Lecture 11: Parser Conflicts, Using Ambiguity, Error Recovery

LR(1) Parsing and CUP/Bison

- Bison and CUP build the kind of machine in the last lecture.
- However, for efficiency reasons, they collapse many of the states together, namely those that differ only in lookahead sets, but otherwise have identical sets of items. Result is called an *LALR(1) parser* (as opposed to LR(1)).
- Causes some additional conflicts, but these are rare.

LR(1) to LALR(1) Example

- The grammar

```
p : expr -|
expr : expr '+' term | term
term : term '*' primary | primary
primary : ID | '(' expr ')'
```

leads to (among others) two different states:

```
primary : '(' expr ')' • / -|, +, *
-------------------------------------------------
primary : '(' expr ')' • / +, *, ')'
```

- LALR(1) converts these to one state, combining lookaheads:

```
primary : '(' expr ')' • / -|, +, *, ')'
```

without any problems.

LR(1) to LALR(1) Problematic Example

- Example:

```
p : '1' q '0'  | '0' q '1'  | '1' r '1' |  '0' r '0' ;

q : 'x' ;

r : 'x' ;
```

- Here, we get two states for reducing 'x':

```
q : 'x' • /  '1'              q : 'x' • /  '0'
r : 'x' • /  '0'              r : 'x' • /  '1'
```

and all is well.

- Almost always, can get away with collapsing these into one state, combining lookaheads.
- But here, Bison would get a conflict:

```
q : 'x' • / '1', '0'
r : 'x' • / '0', '1'
```

Shift/Reduce Conflicts

- If a DFA state contains both [X: $\alpha\bullet a\beta$, b] and [Y: $\gamma\bullet$, a], then we have two choices when the parser gets into that state at the | and the next input symbol is a:
  - Shift into the state containing [X: $\alpha a\bullet\beta$, b], or
  - Reduce with Y: $\gamma\bullet$.
- This is called a *shift-reduce conflict*.
- Often due to ambiguities in the grammar. Classic example: the dangling else

  S ::= "if" E "then" S | "if" E "then" S "else" S | ...
- This grammar gives rise to a DFA state containing

  [S: "if" E "then" S•, "else"] and [S: "if" E "then" S•"else" S, ...]
- So if "else" is next, we can shift or reduce.

More Shift/Reduce Conflicts

- Consider the ambiguous grammar

  E : E + E | E * E | int
- We will have states containing

$$\begin{array}{lcl} [E: E + \bullet E, */+] & & [E: E + E \bullet, */+] \\ [E: \bullet E + E, */+] & \xrightarrow{E} & [E: E \bullet + E, */+] \\ [E: \bullet E * E, */+] & & [E: E \bullet * E, */+] \\ \dots & & \dots \end{array}$$

- Again we have a shift/reduce conflict on input '*' or '+' (in the item set on the right).
- We probably want to shift on '*' (which is usually supposed to bind more tightly than '+')
- We probably want to reduce on '+' (left-associativity).
- Solution: provide extra information (the precedence of '*' and '+') that allows the parser generator to decide what to do.

Using Precedence in Bison/CUP

- In Bison or Horn, you can declare precedence and associativity of both terminal symbols and rules,
- For terminal symbols (tokens), there are precedence declarations, listed from lowest to highest precedence:

Bison	CUP
`%left '+' '-'`	`precedence left PLUS, SUB;`
`%left '*' '%'`	`precedence left MULT, MOD;`
`%right "**"`	`precedence right EXPO;`

  Symbols on each such line have the same precedence.
- For a rule, precedence = that of its last terminal (Can override with `%prec` if needed, cf. the Bison manual).
- Now, we resolve shift/reduce conflict with a shift if:
  - The next input token has higher precedence than the rule, or
  - The next input token has the same precedence as the rule and the relevant precedence declaration was `%right`.

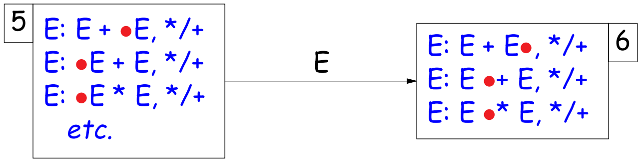
  and otherwise, we choose to reduce the rule.

Example of Using Precedence to Solve S/R Conflict (1)

- Assuming we've declared

```
%left PLUS
%left MULT
```

  the rule E ::= E + E will have precedence 1 (left-associative) and the rule E ::= E * E will have precedence 2.
- So, when the parser confronts the choice in state 6 w/next token '*',



  it will choose to shift because the '*' has higher precedence than the rule E + E.
- On the other hand, with input symbol '+', it will choose to reduce, because the input token then has the same precedence as the rule to be reduced, and is left-associative.

Example of Using Precedence to Solve S/R Conflict (2)

- Back to our dangling else example. We'll have the state

10
S: "if" E "then" S •, "else"
S: "if" E "then" S•"else" S, "else"
etc.

- Can eliminate conflict by declaring the token "else" to have higher precedence than "then" (and thus, than the first rule above).

- HOWEVER: best to limit use of precedence to these standard examples (expressions, dangling elses). If you simply throw them in because you have a conflict you don't understand, you're like to end up with unexpected parse trees or syntax errors.

Reduce/Reduce Conflicts

- The lookahead symbols in LR(1) items are only considered for reductions in items that end in '•'.
- If a DFA state contains both

  [X: α•, a] and [Y: β•, a]

  then on input 'a' we don't know which production to reduce.
- Such *reduce/reduce conflicts* are often due to a gross ambiguity in the grammar.
- Example: defining a sequence of identifiers with

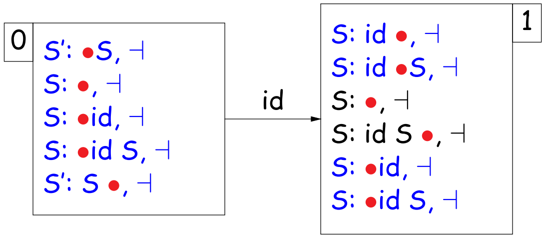
  S: ϵ | id | id S

- There are two parse trees for the string id:

  S $\Rightarrow$id or S $\Rightarrow$id S $\Rightarrow$id.

Reduce/Reduce Conflicts in DFA

- For this example, you'll get states:


0
S': •S, ⊣
S: •, ⊣
S: •id, ⊣
S: •id S, ⊣
S': S •, ⊣

id

1
S: id •, ⊣
S: id •S, ⊣
S: •, ⊣
S: id S •, ⊣
S: •id, ⊣
S: •id S, ⊣

- Reduce/reduce conflict on input '⊣'.
- Better rewrite the grammar: S: ϵ | id S.

Parsing Errors

- One purpose of the parser is to filter out errors that show up in parsing
- Later stages should not have to deal with possibility of malformed constructs
- Parser must *identify* error so programmer knows what to correct
- Parser should *recover* so that processing can continue (and other errors found).
- Parser might even *correct* error (e.g., PL/C compiler could "correct" some Fortran programs into equivalent PL/1 programs!)

Identifying Errors

- All of the valid parsers we've seen identify syntax errors as soon as possible.
- *Valid prefix property:* all the input that is shifted or scanned is the beginning of some valid program. . .
- . . . But the rest of the input might not be.
- So in principle, deleting the lookahead (and subsequent symbols) and inserting others will give a valid program.

Automating Recovery

- Unfortunately, best results require using semantic knowledge and hand tuning.
  - E.g., a(i].y = 5 might be turned to a[i].y = 5 if a is statically known to be a list, or a(i).y = 5 if a function.
- Some automatic methods can do an OK job that at least allows parser to catch more than one error.

Bison's and CUP's Technique

- The special terminal symbol error is never actually returned by the lexer.
- Gets inserted by parser in place of erroneous tokens.
- Parsing then proceeds normally.

Example of Bison's Error Rules

Suppose we want to throw away bad statements and carry on

```
stmt : whileStmt
     | ifStmt
     | ...
     | error NEWLINE
     ;
```

Response to Error

- Consider erroneous text like

```
if x y: ...
```

- When parser gets to the y, will detect error.
- Then pops items off parsing stack until it finds a state that allows a shift or reduction on ‘error’ terminal
- Does reductions, then shifts ‘error’.
- Finally, throws away input until it finds a symbol it can shift after ‘error’, according to the grammar.

Error Response, contd.

- So with our example:

```
stmt : whileStmt
     | ifStmt
     | ...
     | error NEWLINE
     ;
```

  We see ‘y’, throw away the ‘if x’, so as to be back to where a stmt can start.

- Shift ‘error’ and throw away more symbols to NEWLINE. Then carry on.

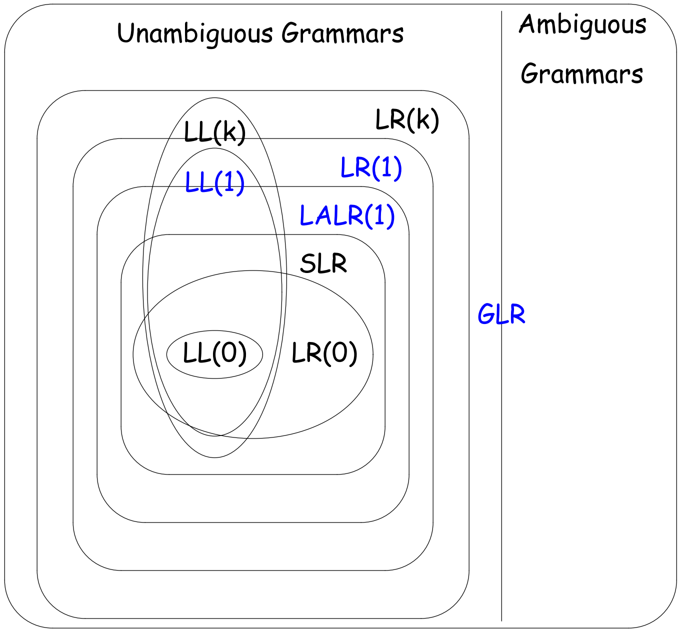
Of Course, It’s Not Perfect

- “Throw away and punt” is sometimes called “panic-mode error recovery”
- Results are often annoying.
- For example, in our example, there could be an INDENT after the NEWLINE, which doesn’t fit the grammar and causes another error.
- Bison compensates in this case by not reporting errors that are too close together
- But in general, can get cascade of errors.
- Doing it right takes a lot of work.

Bison Examples

[See lecture15 directory.]

A Hierarchy of Grammar Classes



Unambiguous Grammars

Ambiguous Grammars

LR(k)

LL(k)

LR(1)

LL(1)

LALR(1)

SLR

GLR

LL(0)

LR(0)

From Andrew Appel, "Modern Compiler Implementation in Java"

Summary

- Parsing provides a means of tying translation actions to syntax clearly.
- A simple parser: LL(1), recursive descent
- A more powerful parser: LR(1)
- An efficiency hack: LALR(1), as in Bison.
- Earley's algorithm provides a complete algorithm for parsing all context-free languages.
- We can get the same effect in Bison by other means (the `%glr-parser` option, for Generalized LR), as seen in one of the examples from lecture #5.