Compiler Design

CLR and LALR

Amey Karkare
Department of Computer Science and Engineering
IIT Kanpur
karkare@iitk.ac.in

Canonical LR Parsing

- Carry extra information in the state so that wrong reductions by A → α will be ruled out
- Redefine LR items to include a terminal symbol as a second component (look ahead symbol)
- The general form of the item becomes [A → α.β, a] which is called LR(1) item.
- Item [A → α., a] calls for reduction only if next input is a. The set of symbols “a”s will be a subset of Follow(A).

Closure(I)

```
repeat
  for each item [A → α.Bβ, a] in I
      for each production B → γ in G'
      and for each terminal b in First(βa)
            add item [B → .γ, b] to I
until no more additions to I
```

Example

Consider the following grammar

S‘→ S
S → CC
C → cC | d

Compute closure(I) where I={[S’ → .S, $]}

S‘→ .S,	$
S → .CC,	$
C → .cC,	c
C → .cC,	d
C → .d,	c
C → .d,	d

Example

Construct sets of LR(1) items for the grammar on previous slide

I_0: S' → .S,	$
S → .CC,	$
C → .cC,	c/d
C → .d,	c/d
I_1: goto(I_0,S)	
S' → S.,	$
I_2: goto(I_0,C)	
S → C.C,	$
C → .cC,	$
C → .d,	$
I_3: goto(I_0,c)	
C → c.C,	c/d
C → .cC,	c/d
C → .d,	c/d

I_4: goto(I_0,d)	
C → d.,	c/d
I_5: goto(I_2,C)	
S → CC.,	$
I_6: goto(I_2,c)	
C → c.C,	$
C → .cC,	$
C → .d,	$
I_7: goto(I_2,d)	
C → d.,	$
I_8: goto(I_3,C)	
C → cC.,	c/d
I_9: goto(I_6,C)	
C → cC.,	$

Construction of Canonical LR parse table

- Construct C={$I_{0,\ldots,}I_n$} the sets of LR(1) items.
- If [A → α.aβ, b] is in I_i and goto(I_i, a)=I_j then action[i,a]=shift j
- If [A → α., a] is in I_i then action[i,a] reduce A → α
- If [S' → S., $] is in I_i then action[i,$] = accept
- If goto(I_i, A) = I_j then goto[i,A] = j for all non terminals A

Parse table

State	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Notes on Canonical LR Parser

- Consider the grammar discussed in the previous two slides. The language specified by the grammar is c*dc*d.
- When reading input cc…dcc…d the parser shifts cs into stack and then goes into state 4 after reading d. It then calls for reduction by C→d if following symbol is c or d.
- IF $ follows the first d then input string is c*d which is not in the language; parser declares an error
- On an error canonical LR parser never makes a wrong shift/reduce move. It immediately declares an error
- Problem: Canonical LR parse table has a large number of states

LALR Parse table

- Look Ahead LR parsers
- Consider a pair of similar looking states (same kernel and different lookaheads) in the set of LR(1) items
  I_4: C → d. , c/d $\qquad$ I_7: C → d., \$
- Replace I_4 and I_7 by a new state I_{47} consisting of (C → d., c/d/\$)
- Similarly I_3 & I_6 and I_8 & I_9 form pairs
- Merge LR(1) items having the same core

Construct LALR parse table

- Construct C={I_0,……,I_n} set of LR(1) items
- For each core present in LR(1) items find all sets having the same core and replace these sets by their union
- Let C' = {J_0,…….,J_m} be the resulting set of items
- Construct action table as was done earlier
- Let J = I_1 U I_2…….U I_k

  since I_1 , I_2……., I_k have same core, goto(J,X) will have the same core

  Let K=goto(I_1,X) U goto(I_2,X)……goto(I_k,X) then goto(J,X)=K

LALR parse table …

State	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Notes on LALR parse table

- Modified parser behaves as original except that it will reduce C→d on inputs like ccd. The error will eventually be caught before any more symbols are shifted.
- In general core is a set of LR(0) items and LR(1) grammar may produce more than one set of items with the same core.
- Merging items never produces shift/reduce conflicts but may produce reduce/reduce conflicts.
- SLR and LALR parse tables have same number of states.

Notes on LALR parse table…

- Merging items may result into conflicts in LALR parsers which did not exist in LR parsers

- New conflicts can not be of shift reduce kind:
  - Assume there is a shift reduce conflict in some state of LALR parser with items
    $\{[X \rightarrow \alpha.,a],[Y \rightarrow \gamma.a\beta,b]\}$
  - Then there must have been a state in the LR parser with the same core
  - Contradiction; because LR parser did not have conflicts

- LALR parser can have new reduce-reduce conflicts
  - Assume states
    $\{[X \rightarrow \alpha., a], [Y \rightarrow \beta., b]\}$ and $\{[X \rightarrow \alpha., b], [Y \rightarrow \beta., a]\}$
  - Merging the two states produces
    $\{[X \rightarrow \alpha., a/b], [Y \rightarrow \beta., a/b]\}$

Notes on LALR parse table…

- LALR parsers are not built by first making canonical LR parse tables

- There are direct, complicated but efficient algorithms to develop LALR parsers

- Relative power of various classes

  - SLR(1) ≤ LALR(1) ≤ LR(1)

  - SLR(k) ≤ LALR(k) ≤ LR(k)

  - LL(k) ≤ LR(k)

Error Recovery

- An error is detected when an entry in the action table is found to be empty.

- Panic mode error recovery can be implemented as follows:

  - scan down the stack until a state **S** with a goto on a particular nonterminal **A** is found.

  - discard zero or more input symbols until a symbol **a** is found that can legitimately follow **A**.

  - stack the state **goto[S,A]** and resume parsing.

- **Choice of A, a:** Normally **A** is chosen from non terminals representing major program pieces such as an expression, statement or a block. For example if **A** is the nonterminal **stmt**, **a** might be **semicolon** or **end**.

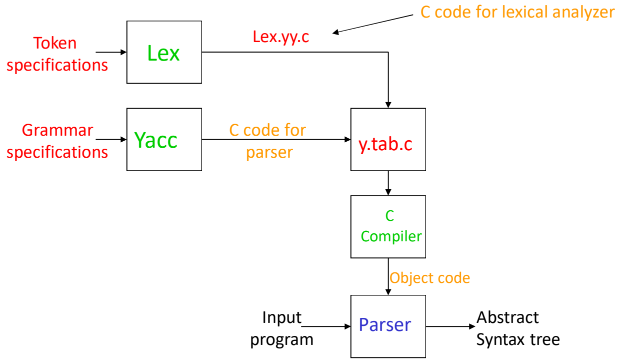
Parser Generator

- Some common parser generators

  - YACC: **Y**et **A**nother **C**ompiler **C**ompiler
  - Bison: GNU Software
  - ANTLR: **AN**other **T**ool for **L**anguage **R**ecognition

- Yacc/Bison source program specification (accept LALR grammars)
  declaration
  %%
  translation rules
  %%
  supporting C routines

Yacc and Lex schema



Refer to YACC Manual

Bottom up parsing …

- A more powerful parsing technique
- LR grammars – more expensive than LL
- Can handle left recursive grammars
- Can handle virtually all the programming languages
- Natural expression of programming language syntax
- Automatic generation of parsers (Yacc, Bison etc.)
- Detects errors as soon as possible
- Allows better error recovery