COMPILER (CSE 4120)

(Lecture 6: Parsing 4 – Bottom-up Parsing)

Sungwon Jung

Mobile Computing & Data Engineering Lab
Dept. of Computer Science and Engineering
Sogang University
Seoul, Korea
Tel: +82-2-705-8930
Email : jungsung@sogang.ac.kr

Dept. of Computer Science and Engineering at Sogang University

Resolving Parsing Conflicts

- Parse conflicts possible when certain LR items are found in the same state.
- Depending on parser, may choose between LR items using lookahead
- Legal lookahead for LR items must be disjoint, else conflict exists.

	Shift-Reduce [A := α. , δ] [B := β.γ , η]	Reduce-Reduce [A := α. , δ] [B := β. , η]
LR(0)	conflict	conflict
SLR(1)	Follow(A) ∩ First(γ)	Follow(A) ∩ Follow(B)
LR(1)	δ ∩ First(γ)	δ ∩ η

SLR(1) and LR(1)

- The Grammar

  S' := G   G := E=E | id   E := E +T | T   T := T*id | id

  LR(0) states

  S_0: [S' := .G] [G := .E=E] [G := .id] [E := .E+T]
  [E := .T] [T := .T*id] [T := .id]

  S_1: [G := id.]   FOLLOW(G) = {$}

  [T := id.]   FOLLOW(T) = {$, *, +, =}

  Reduce-reduce conflict in S_1 for lookahead $!!!

  LR(1) states

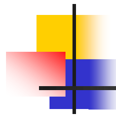
  S_0 : [S' := .G, $] [G := .E=E, $] [G := .id, $] [E := .E+T, =/+]
  [E := .T, =/+] [T := .T*id, =/+/*] [T := .id, =/+/*]

  S_1 : [G := id., $] [T := id., =/+/*]

  Reduce-reduce conflict in S_1 resolved by lookahead !!!

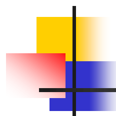
LALR(1) Parsing

- LR(1) parser have many more states than SLR(1) parser
  - approximately factor of ten for Pascal
- LALR(1) parser have same number of states as SLR(1) parsers, but with more power due to lookahead in states
- Define the core of a set of LR(1) items to be the set of LR(0) items derived by ignoring the lookahead symbols
  - Thus, the two sets
    {[A := α.β, a], [A := α.β, b]} and
    {[A := α.β, c], [A := α.β, d]} have the same core.
- Key Idea:
  - If two sets of LR(1) items, I_i and I_j, have the same core, we can merge the states that represent them in the ACTION and GOTO tables



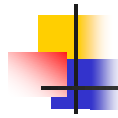
LALR(1) Table Construction

- Build LR(1) sets of items, then merge
  1. For each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union
  2. Update the goto function to reflect the replacement sets

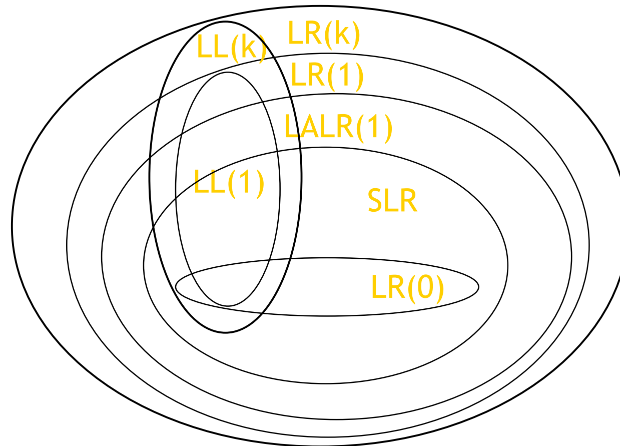


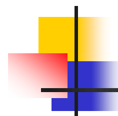
LALR(1) Properties

- LALR(1) parsers have the same number of states as SLR(1) parsers
  - core LR(0) items are the same
- May perform reduce than error
- But will catch error before more input is processed
- LALR derived from LR with no shift-reduce conflict will also have no shift-reduce conflict
- LALR may create reduce-reduce conflict not in LR from which LALR is derived



Classification of Grammars



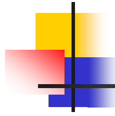


LR Parsing of Ambiguous Grammars

- S := if E then S else S
  S := if E then S
  S := other
- The above grammar rules allows programs such as
  - if *a* then if *b* then *s1* else *s2*
    - (1) if *a* then { if *b* then *s1* else *s2* } --- Correct Interpretation !
    - (2) if *a* then { if *b* then *s1* } else *s2* --- Wrong Interpretation !
- In the LR parsing table there will be a shift-reduce conflict:

***S* := if *E* then *S*.**	**else**	← Reducing
S* := if *E* then *S*. else *S	**(any)**	← Shifting

Shifting : Gives the interpretation (1)
Reducing: Gives the interpretation (2)

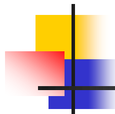


LR Parsing of Ambiguous Grammars

- The ambiguity can be eliminated by introducing auxiliary nonterminals *M* (for matched statement) and *U* (for unmatched statement)

  $S := M \mid U$
  $M :=$ if E then M else M | other
  $U :=$ if E then S | if E then M else U

- Another method: Leave the grammar unchanged and tolerate the shift-reduce conflict
  - Often possible to use ambiguous grammars by resolving shift-reduce conflicts in favor of shifting or reducing, as appropriate
  - In constructing the parsing table, the above shift-reducing conflict can be resolved by shifting over reducing
    - Prefer the interpretation (1) to (2)



Using Parser Generators

- The task of constructing LR(1) or LALR(1) parsing tables is simple enough to automated and so tedious to do by hand for realistic grammars !!!
- Automatic parser generators: Yacc (Yet another compiler-compiler)
  - *Bison* and *occs* : More recent implementations
- A Yacc Specification
  - divided into 3 sections, separated by %% marks

    parser declarations
    %%
    grammar rules
    %%
    programs

An Example of Yacc program without Semantic actions attached

1. P → L
2. S → id := id
3. S → while id do S
4. S → begin L end
5. S → if id then S
6. S → if id then S else S
7. L → S
8. L → L ; S

```
%{
int yylex(void) ;
void yyerror (char *s) { EM_error (EM_tokPos, "%s", s); }
%}
%token ID WHILE BEGIN END DO IF THEN ELSE
       SEMI ASSIGN
%start prog
%%
prog: stmlist
stm : ID ASSIGN ID
    | WHILE ID DO stm
    | BEGIN stmlist END
    | IF ID THEN stm
    | IF ID THEN stm ELSE stm

stmlist : stm
        | stmlist SEMI stm
```


Conflicts

- Yacc reports shift-reduce and reduce-reduce conflicts
  - A shift-reduce conflict: a choice between shifting and reducing
  - A reduce-reduce conflict: a choice of reducing by two different rules
- By default, Yacc resolves
  - Shift-reduce conflict by shifting
  - Reduce-reduce conflicts by using the rule appearing earlier in the grammar
  - See the Figure in the next slide : y.output
    - "yacc -dv filename.y" produces 3 output files:
      - y.tab.c, y.tab.h and y.output
    - Check Yacc's default resolution of the shift-reduce conflict
    - This shift-reduce conflict is not harmful !!! <--- Why ?

Example: LR Parsing Table (Table 3.33)

$E \rightarrow$ id | num | $E * E$ / E / E | $E + E$ | E - E | (E)

	id	num	+	-	*	/	(	)	$	E
1	s2	s3					s4			g7
2			r1	r1	r1	r1		r1	r1	
3			r2	r2	r2	r2		r2	r2	
4	s2	s3					s4			g5
5								s6		
6			r7	r7	r7	r7		r7	r7	
7			s8	s10	s12	s14			a	
8	s2	s3					s4			g9
9			s8,r5	s10,r5	s12,r5	s14,r5		r5	r5	
10	s2	s3					s4			g11
11			s8,r6	s10,r6	s12,r6	s14,r6		r6	r6	
12	s2	s3					s4			g13
13			s8,r3	s10,r3	s12,r3	s14,r3		r3	r3	
14	s2	s3					s4			g15
15			s8,r4	s10,r4	s12,r4	s14,r4		r4	r4	

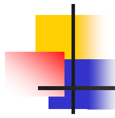
Precedence Directives – 1

- The LR(k) parsing table of an ambiguous grammar will always have conflicts
  - Ambiguous grammars can still be useful if we can find ways to resolve the conflicts
  - Example Grammar:

    $E \rightarrow$ id | num | $E * E$ / E / E | $E + E$ | E - E | (E)
    - Should be parsed so that * and / bind more tightly than + and -, and each operator associates to the left
    - See ***state 13*** with lookahead + in Table 3.33
      - Conflict between *shift into state 8* and *reduce by rule 3*

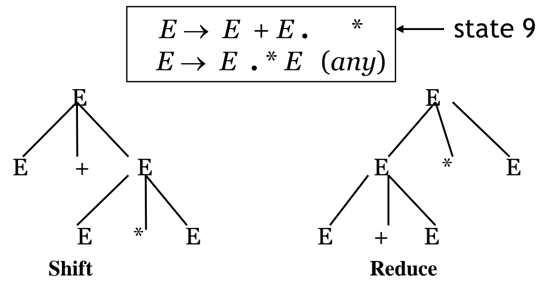
        $$E \rightarrow E * E . \quad +$$
        $$E \rightarrow E . + E \quad (any)$$

      - In this state the top of the stack is ··· E * E



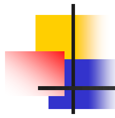
Precedence Directives – 3

- Example Grammar (Cont'd):
  - See ***state 9*** with lookahead *
    - Conflict between *shift into state 12* and *reduce by rule5*: (*s12, r5*)
    - In this state the top of the stack is ··· E + E



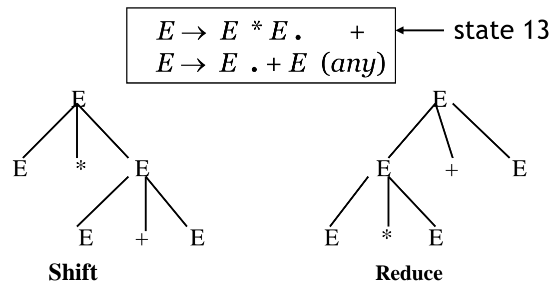
- **If we wish * to bind tighter than +, then shift instead of reduce !!!**

Dept. of Computer Science and Engineering at Sogang University



Precedence Directives – 2

- Example Grammar (Cont'd):
  - See ***state 13*** with lookahead +
    - Conflict between *shift into state 8* and *reduce by rule 3*: (*s8, r3*)
    - In this state the top of the stack is ··· E * E



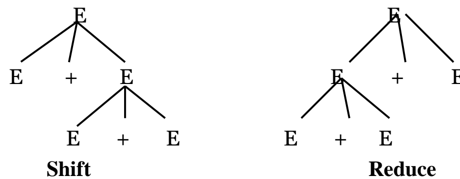
- **If we wish * to bind tighter than +, reduce instead of shift !!!**

Associativity – 1

- Example Grammar (Cont'd):
  - See ***state 9*** with lookahead +
    - Conflict between *shift into state 8* and *reduce by rule5*: (*s8, r5*)
    - In this state the top of the stack is ··· E + E

$$E \to E + E\,.\quad +$$
$$E \to E\,. + E\ (any)$$

← state 9



- Shifting(Reducing) will make the operator right(left)-associative !!!

Associativity – 2

- Left-associativity means
  - Conflict should be broken in favor of reduce !!!
- Right-associativity means
  - Conflict should be broken in favor of shift !!!
- Nonassociative
  - "-" is usually left-associative
    - a - b - c usually means (a - b) - c
  - Can make "-" operator nonassociative
    - Instead of expressing a - b - c , express either (a-b)-c or a-(b-c)

Declaration of Precedence Directives in Yacc

%nonassoc EQ NEQ
%left PLUS MINUS
%left TIMES DIV
%right EXP

- The above declaration indicates
  - + and - are left-associative and bind equally tightly
  - * and / are left-associative and bind more tightly than + and -
  - ^ is right-associative and binds most tightly
  - = and ≠ are non-associative, and bind most weakly than + and -

$$E \rightarrow E * E . \quad +$$
$$E \rightarrow E . + E \ (any)$$

The priority of a rule is given by the last token occurring on the right hand side of that rule