CS 421 Lecture 9: LR parsing and resolving conflicts

- Review
  - Top-down parsing
  - Bottom-up parsing

- Lecture outline
  - What are conflicts?
  - Using parse trees to understand conflicts
  - Fixing conflicts
  - Eliminating conflicts using %prec declarations

Review: Top-down parsing

- A.K.A. recursive descent
  - One parse function per non-terminal
- Ambiguity
- LL(1) condition
- Parse tree construction
  - Precedence
  - Associativity
- How do we choose which production to apply?

Review: Bottom-up parsing

- A.K.A. shift-reduce
  - Keep a stack of partial parse trees
  - Automatic parser generation (ocamlyacc)
- Actions
  - Shift
  - Reduce
  - Accept
  - Reject
- How to decide which action to take?
  - Today: dealing with conflicts

Conflicts

- Big question: how to choose whether to shift or reduce?
  - ocamlyacc uses a method – called *LALR(1)* – to construct tables that say which action to take
- There are times when there is no good way to make this decision
  - ocamlyacc will reject grammar and give an error message
- In bottom-up parsing, these are called *conflicts*
  - As with top-down parsing, these problems can sometimes be resolved by modifying the grammar.

Conflicts

- Ocamlyacc generates tables saying which action to take at each point in the 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" – know to reduce, but can't decide which production to use

Example 1

- Grammar Language??
  - $A \rightarrow B, id$
  - $B \rightarrow id \mid id, B$
- Unambiguous, but consider 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 the two parse trees, and the s-r derivations.

Example 1: parse trees

- Grammar:
  - $A \rightarrow B, id$
  - $B \rightarrow id \mid id, B$
- Parse tree:

x,y,10 x,y,z,10

Example 1: derivations

- Grammar:
  - $A \to B, id$
  - $B \to id \mid id, B$
- Derivation:

Action	Stack	Input	Action	Stack	Input
S		x,y,10	S		x,y,z,10

Example 1: ocamlyacc

- 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)
```

Example 1b

- One way to fix grammar:
  - $A \rightarrow B\ int$
  - $B \rightarrow id\ ,\ |\ id\ ,\ B$
- Conflict resolution:
  - If id on stack – shift
  - If id + ‘,’ on stack, and *lookahead* is:
    - id – shift
    - number – reduce
    - comma – reject

Example 1b: parse trees

- Grammar:
  - $A \rightarrow B\ int$
  - $B \rightarrow id\ ,\ |\ id\ ,\ B$
- Parse tree:

x,y,10 x,y,z,10

Example 1b: derivations

- Grammar:
  - $A \rightarrow B\ int$
  - $B \rightarrow id\ ,\ |\ id\ ,\ B$

Rules for (id + ‘,’) lookahead:

id – shift

number – reduce

comma – reject

- Derivation:

Action	Stack	Input	Action	Stack	Input
S		x,y,10	S		x,y,z,10

Example 1c

- Another way to fix grammar:
  - $A \rightarrow B \text{ , } int$
  - $B \rightarrow id \mid B \text{ , } id$
- Conflict resolution:
  - Stack + lookahead give enough info to take correct parse action

Example 1c: parse trees

- Grammar:
  - $A \rightarrow B\ ,\ int$
  - $B \rightarrow id \mid B\ ,\ id$
- Parse tree:

x,y,10 x,y,z,10

Example 1c: derivations

- Grammar:
  - $A \rightarrow B \text{ , } int$
  - $B \rightarrow id \mid B \text{ , } id$
- Derivation:

Action	Stack	Input	Action	Stack	Input
S		x,y,10	S		x,y,z,10

Example 2

- Ambiguous grammar for conditional expressions:
  - *CondExpr* → *id* | *CondExpr* || *CondExpr*
    | *CondExpr* && *CondExpr* | ! *CondExpr*
- Consider this input:
  - x || y && z
- Stack/lookahead config in which shifting and reducing both work, but produce different parse trees:

Example 2: derivations

- Grammar:
  - $CondExpr \rightarrow id \mid CondExpr \parallel CondExpr$
    $\mid CondExpr \ \&\& \ CondExpr \mid \ ! \ CondExpr$
- Derivation:

Action	Stack	Input
S		x || y && z
R	x	|| y && z
S*2	CE	|| y && z
R	CE || y	&& z
S*2 or R?	CE || CE	&& z

Example 2: derivations

- Grammar:
  - *CondExpr* → *id* | *CondExpr* || *CondExpr*
    | *CondExpr* && *CondExpr* | ! *CondExpr*
- Derivation:

Action	Stack	Input	Action	Stack	Input
S*2	CE || CE	&& z	R	CE || CE	&& z

Example 2: ocamlyacc

- 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 grammr.
- Use “stratified grammar,” as for arithmetic expressions:
  - $CondExpr \rightarrow CondTerm \mid CondExpr \parallel CondTerm$
  - $CondTerm \rightarrow CondPrimary \mid CondTerm \;\&\&\; CondPrimary$
  - $CondPrimary \rightarrow id \mid \,!\, CondPrimary$\
- Parse tree: x || y && z

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, then reduce
  - If t_1 appears in a declaration after t_2, then shift
- Example:

```
%left      token, …
%right     token, …
%nonassoc  token, …
```

Example 2 (cont.)

- Use ambiguous grammar, but add these declarations

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

- x || y && z is now handled correctly. Derivation:

Action	Stack	Input
S		x || y && z

Example 2 (cont.)

- However, ocamlyacc still reports conflicts. 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 the two lines above

More on conflicts and LR parsing

- Prof. Kamin’s note on the “LR theorem”
- *Compilers: Principles, Techniques, and Tools* by Aho, Sethi, and Ullman
  - A.K.A "The Dragon Book"