# Experiments with Error Recovery features of YACC

Few experiments with lex and yacc, with and without use of error recovery features, are illustrated in this report. Note that purpose of these examples is to illustrate the functioning of the generated parser, particularly its two modes of operation, normal parsing mode and recovery mode. Recall that there are exactly two features in YACC / BISON are a) special token, **error** and b) special action, **yyerrok**.

**Experiment 1 : Yacc Generated parser without using error recovery features and without support for debugging.**

## 1.1 Lex and Yacc scripts for an expression grammar

**Contents of the file "err1.l", lex script**

```
%{
#include <stdio.h>
#include "y.tab.h"
%}

%%
"+" {return ('+');}
"-" {return ('-');}
"a" {return ('a');}
%%
```

**Contents of the file "error_plain.y", this uses the regular features of yacc and none related with errors.**

```
/*C declarations*/
%{ #include<stdio.h>
%}

/* YACC Declarations */

%token a
%left  '-' '+'

/* Grammar follows */
%%

exp:      'a'
        | exp '+' 'a'
        | exp '-' 'a'
        ;
```

```
%%
yyerror(s)
     char* s;
     {printf("%s\n", s);
     }


main ()
{ yyparse (); }
```

**1.2 Generating the Parser**

```
$ lex err1.l
$ yacc -dv error_plain.y
$ cc -o error_plain lex.yy.c y.tab.c -ll
```

The parser "error_plain" is now ready. The LALR(1) parser created can be seen through the file, "y.output", which is given below.
Construct LALR(1) parser manually and compare with the yacc generated parser given below with respect to the reduce and error entries in the two parsers.

```
===================================================================
 Terminals which are not used :  a   '*'    '/'

Grammar
    0 $accept: exp $end
    1 exp: 'a'
    2    | exp '+' 'a'
    3    | exp '-' 'a'

Terminals, with rules where they appear
$end (0) 0
'*' (42)
'+' (43) 2
'-' (45) 3
'/' (47)
'a' (97) 1 2 3
error (256)
a (258)

Nonterminals, with rules where they appear
$accept (9)
    on left: 0
exp (10)
    on left: 1 2 3, on right: 0 2 3

state 0
    0 $accept: . exp $end
    'a'  shift, and go to state 1
    exp  go to state 2

state 1
    1 exp: 'a' .
    $default  reduce using rule 1 (exp)

state 2
    0 $accept: exp . $end
    2 exp: exp . '+' 'a'
    3    | exp . '-' 'a'
    $end  shift, and go to state 3
    '-'   shift, and go to state 4
    '+'   shift, and go to state 5
state 3
    0 $accept: exp $end .
    $default  accept

state 4
    3 exp: exp '-' . 'a'
    'a'  shift, and go to state 6

state 5
    2 exp: exp '+' . 'a'
    'a'  shift, and go to state 7

state 6
    3 exp: exp '-' 'a' .
    $default  reduce using rule 3 (exp)

state 7
    2 exp: exp '+' 'a' .
    $default  reduce using rule 2 (exp)
```
===============================================================

### 1.3 Testing Performance of this parser with different inputs

**a) input 1 :   a + +**

```
syntax error
```

**b) input 2 :  + + a - - a**

```
syntax error
```

**c) input 3 :  a a + a a + + - a -**

```
syntax error
```

The output of the parser is shown below each input, which comprises of the message, "**syntax error**" which is given out through yyerror() as a default message from the parser. Outputs of the parser can be easily explained with the help of the LALR(1) automaton given above. However, which symbol causes the error and what does the parser do after detection and reporting of the first error is not clear from the parser output. To get insight into the parser moves for the inputs given above, we do the next experiment.

**Experiment 2 : Create a parser that has debugging capabilities but no error recovery features.**

**2.1 Yacc scripts**

The lex script, err1.l, is used in this experiment without any change. The yacc script, "err1.y" is given below.

```
/*C declarations*/
%{#include<stdio.h>
%}

/* YACC Declarations */

%token a

%left  '-' '+'

/* Grammar follows */
%%

exp:      'a'
        | exp '+' 'a'
        | exp '-' 'a'
        ;

%%
yyerror(s)
    char* s;
    {printf("%s\n", s);
    }


main ()
{  yydebug = 1;
   yyparse ();
 }
```

## 2.3 Generating the Parser

```
$ lex err1.l
$ yacc -dv  -t err1.y
$ cc -o error_debug lex.yy.c y.tab.c -ll
```

The changes in this experiment, as compared to Experiment 1, are shown in **bold face** in blue color. The parser, named as "error_debug", is now ready. The LALR(1) parser created, contents of the file, y.output, is identical to that produced in experiment 1.
==============================================================

## 2.3 Testing Performance of this parser with different inputs

For each input, the verbose display of parser actions are shown alongwith. The compilation errors reported by the parser is shown in bold face, "**syntax error**". In this version of the parser, the internal actions are clearly seen and also the point where the error is detected and displayed is also explicit. Observe that even for inputs with more than one error, the generated parser reports exactly one error and terminates.

**a) input 1 : a + +**

```
Starting parse
Entering state 0
Reading a token: Next token is token 'a' ()
Shifting token 'a' ()
Entering state 1
Reducing stack by rule 1 (line 15):
   $1 = token 'a' ()
-> $$ = nterm exp ()
Stack now 0
Entering state 2
Reading a token: Next token is token '+' ()
Shifting token '+' ()
Entering state 5
Reading a token: Next token is token '+' ()
syntax error
Error: popping token '+' ()
Stack now 0 2
Error: popping nterm exp ()
Stack now 0
Cleanup: discarding lookahead token '+' ()
Stack now 0
```

**Explanation :** Whie most of the the parser moves are obvious, try to understand the dump given above after the line **syntax error**.

**b) input 2 :  + + a - - a**

```
Starting parse
Entering state 0
Reading a token: Next token is token '+' ()
```
**syntax error**
```
Cleanup: discarding lookahead token '+' ()
Stack now 0
```

**c) input 3 :  a a + a a + + - a -**

```
Starting parse
Entering state 0
Reading a token: Next token is token 'a' ()
Shifting token 'a' ()
Entering state 1
Reducing stack by rule 1 (line 15):
   $1 = token 'a' ()
-> $$ = nterm exp ()
Stack now 0
Entering state 2
Reading a token: Next token is token 'a' ()
```
**syntax error**
```
Error: popping nterm exp ()
Stack now 0
Cleanup: discarding lookahead token 'a' ()
Stack now 0
```

**Experiment 3 : Create a parser that has error production rules in the grammar (one feature of Yacc is used for error recovery)**

**3.1 Scripts**

The lex script remains uncanged  but the yacc script is rewritten as shown below. The change is shown in bold face in blue color.

```
/*C declarations*/
%{#include<stdio.h>
%}

/* YACC Declarations */

%token a
%left  '-' '+'
```

```
/* Grammar follows */
%%

exp:      'a'
        | exp '+' 'a'
        | exp '-' 'a'
        | error
        ;

%%
yyerror(s)
    char* s;
    {printf("%s\n", s);
    }


main ()
{  yydebug = 1;
   yyparse ();
}
```

## 3.2 Generating the Parser

```
$ lex err1.l
$ yacc -dv  -t err_rec.y
$ cc -o error_rec lex.yy.c y.tab.c -ll
```

The parser "error_rec" is now ready. The LALR(1) parser created can be seen through the file, "y.output", which is different from the automaton of earlier two experiments.

```
==================================================================
Terminals which are not used :   a   '*'    '/'
Grammar
   0 $accept: exp $end
   1 exp: 'a'
   2    | exp '+' 'a'
   3    | exp '-' 'a'
   4    | error

Terminals, with rules where they appear
$end (0) 0
'+' (43) 2
'-' (45) 3
'a' (97) 1 2 3
error (256) 4
a (258)
```

```
Nonterminals, with rules where they appear
$accept (9)
    on left: 0
exp (10)
    on left: 1 2 3 4, on right: 0 2 3

state 0
    0 $accept: . exp $end
    error  shift, and go to state 1
    'a'    shift, and go to state 2
    exp  go to state 3

state 1
    4 exp: error .
    $default  reduce using rule 4 (exp)

state 2
    1 exp: 'a' .
    $default  reduce using rule 1 (exp)

state 3
    0 $accept: exp . $end
    2 exp: exp . '+' 'a'
    3    | exp . '-' 'a'
    $end  shift, and go to state 4
    '-'   shift, and go to state 5
    '+'   shift, and go to state 6

state 4
    0 $accept: exp $end .
    $default  accept

state 5
    3 exp: exp '-' . 'a'
    'a'  shift, and go to state 7

state 6
    2 exp: exp '+' . 'a'
    'a'  shift, and go to state 8

state 7
    3 exp: exp '-' 'a' .
    $default  reduce using rule 3 (exp)


state 8
    2 exp: exp '+' 'a' .
    $default  reduce using rule 2 (exp)
```

## 3.3 Performance with different inputs

The behaviour of this parser with all the 3 inputs are given in the following (2 column display). For each input, the verbose display of parser actions are shown alongwith. The compilation errors reported by the parser is shown in bold face, "**syntax error**". Shift action on the special token, **error**, is present in exactly one state, namely state 0. In the event of Action [] table entry denoting an error, the parser will pop the stack repeatedly till it exposes stack 0. The shift of **error** on state 0 to reach state 1 prevents the parser from falling off the stack. Error recovery is therefore enabled in this parser at least in one state. The switch of the parser between parsing mode and recovery mode is shown at relevant places in the dump. The errors detected during the recovery mode are shown in red color; these error do not cause the issue of a error message. The successful shift of a token is shown in blue color, since the parser keeps a count of the number of successful consecutive tokens shifted. The detection of parser in a loop is shown in color in the dumps.

**a) input 1 : a + +**

```
Normal Parsing Mode
Starting parse
Entering state 0
Reading a token: Next token is token 'a' ()
Shifting token 'a' ()
Entering state 2
Reducing stack by rule 1 (line 15):
   $1 = token 'a' ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Reading a token: Next token is token '+' ()
Shifting token '+' ()
Entering state 6
Reading a token: Next token is token '+' ()
Syntax error
Entering Recovery Mode
Error: popping token '+' ()
Stack now 0 3
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token '+' ()
Shifting token '+' ()
Entering state 6
Reading a token: Now at end of input.
Error: popping token '+' ()
Stack now 0 3
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Now at end of input.
Shifting token $end ()
Entering state 4
Stack now 0 3 4
Cleanup: popping token $end ()
Cleanup: popping nterm exp ()
```

**b) input 2 : + + a - - a**

```
Normal Parsing Mode
Starting parse
Entering state 0
Reading a token: Next token is token '+' ()
Syntax error
Entering Recovery Mode
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token '+' ()
Shifting token '+' ()
Entering state 6
Reading a token: Next token is token '+' ()
Error: popping token '+' ()
Stack now 0 3
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token '+' ()
Shifting token '+' ()
Entering state 6
Reading a token: Next token is token 'a' ()
Shifting token 'a' ()
Entering state 8
Reducing stack by rule 2 (line 16):
   $1 = nterm exp ()
   $2 = token '+' ()
   $3 = token 'a' ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Reading a token: Next token is token '-' ()
Shifting token '-' ()
Entering state 5
Normal Parsing Mode
Reading a token: Next token is token '-' ()
Syntax error
Entering Recovery Mode
Error: popping token '-' ()
Stack now 0 3
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token '-' ()
Shifting token '-' ()
Entering state 5
Reading a token: Next token is token 'a' ()
Shifting token 'a' ()
Entering state 7
Reducing stack by rule 3 (line 17):
   $1 = nterm exp ()
   $2 = token '-' ()
   $3 = token 'a' ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Reading a token: Now at end of input.
Shifting token $end ()
Entering state 4
Stack now 0 3 4
Cleanup: popping token $end ()
Cleanup: popping nterm exp ()
```

**c) input 3 :  a a + a a + + - a -**

**Normal Parsing Mode**

```
Starting parse
Entering state 0
Reading a token: Next token is token 'a' ()
Shifting token 'a' ()
Entering state 2
Reducing stack by rule 1 (line 15):
   $1 = token 'a' ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Reading a token: Next token is token 'a' ()
```

**Syntax error**

**Entering Recovery Mode**

```
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token 'a' ()
```

**Loop detected at with state = 3; token = 'a'**

```
Error: discarding token 'a' ()
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Reading a token: Next token is token '+' ()
Shifting token '+' ()
Entering state 6
Reading a token: Next token is token 'a' ()
Shifting token 'a' ()
Entering state 8
Reducing stack by rule 2 (line 16):
   $1 = nterm exp ()
   $2 = token '+' ()
   $3 = token 'a' ()
```

```
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Reading a token: Next token is token 'a' ()
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token 'a' ()
```

**Parser in Loop**

```
Error: discarding token 'a' ()
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Reading a token: Next token is token '+' ()
Shifting token '+' ()
Entering state 6
Reading a token: Next token is token '+' ()
Error: popping token '+' ()
Stack now 0 3
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token '+' ()
Shifting token '+' ()
Entering state 6
Reading a token: Next token is token '-' ()
Error: popping token '+' ()
Stack now 0 3
Error: popping nterm exp ()
Stack now 0
```

```
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token '-' ()
Shifting token '-' ()
Entering state 5
Reading a token: Next token is token 'a' ()
Shifting token 'a' ()
Entering state 7
Reducing stack by rule 3 (line 17):
   $1 = nterm exp ()
   $2 = token '-' ()
   $3 = token 'a' ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Reading a token: Next token is token '-' ()
Shifting token '-' ()
Entering state 5
```
**Normal Parsing Mode**
```
Reading a token: Now at end of input.
```
**Syntax error**

**Enetering Recovering Mode**
```
Error: popping token '-' ()
Stack now 0 3
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Now at end of input.
Shifting token $end ()
Entering state 4
Stack now 0 3 4
Cleanup: popping token $end ()
Cleanup: popping nterm exp ()
```

**Experiment 4 : Parser with both error recovery features, error and yyerrok**

**4.1 Scripts**

The changes in the yacc script are showin in bold face in blue color.

```
/*C declarations*/
%{ #include <stdio.h>
%}

/* YACC Declarations */

%token a

%left  '-' '+'
%left '*' '/'

/* Grammar follows */
%%

exp:      'a' {yyerrok;}
        | exp '+' 'a'
        | exp '-' 'a'
        | error
        ;

%%
yyerror(s)
    char* s;
    {printf("%s\n", s);
    }


main ()
{   yydebug = 1;
   yyparse ();
}
```

**4.2 Generating the Parser**

```
$ lex err1.l
$ yacc -dv  -t err_rec.y
$ cc -o error_rec lex.yy.c y.tab.c -ll
```

The parser "error_rec" is now ready. The LALR(1) parser created can be seen through the file, "y.output", which is different from the automaton of previous experiment.

### 4.3 Performance with different inputs

The annotations in the dump are same as explained in Experiment 3. The only addition is because of the presence of yyerrok with the rule, E -> a, which forces the parser to return to normal parsing mode whenever this rule is used in a reduce action. However as shown for all the inputs parsed, this action was either executed without any effect (when executed in normal parsing mode) or was never executed.

**a) input 1 : a + +**

**Normal Parsing Mode**

```
Starting parse
Entering state 0
Reading a token: Next token is token 'a' ()
Shifting token 'a' ()
Entering state 2
Reducing stack by rule 1 (line 15):
   $1 = token 'a' ()
-> $$ = nterm exp ()
```

**yyerrok executed – without any effect**

```
Stack now 0
Entering state 3
Reading a token: Next token is token '+' ()
Shifting token '+' ()
Entering state 6
Reading a token: Next token is token '+' ()
```

**Syntax error**

**Entering Recovery Mode**

```
Error: popping token '+' ()
Stack now 0 3
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token '+' ()
Shifting token '+' ()
Entering state 6
Reading a token: Now at end of input.
Error: popping token '+' ()
Stack now 0 3
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Now at end of input.
Shifting token $end ()
Entering state 4
Stack now 0 3 4
Cleanup: popping token $end ()
Cleanup: popping nterm exp ()
```

**b) input 2 :  + + a - - a**

```
Normal Parsing Mode
Starting parse
Entering state 0
Reading a token: Next token is token '+' ()
Syntax error
Entering Recovery Mode
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token '+' ()
Shifting token '+' ()
Entering state 6
Reading a token: Next token is token '+' ()
Error: popping token '+' ()
Stack now 0 3
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token '+' ()
Shifting token '+' ()
Entering state 6
Reading a token: Next token is token 'a' ()
Shifting token 'a' ()
Entering state 8
Reducing stack by rule 2 (line 16):
   $1 = nterm exp ()
   $2 = token '+' ()
   $3 = token 'a' ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Reading a token: Next token is token '-' ()
Shifting token '-' ()
Entering state 5
Back to Normal Parsing
Reading a token: Next token is token '-' ()
Syntax error
Entering Recovery Mode
Error: popping token '-' ()
Stack now 0 3
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token '-' ()
Shifting token '-' ()
Entering state 5
Reading a token: Next token is token 'a' ()
Shifting token 'a' ()
Entering state 7
Reducing stack by rule 3 (line 17):
   $1 = nterm exp ()
   $2 = token '-' ()
   $3 = token 'a' ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Reading a token: Now at end of input.
Shifting token $end ()
Entering state 4
Stack now 0 3 4
Cleanup: popping token $end ()
Cleanup: popping nterm exp ()
```

**c) input 3 :  a a + a a + + - a -**

**Normal Parsing Mode**

```
Starting parse
Entering state 0
Reading a token: Next token is token 'a' ()
Shifting token 'a' ()
Entering state 2
Reducing stack by rule 1 (line 15):
   $1 = token 'a' ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Reading a token: Next token is token 'a' ()
```

**Syntax error**

**Entering Recovery Mode**

```
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token 'a' ()
```

**Parser in loop : shiftcount = 0**

```
Error: discarding token 'a' ()
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Reading a token: Next token is token '+' ()
Shifting token '+' ()
Entering state 6
Reading a token: Next token is token 'a' ()
Shifting token 'a' ()
Entering state 8
Reducing stack by rule 2 (line 16):
   $1 = nterm exp ()
   $2 = token '+' ()
   $3 = token 'a' ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Reading a token: Next token is token 'a' ()
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token 'a' ()
```
**Parser in loop : shiftcount = 0**
```
Error: discarding token 'a' ()
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Reading a token: Next token is token '+' ()
Shifting token '+' ()
Entering state 6
Reading a token: Next token is token '+' ()
Error: popping token '+' ()
Stack now 0 3
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token '+' ()
Shifting token '+' ()
Entering state 6
Reading a token: Next token is token '-' ()
Error: popping token '+' ()
Stack now 0 3
Error: popping nterm exp ()
```

```
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Next token is token '-' ()
Shifting token '-' ()
Entering state 5
Reading a token: Next token is token 'a' ()
Shifting token 'a' ()
Entering state 7
Reducing stack by rule 3 (line 17):
   $1 = nterm exp ()
   $2 = token '-' ()
   $3 = token 'a' ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Reading a token: Next token is token '-' ()
Shifting token '-' ()
Entering state 5
```
**Return to Normal Parsing Mode**
```
Reading a token: Now at end of input.
```
**Syntax error**

**Entering Recovery Mode**
```
Error: popping token '-' ()
Stack now 0 3
Error: popping nterm exp ()
Stack now 0
Shifting token error ()
Entering state 1
Reducing stack by rule 4 (line 18):
   $1 = token error ()
-> $$ = nterm exp ()
Stack now 0
Entering state 3
Now at end of input.
Shifting token $end ()
Entering state 4
Stack now 0 3 4
Cleanup: popping token $end ()
Cleanup: popping nterm exp ()
```

**End of Document**