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

\[
\begin{align*}
stmt & \rightarrow \text{if} \; expr \; \text{then} \; stmt \; \text{else} \; stmt \\
& \quad | \quad \text{if} \; expr \; \text{then} \; stmt \\
& \quad | \quad expr \; ? \; stmt \; stmt \\
& \quad | \quad \text{arr} \; [\; expr \; ] \; := \; expr \\
expr & \rightarrow \; \text{num} \quad | \quad expr \; + \; expr \\
num & \rightarrow \; \langle\text{digit}\rangle \quad | \quad num \; \langle\text{digit}\rangle
\end{align*}
\]

Yacc

LALR parser generator
Puzzling error messages waste our time

\[
\text{stmt} \rightarrow \text{if expr then stmt else stmt}
\]

| \text{if expr then stmt} |
| \text{expr ? stmt stmt} |
| \text{arr [ expr ] := expr} |

\[
\text{expr} \rightarrow \text{num} \mid \text{expr + expr}
\]

\[
\text{num} \rightarrow \langle \text{digit} \rangle \mid \text{num} \langle \text{digit} \rangle
\]

Warning: *** Shift/Reduce conflict
found in state #10
between reduction on
\[
\text{stmt ::= IF expr THEN stmt •}
\]
and shift on
\[
\text{stmt ::= IF expr THEN stmt • ELSE stmt under symbol ELSE}
\]

Warning: *** Shift/Reduce conflict
found in state #13
between reduction on
\[
\text{expr ::= expr PLUS expr •}
\]
and shift on
\[
\text{expr ::= expr • PLUS expr under symbol PLUS}
\]

Warning: *** Shift/Reduce conflict
found in state #1
between reduction on
\[
\text{expr ::= num •}
\]
and shift on
\[
\text{num ::= num • DIGIT under symbol DIGIT}
\]
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

Why are there 3 parsing conflicts in my tiny grammar?

Finding Counterexamples from Parsing Conflicts 4/34
Succinct explanations?

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

```plaintext
flow: 'F' IF '(' ')' statements elseif

elseif else
    | elseif else

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

elsif: /* empty */
    | elseif '#' ELSEIF statements

The conflicts now are:

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

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

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

// End conflict information for parser
```

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

```plaintext
NUMBER AND NUMBER AND NUMBER
```

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

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

or

```plaintext
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:

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

I already know right recursion solves the problem as `BISON INFORMATION` has said. But I'm looking...
Finding Counterexamples from Parsing Conflicts

In this case, the problem is that the grammar is not LALR(1), so it can't decide what action to take in the conflict. If you give it an input like

```
NUMBER AND NUMBER AND NUMBER
```

it can't decide if it should parse it as equal

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

or

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

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

\[
stmt \rightarrow \text{if } expr \text{ then } stmt \text{ else } stmt \\
\quad | \quad \text{if } expr \text{ then } stmt \\
\quad | \quad expr \ ? \ stmt \ stmt \\
\quad | \quad arr [ expr ] := expr \\
expr \rightarrow \text{num} \ | \ expr + expr \\
num \rightarrow \langle \text{digit} \rangle \ | \ num \langle \text{digit} \rangle \\
\]

ambiguous grammar  
(serious syntactic problem)  
\[\downarrow\]  
want ambiguous counterexample  
\[\downarrow\]  
counterexample should indicate ambiguity in grammar
Good counterexamples are hard to find

\[
\begin{align*}
stmt & \rightarrow \text{if } expr \text{ then } stmt \text{ else } stmt \\
& \quad \mid \text{if } expr \text{ then } stmt \\
& \quad \mid expr \ ? \ stmt \ stmt \\
& \quad \mid arr [ \ expr \ ] := expr \\
expr & \rightarrow \ num \mid expr + expr \\
num & \rightarrow \langle digit \rangle \mid num \langle digit \rangle
\end{align*}
\]

Ambiguous grammar
(serious syntactic problem)
\[
\downarrow
\]
want ambiguous counterexample
\[
\downarrow
\]
counterexample should indicate ambiguity in grammar

Bad news:

Ambiguity detection is undecidable.

Game over?
## Comparison: prior & our approaches

<table>
<thead>
<tr>
<th>approach</th>
<th>ambiguities checked</th>
<th>accurate reports</th>
<th>efficient</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>approximation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NU test [S 07]</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>allows false positives</td>
</tr>
<tr>
<td><strong>brute force</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMBER [S 01]</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>never terminates on unambiguous grammars</td>
</tr>
<tr>
<td>DMS [BPM 04]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>punting on ambiguities</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANTLR 4 [PHF 14]</td>
<td></td>
<td></td>
<td>✔️</td>
<td>can miss unexpected ambiguities</td>
</tr>
<tr>
<td>Elkhound [MN 04]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>our counterexample finder</strong></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
</tbody>
</table>
Succinct explanations

Problem statement

We seek counterexamples that are...

1. easy to understand
2. efficient to find
Good counterexamples

No more concrete than necessary

\[
stmt \rightarrow \text{if } expr \text{ then } stmt \text{ else } stmt
\]
\[
| \text{if } expr \text{ then } stmt
\]
\[
| expr \ ? \ stmt \ stmt
\]
\[
| arr[\ expr\ ] := expr
\]

\[
expr \rightarrow \text{num} \ | \ expr + expr
\]

\[
num \rightarrow \langle\text{digit}\rangle \ | \ num \langle\text{digit}\rangle
\]


too specific, distracting

if \textcolor{yellow}{expr} \textcolor{green}{\text{ then if } \textcolor{yellow}{expr} \text{ then } stmt \text{ else } stmt} \checkmark

general and abstract
use nonterminals when possible
Good counterexamples

Derivation of most specific nonterminal causing ambiguity

\[
\begin{align*}
stmt & \rightarrow \text{if } expr \text{ then } stmt \text{ else } stmt \\
& \quad \mid \text{if } expr \text{ then } stmt \\
& \quad \mid expr \ ? stmt \ stmt \\
& \quad \mid arr [ expr ] := expr
\end{align*}
\]

\[
\begin{align*}
expr & \rightarrow \text{num} \ | \ expr + expr \\
um & \rightarrow \langle \text{digit} \rangle \ | \ num \ \langle \text{digit} \rangle
\end{align*}
\]

\[
stmt \rightarrow^* expr + expr + expr \ ? stmt \ stmt
\]
not specific enough
extra tokens distracting

\[
expr \rightarrow^* expr + expr + expr \checkmark
\]
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
A state contains a collection of production items
The dot (●) indicates progress on completing a production
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**)

```latex
\begin{align*}
stmt & \rightarrow \text{if expr then stmt • else stmt} \; \{\$, \text{else, ...}\} \\
stmt & \rightarrow \text{if expr then stmt •} \; \{\$, \text{else, ...}\}
\end{align*}
```
Parsing conflicts

**shift/reduce conflict:**
shift & reduce possible on same input symbol

**reduce/reduce conflict:**
items ending with $\bullet$ have intersecting lookahead sets
Connection between conflict and ambiguity

\[
\text{conflict} \implies \exists \text{ input taking parser from start to conflict states} \implies \text{parser actions differ at conflict state and diverge for rest of input} \implies \text{keep track of both parses simultaneously to find ambiguous counterexample (unifying counterexample)}
\]

```
if expr then if expr then stmt reduce else stmt
```

Finding Counterexamples from Parsing Conflicts
Simulating copies of parser in parallel

**product parser**: states = Cartesian product of original parser items

\[
\begin{align*}
\text{stmt} & \rightarrow \text{if } \text{expr} \text{ then } \text{stmt} \; \cdot \; \text{else } \text{stmt} \\
\text{stmt} & \rightarrow \text{if } \text{expr} \text{ then } \text{stmt} \; \cdot
\end{align*}
\]

item for 1\textsuperscript{st} parser

item for 2\textsuperscript{nd} parser

ensures: identical input parsed by both copies
**Actions on product parser**

**Intuition:** Keeping the input identical for both copies.

\[
\begin{align*}
\text{stmt} & \rightarrow \text{if} \bullet \ expr \ \text{then} \ \text{stmt} \ \text{else} \ \text{stmt} \\
\text{stmt} & \rightarrow \text{if} \bullet \ expr \ \text{then} \ \text{stmt} \\
\end{align*}
\]

transition on same symbol

\[
\begin{align*}
\text{expr} & \rightarrow \ expr + \ expr \\
\text{stmt} & \rightarrow \text{if} \bullet \ expr \ \text{then} \ \text{stmt} \\
\end{align*}
\]

**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)

\[
\begin{align*}
\text{if} & \quad \text{expr} \\
\text{then} & \quad \text{if} \quad \text{expr} \\
\text{reduce} & \quad \text{stmt} \\
\text{else} & \quad \text{stmt} \\
\end{align*}
\]
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

```
stmt → if expr then • stmt
stmt → if expr then • stmt else stmt
```

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

stmt → if expr then ⋆ stmt
stmt → if expr then ⋆ stmt else stmt

backward transition

stmt → if expr then stmt ⋆
stmt → if expr then stmt ⋆ else stmt

stmt
Our search in action

Stage 1: completing reduce item

reduction on \( \text{stmt} ::= \text{IF expr THEN stmt} \)

shift on \( \text{stmt} ::= \text{IF expr THEN stmt} \text{ ELSE stmt} \)

under symbol ELSE

Start at conflict state; take backward transitions

\[
\begin{align*}
\text{stmt} & \rightarrow \text{if expr then stmt} \\
\text{stmt} & \rightarrow \text{if expr then stmt else stmt}
\end{align*}
\]

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

Finding Counterexamples from Parsing Conflicts
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

\[
\begin{align*}
\text{stmt} & \rightarrow \text{if expr then } \bullet \text{stmt else stmt} \\
\text{stmt} & \rightarrow \text{if expr then } \text{stmt} \\
\text{stmt} & \rightarrow \bullet \text{if expr then stmt else stmt}
\end{align*}
\]

backward production step
Our search in action
Stage 4: completing counterexample

\[
\text{reduction on stmt ::= IF expr THEN stmt • }
\]
\[
\text{shift on stmt ::= IF expr THEN stmt • ELSE stmt}
\]
under symbol ELSE

Keep expanding outward

Complete counterexample:

\[
\text{if expr then if expr then stmt • else stmt}
\]

\[
\text{stmt}
\]
\[
\text{stmt}
\]
\[
\text{stmt}
\]
\[
\text{stmt}
\]
\[
\text{stmt}
\]
Nontermination happens when
- grammar is not ambiguous, and
- production step is taken repeatedly

\[
\text{expr} \rightarrow \bullet \text{expr} + \text{expr} \\
\text{stmt} \rightarrow \text{if} \bullet \text{expr} \text{then stmt}
\]

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 \texttt{expr ::= num •}
and shift on \texttt{num ::= num • ⟨digit⟩}
under symbol \(⟨digit⟩\)

\textbf{Ambiguity detected for nonterminal \texttt{stmt}}

\textbf{Example: expr ? arr [ expr ] ::= num • ⟨digit⟩ ⟨digit⟩ ? stmt stmt}

Derivation using reduction: \texttt{stmt ::= ...}
Derivation using shift : \texttt{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

```expr
\text{arr}[\text{expr}] := \text{num}
```

```
\text{reduce}
```

```
\langle\text{digit}\rangle \quad ? \quad \text{stmt} \quad \text{stmt}
```
Tested on a desktop... our grammars from StackOverflow and StackExchange

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

<table>
<thead>
<tr>
<th>Grammar</th>
<th># nonterms</th>
<th># prods</th>
<th># states</th>
<th># conflicts</th>
<th>Amb?</th>
<th># unif</th>
<th># nonunif</th>
<th># time out</th>
<th>Time (seconds)</th>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>figure1</td>
<td>3</td>
<td>9</td>
<td>24</td>
<td>3</td>
<td>✓</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0.072</td>
<td>figure3</td>
</tr>
<tr>
<td>figure3</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>1</td>
<td>✓</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.010</td>
<td>figure7</td>
</tr>
<tr>
<td>java-ext1</td>
<td>185</td>
<td>445</td>
<td>767</td>
<td>2</td>
<td>✓</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>T/L</td>
<td>java-ext2</td>
</tr>
<tr>
<td>java-ext2</td>
<td>185</td>
<td>445</td>
<td>766</td>
<td>2</td>
<td>✓</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>T/L</td>
<td></td>
</tr>
<tr>
<td>stackexc02</td>
<td>6</td>
<td>11</td>
<td>15</td>
<td>1</td>
<td>✓</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.009</td>
<td></td>
</tr>
<tr>
<td>stackexc03</td>
<td>6</td>
<td>6</td>
<td>10</td>
<td>1</td>
<td>✓</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.017</td>
<td></td>
</tr>
<tr>
<td>stackovf06</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>2</td>
<td>✓</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0.012</td>
<td></td>
</tr>
<tr>
<td>stackovf07</td>
<td>7</td>
<td>12</td>
<td>17</td>
<td>3</td>
<td>✓</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0.028</td>
<td></td>
</tr>
<tr>
<td>stackovf10</td>
<td>6</td>
<td>12</td>
<td>27</td>
<td>1</td>
<td>✓</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.012</td>
<td></td>
</tr>
<tr>
<td>stackovf11</td>
<td>9</td>
<td>20</td>
<td>53</td>
<td>19</td>
<td>✓</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.140</td>
<td></td>
</tr>
</tbody>
</table>

Finding Counterexamples from Parsing Conflicts
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