Don’t Panic! Reducing Cascading Parsing Errors

Lukas Diekmann

Laurence Tratt

KING’S College LONDON

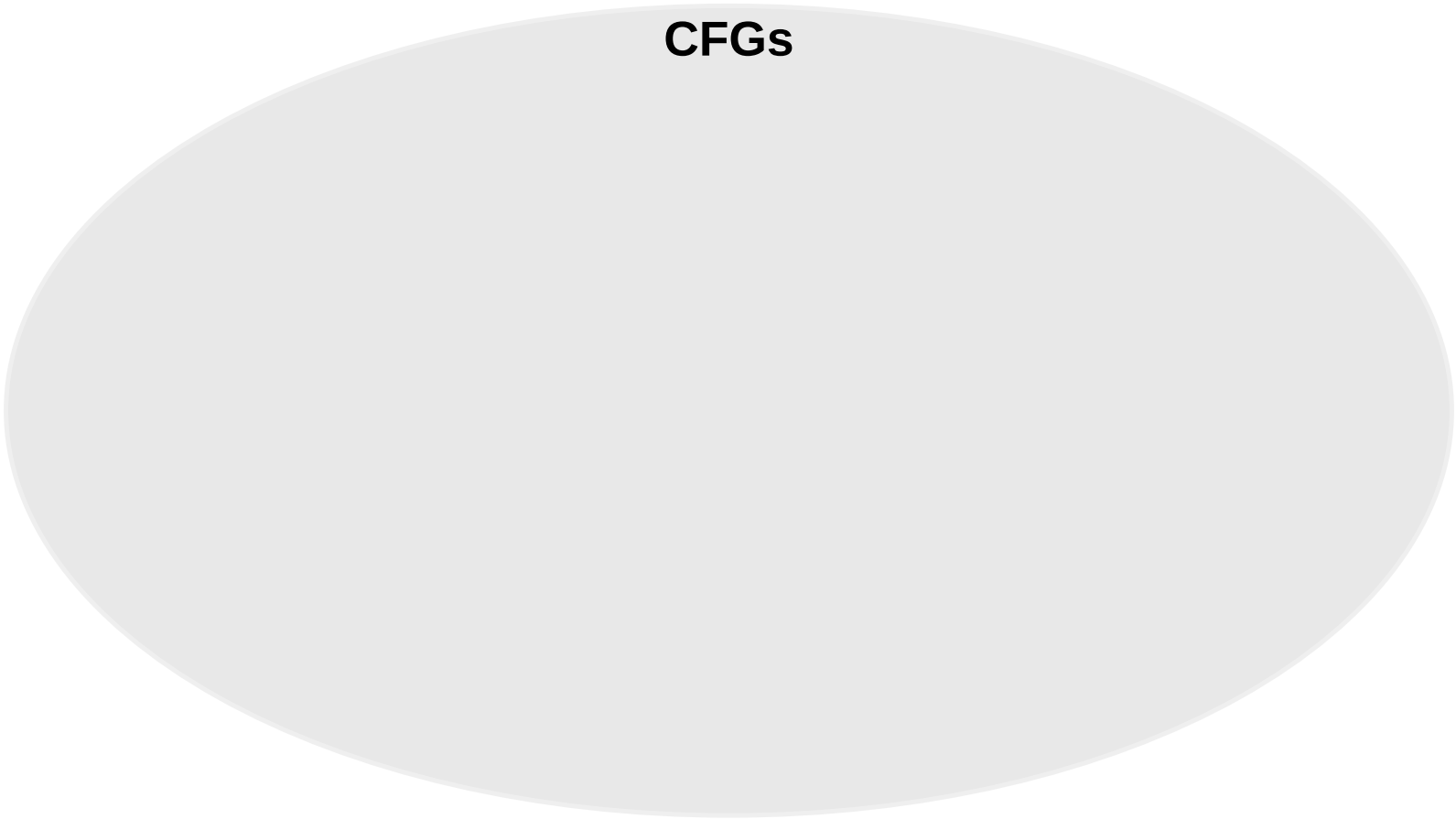
Software Development Team

2018-10-24

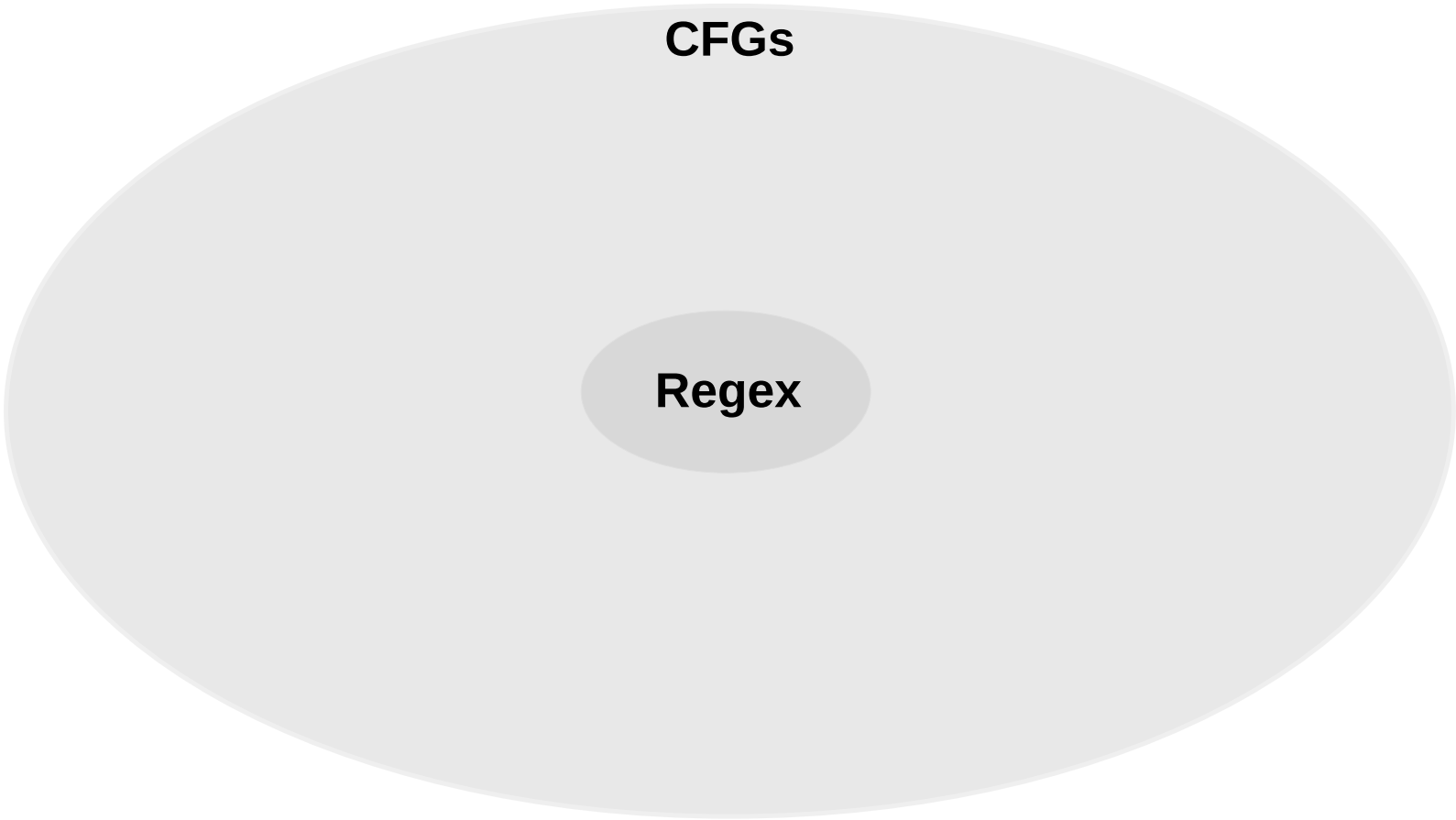
What to expect from this talk

Better, fewer syntax errors

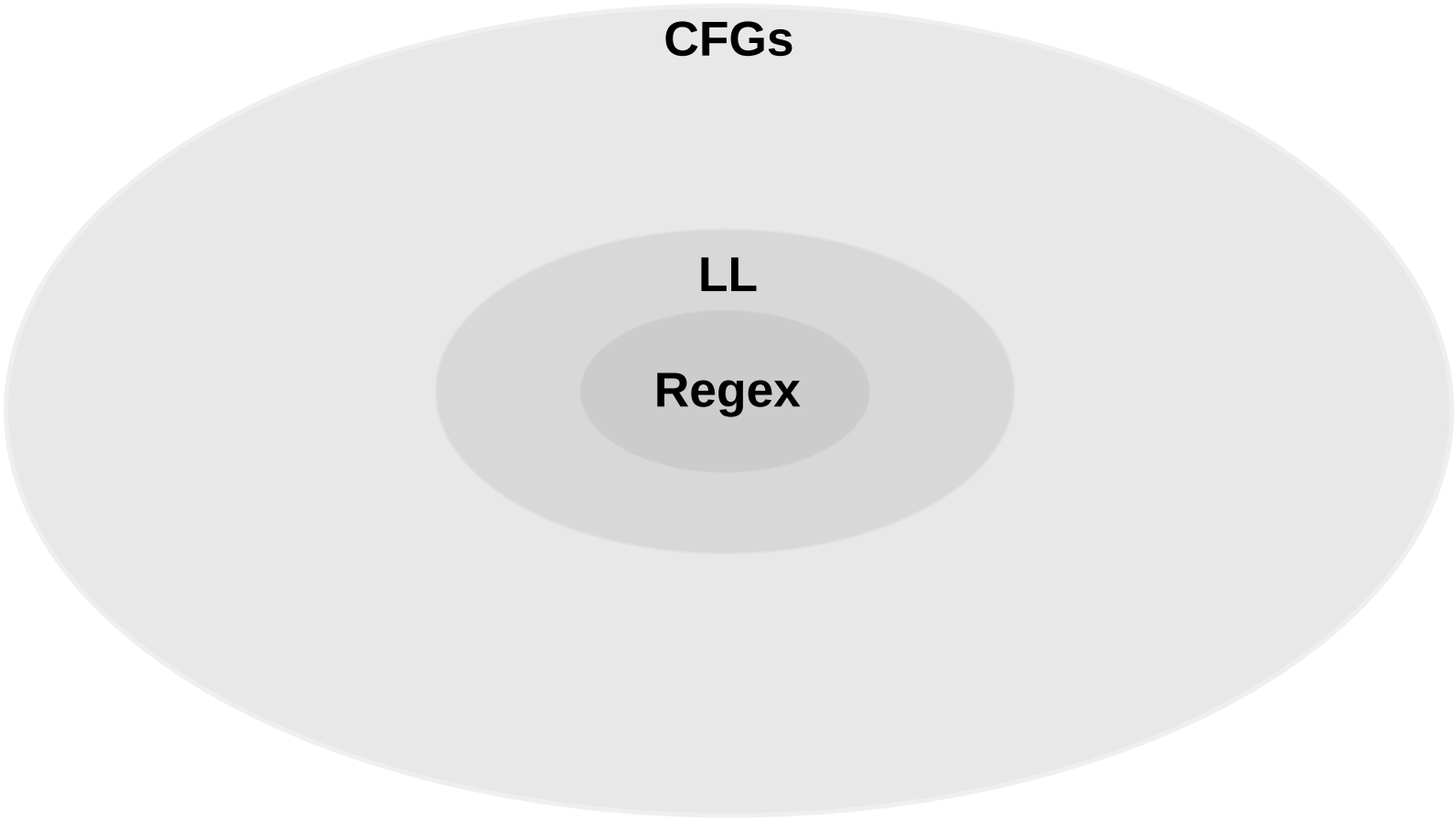
Context-Free Grammars (CFGs)



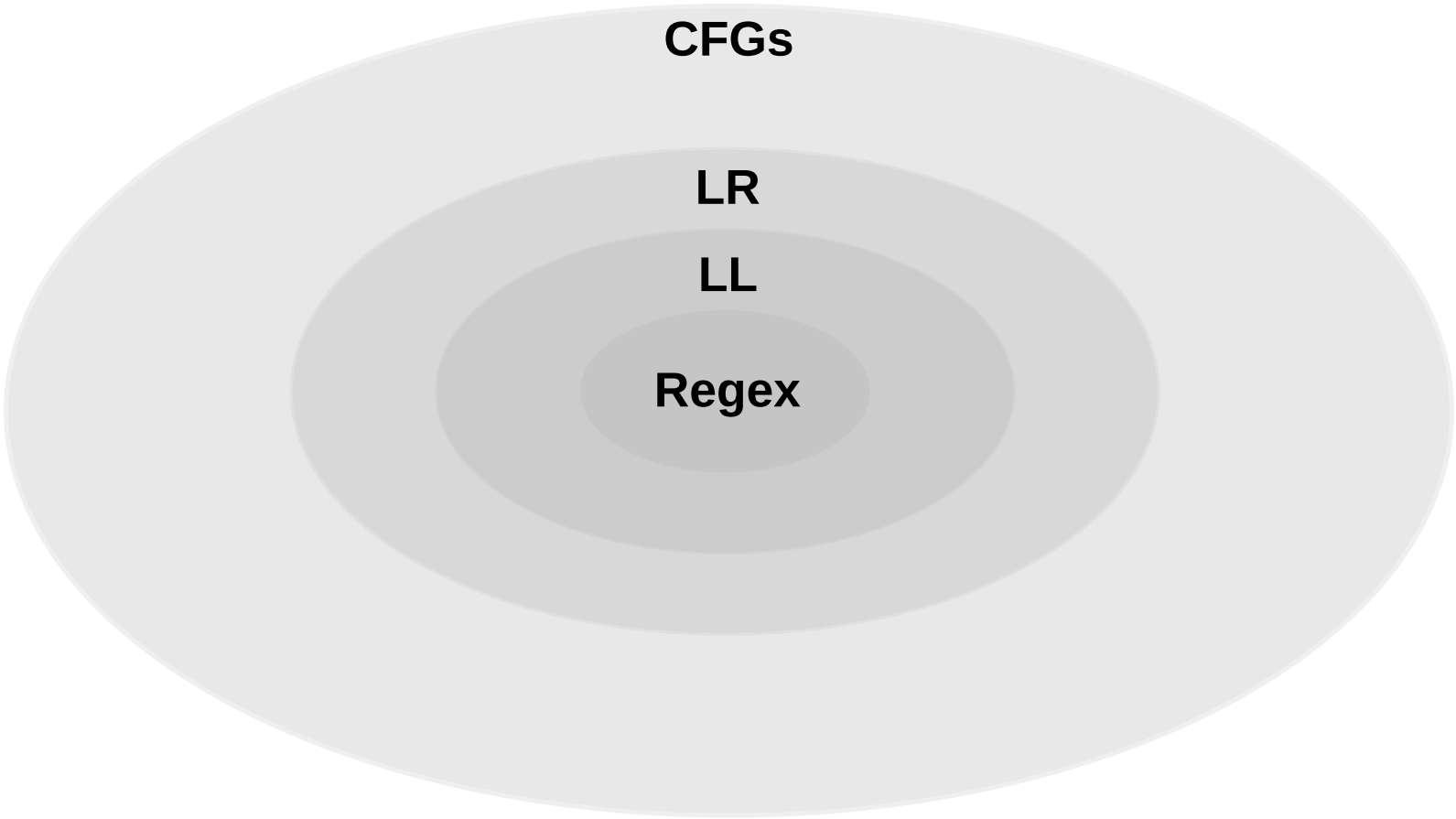
Context-Free Grammars (CFGs)



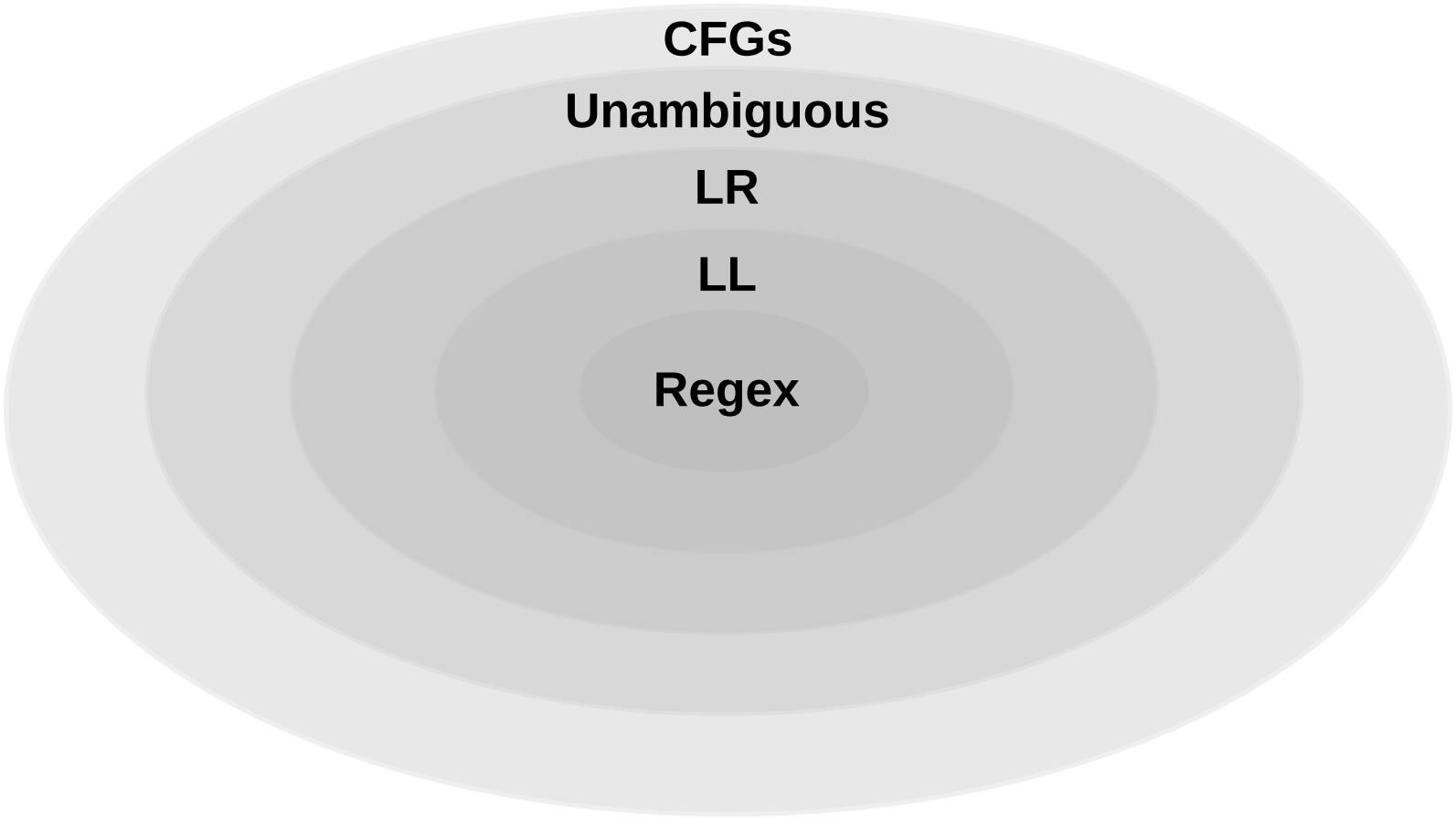
Context-Free Grammars (CFGs)



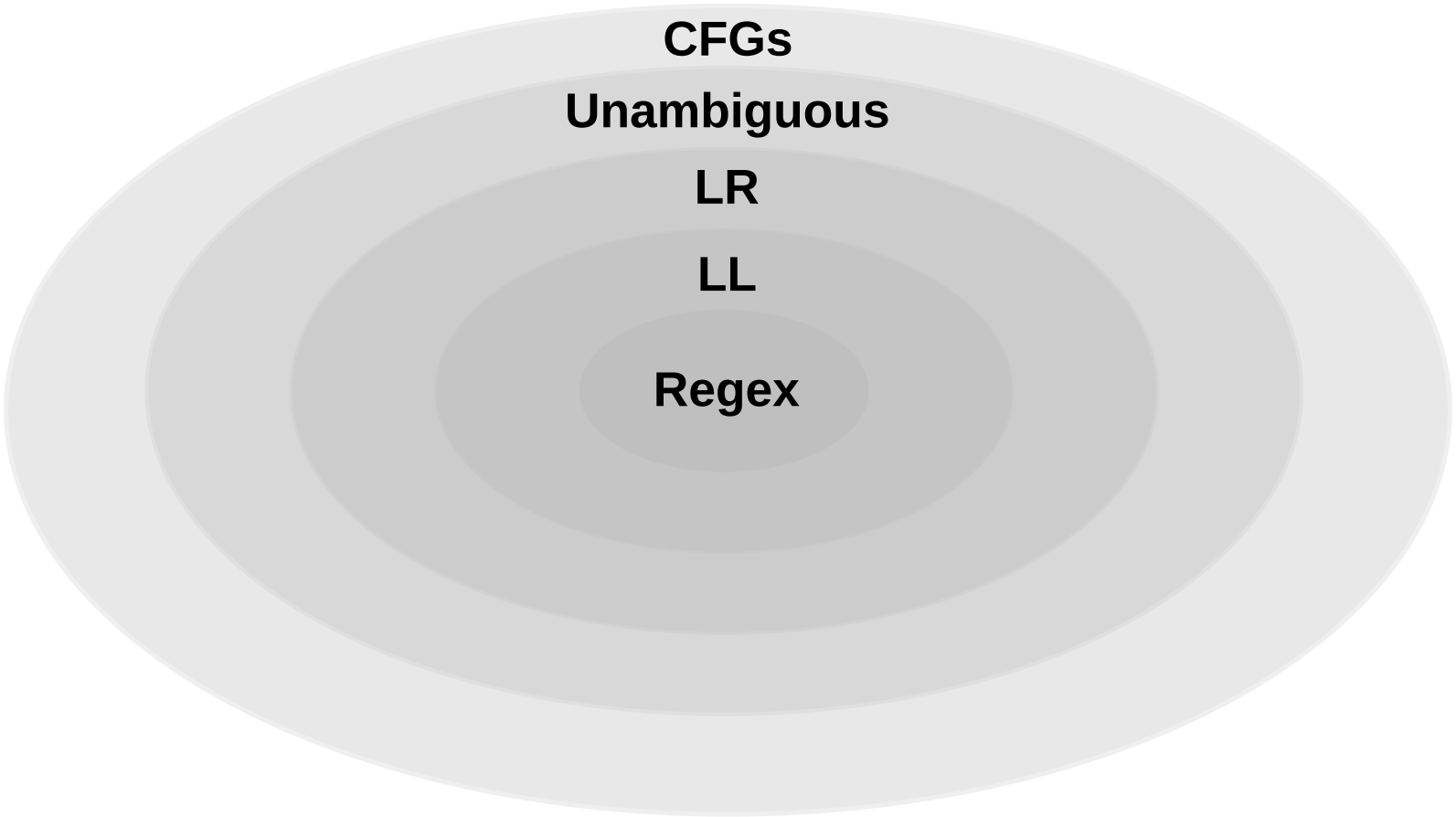
Context-Free Grammars (CFGs)



Context-Free Grammars (CFGs)

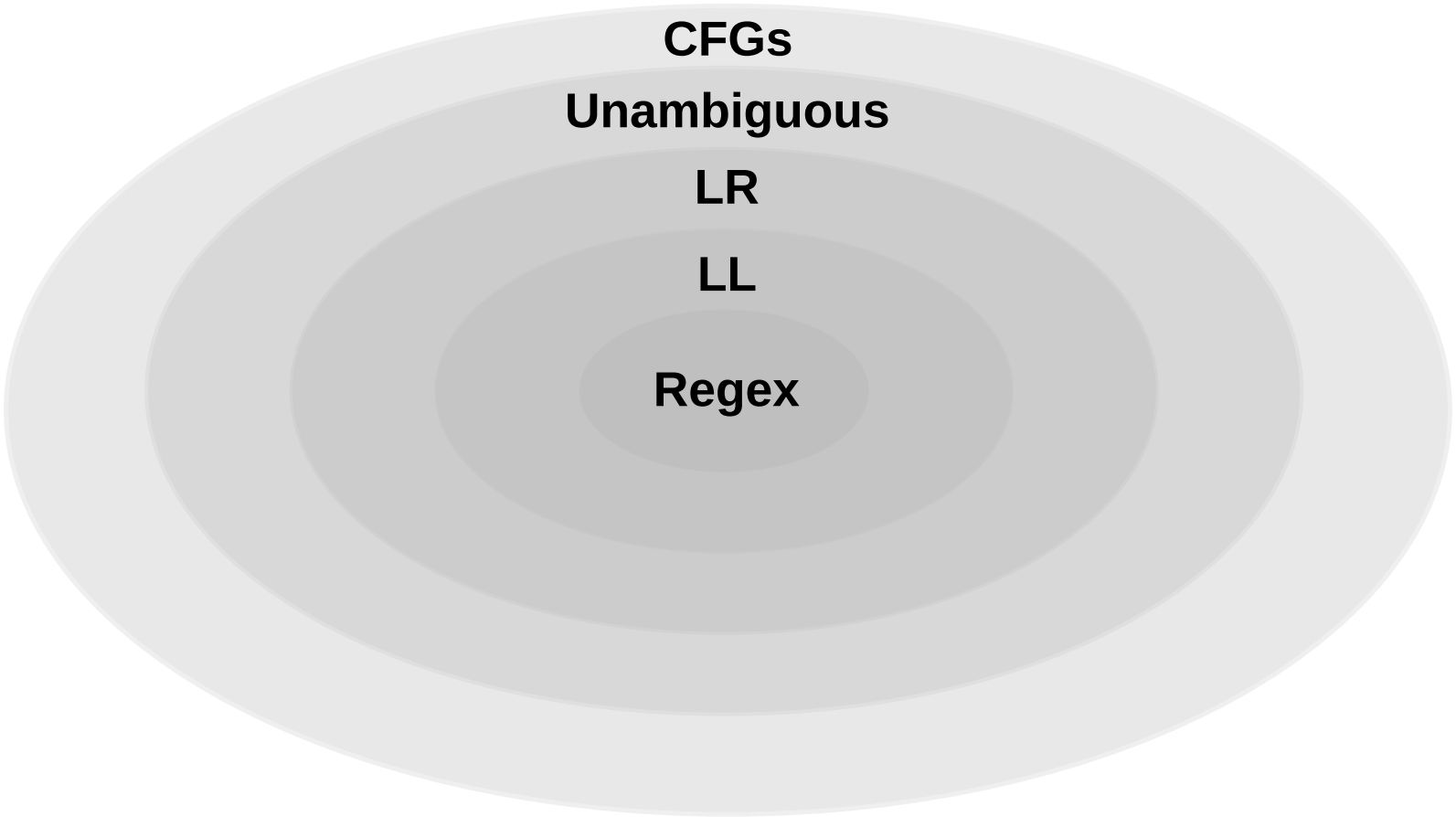


Context-Free Grammars (CFGs)



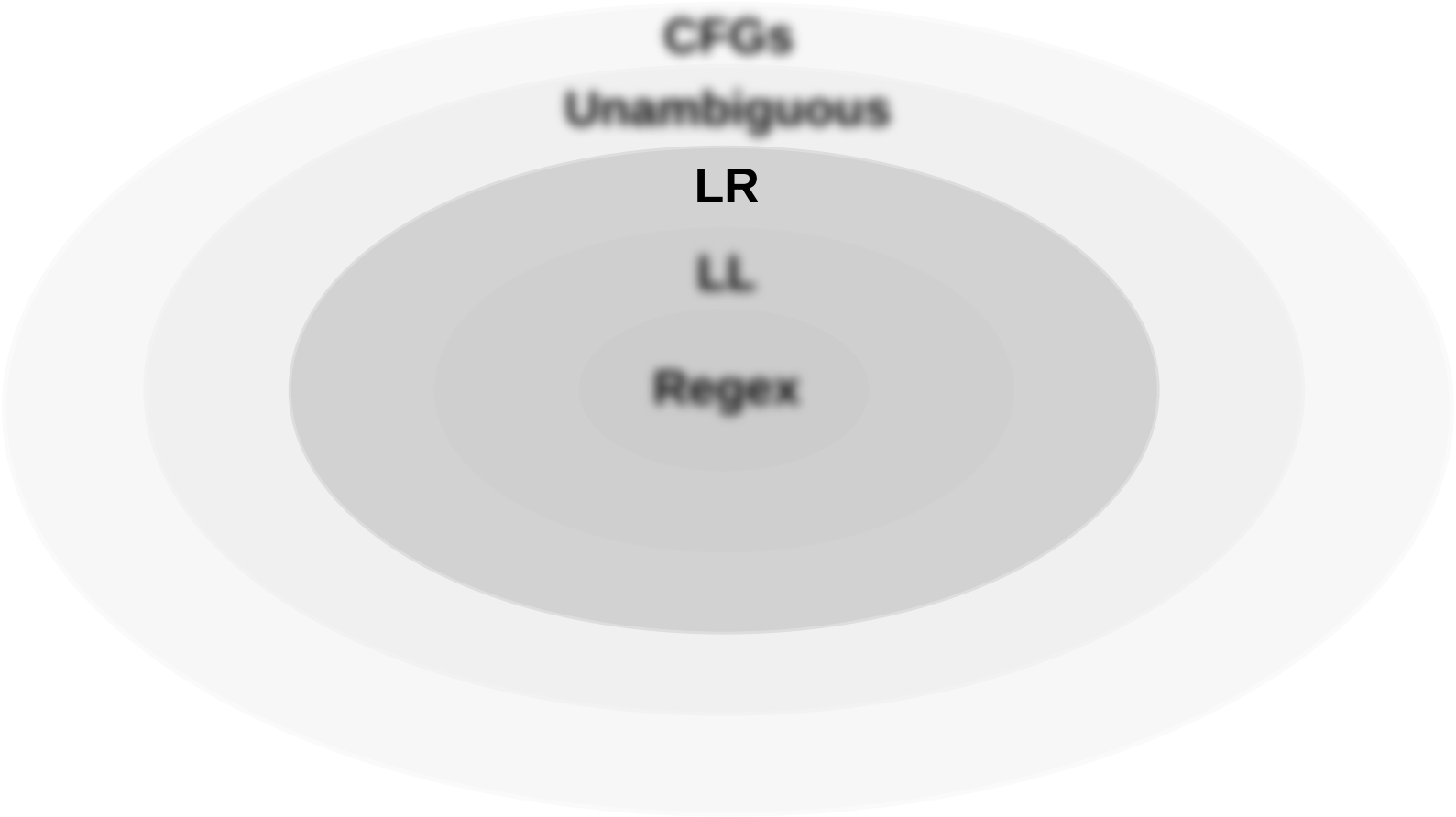
Hard to classify: ALL(*)

Context-Free Grammars (CFGs)



Hard to classify: ALL(*)
Impossible to classify: PEG, recursive descent.

Context-Free Grammars (CFGs)



LR refresher: terminology

```
%start Expr
%%
Expr: Term "+" Expr        // (I)
    | Term ;               // (II)

Term: Factor "*" Term      // (III)
    | Factor ;             // (IV)

Factor: "(" Expr ")"       // (V)
      | "INT" ;            // (VI)
```

LR refresher: terminology

```
%start Expr
%%
Expr: Term "+" Expr          // (I)
    | Term ;                 // (II)

Term: Factor "*" Term        // (III)
    | Factor ;               // (IV)

Factor: "(" Expr ")"         // (V)
      | "INT" ;              // (VI)
```

A grammar consists of *rules*

LR refresher: terminology

```
%start Expr
%%
Expr: Term "+" Expr        // (I)
    | Term ;               // (II)

Term: Factor "*" Term      // (III)
    | Factor ;             // (IV)

Factor: "(" Expr ")"       // (V)
      | "INT" ;            // (VI)
```

Rules have *names*

LR refresher: terminology

```
%start Expr
%%
Expr: Term "+" Expr          // (I)
    | Term ;                 // (II)

Term: Factor "*" Term        // (III)
    | Factor ;               // (IV)

Factor: "(" Expr ")"         // (V)
      | "INT" ;              // (VI)
```

Rules map to one or more *productions*

LR refresher: terminology

```
%start Expr
%%
Expr: Term "+" Expr          // (I)
    | Term ;                 // (II)

Term: Factor "*" Term        // (III)
    | Factor ;               // (IV)

Factor: "(" Expr ")"         // (V)
      | "INT" ;              // (VI)
```

Symbols reference either rules

LR refresher: terminology

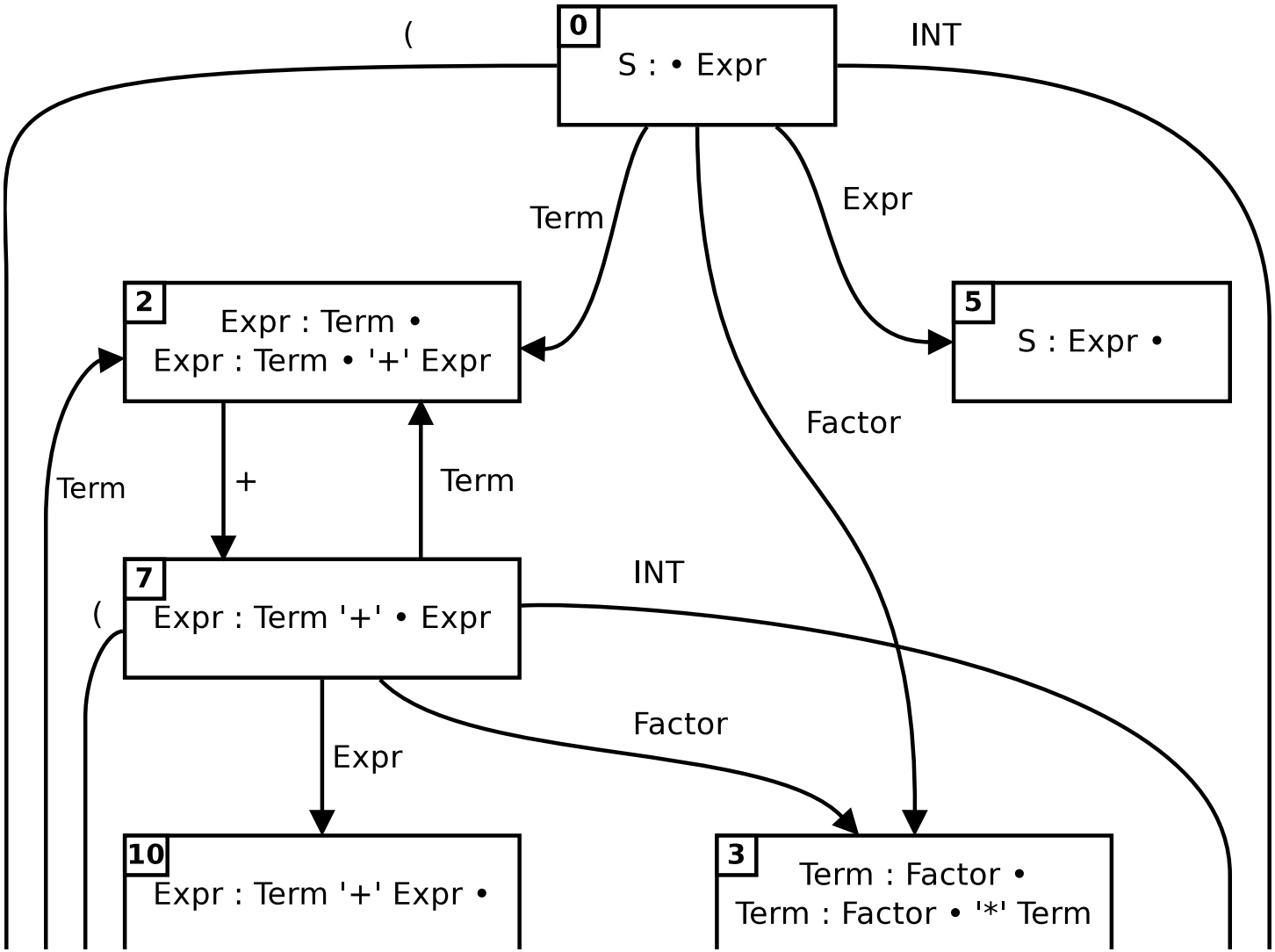
```
%start Expr
%%
Expr: Term "+" Expr        // (I)
    | Term ;               // (II)

Term: Factor "*" Term      // (III)
    | Factor ;             // (IV)

Factor: "(" Expr ")"       // (V)
      | "INT" ;            // (VI)
```

Symbols reference either rules or *tokens*

LR refresher: stategraph



LR refresher: actions and gotos tables

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

LR refresher: actions and gotos tables

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

'S(*i*)' means "consume the next token in the input and push state *i* onto the stack"

LR refresher: actions and gotos tables

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

'R(*i*)' means "reduce the production *i*: pop some elements from the stack; push a state *x* associated with *i*'s parent rule onto the stack"

LR refresher: actions and gotos tables

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

'Accept' indicates that the parse has completed successfully

LR refresher: actions and gotos tables

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

Empty cells indicate errors

The $\rightarrow_{\text{LR}}$ reduction relation

$$\frac{\text{action}(s_n, t_0) = \textit{shift}\ s'}{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow_{\text{LR}} ([s_0 \ldots s_n, s'], [t_1 \ldots t_n])}\ \text{LR Shift}$$

$$\frac{(\text{action}(s_n, t_0) = \textit{reduce}\ N\colon \alpha) \wedge (\text{goto}(s_{n-|\alpha|}, N) = s')}{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow_{\text{LR}} ([s_0 \ldots s_{n-|\alpha|}, s'], [t_0 \ldots t_n])}\ \text{LR Reduce}$$

An LR parse

Input: 2 * 3

An LR parse

Input: 2 * 3 $
Stack: [0]

An LR parse

Input: • 2 * 3 $
Stack: [0]

An LR parse

Input:	• 2 * 3 $
Stack:	[0]
Action:	

An LR parse

Input: • 2 * 3 $
Stack: [0]
Action:

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: • 2 * 3 $
Stack: [0]
Action: S(4)

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 • * 3 $
Stack: [0, 4]
Action:

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 • * 3 $
Stack: [0, 4]
Action: R(VI)

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 • * 3 $
Stack: [0, 4]
Action: R(VI)

```
%start Expr
%%
Expr: Term "+" Expr        // (I)
    | Term ;               // (II)

Term: Factor "*" Term      // (III)
    | Factor ;             // (IV)

Factor: "(" Expr ")"       // (V)
      | "INT" ;            // (VI)
```

An LR parse

Input: 2 • * 3 $
Stack: [0]
Action: R(VI)

```
%start Expr
%%
Expr: Term "+" Expr        // (I)
    | Term ;               // (II)

Term: Factor "*" Term      // (III)
    | Factor ;             // (IV)

Factor: "(" Expr ")"       // (V)
      | "INT" ;            // (VI)
```

An LR parse

Input: 2 • * 3 $
Stack: [0]
Action: goto(0, Factor)

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 • * 3 $
Stack: [0, 3]
Action:

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 • * 3 $
Stack: [0, 3]
Action: S(8)

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * • 3 $
Stack: [0, 3, 8]
Action:

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * • 3 $
Stack: [0, 3, 8]
Action: S(4)

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * 3 • $
Stack: [0, 3, 8, 4]
Action:

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * 3 • $
Stack: [0, 3, 8, 4]
Action: R(VI)

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * 3 • $
Stack: [0, 3, 8, 4]
Action: R(VI)

```
%start Expr
%%
Expr: Term "+" Expr        // (I)
    | Term ;               // (II)

Term: Factor "*" Term      // (III)
    | Factor ;             // (IV)

Factor: "(" Expr ")"       // (V)
      | "INT" ;            // (VI)
```

An LR parse

Input: 2 * 3 • $
Stack: [0, 3, 8]
Action: goto(8, Factor)

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * 3 • $
Stack: [0, 3, 8, 3]
Action:

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * 3 • $
Stack: [0, 3, 8, 3]
Action: R(IV)

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * 3 • $
Stack: [0, 3, 8, 3]
Action: R(IV)

```
%start Expr
%%
Expr: Term "+" Expr        // (I)
    | Term ;               // (II)

Term: Factor "*" Term      // (III)
    | Factor ;             // (IV)

Factor: "(" Expr ")"       // (V)
      | "INT" ;            // (VI)
```

An LR parse

Input: 2 * 3 • $
Stack: [0, 3, 8]
Action: goto(8, Term)

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * 3 • $
Stack: [0, 3, 8, 11]
Action:

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * 3 • $
Stack: [0, 3, 8, 11]
Action: R(III)

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * 3 • $
Stack: [0, 3, 8, 11]
Action: R(III)

```
%start Expr
%%
Expr: Term "+" Expr        // (I)
    | Term ;               // (II)

Term: Factor "*" Term      // (III)
    | Factor ;             // (IV)

Factor: "(" Expr ")"       // (V)
      | "INT" ;            // (VI)
```

An LR parse

Input: 2 * 3 • $
Stack: [0]
Action: goto(0, Term)

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * 3 • $
Stack: [0, 2]
Action:

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * 3 • $
Stack: [0, 2]
Action: R(II)

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * 3 • $
Stack: [0, 2]
Action: R(II)

```
%start Expr
%%
Expr: Term "+" Expr        // (I)
    | Term ;               // (II)

Term: Factor "*" Term      // (III)
    | Factor ;             // (IV)

Factor: "(" Expr ")"       // (V)
      | "INT" ;            // (VI)
```

An LR parse

Input: 2 * 3 • $
Stack: [0]
Action: goto(0, Expr)

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * 3 • $
Stack: [0, 5]
Action:

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

An LR parse

Input: 2 * 3 • $
Stack: [0, 5]
Action: Accept

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

Error recovery

LR parsers halt at the first point that an error has definitely been encountered.

Error recovery

LR parsers halt at the first point that an error has definitely been encountered.

What then?

Error recovery

LR parsers halt at the first point that an error has definitely been encountered.

What then?

Could tell the user the error location and then stop.

Error recovery

LR parsers halt at the first point that an error has definitely been encountered.

What then?

Could tell the user the error location and then stop.

Or try to *recover* (possibly reporting multiple errors).

Hard-coded error recovery algorithms

Most common choice: insert, or search for, a *synchronisation* token.

Hard-coded error recovery algorithms

Most common choice: insert, or search for, a *synchronisation* token.

e.g. ‘;’ is an obvious choice for Java.

Hard-coded error recovery algorithms

Most common choice: insert, or search for, a *synchronisation* token.

e.g. ‘;’ is an obvious choice for Java.

Problems: language specific; skips lots of input; tends to lead to a cascade of parsing errors.

Holub's panic mode algorithm: outline

Simplest generic panic mode algorithm

Holub's panic mode algorithm: outline

Simplest generic panic mode algorithm

On error: search parsing stack for a state that can parse the next input symbol. If no such state is found, skip the input symbol and repeat.

Holub's panic mode algorithm: formal(ish)

```
def holub(pstack, toks):
  while len(toks) > 0:
    npstack = pstack[:]
    while len(npstack) > 0:
      if action(npstack[-1], toks[0]) != error:
        return (npstack, toks)
      npstack.pop()
    del toks[0]
  return None
```

Panic mode: Holub's algorithm (example)

Input '2 + + 3': error encountered with '+ 3 $' left, and the stack at [0, 2, 7].

Panic mode: Holub's algorithm (example)

Input '2 + + 3': error encountered with '+ 3 $' left, and the stack at [0, 2, 7].

	Actions						Gotos		
s	INT	+	*	(	)	$	Term	Factor	Expr
0	S(4)			S(1)			2	3	5
1	S(4)			S(1)			2	3	6
2		S(7)			R(II)	R(II)			
3		R(IV)	S(8)		R(IV)	R(IV)			
4		R(VI)	R(VI)		R(VI)	R(VI)			
5						Accept			
6					S(9)				
7	S(4)			S(1)			2	3	10
8	S(4)			S(1)			11	3	
9		R(V)	R(V)		R(V)	R(V)			
10					R(I)	R(I)			
11		R(III)			R(III)	R(III)			

Panic mode: Holub's algorithm (example)

Input '2 + + 3': error encountered with '+ 3 $' left, and the stack at [0, 2, 7].

Set the stack to [0, 2]. Success!

Panic mode: Holub's algorithm (example)

Input '2 + + 3': error encountered with '+ 3 $' left, and the stack at [0, 2, 7].

Set the stack to [0, 2]. Success!

Deus ex machina recovery: can't be reported to, or emulated by, users.

Panic mode: Holub's algorithm (example)

Input '2 + + 3': error encountered with '+ 3 $' left, and the stack at [0, 2, 7].

Set the stack to [0, 2]. Success!

Deus ex machina recovery: can't be reported to, or emulated by, users.

Tends to lead to a cascade of parsing errors.

The Fischer *et al.* family of error recovery algorithms

Lots of advanced error recovery algorithms.

The Fischer *et al.* family of error recovery algorithms

Lots of advanced error recovery algorithms. We'll be looking at those that trace back to:

A Locally Least-Cost LR-Error Corrector.
C. N. Fischer, B. A. Dion, and J. Mauney. Technical Report 363. University of Wisconsin, 1979.

Our first algorithm *CPCT*+ fixes and extends:

Repairing syntax errors in LR parsers
Rafael Corchuelo, José Antonio Pérez, Antonio Ruiz-Cortés, and Miguel Toro TOPLAS 24 (Nov 2002)

Recovery consists of a *repair sequence*.

Corchuelo *et al.* basics

Recovery consists of a *repair sequence*. Repairs are:

- *insert T:* insert a token of type *T*.

Corchuelo *et al.* basics

Recovery consists of a *repair sequence*. Repairs are:

- *insert T:* insert a token of type *T*.
- *delete:* delete the token at the current index.

Corchuelo *et al.* basics

Recovery consists of a *repair sequence*. Repairs are:

- *insert T:* insert a token of type *T*.
- *delete:* delete the token at the current index.
- *shift:* parse the token at the current index.

Error recovery completes:

- Successfully if: an *accept* state is reached; or 'enough' tokens are shifted.

Error recovery completes:

- Successfully if: an *accept* state is reached; or 'enough' tokens are shifted.
- Unsuccessfully if: 'too many' tokens are deleted or shifted.

Error recovery completes:

- Successfully if: an *accept* state is reached; or 'enough' tokens are shifted.
- Unsuccessfully if: 'too many' tokens are deleted or shifted.

If completed successfully, *apply* the chosen repair sequence and resume parsing.

Corchuelo *et al.* basics

- Search using *configurations*: *(pstack, tokens, repair seq)* triples.
- Given a configuration, we can generate all its neighbours.

The $\rightarrow_{\text{CR}}$ reduction relation

$$\frac{t_0 \neq \$}{([s_0 \ldots s_n], [t_0, t_1 \ldots t_n]) \rightarrow_{\text{CR}} ([s_0 \ldots s_n], [t_1 \ldots t_n], [delete])}$$

CR DELETE

The $\rightarrow_{\text{CR}}$ reduction relation

$$\frac{t_0 \neq \$}{([s_0 \ldots s_n], [t_0, t_1 \ldots t_n]) \rightarrow_{\text{CR}} ([s_0 \ldots s_n], [t_1 \ldots t_n], [\mathit{delete}])}$$

CR DELETE

$$\frac{\text{action}(s_n, t) \neq \mathit{error} \wedge t \neq \$ \wedge ([s_0 \ldots s_n], [t, t_0 \ldots t_n]) \rightarrow^*_{\text{LR}} ([s'_0 \ldots s'_m], [t_0 \ldots t_n])}{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow_{\text{CR}} ([s'_0 \ldots s'_m], [t_0 \ldots t_n], [\mathit{insert}\ t])}$$

CR INSERT

The $\rightarrow_{\text{CR}}$ reduction relation

$$\frac{t_0 \neq \$}{([s_0 \ldots s_n], [t_0, t_1 \ldots t_n]) \rightarrow_{\text{CR}} ([s_0 \ldots s_n], [t_1 \ldots t_n], [\mathit{delete}])}$$

CR DELETE

$$\frac{\text{action}(s_n, t) \neq \mathit{error} \wedge t \neq \$ \wedge ([s_0 \ldots s_n], [t, t_0 \ldots t_n]) \rightarrow^*_{\text{LR}} ([s'_0 \ldots s'_m], [t_0 \ldots t_n])}{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow_{\text{CR}} ([s'_0 \ldots s'_m], [t_0 \ldots t_n], [\mathit{insert}\ t])}$$

CR INSERT

$$\frac{\begin{array}{c}([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow^*_{\text{LR}} ([s'_0 \ldots s'_m], [t_j \ldots t_n]) \wedge 0 < j \leq N_{\mathit{shifts}} \\ j = N_{\mathit{shifts}} \vee \text{action}(s'_m, t_j) \in \{\mathit{accept}, \mathit{error}\}\end{array}}{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow_{\text{CR}} ([s'_0 \ldots s'_m], [t_j \ldots t_n], [\underbrace{\mathit{shift} \ldots \mathit{shift}}_{j}])}$$

CR SHIFT 1

Corchuelo *et al.* algorithm

```
 1 def corchueloetal(pstack, toks):
 2   todo = [[(pstack, toks, [])]]
 3   cur_cst = 0
 4   while cur_cst < len(todo):
 5     if len(todo[cur_cst]) == 0:
 6       cur_cst += 1
 7       continue
 8     n = todo[cur_cst].pop()
 9     if action(n[0][-1], n[1][0]) == accept \
10        or ends_in_N_shifts(n[2]):
11       return n
12     elif len(n[1]) - len(toks) == N_total: continue
13     for nbr in all_cr_star(n[0], n[1]):
14       if len(n[2]) > 0 and n[2][-1] == delete \
15          and nbr[2][-1] == insert:
16         continue
17       cst = cur_cst + rprs_cst(nbr[2])
18       for _ in range(len(todo), cst): todo.push([])
19       todo[cst].append((nbr[0], nbr[1], n[2] + nbr[2]))
20   return None
```

Corchuelo *et al.* algorithm

```
 1 def corchueloetal(pstack, toks):
 2   todo = [[(pstack, toks, [])]]
 3   cur_cst = 0
 4   while cur_cst < len(todo):
 5     if len(todo[cur_cst]) == 0:
 6       cur_cst += 1
 7       continue
 8     n = todo[cur_cst].pop()
 9     if action(n[0][-1], n[1][0]) == accept \
10        or ends_in_N_shifts(n[2]):
11       return n
12     elif len(n[1]) - len(toks) == N_total: continue
13     for nbr in all_cr_star(n[0], n[1]):
14       if len(n[2]) > 0 and n[2][-1] == delete \
15          and nbr[2][-1] == insert:
16         continue
17       cst = cur_cst + rprs_cst(nbr[2])
18       for _ in range(len(todo), cst): todo.push([])
19       todo[cst].append((nbr[0], nbr[1], n[2] + nbr[2]))
20   return None
```

Todo list: lists of (pstack, tokens, repair seq) triples

Corchuelo *et al.* algorithm

```
1  def corchueloetal(pstack, toks):
2    todo = [[(pstack, toks, [])]]
3    cur_cst = 0
4    while cur_cst < len(todo):
5      if len(todo[cur_cst]) == 0:
6        cur_cst += 1
7        continue
8      n = todo[cur_cst].pop()
9      if action(n[0][-1], n[1][0]) == accept \
10        or ends_in_N_shifts(n[2]):
11       return n
12     elif len(n[1]) - len(toks) == N_total: continue
13     for nbr in all_cr_star(n[0], n[1]):
14       if len(n[2]) > 0 and n[2][-1] == delete \
15          and nbr[2][-1] == insert:
16         continue
17       cst = cur_cst + rprs_cst(nbr[2])
18       for _ in range(len(todo), cst): todo.push([])
19       todo[cst].append((nbr[0], nbr[1], n[2] + nbr[2]))
20   return None
```

Pop a lowest cost configuration

Corchuelo *et al.* algorithm

```
1  def corchueloetal(pstack, toks):
2    todo = [[(pstack, toks, [])]]
3    cur_cst = 0
4    while cur_cst < len(todo):
5      if len(todo[cur_cst]) == 0:
6        cur_cst += 1
7        continue
8      n = todo[cur_cst].pop()
9      if action(n[0][-1], n[1][0]) == accept \
10        or ends_in_N_shifts(n[2]):
11       return n
12     elif len(n[1]) - len(toks) == N_total: continue
13     for nbr in all_cr_star(n[0], n[1]):
14       if len(n[2]) > 0 and n[2][-1] == delete \
15          and nbr[2][-1] == insert:
16         continue
17       cst = cur_cst + rprs_cst(nbr[2])
18       for _ in range(len(todo), cst): todo.push([])
19       todo[cst].append((nbr[0], nbr[1], n[2] + nbr[2]))
20   return None
```

If the configuration reaches accept or shifts ‘enough’ tokens: success!

Corchuelo *et al.* algorithm

```
 1 def corchueloetal(pstack, toks):
 2   todo = [[(pstack, toks, [])]]
 3   cur_cst = 0
 4   while cur_cst < len(todo):
 5     if len(todo[cur_cst]) == 0:
 6       cur_cst += 1
 7       continue
 8     n = todo[cur_cst].pop()
 9     if action(n[0][-1], n[1][0]) == accept \
10        or ends_in_N_shifts(n[2]):
11       return n
12     elif len(n[1]) - len(toks) == N_total: continue
13     for nbr in all_cr_star(n[0], n[1]):
14       if len(n[2]) > 0 and n[2][-1] == delete \
15          and nbr[2][-1] == insert:
16         continue
17       cst = cur_cst + rprs_cst(nbr[2])
18       for _ in range(len(todo), cst): todo.push([])
19       todo[cst].append((nbr[0], nbr[1], n[2] + nbr[2]))
20   return None
```

If the configuration has consumed too many tokens: discard

Corchuelo *et al.* algorithm

```
 1 def corchueloetal(pstack, toks):
 2   todo = [[(pstack, toks, [])]]
 3   cur_cst = 0
 4   while cur_cst < len(todo):
 5     if len(todo[cur_cst]) == 0:
 6       cur_cst += 1
 7       continue
 8     n = todo[cur_cst].pop()
 9     if action(n[0][-1], n[1][0]) == accept \
10        or ends_in_N_shifts(n[2]):
11       return n
12     elif len(n[1]) - len(toks) == N_total: continue
13     for nbr in all_cr_star(n[0], n[1]):
14       if len(n[2]) > 0 and n[2][-1] == delete \
15          and nbr[2][-1] == insert:
16         continue
17       cst = cur_cst + rprs_cst(nbr[2])
18       for _ in range(len(todo), cst): todo.push([])
19       todo[cst].append((nbr[0], nbr[1], n[2] + nbr[2]))
20   return None
```

Gather neighbours

Corchuelo *et al.* algorithm

```
 1 def corchueloetal(pstack, toks):
 2   todo = [[(pstack, toks, [])]]
 3   cur_cst = 0
 4   while cur_cst < len(todo):
 5     if len(todo[cur_cst]) == 0:
 6       cur_cst += 1
 7       continue
 8     n = todo[cur_cst].pop()
 9     if action(n[0][-1], n[1][0]) == accept \
10        or ends_in_N_shifts(n[2]):
11       return n
12     elif len(n[1]) - len(toks) == N_total: continue
13     for nbr in all_cr_star(n[0], n[1]):
14       if len(n[2]) > 0 and n[2][-1] == delete \
15          and nbr[2][-1] == insert:
16         continue
17       cst = cur_cst + rprs_cst(nbr[2])
18       for _ in range(len(todo), cst): todo.push([])
19       todo[cst].append((nbr[0], nbr[1], n[2] + nbr[2]))
20   return None
```

Error recovery failed

Corchuelo *et al.* problem

$$\frac{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow^*_{\mathrm{LR}} ([s'_0 \ldots s'_m], [t_j \ldots t_n]) \wedge 0 < j \leq N_{shifts} \quad j = N_{shifts} \vee \mathrm{action}(s'_m, t_j) \in \{accept, error\}}{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow_{\mathrm{CR}} ([s'_0 \ldots s'_m], [t_j \ldots t_n], [\underbrace{shift \ldots shift}_{j}])}$$

CR SHIFT 1

Corchuelo *et al.* problem

$$\frac{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow^*_{\mathrm{LR}} ([s'_0 \ldots s'_m], [t_j \ldots t_n]) \wedge 0 < j \leq N_{shifts} \quad j = N_{shifts} \vee \mathrm{action}(s'_m, t_j) \in \{accept, error\}}{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow_{\mathrm{CR}} ([s'_0 \ldots s'_m], [t_j \ldots t_n], [\underbrace{shift \ldots shift}_{j}])}$$

CR SHIFT 1

Corchuelo *et al.* fix (1)

$$\frac{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow^*_{\mathrm{LR}} ([s'_0 \ldots s'_m], [t_j \ldots t_n]) \wedge 0 \leq j \leq N_{shifts} \quad j = N_{shifts} \vee \mathrm{action}(s'_m, t_j) \in \{accept, error\}}{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow_{\mathrm{CR}} ([s'_0 \ldots s'_m], [t_j \ldots t_n], [\underbrace{shift \ldots shift}_{j}])}$$

CR SHIFT 1.5

Corchuelo *et al.* fix (1)

$$\frac{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow^*_{\mathrm{LR}} ([s'_0 \ldots s'_m], [t_j \ldots t_n]) \wedge 0 \leq j \leq N_{shifts} \quad (j = 0 \wedge [s_0 \ldots s_n] \neq [s'_0 \ldots s'_m]) \vee j = N_{shifts} \vee \mathrm{action}(s'_m, t_j) \in \{accept, error\}}{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow_{\mathrm{CR}} ([s'_0 \ldots s'_m], [t_j \ldots t_n], [\underbrace{shift \ldots shift}_{j}])}$$

CR SHIFT 2

CR SHIFT 2

For the input ‘2 3 +‘ an error is found at ‘3’:

- CR SHIFT 1 finds no repair sequences

CR Shift 2

For the input ‘2 3 +‘ an error is found at ‘3’:

- CR Shift 1 finds no repair sequences
- CR Shift 2 finds:

```
Delete "3", Delete "+"
Delete "3", Shift "+", Insert "INT"
Insert "PLUS", Shift "3", Shift "+", Insert "INT"
Insert "MULT", Shift "3", Shift "+", Insert "INT"
```

Corchuelo *et al.* fix (2)

$$\frac{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow^*_{\mathrm{LR}} ([s'_0 \ldots s'_m], [t_j \ldots t_n]) \wedge 0 \leq j \leq N_{shifts} \quad (j = 0 \wedge [s_0 \ldots s_n] \neq [s'_0 \ldots s'_m]) \vee j = N_{shifts} \vee \mathrm{action}(s'_m, t_j) \in \{accept, error\}}{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow_{\mathrm{CR}} ([s'_0 \ldots s'_m], [t_j \ldots t_n], [\underbrace{shift \ldots shift}_{j}])}$$

CR SHIFT 2

Corchuelo *et al.* fix (2)

$$\frac{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow^*_{\mathrm{LR}} ([s'_0 \ldots s'_m], [t_j \ldots t_n]) \wedge 0 \leq j \leq 1 \quad (j = 0 \wedge [s_0 \ldots s_n] \neq [s'_0 \ldots s'_m] \wedge R = [\,]) \vee (j = 1 \wedge R = [\mathit{shift}])}{([s_0 \ldots s_n], [t_0 \ldots t_n]) \rightarrow_{\mathrm{CR}} ([s'_0 \ldots s'_m], [t_j \ldots t_n], R]}$$

CR SHIFT 3

CR Shift 3

For the input ‘2 3 +‘ an error is found at ‘3’:

- CR Shift 1 finds no repair sequences
- CR Shift 2 finds:

```
Delete "3", Delete "+"
Delete "3", Shift "+", Insert "INT"
Insert "PLUS", Shift "3", Shift "+", Insert "INT"
Insert "MULT", Shift "3", Shift "+", Insert "INT"
```

CR Shift 3

For the input ‘2 3 +‘ an error is found at ‘3’:

- CR Shift 1 finds no repair sequences
- CR Shift 2 finds:

```
Delete "3", Delete "+"
Delete "3", Shift "+", Insert "INT"
Insert "PLUS", Shift "3", Shift "+", Insert "INT"
Insert "MULT", Shift "3", Shift "+", Insert "INT"
```

- CR Shift 3 finds:

```
Delete "3", Delete "+"
Delete "3", Shift "+", Insert "INT"
Insert "PLUS", Shift "3", Shift "+", Insert "INT"
Insert "MULT", Shift "3", Shift "+", Insert "INT"
Insert "MULT", Shift "3", Delete "+"
Insert "PLUS", Shift "3", Delete "+"
```

Implementation considerations (1)

Dijkstra’s algorithm obvious choice for search.

Implementation considerations (1)

Dijkstra's algorithm obvious choice for search.

How to represent the queue?

Implementation considerations (1)

Dijkstra's algorithm obvious choice for search.

How to represent the queue?

Some factors in our favour:

Implementation considerations (1)

Dijkstra's algorithm obvious choice for search.

How to represent the queue?

Some factors in our favour:

- Every configuration has a cost *c*.

Implementation considerations (1)

Dijkstra's algorithm obvious choice for search.

How to represent the queue?

Some factors in our favour:

- Every configuration has a cost *c*.
- Costs monotonically increase over time.

Implementation considerations (1)

Dijkstra's algorithm obvious choice for search.

How to represent the queue?

Some factors in our favour:

- Every configuration has a cost *c*.
- Costs monotonically increase over time.
- A configuration's generated neighbours always cost the same or more.

Implementation considerations (1)

Dijkstra's algorithm obvious choice for search.

How to represent the queue?

Some factors in our favour:

- Every configuration has a cost *c*.
- Costs monotonically increase over time.
- A configuration's generated neighbours always cost the same or more.
- Costs rarely rise into double digits.

The queue

Solution similar to Cerecke:

- The simplest solution works: the queue is a list of lists.

The queue

Solution similar to Cerecke:

- The simplest solution works: the queue is a list of lists.
- A sub-list at index *i* represents a cost *i*.

The queue

Solution similar to Cerecke:

- The simplest solution works: the queue is a list of lists.
- A sub-list at index *i* represents a cost *i*.
- O(1) insert (i.e. `push`) and O(1) take (i.e. `pop`).

Implementation considerations (2)

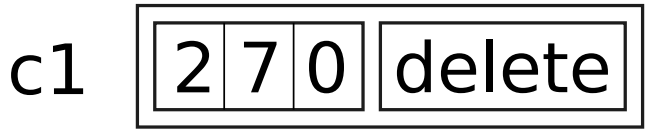
Can store vast numbers of configurations: memory is a concern.

Implementation considerations (2)

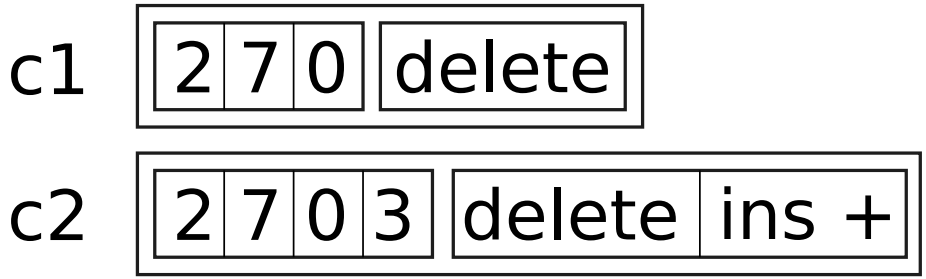
Can store vast numbers of configurations: memory is a concern.

Problem: configurations are variable size (due to parsing stacks and repair sequences).

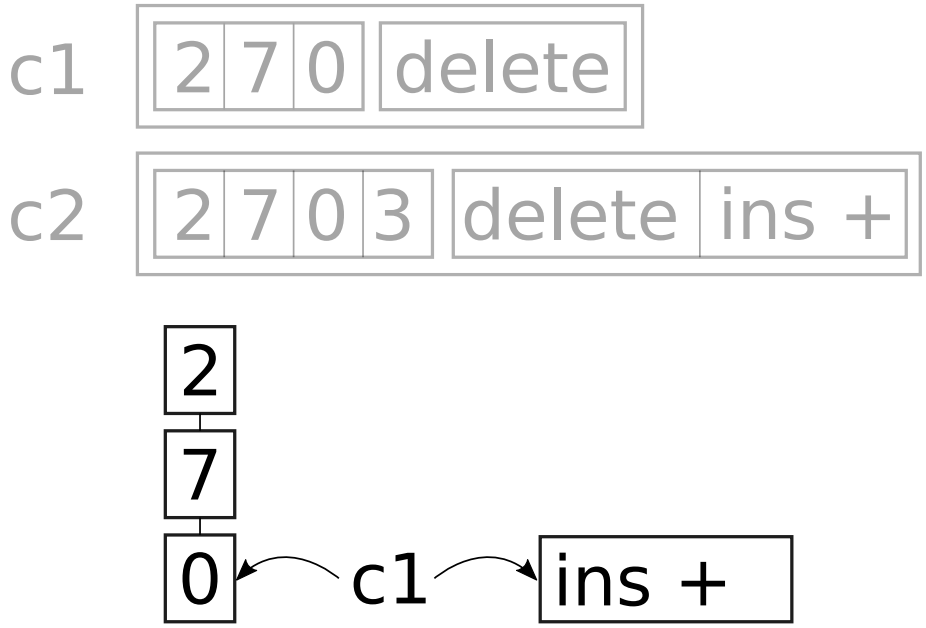
Parent-pointer trees



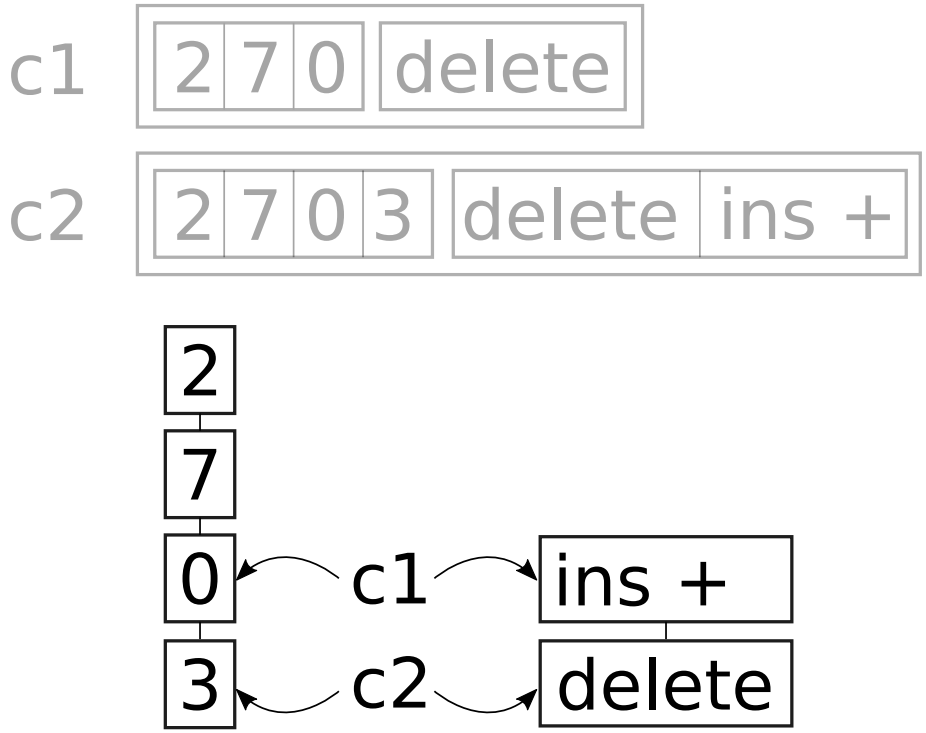
Parent-pointer trees



Parent-pointer trees



Parent-pointer trees



From Corchuelo *et al.* to *CPCT*+

Non-deterministically presenting one repair sequence to users is annoying.

From Corchuelo *et al.* to *CPCT*+

Non-deterministically presenting one repair sequence to users is annoying.

Presenting *all* minimum cost repair sequences is useful.

From Corchuelo *et al.* to *CPCT*$^+$

Non-deterministically presenting one repair sequence to users is annoying.

Presenting *all* minimum cost repair sequences is useful.

Trivial addition to the algorithm, but imposes a massive slowdown.

Merging compatible configurations

Solution: merge *compatible* configurations.

Merging compatible configurations

Solution: merge *compatible* configurations. Two configurations are compatible if:

- their parsing stacks are identical
- they both have an identical amount of input remaining
- their repair sequences are compatible

Merging compatible configurations

Solution: merge *compatible* configurations. Two configurations are compatible if:

- their parsing stacks are identical
- they both have an identical amount of input remaining
- their repair sequences are compatible

Two repair sequences are compatible:

- if they both end in the same number ($n \geq 0$) of shifts
- if one repair sequence ends in a delete, the other repair sequence also ends in a delete

Implementing configuration merging

How to find compatible configurations quickly?

Implementing configuration merging

How to find compatible configurations quickly?

1. Change the queue from list-of-lists to list-of-hashsets.

Implementing configuration merging

How to find compatible configurations quickly?

1. Change the queue from list-of-lists to list-of-hashsets.
2. Hash configurations with only parsing stack and remaining input; configuration equality uses parsing stack, remaining input, and repair sequence.

Implementing configuration merging

How to find compatible configurations quickly?

1. Change the queue from list-of-lists to list-of-hashsets.
2. Hash configurations with only parsing stack and remaining input; configuration equality uses parsing stack, remaining input, and repair sequence.
3. Then use parent-pointer trees to represent merged configurations.

Implementing configuration merging

How to find compatible configurations quickly?

1. Change the queue from list-of-lists to list-of-hashsets.
2. Hash configurations with only parsing stack and remaining input; configuration equality uses parsing stack, remaining input, and repair sequence.
3. Then use parent-pointer trees to represent merged configurations.

Won't find compatible configurations which are already processed; but finds enough to be a major saving.

Ranking repair sequences

```
class C {
    T x = 2 +
    T y = 3;
}
```

Ranking repair sequences

```
class C {
    T x = 2 +
    T y = 3;
}
```

Leads to

```
Error at line 3 col 7. Repairs found:
  Insert "COMMA"
```

Ranking repair sequences

```
class C {
    T x = 2 +
    T y = 3;
}
```

Leads to

```
Error at line 3 col 7. Repairs found:
  Insert "COMMA"
```

However, three repair sequences were initially found: *Insert ‘,’*; *Insert ‘?’*; *Insert ‘(’*.

Ranking repair sequences

```
class C {
    T x = 2 +
    T y = 3;
}
```

Leads to

```
Error at line 3 col 7. Repairs found:
  Insert "COMMA"
```

However, three repair sequences were initially found: *Insert ‘,’*; *Insert ‘?’*; *Insert ‘(’*.

Rank the complete set of minimum cost repair sequences and select one from the best to apply to the input. Throw away the non-best.

CPCT+ takes Corchuelo *et al.* as a base

CPCT$^+$ takes Corchuelo *et al.* as a base :

- CR SHIFT 1 $\rightarrow$ CR SHIFT 3

CPCT$^+$ takes Corchuelo *et al.* as a base :

- CR SHIFT 1 $\rightarrow$ CR SHIFT 3
- Efficient queue and use of parent-pointer trees

CPCT$^+$ takes Corchuelo *et al.* as a base :

- CR SHIFT 1 $\rightarrow$ CR SHIFT 3
- Efficient queue and use of parent-pointer trees
- Can generate the complete set of minimum cost repair sequences

CPCT$^+$ takes Corchuelo *et al.* as a base :

- CR SHIFT 1 $\rightarrow$ CR SHIFT 3
- Efficient queue and use of parent-pointer trees
- Can generate the complete set of minimum cost repair sequences
- Rank the complete set of minimum cost repair sequences

MF uses the A* algorithm and a distance table to speed up recovery.

MF uses the A* algorithm and a distance table to speed up recovery.

	t					
s	INT	+	*	(	)	$
0	0	1	1	0	1	1
1	0	1	1	0	1	1
2	1	0	1	1	0	0
3	1	0	0	1	0	0
4	1	0	0	1	0	0
5	∞	∞	∞	∞	∞	0
6	2	1	1	2	0	1
7	0	1	1	0	1	1
8	0	1	1	0	1	1
9	1	0	0	1	0	0
10	2	1	1	2	0	0
11	1	0	1	1	0	0

MF uses the A* algorithm and a distance table to speed up recovery.

	t					
s	INT	+	*	(	)	$
0	0	1	1	0	1	1
1	0	1	1	0	1	1
2	1	0	1	1	0	0
3	1	0	0	1	0	0
4	1	0	0	1	0	0
5	∞	∞	∞	∞	∞	0
6	2	1	1	2	0	1
7	0	1	1	0	1	1
8	0	1	1	0	1	1
9	1	0	0	1	0	0
10	2	1	1	2	0	0
11	1	0	1	1	0	0

MF is another talk in and of itself...

Experiment

Evaluating error recovery algorithms an historical weakness.

Experiment

Evaluating error recovery algorithms an historical weakness.

Most papers use 200-ish inputs.

Experiment

Evaluating error recovery algorithms an historical weakness.

Most papers use 200-ish inputs.

Cerecke uses ˜60,000 inputs.

Experiment

Evaluating error recovery algorithms an historical weakness.

Most papers use 200-ish inputs.

Cerecke uses ~60,000 inputs.

We have Blackbox! 200,000 Java files are ours.

Experiment (draft data)

	Mean time (s)	Median time (s)	Cost size (#)	Failure rate (%)	Lexemes skipped (%)	Error locations (#)
Panic	0.000002 ±<0.0000001	0.000001 ±<0.0000001	-	-	3.87 ±<0.001	980,848 ±0

Experiment (draft data)

	Mean time (s)	Median time (s)	Cost size (#)	Failure rate (%)	Lexemes skipped (%)	Error locations (#)
Panic	0.000002 ±<0.0000001	0.000001 ±<0.0000001	-	-	3.87 ±<0.001	980,848 ±0
CPCT^{+}	0.014739 ±0.0000052	0.000308 ±0.0000002	1.65 ±<0.001	1.81 ±0.003	0.28 ±<0.001	436,577 ±33

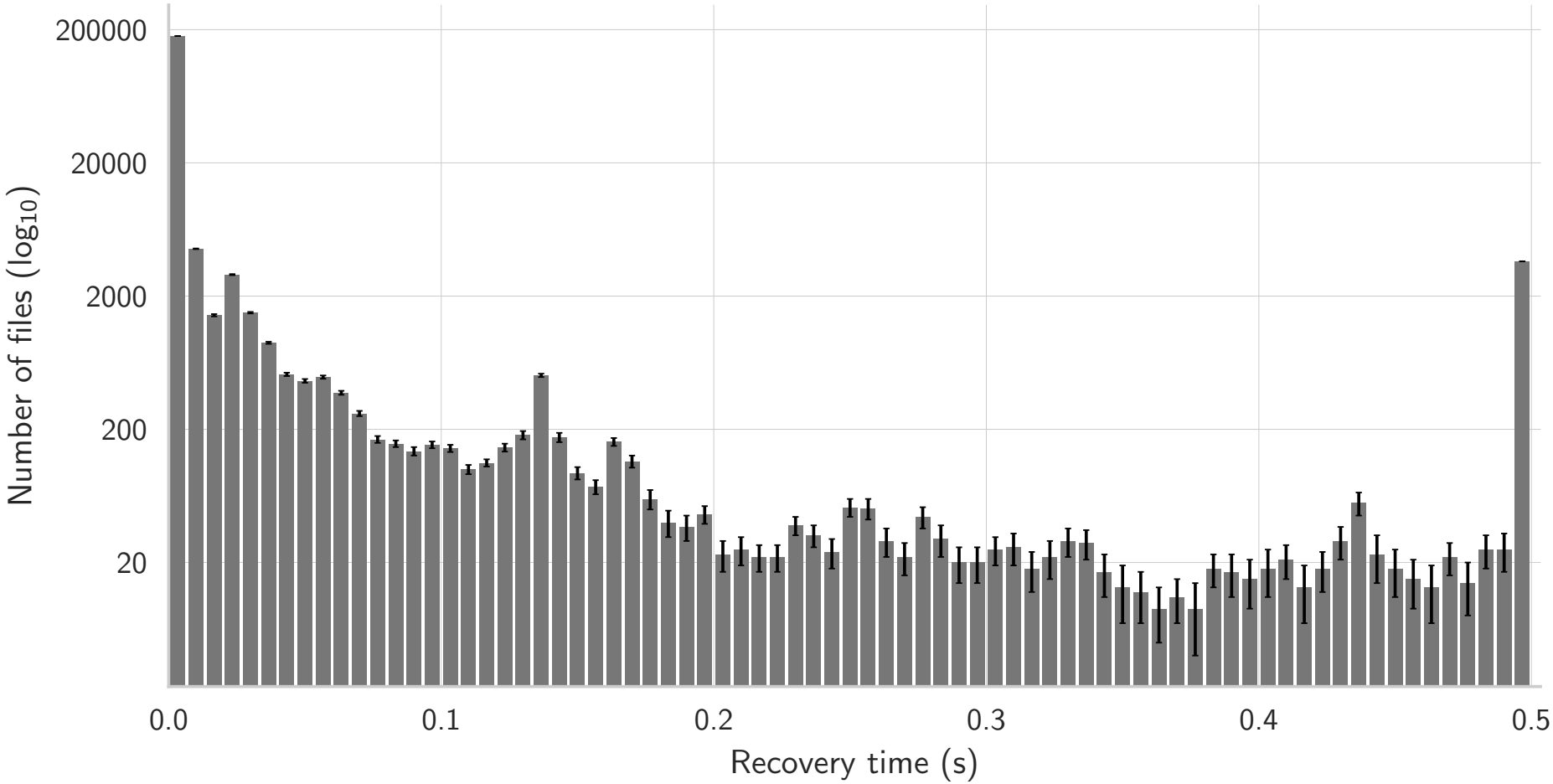
Experiment (draft data)

	Mean time (s)	Median time (s)	Cost size (#)	Failure rate (%)	Lexemes skipped (%)	Error locations (#)
Panic	0.000002 ±<0.0000001	0.000001 ±<0.0000001	-	-	3.87 ±<0.001	980,848 ±0
CPCT$^+$	0.014739 ±0.0000052	0.000308 ±0.0000002	1.65 ±<0.001	1.81 ±0.003	0.28 ±<0.001	436,577 ±33
MF	0.010079 ±0.0000109	0.000162 ±0.0000002	1.66 ±<0.001	1.28 ±0.002	0.28 ±<0.001	440,123 ±36

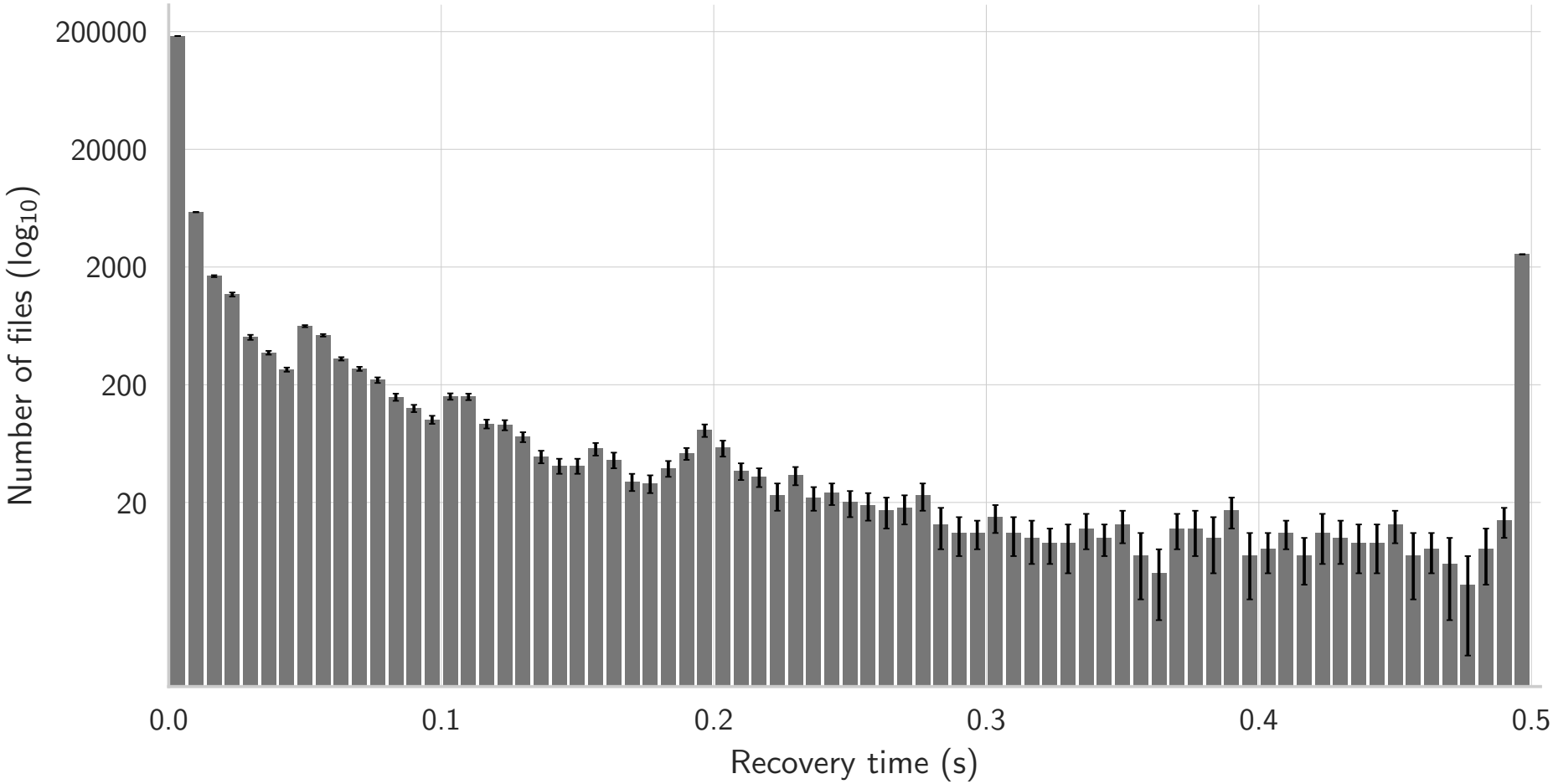
Experiment (draft data)

	Mean time (s)	Median time (s)	Cost size (#)	Failure rate (%)	Lexemes skipped (%)	Error locations (#)
Panic	0.000002 ±<0.0000001	0.000001 ±<0.0000001	-	-	3.87 ±<0.001	980,848 ±0
CPCT^{+}	0.014739 ±0.0000052	0.000308 ±0.0000002	1.65 ±<0.001	1.81 ±0.003	0.28 ±<0.001	436,577 ±33
MF	0.010079 ±0.0000109	0.000162 ±0.0000002	1.66 ±<0.001	1.28 ±0.002	0.28 ±<0.001	440,123 ±36
MF$_{rev}$	0.013570 ±0.0000140	0.000194 ±0.0000003	1.77 ±<0.001	1.77 ±0.002	0.39 ±<0.001	582,305 ±42

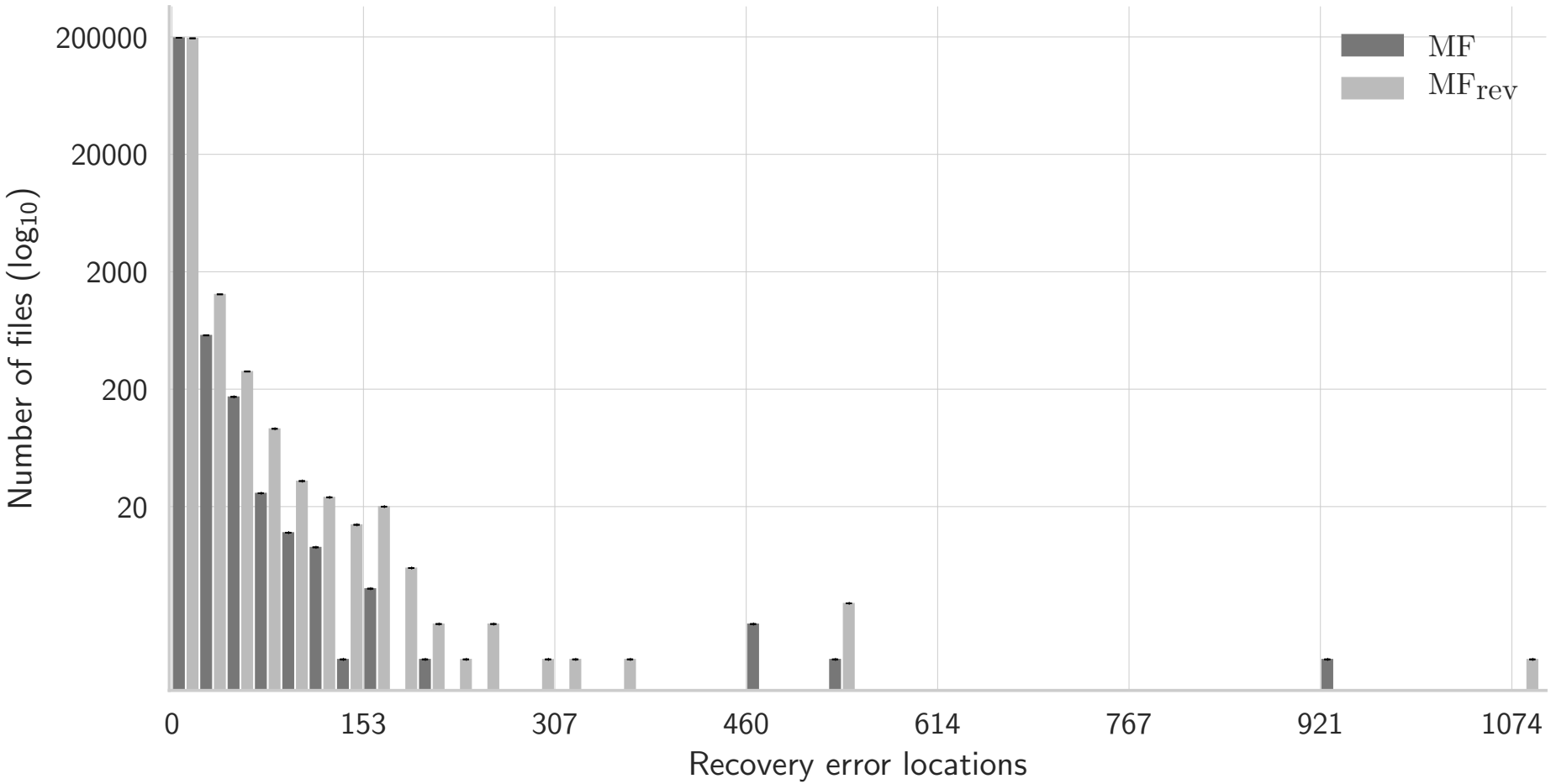
Time spent in recovery (*CPCT*$^+$)



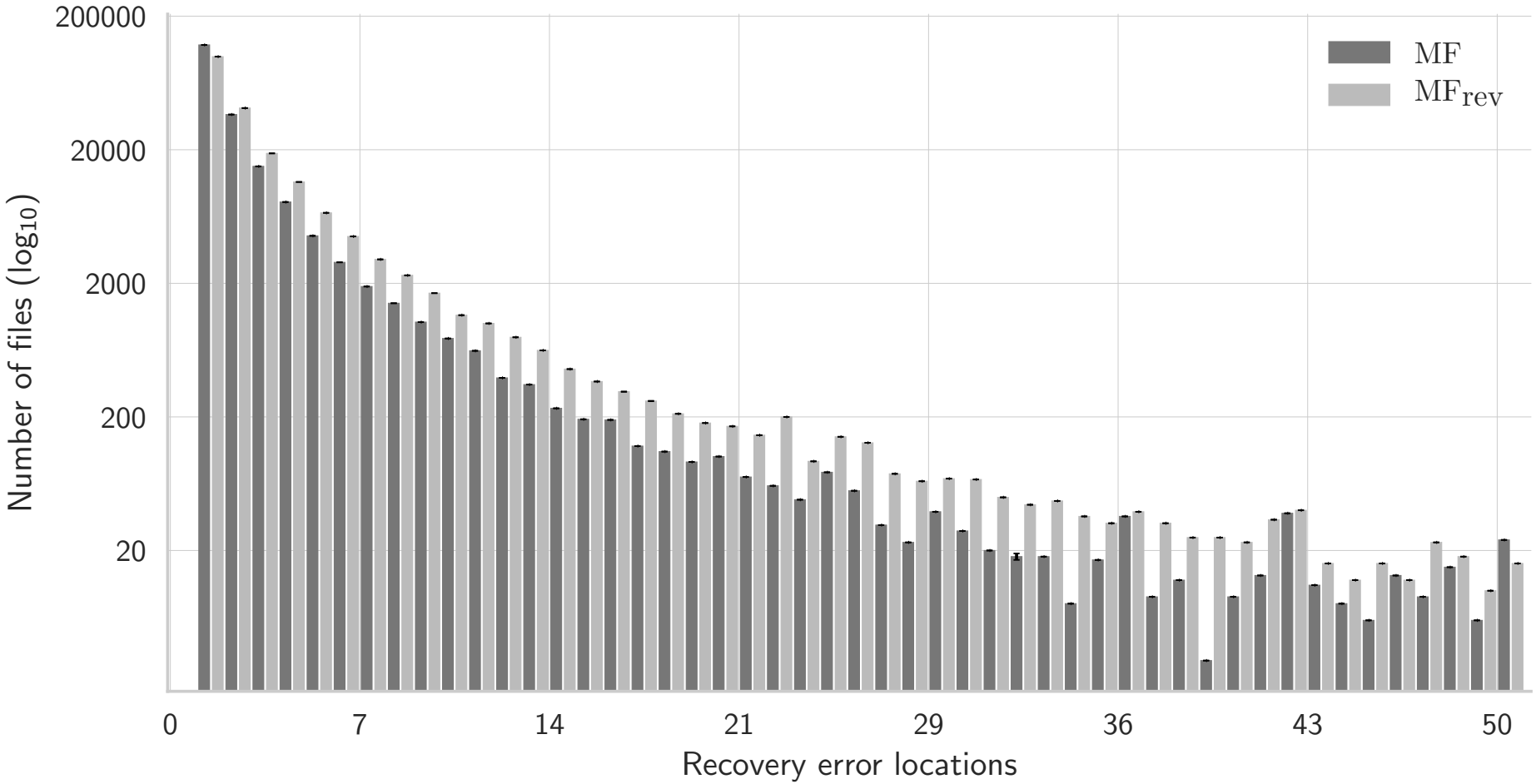
Time spent in recovery (*MF*)



Error locations (*MF* vs. MF_{rev})



Error locations (*MF* vs. MF_{rev})



References

Reducing Cascading Parsing Errors Through Fast Error Recovery
L. Diekmann and L. Tratt

A Locally Least-Cost LR-Error Corrector
C. N. Fischer, B. A. Dion, and J. Mauney. Technical Report 363. University of Wisconsin, 1979.

Locally least-cost error repair in LR parsers
C. Cerecke Ph.D. Dissertation. University of Canterbury. 2003.

Repairing syntax errors in LR parsers
Rafael Corchuelo, José Antonio Pérez, Antonio Ruiz-Cortés, and Migue Toro TOPLAS 24 (Nov 2002)

Software

grmtools Our Rust parser tools

https://github.com/softdevteam/grmtools

Thanks

- EPSRC: *Lecture*.

Thanks for listening

```
class C {
  int x y;
}

Error at line 2 col 9. Repairs found:
  Delete "y"
  Insert "COMMA"
  Insert "EQ"
```