Chad Stewart
Kefu Zhou
Eric Fruchter
Patrick Lucas

Due: 12/09/2012
Dr. Pande

CS 3240 Languages and Computation
Scanner Generator/Language Interpreter Project

Table of Contents:

Project: Scanner Generator, Phase I

This project involved parsing a specification file for a regular language to generate a DFA which can be used to tokenize an input stream. Our design implements a pipeline of components. The entry point, ScannerGenerator, first reads in a specification using SpecReader. This class interprets the file into a list of TokenType objects. For each of these an NFA segment is produced, then the segments combined into a full NFA. The NFA is converted to a DFA, which is then minimized. Finally, the DFA and the input stream are passed to a Tokenizer which, when iterated, returns a stream of Token objects containing the token type and value.

Implementation:

The entry point of the project takes in a spec (the format provided below under assumptions), and an input file. It then calls the spec reader which uses a Regex Expansion utility (regex format base on that of Java, also explained below in assumption) to create a list of token-types and combines and parses these to create an NFA. This NFA is converted into a DFA which is given to the tokenizer. The tokenizer uses the input and starts making tokens by walking over the DFA.

Assumptions:

Our implementation makes assumptions about the specification file format based on the assignment document and sample specification. The format must be two sections, separated by a blank line. The top section contains only character class definitions, and character classes can only reference classes defined before them. The bottom section contains only token type definitions and can only reference character classes from the top section.

Character class and token type names must all be of the format specified by: /\$[A-Z-]+/.

All regular expressions in the spec are parsed by the RegexExpander class. This class only accepts regexes similar to those used in Java. The expander accepts (and is not limited to) the following: [a], [ab], [a-ch-l], [a-z0-9A-Z] – with any characters as long as the second character (char2) in the "char1-char2" sequences have ASCII value(char1) is less than ASCII value(char2). Another assumption with the reg-exs are that every negation regex will be formatted as such: [^<char-sequence>]IN[<char-sequence>]. The regex expander accepts all cases of +,\,(,),[,] normally possible with Java. Cases using {#} were not accounted for, nor were cases using ?'s.

Another set of assumptions relates to the format of input file. The input file should consist of two sections separated by a new line, where the first section is definitions, and the second section is more identifier/ equation-type definitions (see sample inputs). Names for Token-types in the input were read as $<NAME IN ALL CAPS> only accepting all uppercase letters and hyphens.

Problems & Uncompleted Modules:

To our team's knowledge there are no uncompleted parts of the assignment. In addition we were able to complete the bonus.

Conclusion:

Over the course of the project there were several problems that we faced in designing the project. We found the description somewhat vague and confusing at first, but after some discussion we decided on a course of action. In addition, having only a minimal sample specification left us to make decisions about interpreting the input that may not be entirely accurate: for example, the assumption regarding the spec file format as described in the "Assumptions" section above.

A large debugging problem arose when we were combining the parts of the project into one working program: Our program was non-deterministic! After a

few hours of debugging, we narrowed the scope of the bug's origin to the NFA to DFA conversion component and were able to solve it.

Extra Credit NFA Walker:

The DFA walker used in the project is actually this NFA walker, so the token output portion of the extra credit was omitted from this section. Instead, you may see visual representations of NFA's via the extracredit.sh file provided. You may open up a file with nfa specification format as explained below:

You can specify an NFA state by using this format:

nameOfState (*, if state is final) | *transitionString* |

stateToTransitionTo ..

You may use ||*stateToTransitionTo* to create an epsilon transition.

The first state specified is the start state.

Sample:

```
# nfa for (a|b)+bg
s ||1 ||2
1 |a|3
2 |b|3
3 ||s |b|4
4 |g| f
f*
```

You can specify strings to test using $, as seen below:

```
$abg
$bbg
$a
$ababag
```

When you run this file in the NFA walker, you will receive a state table with the transitions specified in an easy-to-read format. Then, you will receive diagnostics for each test string, such as *validity in the NFA* and *number of transitions taken*. The same tests will be run after a comparison to DFA, so you can compare the differences. The NFA walker is designed so that it will refuse to visit any path it has already visited. It does this by representing a "step" as a tuple of *string state* and *NFA Node*, allowing it to keep track of the visited steps via hashing. When an epsilon transition is reached, the resulting next step is the next state with the string unmodified. You can find a sample nfa walker specification in the *doc/* folder.

Project: Language Interpreter, Phase II

This phase of the project involved developing an interpreter for a spall specified language "MiniRE". Scripts in this language find and replace strings in text files, save them into lists, and performs basic operations, and can print to the output.

Implementation:

The enter point for this part of the project is a file with a MiniRE script that is scanned, parsed, and executed by the interpreter. The script is only executed if it is accepted by the scanner and the parser. Upon error, the program exists and outputs the appropriate error message. Without error messages, the script implements the functionality specified.

Assumptions:

We used java's regex and file manipulation utilities to handle the actual file find/replace actions because we felt that specific portion of the project was beyond the scope of the class. The assumption we made in phase I of the project carried over to this portion.

Problems & Uncompleted Modules:

All modules of this project were completed to our knowledge. There were a few technical hiccups caused by confusion with the assignment documents, but overall this phase of the project was a little smoother. We also completed the bonus portion of this phase.

Conclusion:

This project was a lot of work. Each phase required a twelve-hour or more hackathon to get everything together and in working order.

Extra Credit LL(1) Parser:

This module is located in the package This module is locate in the package project.phase2.ll1parsergenerator. The primary class to interact with other modules is the class ParserGenerator. Generally, this class is

used to generate a LL(1) parser with an array of rules usng the static method generateParser(Rule[] rules): LL1Parser. The array of rules is generated from RulesParser which parses a specification file and converts it to a rules array, which represents the grammar. With the LL1Parser that is ultimately created, an AST can be generated with its parse method. The main challenge here is writing ParserGenerator. Calling the static method generateParser(Rule[] rules): LL1Parser first stores all the rules to ParserGenerator. Next, the first and follow sets are initialized and populated using algorithms which can be found in Louden's book *Compiler Construction: Principles and Practice*. Primarily, ParserGenerator is used to generate the LL1Parser with the static method generateParser. It also includes methods to get the first and follow sets for transparency and extensions.

Instructions, Code, and Test Cases

All code is in the src/ directory, with the top-level package called 'project'.

To build, either run 'make' in the top-level project directory, or run 'src/build.sh'.

Then, to run an arbitrary script, run 'run_phase2.sh <path-to-script>'. Be aware that the interpreter will use the current working directory for resolving paths to files referenced by the script. For example, if the script 'example/script.txt' references 'input.txt' which itself resides in the 'example/' directory as well, the user should first 'cd example/' and then run '../run_phase2.sh script.txt'.

A variety of tests exist in the test/ directory. To run one individually, execute its 'run.sh' script. For example, to run the test named 'test1', execute 'test/test1/run.sh'. For convenience, the script 'phase2_tests.sh' is provided which runs all tests in the test/ directory sequentially.

Tests Include:

- all_test: Tests all of the functionality of MiniRE together
  - This test 'descrambles' a ASCII picture of Marilyn Monroe using find, replace, and recursive replace. The output files (1,2,3) match the steps of decoding. "output3.txt" is the completed ASCII picture.
- find_test: Tests the find feature
  - This test matches "asdf" randomly placed in the input file. It exists in "input.txt" 12 times. The script prints out it's count (which is 12.)
- maxfreqstring_test: Tests the functionality of maxfreqstring
  - This test computes the number of 'foo's, the number of 'bar's, then runs a union find on the 'foo's and the 'bar's. It then checks to see which has the highest occurrence assigned by maxfreqstring. The foo/bar count is printed beforehand as a reference.
- find_replace: Tests the ability to find and then replace
  - This test checks to make sure finds and replaces can happen before/after each other.
- print_test: Tests the print feature
  - This test simply finds some things and prints it out.
- recurse_test: Tests the recursive replace feature
  - This test replaces things in "input.txt" with simple versions of itself; that is it replaces "11" with "1" for instance. To test the recursive nature of the function, there are 4 "output#.txt" where different lines in the "input.txt" files were replaced.

Team Contributions

Team Chad:

RegexExpander, Documentation, Test Cases, Made snacks

Team Eric:

NFA Walker and NFA Utils, converter. File IO utilities and find/replace

Team Kefu:

Tokenizer, LL(1) Parser Generator, AST data structure

Team Patrick:

Spec reader, NFA builder, MiniRE Parser, MiniRE Interpreter