CS 256 – Programming Languages and Translators
Assignment 1

- This assignment is due by 1 p.m. on Monday, February 10, 2014
- This assignment will be worth 2% of your grade
- You are to work on this assignment by yourself

Basic Instructions

For this assignment, you are to use **bison** (in conjunction with flex) to create a C++ program that will perform syntax analysis for a small programming language called MFPL (described below). If your bison file is named `mfpl.y` and your flex file is named `mfpl.l`, you should be able to compile and execute it on one of the campus Linux machines (such as rc##ucs213.managed.mst.edu where ## is 01-24) using the following commands:

```
bison mfpl.y
flex mfpl.l
g++ mfpl.tab.c -o mfpl_parser
mfpl_parser < inputFileName
```

Your program should process a **single** expression from an input file (although note taht the expression could be an expression list; see the attached grammar). No attempt should be made to recover from errors; **if your program encounters a syntax error, it should simply output a "syntax error" message that includes the line number** in the input file where the error occurred and terminate. Note that your program should NOT evaluate any expressions in the input program as that is not the perpose of either syntax or lexical analysis.

Programming Language Synatax

What follows if the CFG for the MFPL programming language for which you are writing the syntax analyzer. To help you distinguish nonterminals from terminals, nonterminal names begin with **N_** and terminal names begin with **T_**.

N_EXPR → N_CONST | T_IDENT | T_LPAREN N_PARENTHESIZED_EXPR T_RPAREN

N_CONST → T_INTCONST | T_STRCONST | T_T | T_NIL

N_PARENTHESIZED_EXPR → N_ARITHLOGIC_EXPR | N_IF_EXPR | N_LET_EXPR | N_LAMBDA_EXPR | N_PRINT_EXPR | N_INPUT_EXPR | N_EXPR_LIST

N_ARITHLOGIC_EXPR → N_UN_OP N_EXPR | N_BIN_OP N_EXPR N_EXPR

N_IF_EXPR → T_IF N_EXPR N_EXPR N_EXPR

N_LET_EXPR → T_LETSTAR T_LPAREN N_ID_EXPR_LIST T_RPAREN N_EXPR

N_ID_EXPR_LIST → ϵ | N_ID_EXPR_LIST T_LPAREN T_IDENT N_EXPR T_RPAREN

N_LAMBDA_EXPR → T_LAMBDA T_LPAREN N_ID_LIST T_RPAREN N_EXPR

N_ID_LIST → ϵ | N_ID_LIST T_IDENT

N_PRINT_EXPR → T_PRINT N_EXPR

N_INPUT_EXPR → T_INPUT

N_EXPR_LIST → N_EXPR N_EXPR_LIST | N_EXPR

N_BIN_OP → N_ARITH_OP | N_LOG_OP | N_REL_OP

N_ARITH_OP → T_MULT | T_SUB | T_DIV | T_ADD

N_LOG_OP → T_AND | T_OR

N_REL_OP → T_LT | T_GT | T_LE | T_GE | T_EQ | T_NE

N_UN_OP → T_NOT

Sample Input and Output

You still should output the token and lexeme information for every token processed in the input file. In addition, you should output a statement about each production that is being applied throughout the parse, and clearly identify when a syntax error is encountered and the line number on which it occurred.

Given below is some sample input and output. Because we are using an automated script (program) for grading, with the exception of whitespace and capitalization, the output produced by your program MUST be identical to that of the sample output files! Use EXACTLY the same nonterminal names as given in the grammar. However, to shorten things a bit, when you output the productions (but NOT when you output the TOKEN/LEXEME info from HW 1):

1. drop the **N_** prefix for nonterminal names
2. use the actual MFPL keywords and symbols rather than their full terminal names (e.g. let* instead of T_LETSTAR, etc) for any token whose lexem is not unique for its token class
3. drop the **T_** prefix for the terminals IDENT, INTCONST, STRCONST, T, and NIL
4. output **epsilon** when ϵ appears on the RHS of the production.

Also, for better readibility, output at least one space between each terminal and/or nonterminal in each grammar production.

Input example with no syntax errors:

```
 ; no syntax errors in this example
(let* ( (x (input))
          (y (* x 100))
          (z 42)
       )
   (if (and (> x z) (not (or (/= x 100) (= y "hello"))))
      t
      nil
```

```
  )
)
```

Output for example:

```
TOKEN: LPAREN     LEXEME: (
TOKEN: LETSTAR    LEXEME: let*
TOKEN: LPAREN     LEXEME: (
ID_EXPR_LIST -> epsilon
TOKEN: LPAREN     LEXEME: (
TOKEN: IDENT      LEXEME: x
TOKEN: LPAREN     LEXEME: (
TOKEN: INPUT      LEXEME: input
INPUT_EXPR -> input
PARENTHESIZED_EXPR -> INPUT_EXPR
TOKEN: RPAREN     LEXEME: )
EXPR -> ( PARENTHESIZED_EXPR )
TOKEN: RPAREN     LEXEME: )
ID_EXPR_LIST -> ID_EXPR_LIST ( IDENT EXPR )
TOKEN: LPAREN     LEXEME: (
TOKEN: IDENT      LEXEME: y
TOKEN: LPAREN     LEXEME: (
TOKEN: MULT       LEXEME: *
ARITH_OP -> *
BIN_OP -> ARITH_OP
TOKEN: IDENT      LEXEME: x
EXPR -> IDENT
TOKEN: INTCONST   LEXEME: 100
CONST -> INTCONST
EXPR -> CONST
ARITHLOGIC_EXPR -> BIN_OP EXPR EXPR
PARENTHESIZED_EXPR -> ARITHLOGIC_EXPR
TOKEN: RPAREN     LEXEME: )
EXPR -> ( PARENTHESIZED_EXPR )
TOKEN: RPAREN     LEXEME: )
ID_EXPR_LIST -> ID_EXPR_LIST ( IDENT EXPR )
TOKEN: LPAREN     LEXEME: (
TOKEN: IDENT      LEXEME: z
TOKEN: INTCONST   LEXEME: 42
CONST -> INTCONST
EXPR -> CONST
TOKEN: RPAREN     LEXEME: )
ID_EXPR_LIST -> ID_EXPR_LIST ( IDENT EXPR )
TOKEN: RPAREN     LEXEME: )
TOKEN: LPAREN     LEXEME: (
TOKEN: IF         LEXEME: if
TOKEN: LPAREN     LEXEME: (
TOKEN: AND        LEXEME: and
LOG_OP -> and
BIN_OP -> LOG_OP
TOKEN: LPAREN     LEXEME: (
TOKEN: GT         LEXEME: >
REL_OP -> >
BIN_OP -> REL_OP
TOKEN: IDENT      LEXEME: x
EXPR -> IDENT
TOKEN: IDENT      LEXEME: z
EXPR -> IDENT
ARITHLOGIC_EXPR -> BIN_OP EXPR EXPR
PARENTHESIZED_EXPR -> ARITHLOGIC_EXPR
TOKEN: RPAREN     LEXEME: )
EXPR -> ( PARENTHESIZED_EXPR )
TOKEN: LPAREN     LEXEME: (
TOKEN: NOT        LEXEME: not
UN_OP -> not
TOKEN: LPAREN     LEXEME: (
TOKEN: OR         LEXEME: or
LOG_OP -> or
BIN_OP -> LOG_OP
TOKEN: LPAREN     LEXEME: (
TOKEN: NE         LEXEME: /=
REL_OP -> /=
BIN_OP -> REL_OP
TOKEN: IDENT      LEXEME: x
EXPR -> IDENT
TOKEN: INTCONST  LEXEME: 100
CONST -> INTCONST
EXPR -> CONST
ARITHLOGIC_EXPR -> BIN_OP EXPR EXPR
PARENTHESIZED_EXPR -> ARITHLOGIC_EXPR
TOKEN: RPAREN    LEXEME: )
EXPR -> ( PARENTHESIZED_EXPR )
TOKEN: LPAREN     LEXEME: (
TOKEN: EQ         LEXEME: =
REL_OP -> =
BIN_OP -> REL_OP
TOKEN: IDENT      LEXEME: y
EXPR -> IDENT
TOKEN: STRCONST   LEXEME: "hello"
CONST -> STRCONST
EXPR -> CONST
ARITHLOGIC_EXPR -> BIN_OP EXPR EXPR
PARENTHESIZED_EXPR -> ARITHLOGIC_EXPR
TOKEN: RPAREN     LEXEME: )
EXPR -> ( PARENTHESIZED_EXPR )
ARITHLOGIC_EXPR -> BIN_OP EXPR EXPR
PARENTHESIZED_EXPR -> ARITHLOGIC_EXPR
TOKEN: RPAREN     LEXEME: )
EXPR -> ( PARENTHESIZED_EXPR )
ARITHLOGIC_EXPR -> UN_OP EXPR
PARENTHESIZED_EXPR -> ARITHLOGIC_EXPR
TOKEN: RPAREN     LEXEME: )
EXPR -> ( PARENTHESIZED_EXPR )
```

```
ARITHLOGIC_EXPR -> BIN_OP EXPR EXPR
PARENTHESIZED_EXPR -> ARITHLOGIC_EXPR
TOKEN: RPAREN     LEXEME: )
EXPR -> ( PARENTHESIZED_EXPR )
TOKEN: T          LEXEME: t
CONST -> t
EXPR -> CONST
TOKEN: NIL        LEXEME: nil
CONST -> nil
EXPR -> CONST
IF_EXPR -> if EXPR EXPR EXPR
PARENTHESIZED_EXPR -> IF_EXPR
TOKEN: RPAREN     LEXEME: )
EXPR -> ( PARENTHESIZED_EXPR )
LET_EXPR -> let* ( ID_EXPR_LIST ) EXPR
PARENTHESIZED_EXPR -> LET_EXPR
TOKEN: RPAREN     LEXEME: )
EXPR -> ( PARENTHESIZED_EXPR )
START -> EXPR

---- Completed parsing ----
```

Input example with a syntax error:

```
; there is a syntax error in this example
(let* ( (x (input))
          (y (* x))   ; syntax error in expression on this line
          (z 42)
        )
  (if (and (> x z) (not (or (/= x 100) (= y "hello"))))
      t
      nil
  )
)
```

Output for example with syntax error:

```
TOKEN: LPAREN     LEXEME: (
TOKEN: LETSTAR    LEXEME: let*
TOKEN: LPAREN     LEXEME: (
ID_EXPR_LIST -> epsilon
TOKEN: LPAREN     LEXEME: (
TOKEN: IDENT      LEXEME: x
TOKEN: LPAREN     LEXEME: (
TOKEN: INPUT      LEXEME: input
INPUT_EXPR -> input
PARENTHESIZED_EXPR -> INPUT_EXPR
TOKEN: RPAREN     LEXEME: )
EXPR -> ( PARENTHESIZED_EXPR )
TOKEN: RPAREN     LEXEME: )
ID_EXPR_LIST -> ID_EXPR_LIST ( IDENT EXPR )
TOKEN: LPAREN     LEXEME: (
TOKEN: IDENT      LEXEME: y
```

```
TOKEN: LPAREN     LEXEME: (
TOKEN: MULT       LEXEME: *
ARITH_OP -> *
BIN_OP -> ARITH_OP
TOKEN: IDENT      LEXEME: x
EXPR -> IDENT
TOKEN: RPAREN     LEXEME: )
Line 3: syntax error
```

Submission

You will submit this assignment using `cssubmit`. Your ***single*** lexer file ***must*** have a `.l` extension, and your single yacc/bison file must have a `.y` extension. If it does not, you will receive a zero for this assignment. From the directory containing the `.l` file, you will run
`cssubmit 256 a 2`
on the cs213 Linux machines. This will collect your submission and submit it to me. You may submit as many times as you desire; only your last submission will be graded (previous submissions are overwritten). **READ** the output of cssubmit; it may have changed since you last used it.