## Parsing - 4

- Using ambiguous grammars for parsing
- LALR(k) parsing
  - Space savings over LR(k)
  - Sometimes introduce reduce-reduce conflicts
- Parser generators : Yacc, CUP
  - How to use?
  - Error recovery

## **Using Ambiguous Grammars**

- Sometimes an ambigous grammar will create a smaller parser than an unambiguous one
- Need to resolve conflicts appropriately by setting precedences as desired, to preserve meaning in the grammar
  - Often done with expression grammars
    - e.g., to get small SLR(1) parser for language on *Parsing3*, #8

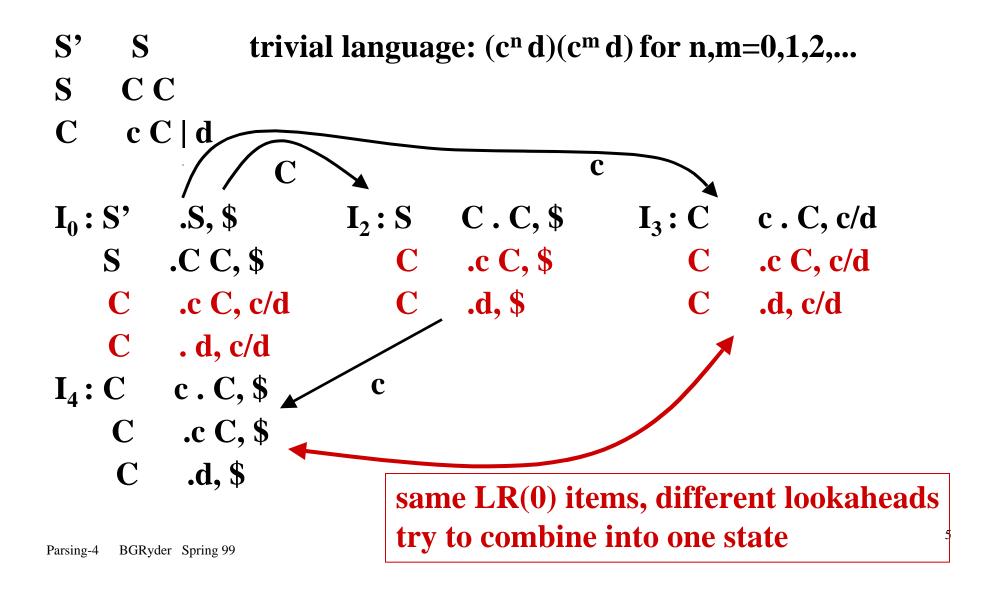
## LALR(k) Parsing

- LALR(k) parsers use *k lookahead* symbols and combine those states of an LR(k) parser that have the same items, except for lookahead symbols
- Provides smaller parsers, usually about the size of an SLR(k) parser
- But sometimes can introduce *reduce-reduce* conflicts in this manner

## LALR(k) Parsing

- When given erroneous input, sometimes an LALR(k) parser will do a few extra reductions which an LR(k) parser would have avoided, but it never will shift another symbol onto the stack, beyond those which would be shifted by an LR(k) parser.
- Can be formed directly from a grammar, although we will reduce an LR(1) parser to LALR(1) form

#### Example, ASU p 236



## LALR(k)

- Complete LALR(1) parser for this language and can see there are no conflicts introduced
- When merge LR(k) states cannot produce shift-reduce conflicts, but can produce reducereduce conflicts

<b>e.g.,</b> A	c. , d	Α	<b>c.</b> , e	two states which when combined
В	c., e	B	<b>c., d</b>	produce a reduce-reduce conflict

#### **CUP: a Parser Generator**

- Yacc 1975 Steve Johnson at AT&T Bell Labs
- CUP, a Java version of Yacc
  - Input: CUP directives, Java code, grammar
  - Output: Java program which parses the language described by grammar (i.e., a Grm object)
  - Grm class extends java\_cup.runtime.lr\_parser class (see proj3/Parse/Parse.java); parse() method is applied to the Grm object within a try block so exceptions will be caught properly

# Parse/Parse.java in proj3

<pre>public class Parse {    public ErrorMsg.ErrorMsg errorMsg.err</pre>	Isg;		
errorMsg = new ErrorMsg.Erro	rMsg(filename);		
java.io.InputStream inp; try {inp=new java.io.FileInputSt catch (java.io.FileNotFoundExce	me exists	it	
<pre>     throw new Error("File not four     Grm parser = new Grm(new Yyl     try { parser./*debug_*/parse();}</pre>	create new		
catch (Throwable e) {	try to parse input		parser
<pre>e.printStackTrace(); throw new Error(e.toString());} finally { try {inp.close();} catch ( }</pre>		on e) {} } <b>cl</b>	eanup

## **Grm.cup**

- Input file to the CUP parser generator
  - Preamble of CUP directives and grammar rules
    - Grammar rules look like:

exp ::= exp PLUS exp {: actions :}

• Directive include identification of terminals and nonterminals

terminal ID, WHILE, BEGIN, END

non terminal prog, stm, stmlist;

start with prog;

 Actions are given in Java and will be executed as the parser reduces using this rule.

### Conflicts

- CUP reports conflicts
  - Default is to shift for shift-reduce conflicts
  - Default is use rule appearing the earliest in the grammar for reduce-reduce conflicts
  - Normally, we rewrite the grammar when conflicts are reported

#### **Precedence Directives**

- Precedence directives
  - Specify both associativity of operators and relative precedence among them
     precedence nonassoc EQ, NEQ; lowest prec
     precedence left PLUS, MINUS;
     precedence right EXP; highest prec
  - Use precedence to break shift-reduce conflicts, given last token on righthand-side of rule
    - If rule and token have same precedence then *left* prec means *reduce*, *right* prec means *shift*, and nonassoc means error

#### Limitations

- Not all language constructs can be expressed in a context-free grammar
  - e.g., Correspondence of types of operands to operator
  - e.g., Finding correct kind of l-value on lefthandside of assignment statement
- Use semantic analysis phase to check these

### **Local Error Recovery**

- Local adjust the parse stack *where* the error was detected
  - Can insert error symbol into grammar in order to go into an error state on improper input
  - Then input is discarded until a synchronizing token is encountered
  - Have to be careful when discarding states from the stack, when associated actions have side effects
    - e.g., construct counting matched parentheses

### **Global Error Recovery**

- Global insert or delete token(s) from input stream at a point *before* where the error was detected
  - Try to find the smallest set of insertions or deletions that turn the source into a parsable string
  - Best replacement allows parsing to continue furthest past current position

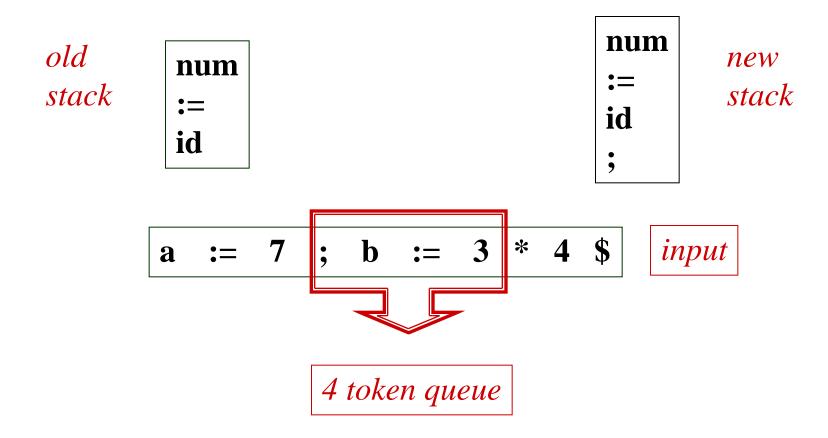
### **Burke-Fisher Error Recovery**

- Burke-Fisher Error Recovery(1987) exhaustively tries single token insertion, deletion or replacement at every point within k tokens before where the error occurs
- If have N kinds of tokens, there are k+kN+kN possible deletions, insertions and substitutions within the k token window (kept on a queue)
- Must delay all semantic actions to prevent unwanted side effects, until parse is validated

### **Burke-Fisher Error Recovery**

- Algorithm uses 2 stacks, *current* and *old*, and a *queue* of *k* tokens
  - *old* stack has successfully parsed string so far (have done actions for reductions to symbols here)
  - *current* stack has rest of possible parse covering the next k tokens
  - *queue* is k tokens back from endpoint of current parse
- Can use *old* stack and *queue* to reparse string after replacement, deletion or insertion of single token into *queue*

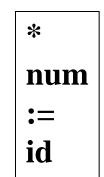
### Example



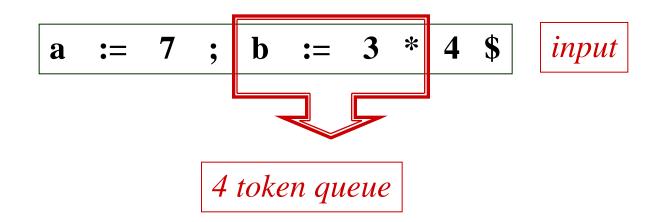
#### Example

old stack

; S



new stack



### **Burke-Fisher Error Recovery**

- Problems:
  - If the semantic action(s) being delayed affect parsing (e.g., typedef)
  - Need to specify values for inserted/replaced tokens
- Common errors can be anticipated with error correcting code

- e.g., in 0 end to close a scope