CS 3240 : Languages and Computation
Course Mini-Project
Total points : 100
Due date: TBA (Final's week)

Guidelines:

- Each team should submit a report and source code for each phase. If multiple source files, they must be tarred or zipped along with the makefile.
- You have to form teams of 3 students. We will set up wiki and forum to help you with that.
- You can program in C, C++ or Java. Do not use automated parser generator (yacc or ANTLR etc), you are asked to develop one here. You are allowed to use Java/C/C++ lexer or you can write your own.
- Code should be properly documented with meaningful variable and function names. Short elegant code is preferred.
- Read the Louden book for Tiny language discussion.
- Provide instructions about how to compile and execute your files along with test inputs as a part of your report. Also, give documentation of the design along with classes used in the implementation of parser generator.
- Any re-use of code (from internet or any other source) for parser generator is prohibited and will constitute act of plagiarism as per Georgia Tech honor code. You are allowed to use any code or prebuilt classes/libraries for lexer though.
- If you run into road-blocks or have any questions about the project, please immediately seek help from the TAs.

Objective:

In this project, we will be writing a LL(1) parser generator. The input will be a grammar specification (see the format of the specification below) for some language X and the output will be a LL(1) parser table for that language. You will also write a LL(1) parser driver (a simple program) which will implement LL(1) parsing algorithm discussed in the class. This driver will use the generated parsing table and perform parsing of a program written in language X by getting the necessary tokens and by applying rules on parsing stack.

We will discuss and demonstrate this project using a simple language. Consider a language called Tiny which is defined by the following grammar

<Tiny-program > : begin <statement-list> end
<statement-list> : <statement-list> <statement> | <statement>
<statement> : print (<exp-list>) ;
<statement> : ID := <exp> ;
<statement> : read (<id-list>) ;
<id-list> : <id-list> , ID | ID
<exp-list> : <exp-list> , <exp> | <exp>

<exp> : ID | INTNUM | (<exp>)
<exp> : <exp> <bin-op> <exp>
<bin-op> : + | - | * | %
Tiny'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 10 characters.

INTNUM :: Can be unsigned integer without any leading zeroes.

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

Keywords : begin end print read

Semantics and precedences :

The language is simple and consists of only straight-line statements – notice the absence of declarative statement in Tiny, 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 only data type present in Tiny is integer and all variables are implicitly declared to be of type integer at the point of their use. 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. One can read integer values into variables through a read statement. An expression can be formed using variables and integers. There are four operators in the language : +, -, * and %. The precedence hierarchy of operators is : parenthesis ( ) has the highest precedence, then comes % (remainder), then comes * and then + and -.

Step 0: First, the Tiny grammar above should be rewritten (refer to the class slides) to reflect the operator precedences – each team submit your revised grammar to the TA and get it checked.

Step I: Write a scanner which will generate tokens mentioned in the lexical classes as above. For this purpose you are allowed to use Java lexer or any other lexer that can generate the tokens given their description. Alternatively you can write it by hand (just for this language Tiny) so that it allows you to test step II below; the tokens are very simple to generate anyway.

Step II: This is the main part of our project. Write a LL(1) parser generator. First you will specify the grammar of the language X with the following format: the first line enumerates all the tokens or terminals of the language, the second line enumerates all the non-terminals, then come the rules of the grammar which show how to expand a non-terminal using the terminals and non-terminals on the right hand side. Note the syntax of

the non-terminals, they are enclosed in <…>, also note the syntax of the rules. In case of Tiny, the grammar file will appear something like this:

%Tokens BEGIN END SEMICOLON COMMA INTNUM PLUS …
%Non-terminals <Tiny-program> <statement-list> <statement> …
%Start <Tiny-program>
%Rules
<Tiny-program > : BEGIN <statement-list> END
<statement-list> : <statement-list> <statement> | <statement>
<statement> : PRINT LEFTPAR <exp-list> RIGHTPAR SEMICOLON
<statement> : ID ASSIGN <exp> SEMICOLON
<statement> : READ LEFTPAR <id-list> RIGHTPAR SEMICOLON
<id-list> : <id-list> COMMA ID | ID
<exp-list> : <exp-list> COMMA <exp> | <exp>
<exp> : ID | INTNUM | LEFTPAR <exp> RIGHTPAR | <exp> <bin-op> <exp> <bin-op> : PLUS | MINUS | MULTIPLY | MODULO

where

LEFTPAR stands for (
RIGHTPAR stands for)
ASSIGN stands for :=
COMMA stands for ,
SEMICOLON stands for ;
PLUS stands for +
MINUS stands for –
MULTIPLY stands for *
MOD stands for %

We will convert the grammar to LL(1) parsing table as follow:

Step II(a) Removal of left recursion and common prefix (factoring): First we will read the grammar specification for language X and automatically convert the grammar internally removing left recursion and common prefix. Note we will only worry about the self left recursion in this project (no indirect recursion).

Step II(b) Building parsing table: Once the grammar is converted into a proper form internally (without left recursion and common prefix), next we will build the LL(1) parsing table. We will compute the necessary First(), Follow() sets for each non-terminal and then build the LL(1) parsing table.

Step II (c) : We will write a simple program that then uses the generated LL(1)

table, gets the necessary tokens from scanner in step I and uses parsing stack to perform parsing. The parsing program should declare successful parse on no error. In case a syntax error is detected, it should be notified to the user by generating the dump of the current partial sentence built on the parsing stack along with the token that has produced the error and halt.

Requirements for Evaluation and Demonstration:

- We will demonstrate the parser generator's working by generating a parse table and by showing the parsing based on it (if the input is fully accepted, you should in the end give a message "Successful parse!"). We will use Tiny programs for doing the same. You can use your own set of Tiny programs for checking, for debugging and for demonstration. Output produced will be either a syntax error with the current production being parsed and token or successful parse – both should be demonstrated using sample runs.
- In addition to (1), your parser generator should be able to accept any LL(1) grammar given to you in the format above and should be able to generate the parse table successfully. This is a very important requirement of the project. Please note that although we will demonstrate it using Tiny, the whole idea behind parser generator is its ability to take <u>any</u> LL(1) grammar and convert it into parsing table. Therefore you should test your parser generator using different LL(1) grammars (you can take some from Sipser and Louden's book and also using internet) and make sure your parser generator works properly.