LING 473 – Project 3
Summer 2015

Andrew Troelsen

Preamble

For this project, we are given an input block of Thai data. As mentioned in the notes, it is common in Thai orthography to not separate word and/or syllable breaks by spaces. The goal here is to use a state machine to parse the input and return the correct word / syllable breaks via the provided sets of vowels, consonant and tone marks.

The input for this program is found at the following location on PATAS:

/opt/dropbox/15-16/473/project3/fsm-input.utf8.txt.

Programming Environment and File Set

Once again, I chose the C# language for this project as implemented under the Mono platform. The code was developed using Xamarin Studio on my Mac OS X computer.

When I began this project, I made use of a simple switch / case construct to model the correct state transitions. After that work was completed, I decided to try and implement a second version of the machine using the Dictionary object states and Func<> objects (with the related lambdas). Because of their common functionally, I also created an interface type to model the core behavior of the machine.

This all being said, I ended up with the following files:

- **Program.cs**: Program Entry Point
- **IThaiSyllabifier.cs**: Interface definition.
- **ThaiSyllabifierCaseLogic.cs**: State machine using switch / case logic.
- **ThaiSyllabifierLambdaLogic.cs**: State machine using lambdas / Func<> objects.
- **StateReturnData.cs**: Currently, my lambda-centric state machine requires the use of a custom class to encapsulate the returned state and text output….this is where I am currently stuck (see below).

Full disclosure! At the time I am authoring this write up- the ThaiSyllabifierLambdaLogic class does not work correctly- I am hoping that I can figure out my error as I go through the write up! But, as far as I can tell, the case logic works fine.

A walkthrough of the key code points can be found over the next several pages.

The Common Interface and Core Main() Logic

Both of my state machine implement the following interface:

```
public interface IThaiSyllabifier
{
   void SyllabifyString(string input);
   string GetOutputString();
}
```

As you can see, the SyllabifyString() method will take a line of Thai input and insert breaks based on the provided state machine notes.

I am returning void from this method. Initially I returned a string- but once I started doing the lambda version of the machine, I realized I needed to return other data (at least it seems that way). Given this, the shared interface also provides a method named GetOutputString() which will returned the syllabified string. I think I would clean this up if I had time- but this is what I am going with for now!

Given that ThaiSyllabifierCaseLogic and ThaiSyllabifierLambdaLogic both implement this interface, I created a simple factory method which will return the correct underlying object based on the presence of –lambda in the command line arguments (this method is in the Program class):

```
static IThaiSyllabifier BreakerFactoryMethod(string[] args)
{
   // See if user wants to use lambda version.
   if (args.Contains ("-lambda"))
      usingLambda = true;

   // Create the buster.
   IThaiSyllabifier breaker = null;
   if (!usingLambda)
      breaker = new ThaiSyllabifierCaseLogic ();
   else
      breaker = new ThaiSyllabifierLambdaLogic ();

   return breaker;
}
```

The Main() method uses the underlying object as so (htmlOutput is a StringBuilder object):

```
// Now loop over each line in the input file and write it out.
foreach (var currLine in File.ReadAllLines(inputFileLocation))
{
  breaker.SyllabifyString (currLine);
  htmlOutput.Append(breaker.GetOutputString());
  htmlOutput.Append("<br/>");
}
```

The last thing to mention at this point is that I did create a secondary interface during my programming time- but it is not currently used. The problem (as you will see) is how each machine handles processing the text characters and how it is returned back.

```
// I made a sub-interface- because the GetOutputString()
// already works for the case logic by returning a string.
// Do I need this?
// Currently not used!!
public interface ILambdaSupport : IThaiSyllabifier
{
    StateReturnData SyllabifyStringLambda(string input);
}
```

The ThaiSyllabifierCaseLogic Implementation

I was *really* surprised how quick (and straightforward) it was to implement the machine using a switch statement (and your notes!). I've read about state machines and did some pencil and paper work before- but this was my first time coding such a construct.

First, my class has member variables for each of the HashSets<> you listed in the assignment notes:

```
public class ThaiSyllabifierCaseLogic : IThaiSyllabifier
{
   #region Vowels, consonants and tones!
   // These HashSet objects contain the possible characters which can correspond to the
   // following input pattern:
   // [V1]C1[C2][V2][T][V3][C3]
   private HashSet<char> vowelSet1 = new HashSet<char>("ุูิีั");
…
   #endregion
…
}
```

Next, I make use of the following C# enum to mark each state (this machine does not use the State.Error value- but the other machine does, so I included it here):

```
// This enum defines the various states the machine can be in.
public enum State {Zero, One, Two, Three, Four, Five, Six, Seven, Eight, Nine, Error};
```

The SyllabifyString() method of the shared interface is implemented using the following algorithm:

- The machine states in State.Zero.
- Take the current input string object and loop over each character.
- Test if the current character is a member of any of the HashSet<> objects using a series of case statements.
- If a match is found, change to the new state (as listed in the assignment notes).
- If we reach an accepting state (State.Seven, State.Eight or State.Nine) we will either add a space before the character in question, or break at that point.

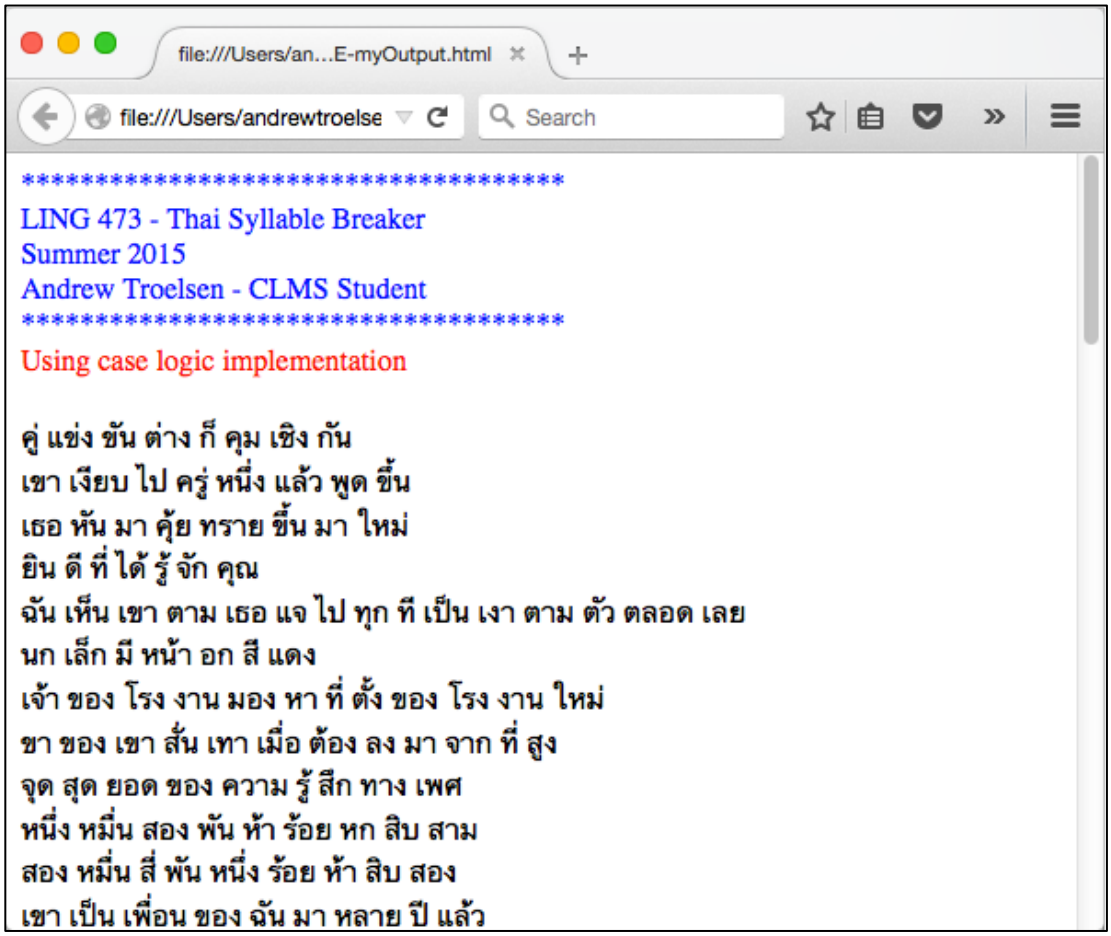
The full implementation of this method is quite long- so I won't repeat everything here. However, the next page showcases the crux of the logic:

```
public void SyllabifyString (string input)
{
   State currentState = State.Zero;

   outputString = new StringBuilder(input);

   // Loop over each character in the input.
   for(int currCharPos = 0; currCharPos < outputString.Length; currCharPos++)
   {
      // Switch on the current state.
      // When in a state, we will check the current character's membership
      // in a given HashSet<> and then transition to next state.
      switch (currentState)
      {
…
   case State.Two:

      if (conSet2.Contains (outputString[currCharPos]))
         currentState = State.Three;
      else if (vowleSet2.Contains (outputString[currCharPos]))
         currentState = State.Four;
      else if (toneSet.Contains (outputString[currCharPos]))
         currentState = State.Five;
      else if (vowelSet3.Contains (outputString[currCharPos]))
         currentState = State.Six;
      else if (conSet3.Contains (outputString[currCharPos]))
         currentState = State.Nine;
      else if (vowelSet1.Contains (outputString[currCharPos]))
         currentState = State.Seven;
      else if (conSet1.Contains (outputString[currCharPos]))
         currentState = State.Eight;
      break;
…
   case State.Seven: // Syllable break!
      currentState = State.One;
      outputString = outputString.Insert (currCharPos - 1, " ");
      break;

   case State.Eight: // Syllable break!
      currentState = State.Two;
      outputString = outputString.Insert (currCharPos - 1, " ");
      break;

   case State.Nine: // Syllable break!
      currentState = State.Zero;
      outputString = outputString.Insert (currCharPos, " ");
      break;
…
}
```

The implementation of the GetOutputString() method is simply to return the "broken" input text:

```
public string GetOutputString()
{
   return outputString.ToString();
}
```

So! If you run the program and do not provide the –lambda command line argument, the program will generate an HTML file named myOutput.html. When I load this up into a web browser, I find the following (partial) output:



By doing an eye-ball comparison of the fsm-output (intended).html file found in the /opt/dropbox/15-16/473/project3 folder, it appear to be the correct output!

The ThaiSyllabifierLambdaLogic Implementation

As mentioned- this is currently not working. I've tried several iterations of this, but I am stuck at the same place each time. I will describe what I have tried so far (you can see all the commented attempts in the C# code file). If I am lucky, maybe I'll have an "AH-HA!" moment while typing.

So, this class still makes use of a custom State enum. No changes there. The class also implements the common interface.

The biggest change is the use of a (massive) Dictionary<> object- I used your slides in lecture 8 as a template. My first iteration of this object mapped each case statement from the other machine using the same technique:

- The Dictionary<> is typed to operate of <State, Func<char,State>> tuples.
- The input to Func<> is the current character to look at. The return value is the new state.

In this partial implementation, you can see where I was trying to go:

```
private Dictionary<State, Func<char, State>> stateMachine = new Dictionary<State, Func<char, State>>
{
   {State.Zero, (ch) => {
                        if (vowelSet1.Contains (outputString[ch]))
                           return State.One;
                        else if (conSet1.Contains (outputString[ch]))
                           return State.Two;
                        else return State.Error; } },

   {State.One, (ch) => {
         if (conSet1.Contains (outputString[ch]))
            return State.Two;
         else return State.Error; }},
  …
}
```

I was able to get a clean compile, but I was receiving a null reference exception because the StringBuilder object (outputString) was not initialized. Suddenly I realized this approach would not really work, because this object can't be constructed to hold the current input string until after the Dictionary is created!

<< Thought- I could move this dictionary object as a local variable to the SyllabifyString() method! I will try that after I talk about my other attempts. >>

In my second attempt, I thought maybe a way around this scope problem was to have the Dictionary not simply return the new state, but a new class which holds the state and the current form of the output string (which was a StringBuilder at this point). The class was defined as so:

```
public class StateReturnData
{
   // The lambda version of the machine requires multiple
   // return values- so using this class type.
   public ThaiSyllabifierLambdaLogic.State newState;
   public StringBuilder outputString;

   public StateReturnData(ThaiSyllabifierLambdaLogic.State newState, StringBuilder outputString)
   {
      this.newState = newState;
      this.outputString = outputString;
   }
}
```

With this, the Func<> now looked like so:

```
Func<char, StateReturnData>
```

Some example dictionary entries now looked like so:

```
{State.Zero, (ch) => {
      if (vowelSet1.Contains (outputString[ch]))
         return new StateReturnData(State.One, outputString);
      else if (conSet1.Contains (outputString[ch]))
         return new StateReturnData(State.Two, outputString);
      else return new StateReturnData(State.Error, outputString); } },
...
{ State.Seven, (ch) => {
      outputString = outputString.Insert (ch - 1, " ");
      return new StateReturnData(State.One, outputString);}},

{ State.Eight, (ch) => {
      outputString = outputString.Insert (ch - 1, " ");
      return new StateReturnData(State.Two, outputString);}},

{ State.Nine, (ch) => {
      outputString = outputString.Insert (ch, " ");
      return new StateReturnData(State.Zero, outputString);}}
```

Again, I did get a clean compile….but got a stack overflow exception at runtime :-/
So it seemed that I was copying way too many StringBuilder objects and growing the internal character buffer too far.

So I changed this class to look like so:

```
public class StateReturnData
{
   // The lambda version of the machine requires multiple
   // return values- so using this class type.
   public ThaiSyllabifierLambdaLogic.State newState;
   public StringBuilder outputString = new StringBuilder();

   public StateReturnData(ThaiSyllabifierLambdaLogic.State newState, string outputString)
   {
      this.newState = newState;
      this.outputString.Append(outputString);
   }
}
```

In this case, notice I am attempting to use Append() rather than a straight assignment.

With this change, I went to my third attempt at the massive Dictionary<> object- which is where I am currently stuck.

Now, rather than having a dictionary entry try to consult the outputString object (which again, is currently not holding the input text!! See my realization in yellow above….) I am trying to take the input character and directly consult the HashSet<> objects. For example:

```
{State.Two, (ch) => {
      if (conSet2.Contains (ch))
         return new StateReturnData(State.Three, ch.ToString());
      else if (vowleSet2.Contains (ch))
         return new StateReturnData(State.Four, ch.ToString());
      else if (toneSet.Contains (ch))
         return new StateReturnData(State.Five, ch.ToString());
      else if (vowelSet3.Contains (ch))
         return new StateReturnData(State.Six, ch.ToString());
      else if (conSet3.Contains (ch))
         return new StateReturnData(State.Nine, ch.ToString());
      else if (vowelSet1.Contains (ch))
         return new StateReturnData(State.Seven, ch.ToString());
      else if (conSet1.Contains (ch))
         return new StateReturnData(State.Eight, ch.ToString());
      else return new StateReturnData(State.Error, ch.ToString()); }},
```

Because I was no longer referencing the outputString object, I had to change how I handled states Seven, Eight and Nine by building a temp string object via string.Format. Basically the location of the {0} insertion point is accounting for where to add a space (or not):

```
{State.Seven, (ch) => {
      //outputString = outputString.AppendFormat (" {0}", ch.ToString());
      return new StateReturnData(State.One, string.Format(" {0}", ch.ToString())); }},

{State.Eight, (ch) => {
      //outputString = outputString.AppendFormat (" {0}", ch.ToString());
      return new StateReturnData(State.Two, string.Format(" {0}", ch.ToString())); }},

{State.Nine, (ch) => {
      //outputString = outputString.AppendFormat ("{0} ", ch.ToString());
      return new StateReturnData(State.Zero, string.Format("{0}", ch.ToString())); }},
```

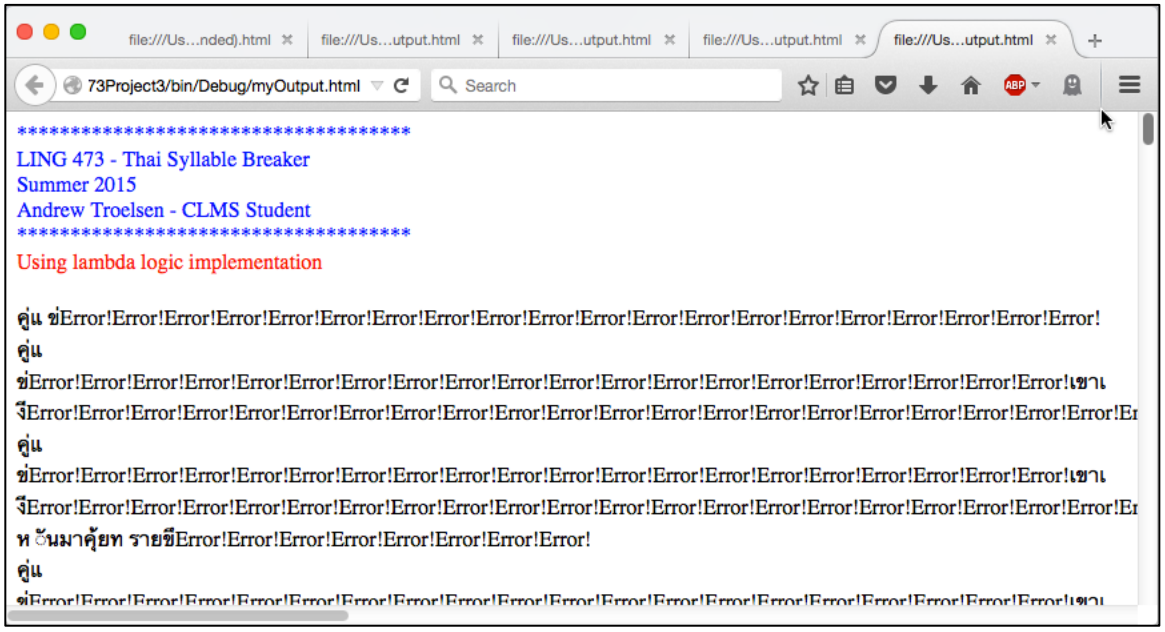
I also added an entry for State.Error like so:

```
{State.Error, (ch) => { return new StateReturnData(State.Error, "Error!");}}
```

Now, here is my current problem!! Most of my Func<> listings have a final “else if” which always transitions to State.Error:

```
{State.Six, (ch) => {
      if (conSet3.Contains (ch))
         return new StateReturnData(State.Nine, ch.ToString());
      else if (vowelSet1.Contains (ch))
         return new StateReturnData(State.Seven, ch.ToString());
      else if (conSet1.Contains (ch))
         return new StateReturnData(State.Eight, ch.ToString());
      else return new StateReturnData(State.Error, ch.ToString()); }},
```

So when I run this iteration- I no longer get a runtime exception (that is nice) but my output is 95% “Error!” tokens:



So this is where I was before I started doing my write up.

See next page for possible breakthrough!

Last Ditch Effort

So during this write up I realized my outputBuffer could never be initialized with the input Thai text, *because* the Dictionary was a class level variable. So I will now try to move the first iteration of the Dictionary into the scope of the SyllabifyString() method (screen shot to prevent huge dump of Dictionary contents):

```
public void SyllabifyString (string input)
{
    MASSIVE Dictionary of states and transitions!

    // StateReturnData state = new StateReturnData(State.Zero, "");
    State state = State.Zero;

    outputString = new StringBuilder(input);

    // Loop over each character in the input.
    for (int currCharPos = 0; currCharPos < outputString.Length; currCharPos++)
    {
        state = stateMachine [state] (outputString[currCharPos]);
    }
}
```

Nope. Now I am getting an index out of range error. Grrrr.

```
Dictionary<State, Func<char, State>> stateMachine = new Dictionary<State, Func<char, State>>
{
    {State.Zero, (ch) => {
                if (vowelSet1.Contains (outputString[ch]))
                    return State.One;
                else if (conSet1.Contains (outputString[ch]))
                    return State.Two;
                else return State.Error; } },
```

System.IndexOutOfRangeException has been thrown
Array index is out of range.
Show Details

~~Well Glenn, I think I am close here- but I really need to move onto the next program assignment this weekend. So I think I will leave this "as-is" for now. If I have time I really want to come back to this and see if I can figure it out. I think I am close! But I have my in-laws coming to visit next week, so I need to start on Project 4!!~~

~~Thanks!~~

I figured it out! I put the Dictionary object within the scope of the SyllabifyString() method. I also realized I was strangely not noticing that I was trying to index into the StringBuilder using the current CHARACTER not the current CHARACTER COUNT!!! That is why I was getting the index out of range exception above. So I fixed that- and I did not need that extra class for the return value after all.

Just because I am so darn happy now- I'll list out the entire implementation on the next page (or two) ☺

```
public void SyllabifyString (string input)
{
   // Make this visible to the whole method.
   int currCharPos = 0;

   #region MASSIVE Dictionary of states and transitions!
   // This dictionary holds a list of Func<> objects, which take the current character as input, and return the
   // new state.
   Dictionary<State, Func<char, State>> stateMachine = new Dictionary<State, Func<char, State>>
   {
      {State.Zero, (ch) => {
            if (vowelSet1.Contains (ch))
               return State.One;
            else if (conSet1.Contains (ch))
               return State.Two;
            else return State.Error; } },

      {State.One, (ch) => {
            if (conSet1.Contains (ch))
               return State.Two;
            else return State.Error; }},

      {State.Two, (ch) => {
            if (conSet2.Contains (ch))
               return State.Three;
            else if (vowleSet2.Contains (ch))
               return State.Four;
            else if (toneSet.Contains (ch))
               return State.Five;
            else if (vowelSet3.Contains (ch))
               return State.Six;
            else if (conSet3.Contains (ch))
               return State.Nine;
            else if (vowelSet1.Contains (ch))
               return State.Seven;
            else if (conSet1.Contains (ch))
               return State.Eight;
            else return State.Error; }},

      {State.Three, (ch) => {
            if (vowleSet2.Contains (ch))
               return State.Four;
            else if (toneSet.Contains (ch))
               return State.Five;
            else if (vowelSet3.Contains (ch))
               return State.Six;
            else if (conSet3.Contains (ch))
               return State.Nine;
            else return State.Error; }},


      {State.Four, (ch) => {
            if (toneSet.Contains (ch))
               return State.Five;
            else if (vowelSet3.Contains (ch))
               return State.Six;
```

```
            else if (conSet3.Contains (ch))
               return State.Nine;
            else if (vowelSet1.Contains (ch))
               return State.Seven;
            else if (conSet1.Contains (ch))
               return State.Eight;
            else return State.Error; }},

        {State.Five, (ch) => {
            if (vowelSet3.Contains (ch))
               return State.Six;
            else if (conSet3.Contains (ch))
               return State.Nine;
            else if (vowelSet1.Contains (ch))
               return State.Seven;
            else if (conSet1.Contains (ch))
               return State.Eight;
            else return State.Error; }},

        {State.Six, (ch) => {
            if (conSet3.Contains (ch))
               return State.Nine;
            else if (vowelSet1.Contains (ch))
               return State.Seven;
            else if (conSet1.Contains (ch))
               return State.Eight;
            else return State.Error; }},


        {State.Seven, (ch) => {
            outputString = outputString.Insert (currCharPos - 1, " ");
            return State.One;}},

        {State.Eight, (ch) => {
            outputString = outputString.Insert (currCharPos - 1, " ");
            return State.Two;}},

        {State.Nine, (ch) => {
            outputString = outputString.Insert (currCharPos, " ");
            return State.Zero;}}
    };
    #endregion

    // StateReturnData state = new StateReturnData(State.Zero, "");
    State state = State.Zero;

    outputString = new StringBuilder(input);

    // Loop over each character in the input.
    for (currCharPos = 0; currCharPos < outputString.Length; currCharPos++)
    {
       state = stateMachine [state] (outputString[currCharPos]);
    }
}
```