Error-Recovery Strategies

Types of Parsers

- Universal
  Can parse any grammmar
  - Cocke-Younger-Kasami parsing method
  - Earley's algorithm

  In-efficient to be used in production of compilers.
- Top-down - build purse trees
  Starts parsing from top (root) to the bottom (leaves).
- Bottom-up
  Starts from leaves and work their way upto the root.
- Both in top-down and bottom up
  - scan the input from left to right.
  - one symbol at a time.

Error Handling Techniques

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

Error Handling Techniques

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

- Stop immediately and signal an error.

Error Handling Techniques

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

- Stop immediately and signal an error.
- Record the error but try to continue.

Error Handling Techniques

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

- Stop immediately and signal an error.
- Record the error but try to continue.

In the first case user must recompile the scratch after 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.

Error Handling Techniques:

There are mainly four types of error. They are as follows:

- Lexical Error: Such as misspelling an identifier, keyword or operator.
- Syntactic Error: Such as an arithmetic expression with unbalanced parentheses.
- Semantic Error: Such as an operator applied to an incompatible operand.
- Logical Error: Such as infinitely recursive call.

Error-Recovery Strategies

- Panic Mode.
- Phrase Level Recovery.
- Error Production.
- Global Correction.

Panic Mode Error Recovery

- It is based on the idea of skipping symbols on the input until in a selected set of synchronizing tokens appears.
- In case of an error like:
  a = b + c // no semi-colon
  d = e + f;
  The compiler will discard all subsequent tokens till a semicolon is encountered.

Phrase Level Recovery.

Perform local correction on the input to repair the error.

- Change input stream by inserting missing tokens
- For example: **int id 5;** is changed into **int id=5;**

Error Production

Augment grammar with productions for erroneous constructs.

Global Correction

Choose a minimal sequence of changes to obtain a global least-cost correction

Error Recovery in Predictive Parsing

- An error detected during predictive parsing when the *terminal* on top of the stack does not match the *next input symbol*.
- when nonterminal A is on top of the stack, a is the next input symbol, and the parsing table entry $M[A, a]$ is empty.

Some heuristics for the choice of Synchronizing Set are as follows:

1. As a starting point, place all symbols in $FOLLOW(A)$ into the synchronizing set for the nonterminal A. Skip the tokens until an element of $FOLLOW(A)$ is seen and pop A from the stack, it is likely that parsing can continue.
2. If we add symbols in FIRST(A) to the synchronizing set for non terminal A, then it may be possible to resume parsing according to A if a symbol in FiRST(A) appears in the input.
3. 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 inserted and continue the parsing.

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}
FIRST(E') ={ +, ϵ}
FIRST(T') ={*,ϵ}
FOLLOW(E) = FOLLOW(E') ={),$}
FOLLOW(T)=FOLLOW(T')={+,),$ }
FOLLOW(F) = {*, +,), $}

Table: Synchronizing Tokens added to Predictive Parsing Table

Non Terminal	*id*	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	Synch	Synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	Synch		$T' \rightarrow FT'$	Synch	Synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	Synch	Synch	$F \rightarrow (E)$	Synch	Synch

- **Synch** the driver pops current nonterminal A and skips input till synch token or skips input until one of FIRST(A) is found
- If the parser looks up entry M[A,a] and finds that it is blank, the input symbol a is skipped.
- If the entry is synch, the the nonterminal on top of the stack is popped.
- If a token on top of the stack does not match the input symbol, then we pop the token from the stack.

Table: Parsing and error recovery moves made by predictive parser.

	Stack	Input	Output
(1)	$E	) id * + id $	error, Skip)

Table: Parsing and error recovery moves made by predictive parser.

	Stack	Input	Output
(1)	$E	) id * + id $	error, Skip)
(2)	$E	id * + id $	**id** is in FIRST(E)

Table: Parsing and error recovery moves made by predictive parser.

	Stack	Input	Output
(1)	$E	) id * + id $	error, Skip)
(2)	$E	id * + id $	**id** is in FIRST(E)
(3)	$E'T	id * + id $	

Table: Parsing and error recovery moves made by predictive parser.

	Stack	Input	Output
(1)	$E	) id * + id $	error, Skip)
(2)	$E	id * + id $	**id** is in FIRST(E)
(3)	$E'T	id * + id $	
(4)	$E'T'F	id * + id $	
(5)	$E'T'id	id * + id $	
(6)	$E'T'	* + id $	
(7)	$E'T'F*	* + id $	

Table: Parsing and error recovery moves made by predictive parser.

	Stack	Input	Output
(1)	$E	) id * + id $	error, Skip)
(2)	$E	id * + id $	**id** is in FIRST(E)
(3)	$E'T	id * + id $	
(4)	$E'T'F	id * + id $	
(5)	$E'T'id	id * + id $	
(6)	$E'T'	* + id $	
(7)	$E'T'F*	* + id $	
(8)	$E'T'F	+ id $	error, M[F, +] = Synch

Table: Parsing and error recovery moves made by predictive parser.

	Stack	Input	Output
(1)	$E	) id * + id $	error, Skip)
(2)	$E	id * + id $	**id** is in FIRST(E)
(3)	$E'T	id * + id $	
(4)	$E'T'F	id * + id $	
(5)	$E'T'id	id * + id $	
(6)	$E'T'	* + id $	
(7)	$E'T'F*	* + id $	
(8)	$E'T'F	+ id $	error, M[F, +] = Synch
(9)	$E'T'	+ id $	F has been popped

Table: Parsing and error recovery moves made by predictive parser.

	Stack	Input	Output
(1)	$E	) id * + id $	error, Skip)
(2)	$E	id * + id $	**id** is in FIRST(E)
(3)	$E'T	id * + id $	
(4)	$E'T'F	id * + id $	
(5)	$E'T'id	id * + id $	
(6)	$E'T'	* + id $	
(7)	$E'T'F*	* + id $	
(8)	$E'T'F	+ id $	error, M[F, +] = Synch
(9)	$E'T'	+ id $	F has been popped
(10)	$E'	+ id $	
(11)	$E'T+	+ id $	
(12)	$E'T	id $	
(13)	$E'T'F	id $	
(14)	$E'T'id	id $	
(15)	$E'T'	$	
(16)	$E'	$	
(17)	$	$	

Phrase Level Recovery

Phrase Level Recovery in Predictive Parsing

- Each cell can be filled with a special-purpose error routine.
- Such routines typically remove tokens from the input, and/or pop an item from the stack.
- It is ill-advised to modify the input stream or the stack without removing items, because it is then hard to guarantee that error recovery will always terminate

Phrase Level Recovery in Predictive Parsing

Change input stream by inserting missing tokens

For example: **id id** is changed into **id * id**

Table: Phrase Level entry added to Predictive Parsing Table

Non Terminal	*id*	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	Synch	Synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	Synch		$T \rightarrow FT'$	Synch	Synch
T'	Insert *	$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	synch	Synch	$F \rightarrow (E)$	Synch	Synch

Note: insert *: driver inserts missing * and retries the production

Error Production

Recovery using *Error Production* in Predictive Parsing

- Include productions for common errors.
- As for example, to ignore missing * in **id id**
- Add **Error production** : $T' \rightarrow FT'$

Table: **Error Production** added to Predictive Parsing Table

Non Terminal	*id*	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	Synch	Synch
E'		$E' \rightarrow TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	Synch		$T' \rightarrow FT'$	Synch	Synch
T'	$T' \rightarrow FT'$	$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	synch	Synch	$F \rightarrow (E)$	Synch	Synch

Note: Powerful recovery method but there is a need of manual addition of productions.

Error Handling in LR Parsing

Phrase level recovery - for each error entry in the table insert a pointer to a particular error procedure, which assumes the most likely cause and takes the appropriate action.
Consider the following grammar production rules

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

Table: SLR Parsing Table For the given grammar

State	Action						Goto		
	id	+	*	(	)	$	*E*	*T*	*F*
0	S_5	e1	e1	S_4	e2	e1	1	2	3
1	e3	S_6	e4	e3	e2	Accept			
2	e3	r_2	S_7	e3	r_2	r_2			
3	e3	r_4	r_4	e3	r_4	r_4			
4	S_5	e1	e1	S_4	e2	e1	8	2	3
5	e3	r_6	r_6	e3	r_6	r_6			
6	S_5	e1	e1	S_4	e2	e2		9	3
7	S_5	e1	e1	S_4	e2	e2			10
8	e3	S_6	e4	e3	S_{11}	e5			
9	e3	r_1	S_7	e3	r_1	r_1			
10	e3	r_3	r_3	e3	r_3	r_3			
11	e3	r_5	r_5	e3	r_5	r_5			

Error Procedures:

e1 /* Expecting an **id** or an "(" , but finding an '+' , '*' or '$' */
put 5 on top of the stack
issue **"missing operand"** message.

e2 /* Finding an Unexpected ')' */
remove ')' from input. /* ignore it */
issue "Unmatched right parentheses " message.

e3 /* Expecting '+' , finding **id** or '(' */
put 6 on top of the stack, /* assume '+' */
issue "missing + " message.

e4 /* Expecting '+' , finding '*' */
put 6 on top of the stack, /* assume '+' */
remove '*' from input.
issue " '*' instead of '+' "message.

e5 /* Expecting ')', finding '$' */
put 11 on stack /* assume ')' */
issue "missing right parenthesis" message.

Parser Generators — Yacc