COMS W4115
Programming Languages and Translators
Lecture 11: Parsing Action Conflicts
February 27, 2013

Lecture Outline

1. Parsing action conflicts
2. Resolving shift/reduce conflicts
3. Using Yacc to generate LALR(1) parsers
4. Using Yacc with ambiguous grammars
5. Error recovery in Yacc

1. Parsing Action Conflicts

- If a grammar is not SLR(1), the SLR parsing table construction produces one or more multiply defined entries in the parsing table action function.
- These entries are either *shift/reduce conflicts* or *reduce/reduce conflicts.*

2. Resolving Shift/Reduce Conflicts

- **Example.** Given the augmented grammar *G′*

```
(0) E' â†’ E
(1) E  â†’ E+E
(2) E  â†’ E*E
(3) E  â†’ id
```

FOLLOW(E) = { `+, *, $`}

- Note that this grammar does not specify the relative precedence of the + and `*` operators nor their associativities.
- *C*, the canonical collection of sets of LR(0) items for *G′*, is

```
I_0: E' â†’ Â·E
     E â†’ Â·E+E
     E â†’ Â·E*E
     E â†’ Â·id

I_1: E' â†’ EÂ·
     E â†’ EÂ·+E
     E â†’ EÂ·*E

I_2: E â†’ idÂ·

I_3: E â†’ E+Â·E
     E â†’ Â·E+E
     E â†’ Â·E*E
     E â†’ Â·id

I_4: E â†’ E*Â·E
     E â†’ Â·E+E
     E â†’ Â·E*E
     E â†’ Â·id

I_5: E â†’ E+EÂ·
     E â†’ EÂ·+E
     E â†’ EÂ·*E

I_6: E â†’ E*EÂ·
     E â†’ EÂ·+E
     E â†’ EÂ·*E
```

- SLR(1) parsing table for *G'*:

State	Action				Goto
	id	+		$	E
0	s2				1
1		s3	s4	acc	
2		r3	r3	r3	
3	s2				5
4	s2				6
5		s3 r1	s4 r1	r1	
6		s3 r2	s4 r2	r2	

- Notes
- There is a shift/reduce conflict in `action[5,+]` between "`shift 3`" and "`reduce by E â†’ E+E`" because the associativity of the operator + is not defined by the grammar. This conflict can be resolved in favor of "`reduce by E â†’ E+E`" if we want + to be left associative.
- There is a shift/reduce conflict in `action[5,*]` between "`shift 4`" and "`reduce by E â†’ E+E`" because the relative precedence of the operators + and `*` is not defined by the grammar. This conflict can be resolved in favor of "`shift 4`" if we want `*` to have higher precedence than +.
- Analogously, there is a shift/reduce conflict in `action[6,+]` between "`shift 3`" and "`reduce by E â†’ E*E`" because the relative precedence of the operators + and `*` is not defined by the grammar. This conflict can be resolved in favor of "`reduce by E â†’ E*E`" if we want `*` to have higher precedence than +.
- There is a shift/reduce conflict in `action[6,*]` between "`shift 4`" and "`reduce by E â†’ E*E`" because the associativity of the operator `*` is not defined by the grammar. This conflict can be resolved in favor of "`reduce by E â†’ E*E`" if we want `*` to be left associative.

3. Using Yacc to Generate LALR(1) Parsers

- Consider the yacc program `expr1.y`:

```
%token id
%%
E : E '+' E
 | E '*' E
 | id
 ;
```

- Invoking `yacc -v expr1.y`, we can see the kernels of the sets of items for this grammar in the yacc output file `y.output`.
- The parsing action conflicts are shown for states 5 and 6.

```
state 0
       $accept : _E $end

       id  shift 2
       .  error

       E  goto 1

state 1
       $accept :  E_$end
       E :  E_+ E
       E :  E_* E

       $end  accept
       +  shift 3
       *  shift 4
       .  error

state 2
       E :  id_    (3)

       .  reduce 3

state 3
       E :  E +_E
       id  shift 2
       .  error

       E  goto 5

state 4
       E :  E *_E

       id  shift 2
       .  error

       E  goto 6

5: shift/reduce conflict (shift 3, red'n 1) on +
5: shift/reduce conflict (shift 4, red'n 1) on *
state 5
       E :  E_+ E
       E :  E + E_    (1)
       E :  E_* E

       +  shift 3
       *  shift 4
       .  reduce 1

6: shift/reduce conflict (shift 3, red'n 2) on +
6: shift/reduce conflict (shift 4, red'n 2) on *
state 6
       E :  E_+ E
       E :  E_* E
       E :  E * E_    (2)

       +  shift 3
       *  shift 4
       .  reduce 2


5/127 terminals, 1/600 nonterminals
4/300 grammar rules, 7/1000 states
4 shift/reduce, 0 reduce/reduce conflicts reported
4/601 working sets used
memory: states,etc. 17/2000, parser 2/4000
3/3001 distinct lookahead sets
0 extra closures
9 shift entries, 1 exceptions
3 goto entries
0 entries saved by goto default
Optimizer space used: input 25/2000, output 9/4000
9 table entries, 3 zero
maximum spread: 257, maximum offset: 257
```

4. Using Yacc with Ambiguous Grammars

- We can specify the associativities and relative precedence of the + and `*` operators in the declarations sections of a yacc program.
- Consider the yacc program `expr2.y`:
- The statement `%left '+'` makes + left associative.
- The statement `%left '*'` makes * left associative.
- Since the statement for + comes before the statement for `*`, + has lower precedence than `*`.

```
%token id
%left '+'
%left '*'
%%
E : E '+' E
 | E '*' E
 | id
 ;
```

- Invoking `yacc -v expr2.y`, we can now see that no shift-reduce conflicts have been generated in the yacc output file `y.output`.

```
state 0
       $accept : _E $end

       id  shift 2
       .  error

       E  goto 1

state 1
       $accept :  E_$end
       E :  E_+ E
       E :  E_* E

       $end  accept
       +  shift 3
       *  shift 4
       .  error

state 2
       E :  id_    (3)

       .  reduce 3

state 3
       E :  E +_E

       id  shift 2
       .  error

       E  goto 5

state 4
       E :  E *_E

       id  shift 2
       .  error

       E  goto 6

state 5
       E :  E_+ E
       E :  E + E_    (1)
       E :  E_* E

       *  shift 4
       .  reduce 1

state 6
       E :  E_+ E
       E :  E_* E
       E :  E * E_    (2)

       .  reduce 2


5/127 terminals, 1/600 nonterminals
4/300 grammar rules, 7/1000 states
0 shift/reduce, 0 reduce/reduce conflicts reported
4/601 working sets used
memory: states,etc. 17/2000, parser 2/4000
3/3001 distinct lookahead sets
0 extra closures
6 shift entries, 1 exceptions
```

```
3 goto entries
0 entries saved by goto default
Optimizer space used: input 19/2000, output 10/4000
10 table entries, 3 zero
maximum spread: 257, maximum offset: 257
```

- Unless otherwise instructed, Yacc resolves all parsing action conflicts using the following two rules:

1. A reduce/reduce conflict is resolved by choosing the conflicting production listed first in the Yacc specification.
2. A shift/reduce conflict is resolved in favor of shift. Note that this rule correctly resolves the shift/reduce conflict arising from the dangling-else ambiguity.

5. Error Recovery in Yacc

- In Yacc, error recovery can be performed with error productions.
- To process errors at the level of a nonterminal A, add an error production `A â†’ error Î±` where `error` is a Yacc reserved word, and Î± is a string of grammar symbols, possibly empty. Yacc will generate a parser including this production.
- If Yacc encounters an error during parsing, it pops symbols from its stack until it finds the topmost state on its stack whose underlying set of items includes an item of the form

  ```
  A â†’ . error Î±
  ```

  The parser then shifts the special token `error` onto the stack as though it saw the token `error` on the input.
- If Î± is not empty, Yacc skips ahead on the input looking for a substring that can be "reduced" to Î± on the stack. Now the parser will have `error Î±` on top of its stack, which it will reduce to A. The parser then resumes normal parsing.
- Example: See Fig. 4.61, p. 296. This Yacc specification contains a translation rule

```
    lines : error '\n'  { yyerror("reenter previous line:"); yyerrok; }
```

On encountering an error, the parser starts popping symbols from its stack until it encounters a state with a shift action on `error`. The parser shifts the token `error` on to the stack and then skips ahead in the input until it finds a newline which it shifts onto the stack. The parser reduces `error '\n'` to `lines` and emits the error message "reenter previous line:". The special Yacc routine `yyerrok` resets the parser to its normal mode of operation.

6. Practice Problems

Consider the following grammar G:

```
(1) S â†’ a S b S
(2) S â†’ b S a S
(3) S â†’ Îµ
```

- What language does G generate?
- Construct an SLR(1) parsing table for G.
- Explain why the parsing action conflicts arise in the parsing table.
- Construct an equivalent LALR(1) grammar for L(G).
- Show that your grammar is LALR(1) by using yacc to construct an LALR(1) parsing table for it.
- Is your grammar LL(1)?

7. Reading

- ALSU, Sects. 4.8 and 4.9

---

aho@cs.columbia.edu