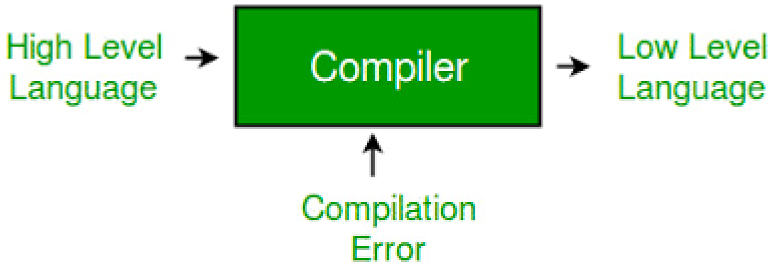
Compiler Design

What is Compiler ?

A compiler is a computer program which helps you transform source code written in a high-level language into low-level language.



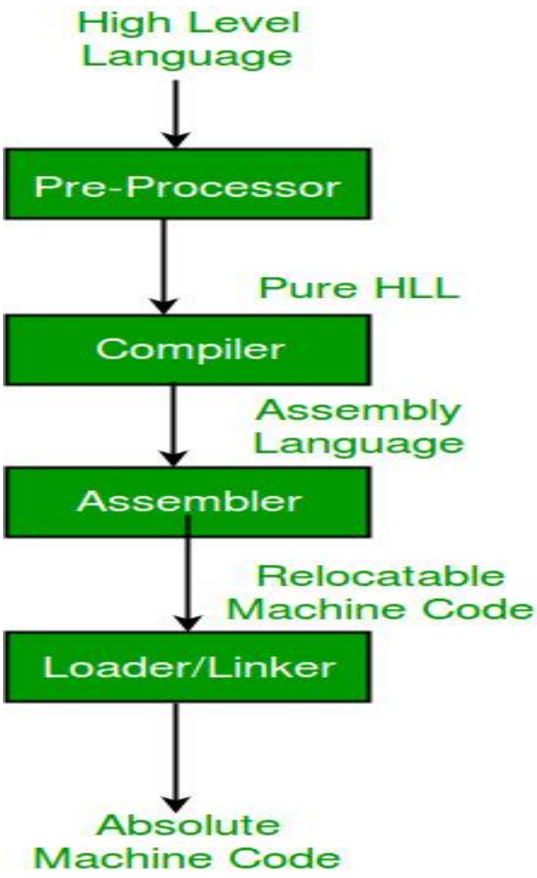
Steps for Language processing systems

Before knowing about the concept of compilers, you first need to understand a few other tools which work with compilers.

Here we are drawing the language process system by using the compiler

Pre-processor

The pre-processor is considered as a part of the Compiler. It is a tool which produces input for Compiler. It deals with macro processing, augmentation, language extension, etc.



Compiler

A compiler is a computer program which helps you transform source code written in a high-level language into low-level machine language.

Assembler:

It translates assembly language code into machine understandable language. The output result of assembler is known as an object file which is a combination of machine instruction as well as the data required to store these instructions in memory.

• Linker:

The linker helps you to link and merge various object files to create an executable file. All these files might have been compiled with separate assemblers.

Loader:

The loader is a part of the OS, which performs the tasks of loading executable files into memory and run them. It also calculates the size of a program which creates additional memory space.

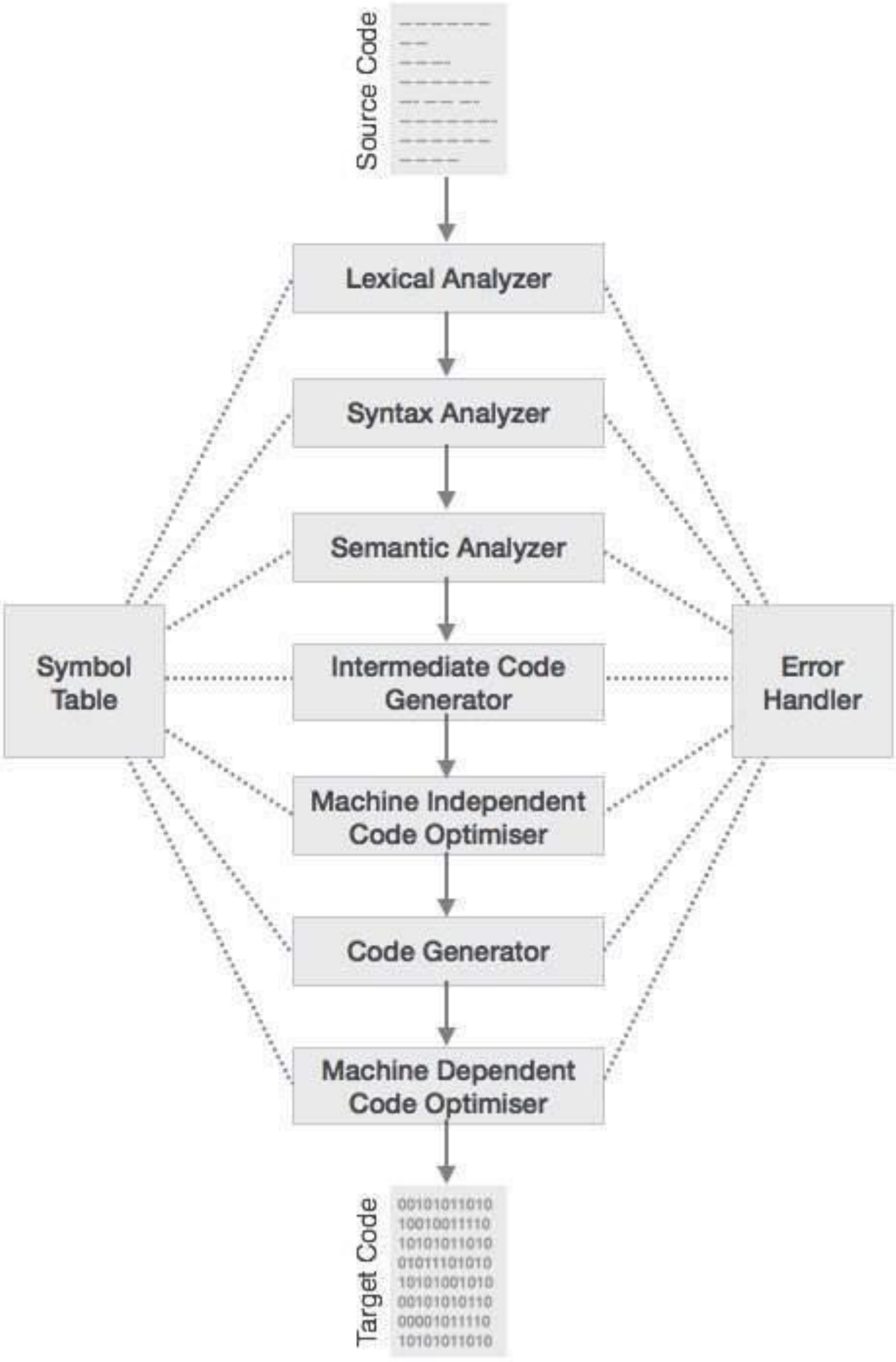
Machine code

a computer programming language consisting of binary or hexadecimal instructions which a computer can respond to directly.

Phases of Compiler

The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. Each phase takes input from its previous stage.

There are the various phases of compiler:



Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyser represents these lexemes in the form of tokens.

<token-name, attribute-value>

Example:

Sum=old sum Rate*50

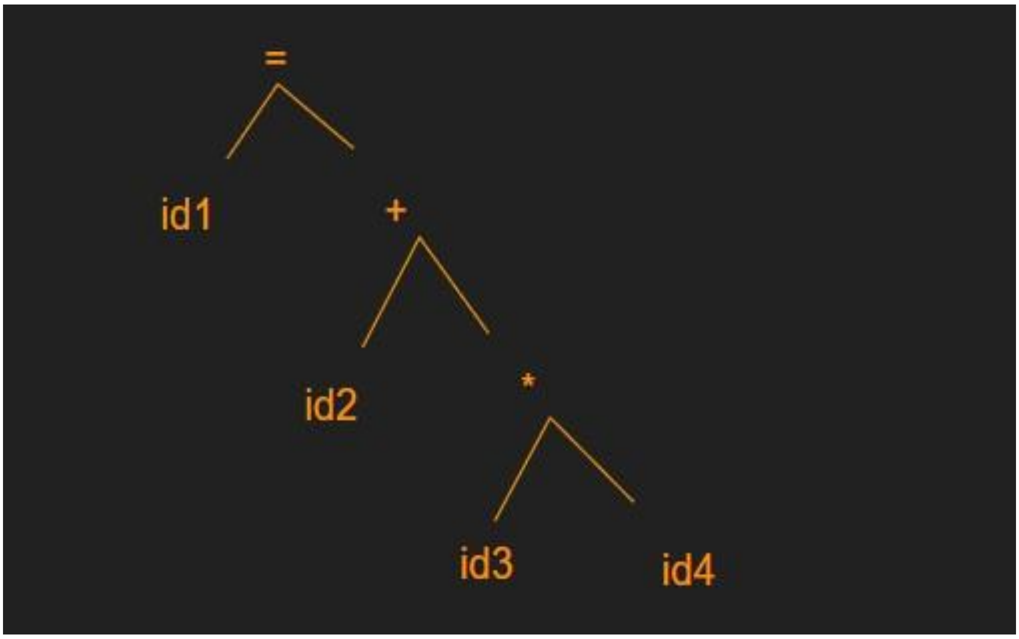
id1=id2+id3*50

Syntax Analysis

The next phase is called the syntax analysis or parsing.

the parser checks if the expression made by the tokens is syntactically correct. id1=id2+id3*50

Example



Semantic Analysis

Semantic analysis is the third phase of compilation process. It checks whether the parse tree follows the rules of language. Semantic analyser keeps track of identifiers, their types and expressions. The output of semantic analysis phase is the annotated tree syntax.

• Intermediate Code Generation

compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language.

Example

id1=id2+id3*50

temp1 = inttoreal(50)

temp2 = id3*temp1

temp3 = id2+temp2

id1 = temp3

Code Optimization

It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

```
Example   id1=id2+id3*50
               temp1=id3*50.0
                id1=id2+temp1
```

• Code Generation

Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language.   Example: id1=id2+id3*50

```
MOV R1, Id3
MUL   R1#50.0
MOV R2, id2
```

```
ADD   R1, R2
MOV Id1, R1
```

Syntax directed translation

In syntax directed translation, along with the grammar we associate some informal notations and these notations are called as semantic rules.

So, we can say that

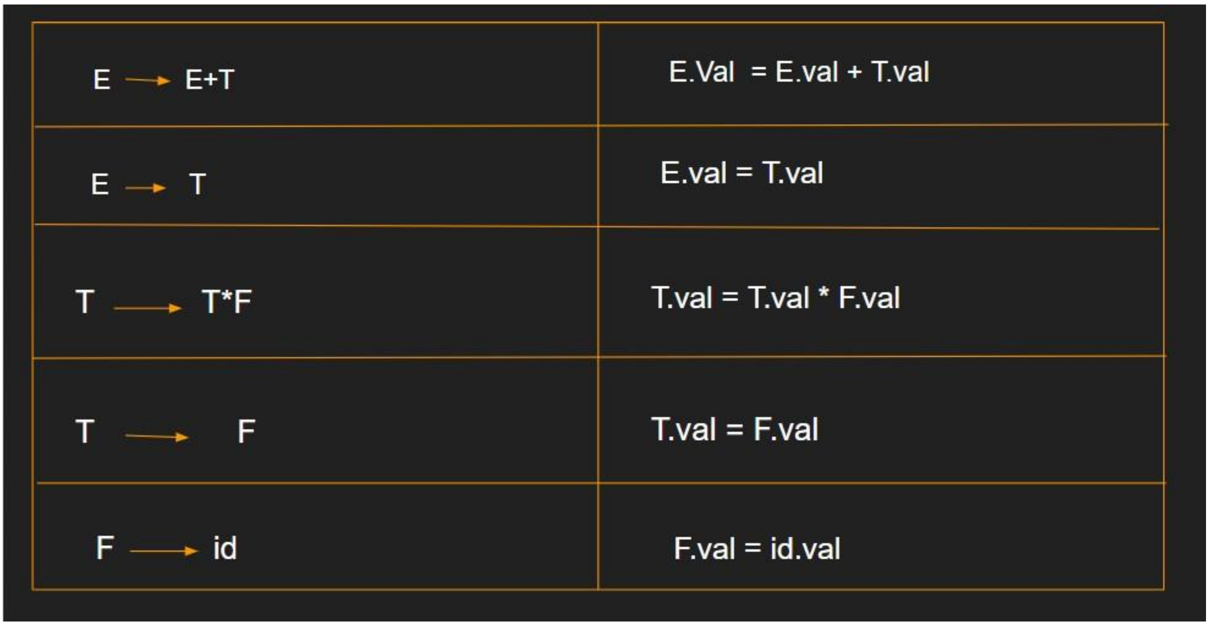
1. Grammar + semantic rule = SDT (syntax directed translation)

In syntax directed translation, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the attribute. The value of these attributes is evaluated by the semantic rules associated with the production rule.

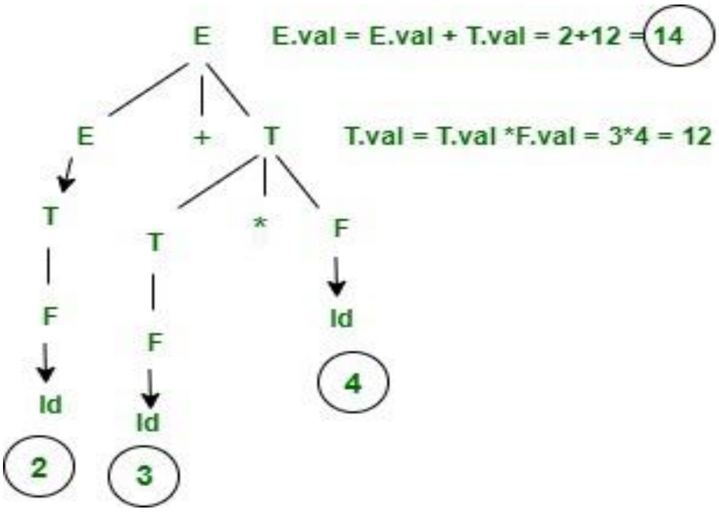
In the semantic rule, attribute is VAL and an attribute may hold anything like a string, a number, a memory location and a complex record

- In Syntax directed translation, whenever a construct encounters in the programming language then it is translated according to the semantic rules define in that particular programming language.

Productions **Semantic rule**



E → E+T	E.Val = E.val + T.val
E → T	E.val = T.val
T → T*F	T.val = T.val * F.val
T → F	T.val = F.val
F → id	F.val = id.val

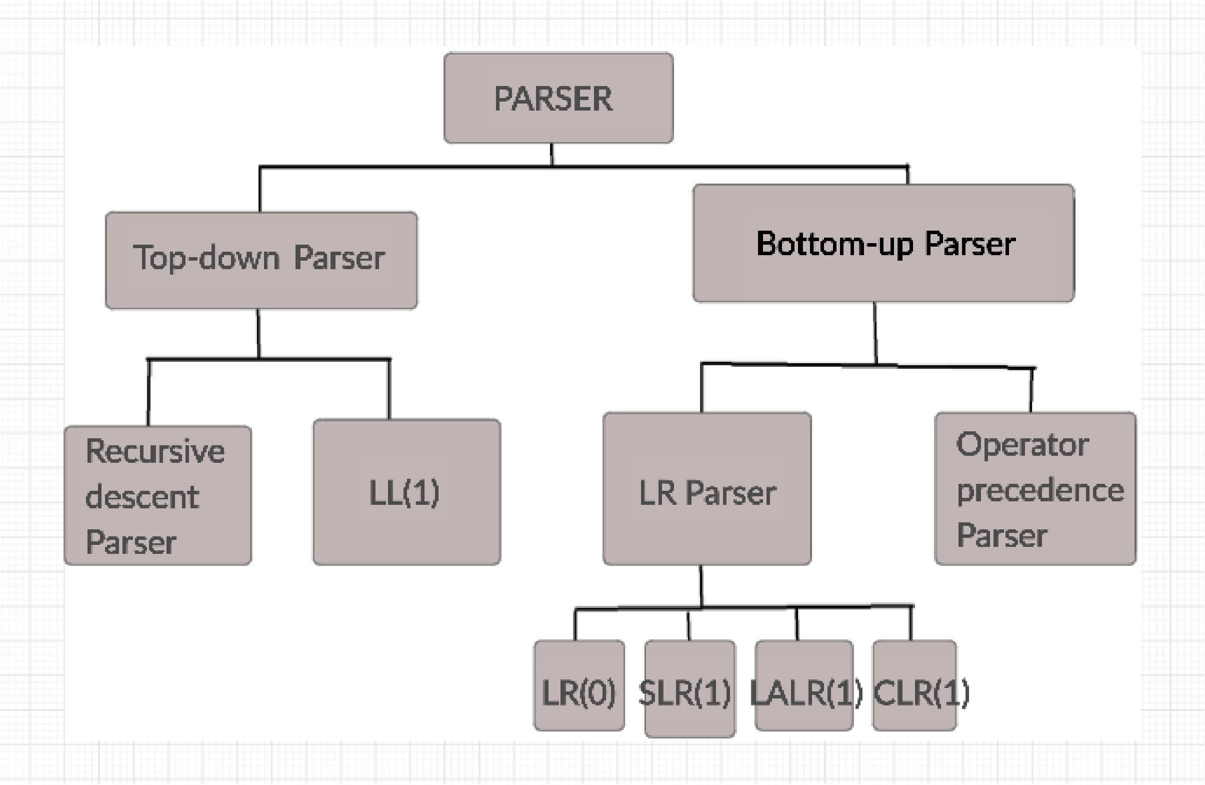

E
E.val = E.val + T.val = 2+12 = 14
E
+
T
T.val = T.val *F.val = 3*4 = 12
T
T
*
F
F
F
Id
Id
Id
2
3
4

Parser

A parser takes input in the form of sequence of tokens and produces output in the form of parse tree.

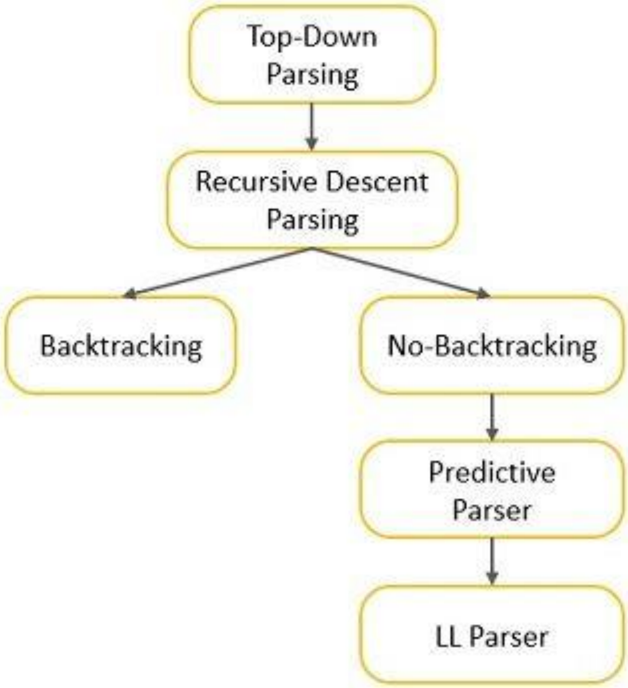
Parsing is of two types: 1.top down parsing

2.bottom up parsing.



Top-down parsing

- The process of constructing the parse tree which starts from the root and goes down to the leaf is Top-Down Parsing.
- Top-Down Parsers constructs from the Grammar which is free from ambiguity and left recursion. Top-Down Parsers uses leftmost derivation to construct a parse tree.



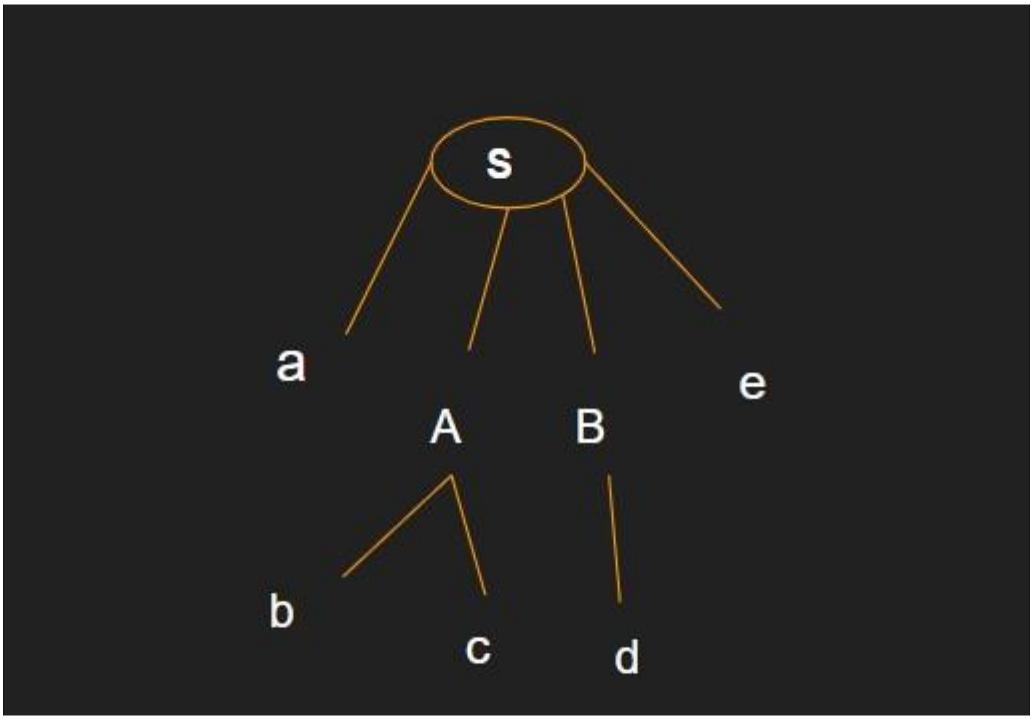
Example

S→ aABe

A→ bc

B → d

Input string is **abcde.**



Recursive descent parser

- Recursive Descent Parser **uses the technique of Top-Down Parsing without backtracking**.
- It can be defined as a Parser that uses the various recursive procedure to process the input string with no backtracking. It can be simply performed using a Recursive language.

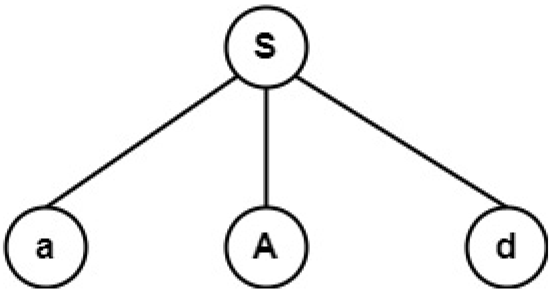
Backtracking

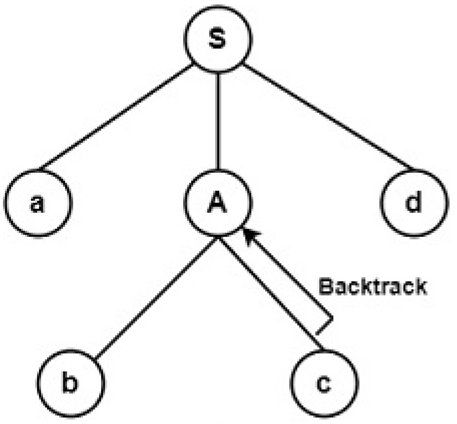
Example1 − Consider the Grammar

S → a A d

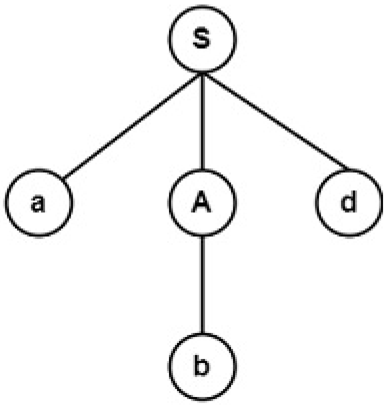
A → b c | b

i/p=abd





(b)



(c)

Predictive Parser:

In this, we will cover the overview of Predictive Parser and mainly focus on the role of Predictive Parser. And will also cover the algorithm for the implementation of the Predictive parser algorithm and finally will discuss an example by implementing the algorithm for precedence parsing.

Example :

Given grammar

```
E->E+T|T
T->T*F|F
F->(E)|id
```

After removing left recursion, left factoring

Computation of FIRST & FOLLOW

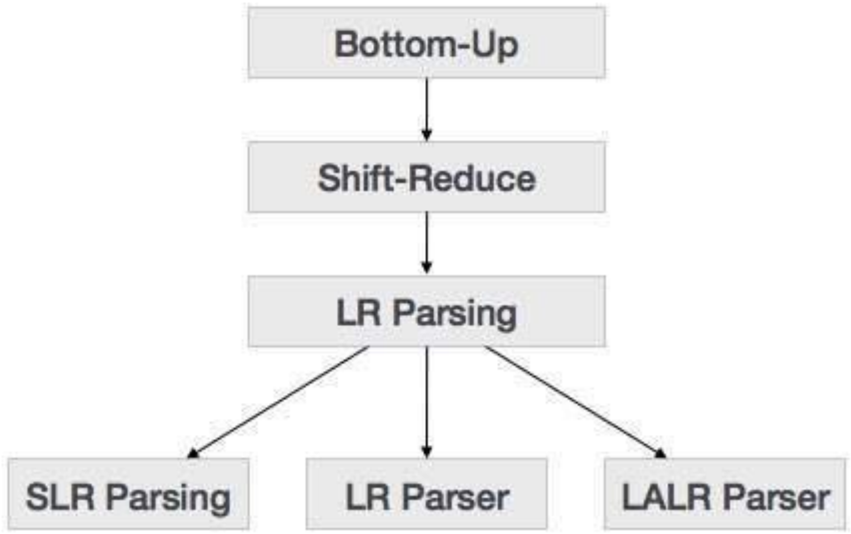
```
First(E)=First(T)=FIRST(F)={(,id}

First(
```

Bottom-Up Parsing

Bottom-up parsing parses the stream of tokens from the lexical analyzer. And after parsing the input string it generates a parse tree.

The bottom-up parser builds a parse tree from the leaf nodes and proceeds towards the root node of the tree. In this section, we will be discussing bottom-up parsing along with its types.



Example:

S aABe

A Abc/b

B d

Input string “ abbcde ”

- abbcde

  aAbcde(A b)

  aAde(A Abc)

  aABe(B d)

  S(S aABe)

Example

E → E+T|T

T → T*F|F

F → (E)|id   i/p=id*id

id*id

F*id   (F   id)

T*id   (T   F)

T*F   (F   id)

T   (E   T)

E

Example

E → E+T|T

T → T*F|F

F → (E)|id   i/p=id*id

id*id

F*id   (F   id)

T*id   (T   F)

T*F   (F   id)

T   (E   T)

E

Shift reduce parser:

Shift Reduce Parser is a type of Bottom-Up Parser. It generates the Parse Tree from Leaves to the Root. In Shift Reduce Parser, the input string will be reduced to the starting symbol. This reduction can be produced by handling the rightmost derivation in reverse, i.e., from starting symbol to the input string.

Shift Reduce Parser requires two Data Structures

- Input Buffer
- Stack

There are the various steps of Shift Reduce Parsing which are as follows −

There are the various steps of Shift Reduce Parsing which are as follows −

- It uses a stack and an input buffer.
- Insert $ at the bottom of the stack and the right end of the input string in Input Buffer.

Stack
$
Input String
w
$

- Shift − Parser shifts zero or more input symbols onto the stack until the handle is on top of the stack.
- Reduce − Parser reduce or replace the handle on top of the stack to the left side of production, i.e., R.H.S. of production is popped, and L.H.S is pushed.

- Accept − Step 3 and Step 4 will be repeated until it has detected an error or until the stack includes start symbol (S) and input Buffer is empty, i.e., it contains $.

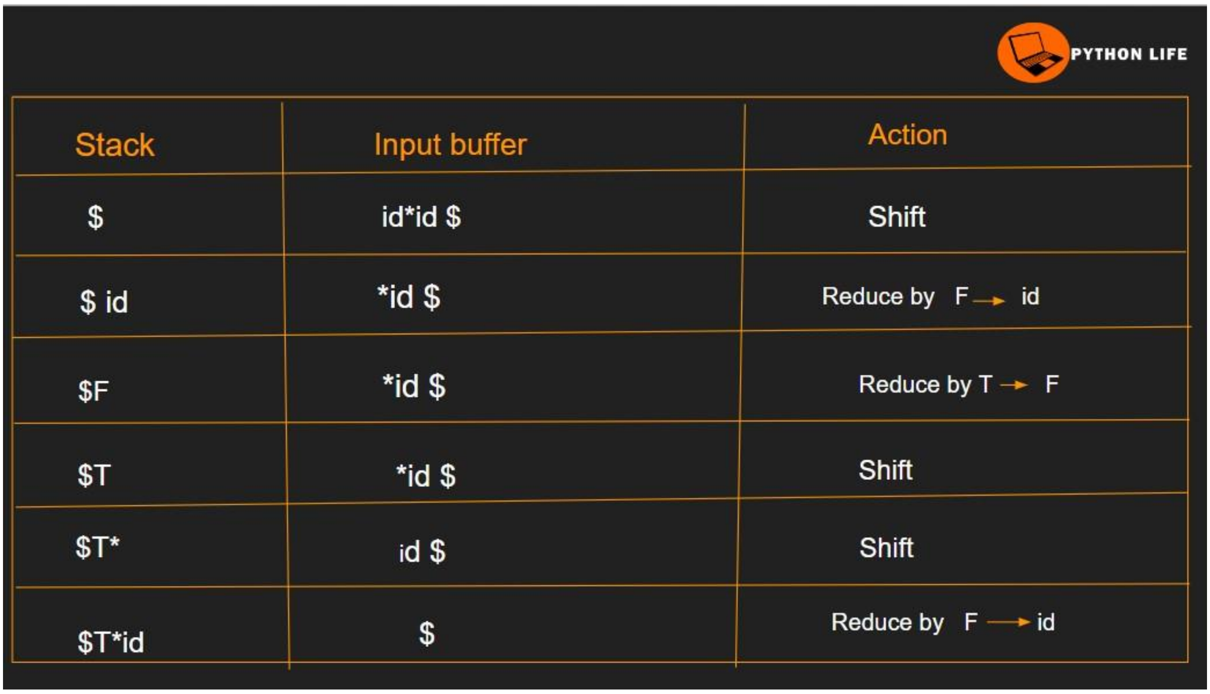
Example

Given grammar

E → E+T|T

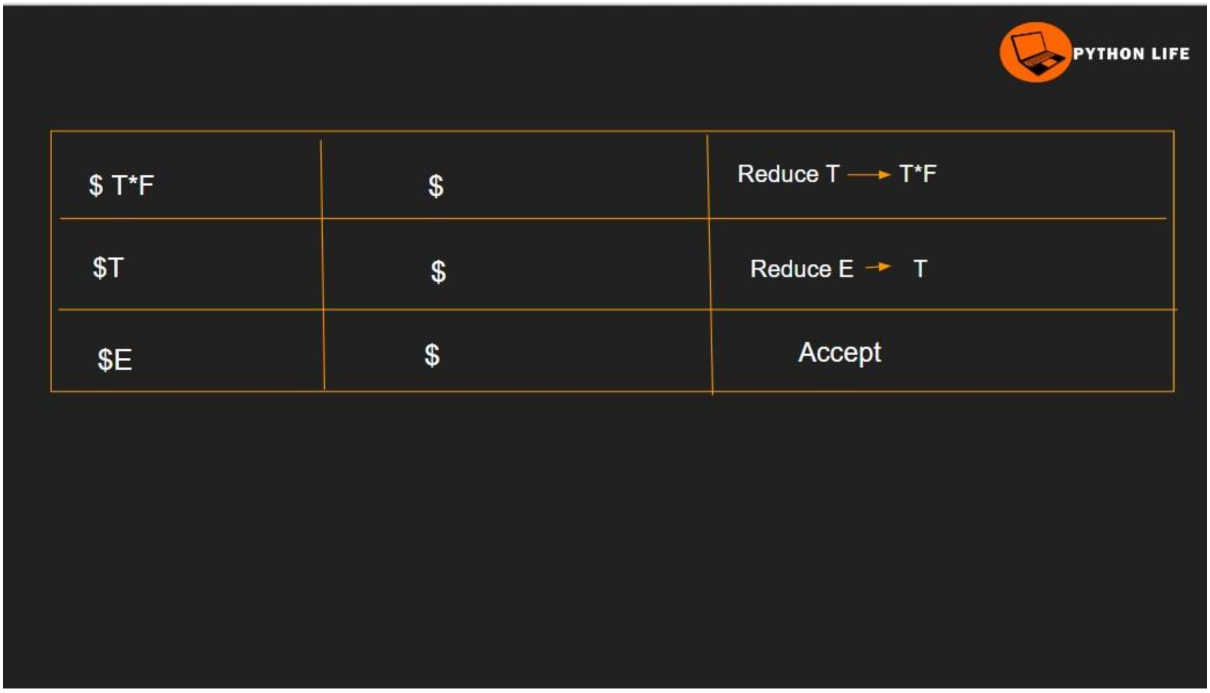
T → T*F|F

F → (E)|id i/p=id*id



PYTHON LIFE

Stack	Input buffer	Action
$	id*id $	Shift
$ id	*id $	Reduce by F → id
$F	*id $	Reduce by T → F
$T	*id $	Shift
$T*	id $	Shift
$T*id	$	Reduce by F → id


PYTHON LIFE

$ T*F	$	Reduce T → T*F
$T	$	Reduce E → T
$E	$	Accept

Canonical Collection of LR(0) items

Example

Given grammar:

1. S → AA
2. A → aA | b

Add Augment Production and insert '•' symbol at the first position for every production in G

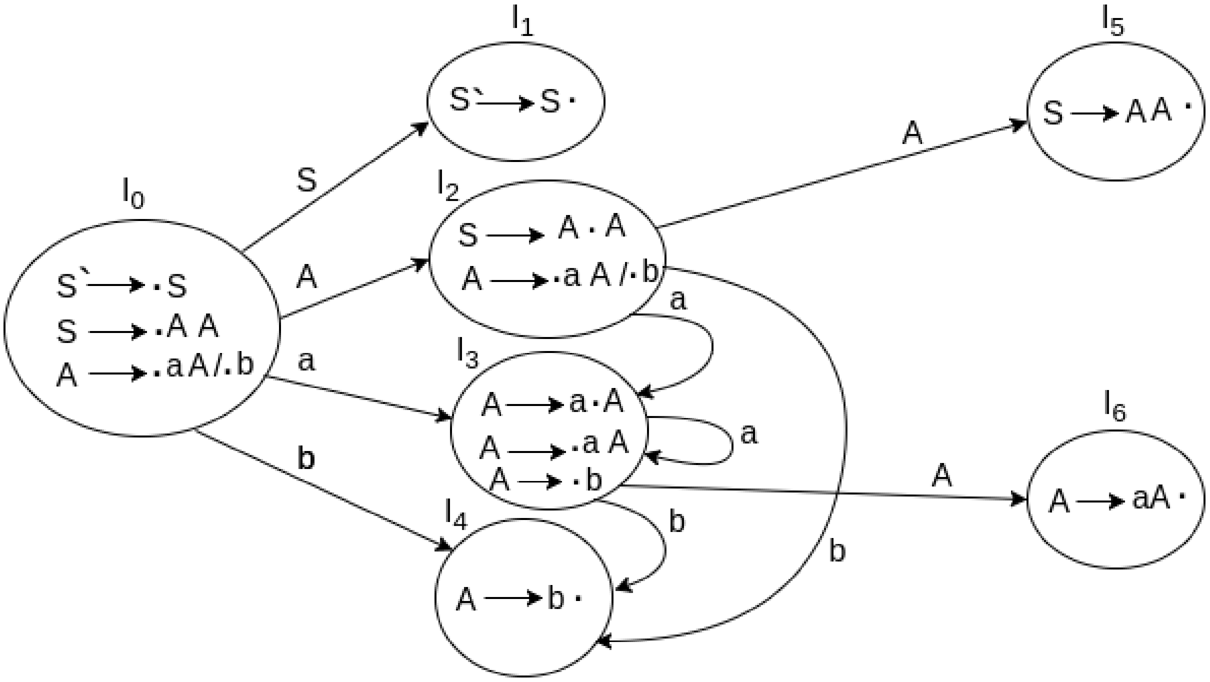
S` → •S

S → •AA

A → •aA

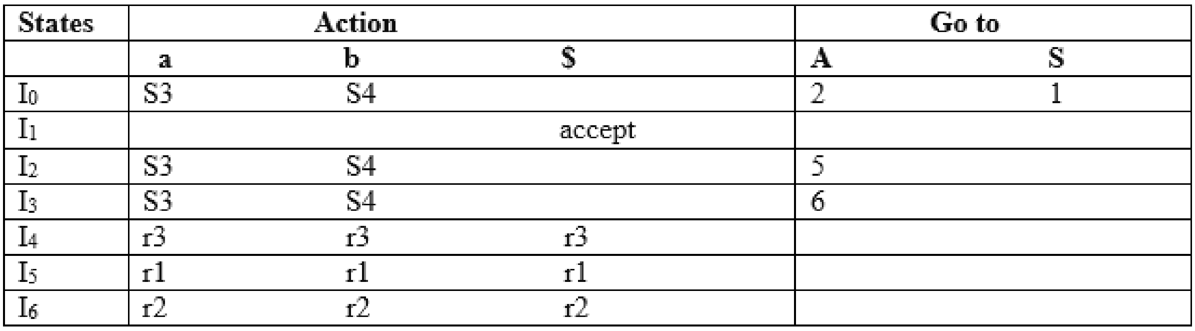
A → •b

Drawing DFA:



LR(0) Table

- If a state is going to some other state on a terminal then it correspond to a shift move.
- If a state is going to some other state on a variable then it correspond to go to move.
- If a state contain the final item in the particular row then write the reduce node completely.



States	Action			Go to	
	a	b	$	A	S
I_0	S3	S4		2	1
I_1			accept		
I_2	S3	S4		5	
I_3	S3	S4		6	
I_4	r3	r3	r3		
I_5	r1	r1	r1		
I_6	r2	r2	r2		

SLR(1) Parser

Steps for constructing the SLR parsing table :

1. Writing augmented grammar
2. LR(0) collection of items to be found
3. Find FOLLOW of LHS of production
4. Defining 2 functions:goto and action in the parsing table

Example

S–>AA

A–>aA|b

Solution:

STEP1 – Find augmented grammar

The augmented grammar of the given grammar is:-

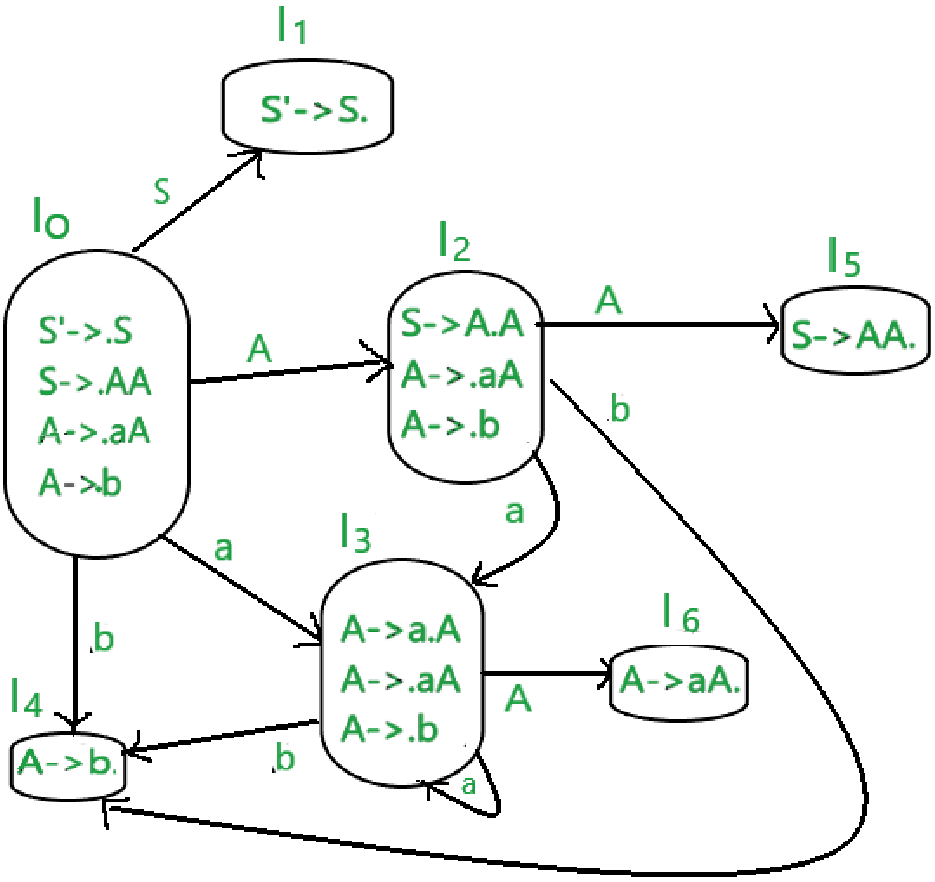
S’–>.S   [0th production]

S–>.AA [1st production]

A–>.aA [2nd production]

A–>.b [3rd production]

STEP2 – Find LR(0) collection of items



The terminals of this grammar are {a,b}.

The non-terminals of this grammar are {S,A}

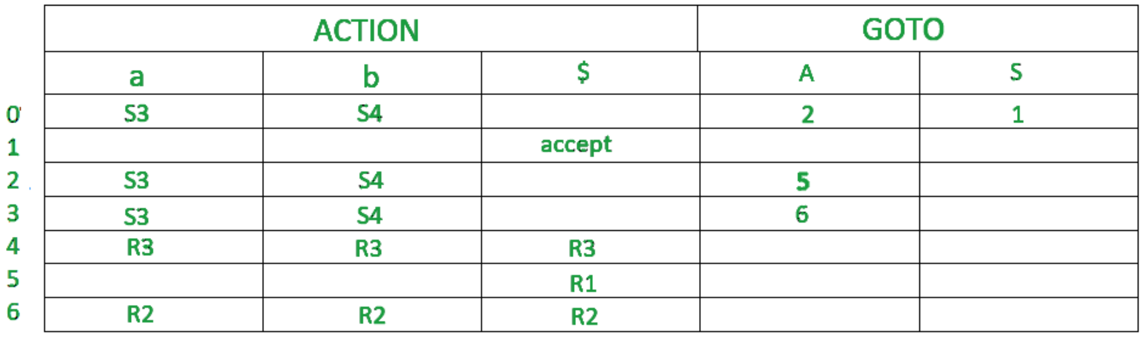
STEP3 –

Find FOLLOW of LHS of production

FOLLOW(S)=$

FOLLOW(A)=a,b,$

Step4-Parsing table



	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

<u>CLR(1) Parser</u>

Steps for constructing CLR parsing table :

1. Writing augmented grammar
2. LR(1) collection of items to be found
3. Defining 2 functions: goto and action in the CLR parsing table

EXAMPLE

S-->AA

A-->aA|b

Solution :

STEP 1 – Find augmented grammar

The augmented grammar of the given grammar is:-

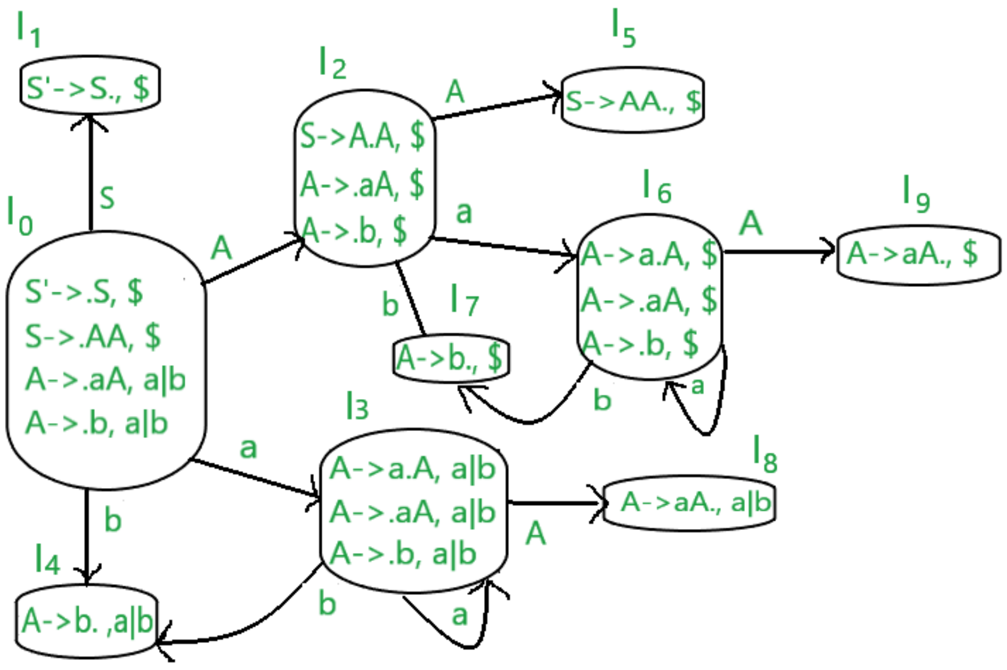
S'-->.S ,$ [0th production]

S-->.AA ,$ [1st production]

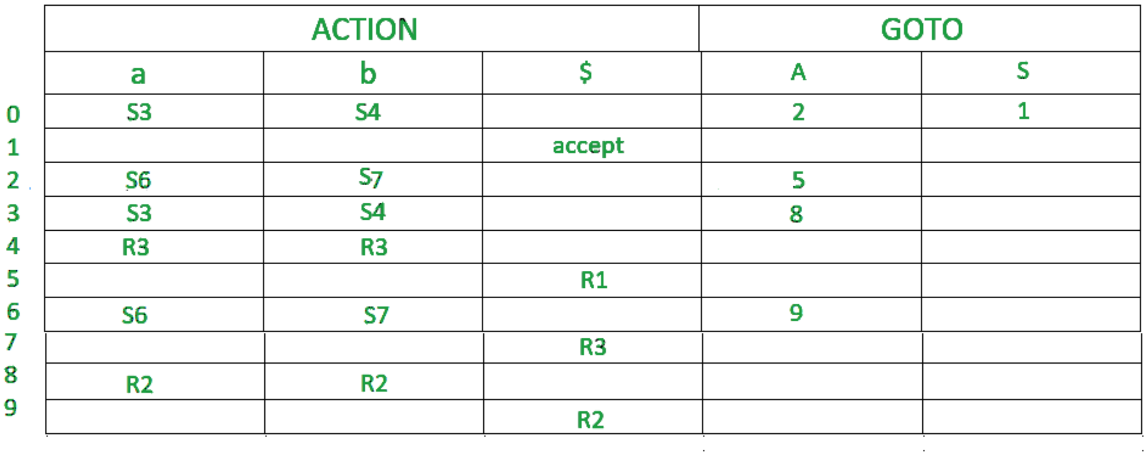
A-->.aA ,a|b [2nd production]

A-->.b ,a|b [3rd production]

STEP 2 – Find LR(1) collection of items



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		

LALR (1) Parsing:

Steps for constructing the LALR parsing table :

1. Writing augmented grammar
2. LR(1) collection of items to be found
3. Defining 2 functions: goto and action in the LALR parsing table

EXAMPLE

S-->AA

A-->aA|b

Solution:

STEP1- Find augmented grammar

The augmented grammar of the given grammar is:-

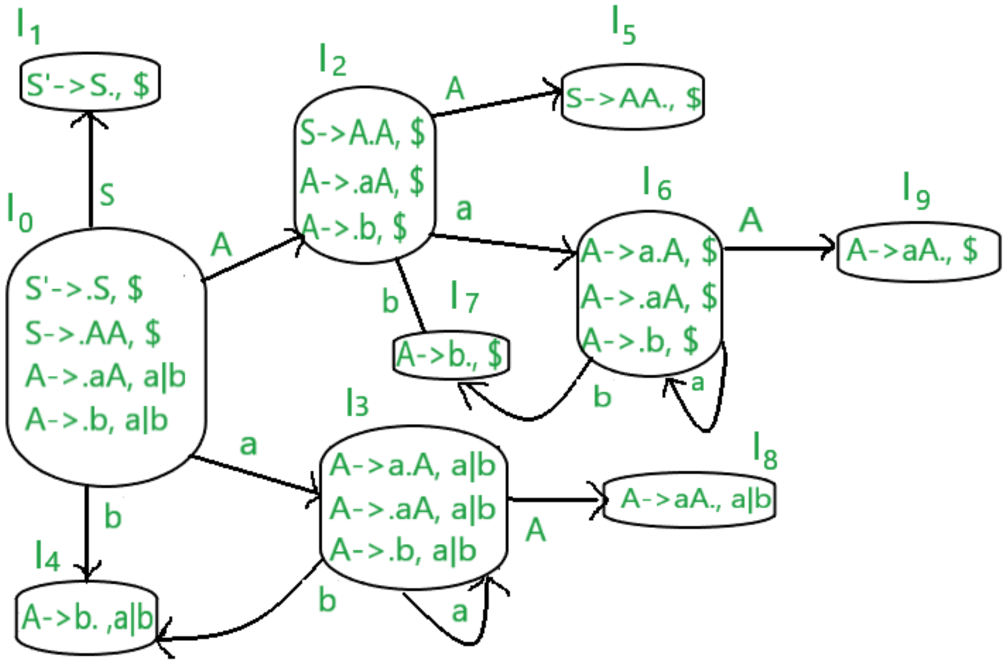
S'-->.S ,$ [0th production]

S-->.AA ,$ [1st production]

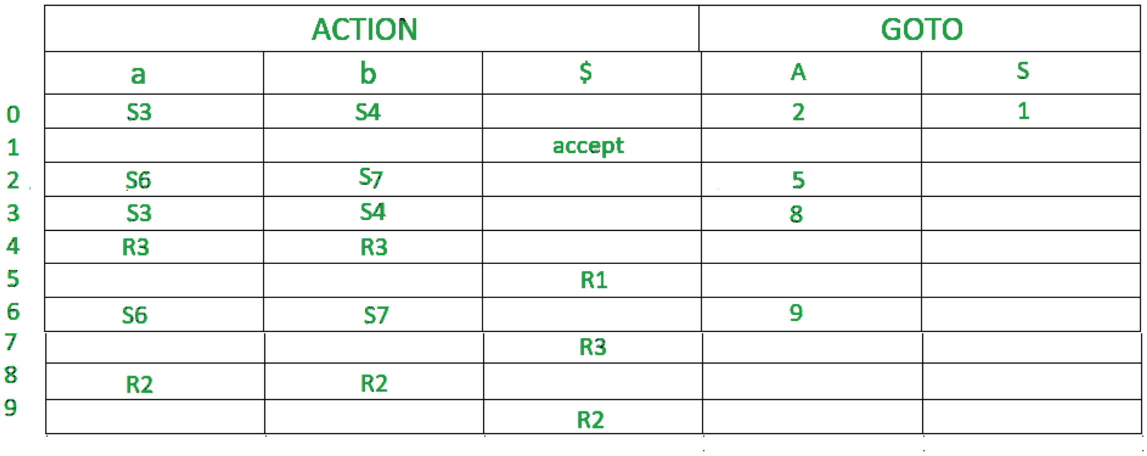
A-->.aA ,a|b [2nd production]

A-->.b ,a|b [3rd production]

STEP2 – Find LR(1) collection of items



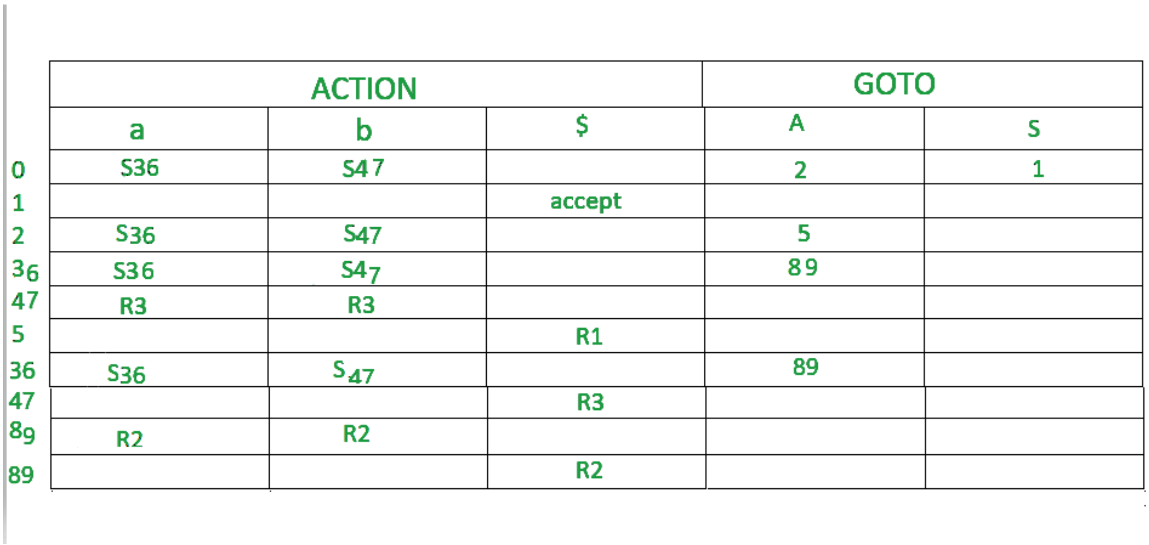
STEP 3 –



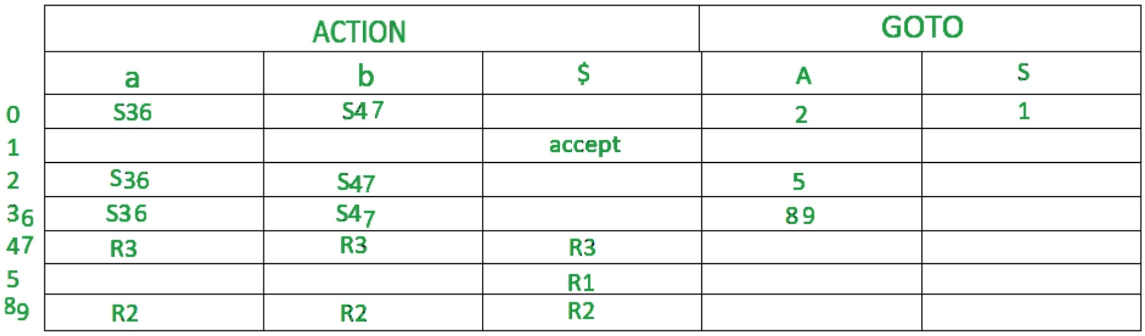
	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S6	S7		5	
3	S3	S4		8	
4	R3	R3			
5			R1		
6	S6	S7		9	
7			R3		
8	R2	R2			
9			R2		

From step 2

- I3 and I6 are similar except their lookaheads.
- I4 and I7 are similar except their lookaheads.
- I8 and I9 are similar except their lookaheads.
- 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)



	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		



	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		

Three address code in Compiler

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code.It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler.

Example – Consider expression a = b * – c + b * – c.

The three address code is:

```
t1 = uminus c

t2 = b * t1

t3 = uminus c

t4 = b * t3

t5 = t2 + t4

a = t5
```

Implementation of Three Address Code –

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. **Quadruple –**
   It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage –

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

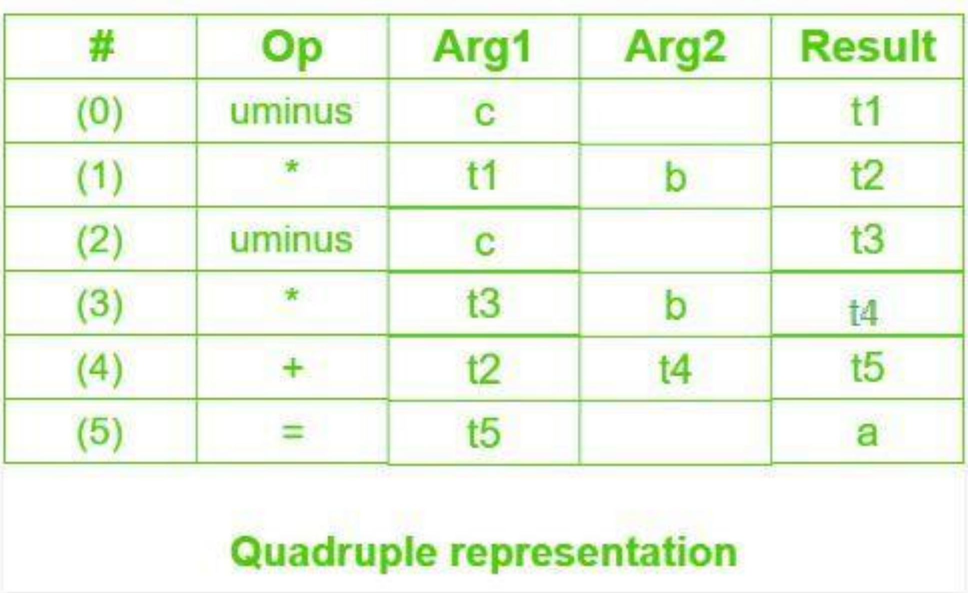
Disadvantage –

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

Example – Consider expression a = b * – c + b * – c.

The three address code is:

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



#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

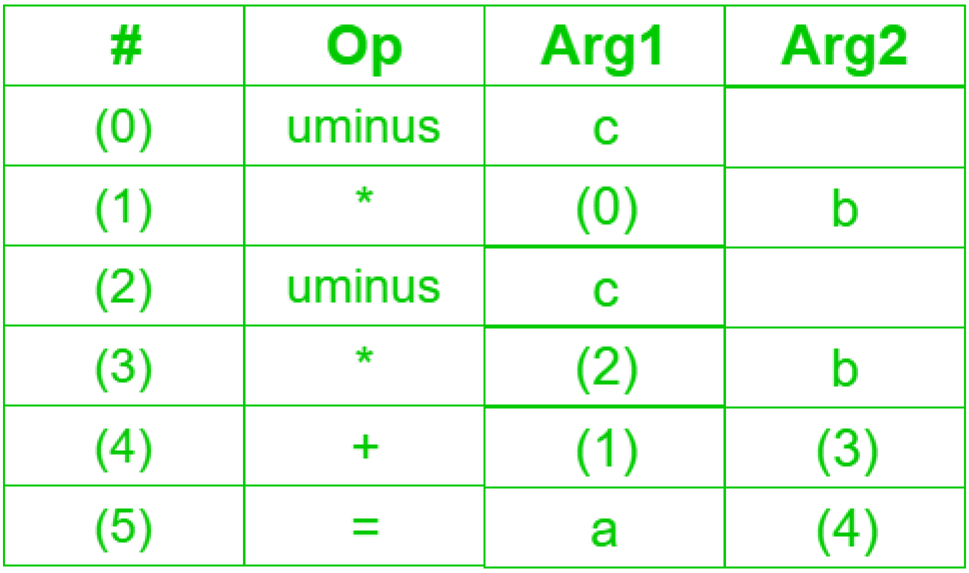
2. **Triples –**

   This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantage –

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Example – Consider expression a = b * – c + b * – c



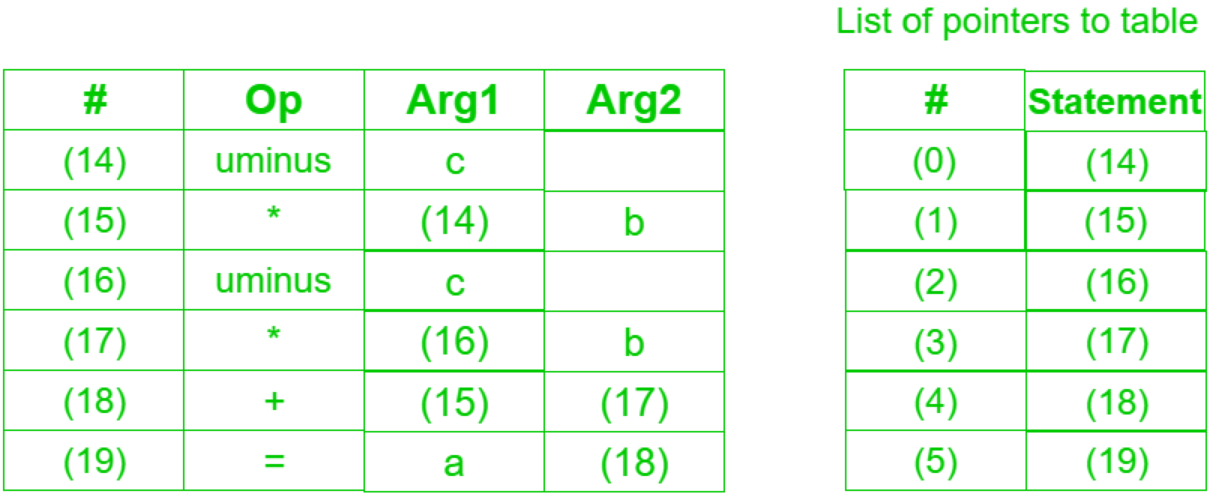
#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

3. Indirect Triples –

This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example – Consider expression a = b * – c + b * – c



#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

List of pointers to table

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

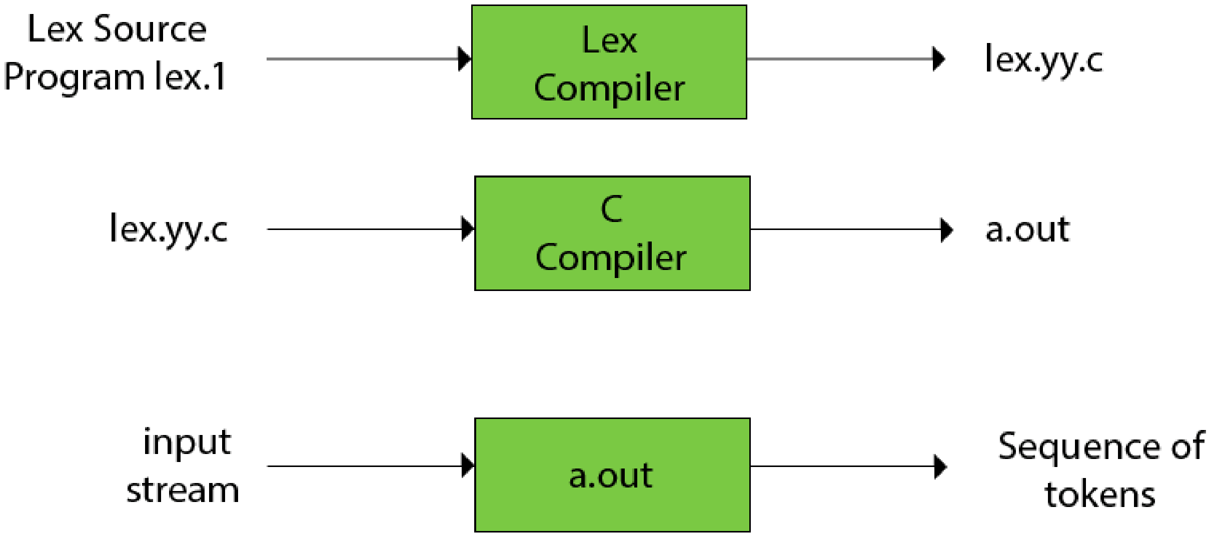
Indirect Triples representation

LEX:

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



Lex file format:

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

1. { definitions }
2. %%
3. { rules }
4. %%
5. { user subroutines

Definitions include declarations of constant, variable and regular definitions.

Rules define the statement of form p1 {action1} p2 {action2}....pn {action}.

Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.

User subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

Storage Organization:

- When the target program executes then it runs in its own logical address space in which the value of each program has a location.
- The logical address space is shared among the compiler, operating system and target machine for management and organization. The operating system is used to map the logical address into physical address which is usually spread throughout the memory.

Storage Allocation

The different ways to allocate memory are:

1. Static storage allocation
2. Stack storage allocation

3. Heap storage allocation

Static storage allocation

- In static allocation, names are bound to storage locations.
- If memory is created at compile time then the memory will be created in static area and only once.
- Static allocation supports the dynamic data structure that means memory is created only at compile time and deallocated after program completion.
- The drawback with static storage allocation is that the size and position of data objects should be known at compile time.
- Another drawback is restriction of the recursion procedure.

Stack Storage Allocation

- In static storage allocation, storage is organized as a stack.
- An activation record is pushed into the stack when activation begins and it is popped when the activation end.
- Activation record contains the locals so that they are bound to fresh storage in each activation record. The value of locals is deleted when the activation ends.
- It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process.

Heap Storage Allocation

- Heap allocation is the most flexible allocation scheme.
- Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.
- Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.
- Heap storage allocation supports the recursion process.

Activation Record

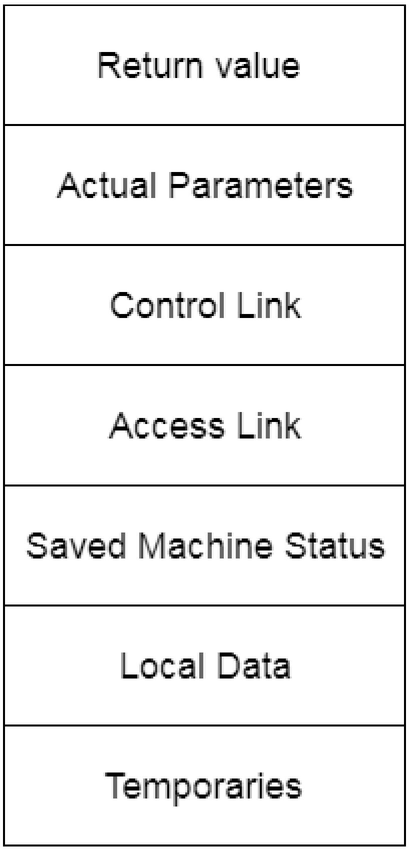
- Control stack is a run time stack which is used to keep track of the live procedure activations i.e. it is used to find out the procedures whose execution have not been completed.

When it is called (activation begins) then the procedure name will push on to the stack and when it returns (activation ends) then it will popped.

Activation record is used to manage the information needed by a single execution of a procedure.

An activation record is pushed into the stack when a procedure is called and it is popped when the control returns to the caller function.

The diagram below shows the contents of activation records:



Return Value: It is used by calling procedure to return a value to calling procedure.

Actual Parameter: It is used by calling procedures to supply parameters to the called procedures.

Control Link: It points to activation record of the caller.

Access Link: It is used to refer to non-local data held in other activation records.

Saved Machine Status: It holds the information about status of machine before the procedure is called.

Local Data: It holds the data that is local to the execution of the procedure.

Temporaries: It stores the value that arises in the evaluation of an expression.

Heap Management

The heap is the portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes it. While local variables typically become inaccessible when their procedures end, many languages enable us to create objects or other data whose existence is not tied to the procedure activation that creates them. For example, both C + + and Java give the programmer new to create objects that may be passed — or pointers to them may be passed — from procedure to procedure, so they continue to exist long after the procedure that created them is gone. Such objects are stored on a heap.

1 The Memory Manager

2 The Memory Hierarchy of a Computer

3 Locality in Programs

1 The Memory Manager

The memory manager keeps track of all the free space in heap storage at all times. It performs two basic functions:

• **Allocation.** When a program requests memory for a variable or object,[3] the memory manager produces a chunk of contiguous heap memory of the requested size. If possible, it satisfies an allocation request using free space in the heap; if no chunk of the needed size is available, it seeks to increase the heap storage space by getting consecutive bytes of virtual memory from the operating system. If space is exhausted, the memory manager passes that information back to the application program.

• **Deallocation**. The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests. Memory managers typically do not return memory to the operating sys-tem, even if the program's heap usage drops.

2. The Memory Hierarchy of a Computer

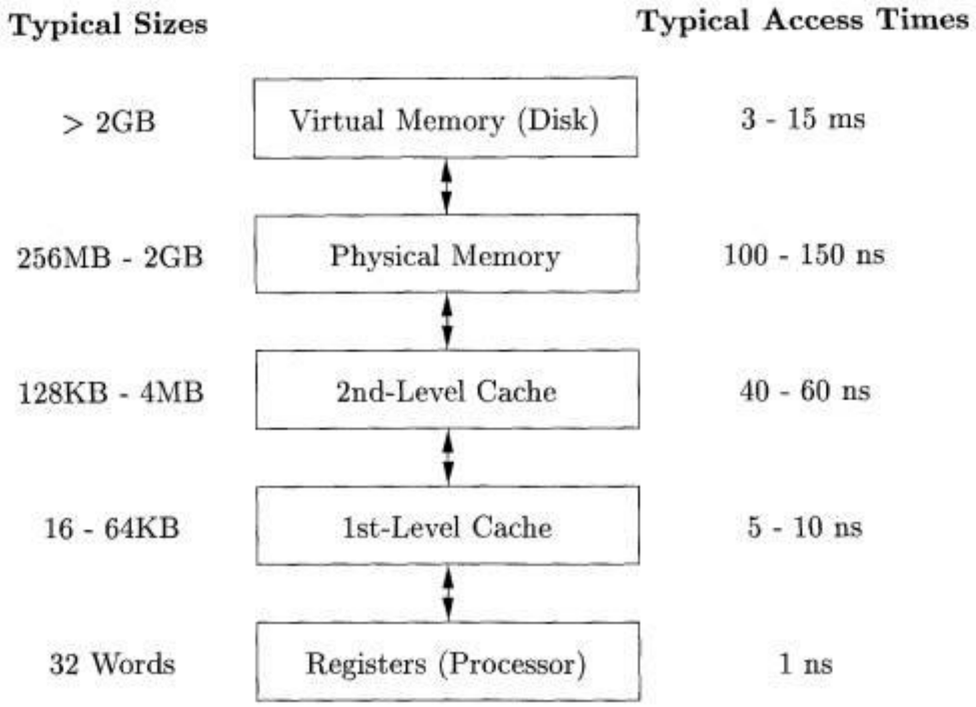


Figure 7.16: Typical Memory Hierarchy Configurations

3. Locality in Programs

Most programs exhibit a high degree of locality; that is, they spend most of their time executing a relatively small fraction of the code and touching only a small fraction of the data. We say that a program has temporal locality if the memory locations it accesses are likely to be accessed again within a short period of time. We say that a program has *spatial locality* if memory locations close to the location accessed are likely also to be accessed within a short period of time.

Peephole Optimization :

Peephole optimization is a type of code Optimization performed on a small part of the code. It is performed on a very small set of instructions in a segment of code.
The small set of instructions or small part of code on which peephole optimization is performed is known as peephole or window.
It basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without a change in output. The peephole is machine-dependent optimization.

Characteristics of peephole optimizations:

Redundant-instructions elimination
Flow-of-control optimizations
Algebraic simplifications
Use of machine idioms
Unreachable

Redundant Loads And Stores:

If we see the instructions sequence
(1) MOV R0,a
(2) MOV a,R0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Unreachable Code:

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1. In C, the source code might look like:

```
#define debug 0
….

If ( debug ) {
Print debugging information

}
```

Flows-Of-Control Optimizations:

The 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 jump sequence

```
        goto L1
        ….

L1: gotoL2 (d)
```

by the sequence

```
        goto L2
        ….

L1: goto L2
```

Algebraic Simplification:

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

x := x+0 or
x := x * 1

are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Use of Machine Idioms:

The target machine may have hardware instructions to implement certain specific operations efficiently. 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 these 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 i : =i+1.

i:=i+1 → i++
i:=i-1 → i- -

Basic Block and Flow Graph:

Basic Block is a straight line code sequence that has no branches in and out branches except to the entry and at the end respectively. Basic Block is a set of statements that always executes one after other, in a sequence.
The first task is to partition a sequence of three-address code into basic blocks. A new basic block is begun with the first instruction and instructions are added until a jump or a label is met. In the absence of a jump, control moves further consecutively from one instruction to another. The idea is standardized in the algorithm below:
Algorithm:
Partitioning three-address code into basic blocks.

Rule 1 : Determining the Leader.
Rule 2: Determining the Basic Block

In Rule 1 we have 3 statement:

1. The first three-address instruction of the intermediate code is a leader.

2. Instructions that are targets of unconditional or conditional jump/goto statements are leaders.

3. Instructions that immediately follow unconditional or conditional jump/goto statements are considered leaders.

Intermediate code to set a 10*10 matrix to an identity matrix:

1) i=1   //Leader 1 (First statement)

2) j=1   //Leader 2 (Target of 11th statement)

3) t1 = 10 * i   //Leader 3 (Target of 9th statement)

```
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= goto (3)
10) i = i + 1                    //Leader 4 (Immediately following Conditional goto statement)
11) if i <= 10 goto (2)
12) i = 1                        //Leader 5 (Immediately following Conditional goto statement)
13) t5 = i - 1                   //Leader 6 (Target of 17th statement)
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

The given algorithm is used to convert a matrix into identity matrix i.e. a matrix with all diagonal elements 1 and all other elements as 0.

Steps (3)-(6) are used to make elements 0, step (14) is used to make an element 1. These steps are used recursively by goto statements.

There are **6 Basic Blocks** in the above code :

B1) Statement 1
B2) Statement 2
B3) Statement 3-9
B4) Statement 10-11
B5) Statement 12
B6) Statement 13-17

Issues in the design of a code generator:

Code generator **converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code.**

The following issue arises during the code generation phase:

- **Input to code generator** – The input to the code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data objects denoted by the names in the intermediate representation.

- **Target program:** The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, or assembly language.

- **Memory Management –** Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for the name.

- **Instruction selection –** Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also play a major role when efficiency is considered.