YAFSM Language Overview

Compilers Project

Group 6

Authors

Vishal Vijay Devadiga (CS21BTECH11061)

Mahin Bansal (CS21BTECH11034)

Harshit Pant (CS21BTECH11021)

Satpute Aniket Tukaram (CS21BTECH11056)

भारतीय प्रौद्योगिकी संस्थान हैदराबाद

Indian Institute of Technology Hyderabad

Introduction

What is YAFSM?

YAFSM full form is "Yet Another Finite State Machine". YAFSM is a domain specific language that simplifies working with Finite State Machines(FSMs).

FSMs include Deterministic Finite Automata(DFAs), Non-Deterministic Finite Automata(NFAs), Pushdown Automata(PDAs), Context Free Grammars(CFGs), Regular Expressions(REs), etc.

Why YAFSM?

YAFSM is a simple language that allows users to define Finite State Machines in a simple and intuitive way. Example:

```
<!-- A simple DFA that accepts strings starting with 01 --!>
dfa x;
x.Q = {q0, q1, q2, q3};
x.Sigma = {a: "0", b:"1"};
x.delta = {
    (q0, a -> q1),
    (q1, b -> q2),
    (q2, a -> q2),
    (q2, b -> q2),
    (q0, b -> q3),
    (q1, a -> q3),
    (q3, a -> q3),
    (q3, b -> q3)
};
x.q0 = q0;
x.F = {q2};
```

Technical Specifications

We have used Flex and Bison for implementing the lexer and parser respectively in C, while the semantic analysis and code generation phases have been implemented in C++.

As per the general practice regarding C++ code,

1) Header files are included in the `include` directory.
2) Source files are included in the `src` directory.

3) During compilation, the object files are generated in the `build` directory.
4) Then, these object files are linked to generate the executable in the `bin` directory.

Compilation Process

We have included a `Makefile` in the root directory of the project, which can be used to compile the compiler.

Compilation of the Compiler

Running default `make` command will compile the compiler and generate the executable in the `bin` directory.

```
make
```

Running the Compiler

Cleaning the Project

The project can be cleaned using the following command:

```
make clean
```

Running the Tests

Our tests have the following naming convention(where x is a number): - `tx.txt` is a test file that passes. - `ex.txt` is a test file that fails.

Note that there might be cases where the test file should pass but fails due to a phase of the compiler after the phase that is being tested.

The tests can be run using the following command:

```
make test
```

This will generate the output in the `output` directory.

Running the FSM code

Let the name of code be 'code.fsm'. Then, the code can be run using the following command:

```
make run <path_to_code_file>
```

This will first transpile the code to C++ code and then compile it using g++ and run the executable.

Difficulties Faced

1) During implementation of lexer/parser, we faced some issues with defining our own regex(part of the language's specification) format. We eventually decide to scrap all of the implementation and use the POSIX regex library provided by C.
2) We also faced some issues in implementing the parser for the FSMs, that is, initially there were a lot of shift/reduce conflicts in the grammar, which we eventually resolved by improving the grammar.
3) We faced a lot of issues in implementing the semantic analysis phase of the compiler, part of the issue being the wide scope of the language. This is evident from the fact that our semantic analysis phase is the largest part of the compiler(Around 4000 lines).
4) Similarlly we faced a lot of issues in coding the code generation phase of the compiler. This is because of the fact that we had to generate efficient code for implementing the FSMs. This required us to use a lot of data structures and algorithms which was a bit difficult to implement.