|  |
| --- |
| Programming Language Final Report |
| StateMap |
| Oren Finard, Jackson Foley, Alex Peters, Brian Yamamoto, Zuokun Yu |

|  |
| --- |
| Fall 2014 |

Contents

[2 An Introduction to StateMap 1](#_Toc406619370)

[2.1 StateMap Nodes 1](#_Toc406619371)

[3 Language Tutorial 2](#_Toc406619372)

[3.1 Getting Started 2](#_Toc406619373)

[3.2 Structure of a Program with Hello World! 3](#_Toc406619374)

[3.3 Multiple States 3](#_Toc406619375)

[3.4 Compiling and Running Programs 5](#_Toc406619376)

[3.5 Multiple DFAs and Concurrency 6](#_Toc406619377)

[4 Language Manual 8](#_Toc406619378)

[4.1 Lexical Conventions 8](#_Toc406619379)

[4.1.1 Comments 8](#_Toc406619380)

[4.1.2 Identifiers (Names) 8](#_Toc406619381)

[4.1.3 Keywords 9](#_Toc406619382)

[4.1.4 Constants 9](#_Toc406619383)

[4.1.5 Strings 10](#_Toc406619384)

[4.1.6 Punctuation 10](#_Toc406619385)

[4.1.7 Operators 11](#_Toc406619386)

[4.1.8 Whitespace 12](#_Toc406619387)

[4.2 Syntax Notation 13](#_Toc406619388)

[4.2.1 Program Structure (Main) 13](#_Toc406619389)

[4.2.2 State Blocks 13](#_Toc406619390)

[4.2.3 Sub-DFA 13](#_Toc406619391)

[4.2.4 Expressions 14](#_Toc406619392)

[4.2.5 Statements 15](#_Toc406619393)

[4.2.6 Scope 16](#_Toc406619394)

[4.3 Type 18](#_Toc406619395)

[4.3.1 Type Declaration 18](#_Toc406619396)

[4.3.2 Fundamental TYpes 18](#_Toc406619397)

[4.4 Built-in Functions 20](#_Toc406619398)

[4.4.1 Concurrent 20](#_Toc406619399)

[4.4.2 State 20](#_Toc406619400)

[4.4.3 Sleep 21](#_Toc406619401)

[4.4.4 Print 21](#_Toc406619402)

[4.4.5 Input 21](#_Toc406619403)

[4.4.6 Conversion Functions 21](#_Toc406619404)

[4.5 Program Execution 23](#_Toc406619405)

[5 Project Plan 24](#_Toc406619406)

[5.1 The plan. 24](#_Toc406619407)

[5.2 Specification. 24](#_Toc406619408)

[5.3 Development. 25](#_Toc406619409)

[5.4 Testing. 25](#_Toc406619410)

[5.5 Programming Style Guide 26](#_Toc406619411)

[5.6 Project Timeline 26](#_Toc406619412)

[5.7 Roles and Responsibilities of Each Team Member 27](#_Toc406619413)

[5.8 Software Development Environment Used (Tools and Languages) 28](#_Toc406619414)

[5.9 Project Log 28](#_Toc406619415)

[6 Architectural Design 34](#_Toc406619416)

[6.1 A Diagram of the Major Components of the Translator 34](#_Toc406619417)

[6.2 Interfaces between the components. 34](#_Toc406619418)

[6.2.1 Scanner 34](#_Toc406619419)

[6.2.2 Parser 35](#_Toc406619420)

[6.2.3 Semantic Check 35](#_Toc406619421)

[6.2.4 Code Generation 35](#_Toc406619422)

[6.3 Implementation Responsibilities 36](#_Toc406619423)

[7 Test Plan 36](#_Toc406619424)

[7.1 Printing the AST 36](#_Toc406619425)

[7.2 Unit Tests 37](#_Toc406619426)

[7.3 Exception Tests 38](#_Toc406619427)

[7.4 Automation 39](#_Toc406619428)

[7.5 Sample Source Language Program and Target Language Program 41](#_Toc406619429)

[8 Lessons Learned 52](#_Toc406619430)

[8.1 What Oren Finard learned and advice for other teams 52](#_Toc406619431)

[8.2 What Jackson Foley learned and advice for other teams. 52](#_Toc406619432)

[8.3 What Alexander Peters learned and advice for other teams. 53](#_Toc406619433)

[8.4 What Brian Yamamoto learned and advice for other teams. 54](#_Toc406619434)

[8.5 What Zuokun Yu learned and advice for other teams. 54](#_Toc406619435)

[9 Appendix 55](#_Toc406619436)

[9.1 Scanner Code (scanner.mll) 55](#_Toc406619437)

[9.2 Parser Code (parser.mly) 56](#_Toc406619438)

[9.3 AST Code (ast.ml) 59](#_Toc406619439)

[9.4 Semantic Check Code (semantic\_check.ml) 62](#_Toc406619440)

[9.5 SAST Code (sast.mli) 73](#_Toc406619441)

[9.6 Code Generator Code (gen\_python.ml) 74](#_Toc406619442)

[9.7 Compiler Code (compiler.ml) 81](#_Toc406619443)

# An Introduction to StateMap

It has been proven that a PDA (push-down automaton) 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 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.

## StateMap Nodes

StateMap programs consist of nodes (also known as states), and within those nodes there are a constant number of operations, as well as transition statements, which allow for control to 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 node. 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.

# Language Tutorial

## Getting Started

Before writing any code in StateMap, draw a picture. The essence of StateMap is the ease in which an existing DFA can be encoded and run. Therefore, having a DFA diagram representation of your program on hand while coding in StateMap makes the entire coding process much easier.

If your program is more complex and requires more than one DFA, all of these DFAs can be written in one StateMap program, just as other programming languages can contain multiple functions or methods in a single file. Along those same lines, each .sm file must have a main DFA, and all other DFAs must be written above main. If you wish to write a single DFA StateMap program, it is up to you whether it should be the main DFA, or if main should call your DFA. Also, you may realize while writting your program that parts of your original DFA can be broken off into smaller sub-DFAs, especially if you do repeated work. All of these options are possible and easy to implement in StateMap.

At first, we will conentrate on single DFA programs. By the end of this tutorial, we will show how to write more complex programs in StateMap (i.e. those requiring multiple DFAs or concurrently running DFAs), and you can refer to our Language Reference Manual in section 3 of this report for more detail.

## Structure of a Program with Hello World!

A single DFA StateMap program consists of the declaration of a void main DFA followed by a series of states, the first of which must be start. Each of the states are contained within the braces of the main DFA, and the code for each state is contained within the braces of the state. Below is an example of the Hello World program in StateMap:

void DFA main()

{

start

{

print("Hello World!");

return;

}

}

This is a single DFA, single state program. When the program begins, the start state of the main method is run, and this program prints "Hello World!" to standard out using the built-in print() function.

## Multiple States

The concept of "if" and "while" doesn't exist directly in StateMap. Instead, we use transitions based on boolean expressions to new states, where new code can then be excecuted. Suppose we wanted to print "Hello World!" ten times, whithout writing ten print statements. This can be done with state transitions, as shown below:

void DFA main()

{

int count = 0;

start

{

hello <- count < 10;

finished <- \*;

}

hello

{

print("Hello World!");

count = count + 1;

start <- \*;

}

finished

{

return;

}

}

The "<-" is used for transition statements and is preceeded by the name of a user-defined state, and suceeded by a boolean statement. At a given transition, the program will immediately go to the given state if the boolean expression is true, and will continue in its current state otherwise. Also, the \* is used for a default transition. This is always the last transition listed, and the transition is always followed. These are required in every state containing transitions in StateMap, and can be used for debugging with an error state if they are not needed for your program to function. It is worth mentioning here that states that contain "return" cannot contain transitions, and vice-versa.

As a final note for this example, you can declare variables inside and outside of states. Those declared outside are considered part of the DFA scope, and can be accessed anywhere within the DFA in which it was decalred. Those declared inside a state are part of the state scope, and can only be accessed withing that state, and are cleared at the end of the state.

## Compiling and Running Programs

After running "make compile" to produce the compiler exectutable, you can compile your .sm file with the following command:

$ ./compiler "name of output file" < "path to your .sm file"

This will compile your StateMap program and produce python code called "name of output file".py. You can then run this file with:

$ python "name of output file".py "command line args"

To supply command line arguments to your program, you add them after the python command. Please see our reference manual in section 3 for details on how to do this.

## Multiple DFAs and Concurrency

The most interesting feature of StateMap is the ability to write a program that contains multiple DFAs, and have them interact while running concurrently. This involves using the built-in concurrent() function, which takes in calls to multiple user designed DFAs which are built to work alongside eachother. The following example illistrates this functionality:

void DFA a()

{

start

{

print("DFA a: start");

afinish <- state("b") == "b2";

start <- \*;

}

afinish

{

print ("DFA a is done.");

return;

}

}

void DFA b()

{

start

{

print("DFA b: start");

b1 <- \*;

}

b1

{

print("DFA b: b1");

b2 <- \*;

}

b2

{

print("DFA b: b2");

bfinish <- \*;

}

bfinish

{

print ("DFA b is done.");

return;

}

}

void DFA main()

{

start

{

concurrent(a(), b());

return;

}

}

In this example, there are two DFAs, labeled "a" and "b". Each of the DFAs have a helpful print statement that prints its current state as soon as it arrives there. Then, DFA b's transitions are defined such that it moves through each of its states unconditionally in order: start -> b1 -> b2 -> bfinish. DFA a only transitions from its start state when DFA b is in state b2. This is accompished using the built-in state() function, which takes in a string name of a DFA and returns a string which represents the name of the state the given DFA is currently in. The line above " state("b") == "b2" " is asking if the DFA labeled "b" is currently in state "b2".

The output of this program is the following:

DFA b: start

DFA a: start

DFA b: b1

DFA a: start

DFA b: b2

DFA a: start

DFA b is done.

DFA a is done.

As you can see, DFA a remains in its start state until DFA b reaches b2, upon which they both finish.

This concurrency functionality allows you to write a program consisting of multiple DFAs designed to interact while they are running. This has great application value in any program seeking synchronous behavior becuase DFAs that run conccurently make transitions simultaneously. The most obvious application here is multiple parts of hardware that are sychronized with a clock, but many other hardware and software applications exist.

# Language Manual

## Lexical Conventions

### 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.

### 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.

### Keywords

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

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

### Constants

There are several types of constants, as follows:

#### 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

#### 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

#### 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.

### 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 asked “Do you have your towel?””

### Punctuation

#### 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.

#### Parenthesis

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

#### Semicolon

Used to denote the end of a statement.

#### Comma

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

Example: String name, address, profession;

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

count(wordCount, num);

### Operators

#### Arithmetic

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

#### Assignment

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

#### Comparison

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

#### Boolean Evaluation

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

### Whitespace

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

## Syntax Notation

### Program Structure (Main)

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(/\*args\*/) {}

If the number of arguments are known beforehand, they can be passed to the main DFA like so:

void DFA main(/\*[type] name1, [type] name2, etc\*/) {}

Otherwise, rely on a stack of primitives:

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

### State Blocks

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

/\*NAME\*/ {

/\*STMT\*/

}

Every state block must either have a catch-all transition (<- \*;) or a return statement. Every DFA must have a state labeled “start”, which will act as the first node acting in a DFA.

### Sub-DFA

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. If sub-DFAs exist, the main DFA must be the last DFA declared in the program.

Sub-DFAs follow a similar structure as the main DFA.

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

Each DFA (including main) must eventually return their type:

void DFA -> return;

int DFA -> return [int];

float DFA -> return [float];

string DFA -> return [string];

Note that a formal passed into any DFA can’t be of type void or EOS.

### Expressions

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

#### 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}

#### 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”.

### Statements

The types of statements in StateMap are declaration, assignment, sub-DFA 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.

#### 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.

#### 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, string s = "hello";

#### Sub-DFA 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);

#### 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}<-\* should be the last transition.

#### Return

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

return {expression};

return i < 4; // returns an int 1

### 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 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.

## Type

### 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.

### Fundamental TYpes

#### int

A 32-bit integer.

#### Float

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

#### string

A sequence of characters.

#### 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. Running this operation on an empty stack return EOS (not a string).

stop <- stack.peek() == EOS;

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

s = stack.pop();

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

string s = “towel”;

stack.push(s)

#### void

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

Calling return in a void DFA will return an int of 1, which allows you to transition on a void sub-DFA call.

## Built-in Functions

These are a list of functions included within StateMap.

### 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. Concurrently-running DFAs can only return ints, strings, 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.

### 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”;

### 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);

### 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!”);

### Input

Input 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 (like print()) – however, it then waits for input from the user until the Enter key is pressed. Input then passes back the input before the Enter key as a string as a return value.

string msg = input(/\* string typed in terminal \*/);

### Conversion Functions

Conversion functions allow for conversion between types – it takes in the constant to be converted and returns the constant as its new converted type.

The available functions are:

stof: converts type string to type float

ftos: converts type float to type string

stoi: converts type string to type int

itos: converts type int to type string

For example:

string a = “3.0”;

float x = ftos(a);

## Program Execution

StateMap programs are saved with .sm extension:

To compile, run the following commands:

1) make

2) ./compiler {output name} < {path to .sm file}

3) python {output name}.py {args}

After compiling, programs are run via command line, in the format:

python outputName.py {args separated by space}

For example:

python outputName.py 0 9 2 3

Stacks can be passed in a command line by separation via commas. No spaces should exist between the elements of a stack:

python outputName.py a,b,c

python outputName.py [a,b,c] // is also allowed

To pass in a string as a stack of strings, with each string consisting as a single character of the string, surround the string to be passed with “’ (double quotes then single quotes):

python output.py “’bitbybit’”

will pass the main DFA b,i,t,b,y,b,i,t as a stack.

# Project Plan

## The plan.

The planning of the project started simple, and (surprisingly) did not vary greatly with time. The original idea for StateMap sprung from the theorem (taught in CS theory) that a Finite Automata with two or more stacks could (theoretically) compute and computable problem. This was the kernel that started the process, and still remains the heart of the language.

There was no over-arching “plan” or specified timeline in our group- taking a page out of Socrate’s handbook (“The only true wisdom is knowing you know nothing”) we iteratively set short term goals for ourselves, guided heavily by our TA, Olivia Byer, to allow ourselves to respond to unexpected difficulties in the project. We worked steadily and consistently throughout the semester, with the bulk of the work being done in the last month, increasing exponentially throughout the month. This was not due to timing issues, but rather that the majority of the work came from fixing issues found through testing. Once the language generated code, through testing we were able to greatly adjust, customize, and improve our language.

## Specification.

Because StateMap never thematically changed, the first round LRM covered the majority of the language throughout the project. The original LRM was written, essentially, by all five members sitting in a room together, spending 30 minutes going over everything that we agreed with each other about the language, and then yelling at each other for an hour about the five things we all disagreed on.

The original LRM was (obviously) written predictively, and was meant as a guiding light: this equated to, later in the project, needing to change specifications, add details, clarify oddities, and add many notes. This was done mostly through meetings and an active Facebook group, and at the end, through an email chain. One member of our group took on the responsibility of adjusting the LRM at all times.

## Development.

Everyone warned us about the troubles of developing in a team, and group dynamics, but development was surprisingly simple. Right from the beginning, the group set aside time to meet weekly in addition to meeting with the TA weekly. We ended up skipping many weeks because the work for the project was either straightforward, or didn’t need to be discussed with the group. We rarely missed a TA meeting, and often used the time after the meeting to sketch out what we would do for the week.

As development got more heavy, who worked on what really fell into who had time to work. Rather than wait for meeting times, we just started texting each other to find out who was free to work. Everyone touched all parts of the project, regardless of their roles, but the roles definitely helped organize members into who worked on what at the end, when there was more than just one thing to do at a time.

## Testing.

We created an automated testing suite that checked the various parts of our language. This was set up in the late stages of the project, primarily once code generation was running. However, along the was the Tester was also making sure that the language was running properly at various milestones: testing was done after finishing the parser, thus breaking the code into a reduction tree, and then also after finishing the SAST, again breaking code into a reduction tree. However, these tests were not automated.

## Programming Style Guide

StateMap is meant to translate DFA diagrams from paper to code easily and clearly. Statements in the language are meant to be short and clear. The lack of ‘if’ statements and loops (instead, we have ‘transition’ statements) forces a very unique style of programming. The ideal is to create simple nodes, with a few, easily read and deterministic (read: non-arbitrary) lines of code. Brevity of node blocks, and clarity of code are prioritized over length of files, and complexity of overall design. The number of tools the coder has are significantly diminished compared to other languages, but the language is extremely simple to understand, use correctly, and use powerfully: it does, however, force the programmer to think through (and often draw out) their program beforehand.

## Project Timeline

As mentioned in the planning section, there was no initial timeline planned, but as we went along we had several soft deadlines:

Finished By:

9/15 First Team Meeting

9/22 Birth of StateMap Idea & Discussion

9/24 Proposal

10/6 Scanner

10/13 Parser

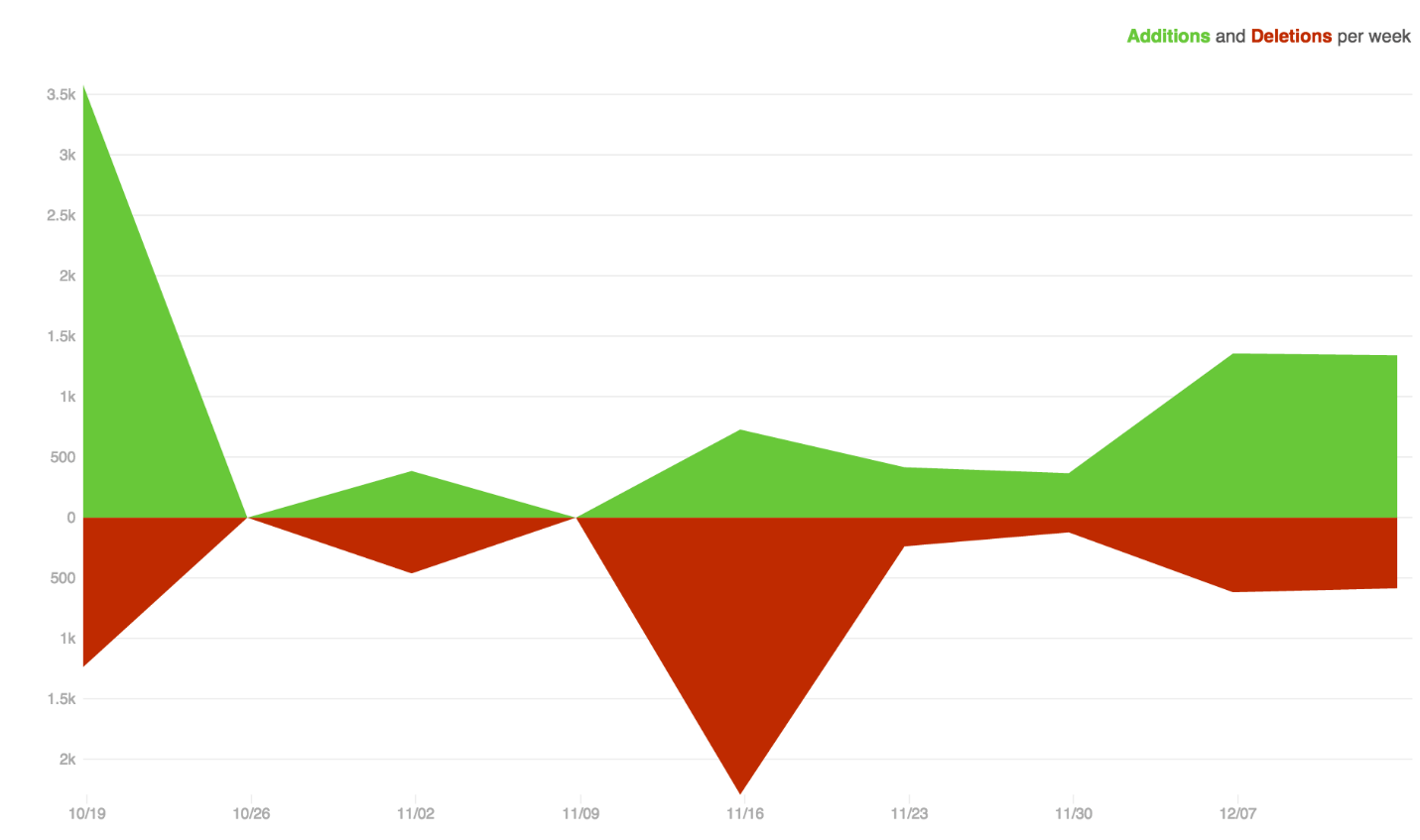
10/27 LRM

11/27 Semantic Check

12/14 Code Generation

12/16 Everything Finished

Below is our git commit timeline for our repository:



## Roles and Responsibilities of Each Team Member

As was mentioned in the development part of the planning section, everyone touched all parts of everything. Every member had a part in making all team decisions (mostly the difficult or tricky ones), and all other decisions were left to the implementer. We assigned team roles, but the roles and responsibilities that people actually fulfilled are as follows:

**Alex “Dread Pirate Roberts” Peters** = StateMap Code Creator, Exercising Common Sense, *Language Guru*

**Brian “LoL” Yamamoto** = LRM Management, Presentation Coordinator, Report Organizer, StateMap Code Creator, *Language Guru*

**Jackson “Swag” Foley** = Testing and Test Suite Management, Exercising Uncommon Sense, *Verification and Validation*

**Oren “DopeDopeDope@Dope.com”** Finard = Team Mom, Python Guru, *Manager*

**Zuokon “AlreadyFixedYourProblem” Yu** = Semantic Check Master, Master of Knowing How The Entire Language Works, *System Architect*

## Software Development Environment Used (Tools and Languages)

We used OCaml 4.02 to write the compiler, and compiled the StateMap language into Python 2.7. Additional tools include the use of Bash scripts for testing, Git for version control, OCamllex for the Scanner, and OCamlyacc for the Parser. Also, Makefiles.

## Project Log

\* Alexander\_Peters Made Makefile get rid of all .py files

\* Zuokun Yu Variadic output file

\* Brian Yamamoto Finalized LRM uploaded, comments made for reg\_ex

\* Jackson Foley test\_all formatting

\* Alexander\_Peters removed extra gcd and updated gcd

\* Jackson Foley Merge branch 'master' into tests

|\

| \* Zuokun Yu Fixed scoping issues

| \* Alexander\_Peters added a concurrency example to be used in the final report

| \* Alexander\_Peters added gcd, and an example of gcd that throws an exception for unknown reason

| \* Alexander\_Peters Added hello10.sm for report. First transition program, prints Hello World ten times.

| \* Brian Yamamoto Merge branch 'tests'

| |\

| \* | Jackson Foley removes test

\* | | Jackson Foley Merge branch 'tests' of https://github.com/jacksonConrad/StateMap into tests

|\ \ \

| |/ /

|/| /

| |/

| \* Brian Yamamoto project CYOA runs

| \* Brian Yamamoto reg\_ex\_test accepts only (ab|c\*)d\*

| \* Brian Yamamoto Merge branch 'master' of https://github.com/jacksonConrad/StateMap into tests

| |\

\* | \ Jackson Foley Merge branch 'master' of https://github.com/jacksonConrad/StateMap into tests

|\ \ \

| |/ /

|/| /

| |/

| \* Nero144 fixed input. Needed to use raw\_input not input

\* | Brian Yamamoto Added simple input test

\* | Brian Yamamoto Merge branch 'master' into tests

|\ \

| |/

| \* Zuokun Yu string == string no longer returns True

\* | Brian Yamamoto Updates to CYOA

|/

\* Brian Yamamoto no return statement test

\* Brian Yamamoto Merge branch 'tests' of https://github.com/jacksonConrad/StateMap into tests

|\

| \* Jackson Foley Merge branch 'tests' of https://github.com/jacksonConrad/StateMap into tests

| |\

| \* | Jackson Foley adds empty print test, and makes it pass

| \* | Zuokun Yu Strings aren't cast to ints anymore

| \* | Nero144 Merge branch 'master' of https://github.com/jacksonConrad/StateMap

| |\ \

| | \* | Alexander\_Peters fixed buugs in shift\_reg

| \* | | Nero144 created mad string stack rules to get strings of all kinds into stacks from the command-line

\* | | | Brian Yamamoto Fixed out files again and renamed to no\_catch\_all

| |\_|/

|/| |

\* | | Brian Yamamoto fixed out files

\* | | Brian Yamamoto Missing return statements and multiple declarations in a state tests

| |/

|/|

\* | Jackson Foley boolean binops now return 1 or 0 instead of True or False

\* | Zuokun Yu Changed permissions/Makefile so it can execute

\* | Jackson Foley Merge branch 'master' of https://github.com/jacksonConrad/StateMap

|\ \

| \* \ Zuokun Yu Merge branch 'master' of https://github.com/jacksonConrad/StateMap

| |\ \

| \* | | Zuokun Yu Added string + string -> string

\* | | | Jackson Foley merges ast\_print into tests

|\ \ \ \

| \* | | | Jackson Foley removes statemap.ml, replaces it with ast\_print.ml. appropriate changes in Makefile

\* | | | | Jackson Foley Merge branch 'master' into tests

|\ \ \ \ \

| |/ / / /

|/| | / /

| | |/ /

| |/| |

| \* | | Alexander\_Peters Merge branch 'master' of https://github.com/jacksonconrad/statemap

| |\ \ \

| | |/ /

| | \* | Zuokun Yu More cleanup

| | \* | Zuokun Yu Cleaning up code

| | |/

| \* | Alexander\_Peters modified counter.sm to count higher and added a new (not yet working) source example of a shift register shift\_reg.sm

| |/

\* | Jackson Foley adds exception testing

\* | Jackson Foley moar tests

|/

\* Nero144 semantically check that only correctly called DFAs are allowed as arguments for the concurrent()

\* Nero144 added the ability to give stacks in at the command line

\* Nero144 added stof fots stoi and input to semantic check. added all but input to gen\_python

\* Nero144 enforces that concurrent only ever returns string values

\* Nero144 merge commit

|\

| \* Zuokun Yu main DFAs must return void. Fixed tests. int->void

\* | Nero144 added self.\_next = None after a return statement to help prevent an accidental infinite loop

|/

\* Jackson Foley Merge branch 'code\_gen' of https://github.com/jacksonConrad/StateMap into code\_gen

|\

| \* Zuokun Yu Added new test and removed error from Makefile

\* | Jackson Foley adds sleep() test and gen\_python fixes

\* | Jackson Foley fixes code gen for state() function. adds test for state()

|/

\* Jackson Foley fixes concurrent test output. fixes args getting passed into main dfa vs subdfa

\* Nero144 fixed the scoping issue of name overshadowing by adding underscores to dfa/node names and researved words

\* Nero144 fixed naming overshadowing issues

\* Zuokun Yu Modified contents of output files

\* Nero144 some minor changes to gen\_python. I actually forget what

\* Zuokun Yu Added \n to end of files so colordiff doesn't complain

\* Jackson Foley fixes test suite again

\* Jackson Foley fixes indentation in test\_all

\* Jackson Foley fixes test suite output

\* Zuokun Yu Removing log.txt

\* Zuokun Yu Passing current test suite

\* Jackson Foley adds all and test targets to Makefile

\* Jackson Foley fixes merge conflicts with tests branch

|\

| \* Jackson Foley adds arithmetic, basic\_stack, dfa\_args, and return\_types tests

| \* Jackson Foley Merge branch 'master' into tests

| |\

| | \* Brian Yamamoto Merge branch 'master' of https://github.com/jacksonConrad/StateMap

| | |\

| | \* | Brian Yamamoto Added LRM and sample CYOA code

| \* | | Jackson Foley adds concurrent test and subdfa\_call test

| \* | | Jackson Foley improves test\_all output, and now generates log.txt file. adds void\_return test

| \* | | Jackson Foley Adds test script, test directory with output files.

\* | | | Jackson Foley Removes .swp files...ORENgit add --allgit add --all

\* | | | Jackson Foley Merge branch 'code\_gen' of https://github.com/jacksonConrad/StateMap into code\_gen

|\ \ \ \

| |/ / /

|/| | |

| \* | | Nero144 made a more complex example code, and logged a bunch more issues in Notes

| \* | | Nero144 fixed the issue with all locals being seen as dfa scope, added push pop and peek and they work, and added state to the list of predefined funcs/dfas

\* | | | Jackson Foley adds output.py to .gitignore. moves wordcount.sm to sample\_programs directory

|/ / /

\* | | Nero144 just some minor 4am adjustments

\* | | Zuokun Yu Rehauled semantic\_check

\* | | Zuokun Yu Location based scoping

\* | | Zuokun Yu Scoping

\* | | Nero144 Merge branch 'code\_gen' of https://github.com/jacksonConrad/StateMap into code\_gen

|\ \ \

| \* | | Zuokun Yu Remove inf. loop in code\_gen

| \* | | Zuokun Yu More bugs in semantic\_check. Correctly propagate envs

\* | | | Nero144 fixing stuff with Zuokon

|/ / /

\* | | Nero144 merged compiler

|\ \ \

| \* | | Zuokun Yu Mutually exclusive return/transition

| \* | | zeeKKR Delete output.py

| \* | | zeeKKR Delete .compiler.ml.swp

\* | | | Nero144 changed compiler stuffs

|/ / /

\* | | Nero144 We got Hello World working (commits wont let me use exclemation marks but imagine a ton of them)

\* | | Jackson Foley fixes 10000 bugs in gen\_python. Makefile lets us debug.

\* | | Jackson Foley Merge branch 'master' into code\_gen

|\ \ \

| | |/

| |/|

| \* | Alexander\_Peters Merge branch 'master' of https://github.com/jacksonconrad/statemap

| |\ \

| \* | | Alexander\_Peters commting changes to source code

\* | | | Jackson Foley Merge branch 'master' into code\_gen

|\ \ \ \

| | |/ /

| |/| |

| \* | | Jackson Foley Merge branch 'ast'.

| |\ \ \

| | |/ /

| |/| |

| | \* | Jackson Foley removes all occurences of ExprAssign. Assignments are explicitly stmts

| \* | | Alexander\_Peters added new Hello World source code, and updated other source code

| \* | | Alexander\_Peters removed unary operators INC and DEC

| |/ /

\* | | Jackson Foley Merge branch 'code\_gen' of https://github.com/jacksonConrad/StateMap into code\_gen

|\ \ \

| \* | | Nero144 wrote gen\_node\_body

| \* | | Nero144 some mucking with the code\_gen

\* | | | Jackson Foley Merge branch 'code\_gen' of https://github.com/jacksonConrad/StateMap into code\_gen

|\ \ \ \

| |/ / /

| \* | | Nero144 Merge branch 'master' into code\_gen

| |\ \ \

| | |/ /

| \* | | Nero144 adds sample programs

\* | | | Jackson Foley Merge branch 'code\_gen' of https://github.com/jacksonConrad/StateMap into code\_gen

|\ \ \ \

| |/ / /

| \* | | Nero144 worked on the callDfa and concurrent dfas with jackson

\* | | | Jackson Foley Merge branch 'master' into code\_gen

|\ \ \ \

| |/ / /

|/| / /

| |/ /

| \* | Zuokun Yu More holistic semantic check

\* | | Nero144 just some more code gen messing around

\* | | Nero144 did some work on the code gen, but it's kind of a mess

\* | | Nero144 better way to make dfa calls

\* | | Nero144 changed the python template

\* | | Jackson Foley starts code gen. fixes program def in sast. adds hypothetical python representation of our code.

\* | | Jackson Foley adds compiler, starts gen\_python based off Slang

|/ /

\* | Zuokun Yu Actually got rid of Doubles

\* | Alexander\_Peters Fixed a bug with assignment statement in parser.mly

\* | Alexander\_Peters Added the ability to assign a value to a variable outside of a vdecl

|/

\* Zuokun Yu semantic\_check compiles

\* Zuokun Yu Double to float promotion. sast. Making semantic\_check compile

\* Zuokun Yu More functional semantic\_check/add sast

\* Nero144 Added dfa as a variable type for the concurrent function to take dfa's as arguments

\* Zuokun Yu Parser properly accepts <> notation for stacks and they're properly printed in the AST

\* Zuokun Yu added void in front of main. Made concurrent a function to match scanner/parser. Changed & to &&.

\* Zuokun Yu Removed main token from scanner. Parser recognizes stack types

\* Alexander\_Peters Merge branch 'master' of https://github.com/jacksonconrad/statemap

|\

| \* Nero144 added the built-in functions String state(String dfa), Void print(String str), Void sleep(Int ms), String itos(Int int) to the semantic check

\* | Alexander\_Peters edited counter.sm to reflect changes from 12-3 meeting

\* | Alexander\_Peters Finished counter.sm

\* | Alexander\_Peters added a start to a new sample program, counter.sm

|/

\* Jackson Foley creates sample program directory. adds statemap executable to .gitignore

\* Jackson Foley adds Makefile to compile everything

\* Jackson Foley adds string\_of\_\* functions for printing the AST

\* Jackson Foley adds printing functions to ast.ml. makefile changed from ast.mli to ast.ml.

\* Jackson Foley Fixes push pop peek parser errors

\* Jackson Foley comments out recklessly added lines. adds new scanner tokens to the top of parser.mly

\* Jackson Foley resolves merge conflicts merging master into sast branch

|\

| \* Jackson Foley Scanner, Parser, Ast compilesgit add .git add .git add .

| \* Jackson Foley removes 'main' from parser, fills in brackets in parser, adds functionality to ast

\* | Nero144 first round semantic check

\* | zeeKKR semantic\_check v2

\* | zeeKKR Added semantic\_Check

|/

\* Jackson Foley Merge branch 'ast'

|\

| \* Alexander\_Peters Added first bit of source code wordcount.sm

\* | Jackson Foley adds basic Makefile

\* | Jackson Foley updates .gitignore

|/

\* Jackson Foley merges ast branch into parser

|\

| \* zeeKKR Actually add ast.

| \* zeeKKR Added ast. Changed parser/scanner to accept double. Simplified stmt in parser.

\* | Jackson Foley removes comment

|/

\* Jackson Foley adds statement and node production to the parser. no shift/reduce errors.

\* Jackson Foley fixes quote error in scanner

\* Jackson Foley removes reduction rules. no shift/reduce errors here

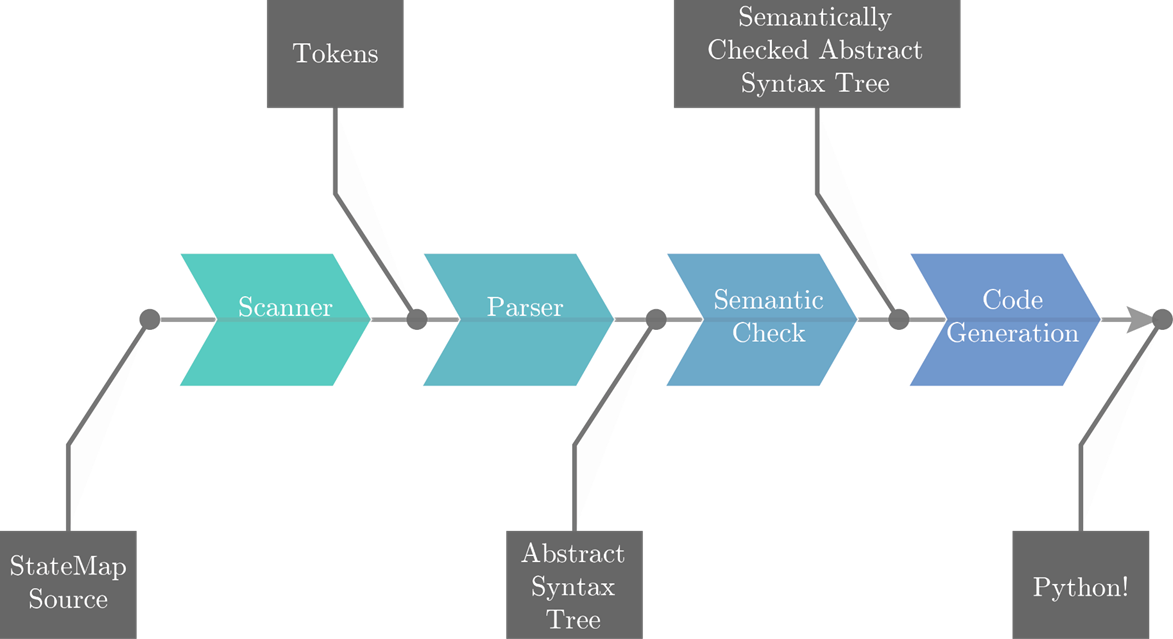
\* Jackson Foley removes superfluous methods from parser

\* Jackson Foley initial commit

\* Jackson Foley Initial commit

# Architectural Design

## A Diagram of the Major Components of the Translator



## Interfaces between the components.

### Scanner

The scanner tokenizes StateMap’s source code.

### Parser

The parser takes the tokens produced by the scanner and produces an abstract syntax tree (AST).

### Semantic Check

The semantic check takes an AST and semantically checks it. Some untraditional conditions it checks include:

1) A DFA called main exists and its return type is void.

2) Every non-void DFA actually returns something.

3) Every DFA has a node called start.

4) Every node in a DFA either has a transition statement or returns.

5) If a node doesn’t have a return, it must at least have an unconditional transition statement.

It also checks traditional conditions such as the existence of a variable within a specified scope or type consistencies for assignments. The final output is a semantically checked AST.

### Code Generation

The code gen takes the semantic checked program, and translates it into python. It turns a DFA into a class, and the states into class methods. It also generates a fair amount of pre-established pure python code that is used to do built-in language functions (wrapped as DFAs), as well as set up the architecture for the main function to run itself (all other functions are run/managed in python by the DFA that calls them).

## Implementation Responsibilities

Even though we all worked on coding the compiler, there were certain parts that had distinct ownership.

Jackson wrote the test suite.

Brian and Alex wrote the overwhelming majority of the sample programs.

Zuokun and Oren were responsible for debugging.

# Test Plan

Our test suite consists of three components – unit tests, exception tests, and AST printing. The unit tests test the smallest building blocks of our language, providing us assurance that we aren’t breaking anything as we add additional functionality. The exception tests test that the semantic check catches things we think shouldn’t be allowed in our language, and forces us to create verbose error messages when an exception is thrown.

The unit tests compile and run the test programs, saving the output of each to a \*.output file. It then compares this output the expected output file. The bare test results are printed to STDIN, while more verbose error messages are appended to a log.txt file.

The exception tests attempt to compile a malformed program, save the compiler output to a \*.output file, and uses ‘egrep’ to verify that the compiler output contains an appropriate error message.

## Printing the AST

The first form of testing we developed before could generate Python code was a script to take in a program and print out the corresponding AST. This can be compiled and run by typing ‘make ast\_print’ and then ‘./ast\_print < ./sample\_programs/counter.sm’.

## Unit Tests

|  |  |
| --- | --- |
| File | Functionality Tested |
| Arithmetic.sm | Integer arithmetic |
| Basic\_stack.sm | Pop, push, and peek stack functions |
| Concurrent.sm | Built-in concurrent() function |
| Dfa\_args.sm | Ability to pass arguments to sub-dfas |
| Hello.sm | The simplest program, printing “Hello World!” |
| Keywords\_as\_states.sm | Tests that our built-in functions don’t pollute the namespace of states |
| Logic\_ops.sm | Boolean logic operators |
| Return\_types.sm | Verifies a sub-dfa can return ints, floats, strings, and void |
| Sleep.sm | Built-in sleep() function |
| Subdfa\_as\_function\_param.sm | Tests that calls to sub\_dfa’s are ultimately evaluated as expressions and can be passed as a parameter to another sub\_dfa |
| Type\_conversions.sm | Built in functions for converting between strings and other types |
| Concurrent\_return\_and\_self\_loop.sm | Popping things off the stack that concurrent() returns |
| Empty\_print.sm | Checks that an empty print statement prints a newline |
| Subdfa\_call.sm | Checks that a subdfa can be called as a function |
| Subdfa\_state.sm | Checks that the state() function can be called from one subdfa to check the state that another subdfa is in |
| Void\_return.sm | Checks that the void return type works properly |

## Exception Tests

|  |  |
| --- | --- |
| **File** | **Functionality Tested** |
| Assign\_void.sm | Ensures you can’t assign a function that returns “void” to a variable. |
| Duplicate\_dfa.sm | Verifies you can’t declare two sub-dfas with the same name |
| No\_decl.sm | Ensures variable declarations must contain the type |
| No\_start.sm | Ensures a DFA must have a state named ‘start’ |
| Wrong\_decl\_order.sm | Ensures error is thrown if you call a DFA that isn’t declared above the one you call it from |
| Multi\_decl.sm | Ensures error is thrown if you declare a variable more than once. (i.e. int x = 1; int x = 2;) |
| No\_catch\_all\_main.sm | Ensures that you must have a \* transition or a return statement in a state in the main DFA |
| No\_catch\_all\_subDFA.sm | Ensures that a state in a subDFA must have a \* transition or a return statement. |
| No\_return.sm | Ensures a subDFA must contain a return statement |

## Automation

These automated test scripts were used for regression testing. To run all the tests, in the StateMap directory, run:

make test

Note: you may have to install colordiff. If you are on a Mac and use homebrew, run:

brew install colordiff

Alternatively, open up unit\_tests.sh and exception\_tests.sh and change each instance of ‘colordiff’ to ‘diff’.

**Unit testing:**

This script compiles and runs each test program, saves the output to a file, and compares the output to the expected output in the corresponding \*.out file.

It prints the test results to the screen, and saves more verbose output to a log.txt file.

#!/bin/bash

#script used for reg testing

COMPILER="./compiler"

COMPFILE='output.py'

LOGFILE='log.txt'

rm -f "$LOGFILE" &>/dev/null

for TESTFILE in ./tests/\*.sm;

do

echo " TESTING $TESTFILE" | tee -a "$LOGFILE"

LEN=$((${#TESTFILE}-3))

OUTFILENAME="${TESTFILE:0:$LEN}.output"

TESTFILENAME="${TESTFILE:0:$LEN}.out"

echo "Compiling ... " >> "$LOGFILE"

("$COMPILER" < "$TESTFILE") 2>> "$LOGFILE"

# if compilation succeeds, run output.py.

if (find output.py &>/dev/null)

then

echo "Python runtime output:" >> "$LOGFILE"

(python "$COMPFILE" > "$OUTFILENAME") 2>> "$LOGFILE"

echo "Diff:\n" >> "$LOGFILE"

touch "$OUTFILENAME"

if (diff "$OUTFILENAME" "$TESTFILENAME" >/dev/null)

then

echo 'OK!' | tee -a "$LOGFILE"

else

colordiff -y "$OUTFILENAME" "$TESTFILENAME"

echo "BAD!" | tee -a "$LOGFILE"

fi

else

echo "BAD!\nCompilation of $TESTFILE FAILED" | tee -a "$LOGFILE"

fi

touch output.py

rm '$COMPFILE' "$OUTFILENAME" &>/dev/null

done

exit 0

**Exception testing:**

This script compiles each test program, and saves the compiler output to a file. It then uses ‘egrep’ to check that the phrase in the corresponding \*.out file appears in the compiler output. It prints the test results to the screen, and saves more verbose output to a log\_fail.txt file.

#!/bin/bash

#script used for reg testing

COMPILER="./compiler"

COMPFILE='output.py'

LOGFILE='log\_fail.txt'

rm -f "$LOGFILE" &>/dev/null

for TESTFILE in ./tests/to\_fail/\*.sm;

do

echo " TESTING $TESTFILE" | tee -a "$LOGFILE"

LEN=$((${#TESTFILE}-3))

OUTFILENAME="${TESTFILE:0:$LEN}.output"

TESTFILENAME="${TESTFILE:0:$LEN}.out"

echo "Compiling ... " >> "$LOGFILE"

("$COMPILER" < "$TESTFILE") 2> "$OUTFILENAME"

if (egrep -f "$TESTFILENAME" "$OUTFILENAME" >> "$LOGFILE" 2>&1)

then

echo "OK!"

else

echo "BAD!"

colordiff -y "$OUTFILENAME" "$TESTFILENAME" 2>> "$LOGFILE"

fi

rm '$COMPFILE' "$OUTFILENAME" &>/dev/null

done

exit 0

## Sample Source Language Program and Target Language Program

**StateMap to Python**

Counter simulation:

The following example simulates a counter using 2 T flip-flops:

//Synchronous Counter with 3 T-Flip-Flops (0 to 7) and Display

// Prints a number to standard out based on

// states of the TFFs

void DFA display()

{

start

{

print0 <- (state("clock") == "rising"

&& state("TFF1") == "high"

&& state("TFF2") == "high"

&& state("TFF3") == "high");

print1 <- (state("clock") == "rising"

&& state("TFF1") == "start"

&& state("TFF2") == "start"

&& state("TFF3") == "start");

print2 <- (state("clock") == "rising"

&& state("TFF1") == "high"

&& state("TFF2") == "start"

&& state("TFF3") == "start");

print3 <- (state("clock") == "rising"

&& state("TFF1") == "start"

&& state("TFF2") == "high"

&& state("TFF3") == "start");

print4 <- (state("clock") == "rising"

&& state("TFF1") == "high"

&& state("TFF2") == "high"

&& state("TFF3") == "start");

print5 <- (state("clock") == "rising"

&& state("TFF1") == "start"

&& state("TFF2") == "start"

&& state("TFF3") == "high");

print6 <- (state("clock") == "rising"

&& state("TFF1") == "high"

&& state("TFF2") == "start"

&& state("TFF3") == "high");

print7 <- (state("clock") == "rising"

&& state("TFF1") == "start"

&& state("TFF2") == "high"

&& state("TFF3") == "high");

start <- \*;

}

print0

{

print("0");

start <- \*;

}

print1

{

print("1");

start <- \*;

}

print2

{

print("2");

start <- \*;

}

print3

{

print("3");

start <- \*;

}

print4

{

print("4");

start <- \*;

}

print5

{

print("5");

start <- \*;

}

print6

{

print("6");

start <- \*;

}

print7

{

print("7");

start <- \*;

}

}

// DFA to represent a clock

// halfPeriod: integer to represent period/2 in ms

void DFA clock(int halfPeriod)

{

// Start == low

// Wait halfPeriod ms, then toggle

start

{

sleep(halfPeriod);

rising <- \*;

}

// state that triggers a toggle for the TFFs

rising

{

high <- \*;

}

high

{

sleep(halfPeriod);

start <- \*;

}

}

// 1st T-FlipFlop in counter

// Toggles on every rising clock

void DFA TFF1()

{

// low output

start

{

high <- (state("clock") == "rising");

start <- \*;

}

// high output

high

{

start <- (state("clock") == "rising");

high <- \*;

}

}

// 2nd T-FlipFlop in counter

// Toggles on every clock only if TFF1 is high

void DFA TFF2()

{

// low output

start

{

high <-(state("clock") == "rising"

&& state("TFF1") == "high");

start <- \*;

}

// high output

high

{

start <-(state("clock") == "rising"

&& state("TFF1") == "high");

high <- \*;

}

}

// 3rd T-FlipFlop in counter

// Toggles on every clock only if TFF1 AND TTF2 is high

void DFA TFF3()

{

// low output

start

{

high <-(state("clock") == "rising"

&& state("TFF1") == "high"

&& state("TFF2") == "high");

start <- \*;

}

// high output

high

{

start <-(state("clock") == "rising"

&& state("TFF1") == "high"

&& state("TFF2") == "high");

high <- \*;

}

}

void DFA main()

{

int halfPeriod = 400;

start

{

print("0");

concurrent(clock(halfPeriod), TFF1(), TFF2(), TFF3(), display());

return;

}

}

Corresponding Python Code:

#########BEGIN AUTOGENERATED FUNCTIONS ###########

from time import sleep

import sys

\_dfa\_Dict = dict()

def \_node\_start():

#do nothing: just exist as a function for the dfas to initially

#point to with `dfa.\_now` so that we can have correct formatting in

#state()

return

def state(dfa):

return \_dfa\_Dict[dfa].\_now.\_\_name\_\_[6:]

def makeStack(stacktype,string\_of\_stack):

if stacktype != str:

return map(stacktype,string\_of\_stack.replace('[','').replace(']','').split(','))

else:

if "'" not in string\_of\_stack and '"' not in string\_of\_stack:

return map(stacktype, string\_of\_stack.split(','))

elif ('"' not in string\_of\_stack or

(string\_of\_stack.find("'") < string\_of\_stack.find('"') and

string\_of\_stack.find("'") != -1)):

startIndex = string\_of\_stack.find("'")

endIndex = string\_of\_stack.find("'",startIndex+1)

if endIndex == -1:

print('RuntimeError:Invalidly formatted string stack')

sys.exit(1)

return [element for element in

string\_of\_stack[:startIndex].split(',') +

list(string\_of\_stack[startIndex+1:endIndex]) +

makeStack(str,string\_of\_stack[endIndex+1:])

if element != '']

else:

startIndex = string\_of\_stack.find('"')

endIndex = string\_of\_stack.find('"',startIndex+1)

if endIndex == -1:

print('RuntimeError:Invalidly formatted string stack')

sys.exit(1)

return [element for element in

string\_of\_stack[:startIndex].split(',') +

[string\_of\_stack[startIndex+1:endIndex]] +

makeStack(str,string\_of\_stack[endIndex+1:])

if element != '']

def concurrent(\*dfasNArgs):

dfas = [dfa(dfasNArgs[i\*2+1]) for i,dfa in enumerate(dfasNArgs[::2])]

finishedDfas = set()

while len(set(dfas) - finishedDfas):

for dfa in (set(dfas) - finishedDfas):

dfa.\_\_class\_\_.\_now()

for dfa in (set(dfas) - finishedDfas):

dfa.\_\_class\_\_.\_now = dfa.\_next

finishedDfas = set([dfa for dfa in dfas if dfa.\_returnVal is not None])

return [str(dfa.\_returnVal) for dfa in dfas]

def callDfa(dfaClass, \*args):

dfaInstance = dfaClass(args)

while dfaInstance.\_returnVal is None:

dfaClass.\_now()

dfaClass.\_now = dfaInstance.\_next

return dfaInstance.\_returnVal

class EOS:

def \_\_init\_\_(self):

return

def \_\_type\_\_(self):

return 'EOSType'

def \_\_str\_\_(self):

return 'EOS'

def \_\_eq\_\_(self,other):

return type(self) == type(other)

def \_\_ne\_\_(self,other):

return type(self) != type(other)

########END AUTOGENERATED FUNCTIONS ##############

########BEGIN DFA DEFINITIONS ##############

class \_main:

\_now = \_node\_start

def \_\_init\_\_(self,\*args):

try:

pass

except IndexError:

print('RuntimeError:Too few arguments provided to dfa "main"')

sys.exit(1)

self.\_returnVal = None

\_main.\_now = self.\_node\_start

self.\_next = None

self.halfPeriod = 400

while self.\_returnVal is None:

\_main.\_now()

\_main.\_now = self.\_next

return

def \_node\_start(self):

print "0"

concurrent(\_clock, [self.halfPeriod], \_TFF1, [], \_TFF2, [], \_TFF3, [], \_display, [])

self.\_returnVal = 1

self.\_next = None

\_dfa\_Dict["main"] = \_main

class \_TFF3:

\_now = \_node\_start

def \_\_init\_\_(self,\*args):

self.\_returnVal = None

\_TFF3.\_now = self.\_node\_start

self.\_next = None

return

def \_node\_high(self):

if(int(int(int(state("clock")=="rising") and int(state("TFF1")=="high")) and int(state("TFF2")=="high"))):

self.\_next = self.\_node\_start

return

if(1):

self.\_next = self.\_node\_high

return

def \_node\_start(self):

if(int(int(int(state("clock")=="rising") and int(state("TFF1")=="high")) and int(state("TFF2")=="high"))):

self.\_next = self.\_node\_high

return

if(1):

self.\_next = self.\_node\_start

return

\_dfa\_Dict["TFF3"] = \_TFF3

class \_TFF2:

\_now = \_node\_start

def \_\_init\_\_(self,\*args):

self.\_returnVal = None

\_TFF2.\_now = self.\_node\_start

self.\_next = None

return

def \_node\_high(self):

if(int(int(state("clock")=="rising") and int(state("TFF1")=="high"))):

self.\_next = self.\_node\_start

return

if(1):

self.\_next = self.\_node\_high

return

def \_node\_start(self):

if(int(int(state("clock")=="rising") and int(state("TFF1")=="high"))):

self.\_next = self.\_node\_high

return

if(1):

self.\_next = self.\_node\_start

return

\_dfa\_Dict["TFF2"] = \_TFF2

class \_TFF1:

\_now = \_node\_start

def \_\_init\_\_(self,\*args):

self.\_returnVal = None

\_TFF1.\_now = self.\_node\_start

self.\_next = None

return

def \_node\_high(self):

if(int(state("clock")=="rising")):

self.\_next = self.\_node\_start

return

if(1):

self.\_next = self.\_node\_high

return

def \_node\_start(self):

if(int(state("clock")=="rising")):

self.\_next = self.\_node\_high

return

if(1):

self.\_next = self.\_node\_start

return

\_dfa\_Dict["TFF1"] = \_TFF1

class \_clock:

\_now = \_node\_start

def \_\_init\_\_(self,\*args):

self.halfPeriod= args[0][0]

self.\_returnVal = None

\_clock.\_now = self.\_node\_start

self.\_next = None

return

def \_node\_high(self):

sleep(self.halfPeriod\*.001)

if(1):

self.\_next = self.\_node\_start

return

def \_node\_rising(self):

if(1):

self.\_next = self.\_node\_high

return

def \_node\_start(self):

sleep(self.halfPeriod\*.001)

if(1):

self.\_next = self.\_node\_rising

return

\_dfa\_Dict["clock"] = \_clock

class \_display:

\_now = \_node\_start

def \_\_init\_\_(self,\*args):

self.\_returnVal = None

\_display.\_now = self.\_node\_start

self.\_next = None

return

def \_node\_print7(self):

print "7"

if(1):

self.\_next = self.\_node\_start

return

def \_node\_print6(self):

print "6"

if(1):

self.\_next = self.\_node\_start

return

def \_node\_print5(self):

print "5"

if(1):

self.\_next = self.\_node\_start

return

def \_node\_print4(self):

print "4"

if(1):

self.\_next = self.\_node\_start

return

def \_node\_print3(self):

print "3"

if(1):

self.\_next = self.\_node\_start

return

def \_node\_print2(self):

print "2"

if(1):

self.\_next = self.\_node\_start

return

def \_node\_print1(self):

print "1"

if(1):

self.\_next = self.\_node\_start

return

def \_node\_print0(self):

print "0"

if(1):

self.\_next = self.\_node\_start

return

def \_node\_start(self):

if(int(int(int(int(state("clock")=="rising") and int(state("TFF1")=="high")) and int(state("TFF2")=="high")) and int(state("TFF3")=="high"))):

self.\_next = self.\_node\_print0

return

if(int(int(int(int(state("clock")=="rising") and int(state("TFF1")=="start")) and int(state("TFF2")=="start")) and int(state("TFF3")=="start"))):

self.\_next = self.\_node\_print1

return

if(int(int(int(int(state("clock")=="rising") and int(state("TFF1")=="high")) and int(state("TFF2")=="start")) and int(state("TFF3")=="start"))):

self.\_next = self.\_node\_print2

return

if(int(int(int(int(state("clock")=="rising") and int(state("TFF1")=="start")) and int(state("TFF2")=="high")) and int(state("TFF3")=="start"))):

self.\_next = self.\_node\_print3

return

if(int(int(int(int(state("clock")=="rising") and int(state("TFF1")=="high")) and int(state("TFF2")=="high")) and int(state("TFF3")=="start"))):

self.\_next = self.\_node\_print4

return

if(int(int(int(int(state("clock")=="rising") and int(state("TFF1")=="start")) and int(state("TFF2")=="start")) and int(state("TFF3")=="high"))):

self.\_next = self.\_node\_print5

return

if(int(int(int(int(state("clock")=="rising") and int(state("TFF1")=="high")) and int(state("TFF2")=="start")) and int(state("TFF3")=="high"))):

self.\_next = self.\_node\_print6

return

if(int(int(int(int(state("clock")=="rising") and int(state("TFF1")=="start")) and int(state("TFF2")=="high")) and int(state("TFF3")=="high"))):

self.\_next = self.\_node\_print7

return

if(1):

self.\_next = self.\_node\_start

return

\_dfa\_Dict["display"] = \_display

#######END DFA DEFINITIONS #############

if \_\_name\_\_ == '\_\_main\_\_':

\_main(sys.argv[1:] if len(sys.argv) else [])

# Lessons Learned

## What Oren Finard learned and advice for other teams

Big takeaway?  Do whatever you have to do to get to code generation.  Things make sense when you start trying to generate code.

Get good teammates, and get smart teammates.  And start early, and work consistently.  It's really not that bad.  Just don't leave it all to the last second.  But don't do that with anything.

## What Jackson Foley learned and advice for other teams.

Testing catalyzes productivity — begin testing as soon as possible.  You can start as soon as you have a scanner.  Always git pull, and check out a new branch before you start working on a new feature.  Git stash is your best friend.  Figuring out OCAML will save you far more time than trying to copy the code from previous projects and generalize it to your language.  Writing verbose error messages in the semantic check, and figuring out how to use the OCAML debugger saves hours.

## What Alexander Peters learned and advice for other teams.

This project really hit home the idea of creating something as a group. I've worked on group projects in research and in lab before, but never anything on this scale. With a project this large, but also this detailed, it is so important to keep constant communication between the group. Tools like group emails, group meetings and git were essential for us in arriving at the end result. This is different from projects I have worked on before because usually they can be modular and the work can be divided easily. While work delegation was also present here, the idea that everyone could do their individual part and then it all comes together at the end would be ridiculous with this project. Everyone's progress was constantly dependent on the progress of everyone else in the group, and because of that, I learned how to understand and build upon other people's work in a way I have never done before.

My advice to future teams would be to hold your LRM to very high standards. It is written early on in the class for a very good reason. The LRM represents a guide for what is and isn't allowed in your language. The first draft should be agreed upon, written and understood by the entire group. From this point on, it should be followed to the tee. This will keep your group on the same page as you move throughout the semester and multiple portions of the project are being written simultaneously. However, there is an interesting dichotomy here because the LRM will absolutely change. Therefore, the LRM should be taken as gospel up until the point where a change is needed. Then, this change should be made in the master copy of the LRM immediately, and the change should be communicated to the entire group immediately. Following this advice will ensure that the "vision" of your language is smooth across your entire group.

## What Brian Yamamoto learned and advice for other teams.

Always find ways to contribute, even if it’s not code**.** Subdivide tasks into pairs for maximum efficiency and time the weekly group meetings to occur shortly before and shortly after the weekly meetings with the TA to prepare questions and delegate tasks immediately after.

**Create a Facebook group for the project** – notifications will be swiftly communicated and you can even have some sort of version control on files posted there. **Rely on someone experienced with Git** to immediately create a repo and lay down ground rules on merging; learn how to use branches. **Keep a version log for the LRM and have one person maintain it** as soon as any changes are made within the language (changes will happen).

**Be friends with your group** – feel comfortable with admitting a lack of familiarity with certain sections of the project so that other team members know to explain it. Really soon into the semester the strengths and weaknesses of various members will be apparent – don’t attempt to divide the tasks to enforce that everyone contribute equally to a module.

This is the first project on which I really had to collaborate with others – it is both an illuminating and essential experience.

## What Zuokun Yu learned and advice for other teams.

What I learned:

1) Design work, on any scale, is difficult. As we were clarifying our language, we thought of many possible solutions for a particular problem. However, finding an optimal solution is non-trivial because a great way to solve a particular problem might not be best for the system as a whole. In other words, a greedy approach to problem solving isn’t sufficient.

2) Programming in pairs is awesome. I was most productive when working with someone else. Making choices was usually painless and having different angles on a problem was valuable. We tried working in larger groups as well (3+), but that wasn’t nearly as effective. It was harder to come to a consensus and harder to bring everyone on the same page.

Advice for other teams:

Test periodically, as parts of the compiler are written, on actual programs. This is also great because it ensures syntactic consistency early on. We had problems with artificial progress. Since our code compiled, we thought we were done with the various parts of the compiler. However, we ended up changing many components at the end while squashing bugs.

# Appendix

## Scanner Code (scanner.mll)

{ open Parser }

rule token =

parse [' ' '\t' '\r' '\n'] { token lexbuf } (\* Whitespace \*)

| "/\*" { comment lexbuf } (\* Multi-line comment \*)

| "//" { singleComment lexbuf } (\* Single-line comments \*)

| '(' { LPAREN }

| ')' { RPAREN }

| '{' { LBRACE }

| '}' { RBRACE }

| ';' { SEMI }

| ':' { COLON }

| ',' { COMMA }

| '.' { DOT }

| '+' { PLUS }

| '-' { MINUS }

| '\*' { STAR }

| '/' { DIVIDE }

| '%' { MOD }

| '=' { ASSIGN }

| "==" { EQ }

| '!' { NOT }

| "!=" { NEQ }

| "&&" { AND }

| "||" { OR }

| '<' { LT }

| "<-" { TRANS }

| "<=" { LEQ }

| '>' { GT }

| ">=" { GEQ }

| "return" { RETURN }

| "int" { INT }

| "float" { FLOAT }

| "string" { STRING }

| "void" { VOID }

| "DFA" { DFA }

| "stack" { STACK }

| "pop" { POP }

| "peek" { PEEK }

| "push" { PUSH }

| "EOS" { EOS }

| ['0'-'9']+ as lxm { INT\_LITERAL(int\_of\_string lxm) }

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

| '"' (('\\' \_ | [^'"'])\* as lxm) '"'{ STRING\_LITERAL(lxm) }

| ((['0'-'9']+('.'['0'-'9']\*|('.'?['0'-'9']\*'e'('+'|'-')?))['0'-'9']\*) |

(['0'-'9']\*('.'['0'-'9']\*|('.'?['0'-'9']\*'e'('+'|'-')?))['0'-'9']+))

as lxm { FLOAT\_LITERAL(float\_of\_string lxm) }

| eof { EOF }

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

and comment = parse

"\*/" { token lexbuf }

| \_ { comment lexbuf }

and singleComment = parse

'\n' { token lexbuf }

| \_ { singleComment lexbuf }

## Parser Code (parser.mly)

%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA RBRAC LBRAC COLON DOT

%token PLUS MINUS STAR DIVIDE ASSIGN STAR PUSH POP PEEK

%token NOT

%token EQ NEQ LT LEQ GT GEQ OR AND MOD

%token RETURN TRANS

%token DFA STACK

%token <int> INT\_LITERAL

%token <string> STRING\_LITERAL TYPE ID

%token <float> FLOAT\_LITERAL

%token EOF EOS

%token MAIN

%token STRING INT VOID FLOAT

%right ASSIGN

%left OR

%left AND

%right NOT

%left EQ NEQ LT GT LEQ GEQ

%left PLUS MINUS

%left STAR DIVIDE MOD

%right UMINUS

%left PUSH POP PEEK

%nonassoc LPAREN RPAREN LBRAC RBRAC

%start program

%type <Ast.program> program

%%

program:

{[]}

| dfa\_decl program { $1 :: $2 }

var\_type:

INT {Int}

|STRING {String}

|FLOAT {Float}

|VOID {Void}

ret\_type:

var\_type {Datatype($1)} |

STACK LT var\_type GT {Stacktype(Datatype($3))}

dfa\_decl:

ret\_type DFA ID LPAREN formals\_opt RPAREN LBRACE vdecl\_list node\_list RBRACE

{ { return = $1;

dfa\_name = Ident($3);

formals = $5;

var\_body = $8;

node\_body = $9}}

vdecl\_list:

{[]}

| vdecl vdecl\_list { $1 :: $2 }

vdecl:

var\_type ID SEMI { VarDecl(Datatype($1), Ident($2)) }

| var\_type ID ASSIGN expr SEMI { VarAssignDecl(Datatype($1), Ident($2), ExprVal($4))}

| STACK LT var\_type GT ID SEMI { VarDecl(Stacktype(Datatype($3)), Ident($5)) }

node\_list:

{[]}

| node node\_list { $1 :: $2 }

node:

ID LBRACE stmt\_list RBRACE { Node(Ident($1), $3) }

stmt\_list:

{[]}

| stmt stmt\_list { $1 :: $2 }

/\* TODO: add method calls \*/

stmt:

RETURN expr SEMI {Return($2)}

| ID TRANS expr SEMI {Transition(Ident($1),$3)}

| ID TRANS STAR SEMI {Transition(Ident($1),IntLit(1))} /\*Star evaluates to IntLit 1 because that's True in StateMap\*/

| vdecl {Declaration($1)}

| ID ASSIGN expr SEMI { Assign(Ident($1), $3) } /\*Assignment post-declaration\*/

| expr SEMI {Expr($1)}

| RETURN SEMI {Return(IntLit(1))}

formals\_opt:

{[]} /\*nothing\*/

| formal\_list { List.rev $1}

formal\_list:

param { [$1] }

| formal\_list COMMA param { $3 :: $1}

param:

var\_type ID { Formal(Datatype($1),Ident($2)) }

| STACK LT var\_type GT ID { Formal(Stacktype(Datatype($3)), Ident($5)) }

expr\_list:

{[]}

| expr COMMA expr\_list { $1 :: $3 }

| expr { [$1] }

expr:

INT\_LITERAL { IntLit($1) }

| STRING\_LITERAL { StringLit($1) }

| FLOAT\_LITERAL { FloatLit($1) }

| ID { Variable(Ident($1)) }

| EOS { EosLit }

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

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

| expr STAR expr { Binop($1, Mult, $3) }

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

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

| expr NEQ expr { Binop($1, Neq, $3) }

| expr LT expr { Binop($1, Lt, $3) }

| expr LEQ expr { Binop($1, Leq, $3) }

| expr GT expr { Binop($1, Gt,$3)}

| expr GEQ expr { Binop($1, Geq, $3) }

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

| expr AND expr { Binop($1, And, $3) }

| expr OR expr { Binop($1, Or , $3) }

| MINUS expr %prec UMINUS { Unop(Neg, $2) }

| NOT expr { Unop(Not, $2) }

| LPAREN expr RPAREN { $2 }

| ID DOT POP LPAREN RPAREN { Pop(Ident($1)) }

| ID DOT PUSH LPAREN expr RPAREN { Push(Ident($1), $5) }

| ID DOT PEEK LPAREN RPAREN { Peek(Ident($1)) }

| ID LPAREN expr\_list RPAREN {Call(Ident($1), $3) (\*call a sub dfa\*)}

## AST Code (ast.ml)

type var\_type = Int | String | Stack | Float | Void | Eos

type binop = Add | Sub | Mult | Div | Mod | Equal | Neq | And | Or| Lt | Leq | Gt | Geq

type unop = Not | Neg

type ident =

Ident of string

type datatype =

Datatype of var\_type |

Stacktype of datatype|

Eostype of var\_type

type expr =

IntLit of int |

StringLit of string |

FloatLit of float |

EosLit |

Variable of ident |

Unop of unop \* expr |

Binop of expr \* binop \* expr |

Call of ident \* expr list |

Push of ident \* expr |

Pop of ident |

Peek of ident

type value =

ExprVal of expr

and decl =

VarDecl of datatype \* ident |

VarAssignDecl of datatype \* ident \* value

type stmt =

Block of stmt list |

Expr of expr |

Declaration of decl |

Assign of ident \* expr |

Transition of ident \* expr |

Return of expr

type formal =

Formal of datatype \* ident

type node =

Node of ident \* stmt list

type dfa\_decl = {

return : datatype;

dfa\_name: ident;

formals : formal list;

var\_body : decl list;

node\_body : node list;

}

type program = dfa\_decl list

(\* "Pretty printed" version of the AST, meant to generate a MicroC program

from the AST. These functions are only for pretty-printing (the -a flag)

the AST and can be removed. \*)

let string\_of\_ident = function

Ident(l) -> l

let rec string\_of\_expr = function

IntLit(l) -> string\_of\_int l

| StringLit(l) -> l

| FloatLit(l) -> string\_of\_float l

| Variable(id) -> string\_of\_ident id

| Unop(o, e) ->

string\_of\_expr e ^ " " ^

(match o with

Not -> "!" |

Neg -> "-")

| Binop(e1, o, e2) ->

string\_of\_expr e1 ^ " " ^

(match o with

Add -> "+" | Sub -> "-" | Mult -> "\*" | Div -> "/"

| Equal -> "==" | Neq -> "!=" | Mod -> "%"

| Lt -> "<" | Leq -> "<=" | Gt -> ">" | Geq -> ">=" | And -> "&&" | Or -> "||" ) ^ " " ^

string\_of\_expr e2

| Call(id, e\_list) -> string\_of\_ident id ^ " " ^

"(" ^ String.concat ", " (List.map string\_of\_expr e\_list) ^ ")"

| Push(id, e) -> string\_of\_ident id ^ " " ^ string\_of\_expr e

| Pop(id) -> string\_of\_ident id

| Peek(id) -> string\_of\_ident id

| EosLit -> "EOSLIT"

let rec string\_of\_datatype = function

Datatype(vartype) ->

(match vartype with

Int -> "int" | String -> "String" | Stack -> "Stack" | Float -> "Float"

| Void -> "Void" | Eos -> "Eos"

)

| Stacktype(datatype) -> "Stack<" ^ string\_of\_datatype datatype ^ ">"

| Eostype(\_) -> "EOS"

let string\_of\_decl = function

VarDecl(dt, id) -> string\_of\_datatype dt ^ " " ^ string\_of\_ident id

| VarAssignDecl(dt,id,value) -> string\_of\_datatype dt ^ " " ^ string\_of\_ident id

^ " = " ^ (match value with

ExprVal(e) -> string\_of\_expr e)

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";

| Assign(id, expr) -> string\_of\_ident id ^ " = " ^ string\_of\_expr expr ^

";\n"

| Declaration(decl) -> string\_of\_decl decl

| Transition(id, expr) -> string\_of\_ident id ^ " <- (" ^ string\_of\_expr expr ^ ")"

let string\_of\_node = function

Node(id, stmtlist) -> string\_of\_ident id ^ " {\n" ^

String.concat "\n" (List.map string\_of\_stmt stmtlist) ^ "\n}"

let string\_of\_formal = function

Formal(dt, id) -> string\_of\_datatype dt ^ " " ^ string\_of\_ident id

let string\_of\_dfadecl dfadecl =

string\_of\_datatype dfadecl.return ^ " " ^ string\_of\_ident dfadecl.dfa\_name ^ "(" ^ String.concat ", " (List.map string\_of\_formal dfadecl.formals) ^ ")\n{\n" ^

String.concat "" (List.map string\_of\_decl dfadecl.var\_body) ^

String.concat "" (List.map string\_of\_node dfadecl.node\_body) ^

"}\n"

let string\_of\_program (program) =

String.concat "" (List.map string\_of\_dfadecl program)

## Semantic Check Code (semantic\_check.ml)

open Ast

open Sast

open Printf

exception Error of string

type symbol\_table = {

parent: symbol\_table option;

variables: (ident \* datatype \* value option) list;

}

type dfa\_table = {

dfas: (datatype \* ident \* formal list \* sstmt list \* snode list) list

}

type translation\_environment = {

return\_type: datatype;

return\_seen: bool;

location: string; (\*Where we are. DFA/Node\*)

node\_scope: symbol\_table;

dfa\_lookup: dfa\_table; (\*Table of all DFAs\*)

}

let get\_ident\_name ident = match ident with

Ident(n) -> n

let find\_dfa (dfa\_lookup: dfa\_table) name =

List.find (fun (\_,s,\_,\_,\_) -> s=name) dfa\_lookup.dfas

let basic\_math t1 t2 = match (t1, t2) with

(Float, Int) -> (Float, true)

| (Int, Float) -> (Float, true)

| (Int, Int) -> (Int, true)

| (Float, Float) -> (Int, true)

| (String, String) -> (String, true)

| (\_,\_) -> (Int, false)

let relational\_logic t1 t2 = match (t1, t2) with

(Int,Int) -> (Int,true)

| (Float,Float) -> (Int,true)

| (Int,Float) -> (Int,true)

| (Float,Int) -> (Int,true)

| (\_,\_) -> (Int, false)

let equal\_logic t1 t2 = match(t1,t2) with

(Int,Int) -> (Int,true)

| (Float,Float) -> (Int,true)

| (Int,Float) -> (Int,true)

| (Float,Int) -> (Int,true)

| (String,String) -> (Int,true)

| (\_,\_) -> (Int,false)

let rec get\_type\_from\_datatype = function

Datatype(t)->t

| Stacktype(ty) -> get\_type\_from\_datatype ty

| Eostype(t) -> Void

let get\_binop\_return\_value op typ1 typ2 =

let t1 = get\_type\_from\_datatype typ1 and t2 = get\_type\_from\_datatype typ2 in

let (t, valid) =

match op with

Add -> basic\_math t1 t2

| Sub -> basic\_math t1 t2

| Mult -> basic\_math t1 t2

| Div -> basic\_math t1 t2

| Mod -> basic\_math t1 t2

| Equal -> equal\_logic t1 t2

| Neq -> equal\_logic t1 t2

| Lt -> relational\_logic t1 t2

| Leq -> relational\_logic t1 t2

| Gt -> relational\_logic t1 t2

| Geq -> relational\_logic t1 t2

| And -> relational\_logic t1 t2

| Or -> relational\_logic t1 t2

in (Datatype(t), valid)

let get\_name\_type\_from\_formal env = function

Formal(datatype,ident) -> (ident,datatype,None)

let update\_variable env (name, datatype, value) =

let ((\_,\_,\_), location) =

try (fun node\_scope -> ((List.find (fun (s,\_,\_) -> s=name) node\_scope),1)) env.node\_scope.variables

with

Not\_found ->

try

let globalScope = match env.node\_scope.parent with

Some scope -> scope

| None -> raise(Error("No Global Scope"))

in

(fun node\_scope -> ((List.find (fun (s,\_,\_) -> s=name)

node\_scope),2)) globalScope.variables

with Not\_found -> raise(Error("Not Found exception in update\_variable"))in

let new\_envf =

match location with

1 -> (\*Node variables\*)

let new\_vars = List.map (fun (n, t, v) -> if(n=name) then (name, datatype, value) else (n, t, v)) env.node\_scope.variables in

let new\_sym\_table = {parent = env.node\_scope.parent; variables = new\_vars;} in

let new\_env = {env with node\_scope = new\_sym\_table} in

new\_env

| 2 -> (\*DFA variables\*)

let globalScope = match env.node\_scope.parent with

Some scope -> scope

| None -> raise(Error("No Global Scope2"))

in

let new\_vars = List.map (fun (n, t, v) -> if(n=name) then (name,

datatype, value) else (n, t, v)) globalScope.variables in

let new\_dfa\_sym\_table = {parent = None; variables = new\_vars;} in

let new\_node\_scope = {env.node\_scope with parent = Some(new\_dfa\_sym\_table);} in

let new\_env = {env with node\_scope = new\_node\_scope} in

new\_env

| \_ -> raise(Error("Undefined scope"))

in new\_envf

let find\_variable env name =

try List.find (fun(s,\_,\_) -> s = name) env.node\_scope.variables

with Not\_found ->

let globalScope = (match env.node\_scope.parent with

Some scope -> scope

|None -> raise(Error("No Global Scope3")))

in List.find(fun (s,\_,\_) -> s=name) globalScope.variables

let find\_local\_variable env name =

List.find (fun (s,\_,\_) -> s=name) env.node\_scope.variables

let rec check\_expr env e = match e with

IntLit(i) ->Datatype(Int)

| FloatLit(f) -> Datatype(Float)

| StringLit(s) -> Datatype(String)

| EosLit -> Eostype(Eos)

| Variable(v) ->

let (\_,s\_type,\_) = try find\_variable env v with

Not\_found ->

raise (Error("Undeclared Identifier ")) in s\_type

| Unop(u, e) ->

let t = check\_expr env e in

(match u with

\_ -> if t = Datatype(Int) then t else if t = Datatype(Float) then t

else

raise (Error("Cannot perform operation on " )))

| Binop(e1, b, e2) ->

let t1 = check\_expr env e1 and t2 = check\_expr env e2 in

let (t, valid) = get\_binop\_return\_value b t1 t2 in

if valid || e1 = EosLit || e2 = EosLit

then t else raise(Error("Incompatible types with binary operator"));

| Push(id, e) -> let (\_,t1,\_) = (find\_variable env id) and t2 =

check\_expr env e

in (if not (t1 = Stacktype(t2)) then (raise (Error("Mismatch in types for

assignment")))); t2

| Pop(id) -> let (\_,t1,\_) = (find\_variable env id) in t1

| Peek(id) -> let (\_,t1,\_) = (find\_variable env id) in t1

| Call(Ident("concurrent"), e\_list) ->

let dfaArgsList = List.filter( function

Call(\_,\_) -> false

| \_ -> true) e\_list

in

if dfaArgsList != [] then raise(Error("Not all arguments passed to

concurrent are dfas")) else Stacktype(Datatype(String))

| Call(id, e) -> try (let (dfa\_ret, dfa\_name, dfa\_args, dfa\_var\_body, dfa\_node\_body) = find\_dfa

env.dfa\_lookup id in

let el\_tys = List.map (fun exp -> check\_expr env exp) e in

let fn\_tys = List.map (fun dfa\_arg-> let (\_,ty,\_) =

get\_name\_type\_from\_formal env dfa\_arg in ty) dfa\_args in

if (

id = Ident("print") ||

id = Ident("concurrent" ) ||

id = Ident("itos") ||

id = Ident("stoi") ||

id = Ident("ftos") ||

id = Ident("stof") ||

id = Ident("sleep") ||

id = Ident("input") ||

id = Ident("state")

)

then dfa\_ret

else

if not (el\_tys = fn\_tys) then

raise (Error("Mismatching types in function call")) else

dfa\_ret)

with Not\_found ->

raise (Error("Undeclared Function: " ^ get\_ident\_name id))

let get\_node\_scope env name =

if env.location = "dfa" then DFAScope

else

try (let (\_,\_,\_) = List.find (fun (s,\_,\_) -> s=name) env.node\_scope.variables in NodeScope)

with Not\_found -> let globalScope = (match env.node\_scope.parent with

Some scope -> scope

|None -> raise(Error("No Global Scope4")))

in try (let (\_,\_,\_) = List.find(fun (s,\_,\_) -> s=name) globalScope.variables in DFAScope)

with Not\_found -> raise(Error("get\_node\_scope is failing"))

let rec get\_sexpr env e = match e with

IntLit(i) -> SIntLit(i, Datatype(Int))

| FloatLit(d) -> SFloatLit(d,Datatype(Float))

| StringLit(s) -> SStringLit(s,Datatype(String))

| Variable(id) -> SVariable(SIdent(id, get\_node\_scope env id), check\_expr env e)

| Unop(u,ex) -> SUnop(u, get\_sexpr env ex, check\_expr env e)

| Binop(e1,b,e2) -> SBinop(get\_sexpr env e1,b, get\_sexpr env e2,check\_expr env e)

| Call(id, ex\_list) -> let s\_ex\_list = List.map(fun exp -> get\_sexpr env

exp) ex\_list in SCall(SIdent(id,StateScope),s\_ex\_list, check\_expr env e)

| Push(id, ex) -> SPush(SIdent(id, get\_node\_scope env id),

get\_sexpr env ex,check\_expr env e)

| Pop(id) -> SPop(SIdent(id, get\_node\_scope env id), check\_expr env e)

| Peek(id) -> SPeek(SIdent(id, get\_node\_scope env id), check\_expr env e)

| EosLit -> SEosLit

let get\_sval env = function

ExprVal(expr) -> SExprVal(get\_sexpr env expr)

let get\_datatype\_from\_val env = function

ExprVal(expr) -> check\_expr env expr

let get\_sdecl env decl =

let scope = match env.node\_scope.parent with

Some(\_) -> NodeScope

|None -> DFAScope

in match decl with

VarDecl(datatype, ident) -> (SVarDecl(datatype, SIdent(ident, scope)), env)

| VarAssignDecl(datatype, ident, value) ->

let sv = get\_sval env value in

(SVarAssignDecl(datatype, SIdent(ident, scope), sv), env)

let get\_name\_type\_from\_decl decl = match decl with

VarDecl(datatype, ident) -> (ident, datatype)

| VarAssignDecl(datatype,ident,value) -> (ident,datatype)

let get\_name\_type\_val\_from\_decl decl = match decl with

VarDecl(datatype, ident) -> (ident, datatype, None)

| VarAssignDecl(datatype, ident, value) -> (ident, datatype, Some(value))

let get\_name\_type\_from\_var env = function

VarDecl(datatype,ident) -> (ident,datatype,None)

| VarAssignDecl(datatype,ident,value) -> (ident,datatype,Some(value))

let add\_to\_var\_table env name t v =

let new\_vars = (name,t, v)::env.node\_scope.variables in

let new\_sym\_table = {parent = env.node\_scope.parent; variables = new\_vars;} in

let new\_env = {env with node\_scope = new\_sym\_table} in

new\_env

let check\_assignments type1 type2 = match (type1, type2) with

(Int, Int) -> true

|(Float, Float) -> true

|(Int, Float) -> true

|(Float, Int) -> true

|(String, String) -> true

|(\_,\_) -> false

let match\_var\_type env v t =

let(name,ty,value) = find\_variable env v in

if(t<>ty) then false else true

let check\_final\_env env =

(if(false = env.return\_seen && env.return\_type <> Datatype(Void)) then

raise (Error("Missing Return Statement")));

true

(\* Default Table and Environment Initializations \*)

let empty\_table\_initialization = {parent=None; variables =[];}

let empty\_dfa\_table\_initialization = {

dfas=[

(\*The state() function to get states of concurrently running dfas\*)

(Datatype(String), Ident("state"),

[Formal(Datatype(String),Ident("dfa"))],[], []);

(\*The built-in print function (only prints strings)\*)

(Datatype(Void), Ident("print"),

[Formal(Datatype(String),Ident("str"))],[], []);

(\*The built-in sleep function\*)

(Datatype(Void), Ident("sleep"), [Formal(Datatype(Int),Ident("ms"))],[],

[]);

(\*The built-in int-to-string conversion function\*)

(Datatype(String), Ident("itos"),

[Formal(Datatype(Int),Ident("int"))],[], []);

(\*The built-in string-to-int conversion function\*)

(Datatype(Int), Ident("stoi"),

[Formal(Datatype(String),Ident("str"))],[],[]);

(\*The built-in float-to-string converstion function\*)

(Datatype(String), Ident("ftos"),

[Formal(Datatype(Float),Ident("float"))],[],[]);

(\*The built-in string-to-float converstion function\*)

(Datatype(Float), Ident("stof"),

[Formal(Datatype(String),Ident("str"))],[],[]);

(\*The built-in get-user-input function\*)

(Datatype(String), Ident("input"),[],[],[]);

(\*The built-in 'get state' function for concurrently running dfas \*)

(Datatype(String), Ident("state"),

[Formal(Datatype(String),Ident("dfa"))],[],[]);

(\*The built-in concurrent string\*)

(Stacktype(Datatype(String)), Ident("concurrent"), [] ,[], []) (\*how to

check formals\*)

]}

let empty\_environment = {return\_type = Datatype(Void); return\_seen = false;

location="in\_dfa"; node\_scope = {empty\_table\_initialization with parent =

Some(empty\_table\_initialization)}; dfa\_lookup = empty\_dfa\_table\_initialization}

let find\_global\_variable env name =

let globalScope = match env.node\_scope.parent with

Some scope -> scope

| None -> raise (Error("No global scope")) in

try List.find (fun (s,\_,\_) -> s=name) globalScope.variables

with Not\_found -> raise (Error("error in find\_global\_variable"))

let rec check\_stmt env stmt = match stmt with

| Block(stmt\_list) ->

let new\_env=env in

let getter(env,acc) s =

let (st, ne) = check\_stmt env s in

(ne, st::acc) in

let (ls,st) = List.fold\_left(fun e s ->

getter e s) (new\_env,[]) stmt\_list in

let revst = List.rev st in

(SBlock(revst),ls)

| Expr(e) ->

let \_ = check\_expr env e in

(SSExpr(get\_sexpr env e),env)

| Return(e) ->

let type1=check\_expr env e in

if env.return\_type <> Datatype(Void) && type1 <> env.return\_type then

raise (Error("Incompatible Return Type"));

let new\_env = {env with return\_seen=true} in

(SReturn(get\_sexpr env e), new\_env)

| Ast.Declaration(decl) ->

let (name, ty) = get\_name\_type\_from\_decl decl in

let ((\_,dt,\_),found) = try (fun f -> ((f env name),true)) find\_local\_variable with

Not\_found ->

((name,ty,None),false) in

let ret = if(found=false) then

match decl with

VarDecl(\_,\_) ->

let (sdecl,\_) = get\_sdecl env decl in

let (n, t, v) = get\_name\_type\_val\_from\_decl decl in

let new\_env = add\_to\_var\_table env n t v in

(SDeclaration(sdecl), new\_env)

| VarAssignDecl(dt, id, value) ->

let t1 = get\_type\_from\_datatype(dt) and t2 = get\_type\_from\_datatype(get\_datatype\_from\_val env value) in

if(t1=t2) then

let (sdecl,\_) = get\_sdecl env decl in

let (n, t, v) = get\_name\_type\_val\_from\_decl decl in

let new\_env = add\_to\_var\_table env n t v in

(SDeclaration(sdecl), new\_env)

else raise (Error("Type mismatch"))

else

raise (Error("Multiple declarations")) in ret

| Ast.Assign(ident, expr) ->

let (\_, dt, \_) = try find\_variable env ident with Not\_found -> raise (Error("Uninitialized variable")) in

let t1 = get\_type\_from\_datatype dt

and t2 = get\_type\_from\_datatype(check\_expr env expr) in

if( not(t1=t2) ) then

raise (Error("Mismatched type assignments"));

let sexpr = get\_sexpr env expr in

let new\_env = update\_variable env (ident,dt,Some((ExprVal(expr)))) in

(SAssign(SIdent(ident, get\_node\_scope env ident), sexpr), new\_env)

| Transition(idState,ex) ->

let t=get\_type\_from\_datatype(check\_expr env ex) in

if not(t=Int) then

raise(Error("Improper Transition Expression Datatype")) else

(STransition(SIdent(idState, StateScope), get\_sexpr env ex), env)

let get\_sstmt\_list env stmt\_list =

List.fold\_left (fun (sstmt\_list,env) stmt ->

let (sstmt, new\_env) = check\_stmt env stmt in

(sstmt::sstmt\_list, new\_env)) ([],env) stmt\_list

let get\_svar\_list env var\_list =

List.fold\_left (fun (svar\_list,env) var ->

let stmt = match var with

decl -> Ast.Declaration(var)

in

let (svar, new\_env) = check\_stmt env stmt in

(svar::svar\_list, new\_env)) ([],env) var\_list

let get\_snode\_body env node\_list =

List.fold\_left (fun (snode\_list, dfa\_env) raw\_node ->

let node\_sym\_tab = {parent = Some(dfa\_env.node\_scope); variables = [];} in

let node\_env = {dfa\_env with node\_scope = node\_sym\_tab;} in

match raw\_node with

Node((Ident(name), node\_stmt\_list)) ->

let transCatchAllList = List.filter( function

Transition(\_,IntLit(1)) -> true

| \_ -> false) node\_stmt\_list in

let transList = List.filter( function

Transition(\_,\_) -> true

| \_ -> false) node\_stmt\_list in

let retList = List.filter (function

Return(\_) -> true

| \_ -> false) node\_stmt\_list in

if retList != [] && transList != [] then

raise(Error("Return statements and Transitions are

mutually exclusive"))

else

let block =

let node\_block = Block(node\_stmt\_list) in

let (snode\_block, new\_node\_env) = check\_stmt node\_env node\_block in

let new\_dfa\_node\_scope = (match new\_node\_env.node\_scope.parent

with

Some(scope) -> scope

| None-> raise(Error("Snode check returns no dfa scope")))

in

let new\_dfa\_env = {dfa\_env with node\_scope =

new\_dfa\_node\_scope; return\_seen = new\_node\_env.return\_seen} in

(SNode(SIdent(Ident(name), NodeScope), snode\_block)::snode\_list,

new\_dfa\_env) in

if retList == [] then

if transCatchAllList != [] then

block

else raise(Error("No catch all"))

else

block

) ([],env) node\_list

let add\_dfa env sdfa\_decl =

let dfa\_table = env.dfa\_lookup in

let old\_dfas = dfa\_table.dfas in

match sdfa\_decl with

SDfa\_Decl(sdfastr, datatype) ->

let dfa\_name = sdfastr.sdfaname in

let dfa\_type = get\_type\_from\_datatype sdfastr.sreturn in

let dfa\_formals = sdfastr.sformals in

let dfa\_var\_body = sdfastr.svar\_body in

let dfa\_node\_body = sdfastr.snode\_body in

let new\_dfas = (Datatype(dfa\_type), dfa\_name, dfa\_formals,

dfa\_var\_body, dfa\_node\_body)::old\_dfas in

let new\_dfa\_lookup = {dfas = new\_dfas} in

let final\_env = {env with dfa\_lookup = new\_dfa\_lookup} in

final\_env

let check\_for\_start node\_list =

let allNodes = List.fold\_left (fun (name\_list) raw\_node ->

match raw\_node with

Node((Ident(name), node\_stmt\_list)) ->

name::name\_list) ([]) node\_list

in if List.mem "start" allNodes = false then raise(Error("No start state in

node"))

let transition\_check node\_list =

let allNodes = List.fold\_left (fun (name\_list) raw\_node ->

match raw\_node with

Node((Ident(name), node\_stmt\_list)) ->

name::name\_list) ([]) node\_list

in let statements = List.map (fun raw\_node ->

match raw\_node with

Node((Ident(name), node\_stmt\_list)) ->

List.map (fun x -> x) node\_stmt\_list) node\_list

in let flat = List.flatten statements

in let states = List.fold\_left (fun (states\_list) stmt ->

match stmt with

Transition(Ident(id), ex) ->

id::states\_list

| \_ -> []) ([]) flat

in List.map (fun id -> try (List.mem id allNodes) with Not\_found ->

raise(Error("Invalid state transition"))) states

let check\_dfa env dfa\_declaration =

try(let (\_,\_,\_,\_,\_) = find\_dfa env.dfa\_lookup dfa\_declaration.dfa\_name in

raise(Error("DFA already declared"))) with

Not\_found ->

let dfaFormals = List.fold\_left(fun a vs -> (get\_name\_type\_from\_formal env vs)::a)[] dfa\_declaration.formals in

let dfa\_env = {return\_type = dfa\_declaration.return; return\_seen = false;

location = "dfa"; node\_scope = {parent = None; variables = dfaFormals;};

dfa\_lookup = env.dfa\_lookup} in

let \_ = check\_for\_start dfa\_declaration.node\_body in

let \_ = transition\_check dfa\_declaration.node\_body in

let (global\_var\_decls, penultimate\_env) = get\_svar\_list dfa\_env

dfa\_declaration.var\_body in

let location\_change\_env = {penultimate\_env with location = "node"} in

let (checked\_node\_body, final\_env) = get\_snode\_body location\_change\_env

dfa\_declaration.node\_body in

let \_ =check\_final\_env final\_env in

let sdfadecl = ({sreturn = dfa\_declaration.return; sdfaname =

dfa\_declaration.dfa\_name; sformals = dfa\_declaration.formals; svar\_body =

global\_var\_decls; snode\_body = checked\_node\_body}) in

(SDfa\_Decl(sdfadecl,dfa\_declaration.return), env)

let initialize\_dfas env dfa\_list =

let (typed\_dfa,last\_env) = List.fold\_left

(fun (sdfadecl\_list,env) dfa-> let (sdfadecl, \_) = check\_dfa env dfa in

let final\_env = add\_dfa env sdfadecl in

(sdfadecl::sdfadecl\_list, final\_env))

([],env) dfa\_list in (typed\_dfa,last\_env)

let check\_main env str =

let id = Ident(str) in

let (dt, \_, \_, \_, \_) = try(find\_dfa env.dfa\_lookup id)

with Not\_found -> raise(Error("Need DFA called main")) in

if dt <> Datatype(Void) then

raise(Error("main DFA needs void return type"))

let check\_program program =

let dfas = program in

let env = empty\_environment in

let (typed\_dfas, new\_env) = initialize\_dfas env dfas in

let (\_) = check\_main new\_env "main" in

Prog(typed\_dfas)

## SAST Code (sast.mli)

open Ast

type scope =

NodeScope

| DFAScope

| StateScope

type sident =

SIdent of ident \* scope

type sval =

SExprVal of sexpr

and sexpr =

SIntLit of int \* datatype

| SFloatLit of float \* datatype

| SStringLit of string \* datatype

| SVariable of sident \* datatype

| SUnop of unop \* sexpr \* datatype

| SBinop of sexpr \* binop \* sexpr \* datatype

| SCall of sident \* sexpr list \* datatype

| SPeek of sident \* datatype

| SPop of sident \* datatype

| SPush of sident \* sexpr \* datatype

| SEosLit

type sdecl =

SVarDecl of datatype \* sident

| SVarAssignDecl of datatype \* sident \* sval

type sstmt =

SBlock of sstmt list

| SSExpr of sexpr

| SReturn of sexpr

| SDeclaration of sdecl

| SAssign of sident \* sexpr

| STransition of sident \* sexpr

type snode =

SNode of sident \* sstmt

type sdfastr = {

sreturn: datatype;

sdfaname : ident;

sformals : formal list;

svar\_body : sstmt list;

snode\_body: snode list;

}

type sdfa\_decl =

SDfa\_Decl of sdfastr \* datatype

type sprogram =

Prog of sdfa\_decl list

## Code Generator Code (gen\_python.ml)

open Ast

open Sast

open Printf

exception Error of string

let py\_start =

"#########BEGIN AUTOGENERATED FUNCTIONS ###########

from time import sleep

import sys

\_dfa\_Dict = dict()

def \_node\_start():

#do nothing: just exist as a function for the dfas to initially

#point to with `dfa.\_now` so that we can have correct formatting in

#state()

return

def state(dfa):

return \_dfa\_Dict[dfa].\_now.\_\_name\_\_[6:]

def makeStack(stacktype,string\_of\_stack):

if stacktype != str:

return map(stacktype,string\_of\_stack.replace('[','').replace(']','').split(','))

else:

if \"'\" not in string\_of\_stack and '\"' not in string\_of\_stack:

return map(stacktype, string\_of\_stack.split(','))

elif ('\"' not in string\_of\_stack or

(string\_of\_stack.find(\"'\") < string\_of\_stack.find('\"') and

string\_of\_stack.find(\"'\") != -1)):

startIndex = string\_of\_stack.find(\"'\")

endIndex = string\_of\_stack.find(\"'\",startIndex+1)

if endIndex == -1:

print('RuntimeError:Invalidly formatted string stack')

sys.exit(1)

return [element for element in

string\_of\_stack[:startIndex].split(',') +

list(string\_of\_stack[startIndex+1:endIndex]) +

makeStack(str,string\_of\_stack[endIndex+1:])

if element != '']

else:

startIndex = string\_of\_stack.find('\"')

endIndex = string\_of\_stack.find('\"',startIndex+1)

if endIndex == -1:

print('RuntimeError:Invalidly formatted string stack')

sys.exit(1)

return [element for element in

string\_of\_stack[:startIndex].split(',') +

[string\_of\_stack[startIndex+1:endIndex]] +

makeStack(str,string\_of\_stack[endIndex+1:])

if element != '']

def concurrent(\*dfasNArgs):

dfas = [dfa(dfasNArgs[i\*2+1]) for i,dfa in enumerate(dfasNArgs[::2])]

finishedDfas = set()

while len(set(dfas) - finishedDfas):

for dfa in (set(dfas) - finishedDfas):

dfa.\_\_class\_\_.\_now()

for dfa in (set(dfas) - finishedDfas):

dfa.\_\_class\_\_.\_now = dfa.\_next

finishedDfas = set([dfa for dfa in dfas if dfa.\_returnVal is not None])

return [str(dfa.\_returnVal) for dfa in dfas]

def callDfa(dfaClass, \*args):

dfaInstance = dfaClass(args)

while dfaInstance.\_returnVal is None:

dfaClass.\_now()

dfaClass.\_now = dfaInstance.\_next

return dfaInstance.\_returnVal

class EOS:

def \_\_init\_\_(self):

return

def \_\_type\_\_(self):

return 'EOSType'

def \_\_str\_\_(self):

return 'EOS'

def \_\_eq\_\_(self,other):

return type(self) == type(other)

def \_\_ne\_\_(self,other):

return type(self) != type(other)

########END AUTOGENERATED FUNCTIONS ##############

########BEGIN DFA DEFINITIONS ##############

"

let py\_end =

"

#######END DFA DEFINITIONS #############

if \_\_name\_\_ == '\_\_main\_\_':

\_main(sys.argv[1:] if len(sys.argv) else [])

"

let print = "print"

let def = "def"

let return = "return"

let gen\_id = function

Ident(id) -> id

let gen\_sid = function

SIdent(id,dt) -> id

let rec gen\_tabs n = match n with

0 -> ""

|1 -> "\t"

| \_ -> "\t"^gen\_tabs (n-1)

let get\_sident\_name = function

SIdent(id,scope) -> match scope with

NodeScope -> "" ^ gen\_id id

|DFAScope -> "self." ^ gen\_id id

|StateScope -> "" ^ gen\_id id

let gen\_unop = function

Neg -> "-"

| Not -> "not "

let gen\_binop = function

Add -> "+"

| Sub -> "-"

| Mult -> "\*"

| Div -> "/"

| Equal -> "=="

| Neq -> "!="

| Lt -> "<"

| Leq -> "<="

| Gt -> ">"

| Geq -> ">="

| Mod -> "%"

| And -> " and "

| Or -> " or "

let gen\_var\_type = function

Int -> "int"

|Float -> "float"

|String -> "str"

|Eos -> "type(EOS())"

|Void -> "Void"

|Stack -> "Stack"

let gen\_formal formal = match formal with

Formal(datatype, id) -> gen\_id id

let rec gen\_sexpr sexpr = match sexpr with

SIntLit(i, d) -> string\_of\_int i

| SFloatLit(f, d) -> string\_of\_float f

| SStringLit(s, d) -> "\"" ^ s ^ "\""

| SVariable(sident, d) -> get\_sident\_name sident

| SUnop(unop, sexpr, d) -> gen\_unop unop ^ "(" ^ gen\_sexpr sexpr ^ ")"

| SBinop(sexpr1, binop, sexpr2, d) ->

(match d with

Datatype(String) ->

(match binop with

Add -> "(" ^ gen\_sexpr sexpr1 ^ gen\_binop binop ^ gen\_sexpr sexpr2

^ ")"

| \_ -> "int(" ^ gen\_sexpr sexpr1 ^ gen\_binop binop ^ gen\_sexpr sexpr2 ^ ")")

| \_ -> "int(" ^ gen\_sexpr sexpr1 ^ gen\_binop binop ^ gen\_sexpr sexpr2 ^ ")")

| SPeek(sident,dt) -> let stackName = get\_sident\_name sident in

"(" ^ stackName ^ "[0] if len(" ^ stackName ^") else EOS())"

| SPop(sident,dt) -> let stackName = get\_sident\_name sident in

"(" ^ stackName ^ ".pop(0) if len(" ^ stackName ^ ") else EOS())"

| SPush(sident,sexpr,dt) -> let stackName = get\_sident\_name sident in

stackName ^ ".insert(0," ^ gen\_sexpr sexpr ^ ")"

| SEosLit -> "EOS()"

| SCall(sident, sexpr\_list, d) -> match gen\_id (gen\_sid sident) with

"print" -> "print " ^ gen\_sexpr\_list sexpr\_list

| "state" -> "state(" ^ gen\_sexpr\_list sexpr\_list ^ ")"

| "sleep" -> "sleep(" ^ gen\_sexpr\_list sexpr\_list ^ "\*.001)"

| "itos" -> "str(" ^ gen\_sexpr\_list sexpr\_list ^ ")"

| "ftos" -> "str(" ^ gen\_sexpr\_list sexpr\_list ^ ")"

| "stof" -> "float(" ^ gen\_sexpr\_list sexpr\_list ^ ")"

| "stoi" -> "int(" ^ gen\_sexpr\_list sexpr\_list ^ ")"

| "input" -> "raw\_input(" ^ gen\_sexpr\_list sexpr\_list ^ ")"

| "concurrent" -> "concurrent(" ^ gen\_concurrency\_list sexpr\_list ^")"

| \_ -> let dfaname = get\_sident\_name sident in

"callDfa(\_" ^ dfaname ^ "," ^ gen\_sexpr\_list sexpr\_list ^ ")"

and gen\_sstmt sstmt tabs = match sstmt with

SBlock(sstmt\_list) -> gen\_sstmt\_list sstmt\_list tabs

| SSExpr(sexpr) -> gen\_tabs tabs ^ gen\_sexpr sexpr ^ "\n"

| SReturn(sexpr) -> gen\_tabs tabs ^ "self.\_returnVal = " ^ gen\_sexpr sexpr ^ "\n" ^

gen\_tabs tabs ^ "self.\_next = None\n"

| SDeclaration(sdecl) -> (match sdecl with

SVarDecl(dt,sident) -> (match dt with

Stacktype(\_) -> gen\_tabs tabs ^ get\_sident\_name sident ^ "= list()\n"

|Datatype(\_) -> gen\_tabs tabs ^ get\_sident\_name sident ^ "= None\n"

|Eostype(\_) -> "type(EOS())")

|SVarAssignDecl(dt,sident,SExprVal(sval)) -> gen\_tabs tabs ^

get\_sident\_name sident ^ " = " ^ gen\_sexpr sval ^ "\n")

| SAssign(sident, sexpr) -> gen\_tabs tabs ^ get\_sident\_name sident ^ " = " ^

gen\_sexpr sexpr ^ "\n"

| STransition(sident, sexpr) -> gen\_tabs tabs ^ "if(" ^ gen\_sexpr sexpr ^ "):\n" ^

gen\_tabs (tabs+1) ^ "self.\_next = self.\_node\_" ^ get\_sident\_name sident ^ "\n" ^

gen\_tabs (tabs+1) ^ "return\n"

and gen\_sdecl decl = match decl with

SVarDecl(datatype, sident) -> "# Variable declared without assignment: " ^ get\_sident\_name sident ^ "\n"

| SVarAssignDecl(datatype, sident, value) -> get\_sident\_name sident ^ " = " ^ gen\_svalue value ^ "\n"

and gen\_svalue value = match value with

SExprVal(sexpr) -> gen\_sexpr sexpr

and gen\_formal\_list formal\_list = match formal\_list with

[] -> ""

| h::[] -> gen\_formal h

| h::t -> gen\_formal h ^ ", " ^ gen\_formal\_list t

and gen\_sstmt\_list sstmt\_list tabs = match sstmt\_list with

[] -> ""

| h::[] -> gen\_sstmt h tabs

| h::t -> gen\_sstmt h tabs ^ gen\_sstmt\_list t tabs

and gen\_sexpr\_list sexpr\_list = match sexpr\_list with

[] -> ""

| h::[] -> gen\_sexpr h

| h::t -> gen\_sexpr h ^ ", " ^ gen\_sexpr\_list t

and gen\_concurrent\_dfa sexpr = match sexpr with

SCall(sident,sexpr\_list,d) -> "\_" ^ get\_sident\_name sident ^ ", [" ^

gen\_sexpr\_list sexpr\_list ^ "]"

| \_ -> ""

and gen\_concurrency\_list sexpr\_list = match sexpr\_list with

[] -> ""

| h::[] -> gen\_concurrent\_dfa h

| h::t -> gen\_concurrent\_dfa h ^ ", " ^ gen\_concurrency\_list t

let rec gen\_node\_list snode\_body = match snode\_body with

[] -> ""

| SNode(sident,snode\_block)::rst -> gen\_tabs 1 ^ "def \_node\_" ^ gen\_id (gen\_sid sident) ^ "(self):\n" ^

gen\_sstmt snode\_block 2 ^ gen\_node\_list rst

let rec get\_type\_from\_datatype = function

Datatype(t) -> t

| Stacktype(ty) -> get\_type\_from\_datatype ty

| Eostype(e) -> e

let gen\_formal\_typeCast dt id = match dt with

Stacktype(Stacktype(\_)) -> raise(Error("Cannot have a formal of Stacks of Stacks"))

|Stacktype(Eostype(\_)) -> raise(Error("Cannot have a formal of Stacks of EOS"))

|Stacktype(Datatype(Eos)) -> raise(Error("Cannot have a formal of Stacks of EOS"))

|Stacktype(Datatype(Void)) -> raise(Error("Cannot have a formal of Stacks of Void"))

|Stacktype(Datatype(vartype)) -> "makeStack(" ^ gen\_var\_type vartype ^ ","

| \_ -> match get\_type\_from\_datatype dt with

Int -> "int("

|Float -> "float("

|String -> "("

|Void -> raise(Error("A formal cannot be of type Void"))

|Eos -> raise(Error("A formal cannot be of type Eos"))

|Stack -> raise(Error("A formal cannot be of type Stack"))

let rec gen\_unpacked\_formal\_list sformals index tabs = match sformals with

[] -> ""

|Formal(dt,id)::rst -> gen\_tabs tabs ^ "self." ^ gen\_id id ^

"= args[0][" ^ string\_of\_int index ^ "]\n" ^

gen\_unpacked\_formal\_list rst(index + 1) tabs

let rec gen\_unpacked\_main\_formal\_list sformals index tabs = match sformals with

[] -> ""

|Formal(dt,id)::rst ->

gen\_tabs tabs ^ "self." ^ gen\_id id ^ "=" ^ gen\_formal\_typeCast dt id ^

"args[0][" ^ string\_of\_int index ^ "])\n" ^ gen\_unpacked\_main\_formal\_list rst (index+1) tabs

let get\_main\_dfa\_str name = match name with

"main" -> gen\_tabs 2 ^ "while self.\_returnVal is None:\n" ^ gen\_tabs 3 ^

"\_main.\_now()\n" ^ gen\_tabs 3 ^ "\_main.\_now = self.\_next\n"

| \_ -> ""

let gen\_sdfa\_str sdfa\_str =

"class \_" ^ gen\_id sdfa\_str.sdfaname ^ ":\n" ^

gen\_tabs 1 ^ "\_now = \_node\_start\n" ^

gen\_tabs 1 ^ "def \_\_init\_\_(self,\*args):\n" ^

let protectedIndexArgs = match gen\_id sdfa\_str.sdfaname with

"main" ->

gen\_tabs 2 ^ "try:\n" ^

gen\_unpacked\_main\_formal\_list sdfa\_str.sformals 0 3 ^

gen\_tabs 3 ^ "pass\n" ^

gen\_tabs 2 ^ "except IndexError:\n" ^

gen\_tabs 3 ^ "print('RuntimeError:Too few arguments provided to dfa \"main\"')\n" ^

gen\_tabs 3 ^ "sys.exit(1)\n"

| \_ -> gen\_unpacked\_formal\_list sdfa\_str.sformals 0 2

in protectedIndexArgs ^

gen\_tabs 2 ^ "self.\_returnVal = None\n" ^

gen\_tabs 2 ^ "\_" ^ (gen\_id sdfa\_str.sdfaname) ^ ".\_now = self.\_node\_start\n" ^

gen\_tabs 2 ^ "self.\_next = None\n" ^

gen\_sstmt\_list sdfa\_str.svar\_body 2 ^

get\_main\_dfa\_str (gen\_id sdfa\_str.sdfaname) ^ gen\_tabs 2 ^ "return\n" ^

gen\_node\_list sdfa\_str.snode\_body ^ "\n" ^

"\_dfa\_Dict[\"" ^ gen\_id sdfa\_str.sdfaname ^ "\"] = \_" ^gen\_id sdfa\_str.sdfaname ^ "\n"

let gen\_sdfa\_decl = function

SDfa\_Decl(sdfa\_str, dt) -> gen\_sdfa\_str sdfa\_str

let gen\_sdfa\_decl\_list sdfa\_decl\_list =

String.concat "\n" (List.map gen\_sdfa\_decl sdfa\_decl\_list)

let gen\_program = function

Prog(sdfa\_decl\_list) -> py\_start ^ gen\_sdfa\_decl\_list sdfa\_decl\_list ^ py\_end

## Compiler Code (compiler.ml)

open Semantic\_check

open Gen\_python

open Sys

let \_ =

let lexbuf = Lexing.from\_channel stdin in

let ast = Parser.program Scanner.token lexbuf in

let sast = Semantic\_check.check\_program ast in

let code = gen\_program sast in

let output = open\_out (Sys.argv.(1) ^ ".py") in

output\_string output code