Finding Counterexamples from Parsing Conflicts

Chin Isradisaikul
chinawat@cs.cornell.edu

Andrew Myers
andru@cs.cornell.edu

Cornell University
Department of Computer Science

PLDI 2015 | Wed, June 17, 2015 | Portland, OR

This paper

improving

parser generators

(awesome for flawless grammars)

with better

error messages

(confusing for faulty grammars)

Puzzling error messages waste our time

$$
\begin{aligned}
\mathit{stmt} &\rightarrow \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt} \\
&\mid\ \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{stmt} \\
&\mid\ \mathit{expr}\ \texttt{?}\ \mathit{stmt}\ \ \mathit{stmt} \\
&\mid\ \texttt{arr [}\ \mathit{expr}\ \texttt{] :=}\ \mathit{expr} \\
\mathit{expr} &\rightarrow \mathit{num}\ \mid\ \mathit{expr}\ \texttt{+}\ \mathit{expr} \\
\mathit{num} &\rightarrow \langle \mathit{digit} \rangle\ \mid\ \mathit{num}\ \langle \mathit{digit} \rangle
\end{aligned}
$$

↓

Yacc

LALR Parser Generator for Java™

Scott Hudson, GVU Center, Georgia Tech

LALR parser generator

Puzzling error messages waste our time

$$
\begin{aligned}
stmt &\rightarrow \texttt{if}\ expr\ \texttt{then}\ stmt\ \texttt{else}\ stmt \\
&\mid \texttt{if}\ expr\ \texttt{then}\ stmt \\
&\mid expr\ \texttt{?}\ stmt\ stmt \\
&\mid \texttt{arr [}\ expr\ \texttt{] :=}\ expr \\
expr &\rightarrow num \mid expr\ \texttt{+}\ expr \\
num &\rightarrow \langle digit\rangle \mid num\ \langle digit\rangle
\end{aligned}
$$

Yacc

LALR parser generator

```
Warning : *** Shift/Reduce conflict
    found in state #10
  between reduction on
    stmt ::= IF expr THEN stmt •
  and shift on
    stmt ::= IF expr THEN stmt • ELSE stmt
  under symbol ELSE

Warning : *** Shift/Reduce conflict
    found in state #13
  between reduction on
    expr ::= expr PLUS expr •
  and shift on
    expr ::= expr • PLUS expr
  under symbol PLUS

Warning : *** Shift/Reduce conflict
    found in state #1
  between reduction on
    expr ::= num •
  and shift on
    num ::= num • DIGIT
  under symbol DIGIT
```

Why error messages are puzzling

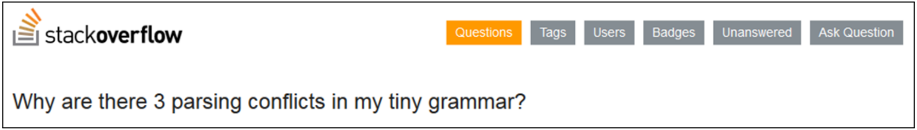
errors reported as **conflicts**
(parser generator internals)

not
in terms of grammar or language

```
Warning : *** Shift/Reduce conflict
    found in state #10
  between reduction on
    stmt ::= IF expr THEN stmt •
  and shift on
    stmt ::= IF expr THEN stmt • ELSE stmt
  under symbol ELSE

Warning : *** Shift/Reduce conflict
    found in state #13
  between reduction on
    expr ::= expr PLUS expr •
  and shift on
    expr ::= expr • PLUS expr
  under symbol PLUS

Warning : *** Shift/Reduce conflict
    found in state #1
```



Succinct explanations?

0

I'm still switching thing arround, and my original question had some errors since the **elseifs** sequence had an **else** allways at the end which was wrong. Here is another take at the question, this time I get two shift/reduce conflicts:

```
flow : '#' IF '(' ')' statements elsebody
     ;

elsebody : else
         | elseifs else
         ;

else : '#' ELSE statements '#' END
     | '#' END
     ;

elseifs : /* empty */
        | elseifs '#' ELSEIF statements
        ;
```

The conflicts now are:

```
// Parser Conflict Information for grammar file "program.y"

Shift/Reduce conflict on symbol "'#'", parser will shift
 Reduce 12: elseifs -> /* empty */
 Shift "'#'":    State-10 -> State-13
  Items for From-state State 10
    7 flow: '#' IF '(' ')' statements . elsebody
    4 statements: statements . stmt
  Items for Next-state State 13
    10 else: '#' . ELSE statements '#' END
    11 else: '#' . END
    7 flow: '#' . IF '(' ')' statements elsebody

Shift/Reduce conflict on symbol "'#'", parser will shift
 Reduce 13: elseifs -> elseifs, '#', ELSEIF, statements
 Shift "'#'":    State-24 -> State-6
  Items for From-state State 24
    13 elseifs: elseifs '#' ELSEIF statements .
    -lookahead: '#'
    4 statements: statements . stmt
  Items for Next-state State 6
    7 flow: '#' . IF '(' ')' statements elsebody

// End conflict information for parser
```

Empty rules just aggravate the gppg i'm affraid. But they seem so natural to use I keep trying them.

I already know right recursion solves the problem as *1800 INFORMATION* has said. But I'm looking

4

A conflict means that the grammar you gave to bison is not LALR(1), so it can't decide what action to take in every possible case in order to correctly parse the grammar.

In your case, the problem is that your grammar is ambiguous. If you give it an input like

```
NUMBER AND NUMBER AND NUMBER
```

it can't decide if it should parse it as equivalent to

```
( NUMBER AND NUMBER ) AND NUMBER
```

or

```
NUMBER AND ( NUMBER AND NUMBER )
```

There are a number of ways you can resolve this:

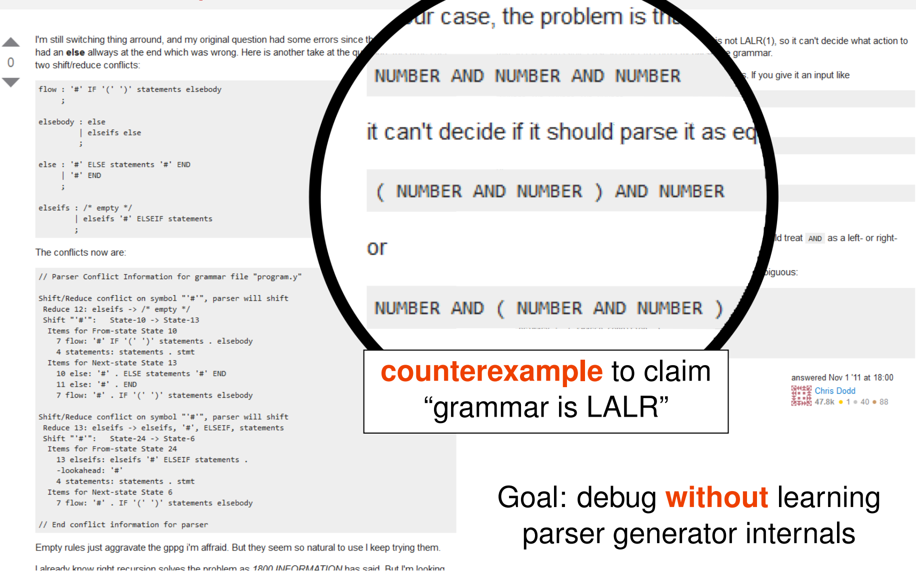
- you can use `%left AND` or `%right AND` to tell bison that it should treat `AND` as a left- or right-associative infix operator
- you can refactor the `search_condition` rule to make it unambiguous:

```
search_condition : search_condition AND primary
                 | primary
                 ;
primary : '( search_condition ')'
        | predicate
        ;
```

share improve this answer

answered Nov 1 '11 at 18:00
Chris Dodd
47.8k ● 1 ● 40 ● 88

Succinct explanations?



counterexample to claim “grammar is LALR”

Goal: debug **without** learning parser generator internals

Succinct explanations

Problem statement

We seek counterexamples that are...

1. easy to understand
2. efficient to find

Good counterexamples are hard to find

$$
\begin{array}{lcl}
\mathit{stmt} & \to & \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt} \\
 & | & \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{stmt} \\
 & | & \mathit{expr}\ \texttt{?}\ \mathit{stmt}\ \ \mathit{stmt} \\
 & | & \texttt{arr}\ \texttt{[}\ \mathit{expr}\ \texttt{]}\ \texttt{:=}\ \mathit{expr} \\
\mathit{expr} & \to & \mathit{num}\ \ |\ \ \mathit{expr}\ \texttt{+}\ \mathit{expr} \\
\mathit{num} & \to & \langle \mathit{digit} \rangle\ \ |\ \ \mathit{num}\ \langle \mathit{digit} \rangle
\end{array}
$$

ambiguous grammar
(serious syntactic problem)
⇓
want ambiguous counterexample
⇓
counterexample should indicate
ambiguity in grammar

Good counterexamples are hard to find

$$
\begin{array}{lcl}
stmt & \rightarrow & \texttt{if}\ expr\ \texttt{then}\ stmt\ \texttt{else}\ stmt \\
 & | & \texttt{if}\ expr\ \texttt{then}\ stmt \\
 & | & expr\ \texttt{?}\ stmt\ \ stmt \\
 & | & \texttt{arr}\ \texttt{[}\ expr\ \texttt{]}\ \texttt{:=}\ expr \\
expr & \rightarrow & num \quad | \quad expr\ \texttt{+}\ expr \\
num & \rightarrow & \langle digit \rangle \quad | \quad num\ \langle digit \rangle
\end{array}
$$

ambiguous grammar
(serious syntactic problem)
⇓
want ambiguous counterexample
⇓
counterexample should indicate
ambiguity in grammar

Bad news:

Ambiguity detection is undecidable.

Game over?

Comparison: prior & our approaches

approach	ambiguities checked	accurate reports	efficient	
approximation NU test [S 07]	✔		✔	allows false positives
brute force AMBER [S 01] DMS [BPM 04]	✔	✔		never terminates on unambiguous grammars
punting on ambiguities ANTLR 4 [PHF 14] Elkhound [MN 04]			✔	can miss unexpected ambiguities
our counterexample finder	✔	✔	✔	

Succinct explanations

Problem statement

We seek counterexamples that are...

1. easy to understand
2. efficient to find

Good counterexamples

No more concrete than necessary

$$
\begin{array}{lcl}
stmt & \rightarrow & \texttt{if}\ expr\ \texttt{then}\ stmt\ \texttt{else}\ stmt \\
 & | & \texttt{if}\ expr\ \texttt{then}\ stmt \\
 & | & expr\ \texttt{?}\ stmt\ \ stmt \\
 & | & \texttt{arr [}\ expr\ \texttt{] :=}\ expr \\
expr & \rightarrow & num \ |\ expr\ \texttt{+}\ expr \\
num & \rightarrow & \langle digit \rangle \ |\ num\ \langle digit \rangle
\end{array}
$$

`if 42 then if 17 then arr[2] := 5 else arr[4] := 7`

too specific, distracting

`if` *expr* `then if` *expr* `then` *stmt* `else` *stmt* ✔

general and abstract

use nonterminals when possible

Good counterexamples

Derivation of most specific nonterminal causing ambiguity

$$
\begin{array}{lcl}
stmt & \rightarrow & \texttt{if}\ expr\ \texttt{then}\ stmt\ \texttt{else}\ stmt \\
 & | & \texttt{if}\ expr\ \texttt{then}\ stmt \\
 & | & expr\ \texttt{?}\ stmt\ \ stmt \\
 & | & \texttt{arr}\ \texttt{[}\ expr\ \texttt{]}\ \texttt{:=}\ expr \\
expr & \rightarrow & num \ \ |\ \ expr\ \texttt{+}\ expr \\
num & \rightarrow & \langle digit\rangle \ \ |\ \ num\ \langle digit\rangle
\end{array}
$$

$$stmt \rightarrow^* expr + expr + expr\ ?\ stmt\ \ stmt$$

not specific enough

extra tokens distracting

$$expr \rightarrow^* expr + expr + expr \quad ✔$$

root cause of ambiguity

Succinct explanations

Problem statement

We seek counterexamples that are...

1. easy to understand ✔
   (most general derivation of most specific nonterminal causing ambiguity)
2. efficient to find

Idea: exploit parser state machine

ambiguity in grammar

$\Downarrow$

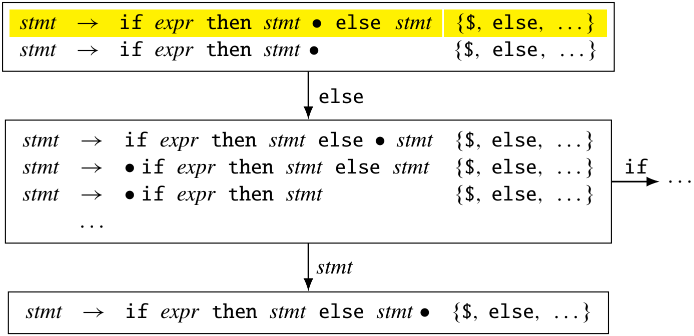
conflict in parser state machine
(parser generator internals)

$\Downarrow$

find counterexample from conflict

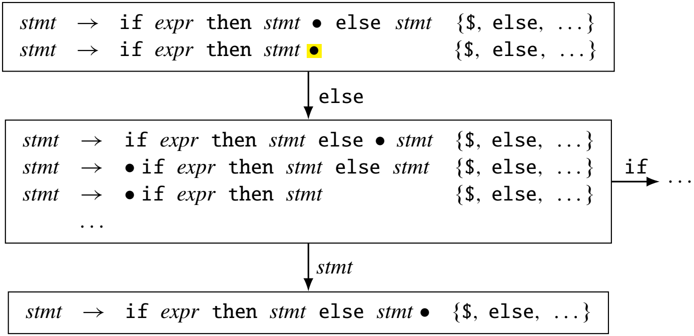
Time to learn parser generator internals...
...one last time

Anatomy of LR parser state machine



A state contains a collection of production items

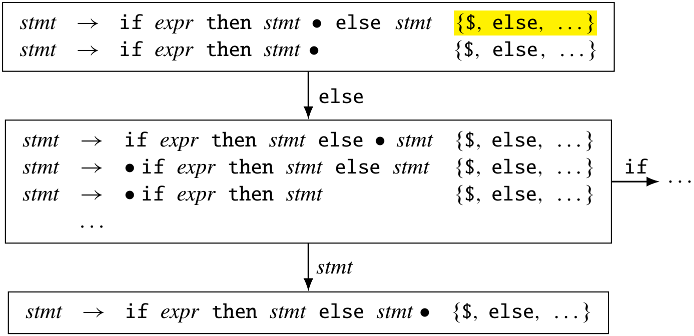
Anatomy of LR parser state machine



A state contains a collection of production items

The dot (•) indicates progress on completing a production

Anatomy of LR parser state machine

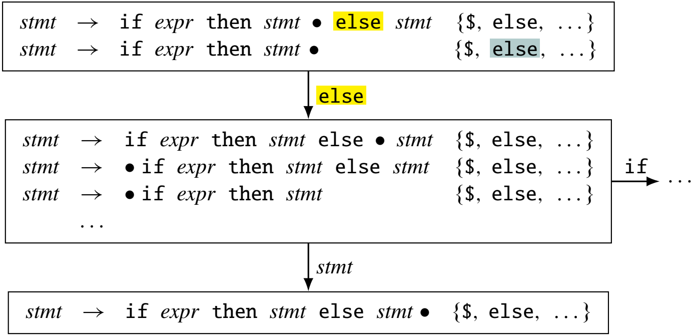


A state contains a collection of production items

The dot (•) indicates progress on completing a production

The lookahead set lists terminal symbols that can follow production

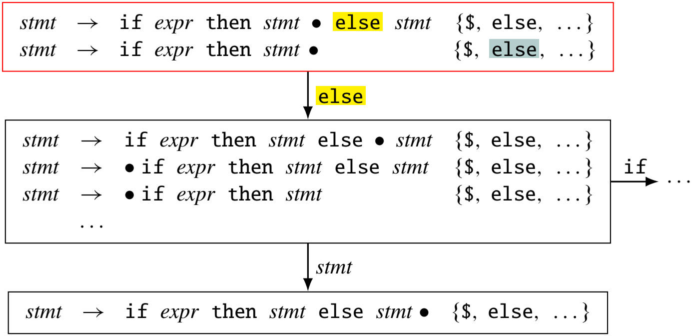
Anatomy of LR parser state machine



Parser actions:

- **shift**: consume next input symbol
  (has outgoing transition)
- **reduce**: finish up a production
  (lookahead set of item ending with • has next symbol)

Parsing conflicts



- shift/reduce conflict:
  shift & reduce possible on same input symbol
- reduce/reduce conflict:
  items ending with • have intersecting lookahead sets

Connection between conflict and ambiguity

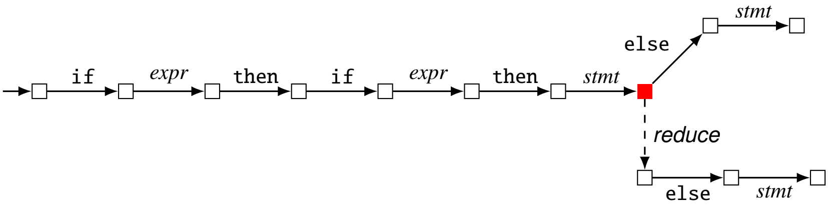
conflict ⇒ ∃ input taking parser from start to conflict states

⇓

ambiguity ⇒ parser actions differ at conflict state and diverge for rest of input

⇓

keep track of both parses simultaneously to find ambiguous counterexample (**unifying counterexample**)



Simulating copies of parser in parallel

product parser: states = Cartesian product of original parser items

$stmt \rightarrow$ `if` $expr$ `then` $stmt \bullet$ `else` $stmt$	$\leftarrow$ item for 1st parser
$stmt \rightarrow$ `if` $expr$ `then` $stmt \bullet$	$\leftarrow$ item for 2nd parser

ensures: identical input parsed by both copies

Actions on product parser

Intuition: Keeping the input identical for both copies.

$stmt \rightarrow$ `if` $\bullet$ $expr$ `then` $stmt$ `else` $stmt$
$stmt \rightarrow$ `if` $\bullet$ $expr$ `then` $stmt$

↓ $expr$

$stmt \rightarrow$ `if` $expr$ $\bullet$ `then` $stmt$ `else` $stmt$
$stmt \rightarrow$ `if` $expr$ $\bullet$ `then` $stmt$

transition on same symbol

$stmt \rightarrow$ `if` $\bullet$ $expr$ `then` $stmt$ `else` $stmt$
$stmt \rightarrow$ `if` $\bullet$ $expr$ `then` $stmt$

↓ [prod]

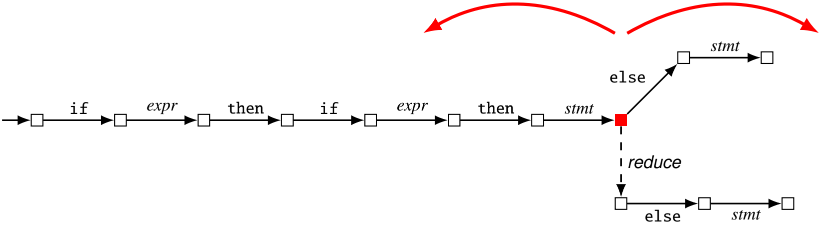
$expr \rightarrow \bullet\ expr + expr$
$stmt \rightarrow$ `if` $\bullet$ $expr$ `then` $stmt$

production step: work on deeper production

Searching forward & backward from conflict state

searching from start state = unguided brute force
(need to use conflict state anyway)

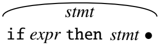
start at conflict state = guided brute force
(well begun is half done)



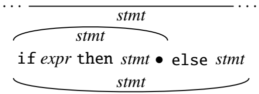
Search stages

```
reduction on stmt ::= IF expr THEN stmt •
shift on     stmt ::= IF expr THEN stmt • ELSE stmt
under symbol ELSE
```

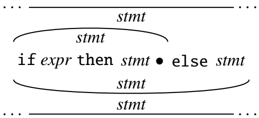
1. completing reduce item



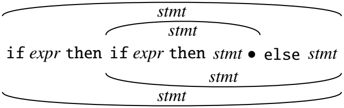
2. completing shift item



3. finding ambiguous nonterminal



4. completing counterexample

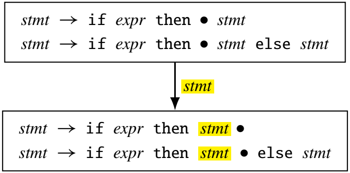


Our search in action

Stage 1: completing reduce item

```
reduction on stmt ::= IF expr THEN stmt •
shift on     stmt ::= IF expr THEN stmt • ELSE stmt
under symbol ELSE
```

Start at conflict state



backward transition

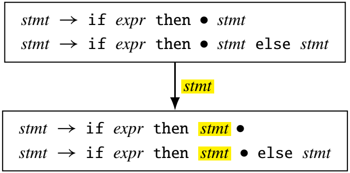
•

Our search in action

Stage 1: completing reduce item

```
reduction on stmt ::= IF expr THEN stmt •
shift on     stmt ::= IF expr THEN stmt • ELSE stmt
under symbol ELSE
```

Start at conflict state; take backward transition



backward transition

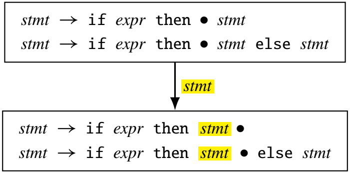
stmt •

Our search in action

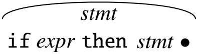
Stage 1: completing reduce item

```
reduction on stmt ::= IF expr THEN stmt •
shift on     stmt ::= IF expr THEN stmt • ELSE stmt
under symbol ELSE
```

Start at conflict state; take backward transitions



backward transition

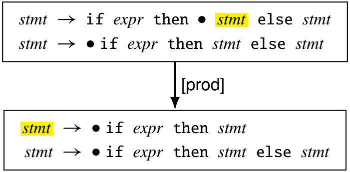


Our search in action

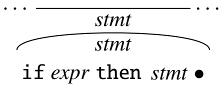
Stage 1: completing reduce item

```
reduction on stmt ::= IF expr THEN stmt •
shift on     stmt ::= IF expr THEN stmt • ELSE stmt
under symbol ELSE
```

Find out who wants this derivation; take backward production step



backward production step

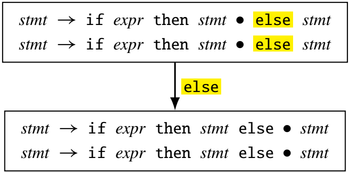


Our search in action

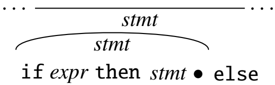
Stage 2: completing shift item

```
reduction on stmt ::= IF expr THEN stmt •
shift on     stmt ::= IF expr THEN stmt • ELSE stmt
under symbol ELSE
```

Try to derive the next symbol; take forward transition



forward transition

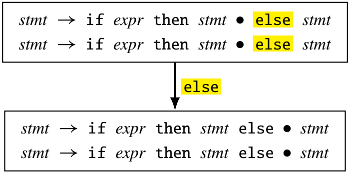


Our search in action

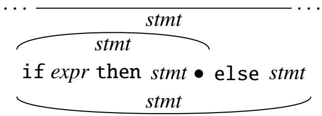
Stage 2: completing shift item

```
reduction on stmt ::= IF expr THEN stmt •
shift on     stmt ::= IF expr THEN stmt • ELSE stmt
under symbol ELSE
```

Try to derive the next symbol; take forward transitions



forward transition



Our search in action

Stage 3: finding ambiguous nonterminal

```
reduction on stmt ::= IF expr THEN stmt •
shift on     stmt ::= IF expr THEN stmt • ELSE stmt
under symbol ELSE
```

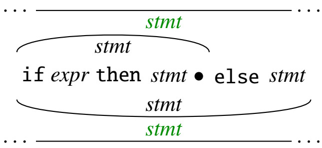
Find out who wants this derivation; take backward production step

$stmt \to$ `if` $expr$ `then` $\bullet$ $stmt$ `else` $stmt$
$stmt \to$ `if` $expr$ `then` $\bullet$ $stmt$

↓ [prod]

$stmt \to$ `if` $expr$ `then` $\bullet$ $stmt$ `else` $stmt$
$stmt \to \bullet$ `if` $expr$ `then` $stmt$ `else` $stmt$

backward production step



Our search in action

Stage 4: completing counterexample

```
reduction on stmt ::= IF expr THEN stmt •
shift on     stmt ::= IF expr THEN stmt • ELSE stmt
under symbol ELSE
```

Keep expanding outward

Complete counterexample:

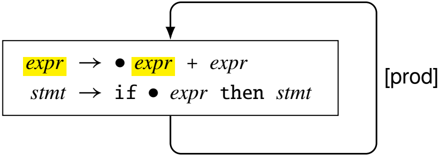
$$\overbrace{\texttt{if}\ \mathit{expr}\ \texttt{then}\ \overbrace{\texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{stmt}\ \bullet}^{\mathit{stmt}}\ \texttt{else}\ \mathit{stmt}}^{\mathit{stmt}}$$

$$\underbrace{\texttt{if}\ \mathit{expr}\ \texttt{then}\ \underbrace{\texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{stmt}\ \bullet\ \texttt{else}\ \mathit{stmt}}_{\mathit{stmt}}}_{\mathit{stmt}}$$

Our search vs undecidability

Nontermination happens when

- grammar is not ambiguous, and
- production step is taken repeatedly



production step: work on deeper production

Implementation

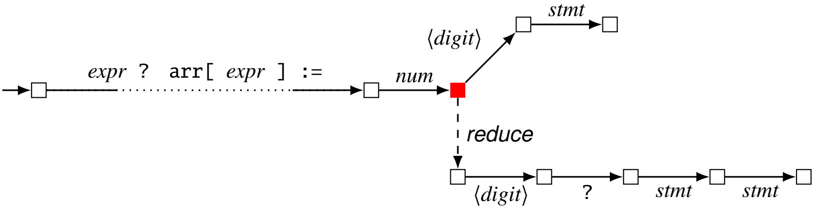
Extended CUP LALR parser generator:

- ~1,500 lines of code added
- counterexample searched for each conflict

```
Warning : *** Shift/Reduce conflict found in state #1
  between reduction on expr ::= num •
  and shift on         num ::= num • ⟨digit⟩
  under symbol ⟨digit⟩
  Ambiguity detected for nonterminal stmt
  Example: expr ? arr [ expr ] := num • ⟨digit⟩ ⟨digit⟩ ? stmt stmt
  Derivation using reduction: stmt ::= ...
  Derivation using shift    : stmt ::= ...
  Resolved in favor of shifting.
```

Implementation vs undecidability

- 5-second timeout
  - duration you're willing to wait
- reports **nonunifying counterexample** instead
  - symbols may differ after conflict state
  - search is decidable



Evaluation

Tested on a desktop. . .

our grammars

grammars from StackOverflow and StackExchange

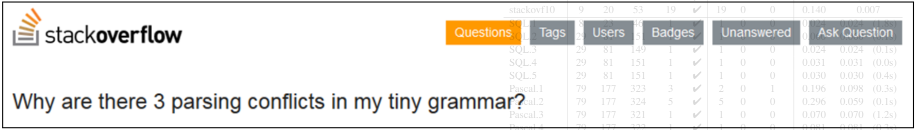
grammars used to evaluate grammar filtering technique (approximation + brute force)

Grammar	# nonterms	# prods	# states	# conflicts	Amb?	# unif	# nonunif	# time out	Time (seconds) Total	Average
figure1	3	9	24	3	✔	3	0	0	0.072	0.024
figure3	4	7	10	1	✗	0	1	0	0.010	0.010
figure7	4	10	16	2	✔	2	0	0	0.016	0.008
ambfailed01	6	10	17	1	✔	0	1	0	0.010	0.010
abcd	5	11	22	3	✔	3	0	0	0.024	0.008
simp2	10	41	70	1	✔	1	0	0	0.548	0.548
xi	16	41	82	6	✔	6	0	0	0.155	0.026
eqn	14	67	133	1	✔	1	0	0	0.169	0.169
java-ext1	185	445	767	2	✗	0	0	2	T/L	T/L
java-ext2	234	599	1255	1	✗	0	0	1	T/L	T/L
stackexc01	2	7	13	3	✔	3	0	0	0.023	0.008
stackexc02	6	11	15	1	✗	0	1	0	0.008	0.008
stackovf01	2	5	9	1	✗	0	1	0	0.009	0.009
stackovf02	2	5	9	4	✔	4	0	0	0.043	0.011
stackovf03	2	6	10	1	✔	1	0	0	0.017	0.017
stackovf04	5	9	13	1	✗	0	1	0	0.009	0.009
stackovf05	5	10	14	1	✔	1	0	0	0.010	0.010
stackovf06	6	10	15	2	✗	0	2	0	0.012	0.006
stackovf07	7	12	17	3	✔	3	0	0	0.028	0.009
stackovf08	3	13	21	8	✗	0	8	0	0.025	0.003
stackovf09	6	12	27	1	✗	0	1	0	0.017	0.017
stackovf10	9	20	53	19	✔	19	0	0	0.140	0.007
SQL.1	8	23	46	1	✔	1	0	0	0.024	0.024 (1.8s)
SQL.2	29	81	151	1	✔	1	0	0	0.060	0.060 (0.1s)
SQL.3	29	81	149	1	✔	1	0	0	0.024	0.024 (0.1s)
SQL.4	29	81	151	1	✔	1	0	0	0.031	0.031 (0.0s)
SQL.5	29	81	151	1	✔	1	0	0	0.030	0.030 (0.4s)
Pascal.1	79	177	323	3	✔	2	0	1	0.196	0.098 (0.3s)
Pascal.2	79	177	324	5	✔	5	0	0	0.296	0.059 (0.1s)
Pascal.3	79	177	321	1	✔	1	0	0	0.070	0.070 (1.2s)
Pascal.4	79	177	322	1	✔	1	0	0	0.081	0.081 (0.3s)
Pascal.5	79	177	322	1	✔	1	0	0	0.113	0.113 (0.3s)
C.1	64	214	369	1	✔	1	0	0	0.327	0.327 (1.3s)
C.2	64	214	368	1	✔	1	0	0	0.219	0.219 (1.11h)
C.3	64	214	368	4	✔	4	0	0	1.015	0.254 (0.5s)
C.4	64	214	369	1	✔	0	0	1	T/L	T/L (1.3s)
C.5	64	214	370	1	✔	1	0	0	0.212	0.212 (4.9s)
Java.1	152	351	607	1	✔	1	0	0	0.569	0.569 (32.4s)
Java.2	152	351	606	1133	✔	141	0	9 (983)	35.384	0.251 (0.4s)
Java.3	152	351	608	2	✔	2	0	0	0.435	0.218 (35.1s)
Java.4	152	351	608	14	✔	6	2	6	2.042	0.255 (6.5s)
Java.5	152	351	607	3	✔	3	0	0	0.526	0.175 (3.3s)

Evaluation results

- effective: 92% of conflicts didn't time out
  (nonunifying counterexamples reported for other 8%)
- efficient: if not timed out, 0.18s spent per conflict
  10.7x faster than grammar filtering
  8ms per conflict for StackOverflow grammars

Grammar	# nonterms	# prods	# states	# conflicts	Amb?	# unif	# nonunif	# time out	Time (seconds) Total	Average	
figure1	3	9	24	3	✔	3	0	0	0.072	0.024	
figure3	4	7	10	1	✗	0	1	0	0.010	0.010	
figure7	4	10	16	2	✔	2	0	0	0.016	0.008	
ambfailed01	6	10	17	1	✔	0	1	0	0.010	0.010	
[illegible]	5	11	22	3	✔	3	0	0	0.024	0.008	
simp2	10	41	70	1	✔	1	0	0	0.548	0.548	
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	[illegible]	0.026	
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	[illegible]	0.169	
java-ext1	185	445	767	2	✗	0	0	2	T/L	T/L	
java-ext2	234	599	[illegible]	1	✗	0	0	1	T/L	T/L	
[illegible]	[illegible]	[illegible]	[illegible]	3	✔	3	0	0	0.023	0.008	
stackexc02	6	11	15	1	✗	0	1	0	0.008	0.008	
[illegible]	[illegible]	5	9	1	✗	0	1	0	0.009	0.009	
[illegible]	[illegible]	5	9	4	✔	4	0	0	0.043	0.011	
stackovf03	2	6	10	1	✔	1	0	0	0.017	0.017	
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1	0	0.009	0.009	
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0	0.010	0.010	
stackovf06	6	10	15	2	✗	0	2	0	0.012	0.006	
stackovf07	7	12	17	3	✔	3	0	0	0.028	0.009	
stackovf08	3	13	21	8	✗	0	8	0	0.025	0.003	
stackovf09	6	12	27	1	✗	0	1	0	0.017	0.017	
stackovf10	9	20	53	19	✔	19	0	0	0.140	0.007	
SQL.1	8	23	46	1	✔	1	0	0	0.024	0.024	(1.8s)
SQL.2	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
SQL.3	29	81	149	1	✔	1	0	0	0.024	0.024	(0.1s)
SQL.4	29	81	151	1	✔	1	0	0	0.031	0.031	(0.0s)
SQL.5	29	81	151	1	✔	1	0	0	0.030	0.030	(0.4s)
Pascal.1	79	177	323	3	✔	2	0	1	0.196	0.098	(0.3s)
Pascal.2	79	177	324	5	✔	5	0	0	0.296	0.059	(0.1s)
Pascal.3	79	177	321	1	✔	1	0	0	0.070	0.070	(1.2s)
Pascal.4	79	177	322	1	✔	1	0	0	0.081	0.081	(0.3s)
Pascal.5	79	[illegible]	322	1	✔	1	0	0	0.113	0.113	(0.3s)
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0.327	0.327	(1.3s)
C.2	64	214	368	1	✔	1	0	0	0.219	0.219	(1.11h)
C.3	64	214	368	4	✔	4	0	0	1.015	0.254	(0.5s)
C.4	64	214	369	1	✔	0	0	1	T/L	T/L	(1.3s)
C.5	64	214	370	1	✔	1	0	0	0.212	0.212	(4.9s)
Java.1	152	351	607	1	✔	1	0	0	0.569	0.569	(32.4s)
Java.2	152	351	606	1133	✔	141	0	9 (983)	35.384	0.251	(0.4s)
Java.3	152	351	608	2	✔	2	0	0	0.435	0.218	(35.1s)
Java.4	152	351	608	14	✔	6	2	6	2.042	0.255	(6.5s)
Java.5	152	351	607	3	✔	3	0	0	0.526	0.175	(3.3s)



3 unifying counterexamples found in 25 milliseconds

Succinct explanations

Problem statement

We seek counterexamples that are...

1. easy to understand ✔
   (most general derivation of most specific nonterminal causing ambiguity)
2. efficient to find ✔
   (search outward from conflict state in parser state machine)
   (applicable to LR parser generators, not just LALR)

Time is always against us

More in the paper

We covered...

- properties of good counterexamples
- unifying counterexamples
- product parser
- outward search from conflict state

We did not cover...

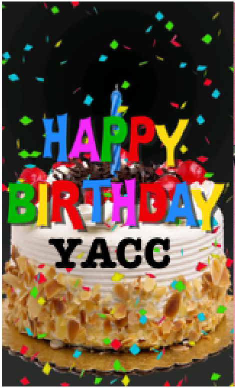
- conflicts not associated with ambiguities
- lookahead-sensitive graph
- shortest lookahead-sensitive path
- implementation optimizations & tradeoffs

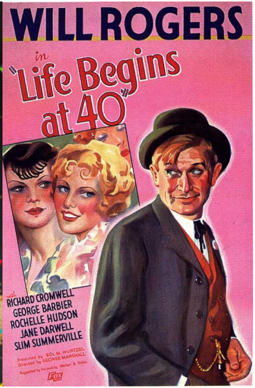
Takeaways

- Easier-to-understand error messages possible for parser generators
- Counterexamples usually found efficiently despite undecidability
- Now part of Polyglot: `https://github.com/polyglot-compiler`
- A new expectation for future parser generators?

Takeaways

- Easier-to-understand error messages possible for parser generators
- Counterexamples usually found efficiently despite undecidability
- Now part of Polyglot: `https://github.com/polyglot-compiler`
- A new expectation for future parser generators?





Finding Counterexamples from Parsing Conflicts

Chin Isradisaikul
chinawat@cs.cornell.edu

Andrew Myers
andru@cs.cornell.edu

Available on GitHub: `http://git.io/vTQp8`

Google: `polyglot java_cup`

Cornell University
Department of Computer Science