Computer Science and Engineering
IIIT Kalyani, West Bengal

Compilers Design Laboratory (CS 511)
(Autumn: 2019 - 2020)
3rd Year CSE: 5th Semester

Assignment - 5 *Marks: 10*
Assignment Out: *5th September, 2019* Report on or before: 20th *September, 2019*

Write C program to implement a *table driven predictive parser* for the language of the following grammar (G) with *terminals* { `eof id num r ,` (comma) = (assignment) + (plus) ∗ (times) ((left-parenthesis)) (right-parenthesis) }, *non-terminals* { S CE AE PE ME BE }, where 'S' as the *start symbol*. The production rules are,

S → CE `eof`
CE → CE `,` AE | AE
AE → `id` = AE | PE
PE → PE + ME | ME
ME → ME ∗ BE | BE
BE → (CE) | `id` | `num` | `r`

The grammar G is not $LL(1)$. Transform it to an equivalent $LL(1)$ grammar G_1 by removing *left recursion*, *substitution* and *left factoring*.

For a *table driven predictive parser* we need a *stack* and a *parsing* table. We also need to encode the production rules and store them with rule numbers. Following are my suggestions. You may decide in a different way.

1. We need to encode the production rules and store them. We already have *token code* for terminals. We also assign distinct code to non-terminals. It is necessary to keep in mind that the *rows* of the parsing table are *indexed* by the non-terminals.

   After encoding, a production rule is a sequence of positive integers (code). In our case it is not necessary to store the left-hand non-terminal as they are already available as the index of the row of the parsing table. So the set of rules may be stored a an array structure as follows:

```
typedef struct{
        int len;       // length of right-hand side of the rule
        int rule[LEN]; // code sequence of terminals and
                       // non-terminals
} rule_t;
```

2. The parsing stack will store the terminals and non-terminals (their code). So a simple integer stack is good enough. Note that rules are inserted in reverse order (rightmost symbol first). A stack is implemented as usual, `stack.h` and `stack.c` files. The header file may be as follows.

```
// stack.h
#include <stdio.h>
#ifndef _STACK_H
#define _STACK_H
#define SIZE 1000
#define ERROR 1
#define OK 0
typedef struct {
        int data[SIZE];
        int tos;
} stack ;

void init(stack *) ;      // Initializes the stack
int push(stack * , int) ;
int pop(stack *) ;
int top(stack *, int *) ;
int isEmpty(stack *) ;
int isFull(stack *) ;
#endif
```

Implement the functions in the `stack.c` file.

3. Use the scanner of *assignment-4* as it is with its `lex.h` and `lex.c` files.

4. The row indices of the parsing table are non-terminals. If there are n non-terminals, there are $0, \cdots, n-1$ rows. The non-terminal codes should be such that the actual table index can be obtained with ease.

   Similarly the column indices of the parsing table are *terminals*. There are 10 terminals in this assignment. So the column indices are $0, \cdots, 9$. We already have code for the terminals. These codes should be mapped to the range of column indices. An 1D-array may be used for the mapping of a *terminal code* to the *column index*.

5. The content of the parsing table of size 10×10 are the rule numbers and error indicators.

6. The parser is implemented as `parser.h` and `parser.c` files. **It will not generate any intermediate code**. Its output is simply an **Accept** or a **Reject**.

7. Modify the Makefile.

8. Prepare a `.tar` file with all the files you have with the following command:
   ```
   $ tar cvf <rollNo>.5.tar lex.c lex.h parser.c parser.h main.c
                                stack.c stack.h Makefile
   ```
   Send it to us on or before the due date.

A few input and output are:

```
$ a.out
1
Accepted
```

```
$ a.out
1+2*3
Accepted
$ a.out
a=2+3
Accepted
$ a.out
a=2, b=a+5
Accepted
$ a.out
2 + 3 4
Rejected
$ a.out
3 % 8
Wrong token: %
Rejected
$ a.out
2 = 3
Rejected
$ a.out
2+a=5
Rejected
$ a.out
2+(a=5)
Accepted
```