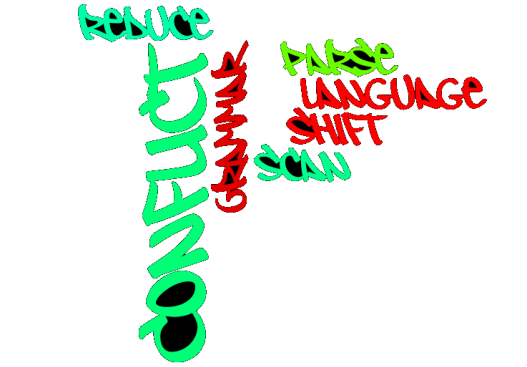
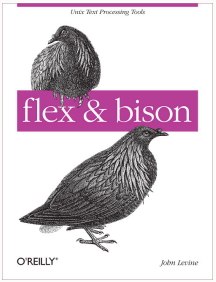
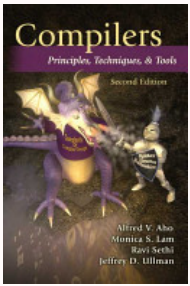
Ambiguity and Conflict
CpSc 8270 Language Translation

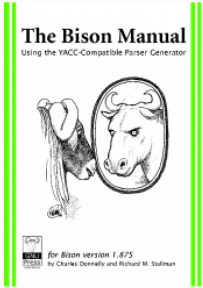
Brian A. Malloy



Resources







1. Overview

- Conflicts occur when bison reports one of two conflicts:
  - shift/reduce
  - reduce/reduce
- These conflicts are reported in: name.output obtained by using -v option to bison
- Even though bison reports where they are, it can still be a challenge to determine why and how to fix
- In examples, capital letters are terminals, lower case letters are non-terminals

2. The Pointer Model

- Shows a pointer moving through grammar
- The pointer starts at the start rule

```
%token A B C
%%
start: ↑ A B C
```

- If A and B are read, move pointer:

```
%token A B C
%%
start: A B ↑ C
```


2.1. More than one Pointer

- Example, assume we read A, B:

```
%%
start: x | y
x: A B ↑ C D
y: A B ↑ E F
```

- Pointers can disappear in two ways:
  1. A rule doesn't work; Eg next token C

     ```
     %%
     start: x | y
     x: A B C ↑ D
     y: A B E F
     ```

  2. Ponters merge into a common subrule

- For merge example, assume we read A

```
start: x | y
x: A ↑ B z R
y: A ↑ B z S
z: C D
```

- After A B C, there is only one pointer:

```
start: x | y
x: A B z R
y: A B z S
z: C ↑ D
```

- After A B C D, there are two pointers

```
start: x | y
x: A B z ↑ R
y: A B z ↑ S
```

2.2. Reductions

- If there is only one pointer at the end of a rule, we reduce. E.g., assume an A is read:

```
%%
start: x | y
x: A ↑
y: B
```

- After A, there is only one pointer in x, and rule x is reduced.
- Similarly, if B is read, there is only one pointer in y, and rule y is reduced.




3. Kinds of Conflicts

- Reduce/reduce conflict: two pointers at the end of two rules.

```
%%
start: x | y
x: A ↑
y: A ↑
```

- Never a conflict if only one arrow:

```
%%
start: x | y
x: z R
y: z S
z: A B ↑
```

- After z is reduced, there are two pointers. Why?

- Shift/reduce conflict: One pointer wants to shift, the other pointer wants to reduce:

```
%%
start: x | y
x: A ↑ R
y: A ↑
```

3.1. Must account for lookahead

- This is a reduce/reduce conflict:

```
%%
start: x B | y B
x: A ↑
y: A ↑
```

- But this is not because bison has one symbol lookahead:

```
%%
start: x B | y C
x: A ↑
y: A ↑
```

- The following would not be a conflict in a parser with two tokens lookahead:

```
%%
start: x B C | y B D
x: A ↑
y: A ↑
```




3.2. Sometimes the parse tree helps

- Consider the following s/r conflict:

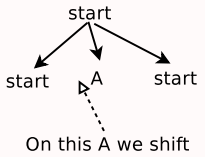
```
%%
start: start A start | A
```

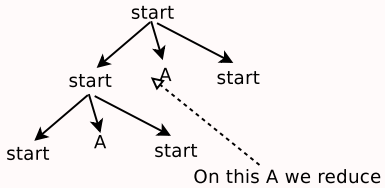
- Which bison reports as:

```
State 5
    1 start: start . A start
    1      | start A start .
    A  shift, and go to state 4
    A          [reduce using rule 1 (start)]

Where rule 1 is:
    1 start: start A start
```

3.3. Parse trees illustrate s/r





4. Bison & Ambiguity

- In the face of ambiguity, bison will make a choice, unless the user specifies otherwise. The user can set the precedence, refactor the grammar to remove conflicts, or suppress conflict warnings using `%expect n`, `%expect-rr n`.
- Default action for Shift/Reduce conflict: `shift`.
- Default action for Reduce/Reduce conflict: choose the rule that appears first in the grammar.

5. Parser States

- You can generate name.output by supplying the -v flag to bison (-v ⇒ verbose)
- name.output is a description of the state machine generated by bison, including:
  - Listing of the grammar rules
  - Rules that are never used
  - Summary of the conflicts
  - All of the parser states

5.1. States in name.output

- For each state, bison lists
  - The rules and positions that correspond to the state,
  - the shifts and reductions the parser will make when it reads various tokens in the state,
  - and the state it will switch to after a reduction produces a non-terminal in that state,
  - the conflicts in each state.




5.2. What are States

- Each state corresponds to a unique combination of possible pointers in your bison grammar.
- Bison assigns a number to each state:

start: A ⟨state 1⟩ B ⟨state 2⟩ C

- Numbers can differ across bison implementations; the actual number isn't significant.




5.3. Different streams can ⇒ same state

- Different input streams can corresond to the same state when they correspond to the same pointer:

```
start: threeAs;
threeAs: A ⟨state 1⟩ A ⟨state 2⟩ A ⟨state 3⟩
       | {empty;}
```

- In the above example, state 1 corresponds to 1, 4, 7,... A's
- state 2 corresponds to 2, 5, 8,... A's
- state 3 corresponds to 3, 6, 9,... A's

```
start: threeAs X;
       | twoAs Y;
threeAs: A A A threeAs;
       | {empty;}
twoAs: A A twoAs;
       | {empty;}
```

- The above accepts multiples of 3 As, followed by X, or 2 As followed by Y.
- Without the X or Y the grammar would have a conflict.
- There would also be a conflict if we had used left recursion.

6. Tracing reduce/reduce

```
start: a Y | b Y;
a:     X ;
b:     X ;
```

- We will trace bison as it parses the above ambiguous grammar
- assume the input is X Y ⟨eof⟩
- bison always starts at state 0, shift 0; i.e., push 0 onto the stack.




```
(1) Starting parse
(2) Entering state 0
(3) Reading a token: –(end of buffer or a NUL)
(4) X
(5) –accepting rule at line 10 ("X")
(6) Next token is token X ()
(7) Shifting token X ()
(8) Entering state 1
```

- (1) Bison announces the start of the parse
- (2) Bison always begins in state 0
- (3) Bison is about to read a token
- (4) reads X
- (5) This line is debug info from Flex
- (8) Bison enters state 1, stack is:
  (1, x) (0, -)

```
state 1
    3 a: X ↑ (arrow shows where we are before next token)
    4 b: X ↑
    Y             reduce using rule 3 (a)
    Y             [reduce using rule 4 (a)]
    $default      reduce using rule 3 (a)
```

```
(1) Reducing stack by rule 3 (line 12):
(2) $1 = token X ()
(3) -> $$ = nterm a ()
(4) Stack now 0
(5) Entering state 3
```

- (1) line 12 is in parse.y
- (3) nterm a is non-terminal a
- (5) stack is: (3, a) (0, -)

(1) Reading a token: –accepting rule at line 12 ("
(2) ")
(3) –(end of buffer or a NUL)
(4) Y
(5) –accepting rule at line 11 ("Y")
(6) Next token is token Y ()
(7) Shifting token Y ()
(8) Entering state 6

- (1), (3), (5) is from Flex
- (7) and (8) push (6, Y) onto stack
- stack is:
  (6, Y)
  (3, a)
  (0, -)




```
state 6
    1 start: a Y ↑
    $default      reduce using rule 1 (start)
```

```
(1) Reducing stack by rule 1 (line 9):
(2)      $1 = nterm a ()
(3)      $2 = token Y ()
(4) -> $$ = nterm start ()
(5) Stack now 0
(6) Entering state 2
```

- recall that stack is: (6, Y) (3, a) (0, -)
- (1) reduce by rule 1 pops Y and a off stack
- $default & (4) pushes start onto stack
- (5) says stack is state 0, start: goto state 2

```
state 0
    0 $accept: ↑ start $end
    ...
    start:      goto state 2
```

- Above, on **start** goto state 2
- stack is:
  (2, start),
  (0, -)




state 2
    0 $accept: ↑ start $end
    $end shift, and goto state 5

- Bison now reads $eof, shifts it onto the stack, and goes to state 5
- stack is:
  (5, $end)
  (2, start)
  (0. -)

(1) –EOF (start condition 0)
(2) Now at end of input.
(3) Shifting token $end ()
(4) Entering state 5


- Recall, stack is:
  (5, $end)
  (2, start)
  (0. -)

```
(1) Entering state 5
(2) Stack now 0 2 5
(3) Cleanup: popping token $end ()
(4) Cleanup: popping nterm start ()
(5) Syntactically correct.
```

6.1. Some notes on reduce/reduce

- The arrow (bison uses a dot) shows where we are in the grammar before reading the next token.
- For **reduce/reduce** conflicts, the two arrows are always at the end of the rules.
- In a conflict, the rule not used is shown in brackets
- In state 1, bison chose to reduce to a by rule 3 because with **reduce/reduce** conflicts it chooses the earliest one in the grammar.




7. shift/reduce

To Identify a shift/reduce conflict:

- Find the shift/reduce conflict in name.output
- Identify the reduce rule
- Identify the relevant shift rule(s).
- Find the state the reduce rule reduces to
- Deduce the token stream that will produce the conflict

The next grammar has a shift/reduce conflict:

```
start: x | y R;
x:     A R ;
y:     A ;
```



- Assume input is A R ⟨eof⟩.
- Bison complains in state 1:
- Bison resolves shift/reduce in favor of the shift, so in this case if it receives an R, it shifts to state 5.

```
state 0
        0 $accept: ↑ start $end
        A shift, and go to state 1 ...

state 1
        3 x: A . R
        4 y: A .
        R shift, and go to state 5
        R [reduce using rule 4 (y)]
```

8. How to Fix Conflicts

- Start with the rules with most conflicts: resolving major conflicts frequently eliminates minor conflicts.
- Determine the cause of the conflict: is it the language definition or the grammar.
- You can improve the language design by changing the language definition.
- We now describe grammar constructs that produce conflicts, and suggest fixes.

8.1. if/else Conflict

```
stmt    : IF expr stmt
        | IF expr stmt ELSE stmt
        | expr
        |
        ;
expr    : IDENT
        ;
```

```
state 7
        1 stmt: IF expr stmt ↑
        2 | IF expr stmt ↑ ELSE stmt
        ELSE shift, and go to state 8
        ELSE [reduce using rule 1 (stmt)]
        $default reduce using rule 1 (stmt)
```




- This is the classic *dangling else* ambiguity
- The fix is to tell bison that the if part has lower precedence than the else.

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE
%%
 stmt    : IF expr stmt %prec LOWER_THAN_ELSE
         | IF expr stmt ELSE stmt
         | expr
         |
         ;
 expr    : IDENT
         ;
```

8.2. Loop Within a Loop

```
start          : outerList Z ;
outerList      : outerList outerListItem
               | ;
outerListItem  : innerList ;
innerList      : innerList innerListItem
               |  ;
innerListItem  : I ;
```

```
state 2
        1 start: outerList ↑ Z
        2 outerList: outerList ↑ outerListItem
        Z shift, and go to state 4
        Z [reduce using rule 6 (innerList)]
        $default reduce using rule 6 (innerList)
        outerListItem go to state 5
        innerList go to state 6
```

- The grammar, as written, does not work.
- It is intended to accept a possibly empty list of Is followed by a Z, and reject the empty string.
- But the generated parser gets confused on empty: it can either shift or reduce: in this case it loops indefinitely.
- The fix is to eliminate one of the loops.

```
start          : outerList Z ;
outerList      : outerList outerListItem
               | ;
innerListItem  : I ;
```

8.3. Expression Precedence

```
expr    : expr '+' expr
        | expr '-' expr
        | expr '*' expr
        ...
```

- If you forget to define the precedence, you get a boatload of shift/reduce conflicts
- The fix is to use %left or %right to define precedence




8.4. Overlap of Alternatives

```
person  : girls
        | boys
        ;
girls   : ALICE
        | BETTY
        | CHRIS
        | DARRYL
        ;
boys    : ALLEN
        | BOB
        | CHRIS
        | DARRYL
        ;
```

- CHRIS and DARRYL ⇒ reduce/reduce conflict: bison can't tell if they're boys or girls




- Best way to fix this:

```
person   : girls | boys | either ;
girls    : ALICE
         | BETTY
         ;
boys     : ALLEN
         | BOB
         ;
eigher   : CHRIS
         | DARRYL
         ;
```

8.5. Limited Lookahead

```
stmt          : command opt_keyword '(' id_list ')'
              ;
opt_keyword   : '(' IDENT ')'
              |
              ;
id_list       : id_list IDENT ','
              | IDENT
              ;
command       : GOTO
              ;
```

- This shift/reduce conflict is caused because bison has only one symbol lookahead
- The fix is to *flatten* the description




```
stmt      : command '(' id_list ')' '(' id_list ')'
          | command '(' id_list ')'
          ;
id_list   : id_list IDENT ','
          | IDENT
          ;
command   : GOTO
          ;
```