Error Recovery

MICHAEL WOLLOWSKI

BASED ON DRAGON BOOK

Error recovery goals

What should happen when your parser finds an error in the user's input?

- stop immediately and signal an error
- record the error but try to continue

In the first case, the user must recompile from scratch after possibly a trivial fix

In the second case, the user might be overwhelmed by a whole series of error messages, all caused by essentially the same problem

Strategies

Many different general strategies

No one strategy has proven itself to be universally acceptable

However, a few methods have broad applicability.

We introduce the following strategies:

- panic mode
- phrase level
- error productions
- global correction

Panic-mode recovery

Parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found.

Typical synchronizing tokens:

- delimiters, such as a semicolon, opening or closing parenthesis

Now you know why some programming languages separate statements with semicolons.

Panic-mode recovery - Evaluation

Simplest method to implement

Can be used by most parsing methods.

Panic-mode correction often skips a considerable amount of input without checking it for additional errors

Advantage of simplicity.

When multiple errors in the same statement are rare, this method is quite adequate.

Phrase-level recovery

On discovering an error, a parser may perform local correction on the remaining input

For example, it may replace a prefix of the remaining input by some string that allows the parser to continue.

A typical local correction would be to:

- replace a comma by a semicolon,
- delete an extraneous semicolon, or
- insert a missing semicolon.

The choice of the local correction is left to the compiler designer.

Phrase-level recovery - Evaluation

This type of replacement can correct any input string and has been used in several error-repairing compilers.

The method was first used with top-down parsing.

Major drawback: Situations in which the actual error has occurred before the point of detection.

Error productions

If we have a good idea of the common errors then augment the grammar with *error productions* that generate the erroneous constructs.

Use the grammar augmented by these error productions to construct a parser.

If an error production is used by the parser, generate an appropriate error diagnostic message.

We will focus on this approach in a few slides.

Global correction

Ideally, we would like a compiler to make as few changes as possible in processing an incorrect input string.

There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction.

Given an incorrect input string *x* and grammar *G,* these algorithms will find a parse tree for a related string *y,* such that the number of insertions, deletions, and changes of tokens required to transform *x* into *y* is as small as possible.

Global correction - Evaluation

Too costly.

At this time, they are of theoretical interest.

The closest correct program may not be what the programmer had in mind.

Nevertheless, the notion of least-cost correction provides a yardstick for evaluating error-recovery techniques

It has been used for finding optimal replacement strings for phrase-level recovery.

Top-down Parsing

An error is detected during predictive parsing when the terminal top of the stack does not match the next input symbol or

When nonterminal A is on top of the stack, *a* is the next input symbol, and the parsing table *M[A, a]* is empty.

Synchronizing Tokens

As a starting point, we can place all symbols in FOLLOW(*A*) into the synchronizing set for nonterminal *A.*

If we skip tokens until an element of FOLLOW(*A*) is seen and pop *A* from the stack, it is likely that parsing can continue.

If we add symbols in FIRST(*A*) to the synchronizing set for nonterminal *A,* then it may be possible to resume parsing according to *A* if a symbol in FIRST(*A*) appears in the input.

Synchronizing Tokens

If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was removed, and continue parsing.

Synchronizing Tokens

NONTER-MINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**	synch	synch	$F \to (E)$	synch	synch

Fig. 4.18. Synchronizing tokens added to parsing table of Fig. 4.15.

Example

Use of parsing table:

- If the parser looks up entry M[*A, a*] and finds that it is blank, then the input symbol *a* is skipped.
- If the entry is *synch*, then either:
  - Pop the nonterminal on top of the stack or
  - Skip input until one in First(A) is found.
- If a token on top of the stack does not match the input symbol, then we pop the token from the stack.

We will attempt to parse: **) id * + id**

Solution

STACK	INPUT	REMARK
$E	) id * + id $	error, skip)
$E	id * + id $	id is in FIRST(E)
$E'T	id * + id $	
$E'T'F	id * + id $	
$E'T'id	id * + id $	
$E'T'	* + id $	
$E'T'F*	* + id $	
$E'T'F	+ id $	error, M[F, +] = synch
$E'T'	+ id $	F has been popped
$E'	+ id $	
$E'T+	+ id $	
$E'T	id $	
$E'T'F	id $	
$E'T'id	id $	
$E'T'	$	
$E'	$	
$	$	

Fig. 4.19. Parsing and error recovery moves made by predictive parser.

Synchronizing Tokens

It is not sufficient to use FOLLOW(*A*) as the synchronizing set for *A.*

For example, if semicolons terminate statements, as in C, then keywords that begin statements will not appear in the FOLLOW set of the nonterminal generating expressions.

A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped.

Often, there is a hierarchical structure on constructs in a language; e.g., expressions appear within statements, which appear within blocks.

We can add to the synchronizing set of a lower construct to the symbols that begin higher constructs.

For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generating expressions.

Synchronizing Tokens

If a nonterminal can generate the empty string, then the production deriving ε can be used as a default.

Doing so may postpone some error detection, but cannot cause an error to be missed.

This approach reduces the number of non-terminals that have to be considered during error recovery.

Bottom-up Parsing

STATE	action						goto
	id	+	*	(	)	$	*E*
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			8
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Fig. 4.47. Parsing table for grammar (4.22).

Bottom-up Parsing

Error recovery: Complete parsing table for all symbols.

Bottom-up Parsing

STATE	action						goto
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			8
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Fig. 4.47. Parsing table for grammar (4.22).

STATE	action						goto
	id	+	*	(	)	$	E
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

Fig. 4.53. LR parsing table with error routines.

Error Routines

E1:

- This routine is called from states 0, 2, 4 and 5, all of which expect the beginning of an operand, either an **id** or a left parenthesis.
- Instead an operator, + or *, or the end of the input was found.
- Push an imaginary **id** onto the stack and cover it with state 3 (the goto of states 0, 2, 4 and 5 on **id**)
- Issue diagnostic "missing operand"

E2:

- This routine is called from states 0, 1, 2, 4 and 5 on finding a right parenthesis.
- Remove the right parenthesis from the input
- Issue diagnostic "unbalanced right parenthesis"

Error Routines

E3:

- This routine is called from states 1 or 6 when expecting an operator and an **id** or left parenthesis is found.
- Push + onto the stack and cover it with state 4.
- Issue diagnostic "missing operator"

E4:

- This routine is called from state 6 when the end of the input is found
- State 6 expects an operator or a right parenthesis
- Push a right parenthesis onto the stack and cover it with state 9
- Issue diagnostic “missing right parenthesis”

Reductions

Change each state that calls for a particular reduction on some input symbols by replacing error entries in that state by the reduction.

This change has the effect of postponing the error detection until one or more reductions are made.

However, the error will still be caught before any shift takes place.

Solution to Worksheet

On the erroneous input **id** +) discussed in Example 4.32, the sequence of configurations entered by the parser is shown in Fig. 4.54. □

STACK	INPUT	ERROR MESSAGE AND ACTION
0	**id**+)$	
0**id**3	+)$	
0*E*1	+)$	
0*E*1+4	)$	
0*E*1+4	$	"unbalanced right parenthesis" e2 removes right parenthesis
0*E*1+4**id**3	$	"missing operand" e1 pushes **id** 3 on stack
0*E*1+4*E*7	$	
0*E*1	$	

Fig. 4.54. Parsing and error recovery moves made by LR parser.

Error Recovery in Lexer

Consider:

```
fi (a == f(x)) ...
```

Possible error-recovery actions:

- Transpose two adjacent characters
- Delete an extraneous character
- Insert a missing character
- Replace an incorrect character by a correct one