CS6109 – COMPILER DESIGN

Module – 6

Presented By
Dr. S. Muthurajkumar,
Assistant Professor,
Dept. of CT, MIT Campus,
Anna University, Chennai.

MODULE - 6

- CLR Parsers
- LALR Parsers
- Parser Generators
- Design of parser generator

CLR PARSING TABLE

- INPUT: An augmented grammar G’.
- OUTPUT: The sets of LR(1) items that are the set of items valid for one or more viable prefixes of G’.
- We need a way to bring the notion of following tokens much closer to the productions that use them.
- An LR(1) item has the form [I, t] where I is an LR(0) item and t is a token.
- As the dot moves through the right-hand side of I, token t remains attached to it. LR(1) item $[A \rightarrow \alpha \cdot, t]$ calls for a reduce action when the lookahead is t.
- The follow context is determined when an LR(1) item $[A \rightarrow \cdot\ \alpha, t]$ is created, and is carried along with it. Token t is a token that can follow A in the context in which the A occurs. This does not suffer from the difficulty of using FOLLOW sets to determine when to do reduce actions.

CLR PARSING TABLE

- Example:
- S → CC
- C → c C | d
- Production with numbers

1. S → CC
2. C → c C
3. C → d

- Add Augment production in the given grammar. **S' → S**
- S' → S
- S → CC
- C → c C
- C → d

CLR PARSING TABLE

- Production with numbers

1. S → CC
2. C → c C
3. C → d

- Add Augment production in the given grammar. **S' → S**
- S' → S
- S → CC
- C → c C
- C → d

First (S) = {c, d}	Follow (S) = {\$}
First (S') = {c, d}	Follow (S') = {\$}
First (C) = {c, d}	Follow (C) = {c, d}

CLR PARSING TABLE

- Canonical collection of LR(1) items

  S’ → S

  S → CC

  C → c C

  C → d

The initial state has kernel {[S′ → · S, $]}

$$\left.\begin{array}{l} \text{S' → .S, \$} \\ \text{S → .CC, \$} \\ \text{C → .c C, c/d} \\ \text{C → .d, c/d} \end{array}\right\} \textbf{I0}$$

GOTO(I0, S)

S’ → S., $ ➔ **I1**

GOTO(I0, C)

S → C.C, $

C → .c C, $

C → .d, $ ➔ **I2**

GOTO(I0, c)

C → c . C, c/d

S → .c C, c/d

C → .d, c/d ➔ **I3**

CLR PARSING TABLE

GOTO(I0, d)
C → d. , c/d ➔ **I4**

GOTO(I2, C)
S → CC. , $ ➔ **I5**

GOTO(I2, c)
C → c . C, $
C → .c C, $
C → .d, $ ➔ **I6**

GOTO(I2, d)
C → d., $ ➔ **I7**

GOTO(I3, C)
C → c C. , c/d➔ **I8**

GOTO(I5, C)
C → c C. , $ ➔ **I9**

GOTO(I3, c)
C → c . C, c/d
S → .c C, c/d
C → .d, c/d ➔ **I3**

GOTO(I2, c)
C → c . C, $
C → .c C, $
C → .d, $ ➔ **I6**

CLR PARSING TABLE

GOTO(I2, c)
C → c . C, $
C → .c C, $
C → .d, $ ➔ **I6**

GOTO(I3, d)
C → d. , c/d ➔ **I4**

GOTO(I6, C)
C → c C., $ ➔ **I9**

GOTO(I6, c)
C → c . C, $
C → .c C, $
C → .d, $ ➔ **I6**

GOTO(I6, d)
C → d., $ ➔ **I7**

CLR PARSING TABLE

- INPUT: An augmented grammar G'.
- OUTPUT: The canonical-LR parsing table function ACTION and GOTO for G'.
- METHOD:

1. Construct C' = {I_0, I_1, … , I_n}, the collection of sets of LR(1) items for G'.
2. State i of the parser is constructed from I_i, The parsing action for state I is determined as follows.

   (a) If [A → α.aβ, b] is in Ii and GOTO(I_i, a) = I_j, then set ACTION[i, a] to "shift j". Here a must be a terminal.

   (b) If [A → α., a] is in I_i, A ≠ S', then set ACTION[I, a] to "reduce A → α".

   (c) If [S' → S., $] is in I_i, then set ACTION[i, $] to "accept".

   If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.

CLR PARSING TABLE

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If GOTO(I_i, A) = I_j , then GOTO[i, A] = j.
4. All entries not defined by rules (2) and (3) are made “error”.
5. The initial state of the parser is the one constructed from the set of items containing [S’ $\rightarrow$ .S, $].

CLR PARSING TABLE

State	Action			GOTO	
	c	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

CLR PARSING TABLE

1) S → CC 2) C → c C 3) C → d

- Input cccdcd $

Stack	Symbols	Input	Action
0		cccdcd $	0 → c → S3 Push 3 and shift input c
0 3	c	ccdcd $	0 → c → S3 Push 3 and shift input c
0 3 3	cc	cdcd $	0 → c → S3 Push 3 and shift input c
0 3 3 3	ccc	dcd $	0 → c → S4 Push 4 and shift input c
0 3 3 3 4	cccd	cd $	4 → c → R3 (C → d) Pop 4 and Reduce d by C
0 3 3 3	cccC	cd $	3 → C → 8 Push 8

State	Action			GOTO	
	c	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

CLR PARSING TABLE

1) S → CC   2) C → c C   3) C → d

- Input cccdcd $

Stack	Symbols	Input	Action
0 3 3 3 8	cccC	cd $	8 → c → R2 (C → cC) Pop 8, 3 and Reduce cC by C
0 3 3 8	ccC	cd $	3 → C → 8 Push 8
0 3	cC	cd $	8 → c → R2 (C → cC) Pop 8, 3 and Reduce cC by C
0 3 8	cC	cd $	3 → C → 8 Push 8
0 3 8	C	cd $	8 → c → R2 (C → cC) Pop 8, 3 and Reduce cC by C
0	C	c d $	0 → C → 2 Push 2

State	Action			GOTO	
	c	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

CLR PARSING TABLE

1) $S \rightarrow CC$ 2) $C \rightarrow c\,C$ 3) $C \rightarrow d$

- Input cccdcd $

Stack	Symbols	Input	Action
0 2	C	c d $	2 → c → S6 Push 6 and Shift input c
0 2 6	Cc	d $	6 → d → S7 Push 7 and Shift input d
0 2 6 7	Ccd	$	7 → $ → R3 (C → d) Pop 7 and Reduce d by C
0 2 6	CcC	$	6 → C → 9 Push 9
0 2 6 9	CcC	$	9 → $ → R2 (C → cC) Pop 9, 6 and Reduce cC by C
0 2 5	CC	$	2 → C → 5 Push 5

State	Action			GOTO	
	c	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

CLR PARSING TABLE

1) S → CC   2) C → c C   3) C → d

- Input cccdcd $

Stack	Symbols	Input	Action
0 2 5	CC	$	5 → $ → R1 (S → CC) Pop 5,2 and Reduce CC by S
0	S	$	0 → 1 → 1 Push 1
0 1	S	$	1 → $ → Accept

State	Action			GOTO	
	c	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

LALR PARSING TABLE

- The LALR (lookahead-LR) technique.
- This method is often used in practice, because the tables obtained by it are considerably smaller than the canonical LR tables, yet most common syntactic constructs of programming languages can be expressed conveniently by an LALR grammar.
- The same is almost true for SLR grammars, but there are a few constructs that cannot be conveniently handled by SLR Techniques.

LALR PARSING TABLE

- Example:
- S → CC
- C → c C | d
- Production with numbers

1. S → CC
2. C → c C
3. C → d

- Add Augment production in the given grammar. **S' → S**
- S' → S
- S → CC
- C → c C
- C → d

LALR PARSING TABLE

- Production with numbers

1. S → CC
2. C → c C
3. C → d

- Add Augment production in the given grammar.   **S' → S**
- S' → S
- S → CC
- C → c C
- C → d

First (S) = {c, d}	Follow (S) = {$}
First (S') = {c, d}	Follow (S') = {$}
First (C) = {c, d}	Follow (C) = {c, d}

LALR PARSING TABLE

- Canonical collection of LR(1) items

  S’ → S

  S → CC

  C → c C

  C → d

  The initial state has kernel {[S′ → · S, $]}

$$\left.\begin{array}{l} S' \to .S,\ \$ \\ S \to .CC,\ \$ \\ C \to .c\,C,\ c/d \\ C \to .d,\ c/d \end{array}\right\}\ \textbf{I0}$$

GOTO(I0, S)
S’ → S., $ ➔ **I1**

GOTO(I0, C)
S → C.C, $
C → .c C, $
C → .d, $ ➔ **I2**

GOTO(I0, c)
C → c . C, c/d
S → .c C, c/d
C → .d, c/d ➔ **I3**

LALR PARSING TABLE

GOTO(I0, d)
C → d. , c/d **→ I4**

GOTO(I2, C)
S → CC. , $ **→ I5**

GOTO(I2, c)
C → c . C, $
C → .c C, $
C → .d, $ **→ I6**

GOTO(I2, d)
C → d., $ **→ I7**

GOTO(I3, C)
C → c C. , c/d **→ I8**

GOTO(I5, C)
C → c C. , $ **→ I9**

GOTO(I3, c)
C → c . C, c/d
C → .c C, c/d
C → .d, c/d **→ I3**

GOTO(I2, c)
C → c . C, $
C → .c C, $
C → .d, $ **→ I6**

LALR PARSING TABLE

GOTO(I2, c)
C → c . C, $
C → .c C, $
C → .d, $ ➔ **I6**

GOTO(I3, d)
C → d. , c/d ➔ **I4**

GOTO(I6, C)
C → c C., $ ➔ **I9**

GOTO(I6, c)
C → c . C, $
C → .c C, $
C → .d, $ ➔ **I6**

GOTO(I6, d)
C → d., $ ➔ **I7**

LALR PARSING TABLE

- INPUT: An augmented grammar G’.
- OUTPUT: The LALR parsing-table functions ACTION and GOTO for G’.
- METHOD:

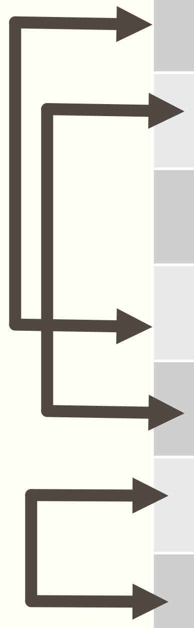
1. Construct C = $\{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items.
2. For each core present among the set of LR(1) items, nd all sets having that core, and replace these sets by their union.
3. Let C’ = $\{J_0, J_1, \ldots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as in CLR Algorithm. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

LALR PARSING TABLE

4. The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is, $J = I_1 \quad I_2 \quad \dots \quad I_k$, then the cores of GOTO(I_1,X), GOTO(I_2,X), … , GOTO(I_k,X) are the same, since I_1, I_2, …, I_k all have the same core. Let K be the union of all sets of items having the same core as GOTO(I_1,X). Then GOTO(J, X) = K.

CLR PARSING TABLE

State	Action			GOTO	
	c	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		



LALR PARSING TABLE

State	Action			GOTO	
	c	d	$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

CLR PARSING TABLE

1) S → CC   2) C → c C   3) C → d

- Input ccd $

Stack	Symbols	Input	Action
0		ccd $	0 → c → S3 Push 3 and shift input c
0 3	c	cd $	0 → c → S3 Push 3 and shift input c
0 3 3	cc	d $	3 → c → S3 Push 3 and shift input c
0 3 3 4	cc	d $	3 → d → S4 Push 4 and shift input d
0 3 3 4	ccd	$	4 → $ → Empty Error
			Not Accept

State	Action			GOTO	
	c	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

LALR PARSING TABLE

1) S → CC   2) C → c C   3) C → d

- Input ccd $

Stack	Symbols	Input	Action
0		ccd $	0 → c → S36 Push 36 and shift input c
0 36	c	cd $	0 → c → S36 Push 36 and shift input c
0 36 36	cc	d $	36 → c → S36 Push 36 and shift input c
0 36 36 47	cc	d $	36 → d → S47 Push 47 and shift input d
0 36 36 47	ccd	$	47 → $ → Empty Error
			Not Accept

State	Action			GOTO	
	c	d	$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

PARSER GENERATORS

- This section shows how a parser generator can be used to facilitate the construction of the front end of a compiler.
- We shall use the LALR parser generator Yacc as the basis of our discussion, since it implements many of the concepts discussed in the previous two sections and it is widely available.
- Yacc stands for “yet another compiler-compiler”, reflecting the popularity of parser generators in the early 1970s when the first version of Yacc was created by S. C. Johnson.
- Yacc is available as a command on the UNIX system, and has been used to help implement many production compilers.

PARSER PROBLEMS

1. Example 1

   E → E + T | T

   T → T * F | F

   F → (E) | id

2. Example 2

   S → i S e S | i S | a Input iiaea $

3. Show that the following grammar

   S → A a | b A c | d c | b d a

   A → d

   is LALR(1) but not SLR(1).

PARSER PROBLEMS

1. Example LALR

   S → a A d | b B d | a B e | b A e

   A → c

   B → c

2. LALR

   S → Aa | b

   A → Ac | sd | ϵ

3. LALR

   S → (L) | a

   L → L, S | S

PARSER PROBLEMS

1. Show that the following grammar

   S → A a | b A c | B c | b B a

   A → d

   B → d

   is LR(1) but not LALR(1).

2. LALR

   S → L = R | R

   L → *R | id

   R → L

3. LALR

   S → CC

   C → c C | d

PARSER PROBLEMS

1. Show that the following grammar

   E → E sub E sup E

   E → E sub E

   E → E sup E

   E → {E}

   E → c

   is LR(1) but not LALR(1).

2. CLR and LALR

   S → SS+ | SS* | a

PARSER PROBLEMS - Dangling-Else

- The “Dangling-Else” Ambiguity
- Consider again the following grammar for conditional statements:
- stmt → if expr then stmt else stmt

  | if expr then stmt

  | other
- This grammar is ambiguous because it does not resolve the dangling-else ambiguity.
- To simplify the discussion, let us consider an abstraction of this grammar, where i stands for if expr then, e stands for else, and a stands for “all other productions”. We can then write the grammar, with augmenting production
- S’ → S, as S’ → S
- S → i S e S | i S | a

PARSER PROBLEMS

2. Example 2

   $S \rightarrow i\ S\ e\ S \mid i\ S \mid a$ Input iiaea $

DESIGN OF PARSER GENERATOR

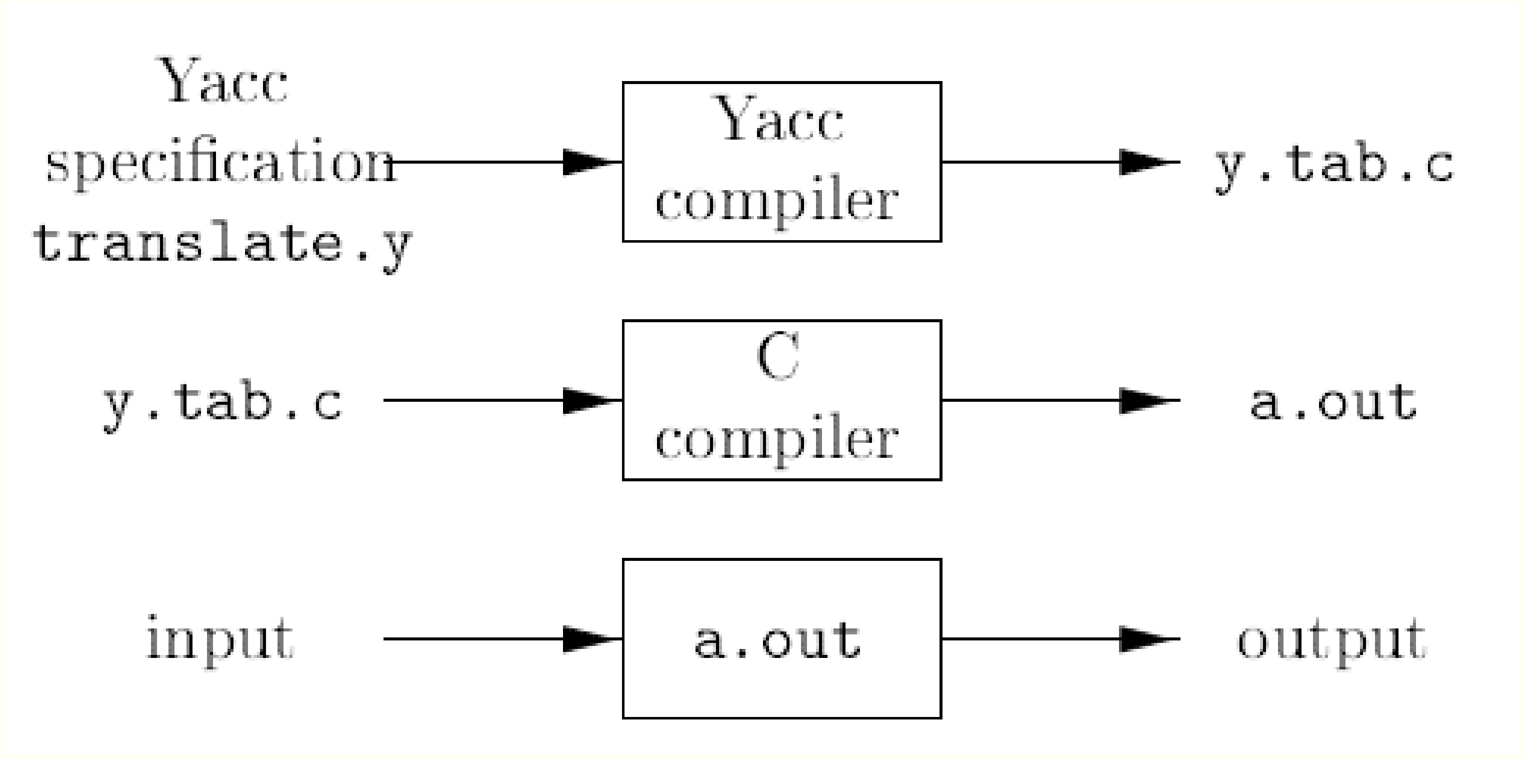
- The Parser Generator Yacc
- Using Yacc with Ambiguous Grammars
- Creating Yacc Lexical Analyzers with Lex
- Error Recovery in Yacc

PARSER GENERATORS

- **The Parser Generator Yacc**
  - First, a file, say translate.y, containing a Yacc specification of the translator is prepared. The UNIX system command
    yacc translate.y
  - Transforms the file translate.y into a C program called y.tab.c using the LALR method
  - The program y.tab.c is a representation of an LALR parser written in C, along with other C routines that the user may have prepared.
  - The LALR by compiling y.tab.c along with the ly library that contains the LR parsing program using the command
    cc y.tab.c -ly
  - we obtain the desired object program a.out that performs the translation specified by the original Yacc. If other procedures are needed, they can be compiled or loaded with y.tab.c, just as with any C program.

PARSER GENERATORS

- **A Yacc source program has three parts:**



PARSER GENERATORS

- **A Yacc source program has three parts:**
- Creating an input/output translator with Yacc

```
declarations
%%
translation rules
%%
supporting C routines
```

PARSER GENERATORS

- **Using Yacc with Ambiguous Grammars**
- Let us now modify the Yacc specification so that the resulting desk calculator becomes more useful. First, we shall allow the desk calculator to evaluate a sequence of expressions, one to a line. We shall also allow blank lines between expressions. We do so by changing the first rule to

```
lines : lines expr '\n' { printf("%g\n", $2); }
| lines '\n'
| /* empty */
;
```

- In Yacc, an empty alternative, as the third line is, denotes .

PARSER GENERATORS

- **Creating Yacc Lexical Analyzers with Lex**
- Lex was designed to produce lexical analyzers that could be used with Yacc. The Lex library ll will provide a driver program named yylex(), the name required by Yacc for its lexical analyzer. If Lex is used to produce the lexical analyzer, we replace the routine yylex() in the third part of the Yacc specification by the statement

  ```
  #include "lex.yy.c"
  ```

- and we have each Lex action return a terminal known to Yacc. By using the #include "lex.yy.c" statement, the program yylex has access to Yacc's names for tokens, since the Lex output file is compiled as part of the Yacc output file y.tab.c.

PARSER GENERATORS

- **Error Recovery in Yacc**
- In Yacc, error recovery uses a form of error productions.
- First, the user decides what “major" nonterminals will have error recovery associated with them.
- Typical choices are some subset of the nonterminals generating expressions, statements, blocks, and functions.
- The user then adds to the grammar error productions of the form A $\rightarrow$ error α, where A is a major nonterminal and α is a string of grammar symbols, perhaps the empty string; error is a Yacc reserved word.
- Yacc will generate a parser from such a specification, treating the error productions as ordinary productions.

REFERENCE

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, “Compilers: Principles, Techniques and Tools”, Second Edition, Pearson Education Limited, 2014.
2. Randy Allen, Ken Kennedy, “Optimizing Compilers for Modern Architectures: A Dependence-based Approach”, Morgan Kaufmann Publishers, 2002.
3. Steven S. Muchnick, “Advanced Compiler Design and Implementation”, Morgan Kaufmann Publishers - Elsevier Science, India, Indian Reprint, 2003.
4. Keith D Cooper and Linda Torczon, “Engineering a Compiler”, Morgan Kaufmann Publishers, Elsevier Science, 2004.
5. V. Raghavan, “Principles of Compiler Design”, Tata McGraw Hill Education Publishers, 2010.
6. Allen I. Holub, “Compiler Design in C”, Prentice-Hall Software Series, 1993.

Thank you