CS 3240 : Languages and Computation
Phase II : Course Project (Including Bonus Part)

Due date: December 9th 2012, 11:59 pm (No extensions!)

Guidelines:

1. This part of the project will use the scanner built in phase I of the project. Thus, please fix any scanner problems you may have and get it to fully work before you use it in this phase II.
2. Each member must document her/his role in the project and the parts that she/he worked on.
3. Each group must submit a report and the source code. If multiple source files, they must be packed into a single file (e.g. tarred along with the makefile).
4. You have to set up a 20 min appointment with the TA for the week of December 12-16 to do a demonstration. All members of the team must be present and explain their contributions.
5. You can continue to program in the same language that you used in phase I. Do **not** use automatic tools (like lex and yacc) -- such solutions will get ZERO points.
6. Code should be properly documented with meaningful variable and function names. Short elegant code is preferred.
7. You will find the course slides on DFA/NFA/scanner/recursive descent parser and LL(1) parsing useful along with the relevant chapters from the Louden book.
8. Provide instructions about how to compile and execute your files along with test inputs.
9. Any re-use of code (from internet or any other source) for any part of the project is prohibited and will constitute act of plagiarism as per Georgia Tech honor code.
10. **If you run into road-blocks or have any questions, seek help from the TA (use the Forums and e-mail us – we check those round the clock and will respond expediently).**

Goal for main and bonus project:

Develop **an interpreter** for a small language that we call **MiniRE** (as in "mini reg-ex"). You can think of it as a mini Awk or mini Perl. Scripts in MiniRE search and replace strings in **text** files, can save them in string lists, can perform some operations on the string-list and also print to the output. Strings are found by matching regular expressions with the text in the files. Matched strings could be saved into lists; strings inside lists can be operated and arranged with some additional operations that are supported on the lists. Key statistics can be generated on the strings in the string-lists. Scripts written using Mini-RE could be very helpful for document analysis and processing. The **bonus part** of this project involves developing a LL(1) parser generator which can take *any* LL(1) grammar using the specified syntax and convert it into a LL(1) parsing table which can be used by a LL(1) driver to parse the script.

The input to the interpreter is a file with a MiniRE script that is scanned, parsed, and executed by the interpreter. The script is executed only if the scanning and parsing accept the script. If the scanner or parser finds an error, it stops and reports the error type and location in the script file. The error type for the scanner is the start location of the unrecognized token and the unrecognized token itself, and for the parser, it is the token (including location) for which no grammar rule was found. When executed without error, MiniRE script implements the functionality specified. Following is the specification for MiniRE in terms of its lexical rules, grammars and semantics supported by the different statements. In this phase you will use the scanner developed for **REGEX in phase I of the project.**

Lexical classes and Scanning

ID :: variable identifier that starts with any letter and is optionally followed by a sequence of up to 9 more characters consisting of any combination of letters, digits, and underscores. Note: letters can be upper or lower case, and identifiers are case-sensitive (e.g. ‘var’ is not equal to ‘Var’).

REGEX Specification for **REGEX** will be the same as phase I.

ASCII-STR :: any string of ASCII printable characters, including empty string, excluding \n, surrounded by "’s. A " in the string must be escaped (preceded) by \. No other characters need escaping.

Mini-Re keywords: The remaining tokens are those corresponding to the language description in the rest of this document (e.g**, begin**, **end**, **=**, **replace**, **(**, **)**, **union, inters, print**).

Parser: Syntax and Semantics

This is the syntax of MiniRE specification file. MiniRE parser can be written as recursive descent (no bonus) or those who are doing bonus will write a generalized LL(1) parser generator based on this format (you can assume that the grammar will be specified to the parser generator using the following format) and then automatically generate a specific Mini-RE LL(1) parser based on the specifications below:

Grammar specification format and the specification for Mini-RE:

%% Tokens /* This section specifies a list of all the tokens or terminals used in the grammar separated by spacing*/
begin end ID = replace with in ; recursivereplace >! print ( ) , # find diff union inters maxfreqstring
%% Start /* This section specifies the start symbol of the grammar */

<MiniRE-program>

%% Rules /* This section specifies the rules – non terminals are enclosed in < > - BNF notation is used */

<MiniRE-program> begin <statement-list> end
<statement-list> <statement><statement-list-tail>
<statement-list-tail> <statement><statement-list-tail> |
<statement> ID = <exp> ;
<statement> ID = # <exp> ;
<statement> ID = maxfreqstring (ID);
<statement> replace REGEX with ASCII-STR in <file-names> ;
<statement> recursivereplace REGEX with ASCII-STR in <file-names> ;
<file-names> <source-file> >! <destination-file>
<source-file> ASCII-STR
<destination-file> ASCII-STR
<statement> print (<exp-list>) ;
<exp-list> <exp> <exp-list-tail>

<exp-list-tail> , <exp> <exp-list-tail>
<exp> ID | (<exp>)
<exp> <term> <exp-tail>
<exp-tail> <bin-op> <term> <exp-tail>
<exp-tail>
<term > find REGEX in <file-name>
<file-name> ASCII-STR
<bin-op> diff | union | inters

Those not doing the bonus part, please write the Recursive Descent method (see class slides) to implement the parser for MiniRE. **Those doing the bonus part will first implement the parser generator for LL(1) and then use it to generate the LL(1) parser for the above grammar.**

If a syntax error is found, including checks of correctness for REGEXs, the parser indicates an error and stops (the script is not executed).

Variables in MiniRE are of two types: signed integers and *string-match lists*. There are no explicit declaration statements (i.e., declarations are implicit) for ID. ID can correspond to a string-match list. A string match list is a (possibly empty) list where each element is a matched found by a *find* operation (see below). The string list consists of strings; each string is represented by the tuple : <file-name, line, start-index, end-index> .

A MiniRE script contains no functions and is a sequence of the following statements and blocks:

- Assignment statement. ID = <exp> ;
    - Assigns an expression (see below) to a variable and ends in a semi-colon. Examples, s = (find [a-z]+ in "myfile.txt");. The variable on the left of = is implicitly declared the first time it is assigned to. If an existing variable is assigned an expression of a different type, it switches to the type of the newly assigned expression. In the above example, s contains a list of all strings that match the given regular expression [a-z]+ in the file myfile.txt
    - The right hand side could be an expression as well such as : (find [a-z]+ in “myfile.txt”) union (find [0-9]+ in “myfile.txt”) This expression finds a union of two string lists returned by the respective find operations. In addition to union, other operators are also supported. **diff** operator finds a difference by subtracting the second string list from the first one, similarly **inters** operator allows one to find intersection of two lists. All the operators are left associative.
- Assignment statement : ID = # <exp> ;
    - Defines an integer variable ID and assigns to it the length of the string list in <exp>
- Assignment statement ID1 = maxfreqstring (ID2);
    - Defines a string list variable ID1 and assigns to it the string in ID2 that occurs with the highest frequency.
- Replace statement: “replace REGEX with STR in filename1 >! filename2;” where STR replaces every substring in filename1 that matches the REGEX, filename1 is source file and filename2 is the destination file. After applying the replacement the modified copy of filename1 is saved in filename2.

If filename2 exists, overwrite it. A parser error occurs if filename1 = filename2. A runtime error occurs if filename1 doesn't exist or filename2 can't be written to.

- Recursive replace statement: "recursivereplace REGEX with STR in filename1 >! filename2;" Depending on the string that you replace the regex with, it may induce another instance of the string to be replaced. By recursively applying replace we can remove all of these, unless the replacement is cyclic (e.g. replace abc with abc). Thus, the only difference with respect to replace is the repeated application of recursive replace. Additionally a parser error should occur if REGEX and STR are the same.
- Print statement, which prints a list of expressions to the output. If an expression is an integer, it prints the integer and a newline. If it's a string-match list, prints the elements in the list in order in some readable format (include the matched string, the filename, and the index), and a newline.

An expression in MiniRE can be:

- A find expression, whose format is "find REGEX in filename" where filename is the name of a text file surrounded by "'s
. A variable, of type integer or string-match list. Using a variable not (yet) assigned to is an error.
- #v which returns the length (as an integer) of string-match list variable v, ie. the number of strings contained in the string list.
- Set operations applied to string-match lists that return modified lists: union (returns union of the two lists), intersection (returns intersection of the two lists), and difference (first list minus second). Represented by literal tokens *union*, *inters*, and -, respectively. Associativity is by parentheses and left to right.

Steps

An interpreter of a language consists of a scanner, parser and evaluator. It will first convert the MiniRE script into an Abstract Syntax Tree and then will evaluate it generating output.

Our first goal will be to build the scanner and parser of the interpreter which will scan and parse the input Mini-RE scripting program given to it and once found syntactically correct, convert it into an abstract syntax tree (AST). This can be accomplished through the following steps:

Step I: Expand the scanner to generate additional tokens needed by Mini-RE language.

Step II: Write a recursive descent parser (non-bonus) or LL(1) parser (generated via parser generator) (for those doing bonus) which demands and matches tokens one by one generated by the scanner above and builds the MiniRE scripting program into the ASTs. You can use the format of AST as given on page 138 of Louden book.

Step III : The next goal will be to build the evaluator that will walk the AST and generate results of the expression and will propagate those. For example, expression trees will get evaluated on the right hand side and the result will be assigned to left hand side ID in an assignment statement ; when this ID is used further, the value stored in ID will be the one that will be used. When a print statement is encountered, the values will be dumped to the screen by following the program flow.

Main and Bonus parts

Main part: Mini-RE interpreter built using recursive descent parser.

Bonus part: LL(1) parser generator, Mini-RE LL(1) parser generated using the same, Mini-RE interpreter built using the generated LL(1) parser.

Grading criteria

Main part: **100** points.

Recursive descent parser and Mini-RE interpretor based on it.

Bonus part: **25** extra points : LL(1) parser generator that can generate a parser from a general LL(1) specification and LL(1) parser for Mini-RE generated using the parser generator. Mini-RE interpreter based on this parser.

For each part and subpart, the team must bring a collection of examples (scripts + input text files for find/replace statements). You must include examples that cause different scanner/parser/runtime errors. Please try to write simple and useful MiniRE programs and use them to test your implementation. A sample program will also be given for testing by the TA>.

The TA will also test your project with his own MiniRE scripts and text files.