

## 1. Compare the LR parsers.

### 2.6 Introduction to LR Parsing:

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where

- L stands for left-to-right scanning of the input stream;
- R stands for the construction of right-most derivation in reverse, and
- k denotes the number of lookahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

- **SLR() – Simple LR Parser:**
  - Works on smallest class of grammar
  - Few number of states, hence very small table
  - Simple and fast construction
- **CLR() – Canonical LR Parser:**
  - Works on complete set of LR(1) Grammar
  - Generates large table and large number of states
  - Slow construction
- **LALR() – Look-Ahead LR Parser:**
  - Works on intermediate size of grammar
  - Number of states are same as in SLR(1)

#### 2.6.1 Why LR Parsers?

LR parsing is attractive for a variety of reasons:

- LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.
- The LR-parsing method is the most general non backtracking shift-reduce parsing method known.

## 2. Explain the forms of 3-address codes with an example

Three-address code is a sequence of statements of the general form

**$x = y \text{ op } z$**

Where x, y and z are names, constants, or compiler-generated temporaries; *op* stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on Boolean valued data. Thus a source language expression like  $x + y * z$  might be translated into a sequence

$t1 = y * z$

$t2 = x + t1$

Where t1 and t2 are compiler-generated temporary names.

***The reason for the term “three-address code” is that each statement usually contains three addresses, two for the operands and one for the result.***

Three such representations are:

- Quadruples
- Triples
- Indirect triples

#### **Quadruples:**

Three-Address Code does not specify the internal representation of 3-Address instructions. This limitation is overcome by Quadruple.

- A quadruple is a record structure with four fields, which are, ***op***, ***arg1***, ***arg2*** and ***result***.
- The *op* field contains an internal code for the operator. The three-address statement  $x = y \text{ op } z$  is represented by placing y in *arg1*, z in *arg2* and x in *result*.
- The contents of field's *arg1*, *arg2* and *result* are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

**Example:  $a = b * c + b * c$  represent the expression using Quadruple, Triples and Indirect Triples.**

	<i>op</i>	<i>arg1</i>	<i>arg2</i>	<i>result</i>
(0)	uminus	c		t <sub>1</sub>
(1)	*	b	t <sub>1</sub>	t <sub>2</sub>
(2)	uminus	c		t <sub>3</sub>
(3)	*	b	t <sub>3</sub>	t <sub>4</sub>
(4)	+	t <sub>2</sub>	t <sub>4</sub>	t <sub>5</sub>
(5)	=	t <sub>5</sub>		a

(a) Quadruples

**Triples:**

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- If we do so, three-address statements can be represented by records with only three fields: ***op*, *arg1* and *arg2***.
- The fields *arg1* and *arg2*, for the arguments of *op*, are either pointers to the symbol table or pointers into the triple structure ( for temporary values ).
- Since three fields are used, this intermediate code format is known as *triples*.

	<i>op</i>	<i>arg1</i>	<i>arg2</i>
(0)	minus	c	
(1)	*	b	(0)
(2)	minus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

**\*\*Note:** 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 instruction that uses t require no change. With Triples the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result. This problem does not occur in Indirect Triples.

**Indirect Triples:**

- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.
- For example, let us use an array statement to list pointers to triples in the desired order.



	<i>statement</i>		<i>op</i>	<i>arg1</i>	<i>arg2</i>
(0)	(14)	(14)	minus	c	
(1)	(15)	(15)	*	b	(14)
(2)	(16)	(16)	minus	c	
(3)	(17)	(17)	*	b	(16)
(4)	(18)	(18)	+	(15)	(17)
(5)	(19)	(19)	assign	a	(18)

Indirect triples representation of three-address statements

3.Translate the expression  $a=b* -c + b* -c$  into quadruples. (Refer q-2)

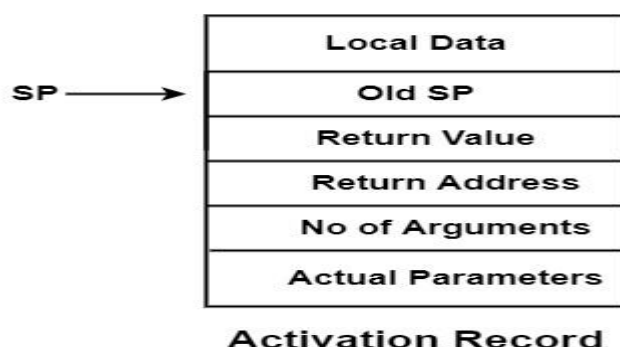
4.Construct abstract syntax tree for the  $x=(a-b)*(c+d)$  using mknnode ,mkleaf functions

5.Explain synthesized and inherited attributes with an example.

S.NO	Synthesized Attributes	Inherited Attributes
1.	An attribute is said to be Synthesized attribute if its parse tree node value is determined by the attribute value at child nodes.	An attribute is said to be Inherited attribute if its parse tree node value is determined by the attribute value at parent and/or siblings node.
2.	The production must have non-terminal as its head.	The production must have non-terminal as a symbol in its body.
3.	A synthesized attribute at node n is defined only in terms of attribute values at the children of n itself.	A Inherited attribute at node n is defined only in terms of attribute values of n's parent, n itself, and n's siblings.
4.	It can be evaluated during a single bottom-up traversal of parse tree.	It can be evaluated during a single top-down and sideways traversal of parse tree.
5.	Synthesized attributes can be contained by both the terminals or non-terminals.	Inherited attributes can't be contained by both, It is only contained by non-terminals.
6.	Synthesized attribute is used by both S-attributed SDT and L-attributed SDT.	Inherited attribute is used by only L-attributed SDT.
7.	<p>EX:-</p> $E.val \rightarrow F.val$ 	<p>EX:-</p> $E.val = F.val$ 

6.Discuss about the stack allocation technique in detail.

Almost all compilers for languages that use procedures, functions, or methods as units of user-defined actions manage at least part of their run-time memory as a stack. Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack. This arrangement not only allows space to be shared by procedure calls whose durations do not overlap in time, but it allows us to compile code for a procedure in such a way that the relative addresses of its nonlocal variables are always the same, regardless of the sequence of procedure calls.



## 7.Describe Machine Independent Optimization techniques.

Ans.

<https://www.javatpoint.com/machine-independent-optimization#:~:text=Machine%20independent%20optimization%20attempts%20to,location%20or%20any%20CPU%20registers.>

## 8.Explain about Loop Optimization with suitable examples.

Ans. Loops are a very important place for optimizations, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.

There are three techniques:

1. Code Motion
2. Elimination of induction variables
3. Strength reduction

### 1. Code Motion:

Code motion reduces the number of instructions in a loop by moving instructions outside a loop. It moves loop invariant computations i.e, those instructions or expressions that result in the same value independent of the number of times a loop is executed and places them at the beginning of the loop. The relocated expressions become an entry for the loop.

Example:

```
While(X!=n-2)
```

```
{
```

```
  X=X+2;
```

```
}
```

here the expression  $n-2$  is a loop invariant computation i.e, the value evaluated by this expression is independent of the number of times the while loop is executed. In other words the value of  $n$  remains unchanged. The code relocation places the expressions  $n-2$  before the while loop begins as shown below.

```
M=n-2;
```

```
While(X!=M)
```

```
{
```

```
  X=X+2;
```

```
}
```

## 2. Elimination of Induction Variables:

An induction variable is a loop control variable or any other variable that depends on the induction variable in some fixed way. It can also be defined as variable which is incremented or decremented by a fixed number in a loop each time the loop is executed. If there are two or more induction variables in a loop then by the induction variable elimination process all can be eliminated except one.

Example:

```
void fun(void)
{
    int i,j,k;
    for(i=0,j=0,k=0;i<10;i++)
        a[j++]=b[k++];
    return;
}
```

In the above code there are three induction variables i,j and k which take on the values 1,2,3...10 each time through the beginning of the loop. Suppose that the values of the variables j and k are not used after the end of the loop then we can eliminate them from the function fun() by replacing them by variable i. After induction variable elimination, the above code becomes

```
void fun(void)
{
    int i,j,k;
    for(i=0;i<10;i++)
        a[i]=b[i];
    return;
}
```

Thus induction variable elimination reduces the code and improves the run time performance.

## 3. Strength Reduction:

Strength reduction is an optimization technique in which expensive operations are replaced with equivalent but less expensive operations. The classic example of strength reduction converts "strong" multiplications inside a loop into "weaker" additions – something that frequently occurs in array addressing. By doing this the execution speed can be increased. For example, consider the code

```
for(i=1;i<=5;i++)
{
    x=4*i;
}
```



The instruction  $x=4*i$  in the loop can be replaced by equivalent additions instruction as

$x=x+4$ ;

## 9. Discuss various object code forms.

<https://www.geeksforgeeks.org/introduction-of-object-code-in-compiler-design/>

## 10. Explain about peephole Optimization technique.

A simple but effective technique for locally improving the target code is *peephole optimization*, which is done by examining a sliding window of target instructions (called the *peephole*) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible. Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.

The following peephole optimization techniques may be applied to improve the performance of the target program:

- Redundant-instruction elimination
- Elimination of Unreachable Code.
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

### i) Eliminating Redundant Loads and Stores

If we see the instruction sequence

```
LD a, R0
ST R0, a
```

in a target program, we can delete the store instruction. Note that if the store instruction has a label, we could not be sure that the first instruction is always executed before the second, so we could not remove the store instruction. The two instructions have to be in the same basic block for this transformation to be safe.

### ii) Eliminating Unreachable Code

Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed.

```
Goto L2
Print Debug Information
L2:
```

In above example Print Statement can be eliminated.

### iii) Flow-of-Control Optimizations

Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the sequence

```
goto L1
```

```
...
```

```
L1: goto L2
```

by the sequence

```
goto L2
```

```
...
```

```
L1: goto L2
```

If there are now no jumps to L1, then it may be possible to eliminate the statement **L1: goto L2** provided it is preceded by an unconditional jump.

Similarly, the sequence

```
If a < b goto L1
```

```
-----
```

```
L1: goto L2
```

can be replaced by the sequence

```
If a < b goto L2
```

```
-----
```

```
L1: goto L2
```

### iv) Algebraic Simplification and Reduction in Strength

The algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as  $X = X + 0$  and  $X = X * 1$  in the peephole. Similarly, reduction-in-strength transformations can be applied in the peephole to replace expensive operations by equivalent cheaper ones. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example,  $x^2$  can be implemented by using  $x*x$ . Similarly  $x*x$  can be implemented by using  $x+x$ .

#### **v) Use of Machine Idioms**

The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of the modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like  $x = x + 1$ .