CS412/413

Introduction to Compilers
Radu Rugina

Lecture 9: LR(1) Parsing
13 Feb 04

LR(0) Parsing Summary

- LR(0) state = set of LR(0) items
- LR(0) item = a production with a dot in RHS
- Compute LR(0) states and build DFA:
  - Use the closure operation to compute states
  - Use the goto operation to compute transitions between states
- Build the LR(0) parsing table from the DFA
- Use the LR(0) parsing table to determine whether to reduce or to shift

LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a single reduce action
- With more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Need to use look-ahead to choose

ok	shift /reduce	reduce / reduce
L → L , S .	L → L , S . S → S . , L	L → S , L . L → S .

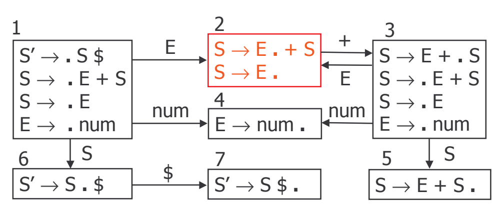
LR(0) Parsing Table

	(	)	id	,	$	S	L
1	s3		s2			g4	
2	S→id	S→id	S→id	S→id	S→id		
3	s3		s2			g7	g5
4					accept		
5		s6		s8			
6	S→(L)	S→(L)	S→(L)	S→(L)	S→(L)		
7	L→S	L→S	L→S	L→S	L→S		
8	s3		s2			g9	
9	L→L,S	L→L,S	L→L,S	L→L,S	L→L,S		

A Non-LR(0) Grammar

- Grammar for addition of numbers:
  S → S + E | E
  E → num
- Left-associative version is LR(0)
- Right-associative version is not LR(0)
  S → E + S | E
  E → num

LR(0) Parsing Table



What to do in state 2: shift or reduce?

	num	+	$	E	S
1	s4			g2	g6
2	S→E	s3/S→E	S→E		

SLR Parsing

- SLR Parsing = easy extension of LR(0)
  - For each reduction $X \to \gamma$: look at the next symbol C
  - Apply reduction only if C is in FOLLOW(X)
- SLR parsing table eliminates some conflicts
  - Same as LR(0) table except reduction rows
  - Adds reductions $X \to \gamma$ only in the columns of symbols in FOLLOW(X)
- Example:

  FOLLOW(S)={$}

	num	+	$	E	S
1	s4			g2	g6
2		s3	S→E		

SLR Parsing Table

- Reductions do not fill entire rows
- Otherwise, same as LR(0)

	num	+	$	E	S
1	s4			g2	g6
2		s3	S→E		
3	s4			g2	g5
4			S→E		
5			S→E+S		
6			s7		
7			accept		

LR(1) Parsing

- Get as much power as possible out of 1 look-ahead symbol parsing table
- LR(1) grammar = recognizable by a shift/reduce parser with 1 look-ahead
- LR(1) parsing uses similar concepts as LR(0)
  - Parser states = sets of items
  - LR(1) item = LR(0) item + look-ahead symbol possibly following production

LR(0) item : $\boxed{S \to .S + E}$

LR(1) item : $\boxed{S \to .S + E \quad +}$

LR(1) States

- LR(1) state = set of LR(1) items
- LR(1) item = ($X \to \alpha . \beta$, y)
- Meaning: α already matched at top of the stack; next expect to see β y
- Shorthand notation

  ($X \to \alpha . \beta$, $\{x_1, \ldots, x_n\}$)

  means:

  ($X \to \alpha . \beta$, x_1)

  ...

  ($X \to \alpha . \beta$, x_n)

S → S . + E	+,$
S → S + . E	num

- Extend closure and goto operations

LR(1) Closure

- LR(1) closure operation:
  - Start with Closure(S) = S
  - For each item in S:

    $X \to \alpha . Y \beta$, z

    and for each production $Y \to \gamma$, add the following item to the closure of S:

    $Y \to . \gamma$, FIRST(βz)
  - Repeat until nothing changes
- Similar to LR(0) closure, but also keeps track of the look-ahead symbol

LR(1) Start State

- Initial state: start with (S' → . S , $), then apply the closure operation
- Example: sum grammar

  S' → S $

  S → E + S | E

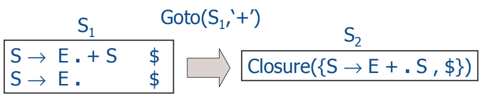
  E → num

$\boxed{S' \to . S \quad \$}$ $\xrightarrow{\text{closure}}$

S' → . S	$
S → . E + S	$
S → . E	$
E → . num	+,$

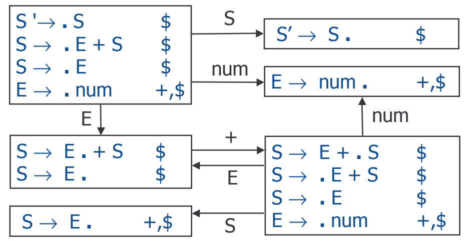
LR(1) Goto Operation

- LR(1) goto operation = describes transitions between LR(1) states
- Algorithm: for a state S and a symbol Y
  - S′ = {(X → α**Y.**β, z) | (X → α**.Y**β, z) ∈ S}
  - Goto(S, Y) = Closure(S′)



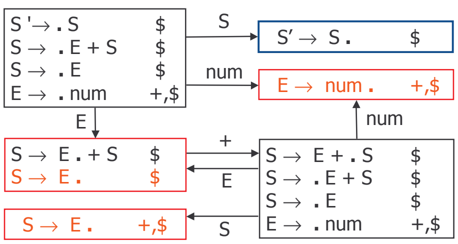
LR(1) DFA Construction

- If S′ = goto (S,x) then add an edge labeled x from S to S′



LR(1) Reductions

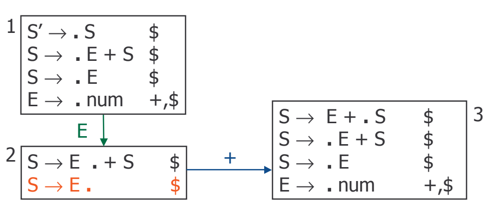
- Reductions correspond to LR(1) items of the form (X → γ **.** , y)



LR(1) Parsing Table Construction

- Same as construction of LR(0) parsing table, except for reductions
- For a transition S → S′ on terminal x:
  Shift(S′) ⊆ Table[S,x]
- For a transition S → S′ on non-terminal N:
  Goto(S′) ⊆ Table[S,N]
- If (X → γ **.** , y) ∈ S, then:
  Reduce(X → γ) ⊆ Table[S,y]

LR(1) Parsing Table Example



Fragment of the Parsing table:

	+	$	E
1			2
2	s3	S→E	

LALR(1) Grammars

- Problem with LR(1): too many states
- LALR(1) Parsing (Look-Ahead LR)
  - Constructs LR(1) DFA and then merge any two LR(1) states whose items are identical except look-ahead
  - Results in smaller parser tables
  - Theoretically less powerful than LR(1)

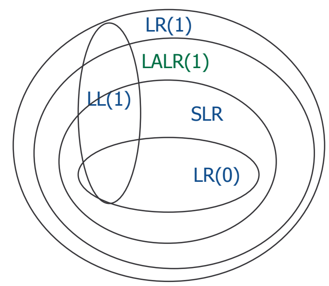
S → id . + S → E . $	+	S → id . $ S → E . +	= ?

- LALR(1) Grammar = a grammar whose LALR(1) parsing table has no conflicts

LL/LR Grammar Summary

- LL parsing tables
  - Nonterminals x terminals → productions
  - Computed using FIRST/FOLLOW
- LR parsing tables
  - LR states x terminals → {shift/reduce}
  - LR states x non-terminals → goto
  - Computed using closure/goto operations on LR states
- A grammar is:
  - LL(1) if its LL(1) parsing table has no conflicts
  - LR(0) if its LR(0) parsing table has no conflicts
  - SLR if its SLR parsing table has no conflicts
  - LALR(1) if its LALR(1) parsing table has no conflicts
  - LR(1) if its LR(1) parsing table has no conflicts

Classification of Grammars



$LR(k) \subseteq LR(k+1)$
$LL(k) \subseteq LL(k+1)$
$LL(k) \subseteq LR(k)$

$LR(0) \subseteq SLR$

$LALR(1) \subseteq LR(1)$

Automate the Parsing Process

- Can automate:
  - The construction of LR parsing tables
  - The construction of shift-reduce parsers based on these parsing tables
- Automatic parser generators: yacc, bison, CUP
- LALR(1) parser generators
  - No much difference compared to LR(1) in practice
  - Smaller parsing tables than LR(1)
  - Augment LALR(1) grammar specification with declarations of precedence, associativity
- output: LALR(1) parser program

Associativity

S → S + E | E
E → num

⇒

E → E + E
E → num

What happens if we run this grammar through LALR construction?

Shift/Reduce Conflict

E → E + E
E → num

E → E + E .	+
E → E . + E	+,$

$\xrightarrow{+}$

shift/reduce conflict

shift: 1+(2+3)
reduce: (1+2)+3

1+2+3
 ^

Grammar in CUP

non terminal E; terminal PLUS, LPAREN...
precedence left PLUS;

"when shifting '+' conflicts with reducing a production, choose reduce"

```
E ::= E  PLUS  E
    |  LPAREN  E  RPAREN
    |  NUMBER ;
```

Precedence

- CUP can also handle operator precedence

$E \rightarrow E + E \mid T$
$T \rightarrow T \times T \mid num \mid (E)$

⇩

$E \rightarrow E + E \mid E \times E$
$\quad \mid num \mid (E)$

Conflicts without Precedence

$E \rightarrow E + E \mid E \times E$
$\quad \mid num \mid (E)$

$E \rightarrow E . + E$	...	
$E \rightarrow E \times E .$	+	

$E \rightarrow E + E .$	$\times$	
$E \rightarrow E . \times E$	...	

Predecence in CUP

precedence left PLUS;
precedence left TIMES; // TIMES > PLUS
E ::= E PLUS E | E TIMES E | ...

RULE: in conflict, choose reduce if production symbol higher precedence than shifted symbol; choose shift if vice-versa

$E \rightarrow E . + E$	...
$E \rightarrow E \times E .$	+

reduce $E \rightarrow E \times E$

$E \rightarrow E + E .$	$\times$
$E \rightarrow E . \times E$	...

Shift $\times$

Summary

- Look-ahead information makes SLR(1), LALR(1), LR(1) grammars expressive
- Automatic parser generators support LALR(1) grammars
- Precedence, associativity declarations simplify grammar writing