JYOTHISHMATHI INSTITUTE OF TECHNOLOGY & SCIENCE, KARIMNAGAR

COMPILER DESIGN

K.RAJ KUMAR

ASST. PROFESSOR

CSE

Ambiguity in Grammar

- Every ambiguous grammar fails to be LR.
- Certain types of ambiguous grammar are useful in the specification and implementation of the languages.
  - For language constructs like expressions an ambiguous grammar provides a shorter more natural specification than any equivalent unambiguous grammar.
  - Used in isolating common occurring syntactic constructs for special case optimization.

- We can specify ***disambiguating rules*** that allow only one parse tree for each sentence. So that overall language construct still remains unambiguous.

- Using Precedence and Associativity to Resolve Parsing Conflicts :
- Consider arithmetic expressions in programming languages with operator + and * .
- Grammar rules:
- **E → E + E | E * E | (E) | id**

---- (1)

- The corresponding unambiguous grammar is:
- E → E + T | T
- T → T * F | F --(2)

- F → (E) | **id**

- This unambiguous grammar generates the same language but gives lower precedence to + over * and makes both operators left associative.

- Benefits of using ambiguous grammar:
  - Using ambiguous grammar makes us able to change the associativities or/and precedence levels of + and * without disturbing the productions of (1) or the number of states in the resulting parser.
  - Parser for (2) will spend substantial fractions of its time reducing by the production E → T, T→ F .Whose sole function is to enforce associativity and precedence.

- Sets of LR (0) for augmented grammar:

I0 :

E’ → .E

E → .E - E

E → .E * E

E → .(E)

E → .**id**

I1 :

E’ → E .

E → E. + E

E → E . * E

I2 :

E → (.E)

E → E. + E

E → .E * E

E → .(E)

E → .id

I3 : E → id.

I4 :

E → E + .E

E → .E + E

E → .E * E

E → .(E)

E → .id

I5 :

E → E * .E

E → .E - E

E → .E * E

E → .(E)

E → .**id**

I6 : E → (E.)

E → E. + E

E → E. * E

I7 :

E → E + E.

E → E. + E

E → .E * E

I8 : E → E * E.

E → E. + E

E → E.* E

I9 : E → (E).

- Since the grammar is ambiguous parsing actions conflict will occur. The states corresponding to items I7 and I8 will generate these conflicts. These conflicts can be resolved using the precedence and associativity information for + and *.
- Consider the input **id + id * id** which causes a parser based upon above states to enter state 7 after processing **id + id**; in particular parser reaches configuration

- ***Stack***
  Input
- 0 E 1 + 4 E 7
  *** id $**

Assuming that * takes precedence over +, we know that the parser should shift * on to the stack, preparing to reduce the * and its surrounding id's to an expression. On the other hand, if + takes the precedence over *, we know that parser should reduce E + E to E.

Thus the relative precedence of + followed by * uniquely determines how the parsing conflict between reducing E → E + E and shifting on * in state 7 should be resolved.

Similarly if the input has been **id + id +id** the parser would still reach the configuration in which it had stack 0 E 1 + 4 E 7 after processing input id + id .

On input + there is again shift/reduce conflict in state 7. Now, however, the associativity of the + operator determines how this conflict should be resolved.

If + is left associative, the correct action should to reduce by E → E + E i.e the id's surrounding the first + must be grouped first.

So, we can choose from the conflict which rule should be given preference according to our requirements. Proceeding in this way we get the unambiguous parsing table as follows

STATE	Action Id	+	*	(	)	$	goto
0	s3			s2			1
1		s4	s5			accp	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			8
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	