CS 3240 : Languages and Computation
Course Mini-Project

Due dates:
Phase I: April 11th 2008, 5 pm
Phase II and Bonus: April 25th 2008, 5 pm

Guidelines:

1. This project has two phases. Phase 1 is due on April 11th by 5pm. Phase 2 is due on April 25th by 5pm including the bonus part.
2. There will be no extensions for either of the phases.
3. You will work in groups of three
4. Each group should submit a report and source code for each phase. If multiple source files, they must be tarred along with the makefile.
5. You can program in C, C++ or Java. Do not use tools (like lex and yacc) or the standard template library -- 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 useful. In addition, read the Louden book for Tiny language discussion.
8. The first phase of the project is worth 100 points, the second is worth 70 and the last bonus part is worth 80 points.
9. Provide instructions about how to compile and execute your files along with test inputs during each phase of submission.

Objective: Consider a language called **Micro** which is defined by the following grammar

<Micro-program > → begin <statement-list> end
<statement-list> → <statement-list> ; <statement> | <statement>
<statement> → print (<exp-list>)
<statement> → ID := <exp>
<exp-list> → <exp-list>, <exp> | <exp>
<exp> → ID | INTNUM | (<exp>)
<exp> → <exp> <bin-op> <exp>
<bin-op> → + | - | * | **

Micro's Lexical classes are defined as follows:

ID :: An identifier can consist of letters, digits and underscores and must start with a letter or an underscore and can be followed optionally by letters, digits or underscores; an underscore must be followed by at least a letter or a digit and identifier can not be longer than 15 characters.

INTNUM :: Can be a signed or unsigned integer without any leading zeroes.

:= is an assignment operator, , (comma) is a separator, * is a multiplication operator and ** is an exponentiation operator, ; is a statement delimiter and + and – are addition and subtraction operators respectively.

Our objective is to write an interpreter for this language. The interpreter will take any legal program written in Micro as an input and will interpret it statement by statement and generate the output. The language is simple and consists of only straight-line statements – notice the absence of declarative statement in Micro, all variable names are implicitly declared at the point of their definition or use and the language consists of only an assignment and print statement in the language. The assignment statement assigns the value of <exp> generated on the right hand side to its left hand side variable. One can print the values of expressions inside a print statement. An expression can be formed using variables and integers. There are four operators in the language : +, -, * and **. The precedence hierarchy of operators is : ** has the highest precedence, then comes * and then + and -. Parenthesized expressions are evaluated first. The operator ** is right associative whereas all the other operators are left associative. Thus, an expression : 1 + 8 * 9 **7**2 – 4 will be evaluated as ((1 + (8 * (9 ** (7**2))) – 4) (parentheses are showing order of evaluation – innermost parenthesis evaluated first going towards the outer ones).

An interpreter of a language consists of a scanner, parser and evaluator which will first convert the Micro program into an internal representation based on Abstract Syntax Tree and then will evaluate it generating output.

Phase I : In this phase of the project, our goal will be to build the scanner and recursive descent parser of the interpreter which will scan and parse the input Micro 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: First rewrite the grammar converting it into an appropriate form removing left recursion and to enforce correct operator precedence.
Step II : Write a scanner which will generate tokens mentioned in the lexical classes as above. Along with ID token you should send the exact identifier as a string for holding it in AST and along with INTNUM token you should also send its value. Remember you have to write the scanner that reads the input Micro program character by character and thus, you will have to *assemble* the value of INTNUM token from its constituent characters (points will be deducted for solutions which skip/short-circuit such key steps). Apart from generating tokens, the scanner will also skip white spaces, tabs, newlines etc.
Step III : Write a recursive descent parser which demands and matches tokens one by one generated by the scanner above and builds the sentence into an AST. You can use the format of AST as given on page 138 of Louden book. In general read section 3.6 and 3.7 for Tiny language implementation which should be helpful. In case a syntax error is detected, it should be notified to the user by generating the dump of the current partial AST that is built for the current sentence along with the token that has produced the error. The parser should be able to continue discarding the remainder of the current erroneous sentence and parse the remainder of Micro program. Thus, this phase should be able to

catch and report multiple errors in a given Micro program. Such error reporting is required and is a part of design requirements for this phase. For a Micro program that is free of any syntax errors, this phase will output the ASTs built. The format for reporting an AST is (root-label label-of-1st-left-child label-of-2nd-left-child label-of-3rd-left-child …)

Phase II: Phase II of this project will 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. In debugging mode of the evaluator, a user should be able to step through the program statement by statement and at each statement, the evaluator will show the lhs value of an assignment statement. The expressions will be evaluated as per the operator precedence and associativity rules of the language. In a normal evaluation mode, the whole program will get interpreted without interruption producing the output results.

Bonus Part: **Total points : 80, Part I : 30, Part II : 50**

Part I. As one can see a user could write buggy program in Micro in which one could use a variable uninitialized ie., without assigning a value to it before using it. Catch and report such bugs during the parsing phase on the basic Micro language defined above.
Part II. Once the initial design of Micro was done, it was quickly found that some type of control flow was essential. So the designers added the following grammar rules for if…then…else…endif:

```
<statement> → if (<b-exp>) then <statement> else <statement> endif
<b-exp> → <exp> <rel-op> <exp>
<rel-op> → < | > | = | <>
```

In the above grammar <b-exp> is evaluated to TRUE or FALSE and the control flow goes into the “then” or “else” part respectively. The relational operators supported are < (less than), > (greater than), = (equal to) or <> (not equal to). Implement the new features modifying, scanner, parser and evaluator. Note that nested “if … then … else” are allowed in the language.