1 Introduction

The purpose of this phase of the compiler is to aid the programmer in identifying all possible inconsistencies in his/her source program. This is the *user interface* part of a compiler, and must be designed carefully. It is desirable that error analysis be integrated with the compilation process and not be treated as an add-on feature. The typical aspects of error analysis are given below.

1. Detection of errors in various phases of the compiler.
2. Reporting of errors in a desirable form.
3. Recovery from errors in order to detect as many errors as possible in the entire input.
4. Repair of errors. This is different from (3) above, in the sense that error is attempted to be corrected, so that the compiled program is executable.

The first three issues are mandatory and can be reasonably handled by a compiler.

1.1 Classification of Errors

Errors are classified by the phase of the compiler where an error can be potentially detected; thus *lexical error, syntax error, semantic error, logical error* are among the types of errors that are defined in this context. Example of errors in various phases are as given in the following.

- Lexical Error : Errors in token formation, illegal characters, etc.
- Syntax Errors : Errors in structure, missing operators, unbalanced parenthesis, etc.
- Semantic Errors : Errors in agreement, operands whose types are incompatible, undeclared variables, actual arguments of function or procedure call not conforming to the declared formal arguments, etc.

The scope of this module is confined to syntax errors. Semantic errors, their detection and recovery aspects (if any), are dealt in module 5. Most of the lexical errors may also be handled by the parser.

2 Error Detection

The basic question that is addressed here concerns with the detection of syntax errors. For the table driven parsers, such as the top down LL(1) parser and the bottom up parsers SLR, LALR and LR, we have seen that the tables contain explicit entries for error situations. All table driven parsers have the valid prefix property. This implies that the part of the string that has been parsed successfully never needs to be reexamined on encountering an error. Among all the table driven parsers, only the LR(1) parser has immediate error detection property. As a consequence, this parser has the capability of giving better error diagnostics as compared to all the parsers of this family.

LALR parser , as mentioned in the previous module, may proceed to reduce around the error point, it would never shift an erroneous token however. This causes delay in detection of error. Proper reporting of error is also difficult since the left context may have become different than what it was at the actual point of error.

3 Error Reporting

This issue is very important for the end user. Some desirable features are given in the following.

1. Errors should be reported in terms of source program entities.
2. Error messages must be clear and unambiguous.
3. Error messages must highlight the source of error and localize it,as close as possible, to the point of occurrence.
4. Redundant error messages (cascading effect) should be avoided.
5. An error may be introduced because of the error recovery method used. In such a case, the user must be notified of such errors (and distinguished from genuine ones) by an appropriate message.

3.1 Reporting of errors in table driven parsers

Since the error situations are explicitly known, reporting of errors is not difficult. The user may also be provided with useful tips about the tokens that are acceptable at the error point. A general method for error reporting would be to associate appropriate error messages for each error point (group similar ones) and invoke a routine to issue out the error message.

We illustrate the method by writing error messages for the following grammar. The corresponding SLR parsing table and the SLR automaton are given as Figures 17 and 21 respectively, in *module 3.*

$$\begin{aligned} &E \rightarrow E + T \mid T \\ &T \rightarrow T * F \mid F \\ &F \rightarrow (E) \mid \texttt{id} \end{aligned}$$

We assume that error messages give indication about the locality of error, such as line number and the erroneous token. The LR parsing table is also assumed to be available.

In the following we develop guidelines for constructing appropriate messages such that they are helpful to the user in correcting the error.

3.2 First Approach

In this approach, the error entries of each state are examined one by one. The constructed message essentially indicates the valid tokens of the state. The basic steps are:
For each state i of the parsing table do the following.

1. For each error entry of state i, create an routine e_i. This routine is invoked whenever an error is encountered in state i.

2. The set of valid tokens, say T, for state i is constructed. T = { t | Action[i,t] $\neq$ `error` }.

3. Routine e_i, after reporting locality of the error, issues a message which states that, *symbols expected are* "T".

Clearly if two states i and j have the same error entries, the same error routine may be used for both.
EXAMPLE : Consider state 2 of the parsing table of Figure 17 of module 3, reproduced in the figure below. The set T = { + , $*$,) , \$ }. The New contents of this state also shown in the figure. The routine e_2 would report error in the following format.

Error at line number "..." near symbol "*next_token*";
Expected symbols are +, $*$,) and end_of_input

In view of the fact that states 3,5,9,10 and 11 have the same error positions as that of state 2, routine e_2 may be used for all these states.

Action

State	id	+	*	(	)	$
2		r2	s7		r2	r2

Original entry of State 2

Action

State	id	+	*	(	)	$
2	e2	r2	s7	e2	r2	r2

Revised entry of State 2

3.3 Second Approach

This approach is essentially the same as the first approach except for the use of set T. It is possible that T for a state may contain several tokens. T is therefore a gross information and instead of helping may confuse the user in correcting the error. The purpose is to explore the possibility of giving more precise and accurate information (subset of T) for every distinct error entry of a state.
We examine state 2 of Figure 17 of module 3 again, given in the figure. The following observations may be made.

The erroneous tokens for this state are `id` and (. The valid token set T = { + , $*$,) , \$ }.

Consider token `id`. We examine which tokens from T are less likely candidates in this state. A token t is *more likely* if the sequence t `id` gets parsed without an error.

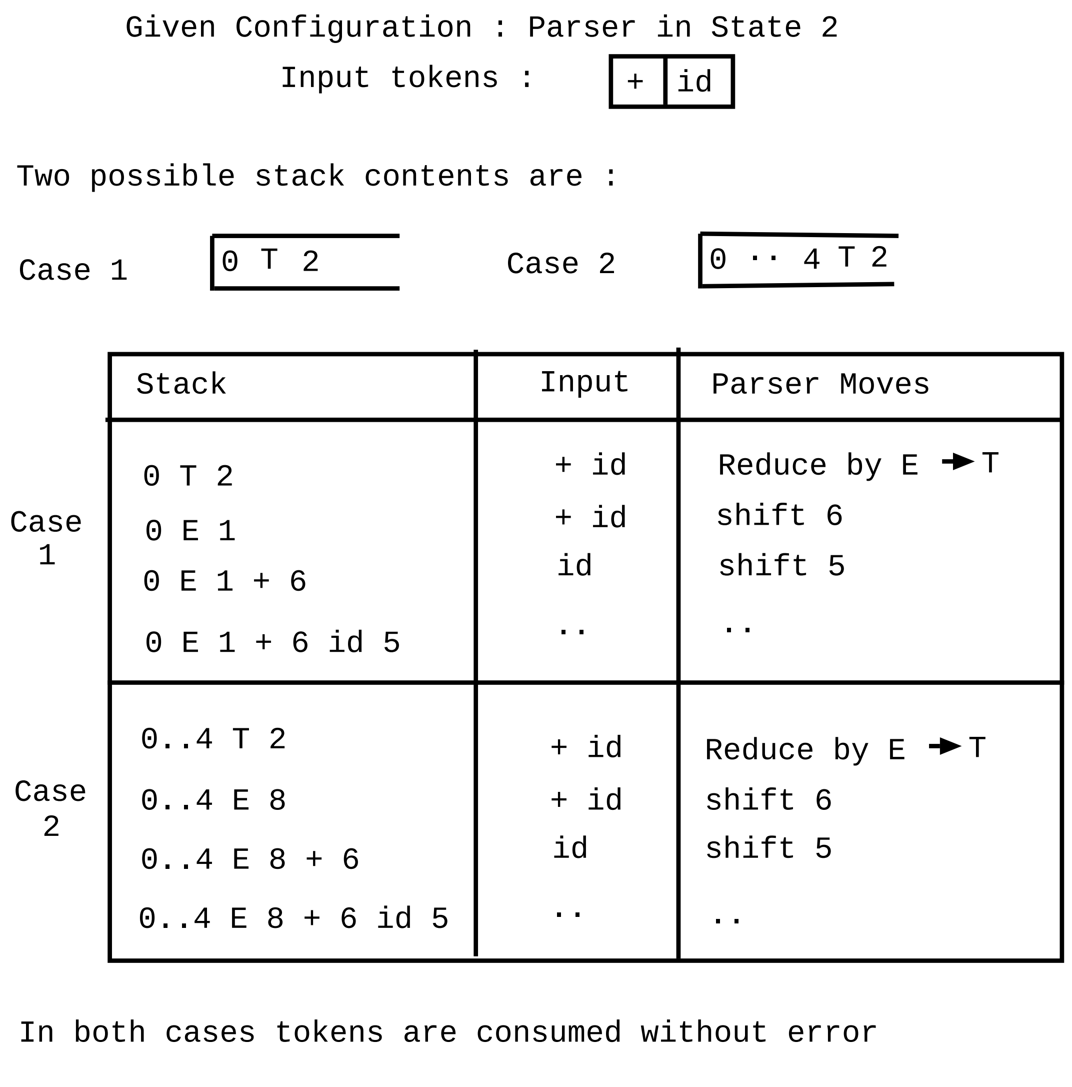
Consider + `id`. The sequence of parser moves shown in the figure below justifies the inclusion of symbol + for reporting. It may be seen that $*$ is also a *more likely* candidate by a similar reasoning.

Consider) `id`. The sequence of parser moves shown in Figure **??** explains why it is less likely as compared to the previous ones. Similar result holds true for \$ `id`.

A similar exercise for the other error token (may be done. It can be seen that *more* *likely* tokens are again $+$ and $*$. The new entry for state 2 remains the same as in Figure **??**. However, the message issued by routine e_2 now changes to :

Error at line number "..." near symbol "*next_token*";
Missing operator;

Both the approaches described above are cumbersome and in the worst case may require an effort proportional to the number of error entries in the entire table.



Given Configuration : Parser in State 2

Input tokens : $\boxed{+}\boxed{\text{id}}$

Two possible stack contents are :

Case 1 $\boxed{0\ T\ 2}$ Case 2 $\boxed{0 \cdot\cdot\ 4\ T\ 2}$

	Stack	Input	Parser Moves
Case 1	0 T 2	+ id	Reduce by E $\rightarrow$ T
	0 E 1	+ id	shift 6
	0 E 1 + 6	id	shift 5
	0 E 1 + 6 id 5	..	..
Case 2	0..4 T 2	+ id	Reduce by E $\rightarrow$ T
	0..4 E 8	+ id	shift 6
	0..4 E 8 + 6	id	shift 5
	0..4 E 8 + 6 id 5	..	..

In both cases tokens are consumed without error

3.4 Third Approach

The key step of this approach is to reduce the number of error entries as explained below.

Given Configuration : Parser in State 2

Input tokens : $\boxed{)}\boxed{\text{id}}$

Two possible stack contents are :

Case 1 $\boxed{0\ T\ 2}$ **Case 2** $\boxed{0 \cdot\cdot\ 4\ T\ 2}$

	Stack	Input	Parser Moves
Case 1	0 T 2	) id	Reduce by E → T
	0 E 1	) id	error
Case 2	0..4 T 2	) id	Reduce by E → T
	0..4 E 8	) id	shift 11
	0..4 E 8) 11	id	Reduce by F → (E)
	Stack must have 0..t (4 E 8) 11 Possible values of t are : 0,4,6 and 7 There are four subcases to examine		
t=0	0 (4 E 8) 11	id	Reduce by F → (E)
	0 F 3	id	Reduce by T → F
	0 T 2	id	error
t=4	0..4 (4 E 8) 11	id	Reduce by F → (E)
	0..4 F 3	id	Reduce by T → F
	0..4 T 2	id	error
t=6	0..6 (4 E 8) 11	id	Reduce by F → (E)
	0..6 F 3	id	Reduce by T → F
	0..6 T 9	id	error
t=7	0..7 (4 E 8) 11	id	Reduce by F → (E)
	0..7 F 10	id	error

In all cases token id is not consumed.

1. If a state has at one *reduce* entry, say r_j, then all error entries of the state are replaced by r_j.
2. In case a state has several reduce entries, any one may be selected (such as the most frequent one) for replacing the error entries.
3. A state that has only shift (*accept* is also allowed) and error entries remains unchanged.

This is the method used in YACC to store a LALR(1) parsing table. It can be shown that using the transformed parsing table as mentioned above, delays the error detection capability of the parser without affecting its correctness.

The effect of making the changes suggested above to that of Figure 17 of module 3, is shown in Figure **??**. Initially there are 36 error entries and 36 valid entries. After the transformation, six states, namely 2, 3, 5, 9, 10 and 11 are free of errors. A total of 12 error entries have been eliminated.

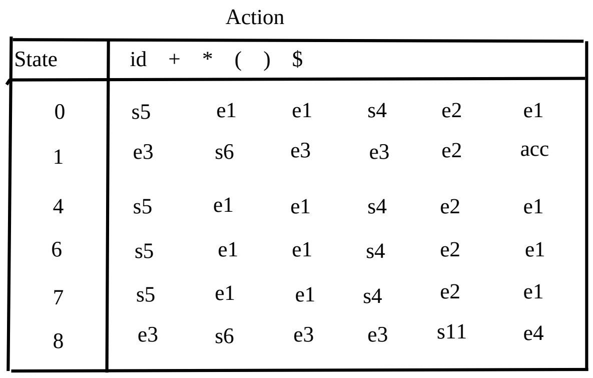
Action

State	id	+	*	(	)	$
0	s5			s4		
1		s6				acc
2	r2	r2	s7	r2	r2	r2
3	r4	r4	r4	r4	r4	r4
4	s5			s4		
5	r6	r6	r6	r6	r6	r6
6	s5			s4		
7	s5			s4		
8		s6			s11	
9	r1	r1	s7	r1	r1	r1
10	r3	r3	r3	r3	r3	r3
11	r5	r5	r5	r5	r5	r5

The error routines for the remaining entries may be written as outlined in the second approach. The resulting parsing table alongwith a description of the error routines are given in Figure **??**.

4 Error Recovery

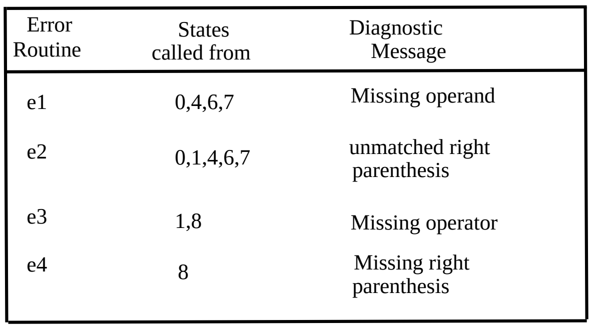
There are several methods suggested in the literature for *Recovery from Syntax Errors* but very few have actually been implemented and used.



Action

State	id	+	*	(	)	$
0	s5	e1	e1	s4	e2	e1
1	e3	s6	e3	e3	e2	acc
4	s5	e1	e1	s4	e2	e1
6	s5	e1	e1	s4	e2	e1
7	s5	e1	e1	s4	e2	e1
8	e3	s6	e3	e3	s11	e4

States which are free of error entries are omitted



Error Routine	States called from	Diagnostic Message
e1	0,4,6,7	Missing operand
e2	0,1,4,6,7	unmatched right parenthesis
e3	1,8	Missing operator
e4	8	Missing right parenthesis

Panic mode : In this method a synchronizing token has to be defined for each important construct of the language. For instance ';' for an assignment statement, `'end'` for list of statements enclosed in a `begin - end` , etc.

The method examines the left context at the point of error and discards tokens from the input till the matching token is encountered.

4.1 Exhaustive error recovery

This method comprises of examining each error possibility, finding the cause, and then deciding on an appropriate recovery. Though the method can be employed in principle for the table driven parsers, the task is complex and tedious. In the following recovery actions for the table in Figure **??** are worked out in the following. Our approach, though based on analysis, is not exhaustive and may be termed as an *ad-hoc* method of recovery. To write the recovery actions, we examine each of the error routines as given below.

Routine e_1 is called when nexttoken is one of $+$, $*$ or \$ and valid symbols are id and (. It can be shown that id is a *more likely* token as compared to (. The recovery decision is therefore to shift a dummy id. *It is an insertion type of recovery.*

Routine e_2 is called when nexttoken is) and valid symbols are id and (. It can be shown that neither id nor (qualify to be *likely* tokens. The recovery decision is therefore to delete ")". *It is a deletion type of recovery.*

Routine e_3 is called when nexttoken is one of id or (and valid symbols for state 1 are + and \$ and), while for state 8 they are + and). Token + is the *more likely* token for both the states 1 and *. The recovery decision is therefore to shift +. *It is an insertion type of recovery.*

Routine e_4 is called when nexttoken is \$ and valid symbols are + and). It can be seen that) is a *more likely* token as compared to +. The recovery decision is therefore to shift a ")". *It is also an insertion type of recovery.*

The recovery actions are shown in the table below. It is assumed that the parser will return to normal parsing after performing the recovery actions as stated in each error routine.

Error Routine	States called from	Diagnostic Message	Recovery Actions
e1	0,4,6,7	Missing operand	Push id and State 5
e2	0,1,4,6,7	Unmatched right parenthesis	Delete token i.e.,)
e3	1,8	Missing operator	Push + and State 6
e4	8	Missing right parenthesis	Push) and State 11

4.2 Systematic methods for error recovery

The basic steps of these methods are : i) to suspend normal parsing on reaching an error configuration, ii) to change the error configuration by modifying either the stack contents or the input buffer, or perhaps both, in order to arrive at a different configuration, and iii) to resume normal parsing from the new configuration.

The issues involved in performing systematic error recovery are as follows.

1. How to characterize the new configuration?
2. There may be several candidate configurations. How to identify the best among them ?

3. Assigning costs for deletion and insertion of a token may be used to locally order candidate configurations in terms of their cost.

Finding the globally optimum configuration is a hard problem. These methods, which have a significant engineering component, are usually reliable but tend to be expensive in terms of their time requirement.

5 Error Recovery in YACC

The parser generator YACC has provided some features for error recovery. While these features are minimal and restrictive, it is possible to incorporate reasonable error recovery in a compiler by using them. We discuss this method in detail for the rest of this module.

YACC generates a LALR(1) parser. What are the features provided by YACC for error recovery ? A predefined terminal symbol, called `error` and a special routine named *yyerrok* are provided by YACC. The purpose of the symbol `error` is to treat input error as a special instance of this symbol.

The symbol `error` is used as a terminal in all aspects, except that it is not produced by the scanner. If the parser is in a configuration, where the parsing table entry is an *error*, i.e., action[u,a] := *error*, then `error` symbol plays a crucial role. The general approach of the error recovery method used in YACC can be described as follows.

5.1 Error Productions and their use

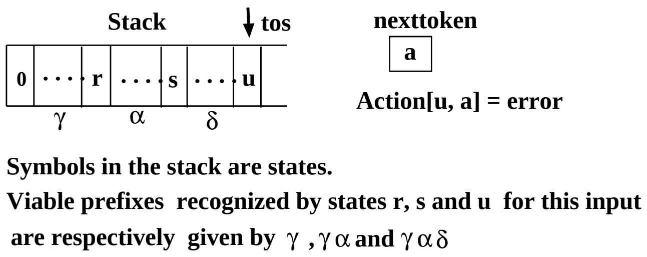
Compiler writer is permitted to add extra rules that have the symbol `error` in the rhs, to take care of the erroneous inputs. A form such as $A \rightarrow \alpha$ `error` β is permitted. Such rules are referred to as *error-productions.*

When the parser encounters an error, it tries to use a relevant *error-production.* Let the configuration at an error point be as shown in Figure **??**(a). The relevant slice of the LALR automaton is also given in Figure **??**(b).

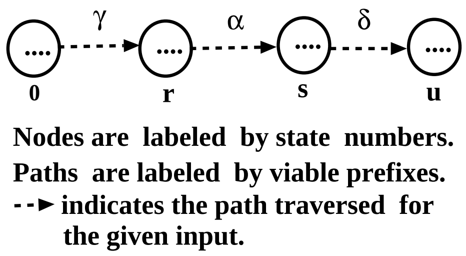
States from the stack are popped off till a state, say s, is found that contains an item of the form $A \rightarrow \alpha \bullet$ `error` β.

The corresponding configuration and automaton are given in Figure **??**. The assumption is that state u and the states on the path labeled δ do not have an *error-production* of the kind given above. The absence of such a state results in an abnormal termination of the parser. This situation is called as the *parser falling off the stack.*

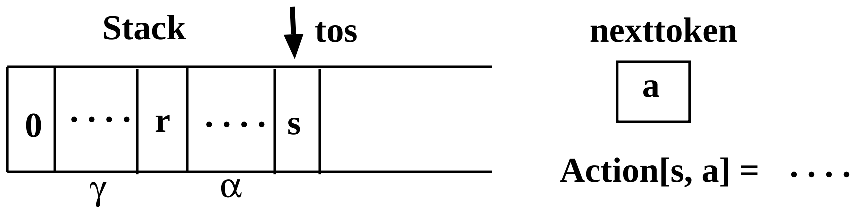
The parser resumes with state s and makes a fictitious *shift* by moving the symbol `error` and the state t onto the stack, reaching the configuration as shown in Figure **??**.


Stack
tos
nexttoken
a
0
r
s
u
γ
α
δ
Action[u, a] = error
Symbols in the stack are states.
Viable prefixes recognized by states r, s and u for this input are respectively given by γ , γα and γαδ

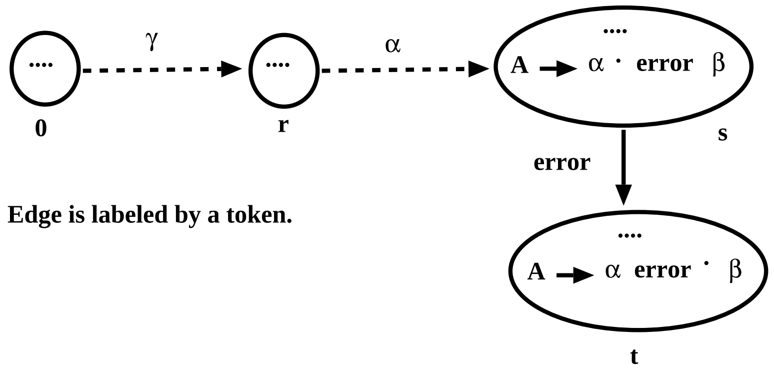
(a) Parser Configuration


γ
α
δ
0
r
s
u
Nodes are labeled by state numbers.
Paths are labeled by viable prefixes.
indicates the path traversed for the given input.

(b) Relevant Slice of the LALR Automaton



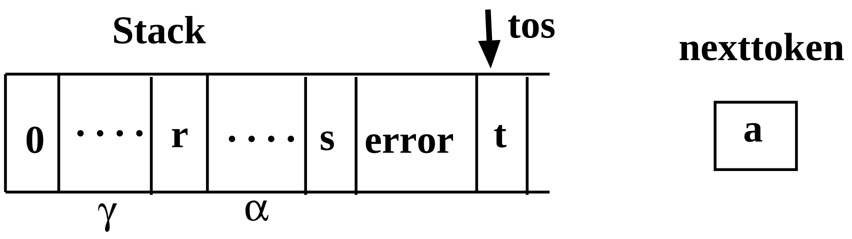
(a) Parser Configuration



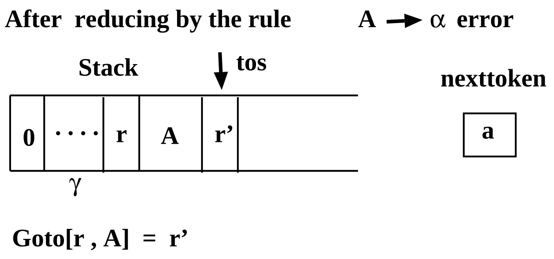
(b) Relevant Slice of the LALR Automaton

From the state t, which contains the item $A \rightarrow\ \alpha$ `error` $\bullet\beta$, the parser takes recovery action depending upon the string β. We discuss two specific instances of β.

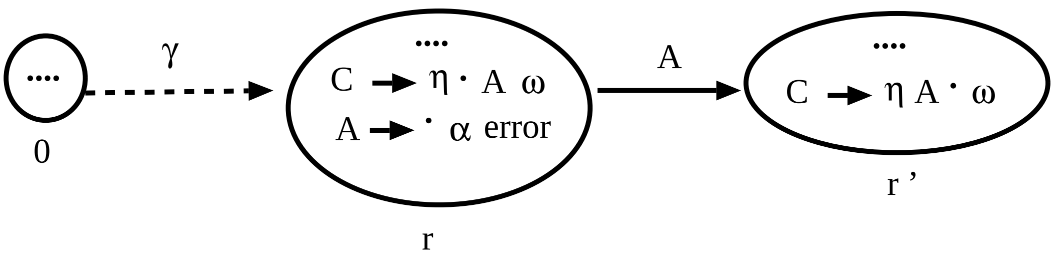
Case $\beta = \epsilon$: In this situation, α `error` is reduced to A and the semantic action specified



with the rule is executed (user specified error recovery). The resulting configuration and automaton are shown in Figure ??. If Action[r',a] is *shift*, then recovery has been successful. In the worst case input tokens may be discarded, one by one, till a token is found on which the parser can resume.



(a) Parser Configuration

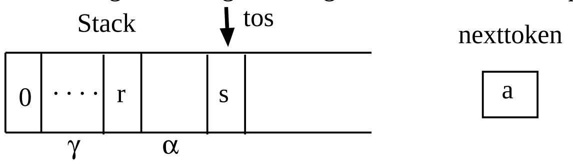


(b) Relevant Slice of the LALR Automaton

Case $\beta = \texttt{w}$: In this case, the parser tries to find the string of terminals `w` on the input, deleting symbols that do not match. The deletion of nonmatching tokens are necessary, since otherwise the parser may get into an endless loop. This situation is explained in Figure ??. When `w` has been found and also shifted, a reduction by the error production takes place. The configuration reached has A on *tos*. Recovery is successful since tokens beyond the error point have been shifted and consumed.

In case `w` is not found in the input, the parser may again terminate abnormally. This situation is known as the *parser falling off the input buffer*. In both the situations described above, it is possible for the parser to resume parsing beyond the error point.

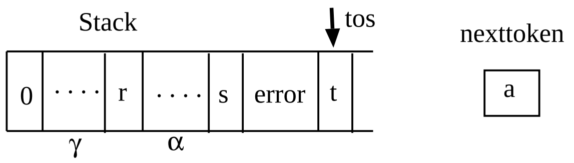
1 : Reaching the configuration given below in normal parsing.



Action [s, a] = error

Relevant part of LALR automaton as shown in Figure 4.7(b)

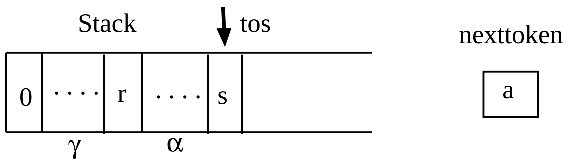
2 : Reaching the configuration given below after pushing error and state t on the stack in recovery phase.



Action [t, a] = error

3: Reaching the configuration shown below on encountering another error in situation 2, in the recovery phase.

It is assumed that state t does not contain an error production of the form, $C \rightarrow \delta \cdot$ error η , i.e., a production in which $\cdot$ precedes the error symbol. Recovery pops off state t and error resulting in :



The configuration is same as that of situation 1.

The parser is in an endless loop.

However, it is possible that more errors are encountered immediately after recovery and on resumption of normal parsing. This, in turn, could lead to a cascade of error messages. YACC controls this problem by adopting a policy that *the parser must shift three terminal symbols beyond the error point, before another error results in an error message.*

5.2 Parsing and Recovery Algorithm

The basic approach is briefly described in the following.

1. The parser normally works in the parsing phase.

2. On encountering an error, the parser enters the error recovery phase. An error production is used to continue parsing in the recovery phase.
3. If the parser can shift three consecutive tokens (without encountering another error) in the recovery phase, then it returns to the normal parsing phase.
4. Failing above, the parser remains in the recovery phase (inserting or deleting tokens, as appropriate), till the condition for return to normal parsing is met.

The special routine, *yyerrok*, when invoked from within the semantic actions, causes the parser to abandon recovery and return back to normal parsing. As a consequence of *yyerrok*, user can force the parser to resume reporting of error messages even before three consecutive tokens are successfully consumed after an error.

The context in which the error recovery algorithm is invoked from within the parser is given in Algorithm 4.1.

```
Algorithm 4.1
Initialize Stack ; get a token in next_token ;
repeat
   case
       : ACTION[ tos(Stack), next_token ] = shift :
                execute the shift action ;

       : ACTION[ tos(Stack), next_token ] = reduce :
                execute the reduce action ;

       : ACTION[ tos(Stack), next_token ] = accept :
                yyparse = 0 ; { normal termination } exit ;

       : ACTION[ tos(Stack), next_token ] = – :
                begin
                    { parsing table entry indicates error}
                    issue error message ;
                    call recovery ( Stack, next_token ) ;
                end
   end case
forever
```

```
end Algorithm 4.1
```

Execute an action implies changing of the stack and/or next_token as appropriate. Reporting of error messages can be done by using the YACC variables that refer to the line number, current token, etc. A parser can be viewed as working in two separate modes, normal parsing mode and error recovery mode.

The parser starts off in the normal parsing mode and enters the error recovery mode only when it encounters an error. It continues to operate in the recovery mode till either
(i) it meets the built-in criteria for end of recovery, or
(ii) it is forced by the user to come out of the recovery mode.
In either case the parser returns to the normal parsing mode (with a different configuration usually).

A pseudo code of the recovery scheme employed by YACC is given in Algorithm 4.2.

```
Algorithm 4.2
  procedure recovery( Stack, next_token )
    begin
      shift_cnt = 0 ; { the number of consecutive symbols
      shifted so far }
  start: mark the pair ( tos( Stack ), next_token ) ;

      while ( ACTION[ tos(Stack), error ] $\neq$ shift
      and Stack $\neq$ $\Phi$ ) do pop ( Stack );
      { pop deletes the tos state symbol and also
      the grammar symbol below it }

      if Stack = $\Phi$ then
      begin
        yyparse = 1 ; { parser falls off the stack }
        exit
      end;

      execute the move shift error ;

     repeat
```

```
case
   : ACTION[ tos(Stack), next_token ] = shift :
      begin
         execute the shift action ;
         shift_cnt ++ ;
         if shift_cnt = 3
         then return to normal parsing ;
      end
   : ACTION[ tos(Stack), next_token ] = reduce :
      begin
         execute the reduce action ;
         if *yyerrok* is invoked in the
         semantic part of this rule
         then return to normal parsing
      end
   : ACTION[ tos(Stack), next_token ] = accept :
      begin
         yyparse = 0 ;
         exit ; { end of parsing }
      end

   : ACTION[ tos(Stack), next_token ] = - :
      begin
         { situation when parsing table
         entry indicates error while in
         recovery phase. }

         if ( tos(Stack), next_token ) is *marked*
         then begin
            { current token is deleted }

            if  current token is end of input
            then begin
               yyparse = 1 ;
               exit ;
               { *parser falls off the input* }
            end
```

```
                    else begin
                       next_token = lex( ) ;
                       { new token obtained }
                       shift_cnt = 0 ;
                       { tokens shifted so far discounted }
                    end

                 else goto  start
          end case
       forever
    end procedure recovery
  end Algorithm 4.2
```

5.3 Example Grammar without using Error Recovery

How does yacc generated parser work if the recovery features are not used ? We illustrate using an example grammar. Consider the grammar : $E \rightarrow E + E \mid$ `id`.

The LALR parsing table generated by YACC is given below. The format of the parsing table entries is explained in the following. It is usually contained in a file called *y.output*. The states of the parsing table are listed in sequence. For a state, the entries in order are as given below. Kernel items of the state (nonkernel items are not listed) are listed first followed by the action part of the table and then the goto part of the table.

The ACTION table is listed in a compact form.

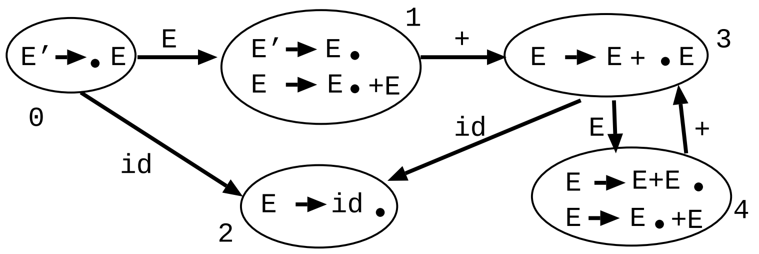
1. Shift entries, if any, are listed first
2. Reduce entries, if any, are listed next
3. Error entries, if any, are listed at the end
4. If a state has reduce entries, then all the error entries are replaced by the most frequently used reduction in this state. See entries for states 2 and 4.
5. If a state has no reduce entries then after the shift entries, the single entry, . error, is used.

The symbol . is used to denote the remaining terminals, i.e., those which are not listed above in the shift/reduce entries. See entries for states 0, 1 and 3. Only the nonblank transitions of the GOTO table are listed. See the entries for the states 0 and 3. The LALR automaton for this grammar is given in Figure **??** and the compressed parsing table is given below.

```
    state 0:                       state 3:
$accept : • E $end             E : E + • E
id  shift 2                    id  shift 2
.  error                       .  error
E  goto 1                      E  goto 4

    state 1:                       state 4:
$accept : E • $end             E : E • + E
E : E • + E                    E : E + E •
$end  accept                   +  shift 3
+  shift 3                     .  reduce 1
.  error

    state 2:  E : id •
              .  reduce 2
```



Parse the input : `a + +`
The actions of the parser (can be seen explicitly by debugging the parser) are as follows.

Normal Parsing Mode :
push 0; reading '`id`'; push 2; reduce (uncovers 0); push 1; reading '+'; push 3; reading '+'; { *error found* }

Recovery Mode :
error 1 : line 1 near "+" ;
pops 3 (uncovers 1); { no *error-production* }; pops 1 (uncovers 0); { no *error-production*

};pops 0; { no *error-production* }; stack empty; yyparse() == 1; { *abnormal termination* }

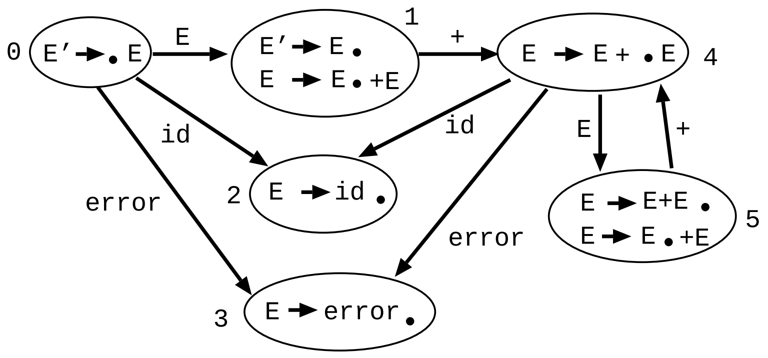
The parser used the built-in recovery , but failed to parse beyond the first error point. No features provided for recovery were used. In fact the parser *fell off the stack* in this situation.

5.4 Example Grammar using Error Recovery

We add an *error-production* to our previous grammar, as indicated below.

$$\begin{aligned} \mathrm{E} &\rightarrow \mathrm{E{+}E} \mid \texttt{id} \\ &\quad \mid \texttt{error} \end{aligned}$$

The LALR automaton for this grammar is given in Figure **??** and the compressed parsing table is given below.



Only Kernel Items are shown

Parse the input : `a + +` and note the difference in behaviour. One error message and successful parsing.

<u>state 0</u>:	<u>state 4</u>:
$accept : • E $`end`	E : E + • E
`error shift 3`	`error shift 3`
`id shift 2`	`id shift 2`
. `error`	. `error`
E `goto 1`	E `goto 5`
<u>state 1</u>:	<u>state 5</u>:
$accept : E • $`end`	E : E • + E

```
E : E • + E                E : E + E •
$end  accept               + shift 4
+  shift 4                 .  reduce 1
.  error
     state 2:                    state 3:
 E : id •                   E : error •
.  reduce 2                .  reduce 3
```

To get a better view, parse an input for which the parser encounters more than one error, such as, a + + b − + c. Single error message and normal termination of parsing. The shifting of three symbols beyond the error point criterion gets applied here.

5.5 Use of YYERROK

We augment our grammar by including the other feature of YACC, namely *yyerrok*, in the semantic action part, as shown below.

```
E
  : E + E
  | id
    { yyerrok }
  | error
```

The placement of this action is important. If used for the input of the preceding example, a + + b − + c, it causes generation of the missing error message, as explained below.

Normal Parsing Mode :
push 0; reading `id`; push 2; reduce (uncover 0); push 1; reading '+'; push 4; reading '+'; { *error found* ; enters recovery }

Recovery Mode :

error 1 : line 1 near symbol "+";

reading '`error`'; push 3; reduce (uncovers 4); push 5; reading '+'; push 4; reading '`id`'; push 2; reduce (uncovers 4); { *yyerrok* invoked ; resumes normal parsing }

Normal Parsing Mode :
push 5; reduce (uncovers 4); push 5; reduce (uncovers 0); push 1; reading '−'; { *error found* ; enters recovery }

Recovery Mode :

error 2 : line 1 near symbol "−";

pops 1; { no *error-production in 1* }; reading '`error`'; push 3; reduce (uncovers 0); push 1; reading '−'; { *looping detected* }; delete '−'; reading '+'; push 4; reading '`id`'; push 2; reduce (uncovers 4); { *yyerrok* invoked ; resumes normal parsing }

Normal Parsing Mode :
push 5; reduce (uncovers 0); push 1; reading '\$'; yyparse == 0; *normal termination*

5.6 Guidelines for writing Error Productions

In the following some general guidelines are suggested for using these features. An obvious requirement is to place an `error` symbol close to the start symbol of the grammar in order to avoid the parser from falling off the stack. Placement of the `error` token in the vicinity of terminals is also important. This is because if an error is detected there, few source tokens may be lost. In recursive rules, the `error` token needs to be placed with care . The following observations highlights this point.

Consider a rule of the form given below.

$$\begin{aligned} A \to & A\ \texttt{a} \\ & \mid \epsilon \end{aligned}$$

The only state of the parser which may encounter an error is the one that contains the item, A → A • `a`. In order to recover, one must admit errors in this state. An item of the form, A → A • `error`, must be present in the state. The required grammar is given below.

$$\begin{aligned} A \to & A\ \texttt{a} \mid \epsilon \\ & \mid A\ \texttt{error} \end{aligned}$$

For a rule of the form given below :

$A \rightarrow A$ `a` | `a` , we rewrite it as
$A \rightarrow A$ `a` | `a`
$A \rightarrow$ `error` | A `error`

For a rule of the form given below :

$A \rightarrow A$ `t` `a` | `a` , we rewrite it as
$A \rightarrow$ `error` | A `error`
$A \rightarrow A$ `t` `error` | A `error` `a`

The absence of the last two *error-productions* may lead to *deletion* of input tokens, including *endmarker* \$.

yyerrok is best invoked after the first rule in each of the three situations. It should also be invoked after the last rule in the third situation.

The rules that have `error` in their rhs followed by a terminal are typical candidates that may invoke *yyerrok* in their semantic actions. The reason being that a new token has been consumed (beyond the error point) at this stage , so normal parsing may be forced, if so desired.

The important terminals, typically } `)` `and` ; are used in this manner. Regardless of the context, they may signal resumption of normal parsing by invoking *yyerrok* in the rules they belong to. Adding the extra *error-production* rules may result in conflicts in an otherwise functional parser. These conflicts are resolved by examining the offending items and taking appropriate decisions. The error productions and *yyerrok* have to used with caution since their implication on the behaviour of the generated parser is difficult to ascertain.

6 Graded Exercises

1 Consider LR parsing tables and the error entries in the GOTO part of the table. What role are played by these entries as far error detection is concerned ?

2 In YACC, the LALR table of the generated parser is stored in a compact form. The compaction is typically concerned with the storing of the error entries since these dominate the other entries usually. One possible compaction was mentioned in Exercise 6 in module 3.
(a) Find out the strategy employed by YACC to store the entries of a LALR(1) state. In particular, for a state that has several shift, reduce and error entries how are they stored internally.
(b) Reason out the correctness of a LALR(1) parser which stores its states as done by YACC. Comment particularly on the error detection capability of such a parser as compared to a LALR(1) parser that uses the entire table.

3 Consider the following declaration grammar.

$$\begin{aligned}
&\text{D} \rightarrow \text{D var I : T ;} \mid \epsilon \\
&\text{T} \rightarrow \text{S} \mid \text{A} \\
&\text{I} \rightarrow \text{id} \mid \text{I , id} \\
&\text{S} \rightarrow \text{integer} \mid \text{real} \\
&\text{A} \rightarrow \text{array [num .. num] of S}
\end{aligned}$$

(a) Construct a LALR(1) parsing table for the grammar.
(b) For each error entry of this table, identify the causes of error and write appropriate error messages. Your design should be able to classify entries that may have similar error messages.
(c) Provide manual error recovery for each of the error situations obtained in (b) above.

4 Consider the grammar given in Exercise 3. It is assumed that the following rule S → D has been added.
(a) Construct an input for which the YACC generated parser would fall off the stack while parsing it.
(b) Construct an input for which the YACC generated parser would fall off the input buffer while parsing it. No error recovery features of YACC are to used in

the grammar.

(c) Add the rule D' $\rightarrow$ error to the grammar of part (a). Construct an input for which the parser would fall off the input buffer. Is it possible to construct an input for which this parser may fall off the stack ?

5 Consider the facility *yyerrok* provided by YACC. Is it possible that an improper use of *yyerrok* leads to the resulting parser in falling off the stack (or input buffer)? Provide an example if your answer is yes else prove why such a situation is not possible.

6 Consider the following grammar.

declist $\rightarrow$ declist decl ; | ϵ
decl $\rightarrow$ var list : integer
list $\rightarrow$ id | list , id

(a) The following error rules are added to this grammar. Detect all conflicts that arise because of additions of these error productions. Analyse the entries of the states containing the conflicts and applying the guidelines suggested to eliminate the redundant error productions from the grammar.

declist $\rightarrow$ error
declist $\rightarrow$ declist error
declist $\rightarrow$ declist error decl
declist $\rightarrow$ declist decl error ;
declist $\rightarrow$ declist decl ; error
decl $\rightarrow$ error
decl $\rightarrow$ var error
decl $\rightarrow$ var list error
decl $\rightarrow$ var list error integer
decl $\rightarrow$ var list : error
list $\rightarrow$ error
list $\rightarrow$ list error
list $\rightarrow$ list error id
list $\rightarrow$ list , error

(b) Using the method suggested in part (a) above, construct a grammar using error production rules which give a conflict free parser. Add *yyerror* against the relevant rules to complete the error recovery capability of your parser.
(c) Examine your error recovery parser with respect to the following.
i) provide an erroneous input for which the parser's recovey action leads to insertion kind of recovery.
ii) provide an erroneous input for which the parser's recovey action leads to deletion kind of recovery.
iii) provide inputs to show the effectiveness of the *yyerrok* actions.

7 References

1. Aho, A.V., R. Sethi and J.D. Ullman (1986) : *Compilers – Principles, Techniques and Tools*, Addison-Wesley, Reading.
2. Holub, A.I. (1993): *Compiler Design in C* Prentice-Hall, Englewood-Cliffs.
3. Levine, J.R., T. Mason and D. Brown (1990): *Lex and Yacc*, 2^{nd} edition, O'Reilly & Associates, Sebastopol.
4. Schreiner, A.T., H.G. Friedman Jr. (1985): *Introduction to Compiler Construction with UNIX* Prentice-Hall, Englewood Cliffs.
5. Tremblay, J.P. and P.G. Sorenson (1985): *The Theory and Practice of Compiler Writing*, McGraw-Hill, New York.