# Lecture 8: 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

  [E: E + •E, */+]  [E: E + E •, */+]
  [E: •E + E, */+] $\xrightarrow{E}$ [E: E •+ E, */+]
  [E: •E * E, */+]  [E: E •* E, */+]
  ...  ...

- 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 CUP, 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 equals that of its last terminal (Can override with `%prec` if needed, cf. the Bison or CUP 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.

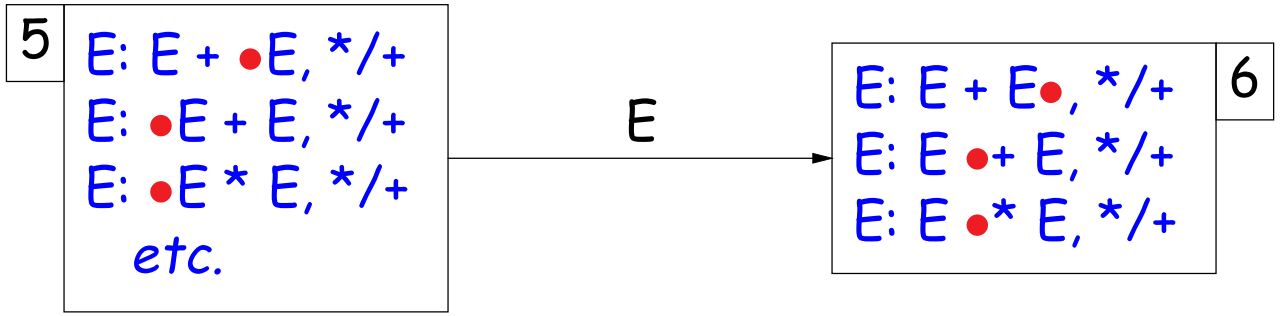
# Example of Using Precedence to Solve S/R Conflict (1)

- Assuming we've declared

```
precedence left PLUS
precedence 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 when the next token is '*',




  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"<br>S: "if" E "then" S•"else" S, "else"<br>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 likely 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: $\alpha$•, a] and [Y: $\beta$•, 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

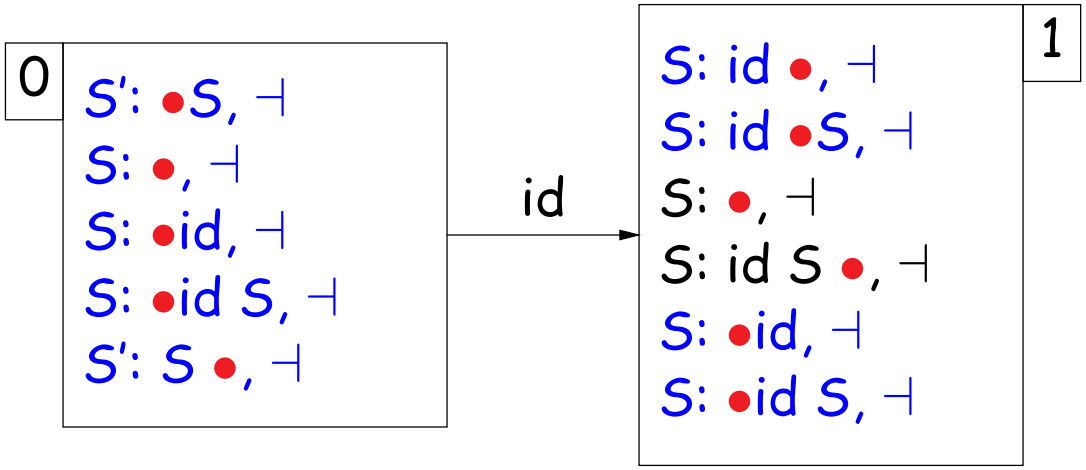
  S: $\epsilon$ | id | id S

- 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:




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

# Bison Examples

[See lecture8 directory.]

# GLR

- LR($k$) parsers provide fast parsing and their limitations can actually help find errors in a grammar (e.g., unintentional ambiguities).
- The cost is that they don't cover all context-free languages, and have difficulty with some language constructs, such as situations that require multiple symbols of lookahead.
- A technique called GLR parsing can handle general context-free languages while preserving compilation speed (and space) on inputs that can be handled by plain LR parsing.
- Idea is simple: when an ordinary LR parser comes to an S/R or R/R conflict, it tries both, in effect making a copy of the parsing stack for each additional choice of action.
- It then proceeds by working on each stack copy until it has shifted the next token onto each stack, and continuing in the same way, token-by-token.
- At some point, a stack copy either runs into a parsing error and is eliminated, or becomes equal to another copy, in which case the two stacks are merged back into one.

# GLR Example

- Example from the Bison manual: in Extended Pascal, one can define subtypes and enumerated types with

```
type_defn : "type" id "=" expr ".." expr
          | "type" id "=" "(" id_list ")"
          ...
id_list : id | id_list "," id ;
expr : ... | id | ...;
```

- So consider what happens with the declaration

```
type aSubrange = (x) .. y;
```

- Until the parser sees "..", it doesn't know whether this is a subrange definition or an enumerated type definition.
- Therefore, after it shifts the id 'x', it doesn't know whether to reduce it to an id_list or to an expr until *after* shifting the ')', which is too late.
- GLR simply does both reductions. The stack where the reduction to id_list happens eventually dies, leaving just the stack where the correct choice (reducing to expr) was made.

# 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 an array, or

    ```
    a(i).y = 5
    ```

    if a is known to be 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

- When a syntax error is detected,
  - Throw away states on the stack until we come to one that allows us to shift the token '**error**'.
  - The special terminal symbol '**error**' is never actually returned by the lexer.
  - Instead, the parser inserts it, and then proceeds normally.
  - If after shifting **error**, the next input symbol still causes an error, we discard tokens until the parser can proceed.

# Example of Bison's Error Rules

Suppose we want to throw away bad statements and carry on

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

- 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
     ;
```

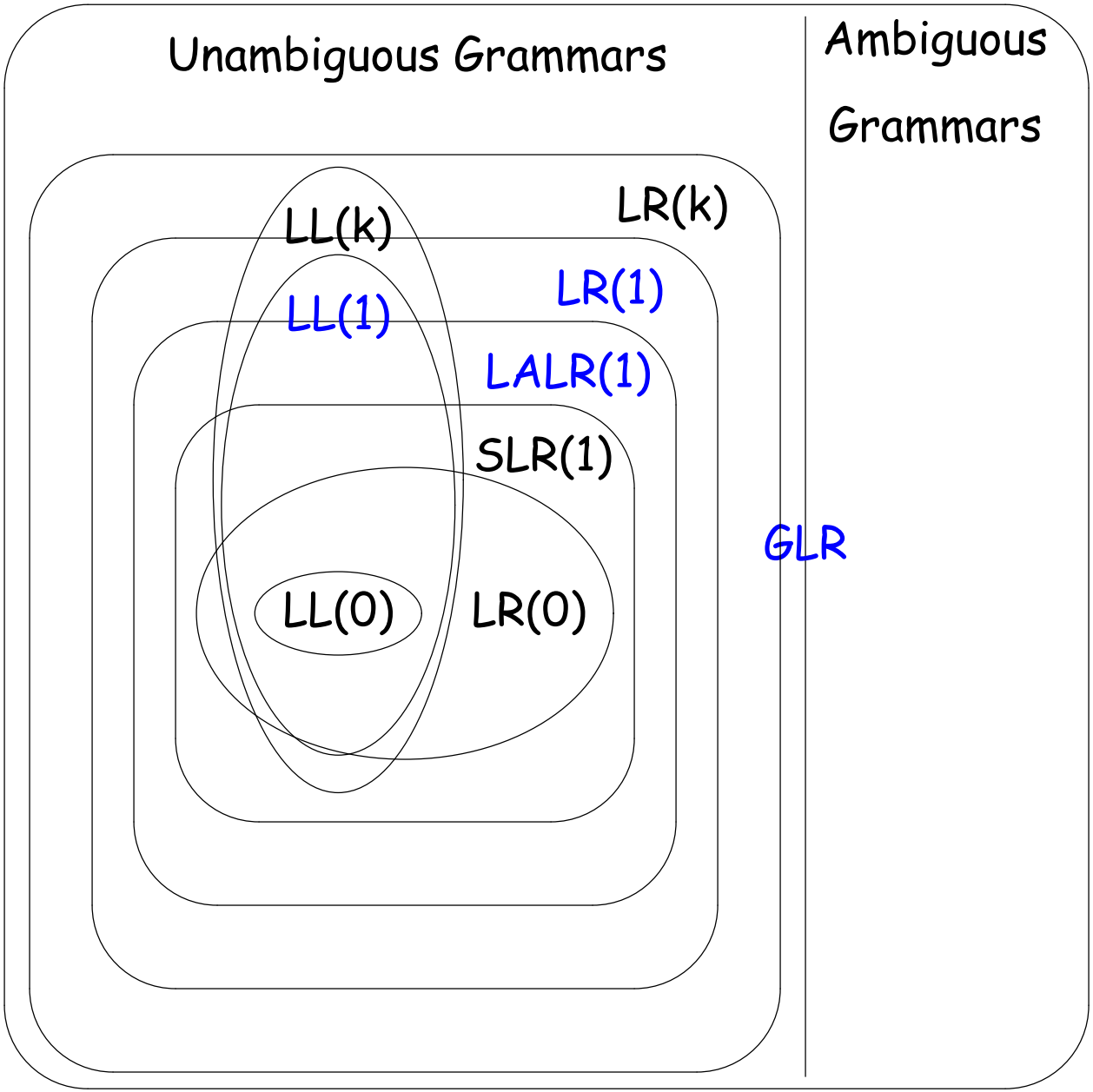
  We see 'y', throw away the 'if x', so as to be back to where a stmt can start.

- 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.

# A Hierarchy of Grammar Classes




Andrew Appel, *Modern Compiler Implementation in Java*

(SLR(1) (Simple LR) is LR(0) with FOLLOW(e) as lookaheads for reductions to e.)

# 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 #4.