CS3423: Compilers 2
Professor : Ramakrishna Upadrasta
Department of Computer Science
Indian Institute of Technology Hyderabad

AUTOMATON

V.Sathwik - Project manager

Nalari Sushma - System Architect

P Rashmitha - System Integrator

L Harsha Vardhan - Language Guru

Danda Sarat Chandra - System Integrator

K Lakshmi Sravya - Tester

K Sri Teja - Tester

December 6, 2021

Contents

1 Introduction

Our idea for the Automaton is to enable a programming language that emulates DFA (Deterministic Finite Automata).The purpose of the language is to shrink the gap between the paper model of DFA and the running code. Automaton is organized and executed similar to an Automata diagram, emphasizes modularity and organization of code into short nodes that transition to each other until reaching one of the end states. Our language blends the functional and imperative programming styles to allow programmers to abstract away implementation details.

1.1 States in Automaton

Automaton programs contain nodes/states, and the transition between these states takes place based on the conditions we check.

Transition nodes include transition statements, which check the expressions, and use them if the statement is true. All transition nodes should be kept up to date. End node.

The state of an automata is defined by name, followed by flower brackets , which contains multiple functions inside it, with either the transition or return statements.

Start state: All the Automaton codes must begin with the "start" state, where the 1st state to be chosen is given.

Accepting state: If a finite state machine finishes an input string and is in an accepting state, the string is considered to be valid.

1.2 Modularity

The key to a good Automaton program is extreme flexibility. Being able to draw a program, on paper, like an automata means that you are probably on the right track. Nodes are systematic in a short concise manner along with ability to create new variables, the decisions mainly contain global information, and local variables are also available primarily for simplicity, efficiency, and shortening code length.

2 Language Tutorial

2.1 Getting Started

First we need to analyse the picture of the DFA diagram representation before writing any code, which is the main essence of our project. After getting a picture now it will be easier to write the code.

If our program needs multiple dfa then we can write them as functions with one of the dfa as the main function like in c, c++, python etc (other programming languages) to decrease the complexity. If our program needs one dfa then we can write it in the main function or create a dfa function if we use it multiple times, we can write in any way which is possible and easy to implement in the automaton. First we understand the language using single DFA codes and then we will write complex programs in the next sections to understand the language in depth.

2.2 Structure of a program

A single DFA 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 Automaton:

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

the program begins, the start state of the main method is run, and this program prints ”Hello World!” to standard out using the print() function.

2.3 Variable declaration

Below given are the syntax of the variables used in the Automaton. We must explicitly declare the variable before using it, we can also assign the value of the variable while declaring.

int num;
float n;
string str;
stack s;

2.4 print statement

Syntax for the print statement :
print("Hello World");
print("Our language name id Automaton");

We can print the statements using the built-in **print()** function.

2.5 Function Declaration

The key word used for the main function in our Automaton is **DFA main** and for defining the functions it is **def.** The main function should be present in every program. We will understand the structure of the function using a simple program called "Hello World". First we have to declare the void DFA main followed by the states. The states are contained within the curly braces of the main function and the code part in each transition is defined in the curly braces of that transition.

The hello world code written below is the single DFA program where the code must begin with the start state and end with return type defined in the function statement and the program prints **'Hello World!"** and we know that the function prints using the built-in function print().

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

2.6 Multiple States

In our language automaton there are no if else statements, no for and while loops, it consists of states where we can jump from one state to another based upon the condition given. Let us understand this using an example to find the greatest integer between two numbers.

```
int DFA main ()
{

    Start
    {
      int a = 10;
      int b = 5;
      Printx < -(a >= b);
```


```
        Printy <- $;
      }
      Printx
      {
        print ("a is greater than or equal to b");
        return 0;
      }
      Printy
      {
        print ("b is greater than a");
        return 0;
      }
}
```

this symbol is used to change states in the program, if the condition present on the right side is true then it will go to the state name given on the left side or else it will go the default state which is represented as $ in the program.

Here instead of if else statements we used the states Start, Printx, Printy. First we define two integers a and b in the start state and in the next line first we check the condition which is present on the right side of the symbol "<-", if it is true then it goes to the Printx or else it will go to the Printy as the default statement. As given a = 10 and b = 5, transition is from start to Printx and then it prints **"a is greater than or equal to b"** using the built-in **print()** function.

2.7 compilation and running

After running "make" to produce the make file executable, we can compile our .sm file with the command given below.

```
$   ./gen\_python   filename  < path to your .am  file
```

The above command produces a python code and now we have to run the python code to get the output of our code. To supply command line arguments to your program, you add them after the python command.

The command to run the python file is given below

```
$ python filename.py "command line args"
```

3 Language Reference Manual

3.1 lexical Conventions

3.1.1 comments

3.1.1.1 Single line comments:

They start with these characters.
Ex: Automaton is a domain specific language.

3.1.1.2 Multi line comments:

They start with these characters /* and end with these characters */ .Between these characters /* & */ there can be any characters between them except the end symbol.
Ex: /* Automaton is a domain specific language, based on DFA */

3.1.2 Identifiers

An identifier is a sequence of letters, underscores, and digits. the first letter must be small and it is a case sensitive language.

3.1.3 Key Words

These are the keywords and can't be used as variables :
main, int, float, char, DFA, void, stack, start, exit, return, true, false, boolean, print.

3.1.4 Constants

These are some of the constants used in Automaton.

3.1.4.1 Integer constants

It consists of one or more than one digits and also one optional '-' (negative) sign.
Ex:
Valid : 10, 0, -12, etc.
Invalid : +10, 0.01, 00, etc.

3.1.4.2 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 and any other value will be evaluated as true.

3.1.4.3 Float Constants

It is a 64-bit signed floating point. It contains one or more digits followed by a decimal and another integer or a decimal point followed by an integer(positive integer).
Ex:
Valid: 2.43, 90.8, .6, -2.3.
Invalid: 42, 0, .-3.

3.1.5 Operators

All the operators in this language are similar to C++ operators. But the arrow operator(->) is used for state transition.

3.1.5.1 Arithmetic Operators

Operator	Use	Example
+	Addition	
-	Subtraction	Left
*	Multiplication	Left
/	Division	Left

3.1.5.2 Relational Operators

Operator	Use	Example
!=	Test inequality	Left
>	Greater than	Left
<	Less than	Left
>=	Greater than or equal to	Left
<=	Less than or equal to	Left

3.1.5.3 Logical Operators

&	BITWISE AND	Left
|	BITWISE OR	Left
.	Access	Left

3.1.5.4 Assignment Operator

= is assignment Operator. It assigns values from right side operand to left side operand.

3.1.5.5 Arrow Operator

-> is Arrow Operator.This operator is used for transition between the states in the Definite Finite Automata.

3.1.6 Precedence and Associativity

->	Left to Right
,	Left to Right
=	Right to Left
||	Left to Right
& &	Left to Right
== and !=	Left to Right
> < >= <=	Left to Right
+ -	Left to Right
* / %	Left to Right
()[]	Left to Right

3.2 Types

3.2.1 Type Declaration

A set of values with similar characteristics are considered as data types. Primitive data types are predefined by the language and reserved by a keyword. Derived data types are the aggregation of fundamental data types.

3.2.2 Primitive Types

The basic data types in Automaton are:

- int - It is a keyword used for the integer data type. It is represented in 32-bits and in signed two's complement form. Its range is from -2^{31} and $2^{31} - 1$
  Ex: int a = 3, b = -2;
- char - It is a keyword used for the character data type. It is represented in 8-bits. It stores all the ASCII characters in a fixed-length field.
  Ex: char ch1 = 'X, ch2 = ' ';
- float - It is a keyword for 64-bit signed which also includes decimal/exponential portion.
  Ex: float f = 1.32;
- double - It is a keyword for 32- bit floating bit number.
  Ex: double d = 1.7 e-3;
- bool - It is a keyword used for the data type Boolean. It represents the correctness of a statement/expression. It has 2(binary) possibilities - TRUE or FALSE.
  Ex: bool var = true;

3.2.3 Derived Types

The derived data types in Automaton are:

- stack - It is a collection of data types. It follows LIFO(last in first out). It uses push(), pop(), top() to manipulate a given stack.
- push - Used to push a given item to the top of a stack.
- pop - Used to remove the top element of a stack.
- top - Used to return the top element of a stack.
- string - It is a sequence of characters. It is represented in double-quotes.
  Ex: string s = "AbC";

- vector - It is used to store a collection of same data types.
  Ex: vector<int> v;
  push_back(), pop() , size() is used to add an element, remove an element, return the size of a vector.

3.3 Syntax notation

3.3.1 Program structure

Code consists of a series of DFA's with the main dfa to which command line arguments are written in the form of a stack of strings and the return type is written first in the declaration. The syntax of the main dfa is:

```
void DFA main(/*  arguments */) {
/* statements */
}
```

If we know the command arguments beforehand then it can be passed to the main function directly or else we can rely on the stack of primitives :

```
void DFA main(/*[datatype] n1,  [datatype] n2,  [datatype] n3, */) {
/* statements */
}

Or else

void DFA main (stack<string> args){
/* statements */
}
```

The main DFA must return their type as mentioned in the function :

- void DFA -> return;
- int DFA -> return [int];
- float DFA -> return [float];
- string DFA -> return [string];

3.3.2 States

The program consists of the states where they can go from one state to another based on the condition given, each state block contains statements between the curly braces. The state names must start with a capital letter and the starting node in the DFA should be labeled as "Start".

Every state block must end with the return statement or the transition (<-). The basic syntax of these states is:

```
/* name starting with capital letter */ {
/* statements */
}
```

3.3.3 Sub-Dfa

A program can have multiple dfa's which are functions in a program. These dfa's are implemented with their own states and transitions ending with the return type declared in the function name. The main dfa must be declared last after defining all the sub-dfa's in the program. The syntax for the sub-dfa is same as the main dfa :

```
/* return data type */  DFA  /*name of the function*/ (/* arguments */) {
/* states/statements */
}
```

The main DFA must return their type as mentioned in the function :

- void DFA -> return;
- int DFA -> return [int];
- float DFA -> return [float];
- string DFA -> return [string];

3.3.4 Expressions

3.3.4.1 Literals and Operators

Literals are data used for representing fixed values which can be directly used in the code. The operators consist of two groups: booleans test operators and the expression operators.

The boolean operators include "=", "<", ">", ">=", and "<=". They can be used for condition checking while going to another state. The expression operators contain "()", "||", "+", "-", and "*".

```
{Id}
{Id} {Operator} {Id}.
```

3.3.4.2 Method Calls

Method calls that return a value will evaluate as expressions.

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

Assume a stack called s was declared. A valid method call is: s.push(10) and will return 10, s.peek() will return the top element in the stack.

3.3.5 Statements

The types of statements in the Automaton are declaration, assignment, sub-dfa, transition and return. Only declaration and assignment are the types which can be defined outside the states i.e a node and every statement should end with semicolon ";".

3.3.5.1 Literals and Operators

A declaration is a statement where it consists of the variable type followed by the id. The variable must start with the small letters because we used capital letters for defining the state names. There can be multiple declarations in the same line separated by comma ",".

```
Syntax : {datatype} {ID};
Ex:
int num;
int a, string b, stack<float> s;
```

3.3.5.2 Assignment

An assignment is used to set the values for the variables which can be done in two ways i.e while declaring the variable or later in a different line using variable id. There can be multiple assignments made in the same line separated by comma ",".

```
{datatype}{Id} = {Expression}

Or

{datatype}{Id};
{Id} = {Expression}
```


```
Ex:
int n = 10;

float f;
f = 4;
String str = "Automaton";
```

3.3.5.3 Sub-Dfa Call

A sub-DFA call is a function call expression which can also be used in the assignment statement because of the return type in the dfa function.

```
DFA_1 (/* arguments */);
int n = DFA_2 (arg_1, arg_2);
```

3.3.5.4 Transition

A transition statement consists of a state id, the transition operator (<-) is used to denote a transition from one state to another state. The transition occurs if the expression written on the right of the operator is true. And the dollar symbol "$" is used as a default statement.

```
{State_name}<-{condition}
{State_name}<-$
State_1 <- (a >= b)
State_2 <- $
```

The transition occurs if the value of a is greater than or equal to b(the condition is written on the right side of the arrow "<-"). The dollar operator indicates the default transition state i.e if the condition is false then the transition takes place with the default state named as State_2, and the state must end with the default transition or the return statement.

3.3.5.5 Return

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

```
return {expression};

int n = 12;
return n > 9; // returns an integer 1
```

3.3.6 Scope

Scope of the Automaton is divided into global types and local types. Global scope is particular to the DFA whereas local scope is for a state.

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

3.4 built-in functions

3.4.1 Print

It is a simple built -in function which takes a single argument of type string and then prints it. The keyword used for is function is print and the syntax of this function is

print(“string to be printed”);

```
Examples
  print("Hello World");
  print("Our language name is Automaton");
```

3.4.2 Input

It is a built-in function which takes a single argument of type string and then it prints the argument in the terminal where the function has been called. Now it waits for the user to give the input and then it stores the string as the return value. The keyword used for is function is input and the syntax of this function is :

String str = input(/*input taken from the user */);

4 Project Plan

- **Week 0:**
  - we had a discussion on each other's ideas and came up to a conclusion on what language should be taken, what lexical tokens should be used,what are domain-specific things to be done..
  - later we studied Ocaml for building compiler architecture. after coming to the end of the discussion we ended up with a language name called Automaton with what operators, what types to be used, and what extra operators to be added.
- **Week 1:**
  - we came up with language syntax and submitted Assignment-1. To do the Lexical part .we explored about Ocamyllex and we started working on it.
- **Week 2:**
  - We have written the code for a lexical analyser using Ocaml.
  - We tested our lexer with several testcases to improve its performance. We went through the comments on our language manual and trying to improve the features.
  - We completed the presentation, implementation and demo videos for the lexical analysis phase. Discussed how to extend this to parser phase.
  - Currently exploring different methods to implement the parser.
- **Week 3:**
  - We have finished the lexical analyzer part and submitted the required working model. We started learning grammer rules for our language. We edited the parser file with statements and currently exploring more about the parser.
- **Week 4:**
  - we have completed writing code part of the parser.added some more tokens in lexer, updated input test cases,made PPT for the parser part and started working on the demo and presentation video.
- **Week 5:**
  - We completed the implementation, demo videos for the parser phase.
  - Made a note of all the semantic checks to be done. Currently exploring different methods to implement this semantic phase.
- **Week 6:**

  - We are working on semantic analyser, resolved some errors in parser and lexer parts and now we are implementing grammar part according to the semantic rules.

- **Week 7:**
  - We used the obtained syntax tree in the parser phase and trying to build a symbol table.
  - We are trying to make our semantic phase work using different testcases written earlier.
- **Week 8:**
  - Checked whether the parameters and the tokens have been declared properly or not in the list of functions.
  - Wrote some more evaluation rules by updating the previous examples. Modified the parser according to the updated examples.
- **Week 9:**
  - We completed the implementation, demo videos for the semantic phase.
  - submitted semantic assignment with all the required delivarables.
- **Week 10:**
  - We are remodelling the semantic analyzer. Started learning about code generation phase and working on it.
- **Week 11:**
  - We started analysing code generation phase after making some modifications in examples for the semantic phase.
- **Week 12:**
  - We are trying to understand the code generation phase to optimize the compilation process. We are continuing our progress in this phase using the examples created.

5 Language Evolution

Our project plan was simple, it started with an idea to shrink the gap between the paper model of DFA and the running code. The original idea for Automaton is from the concept that a Finite Automata with two or more stacks could compute a computable problem. We didn't change much of our plan throughout the timeline, but we were unable to do some parts of the code through the semantic phase and code generation so we modified our code so that we can implement it a bit easier.

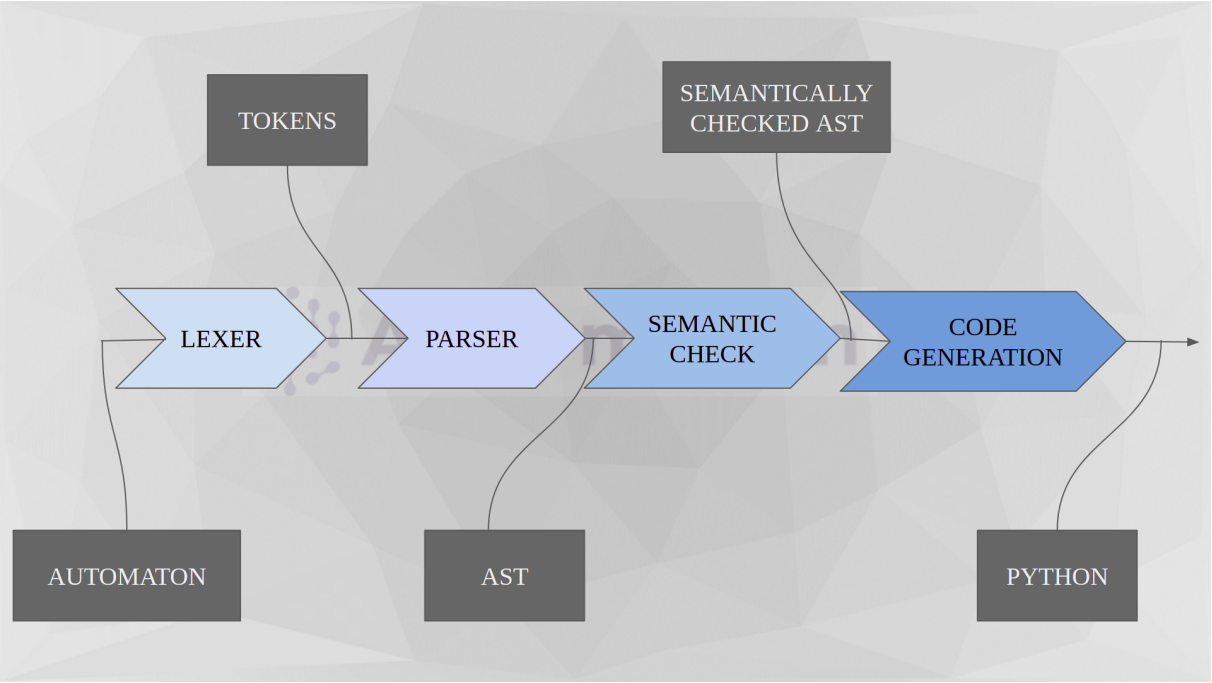
First we thought of adding built in functions like concurrent, conversion functions, etc but it became complicated while implementing and hence we left it out. We also thought of implementing the multiple dfa functions but we were unable to understand the whole implementation process so we decided to leave that part.

We worked steadily throughout the timeline of 7 weeks, but the work in the last month increased a lot for fixing the errors found while testing, customizing and improving our language, and implementing the semantic_check and code generation part. Once the language generated code, through testing we were able to greatly adjust, customize, and improve our language.

The code generation part was not easy to implement. We tried using python script to return output files as it has many in-built functions which makes our implementation easier but we are not able to return output.py file and return the output. we tried our best but we were unable to complete the whole part due to the time constraint and python execution errors..

6 Compiler Architecture

The components of the compiler are:
1.Lexer
2.Parser
3.Semantic
4.Code Generation



6.1 Lexer

Lexer converts a sequence of characters into a sequence of tokens.errors like invalid characters,incomplete multi line comments are handled with white spaces getting ignored.

6.2 Parser

Parser takes the tokens produced by lexer and matches with grammar rules to form Abstract Syntax Tree.Syntax errors will be handled here.

6.3 Semantic

The semantic check takes an AST and semantically checks it.Semantic Analysis is the process of drawing meaning from a text,Ensuring the declarations and statements of a program is done in this process.Functions of Semantic Analysis are:-

- Type Checking: Makes sure that each operator has matching operands or in other words ensures that data types are used in a way consistent with their definition.
- Label Checking: Every program must contain labels references.
- Flow Control Check: Keeps a track of whether the control structures are used in proper manner or not.It occurs during compile time and run time.

  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.

6.4 Intermediate Code Generation

The code gen takes the semantic checked program, and translates it into python.It generates a fair amount of pre-established pure python code that is used to do built-in language functions and set up the architecture for the main function to run itself.

7 Development Environment

7.1 GNU MAKE

without make file our compiling would be difficult as there are many commands to run and the probability of doing mistakes while typing would be higher. The advantages we got are :

- It automatically determines the proper order for updating files, in case one non source file depends on another non-source file.
- if we change a few source files and then run Make, it does not need to recompile all of your program. It updates only those non-source files that depend directly or indirectly on the source files that you changed.
- It can also regenerate, use, and then delete intermediate files which need not be saved using -rf clean commands.

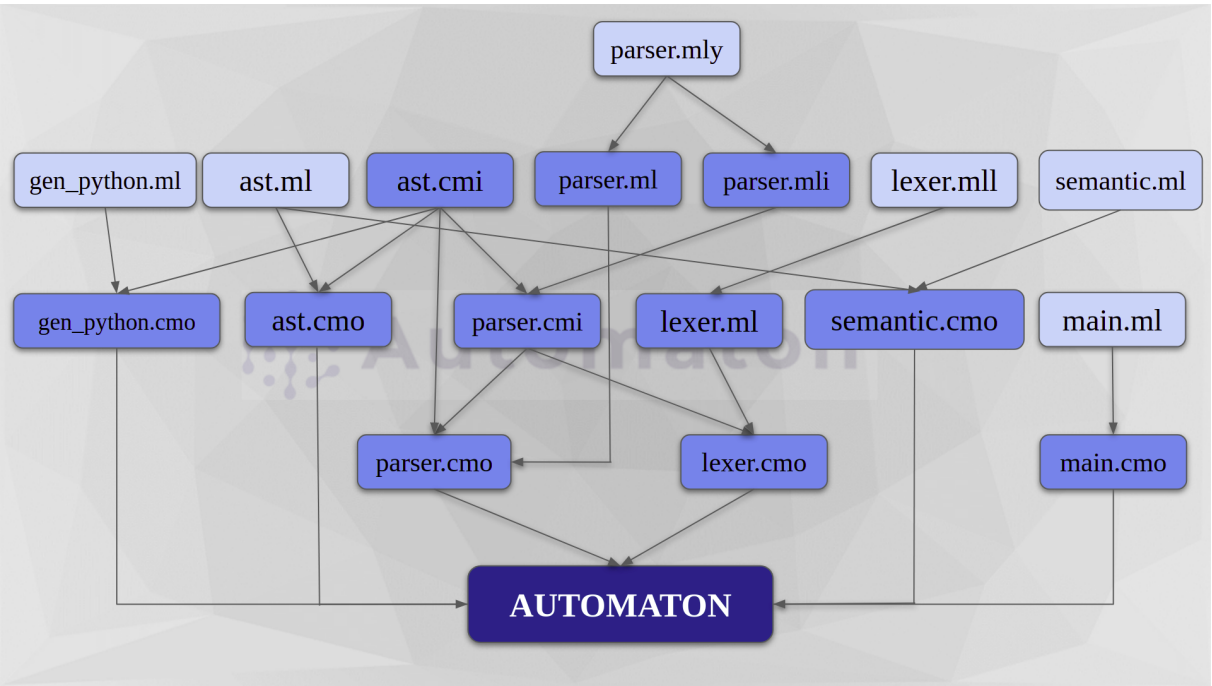


Figure 1: Makefile Dependencies

7.2 Github

we used Git for pushing our work whenever a part is done so that every other team can access it and can commit if there were any modifications.

7.3 VSCode

we used VSCode for liveshare as our team members are not at one place. It helped in rectifying our mistakes. The terminal share feature helped our team to understand better way of execution.

8 Conclusions & Lessons Learnt

It was a good experience for us. During sem 4 we learnt about DFA and Turing machines taught by subramanyam sir and we thought of developing a compiler on Finite Automata. There came the idea of Automaton and we started working on it.

With a project this large, but also this detailed, it is so important to keep constant communication between the group. Tools like group meetings and github were essential for us in arriving at the end result. While a work delegation was 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, we learned how to understand and build upon other people's work .

We first learnt that we should begin testing as soon as we are done with the code. We realized that we should check out a new branch before you start working on a new feature. We 1st searched for command lines required for various phases but then figuring out ocaml is better than writing from scratch.

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 greedy approach to problem solving isn't sufficient. Test periodically, as parts of the compiler are written, on actual programs 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 Test Plan And Test Suites

9.1 Introduction

We wrote a good number of test cases and error cases to check our code which handled all the syntax types like multiple states, arithmetic, loops using states, stacks, etc. Below we have written 10 test cases with their expected output and error cases to understand our language properly.

9.2 Test Cases

```
// basic syntax of the dfa program
int DFA main()
{
          Start
          {
                      return 0;
           }
}
```

Listing 1: Test 1

```
0
```

Listing 2: Expected Output

```
// different types of datatypes used in Automaton
void DFA main()
{
          Start
          {
                    int number = 10;
                    float f= 5.0;
                    string s = ""Automaton;
                    print(s + number);
                    print(f);
           }
}
```

Listing 3: Test 2

```
Automaton 10
5.0
```

Listing 4: Expected Output

```
// check whether the given integer is even or odd
void DFA main()
{
          Start
```


```
        {
                    int n = 7;
                    Even_state <- (n%2 == 0)
         }
          Even_state
          {
                     print"(n is an even "number);
                     return;
          }
          Odd_state
          {
                     print"(n is an odd "number);
                     return;
           }
}
```

Listing 5: Test 3

```
n is an odd number
```

Listing 6: Expected Output

```
// increment the a value i.e a = a+5;
void DFA main()
{
         Start
         {
          int a;
          print"(Enter a value: ");
          a = input();
          a = a + 5;
          print"(Updated a value is " + a);
          return ;
          }
}
```

Listing 7: Test 4

```
Enter a value :
User: 5
Updated a value is 10
```

Listing 8: Expected Output

```
// Multiple states in a DFA
void DFA main()
{
  int a = 1;
  Start
  {
    Bool_one <- (a > 0);
```

```
    Bool_two <- $;
  }

  Bool_one
  {
    Bool_two <- $;
  }

  Bool_two
  {
    print("present state: bool_two");
    return;
  }
}
```

Listing 9: Test 5

```
present state: bool_two
```

Listing 10: Expected Output

```
// print statements (built-in function)
void DFA main()
{
    Start
    {
        print("Hello! What's your name?");
        string s = input();
        print("Cool, " + s + " is your name?");
        print("Testing  and other stuff");
        return;
    }
}
```

Listing 11: Test 6

```
Hello! What's your name?
Sathwik
Cool, Sathwik is your name?
Testing  and other stuff
```

Listing 12: Expected Output

```
// Arithmetic operations using multiple 'DFAs
// functions - add, sub, mul


int DFA fun_add(int a, int b)
{
  Start
  {
```


```
    return a + b;
  }
}

int DFA fun_sub(int a, int b)
{
  Start
  {
    return a - b;
  }
}

int DFA fun_mul(int a, int b)
{
  Start
  {
    return a * b;
  }
}

void DFA main()
{
  Start
  {
    int a ;
    a = fun_add(1,2);
    print"(Addition: " + a);

    int b;
    b = fun_sub(1, 2);
    print"(Subtraction: "+ b);

    int c;
    c = fun_mul(1, 2);
    print"(Multiplication: "+c);

    return 0;
  }
}
```

Listing 13: Test 7

```
Addition: 3
Subtraction: -1
Multiplication: 2
```

Listing 14: Expected Output

```
//comparing two integers
void DFA main()
{
```


```
  Start
  {
    int a = 10;
    int b = 5;
    a = a + 1;
    Printx <- (a >= b);
    Printy <- $;
  }
  Printx
  {
    print("a is greater than or equal to b");
    return;
  }
  Printy
  {
    print("a  is less than b");
    return;
  }

}
```

Listing 15: Test 8

```
a is greater than or equal to b
```

Listing 16: Expected Output

```
//stack operations on DFA
void DFA main()
{
  Start
  {
    stack<float> s;
    s.push(10.0);  // 10.0
    s.push(3.5);    //3.5 10.0
    float a = s.peek();  // a = 3.5
    State_1 <- (a > 0);
    End <- $;
  }
  State_1
  {
    print("peek value is positive");
    return;
  }
  End
  {
    print("peek value is negative");
    return;
  }

```


```
}
```

Listing 17: Test 9

```
peek value is positive
```

Listing 18: Expected Output

```
// print hello world 10 times (loop)
int DFA main()
{
    Start
    {
      int i = 0;
      int n = 10;
      Print <- (i = 0);
      Exit <- $;
    }

    Print
    {
      i = i+1;
      print("Hello world " + i);
      Print <- (i < n);
      Exit <- $;
    }

    Exit
    {
      return 0;
    }
}
```

Listing 19: Test 10

```
Hello World 1
Hello World 2
Hello World 3
Hello World 4
Hello World 5
```

Listing 20: Expected Output

9.3 Error Codes

```
//comparing two integers

void DFA main()
{
```

```
  int a = 1;
  Start
  {
    Bool_one <- (c > 0);  //error(c not defined)
    Bool_two <- $;
  }

  Bool_one
  {
    Bool_two <- $;
  }

  Bool_two
  {
    print("Second");
    return;
  }
}
```

Listing 21: Test 1

```
// print statements (using built-in function)

void DFA main()
{
    Start
    {
        print("Hello world!");
        int num = input();
        print("Hello world! " + num);
        return 0;               //error(wrong return type)
    }
}
```

Listing 22: Test 2

```

//Arithmetic operations using multiple 'DFAs

// functions - add, sub, mul

int DFA fun_add(int a, int b)
{
  Start
  {
    return a + b;
  }
}

int DFA fun_sub(int a, int b)
{
```


```
  Start
  {
    return a - b;
  }
}

int DFA fun_mul(int a, int b)
{
  Start
  {
    return a * b;
  }
}

void DFA main()
{
  start  // error state must start with capital letter
  {
    int a ;
    a = fun_add(1,2);
    print(a);

    int b;
    b = fun_sub(1, 2);
    print(b);

    int c;
    c = fun_mul(1, 2);
    print(c);

    return 0;
  }
}
```

Listing 23: Test 3

```
//stack operations on DFA

void DFA main()
{
  State_1  //error - first transition must be Start state
  {
    stack<float> s;
    s.push(10.0);
    s.push(3.5);
    float a = s.peek();
    State_2 <- (a > 0);
    End <- $;
  }
  State_2
```

```
15   {
16     print("peek value is positive");
17     return;
18   }
19   End
20   {
21     print("peek value is negative");
22     return;
23   }
24
25 }
```

Listing 24: Test 4

```
1  // compare two integers
2
3  void DFA main()
4  {
5    Start
6    {
7      int a = 10;
8      int b = 5;
9      a = a + 1.0; //error (1.o is float and a is int)
10     Printx <- (a >= b);
11     Printy <- $;
12   }
13   Printx
14   {
15     print("a is greater than or equal to b");
16     return;
17   }
18   Printy
19   {
20     print("a  is less than b");
21     return;
22   }
23
24 }
```

Listing 25: Test 5

10 Appendix

10.1 Lexer Code (Lexer.mll)

```
{
open Printf
open Parser
}
let digit  = ['0'-'9']
let character = ['A'-'Z' 'a'-'z']

rule tokens =
parse
| [' ' '\t' '\r' '\n'] { tokens lexbuf }
| "/*"     { print_endline "multiline comments start"; comment lexbuf }
| "//"     { print_endline "singleline comments start";singleComment lexbuf }
| ';'      { SEMICOLON}
| ':'      { COLON }
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LBRACKET }
| ']'      { RBRACKET }
| ','      { COMMA }
| '.'      { DOT }
| '"'      { APOS}

| '+'      { PLUS }
| '-'      { MINUS}
| '*'      { MULTIPLICATION }
| '/'      { DIV }
| '^'      { CONCATENATE }
| '='      { ASSIGN }
| "=="     { EQ }
| '!'      { NOT }
| "!="     { NEQ }
| "&&"     { AND }
| "||"     { OR }
|"true"    { TRUE }
|"false"   { FALSE }
| '<'      { LT }
```


```
| "<-"     { ARROW }
| "<="     { LEQ }
| '>'      { GT }
| ">="     { GEQ }
| '%'      { MOD }
| "return" { RETURN }
| "print"  { PRINT}
| '$'      { DEFSTATE}

| "int"    { INT }
| "float"  { FLOAT }
| "string" { STRING }
| "void"   { VOID }
| "DFA"    { DFA }
| "stack"  { STACK }
| "pop"    { POP }
| "peek"   { PEEK }
| "push"   { PUSH }
| "EOS"    { EOS }
| (digit)+ as lexemme                {INTEGER_LITERAL(int_of_string lexemme)}
| (digit)+ '.'(digit)+ as lexemme    {FLOAT_LITERAL(float_of_string lexemme)}
| ['a'-'z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| ['A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { STATE(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 }
| _        {printf "INVALID TOKEN";tokens lexbuf}
| _ {tokens lexbuf }
and comment = parse
|  "*/" { print_endline "multiline comments end"; tokens lexbuf }
| _    { comment lexbuf }

and singleComment = parse
|  '\n' { print_endline "singleline comment end"; tokens lexbuf }
| _    { singleComment lexbuf }
```

10.2 Parser Code (parser.mly)

```
%{ open Ast
```


```
    let parse_error s = print_endline s; (* Called by parser on error *)
flush stdout
%}

/* Tokens defined according to the preference */
%token SEMICOLON LPAREN RPAREN LBRACE RBRACE COMMA RBRAC
LBRAC COLON DOT LBRACKET RBRACKET APOS
%token PLUS MINUS MULTIPLICATION DIV ASSIGN PUSH POP PEEK CONCATENATE
%token EQ NEQ LT LEQ GT GEQ MOD
%token BITAND BITOR AND OR NOT TRUE FALSE
%token RETURN SSTATE DEFSTATE ESTATE STATE PRINT DEFSTATE ARROW
%token DFA STACK
%token <int> INTEGER_LITERAL
%token <string>  ID
%token <string> STRING_LITERAL
%token <string> STRING_LITERAL TYPE STATE
%token <float> FLOAT_LITERAL
%token EOF EOS
%token MAIN BOOLEAN
%token STRING INT VOID FLOAT DOUBLE

%right ASSIGN
%left OR BITAND BITOR
%left AND
%right NOT
%left EQ NEQ LT GT LEQ GEQ
%left PLUS MINUS
%left MULTIPLICATION DIV MOD
%right UMINUS
%left PUSH POP PEEK
%nonassoc LPAREN RPAREN LBRAC RBRAC LBRACKET RBRACKET


/* start program */
%start program
%type <Ast.program> program

%%

/* Grammer Rules */
```


```
program:
  {[]}
  | automaton_declaration program { $1 :: $2 }

Data_type:
    | VOID                  {print_endline "found void"; Void}
    | INT                   {print_endline "found int"; Int}
    | FLOAT                 {print_endline "found float"; Float}
    | STRING                {print_endline "found string"; String}

return_type:
    | Data_type {Datatype($1)}
    | STACK LT Data_type GT {Stacktype(Datatype($3))}

automaton_declaration:
    return_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:
    | {[]}
    | variable_declaration vdecl_list { $1 :: $2 }

variable_declaration:
    | Data_type ID SEMICOLON             { print_endline
    "int declared";VarDecl(Datatype($1), Ident($2)) }
    | Data_type ID ASSIGN expr SEMICOLON { print_endline "int assigned
    to identifier";VarAssignDecl(Datatype($1), Ident($2), ExprVal($4))}
    | STACK LT Data_type GT ID SEMICOLON { print_endline
    "stack declared";VarDecl(Stacktype(Datatype($3)), Ident($5)) }

node_list:
    | {[]}
    | node node_list { $1 :: $2 }
```


```
node:
  | STATE LBRACE statement_list RBRACE {print_endline
  "found transition"; Node(Ident($1), $3) }

statement_list:
| {[]}
| statement statement_list { $1 :: $2 }

/* TODO: add method calls */
statement:
   | RETURN expr SEMICOLON  {print_endline
   "found return statement";Return($2)}
   | STATE ARROW expr SEMICOLON {print_endline
   "change in transition";Transition(Ident($1),$3)}
   | STATE ARROW DEFSTATE SEMICOLON {print_endline
   "default transition";Transition(Ident($1),IntLit(1))}
   /*Star evaluates to IntLit 1 because that's True in StateMap*/
   | variable_declaration {print_endline
   "variable declaration"; Declaration($1)}
    | ID ASSIGN expr SEMICOLON { print_endline
    "assignig value";Assign(Ident($1), $3) }
    /*Assignment post-declaration*/
   | expr SEMICOLON {Expr($1)}
    | RETURN SEMICOLON {print_endline
    "found return";Return(IntLit(1))}
    | PRINT LPAREN expr RPAREN SEMICOLON {Return($3)}
    /*| PRINT LPAREN APOS ID APOS RPAREN SEMICOLON
    {print_endline "found printf statement"; Print2($2)}
    | PRINT LPAREN APOS expr APOS RPAREN SEMICOLON        */

formals_opt:
    | {[]} /*nothing*/
    | formal_list { List.rev $1}

formal_list:
    | param { [$1] }
    | formal_list COMMA param {print_endline "function parameters"; $3 :: $1}

param:
      | Data_type ID { print_endline "function parameter";Formal(Datatype($1),Ident($2))
```


```
      | STACK LT Data_type GT ID { print_endline
      "function parameter(stack)";
      Formal(Stacktype(Datatype($3)), Ident($5)) }

expr_list:
    | {[]}
    | expr COMMA expr_list { $1 :: $3 }
    | expr { [$1] }

expr:
  | INTEGER_LITERAL    {  print_endline "found digit";IntLit($1)   }
  | STRING_LITERAL     {  print_endline "found string";StringLit($1) }
  | FLOAT_LITERAL      {  print_endline "found float";FloatLit($1) }
  | ID                 {  print_endline "found ID";Variable(Ident($1))  }
  | EOS                {  EosLit }
  | expr PLUS   expr   {  print_endline "found +";BinaryOp($1, Add,   $3) }
  | expr MINUS  expr   {  print_endline "found -";BinaryOp($1, Sub,   $3) }
  | expr MULTIPLICATION  expr    {  print_endline "found *";BinaryOp($1, Mult,  $3) }
  | expr DIV expr      {  print_endline "found /";BinaryOp($1, Div,   $3) }
  | expr EQ     expr   {  print_endline "found =";BinaryOp($1, Equal, $3) }
  | expr NEQ    expr   {  print_endline "found !=";BinaryOp($1, Neq,   $3) }
  | expr LT     expr   {  print_endline "found <";BinaryOp($1, Lt,  $3) }
  | expr LEQ    expr   {  print_endline "found <=";BinaryOp($1, Leq,   $3) }
  | expr GT     expr   {  print_endline "found >";BinaryOp($1, Gt,$3)}
  | expr GEQ    expr   {  print_endline "found >=";BinaryOp($1, Geq,   $3) }
  | expr MOD    expr   {  print_endline "found mod";BinaryOp($1, Mod,   $3) }
  | expr AND expr   {  print_endline "found AND";BinaryOp($1, And, $3) }
  | expr OR expr    {  print_endline "found OR"; BinaryOp($1, Or, $3) }
  | MINUS expr %prec UMINUS           {  Unop(Neg, $2) }
  | NOT   expr               {  Unop(Not, $2) }
  | LPAREN expr RPAREN       {  $2 }
  | ID DOT POP LPAREN RPAREN    {  print_endline "found pop";Pop(Ident($1)) }
  | ID DOT PUSH LPAREN expr RPAREN  {  print_endline "found push";Push(Ident($1), $5) }
  | ID DOT PEEK LPAREN RPAREN    {  print_endline "found peek";Peek(Ident($1)) }
  | ID LPAREN expr_list RPAREN  {  Call(Ident($1), $3) (*call a sub dfa*)}
```

10.3 Ast Code (ast.ml)

```
type var_type =
| Int
| String
```


```
| Stack
| Float
| Void
| Eos

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


```
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
```

10.4 Semantic Code (semantic.ml)

```
open Printf
open Ast


exception Error of string

type dfa_table =
{
dfas: (datatype * ident * formal list * sstmt list * snode list) list
}

type symbol_table =
{
variables: (ident * datatype * value option) list;
parent: symbol_table option;
}

type translation_environment =
{
dfa_lookup: dfa_table;     (* defined above*)
node_scope: symbol_table; (* defined above*)
return_seen: bool;         (* bool type *)
return_type: datatype;     (* int, float, string or stacktype*)
location: string;
}
```


```
let find_dfa (dfa_lookup: dfa_table) name =
List.find (fun (_,s,_,_,_) -> s=name) dfa_lookup.dfas

let get_ident_name ident = match ident with
Ident(n) -> n

let rec get_type_from_datatype = function
| Datatype(t)->t
| Stacktype(ty) -> get_type_from_datatype ty
| Eostype(t) -> Void

let math_logic x y = match (x, y) with     (* math logic =, !=, etc*)
| (String,String) -> (Int,true)
| (Int,Int) -> (Int,true)
| (Int,Float) -> (Int,true)
| (Float,Float) -> (Int,true)
| (Float,Int) -> (Int,true)
| (_,_) -> (Int,false)

let math_sym x y = match (x, y) with       (* math symbols >,<,=>,<= etc*)
| (Int,Int) -> (Int,true)
| (Int,Float) -> (Int,true)
| (Float,Int) -> (Int,true)
| (Float,Float) -> (Int,true)
| (_,_) -> (Int, false)

let math_operators x y = match (x, y) with  (* math logic =, !=, etc*)
| (String, String) -> (String, true)
| (Int, Int) -> (Int, true)
| (Int, Float) -> (Float, true)
| (Float, Float) -> (Int, true)
| (Float, Int) -> (Float, true)
| (_,_) -> (Int, false)


let get_binop_return_value operator p q =
let x = get_type_from_datatype p and
y = get_type_from_datatype q in
let (t, valid) =
    match operator with
        | Neq -> math_logic x y
```


```
        | Equal -> math_logic x y
        | Mult -> math_operators x y
        | Div -> math_operators x y
        | Lt -> math_sym x y
        | Leq -> math_sym x y
        | Gt -> math_sym x y
        | Add -> math_operators x y
        | Sub -> math_operators x y
        | Geq -> math_sym x y
        | And -> math_sym x y
        | Or -> math_sym x y
        | Mod -> math_operators x y

    in (Datatype(t), valid)



let update_variable env (temp, dt, value) = (* semantic check for scope errors*)
let ((_,_,_), location) =
try (fun node_scope -> ((List.find (fun (s,_,_) -> s = temp ) 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 - 1"))
    in
    (fun node_scope -> ((List.find (fun (s,_,_) -> s = temp)
    node_scope),2)) globalScope.variables
        with Not_found -> raise(Error("Not Found exception in update_variable"))in
let new_envf =
match location with
    | 1 ->                          (* Variables in the node *)
        let new_vars = List.map (fun (n, t, v) -> if(n = temp) then
        (temp, dt, 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 ->                          (* Variables in the DFA *)
        let globalScope = match env.node_scope.parent with
          Some scope -> scope
        | None -> raise(Error("No Global Scope - 2"))
        in
        let new_vars =
        List.map (fun (n, t, v) -> if(n = temp) then
        (temp, dt, 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"))  (* scope Not defined *)
in new_envf

let find_local_variable env temp =  (* find the local variable *)
List.find (fun (s,_,_) -> s = temp) env.node_scope.variables

let get_name_type_from_formal env = function
Formal(datatype,ident) -> (ident,datatype,None)

let find_variable env temp =  (*find the variable*)
try List.find (fun(s,_,_) -> s = temp) env.node_scope.variables
with Not_found ->
let globalScope = (match env.node_scope.parent with
  Some scope -> scope
  |None -> raise(Error("No Global Scope - 3")))  (*global scope not present*)
  in List.find(fun (s,_,_) -> s=temp) globalScope.variables



let rec check_expr env exp =(*check the given expression compatible or not*)
match exp with
| StringLit(s) -> Datatype(String)
```


```
| IntLit(i) ->Datatype(Int)

| FloatLit(f) -> Datatype(Float)

| EosLit -> Eostype(Eos)

| Variable(v) ->  let (_,s_type,_) = try find_variable env v with
       Not_found ->
           raise (Error("Undeclared Identifier ")); in s_type
           (*If there is an undefined id used in the code *)
| Unop(u, exp) -> let t = check_expr env exp in
    (match u with
         _ -> if t = Datatype(Int) then t else if t = Datatype(Float) then t
                  else
                      raise (Error("Cannot perform operation on " )))
                      (* operation can not be performed*)

| BinaryOp(e1, b, e2) -> let p = check_expr env e1 and q = check_expr env e2 in
    let (t, valid) = get_binop_return_value b p q in
    if valid || e1 = EosLit || e2 = EosLit
    then t else raise(Error("Incompatible types with binary operator"));
    (*binary operations are not compatible*)

| Peek(id) -> let (_,p,_) = (find_variable env id) in p
          (*peek operation*)

| Push(id, exp) -> let (_,p,_) = (find_variable env id) and q = (*push operation*)
    check_expr env exp
in (if not (p = Stacktype(q)) then (raise (Error("Mismatch in types for assignment"))));

| Pop(id) -> let (_,p,_) = (find_variable env id) in p         (*pop operation*)

| 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("state")  || id = Ident("input") ||
            id = Ident("concurrent" ) )
            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 temp =
if env.location = "dfa" then DFAScope
else
try (let (_,_,_) = List.find (fun (s,_,_) -> s = temp)
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 Scope - 4")))
in try (let (_,_,_) = List.find(fun (s,_,_) -> s = temp)
globalScope.variables in DFAScope)
with Not_found -> raise(Error("get_node_scope is failing"))

let rec get_sexpr env exp =
match exp with
  | Peek(id) -> SPeek(SIdent(id, get_node_scope env id),
    check_expr env exp)
  | Push(id, ex) -> SPush(SIdent(id, get_node_scope env id),
    get_sexpr env ex,check_expr env exp)
  | Pop(id) -> SPop(SIdent(id, get_node_scope env id),
    check_expr env exp)

  | StringLit(s) -> SStringLit(s,Datatype(String))
  | IntLit(i) -> SIntLit(i, Datatype(Int))
  | FloatLit(d) -> SFloatLit(d,Datatype(Float))
```


```
  | 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 exp)
  | Variable(id) -> SVariable(SIdent(id, get_node_scope env id), check_expr env exp)
  | BinaryOp(e1,b,e2) -> SBinop(get_sexpr env e1,b, get_sexpr env e2,check_expr env exp)
  | Unop(u,ex) -> SUnop(u, get_sexpr env ex, check_expr env exp)

  | EosLit -> SEosLit

let get_sval env = function
ExprVal(expr) -> SExprVal(get_sexpr 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_val_from_decl temp =
match temp with
| VarDecl(datatype, ident) -> (ident, datatype, None)
| VarAssignDecl(datatype, ident, value) -> (ident, datatype, Some(value))

let get_name_type_from_decl temp =
match temp with
| VarDecl(datatype, ident) -> (ident, datatype)
| VarAssignDecl(datatype,ident,value) -> (ident,datatype)

let get_name_type_from_var env = function
| VarDecl(datatype,ident) -> (ident,datatype,None)
| VarAssignDecl(datatype,ident,value) -> (ident,datatype,Some(value))

let get_datatype_from_val env = function
ExprVal(expr) -> check_expr env expr


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 add_to_var_table env name t v =

let new_vars = (name,t, v)::env.node_scope.variables in

let new_sym_table = { variables = new_vars;
                      parent = env.node_scope.parent; } in
let new_env = {env with node_scope = new_sym_table} in
new_env


let check_final_env env =
(if(false = env.return_seen && env.return_type <> Datatype(Void)) then
    raise (Error("Missing Return Statement")));

true

let match_var_type env v t =
let(name,ty,value) = find_variable env v in
if(t<>ty) then false else true


(* defined names and thier initialization *)
let empty_table_initialization = {parent=None; variables =[];}
let empty_dfa_table_initialization = {
dfas=[
    (Datatype(String), Ident("input"),[],[],[]);
    (*The built-in get-user-input function*)

    (Datatype(String), Ident("state"),
    (*The built-in 'get state' function for concurrently running dfas *)
    [Formal(Datatype(String),Ident("dfa"))],[],[]);

    (Datatype(Void), Ident("print"),
    (*The built-in print function (only prints strings)*)
```


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

    (Datatype(String), Ident("state"),
    (*The state() function to get states of concurrently running dfas*)
    [Formal(Datatype(String),Ident("dfa"))],[], []);

    (Stacktype(Datatype(String)), Ident("concurrent"), [] ,[], [])
    (* The built-in concurrent stringhow 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

| 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"));
        (*if the return wrote is wrong, incompaitable*)

    let new_env = {env with return_seen=true} in
    (SReturn(get_sexpr env e), new_env)

| 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)

| 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)

| 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)

| 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
                (*If they declared multiple times*)

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_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 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 get_snodeBody 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"))(*error prints no catch at all*)

          else
            block
) ([],env) node_list


let check_for_Start node_list =
(*The first transition must be Start or else semantic error*)
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 =  (*semantic check for all the other transitions*)
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
(*It prints invalid state if the transition does not follow all the rules*)


let check_main env str =    (*check whether dfa main name is proper or not*)
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 there is any error in defining the name of function*)
if dt <> Datatype(Void) then
  raise(Error("main DFA needs void return type"))
  (*If there is an error in the return type of the function it prints this statement*)

let check_dfa env dfa_declaration =   (*semantic check on the dfa declaration *)

try(let (_,_,_,_,_) =  find_dfa env.dfa_lookup dfa_declaration.dfa_name in
raise(Error("DFA already declared"))) with
(*prints it if the dfa is aldready declared*)

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_snodeBody 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_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)
```

10.5 Code-Gen code (gen_python.ml)

```
open Ast
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
```

10.6 main.ml

```
open Printf

let main () =
  try
    let lexbuf = Lexing.from_channel stdin in
    let prog = Parser.program Lexer.tokens lexbuf in
    let code=Semantic.check_program prog in
         Gen_python.gen_program code
 with End_of_file -> printf "Hurray"; exit 0
let _ = Printexc.print main ()
```

10.7 make file

```
OBJS = ast.cmo parser.cmo lexer.cmo semantic.cmo gen_python.cmo main.cmo

gen_python: $(OBJS)
ocamlc str.cma -o gen_python $(OBJS)

lexer.ml:lexer.mll
ocamllex lexer.mll
parser.ml parser.mli:parser.mly
ocamlyacc -v $<
%.cmo : %.ml
ocamlc -c $<
%.cmi : %.mli
ocamlc -c $<
# parser.cmo:parser.cmi
ast.cmo : ast.cmi
ast.cmx : ast.cmi
parser.cmo : ast.cmi parser.cmi
parser.cmx : ast.cmx parser.cmi
lexer.cmo : parser.cmi
lexer.cmx : parser.cmx
semantic.cmo : ast.cmi
semantic.cmx : ast.cmx
ast.cmi :
parser.cmi : ast.cmi
gen_python.cmo : ast.cmi
gen_python.cmx : ast.cmx
```


```
.PHONY : clean
clean:
rm -rf parser.mli lexer.ml parser.ml *.cmi *.cmo gen_python parser.output
```