CSCI-621 Programming Languages
Programming Assignment 2: A Recursive Descent Parser

Design and implement a Recursive Descent Parser (RDP) for the following grammar:

<elist> → <elist> , <e> | <e>
<e> → <n> ^ <e> | <n>
<n> → <n> <d> | <d>
<d> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Where ^ is an exponentiation operator (associate to right). This grammar generates statements of the form 2^2^3, 15, 20^2 for which the parser outputs 256 15 400.

Solution:
In order to design this RDP, the grammar should satisfy two conditions:

(1). No left recursive non-terminals (in a production of the form <x> → <x> <y>, <x> is a left recursive non-terminal). To remove left recursion:
<elist> → <elist> , <e>
<elist> → <e>
will be changed to
<elist> → <e> <elist_tail>
<elist_tail> → , <elist>
<elist_tail> →

(2). No two productions having the same LHS can start with the same symbol on the RHS (This condition is not precisely stated, but it serves the purpose). To solve this problem, you need to factorize the productions:
<e> → <n> ^ <e>
<e> → <n>
will be changed to
<e> → <n> <etail>
<etail> → ^ <e>
<etail> →
Applying the same techniques to the <n> productions, you get:
<n> → <d> <ntail>
<ntail> → <n>
<ntail> →
Now the grammar becomes:
<elist> → <e> <elist_tail>
<elist_tail> → , <elist>
<elist_tail> →
<e> → <n> <etail>
<etail> → ^ <e>
<etail> →
<n> → <d> <ntail>
<ntail> → <n>
<ntail> →
<d> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

(3) The Parser

an RDP is a set of mutually recursive procedures, one for parsing each non-terminal, together with some supporting procedures. In an algorithmic language, the RDP of the above grammar would be:

```
procedure RDPARSER;
    while NOT EOF  do
            SUCCEEDED = TRUE;
            GET_INP_LINE; (/* reads in the next input line*/)
            GET_NEXT_SYMBOL; (/* returns the next input symbol */)
            ELIST;
             if  SUCCEEDED
            then SUCCESS_MESSAGE
            else FAILURE_MESSAGE endif
    endwhile
end RDPARSER;

procedure ELIST;
    E;
    if SUCCEEDED
    then ELIST_TAIL endif
end ELIST;

procedure ELIST_TAIL;
    if EOL
    then    print E_Value
    else    if next_inp_symbol = ‘,’
            then    print E_value;
                    GET_NEXT_SYMBOL;
                    ELIST;
            else SUCCEEDED = FALSE endif
    endif
end ELIST_TAIL;

procedure E;
    N_value = 0;
    N;
    if SUCCEEDED
    then ETAIL endif
end E;
```

```
procedure ETAIL;
    if (NOT ((next_inp_symbol = ‘ ,’) OR EOL))
    then    if next_inp_symbol = ‘^’
            then    GET_NEXT_SYMBOL;
                    E;
                    E_value = N_value ** E_value;
            else SUCCEEDED = FALSE endif
    else E_value = N_value endif
end ETAIL;

procedure N;
    D;
    if SUCCEEDED
    then    N_value = N_value * 10 + D_value;
            NTAIL endif
end N;

procedure NTAIL;
    if (NOT ((next_inp_symbol = ‘^’ | ‘,’) OR EOL))
    then N endif
end NTAIL;

procedure D;
    if next_inp_symbol is a digit
    then    compute D_value;
            GET_NEXT-SYMBOL
    else SUCCEEDED = FALSE endif
end d;
```

This is a simple example which explains the basic idea of RDP. There are many other issues you need to think about them. In particular, how to impose the left associativity on the binary + and – after removing the left recursion, when to evaluate the terms so that * and / associate to right, and how to detect an integer overflow or an uninitialized identifier whose value is needed to evaluate an expression. You also need to find a way with which you can specify the position of syntax error; you don’t have to specify the error type.

Note:

- Keep all variables declared globally as they are, except N_value.
- Declare N_value locally to procedure E.
- Make N_value a pass-by-value parameter to procedure ETAIL.
- Make N_value a pass-by-reference parameter to both procedures N and NTAIL.

The parse table for this parser is shown as follows:

	d	^	**,**	**$**
elist	**elist→ e elist'**			
elist'			**elist' → , elist**	**elist' →** ε
e	e → n e'			
e'		e'→ ^ e	e' → ε	e' → ε
n	n → d n'			
n'	n' →n	n' → ε	n' → ε	n' → ε