# CS-3240 Project Phase 1

James Bowland-Gleason, Wisdom Chen, Man Fong, Victor Lee
*jebg3@gatech.edu, cwisdom3@gatech.edu, mfong8@gatech.edu, victorlee42@gatech.edu*

Spring 2013

## Introduction

In this project we are implementing a scanner generator. We do this by two main parts: 1. fitting specifications to character classes and tokens through the help of a recursive descent parser and 2. creating a DFA table and a corresponding table walker.

## Building and Running

Apache Ant is needed to run the ant files to build the project along with Java 1.6:
In the folder with the ant file "build.xml", run:

```
ant build (Compiles and builds the class files)
ant Driver -Dspec="path to spec file" -Dinput="path to input file"
```

## Implementation

Our implementation language of choice was Java. We split up the work into two main parts as described above. Let us first discuss the objects we made in the project briefly:

1. `State` - A state in a NFA. Knows if it is an accept state or not, and has a list of `Transitions`. In addition it can also be set to be a "true" accept state – one that also has an identifier (e.g. $PRINT, $INT, etc) tied to also being an accept state.

2. `Transition` - A transition in a DFA/NFA. Knows the character it represents and the destination `State`

3. `NFA` - Has two `States` - one starting and one ending.

4. `DFAState` - A state in a DFA. Encapsulates one or more `States` (because of the conversion from a NFA to a DFA using epsilon closure techniques).

5. `DFA` - Represents a DFA. Has a HashMap<DFAState, HashMap<Character, DFAState>>, essentially the DFA table

With these objects established, we can move into our discussion of the recursive descent parser.

**Recursive Descent Parser**

For our recursive descent parser (RDP) we have a class that validates a specification regex (as provided by a specification file). We were given the regex grammmar in **Appendix I** for which we also were given the left-recursiveless, prioritized versions for the grammars. For each part of the grammar we have for it its own function and we call other functions or peek/match tokens as per that particular grammar

rule. In essence we have hard-coded the grammar.

If we fail on an input line, we throw an error and stop the program. Otherwise, function that represents a production rule returns a `NFA`. Therefore, a successful parsing of any line in the specification file returns a well produced `NFA` back to the calling function. (Assumptions for the specification will be discussed in a later section).

**DFA Table and Table Walker**

At the end of reading the specification file, we now have a plethora of `NFA` objects; some of which are character classes, some are token specifications. We then union all the token `NFA` objects and call a static function `DFA.convertToDFA(NFA)` which will, as the function header implies, perform the NFA to DFA algorithm including epsilon closure. Reminder that our `DFA` object contains a HashMap of a HashMap, so we have already made our DFA table inside the object.

Our table walker, then, is a function of `DFA` that can tokenize input strings. Because we assume (elaboration in the next section) that 1. token classes are disjoint and 2. we ignore / skip over all white space, our DFA should always be correct in our tokenization. The table walk algorithm is as such:

```
def tableWalk(DFA dfa, String input, int start, int end):
    while(input[start:end] not accept by table) // look for first accept
        end ← end + 1
        if end == |input|:
            return bad-input
    int min-end ← end
    end ← end + 1
    while(input[start:end] accept by table) // keep advancing string until we fail
        min-end ← end
        end ← end + 1
    PRINT input[start:min-end]
    PRINT table.token at input[start:min-end]
    tableWalk(dfa, input, min-end+1, min-end+2)
```

The pseudocode above does not handle all index out of bounds errors but it does show the spirit of the table walk algorithm. In addition the algorithm does not have to be recursive but can be thought of in such a way.

**Driver Program**

This is the program that puts it all together. It reads in a specification file by calling the recursive descent parser, creates the DFA table, and will tokenize an input file. Nothing special, just a program that wraps everything up in one clean method.

## Assumptions

Even with the proper implementation of a recursive descent parser and table walker, like for all programs, there has to be a set of assumptions made.

1. The specification file for classes and tokens are in this format: All character classes first followed by an empty line, then all tokens.
2. Every line in the specification file has this format: [class/token name] SPACE [regex]
3. Every class or token name begins with a dollar sign $.

4. All white space in the input file (what we tokenize) is ignored, and EOL or `\n` ends a line.

5. Corollary to the above: There will not be a token that character class that uses white space "\ " even though it is technically allowed. Or alternate option is to NOT ignore white space in tokenization, and then allow usage of white space as a token or inside a class.

6. All tokens are disjoint in acceptance, ex:
   Allowed:
   ```
   $INT $DIGIT+
   $FLOAT $DIGIT+\.$DIGIT+
   ```
   Not allowed:
   ```
   $INT $DIGIT+
   $SINGLEINT $DIGIT
   ```
   It is obvious to see that what is listed as Allowed is disjoint. For Not allowed, we notice that when parsing a line such as `int a = 1`, and ignoring white space we get `inta=10` we get the ambiguity between `1` as $INT or $SINGLEINT.

7. There is at least one character class and one token.

## Problems

We did not have any significant problems nor did we not implement anything that was asked of us.

## Test Cases

Our test cases are sectioned off into folders (`test_cases`), here is a brief overview of the different test situations: what we were testing and what happened. **Even with the test cases that error out, we always assume our assumptions are met, unless otherwise specified.**

1. test01
   - Testing what: Just the sample files
   - What happens: Should pass everything and properly match the output
2. testIntFloat
   - Testing what: $INT and $FLOAT should be properly distinguished even though both start with DIGIT+
   - What happens: 1 and 1.0 are labeled as $INT and $FLOAT respectively
3. testPassEverything
   - Testing what: We will have a character class for all printable ASCII characters, and a token that is just one printable ASCII character. Input is just one line.
   - What happens: Every single character in the line is individually tokenized; there are as many as the length of the input line.
4. testPassOnlyA
   - Testing what: We have a token that represents just one character, `A`.

- What happens: Our input will `A`'s then something not an `A`. The output will tokenize correctly until that point, then error out.

5. testBadSpecRange
   - Testing what: A bad input range in the RDP.
   - What happens: Error out.
6. testJunkSpec
   - Testing what: Junk string spec file in the RDP.
   - What happens: Error out. (No special error b/c this violates an assumption).
7. testUnknownClass
   - Testing what: A regex references a character class that was never defined.
   - What happens: Error out.

## Conclusion

We have implemented a scanner generator – we read in lexical specifications and create NFAs using a recursive descent parser and a known grammar (with some assumptions made). We then create a DFA table from our NFAs and using our table walker, we can now tokenize input.