

Section Handout: LR(1) and Precedence

Problem 1: Understanding Ambiguity in `yacc`

A CS143 student tests the "classical" desk calculator written in `yacc`. This desk calculator reads integer **NUMBERS** as well as the characters `'+' , '-' , '*' , '/' , '(' , ')' ,` and `'\n'` as tokens from the input. It uses semantic actions to compute and output the value of simple arithmetic expressions entered by the user.

Unfortunately, the student made a mistake when entering the `yacc` description for the calculator: he simply forgets to assign a precedence to `'/'`. Here is the (buggy) code:

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double    // so division doesn't truncate
%}
%token NUMBER
%left '+' '-'
%left '*'
%right UnaryMinus

%%
lines : lines expr '\n' { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;

expr  : expr '+' expr { $$ = $1 + $3; }
      | expr '-' expr { $$ = $1 - $3; }
      | expr '*' expr { $$ = $1 * $3; }
      | expr '/' expr { $$ = $1 / $3; }
      | '(' expr ')' { $$ = $2; }
      | '-' expr %prec UnaryMinus { $$ = -$2; }
      | NUMBER
      ;

%%
```

Considering `yacc`'s rules for using precedence and associativity, and its default conflict resolution strategy of preferring shift to reduce, what does this buggy version of the calculator output for the following inputs?

- $3+5*2/2$
- $12/4/2$
- $-16/4-2$
- $16*-4/2-3$

Assume now that `'/'` has been given the same precedence as `'*'`, but that all references to `UnaryMinus` have been removed.

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double    // so division doesn't truncate
%}
%token NUMBER
%left '+' '-'
%left '*' '/'

%%
lines : lines expr '\n' { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;

expr  : expr '+' expr { $$ = $1 + $3; }
      | expr '-' expr { $$ = $1 - $3; }
      | expr '*' expr { $$ = $1 * $3; }
      | expr '/' expr { $$ = $1 / $3; }
      | '(' expr ')' { $$ = $2; }
      | '-' expr { $$ = -$2; }
      | NUMBER { $$ = $1; }
      ;

%%
```

What does the calculator output for:

- $10-8*-3$
- $12*-2/8$

Assume now that the precedences are set to:

```
%left '+' '-'
%nonassoc '*' '/'
%right unaryminus
```

What does the calculator output for:

- $10-8*-3$
- $3*3*3$

Lastly, assume the precedence for `*` and `/` has been set to “right”. What does the calculator output for:

- $16/4/2$

One of these strings should have caused an error. Using bison, how would you find out what state the error occurred in? How would you view that state?

Problem 2: LR(1) Configuring Sets

Given the following already augmented grammar:

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow AB \mid AA \mid bC \\ A &\rightarrow bCa \mid b \\ B &\rightarrow Bd \mid \varepsilon \\ C &\rightarrow c \end{aligned}$$

Draw the goto graph including just those states of an **LR(1)** parser that are pushed on the parse stack during a parse of the input **bdd** (this will be a subset of all configuring sets). If you accidentally construct irrelevant states, cross them out. Do not add or change any productions in the grammar.

Solution 1: Understanding Ambiguity in `yacc`

What happens when precedence information for `/` is missing?

- $3+5*2/2$ evaluates to 8

Because `*` has higher precedence than `+`, five tokens are shifted, so that the state of the stack looks like:

`3 + 5 * 2`

Here `yacc` could reduce, or it can shift in the `/` and reduce later. By omitting precedence information, we're forcing `yacc` to resolve a shift-reduce conflict, and we know full well that it always chooses the shift. That means everything the entire expression gets shifted in before any significant reductions take place (other than the `expr` \rightarrow `number` reductions, which aren't computationally interesting here)

<code>3 + 5 * 2 / 2</code>	(reduce <code>expr</code> \rightarrow <code>expr / expr</code> : replace <code>2 / 2</code> with <code>1</code>)
<code>3 + 5 * 1</code>	(reduce <code>expr</code> \rightarrow <code>expr * expr</code> : replace <code>5 * 1</code> with <code>5</code>)
<code>3 + 5</code>	(reduce <code>expr</code> \rightarrow <code>expr + expr</code> : replace <code>3 + 5</code> with <code>8</code>)
<code>8</code>	(reduce <code>lines</code> \rightarrow <code>expr '\n'</code> : print <code>8</code>)

So, in this case, the decisions `yacc` makes for us don't interfere with the evaluation.

- $12/4/2$ evaluates to 6

Again, no precedence or associativity information on `/` translates into a preference for a shift over a reduction. Everything gets shifted in before any interesting reductions take place:

<code>12 / 4 / 2</code>	(reduce <code>expr</code> \rightarrow <code>expr / expr</code> : replace <code>4 / 2</code> with <code>2</code>)
<code>12 / 2</code>	(reduce <code>expr</code> \rightarrow <code>expr / expr</code> : replace <code>12 / 2</code> with <code>6</code>)
<code>6</code>	(reduce <code>lines</code> \rightarrow <code>expr '\n'</code> : print <code>6</code>)

- $-16/4-2$ evaluates to -8

<code>- 16 / 4 - 2</code>	(reduce <code>expr</code> \rightarrow <code>expr - expr</code> : replace <code>4 - 2</code> with <code>2</code>)
<code>- 16 / 2</code>	(reduce <code>expr</code> \rightarrow <code>expr / expr</code> : replace <code>16 / 2</code> with <code>8</code>)
<code>- 8</code>	(reduce <code>expr</code> \rightarrow <code>- expr</code> : replace <code>- 8</code> with <code>-8</code>)
<code>-8</code>	(reduce <code>lines</code> \rightarrow <code>expr '\n'</code> : print <code>-8</code>)

Note that the negation doesn't actually take place until the very end; at no point during the parse is there ever a `yyval` value of `-16` anywhere on the stack.

- $16 * -4 / 2 - 3$ evaluates to 64

$16 * - 4 / 2 - 3$	(reduce expr \rightarrow expr - expr : replace $2 - 3$ with -1)
$16 * - 4 / -1$	(reduce expr \rightarrow expr / expr : replace $4 / -1$ with -4)
$16 * - -4$	(reduce expr \rightarrow - expr : replace $- -4$ with 4)
$16 * 4$	(reduce expr \rightarrow expr * expr : replace $16 * 4$ with 64)
64	(reduce lines \rightarrow expr '\n': print 64)

be careful to distinguish between a lexical minus sign (which has no meaning all by itself) and a negative number (where unary minus is understood to mean negative)

What happens when precedence information for unary minus is missing?

Unfortunately, precedence policy is then dictated by the precedence and associativity rules of binary infix minus.

- $10 - 8 * -3$ evaluates to 34

$10 - 8$	(* is shift character, and has higher precedence than $-$ on stack, so reduction possibility is ignored)
$10 - 8 *$	($-$ is shift character, but there's no conflict, because top of parse stack isn't reducible; no choice but shift and then shift again)
$10 - 8 * - 3$	(reduce expr \rightarrow - expr : replace $- 3$ with -3)
$10 - 8 * -3$	(reduce expr \rightarrow expr * expr : replace $8 * -3$ with -24)
$10 - -24$	(reduce expr \rightarrow expr - expr : replace $10 - -24$ with 34)
34	(reduce lines \rightarrow expr '\n': print 34)

- $12 * -2 / 8$ evaluates to -3

$12 * - 2$	(no interesting reduction possibilities until the 2 has been shifted in. Now (sadly) the lookahead character of $/$ is compared to the rightmost token of the reducible handle, and $/$ wins, because the $-$ token has lower precedence; we shift (and then shift again))
$12 * - 2 / 8$	(reduce expr \rightarrow expr / expr : replace $2 / 8$ with $.25$)
$12 * - .25$	(reduce expr \rightarrow - expr : replace $- .25$ with $-.25$)
$12 * -.25$	(reduce expr \rightarrow expr * expr : replace $12 * -.25$ with -3)
-3	(reduce lines \rightarrow expr '\n': print -3)

What happens when multiplication and division are set not to associate?

- $10-8*-3$ evaluates to 34 (there is no change because it does not involve association of * or /)

10 - 8	(* is shift character, and has higher precedence than - on stack, so reduction possibility is ignored)
10 - 8 *	(- is shift character, but there's no conflict, because top of parse stack isn't reducible; no choice but shift and then shift again)
10 - 8 * - 3	(reduce expr \rightarrow - expr : replace - 3 with -3)
10 - 8 * -3	(reduce expr \rightarrow expr * expr : replace 8 * -3 with -24)
10 - -24	(reduce expr \rightarrow expr - expr : replace 10 - -24 with 34)
34	(reduce lines \rightarrow expr '\n': print 34)

- $3*3*3$ causes a syntax error. Bison prevents the * operator from associating in a chain.

What happens when multiplication and division are set to associate right?

- $16/4/2$ evaluates to 8

16/4	(/ is shift character, and right precedence causes us to shift)
16/4/2	(reduce expr \rightarrow expr / expr : replace 4/2 with 2)
16/2	(reduce expr \rightarrow expr / expr : replace 16/2 with 8)

How do you view the state at which a syntax error is caught?

First, set the yydebug variable in parser.y to true. Run make to get your parser code up to date, then run your parser on the file containing the syntax error. The extra output produced by yydebug should tell you the state at which the error occurred. That state is then described in the file y.output, which was made by bison at compile time.

Solution 2: LR(1) Configuring Sets

Given the following already augmented grammar:

```

S'  $\rightarrow$  S
S  $\rightarrow$  AB | AA | bC
A  $\rightarrow$  bCa | b
B  $\rightarrow$  Bd |  $\epsilon$ 
C  $\rightarrow$  c

```

Draw the goto graph including just those states of an **LR(1)** parser that are pushed on the parse stack during a parse of the input **bdd** (this will be a subset of all configuring sets). If you accidentally construct irrelevant states, cross them out. Do not add or change any productions in the grammar.

