Lecture 10: LR parsing and resolving conflicts

- What are conflicts
- Example 1: a simple, unambiguous grammar
  - ocamlyacc output
  - Using parse trees to understand conflict
  - Fixing conflict
- Example 2: ambiguous grammar for conditional expressions
  - Eliminating conflicts using %prec declarations

Conflicts

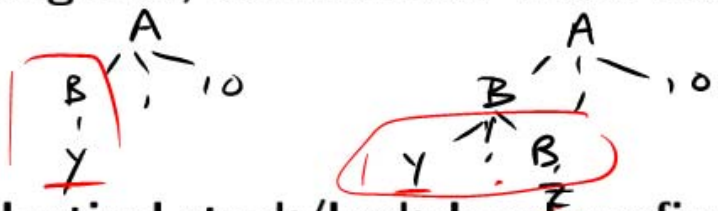
- ocamlyacc generates tables saying what action to take at each point in parse
  - Method is called "LALR(1)"
  - "LR(1)" is a similar, but somewhat more powerful, method — will often use "LR(1)" and "LALR(1)" as synonyms.
- Not every grammar can be parsed using this method.
  - Problem is *always* that ocamlyacc cannot decide on the proper action in some cases
  - "Shift/reduce conflict" — cannot decide whether to shift or reduce
  - "Reduce/reduce conflict" — knows to reduce, but can't decide which production to use

Example 1

- $A \rightarrow B\ ,\ int$
  $B \rightarrow id \mid id\ ,\ B$

A → B int
B → id , | id , B

- Grammar is unambiguous, but consider these two inputs:
  - x,y,10
  - x,y,z,10

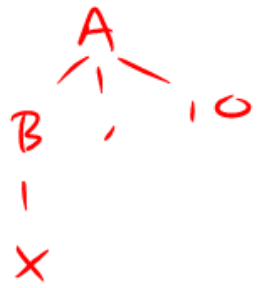


- Both lead to an identical stack/lookahead configuration, but the correct action in one case is shift and in the other is reduce.
- Look at s-r parse, and at two parse trees.

A → B , int
B → id | C
C → id , B

x , 10

x y , 10

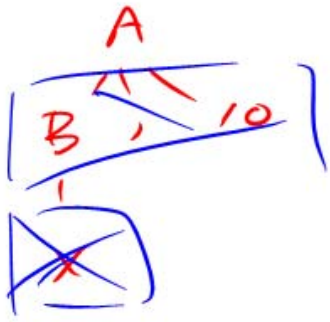


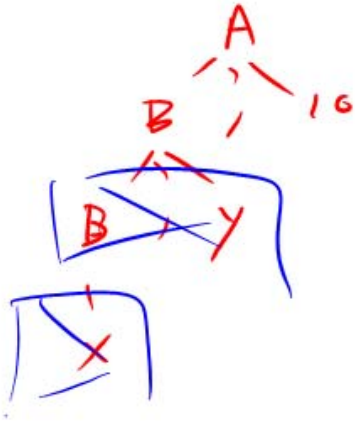
A → B , int

B → id | B , id

x, 10

x, y, 10





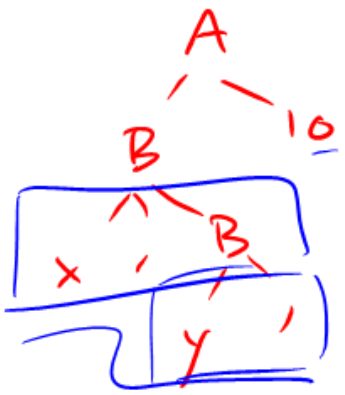
A → B int

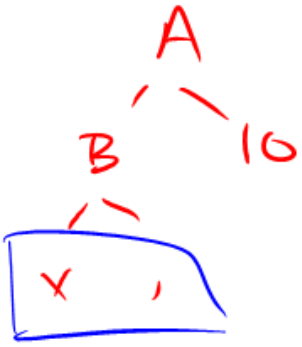
B → id , | id , B

x , 10

x , y , 10

x , y , z , 10





Rule: id , on stack:

Sh on id

R on int

x , B on stack -

R on int

Example 1 (cont.)

- **Presented to ocamlyacc:**

```
%token int id comma
%start A
%type <int> A
%%
A: B comma int   {0}
B: id            {0}
 | id comma B    {0}
```

- **Using "ocamlyacc -v", file simple.output contains:**

```
3: shift/reduce conflict (shift 6, reduce 2) on comma
state 3
B : id .  (2)
B : id . comma B  (3)

comma  shift 6
```

Example 1 (cont.)

- **One way to fix grammar:**

$$A \rightarrow B\ int$$
$$b \rightarrow id\ ,\ |\ id\ ,\ B$$

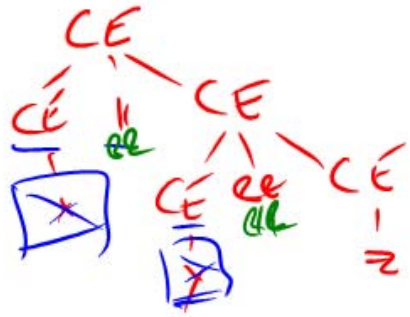
Example 1 (cont.)

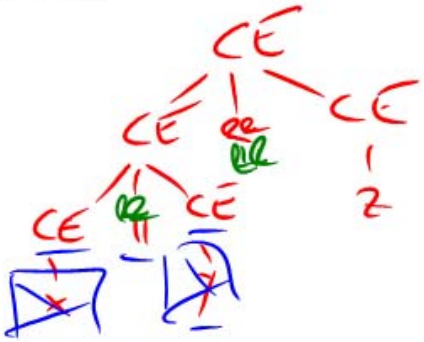
- **Another way to fix grammar:**

$$A \rightarrow B\ ,\ int$$
$$b \rightarrow id \mid B\ ,\ id$$

Example 2

- **Ambiguous grammar for conditional expressions:**
- $CondExpr \rightarrow \quad id \mid CondExpr \;||\; CondExpr$
  $\mid CondExpr \;\&\&\; CondExpr \mid \;!\; CondExpr$
- **Consider this input:**
  - `x || y && z`
- **Leads to a stack/lookahead configuration in which shifting and reducing both work, but produce different parse trees.**
- **Look at s-r parse, and at two parse trees.**





Example 2 (cont.)

- **ocamlyacc -v output contains:**

```
10: shift/reduce conflict (shift 7, reduce 2) on and
10: shift/reduce conflict (shift 8, reduce 2) on or
state 10
CondExpr : CondExpr . or CondExpr  (2)
CondExpr : CondExpr or CondExpr .  (2)
CondExpr : CondExpr . and CondExpr  (3)

and  shift 7
or  shift 8
$end  reduce 2
```

Example 2 (cont.)

- **One way to resolve conflict: fix grammar.**
- **Use "stratified grammar," as for arithmetic expressions:**

$$CondExpr \rightarrow CondTerm \mid CondExpr \;||\; CondTerm$$
$$CondTerm \rightarrow CondPrimary \mid CondTerm \;\&\&\; CondPrimary$$
$$CondPrimary \rightarrow id \mid !\, CondPrimary$$

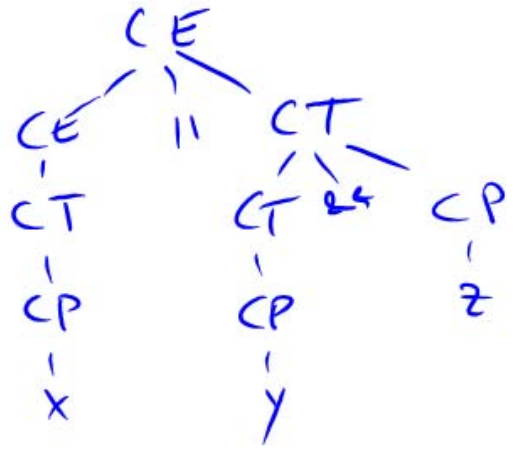
x || y && z

Ident on stack: R CP → id

CP " ", lookahead || or &&:
R CT → CP

CT " ", lookahead ||:
R CE → CT

lookahead &&: sh



Example 2 (cont.)

- Another way to resolve conflict: precedence declarations.
- Suppose t_1 is the topmost terminal symbol on the stack, and t_2 is the lookahead symbol. Then:
  - If t_1, t_2 appear in the same `%left` declaration, then reduce
  - If t_1, t_2 appear in the same `%right` declaration, then shift
  - If t_1 appears in a declaration *before* t_2, shift
  - If t_1 appears in a declaration *after* t_2, reduce

%left ||
%left &&

%right

%nonassoc

CE : CE && CE
| CE || CE
| id

Example 2 (cont.)

- **Use the ambiguous grammar, but add these declarations:**

```
%left or
%left and
```

- `x || y && z` **is now handled correctly.**

Example 2 (cont.)

- **However, ocamlyacc still reports conflicts. Verbose output:**

```
6: shift/reduce conflict (shift 7, reduce 4) on and
6: shift/reduce conflict (shift 8, reduce 4) on or
state 6
CondExpr : CondExpr . or CondExpr  (2)
CondExpr : CondExpr . and CondExpr  (3)
CondExpr : not CondExpr .  (4)

and  shift 7
or  shift 8
$end  reduce 4
```

- **Problem is that we didn't resolve ambiguity involving "!". Add "%nonassoc not" after above two lines.**

Notes on shift-reduce parsing and resolving conflicts in ocamlyacc - Supplementary notes for lecture 10, 2/12/10

CS 421, Prof. Kamin

In this note, we go through four grammars and show how to resolve conflicts reported by ocamlyacc. For each grammar, we do the following:

1. Run ocamlyacc. To simplify the process, we give {0} as the semantic action for each rule. Each of the grammars has a conflict. (In every case, it is a shift/reduce conflict; we have no reduce/reduce conflicts.)

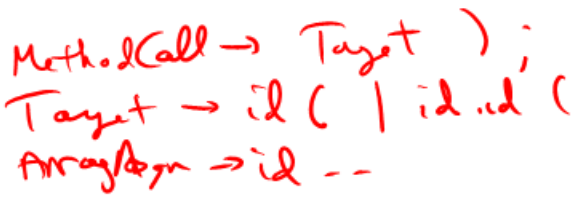
2. We look at the ocamlyacc verbose ("ocamlyacc -v") output, to see where the conflict is.

3. Based on that output, we attempt to construct an example that shows the conflict. A conflict means that an input string leads to a stack/lookahead configuration in which the parser cannot make a clear choice between shifting and reducing. If the grammar is ambiguous, we will just show one input that leads to such a configuration, and which therefore has multiple parses. If the grammar is unambiguous, we will show two inputs that lead to the same configuration, but differ after the lookahead symbol.

4. Using our understanding of the source of the conflict, we will resolve it in one of the three ways described in class: modify the grammar; use precedence and associativity declarations; or do nothing. We will show first that this resolves the conflict illustrated by the example we obtained in step 3; running ocamlyacc proves that we did in fact eliminate all conflicts.

Grammar 1

Stmt → MethodCall | ArrayAsgn
MethodCall → Target ( ) ;
Target → id | id . id
ArrayAsgn → id (int) = int ;



This grammar is unambiguous. ocamlyacc reports a shift/reduce conflict.

(This is the exact input we presented to ocamlyacc:

```
%token oparen cparen id semic dot int equal
%start STMT
%type <int> STMT
%%
STMT : METHODCALL {0} | ARRAYASGN {0}
```

```
METHODCALL : TARGET oparen cparen semic  {0}
TARGET : id  {0}  | id dot id {0}
ARRAYASGN : id oparen int cparen equal int semic {0}
```

We will not show this for the other grammars we study. Note that we used semantic action {0} for every production, because we were only interested in seeing the parsing conflicts.)

Specifically, the verbose output from ocamlyacc includes the following. (The entire output file, example1.output, is about 160 lines long; the interesting part starts where it gives the conflict, which happens to be at line 43, and ends when the next "state" is shown):

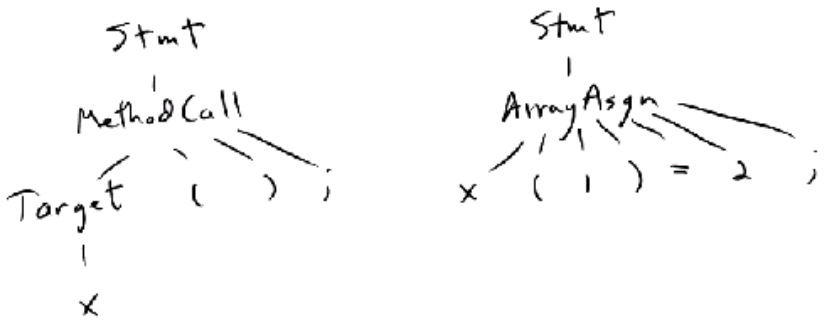
```
3: shift/reduce conflict (shift 8, reduce 4) on oparen
state 3
      TARGET : id .  (4)
      TARGET : id . dot id  (5)
      ARRAYASGN : id . oparen int cparen equal int semic  (6)

      oparen  shift 8
      dot  shift
```

f ( ) - -
f (5 - -

This says: If there is an id on the stack and the lookahead symbol is '(', the parser cannot choose between shifting and reducing. We know the lookahead symbol is '(' because it says that on the first line. We know the top symbol on the stack is id because the three productions have a period after an id. Most importantly, in the line "TARGET : id .", the period is *at the end* of the production, while in the other two lines it is in the middle. This is where the shift/reduce conflict comes in: there can be an input where, with id on the stack and '(' in the input, the correct action is to reduce Target → id, and another input that leads to the same situation but where shift is the correct action.

Before proceeding, see if you can look at the grammar and find two such inputs.

Looking back at the grammar, it is easy to see where an id on the stack should be reduced using Target→ id. Target is used in MethodCall, and is followed by a '('. So we can consider a simple sentence like "x();". It is also easy to see where this same configuration requires a shift action: in the rule for ArrayAsgn, an id is followed by '('; the '(' should be shifted so that, eventually, this production can be used. This suggests a sentence like "x(1) = 2;". Let's look at the parse trees for these two inputs:

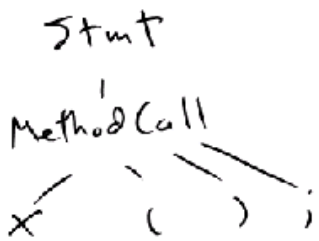


We can easily see where the conflict comes in: on the left, the x needs to be reduced to put Target on the stack, on the right, the '(' should be shifted. (Keep in mind that reductions always consume the *top* of the stack. On the left, we can't shift '(' and then *later* do the Target → id reduction. It has to be done as soon as the '(' is the lookahead symbol.)

We can resolve the conflict here by observing that the Target → id production is what's causing the problem. Suppose we eliminate it and just put the id directly in the MethodCall production:

Stmt → MethodCall | ArrayAsgn
MethodCall → Target ( ) ; | id () ;
Target → id . id
ArrayAsgn → id (int) = int ;

Now, the parse tree for "x();" is



and there is no conflict: with x on the stack and '(' as the lookahead symbol, the correct action in both cases is to shift. (Note that the other kind of target, id . id, does not present a problem; a period in the input should always be shifted; while, if the stack contains "id . id" and the lookahead is '(', the correct action is reduce Target → id . id.) Running ocamlyacc on the modified grammar shows that we have eliminated the conflict.

More on conflicts

- Posted supplementary notes discuss four grammars that have conflicts, and how to resolve them.
- All are relevant to the current MP.