

POLITECNICO DI TORINO

(01JEUHT) Formal Languages and Compilers
Laboratory N°3

Stefano Scanzio

Mail: stefano.scanzio@polito.it

Web: <http://www.skenz.it/compilers>



Lab 3

1

Cup Advanced Use

- Grammars with ambiguities
- Lists
- Operator precedence
- Handling syntax errors



Lab 3

2

Ambiguous grammars in CUP

- Conflicts can arise when the grammar is ambiguous
- This implies that the parser must choose between two or more alternative actions.
- The problem can be solved by modifying the grammar (in order to make it non-ambiguous) or by instructing the parser on how to handle ambiguity.
- The latter option requires that the parsing algorithm is fully understood, in order to avoid unwanted / wrong behaviors.



Ambiguous Grammar

- A grammar is ambiguous if there is at least one sequence of symbols for which two or more distinct parse trees exist.
- Exercise: find all parse trees for
`if (i==1) if (j==2) a=0; else a=1;`

given the grammar:

- $S ::= M ;$
- $M ::= \text{'if' } C M ;$
- $M ::= \text{'if' } C M \text{'else' } M ;$
- $M ::= \text{ID '=' NUM ';' } | \text{ID '=' ID ';' } ;$
- $C ::= \text{'(VAR '=' NUM ')'$



Non-ambiguous grammar: if-then-else statement

- It is possible to write a non-ambiguous grammar for the if-else statements, as follows:
 - $S ::= M \mid U ;$
 - $U ::= \text{'if' } C S ;$
 - $U ::= \text{'if' } C M \text{'else' } U ;$
 - $M ::= \text{'if' } C M \text{'else' } M ;$
 - $M ::= \text{ID '=' NUM ';' } \mid \text{ID '=' ID ';' } ;$
 - $C ::= \text{'(ID '=' NUM ')} ;$
- `if (i==1) if (j==2) a=0; else a=1;`



Non-ambiguous grammar : Algebraic expressions

- The non-ambiguous grammar that describes algebraic expressions is:
 - $S ::= E$
 - $E ::= E \text{'+' } T$
 - $E ::= E \text{'-' } T$
 - $E ::= T$
 - $T ::= T \text{'*' } F$
 - $T ::= T \text{'/' } F$
 - $T ::= F$
 - $F ::= \text{'(' } E \text{')'}$
 - $F ::= \text{NUM}$

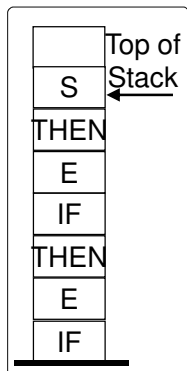
- The symbols T and F are used to solve the ambiguity given by the priority of operators '*' and '/' over the operators '+' e '-' .



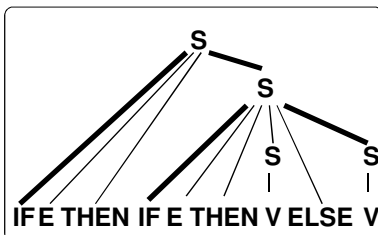
Ambiguous grammars in Cup: shift-reduce conflict (I)

- 1) $S ::= IF\ E\ THEN\ S$
- 2) $S ::= IF\ E\ THEN\ S\ ELSE\ S$
- 3) $S ::= V$

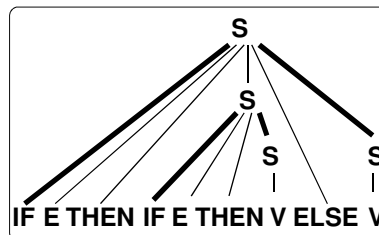
- Input: `IF E THEN IF E THEN S (*) ELSE S`
- The next token is 'ELSE'
- 2 possible actions:



■ **SHIFT** 'ELSE' token into the Stack
=> Rule 2



■ **REDUCE** the first 4 top elements of the Stack
=> Rule 1

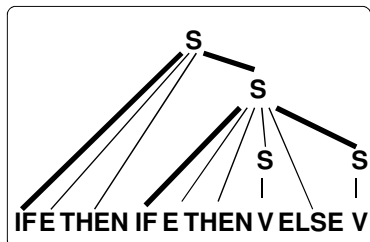


Ambiguous grammars in Cup: shift-reduce conflict (II)

- 1) $S ::= IF\ E\ THEN\ S$
- 2) $S ::= IF\ E\ THEN\ S\ ELSE\ S$
- 3) $S ::= V$

Input

IF E THEN IF E THEN
V ELSE V



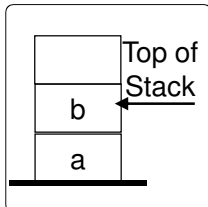
*** Shift/Reduce conflict found in state #8
between $S ::= IF\ E\ THEN\ S\ (*)$
and $S ::= IF\ E\ THEN\ S\ (*)\ ELSE\ S$
under symbol ELSE

Resolved in favor of shifting.

Cup performs a **shift** action.

Ambiguous grammars in Cup: reduce-reduce conflict (I)

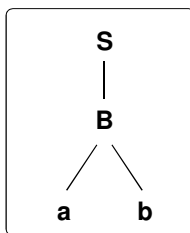
Input
a b



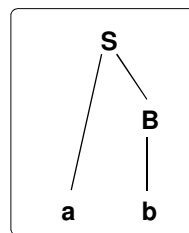
- 1) $S ::= a B$
- 2) $S ::= B$
- 3) $B ::= a b$
- 4) $B ::= b$

- The next token is EOF
- 2 possible actions:

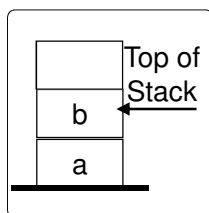
■ **REDUCE** the first 2 top elements of the Stack
=> Rule 3



■ **REDUCE** the first top element of the Stack
=> Rule 4



Ambiguous grammars in Cup: reduce-reduce conflict (II)



- 1) $S ::= a B$
- 2) $S ::= B$
- 3) $B ::= a b$
- 4) $B ::= b$

*** Reduce/Reduce conflict found in state #7
between $B ::= b (*)$
and $B ::= a b (*)$
under symbols: {EOF}

Resolved in favor of the second production.

Cup performs a reduction using the first defined rule (3) .

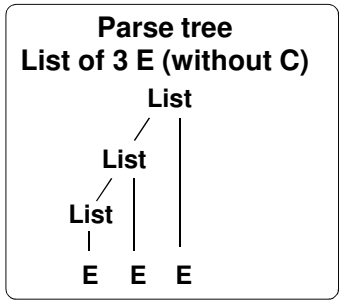


Lists (I)

● Examples of lists:

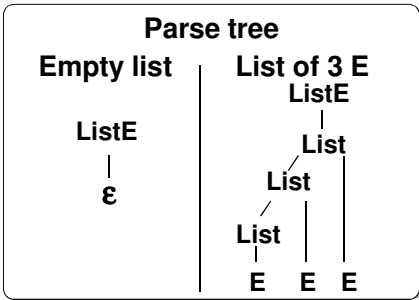
- List with at least one element E, separated with commas C:

```
List ::= List E | E ; //without C
List ::= List C E | E ;
```



- List of elements, possibly empty (first example):

```
ListE ::= ε | List ;
List ::= List E | E ;
```



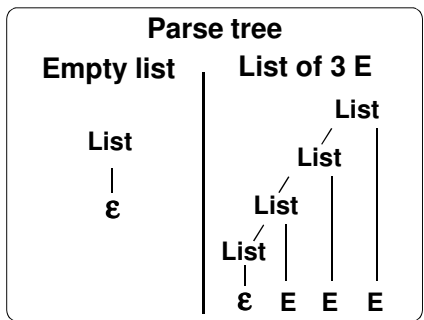
Lists (II)

Same sequence of input tokens, 2 different parse trees => **AMBIGUOUS GRAMMAR**

● Examples of lists:

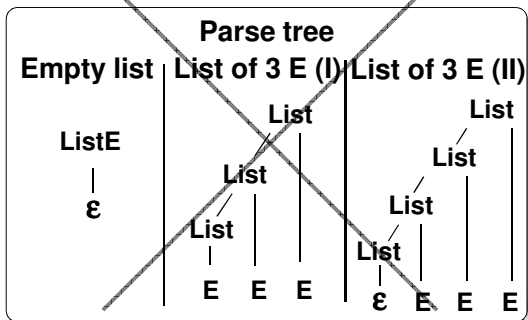
- List of elements, possibly empty (second example):

```
List ::= List E | ε ;
```



- List of elements, possibly empty (WRONG example):

```
List ::= List E | E | ε ;
```

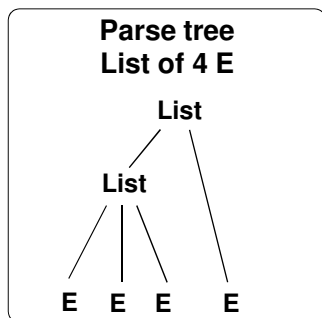


Lists (III)

● Examples of lists:

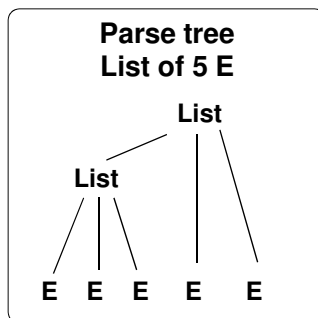
- List of at least 3 elements:

List ::= List E | E E E ;



- List of at least 3 elements in an odd number:

List ::= List E E | E E E ;



Precedence Section: Ambiguous grammars

- Ambiguous grammars can result in fewer, simpler rules, and hence can be sometimes preferred.
- It is necessary to provide disambiguating rules in those cases.
- A typical example is given by algebraic expressions:

Non-ambiguous grammar

```

S ::= E
E ::= E '+' T
E ::= E '-' T
E ::= T
T ::= T '*' F
T ::= T '/' F
T ::= F
F ::= '(' E ')'
F ::= INTEGER
    
```

Ambiguous grammar

```

E ::= E '+' E
E ::= E '-' E
E ::= E '*' E
E ::= E '/' E
E ::= '(' E ')'
E ::= INTEGER
    
```



Associativity

- Left-associative operator ($E ::= E '+' E$)
 - $1+2+3+4 \rightarrow 3+3+4 \rightarrow 6+4 \rightarrow 10$
- Right-associative operator ($E ::= E '+' E$)
 - $1+2+3+4 \rightarrow 1+2+7 \rightarrow 1+9 \rightarrow 10$
- The assignment operator '=' is right-associative:
 - $a = b = 3$
 - The power operator is also right-associative
 - $3^{2^2} \rightarrow 3^4 \rightarrow 81$



Precedence Section: Operators

- Rule #1 (as well as Rule #2) is ambiguous
 - Associativity of the '+' (*) operator is not specified
- Moreover, the precedence of the '+' and '*' is not specified by Rules #1 and #2
- It is possible to make these rules non-ambiguous by adding information in the precedence section.
- The keyword precedence left defines a left-associative operator, precedence right a right-associative operator, whereas precedence nonassoc defines a non-associative operators.
- The order in which precedence keywords are declared is inversely proportional to their priority.

- 1) $E ::= E '+' E$
- 2) $E ::= E '*' E$
- 3) $E ::= '(' E ')'$
- 4) $E ::= \text{INT}$



Precedence Section: Disambiguating rules

- To each production that contains at least one terminal defined as operator, Cup associates the precedence and associativity of the rightmost operator.
- If the rule is followed by the keyword `%prec`, the precedence and associativity are those of the specified operator.
- In the case of a shift-reduce conflict, the action corresponding to the highest precedence production is executed.
- If the precedence is the same, associativity is used: left-associativity results in a reduce action, right-associativity in a shift action.



Precedence Section: Example

```

terminal uminus;

precedence left '+' , '-' ; /* Low priority */
precedence left '*' , '/' ;
precedence left uminus;    /* High priority */

start with E;

E ::=  E '+' E
      | E '-' E
      | E '*' E
      | E '/' E
      | '-' E    %prec uminus
      | '(' E ')'
      | INTEGER
;

```



User code

- Directives are available to insert user code directly in the parser.
- They are useful for
 - Personalizing the parser behavior
 - Adding code directly in the class that implements the parser
 - Using a scanner generator different from the default one (JFlex)
- They are:
 - init with { : ... ; }
 - ⤴ This code is executed before calling any scanner method, hence before any terminal symbol is passed to the parser
 - ⤴ It is used to initialize variables or to initialize the scanner in the case JFlex is not used.



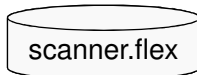
User code (II)

- scan with { : ... ; }
 - ⤴ Indicates to the parser which procedure to use to request the next terminal to the scanner
 - ⤴ It must return an object of the class `java_cup.runtime.Symbol`
 - ⤴ It is used for non-default scanner generators (different than JFlex)
 - ⤴ scan with { : return scanner.next_token(); ; }
- When CUP generates the java file that implements the parser, two classes are defined:
 - ⤴ `public class parser extends java_cup.runtime.lr_parser`
 - ⤴ `parser` is the java class that implements the parser and inherits different methods from the `java_cup.runtime.lr_parser` class
 - ⤴ `class CUP$parser$actions`
 - ⤴ `CUP$parser$actions` is the class where declared grammar rules are translated into a java program. Here, also semantic actions (i.e., the java code related to each rule) are reported



User code (III)

- The `java_cup.runtime.lr_parser` class is implemented in the file `java_cup/runtime/lr_parser.java`, in the CUP installation directory
- parser code `{: ... :}`
 - ⤴ The code is included in the parser class
 - ⤴ It is used to include scanning methods within the parser but usually to override parser methods (e.g. to override methods for error handling)
- action code `{: ... :}`
 - ⤴ The code included in this directive is copied as is in the `CUP$parser$actions` class
 - ⤴ The code is reachable only in the semantic actions associated with grammar rules
 - ⤴ It is used to define procedures and variables to be used in the actions associated to the grammar (e.g., symbol table)



Errors: Printing line and column

```
import java_cup.runtime.*;
...
%%
%cup
%line
%column
```

```
%{
  private Symbol symbol(int type){
    return new Symbol(type, yylines, yycolumn);
  }
  private Symbol symbol(int type, Object value){ //Semantic analysis
    return new Symbol(type, yylines, yycolumn,value);
  }
}%
...
%%
[a-z]      { return symbol(sym.EL); }
','        { return symbol(sym.CM); }
```

Symbol constructors:

```
public Symbol( int sym_id)
public Symbol( int sym_id, int left, int right)
public Symbol( int sym_id, Object o)
public Symbol( int sym_id, int left, int right, Object o)
```





Errors: Printing line and column

```
import java_cup.runtime.*;

parser code {:
  public void report_error(String message, Object info) {
    StringBuffer m = new StringBuffer(message);
    if (info instanceof Symbol) {
      if (((Symbol)info).left != -1 && ((Symbol)info).right != -1) {
        int line = (((Symbol)info).left)+1;
        int column = (((Symbol)info).right)+1;
        m.append(" (line "+line+", column "+column+"");
      }
    }
    System.err.println(m);
  }
:}
```



Handling syntax error (I)

- Generally speaking, when a parser finds an error it should not immediately terminate the execution
 - A compiler usually tries to recover from the error in order to analyze the rest of the input and signal the highest possible number of errors
- As default, a CUP-generated parser when an error is detected:
 - Signals by means of the method `public void syntax_error(Symbol cur_token)` defined in the `java_cup.runtime.lr_parser` class a syntax error, writing "Syntax error" in `stderr`.
 - If the error is not managed by the parser through the predefined error symbol, the parser call the `public void unrecovered_syntax_error(Symbol cur_token)` method, also defined in `java_cup.runtime.lr_parser`. This function, after writing "Couldn't repair and continue parse" in `stderr` (to notify the user of an unrecoverable syntax error), stops the execution of the parser.



Handling syntax error (II)

Analyzing the two functions in detail:

- `public void syntax_error(Symbol cur_token)`
 - Calls the function `report_error` with the following parameters
`report_error("Syntax error", cur_token);`
 - ⤴ Where, when an error occurs, `cur_token` is the currently lookahead symbol
- `public void unrecovered_syntax_error(Symbol cur_token)`
 - Calls the function `report_fatal_error`, with the following parameters
`report_fatal_error("Couldn't repair and continue parse", cur_token);`
 - The `report_fatal_error` function calls with the same parameters
`report_error` and it launches an exception that causes the end of
the parser
- A suitable redefinition, in parser code `{: ... ;}`, of the listed
functions, allow to customize errors management



'error' predefined symbol

- The '`error`' predefined symbol signals an error condition. It can
be used within the grammar in order to enable the parser to
continue execution when an error is encountered.
- Example:

```

ass ::= ID EQ E S
      | ID EQ error S
;

```



How does Cup handle the 'error' symbol?

- When an error occurs, the parser will start emptying the stack until a state is found in which the 'error' symbol is allowed
 - In the previous example, uncorrect E (i.e. symbol sequences that cannot be reduced as E) are removed from the stack, until the terminal EQ is found on the top of the stack.
- The **error** token is *shifted* in the stack
- If the next token is acceptable, the parser resumes syntax analysis.
- Otherwise the parser will continue to read and discard tokens, until an acceptable one is found
 - In the previous example, the parser will read and discard all tokens until S is found.



Some general rules

- A simple strategy for error handling is skipping the current *statement*:


```
stmt ::= error `;`
```
- Sometimes it can be useful to find a closing symbol corresponding to an opening symbol:


```
expr ::= `(` expr `)`
      | `(` error `)`
```
- Note: to limit the generation of spurious error messages, after an error occurs, error signaling is suspended until at least three consecutive tokens are *shifted*.



Grammar

```

file ::= funcs
;

funcs ::= /* empty */
        | funcs func
;

func  ::= ID '(' ')'
        compound
;

compound ::= '{' stmts '}'
;

stmts ::= /* empty */
        | stmts stmt
;

stmt  ::= exp ';'
        | compound
;

exp   ::= NUM
        | exp '+' exp
        | exp '-' exp
        | exp '*' exp
        | exp '/' exp
        | '-' exp %prec NEG
        | '(' exp ')'
;

```



Statements and expressions

```

stmt ::= exp ';'
        | compound
        | error ';' { : System.err.println("syntax error in statement"); :}
;

compound ::= '{' stmts '}'
            | '{' stmts error '}' { : System.err.println("missing ; before '{' "); :}
;

exp ::= ...
        | '(' error ')' { : System.err.println(" syntax error in expression"); :}
;

```

