Oren Finard - obf2107

Jackson Foley - jcf2172

Alex Peters - arp2169

Brian Yamamoto - bky2102

Zuokun Yu - zy2170

StateMap

### Language Reference Manual

StateMap

### Language Reference Manual

Contents

[1. Introduction 4](#_Toc405925779)

[1.1 Deterministic Finite Automata 4](#_Toc405925780)

[1.2 Statemap Nodes 4](#_Toc405925781)

[1.3 Start State 5](#_Toc405925782)

[1.4 Modularity 6](#_Toc405925783)

[2. Lexical Conventions 7](#_Toc405925784)

[2.1 Comments 7](#_Toc405925785)

[2.2 Identifiers (Names) 7](#_Toc405925786)

[2.3 Keywords 7](#_Toc405925787)

[2.4 Constants 7](#_Toc405925788)

[2.4.1 Integer Constants 7](#_Toc405925789)

[2.4.2 Float Constants 8](#_Toc405925790)

[2.4.3 Character Constants 8](#_Toc405925791)

[2.4.4 Boolean Values 8](#_Toc405925792)

[2.5 Strings 8](#_Toc405925793)

[2.6 Punctuation 9](#_Toc405925794)

[2.6.1 Braces 9](#_Toc405925795)

[2.6.2 Parenthesis 9](#_Toc405925796)

[2.6.2 Semicolon 9](#_Toc405925797)

[2.6.3 Comma 9](#_Toc405925798)

[2.7 Operators 9](#_Toc405925799)

[2.7.1 Arithmetic 9](#_Toc405925800)

[2.7.2 Assignment 10](#_Toc405925801)

[2.7.3 Comparison 10](#_Toc405925802)

[2.7.4 Boolean Evaluation 10](#_Toc405925803)

[2.8 Whitespace 10](#_Toc405925804)

[3. Syntax Notation 11](#_Toc405925805)

[3.1 Program Structure 11](#_Toc405925806)

[3.2 Expressions 11](#_Toc405925807)

[3.2.1 Literals and Operators 11](#_Toc405925808)

[3.2.2 Method Calls 12](#_Toc405925809)

[3.3 Statements 12](#_Toc405925810)

[3.3.1 Declaration 12](#_Toc405925811)

[3.3.2 Assignment 12](#_Toc405925812)

[3.3.3 Function Call 13](#_Toc405925813)

[3.3.4 Transition 13](#_Toc405925814)

[3.3.5 Return 13](#_Toc405925815)

[3.4 Scope 14](#_Toc405925816)

[4. Type 15](#_Toc405925817)

[4.1 Type Declaration 15](#_Toc405925818)

[4.2 Fundamental TYpes 15](#_Toc405925819)

[4.2.1 int 15](#_Toc405925820)

[4.2.2 Float 15](#_Toc405925821)

[4.2.3 char 15](#_Toc405925822)

[4.2.4 string 15](#_Toc405925823)

[4.2.5 stack 15](#_Toc405925824)

[4.2.6 void 16](#_Toc405925825)

[4.3 Non-Fundamental Types 16](#_Toc405925826)

[4.3.1 map 16](#_Toc405925827)

[5. Standard Library 17](#_Toc405925828)

[5.1 Declaring a map 17](#_Toc405925829)

[5.2 Inserting into a map 17](#_Toc405925830)

[5.3 Deleting from a map. 17](#_Toc405925831)

[5.4 Finding a Key 17](#_Toc405925832)

[5.5 Finding the size of the map 17](#_Toc405925833)

[5.6 Deleting the entire map 17](#_Toc405925834)

[6. Built-in Functions 18](#_Toc405925835)

[6.1 Concurrent 18](#_Toc405925836)

[6.2 State 18](#_Toc405925837)

[6.3 Sleep 18](#_Toc405925838)

[7. Program Execution 18](#_Toc405925839)

[./{PATH TO directory}/{Name of Executable} {Arguments} 18](#_Toc405925840)

# 1. Introduction

It has been proven that a PDA with two (or more) stacks can accept any language that a Turing Machine can. From this theorem comes the programming language, StateMap. StateMap is a programming language that is organized and executed in a manner analogous to an Automata diagram, like those seen for DFA’s or PDA’s. It emphasizes modularity and organization of code into short nodes, which transition to each other until reaching some end state. It shrinks the gap between paper diagram and running code to let the programmer go from algorithmic organization to actual execution quickly and simply.

## 1.1 Deterministic Finite Automata

StateMap is a programming language that emulates a DFA (Deterministic Finite Automata). Inspired by the knowledge that a DFA with two stacks can perform any computable operation, the languages purpose is to make the transition from a paper model of a DFA into a program simple to write, and concise in length. The language blends functional and imperative programming styles to allow programmers to abstract away implementations details.

## 1.2 Statemap Nodes

StateMap programs consist of nodes, and within those nodes there are a constant number of operations, as well as transition statements, which allow for control to permanently leave the current node and execute on a new node. Aside from information stored on globally-scoped stacks, no information is preserved from node to node.

There are two types of nodes: transition nodes, and end nodes. Transition nodes can include transition statements, which evaluate expressions, and execute if the expression is true. All transition nodes must end with a default, catch-all transition, to ensure that code execution makes its way to an end node. A return node cannot have any transition statements, but it can return data, and control, to the caller. All return nodes must end with a return statement.

Nodes can call sub-automata, which then execute until they reach an end state. Nodes can also make decisions based on the states of sibling automata, which run in parallel to them.

A node within an automata is defined by a name, followed by curly brackets, within which consist of a number of operations (see ‘operations’ section), with either transition or return statements included. There is no keyword needed to define a state as of type ‘end’ or ‘transition’: the language will infer based on whether the last statement in the node is of type transition or return.

## 1.3 Start State

A StateMap automata always begins at the ‘start’ state. This necessitates that every automata include a state labeled ‘start’. Automata are organized by declaring the name as “DFA name”, and then within curly brackets defining the rest of the automata.

An automata definition consists of, first, its global stack declarations, followed by an (unordered) list of its nodes, and their definitions. The stacks are typed, and must be declared as such (see code examples).

## 1.4 Modularity

The key to a good StateMap program is extreme modularity. Being able to draw the program, on paper, as an automata means that you are probably on the right track. Nodes are, idealistically, short and concise, and, while the ability to create variables does exist, decisions mostly consist of global information, and local variables exist mostly for convenience, efficiency, and the shortening of code length.

# 2. Lexical Conventions

## 2.1 Comments

Both C and C++ style comments are supported.

Multi-line comments begin with characters /\* and end with characters \*/. Any characters may appear inside a multi-line comment except for the string ‘\*/’.

Single line comments begin with the characters // and end with a line terminator.

## 2.2 Identifiers (Names)

An identifier is a sequence of letters, digits, or underscores, the first of which must be a letter. There is no limit to the length of an identifier.

## 2.3 Keywords

The following identifiers are keywords and may only be used as such:

**return int float string void DFA main stack char start**

## 2.4 Constants

There are several types of constants, as follows:

### 2.4.1 Integer Constants

An integer constant consists of one optional minus sign followed by a sequence of one or more digits. The first digit in an integer constant cannot be a zero, unless it’s the only digit.

Valid: 42, 0, -13

Invalid: 042, +13, 00, .25

### 2.4.2 Float Constants

A float constant is a 64-bit signed floating point represented with an optional negative, then either an integer followed by a decimal and another integer or a decimal followed by an integer.

Valid: .3, 1.34, -2.3

Invalid: 42, 0

### 2.4.3 Character Constants

Character constants are represented via enclosure with single quotes “’”. No more than two characters can be enclosed.

Valid: ‘a’, ‘\n’, ‘ ‘

Invalid: ‘hello world’, ‘32+1’

### 2.4.4 Boolean Values

While no explicit Boolean constant type is expressed, any empty value (such as an empty sequence or list) or zero will evaluate to false. Any other value will be evaluated as true.

## 2.5 Strings

Strings are represented via enclosure with double quotes ‘”’. To represent the character ‘”’ without closing the string, it must be preceded with a ‘\’. The empty string is represented with ‘””’, with no characters in between the quotes.

Valid: “hello world”, “ “, “42”, “he told me \”yo\””, “”

Invalid: “He said “This was generally regarded as a bad””

## 2.6 Punctuation

### 2.6.1 Braces

Braces are used to denote the body of a DFA, or the body of a state in the DFA. The body of a DFA may contain variable declarations and state definitions. The body of a state may contain any number of statements.

### 2.6.2 Parenthesis

An expression may include expressions inside parenthesis. Parentheses can also indicate a function call, or a list of parameters for a state.

### 2.6.2 Semicolon

Used to denote the end of a statement.

### 2.6.3 Comma

Used to separate multiple variable names during type assignment and DFA arguments.

Example: String name, address, profession;

DFA count(stack<int> a, int b)

count(wordCount, num)

## 2.7 Operators

### 2.7.1 Arithmetic

|  |  |
| --- | --- |
| Operator | Name |
| + | Addition and String concatenation |
| - | Subtraction and unary negation |
| \* | Multiplication |
| / | Division |
| % | Modulo |

### 2.7.2 Assignment

The assignment operator is ‘=’. This assigns the value of the right side of the operator to the left side variable.

### 2.7.3 Comparison

|  |  |
| --- | --- |
| Operator | Name |
| == | Equality |
| != | Inequality |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |

### 2.7.4 Boolean Evaluation

|  |  |
| --- | --- |
| Operator | Name |
| ! | Not (Negation) |
| && | And (Conjunction) |
| || | Or (Disjunction) |

## 2.8 Whitespace

Whitespace is defined as the ASCII space, horizontal tab, new-line, carriage return, and comments. Whitespace does not affect the program.

# 3. Syntax Notation

## 3.1 Program Structure

Programs are composed of a series of DFAs with a single main DFA to which command line arguments are passed in the form of a stack of strings. The main DFA declaration looks like:

void DFA main(stack<string> main) {}

Additional DFAs without the main identifier follow a similar structure.

/\*TYPE\*/ DFA /\*NAME\*/( /\*ARG1\*/, /\*ARG2\*/) {}

A DFA consists of state blocks separated via braces. Each state block may have any number of statements.

/\*NAME\*/ {

/\*STMT\*/

}

Sub-DFAs (also known as functions) are implemented as a separate DFA, with their own states and transitions. A single StateMap program may contain any number of sub-DFAs.

## 3.2 Expressions

Expressions in StateMap are divided into two categories – both of which returns values.

### 3.2.1 Literals and Operators

Any of the constants listed in section 2.4 or strings will evaluate as expressions. Valid combinations of these constants and operators defined in 2.7 will also evaluate as expressions.

{Id}

{Id} {Operator} {Id}

### 3.2.2 Method Calls

Method calls that return a value will evaluate as expressions.

{Id}.{Method}({Arguments})

Assume a stack called foo was declared. A valid method call is: foo.push(“bar”) and will return “bar”.

## 3.3 Statements

The types of statements in StateMap are declaration, assignment, function call, transition, concurrency and return. Declaration and assignment are the only two types that can be called outside of a node, i.e. globally in a DFA. Every type of statement must be terminated by a semicolon.

### 3.3.1 Declaration

A declaration statement consists of a variable type followed by an id. Multiple declarations can be made in a single line separated by commas.

{TYPE}{ID};

int i;

Stack<double> s, char c, string s;

Note that functions include sub-DFAs. Thus, DFA output may be assigned to variables.

### 3.3.2 Assignment

An assignment statement is used to set the value of a variable, which can be done during the declaration of a variable, or later using the variable's id. Multiple assignment can be made in a single line separated by commas.

{Type}{Id} = {Expression}

int i = 4;

double d = 3.0, char c = 'a', string s = "hello";

### 3.3.3 Function Call

A sub-DFA call (or a function call) statement is a function call expression, but also can be used in an assignment statement taking advantage of the fact that a function call statement has type of the return type of the function.

DFA1(arg1);

string s = DFA2(arg2, arg3);

### 3.3.4 Transition

A transition statement consists of a node id, the transition operator and an expression and is used to denote a transition from one node to another. The transition occurs if the expression evaluates to true.

{State}<-\*

{State}<-{Expression}

state1 <- foo >= bar;

Transition to a state occurs after evaluating the expression on the right side of the arrow. The star operator indicates unconditional transition to the state. Since the transitions are evaluated in order, the {State}<-\* must be the last transition.

### 3.3.5 Return

A return statement consists of the return keyword followed by an expression.

return {expression};

return i < 4;

## 3.4 Scope

Scope in StateMap is divided into local and global types. Local scope is particular to a node where global scope is particular to a DFA.

A variable declared within the curly braces of a DFA is accessible anywhere within that DFA, but not in functions (sub-DFAs called by that DFA. Arguments must be used to pass variables between DFAs.

A variable declared within the curly braces of a node is only accessible within that node.

# 4. Type

## 4.1 Type Declaration

In StateMap, it is required to explicitly declare type when declaring a variable or DFA. The type of a variable will not change during the lifetime of that variable, i.e. StateMap is statically typed. The type of a DFA denotes the type that is returned when that DFA is called.

## 4.2 Fundamental TYpes

### 4.2.1 int

A 32-bit integer.

### 4.2.2 Float

A 64-bit signed floating point number including an exponent portion.

### 4.2.3 char

An 8-bit character.

### 4.2.4 string

A sequence of characters.

### 4.2.5 stack

Normally considered a "non-fundamental" data type, but they are fundamental in StateMap because of their connection to DFAs. Must be declared with a type as follows:

stack<int> s;

Stacks, on the fundamental level, support the following operations:

peek - return the item on the top of the stack

pop - remove and return the item on the top of the stack

push - push a given item in the top of the stack

### 4.2.6 void

While not a type used in variable declaration, DFAs can have return type void if they do not return anything.

## 4.3 Non-Fundamental Types

### 4.3.1 map

StateMap's version of a dictionary. Must be declared with two types as follows:

map<string, int> m;

The map functions are included in the standard library and allow users to add entries, delete entries, return values for given key, return keysets, etc.

# 5. Standard Library

The standard library provides an unordered map implementation.

### 5.1 Declaring a map

Map<key, value> foo

### 5.2 Inserting into a map

insert(key, value)

This function returns true if successful. Otherwise, it returns false.

### 5.3 Deleting from a map.

erase(key)

This function returns true if successful. Otherwise, it returns false.

### 5.4 Finding a Key

find(key)

This function returns the key’s value if the key is valid. Otherwise, it returns void.

### 5.5 Finding the size of the map

size()

### 5.6 Deleting the entire map

clear()

This function returns true if successful. Otherwise, it returns false.

# 6. Built-in Functions

These are a list of functions included innately within StateMap.

## 6.1 Concurrent

Concurrent is a function that takes in any number of sub-DFA calls as arguments. This function will ensure that all sub-DFAs will make their transitions concurrently to allow for synchronized stepping through states. Concurrent will return a stack of strings, where each string represents the output returned by the DFA. The stack is created using Last-In-First-Out ordering – popping the top of the stack returns the output of the last DFA call argument in concurrent(). Only DFA calls are accepted as arguments. Any concurrently-running can only return ints, strings, chars, floats, and void.

concurrent(/\*sub-DFA call\*/, /\*sub-DFA call\*/, /\*sub-DFA call\*/);

concurrent(clock(halfPeriod), TFF1(), TFF2(), display());

The above example runs a clock DFA (which is given an integer), two DFAs that each represent a T-Flip-Flop, and a final DFA that runs a display concurrently.

## 6.2 State

State is a function that takes in a single string argument that represents the name of a DFA. It returns a string that represents the name of the state that the argument DFA is currently in at the moment the function is called. State can only be called within a DFA running concurrently with the desired DFA argument.

state(/\*NAME OF DFA\*/);

state(“clock”) == “rising”;

## 6.3 Sleep

Sleep is a function that takes in a single integer argument and halts the DFA, preventing it from making any further evaluations for the integer argument in milliseconds.

sleep(/\*integer in milliseconds\*/);

sleep(1000);

## 6.4 Print

Print is a function that takes in a single argument of type String. It prints out the argument in the terminal from which the program is being called.

print(/\*string to be printed\*/);

print(“Hello Planet!”);

put input here-------------------------------------------

# 7. Program Execution

Programs are run via command line, in the format:

### ./{PATH TO directory}/{Name of Executable} {Arguments}

For example:

./sorts/quicksort 0 9 2 3