UNIT-4

- **More powerful LR parsers:** Canonical LR(1) items - Constructing LR(1) Set of Items – Canonical LR(1) Parsing Tables - Constructing LALR Parsing tables

- To remove shift reduce conflicts in the SLR parsing table, we are going for another alternative and most expensive and most efficient method is called CLR.
- It works on very large class of grammars

CLR

- The CLR parser stands for canonical LR parser.
- It is a more powerful LR parser.
- It makes use of look ahead symbols.
- This method uses a large set of items called LR(1) items.
- The main difference between LR(0) and LR(1) items is that, in LR(1) items, it is possible to carry more information in a state, which will rule out useless reduction states.
- This extra information is incorporated into the state by the look ahead symbol.

- The general syntax becomes [A->∝.B, a]
- where A->∝.B is the production and a is a terminal or right end marker $
- LR(1) items=LR(0) items + look ahead

- **CASE 1 –**

  A->∝.BC, a [0^{th} production]

  B->.D [1st production]

  After B There is C, So FIRST(C) is look ahead symbol for 1^{st} production. For Ex. If FIRST(C)={d} then

  B->.D, d

- **CASE 2 –**

  A->∝.B, a

  B->.D, a

  Here, we can see there's nothing after B. So the look ahead of 0^{th} production will be the look ahead of 1^{st} production.

- **CASE 3 –**
  Assume a production A->a|b

  A->a,$ [0^{th} production]

  A->b,$ [1^{st} production]

  Here, the 1st production is a part of the previous production, so the look ahead will be the same as that of its previous production.

Steps for constructing CLR parsing table

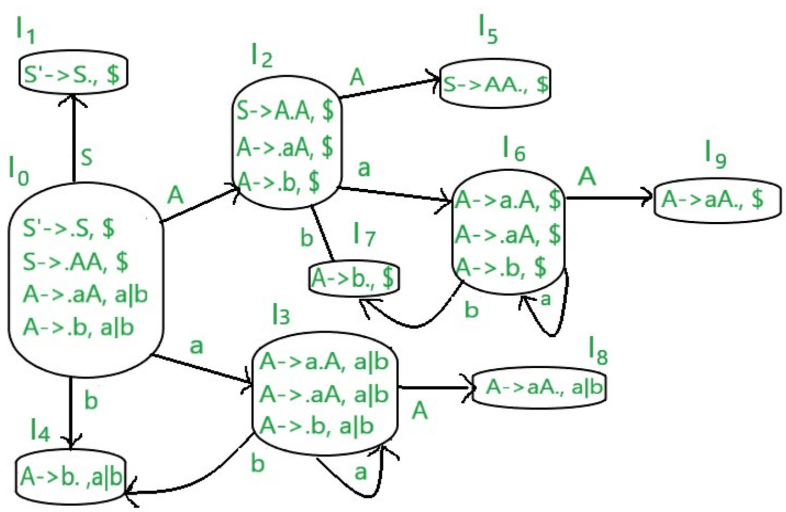
- Writing augmented grammar
- LR(1) collection of items to be found
- Defining 2 functions: goto[list of terminals] and action[list of non-terminals] in the CLR parsing table

Construct a CLR parsing table for the given context-free grammar
S-->AA
A-->aA|b

STEP 1 – Find augmented grammar

- The augmented grammar of the given grammar is:-
  - S'-->.S ,$ [0th production]
  - S-->.AA ,$ [1st production]
  - A-->.aA ,a|b [2nd production]
  - A-->.b ,a|b [3rd production]

STEP 2



STEP 3

	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
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		

STACK	I/P BUFFER	ACTION TABLE	GOTO TABLE	PARSING ACTION
$0	aabb$	[0,a]=S3		Shift
$0a3	abb$	[3,a]=S3		Shift
$0a3a3	bb$	[3,b]=S4		Shift
$0a3a3(b4)	b$	[4,b]=r3	[3,A]=8	Reduce A → b
$0a3(a3A8)	b$	[8,b]=r2	[3,A]=8	Reduce A → aA
$0(a3A8)	b$	[8,b]=r2	[0,A]=2	Reduce A → aA
$0A2	b$	[2,b]=s7		Shift
$0A2(b4)	$	[7,$]=r3	[2,A]=5	Reduce A → b
$0(A2A5)	$	[5,$]=r1	[0,5]=1	Reduce S → AA
$0S1	$	[1,$]=accept		

Canonical LR(1) Collection -- Example

S → AaAb
S → BbBa
A → ε
B → ε

I_0: S' → .S ,$
S → .AaAb ,$
S → .BbBa ,$
A → . ,a
B → . ,b

I_0 --S--> I_1: S' → S. ,$
I_0 --A--> I_2: S → A.aAb ,$ --a--> to I_4
I_0 --B--> I_3: S → B.bBa ,$ --b--> to I_5

I_4: S → Aa.Ab ,$
A → . ,b
I_4 --A--> I_6: S → AaA.b ,$ --a--> I_8: S → AaAb. ,$

I_5: S → Bb.Ba ,$
B → . ,a
I_5 --B--> I_7: S → BbB.a ,$ --b--> I_9: S → BbBa. ,$

11

Canonical LR(1) Collection – Example2

S' → S
1) S → L=R
2) S → R
3) L→ *R
4) L → id
5) R → L

I_0: S' → .S,$
S → .L=R,$
S → .R,$
L → .*R,{$,=}
L → .id, {$,=}
R → .L,$

I_0 --S--> I_1: S' → S.,$
I_0 --*--> I_4
I_0 --L--> I_2: S → L.=R,$
R → L.,$
I_2 --> to I_6
I_0 --R--> I_3: S → R.,$
I_0 --id--> I_5: L → id., {$,=}

I_4: L → *.R, {$,=}
R → .L, {$,=}
L→ .*R, {$,=}
L → .id, {$,=}
--R--> to I_7
--L--> to I_8
--*--> to I_4
--id--> to I_5

I_6: S → L=.R,$
R → .L,$
L → .*R,$
L → .id,$
--R--> to I_9
--L--> to I_{10}
--*--> to I_{11}
--id--> to I_{12}

I_7: L → *R., {$,=}

I_8: R → L., {$,=}

I_9: S → L=R.,$

I_{10}: R → L.,$

I_{11}: L → *.R,$
R → .L,$
L→ .*R,$
L → .id,$
--R--> to I_{13}
--L--> to I_{10}
--*--> to I_{11}
--id--> to I_{12}

I_{12}: L → id.,$

I_{13}: L → *R.,$

I_4 and I_{11}
I_5 and I_{12}
I_7 and I_{13}
I_8 and I_{10}

12

LALR

- Once we make a CLR parsing table, we can easily make a LALR parsing table from it.
- In the step2 diagram, we can see that
- I3 and I6 are similar except their lookaheads.
- I4 and I7 are similar except their lookaheads.
- I8 and I9 are similar except their lookaheads.
- In LALR parsing table construction , we merge these similar states.
- Wherever there is 3 or 6, make it 36(combined form)
- Wherever there is 4 or 7, make it 47(combined form)
- Wherever there is 8 or 9, make it 89(combined form)

LALR PARSING TABLE

	ACTION			GOTO	
	a	b	$	A	S
0	S36	S47		2	1
1			accept		
2	S36	S47		5	
36	S36	S47		89	
47	R3	R3			
5			R1		
36	S36	S47		89	
47			R3		
89	R2	R2			
89			R2		

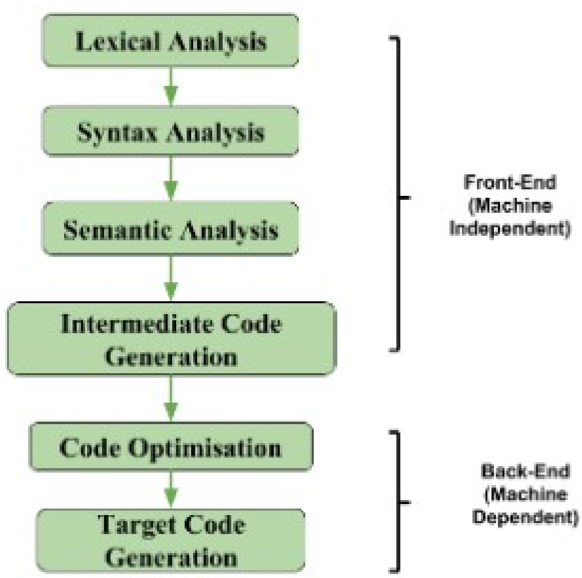
FINAL LALR PARSING TABLE

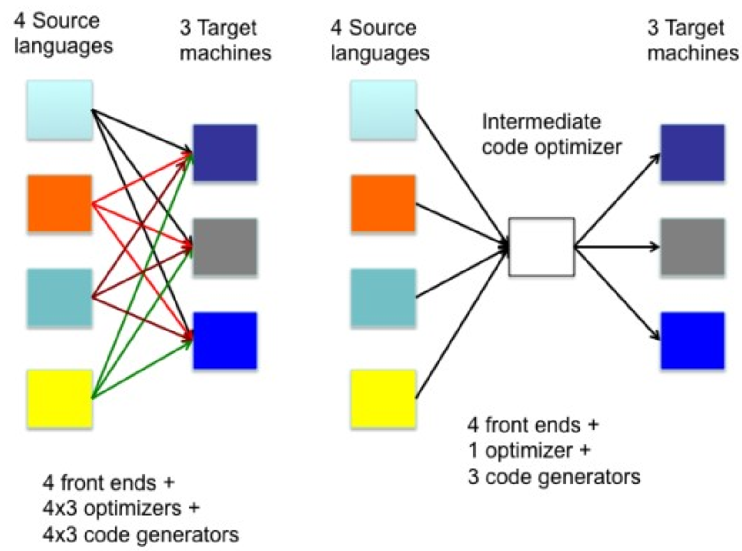
	ACTION			GOTO	
	a	b	$	A	S
0	S36	S47		2	1
1			accept		
2	S36	S47		5	
36	S36	S47		89	
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

- **Intermediate code:**
- Variants of Syntax Trees: Directed Acyclic Graphs for Expressions
- Three address code: Addresses and Instructions- Quadruples - Triples - Indirect Triples.

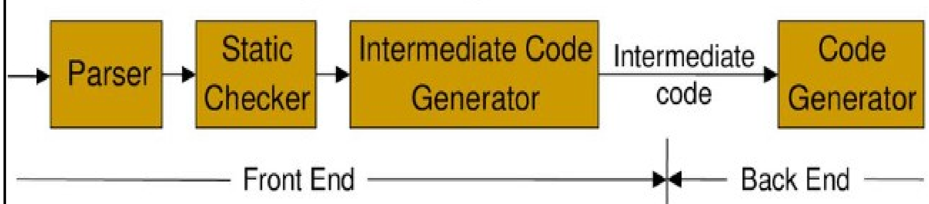
INTERMEDIATE CODE GENERATION

- In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code.
- Ideally, details of the source language are confined to the front end, and details of the target machine to the back end.
- With a suitably defined intermediate representation, a compiler for language *i* and machine *j* can then be built by combining the front end for language *i* with the back end for machine *j.*





Logical structure of front end of a compiler



- Static checking:
  - Type checking: ensures that operators are applied to compatible operands
  - Any syntactic checks that remain after parsing

- In the process of translating a program in a given source language into code for a given target machine, compiler may construct a sequence of intermediate representations.
- High level representations are close to the source language and low-level representations are close to the target machine.
- Syntax trees are high level intermediate representations.
- These are well suited for static type checking.

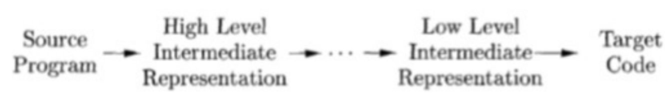


Figure 6.2: A compiler might use a sequence of intermediate representations

- A low level representations is suitable for machine dependent tasks such as register allocation and instruction selection.
- Three address code can range from high to low level depending on choice of program.
- The choice of intermediate representations varies from compiler to compiler.

An intermediate representation may either be:

- actual language or
- it may consist of internal data structures that are shared by phases of the compiler.

intermediate code

The following are commonly used intermediate code representation：

- Syntax tree
- Postfix Notation
- Three-Address Code

4

VARIANTS IN SYNTAX TREE

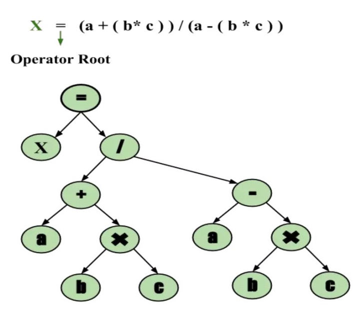
- Nodes in a syntax tree represent constructs in the source program;
- The children of a node represent the meaningful components of a construct.
- A directed acyclic graph (hereafter called a *DAG)* for an expression identifies the *common sub expressions* (subexpressions that occur more than once) of the expression.

1. Directed Acyclic Graphs for Expressions

- Like the syntax tree for an expression,
  - a DAG has leaves corresponding to atomic operands and interior codes corresponding to operators.
- The difference is that a node *N* in a DAG has more than one parent if *N* represents a common sub expression;
- In a syntax tree, the tree for the common sub expression would be replicated as many times as the sub expression appears in the original expression.
- Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

Syntax tree

Example –
x = (a + b * c) / (a – b * c)



6

a + a * (b - c) + (b - c) * d

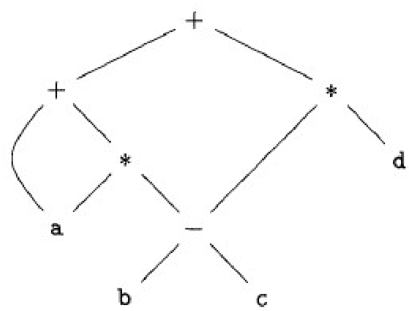
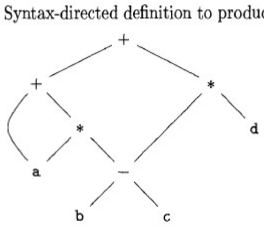


Figure 6.3: Dag for the expression a + a * (b - c) + (b - c) * d

Syntax-directed definition to produce syntax trees or DAG's

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 6.4: Syntax-directed definition to produce syntax trees or DAG'



6.3: Dag for the expression a + a * (b - c) + (b - c) * d

1) $p_1 = Leaf(\textbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\textbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\textbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\textbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\textbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\textbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\textbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

Figure 6.5: Steps for constructing the DAG of Fig. 6.3

Postfix Notation

- The ordinary (infix) way of writing the sum of a and b is with operator in the middle : a + b
- The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation the operator follows the operand.

7

Postfix Notation

Example – The postfix representation of the expression (a – b) * (c + d) + (a – b) is

ab – cd + ab -+*.

8

Three-Address Code

- In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted.
- Example: A source-language expression x+y*z

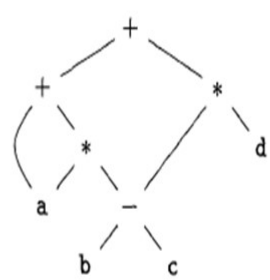
  might be translated into the sequence of three-address instructions below where tl and tz are compiler-generated temporary names.

:

$$t_1 = y * z$$
$$t_2 = x + t_1$$

- Where t1 and t2 are compiler generated temporary names.

Example 6.4: Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.



(a) DAG

$$t_1 = b - c$$
$$t_2 = a * t_1$$
$$t_3 = a + t_2$$
$$t_4 = t_1 * d$$
$$t_5 = t_3 + t_4$$

(b) Three-address code

Addresses and Instructions

- An address can be one of the following:
  - **A name** : For convenience, allow source-program names to appear as addresses in three-address code. In an implementation, ***a source name is replaced by a pointer to its symbol-table entry***, where all information about the name is kept.
  - **A constant** : In practice, a compiler must deal with many different types of constants and variables.
  - ***A* compiler-generated temporary** . It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

list of the common three-address instruction forms:

- Assignment instructions of the form x = y op z, op is binary operator
- x = op y, where op is a unary operation.
- Copy instructions of the form x = y,
- An unconditional jump goto L.
- Conditional jumps of the form if x goto L and if False x goto L.
- Conditional jumps such as if x relop y goto L, which apply a relational operator (<, ==, >=, etc.) to x and y, and execute the instruction with label L next if x stands in relation relop to y. If not, the three-address instruction following if x relop y goto L is executed next, in sequence.

- Procedure calls and returns are implemented using the following instructions:
  - **param x** for parameters; **call p,n** and **y = call p,n** for procedure and function calls, respectively; and **return y** , is optional.
  - Example: a call of the procedure p(xl,x2,. . . ,x,).

```
param x1
param x2
...
param xn
call p, n
```

•Indexed copy instructions of the form x = y[i] and x[i]= y.

•Address and pointer assignments of the form

x = & y, x = * y, and * x = y.

Example

- Consider the statement

  do { i = i+l;} while (a[i] < v) ;

Two ways of assigning labels to three-address statements

```
L:   t1 = i + 1              100:  t1 = i + 1
     i = t1                  101:  i = t1
     t2 = i * 8              102:  t2 = i * 8
     t3 = a [ t2 ]           103:  t3 = a [ t2 ]
     if t3 < v goto L        104:  if t3 < v goto 100
```

(a) Symbolic labels. (b) Position numbers.

Figure 6.9: Two ways of assigning labels to three-address statements

The multiplication i * 8 is appropriate for an array of elements that each take 8 units of space.

Example:

- Then, the assignment

  N=f(a[i]);

- might translate into the following three-address code:

  1) t1 = i * 4 // integer take 4 bytes

  2) t2 = a [t1]

  3) param t2

  4) **t3** = **call f,** 1 // 1 for 1 parameter

  5) n = t3

Data structure of three address code

- Three address code instructions can be implemented as objects or as records with fields for the operator and the operands. Three such representations are called
  - ***Quadruples*** A quadruple (or just "quad') has four fields, which we call op, arg,, arg2, and result
  - ***Triples:*** A triple has only three fields, which we call op, arg1, and arg2. the DAG and triple representations of expressions are equivalent
  - ***Indirect Triples:*** consist of a listing of pointers to triples, rather than a listing of triples themselves.
- The benefit of **Quadruples** over **Triples** can be seen in an optimizing compiler, where instructions are often moved around.
- With ***quadruples***, if we move an instruction that computes a temporary ***t***, then the instructions that use ***t*** require no change. With ***triples***, the result of an operation is referred to by its position, so moving an instruction may require to change all references to that result. ***This problem does not occur with indirect triples*** .

Three-address code for the assignment `a = b*-c+b*-c;`

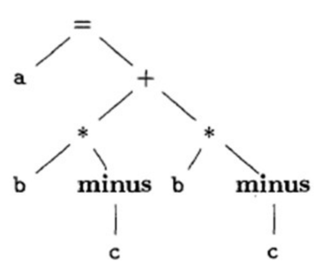
```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

	op	*arg*$_1$	*arg*$_2$	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		. . .		

(b) Quadruples

Figure 6.10: Three-address code and its quadruple representation



(a) Syntax tree

	op	*arg*$_1$	*arg*$_2$
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		. . .	

(b) Triples

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	. . .

	op	*arg*$_1$	*arg*$_2$
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		. . .	

Figure 6.12: Indirect triples representation of three-address code