Programming Language Final Report

StateMap

Oren Finard, Jackson Foley, Alex Peters,
Brian Yamamoto, Zuokun Yu

Fall 2014

1 CONTENTS

2 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.

2.1 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.

3 Language Tutorial

3.1 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.

3.2 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.

3.3 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.

3.4 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.

3.5 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.

4 LANGUAGE MANUAL

4.1 LEXICAL CONVENTIONS

4.1.1 Comments

Both C and C++ style comments are supported.

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

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

4.1.2 Identifiers (Names)

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

4.1.3 Keywords

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

return int float string void DFA main stack start

4.1.4 Constants

There are several types of constants, as follows:

4.1.4.1 Integer Constants

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

Valid: 42, 0, -13

Invalid: 042, +13, 00, .25

4.1.4.2 Float Constants

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

Valid: .3, 1.34, -2.3

Invalid: 42, 0

4.1.4.3 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.

4.1.5 Strings

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

Valid: "hello world", " ", "42", "he told me \"yo\"", ""

Invalid: "He asked "Do you have your towel?""

4.1.6 Punctuation

4.1.6.1 Braces

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

4.1.6.2 Parenthesis

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

4.1.6.3 Semicolon

Used to denote the end of a statement.

4.1.6.4 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);
```

4.1.7 Operators

4.1.7.1 Arithmetic

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

4.1.7.2 Assignment

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

4.1.7.3 Comparison

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

4.1.7.4 Boolean Evaluation

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

4.1.8 Whitespace

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

4.2 Syntax Notation

4.2.1 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) {}
```

4.2.2 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.

4.2.3 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.

4.2.4 Expressions

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

4.2.4.1 Literals and Operators

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

{Id}

{Id} {Operator} {Id}

4.2.4.2 Method Calls

Method calls that return a value will evaluate as expressions.

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

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

4.2.5 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.

4.2.5.1 Declaration

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

```
{TYPE}{ID};
int i;
stack<double> s, char c, string s;
```

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

4.2.5.2 Assignment

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

```
{Type}{Id} = {Expression}
int i = 4;
double d = 3.0, string s = "hello";
```

4.2.5.3 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);
```

4.2.5.4 Transition

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

```
{State}<-*

{State}<-{Expression}

state1 <- foo >= bar;
```

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

4.2.5.5 Return

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

```
return {expression};

return i < 4;  // returns an int 1
```

4.2.6 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.

4.3 TYPE

4.3.1 Type Declaration

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

4.3.2 Fundamental TYpes

4.3.2.1 int

A 32-bit integer.

4.3.2.2 Float

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

4.3.2.3 string

A sequence of characters.

4.3.2.4 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)

4.3.2.5 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.

4.4 Built-in Functions

These are a list of functions included within StateMap.

4.4.1 Concurrent

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

4.4.2 State

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

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

```
state("clock") == "rising";
```

4.4.3 Sleep

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

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

sleep(1000);
```

4.4.4 Print

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

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

print("Hello Planet!");
```

4.4.5 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 */);
```

4.4.6 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);
```

4.5 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.

5 Project Plan

5.1 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.

5.2 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.

5.3 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.

5.4 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.

5.5 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.

5.6 Project Timeline

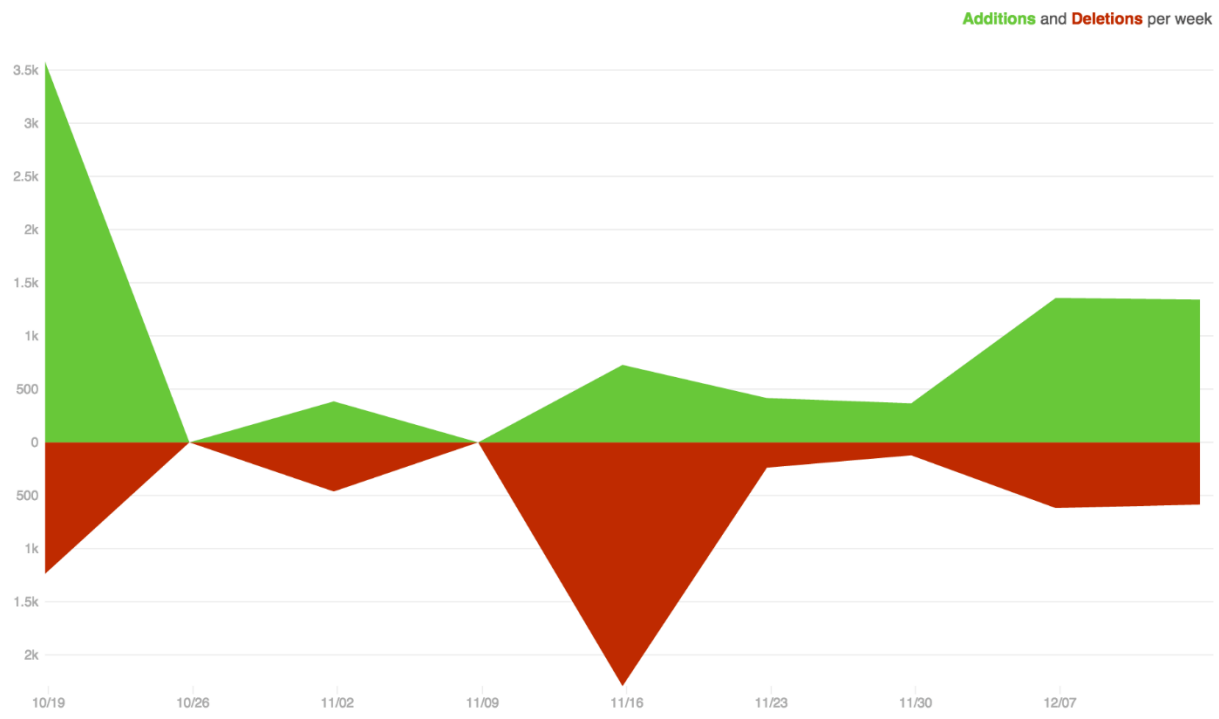
As mentioned in the planning section, there was no initial timeline planned, but as we went along we had several soft deadlines:

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:



5.7 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*

5.8 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.

5.9 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
|/ / /
```

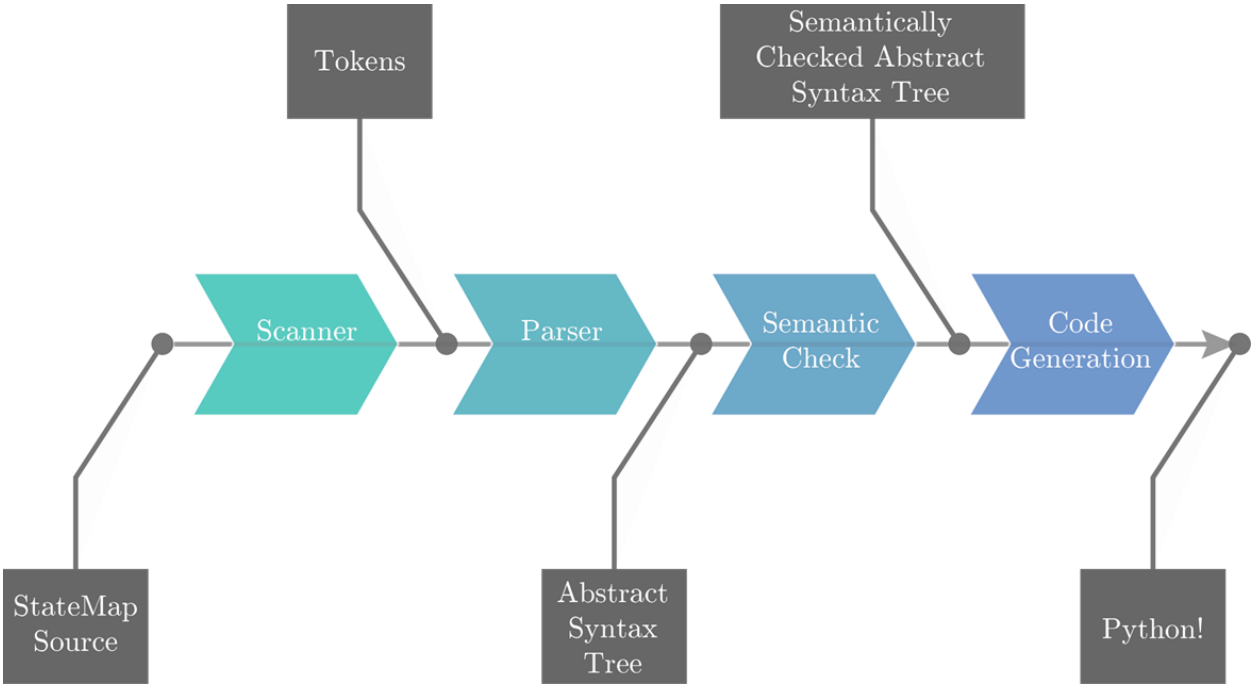
```
* | | Nero144 We got Hello World working (commits wont let me use exclemation
marks but imagine a ton of them)
* | | Jackson Foley fixes 10000 bugs in gen_python. Makefile lets us debug.
* | |   Jackson Foley Merge branch 'master' into code_gen
|\ \ \
| | |/
| |/|
| * |   Alexander_Peters Merge branch '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

6 Architectural Design

6.1 A Diagram of the Major Components of the Translator



6.2 Interfaces between the components.

6.2.1 Scanner

The scanner tokenizes StateMap's source code.

6.2.2 Parser

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

6.2.3 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.

6.2.4 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).

6.3 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.

7 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.

7.1 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'.

7.2 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

7.3 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

7.4 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
```

7.5 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 [])
```

8 Lessons Learned

8.1 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.

8.2 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.

8.3 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.

8.4 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.

8.5 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.

9 APPENDIX

9.1 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 }
```

9.2 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*)}
```

9.3 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)
```

9.4 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)
```

9.5 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
```

9.6 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
```

9.7 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
```

9.8 SOURCE CODE

```
//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;
      }
}
```

```
// Run with two command line integers separated by commas
//          python output.py 9,3

void DFA main(stack<string> args)
{
      int a = stoi(args.pop());
      int b = stoi(args.pop());

      start
      {
            s1    <-    a > b;
            s2    <-    a < b;
            s3    <-    *;
      }
```

```
	s1
	{
		a = a - b;
		start <- *;
	}

	s2
	{
		b = b - a;
		start <- *;
	}

	s3
	{
		print (itos(a));
		return;
	}
}
```

```
void DFA main()
{
	start
	{
	print("Hello World!");
	return;
	}
}
```

```
void DFA main(stack<string> args)
{
	int count = 0;

	start
	{
		hello		<- count < 10;
		finished	<-	*;
	}

	hello
	{
		print("Hello World!");
		count = count + 1;
		start <-	*;
	}

	finished
```

```
      {
            return;
      }
}
```

```
void DFA main()
{
    string setGroup = "";
    string choice = "";

    start
    {
        print("Welcome to the first Choose Your Own Adventure written in
the StateMap language ever! \nTo indicate your choices, simply hit the
corresponding number and the enter key.");
        input("1) Start your adventure.\n");
        wakeUp <- *;
    }

    wakeUp
    {
        print("\nOnly the first day of school and you're already
struggling with the question: Should I stay or should I go?\n");

        print("It's 3:30 PM and you're lying in bed. The sun gleams
through the room-wide window placed directly over the bedpost onto your
eyes, causing you to uncomfortably roll onto the floor. Your prior
Circuits class has placed you in a deep stupor near impossible to
shake.\n");

        print("As you trudge towards the bathroom, you think to yourself -
Surely going to that PLT class in this state would be a waste. I wouldn't
retain anything. Lectures will be posted online anyway. My friends will
have notes. No one else goes. He won't take attendance. Class
participation isn't graded.\n");

        print("Furthermore, during the last days of summer, you were just
a few games away from hitting Platinum tier in the PC game League of
Legends. The end of the season approaches and Kenny has been laughing at
you for not hitting Platinum sooner than Lee.\n");

        print("What will you do?:\n");

        print("1) Go to PLT anyway.");
        print("2) Play League and strive for Platinum!");
        choice = input("\n");

        firstClass  <- choice == "1";
        hahLoser    <- choice == "2";
        print("Type 1 or 2 for your choices.");
        wakeUp      <- *;
```

```
    }

     hahLoser
    {
        print("\nYou decided that enough is enough and you need to do the
responsible thing. Staying in Gold tier while Lee is in Platinum simply
won't do.\n");

        print("You skipped the very first day of PLT to play League of
Legends. Unfortunately every one of the games you played somehow had an
incredibly fed Master Yi killing your entire team. Furthermore, you ended
up dropping PLT, which turned out to be a required course. For shame.\n");

        print("You live life in regret, wondering about what could've
been.\n");

        input("The end.\n");

        returnNode    <- *;
    }

    firstClass
    {
        print("\nYou decided that it's probably a bad idea to skip the
first day to play League. Platinum can wait.\n");

        print("A curtain of heat brush against your face as you walk into
the large square room of Mudd 535. It just hit 4 o'clock and class is
supposed to start in ten minutes. The room is overflowing with students -
you can see piles of students conglomerating in the back where the seats
are closest and even more bursting out through the doors on the side into
the hallways.\n");

        print("You notice that despite the shortage of places to sit, a
left-handed seat at the front remains unclaimed. This chair wasn't near
the front; it WAS at the front, almost touching the Instructor desk. You
would literally be staring up the instructor's nostrils if he lectured
there. Sitting here would certainly make you the class pariah.\n");

        print("You also notice an open seat in the very back corner. While
you don't know the exact path needed to reach that seat, you're certain
that it'll involve physically climbing over the hoards of students around
there.\n");

        print("As you wrestle with your decision, you feel the oppressive
heat bear down on you. The air thickens, and you struggle to take a
breath. Weakened, you notice that several students are sitting on large
window-sills. One of the open windows has an open window-sill, and you can
practically feel the wind blowing through the wide frame on your
face.\n");

        print("You choose to:");

        print("1) Make your way to the back of the room");
```

```
        print("2) Sit on the window-sill");
        print("3) Sit in the very front.");

        choice = input("\n");

        kickingItInTheFrontSeat <- choice == "3";
        sittingInTheBackSeat    <- choice == "1";
        defenestration          <- choice == "2";
        print("Type 1, 2, or 3 to indicate your response.\n");
        firstClass              <- *;
    }

    defenestration
    {
        print("\nYou hastily make your way to the window-sill. Perching
yourself onto the jutting shelf, you feel refreshed by the cool winds
blowing through the large opening. Stretching back and leaning on the
frame, you yawn and wait for class to start.\n");

        print("On 4:09 PM, a man with blazing eyes charges into the room,
gripping a netbook. His sudden approach startles you and you attempt to
straighten up from your eased position. In doing so, you managed to slip
and fall out of the window. As the ground rises up to meet you, you think
to yourself: That's one way to drop out of school.\n");

        input("The end.\n");

        returnNode <- *;
    }

    sittingInTheBackSeat
    {
        print("\nYou claw your way towards the seat in the back, needing
to step on the desks of several students along the way, much to their
distate. As soon as you make it there, you immediately regret your
decision. The air back here is noticeably more dense, and you note that
the students around you seem dazed and unfocused. The desks here are of an
older make, with scrawlings from past bored students who have graduated
years ago.\n");

        print("Class hasn't even started yet and you feel a heavy weight
pulling your eyelids down. The room is bustling with conversation, but
most of it is loaded in the middle and front. If you stayed here for an
entire lecture, you'd surely go comatose.\n");

        print("The sound of quick footsteps made their way towards the
back, and you look up to see a tall figure laying something down on the
desk at front. While it's difficult to see much from your location, you
pick out a green shirt tucked into khaki slacks and glasses. For a moment,
you cease thinking about oven analogies and focus on the newcomer.\n");

        print("\"Okay, so I'm Stephen Edwards. This is Programming
Languages and Translators - COMS 4115, all that good stuff. Fall 2014.
You've figured that stuff out. My goal today is to convince you all not to
```

```
take this class because obviously there are far too many of you ...\"
Light chuckles and murmurs of agreement follow this statement, especially
from the people attempting to see the speaker through the doors leading
into the hallways.\n");

        print("\"... Final project presentations are due during the finals
week ... there's going to be a huge final project report due December 17th
...\" Already you're beginning to feel uncomfortable about this class. You
dread public speaking. You dread massive, final reports.\n");

        print("He continued, \"So, the main thing you're going to do in
this class, assuming that you don't drop it like I want you to, is a
semester long TEAM project. This is a team programming project. Now, the
team part of it is easily the worst, most difficult aspect of all of it,
but just to make it more difficult, I'm going to make you design and
implement your own language and a compiler for it ... you're going to have
to work with other human beings. This really sucks.\"\n");

        print("At this point, you're strongly considering packing up and
leaving. You don't know anyone in this class and stepping on people's desk
did not make a great first impression on your prospective classmates. As
you sit through the rest of the lecture and attempt to focus on Professor
Edwards' words through your rising panic, you've come to a decision.\n");

        print("You've decided to:");

        print("1) Stay in the course and tough it out.");
        print("2) Drop out of the course.");
        choice = input("\n");

        dropkicked  <- choice == "2";
        randomGroup <- choice == "1";
        print("Type 1 or 2 to indicate your choice.\n");
        sittingInTheBackSeat <- *;
    }

    randomGroup
    {
        setGroup = "bad";
        print("Class finishes, and you decide to leave to grab a bite.
While waiting in line at Hamilton Deli, you make the firm decision that
you'll stay in this class - after all, you have a very uncertain feeling
that this might be a required course.");

        print("Days pass and your attendance in class gradually falls. It
just happens that PLT occurs when you NEED to eat and take an afternoon
nap - it also helps that the lectures are posted in video form online
later. However, you realize that you need to join a group and you still
don't know anyone in class. You message the TA and she pairs you with
other students who appear to also be skipping class and thus do not know
anyone.");

        print("More weeks pass and an email was sent out, reminding you
that your language proposal is due soon. Wait. Language proposal?");
```

```
        print("You quickly attempt to arrange a meeting with your group.
However, due to conflicting schedules, you all decide to talk via emails.
After a few weak arguments, the group came to a consensus on a
language.");
        print("What language do you guys decide on?");
        print("1) The Whitespace Language. Dude, it'll be awesome. You
only need two keys.");
        print("2) Lava, which is kinda like Java, but better!");

        choice = input("\n");
        whiteSpace  <- choice == "1";
        Lava        <- choice == "2";
        print("Type 1 or 2 to indicate your choice.");
        randomGroup <- *;
    }

    kickingItInTheFrontSeat
    {
        print("\nYou inch your way to the front seat. As you place your
items on the desk and prepare to sit, you hear someone behind you ask
\"Dude, do you want to move more back? We can make some space. I wouldn't
want to cramp the professor's style.\"");

        print("Your angel took the form of a young man with large eyes and
a scraggly beard. He looked at you and you felt like an ant being examined
by a magnifying glass held by an entire committee of entomologists. You
express your thanks and slide your seat slightly back.");

        print("Footsteps resounded across the hallway and a giant of a man
marched into the room. Severity embodies Professor Stephen Edwards - a man
with steely eyes, a hard nose, and a brow set rigid by intelligence.
Without looking at the student masses flooding the room, he strides to the
desk, and sets down a small network. Conversation continued near the back,
but the front immediately was silenced as they waited for the first words
of the class.");

        print("Finished with setting his mic and slides, he spoke: \"Okay,
so I'm Stephen Edwards. This is Programming Languages and Translators -
COMS 4115, all that good stuff. Fall 2014. You've figured that stuff out.
My goal today is to convince you all not to take this class because
obviously there are far too many of you ...\" Light chuckles and murmurs
of agreement follow this statement, especially from the people attempting
to see the speaker through the doors leading into the hallways.\n");

        print("He continued, \"So, the main thing you're going to do in
this class, assuming that you don't drop it like I want you to, is a
semester long TEAM project. This is a team programming project. Now, the
team part of it is easily the worst, most difficult aspect of all of it,
but just to make it more difficult, I'm going to make you design and
implement your own language and a compiler for it ... you're going to have
to work with other human beings. This really sucks.\"\n");
```

```
        print("You disagree - you haven't had many opportunities to work
with a team on a large programming project and have been wanting
experience in it. And the idea of creating your own working language
excites you - you can't even fathom how such a thing is possible with your
current knowledge.");

        print("He continues lecturing, and hunger sets in. You realize
that this class occurs right around the time you have lunch - you won't
have a chance before due to your work-study. A fleeting desire to free up
this time slot crosses your mind, but you quickly wave it away. Focusing
on the lecture helps - Professor Edwards slips in occasional dry jokes and
frequent jabs at Java. All you really know is Java.");

        print("The class eventually finishes and students began to pool
around the front desk. You stand up and stretch, picturing biting into a
Hungry Man sandwich from Hamilton Deli with great detail. Before you
leave, you hear the student from before laughing. The thought of
introducing yourself to him occurs to you - after all, not knowing anyone
in a class centered around a massive group project only hurts you.");

        print("You make a choice to:");

        print("1) Leave and sign up for a later class so that you won't
die from hunger.");
        print("2) Ignore your hunger and introduce yourself to the
student.");
        print("3) Leave and order a Hungry Man from Hamilton Deli.");
        choice = input("\n");

        dropkicked  <- choice == "1";
        randomGroup <- choice == "3";
        goodGroup   <- choice == "2";
        print("Type 1, 2, or 3 to indicate your choice.");
        kickingItInTheFrontSeat <- *;
    }

    dropkicked
    {
        print("You drop the class and decided to sign up for another class
later in the evening. The class doesn't matter, as this CYOA story is
centered around PLT, and you've ended that line.");

        print("You leave Columbia, a unhireable disgrace.");

        input("The end.");

        returnNode <- *;
    }

    goodGroup
    {
        setGroup = "good";
```

```
        print("You quickly make friends with the helpful student. Luckily,
he had three friends set in the group and they were considering adding one
more person. You immediately accept his offer to join the group.");

        print("After the very next class, your newfound friends decided to
stay back and discuss a bit about potential language ideas. Concepts were
tossed back and forth and pondered, and one language eventually rose to
the top of consideration.");

        print("The language you decide on is:");
        print("1) Lava. It's not gonna be Java, we promise. It'll be
better!");
        print("2) Hey, DFAs are dope and there's a lot you can do with
them. What about a DFA simulating language?");
        choice = input("\n");

        Lava        <- choice == "1";
        StateMap    <- choice == "2";
        print("Type 1 or 2 to indicate your choice.");
        goodGroup <- *;
    }

    whiteSpace
    {
        print("You attempt to make the Whitespace language. It didn't go
past the proposal state. The TAs and other students laughed you out of
Mudd.");
        input("The end.");

        returnNode <- *;
    }

    Lava
    {
        print("After some thought, your group decides on the Lava
language. It's going to be a general purpose computer programming language
that is concurrent, class-based, object-oriented, and specifically
designed to have as few implementation dependencies as possible. It is
intended to let application developers \"write once, run anywhere\"
(WORA), meaning that code that runs on one platform does not need to be
recompiled to run on another! Lava applications are typically compiled to
bytecode that can run on any Lava virtual machine (LVM) regardless of
computer architecture.");

        print("After a few weeks of meetings in person in the computer
science lounge, an argument over leadership arises. You realize after
working with this group for a while that you're probably the best suited
to take this project to completion. However, another student in the group
seems to want the leadership position.");

        print("What do you do?");
        print("1) Attempt to obtain leadership.");
        print("2) Give up the position of leadership.");
        choice = input("\n");
```

```
        attemptCoup     <- choice == "1";
        forgetProject   <- ((setGroup == "bad") && (choice == "2"));
        okaaay          <- ((setGroup == "good") && (choice == "2"));
        print("Type 1, 2, or 3 to indicate your choice.");
        Lava            <- *;
    }

    StateMap
    {
        print("You pick a great language. You guys work hard and create a
final report that rivals the one listed in the directory hosting a folder
called sample_programs that holds this CYOA.");
        input("Hit enter to continue.");

        goodEnding      <- *;
    }

    goodEnding
    {
        print("During the presentation, Professor Edwards only seemed
unimpressed rather than disgusted. Your class performance was a triumph
and you walk away, an individual carved by trial and cast in victory.");
        input("The end.");

        returnNode <- *;
    }

    forgetProject
    {
        print("You decided to give up the leadership position.");
        print("The meetings occurred less and less frequently, and soon
you even stop watching the lectures online. By the time of the first
midterm, you realize that you've missed two of three homeworks required in
the semester, and your entire group forgot about the group project.");
        print("You drop the class.");
        input("The end.");

        returnNode <- *;
    }

    attemptCoup
    {
        print("You attempt to wrest control of leadership.");
        print("Do you:");
        print("1) Attempt diplomacy?");
        print("2) Attempt intimidation?");
        choice = input("\n");

        defenestration  <- choice == "2";
        notOkay         <- choice == "1";
        print("Type 1 or 2 to indicate your choice.");
        attemptCoup <- *;
    }
```

```
    defenestration
    {
        print("You get forcibly thrown out a window by the entire group.
As you fall, you note that this situation feels a bit familiar ...");

        input("The end.");
        returnNode <- *;
    }

    notOkay
    {
        print("You became the leader and attempt to pull the group
together as hard as you can. Another student notes your efforts and does
her best to contribute, but in the end, the project was too big for two
students. For some reason, the project wasn't very well received, and you
take your anger out in your discussion about the other students' role in
the final report - which YOU have to put together.");
        input("The end.");

        returnNode <- *;
    }

    okaaay
    {
        print("Your team managed to pull something out but there was a
crucial error with the language itself. It's a Java copy. Professor
Edwards tosses you out of a window.");
        input("The end.");

        returnNode <- *;
    }

    returnNode
    {
        return;
    }

}
```

```
/* A StateMap DFA that accepts the
reg ex (ab|c*)d* */
void DFA main(stack<string> args) {
    int accepted = 1; /* acceptance
        state if reach end of stack*/

    start {
        string s = args.peek();

        stateOne    <- s == "a";
        stateTwo    <- s == "c";
```

```
    stateThree  <- s == "d";
    accept      <- s == EOS;
    notAccept   <- *;
}

stateOne {
    accepted = 0;
    args.pop();
    string s = args.peek();

    stateFour   <- s == "b";
    notAccept   <- *;
}

stateTwo {
    accepted = 1;
    args.pop();
    string s = args.peek();

    stateThree  <- s == "d";
    stateTwo    <- s == "c";
    accept      <- s == EOS;
    notAccept   <- *;
}

stateThree {
    accepted = 1;
    args.pop();
    string s = args.peek();

    stateThree  <- s == "d";
    accept      <- s == EOS;
    notAccept   <- *;
}

stateFour {
    accepted = 1;
    args.pop();
    string s = args.peek();

    stateThree  <- s == "d";
    accept      <- s == EOS;
    notAccept   <- *;
}

notAccept {
    print("Not accepted by the DFA");
    return;
}

accept {
    print("Accepted by the DFA.");
    return;
}
```

```
}
```

```
//4-bit SIPO Shift Register
//    Data input given as a command line argument of 0s and 1s separted by
commas.
//    Will accept any reasonably lengthed input.

void DFA dataIn(stack<string> data)
{
      int counter = 0;

      //Read state
      start
      {
            counter = 0;
            low <- data.peek() == EOS;
            string currData = data.pop();
            high <- currData == "1";
            low  <- currData == "0";
            error <- *;
      }

      //high and low states to represent the current data input
      //    counter is used for synchronicity
      high
      {
            start <- counter == 1;
            counter = counter + 1;
            high <- *;
      }

      low
      {
            start <- counter == 1;
            counter = counter + 1;
            low <- *;
      }

      error
      {
            print("invalid input");
            return;
      }
}


// 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 catch for the DFFs
	rising
	{
		high	<- *;
	}

	high
	{
		sleep(halfPeriod);
		start	<- *;
	}
}

// 1st T-FlipFlop in Shift Register
//  Catches data on every rising clock
void DFA DFF1()
{
	// low output
	start
	{
		high  <- (state("clock") == "rising"
					&& state("dataIn") == "high");
		start	<- *;
	}

	// high output
	high
	{
		start <- (state("clock") == "rising"
					&& state("dataIn") == "low");
		high  <- *;
	}
}

// 2nd T-FlipFlop in Shift Register
//  Catches DFF1 on every rising clock
void DFA DFF2()
{
	// low output
	start
	{
		high  <- (state("clock") == "rising"
					&& state("DFF1") == "high");
		start	<- *;
	}
```

```
      // high output
      high
      {
            start <- (state("clock") == "rising"
                              && state("DFF1") == "start");
            high  <- *;
      }
}

// 3rd T-FlipFlop in Shift Register
//  Catches DFF2 on every rising clock
void DFA DFF3()
{
      // low output
      start
      {
            high  <- (state("clock") == "rising"
                              && state("DFF2") == "high");
            start      <- *;
      }

      // high output
      high
      {
            start <- (state("clock") == "rising"
                              && state("DFF2") == "start");
            high  <- *;
      }
}

// 4th T-FlipFlop in Shift Register
//  Catches DFF3 on every rising clock
void DFA DFF4()
{
      // low output
      start
      {
            high  <- (state("clock") == "rising"
                              && state("DFF3") == "high");
            start      <- *;
      }

      // high output
      high
      {
            start <- (state("clock") == "rising"
                              && state("DFF3") == "start");
            high  <- *;
      }
}

// display DFA to read and print out the current state of
//    the shift register
void DFA display()
```

```
{
	int d = 0;
	int b1 = 0;
	int b2 = 0;
	int b3 = 0;
	int b4 = 0;

	start
	{
		read <- *;
	}

	read
	{
		d = state("dataIn") == "high";
		b1 = state("DFF1") == "high";
		b2 = state("DFF2") == "high";
		b3 = state("DFF3") == "high";
		b4 = state("DFF4") == "high";
		print <- *;
	}

	print
	{
		print ("Current parallel output: " + itos(b1) + itos(b2) +
itos(b3) + itos(b4));
		print ("About to read in a bit of " + itos(d));
		print ();
		start <- *;
	}
}

void DFA main(stack<string> args)
{
	int halfPeriod = 1000;

	start
	{
		concurrent(clock(halfPeriod), dataIn(args), DFF1(), DFF2(),
DFF3(), DFF4(), display());
		return;
	}
}
```

```
DFA main (stack<string> main)
{
	Map<string, int> wordCount;

	start
	{
		print		<- main.isEmpty;
```

```
            new             <- !wordCount.contains(main.peek);
            increment  <- *;
      }

      new
      {
            wordCount.put(main.pop, 1);
            start           <- *;
      }

      increment
      {
            string word = main.pop;
            wordCount.put(word, ++wordCount.get(word));
            start      <- *;
      }

      print
      {
            printMap(wordCount);
      }
}

void DFA printMap(Map<string, int> map)
{
      Stack<string> words;

      start
      {
            words = map.keyStack;
            done       <- words.isEmpty;
            printEntry <- *;
      }

      printEntry
      {
            string word = words.pop;
            print(word + " " + map.get(word) + "\n;
            start      <- *;
      }

      done
      {}
}
```

```
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;
        }
}
```

```
void DFA main(int lol, double gg, stack<string> huh, string welp) {
  int a;
  string b;
  stack<int> c;
  double d;
```

```
  void e;

  whee {
    return 5;
  }

  wowza {
    ayo <- *;
    int kldsa;
    ayo2 <- kldsa + ladkasl == 5;
    main2(kldsa, c);
  }

  start {
    Concurrent(clock(), yo(), yoyo());
  }
}

void DFA main2(string foo, int bar) {
  void a;
  string br0;
}
```
