450 Programming Project #1

Due Date: Saturday, 9/25/2010

Overview

In this assignment you will write a program named `DFAGen` that given a specification for a deterministic finite automata, will generate a C# program that provides an implementation of the DFA. Like the program Lex and YACC, this program will take as input a file that describes the DFA to be generated. The specifications for the content of this file follow.

A `DFAGen` program consists of one or more state declarations in any order, followed by the definition of the start state. Each state declaration starts with a state identifier, consisting of any number of letters or digits, followed by a ':' for an ordinary state or a '@' for an accept/terminal state. States can be defined in an order, and referred to both before and after they are declared.

Each state identifier is followed by one or more pairs of symbol-string and state identifier separated by ',' (coma) and terminated by a ';' (semi-colon). The symbol-string consists of one or more characters between a pair of apostrophes ('), any one of which when input by the DFA causes a transition to the state given by the state identifier. Almost any characters can appear between the apostrophes containing a symbol-string with the exception of a newline character and an apostrophe. Spaces, tabs and newlines can appear between any components of the input, but not inside a state identifier. A space or tab (or both) can appear inside a symbol-string, in which case it is significant, but a newline cannot.

Single line comments that begin with '//' may be placed in a program file.

The sample DFAGen program below, defines a DFA that will recognize integers or real numbers, proceeded by an optional sign.

```
// DFA to accept integers and real numbers
Start:   ' ' Start, '+' Sign, '-' Sign, '0123456789'
Integer; // optional sign
Sign:     '0123456789' Integer;
Integer@  '0123456789' Integer, '.' Decimal;
Decimal:  '0123456789' Real;
Real@     '0123456789' Real;
*Start
```

Program Specifications

You are to write a program named `DFAGen` that will generate a C# class definition (source code) that contains a public method `scan()`. The `scan()` method takes a string as an argument and returns a boolean value. When invoked `scan()` will "run" the DFA encoded by the program on the string passed as an argument. If the DFA Terminates in a final state, the method will return `true`, otherwise it will return `false`. If the DFA is given a string for which a transition has not been defined, `scan()` will stop processing the string and return `false`.

The class generated by your program will also provide a public method named `setDebug()` that takes a single `boolean` argument. If invoked with the argument `true` the DFA will be placed in debug mode, which will cause it to print out each input character in the string passed to `scan()`, as the characters are processed by the DFA. The input character will be followed by the identifier of the new state.

The C# program below illustrates how to use the code generated by `DFAGen`. Note that this example assumes that the name of the class generated by `DFAGen` is `NumberDFA`.

```
namespace ConsoleApplication1
{
    class TestDFA
    {
        static void Main(string[] args)
        {
            NumberDFA dfa = new NumberDFA();
            Console.WriteLine(dfa.scan("1234"));
            Console.WriteLine(dfa.scan("1234.56"));
            Console.WriteLine(dfa.scan("123-45"));

            dfa.setDebug(true);
            Console.WriteLine(dfa.scan("-123"));
        }
    }
}
```

Output generated by the program:

```
C:\> TestDFA
true
true
false
{Start}-{Sign}1{Integer}2{Integer}3{Integer}
true
```

The DFA Generator program, DFAGen is invoked as shown below:

`DFAGen [-d] DFAProgramFile C#ClassName`
where *`DFAProgramFile`* is a text file that contains a `DFAGen` program as defined above and *`C#ClassName`* is a valid C# identifier that specifies the name of the class file to generate (which also means it gives the name of the output file).

The `-d` option specifies that the program is to run in debug mode (note this applies to the `DFAGen` program and not necessarily to the generated code). When run in debug mode `DFAGen` will print messages to standard output that explain what it is currently doing. I am not going to specify the format of the debugging output, except to say that minimally your program must print out the productions it is using as it parses the DFA program file.

If your program is invoked incorrectly (i.e., command line arguments incorrect, can't open input file, etc.) it will print an appropriate message to standard error and terminate. If your program is given an invalid DFA program file (i.e., syntactically or semantically invalid) you only need to print a message describing the error in very general terms and terminate. If you wish to, and have the time, you may want to add more sophisticated error handling to the program.

Finally your programs must be written in C# and generate C# code. Additionally you must use a tool to generate the C# code for the scanning portion of this project (C# Lex Generators). You may use a different tool if you wish. Finding and installing those tools are part of this assignment.

It is not required to use a parsing generator for this assignment. Feel free to use one if you wish, but you might find it easier to simply write your own recursive descent compiler.

Getting Started

Obviously you need to do a bit of work to complete this assignment. I would recommend that you start out by finding the scanner generator that you will use, installing it on your machine, and learning how to use it. If you decide to use a parser generator you should take care of that next.

After you have decided on the tools you plan to use for this project, the next step would be to write the regular expressions that define the tokens your program will recognize and the grammar for the input file. You will note that I have given you the general format of the input file, but I purposely did not provide a grammar. Again that is part of the assignment.

After you have defined the syntax of your input file, I would recommend using the scanner generator to create a scanner for your program. Be sure to run the scanner by itself and verify that it recognizes the tokens correctly and handles

malformed input. Once you have finished this step you will be ready to work on the parser.

As you develop the parser for your program, I would suggest doing this step by step. First simply build a parser that will recognize a valid file and prints out the productions as it processes the file (similar to the program I showed in class). As you do this think about the debug output that you will generate and make sure you have a framework for handling errors. Remember to get full credit for this assignment your error handling can be as simple as printing out "Invalid Input File" and terminating.

Once you have your parser, well parsing, then you can start adding the code necessary to generate the DFA recognizer. Before you start attempting to generate code, you should probably have a good idea what the code you are trying to generate will look like. At this point it might be a good idea to code up a simple DFA recognizer that meets the specifications of this assignment. I would recommend using a table driven approach. This would mean that all your DFA generator needs to do is figure out the table required for each DFA.

You may find it useful to put a template for your recognizer in a file that will be read and modified by your program. If you decide to do this you may assume that file will be in the same directory as your program when it is executed. Note that your program does not have to generate well-commented "stylized" code. It may be easier to locate problems if the generated code is readable, but you will not be graded on the style of the generated code.

Once you are comfortable with the code you plan to generate, start working on modifying your parser to actually generate the code.

What to Turn In

In order to obtain full credit for this assignment you must turn in all the source code required to generate your program. You **do not** have to turn in the tools you used to generate code for this project, but you **must** turn in the generated files so that can compile and run your program.

In addition to the source code, you must also turn in a report that gives the regular expressions you used to define your tokens, and the grammar you used to generate your parser. This file should also contain a few words about the tools that you used to generate you program and any information you think I might need when running your program. This file **must** be in PDF form.

Style

Write clean, modular, well-structured, and well-documented code.

Here are some general guidelines:

1. Make sure that your name appears somewhere in each source file.

2. Each class and every method in the class should have a header that describes what the class/method does.

3. You do not have to document every single line of code you write. Provide enough documentation so someone who knows C# can understand what your program is doing

4. Names of classes, method, constants, and variables should be indicative of their purpose.

5. All literal constants other than trivial ones (e.g. 0 and 1) should be defined symbolically.

Submitting Your Work

Place all the files in a single directory and create a zip file of the directory and call it assign1. This zip file should be submitted through myCourses under the project 1 dropbox.

You may include a README file in this directory, if you wish, but no other files are acceptable.