CS6660- COMPILER DESIGN – UNIT IV

SYNTAX DIRECTED TRANSLATION & RUN TIME ENVIRONMENT

Syntax directed Definitions-Construction of Syntax Tree-Bottom-up Evaluation of S-Attribute Definitions- Design of predictive translator - Type Systems-Specification of a simple type checker-Equivalence of Type Expressions-Type Conversions.RUN-TIME ENVIRONMENT: Source Language Issues-Storage Organization-Storage Allocation-Parameter Passing-Symbol Tables-Dynamic Storage Allocation-Storage Allocation in FORTAN.

TWO MARKS

1. Define S-attribute and L-attribute definitions.

S-attribute is a syntax-directed definition that uses synthesized attributes only.

L-attribute is an attributed definition in which:

- Each attribute in each semantic rule for the production A→X1,X2,..Xn is either a synthesized attribute or an inherited attribute Xj which depends only on the inherited attribute of A and/or the attributes of X1,…Xj-1.
- Independent of the evaluation order.

Every S-attributed definition is an L-attributed definition.

2. What are the parameter transmission mechanisms?

1. Call by value
2. Call by value-result
3. Call by reference
4. Call by name

3. What do you mean by syntax directed translation?

Syntax-directed translation specifies the translation of a construct in terms of Attributes associated with its syntactic components. Syntax-directed translation uses a context free grammar to specify the syntactic structure of the input. It is an input- output mapping

4. What is the purpose of symbol table?

A symbol table is a containing a record for each identifier, with fields for the attributes of the identifier. The symbol table is used by semantic analysis phase, intermediate code generation phase and code generation phase.

5. Define an attribute. Give the types of an attribute?

An attribute may represent any quantity, with each grammar symbol, it associates a set of attributes and with each production, a set of semantic rules for computing values of the attributes associated with the symbols appearing in that production.

Example: a type, a value, a memory location etc.,

i) Synthesized attributes.

ii) Inherited attributes.

6. What are the functions used to create the nodes of syntax trees?

i) Mknode (op, left, right)
ii) Mkleaf (id,entry)

iii) Mkleaf (num, val)

7. What are the functions for constructing syntax trees for expressions?
i) The construction of a syntax tree for an expression is similar to the translation of the expression into postfix form.
ii) Each node in a syntax tree can be implemented as a record with several fields.

8. What is the various data structure used for implementing the symbol table?
1. Linear list
2. Binary tree
3. Hash table

9. Give short note about call-by-name?
Call by name, at every reference to a formal parameter in a procedure body the name of the corresponding actual parameter is evaluated. Access is then made to the effective parameter.

10. How parameters are passed to procedures in call-by-value method?
This mechanism transmits values of the parameters of call to the called program.
The transfer is one way only and therefore the only way to returned can be the value of a function.

```
Main ( )
{ print (5); }
Int
Void print (int n)
{ printf (“%d”, n); }
```

11. Define register interference graph.

Each node corresponds to m a unique variable**.** An edge exits between two nodes if an only if the corresponding variables are live simultaneously at some point in the program

12. Write the pseudo code of graph coloring problem?

i/p interference graph, k colors

- The graph is k-colorable
- Pick a node t with fewer than k neighbors.
- Push the variable t onto stack and remove it form the graph along with all adjoining edges.
- Repeat the steps 1 and 2 until no node is left. While stack is not empty do
- Pop a node, color the node with a color different from those end do assigned to already colored Neighbors of x.

13. What are the objects to be maintained during runtime?

- Generated code
- Data objects
- Stack

14. What are the fields in an activation record?(Nov/Dec 2013)

- Parameters passed to the procedure
- Book keeping information includes return address
- Storage for local variables
- Storage for local temporaries

15. Define register descriptor.

A register descriptor keeps track of what is currently in each register.

16. What is address descriptor?

An address descriptor keeps track of the location where the current value of the name can be found at run time.

17. What are the storage allocation strategies during runtime?(Nov/Dec 2016)

- Stack allocation
- Heap allocation
- Static allocation

18. What are the different ways of parameters to a procedure?

- Call-by-value
- Copy-by-reference
- Copy restore
- Call-by-name
- Macro expansion

19. What is Activation tree?

A tree used to represent the control enters and leaves activations where the activation is the execution of a procedure.

20. What are the limitations of Static Allocation?

- Size of a data object and constraints must be known at compile time
- Recursive procedures are restricted
- Data structures cannot be created dynamically

21. What is Register Allocation?

- Instructions involving register operands are shorter and faster than those involving operands in memory.

The use of registers is subdivided into two subproblems:

- ***Register allocation*** – the set of variables that will reside in registers at a point in the program is selected.
- ***Register assignment*** – the specific register that a variable will reside in is picked.
- Certain machine requires even-odd *register pairs* for some operands and results.

  For example, consider the division instruction of the form:

  D x, y

  where, x – dividend even register in even/odd register pair y – divisor

  even register holds the remainder odd register holds the quotient

22. What are the issues in static allocation? (Nov/Dec-09)

- Size of a data object and constraints must be know at compile time
- Recursive procedures are restricted

- Data structures cannot be created dynamicaly

23. What are the difference between static allocation and stack allocation?

- **In static allocation**, the position of an activation record in memory is fixed at compile time.
- **In stack allocation**, a new activation record is pushed onto the stack for each execution of a procedure. The record is popped when the activation ends.

24. Define symbol table. [May/June-14]

- The symbol table includes the name and value for each symbol in the source program, together with flags to indicate error conditions. Sometimes it may contain details about the data area. SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval.

25. What are the limitations of Static Allocation? (Apr/May -2011)

- Size of a data object and constraints must be known at compile time
- Recursive procedures are restricted
- Data structures cannot be created dynamically

26. Give two examples for each of top down parser and bottom up parser? (Apr/May – 2010)

Top – down parser

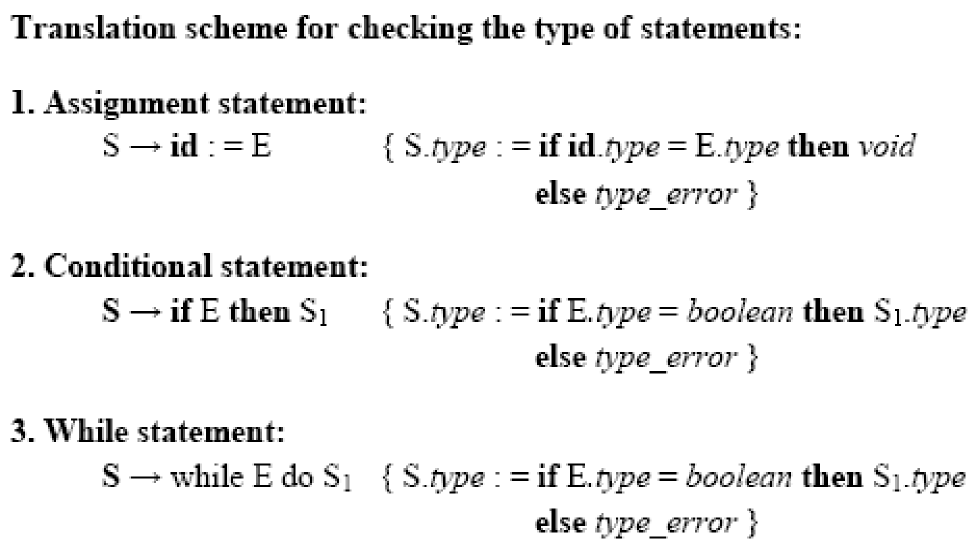
- Predictive parser
- Non Predictive parser

Bottom – up parser

- LR parser
- Operator precedence parser

27. Mention the two rules for type checking (Nov/Dec – 2011)

We can now write the semantic rules for type checking of statements as follows



Translation scheme for checking the type of statements:

1. Assignment statement:

S → **id** : = E { S.*type* : = **if id**.*type* = E.*type* **then** *void* **else** *type_error* }

2. Conditional statement:

S → **if** E **then** S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type* **else** *type_error* }

3. While statement:

S → while E do S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type* **else** *type_error* }

28. Give examples for static check(May/June-2013)

Examples of static checks are

1.Type checks
2.Flow-of-control checks
3.Uniqueness checks
4.Name –related checks

29. What is meant by coercion? (Nov/Dec 2013)

Process by which a compiler automatically converts a value of one type into a value of another type when that second type is required by the surrounding context.

30. What is SDD?

A ***syntax-directed definition*** associates

1. With each grammar symbol, a set of attributes, and

2. With each production, a set of *semantic rules* for computing the values of the attributes associated with the symbols appearing in the production.

31. Define annotated parse tree.

A parse tree, showing the value(s) of its attribute(s) is called an ***annotated parse tree***.

32. What is dangling reference? May/Jun -2012 ,May/Jun -2016

A problem which arises when there is a reference to storage that has been deallocated

Eg:

Int I = 15 ,* p

P = &I;

```
Free (p);
Printf(“%u”,p);
```

33. What is DAG?May/June-2016

A DAG for a basic block is a **directed acyclic graph** with the following labels on nodes:

1. Leaves are labeled by unique identifiers, either variable names or constants.
2. Interior nodes are labeled by an operator symbol.
3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.

34. Write short notes on access to non — local names.

The scope rules of a language determine the treatment of references to non — local names.

i. A common rule called the lexical or static scope rule determines the declaration that applies to a name by examining the program text alone. Eg. Pascal, C, Ada
ii. Dynamic scoping determines the declaration applicable

35. What are inherited attributes?

An inherited attribute is one whose value at a node in a parse tree is defined in terms of attributes at the parent and/ or siblings of that node.

36. How do you evaluate semantic rules?

a) May generate intermediate code
b) May put information into the symbol table
c) May perform type checking
d) May issue error messages
e) May perform some other activities
f) In fact, they may perform almost any activities

37. What are Translation scheme?

(a) Indicate the order of evaluation of semantic actions associated with a production rule.

(b) In order words, translation schemes give a little bit information about implementation details

38.What are the different representations of intermediate languages?

a) Syntax tree
b) Postfix
c) There address code

39. What are the implementations of three- address statements?

a) Quadruples
b) Triples
c) Indirect triples

40. Give syntax directed translation for the following statement call p1 (int a, int b).

S-> call id (Elist) — generate a param statement for each item on queue, causing these statements to follows the statements evaluating the argument expressions { for each item p on queue do emit (`param' **P);**

Emit('call'id.place)}

Elist-* Elist, E {append E. place to the end of the queue}

Elist –> E {initialize queue to contain only E. place}

41. Define Type checking.

A compiler must check that the source program follows both syntactic and semantic conventions of the source language. This checking is also called as static checking which detects and reports errors.

42. Define type systems.

It contains a collection of rules for assigning type expressions to various parts of a program. The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types and the rules for assigning types to language constructs.

43. Define type expression.

A type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions.

Eg. Boolean, real, int are type expressions

44. Define strongly typed language.

A language is strongly typed if its compiler can guarantee that the program it accepts will execute without type errors.

45. What are the contents of activation record?

The activation record is a block of memory used for managing the information needed by a single execution of a procedure various fields of activation record are:

- Temporary variables
- Local variables
- Saved machine registers
- Control link
- Access link
- Actual parameters
- Return values

46. Suggest a suitable approach for computing hash function.

Using hash function we should obtain exact locations of name in symbol table.

The hash function should result in uniform distribution of names in symbol table.

The hash function should be such that there will be minimum number of collisions. Collision is such a situation where hash function results in same location for storing the names.

47. What are the data structures used for designing symbol table?

a. List
b. Self-organizing list
c. Binary tree
d. Hash table

48. Define Compaction.

The process where all the used blocks are moved at the one end of heap memory so that all the free blocks are available in one large free block.

49. What is Back patching?

The labels left unspecified in the jumps statements will be filled in when the proper label can be determined. This process of filling –in of labels is backpacthing.

source:

1) If a or b then
2) If c then
3) X = y + 1

	Translation:		After back patching
1)	If a goto :L	1)	100: if a goto <u>103</u>
2)	If b goto L 1	2)	101 : if b goto <u>103</u>
3)	Goto L3	3)	goto <u>106</u>
4)	Ll : if c goto L2		
5)	goto L3		
6)	L2: x = y+1		
7)	L3:		

50. List of functions used to manipulate list of labels in back patching.

(i) Makelist(i) — create a newlist and put i in the list
(ii) Merge (p 1, p2) — merges two lists pointed by pl and p2
(iii) Back patch (pj) insert the target label j for each list pointed by p.

PART B

1.Explain about Syntax-Directed Definitions.

A ***syntax-directed definition*** associates

1. With **each grammar symbol**, a **set of attributes**, and
2. With **each production**, a set of ***semantic rules*** for computing the values of the attributes **associated with the symbols** appearing in the production.

(OR)

A ***syntax-directed definition*** (**SDD**) is a context-free grammar together with **attributes and rules**. Attributes are **associated with grammar symbols** and rules are **associated with productions**.

If ***X*** is a **symbol** and ***a*** is one of its **attributes**, then we write ***X.a*** to denote the value of a at a particular parse-tree node labeled *X*.

Attributes may be of any kind: **numbers, types, table references, or strings, for instance**.

INHERITED AND SYNTHESIZED ATTRIBUTES

There are two kinds of attributes for nonterminals:

1. A ***synthesized attribute*** for a nonterminal *A* at a parse-tree node *N* is defined by a semantic rule associated with the production at *N*. Note that the production must have *A* as its head. A synthesized attribute at node *N* is defined only in terms of **attribute values at the children** of *N* and at *N* itself.
2. An ***inherited attribute*** for a non-terminal B at a parse-tree node *N* is defined by a semantic rule associated with the production at the parent of *N*. Note that the production must have *B* as a symbol in its body. An inherited attribute at node *N* is defined only in terms of attribute values at **N's parent, *N* itself, and *N's* siblings**.

SYNTAX-DIRECTED DEFINITIONS

Example : The SDD given in the example is based on our familiar grammar for arithmetic expressions with operators + and *. It evaluates expressions terminated by an end marker **n.** In the SDD, **each of the non-terminals has a single synthesized attribute**, called *val.* We also suppose that the terminal **digit** has a synthesized attribute *lexval*, which is an integer value returned by the lexical analyzer.

P R O D U C T I O N	**SEMANTIC RU L ES**
1) $L \rightarrow En$	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow$ digit	$F.val$ = digit.lexval

Syntax-directed definition of a simple desk calculator

The rule for **production 1, *L* ->• *E* n,** sets *L.val* to *E.val*.

Production 2, *E* → *E* + *T*, also has one rule, which computes the *val* attribute for the head *E* as the sum of the values at *E* and *T*. At any parse tree node *N* labeled *E*, the value of *val* for *E* is the sum of the values of *val* at the children of node *N* labeled *E* and *T*.

Production 3, *E* → *T*, has a single rule that defines the value of *val* for *E* to be the same as the value of *val* at the child for *T*.

Production 4 is similar to the second production; its rule multiplies the values at the children instead of adding them.

The rules for productions 5 and 6 copy values at a child, like that for the third production.

Production 7 gives *F.val* the value of a digit, that is, the numerical value of the token digit that the lexical analyzer returned.

An SDD that involves only **synthesized attributes** is called ***S-attributed***. In an S-attributed SDD, each rule computes an attribute for the non-terminal at the head of a production from attributes taken from the body of the production.

An S-attributed SDD can be implemented naturally in conjunction with an LR parser.

An SDD without side effects is sometimes called an ***attribute grammar***. The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

Evaluating an SDD at the Nodes of a Parse Tree

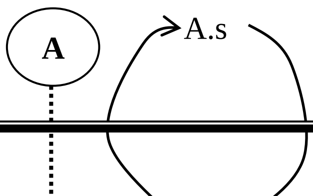
A parse tree, showing the value(s) of its attribute(s) is called an ***annotated parse tree***.

With synthesized attributes, evaluate attributes in any bottom-up order, such as that of a post-order traversal of the parse tree.

For SDD's with both inherited and synthesized attributes, there is no guarantee that there is even one order in which to evaluate attributes at nodes. For instance, consider non-terminals *A* and *B*, with synthesized and inherited attributes *A.s* and *B.i*, respectively, along with the production and rules.

PRODUCTION	**SEMANTIC RULES**
$A \rightarrow B$	$A.s = B.i;$
	$B.i = A.s + 1$

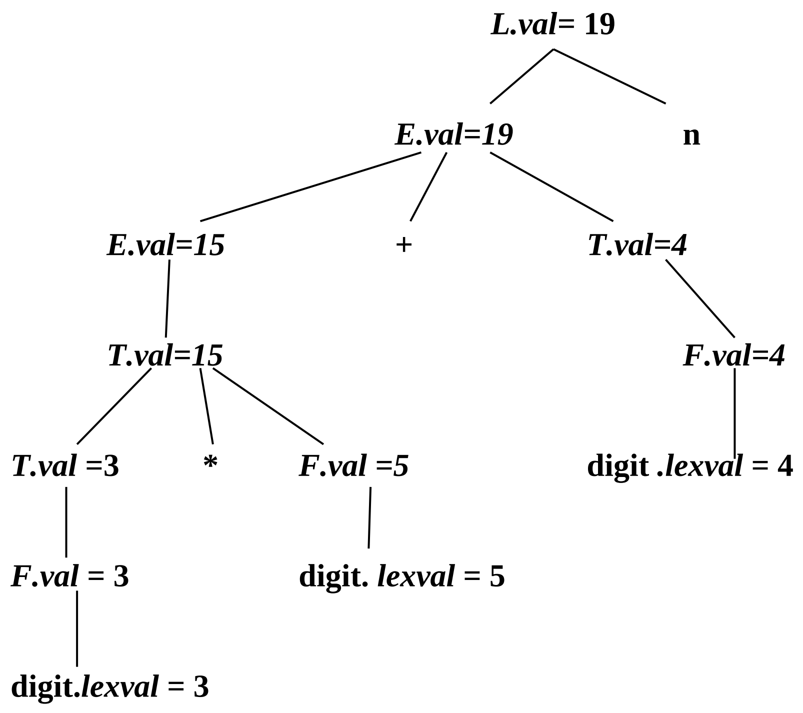
These rules are circular; it is impossible to evaluate either *A.s* at a node *N* or *B.i* at the child of *N* without first evaluating the other.



The circular dependency of *A.s* and *B.i* on one another

It is computationally difficult to determine whether or not there exists any circularity in any of the parse trees that a given SDD could have to translate.

E x a m p l e: An annotated parse tree for the input string 3 * 5 + 4 **n,** constructed using the grammar and rules. The values of ***lexval*** are presumed supplied by the lexical analyzer. Each of the nodes for the non-terminals has attribute ***val*** computed in a bottom-up order.
Inherited attributes are useful when the structure of a parse tree does not "match" the abstract syntax of the source code.



Annotated parse tree for 3 * 5 + 4 n

E x a m p l e: Consider the SDD computes terms like 3 * 5 and 3 * 5 * 7 .

The top-down parse of input 3*5 begins with the production $T \rightarrow FT'$. Here, *F* generates the digit 3, but the operator * is generated by T'. Thus, the left operand 3 appears in a different sub-tree of the parse tree from *. An inherited attribute will therefore be used to pass the operand to the operator.

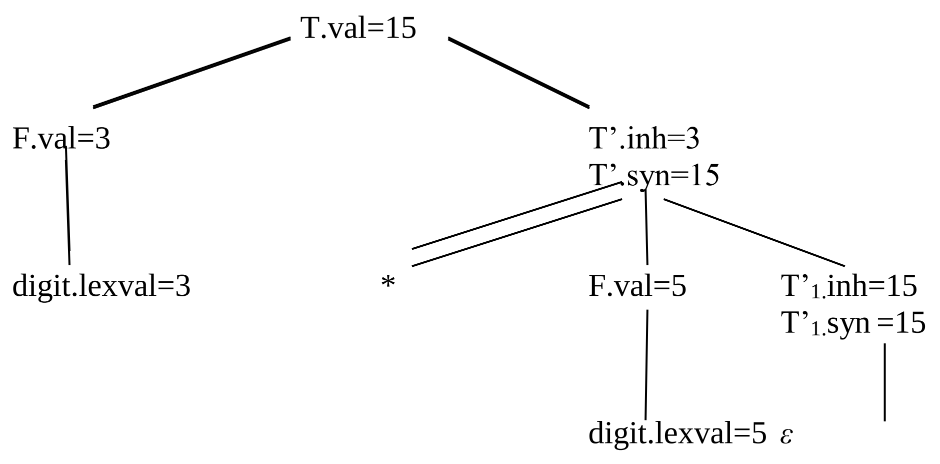
P R O D U C T I O N	**SEMANTIC RU L ES**
1) $T \rightarrow FT'$	$T'.inh = F.val$
	$T.val = T'.syn$
2) $T' \rightarrow *FT_1'$	$\{ T'_1.inh = T'.inh \times F.val$
	$T'.syn = T'_1.syn\}$
3) $T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
4) $F \rightarrow$ digit	$F.val$ = digit. *lexval*

Figure 5.4: An SDD based on a grammar suitable for top-down parsing

Each of the non-terminals *T* and *F* has a synthesized attribute ***val;*** the terminal **digit** has a synthesized attribute ***lexval.***
The non-terminal *T* has two attributes: an **inherited** attribute ***inh*** and a **synthesized** attribute ***syn.***
Consider the annotated parse tree for 3 * 5. The leftmost leaf in the parse tree, labeled digit, has attribute value ***lexval* = 3**, where the 3 is supplied by the lexical analyzer. Its parent is for **production 4, *F* → digit**. The only semantic rule associated with this production defines ***F.val* = digit.*lexval*,** which equals 3.

At the second child of the root, the inherited attribute ***T'.inh*** is defined by the semantic rule ***T'.inh* = *F.val*** associated with production 1. Thus, the left operand, 3, for the * operator is passed from left to right across the children of the root.
The production at **the node for *T'* is T' → * *FT'₁***. The inherited attribute $T'_1.inh$ is defined by the semantic rule $T'_1.inh = T'.inh \times F.val$ associated with production 2.

With ***T'.inh* = 3** and ***F.val* = 5**, we get $T'_1.inh$ **= 1 5** . At the lower node for T'_1, the production is **T' -> ε.** The semantic rule ***T'.syn* = *T'.inh*** defines $T'_1.syn$ **= 15**. The ***syn*** attributes at the nodes for **T'** pass the value 15 up the tree to the node for T, where ***T.val* = 15.**



Annotated parse tree for 3 * 5

2. Explain about the construction of syntax tree (or) Applications of Syntax-Directed Translation.

The main application of syntax-directed translation is the **construction of syntax trees**.

Since some compilers use syntax trees as an intermediate representation, a common form of SDD turns its input string into a tree. To complete the translation to intermediate code, the compiler may then walk the syntax tree, using another set of rules that are in effect an SDD on the syntax tree rather than the parse tree.

Two SDD's considered for constructing syntax trees for expressions.

The first, an **S-attributed** definition, is suitable for use during **bottom-up parsing**.

The second, **L-attributed**, is suitable for use during **top-down parsing**.

Construction of Syntax Trees

Each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.

A syntax-tree node representing an expression $E_1 + E_2$ has label + and two children representing the sub-expressions E_1 and E_2 .

Implement the nodes of a syntax tree by objects with a suitable number of fields. Each object will have an *op* field that is the label of the node.

The objects will have additional fields as follows:

• If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function ***Leaf (op, val)*** creates a leaf object. Alternatively, if nodes are viewed as records, then ***Leaf*** returns a pointer to a new record for a leaf.

• If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function ***Node*** **takes two or more arguments:** ***Node(op,c_1,c_2,... ,c_k)*** creates an object with first field *op* and *k* additional fields for the *k* children ***c_1,c_2,... ,c_k.***

E x a m p l e: The S-attributed definition constructs syntax trees for a simple expression grammar involving only the binary operators **+ and -.** As usual, these operators are at the same precedence level and are jointly left associative. All non-terminals have **one synthesized** attribute ***node***, which represents a node of the syntax tree.

Every time the first production ***E-> E + T*** is used, its rule creates a node with **' + ' for *op*** and two children, ***E.node*** and ***T.node***, for the sub expressions. The second production has a similar rule.

P R O D U C T I O N	**SEMANTIC RU L ES**
1) *E →E + T*	*E.node* = **new** *Node(''+',E.node,T.node)*
2) *E →E—T*	*E.node* = **new** *Node('-',E.node,T.node)*
3) *E→T*	*E.node = T.node*
4) *T→(E)*	*T.node = E.node*
5) T → M d	*T.node* **= new** *Leaf (id, id. entry)*
6) *T* → n um	*T.node* **= new** Leaf(num, num. va l)

Constructing syntax trees for simple expressions

For production 3, *E -> T*, no node is created, since ***E.node*** is the same as ***T.node***. Similarly, no node is created **for production 4, T→ (*E*).** The value of ***T.node*** is the same as ***E.node***, since parentheses are used only for grouping; they influence the structure of the parse tree and the syntax tree, but once their job is done, there is no further need to retain them in the syntax tree.

The last two T-productions have a single terminal on the right. We use the constructor ***Leaf*** to create a suitable node, which becomes the value of ***T.node.***

The construction of a syntax tree for the input ***a* — 4 + *c.***

The nodes of the syntax tree are shown as records, with the ***op*** field first. Syntax-tree edges are now shown as solid lines. The underlying parse tree, which need not actually be constructed, is shown with dotted edges. The third type of line, shown dashed, represents the values of ***E.node*** **and *T.node;*** each line points to the appropriate syntax-tree node.

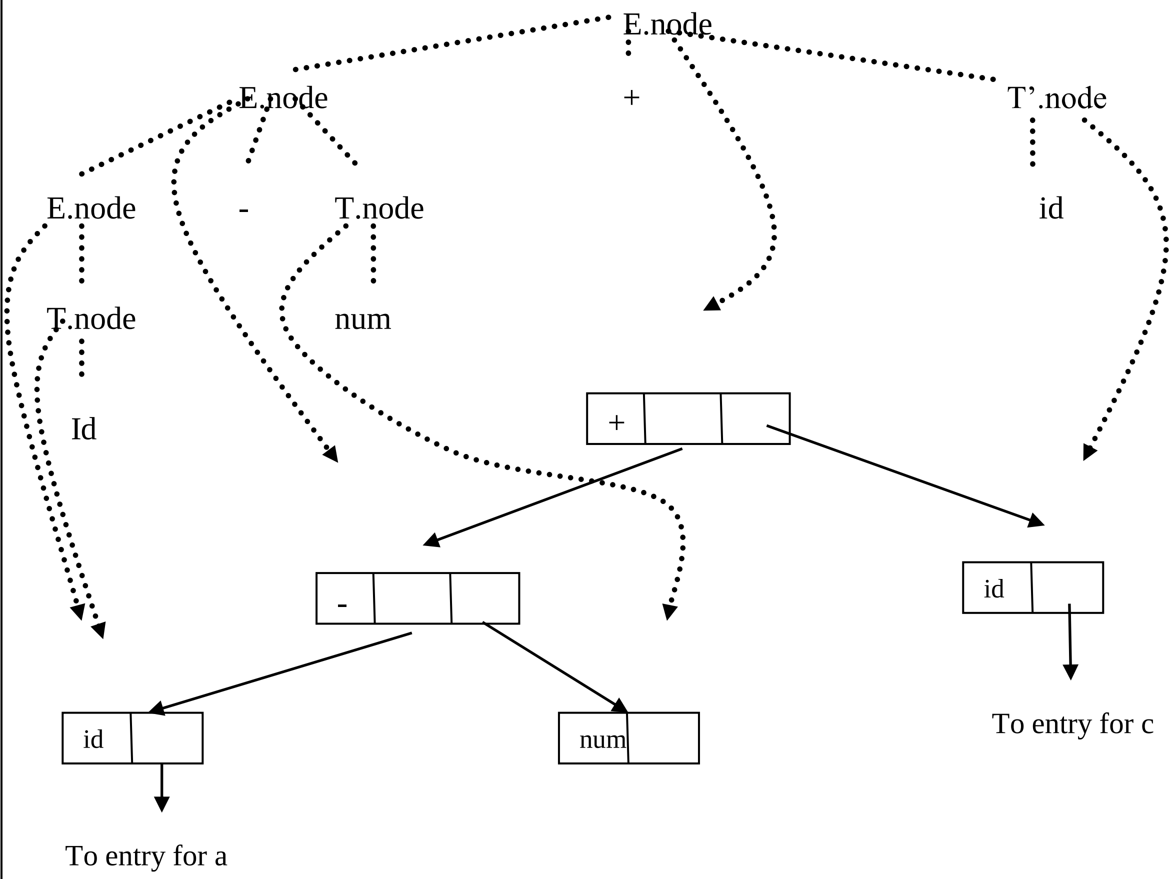
At the bottom we see leaves for ***a*, 4 and c,** constructed by ***Leaf***. We suppose that the lexical value **id.** ***entry*** points into the symbol table, and the lexical value **num**.**val** is the numerical value of a constant. These leaves, or pointers to them, become the value of ***T.node*** at the three parse-tree nodes labeled *T*, according to rules 5 and 6. Note that by rule 3, the pointer to the leaf for *a* is also the value of ***E.node*** for the leftmost *E* in the parse tree.

Rule 2 causes us to create a node with *op* equal to the minus sign and pointers to the first two leaves. Then, rule 1 produces the root node of the syntax tree by combining the node for — with the third leaf.

If the rules are evaluated during a post-order traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps ends with p_5 pointing to the root of the constructed syntax tree.

With a grammar designed for top-down parsing, the same syntax trees are constructed, using the same sequence of steps, even though the structure of the parse trees differs significantly from that of syntax trees.

Example : The L-attributed definition performs the same translation as the S-attributed definition.



Syntax tree for a- 4+c

1) p_1 = **new** Leaf**(id,** *entry-a);*

2) p_2 = **new** *Leaf* **(num,** 4);
3) p_3 = **new** *Node('-',p1,p2);*
4) p_4 = **new** *Leaf*'**(id,** *entry-c);*
5) p_5 = **new** *Node('+',p_3,p_4);*

Steps in the construction of the syntax tree for *a* — 4 + c

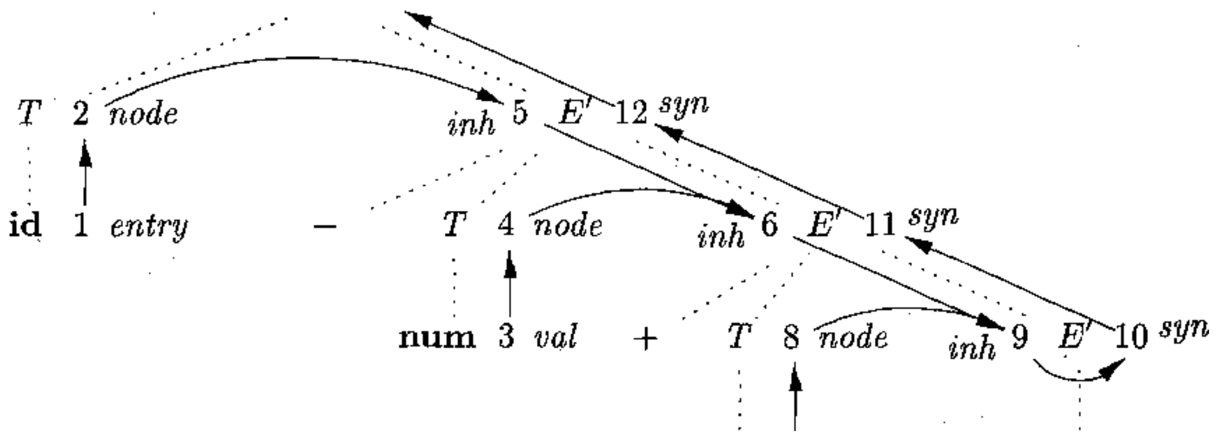
The rules for building syntax trees in this example are similar to the rules for the **desk calculator**. In the desk-calculator example, a term ***x* * *y*** was evaluated by passing ***x* as an inherited attribute**, since ***x* and * *y*** appeared **in different portions of the parse tree**. Here, *the* idea is to build a **syntax tree for *x* + *y*** by passing ***x* as an inherited attribute**, since ***x* and + *y* appear in different sub trees**. Non-terminal ***E'*** is the counterpart of non-terminal **T'**. Compare the dependency graph for ***a* — 4 + c** with that for **3 * 5.**

Non-terminal *E'* has an inherited attribute *inh* and a synthesized attribute *syn.* Attribute *E'.inh* represents the partial syntax tree constructed so far. Specifically, it represents the root of the tree for the prefix of the input string that is to the left of the subtree for *E'.* At node 5 in the dependency graph, *E'.inh* denotes the root of the partial syntax tree for the identifier a; that is, the leaf for *a.* At node 6, *E'.inh* denotes the root for the partial syntax tree for the input *a* — 4. At node 9, *E'.inh* denotes the syntax tree for *a* — 4 + c.

P**RODUCTION**	**SEMANTIC** R**U L ES**
1) *E→T E'*	*E.node = E' .syn* *E'.inh = T.node*
2) *E'→+TE'$_1$*	*E'$_1$.inh* **= new** *Node('+', E' .inh,T.node)* *E' .syn = E'$_1$ .syn*
3) *E' →-T E'$_1$*	*E'$_1$.inh* **= new** *Node('-',E'.inh,T.node)* *E' .syn = E'$_1$.syn*
4) *E' → ε*	*.syn = E'.inh*
5) *T→(E)*	*T.node = E.node*
6) T → **i d**	*T.node* = **new** *Leaf (id, id. entry)*
7) *T* → **num**	*T.node* = **new** Leaf**(num,** *num.val)*

Constructing syntax trees during top-down parsing

E 13 node



Dependency graph for *a* - 4 + c, with the SDD

Since there is no more input, at node 9, *E'.inh* points to the root of the entire syntax tree. The *syn* attributes pass this value back up the parse tree until it becomes the value of *E.node.* Specifically, the attribute value at node 10 is defined by the rule *E'.syn = E'.inh* associated with the production *E'* —> *ε* . The attribute value at node 11 is defined by the rule *E'.syn = E'*$_1$ *.syn* associated with production 2. Similar rules define the attribute values at nodes 12 and 1 3 .

THE STRUCTURE OF A TYPE

Inherited attributes are useful when the **structure of the parse tree differs from the abstract syntax of the input**; attributes can then be used to carry **information from one part of the parse tree to another**. The next example shows how a mismatch in structure can be due to the design of the language, and not due to constraints imposed by the parsing method.

Example: In C, the type **int** [2][3] can be read as, **"array of 2 arrays of 3 integers."** The corresponding type expression ***array(2,* array(3, *integer))*** is represented by the tree. The operator ***array*** takes two parameters, **a number and a type**. If types are represented by **trees**, then this operator returns a tree node labeled ***array*** with two children for a number and a type.

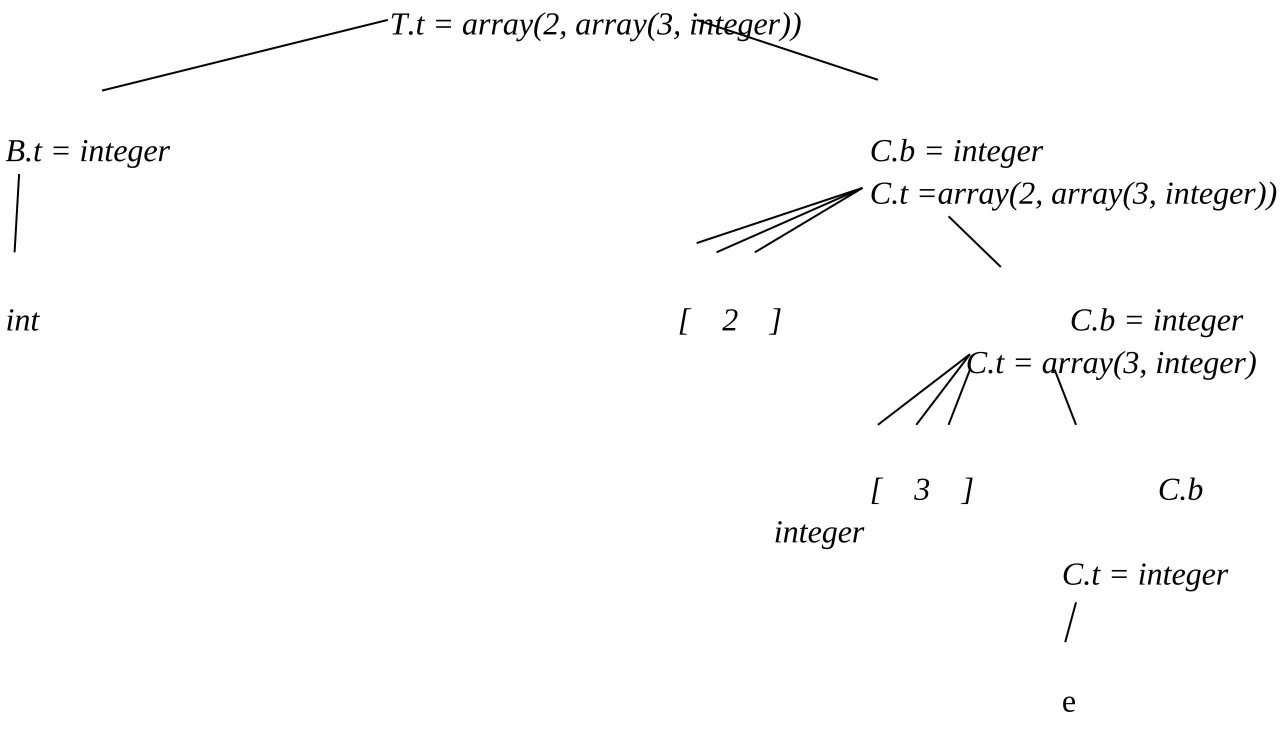
With the SDD, non-terminal *T* generates either a basic type or an array type. Non-terminal *B* generates one of the basic types **int** and **float.** *T* generates a basic type when *T* derives *B C* and *C* derives ε. Otherwise, *C* generates array components consisting of a sequence of integers, each integer surrounded by brackets.

P R O D U C T I O N	**SEMANTIC RULES**
T →*B C*	*T.t = C.t*
	C.b = B.t
B →int	*B.t = integer*

$B \rightarrow$ float	$B.t = float$
$C \rightarrow$[num] C_1	$C.t = array(num.val, C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \varepsilon$	$C.t = C.b$

***T* generates either a basic type or an array type**

The non-terminals *B* and *T* have a synthesized attribute *t* representing a type. The non-terminal *C* has two attributes: an inherited attribute *b* and a synthesized attribute *t*. The inherited *b* attributes pass a basic type down the tree, and the synthesized *t* attributes accumulate the result.
The corresponding type expression is constructed by passing the type *integer* from *B,* down the chain of C's through the inherited attributes *b.* The array type is synthesized up the chain of C's through the attributes *t.* In more detail, at the root for *T* $\rightarrow$ *B C,* non-terminal *C* inherits the type from *B,* using the inherited attribute *C.b.* At the rightmost node for C, the production is *C* e, so *C.t* equals C.6. The semantic rules for the production *C* **[n um]** C_1 form C.t by applying the operator *array* to the operands **num**.val and $C_1.t$.



Syntax-directed translation of array types

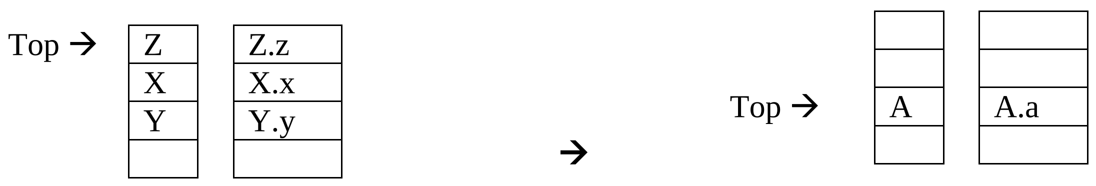
3. Explain in detail about Bottom-Up Evaluation of S-Attributed Definitions.

The S-attributed definitions, that is, the syntax-directed definitions with **only synthesized attributes**. We put the values of the synthesized attributes of the **grammar symbols into a parallel stack**. When an entry of the parser stack holds a grammar symbol X (terminal or non-terminal), the corresponding entry in the parallel stack will hold the synthesized attribute(s) of the symb lX o X.

• We evaluate the values of the attributes during reductions.

A → XYZ A.a=f(X.x,Y.y,Z.z) where all attributes are synthesized. stack parallel-stack

Stack parallel stack



Example:

PRODUCTION	SEMANTIC RULES
L → E \n	print val[top]
E → E1 + T	val[ntop] = val[top-2] + val[top]
E → T	
T → T ∗ F	val[ntop] = val[top-2] * val[top]
T → F	
F → (E)	val[ntop] = val[top-1]
F → num	

- At each shift of **digit**, we also push **digit.lexval** into *val-stack.*

• At all other shifts, we do not put anything into *val-stack* because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).

Bottom-Up Evaluation -- Example

The following Figure shows the moves made by the parser on input 3*5+4n.

– Stack states are replaced by their corresponding grammar symbol;

– Instead of the

Top-Down Translation

• We will look at the implementation of L-attributed definitions during predictive parsing.

• Instead of the syntax-directed translations, we will work with translation schemes.

• We will see how to evaluate inherited attributes (in L-attributed definitions) during recursive predictive parsing.

• We will also look at what happens to attributes during the left-recursion elimination in the left-recursive grammars.

3*5+4n	-	-	
*5+4n	3	3	
*5+4n	F	3	F → digit
*5+4n	T	3	T→ F
5+4n	T*	3-	
+4n	T*5	3-5	
+4n	T*F	3-5	F → digit
+4n	T	15	T → T * F
+4n	E	15	E → T
4n	E+	15-	
n	E+4	15-4	
n	E+F	15-4	F → digit
n	E+T	15-4	T→ F
n	E	19	E →E + T
	E n	19-	
	L	19	L → E n

Eliminating Left Recursion from Translation Scheme

• A translation scheme with a left recursive grammar.

E → E1 + T { E.val = E1.val + T.val }

E → E1 - T { E.val = E1.val - T.val }

E → T { E.val = T.val }

T → T1 * F { T.val = T1.val * F.val }

T → F { T.val = F.val }

F → (E) { F.val = E.val }

F → **digit** { F.val = **digit**.lexval }

• When we eliminate the left recursion from the grammar (to get a suitable grammar for the top-down parsing) we also have to change semantic actions.

E → T { R.i=T.nptr }

    R { E. nptr = R.syn }

R → +

    T { R1.i=mknode('+',R.i,T.nptr) }

    R1 { R.s = R1A1.s}

R → -

    T { R1.i= mknode('-',R.i,T.nptr)}

    R1 { R.s = R1.s}

R→ ε { R.s = R.i }

F → (

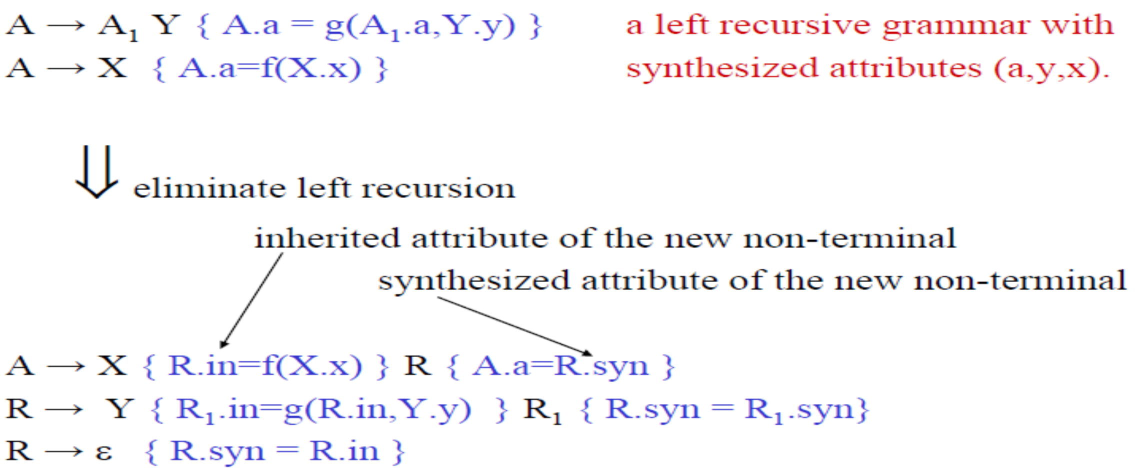
    E

) { T.nptr = ET.nptr }

F → id { T.nptr :=mkleaf(id,id,entry)}

T → num {T.nptr :=mkleaf(num,num,val)}

Eliminating Left Recursion (in general)



$A \rightarrow A_1\ Y\ \{\ A.a = g(A_1.a, Y.y)\ \}$ a left recursive grammar with

$A \rightarrow X\ \{\ A.a = f(X.x)\ \}$ synthesized attributes (a,y,x).

⇓ eliminate left recursion

inherited attribute of the new non-terminal

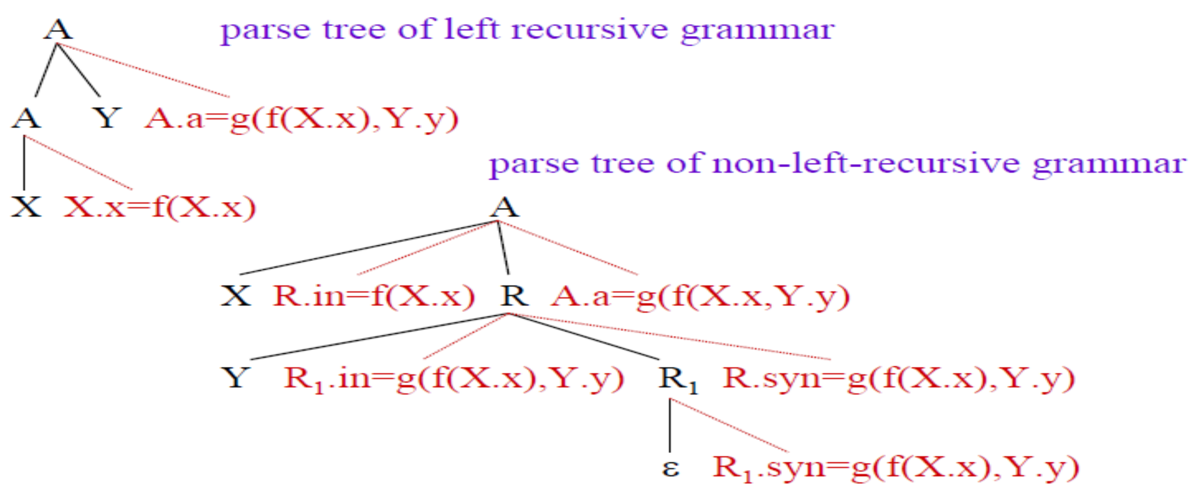
synthesized attribute of the new non-terminal

$A \rightarrow X\ \{\ R.in = f(X.x)\ \}\ R\ \{\ A.a = R.syn\ \}$

$R \rightarrow\ Y\ \{\ R_1.in = g(R.in, Y.y)\ \}\ R_1\ \{\ R.syn = R_1.syn \}$

$R \rightarrow \varepsilon\ \ \{\ R.syn = R.in\ \}$

Evaluating attributes



Translation Scheme - Intermediate Code Generation

E → T { A.in=T.loc } A { E.loc=A.loc }

A → + T { A1.in=newtemp(); emit(add,A.in,T.loc,A1.in) }

A1 { A.loc = A1.loc}

A → ε { A.loc = A.in }

T → F { B.in=F.loc } B { T.loc=B.loc }

B → * F { B1.in=newtemp(); emit(mult,B.in,F.loc,B1.in) }

B1 { B.loc = B1.loc}

B → ε { B.loc = B.in }

F → (E) { F.loc = E.loc }

F → **id** { F.loc = **id**.name }

4. Discuss about Design of Predictive translation.

(or)

Construction of a predictive syntax-directed translator.

Input: translation scheme based on a grammar suitable for predictive parsing

Output: Code for a syntax-directed translator

Method:

1. For each non-terminal A, construct a function with Input parameters: one for each inherited attribute of A; Return value: synthesized attributes of A; Local variables: one for each attribute of each grammar symbol that appears in a production for A.

2. Code for non-terminal A decides what production to use based on the current input symbol (switch statement). Code for each production forms one case of a switch statement.

3. In the code for a production, tokens, nonterminals, actions in the RHS are considered left to right.

(i) For token X: save X.s in the variable created for X; generate a call to match X and advance input.

(ii) For nonterminal B: generate an assignment c = B(b1, b2, ..., bk); where: b1, b2, ... are variables corresponding to inherited attributes of B, c is the variable for synthesized attribute of B, B is the function created for B.

(iii) For an action, copy the code into the function, replacing each reference to an attribute by the variable created for that attribute

Example:

```
E → T { R.i=T.nptr }
        R { E. nptr = R.syn }
R → +
        T { R1.i=mknode('+',R.i,T.nptr) }
        R1 { R.s = R1A1.s}
R → -
        T { R1.i= mknode('-',R.i,T.nptr )}
        R1 { R.s = R1.s}
R → ε { R.s = R.i }
F → (
        E
         ) { T.nptr = ET.nptr }
F → id { T.nptr :=mkleaf(id,id,entry)}
T → num {T.nptr :=mkleaf(num,num,val)}
```

Above grammar obtain types for the arguments and results of the functions for E,R.T.Since E and T do not have inherited attributes, they have no arguments.

Function E:↑syntax_tree_node;

Function R(i:↑ syntax_tree_node): syntax_tree_node;

Function T:↑syntax_tree_node;

We combine two of the productions above to make translator smaller.
The new productions use token addop to represent + and -:

```
R → addop
        T { R1.i= mknode(addop,lexeme,R.i,T.nptr )}
        R1 { R.s = R1.s}
A → ε { R.s = R.i }
```

<u>Parsing procedure for the productions R→ addop T R | ε</u>

```
procedure R;
begin
        if lookahead = addop then begin
                match(addop);T;R
```

```
          end
          else begin
          end
end;
```

Recursive descent construction of syntax tress.

```
function R (i: syntax_tree_node): syntax_tree_node;
          var nptr, il,sl,s: syntax_tree_node
                    addoplexeme:char;
begin
          if lookahead=addop then begin
                    addoplexeme := lexval;
                    match(addop);
                    nptr ;=T;
                    il:=mknode(addoplexeme,i,nptr);
                    sl :=R(il);
                    s:=sl;
          end
          else s:=I;
          return s
end;
```

5. Explain the details about Type Checking with necessary diagram.(Nov/Dec 2016)

- A compiler must check that the source program follows both syntactic and semantic conventions of the source language.
- This checking, called *static checking*, detects and reports programming errors.
- **Over View**
    - **Type system**
        - Type expressions
    - **Specification of a simple type checker**
        - A simple language
        - Type checking of expression
        - Type checking of statement
        - Type checking of functions
- Some examples of static checks:
    1. **Type checks**
    2. **Flow-of-control checks**
    3. **Uniqueness checks**
    4. **Name –related checks-**

:

1. Type checks – A compiler should report an error if an operator is applied to an incompatible operand.

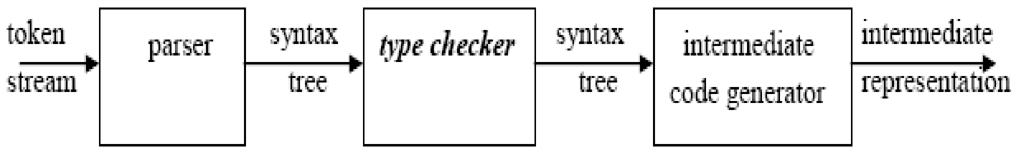
Example: If an array variable and function variable are added together.

2. Flow-of-control checks – Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control.
Example: An error occurs when an enclosing statement, such as break, does not exist in switch statement.

3. Uniqueness checks – there are situation in which an object must be defined exactly once
Example: in Pascal, an identifier must be declared uniquely, labels in a case statement must be distinct

4. Name –related checks - some times, the same name must appear two or more times
Example: in ada, a loop or block may have a name that appears at the beginning and end of the construct

Position of type checker



- A ***type checker*** verifies that the type of a construct matches that expected by its context.
- **For example** : arithmetic operator *mod* in Pascal requires integer operands, so a type checker verifies that the operands of *mod* have type integer.
- Type information gathered by a type checker may be needed when code is generated.

TYPE SYSTEMS

- The design of a type checker for a language is based on information about the syntactic
- Constructs in the language, the notion of types, and the rules for assigning types to language constructs.
  For example: " if both operands of the arithmetic operators of +,- and * are of type integer, then the result is of type integer "

Type Expressions

- The type of a language construct will be denoted by a "type expression."
- A type expression is either a basic type or is formed by applying an operator called a ***type***

Constructor to other type expressions.

The sets of basic types and constructors depend on the language to be checked.

The following are the definitions of type expressions:

- Basic types such as *boolean, char, integer, real* are type expressions . A special basic type, *type_error* , will signal an error during type checking; *void* denoting "the absence of a value" allows statements to be checked.

- Since type expressions may be named, a type name is a type expression.
- A type constructor applied to type expressions is a type expression.

Constructors include:

Arrays: If T is a type expression then *array* (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

Products: If T1 and T2 are type expressions, then their Cartesian product T1 X T2 is a type expression.

- parser ***type checker*** intermediate
- code generator

Records: The difference between a record and a product is that the fields of a record have names. The *record* type constructor will be applied to a tuple formed from field names and field types.

For example:

```
type row = record
                address: integer;
                lexeme: array[1..15] of char
            end;
var table: array[1...101] of row;
```

declares the type name *row* representing the type expression ***record((address X integer) X (lexeme X array(1..15,char)))*** and the variable *table* to be an array of records of this type.

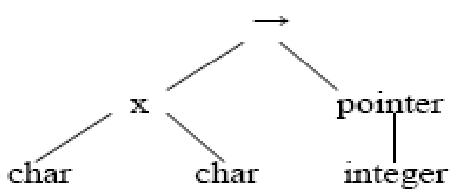
Pointers: If T is a type expression, then *pointer*(T) is a type expression denoting the type "pointer to an object of type T".

For example, ***var p: ↑ row*** declares variable p to have type ***pointer*(row)**.

Functions: A function in programming languages maps a ***domain type D*** to **a *range type R***. The type of such function is denoted by the type expression D → R

Type expressions may contain variables whose values are type expressions.

Tree representation for char x char → *pointer* (integer)



Type systems

- A ***type system*** is a collection of rules for **assigning type expressions** to the various parts of a program.
- A **type checker** implements a type system. It is specified in **a syntax-directed** manner.

- Different type systems may be used by different compilers or processors of the same language.

Static and Dynamic Checking of Types

- Checking done by a compiler is said to be static, while checking done **when the target program runs is termed dynamic.**
- Any check can be done dynamically, if the **target code carries the type of an element along with the value of that element.**
-

Sound type system

- A ***sound*** type system **eliminates the need for dynamic checking for type errors** because it allows us **to determine statically that these errors** cannot occur when the target program runs.
- That is, if a sound type system assigns a type other than ***type error*** to a program part, then type errors cannot occur when the target code for the program part is run.

Strongly typed language

- A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.

Error Recovery

- ❖ Since type checking has the potential for catching errors in program, it is desirable for type checker to recover from errors, so it can check the rest of the input.
- ❖ Error handling has to be designed into the type system right from the start; the type checking rules must be prepared to cope with errors.

6. Explain the detail about the specification of a simple type checker
(16 Marks)(May/Jun -2012) (May/Jun -2013) (May/Jun -2016)

Specify a type checker for a simple language in which the type of each identifier must be declared before the identifier is used. The type checker is a translation scheme that synthesizes the type of each expression from the types of its sub expressions. The type checker can handle arrays, pointers, statements and functions.

- A simple language
- Type checking of expression
- Type checking of statement
- Type checking of functions

A Simple Language

Consider the following grammar:
P → D ; E
D → D ; D | id : T
T → char | integer | array [num] of T | ↑ T
E → literal | num | id | E mod E | E [E] | E ↑

Translation scheme:

P → D ; E	
D → D ; D	
D → id : T	{ *addtype* (id.*entry* , T.*type*) }
T → char	{ T.*type* : = char }
T → integer	{ T.*type* : = integer }
T → ↑ T1	{ T.*type* : = pointer(T1.*type*) }
T → array [num] of T1	{ T.*type* : = array (1… num.val , T1.*type*) }

In the above language,
→ There are two basic types : char and integer ;
→ *type_error* is used to signal errors;
→ the prefix operator ↑ builds a pointer type. Example , ↑ **integer** leads to the type expression **pointer (integer)**.

Type checking of expressions

In the following rules, the attribute *type* for E gives the type expression assigned to the expression generated by E.

1. E → **literal** { E.*type* : = *char* }
E → **num** { E.*type* : = *integer* }

Here, constants represented by the tokens **literal** and **num** have type *char* and *integer*.

2. E → **id** { E.*type* : = *lookup* (**id**.*entry*) }
lookup (e) is used to fetch the type saved in the symbol table entry pointed to by e.

```
3. E → E1 mod E2 { E.type : = if E1. type = integer and
                   E2. type = integer then integer
                   else type_error }
```
The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is *type_error*.

```
4. E → E1 [ E2 ] { E.type : = if E2.type = integer and
                   E1.type = array(s,t) then t
                   else type_error }
```

In an array reference E1 [E2] , the index expression E2 must have type integer. The result is the element type *t* obtained from the type *array(s,t)* of E1.

```
5. E → E1 ↑ { E.type : = if E1.type = pointer (t) then t
              else type_error }
```

The postfix operator ↑ yields the object pointed to by its operand. The type of E ↑ is the type *t* of the object pointed to by the pointer E.

Type checking of statements

Statements do not have values; hence the basic type *void* can be assigned to them. If an error is detected within a statement, then *type_error* is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement:

S → **id** : = E { S.*type* : = **if id**.*type* = E.*type* **then** *void*
else *type_error* }

2. Conditional statement:

S → **if** E **then** S1 { S.*type* : = **if** E.*type* = *boolean* **then** S1.*type*
else *type_error* }

3. While statement:

S → while E do S1 { S.*type* : = **if** E.*type* = *boolean* **then** S1.*type*
else *type_error* }

4. Sequence of statements:

S → S1 ; S2 { S.*type* : = **if** S1.*type* = *void* and
S1.*type* = *void* **then** *void*
else *type_error* }

Type checking of functions

The rule for checking the type of a function application is :

E → E1 (E2) { E.*type* : = **if** E2.*type* **=** *s* **and**
E1.*type* **=** *s* → *t* **then** *t*
else *type_error* }

7. Explain the run time environment

- The **allocation and de-allocation of data objects** is managed by the **runtime support package**, consisting of **routines loaded with the generated target code**
- The design of the runtime support package is influenced by the **semantics of procedures**
- Support packages for language like **Fortran , Pascal and Lisp** can be constructed using the techniques
- Each execution of a procedures is referred to as an activation of the procedure
- If the procedure is recursive , several of its activation may be alive at the same time

- **Source Language Issues**
- **Storage Organization**
- **Storage Allocation Strategies**
-

Source Language Issues

For specificity, suppose that a program is made up of procedure**,**

- **Procedures:**
- **Activation trees:**
- **Control stack:**
- **The Scope of a Declaration:**
- **Binding of names:**

Procedures:

A procedure definition is a declaration that associates an identifier with a statement. The Identifier is the procedure name, and the statement is the procedure body.

For example, the following is the definition of procedure named readarray :

```
Procedure readarray;
        var i : integer;
        begin
        for i : = 1 to 9 do read(a[i])
        end;
```

When a procedure name appears within an executable statement, the procedure is said to be *called* at that point.

Activation trees:

An activation tree is used to depict **the way control enters and leaves activations**. In an activation tree,

1. **Each node** represents an **activation of a procedure**.
2. The **root** represents the **activation of the main program**.
3. **The node for *a*** is the parent of the node for *b* if and only if control flows from activation *a* to *b*.
4. The node for *a* is to the left of the node for *b* if and only if the lifetime of *a* occurs before the lifetime of *b*.

Control stack:

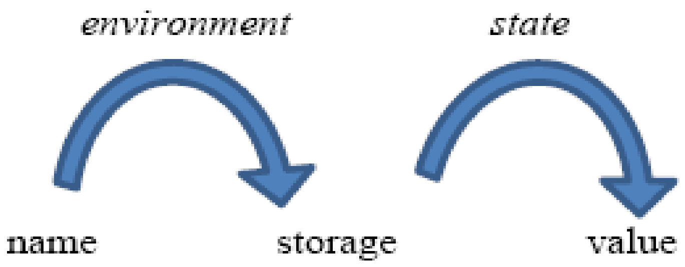
- A ***control stack*** is used to keep track of **live procedure activations**. The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.
- The contents of the control stack are related to paths to the **root of the activation tree**. When node *n* is at the top of control stack, the stack contains the nodes along the path from *n* to the root.

The Scope of a Declaration:

- A declaration is a syntactic construct that associates information with a name.
- Declarations may be explicit, such as:
    - **var i : integer ;**
- or they may be implicit. Example, any variable name starting with I is assumed to denote an integer. The portion of the program to which a declaration applies is called the ***scope*** of that declaration.

Binding of names:

- Even if each name is declared once in a program, the same name may denote different data objects at run time. **“Data object”** corresponds to a storage location that holds values.
- The term ***environment*** refers to a function that **maps a name to a storage location**.
- The term ***state*** refers to a function that **maps a storage location to the value held there**.

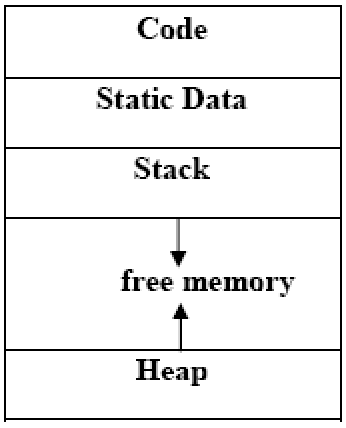


- When an ***environment*** associates storage location *s* with a name *x*, we say that *x* is *bound* to *s*. This association is referred to as a *binding* of *x*.

Storage Organization

- The executing target program runs in its **own logical address space** in which each program value has a location.
- The management and organization of this logical address space is **shared between the complier, operating system and target machine**. The operating system maps the logical address into physical addresses, which are usually spread throughout memory.

Typical subdivision of run-time memory:

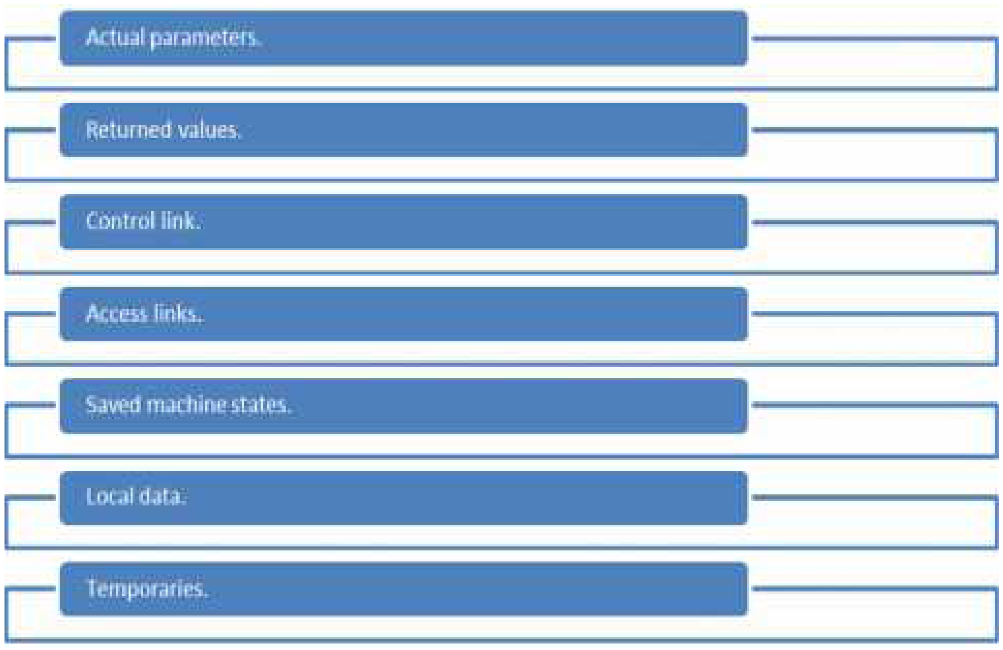


- Run-time storage comes in blocks, where a **byte** is the **smallest unit of addressable memory**. **Four bytes** form **a machine word. Multibyte objects** are stored in **consecutive bytes** and given the **address of first byte.**
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.

- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static. The dynamic areas used to maximize the utilization of space at run time are stack and heap.

Activation records:

- Procedure calls and returns are usually managed by a run time stack called **the control stack**. Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom; the latter activation has its record at the top of the stack.
- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.



- Temporary values such as those arising from the evaluation of expressions.
- Local data belonging to the procedure whose activation record this is.
- A saved machine status, with information about the state of the machine just before the call to procedures.
- An access link may be needed to locate data needed by the called procedure but found elsewhere.
- A control link pointing to the activation record of the caller.

- Space for the return value of the called functions, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
- The actual parameters used by the calling procedure. These are not placed in activation record but rather in registers, when possible, for greater efficiency.

8. What are the Storage Allocation Strategies? Explain (16 Marks) (Nov /Dec 2011)(May/June-2013)

Or

Discuss the various storage allocation strategies in details(8 Marks)(Apr/May 2011) (May/June 2016)

The different storage allocation strategies are :

1. **Static allocation** – lays out storage for all data objects at compile time

Dynamic Storage allocation

2. **Stack allocation** – manages the run-time storage as a stack.
3. **Heap allocation** – allocates and deallocates storage as needed at run time from a data area known as heap.

STATIC ALLOCATION

- In static allocation, names are **bound to storage** as the program is compiled, so there is **no need for a run-time support package.**
- Since the bindings do not change at run-time, every time a procedure is activated, its names are bound to the same storage locations.
- Therefore values of local names are ***retained*** across activations of a procedure. That is,
- When control returns to a procedure the values of the locals are the same as they were when control left the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

STACK ALLOCATION OF SPACE

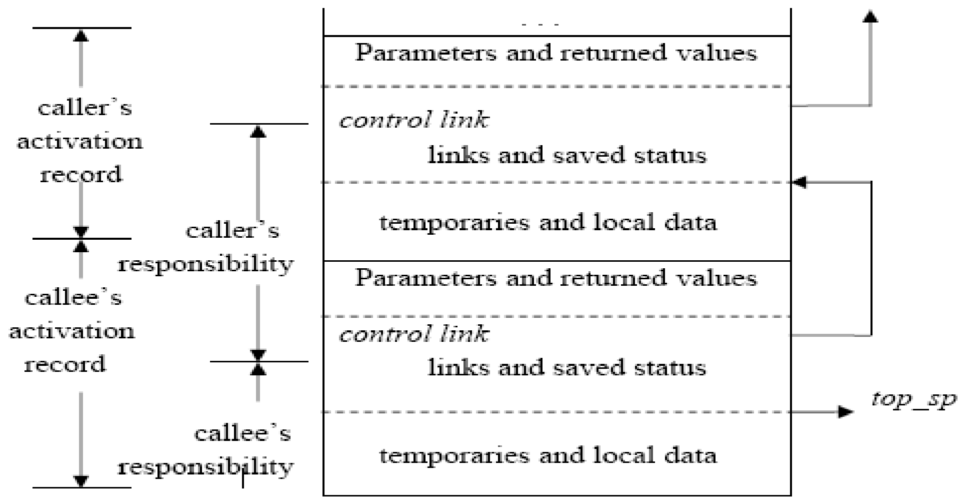
- All compilers for languages that use procedures, functions or methods as units of user defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

Calling sequences:

- Procedures called are implemented in what is called as calling sequence, which consists of code that **allocates an activation record on the stack** and enters information into its fields.
- A return sequence is similar to code to restore the state of machine so the calling procedure can continue its execution after the call.
- The code in calling sequence is often divided between the **calling procedure (caller) and the procedure it calls (callee).**
- When designing calling sequences and the layout of activation records, the following principles are helpful:

  Values communicated between **caller and callee** are generally placed at the **beginning of the callee's activation record,** so they are as **close as possible to the caller's activation record**

- **Fixed length items** are going nearly placed in the **middle**. Such items typically include the **control link, the access link,** and the machine status fields.
- Items whose size may not be known early enough are placed at the end of the activation record. The most common example is dynamically sized array, where **the value of one of the callee's parameters determines the length of the array.**
- We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of fixed-length fields in the activation record. Fixed-length data can then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer



...
Parameters and returned values
control link
links and saved status
temporaries and local data
Parameters and returned values
control link
links and saved status
temporaries and local data
caller's activation record
callee's activation record
caller's responsibility
callee's responsibility
top_sp

Division of tasks between caller and callee

- The calling sequence and its division **between caller and callee** are as follows.
- The **caller** evaluates the **actual parameters**.

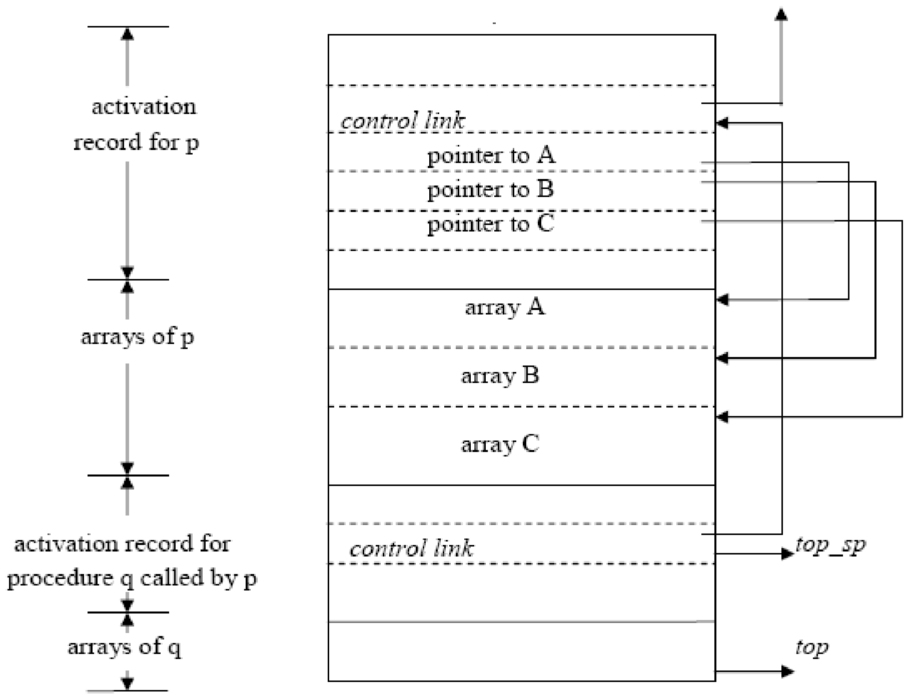
- The **caller** stores a **return address** and the **old value** of ***top_sp*** into the callee's activation record. The caller then **increments the *top_sp*** to the respective positions.
- The **callee** saves the **register values** and **other status information** .
- The **callee** initializes its **local data and begins execution.**

A suitable, corresponding return sequence is:

- The **callee** places the return value next to the parameters.
- Using the information in the machine-status field, the **callee** restores ***top_sp*** and other registers, and then branches to the return address that the caller placed in the status field.
- Although ***top_sp*** has been decremented, the caller knows where the return value is, relative to the current value of ***top_sp*;** the caller therefore may use that value.

Variable length data on stack:

- The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.
- The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space.
- The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.



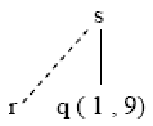
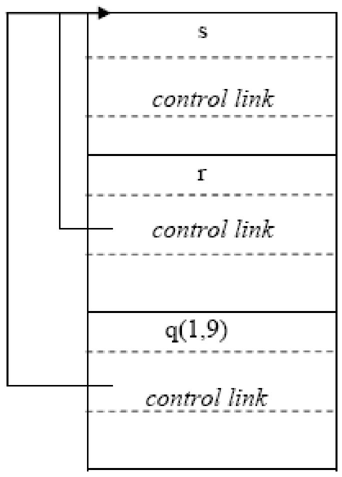
Access to dynamically allocated arrays

- Procedure p has three local arrays, whose sizes cannot be determined at compile time. The storage for these arrays is not part of the activation record for p.
- Access to the data is through two pointers***, top* and *top-sp***. Here the ***top*** marks **the actual top of stack**; it points the position at which the **next activation record will begin**.
- The second ***top-sp*** is used to find local, fixed-length fields of the top activation record.
- The code to reposition ***top* and *top-sp*** can be generated at compile time, in terms of sizes that will become known at run time.

HEAP ALLOCATION
Stack allocation strategy cannot be used if either of the following is possible:

1. **The values of local names must be retained when activation ends.**
2. **A called activation outlives the caller.**

- Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
- Pieces may be de-allocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

Position in the activation tree	Activation records in the heap	Remarks
	 s control link r control link q(1,9) control link	Retained activation record for r

- The record for an activation of procedure **r** is retained when the activation ends.
- Therefore, the record for the new activation **q(1 , 9)** cannot follow that for s physically.
- If the retained activation record for r is de-allocated, there will be free space in the heap between the activation records for s and q.

9. Distinguish between the source test of a procedure and its activation at run time (8 Marks)(Apr/May 2011)

Procedures:
A ***procedure definition*** is a declaration that associates an identifier with a statement. The identifier is the ***procedure name***, and the statement is the ***procedure body.***

For example, the following is the definition of procedure named ***readarray*** :

```
procedure readarray;
        var i : integer;
        begin
        for i : = 1 to 9 do read(a[i])
        end;
```

When a procedure name appears within an executable statement, the procedure is said to be ***called*** at that point.

Activation trees:

An ***activation tree*** is used to depict the way control enters and leaves activations. In an activation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for *a* is the parent of the node for *b* if and only if control flows from activation *a* to *b*.
4. The node for *a* is to the left of the node for *b* if and only if the lifetime of *a* occurs before the lifetime of *b*.

Parameter Passing

The communication medium among procedures is known as **parameter passing**. The values of the variables **from a calling procedure are transferred to the called procedure** by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

r-value

The **value of an expression** is called its **r-value**. The value contained in a **single variable** also becomes **an r-value** if it appears on the **right-hand side of the assignment operator**. **r-values** can always be assigned to **some other variable.**

l-value

The **location of memory (address)** where an expression is stored is known as **the l-value** of that expression. It always appears at the left hand side of an assignment operator. For example:

```
day = 1;
 week = day * 7;
 month = 1;
 year = month * 12;
```

From this example, we understand that constant values like 1, 7, 12, and variables like day, week, month, and year, all have r-values. Only variables have l-values, as they also represent the memory location assigned to them.

For example: 7 = x + y;

is an l-value error, as the constant 7 does not represent any memory location.

Formal Parameters

Variables that take the **information passed by the caller procedure** are called **formal parameters**. These variables are declared in the definition of the called function.

Actual Parameters

Variables **whose values or addresses** are being **passed** to the **called procedure** are called **actual parameters**. These variables are specified in the function call as arguments.

Example:

```
 fun_one()
{
 int actual_parameter = 10;
 call fun_two(int actual_parameter);
 }
 fun_two(int formal_parameter)
 {
 print formal_parameter;
}
```

Formal parameters hold the information of the actual parameter, depending upon the parameter passing technique used. It may be a value or an address.

Pass by Value

In pass by value mechanism, the calling procedure passes the **r-value of actual parameters** and the **compiler** puts that into the **called procedure's activation record**. Formal parameters then **hold the values passed** by the **calling procedure**. If the values held by the formal parameters are **changed**, it should have **no impact** on the **actual parameters**.

Pass by Reference

In pass by reference mechanism, the **l-value of the actual parameter** is copied to the **activation record** of the called procedure. This way, the called procedure now has the address (memory location) of the actual parameter and the formal parameter refers to the same memory location. Therefore, if the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter, as they should also point to the same value.

Pass by Copy-restore

This parameter passing mechanism works similar to **'pass-by-reference'** except that the changes to actual parameters are made when the called procedure ends. Upon function call, the values of actual parameters are copied in the activation record of the called procedure. Formal parameters, if manipulated, have no real-time effect on actual parameters (as l-values are passed), but when the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters.

Example:

```
 int y;
 calling_procedure()
{
y = 10;
copy_restore(y); //l-value of y is passed
printf y; //prints 99
}
opy_restore(int x)
{
x = 99; // y still has value 10 (unaffected)
 y = 0; // y is now 0
}
```

When this function ends, the l-value of formal parameter x is copied to the actual parameter y. Even if the value of y is changed before the procedure ends, the l-value of x is copied to the l-value of y, making it behave like call by reference.

Pass by Name

Languages like Algol provide a new kind of parameter passing mechanism that works like preprocessor in C language. In pass by name mechanism, the name of the procedure being called is replaced by its actual body. Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so that it can now work on actual parameters, much like pass-by-reference.

10. Discuss about source language issues.

<u>Procedures</u>

A procedure definition is a declaration that, in its simplest form, associates an identifier with a statement. The identifier is the procedure name, and the statement is the procedure body. Procedures that return values are called function in many languages; however, it is convenient to refer them as procedures. A complete will also be treated as a procedure.

```
procedure readarray
var i: integer;
begin
    for i=1 to 9 do read(a[i])
end;
```

When a procedure name appears within an executable statement, we say that the procedure is called at that point. The basic idea is that a procedure call executes the procedure body.

Some of the identifiers appearing in a procedure definition are special, and are called formal parameters (or just formals) of the procedure. Arguments known as actual parameters may be passed to a called procedure they are substituted for the formals in the body.

Activation Trees

We make the following assumptions about the flow of control among procedure during the execution of a program:

1. Control flows sequentially, that is, the execution of a program consists of a sequence of steps, with control being at some point in the program at each step.
2. Each execution of a procedure starts at the beginning of the procedure body and eventually returns control to the point immediately following the place where the procedure was called. This means the flow of control between procedures can be depicted using trees.

Each execution of a procedure body is referred to as an activation of the procedure. The lifetime of an activation of a procedure p is the sequence of steps between the first and last steps in the execution of the procedure called by them and so on. In general the term “lifetime” refers to a consecutive sequence of steps during the execution of a program.

If a, b are procedure then their lifetimes are either non-overlapping or are nested. That is if b is entered before a, is left then control must leave b before it leaves a. this nested property of activation lifetime can be illustrated by inserting two print statements in each procedure one before the first statement of the procedure body and the other after the last. The first statement prints enter followed by the name of the procedure and the values of the actual parameters; the last statement prints leave followed by the same information.

A procedure is recursive if a new activation can begin before an earlier activation of the same procedure has ended. A recursive procedure need not call itself directly; p may call another procedure q, which may then call p through some sequence of procedure calls. We can use tree, called an activation tree, to depict the way control enters and leaves activations. In an activation tree,

1. Each node represents an activation of a procedure,

2. The root represents the activation of the main program,

3. The node for a is the parent of the node for b if and only if the control flows from activation a to b, and

4. The node for a, is to the left of the node for b if and only if the lifetime of a, occurs before the lifetime of b.

Since each node represents a unique activation and vice versa it is convenient to talk of control being at a node it is in the activation represented by the node.

Control Stacks

the flow of control in a program corresponds to a depth-first traversal of the activation tree that starts at the root ,visits a node before its children, and recursively visits children at each node left to right order. the output in fig 7.2 can therefore be reconstructed by traversing the activation tree in fig7.3,printing enter when the node for an activation is reaches for the first time and printing leave after the entire sub tree of the node has been visited during the traversal.

We can use a stack, called a control stack to keep track of live procedure activations. The idea is to push the node for activation onto the control stack as the activation begins and to pop the node when the activation ends.

Then the contents of the control stack are related to the paths to the root f the activation tree. When the node n is at the top of the control stack, the stack contains the nodes along the path from n to the root.

Example 7.2:fig 7.4 shows nodes from the activation tree of fig 7.3 that that have been reached when control enters the activation represented by q(2,3).Activations with labels r, p(1,9),p(1.3),and q(1,0) have executed to completion, so the figure contains dashed lines to their nodes. The solid lines mark the path from q (2, 3) to the root.

At this point the control stack contains the following nodes along this path to the root (the top of the stack is to the right)

s, q(1,9),q(1,3),q(2,3) and the other nodes.

The Scope of a Declaration

A declaration in a language is a syntactic construct that associates information with a name. Declarations may be explicit, as in the Pascal fragment

Var i : integer;

Or they may be explicit. For example, any variable name starting with I is or assumed to denote an integer in a FORTRAN program unless otherwise declared.

There may be independent declarations of the same name in the different parts of the program. The scope rules of a language determine which declaration of a name applies when the name appears in the text of a program.

The portion of the program to which a declaration applies is called the scope of the declaration applies is called the scope of the declaration .An occurrence of a name in a procedure is called to be local to the procedure if it is the scope of a declaration within the procedure; otherwise, the occurrence is said to be non-local.

The distinction between local and the non-local names carries over to any syntactic construct that can have declarations within it.

While the scope is a property of the declaration of a name, it is sometimes convenient to use the abbreviation “the scope of a name x” for “the scope of the declaration of name x that applies to this occurrence of x”. In this sense, the scope of ion line 17 is the body of quick sort. At compile time, the symbol table can be to find the declaration that applies to an occurrence of a name. When a declaration is seen, a symbol table entry is created for it. As long as we are in the scope of the declaration, its entry is returned when the name in it is looked up.

11. Explain in detail about symbol tables.

Symbol tables are **data structures** that are used by **compilers to hold information about source-program constructs.**

The information is **collected incrementally** by the analysis phases of a compiler and **used** by the **synthesis phases** to generate the target code. Entries in the symbol table contain information about an identifier such as its character string (or lexeme), its type, its position in storage, and any other relevant information.

Symbol tables typically need to support multiple declarations of the same identifier within a program.

The scope of a declaration is implemented by setting up a separate symbol table for each scope. A program block with declarations will have its own symbol table with an entry for each declaration in the block.

Symbol-Table Entries

Symbol-table entries are created and used during the analysis phase by the lexical analyzer, the parser, and the semantic analyzer.

In some cases, a lexical analyzer can create a symbol-table entry as soon as it sees the characters that make up a lexeme. More often, the lexical analyzer can only return to the parser a token, say id, along with a pointer to the lexeme. Only the parser, however, can decide whether to use a previously created symbol-table entry or create a new one for the identifier.

Symbol Table Per Scope

The term "**scope of identifier really refers to the scope of a particular declaration of x.** The term scope by itself refers to a portion of a program that is the scope of one or more declarations. Scopes are important, because the same identifier can be declared for different purposes in different parts of a program. Common names like i and x often have multiple uses.

As another example, subclasses can redeclare a method name to override a method in a superclass. If blocks can be nested, several declarations of the same identifier can appear within a single block. The following syntax results in nested blocks when stmts can generate a block:

block -> '{' decls stmts '}'

Optimization of Symbol Tables for Blocks

Implementations of symbol tables for blocks can take advantage of the most-closely nested rule. Nesting ensures that the chain of applicable symbol tables forms a stack. At the top of the stack is the table for the current block. Below it in the stack are the tables for the enclosing blocks.

Thus, symbol tables can be allocated and deallocated in a stacklike fashion. Some compilers maintain a single hash table of accessible entries; that is, of entries that are not hidden by a declaration in a nested block. Such a hash table supports essentially constant-time lookups, at the expense of inserting and deleting entries on block entry and exit.

Upon exit from a block B, the compiler must undo any changes to the hash table due to declarations in block B. It can do so by using an auxiliary stack to keep track of changes to the hash table while block B is processed.

Moreover, a statement can be a block, so our language allows nested blocks, where an identifier can be redeclared. The most-closely nested rule for blocks is that an identifier x is in the scope of the most-closely nested declaration of x; that is, the declaration of x found by examining blocks inside-out, starting with the block in which x appears.

Example: The following pseudocode uses subscripts to distinguish among distinct declarations of the same identifier:

```
   1)  { int xi; int yx; 2)
   2)  { int w2; bool y2;
   3)   int z2; 3) • • • w2 ••;
   4)  ••• xi y2 z2 •••;
   5)   }
   6)   • • • w0 • • •;
   7)   ••• Xi ••• yi • • •;
   8)  }
}
```

The subscript is not part of an identifier; it is in fact the line number of the declaration that applies to the identifier. Thus, all occurrences of x are within the scope of the declaration on line 1. The occurrence of y on line 3 is in the scope of the declaration of y on line 2 since y is redeclared within the inner block. The occurrence of y on line 5, however, is within the scope of the declaration of y on line 1. The occurrence of w on line 5 is presumably within the scope of a declaration of w outside this program fragment; its subscript 0 denotes a declaration that is global or external to this block. Finally, z is declared and used within the nested block, but cannot be used on line 5, since the nested declaration applies only to the nested block. •

The most-closely nested rule for blocks can be implemented by chaining symbol tables. That is, the table for a nested block points to the table for its enclosing block.

The Use of Symbol Tables

In effect, the role of a symbol table is to pass information from declarations to uses.

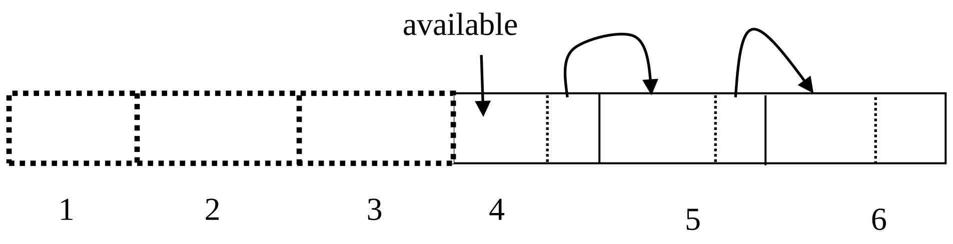
A semantic action "puts" information about identifier x into the symbol table, when the declaration of x is analyzed. Subsequently, a semantic action associated with a production such as factor ->• id "gets" information about the identifier from the symbol table. Since the translation of an expression E1 op E2, for a typical operator op, depends only on the translations of E1 and E2, and does not directly depend on the symbol table, we can add any number of operators without changing the basic flow of information from declarations to uses, through the symbol table.

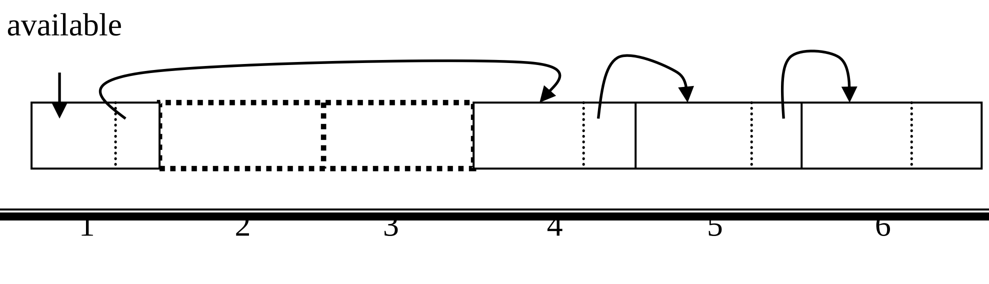
11. Explain in detail about dynamic storage allocation techniques

The techniques needed to implement dynamic storage allocation depend on how storage is deallocated. If deallocation is implicit, then the run-time support package is responsible for determining when a storage block is no longer needed. There *is* less a compiler has to do if deallocation is done explicitly by the programmer. We consider explicit deallocation first.

Explicit Allocation of Fixed-Sized Blocks

The simplest form of dynamic allocation involves blocks of a fixed size. By linking the blocks in a list, allocation and deallocation can be done quickly with little or no storage overhead.





A de-allocated block is added to the list of available blocks.

Suppose that blocks are to be drawn from a contiguous area of storage. Initialization of the area is done by using a portion of each block for a link to the next block.

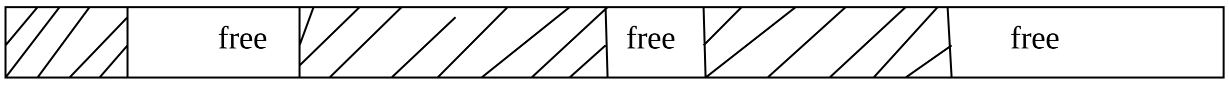
A pointer available points to the first block. Allocation consists of taking a block off the list and de-allocation consists of putting the block back on the list.

The compiler routines that manage blocks do not need to know the type of object that will be held in the block by the user program. We can treat each block as a variant record, with the compiler routines viewing the block as consisting of a link to the next block and the user program viewing the block as being of some other type.

Thus, there is no space overhead because the user program can use the entire block for its own purposes. When the block is returned, then the compiler routines use some of the space from the block itself to link it into the list of available blocks.

Explicit Allocation of Variable-Sized Blocks

When blocks are allocated and de-allocated, storage can become fragmented; that is, the heap may consist of alternate blocks that are free and in use.



Free and used blocks in a heap

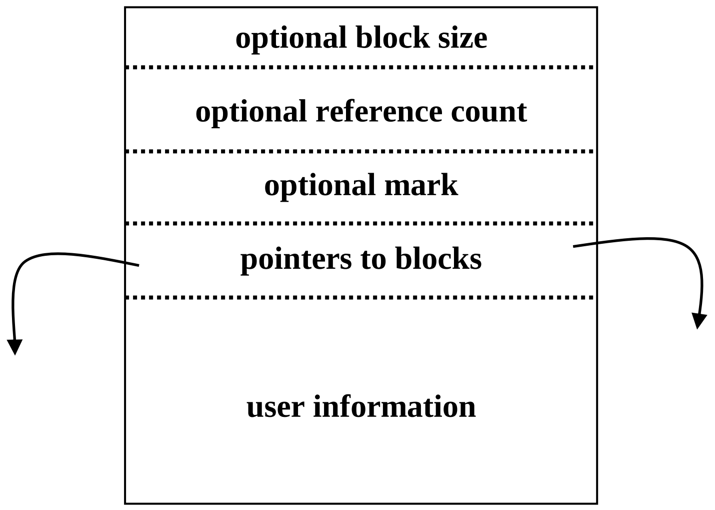
This situation can occur if a program allocates five blanks and then de-allocates the second and fourth, for example. Fragmentation is of no consequence if blocks are of fixed size, but if they are of variable size. It is a problem, because we could not allocate a block larger than any one of the free blocks, even though the space is available in principle.

One method for allocating variable-sized blocks is called the ***first-fit method***. When a block of size s is allocated, *we* search for the first free block that is of size f ≥ s. This block is then subdivided into a used block of size s, and a free block of size f -s. Note that allocation incurs a time overhead because we must search for a free block that is large enough.

When a block is de-allocated, we check to see if it is next to a free block. If possible, the de-allocated block is combined with a free block next to it to create a larger free block. Combining adjacent free blocks into a larger free block prevents further fragmentation from occurring. There are *a* number of subtle details concerning how free blocks are allocated, de-allocated, and maintained in an available list or lists. There are also several tradeoffs between time, space, and availability of large blocks.

Implicit De-allocation

Implicit deallocation requires cooperation between the user program and the run-time package, because the latter needs to know when a storage block is no longer in use. This cooperation is implemented by fixing the format of storage blocks.



The format of a block

The first problem is that of recognizing **block boundaries**. If the size of blocks is fixed, then position information can be used. For example, if each block occupies 20 words, then a new block begins every 20 words. Otherwise, in the inaccessible storage attached to a block we keep the size of a block, so we can determine where the next block begins.

The second problem is that of recognizing if a block is in use. We assume that a block is in use if it *is* possible for the user program to refer to the information in the block. The reference may occur through a pointer or after following a sequence of pointers, *so* the compiler needs to know the position in storage of all pointers. The pointers are kept in a fixed position in the block. Perhaps more to the point, the assumption is made that the user-information area *of* a block does not contain any pointers.

Two approaches can be used for implicit de-allocation.

1. **Reference counts:** We keep track of the number of blocks that point directly to the present block. If this count ever drops to 0, then the block can be deallocated because it cannot be referred to. In other words, the block has become garbage that can be collected. Maintaining reference counts can be costly in time; the pointer assignment p:=q leads to changes in the reference counts of the blocks pointed to by both p and q.

The count for the block pointed to by p goes down by one, while that for the block pointed to by q goes up by one. Reference counts are best used when pointers between blocks never appear in cycles.

2. ***Marking technique:*** An alternative approach is to suspend temporarily execution of the user program and use the frozen pointers to determine which blocks are in use. This approach requires all the pointers into the heap to be known. Conceptually, we pour paint into the heap through these pointers. Any block that is reached by the paint is in use and the rest can be deallocated. In more detail, we go through the heap and mark all blocks unused. Then, we follow pointers marking as *used* any block that is reached in the process. A final sequential scan of the heap allows all blocks still marked *unused* to be collected.

With variable-sized blocks, we have the additional possibility of moving used storage blocks from their current positions. This process, called compaction moves all used blocks to one end of the heap, so that all the free storage can be collected into one large free block. Compaction also requires information about the pointers in blocks because when a used block is moved, all pointers to it have to be adjusted to reflect the move. Its advantage is that afterwards fragmentation of available storage is eliminated.

12. Discuss about storage allocation in FORTRAN.

Fortran was designed to permit static storage allocation.

However, there are some issues, such as the treatment of **COMMON** and **EQUIVALENCE** declarations, that are fairly special to Fortran.

A Fortran compiler can create a number of *data* areas, i-e., blocks of storage in which the values of objects can be stored. In Fortran, there is one data area for each procedure and one data area for each named **COMMON** block and for blank **COMMON**, if used.

The symbol table must record for each name the data area in which it belongs and its offset in that data area, that is, its position relative to the beginning of the area.

The compiler must compute the size of each data area.

For the data areas of the procedures, a single counter suffices, since their sizes are known after each procedure is processed.

For COMMON blocks, a record for each block must be kept during the processing of all procedures, since each procedure using a block may have its own idea of how big the block is, and the actual size is the maximum of the sizes implied by the various procedures.

For each data area the compiler creates a memory *map,* which is a description of the contents of the area. This "memory map" might simply consist of an indication, in the symbol-table entry for each name in the area, of its offset in the area.

A Fortran program consists of a main program, subroutines, and functions (we call them all *procedures).* Each occurrence of a name has a scope consisting of one procedure only. We can generate object code for each procedure upon reaching the end of that procedure.

Data in COMMON Areas

We create for each block a record giving the first and last names, belonging to the current procedure, that are declared to be in that COMMON block. When processing a declaration like

COMMON /BLOCKI/ NAMEl, NAME2

the compiler must do the following.

1. In the table for COMMON block names, create a record for BLOCK1, if one does not already exist.

2. In the symbol-table entries for NAME1 and NAME2, set a pointer to the symbol-table entry for BLOCK 1, indicating that these are in COMMON and members of BLOCK1.

3. a) If the record has just now been created for BLOCK1, set a pointer in that record to the symbol-table entry for NAME1, indicating the first name in this COMMON block. Then, link the symbol-table entry for NAME1 to that for NAME2, using a field of the symbol table reserved for linking members of the same COMMON block. Finally, set a pointer in the record for BLOCK1 to the symbol-table entry for NAME2, indicating the last found member of that block.

b) If, however, this is not the first declaration of BLOCK 1, simply link NAME1 and NAME2 to the end of the list of names for BLOCK1. The pointer to the end of the list for BLOCK1, appearing in the record for BLOCK1 is updated of course.

After a procedure has been processed, we apply the equivalencing algorithm. We may discover that some additional names belong in COMMON because they are equivalenced to names that are themselves in COMMON.

After performing the equivalence operations, we can create a memory map for each COMMON block by scanning the list of names for that block.

Initialize a counter to zero, and for each name on the list, make its offset equal to the current value of the counter. Then, add to the counter the number of memory units taken by the data object denoted by the name. The COMMON block records can then be deleted and the space reused by the next procedure.

A Simple Equivalence Algorithm

The first algorithms for processing equivalence statements appeared in assemblers rather than compilers. Since these algorithms can be a bit complex, especially when interactions between COMMON and EQUIVALENCE statements are considered, let us treat first a situation typical of an assembly language, where the only EQUIVALENCE statements are of the form

EQUIVALENCE A, B+offset

where A and B are the names of locations. This statement makes A denote the location that is offset memory units beyond the location for B.

A sequence of EQUIVALENCE statements groups names into equivalence sets whose positions relative to one another are all defined by the EQUIVALENCE statements, For example, the sequence of statements

EQUI VALENCE A, *B*+ 100
EQUIVALENCE C, D-40
EQUIVALENCE A, C+30
EQUIVALENCE E, F

groups names into the sets {A, B, C, D) and {E, F), where *E* and F denote the same location. C is 70 locations after B, A is 30 after C, and D L 10 after A.

To compute the equivalence sets we create a tree for each set. Each node of a tree represents a name and contains the offset of that name relative to the name at the parent of this node. The name at the root of a tree we call the *leader*. The position of any name relative to the leader can be computed by following the path from the node for that name and adding the offsets along the way.

An Equivalence Algarithm for Fortran

Algorithm : Construct ion *of* equivalence trees.

Input, A list of equivalence-defining statements of the form

EQUIVALENCE A, B+dist

Ourput. A collection of trees such that, for any name mentioned in the input list of equivalences, we may, by following the path from that name to the root and summing the *offset's* found along the path, determine the position of the name relative to the leader.

Method. *R*epeat the steps for each equivalence statement EQUIVALENCE A, B+dist, in turn. The justification for the formula in line

(12) for the offset of the leader of A relative to the leader of B is as follows.

The location of *A*, say *lA*, is equal to c plus the location of the leader of A, say *mA*, The location of B, say l_B, equals *d* plus the location of the leader of B, say m_B .But $l_A = l_B + dist$, so $c + m_A = d + m_B + dist$. Hence $m_A - m_B$ equals $d - c + dist$.

```
begin
(1) let p and q point to the nodes for A and 8, respectively;
(2) c := 0; d : = 0;
(3) while parent ( p ) # null do begin
(4) c:=c+offset(p);
(5) p : = parent (p)
end;
(6) while purent( q ) # null do begin
(7) d : = d + offset(q);
(8) q := parent ( q )
ead;
(9) if p = q then
(10) if c -d # dist then error;
else begin
        (11) parent(p):= q
        (12) offset (p ) : = d - c + dist
        end
end
```

Mapping Data Areas

We may now describe the rules whereby space in the various data areas is assigned for each routine's names.

1. For each COMMON block, visit all names declared to be in that block in the order of their declarations (use the chains of COMMON names created in the symbol table for this purpose). Allocate the number of words needed for each name in turn, keeping a count of the number of words allocated, so offsets can be computed for each name.

2. Visit all names for the routine in any order.

a) If a name is in COMMON, do nothing. Space has been allocated in (1)

b) If a name is not in COMMON and not equivalence, allocate the necessary number of words in the data area for the routine.

C) If a name A is equivalence, find its leader, say L. If L has already been given a position in the data area for the routine, compute the position of A by adding to that position all the ***offset's*** found in the path from A to L in the tree representing the equivalence set of A and L. If L has not been given a position, allocate the next
high-low words in the data area for the equivalence set. The position of L in these words is low words from the beginning, and the position of A can be calculated by summing offsets as before.

Anna University Question Bank

Nov/Dec 2016

PART A

1. Write down syntax directed definition of a simple desk calculator.
2. List Dynamic Storage allocation techniques. **(Q.No:17)**

PART B

1. (a) (i)A Syntax-Directed Translation scheme that takes strings of a's, b's and c's as input and produces as output the number of substrings in the input string that correspond to the pattern a(a|b)*c+(a|b)*b.For example the translation of the input string "abbcabcababc" is "3".

(1) Write a context-free grammar that generate all strings of a's. b's and c's.

(2) Give the semantic attributes for the grammar symbols.

(3) For each production of the grammar present a set of rules for evaluation of the semantic attributes. (8)

(ii) Illustrate type checking with necessary diagram.(8)**(Q.No:5)**

2. (b) Explain the following with respect to code generation phase.(16) **(Q.No:10)unit-v**

(i) Input to code generator

(ii) Target program

(iii) Memory management

(iv) Instruction selection

(v) Register allocation

(vi)Evaluation order.

May/June 2016

PART A

1. What is DAG? **(Q.No:33) unit-v**
2. When does a Dangling reference occur? **(Q.No:38)unit-v**

PART B

1. (a) (I) Construct a syntax directed definition for constructing a syntax tree for assignment statements.(8)

S → id := E
E → E1+E2
E → E1*E2
E → -E1
E → (E1)
E → id

(ii) Discuss specification of a simple type checker. (8)**(Q.No:6)**

2. (b) Discuss different storage allocation strategies. (16) **(Q.No:8)**

MAY/JUNE 2015
PART A

1. Construct a decorated parse tree according to the syntax directed definition ,for the following input statement:(4+7.5*3)/2

PART B

1. Write the semantic rule and derive the parse tree for the given code.
2. What is an Activation record? Explain how its relevant to the intermediate code generation phase with respect to procedure declarations.
3. Write the procedure to perform Register Allocation and Assignment with Graph coloring.
4. Mention in dretail any 4 issues in storage organization.(4)
5. Give a syntax-directed definition to differentiate expressions formed by applying the arithmetic operators + and * to the variable X and constants; expression: X*(3*x+x*x).

NOV/DEC-2013

2. (i) Discuss in details about storage allocation strategies.(8) **(Ref .Pg.no.68 Qus .no.16)**
(ii) Explain about various parameter passing methods in procedure calls.(8) **(Ref .Pg.no.42 Qus .no.9)**

MAY/JUNE 2013

2. (i)What are the different storage allocation strategies? Explain.(8) **(Ref .Pg.no.68 Qus .no.16)**
(ii) Specify a type checker which can handle expressions, statements and functions.(8) **(Ref .Pg.no.61 Qus .no.14)**

May/Jun -2012

3. Explain the detail about the specification of a simple type checker (16 Marks) **(Ref .Pg.no.61 Qus .no.14)**

Apr/May -2011

Part- A

1.What is handle pruning? **(Ref .Pg.no.9 Qus .no.15)**

2.What are the limitations of static allocation? **(Ref .Pg.no.11 Qus .no.22)**

3.Distinguish between the source text of a procedure and its activation at runtime (8 Marks) **(Ref .Pg.no.73 Qus .no.17)**

4.Discuss the various storage strategies in detail (8 Marks) **(Ref .Pg.no.68 Qus .no.16)**

<u>Nov/Dec – 2011</u>

Part- A

1.Define handle pruning**(Ref .Pg.no.9 Qus .no.15)**
2.Mention the two rules for type checking **(Ref .Pg.no.11 Qus .no.24)**
Part B
2. What are the different storage allocation strategies? Explain**(Ref .Pg.no.68 Qus .no.16)**

SYNTAX ANALYSIS

Need and Role of the Parser-Context Free Grammars -Top Down Parsing -General Strategies-Recursive Descent Parser Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR (0)Item-Construction of SLR Parsing Table -Introduction to LALR Parser - Error Handling and Recovery in Syntax Analyzer-YACC-Design of a syntax Analyzer for a Sample Language .

PART-A

1. Define Context free grammar.

Context-free Grammars

A *context-free grammar* for a language specifies the syntactic structure of programs in that language.

Components of a grammar:

- a finite set of tokens (obtained from the scanner);
- a set of variables representing "related" sets of strings, e.g., *declarations*, *statements*, expressions.
- a set of rules that show the structure of these strings.
- an indication of the "top-level" set of strings we care about.

Context-free Grammars: Definition

- Formally, a context-free grammar *G* is a 4-tuple *G* = (V, T, P, S), where:
  - V is a finite set of *variables* (or *nonterminals*). These describe sets of "related" strings.
  - T is a finite set of *terminals* (i.e., tokens).
  - P is a finite set of *productions*, each of the form $A \rightarrow \alpha$

Where $A \in$ V is a variable, and $\alpha \in (V \cup T)^*$ is a sequence of terminals and nonterminals.

  - S $\in$ V is the *start symbol*.

Example of CFG :

E ==>EAE | (E) | -E | id
A==> + | - | * | / |

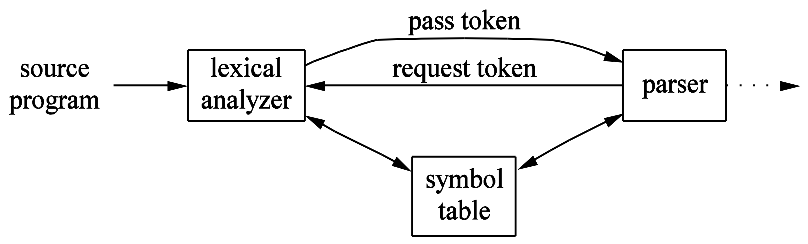
Where E,A are the non-terminals while id, +, *, -, /,(,) are the terminals.

2. What are parsers?

Parser

- Accepts string of tokens from lexical analyzer (usually one token at a time)
- Verifies whether or not string can be generated by grammar
- Reports syntax errors (recovers if possible)

THE ROLE OF A PARSER



Parser obtains a string of tokens from the lexical analyzer and verifies that it can be generated by the language for the source program. The parser should report any syntax errors in an intelligible fashion.

The two types of parsers employed are:

- Top down parser: which build parse trees from top(root) to bottom(leaves)
- Bottom up parser: which build parse trees from leaves and work up the root.

Therefore there are two types of parsing methods– top-down parsing and bottom-up parsing.

3. What are parse trees?

Parse Trees

- Nodes are non-terminals.
- Leaves are terminals.
- Branching corresponds to rules of the grammar.
- The leaves give a sentence of the input language.
- For every sentence of the language there is at least one parse tree.
- Sometimes we have more then one parse tree for a sentence.
- Grammars which allow more than one parse tree for some sentences are called ambiguous and are usually not good for compilation.

4. What are different kinds of errors encountered during compilation?

Compiler Errors

- Lexical errors (e.g. misspelled word)
- Syntax errors (e.g. unbalanced parentheses, missing semicolon)
- Semantic errors (e.g. type errors)
- Logical errors (e.g. infinite recursion)

Error Handling

- Report errors clearly and accurately
- Recover quickly if possible
- Poor error recover may lead to avalanche of errors

5. Define derivation and reduction.

Substitution of non-terminals to right side of production rule is called a derivation.

Replacing of string symbol by left side of the production is called reduction.

6. Define production rule (or rewriting rule).

The *production rule* is defined as the ways in which the syntactic categories may be built up from one another and from the terminal. Each production consists of non terminal followed by an arrow (or:: =) followed by a string of non-terminals and/or terminals.

7. Define recursive grammar with types.

A grammar that has a possibility of infinite substitution then the grammar is called recursive grammar. There are two types of recursion left recursion and right recursion. A grammar G is said to be left recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow A\propto$. A grammar G is said to be right recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow \propto A$

8.What are different error recovery strategies?

Error Recovery strategies

- **Panic mode**: discard tokens one at a time until a synchronizing token is found
- **Phrase-level recovery**: Perform local correction that allows parsing to continue
- **Error Productions**: Augment grammar to handle predicted, common errors
- **Global Production**: Use a complex algorithm to compute least-cost sequence of changes leading to parseable code

9.Explain Recursive descent parsing.

Recursive descent parsing:

Corresponds to finding a leftmost derivation for an input string
Equivalent to constructing parse tree in pre-order

Example:

Grammar: S ! cAd A ! ab j a
Input: cad

Problems:

- backtracking involved ()buffering of tokens required)
- left recursion will lead to infinite looping
- left factors may cause several backtracking steps

10. Define an ambiguous grammar (May/Jun -2012) (Nov\ Dec – 2007) (Nov\ Dec – 2005) (May/June 2016)

- A grammar that produces more than one parse for some sentence is said to be **ambiguous grammar**

Give an example of ambiguous grammar.

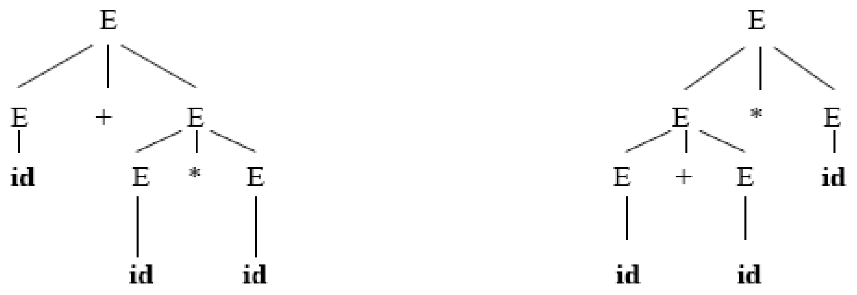
Example : Given grammar G : $E \rightarrow E+E \mid E*E \mid (E) \mid - E \mid id$

The sentence id+id*id has the following two distinct leftmost derivations:

$E \rightarrow E + E$	$E \rightarrow E * E$
$E \rightarrow id + E$	$E \rightarrow E + E * E$
$E \rightarrow id + E * E$	$E \rightarrow id + E * E$
$E \rightarrow id + id * E$	$E \rightarrow id + id * E$
$E \rightarrow id + id * id$	$E \rightarrow id + id * id$

The two corresponding parse trees are

The two corresponding parse trees are :



11. What is left recursion? How it is eliminated?

Left recursion: G istb left recursive if for some non-terminal A,
$A \overset{+}{\Rightarrow} A\alpha$

Simple case I: $A \rightarrow A\alpha \mid \beta \quad \Rightarrow \quad \begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \epsilon \end{array}$

Simple case II:

$$\begin{array}{lll} A & \rightarrow & A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \\ & & \beta_1 \mid \ldots \mid \beta_n \end{array}$$

$$\Downarrow$$

$$\begin{array}{lll} A & \rightarrow & \beta_1 A' \mid \ldots \mid \beta_n A' \\ A' & \rightarrow & \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon \end{array}$$

12. What is left factoring?

Left Factoring

- Rewriting productions to delay decisions
- Helpful for predictive parsing
- Not guaranteed to remove ambiguity

A → αβ1 | αβ2

$$\Downarrow$$

A → αA'
A' → β1 | β2

Left factoring:

Example: *stmt* → **if (** *expr* **)** *stmt*
| **if (** *expr* **)** *stmt* **else** *stmt*

Algorithm:

```
while left factors exist do
  for each non-terminal $A$ do
    Find longest prefix $\alpha$ common to $\geq 2$ rules
    Replace $A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \ldots$
    by  $A \rightarrow \alpha A' \mid \ldots$
        $A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$
  end for
end while
```

13. What is top down parsing?

Top down Parsing

- **Can be viewed two ways**:
  - Attempt to find leftmost derivation for input string
  - Attempt to create parse tree, starting from at root, creating nodes in preorder
- **General form is recursive descent parsing**
  - May require backtracking
  - Backtracking parsers not used frequently because not needed

14. What is predictive parsing?(Nov\ Dec – 2007)

- A special case of recursive-descent parsing that does not require backtracking
- Must always know which production to use based on current input symbol
- Can often create appropriate grammar:
  - removing left-recursion
  - left factoring the resulting grammar

15. Define LL(1) grammar.

LL(1) Grammars

- Algorithm covered in class can be applied to any grammar to produce a parsing table
- If parsing table has no multiply-defined entries, grammar is said to be "LL(1)"
  - First "L", left-to-right scanning of input
  - Second "L", produces leftmost derivation
  - "1" refers to the number of lookahead symbols needed to make decisions

16. What is shift reduce parsing?

Shift-Reduce Parsing

- One simple form of bottom-up parsing is shift-reduce parsing
- Starts at the bottom (leaves, terminals) and works its way up to the top (root, start symbol)
- Each step is a "reduction":
  - Substring of input matching the right side of a production is "reduced"
  - Replaced with the nonterminal on the left of the production
- If all substrings are chosen correctly, a rightmost derivation is traced in reverse

Shift-Reduce Parsing Example

```
S → aABe
A → Abc | b
B -> d
```

```
abbcde
aAbcde
aAde
aABe
S
```

$S \;_{rm}\Rightarrow aABe \;_{rm}\Rightarrow aAde \;_{rm}\Rightarrow aAbcde \;_{rm}\Rightarrow abbcde$

17. Define Handle. Apr/May – 2005 Nov\ Dec – 2004

Handles

- Informally, a "**handle**" of a string:
  - Is a substring of the string
  - Matches the right side of a production
  - Reduction to left side of production is one step along **reverse of rightmost derivation**
- Leftmost substring matching right side of production is not necessarily a handle
  - Might not be able to reduce resulting string to start symbol
  - In example from previous slide, if reduce aAbcde to aAAcde, can not reduce this to S
  -

Formally, a handle of a right-sentential form γ:

  - Is a production A ◊ β and a position of γ where β may be found and replaced with A
  - Replacing A by β leads to the previous right-sentential form in a rightmost derivation of γ
- So if S rm*=> αAw rm=> αβw then A ◊ β in the position following α is a handle of αβw
- The string w to the right of the handle contains only terminals
- Can be more than one handle if grammar is ambiguous (more than one rightmost derivation)

18. What is handle pruning? (Apr/May -2011) (Apr/May – 2004) (Nov/Dec 2016)

- Repeat the following process, starting from string of tokens until obtain start symbol:

  - Locate handle in current right-sentential form
  - Replace handle with left side of appropriate production
- Two problems that need to be solved:
  - How to locate handle
  - How to choose appropriate production

19. What are viable prefixes? Nov\ Dec – 2004

Viable Prefixes

- Two definitions of a viable prefix:
  - A prefix of a right sentential form that can appear on a stack during shift-reduce parsing
  - A prefix of a right-sentential form that does not continue past the right end of the rightmost handle
- Can always add tokens to the end of a viable prefix to obtain a right-sentential form

20. What is an operator grammar?

- A grammar having the property (among other essential requirements) that no production right side is ε or has two adjacent nonterminals is called an **operator grammar**.

Operator Grammar Example

E C→ E + E | E – E | E * E | E / E
| E ^ E | (E) | -E | id

Where

- ^ is of highest precedence and is right-associative
- * and / are of next highest precedence and are left-associative
- + and – are of lowest precedence and are left-associative

21. What are LR parsers?

LR Parsers

- LR Parsers us an efficient, bottom-up parsing technique useful for a large class of CFGs
- Too difficult to construct by hand, but automatic generators to create them exist (e.g. Yacc)
- LR(k) grammars
  - “L” refers to left-to-right scanning of input
  - “R” refers to rightmost derivation (produced in reverse order)
  - “k” refers to the number of lookahead symbols needed for decisions (if omitted, assumed to be 1)

22. What are the benefits of LR parsers?

Benefits of LR Parsing

- Can be constructed to recognize virtually all programming language construct for which a CFG can be written
- Most general non-backtracking shift-reduce parsing method known
- Can be implemented efficiently
- Handles a class of grammars that is a superset of those handled by predictive parsing
- Can detect syntactic errors as soon as possible with a left-to-right scan of input

23. What are three types of LR parsers?

Three methods:

a. **SLR (simple LR)**
   i. Not all that simple (but simpler than other two)!
   ii. Weakest of three methods, easiest to implement
b. **Constructing canonical LR parsing tables**
   i. Most general of methods
   ii. Constructed tables can be quite large
c. **LALR parsing table (lookahead LR)**
   i. Tables smaller than canonical LR
   ii. Most programming language constructs can be handled

24. What is dangling reference? May/Jun -2012

A problem which arises when there is a reference to storage that has been deallocated

Eg:

```
Int I = 15 ,* p
P = &I;
Free (p);
Printf("%u",p);
```

25. What are the limitations of Static Allocation?(Apr/May -2011)

- Size of a data object and constraints must be known at compile time
- Recursive procedures are restricted
- Data structures cannot be created dynamically

26. Give two examples for each of top down parser and bottom up parser?

(Apr/May – 2010)

Top – down parser

- Predictive parser
- Non Predictive parser

Bottom – up parser

- LR parser
- Operator precedence parser
- Shift Reduce Parser

27. Mention the two rules for type checking (Nov/Dec – 2011)

We can now write the semantic rules for type checking of statements as follows

Translation scheme for checking the type of statements:

1. Assignment statement:

S → **id** : = E { S.*type* : = **if id**.*type* = E.*type* **then** *void*
else *type_error* }

2. Conditional statement:

S → **if** E **then** S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type*
else *type_error* }

3. While statement:

S → while E do S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type*
else *type_error* }

8. Differentiate SLR parser from LALR parser (Nov/Dec – 2010)

Demerits of SLR

- It will not produce uniquely defined parsing action tables for all grammars
- Shift reduce conflict

Demerits of LALR

- Merger will produce reduce / reduce conflict
- On erroneous input, LALR parser may proceed to do some reductions after the LR parser has declared an error , but LALR parser never shift a symbol after the LR Parser declares an errors

29. Differentiate Top down approach from Bottom Up approach to parsing with an example (Nov/Dec – 2010)

Top–down parsing:

- A parser can start with the start symbol and try to transform it to the input string.

Example : LL Parsers.

- It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of top-down parsing:

- Recursive descent parsing
- Predictive parsing

Bottom–up parsing :

- A parser can start with input and attempt to rewrite it into the start symbol.

Example: LR Parsers.

- Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.
- A general type of bottom-up parser is a **shift-reduce parser**.

30. What is the signification of look ahead in LR (1) items (Apr/May – 2010)

The general form of LR (1) item is

S-> X.Y, a;

A where ‘a’ is called look ahead .this is an extra information. we are looking a character ahead the ‘a’ may be terminal or the right end marker $

Example:

S^1->. S , $

$ is a look ahead

Here we use “closure “& “goto” functions for constructing LR (1) items taking look aheads in to account

31. List the factors to be considered for top – down parsing (May/ Jun – 2009)

- Elimination of left recursion
- Elimination of left factoring

32. Eliminate the left recursion from the following grammar

A -> Ac/Aad/bd/c (Nov\ Dec – 2007)(May/June-2013)

Ans:

A -> Ac/Aad/bd/c

After removal of left recursion

A ->bd A’/A’

A’->CA’/adA’/C

33. What is phrase level error recovery?(May/ Jun – 2006)

Phrase level error recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, or delete symbols on the input and issue appropriate error messages. They may also pop from the stack.

34. **What are the goals of error handler in a parser? May/ Jun – 2006 Apr/May – 2005**

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs

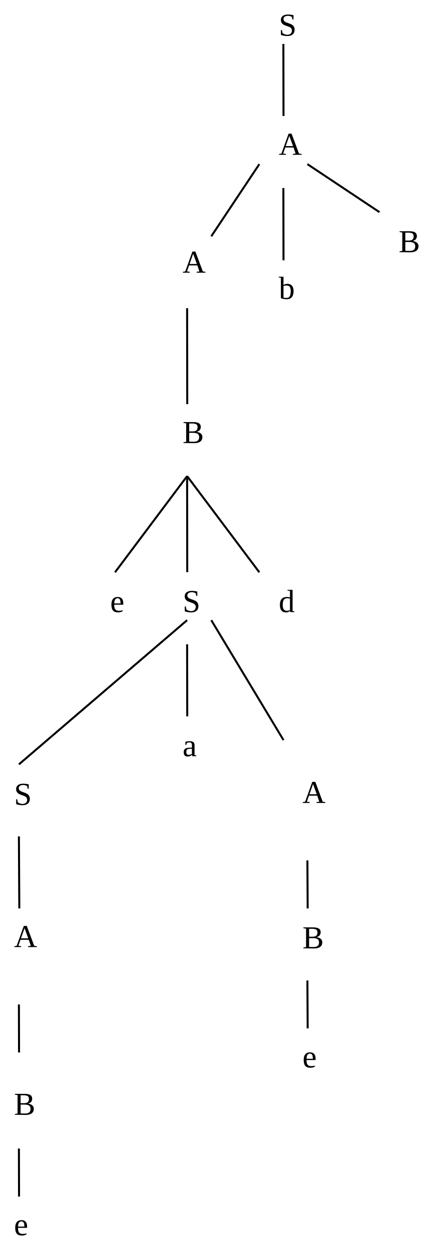
35. What do you mean by viable prefixes? (Nov\ Dec – 2004)

The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes. An equivalent definition of a viable prefix is that it is a prefix of a right sentential form that does not continue past the right end of the rightmost handle of that sentential form.

36. What are kernel and non kernel items? (Nov\ Dec – 2005)

Kernel items, whish include the initial item, S'→ .S, and all items whose dots are not at the left end.
Non-kernel items, which have their dots at the left end.

37. Derive the string and construct a syntax tree for the input string ceaedbe using the grammar S -> SaA | A,A ->AbB | B, B -> c Sd | e (May/ Jun – 2009)



38. Define LR(0) items (Apr/May – 2004)

- An LR(0) item of a grammar G is a production of G with a dot at some position of the right side.
- Thus, production A → XYZ yields the four items

A→ . XYZ

A→X .YZ

A→XY . Z

A→ XYZ .

39. How will YACC resolve the parsing action conflicts?(Nov\ Dec – 2005)

- YACC is an automatic tool for generating the parser program.
- YACC stands for yet another Compiler Compiler which is basically the utility available

from UNIX.
- Basically YACC is LALR parser generator.
- It can report conflict or ambiguities in the form of error messages.

40. What are the components of LR parser?
- An input
- An output
- A stack
- A driver program
- A parsing table

41. What are the different kinds of errors faced by a program?
- Lexical error
- Syntactic error
- Semantic error
- Logical errors

42. What are the semantic errors?
The errors due to undefined variables incompatible operands, etc., are called semantic errors

43. List the properties of LR parser.
- LR parsers can be constructed to recognize most of the programming Languages for which the context free grammar can be written.
- The class of grammar that can be parsed by LR parser is a superset of class of grammars that can be parsed using predictive parsers.
- LR parsers work using non backtracking shift reduce technique yet it is efficient one

44. Mention the types of LR parser.
- SLR parser- simple LR parser
- LALR parser- lookahead LR parser
- Canonical LR parser

45. What are the problems with top down parsing?
The following are the problems associated with top down parsing:
- Backtracking
- Left recursion
- Left factoring
- Ambiguity

46. What is dangling else problem?
Ambiguity can be eliminated by means of dangling-else grammar which is show below:

stmt → if expr then stmt
| if expr then stmt else stmt
| other

47. Define sentential form?
If G = (V, T, P, S) is a CFG, then any string ‘α’ in (VUT)* such that S→* α is a sentential form.

48. What are the difficulties with top down parsing?
a) Left recursion
b) Backtracking
c) Left factoring - The order in which alternates are tried can affect the language accepted
d) Ambiguity
e) When failure is reported. We have very little idea where the error actually occurred.

49. What is meant by recursive-descent parser?
A parser that uses a set of recursive procedures to recognize its input with no backtracking is called a recursive-descent parser. To avoid the necessity of a recursive language, we shall also consider a tabular implementation of recursive descent called predictive parsing.

50. What are the actions available in shift reduce parser?
a) Shift
b) Reduce
c) Accept
d) Error

51. What are the two common ways of determining precedence relations should hold between a pair of terminals?
a) Based on associativity and precedence of operators.
b) Construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse tree.

52. Define augmented grammar?
If G is a grammar with start symbol S, then G' the augmented grammar for G, is G with a new start symbol S' and production S'→S. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input.

53. Write short notes on YACC.

YACC is an automatic tool for generating the parser program.

YACC stands for Yet Another Compiler which is basically the utility available from UNIX.

Basically YACC is LALR parser generator.

It can report conflict or ambiguities in the form of error messages.

54. Give examples for static check.(May/June-2013)

Examples of static checks are

1.Type checks
2.Flow-of-control checks
3.Uniqueness checks
4.Name –related checks

55. What is meant by coercion? (Nov/Dec 2013)
Process by which a compiler automatically converts a value of one type into a value of another type

when that second type is required by the surrounding context.

56. Write the algorithm for FIRST and follow in parser.(May/June 2016)

Algorithm for first( ):

1. If *X* is terminal, then FIRST(*X*) is {X}.

2. If $X \to \varepsilon$ is a production, then add ε to FIRST(*X*).

3. If *X* is non- terminal and $X \to a\alpha$ is a production then add *a* to FIRST(X).

4. If X is non- terminal and $X \to Y1\ Y2 \ldots Yk$ is a production, then place *a* in FIRST(*X*) if for some *i*, *a* is in FIRST(Yi), and ε is in all of FIRST(Y1),…,FIRST(Yi-1); that is, Y1,….Yi-*1* => ε. If ε is in FIRST(*Y j*) for all j=1,2,..,k, then add ε to FIRST(*X*).

Algorithm for follow( ):

1. If *S* is a start symbol, then FOLLOW(*S*) contains $.

2. If there is a production $A \to \alpha B\beta$, then everything in FIRST(β) except ε is placed in follow(*B*).

3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B\beta$ where FIRST(β) contains ε, then everything in FOLLOW(*A*) is in FOLLOW(*B*).

57. What is the purpose of syntax analysis?

The purpose of syntax analysis or parsing is to chech that whether the sequence of token is valid in a pariticular programming language.

58. What are he measures of syntax analyser?

The syntax analyser is expected to take the following measures on the ocuurence of the syntax eroors:

i)Report the errors

ii) Recover from the error and find the next error

iii) should not degrade the execution speed of the program.

59. What are thr limitations of syntax analyser?

There are some situations where the syntax analysers may not be helpful. Syntax analysers have the following limintations:

- It cannot determine if a token is valid
- It cannot determine if a token is declared before it is being used
- It cannot determine if a token is initilaised before it is used
- It cannot determine if an operation performed on a token type is valid or not.

60. What is leftmost derivation and rightmost derivation?

Leftmost derivation: If the leftmost non-terminal is replaced first by its production, then it is termed as leftmost derivation.

Rightmost derivation: If the rightmost non-terminal is replaced first by its production, then it is termed as leftmost derivation.

61. Define inherent amgiuity.

A language that has every grammar to be ambiguous is said to be inherenly ambiguous.

62. How to avoid ambiguity in grammar?

There is no algorithm to solve the ambiguity problem. But there are two ways to avoid writing ambiguous grammar:

- Follow the operator precdence rules while writng the grammar.
- Introduce new variables.

63. What is GOTO function?

GOTO function is used to determine possible states reachable from existing state.

Goto(state stack element) is the closure of the set of items that result from shifting stack element in state.

64. What is canonical LR()?

A canonical LR perser or LR(1) parser is an LR(K) parser for k=1, i.e with a single lookhead terminal.

65.What are the operator precedence relations?

Relation	**Meaning**
a<. b	A yields precedence to b (b has higher precedence)
a =. b	A has the same precedence as b (a and b are of same precedence)
a.> b	A talkes precednce over b (a has higher precedence)

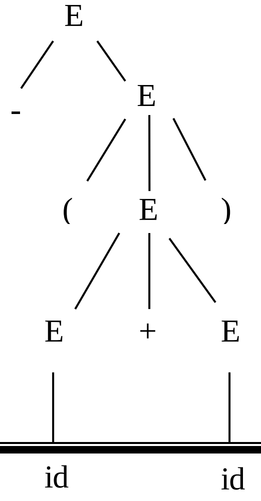
66. Construct a parse tree for –(id+id). (Nov/Dec 2016)

Consider grammar G : E → E+E |E*E |(E) | - E |id

Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	RIGHTMOST DERIVATION
E → - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E)
E → - (id+E)	E → - (E+id)
E → - (id+id)	E → - (id+id)

Parse tree:



PART-B

1. Discuss about introduction of Syntax Analysis.

SYNTAX ANALYSIS

Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree.

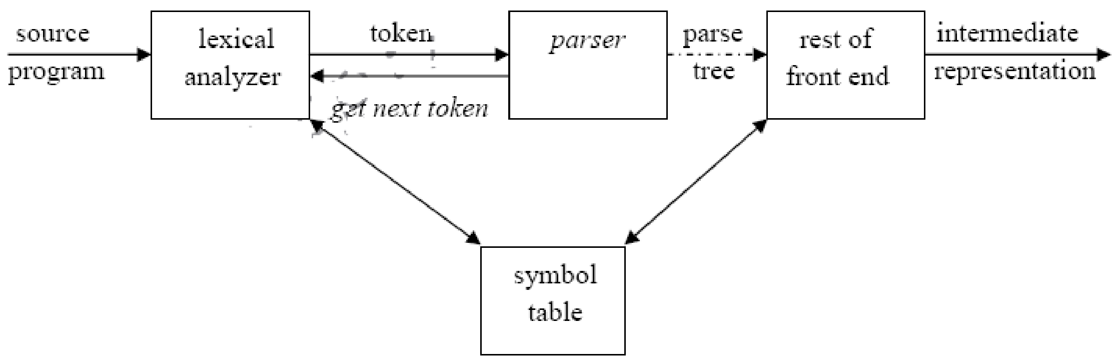
Advantages of grammar for syntactic specification:

1. A grammar gives a precise and easy-to-understand syntactic specification of a programming language.
2. An efficient parser can be constructed automatically from a properly designed grammar.
3. A grammar imparts a structure to a source program that is useful for its translation into object code and for the detection of errors.
4. New constructs can be added to a language more easily when there is a grammatical description of the language.

2. Explain the role of parser. (Nov\ Dec – 2007)(6marks)

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

Position of parser in compiler model



Functions of the parser :

- It verifies the structure generated by the tokens based on the grammar.
- It constructs the parse tree.
- It reports the errors.
- It performs error recovery.

Issues :

Parser cannot detect errors such as:

- Variable re-declaration
- Variable initialization before use.
- Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

Syntax error handling :

Programs can contain errors at many different levels. For example :

- Lexical, such as misspelling a keyword.
- Syntactic, such as an arithmetic expression with unbalanced parentheses.
- Semantic, such as an operator applied to an incompatible operand.
- Logical, such as an infinitely recursive call.

Functions of error handler :

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs.

3. Explain the error recovery strategies on predictive parser. (or)
Explain the error recovery strategies in syntax analysis (6 Marks)(Apr/ May 2011)
Error recovery strategies: May/ Jun – 2006

The different strategies that a parse uses to recover from a syntactic error are:

- Panic mode
- Phrase level
- Error productions
- Global correction

Panic mode recovery:

On discovering an error, the parser discards input symbols one at a time until a synchronizing token is found. The synchronizing tokens are usually delimiters, such as semicolon or **end**. It has the advantage of simplicity and does not go into an infinite loop. When multiple errors in the same statement are rare, this method is quite useful.

Phrase level recovery:

On discovering an error, the parser performs local correction on the remaining input that allows it to continue. Example: Insert a missing semicolon or delete an extraneous semicolon etc.

Error productions:

The parser is constructed using augmented grammar with error productions. If an error production is used by the parser, appropriate error diagnostics can be generated to indicate the erroneous constructs recognized by the input.

Global correction:

Given an incorrect input string x and grammar G, certain algorithms can be used to find a parse tree for a string y, such that the number of insertions, deletions and changes of tokens is as small as possible. However, these methods are in general too costly in terms of time and space.

4. Explain the Context-Free Grammars(6 marks) Or

Explain Context free grammar with examples.(10) May/June 2016

A Context-Free Grammar is a quadruple that consists of **terminals**, **non-terminals, start symbol** and **productions.**

- **Terminals:** These are the basic symbols from which strings are formed.
- **Non-Terminals:** These are the syntactic variables that denote a set of strings. These help to define the language generated by the grammar.
- **Start Symbol:** One non-terminal in the grammar is denoted as the “Start-symbol” and the set of strings it denotes is the language defined by the grammar.
- **Productions:** It specifies the manner in which terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal, followed by an arrow, followed by a string of non-terminals and terminals.

Example of context-free grammar: The following grammar defines **simple** arithmetic expressions:

expr → *expr op expr*
expr → (*expr*)
expr → - *expr*
expr → **id**
op → +
op → -
op → *
op → /
op → ↑

In this grammar,

- **id** + - * / ↑( ) are terminals.
- *expr* , *op* are non-terminals.
- *expr* is the start symbol.
- Each line is a production.

5. Explain the Derivations: (8 Marks)

Two basic requirements for a grammar are :

- To generate a valid string.
- To recognize a valid string.

Derivation is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

Example : Consider the following grammar for arithmetic expressions :

$E \rightarrow E+E \mid E*E \mid (E) \mid - E \mid id$

To generate a valid string - (id+id) from the grammar the steps are

1. $E \rightarrow - E$
2. $E \rightarrow - (E)$
3. $E \rightarrow - (E+E)$
4. $E \rightarrow - (id+E)$
5. $E \rightarrow - (id+id)$

In the above derivation,

- E is the start symbol.
- - (id+id) is the required sentence (only terminals).
- Strings such as E, -E, -(E), . . . are called sentinel forms.

Types of derivations:

The two types of derivation are:

- Left most derivation
- Right most derivation.

- In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.
- In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement.

Example:

Given grammar G : $E \rightarrow E+E \mid E*E \mid (E) \mid - E \mid id$

Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	RIGHTMOST DERIVATION
$E \rightarrow - E$	$E \rightarrow - E$
$E \rightarrow - (E)$	$E \rightarrow - (E)$
$E \rightarrow - (E+E)$	$E \rightarrow - (E+E)$
$E \rightarrow - (id+E)$	$E \rightarrow - (E+id)$
$E \rightarrow - (id+id)$	$E \rightarrow - (id+id)$

- Strings that appear in leftmost derivation are called **left sentinel forms.**
- Strings that appear in rightmost derivation are called **right sentinel forms.**

Sentinels:

Given a grammar G with start symbol S, if $S \rightarrow \alpha$, where α may contain non-terminals or terminals, then α is called the sentinel form of G.

Yield or frontier of tree:

Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called **yield** or **frontier** of the tree.

6. Explain the Ambiguity or Ambiguous grammar G **: E → E+E | E*E | (E) | - E | id . (Nov/Dec 2016)(6 marks)**

A grammar that produces more than one parse for some sentence is said to be **ambiguous grammar**.

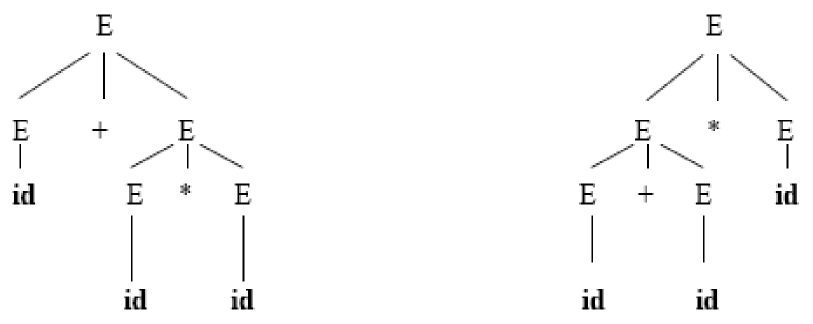
Example : Given grammar G : E → E+E | E*E | (E) | - E | id

The sentence id+id*id has the following two distinct leftmost derivations:

E → E+ E	E → E* E
E → id + E	E → E + E * E
E → id + E * E	E → id + E * E
E → id + id * E	E → id + id * E
E → id + id * id	E → id + id * id

The two corresponding parse trees are

The two corresponding parse trees are :



7. Explain the Writing a Grammar

There are four categories in writing a grammar:

- Regular Expression Vs Context Free Grammar
- Eliminating ambiguous grammar.
- Eliminating left-recursion
- Left-factoring.

Each parsing method can handle grammars only of a certain form hence, the initial grammar may have to be rewritten to make it parable

Regular Expressions vs. Context-Free Grammars:

REGULAR EXPRESSION	**CONTEXT-FREE GRAMMAR**
It is used to describe the tokens of programming languages.	It consists of a quadruple where S → start symbol, P → production, T → terminal, V → variable or non- terminal.

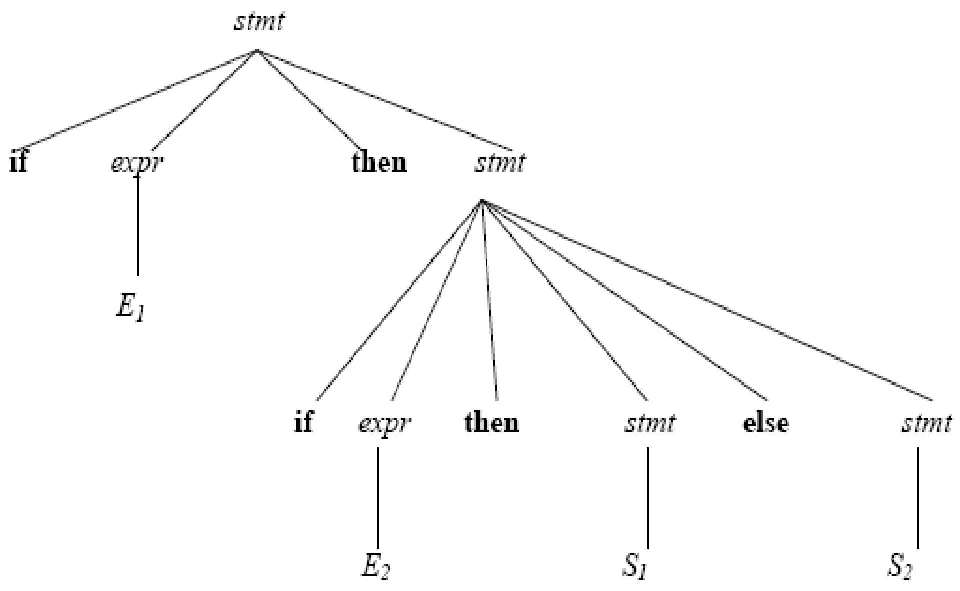
It is used to check whether the given input is valid or not using **transition diagram.**	It is used to check whether the given input is valid or not using **derivation**.
The transition diagram has set of states and edges.	The context-free grammar has set of productions.
It has no start symbol.	It has start symbol.
It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.	It is useful in describing nested structures such as balanced parentheses, matching begin-end's and so on.

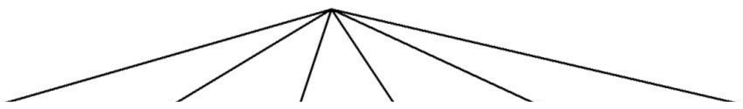
- The lexical rules of a language are simple and RE is used to describe them.
- Regular expressions provide a more concise and easier to understand notation for tokens than grammars.
- Efficient lexical analyzers can be constructed automatically from RE than from grammars.
- Separating the syntactic structure of a language into lexical and non lexical parts provides a convenient way of modularizing the front end into two manageable-sized components.

Eliminating ambiguity:

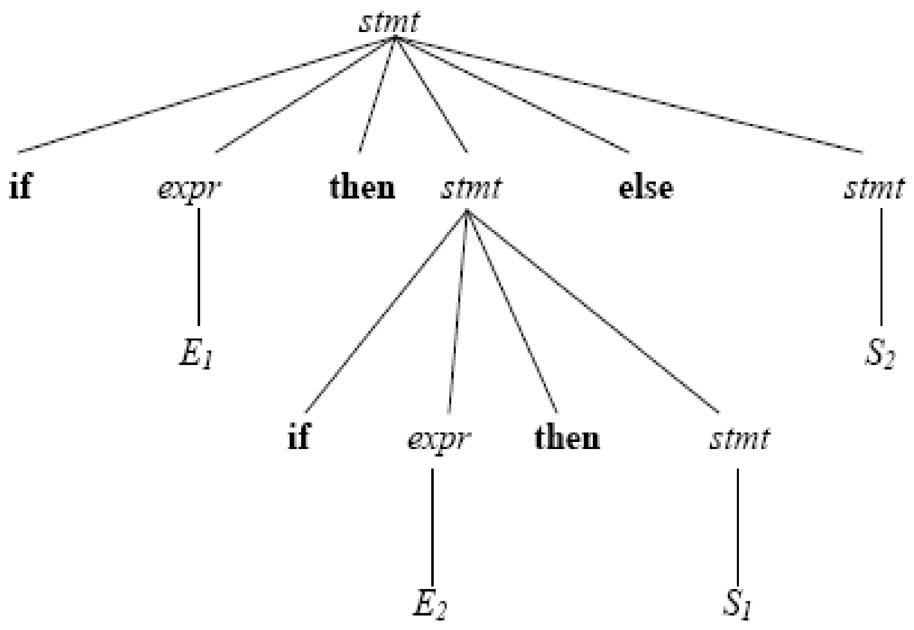
- Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.
- Consider this example, G: *stmt* → **if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | **other**
- This grammar is ambiguous since the string **if E1 then if E2 then S1 else S2** has the following two parse trees for leftmost derivation :

1.





2.



Eliminating Left Recursion:

A grammar is said to be *left recursive* if it has a non-terminal *A* such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production A → Aα |β it can be replaced with a sequence of two productions

A → βA'

A'→ αA' | ε

Without changing the set of strings derivable from A.

Example: Consider the following grammar for arithmetic expressions:

E → E+T | T
T → T*F | F
F → (E) | id

First eliminate the left recursion for E as

E → TE'

E' → +TE' |ε

Then eliminate for T as

T → FT'

T'→ *FT' | ε

Thus the obtained grammar after eliminating left recursion is

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

Algorithm to eliminate left recursion:

```
1. Arrange the non-terminals in some order A1, A2 . . . An.
2. for i := 1 to n do begin
       for j := 1 to i-1 do begin
           replace each production of the form Ai → A j γ by the
               productions Ai → δ1 γ | δ2γ | . . . | δk γ
               where Aj → δ1 | δ2 | . . . | δk are all the current Aj-productions;
       end
       eliminate the immediate left recursion among the Ai-productions
   end
```

Eliminating the Left factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any production A →αβ1 | αβ2 , it can be rewritten as

A → αA'

A'→ β1 | β2

Consider the grammar, G: S → iEtS | iEtSeS | a
E → b

Left factored, this grammar becomes

S → iEtSS' | a
S'→ eS |ε
E → b

8. Explain the Parsing. Or
Construct parse tree for the input string w=cad using top down parser. (Nov/Dec 2016)

It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

Parse tree:

Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals.

Types of parsing:

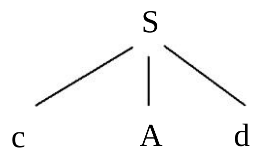
- Top down parsing
- Bottom up parsing
- Top–down parsing: A parser can start with the start symbol and try to transform it to the input string.
    - Example: LL Parsers.
- Bottom–up parsing: A parser can start with input and attempt to rewrite it into the start symbol.
    - Example: LR Parsers.

Construct parse tree for the input string w=cad using top down parser. (Nov/Dec 2016)

S → cAd
A → ab | a

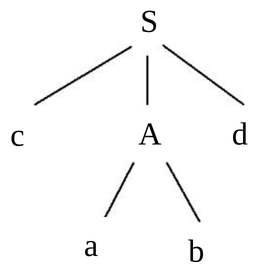
The parse tree can be constructed using the following top-down approach :

Step1:
Initially create a tree with single node labeled S. An input pointer points to ‘c’, the first symbol of w. Expand the tree with the production of S.



Step2:

The leftmost leaf ‘c’ matches the first symbol of w, so advance the input pointer to the second symbol of w ‘a’ and consider the next leaf ‘A’. Expand A using the first alternative.



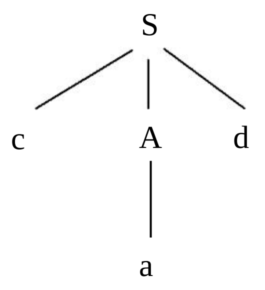
Step3:

The second symbol ‘a’ of w also matches with second leaf of tree. So advance the input pointer to third symbol of w ‘d’**.** But the third leaf of tree is b which does not match with the input symbol **d.**

Hence discard the chosen production and reset the pointer to second position. This is called **backtracking.**

Step4:

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

9. Explain the Top-Down Parsing (16 marks)

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of top-down parsing:

1. Recursive descent parsing (with back tracking)
2. Predictive parsing (without backtracking)
3. Non Recursive descent parsing (without backtracking)

Recursive Descent Parsing or backtracking parser (8 Marks)

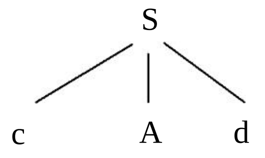
- Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input**.**
- This parsing method may involve **backtracking**, that is, making repeated scans of the input.

Example for backtracking :

Consider the grammar G : $S \rightarrow cAd$
$A \rightarrow ab \mid a$
and the input string w=cad.

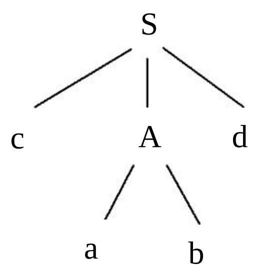
The parse tree can be constructed using the following top-down approach :

Step1:
Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w. Expand the tree with the production of S.



Step2:

The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.



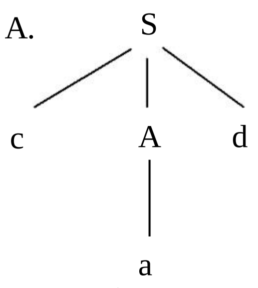
Step3:

The second symbol ‘a’ of w also matches with second leaf of tree. So advance the input pointer to third symbol of w ‘d’**.** But the third leaf of tree is b which does not match with the input symbol **d.**

Hence discard the chosen production and reset the pointer to second position. This is called **backtracking.**

Step4:

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

<u>Example for recursive decent parsing</u>:

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop. Hence, **elimination of left-recursion** must be done before parsing.

Consider the grammar for arithmetic expressions
E → E+T | T
T → T*F | F
F → (E) | id
After eliminating the left-recursion the grammar becomes,

E → TE’
E’ → +TE’ | ε
T → FT’
T’ → *FT’ | ε
F → (E) | id
Now we can write the procedure for grammar as follows:
Recursive procedure:

```
Procedure E()
begin
        T( );
        EPRIME( );
End
```

```
Procedure EPRIME( )
begin
        If input_symbol='+' then
        ADVANCE( );
        T( );
        EPRIME( );
end

Procedure T( )
begin
        F( );
        TPRIME( );
end

Procedure TPRIME( )
begin
        If input_symbol='*' then
        ADVANCE( );
        F( );
        TPRIME( );
end

Procedure F( )
begin
        If input-symbol='id' then
        ADVANCE( );
        else if input-symbol='(' then
        ADVANCE( );
        E( );
        else if input-symbol=')' then
        ADVANCE( );
end

else ERROR( );
```

Stack implementation:

To recognize input **id+id*id :**

PROCEDURE	**INPUT STRING**
E( )	**id**+id*id
T( )	**id**+id*id
F( )	**id**+id*id
ADVANCE( )	id**+**id*id
TPRIME( )	id**+**id*id
EPRIME( )	id**+**id*id
ADVANCE( )	Id+**id***id
T( )	id+**id***id

F( )	id+**id***id
ADVANCE( )	id+id***id
TPRIME( )	id+id***id
ADVANCE( )	id+id* **id**
F( )	id+id* **id**
ADVANCE( )	id+id* **id**
TPRIME( )	id+id* **id**

10. Explain the Predictive Parsing. (16 marks)]

- Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- The key problem of predictive parsing is to determine the production to be applied for a non- terminal in case of alternatives.

Two types

- Recursive Predictive parsing (without backtracking)
- Non Recursive descent parsing (without backtracking

Recursive Predictive parsing (without backtracking)

Transition diagram for predictive parser

The Transition diagram for predictive parser

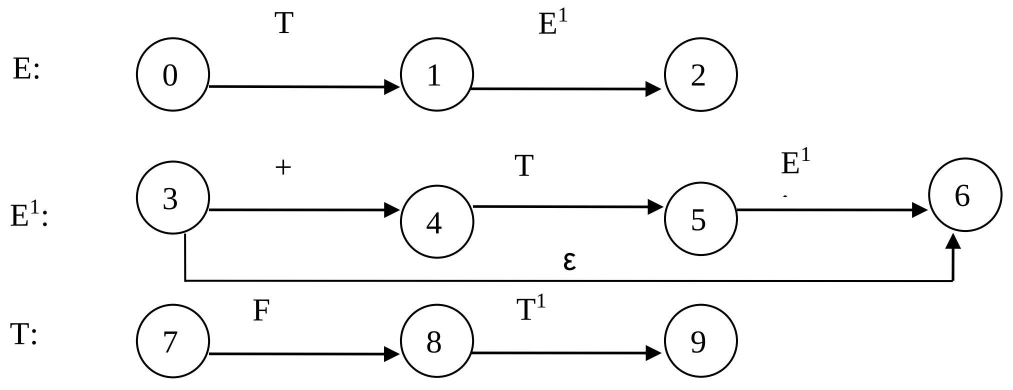
- One diagram for each non terminal
- Labels of edges are tokens and non terminals
- Transition on a token means that transition has to be done if that token is the next input symbol
- A transition on a nonterminal 'A' is a call of the procedure for A

To construct the transition diagram of a predictive parser, do the following

- Eliminate left recursion
- Left factor

For each nonterminal A do following

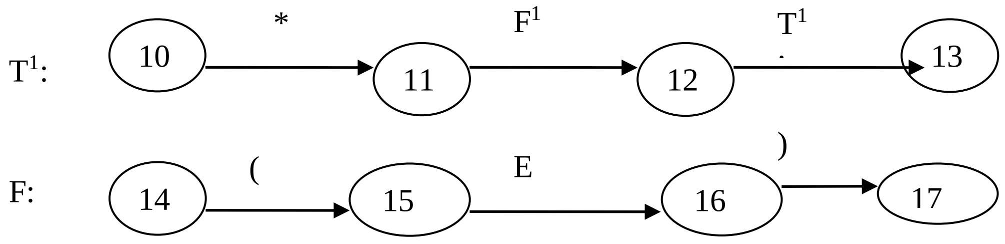
- Create an initial and final state
- For each production A → X1,X2,……Xn, create a path initial to final state with edges labled X1,X2,…..Xn



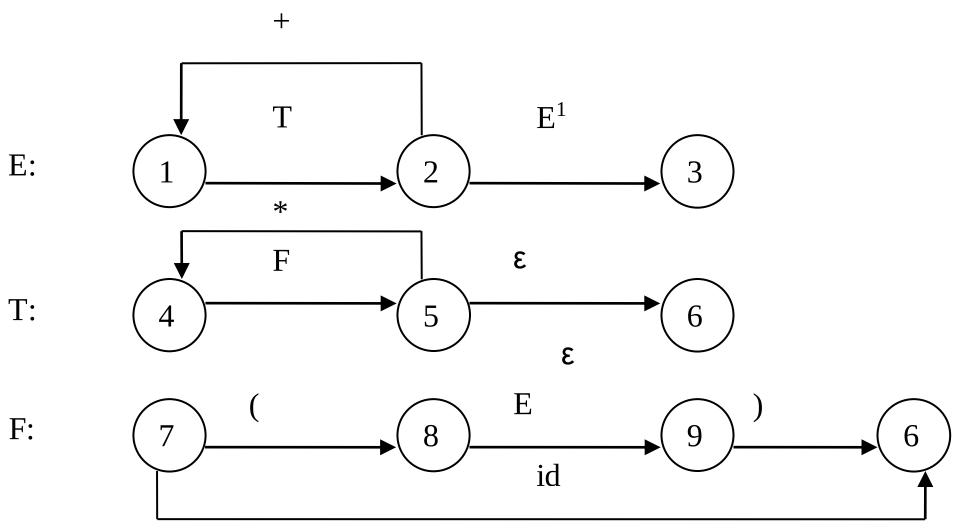
A recursive predictive parsing program based on a transition diagram tries to match terminal symbols against the input . When the parsing has to follow an edge labeled a nonterminal, it makes a recursive procedure call to the corresponding transition diagram

Working of predictive parser

- Parser beings in the start state for the start symbol
- If after some actions, it is in state 'S' with an edge labeled by terminal 'a' to state 't' and if the next input symbol is a and if the next input symbol is 'a' then parser moves the input pointer one position right and goes to state 't'
- If an edge is labeled by a nonterminal 'A' the parser goes to start state for 'A' without moving the input pointer. if it reaches the final state for 'A', it immediately goes to state 't'.
- If there is an edge4 from S to t labeled ε, then from state S the parser goes to t without advancing the input pointer.



After simplification



Error recovery in predictive parsing

- **Panic mode error recovery**
- **Phrase –level recovery**

Panic mode error recovery

It is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. The synchronizing set should be chosen so that the parser recovers quickly from errors that are likely to occur in practice.

Some Heuristics are as follows:

1. We can place all symbols in Follows (A) into the synchronizing set for nonterminal A. If we skip tokens until an element of follow(A) is seen and pop A from the stack, it is likely that parsing can continue.

2. It is not enough to use FOLLOW(A) as the synchronizing set for A. Often, there is a hierarchical structure on constructs in a language eg, expressions appears within statements , which appears within blocks and so on. We can add to the synchronizing set of a lower constructs the symbols that begins higher constructs the symbols that begin higher constructs.

3.If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

4. If a non terminal can generate the empty string ,then the production deriving can be used as a default . This approaches reduces a numberof nonterminals that have to be consider during error recovery.

5. If a terminal on the top of the stack cannot be matched, a simple idea is to pop the terminal issue a message saying that the terminal was inserted and continued parsing.

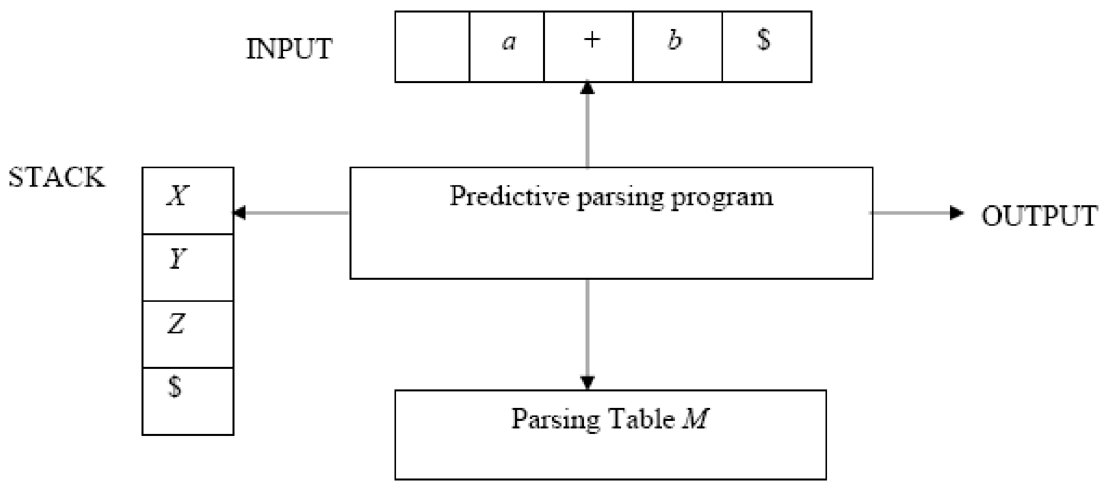
Phrase –level recovery:

It is implemented by implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. The routines may change, insert or delete symbols on the input and issue appropriate error message.

Write an algorithm for Non recursive predictive parsing. (6) May/June 2016

Non-recursive predictive parser

- **Introduction**
- **Algorithm for non-recursive predictive parsing**
- **Predictive parsing table construction**
- **Algorithm for construction of predictive parsing table**
- **Example**



Introduction

The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

Input buffer:

It consists of strings to be parsed, followed by \$ to indicate the end of the input string.

Stack:

It contains a sequence of grammar symbols preceded by \$ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of \$.

Parsing table:

It is a two-dimensional array *M*[*A*, *a*], where **'*A*'** is anon-terminal and **'*a*'** is aterminal.

Predictive parsing program:

The parser is controlled by a program that considers *X*, the symbol on top of stack, and *a*, the current input symbol. These two symbols determine the parser action. There are three possibilities:

1. If $X = a =\$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, the parser pops *X* off the stack and advances the input pointer to the next input symbol.
3. If *X* is a non-terminal , the program consults entry *M*[*X*, *a*] of the parsing table *M*. This entry will either be an *X*-production of the grammar or an error entry.
   If $M[X , a] = \{X \rightarrow UVW\}$,the parser replaces *X* on top of the stack by *WVU*. If
   M[*X* , *a*] = **error**, the parser calls an error recovery routine.

<u>Algorithm for non recursive predictive parsing:</u>

Input : A string *w* and a parsing table *M* for grammar *G*.

Output : If *w* is in *L*(*G*), a leftmost derivation of *w*; otherwise, an error indication.

Method : Initially, the parser has \$*S* on the stack with *S*, the start symbol of *G* on top, and *w*\$ in the input buffer. The program that utilizes the predictive parsing table *M* to produce a parse for the input is as follows:

```
set ip to point to the first symbol of w$;

repeat
let X be the top stack symbol and a the symbol pointed to by ip;
if X is a terminal or $ then
if X = a then
pop X from the stack and advance ip
else error()
else /* X is a non-terminal */
if M[X, a] = X →Y1Y2 … Yk then begin
pop X from the stack;
push Yk, Yk-1, … ,Y1 onto the stack, with Y1 on top;
output the production X → Y1 Y2 . . . Yk
```

```
end
else error()
until X = $ /* stack is empty */
```

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G :

1. FIRST

2. FOLLOW

Rules for first( ):

1. If X is terminal, then FIRST(X) is {X}.
2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).
3. If X is non- terminal and $X \rightarrow a\alpha$ is a production then add a to FIRST(X).
4. If X is non- terminal and $X \rightarrow Y1\ Y2 \ldots Yk$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1),…,FIRST(Yi-1); that is, Y1,….Yi-*1* => ε. If ε is

in FIRST($Y\,j$) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for follow( ):

1. If S is a start symbol, then FOLLOW(S) contains $.
2. If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST(β) except ε is placed in follow(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Algorithm for construction of predictive parsing table:

Input: Grammar *G*

Output: Parsing table *M*

Method:

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.

2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.

3. If ε is in FIRST(α), add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A) , add $A \rightarrow \alpha$ to $M[A, \$]$.

4. Make each undefined entry of *M* be **error**.

Example:

Consider the following grammar:

E → E+T |T
T → T*F |F
F→ (E) |id

After eliminating left-recursion the grammar is

E → TE’
E’ → +TE’ |ε
T → FT’
T’ → *FT’ | ε
F → (E) |id

First( ) :

FIRST(E) ={ (, id}

FIRST(E’) ={+ , ε }

FIRST(T) = { (, id}

FIRST(T’) ={*, ε }

FIRST(F) ={ (, id }

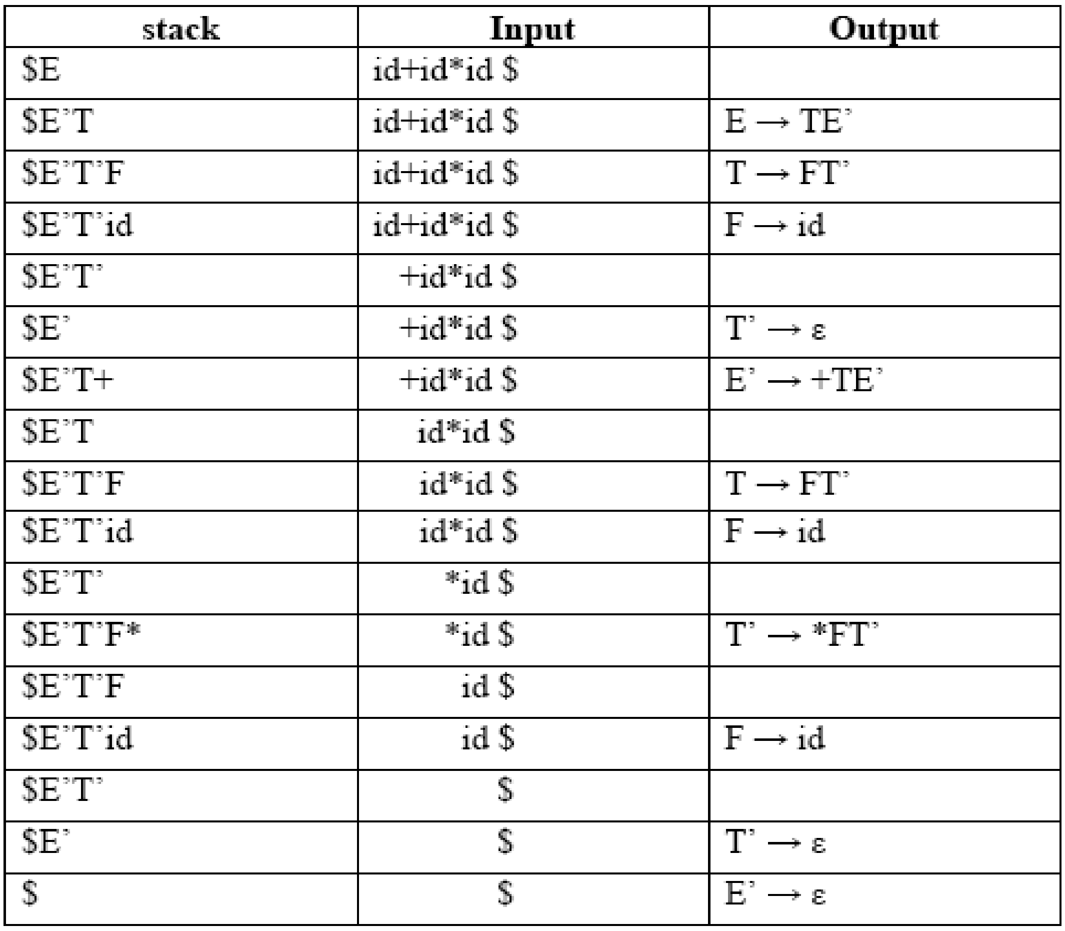
Follow( ) :

FOLLOW(E) = { $,) }
FOLLOW(E’) = { $,) }
FOLLOW(T) = { +, $,) }
FOLLOW(T’) = { +, $,) }
FOLLOW(F) = {+, * , $,) }

Predictive parsing table :

NONTERMINAL	**id**	**+**	*****	**(**	**)**	**$**
E	E → TE’			E → TE’		
E’		E’→+TE’			E’→ ε	E’→ ε
T	T → FT’			T → FT’		
T’		T’→ ε	T’→ *FT’		T’→ ε	T’→ ε
F	F → id			F → (E)		

Stack implementation:



stack	Input	Output
$E	id+id*id $	
$E'T	id+id*id $	E → TE'
$E'T'F	id+id*id $	T → FT'
$E'T'id	id+id*id $	F → id
$E'T'	+id*id $	
$E'	+id*id $	T' → ε
$E'T+	+id*id $	E' → +TE'
$E'T	id*id $	
$E'T'F	id*id $	T → FT'
$E'T'id	id*id $	F → id
$E'T'	*id $	
$E'T'F*	*id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

11. Consider Parsing table for the grammar and find moves made by predictive parser on input id+id*id and find FIRST and FOLLOW. (Nov/Dec 2016)

E → E+T |T
T → T*F |F
F→ (E) |id

After eliminating left-recursion the grammar is

E → TE'
E' → +TE' |ε
T → FT'
T' → *FT' | ε
F → (E) |id

First( ) :

FIRST(E) ={ (, id}

FIRST(E') ={+ , ε }

FIRST(T) = { (, id}

FIRST(T') ={*, ε }

FIRST(F) ={ (, id }

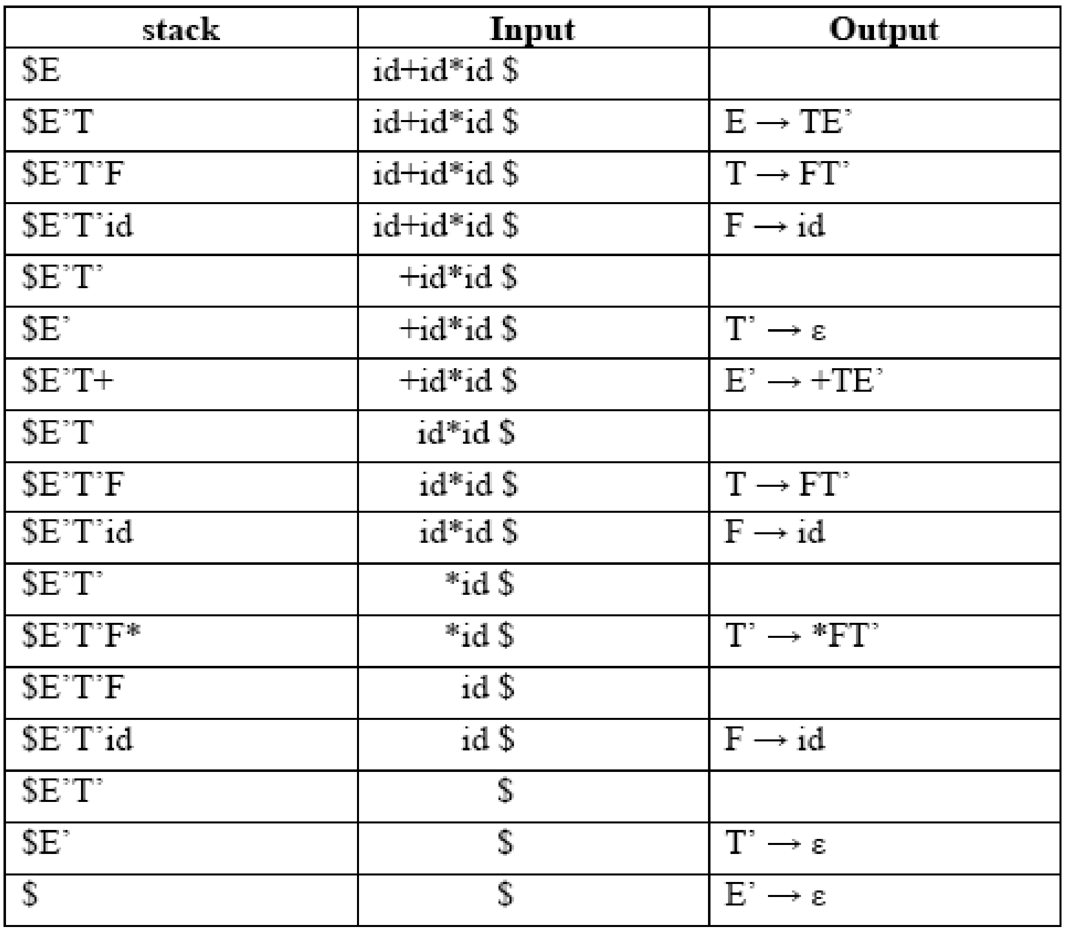
Follow( ) :

FOLLOW(E) = { \$,) }
FOLLOW(E') = { \$,) }
FOLLOW(T) = { +, \$,) }
FOLLOW(T') = { +, \$,) }
FOLLOW(F) = {+, * , \$,) }

Predictive parsing table :

NONTERMINAL	id	+	*	(	)	\$
E	E → TE'			E → TE'		
E'		E'→+TE'			E'→ ε	E'→ ε
T	T → FT'			T → FT'		
T'		T'→ ε	T'→ *FT'		T'→ ε	T'→ ε
F	F → id			F → (E)		

Stack implementation:



stack	Input	Output
\$E	id+id*id \$	
\$E'T	id+id*id \$	E → TE'
\$E'T'F	id+id*id \$	T → FT'
\$E'T'id	id+id*id \$	F → id
\$E'T'	+id*id \$	
\$E'	+id*id \$	T' → ε
\$E'T+	+id*id \$	E' → +TE'
\$E'T	id*id \$	
\$E'T'F	id*id \$	T → FT'
\$E'T'id	id*id \$	F → id
\$E'T'	*id \$	
\$E'T'F*	*id \$	T' → *FT'
\$E'T'F	id \$	
\$E'T'id	id \$	F → id
\$E'T'	\$	
\$E'	\$	T' → ε
\$	\$	E' → ε

12. Explain the LL(1) grammar : (8 marks) Or
Explain LL(1) grammar for the sentences S→iEts|iEtSeS|a E→b.(8) May/June 2016)

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.

Problem 1:

Consider this following grammar:

S → iEtS | iEtSeS | a
E → b

After eliminating left factoring, we have

S → iEtSS' |a
S'→ eS | ε
E → b

FIRST()

FIRST(S) = {i, a}
FIRST(S') = {e, ε}
FIRST (E) = {b}

FOLLOW ()
FOLLOW(S) ={ $,e }
FOLLOW(S') = { $,e }
FOLLOW(E) = {t}

Parsing table :

NONTERMINAL	**a**	**b**	**e**	**i**	**t**	**$**
S	S →a			S→iEtSS'		
S'			S' → eS S' → ε			S' → ε
E		E → b				

Problem 1:
S→CC
C→cC
C→d
Solution:

FIRST(S) = FIRST © ={c,d}
FOLLOW
C S C cC FOLLOW (C) → { FIRST (C)….} [by rule 2]

αB **Bβ**

FOLLOW (S) = FOLLOW (C)
FOLLOW (S) = { $}
FOLLOW (C) = { $, c,d }

FIRST(S) = c, d
FIRST(C) = c,d
FOLLOW (S) = $
FOLLOW (C) = $, c, d

Non terminal	**input symbol**		
	c	d	$
S	**S**→CC	**S**→CC	
C	**C**→cC	**C**Cd	

Example input : cdcd

	stack	**input**	**output**
(1)	$S	cdcd $	
(2)	$CC	Cdcd $	E→TE’
(3)	$CCc	Cdcd $	T→FT’
(4)	$CC	dcd $	F→ id
(5)	$Cd	dcd $	
(6)	$C	cd $	T’→ε
(7)	$Cc	cd $	E’→TE’
(8)	$C	d $	
(9)	$d	d $	T→FT’
(10)	$	$	F→ id

Accepted

LL(1) Grammars

- Algorithm covered in class can be applied to any grammar to produce a parsing table
- If parsing table has no multiply-defined entries, grammar is said to be “LL(1)”
    - First “L”, left-to-right scanning of input
    - Second “L”, produces leftmost derivation
    - “1” refers to the number of lookahead symbols needed to make decisions

Actions performed in predictive parsing:

1. Shift
2. Reduce
3. Accept
4. Error

Implementation of predictive parser:

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST () and FOLLOW () for all non-terminals.
3. Construct predictive parsing table.

4. Parse the given input string using stack and parsing table.

13. Explain the Bottom-Up Parsing. Or
Construct stack implementaion of shift reduce parsing parsing for the grammar
E → E+E
E → E*E
E → (E)
E → id
And the input string id1+id2*id3.(8) (May/June 2016)

Bottom-Up Parsing:

- Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.
- A general type of bottom-up parser is a **shift-reduce parser**.

SHIFT-REDUCE PARSING

- Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).
- The reductions trace out the right-most derivation in reverse.

Handles:

A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

Example:

Consider the grammar:

E → E+E
E → E*E
E → (E)
E → id
And the input string id1+id2*id3

The rightmost derivation is :

E → $\underline{\textbf{E+E}}$
→ E+$\underline{\textbf{E*E}}$
→ E+E*$\underline{\textbf{id3}}$
→ E+$\underline{\textbf{id2}}$*id 3
→ $\underline{\textbf{id1}}$+id2*id 3

In the above derivation the underlined substrings are called **handles.**

Handle pruning:
A rightmost derivation in reverse can be obtained by "**handle pruning**".
(i.e.) if *w* is a sentence or string of the grammar at hand, then $w = \gamma n$, where γn is the n^{th} right-sentinel form of some rightmost derivation.

Stack implementation of shift-reduce parsing :

Stack	Input	Action
\$	$id_1+id_2*id_3$ \$	shift
\$ id_1	$+id_2*id_3$ \$	reduce by E→id
\$ E	$+id_2*id_3$ \$	shift
\$ E+	id_2*id_3 \$	shift
\$ E+id_2	$*id_3$ \$	reduce by E→id
\$ E+E	$*id_3$ \$	shift
\$ E+E*	id3 \$	shift
\$ E+E*id3	\$	reduce by E→id
\$ E+E*E	\$	reduce by E→ E *E
\$ E+E	\$	reduce by E→ E+E
\$ E	\$	accept

Actions in shift -reduce parser:

- shift – The next input symbol is shifted onto the top of the stack.
- reduce – The parser replaces the handle within a stack with a non-terminal.
- accept – The parser announces successful completion of parsing.
- error – The parser discovers that a syntax error has occurred and calls an error recovery routine.

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift shift-reduce parsing:

- **Shift-reduce conflict**: The parser cannot decide whether to shift or to reduce.
- **Reduce-reduce conflict**: The parser cannot decide which of several reductions to make.

1. Shift-reduce_____conflict:

Example:

Consider the grammar:

E→E+E |E*E |id and input id+id*id

Stack	Input	Action	Stack	Input	Action
$ E+E	*id $	Reduce by E→E+E	$E+E	*id $	Shift
$ E	*id $	Shift	$E+E*	id $	Shift
$ E*	id $	Shift	$E+E*id	$	Reduce by E→id
$ E*id	$	Reduce by E→id	$E+E*E	$	Reduce by E→E*E
$ E*E	$	Reduce by E→E*E	$E+E	$	Reduce by E→E*E
$ E			$E		

2. Reduce-reduce conflict:

Consider the grammar:

M → R+R |R+c |R R → c
and input c+c

Stack	Input	Action	Stack	Input	Action
$	c+c $	Shift	$	c+c $	Shift
$ c	+c $	Reduce by R→c	$ c	+c $	Reduce by R→c
$ R	+c $	Shift	$ R	+c $	Shift
$ R+	c $	Shift	$ R+	c $	Shift
$ R+c	$	Reduce by R→c	$ R+c	$	Reduce by M→R+c
$ R+R	$	Reduce by M→R+R	$ M	$	
$ M	$				

Viable prefixes:

- α is a viable prefix of the grammar if there is *w* such that α*w* is a right sentinel form.
- The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes.
- The set of viable prefixes is a regular language.

14. Explain the LR Parsers

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(*k*) parsing.

The 'L' is for left-to-right scanning of the input, the
'R' for constructing a rightmost derivation in reverse, and the
'*k*' for the number of input symbols.
When '*k*' is omitted, it is assumed to be 1.

Advantages of LR parsing:

- It recognizes virtually all programming language constructs for which CFG can be written.
- It is an efficient non-backtracking shift-reduce parsing method.
- A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
- It detects asyntactic error as soon as possible.

Drawbacks of LR method:

It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

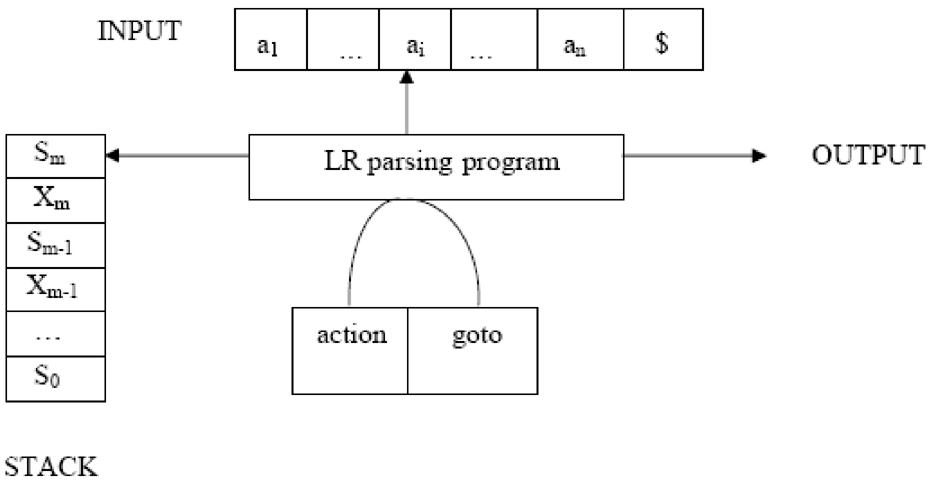
Cannot handle ambiguous grammar without special tricks

Types of LR parsing method:

1. SLR- Simple LR
    Easiest to implement, least powerful.
2. CLR- Canonical LR
    Most powerful, most expensive.
3. LALR- Look -Ahead LR
    Intermediate in size and cost between the other two methods.

The LR parsing algorithm:

The schematic form of an LR parser is as follows:



It consists of : an input, an output, a stack, a driver program, and a parsing table that has two parts (*action* and *goto*).

- The driver program is the same for all LR parser.
- The parsing program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form s0X1s1X2s2…Xmsm, where sm is on top. Each Xi is a grammar symbol and each si is a state.
- The parsing table consists of two parts : *action* and *goto* functions.

Action : The parsing program determines sm, the state currently on top of stack, and ai, the current input symbol. It then consults *action*[sm,ai] in the action table which can have one of four values :

- shift s, where s is a state,
- reduce by a grammar production A → β,
- accept, and
- error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

LR Parsing algorithm:

Input: An input string *w* and an LR parsing table with functions *action* and *goto* for grammar G.

Output: If *w* is in L(G), a bottom-up-parse for *w*; otherwise, an error indication.

Method: Initially, the parser has s0 on its stack, where s0 is the initial state, and *w*$ in the input buffer. The parser then executes the following program :

```
set ip to point to the first input symbol of w$;
repeat forever begin
    let s be the state on top of the stack and a the
      symbol pointed to by ip;

    if action[s, a] =shift s' then begin push a
      then s' on top of the stack; advance ip to
      the next input symbol
    end

    else if action[s, a]=reduce A→β then begin pop 2*
        |β |symbols off the stack;

        let s' be the state now on top of the stack; push A
        then goto[s', A] on top of the stack; output the
        production A→ β
    end
```

```
        else if action[s, a]=accept then return
        else error( )

    end
```

CONSTRUCTING SLR(1) PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:

- Find LR(0) items.
- Completing the closure.
- Compute *goto*(I,X), where, I is set of items and X is grammar symbol.

LR(0) items:

An *LR(0) item* of a grammar G is a production of G with a dot at some position of the right side. For example, production A → XYZ yields the four items :

A → **.** XYZ
A → X **.** YZ
A → XY **.** Z
A → XYZ **.**

Closure operation:

If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

- Initially, every item in I is added to closure(I).
- If A → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ αX . β] such that [A→ α . Xβ] is in I.

Steps to construct SLR parsing table for grammar G are:

- Augment G and produce G'
- Construct the canonical collection of set of items C for G'
- Construct the parsing action function *action* and *goto* using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G'
Output : The SLR parsing table functions *action* and *goto* for G'

Method :

1. Construct C = {$I_0, I_1, \dots I_n$}, the collection of sets of LR(0) items for G'.
2. State *i* is constructed from I_i. The parsing functions for state *i* are determined as follows:
   (a) If [$A \to \alpha \cdot a\beta$] is in I_i and goto(I_i,*a*) = I_j, then set *action*[*i*,*a*] to "shift j". Here *a* must be terminal.
   (b) If [$A \to \alpha \cdot$] is in I_i , then set *action*[*i*,*a*] to "reduce $A \to \alpha$" for all *a* in FOLLOW(A).
   (c) If [$S' \to S.$] is in I_i, then set *action*[*i*,$] to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The *goto* transitions for state *i* are constructed for all non-terminals 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 [$S' \to .S$].

Example for SLR parsing:

Construct SLR parsing for the following grammar :

G : $E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

The given grammar is :

G :	$E \rightarrow E + T$	------ (1)
	$E \rightarrow T$	------ (2)
	$T \rightarrow T * F$	------ (3)
	$T \rightarrow F$	------ (4)
	$F \rightarrow (E)$	------ (5)
	$F \rightarrow id$	------ (6)

Step 1 : Convert given grammar into augmented grammar.
Augmented grammar :

$E' \rightarrow E$
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

Step 2 : Find LR (0) items.

I_0 : $E' \rightarrow . E$
$E \rightarrow . E + T$
$E \rightarrow . T$
$T \rightarrow . T * F$
$T \rightarrow . F$
$F \rightarrow . (E)$
$F \rightarrow . id$

GOTO (I_0 , E)
I_1 : $E' \rightarrow E .$
$E \rightarrow E . + T$

GOTO (I_4 , id)
I_5 : $F \rightarrow id .$

GOTO (I_0 , T)
I_2 : E $\rightarrow$ T .
T $\rightarrow$ T . * F

GOTO (I_0 , F)
I_3 : T $\rightarrow$ F .

GOTO (I_0 , ( )
I_4 : F $\rightarrow$ (. E)
E $\rightarrow$. E + T
E $\rightarrow$. T
T $\rightarrow$. T * F
T $\rightarrow$. F
F $\rightarrow$. (E)
F $\rightarrow$. id

GOTO (I_0 , id)
I_5 : F $\rightarrow$ id .

GOTO (I_1 , +)
I_6 : E $\rightarrow$ E + . T
T $\rightarrow$. T * F
T $\rightarrow$. F
F $\rightarrow$. (E)
F $\rightarrow$. id

GOTO (I_2 , *)
I_7 : T $\rightarrow$ T * . F
F $\rightarrow$. (E)
F $\rightarrow$. id

GOTO (I_4 , E)
I_8 : F $\rightarrow$ (E .)
E $\rightarrow$ E . + T

GOTO (I_4 , T)
I_2 : E $\rightarrow$ T .
T $\rightarrow$ T . * F

GOTO (I_4 , F)
I_3 : T $\rightarrow$ F .

GOTO (I_6 , T)
I_9 : E $\rightarrow$ E + T .
T $\rightarrow$ T . * F

GOTO (I_6 , F)
I_3 : T $\rightarrow$ F .

GOTO (I_6 , ( )
I_4 : F $\rightarrow$ (. E)

GOTO (I_6 , id)
I_5 : F $\rightarrow$ id .

GOTO (I_7 , F)
I_{10} : T $\rightarrow$ T * F .

GOTO (I_7 , ( )
I_4 : F $\rightarrow$ (. E)
E $\rightarrow$. E + T
E $\rightarrow$. T
T $\rightarrow$. T * F
T $\rightarrow$. F
F $\rightarrow$. (E)
F $\rightarrow$. id

GOTO (I_7 , id)
I_5 : F $\rightarrow$ id .

GOTO (I_8 ,))
I_{11} : F $\rightarrow$ (E) .

GOTO (I_8 , +)
I_6 : E $\rightarrow$ E + . T
T $\rightarrow$. T * F
T $\rightarrow$. F
F $\rightarrow$. (E)
F $\rightarrow$. id

GOTO (I_9 , *)
I_7 : T $\rightarrow$ T * . F
F $\rightarrow$. (E)
F $\rightarrow$. id

GOTO (I_4 , ( )

I_4 : F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → id

FOLLOW (E) = { $,) , +)
FOLLOW (T) = { $, + ,) , * }
FOOLOW (F) = { * , + ,) , $ }

SLR parsing table:

	ACTION						GOTO		
	id	+	*	(	)	**$**	**E**	**T**	**F**
I_0	s5			s4			1	2	3
I_1		s6				ACC			
I_2		r2	s7		r2	r2			
I_3		r4	r4		r4	r4			
I_4	s5			s4			8	2	3
I_5		r6	r6		r6	r6			
I_6	s5			s4				9	3
I_7	s5			s4					10
I_8		s6			s11				
I_9		r1	s7		r1	r1			
I_{10}		r3	r3		r3	r3			
I_{11}		r5	r5		r5	r5			

Blank entries are error entries.

Stack implementation:

Check whether the input **id** + **id** * **id** is valid or not.

STACK	INPUT	ACTION
0	id + id * id $	GOTO (I_0 , id) = s5 ; **shift**
0 id 5	+ id * id $	GOTO (I_5 , +) = r6 ; **reduce** by F→id
0 F 3	+ id * id $	GOTO (I_0 , F) = 3 GOTO (I_3 , +) = r4 ; **reduce** by T → F
0 T 2	+ id * id $	GOTO (I_0 , T) = 2 GOTO (I_2 , +) = r2 ; **reduce** by E → T
0 E 1	+ id * id $	GOTO (I_0 , E) = 1 GOTO (I_1 , +) = s6 ; **shift**
0 E 1 + 6	id * id $	GOTO (I_6 , id) = s5 ; **shift**
0 E 1 + 6 id 5	* id $	GOTO (I_5 , *) = r6 ; **reduce** by F → id
0 E 1 + 6 F 3	* id $	GOTO (I_6 , F) = 3 GOTO (I_3 , *) = r4 ; **reduce** by T → F
0 E 1 + 6 T 9	* id $	GOTO (I_6 , T) = 9 GOTO (I_9 , *) = s7 ; **shift**
0 E 1 + 6 T 9 * 7	id $	GOTO (I_7 , id) = s5 ; **shift**
0 E 1 + 6 T 9 * 7 id 5	$	GOTO (I_5 , $) = r6 ; **reduce** by F → id
0 E 1 + 6 T 9 * 7 F 10	$	GOTO (I_7 , F) = 10 GOTO (I_{10} , $) = r3 ; **reduce** by T → T * F
0 E 1 + 6 T 9	$	GOTO (I_6 , T) = 9 GOTO (I_9 , $) = r1 ; **reduce** by E → E + T
0 E 1	$	GOTO (I_0 , E) = 1 GOTO (I_1 , $) = **accept**

15. Give an algorithm for finding the FIRST and Follow positions for a given non terminal. (Nov\ Dec – 2005)

FIRST:

If α is any string of grammar symbols, let FIRST(α) be the set of terminals that begins the strings derived from α. If α ⇒ ε, then ε is FIRST (α).

Rules for computing FIRST(X)

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set.

1. If X is terminal, then FIRST(X) is {X}
2. If X → ε is a production, then add ε to FIRST(X)
3. If X is nonterminal and X→ Y_1, Y_2 … Y_k is a production, then place a in FIRST(X) if for some i, a is in first(Y_i). if ε is in first(Y_j) for all j= 1, 2, … k then add ε to first(X)

FOLLOW:

FOLLOW(A) for nonterminal A to be the set of nonterminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exist a derivation of the form $S \Rightarrow^* \alpha A a \beta$ for some α and β. If A can be the rightmost symbol in some sentential form, then \$ is in FOLLOW (A).

Rules for computing FOLLOW (A):

To compute FOLLOW(A) for all grammar symbols A, apply the following rules until no more terminals or ε can be added to any FOLLOW set.

1. Place \$ in FOLLOW (S), where S is the start symbol and \$ is the input right end marker.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for ε is placed in follow(B).
3. if there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε (i.e., $\beta \Rightarrow^* \varepsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

16. Briefly explain error recovery in LR parsing (Nov\ Dec – 2005)

Panic mode Error Recovery:

It is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. The synchronizing set should be chosen so that the parser recovers quickly from error that are likely to occur in practice . Some Heuristics are as follows:

1. We can place all symbols in Follows (A) into the synchronizing set for nonterminal A. If we skip tokens until an element of follow(A) is seen and pop A from the stack, it is likely that parsing can continue.

2. It is not enough to use FOLLOW(A) as the synchronizing set for A. Often ,there is a hierarchical structure on constructs in a language eg, expressions appears within statements, which appears within blocks and so on. We can add to the synchronizing set of a lower constructs the symbols that begins higher constructs the symbols that begin higher constructs.

3.If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

4.If a non terminal can generate the empty string ,then the production deriving can be used as a default. This approach reduces a number of nonterminals that have to be considered during error recovery.

5. If a terminal on the top of the stack cannot be matched, a simple idea is to pop the terminal issue a message saying that the terminal was inserted and continued parsing.

Example :

The parsing table for the grammar :

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$T' \rightarrow FT'$

$T' \rightarrow *FT|\varepsilon$

$F \rightarrow (E)|id$

with “synch” indicating synchronizing tokens obtained from the follow set of the nonterminal.

The parsing table is shown in below table.

non terminal	**input symbol**					
	id	+	*	**(**	**)**	**$**
E	E→TE’			E→TE’	Synch	Synch
E’		E’→+TE’			E’→ε	E’→ε
T	T→FT’	Synch		T→FT’	synch	Synch
T’		T→ε	T’→*FT’		T’→ε	T’→ε
F	F→id	Synch	Synch	F→(E)	Synch	Synch

Synchronizing tokens added to parsing table

If the parser looks up the entry M[A, a] and finds that it is blank, then the input symbol a is skipped. If the entry is synch, then the nonterminal on the top of the stack is popped in an attempt to resume parsing. If a token on the top of the stack does not match the input symbol, then we pop the token from the stack.
On erroneous input) id*+ id the parser and error recovery mechanism behave as

stack	**input**	**remark**
$ E	) id * + id $	error, skip)
$ E	id * + id $	id is in FIRST(E)
$ E ’ T	id * + id $	
$ E ’ T ’F	id * + id $	
$ E ’ T ’ id	id * + id $	
$ E ‘T’	* + id $	
$ E ‘T ‘ F *	* + id $	
$ E ‘ T ‘ F	+ id $	error, M[F,+]=synch
$ E ‘ T ‘	+ id $	F has been popped
$ E ‘	+ id $	
$ E ‘ T +	+ id $	
$ E ‘ T	id $	
$ E ‘ T ‘ F	id $	
$ E ‘ T ‘ id	id$	
$ E ‘ T ‘	$	
$ E ‘	$	

Parsing and error recovery moves made by predictive parser

Phrase –level recovery:

It is implemented by implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. The routines may change, insert or delete symbols on the input and issue appropriate error message.

17. Explain the LR parsing algorithm with example. (Apr/May – 2005)

INPUT: An input string w and LR parsing table with functions actions and goto for a grammar G.

OUTPUT: If w is in L(G), a bottom up parser for w otherwise an error indication.

METHOD: Initially the parser has S0 on its stack, where S0 is in the initial state and w$ in the input buffer. The parser then executes the program in until an accept or error action is encountered.

```
Set ip to point to the first symbol of w$;
repeat forever begin
        let S be the state on top of the stack and a the symbol pointed to by ip;
if action[s,a]=shift S' then begin
        push a then state on top of the stack;
        advance ip to the next input symbol
end
else if action[S,a]=reduce A→β then begin
        pop 2*|β| symbols off the stack;
        lets S' be the state now on top of the stack;
        push A then goto [S' , A] on top of the stack;
        output the production A→β
end
else if action[S,a]=accept then
        return
else
        error()
end
```

LR Parsing program

Example:

This shows the parsing action and goto functions of LR parsing table for the following grammar for the arithmetic expression with binary operators +,*.

(1) $E \rightarrow E+T$
(2) $E \rightarrow T$
(3) $T \rightarrow T*F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow id$

State	**action**						**Goto**		
	Id	+	*	(	)	$	E	T	F
0	S	5		S4			1	2	3
1		S6				acc			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			

6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Parsing table for expression grammar

The codes for the actions are

1. Si means shift and stack I
2. rj means reduce by production numbered j
3. acc means accept
4. blank means error

Parse the input id * id + id

	stack	**Input**	action
(1)	0	id * id + id $	shift
(2)	0 id 5	* id + id $	reduce by F$\rightarrow$id
(3)	0 F 3	* id + id $	reduce by T$\rightarrow$F
(4)	0 T 2	* id + id $	shift
(5)	0 T 2 * 7	id + id $	shift
(6)	0 T 2 * 7 id 5	+ id $	reduce by F$\rightarrow$id
(7)	0 T 2 * 7 F 10	+ id $	reduce by T$\rightarrow$T*F
(8)	0 T 2	+ id $	reduce by E$\rightarrow$T
(9)	0 E 1	+ id $	shift
(10)	0 E 1 + 6	id $	shift
(11)	0 E 1 + 6 id 5	$	reduce by F$\rightarrow$id
(12)	0 E 1 + 6 F 3	$	reduce by T$\rightarrow$F
(13)	0 E 1 + 6 T 9	$	E$\rightarrow$E+T
(14)	0 E 1	$	accept

Moves of LR parser on id * id + id

1. action [0,id]=S5,shift the id and cover the satack with 5.
2. action [5,*]=r6,reduce by 6th production ie,F$\rightarrow$id,pop 2 symbnols(id and 5) from the stack, push F then goto [0,F]ie,3 on top of the stack.
3. action [3,*]=r4,reduce by T$\rightarrow$F ie, 4th production pop2 symbols (F and 3) from the stack.push T goto [0,T]ie,2 onto the stack.
4. The process repeated until an accept or error action is encountered

18. Construct the predictive parser for the following grammar (Apr/May – 2010) (Apr/May – 2005)

S $\rightarrow$ a | $\uparrow$ | (T)
T $\rightarrow$ T, S | S

Write down the necessary algorithms and define FIRST and FOLLOW. Show the behaviour of the parser in the sentences:

(i) **(a, (a,a))**
(ii) **(((a,a), $\uparrow$, (a), a)**

Solution:

1. Eliminate left recursion from the grammar (1). The non-recursive grammer is

$S \rightarrow a \mid \uparrow \mid (T)$
$T \rightarrow ST'$
$T' \rightarrow\ , ST' \mid \varepsilon$

2. Compute FIRST (x) for all grammar symbols.
   FIRST (S) = { a, ↑ , (}
   FIRST (T) = { a, ↑ , (}
   FIRST (T') = { , , ε}
3. Compute FOLLOW(A) for all non terminals A by using FOLLOW rules.
   FOLLOW(S) = {$,) }
   FOLLOW(T) = {) }
   FOLLOW(T') = {) }
4. The parsing table for this grammar is constructed using the "Construction of a predictive Parsing table" Algorithm.

Nonterminal	**Input symbol**					
	a	↑	(	)	,	$
S	S → a	S→ ↑	S → (T)			
T	T → ST'	T → ST'	T → ST'			
T'				T' → ε	T' → ,ST'	

Parsing Table for M for grammar (2)

(i) Parse the sentence (a, (a, a))

STACK	**INPUT**	**SYMBOL**
$ S	(a, (a, a))$	S→ (T)
$) T	a, (a, a))$	T → ST'
$) T ' S	a, (a, a))$	S → a
$) T '	, (a, a))$	T'→,ST'
$) T ' S	(a, a))$	S → (T)
$) T ') T	a, a))$	T → ST'
$) T ') T ' S	a, a))$	S → a
$) T ') T '	, a))$	T' → , ST'
$) T ') T ' S	a))$	S → a
$) T ') T'	))$	T' → ε
$) T ')	))$	
$) T '	)$	T' → ε
$)	)$	
$	$	

The given sentence (a, (a, a)) is successfully parsed using the grammar (2).

(ii) Parse the sentence (((a, a), ↑, (a), a)

STACK	**INPUT**	**SYMBOL**
$ S	(((a, a), ↑, (a), a) $	S → T
$ (T	((a, a), ↑, (a), a) $	T → ST'
$ (T' S	((a, a), ↑, (a), a) $	S → (T)
$) T') T	(a, a), ↑, (a), a) $	T → ST'
$) T') T' S	(a, a), ↑, (a), a) $	S→ST'

$) T') T') T	a, a), ↑, (a), a) $	T→ ST'
$) T') T') T' S	a, a), ↑, (a), a) $	S → a
$) T') T') T'	, a), ↑, (a), a) $	T' → , ST'
$) T') T') T' S	a), ↑, (a), a) $	S → a
$) T') T') T'	), ↑, (a), a) $	T→ ε
$) T') T')	), ↑, (a), a) $	
$) T') T'	, ↑, (a), a) $	T' → ,ST'
$) T') T') S	↑, (a), a) $	S → ↑
$) T') T'	, (a), a) $	T' → , ST'
$) T') T' S	(a), a) $	S → (T)
$) T') T') T	a), a) $	T → ST'
$) T') T') T' S	a), a) $	S → a
$) T') T') T'	), a) $	T' → ε
$) T') T')	), a) $	
$) T') T'	, a) $	T' → , ST'
$) T') T') S	a) $	S → a
$) T') T'	) $	T' → ε
$) T')	) $	
$) T'	$	

Since the stack contains grammar symbol but no input symbol to parse. It implies that the given sentence (((a, a),↑, (a), a) cannot be parsed.

19. What is Shift reducer parser? Explain in detail the conflict that may occurring Shift reducer parsing. May/Jun -2012(Nov\ Dec – 2005)

A way to implement a shift reduce parser is to use a stack to hold grammar symbols and an input buffer to hold the string w to be parsed.

We use $ to mark the bottom of the stack and also the right end of the input .Initially the stack is empty and the atring w is on the input as follows:

STACK	INPUT
$	w$

The parser operates by shifting zero or more inputs symbols onto the stack onto the stack until a handle β is on the top of the stack. The parser then reduces β to the left side of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack the stack contains the start symbols and the input is empty.

STACK	INPUT
$S	$

After entering this configuration, the parser halts and announces successfully completion of parsing.

while the primary operation of the parser are shift and reduces , there are actually four possible actions a shift reduce parser can make: They are

(1) Shift - In a shift action, the next input symbol is shifted onto the top of the stack.

(2)reduce -In a reduce action, the parser knows the right and of the handle is of the top of the stack . It must then located the left end of the handle within the stack and decide with what non terminal to replace the handle.

(3)Accept -In an accept action, the parser announces successfully completion of the parsing.

(4) error -In a error action , the parser discovers that a syntax error has occurred and calls an error recovery routine.

Viable prefixes:

A viable prefix is that it is a prefix of the right sentential form that does not continue past the right end of the rightmost handle of that sentential form.

Conflicts during shift reduce parsing: (6 Marks) (Apr /May 2011)

There are context free grammar for which shift-reduce parsing cannot be used. In which the parser, knowing the entire stack contents and the next input symbol cannot decide whether to shift or to reduce (a shift/reduce conflict) or cannot decide which of the several reductions to make (a reduce/reduce conflict). These grammar are not in LR(k) class of grammar.

LR(k)

(k)-The number of symbols of lookahead on the input

An ambiguous grammar can never be LR.

(i) shift /reduce conflict : or cannot decide which of the several reductions to perform

(ii). Reduce/Reduce conflict: theses types of grammars are called non –LR- grammars, Ambiguous grammars are not LR grammars

20. Check whether the following grammar is SLR (1) or not . Explain Your answer with reasons (8 Marks) Apr/May – 2004

$S \rightarrow L=R$
$S \rightarrow R$
$L \rightarrow * R$
$L \rightarrow id$
$R \rightarrow L$

Solution:

Follow (S) = {\$}
Follow (L) = {=, Follow (R)}
= {=,\$ }
Follow (R) = Follow (s) + Follow (L)
= { \$ } => { =,\$}

Step 1

Equivalent augmented grammar G is

$S' \rightarrow .S$
$S \rightarrow L=R$
$S \rightarrow R$
$L \rightarrow * R$
$R \rightarrow L$

Step 2

Canonical collection of LR (0) items

Initially={$S' \rightarrow .S$}

I_0 = Closure (1)=closure{$S' \rightarrow .S$}

: $S' \rightarrow .S$
$S \rightarrow .L=R$
$S \rightarrow .\ R$
$L \rightarrow .\ * R$

$L \rightarrow .\ id$

I_0: $R \rightarrow .\ L$

I_1 : goto (I_0, S) : S’→S**.**

I_2 : goto(I_0, L) : L→L **.** = R

R→L **.**

I_3 : goto (I_0, R) : S→R **.**

I_4 : goto (I_0, *) : L→* **.** R

R→**.** L

L→ **.** * R

L→**.** id

I_5 : goto (I_0, id) : L→id **.**

I_1 : null set

I_2:

I_6 : goto (I_2, =) : S→L= **.** R

R→ **.** L

→**.** * R

L→**.** id

I_3 : null set

I_4 :

I_7 : goto (I_4, L) : => R→ L**.**

I_8 : goto (I_4, R) : L→*R **.**

I_4 : goto (I_4, *) : L→* **.** R

R→**.** L

L→**.** * R

L→ **.** id

I_5 : goto (I_4, id) : L→id **.**

I_5 : null set

I_6:

I_8: goto (I_6, L) : R→L **.**

I_9 : goto (I_6, R) : S→=R**.**

I_4 : goto (I_6, *) : L→* **.** R

R→ **.**L

L→ **.** * R

L→**.** id

16. I_5: goto (I_6, id) : L→id **.**

I_7 : null set

I_8 : null set

I_9 : null set

s

State	**action**				**Goto**		
	=	*****	**id**	**$**	**S**	**L**	**R**
0		S4	S5		1	2	3
1				Accept			
2	r5 s6			r5			
3	r2			r2			
4		S4	S5			7	8
5	r4			r4			
6		S4	S5			7	9
7	r5			r5			
8	r3			r3			
9	r1			r1			

The above grammar is not SLR (1)
Thus entry action [2, =] is multiply defined since there is both a shift and a reduce entry in action [2, =], state 2 has a shift/reduce conflict on input symbol. Above grammar is not ambiguous.

21. Consider the grammar (Nov\ Dec – 2005)

$E \rightarrow TE'$

$E' \rightarrow +TE' / \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \varepsilon$

$F \rightarrow (E)/id$

Construct the predictive parsing table for the above grammar & verify whether the input string id + id * id is accepted by the grammar or not.

Solution

FIRST(E) = FIRST(T) = FIRST(F) = { (,id}; By rule (3) of FIRST
FIRST(E') = {=, ε}
FIRST(T') = {*, ε}
FOLLOW(E) = {$,)} ; By rule (1) of follow
FOLLOW(E') = FOLLOW(E) ; By rule (3) of follow = {$,) }
FOLLOW(T) = FOLLOW(T') = { +,), $}
FOLLOW(F) = {), + , * , $}

Non terminal	**input symbol**					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$		$F \rightarrow (E)$			

Parsing table M for grammar above

- Blanks are error entries
- Non Blanks indicate a production with which to expand the top non-terminal on the state.

Parse the input id + id * id using non recursive predictive parser

1. Initially the stack contains start symbol. i.e., E

   Stack : E   input : id + id * id
                                              ↑

   The input pointer points to the leftmost symbol of the string
2. Here the top of the stack is nonterminal i.e., E. Program refers the table entry M[E, id], its equivalent production is $E \rightarrow TE'$,

Stack	***Input***	***Output***
$E	id + id * id $	

	↑	
$E’T	id + id * id $ ↑	E ↑ TE’
$E’T’F	id + id * id $ ↑	T ↑ FT’
$E’T,id	id + id * id $ ↑	F→id
$E’T’	+ id * id $ ↑	
$E’	+ id * id $ ↑	T’ → ε

Moves made by predictive parsing on input id + id * id

this productive is pushed onto the stack. Now ,

stack= E ‘ T   Input: id + id * id $
                                   ↑

3.Again the top of the stack is non terminal i.e, T .It is not matched with input.Find the table entry M[T,id],its equivalent production T→FT ‘.It is pushed onto the stack.now,

stack= E ‘ T ‘ F   Input: id + id * id $
                                   ↑

4.Now also the top of the stack is non terminal i.e, F parser refers the table entry M[F,id],its production F→id is pushed onto the stack.Now,

stack=E ‘ T ‘ id   Input= id + id * id $
                                   ↑

5. Now top of the stack i.e.,id and input symbol is matched . The parser pops the top of the stack i.e, id from the stack and increments the inpuit pointers

stack= E ‘ T ‘   Input= + id * id $
                                   ↑

6. This process is repeated until the end of the input string (ie, $).

22. Consider the following grammar. (Nov\ Dec – 2005)

E→E+T|T
T→T*F|F
F→(E)|id

Construct an LR parsing table for the following grammer. View the moves of LR parser on id * id + id.

Solution:

State	**action**						**Goto**		
	Id	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	S5			S4			1	2	3
1		S6				acc			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10

8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Parsing table for expression grammar

The codes for the actions are

1. Si means shift and stack I
2. Rj means reduce by production numbered j
3. Acc means accept
4. blank means error

parse the input id * id + id

	Stack	Input	Action
(1)	0	id * id + id $	shift
(2)	0 id 5	* id + id $	reduce by F→id
(3)	0 F 3	* id + id $	reduce by T→F
(4)	0 T 2	* id + id $	shift
(5)	0 T 2 * 7	id + id $	shift
(6)	0 T 2 * 7 id 5	+ id $	reduce by F→id
(7)	0 T 2 * 7 F 10	+ id $	reduce by T→T*F
(8)	0 T 2	+ id $	reduce by E→T
(9)	0 E 1	id $	shift
(10)	0 E 1 + 6	id $	shift
(11)	0 E 1 + 6 id 5	$	reduce by F→id
(12)	0 E 1 + 6 F 3	$	reduce by T→F
(13)	0 E 1 + 6 T 9	$	E→E+T
(14)	0 E 1	$	accept

Moves of LR parser on id * id + id

1. Action [0,id]=S5,shift the id and cover the satack with 5.
2. Action [5,*]=r6,reduce by 6th production ie,F→id,pop 2 symbnols(id and 5) from the stack, push F then goto [0,F]ie,3 on top of the stack.
3. Action [3,*]=r4,reduce by T→F ie, 4th production pop2 symbols (F and 3) from the stack. push T goto [0,T]ie,2 onto the stack.
4.The process repeated until an accept or error action is encountered.

23. Construct the predictive parser for the following grammar (May/Jun -2012, Apr/May – 2005)(May/June-2013)

S→(L)|a
L→L;S|S

Solution:

1. Eliminate the left recursion from the grammar(3).The equivalent non recursive grammar is
S→(L)|a
L→SL'--------------------- {4}
L'→,SL'|ε

2. Compute first(x) FOR ALL THE GRAMMAR SYMBOLS x

first(S)={$, ,)}
first(L)={(, a}
first(L)={, , ε}

3. Compute follow (A) for all non terminals A by using follow rules.
follow(S)={$, , ,)}
follow(L)={)}
follow(L')={)}

4. The parsing table for this grammar is constructed using the "Construction of a predictive table" Algorithm.

non terminal	input symbol				
	a	(	)	,	$
S	S→a	S→(L)			
L	L→SL'	L→SL'			
L'			L'→ε	L'→,SL'	

Parsing table for M for the grammar

construct the behavior of the parser on the sentence (a,a) using the grammar specified above.

STACK		INPUT	ACTION
$ S		(a, a)$	
$)L**(**	=	**(**a, a)$	S→ (L)
$)L		a, a)$	L→SL'
$)L'S		a, a)$	S→ a
$)L'**a**	=	**a**, a)$	
$)L'		, a)$	L'→,SL'
$)L'S **,**	=	**,** a)$	
$)L'S		a)$	S → a
$)L'**a**	=	**a**)$	
$)L'		)$	L' → ε
$**)**	=	**)**$	
$		$	Accept

24. Construct LR (0) parsing table for the given grammar (Nov/Dec – 2010)

1. **E→E*B**
2. **E→E +B**
3. **E→B**
4. **B→0**
5. **B→1**

Solution

LR (0) parsing table for the grammar

1. $E \rightarrow E*B$
2. $E \rightarrow E+B$
3. $E \rightarrow B$
4. $B \rightarrow 0$
5. $B \rightarrow 1$

Step 1:

Augmented Grammar

E ‘$\rightarrow$ E

E$\rightarrow$E*B

E$\rightarrow$E +B

E$\rightarrow$B

B$\rightarrow$0

B$\rightarrow$1

Step 2:

Canonical collection of LR (0) items

I_0: E ‘$\rightarrow$. E

E$\rightarrow$. E * B

E$\rightarrow$. E +B

E$\rightarrow$. B

B$\rightarrow$. 0

B$\rightarrow$ .1

Go to (I_0 , E)

I_1: E ‘$\rightarrow$ E .

E$\rightarrow$ E * B .

E$\rightarrow$ .E +B .

Go to (I_0 , B)

I_2: E $\rightarrow$ B

Go to (I_0 , 0)

I_3: E $\rightarrow$ 0

Go to (I_0 , 1)

I_4: E $\rightarrow$ 1

Go to (I_1 , *)

I_5 : E$\rightarrow$E * . B

B$\rightarrow$. 0

B$\rightarrow$ .1

Go to (I_0 , +)

I_6: E→E + . B

B→ . 0

B→ .1

Go to (I_5 , B)

I_7: E→ E * B .

Go to (I_5 , 0) : I_3

Go to (I_5 , 1) : I_4

Go to (I_6 , B)

I_8 : E→E + B .

Go to (I_6, 0): I_3

Go to (I_6, 1): I_4

State: I_0 f_0 I_8

FIRST (E): {0, 1}

FIRST (B): {0, 1}

FOLLOW (E) = {*, +}

FOLLOW (B) = {*, +}

LR (0) Parsing Table

State	action					Go to	
	*	+	0	1	$	E	B
0			S_3	S_4		1	2
1	S_5	S_6			Accept		
2	r_3	r_3					
3	r_4	r_4					
4	r_5	r_5					
5			S_3	S_4			7
6			S_3	S_4			8
7	r_1	r_1					
8	r2	r2					

Parsing table for expression grammar

Since all the entries are unique, the grammar is LR(0) Grammar 0r SLR Grammar

25.Write an algorithm for the construction of LR(1) or CLR items for grammar G. Nov/Dec – 2010(May/June-2013)

The general form of LR (1) item is

S-> X.Y, a;

A where ‘a’ is called look ahead this is an extra information. We are looking a character ahead. The ‘a’ may be terminal or the right end marker $

Example:

S^1->. S , $

$ is a look ahead

Here we use "closure "& "goto" functions for constructing LR (1) items taking look aheads in to account

Closure operation:

If I is a set of items for a grammar G, then closure (I) is the set of items constructed from I by the two rules:

Initially, every item in I is added to closure(I).

If A → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Example:

1. S -> AS
2. S -> b
3. A -> SA
4. A ->a

Solution

```
FIRST (S) = FIRST (A) = {a,b}
FOLLOW (S)          = { $ + FIRST (A) }
                      = {$, a, b}
FOLLOW (A)          = FIRST (S)
                      = {a , b}
```

Step 1:
Augmented Grammar

S' -> S

S -> AS

S -> b

A -> SA

A ->a

Step 2:

Canonical collection of LR (0) items

Closure of (S' -> . S)

S' ->. S

S ->. AS

S ->. b

A -> . SA

A -> . a

Canonical collection of LR (0) items

Closure of (S’ -> . S)

S’ ->. S

S ->. AS

S ->. b

A -> . SA

A -> . a

26. Given the following grammar S - > ASb, A -> SA |a construct a SLR parsing table for the string baab (May/ Jun – 2009)

1. S -> AS
2. S -> b
3. A -> SA
4. A ->a

Solution

FIRST (S) = FIRST (A) = {a,b}

FOLLOW (S) = { \$ + FIRST (A) }

= {\$, a, b}

FOLLOW (A) = FIRST (S)

= {a , b}

Step 1:

Augmented Grammar

S’ -> S

S -> AS

S -> b

A -> SA

A ->a

Step 2:

Canonical collection of LR (0) items

Closure of (S’ -> . S)

S’ ->. S

S ->. AS

S ->. b

A -> . SA

A -> . a

I_0 :

Goto (I_o , S) => S’ -> S.

S’ -> S.

S -> A. S

S ->. AS

A -> . SA

S ->. AS

Goto (I_o , A) => S -> A. S

S ->. AS

S ->. b

A -> . SA

A -> . a

Goto (I_o , a) => A -> a .
Goto (I_o , b) => A -> b .

I_1 :

Goto (I_1 , S) => S -> S . **A**

A -> . SA

A -> . a

S ->. AS

S ->. b

Goto (I_1 , A) => A -> SA.

S -> A. S

S ->. AS

S ->. b

A -> . SA

A -> . a

Goto (I_1 , a) => A -> a .
Goto (I_1 , b) => A -> b .

I_2

Goto (I_2 , S) => S -> AS .

S -> S . **A**

S ->. AS

S ->. b

A -> . SA

A -> . a

Goto (I_2 , A) => S -> A . S

S ->. AS

S ->. b

A -> . SA

A -> . a

Goto (I_2 , a) => A -> a .
Goto (I_2 , b) => A -> b .

I_3 :

=> Null set

I_4 :

=> Null set

I_5 :
Goto (I_5 , S) => A -> A . S

A -> . SA

A -> . a

S ->. AS

S ->. b

Goto (I_5 , A) => A -> SA .

A -> . SA

A -> A . S

A -> . a

S ->. AS

S ->. b

Goto (I_5 , a) => A -> a .
Goto (I_5 , b) => A -> b .
I_6:
Goto ($I_{6,}$ S) => A -> S . A

A -> . SA

A -> . a

S ->. AS

A -> A S .

S ->. b

Goto ($I_{6,}$ A) => A -> A . S

S ->. AS

S ->. b

A -> . SA

A -> . a

Goto (I_6 , a) => A -> a .
Goto (I_6 , b) => A -> b .

I_7:
Goto ($I_{7,}$ S) => A -> S . A

A -> . SA

A -> . a

S ->. AS

S ->. b

Goto ($I_{7,}$ A) => A -> SA .

A -> S . A
S ->. AS

A -> . a

S ->. b

A -> . SA

Goto (I_7 , a) => A -> a .
Goto (I_7 , b) => A -> b .

LR (0) Parsing Table

States	**action**			**Go to**	
	a	**b**	**$**	**S**	**A**
0	S_3	S_4		1	2
1	S_3	S_4	accept	5	6
2	S_3	S_4		7	2
3	r_4	r_4			
4	r_2	r_2	r_2		
5	S_3	S_4		5	6
6	S_3 r_3	S_4 r_3	S_3	S_4	
7	S_3r_1	S_4 r_1		7	2

Parsing table for expression grammar

- Since some of the entries are mul defined in the parsing table, so, the
- given grammar is **not SLR (1)** Grammar
- The circled entries are having 2 parsing actions , namely shift & reduce . the parses cannot perform both the actions. This is called **shift reduce conflict**

27. Consider the grammar, (May/ Jun – 2009)

E→ E+T
E→ T
T→ T*F
T→ F
F→ (E)
F→ id

Using predictive parsing the string id + id * id

Solution:
Int this grammar,
E → start symbol
E and T has left recursion, so it must be eliminated

A→AX|Y
Elimination of left recursion
A→YA'
A; →XA'/ €

So
After eliminating left-recursion the grammar is
E→ E+T\T
E → TE'
E' → +TE'
T→ T*F\F
T → FT'
T' → *FT' | ε
Now , the grammar is

E → TE'
E' → +TE' |ε
T → FT'
T' → *FT' | ε
F → (E)
F→ id

Step1: (computation of FIRST and FOLLOW)

computation of FIRST( ) :

FIRST(E) ={ (, id}

FIRST(E') ={+ , ε }

FIRST(T) = { (, id}

FIRST(T') ={*, ε }

FIRST(F) ={ (, id }

computation of FOLLOW( ):

FOLLOW(E) ={ $,) }

FOLLOW(E’) ={ $,) }

Stack	Input	Output
$E	id+id*id $	
$E’T	id+id*id $	E → TE’
$E’T’F	id+id*id $	T → FT’
$E’T’id	id+id*id $	F→ id
$E’T’	+id*id $	
$E’	+id*id $	T’ → ε
$E’T+	+id*id $	E’ → +TE’
$E’T	id*id $	
$E’T’F	id*id $	T → FT’
$E’T’id	id*id $	F→ id

FOLLOW(T) ={ +, $,) }

FOLLOW(T’) = { +, $,) }

FOLLOW(F) ={+, * , $,) }

Construction of Predictive parsing table

NON-TERMINAL	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’→ ε
T	T → FT’			T → FT’		
T’		T’→ ε	T’→ *FT’		T’ → ε	T’ → ε
F	F→ id			F→ (E)		

This grammar is a LL(1) Grammar

Step 3:

$E’T’	*id $	
$E’T’F*	*id $	T’ → *FT’
$E’T’F	id $	
$E’T’id	id $	F→ id
$E’T’	$	
$E’	$	T’ → ε
$	$	E’ → ε

. . . .

. . . .

. . . .

. . . .

28. Construct a canonical parsing table for the grammar given below. Also explain the algorithm used

E → E + T | T
T → T * F | F
F→ (E) | id

Construct SLR parsing for the following grammar: (Nov/Dec 2016)

G : **E → E + T | T**
T → T * F | F
F→ (E) | id

Solution:

Canonical passing table for

E→ E+T
E→ T
T→ T*F
T→ F
F→ (E)
F→ id

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar G’ :

E’ → E
E → E + T
E → T
T → T * F
T → F
F→ (E)
F→ id

Step 2 : Canonical collection of LR (0) items.

First step is closure of E’ → **. E**

E’ → **.** E
E → **.** E + T
E → **.** T
T → **.** T * F
T → **.** F
F → **.** (E)
F → **.** id

G Let us name it I0:

I0 : First step is closure of **E’** → **. E**

E’ → **.** E
E→ **.** E + T
E→ **.** T
T→ **.** T * F
T **.** F
F **.** (E)
F **.** id

Then find goto (I0, X)
X→ all the terminal and non terminals
GOTO(I0 , E) :=>
I1: E’ → E **.**
E→ E **.** + T

GOTO (I0 , T)=>
I2 **:** E → T **.**
T → T **.** * F

GOTO (I0 , F) =>
I3 : T → F **.**

GOTO (I0 , +) => Null set
GOTO (I0, *)=> Null set

GOTO (I0 , ( ) =>
I4:
F → (**.**E)
E → **.** E + T
E → **.** T
T → **.** T * F
T → **.** F
F→ **.** (E)
F → **.** id

GOTO (I0 , id) =>

I5 : F→ id **.**

GOTO (I0 ,)) Null set

Now we go for I1:

GOTO (I1 , E) => Null set
GOTO (I1 , T) => Null set
GOTO (I1 , F) => Null set

GOTO (I1 , +) =>

I_6: E → E + **.** T
T → **.** T * F
T → **.** F
F → **.** (E)
F → **.** id

GOTO (I1 , *) => Null set
GOTO (I1 , ( ) => Null set
GOTO (I1 ,)) => Null set
GOTO (I1 , id) => Null set

Then State I2:
GOTO (I2 , E) => Null set
GOTO (I2, T,F) => Null set
GOTO (I2 , +) => Null set

GOTO (I2 , *) =>

I7 : T → T * **.** F
F → **.** (E)
F → **.** id

State I3 =>

From state I_3 all the terminals and non terminals having null sets

State I4:

GOTO (I4 , E) =>

I8 : F→ (E **.**)
E → E **.** + T

GOTO (I4 , T) =>

I2 : E →T **.**
T → T **.** * F

GOTO (I4 , F) =>

I3 : T → F **.**

GOTO (I4, ( ) =>

F→ (.E)
E→ **.** E + T

E→ **.** T
T→ **.** T * F
T→ **.** F
F→ **.** (E)
F→ **.** id

GOTO (I_4, id) => F→ id **.**

State I6:

GOTO (I6 , E) => Null set

GOTO (I6 , T) =>

$$\left.\begin{array}{l} I9 : E \rightarrow E + T\,. \\ \quad T \rightarrow T\,.\, * F \end{array}\right\}$$

GOTO (I6 , F) =>

I3 : T → F **.**

GOTO (I6 , ( ) =>

$$I4: \left.\begin{array}{l} F\rightarrow (.E) \\ E\rightarrow .\,E + T \\ E\rightarrow .\,T \\ T\rightarrow .\,T * F \\ T\rightarrow .\,F \\ F\rightarrow .\,(E) \\ F\rightarrow .\,id \end{array}\right\}$$

GOTO (I6 , id) =>

I5 : F→ id **.**

State I_7:

GOTO (I7 , F) =>

I10 : T → T * F **.**

GOTO (I7 , ( ) =>

$$I4 : \left.\begin{array}{l} F\rightarrow (\,.\,E\,) \\ E \rightarrow .\,E + T \\ E \rightarrow .\,T \\ T \rightarrow .\,T * F \\ T \rightarrow .\,F \\ F \rightarrow .\,(E) \\ F \rightarrow .\,id \end{array}\right\}$$

GOTO (I7 , id) =>

I5 : F → id **.**

State I_8:

GOTO (I8 ,)) =>

I_{11}: F→ (E) **.**

GOTO (I8 , +) =>

$$I6 : \left.\begin{array}{l} E \rightarrow E + . T \\ T \rightarrow . T * F \\ T \rightarrow . F \\ F \rightarrow . (E) \\ F \rightarrow . id \end{array}\right\}$$

State I_9:

GOTO (I9 , *) =>

$$I7 : \left.\begin{array}{l} T \rightarrow T * . F \\ F \rightarrow . (E) \\ F \rightarrow . id \end{array}\right\}$$

State I_{10}:

GOTO (I_{10}, all the terminals and no terminals) => Null set

State I_{11}:

Null set

Parsing table construction:

State	**action**						**Goto**		
	id	+	*	**(**	**)**	**$**	**E**	**T**	**F**
0	S5			S4			1	2	3
1		S6				acc			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Parsing table for expression grammar

29. Check whether the following grammar is LL(1) grammar (Apr/May – 2005)

S→iEtS \ iEtSeS \ a

E→b

Consider this following grammar:

S → iEtS | iEtSeS | a E →

b

After eliminating left factoring, we have

S → iEtSS' |a
S'→ eS | ε
E → b

To construct a parsing table, we need FIRST()and FOLLOW() for all the non-terminals.
FIRST()

FIRST(S) = {i, a}
FIRST(S') = {e, ε}
FIRST (E) = {b}

FOLLOW ()
FOLLOW(S) ={ $,e }
FOLLOW(S') = { $,e }
FOLLOW(E) = {t}
Parsing table

NON-TERMINAL	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S'→ eS S'→ ε			S'→ ε
E		E → b				

Since there are more than one production, the grammar is not LL(1) grammar

30. Consider the grammar (8 Marks) (May/ June – 2006) (May/June 2016)

E→ E+E
E→ E*E
E→(E)
E→ id

Show the sequence of moves made by the shift-reduce parser on the Input id + id * id and determine whether the given string is accepted by the parser or not.

Consider the grammar:

E → E+E
E → E*E
E → (E)
E → id

And the input string id1+id2*id3

The rightmost derivation is :

F → <u>**E+E**</u>
→ E+<u>**E*E**</u>
→ E+E*<u>**id3**</u>
→ E+<u>**id2***</u>id 3

→ **id1**+id2*id3

In the above derivation the underlined substrings are called **handles.**

Handle pruning:

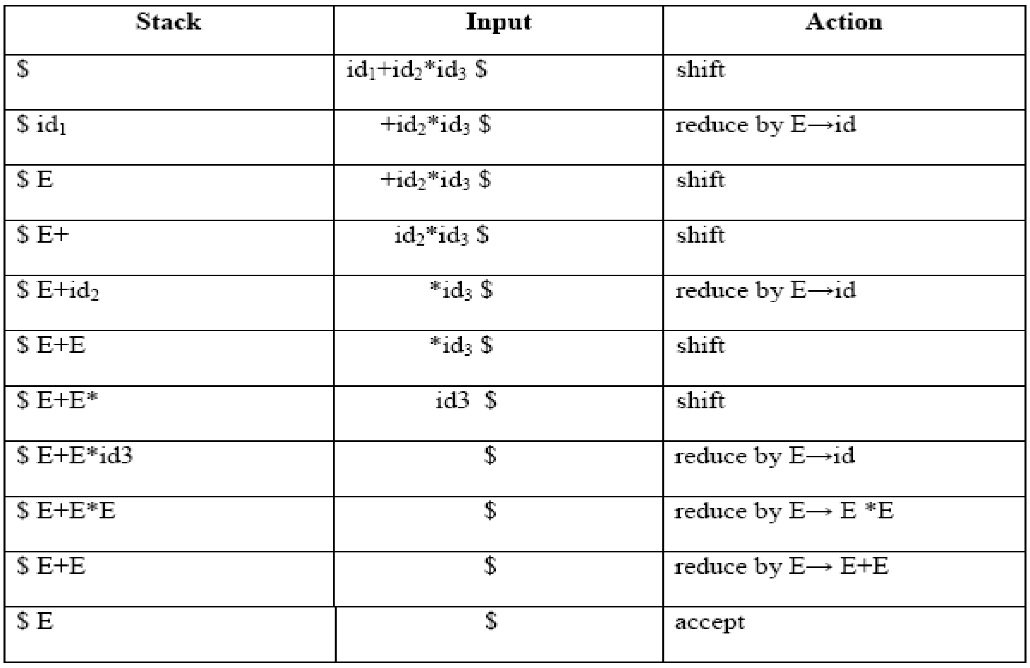
A rightmost derivation in reverse can be obtained by "**handle pruning**".

(i.e.) if *w* is a sentence or string of the grammar at hand, then *w* = γ*n*, where γ*n* is the *n*th right-sentinel form of some rightmost derivation.

Actions in shift -reduce parser:

- shift – The next input symbol is shifted onto the top of the stack.
- reduce – The parser replaces the handle within a stack with a non-terminal.
- accept – The parser announces successful completion of parsing.
- error – The parser discovers that a syntax error has occurred and calls an error recovery routine.

Stack implementation of shift-reduce parsing :



Stack	Input	Action
\$	id_1+id_2*id_3 \$	shift
\$ id_1	+id_2*id_3 \$	reduce by E→id
\$ E	+id_2*id_3 \$	shift
\$ E+	id_2*id_3 \$	shift
\$ E+id_2	*id_3 \$	reduce by E→id
\$ E+E	*id_3 \$	shift
\$ E+E*	id3 \$	shift
\$ E+E*id3	\$	reduce by E→id
\$ E+E*E	\$	reduce by E→ E *E
\$ E+E	\$	reduce by E→ E+E
\$ E	\$	accept

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift shift-reduce parsing:

Shift-reduce conflict: The parser cannot decide whether to shift or to reduce.

Reduce-reduce conflict: The parser cannot decide which of several reductions to make.

31. Briefly explain error recovery in LR parsing (Nov\ Dec – 2005)

Panic mode error recovery

It is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. the synchronizing set should be chosen so that the parser recovers quickly from error that are likely to occur in practice . Some Heuristics are as follows:

1. We can place all symbols in Follows (A) into the synchronizing set for nonterminal A. If we skip tokens until an element of follow(A) is seen and pop A from the stack, it is likely that parsing can continue.

2. It is not enough to use FOLLOW(A) as the synchronizing set for A. Often, there is a hierarchical structure on constructs in a language eg, expressions appears within statements , which appears within blocks and so on. We can add to the synchronizing set of a lower constructs the symbols that begins higher constructs the symbols that begin higher constructs.

3.If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

4.If a non terminal can generate the empty string ,then the production deriving can be used as a defsult . This approaches reduces a numberof nonterminals that have to be consider during error recovery.

5. If a terminal on the top of the stack cannot be matched , a simple idea is to pop the terminal issue a message saying that the terminal was inserted and continued parsing.

Example :
The parsing table for the grammar :
E→TE’
E’→+TE’
T’→FT’
T’→*FT|ε
F→(E)|id
with “synch”indicating synchronizing tokens obtainined from the follow set of the nonterminal.The parsing table is shown in below table.

non terminal	**input symbol**					
	id	**+**	*****	**(**	**)**	**$**
E	E→TE’			E→TE’	synch	synch
E’		E’→+TE’			E’→ε	E’→ε
T	T→FT’	synch		T→FT’	synch	synch
T’		T→ε	T’→*FT’		T’→ε	T’→ε
F	F→id	synch	synch	F→(E)	synch	synch

Synchronizing tokens added to parsing table

If the parser looks up the entry M[A,a] and finds that it is blank, then the input symbol a is skipped. If the entry is synch, then the nonterminal on the topo of the stack is popped in an attempt to resume parsing.If a token on the top of the stack does not match the input symbol, then we pop the token from the stack.

on errorneous input) id*+ id the parser and error recovery mechanism behave as

stack	**input**	**remark**
$ E	) id * + id $	error, skip)
$ E	id * + id $	id is in FIRST(E)
$ E ’ T	id * + id $	

$ E ’ T ’F	id * + id $	
$ E ’ T ’ id	id * + id $	
$ E ‘T’	* + id $	
$ E ‘T ‘ F *	* + id $	
$ E ‘ T ‘ F	+ id $	error, M[F,+]=synch
$ E ‘ T ‘	+ id $	F has been popped
$ E ‘	+ id $	
$ E ‘ T +	+ id $	
$ E ‘ T	id $	
$ E ‘ T ‘ F	id $	
$ E ‘ T ‘ id	id$	
$ E ‘ T ‘	$	
$ E ‘	$	

Parsing and error recovery moves made by predictive parser

Phrase –level recovery:

It is implemented by implemented by filling in the blank entries in the predictive parsing table with pointers to error routines . The routines may change, insert or delete symbols on the input and issue appropriate error message.

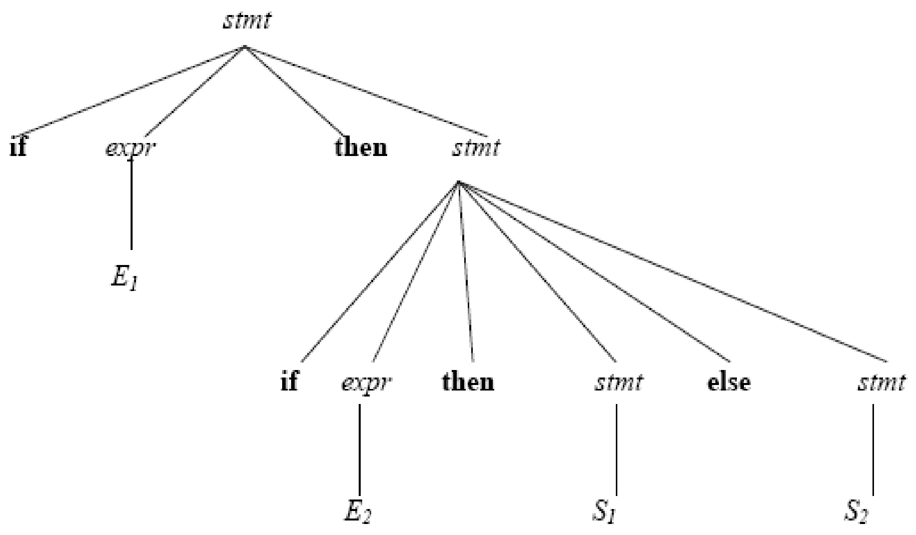
32. What are the preliminary steps that are to be carried out during parsing? Explain with suitable example (6Marks) (Apr/May – 2005)

- Elimination of left recursion, left factoring and ambiguous grammar.
- Construct FIRST() and FOLLOW() for all non-terminals.
- Construct predictive parsing table.
- Parse the given input string using stack and parsing table

Eliminating ambiguity:

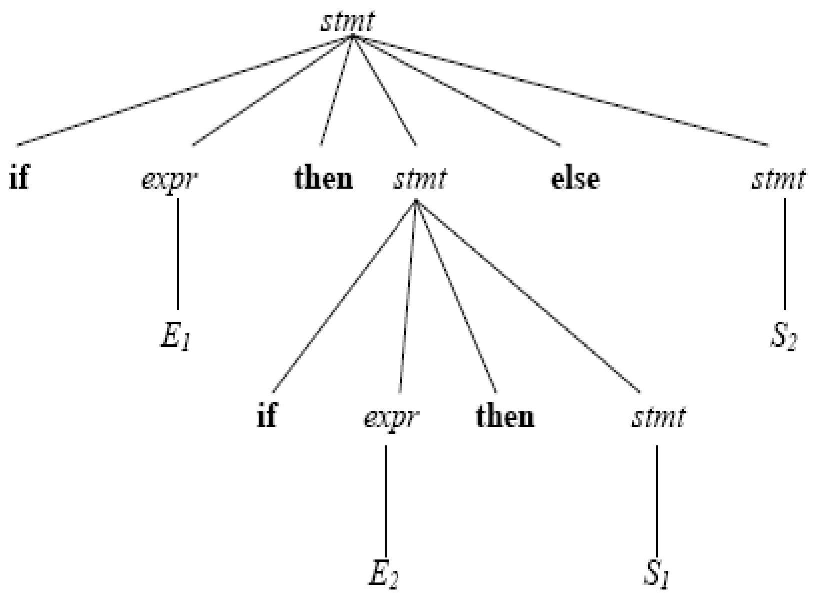
- Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.
- Consider this example, G: *stmt* → **if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | **other**
- This grammar is ambiguous since the string **if E1 then if E2 then S1 else S2** has the following two parse trees for leftmost derivation :

1.



•

2.



Eliminating Left Recursion:

A grammar is said to be *left recursive* if it has a non-terminal *A* such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production A → Aα |β it can be replaced with a sequence of two productions

A→ βA'

A'→ αA' | ε

without changing the set of strings derivable from A.

Example : Consider the following grammar for arithmetic expressions:

E → E+T |T

T → *F|F

F→ (E) |id

First eliminate the left recursion for E as

E → TE’

E’ → +TE’ |ε

Then eliminate for T as T → FT’

T’→ *FT’ | ε

Thus the obtained grammar after eliminating left recursion is

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε F → (E)

|id

Eliminating the Left factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any production A →αβ1 | αβ2 , it can be rewritten as

A →αA’

A’→β1 | β2

Consider the grammar , G : S→iEtS | iEtSeS | a
E → b

Left factored, this grammar becomes

S → iEtSS’ | a
S’→ eS |ε
E → b

Construct FIRST () and FOLLOW () for all non-terminals

The construction of a predictive parser is aided by two functions associated with a grammar G :

1.FIRST

2.FOLLOW

Rules for first( ):

1.If *X* is terminal, then FIRST(*X*) is {X}.

2.If *X* → ε is a production, then add ε to FIRST(*X*).

3.If *X* is non- terminal and X → *a*α is a production then add *a* to FIRST(X).

4.If X is non- terminal and *X* → *Y1 Y2…Yk* is a production, then place *a* in FIRST(*X*) if for some *i*, *a* is in FIRST(Yi), and ε is in all of FIRST(Y1),…,FIRST(Yi-1); that is, Y1,….Yi-*1* => ε. If ε is in FIRST(*Y j*) for all j=1,2,..,k, then add ε to FIRST(*X*).

Rules for follow( ):

1.If *S* is a start symbol, then FOLLOW(*S*) contains $.

2.If there is a production *A* → α*B*β, then everything in FIRST(β) except ε is placed in follow(*B*).

3.If there is a production *A* → α*B*, or a production *A* → α*B*β where FIRST(β) contains ε, then everything in FOLLOW(*A*) is in FOLLOW(*B*).

Algorithm for construction of predictive parsing table:

Input: Grammar *G*

Output: Parsing table *M*

Method:

1. For each production *A* →α of the grammar, do steps 2 and 3.

5. For each terminal *a* in FIRST(α), add *A* → α to *M*[*A*, *a*].

6. If ε is in FIRST(α), add A → α to *M*[*A*, *b*] for each terminal *b* in FOLLOW(*A*). If ε is in FIRST(α) and $ is in FOLLOW(*A*) , add *A* → α to *M*[*A*, $].

7. Make each undefined entry of *M* be **error**.

Parse the given input string using stack and parsing table

Algorithm for nonrecursive predictive parsing: Input: A

string *w* and a parsing table *M* for grammar *G*.

Output : If *w* is in *L*(*G*), a leftmost derivation of *w*; otherwise, an error indication.

Method : Initially, the parser has $*S* on the stack with *S*, the start symbol of *G* on top, and *w*$ in the input buffer. The program that utilizes the predictive parsing table *M* to produce a parse for the input is as follows:

```
set ip to point to the first symbol of $w$$;
repeat
    let $X$ be the top stack symbol and $a$ the symbol pointed to by ip;
    if $X$ is a terminal or $ then
        if $X = a$ then
            pop $X$ from the stack and advance ip
        else error()
    else        /* $X$ is a non-terminal */
        if $M[X, a] = X \rightarrow Y_1 Y_2 \ldots Y_k$ then begin
                pop $X$ from the stack;
                push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with $Y_1$ on top;
                output the production $X \rightarrow Y_1 Y_2 \ldots Y_k$
            end
            else error()
  until $X$ = $        /* stack is empty */
```

33. Explain briefly about LALR PARSING

MOTIVATION

The LALR (Look Ahead-LR) parsing method is between SLR and Canonical LR both in terms of power of parsing grammars and ease of implementation. 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 can not be handled by SLR techniques.

CONSTRUCTING LALR PARSING TABLES

CORE: A core is a set of LR (0) (SLR) items for the grammar, and an LR (1) (Canonical LR) grammar may produce more than two sets of items with the same core.
The core does not contain any look ahead information.

Example: Let s1 and s2 are two states in a Canonical LR grammar.
S1 – {C ->c.C, c/d; C -> .cC, c/d; C -> .d, c/d}
S1 – {C ->c.C, $; C -> .cC, $; C -> .d, $}

These two states have the same core consisting of only the production rules without any look ahead information.

CONSTRUCTION IDEA:

1. Construct the set of LR (1) items.
2. Merge the sets with common core together as one set, if no conflict (shift-shift or shift-reduce) arises.
3. If a conflict arises it implies that the grammar is not LALR.
4. The parsing table is constructed from the collection of merged sets of items using the same algorithm for LR (1) parsing.

ALGORITHM:

Input: An augmented grammar G'.

Output: The LALR parsing table actions and goto for G'.

Method:

1. Construct C= {I0, I1, I2,… , In}, the collection of sets of LR(1) items.
2. For each core present in among these sets, find all sets having the core, and replace these sets by their union.
3. Parsing action table is constructed as for Canonical LR.
4. The goto table is constructed by taking the union of all sets of items having the same core. If J is the union of one or more sets of LR (1) items, that is, J=I1 U I2 U … U Ik, then the cores of goto(I1,X), goto(I2,X),…, goto(Ik, X) are the same as all of them have same core. Let K be the union of all sets of items having same core as goto(I1, X). Then goto(J,X)=K.

EXAMPLE

GRAMMAR:

1. S' -> S
2. S -> CC
3. C -> cC
4. C -> d

STATES:

- I0 : S' -> .S, $
  S -> .CC, $
  C -> .c C, c /d
  C -> .d, c /d
- I1: S' -> S., $
- I2: S -> C.C, $
  C -> .Cc, $
  C -> .d, $
- I3: C -> c. C, c /d
  C -> .Cc, c /d
  C -> .d, c /d
- I4: C -> d., c /d
- I5: S -> CC., $
- I6: C -> c.C, $
  C -> .cC, $

C -> .d, $
- I7: C -> d., $
- I8: C -> cC., c /d
- I9: C -> cC., $

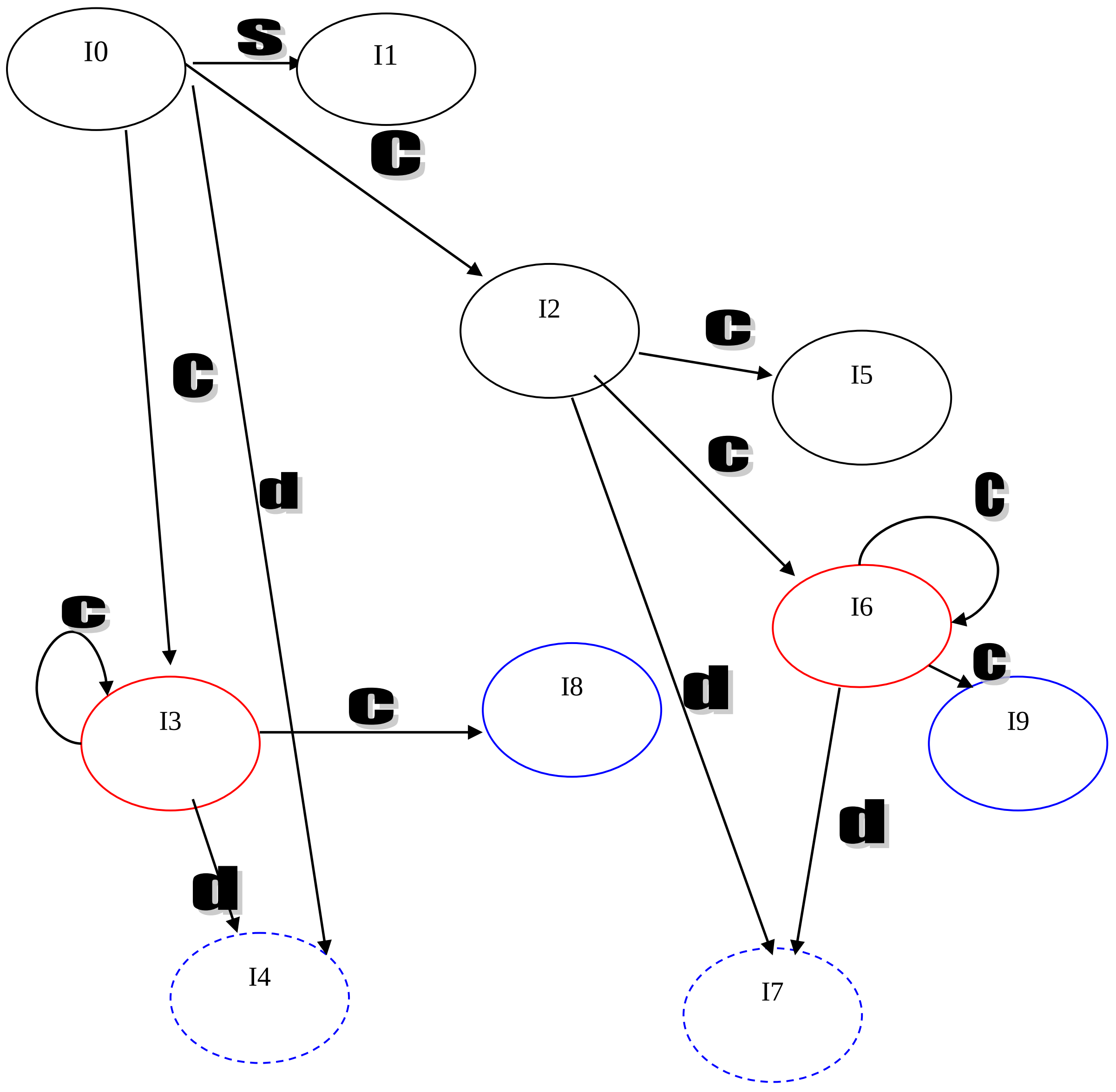
CANONICAL PARSING TABLE:

STATE	**Actions**			**Goto**	
	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		

NOTE: For goto graph see the construction used in Canonical LR.

LALR PARSING TABLE:

START	**Actions**			**goto**	
	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		



- Showing states with same core with same colour which get merged in conversion from LR(1) to LALR.

- States merged together: 3 and 6
  4 and 7
  8 and 9

<u>SHIFT-REDUCE CONFLICT</u>

COMPARISON OF LR (1) AND LALR:

- If LR (1) has shift-reduce conflict then LALR will also have it.
- If LR (1) does not have shift-reduce conflict LALR will also not have it.
- Any shift-reduce conflict which can be removed by LR (1) can also be removed by LALR.
- For cases where there are no common cores SLR and LALR produce same parsing tables.

COMPARISON OF SLR AND LALR:

- If SLR has shift-reduce conflict then LALR may or may not remove it.
- SLR and LALR tables for a grammar always have same number of states.

Hence, LALR parsing is the most suitable for parsing general programming languages. The table size is quite small as compared to LR (1) , and by carefully designing the grammar it can be made free of conflicts. For example, in a language like Pascal LALR table will have few hundred states, but a Canonical LR will have thousands of states. So it is more convenient to use an LALR parsing.

REDUCE-REDUCE CONFLICT

However, the Reduce-Reduce conflicts still might just remain .This claim may be better comprehended if we take the example of the following grammar:

S’-> S

S-> aAd

S-> bBd

S-> aBe

S-> bAe

A-> c

B-> c

Generating the LR (1) items for the above grammar,

I0 : S’-> .S , $

S-> . aAd, $

S-> . bBd, $

S-> . aBe, $

S-> . bAe, $

I1: S’-> S ., $

I2: S-> a . Ad, $

S-> a . Be, $

A-> .c, d

B->.c, e

I3: S-> b . Bd, $

S-> b . Ae, $

A->.c, e

B->.c,d

I4: S->aA.d, $

I5: S-> aB.e,$

I6: **A->c. , d**

B->c. , e

I7: S->bB.d, $

I8: S->bA.e, $

I9: **B->c. , d**

A->c. , e

I10: S->aAd. , $

I11: S->aBe., $

I12: S->bBd., $

I13: S->aBe., $

The underlined items are of our interest. We see that when we make the Parsing table for LR (1), we will get something like this…

The LR (1) Parsing Table. (partly filled)

	a ………………	d	e	
I1 I2 . . .		. . . .	. . . .	
I6	……………….	**r6**	**r7**	
.. . .		. . . .	. . . .	
I9	………………….	**r7**	**r6**	
.				

This table on reduction to the LALR parsing table, comes up in the forms of-

The LALR Parsing table. (partly filled)

	a ………………	d	e	

I1 I2 . . .		. . . .	. . . .	
I69	**……………….**	**r6/r7**	**r7/r6**	
.. . .		. . . .	. . . .	
~~I9~~	~~………………….~~	~~r7~~	~~r6~~	
.				

So, we find that the LALR gains reduce-reduce conflict whereas the corresponding LR (1) counterpart was void of it. This is a proof enough that LALR is less potent than LR (1).
But, since we have already proved that the LALR is void of shift-reduce conflicts (given that the corresponding LR(1) is devoid of the same), whereas SLR (or LR (0)) is not necessarily void of shift-reduce conflict, the LALR grammar is more potent than the SLR grammar.

SHIFT-REDUCE CONFLICT present in SLR

⇓ Some of them are solved in….

LR (1)

⇓ All those solved are preserved in…

LALR

So, we have answered all the queries on LALR that we raised intuitively.

34. Design an LALR parser for the following grammar and parse the input id=id.(16)(Nov/Dec 2013)

S->L=R|R

L->*R|id

R->L

Augmented Grammar

S'->S

S->L=R|R

L->*R|id

R->L

Canonical collection of LR(1) items

Iₒ: S’->.S,$	b=first (Ɛ$)={$}
S->.L=R,$	first(=$)={=}
S->.R,$	first(Ɛ$)={$}
L->.*R,=	
L->.id,=	
R->.L,$	

Goto(0,S)

I1:S”->S.,$

Goto(0,L)

I2: S->L.=R,$

R->L.,$

Goto(0,R)

I3:S->R.,$

Goto(0,*)

I4:L->*.R,=	first(Ɛ=)={=}
R->.L,=	first(Ɛ=)={=}
L->.*R,=	
L->.id,=	

Goto(0,id)

I5:L->id.,=

Goto(2,=)

I6:S->L=.R,$	first(Ɛ$)={$}
R->.L,$	first(Ɛ$)={$}
L->.*R,S	
L->.id,$	

Goto(4,R)

I7:L->*R.,=

Goto(4,L)

I8:R->L.,=

Goto(4,*)

I4

Goto(4,id)

I5

Goto(6,R)

I11:L->*.R,$	first(Ɛ$)={$}
R->.L,$	first(Ɛ$)={$}
L->.*R,$	
L->.id,$	

Goto(6,id)

I12:L->id.,$

Goto(11,R)

I9:S->L=R.,$

Goto(6,L)

I10:R->L.,$

Goto(6,*)

I13:L->*R.,$

Goto(11,L)

I10

Goto(11,*)

I11

Goto(11,id)

I12

This is states for DFA I_o to I13(fourteen states)

LR(1) table construction

States	Action				Goto		
	=	*	id	$	S	L	R
0		S4	S5		1	2	3
1				Acc			
2	S6			r5			
3				r2			
4		S4	S5			8	7
5	r4						
6		S11	S12			10	9
7	r3						

8	r5	
9	r1	
10	r5	
11	S11 S12	10 13
12	r4	
13	r3	

This grammar is an LR(1),since it does not produce may multi-defined entry for its parsing table.

LALR Table Construction

I4 and I11 are similar. Combine them as

I11: L->*.R,=/$
or R->.L,=/$
I4 L->.*R,=/$
L->.id,=/$

I5 and I12 are similar.

I512: L->id.,=/$

or I5

I7 and I13 are similar,

I713: L->*R.,=/$

or I7

I8 and I10 are similar.

I810 :R->L.,=/$

or I8

Now the LALR parsing table after graphing the states is follows:

States	Action				Goto		
	=	*	id	$	S	L	R

0	S4 S5	1 2 3
1	acc	
2	S6 r5	
3	r2	
4	S4 S5	8 7
5	r4 r4	
6	S4 S5	8 9
7	r3 r4	
8	r5 r5	
9	r1	

Note that the number of states we obtained in LALR is same as SLR.SLR has multiply defined entry for[2,=] that leads to shift/reduced conflict that has been made into single entry in both LR(1) and LALR parsing .thus there will not be any shift/reduce conflict.

Parsing the string id=id using LR(1)

Stack	**Input**	**Action**
0	Id=id $	Shift 5
0 id 5	=id $	Reduce by L->id
0 L 2	=id $	Shift 6
0 L 2= 6	id $	Shift 12
0 L 2 = 6 id 12	$	Reduce by L->id
0 L 2 = 6 L 10	$	reduce by R->L
0 L 2 = 6 R 9	$	reduce by S->L=R
0 S 1	$	Accept

Parsing the string id=id using LALR

Stack	**Input**	**Action**
0	Id=id $	Shift 5
0 id 5	=id $	Reduce by L->id
0 L 2	=id $	Shift 6

0 L 2= 6	id $	Shift 5
0 L 2 = 6 id 5	$	Reduce by L->id
0 L 2 = 6 L 8	$	reduce by R->L
0 L 2 = 6 R 9	$	reduce by S->L=R
0 S 1	$	Accept

35. Explain briefly about the Parser Generator Yacc.

Parser Generator Yacc

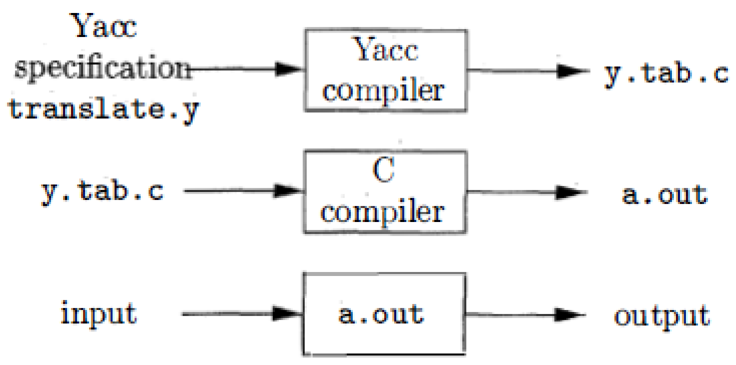
A translator can be constructed using Yacc in the manner illustrated in Fig.. First, a file, say translate . y, containing a Yacc specification of the translator is prepared. The UNIX system command

yacc translate . y

transforms ttte file trans late . y into a C program called y. tab . c using the LALR method outlined in Algorithm. 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 parsing table is compacted as described. 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 Yac c program.7 If other procedures are needed, they cap be compiled or loaded with y. tab . c, just as with any C program.
A Yacc source program has three parts:



Creating an input/output translator with Yacc

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

Example : To illustrate how to prepare a Yacc source program, let us construct a simple desk calculator that reads an arithmetic expression, evaluates it, and then prints its numeric value. We shall build the desk calculator starting with the with the following grammar for arithmetic expressions:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \textbf{digit} \end{array}$$

The token **digit** is a single digit between 0 and 9. A Yacc desk calculator program derived from this grammar is shown in Fig.

The Declarations Part

There are . two sections in the declarations part of a Yacc program; both are optional. In the first section, we put ordinary C declarations, delimited by %{and % }. Here we place declarations of any temporaries used by the translation rules or procedures of the second and third sections. In Fig. 4.58, this section contains only the include-statement

#include <ctype .h>

that causes the C preprocessor to include the standard header file <ctype . h> that contains the predicate isdigi t.

Also in the declarations part are declarations of grammar tokens. In Fig. 4.58, the statement

%token DIGIT

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
line    : expr '\n'         { printf("%d\n", $1); }
        ;
expr    : expr '+' term     { $$ = $1 + $3; }
        | term
        ;
term    : term '*' factor   { $$ = $1 * $3; }
        | factor
        ;
factor  : '(' expr ')'      { $$ = $2; }
        | DIGIT
        ;
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

Yacc specification of a simple desk calculator

declares DIGIT to be a token. Tokens declared in this section can then be used in the second and third parts of the Yacc specification. , If Lex is used to create the lexical analyzer that passes token to the Yacc parser, then these token declarations are also made available to the analyzer generated by Lex.

The Translation Rules Part

In the part of the Yacc specification after the first %% pair, we put the translation rules. Each rule consists of a grammar production and the associated semantic action. A set of productions that we have been writing:

$$\langle\text{head}\rangle \rightarrow \langle\text{body}\rangle_1 \mid \langle\text{body}\rangle_2 \mid \cdots \mid \langle\text{body}\rangle_n$$

Would be written in Yacc as

```
⟨head⟩  :  ⟨body⟩₁   { ⟨semantic action⟩₁ }
        |  ⟨body⟩₂   { ⟨semantic action⟩₂ }
             ...
        |  ⟨body⟩ₙ   { ⟨semantic action⟩ₙ }
        ;
```

In the Yacc specification, we have written the two E-productions

$$E \rightarrow E + T \mid T$$

and their associated semantic actions as:

```
expr : expr '+' term    { $$ = $1 + $3; }
     | term
     ;
```

Note that the nonterminal term in the first production is the third grammar symbol of the body, while + is the second. The semantic action associated with the first production adds the value of the **expr** and the **term** of the body and assigns the result as the value for the nonterminal **expr** of the head. We have omitted the semantic action for the second production altogether, since copying the value is the default action for productions with a single grammar symbolin the body. In general, { $$ = $ 1 ; } is the default semantic action.

Notice that we have ad ded a new starting production

line : expr ' \n ' { print f C " %d\n" , $ 1) ; }

to the Yacc specification. This production says that an input to the desk calculator is to be an expression followed by a newline character. The semantic action associated with this production prints the decimal value of the expression followed by a newline character.

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 €.

Second, we shall enlarge the class of expressions to include numbers instead of single digits and to include the arithmetic operators +, -, (both binary and unary), * , and /. The easiest way to specify this class of expressions is to use the ambiguous grammar

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid - E \mid \mathbf{number}$$

The resulting Yacc specification is

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double  /* double type for Yacc stack */
%}
%token NUMBER

%left '+' '-'
%left '*' '/'
%right UMINUS
%%

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

expr  : expr '+' expr     { $$ = $1 + $3; }
      | expr '-' expr     { $$ = $1 - $3; }
      | expr '*' expr     { $$ = $1 * $3; }
      | expr '/' expr     { $$ = $1 / $3; }
      | '(' expr ')'      { $$ = $2; }
      | '-' expr  %prec UMINUS { $$ = - $2; }
      | NUMBER
      ;
%%
yylex() {
    int c;
    while ( ( c = getchar() ) == ' ' );
    if ( (c == '.') || (isdigit(c)) ) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
```

Yacc specification for a more advanced desk calculator

Creating Yac c Lexical Analyzers with Lex

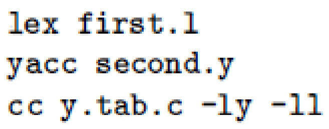
Lex was designed to produce lexical analyzers that could be used with Yacc . The Lex library 11 will provide a driver program named yylex 0 , the name required by Yacc for its lexical analyzer. If Lex is used to produce the lexical analyzer, we replace the routine yylex ( ) in the third part of the Yace specification by the statement

# include 1 1ex . yy . c "

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

output file y . t ab . e.

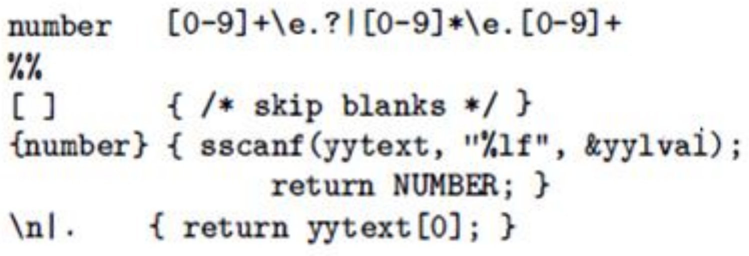
Under the UNIX system, if the Lex specification is in the file f irst . 1 and the Yaee specification in seeond . y, we can say



```
lex first.l
yacc second.y
cc y.tab.c -ly -ll
```

to obtain the desired translator.

The last pattern, meaning "any character" must be written \n 1 • since the dot in Lex matches any character except newline.



```
number   [0-9]+\e.?|[0-9]*\e.[0-9]+
%%
[ ]      { /* skip blanks */ }
{number} { sscanf(yytext, "%lf", &yylval);
                return NUMBER; }
\n|.     { return yytext[0]; }
```

Lex specification for yylex()

Error Recovery in Yacc

Desk calculator with error recovery

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double  /* double type for Yacc stack */
%}
%token NUMBER

%left '+' '-'
%left '*' '/'
%right UMINUS
%%

lines : lines expr '\n'   { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      | error '\n' { yyerror("reenter previous line:");
                     yyerrok; }
      ;
expr  : expr '+' expr     { $$ = $1 + $3; }
      | expr '-' expr     { $$ = $1 - $3; }
      | expr '*' expr     { $$ = $1 * $3; }
      | expr '/' expr     { $$ = $1 / $3; }
      | '(' expr ')'      { $$ = $2; }
      | '-' expr  %prec UMINUS { $$ = - $2; }
      | NUMBER
      ;
%%
#include "lex.yy.c"
```

Anna University Question Bank

NOV/DEC-16
PART A

1. Construct a parse tree for —(id + id)
2. What is meant by handle pruning?

PART B

1.(a) (i) construct parse tree for the input string w = cad using top down parser. (6)

S ----> cAd

A ----> ab | a

(ii) Construct parsing table for the grammar and find moves made by predictive parser on input id+id*id and find FIRST and FOLLOW.(10)

E--->E+T

E ---> T

T-->T*F

T ---> F

F ---> (E)/id

2. (b) (i) Explain ambiguous grammar G : E --> E + E | E * E | (E) | -E | id for the sentence id+id*id.(6)

(ii) Construct SLR parsing table for the following grammar :

G:E->E+T|TT->T*F|FF->(E)|id(10)

May/June-16
PART A

1. Write the algorithm for FIRST and follow in parser.
2. Define ambiguous grammar.

PART B

13. (a)(i) Construct stack implementation of shift reduce parsing for the grammar (8)

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$ and the input string id1+id2*id3

(ii) Explain LL(1) grammar for the sentences $S \rightarrow iEts|iEtSeS|a$ $E \rightarrow b$.(8)

Or

(b) (i) Write an algorithm for Non recursive predictive parsing. (6)

(ii) Explain Context free grammar with examples.(10)

May/June-14

PART-B

1.a)Consider the following grammar

S-AS|b

A-SA|a.

Construct the SLR parse table for the grammar .Shoe the actions of the parser for the input string “abab”. (16)

2)i)What is an ambiguous grammar? Is the following grammar ambiguous? Prove EE+|E(E)|id.The grammar should be moved to the next line ,centered.

NOV/DEC-2013

PART-A

1.Eliminate the left recursion for the grammer

S->Aα|b

A->Ac|Sd|ϵ

2.What is meant by coercion?

PART-B

1.Design an LALR parser for the following grammar and parse the input id=id.(16)

S->L=R|R

L->*R|id

R->L

MAY/JUNE 2013

PART-A

1.Eliminate left recursion from the following grammar A->Ac/Aad/bd/ε2.Give examples for static check

PART-B

1. (i)Construct predictive parser for the following grammar. (10

S->(L) | a

L->L, S |S.

(ii) List all LR(0) items for following grammar.(6)

S->AS/b

a->Sa/A

May/Jun -2012

Part- A

1.Define an ambiguous grammar.

2.What is dangling reference?

Part-B

1. Construct the predictive parser for the following grammar

S-> (L)\a

L->L,S\S (10 Marks)

2. Describe the conflict that may occur during shift reduce parsing (6 Marks)

Apr/May -2011

Part-B

1. Explain the recovery strategies in syntax analysis (6 Marks)

2. Construct a SLR construction table for the following grammar (16 Marks)

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow |id$

Nov/Dec – 2011

Part-B

1.Construct a canonical parsing table for the grammar given below. Also explain the algorithm used

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow |id$

2. What are the different storage allocation strategies? Explain

Nov/Dec – 2010

Part- A

1. Differentiate Top Down approach from Bottom Up approach to parsing with an example
2.Differentiate SLR parser from LALR parser

Part-B

1.Construct LR (0) parsing table for the given grammar (10 Marks)

$E \rightarrow E*B$

$E \rightarrow E +B$

$E \rightarrow B$

$B \rightarrow 0$

$B \rightarrow 1$

2. Write an algorithm for the construction of LR(1) or CLR items for grammar G
.

Apr/May – 2010

Part- A

1.Give two examples for each of top down parser and bottom up parser? 2.What is the signification of look ahead in LR(1) items

Part- B

1. Construct the predictive parsing table for the following grammar how the string (a,a) is parsed by the predictive parser (16 Marks)

$S \rightarrow a \mid \uparrow \mid (T)$

$T \rightarrow T, S \mid S$

May/ Jun – 2009

Part- A

1.Derive the string and construct a syntax tree for the input string ceaedbe using the grammar S -> SaA | A,A ->AbB | B, B -> c Sd | e

2.List the factors to be considered for top – down parsing

Part- B

1.Given the following grammar S - > ASb, A -> SA |a construct a SLR parsing table for the string baab

2.Consider the grammar (16 Marks)

$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

Using predictive parsing the string id + id * id

Nov\ Dec – 2007

Part- A

1.Eliminate the left recursion from the following grammar

A -> Ac/Aad/bd/c

2. What is an ambiguous grammar?

3. What is a predictive parser?

Part- B

1. Explain the role of parser in detail (4 Marks)

May/ Jun – 2006

Part- A

1.What are the goals of error handler in a parser?

2.What is phrase level error recovery?

Part- B

1.Explain in detail about the error recovery strategies in parsing (8 Marks)

2.Consider the grammar (8 Marks)

$E \rightarrow E+E$
$E \rightarrow E*E$
$E \rightarrow (E)$
$E \rightarrow id$

Show the sequence of moves made by the shift-reduce parser on the Input id + id * id and determine whether the given string is accepted by the parser or not

Nov\ Dec – 2005

Part- A

1.What is an ambiguous grammar? Give an example

2.What are kernel and non kernel items?

3.How will YACC resolve the parsing action conflicts?

Part- B

1. Give an algorithm for finding the FIRST and Follow positions for a given non terminal (4 marks)
2.Consider the grammar (12 Marks)

$E \rightarrow TE'$
$E' \rightarrow +TE' / \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' / \varepsilon$
$F \rightarrow (E)/id$

Construct the predictive parsing table for the above grammar & verify Whether the input string id + id * id is accepted by the grammar or not.

3. What is Shift reducer parser? Explain in detail the conflict that may occur during Shift reducer parsing.(6 marks)
4. Consider the following grammar.(12 Marks)

$E \rightarrow E+T|T$
$T \rightarrow T*F|F$
$F \rightarrow (E)|id$

Construct an LR parsing table for the following grammar. View the moves of LR parser on id * id + id.

5. Briefly explain error recovery in LR parsing (4 Marks)

Apr/May – 2005

Part- A

1. What are the goals of error handler in a parser?

2.Define handle.

Part- B

1. What are the preliminary steps that are to be carried out during parsing ?

Explain with suitable example (6Marks)
2. Construct the predictive parser for the following grammar(16 Marks)

$S \rightarrow (L)|a$
$L \rightarrow L;S|S$

Write down the necessary algorithms and define FIRST and FOLLOW.
Show the behaviors of the parser in the sentences:

i. (a, (a,a))
ii.(((a,a), $\uparrow$, (a), a)

3. Explain the error recovery strategies on predictive parser. (4 marks)
4. Explain the LR parsing algorithm with example.(6 Marks)
5. Check whether the following grammar is LL(1) grammar

$S \rightarrow iEtS \setminus iEtSeS \setminus a$

$E \rightarrow b$

Apr/May – 2004

Part- A

1. What do you mean by handle pruning?

2. Define LR(0) items

Part- B

1. Construct the predictive parser for the following grammar

$S \rightarrow (L)|a$

$L \rightarrow L;S|S$

2. Check whether the following grammar is SLR (1) or not . explain Your answer with reasons (8 Marks)

$S \rightarrow L=R$

$S \rightarrow R$

$L \rightarrow * R$

$L \rightarrow id$

$R \rightarrow L$

Construct the behavior of the parser on the sentence (a,a) using the grammar specified above.

Nov\ Dec – 2004

Part- A

1. What do you mean by viable prefixes?

2. Define handle

Part- B

1. What are preliminary steps that are to be carried out during parsing? Explain with suitable example (6 marks)

2. Explain the error recovery in predictive parsing (8 Marks)

3. Write a parser generator program for desk calculator (8marks)

4. Check whether the following grammar is LL(1) grammar

$S \rightarrow iEtS \backslash iEtSeS \backslash a$

$E \rightarrow b$

CS6660- COMPILER DESIGN – UNIT V

CODE OPTIMIZATION AND CODE GENERATION

Principal Sources of Optimization-DAG- Optimization of Basic Blocks-Global Data Flow Analysis-Efficient Data Flow Algorithms-Issues in Design of a Code Generator - A Simple Code Generator Algorithm.

PART-A

1. What is meant by optimization?

It is a program transformation that made the code produced by compiling algorithms run faster or takes less space.

2. Define optimizing compilers?

Compilers that apply code improving transformation are called optimizing compilers.

3. List down the criteria for code improving transformation? Nov/Dec -2011

- A transformation must presence the meaning of program
- A transformation must speed up programs by a measurable amount.
- A transformation must be worth the effort.

4. When do you say a transformation of a program is local?

A transformation of a program is called local, if it can be performed by looking only at the statement in a basic block.

5. Write a note on function preserving transformation?

A compiler can improve a program by transformation without changing the function it compilers.

6. List the function preserving transformation Nov/Dec -2011

- Common sub expression elimination.
- Copy propagation.
- Dead code elimination.
- Constant folding.

7. Define common sub expression?

An occurrence of an expression E is called a common sub expression if E was previously computed and the values of variables in E have not changed since the previous computations.

8. Define live variable?

A variable is live at appoint in a program if its value can be used subsequently.

9. What is meant by loop optimization?

The running time of a program may be improved if we decrease the number of instruction in an inner even if we increase the amount of code outside that loop.

10. What are the three techniques for loop optimization?

- Code motion.
- induction variable elimination
- Reduction in strength.

11. Define basic block? Nov/Dec -2004

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end possibility or branching except at the end.

12. What do you mean by data flow equations?

Typical equation has the form

Out[s] = gen[s] U (in[s] – kill[s])

It can be read as "information at the end of a statement is either generated within the statement or enters at the beginning and is not And is not killed as control flows through the statement.

13. State the meaning of in[s],out[s],kill[s],gen[s] ?

In[s] – The set of definitions reaching the beginning of S

Out[s] – End of s

Gen[s] – The set of definitions generated by S

Kill[s] – The set of definitions that never reach the end of S.

14. Define formal parameters?

The identifiers appearing in procedure definitions are special and are called formal parameters.

15. Define actual parameters?

The arguments passed to a called procedure are known as actual parameters.

16. What is meant by an activation of the procedure?

An execution of a procedure is referred to as an activation of the procedure.

17. Define activation tree?

An activation tree depicts the way control enters and leaves activations.

18 .What is the use of control stack?

It keeps track of live procedure activations. Push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.

19. What is meant by scope of declaration?

The portion of the program to which a declaration applies is called the scope of that declaration. An occurrence of a name in a procedure is said to be local to the procedure if is in the scope of a declaration within the procedure: otherwise is occurrence is said to be no local.

20. What does the runtime storage hold?

Runtime storage holds

- The generated target code.
- Data objects.
- A counter part of the control stack to keep track of procedure activations.

21. Define activation record?

Information needed by a single execution of procedure is managed using a contiguous block of storage called an activation record.

- Temporary values
- Local data
- Saved machine status
- Control link
- Access disk
- Actual parameters
- Return value

22. Define access link?

It refers to no local data held in other activation records.

23.What do you mean by dangling reference?

A dangling reference occurs when there is a reference to storage that has been deallocated.

24. What is meant by static scope rule?

Determines the declaration that applies to a name by examine the program text alone.

25. What do you mean by dynamic scope rule?

It determines the declaration applicable to a name at run time by considering the current activations.

26. Define block?

A block is a statement containing its own local data declarations.

27. Define l-value and r- value?

The l- value refers to the storage represented by an expression and r- value refers to the value contained in the storage.

28. What are the features to be maintained during optimization?

Semantic equivalence with the program has to be maintained

The transformation must speed up program to a measurable amount

30. What are the factors influencing the optimization?

- Characteristics of the target machine
- Number of CPU registers
- Number of functional units
- cache size

31. What is dead code?

The portion of the program that can never get executed for any control flow through the program.

32. Define program point.

A program point wj is the instant between the end of the execution of instruction i_j and the beginning of the of the instruction i_{j+1}

33. What is code motion? Apr/May -2004

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the

Number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

Eg:. While (i<limit -2)

{

…..

```
…..
}
Can be restated as
T= limit-2;
While (I,t)
{
….
…
}
```

34. List out two properties of reducible flow graph (May/Jun-2012)

The forward edges form an acyclic graph in which every node can be reached from initial node of G

The back edges consist only of edges whose heads dominate their tails

35. What is the use of algebraic identifies in optimizations of basic blocks? (May/Jun-2012)

Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

Examples:

i) x : = x + 0 or x : = x * 1 can be eliminated from a basic block without changing the set of expressions it computes.

ii) The exponential statement x : = y * * 2 can be replaced by x : = y * y.

36. Define dead-code elimination. (Apr/May-2011)

Dead code is a code that is either never executed or if it is executed, its result is never used by the program

Suppose *x* is dead, that is, never subsequently used, at the point where the statement

x : = y + z appears in a basic block. Then this statement may be safely removed without

changing the value of the basic block.

37. What is loop optimization?(Apr/May -2011)

The way of decreasing the number of instructions in the inner loop by increasing the number of instructions in the outer loop Methods

- Code motion
- Induction variable elimination
- Strength reduction

38. When does dangling reference occur? (Nov/Dec -2011)(May/June 2016)

Whenever storage can be de-allocated, the problem dangling reference arises. Dangling reference occurs when there is a reference to storage that has been de-allocated

39. How Would You Map Names To Values? (Nov/Dec 2010)

An Address Descriptor Keeps Track Of The Location Where The Current Value Of The Name Can Be Found At Run Time.

40. List the characteristics of peephole optimization. (Nov/Dec 2010) (Nov/Dec 2016)

- Redundant instruction elimination
- Flow of control optimization
- Algebraic simplification
- Use of machine idioms
-

41. List out the primary structure preserving transformations on basic block. (Apr/May -2011)

- Common sub-expression elimination
- Dead code elimination
- Algebraic transformation like reduction in strength
- Use of algebraic identifiers

42. Define dead code elimination (Apr/May -2011) (Nov/Dec 2010

Dead code is a code that is either never executed or if it is executed, its result is never used by the program

Eg: x= 5 ; // dead code

y= y+1:

X= y*2;

Print y, x

43. What are the fields available in activations record? Apr/May -2010

- Temporary values
- Local data
- Saved machine status
- Control link
- Access disk
- Actual parameters
- Return value

44. Give any two examples for strength reduction Apr/May -2010

Which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

Example

- X**2 = X*x
- 2*x=x + x
- x/2=x*0.5

45. What are the categories of code optimization?

- Peephole optimizations
- Local optimizations
- Loop optimizations
- Global or intraprocedural optimizations
- Interprocedural or whole program optimization

46. What are the different data flow properties?

- Available expression
- Reaching definitions
- Live variables
- Busy variable

47. What are the techniques behind code optimization phase?

- Constant folding
- Loop constant code motion
- Induction variable elimination
- Common sub expression elimination
- Strength reduction
- Mathematical identities

48. What are the methods available in loop optimization?

- Code movement
- Strength reduction
- Loop test replacement

- Induction variable elimination

49. What are dominators?

In a flow graph, a node d **dominates** node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by ***d, dom ,n.*** Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop.

Similarly every node dominates itself.

50 .What is meant by constant folding (May/June-2013)

We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.
One advantage of copy propagation is that it often turns the copy statement into dead code.
For example,
a=3.14157/2 can be replaced by
a=1.570 there by eliminating a division operation

51. What is meant by copy restore? May/Jun -2009

A hybrid between call by value and call by reference is call-restore linkage (also known as copy in copy out or value result)

52. What is a peehole optimization? (Nov/Dec 2010)

A simple technique for locally improving target code (can also be applied to intermediate code).The peephole is a small, moving window on the target program.

53. What are the properties of optimizing compilers? May/June-2016

- The transformation must preserve the meaning of programs.
- A transformation must, on the average, speed up programs by a measurable amount
- The transformation must be worth the effort.

54. Identify the constructs for optimization in basic block. (Nov/Dec-2016)

Use of algebraic identities greatly optimizes the basic flow graphs. They are important class of optimization technique. The following identities are widely used:

- x+0=0+x=x
- x-0=x
- x* 1=1 *x=x
- x/ 1=z
- x* *2=x*x

- x*3.0=x+ x+ x
- x/2=x* 0.5
- Constant folding
- Associative laws
- Commutative laws

In constant folding, the expressions involving constants are evaluated and they are directly substituted. For instance,

a=6.45+2

b=a+4

These could be transformed as: 6.45+2+4=12.45. Now b=12.45. The value of b could be directly substituted instead of computing them.

55. Write three address code sequences for the assignment statement
D :=(a-b) + (a-c) + (a-c)(May/June 2016)

The assignment d: = (a-b) + (a-c) + (a-c) might be translated into the following three- address code sequence:

t : = a – b

u : = a – c v : = t + u d : = v + u

With d live at the end.

PART-B

INTRODUCTION

- The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.

Optimizations are classified into two categories. They are

- Machine independent optimizations:
- Machine dependant optimizations:

Machine independent optimizations:

- Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

Machine dependant optimizations:

- Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

The criteria for code improvement transformations:

- Simply stated, the best program transformations are those that yield the most benefit for the least effort.

The transformation must preserve the meaning of programs.

- That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the" safe" approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

A transformation must, on the average, speed up programs by a measurable amount.

- We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an "optimization" may slow down a program slightly.

The transformation must be worth the effort.

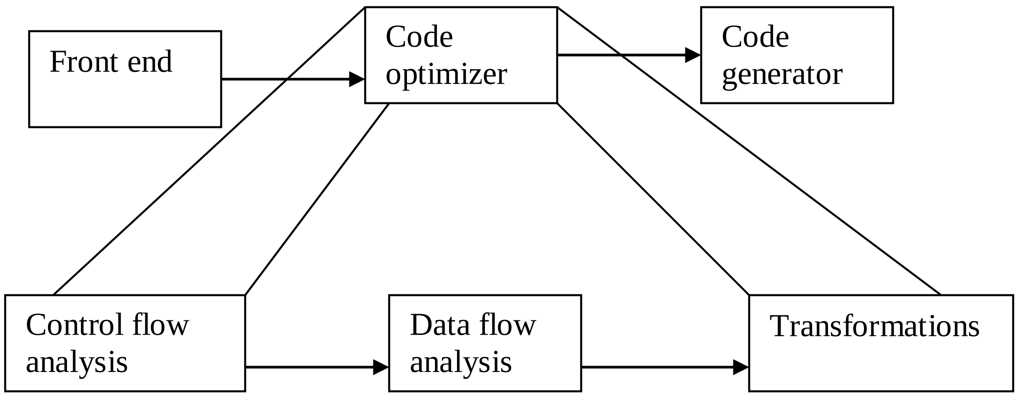
- It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. "Peephole" transformations of this kind are simplevbenough and beneficial enough to be included in any compiler.

Organization for an Optimizing Compiler:

- Flow analysis is a fundamental prerequisite for many important types of code Improvement.

✓ Generally control flow analysis precedes data flow analysis.
✓ Control flow analysis (CFA) represents flow of control usually in form of graphs,
✓ CFA

Fig : **Organization for an Optimizing Compiler:**



Constructs such as

- control flow graph
- Call graph
- Data flow analysis (DFA) is the process of asserting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

1. Explain the principal sources of optimization in details (8 marks) May/Jun-2012)(May/June-2013) Apr/May -2005 Nov/Dec -2004 Apr/May -2011(May/June-2016)

Local

A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.

Global

Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

- **Function-Preserving Transformations**
- **Loop optimizations**

Function-Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes.

The transformations

- **Common sub expression elimination,**
- **Copy propagation,**
- **Dead-code elimination, and**

- **Constant folding**

are common examples of such function-preserving transformations.

- The other transformations come up primarily when global optimizations are performed.
- Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

Common Sub expressions elimination:

- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recompiling the expression if we can use the previously computed value.

For example

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t4: = 4*i
t5: = n
t6: = b [t4] +t5
```

The above code can be optimized using the common sub-expression elimination as

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t5: = n
t6: = b [t1] +t5
```

The common sub expression t4: =4*i is eliminated as its computation is already in t1. And value of i is not been changed from definition to use.

Copy Propagation (Variable propagation)

- Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

For example:

x=Pi;

……

A=x*r*r;

The optimization using copy propagation can be done as follows:

A=Pi*r*r;

Here the variable x is eliminated

Dead-Code Eliminations:

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.

Example:

i=0;

if(i=1)

{

a=b+5;

}

Here, 'if' statement is dead code because this condition will never get satisfied.

Constant folding:

- We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.
- One advantage of copy propagation is that it often turns the copy statement into dead code.

For example,

a=3.14157/2 can be replaced by

a=1.570 there by eliminating a division operation.

Loop Optimizations:

- We now give a brief introduction to a very important place for optimizations, namely loops, 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.

Three techniques are important for loop optimization:

code motion,

This moves code outside a loop;

Induction-variable

Elimination, which we apply to replace variables from inner loop.

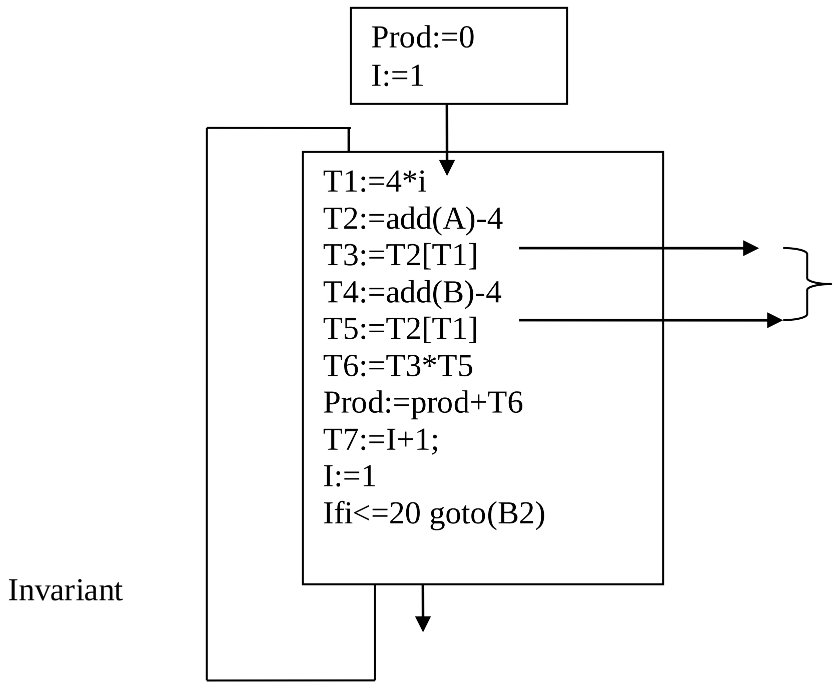
Reduction in strength,

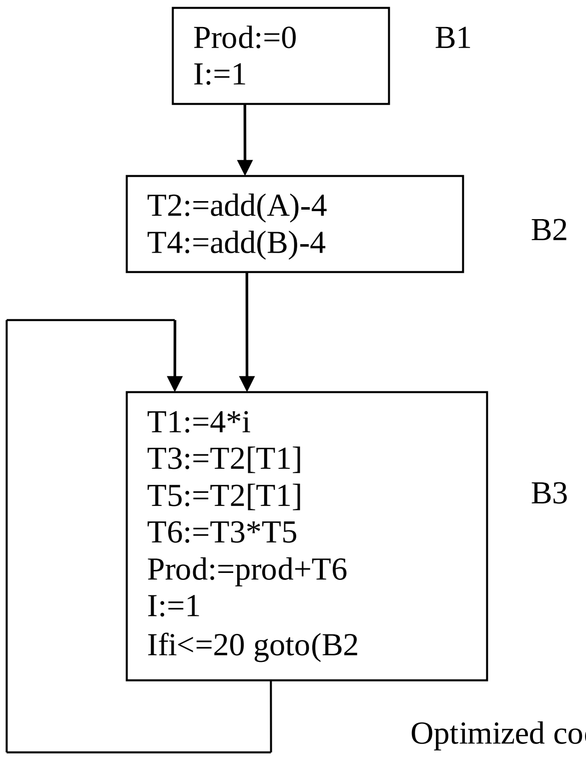
Which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

Code Motion:

- An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result **independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop.**
- Note that the notion "before the loop" assumes the existence of an entry for the loop.

Unoptimized Code



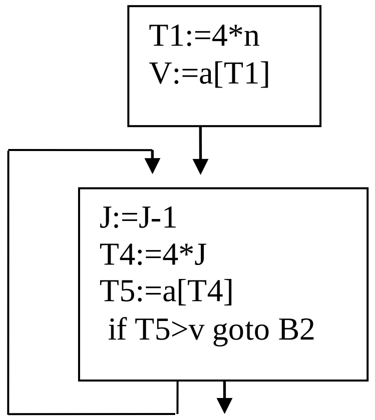


Optimized code

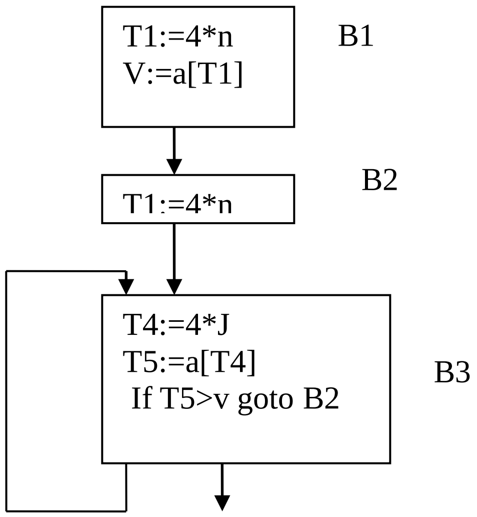
Induction Variables elimination:

- Loops are usually processed inside out. Any two variables are said to be induction variables if there is a change in any one of the variable, then there will be corresponding change in the other variable. The variables X,Y are said to be induction variables ,if X is incremented then Y will also be correspondingly incremented.

Un optimized code



Optimized code



Reduction In Strength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example,

- x^2 is invariably cheaper to implement as x*x than as a call to an Exponentiation routine.
- Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented
- as multiplication by a constant, which may be cheaper

2. Explain the DAG Representation For Basic Block with an Example.
(APR/MAY – 2011)(May/June-2013) (May/June-2014)

A DAG for a basic block is a **directed acyclic graph** with the following labels on nodes:

1. Leaves are labeled by unique identifiers, either variable names or constants.
2. Interior nodes are labeled by an operator symbol.
3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.

- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent statements.
- It provides a good way of determining common sub - expressions.

Algorithm for construction of DAG

Input: A basic block

Output: A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.

Case (i) x: = y OP z

Case (ii) x: = OP y

Case (iii) x: = y

Method:

Step 1: If y is undefined then create node(y).

If z is undefined, create node (z) for case (i).

Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is node(z) (Checking for common sub expression). Let n be this node.

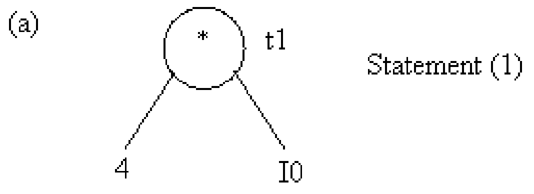
For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node. For case(iii), node n will be node(y).

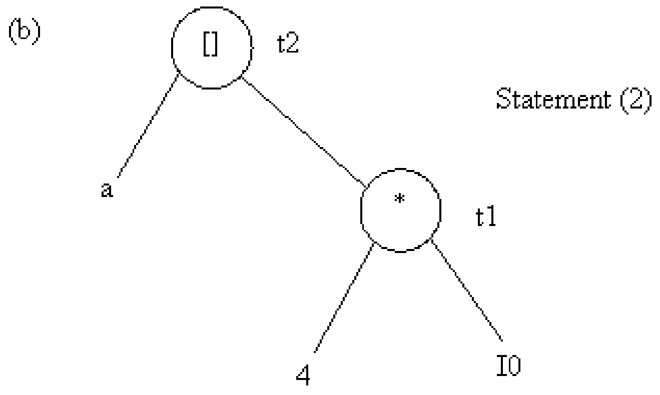
Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached identifiers for the node n found in step 2 and set node(x) to n.

Example: Consider the block of three- address statements:

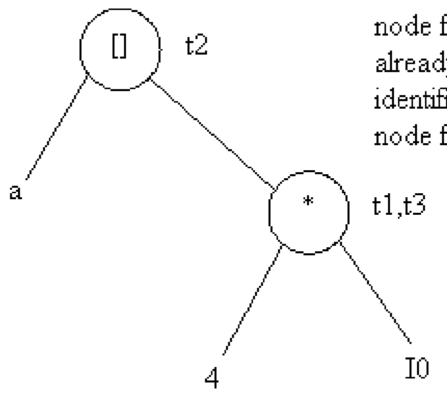
```
1.  t1  := 4* i
2.  t2  := a[t1]
3.  t3  := 4* i
4.  t4  :=  b[t3]
5.  t5  := t2*t4
6.  t6  := prod+t5
7.  prod := t6
8.  t7  := i+1
9.  i := t7
10. if i<=20 goto (1)
```

Stages in DAG Construction



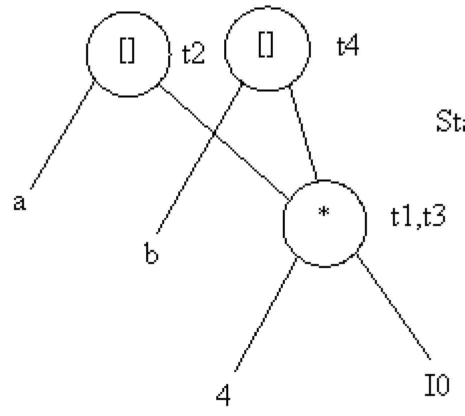


(c)



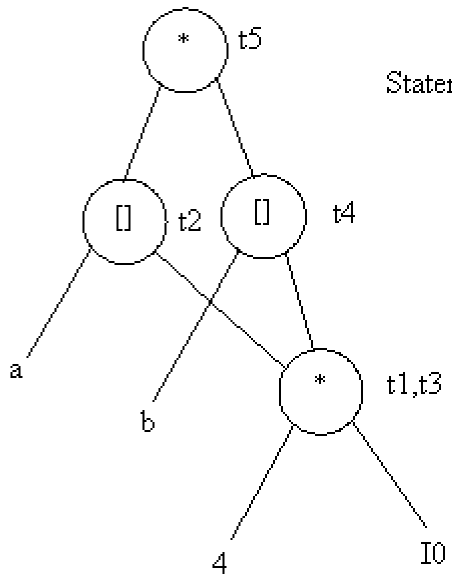
node for 4*I0 exist already, hence attach identifier t3 to the existing node for Statement (3)

(d)



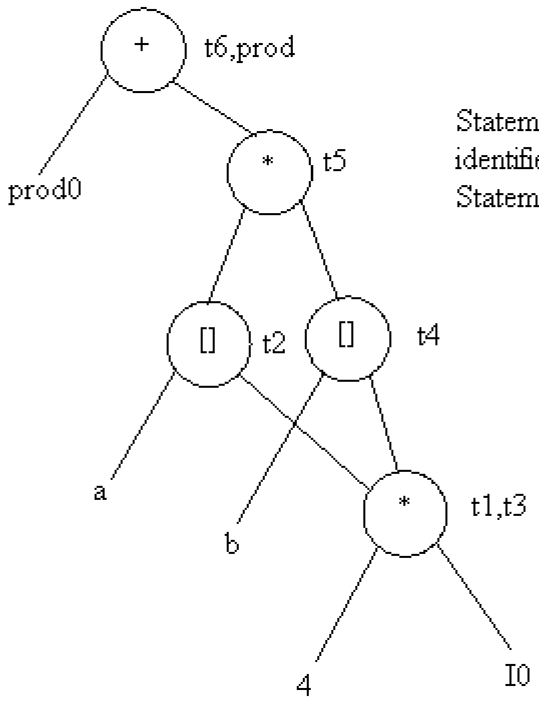
Statement (4)

(e)



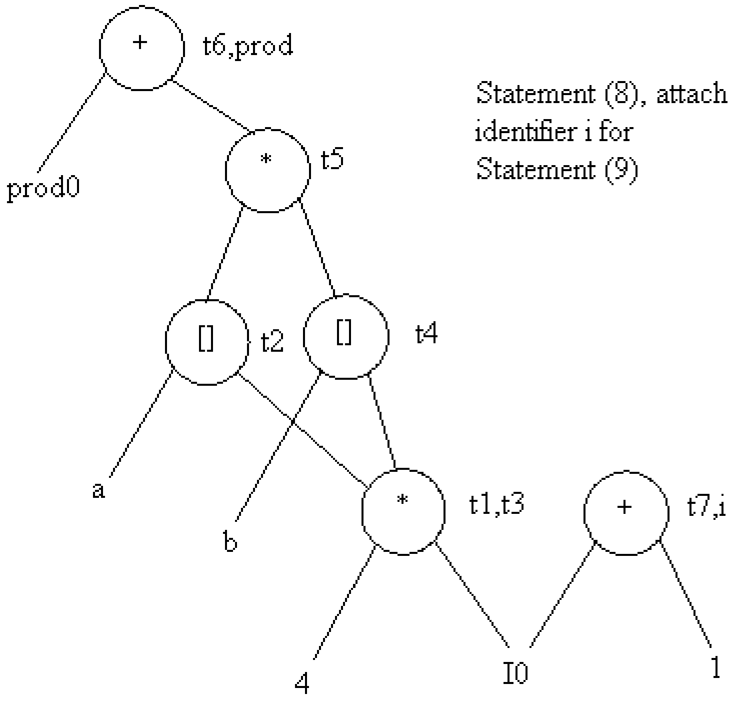
Statement (5)

(f)

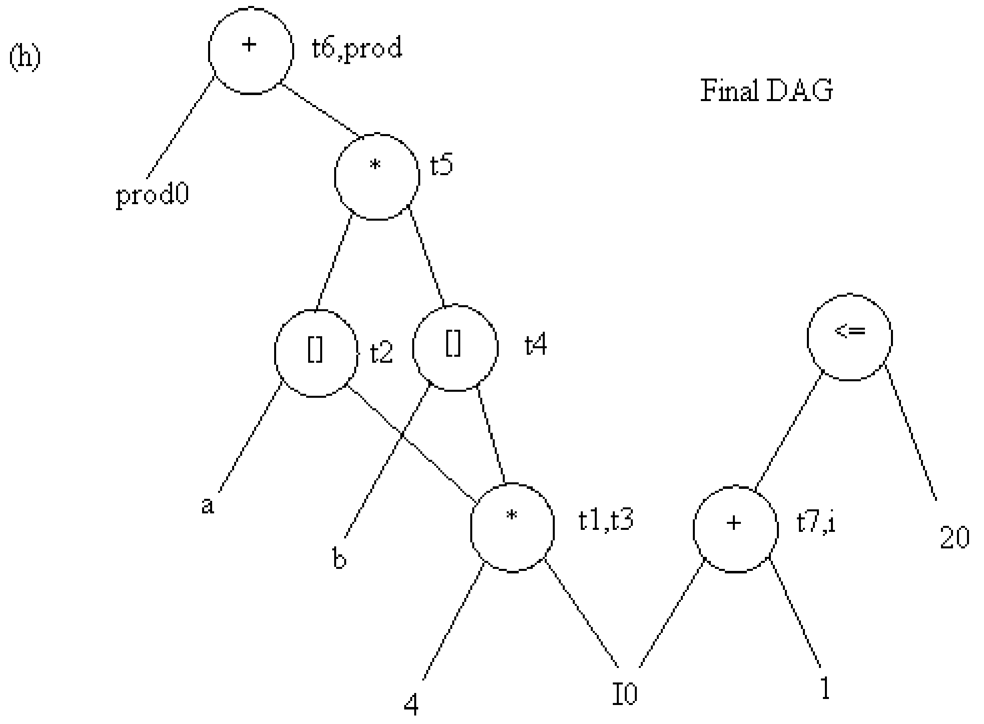


Statement (6), attach identifier prod for Statement (7)

(g)



Statement (8), attach identifier i for Statement (9)



Application of DAGs:

1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.

3. Explain the Generating Code From DAGS

The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than we can start from a linear sequence of three-address statements or quadruples.

Rearranging the order

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

t_1: = a + b

t_2: = c + d

t_3: = e – t_2

t_4: = t_1 – t_3

Generated code sequence for basic block:

MOV a, R_0

ADD b, R_0

MOV c, R_1

```
ADD d, R1
MOV R0, t1
MOV e, R0
SUB R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4
```

Rearranged basic block:

Now t1 occurs immediately before t4.

t_2: = c + d

t_3: = e – t_2

t_1: = a + b

t_4: = t_1 – t_3

Revised code sequence:

```
MOV c, R0
ADD d, R0
MOV a, R0
SUB R0, R1
MOV a, R0
ADD b, R0
SUB R1, R0
MOV R0, t4
```

In this order, two instructions **MOV R0, t1 and MOV t1, R1** have been saved.

Heuristic ordering for Dags

The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.

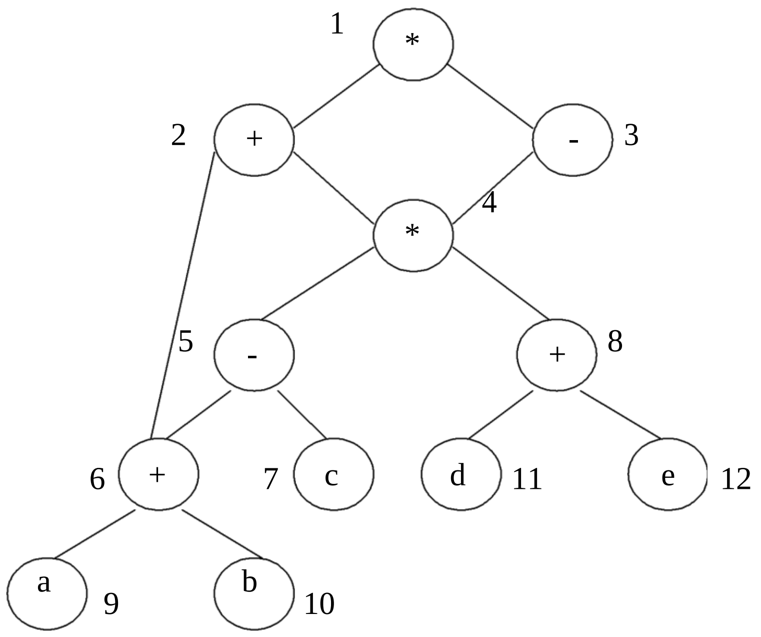
The algorithm shown below produces the ordering in reverse.

Algorithm:

```
1) while unlisted interior nodes remain do begin
2)     Select an unlisted node n, all of whose parents have been listed;
3)     list n;
4)     while the leftmost child m of n has no unlisted parents and is not a leaf do begin
5)        list m;
```

```
6)        n : = m
end
end
```

Example: Consider the DAG shown below:



Initially, the only node with no unlisted parents is 1 so set n=1 at line (2) and list 1 at line (3).

Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set n=2 at line (6).

Now, at line (4) we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus we select a new n at line (2), and node 3 is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that.

The resulting list is 1234568 and the order of evaluation is 8654321.

Code sequence:

t_8 : = d + e

t_6 : = a + b

t_5 : = t_6 – c

t_4 : = t_5 * t_8

t_3 : = t_4 – e

t_2 : = t_6 + t_4

t_1 : = t_2 * t_3

This will yield an optimal code for the DAG on machine whatever be the n umber of registers.

4. Generate DAG representation of the following of the following code and list out the application of DAG representation (Nov/Dec 2013)

i= 1;

s = 0;

While (i <= 10)

s = s +a[i][j];

i = i+1;

Solution:

The three address code generated for the following code segment is as follows

I. i = 1;
II. S = 0;
III. If i <= 10 goto (5)
IV. goto (13)
V. t1 = i * n2
VI. t1 = t1 + j

VII. t2 = c

VIII. t3 = 4 * t1

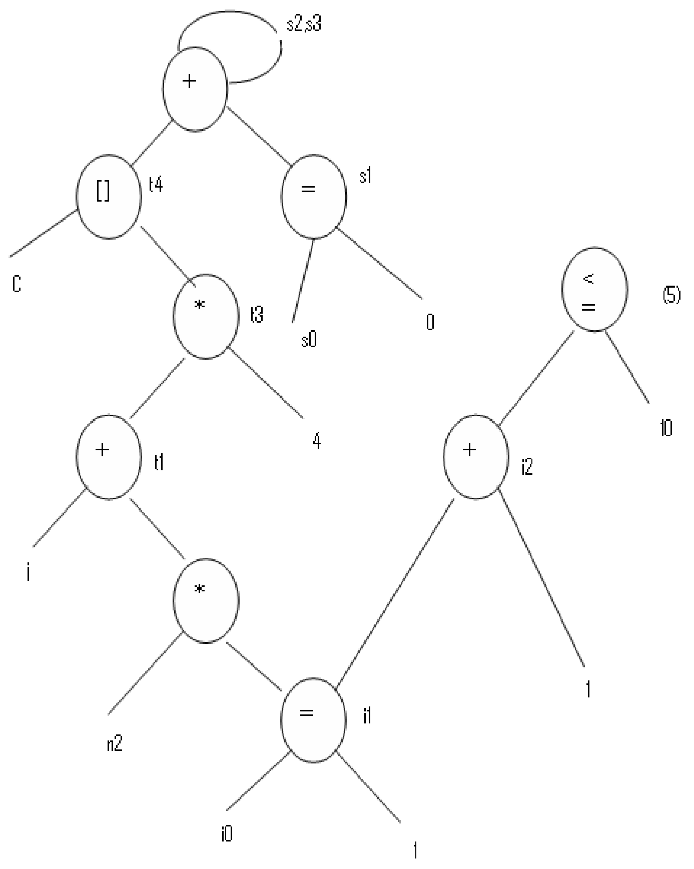
IX. t4 = t2[t3]

X. S = S +t4

XI. i = i+1;

XII. Go to (3)

DAG



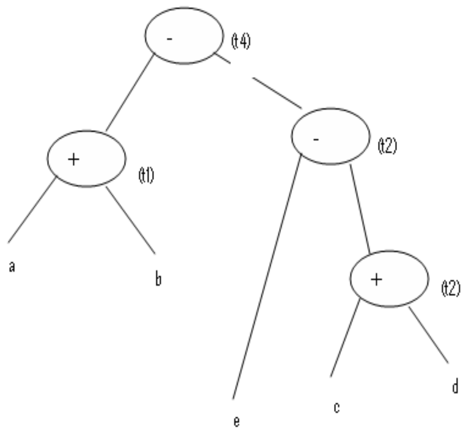
5. Construct DAG for the following basic block (8 marks) APR/MAY-2010

T_1:= A+B;

T_2:= C+D;

T_3:= E-T2;

T_4:= $T_1 - T_3$



6. Construct the DAG for the following basic block. (Ref .Pg.no.25 Qus .no .8)

(Apr/May-12) Nov/Dec 13)

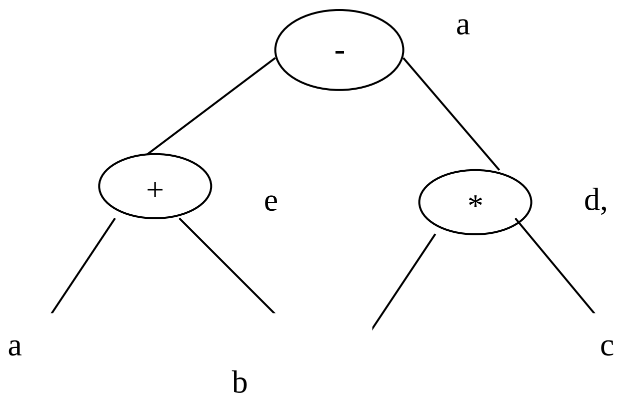
d := b * c

e :=a + b

b := b * c

a := e – d

Ans:



7. Explain in detail optimization of basic blocks with an example (8 marks Nov/Dec-2011) May/June -2009, May/June-14

There are two types of basic block optimizations. They are:

- Structure-Preserving Transformations
- Algebraic Transformations

Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

- Common sub-expression elimination
- Dead code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements.

Common sub-expression elimination:

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

```
a: =b +c
b: =a-d
c: =b +c
d: =a-d
```

The 2nd and 4th statements compute the same expression: b + c and a-d

Basic block can be transformed to

```
a: = b +c
b: = a-d
c: = a
d: = b
```

Dead code elimination:

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

Renaming of temporary variables:

A statement t:= b +c where t is a temporary name can be changed to u:= b+ c where u is another temporary name, and change all uses of t to u.

In this we can transform a basic block to its equivalent block called normal-form block.

Interchange of two independent adjacent statements:

Two statements
t1:=b +c
t2:=x +y

can be interchanged or reordered in its computation in the basic block when value of t1 does not affect the value of t2.

Algebraic Transformations:

Algebraic identities represent another important class of optimizations on basic blocks.

Use of algebraic identifiers:

Arithmetic identities:

- X + 0 = 0 + x =x

- X * 1 = 1 1* x= x
- x-0=x
- x/1=x

This includes simplifying expressions or replacing expensive operation by cheaper ones

Reduction in strength

- X**2 = X*x
- 2*x=x + x
- x/2=x*0.5

Constant folding

Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.

8. Explain the global data flow analysis [Nov/Dec 2013] [Nov/Dec 2016]

- Compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.
- A compiler could take advantage of "reaching definitions" , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as **data-flow analysis**.
- **Data-flow information** can be collected by setting up and solving systems of equations of the form :

  out [S] = gen [S] U (in [S] – kill [S])
- This equation can be read as "the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement." Such equations called **dataflow equations.**

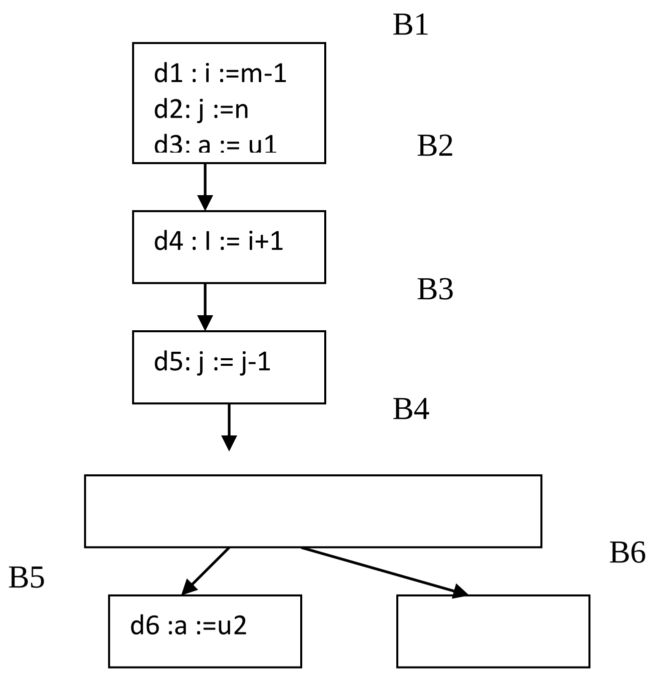
The details of how data-flow equations are set and solved depend on three factors.

- The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms of out[s].
- Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.

- There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Points and Paths:

- Within a basic block, the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.



- Now let us take a global view and consider all the points in all the blocks. A path from p1 to pn is a sequence of points p1, p2,….,pn such that for each i between 1 and n-1, either

1. Pi is the point immediately preceding a statement and pi+1 is the point immediately following that statement in the same block, or
2. Pi is the end of some block and pi+1 is the beginning of a successor block.

Reaching definitions:

- A definition of variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x.

- These statements certainly define a value for x, and they are referred to as **unambiguous Definitions** of x.
- There are certain kinds of statements that may define a value for x; they are called **ambiguous** definitions. The most usual forms of **ambiguous** definitions of x are:
- A call of a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure.
- An assignment through a pointer that could refer to x.
- **For example,** the assignment *q: = y is a definition of x if it is possible that q points to x. we must assume that an assignment through a pointer is a definition of every variable.
- We say a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path. Thus a point can be reached by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.

9. How to trace Data-flow analysis of structured programs (6 marks)
May /Jun2012) (May/June-2013) Nov/Dec -2011[May/June-14]

- Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.

  S ->id: = E| S; S | if E then S else S | do S while E

  E-> id + id| id

- Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.

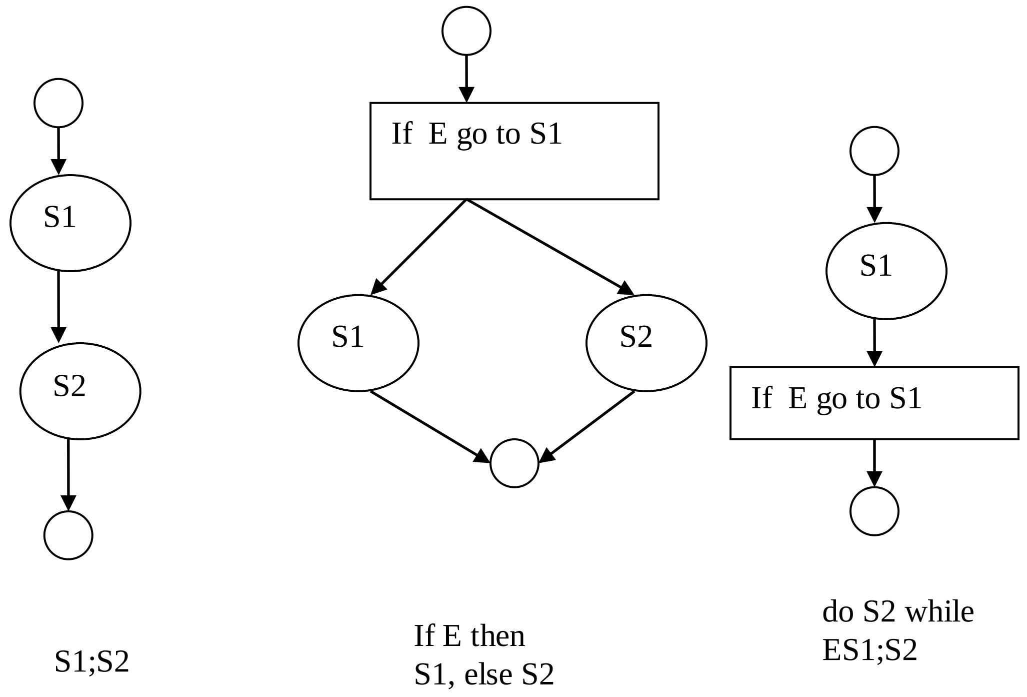
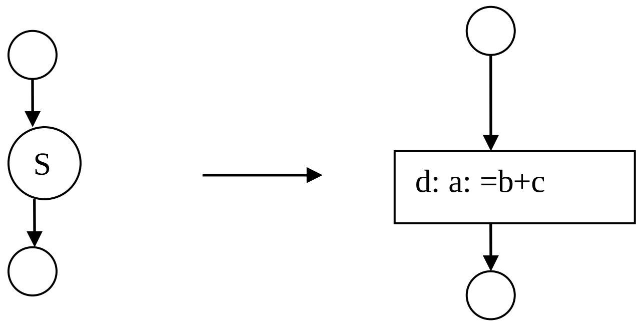


Fig : some structured controlled constructs

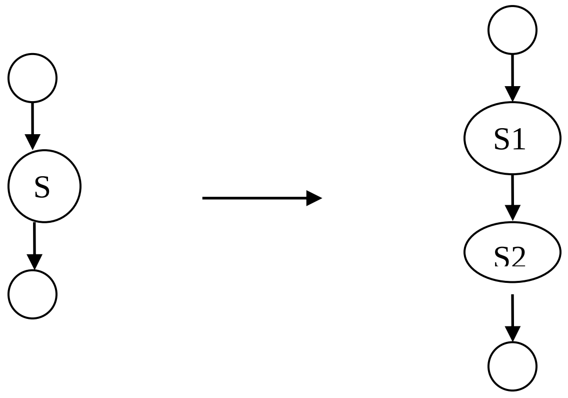
- We define a portion of a flow graph called a *region* to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header.
- The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.
- We say that the beginning points of the dummy blocks at the entry and exit of a statement's region are the beginning and end points, respectively, of the statement. The equations are inductive, or syntax-directed, definition of the sets **in[S], out[S], gen[S], and kill[S]** for all statements S.
- **gen[S] is the set of definitions "generated" by S while kill[S] is the set of definitions that never reach the end of S.**
- Consider the following data-flow equations for reaching definitions:



gen [S] = { d }
kill [S] = D_a – { d }
out [S] = gen [S] U (in[S] – kill[S])

- Observe the rules for a single assignment of variable a. Surely that assignment is a definition of a, say d. Thus
    - **Gen[S]={d}**
- □On the other hand, d "kills" all other definitions of a, so we write
    - **Kill[S] = Da – {d}**
- Where, Da is the set of all definitions in the program for variable a.

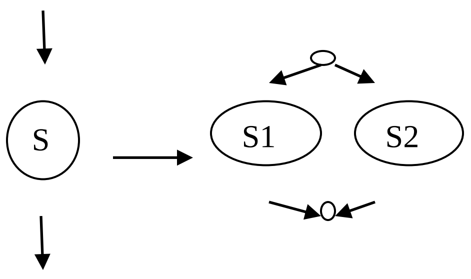
ii)



gen[S]=gen[S_2] U (gen[S_1]-kill[S_2])
Kill[S] = kill[S_2] U (kill[S_1] – gen[S_2])
in [S_1] = in [S]
in [S_2] = out [S_1]
out [S] = out [S_2]

- Under what circumstances is definition d generated by S=S1; S2? First of all, if it is generated by S2, then it is surely generated by S. if d is generated by S1, it will reach the end of S provided it is not killed by S2. Thus, we write
    - **gen[S]=gen[S2] U (gen[S1]-kill[S2])**
- Similar reasoning applies to the killing of a definition, so we have **Kill[S] = kill[S2] U (kill[S1] – gen[S2])**

iii)



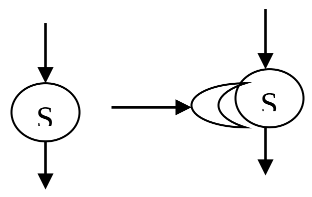
gen[S]=gen[S1]Ugen[S2]

kill[S]=kill[S2]∩kill[S1]

in[S1]=in[S]

in[S2]=in[S]
out[S]=out[S1]U out[S2]

iv)



gen[S]=gen[S1]
kill[S]=kill[S1]
in[S1]=in[S]U gen[S1]
out[S] = out[S1]

Conservative estimation of data-flow information:
There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are "Uninterrupted"; that is, there exists inputs to the program that make their branches go either way.

We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input.
When we compare the computed gen with the "true" gen we discover that the true gen is always a subset of the computed gen. on the other hand, the true kill is always a superset of the computed kill.
These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer lies in the use intended for these data.
Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth.

Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached.

Decreasing kill can only increase the set of definitions reaching any given point.

Computation of in and out:

- Many data-flow problems can be solved by synthesized translations similar to those used to compute gen and kill. It can be used, for example, to determine loop-invariant computations.
- However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in. we intend that in[S] be the set of definitions reaching the beginning of S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.
- The set out[S] is defined similarly for the end of s. it is important to note the distinction between out[S] and gen[S]. The latter is the set of definitions that reach the end of S without following paths outside S.
- Assuming we know in[S] we compute out by equation, that is

  Out[S] = gen[S] U (in[S] - kill[S])
- Considering cascade of two statements S1; S2, as in the second case. We start by observing in[S1]=in[S]. Then, we recursively compute out[S1], which gives us in[S2], since a definition reaches the beginning of S2 if and only if it reaches the end of S1. Now we can compute out[S2], and this set is equal to out[S].
- Considering if-statement we have conservatively assumed that control can follow either branch, a definition reaches the beginning of S1 or S2 exactly when it reaches the beginning of S.

  In[S1] = in[S2] = in[S]
- If a definition reaches the end of S if and only if it reaches the end of one or both sub statements; i.e,

  Out[S]=out[S1] U out[S2]

Representation of sets:

- Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph.

Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is in the set.

- The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.
- A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming languages. The difference A-B of sets A and B can be implemented by taking the complement of B and then using logical and to compute A .
- **Local reaching definitions:**
- Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, re computing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.
- Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

Use-definition chains:

- It is often convenient to store the reaching definition information as" use-definition chains" or "ud-chains", which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable a in block B is preceded by no unambiguous definition of a, then ud-chain for that use of a is the set of definitions in in[B] that are definitions of a.in addition, if there are ambiguous definitions of a ,then all of these for which no unambiguous definition of a lies between it and the use of a are on the ud-chain for this use of a.

Evaluation order:

- The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after all uses of it have occurred.

- Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

General control flow:

- Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in syntax directed manner.
- When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.
- Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval-based methods
- However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

10. Write in detail about the issues in the design of a code generator (APR\MAY-2011& NOV\DEC)(May/June-14) (Nov/Dec-16)

(Or)

Write on the issues in code generation and generate the assembly language for the statement

W:= (A+B) +(A+C) +(A+C) (16 Marks) (Nov/Dec-16)

The following issues arise during the code generation phase:

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

- The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

- Intermediate representation can be :
  - Linear representation such as postfix notation
  - Three address representation such as quadruples
  - Virtual machine representation such as stack machine code d. Graphical representations such as syntax trees and dags.

Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free

2. Target program:

- The output of the code generator is the target program. The output may be :

  a. Absolute machine language

- It can be placed in a fixed memory location and can be executed immediately.

  b. Relocatable machine language

- It allows subprograms to be compiled separately.

  c. Assembly language

- Code generation is made easier.

3. Memory management:

- Names in the source program are mapped to addresses of data objects in run- Time memory by the front end and code generator.
- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
- Labels in three-address statements have to be converted to addresses of Instructions.

For example,

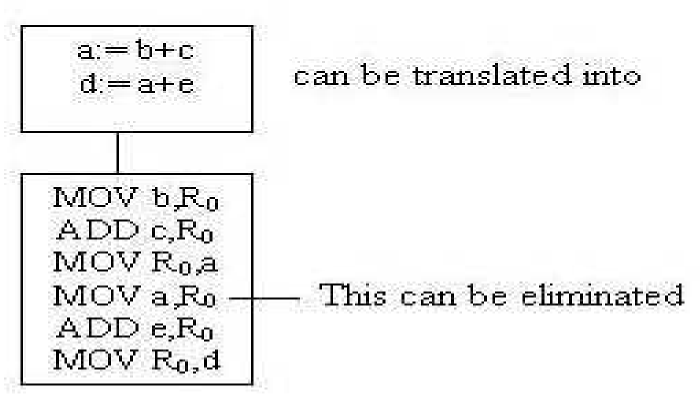
j : **goto** *i* generates jump instruction as follows :

- if $i < j$, a backward jump instruction with target address equal to location of code for quadruple *i* is generated.
- if $i > j$, the jump is forward. We must store on a list for quadruple *i* the

location of the first machine instruction generated for quadruple *j*. When *i* is processed, the machine locations for all instructions that forward jumps to *i* are filled.

4. **Instruction selection:**

- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size.
- The former statement can be translated into the latter statement as shown below:



5. **Register allocation**

- Instructions involving register operands are shorter and faster than those involving operands in memory.
- The use of registers is subdivided into two subproblems :
- ***Register allocation*** – the set of variables that will reside in registers at a point in the program is selected.
- ***Register assignment*** – the specific register that a variable will reside in is picked.
- Certain machine requires even-odd *register pairs* for some operands and results.

For example , consider the division instruction of the form :

D x, y

where, x – dividend even register in even/odd register pair y – divisor

even register holds the remainder odd register holds the quotient

6. **Evaluation order**

- The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

QUS: W :=(A+B) + (A+C) + (A+C)

ANS: THREE ADDRESS CODE IS

```
T1:= A+B
T2:= A+C
T3:= T2+T2
T4:= T1+T3
```

CODE GENERATION IS

```
MOV A, R0
ADD B,R0
MOV A, R1
ADD C,R1
ADD C,R1
ADD R1,R1
ADD R0,R1
```

11. Briefly explain about simple code generator (APR/MAY 2011 & NOV/DEC - 10 marks) (May/June-2013)(May/June-2016)

- For simplicity, we will assume that for each intermediate code operator we have a corresponding target code operator.
- We will also assume that computed results can be left in registers as long as possible.
- If the register is needed for another computation, the value in the register must be stored.
- Before we leave a basic block, everything must be stored in memory locations.
- We will try to produce reasonable code for a given basic block.
- The code-generation algorithm will use descriptors keep track of register contents and addresses for names.

Register Descriptors

- A register descriptor keeps track of what is currently in each register.
- It will be consulted when a new register is needed by code-generation algorithm.
- We assume that all registers are initially empty before we enter into a basic block. This is not true if the registers are assigned across blocks.
- At a certain time, each register descriptor will hold zero or more names.

```
                    R1 is empty
MOV  a,R1
                    R1 holds a
```

```
MOV  R1,b
R1 holds both a and b
```

Address Descriptors

- An address descriptor keeps track of the locations where the current value of a name can be found at run-time.
- The location can be a register, a stack location or a memory location (in static area). The location can be a set of these.
- This information can be stored in the symbol table.

```
                        a is in the memory
MOV  a,R1
                        a is in R1 and in the memory
MOV  R1,b
                        b is in R1 and in the memory
```

A Code Generation Algorithm

- This code generation algorithm takes a basic block of three-address codes, and produces machines codes in our target architecture.
- For each three-address code we perform certain actions.
- We assume that we have *getreg* routine to determine the location of the result of the operation.

Code Generation Actions for x := y op z

1. Invoke getreg function determine the location Lx of the result (x).
2. Consult the address descriptor of y to determine the location Ly for y. Prefer a register for Ly if y is in both in a register and in a memory location. If y is not already in Lx, generate the instruction MOV Ly,Lx
3. Generate the instruction OP Lz, Lx where Lz is the location of z. If z is in both in a register and in memory, prefer the register. Update the address descriptor of x to indicate that x is in Lx. If Lx is a register update its descriptor to indicate that it contains x. Remove x from all other register descriptors.
4. If y and z has no next uses, are not live on the exit from block, and they are in registers, change the register descriptors so that they do not contain y or z respectively.
5. At the end of block, if a name is live after block and it is not in memory, we generate a move instruction for this name.

Function getreg for Lx (x: = y op z)

1. If y is in a register that does not hold no other names, and y is not live and has no next use after x:=y op z, then return the register of y as Lx. Update the address descriptor of y to indicate that y is no longer in Lx.

2. If (1) fails, return an empty register for Lx if there is one.

3. If (2) fails ➔ if x has a next use in the block (or OP is a special instruction needs a register), find a suitable occupied register and empty that register.

4. If x is not used in the block, or no suitable register can be found, select the memory location of x as Lx.

5. A more sophisticated getreg function can be designed.

Code Generation Example

Generating Code for Assignment Statements:

- The assignment d: = (a-b) + (a-c) + (a-c) might be translated into the following three- address code sequence:

t : = a – b

u : = a – c v : = t + u d : = v + u

with d live at the end.

Code sequence for the example is:

Statements	**Code Generated**	**Register descriptor**	**Address descriptor**
		Register empty	
t : = a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a - c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d : = v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 and memory

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements

a : = b [i] and **a [i] : = b**

Statements	Code Generated	Cost
a : = b[i]	MOV b(R_i), R	2
a[i] : = b	MOV b, a(R_i)	3

Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments **a : = *p** and ***p : = a**

Statements	Code Generated	Cost
a : = *p	MOV *R_p, a	2
*p : = a	MOV a, *R_p	2

Generating Code for Conditional Statements

Statement	Code
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */
x : = y +z if x < 0 goto z	MOV y, R_0 ADD z, R_0 MOV R_0,x CJ< z

12. Discuss about efficient data flow algorithms.

There are two ways to use flow-graph theory to speed data-flow analysis is

- an application of depth-first ordering to reduce the number of passes that the iterative algorithms take, and
- the second uses intervals or the T_1 and T_2 transformations to generalize the syntax-directed approach.

Depth-First Ordering in iterative Algorithms

In all the problems studied *so far,* such as reaching definitions available expressions, or live variables, any event of significance at a node will be propagated to that node along an acyclic path. For example, if a definition d is in in[B], then there is some acyclic path from the block containing *d* to B such that *d* is in the *in's* and out's all along that path, Similarly, if an expression x+y is not available at the entrance to block *B,* then there is some acyclic path that demonstrates that fact; either the path is from the initial node and includes no statement that kills or generates x+y, or the path is from a block that kills x+y and along the path there is no subsequent generation of x+y.

Finally, for live variables, if *x* is live on exit from block *B,* then there is an acyclic path from B to a use of x, along with there are no definitions of x.

Check that in each of these cases, paths with cycles add nothing. For example, if a use of x is reached from the end of block B along a path with a cycle, we can eliminate that cycle to find a shorter path along which the use *of* x is still reached from B.

If all useful information propagates along acyclic paths, we have an opportunity to tailor the order in which we visit nodes in iterative data-flow algorithms *so* that after relatively few passes through the nodes, we can be sure information has passed along all the acyclic paths. In particular, show that typical flow graphs have a very low interval *depth,* which is the number of times one must apply the interval partition to reach the limit flow graph; an average of 2.75 was found. Furthermore, it can be shown that the interval depth of a flow graph is never less than what we have called the "depth," the maximum number of retreating edges on any acyclic path. (If the flow graph is: not reducible, the depth may depend on the depth-first spanning tree chosen)

We note that if a→ b is an edge, then the depth-first number of *b* is less than that of *b* only when the edge is a retreating edge. Thus, replace line (5), which tells us to visit each block *B* of the flow graph for which we are computing reaching definitions, by:

```
                **for** each block *B* in depth-first order **do**
```

Suppose we have a path along which a definition *d* propagates, such as

3→5→19→35→16→23→45→4→10→17

where integers represent the depth-first numbers of the blocks along the path.

Then the first time through the loop of lines (5)-(9) . *d* will propagate from out[3] to *in[5] and* so on, up to out[35]. It will not reach in[16] on that round, because as 16 precedes 35, we had already computed in[16] by the time *d* was put in out[35].However, the next time we run through the Ioop of Lines (5)-(9), when we compute in[16], *d* will be included *because* it is in *out[* 35]. Definition *d* will also propagate to out[16], *in[23],* and so on, up to out[45], where it must wait because in[4] was already computed.

On the third pass, d travels to in[4], out[4], in[10], out[10], and in[17], *so* after three passes we establish that d reaches block 17.

It should not be hard to extract the general principle from this example. If we use depth-first order, then the number of passes needed to propagate any reaching definition along any acyclic path is no more than one greater than the number of edges along that path that go from a higher numbered block to a lower numbered block. Those edges are exactly the retreating edges, *so* the number of passes needed is one plus the depth. Of course Algorithm does not detect the fact that all definitions have reached wherever they can reach for one more pass, *so* the upper bound on the number of passes taken by that algorithm with depth-first block ordering is actually two plus the depth.
The depth-first order is also advantageous for available expressions or any data flow problem that we solved by propagation in the forward direction. For problems like live variables, where we propagate backwards, the same average of five passes can be achieved if we chose the reverse of the depth-first order. Thus, we may propagate a use of a variable in block 17 backwards along the path

3→5→19→35→16→23→45→4→10→17

in one pass to in[4], where we must wait for the next pass to in order reach out[45]. On the second pass it reaches *in[*16], and on the third pass it goes from our[35] to out[3]. In general, one the depth passes suffices to carry the use of a variable backward, along any acyclic path, if we choose the reverse of depth-first order to visit the nudes in a pass, because then, uses propagate along any decreasing sequence in a single pass,

Structure-Based Data-Flow Analysis

With a bit more effort, we can implement data-flow algorithms that visit nodes (and apply data-flow equations) no more times than the interval depth of the flow graph, and frequently the average node will be visited even fewer times than that. Whether the extra effort results in a true time savings has not been firmly established, but a technique like this one, based on interval analysis, has been used in several compilers. Further, the ideas exposed here apply to syntax-directed data-flow algorithms For all sorts of structured control statements, not just the if ... then and do . . while, and these have also appeared in several compilers.

We shall base our algorithm on the structure induced on flow graphs by the T_1, and T_2 transformations. We are concerned with the definitions that are generated and killed as control flows through a region. Unlike the regions defined by if or while statements, a general region can have multiple exits, so for each block *B* in region R we shall compute sets $gen_{R,B}$ and kill $_{R,B}$ of definitions generated and killed, respectively, along paths within the region from the header to *the* end of block B, These sets will be used to define a t*ransfer* function $trans_{R,B}(S)$ *th*at tells for any set S of definitions, what set of definitions reach the end of block B by traveling along paths wholly within R, given that all and only the definitions in *S* reach the header of *R.*

The definitions reaching the end of block *B* fall into two classes.

I . Those that are generated within *R* and propagate to the end of *B i*ndependent of S.

2. Those that are no1 generated in *R,* but that also are not killed along some path from the header of *R* to the end of *B,* and therefore are in $trans_{R,B}(S)$ if and only if they are in S.

Thus, we may write *trans* in the form:

$$\mathbf{trans_{R,B}(S)=gen_{R,B}\ U\ (S-kill_{R,B})}$$

The heart of the algorithm is a way to compute $trans_{R,B}$ for progressively larger regions defined by some (*T1I* , T2)-decomposition of a flow graph. For the moment, we assume

that the flaw graph is reducible, although a simple modification allows the algorithm to work for nonreducible graphs as well.
The basis is a region consisting of a single block, *B*. Here the transfer function of the region is the transfer function of the block itself, since a definition reaches the end of the block if and only if it is either generated by the block or is in the set S and not killed. That is,

$\mathbf{gen_{R,B}=gen[B]}$
$\mathbf{kil_{R,B}=kill[B]}$

NOW, let us consider the construction of a region *R* by *T2;* that is, *R* is formed when *R ,consumes R2*. First, note that within region *R* there are no edges from *R2* back to *R*, since any edge from *R2* to the header of *R* , is not a part of R. Thus any path totally within *R* goes through R first, then through *R2,* but cannot then return to *R1* . A h note that the header of *R* is the header of , . We may conclude that within R, *R2* does not affect the transfer function of nodes in *R1;* that is,

$\mathbf{gen_{R,B}=gen_{R1,B}}$
$\mathbf{kill_{R,B}=kill_{R1,B}}$

for all *B* in *R1*
For *B* in R2, a definition can reach the end of 8 if any of the following conditions hold.
The definition is generated within R 2.
The definition is generated within *R1*, reaches the end of some predecessor of the header of R2, and is not killed going from the header of *R* to *B*.
The definition is in the set S available at the header of R1, not killed going to some predecessor of the header of *R1*, and not killed going from the header of *R2* to *B*.
Hence, the definitions reaching the ends of those blocks in R 1 that are predecessors of the header of *R2* play a special role, In essence, we see what happens to a set *S* entering the header of R , as its definitions try to reach the header of *R2,* via one of its predecessors. The set of definitions that reach one of the predecessors of the header of *RI* becomes the input set for *R 2* , and we apply the transfer functions for *R2* to that set.

Thus, let G be the union of genR1,P, for all predecessors *P* of the header of *R* 2, and let K be the intersection of killR1,P for all those predecessors P. Then if S the set of definition that reach the header of R I , the set of definitions that reach the header of *R2* along paths staying wholly within *R* is *G* U (S - *K)* . Therefore, the transfer function in *R* for those blocks *B* in *R2* may be computed by

$\mathbf{gen_{R,B}=gen_{R2,B}\ U\ (G - kill_{R2,B})}$
$\mathbf{kill_{R,B}=kill_{R2,B}\ U\ (K\text{-}gen_{R2,B})}$

Next, consider what happens when a region *R* is built from a region R1using transformation T1+ The general situation is note that *R* consists of *R1* plus some back edges to the header of *R1* (which is also the header of *R*, of course). A path going through the header twice would be cyclic and, as we argued earlier in the section, need not be considered. Thus, all definitions generated at the end of block B are generated in one of two ways.
1.The definition is generated within *R1* and does not need the back edges incorporated into *R* in order to reach the end of *B*.
2. The definition is generated somewhere within *R1,* reaches a predecessor of the header, follows one back edge, and is not killed going from the header to B.

If we let *G* be the union of genR1,P for all predecessors of the header in *R, t*hen
$\mathbf{gen_{R,B}=gen_{R1,B}\ U\ (G - kill_{R1,B})}$
A definition is killed ping from the header to *B* if and only if it is killed along all acyclic paths, so the back edges Incorporated into R do not cause more definitions to be killed. That is,
$\mathbf{kill_{R,B}=kill_{R1,B}}$

Algorithm: Structure-based reaching definitions.
Input: A reducible flow graph *G* and sets of definitions gen *[B]* and *kill[B]* for each block *B* of *G*.
Output: *in[B]* for each block *B*.
Method:

1. Find the (T1,T 2) decomposition for G.
2. For each region *R* in the decomposition, from the inside out, compute
$gen_{R,B}$ and $kill_{R,B}$ for each block *B* in *R,*
3. If U is the name of the region consisting of the entire graph, then for each block *B,* set in[B] to the union, over all predecessors *P* of block *B,* of *genU,P.*

Some Speedups to the Structure-Based Algorithm
First, notice that if we have a transfer function G U *(S - K)* , the function is not changed if we delete from *K* some members of *G.* Thus, when we apply *T2* , instead of using the formulas

$\mathbf{gen_{R,B}=gen_{R2,B}\ U\ (G\text{-}kill_{R2,B})}$
$\mathbf{kill_{R,B}=kill_{R2,B}\ U\ (K\text{-}gen_{R2,B})}$

we can replace the second by

$\mathbf{kill_{R,B}=kill_{R2,B}\ U\ K}$

thus saving an operation for each block in region *R2.*
Another useful idea is to notice that the only time we apply *T1,* is after we have first consumed some region *R2* by *R1,* and there are some back edges from *R2 to* the header of R1.Instead of first making changes in *R1 and R2* due to the T1 operation, we can combine the two sets of changes *if* we do the following.
1. Using the *T2* rule, compute the new transfer function for those nodes in *R2* that are predecessors *of* the header of *R* 1.
2. Using the *T1* rule, compute the new transfer function for all the nodes of R1.
3. Using the *T2* rule, compute the new transfer function for all the nodes of *R2.* Note that feedback due to the application of T1 has reached the predecessors of *R2* and is passed to all of *R2* by the *T1* rule; there is no need to apply the T1 rule for *R2.*

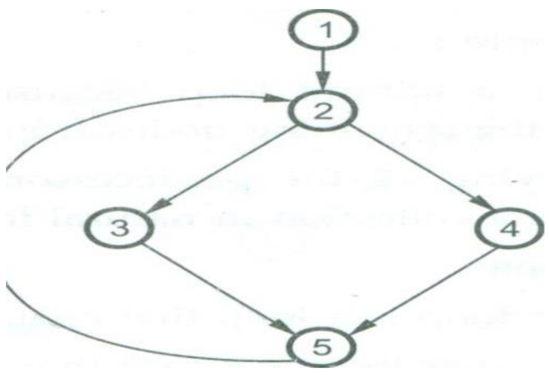
13. Write an algorithm for constructing natural loop of a back edge.Nov/Dec-2016

Let us get introduced with some common terminologies being used for loops in flow graph.

1. Dominators

In a flow graph, anode d dominates n if every path to node n from initial node goes through d only. This can be denoted as'd dom n'.every initial node dominated all the remaining nodes in the flow graph. Similarly every node dominates itself.

Example: in flow graph as shown in fig 8.6.1



Node 1 is initial node and it dominates every node as it is a initial node. Node 2 dominates 3, 4 and 5 as there exists only one path from initial node to node 2 which is going through 2 (it is 1-2-3). Similarly for node 4 the only path exists is 1-2-4 and for node 5 the only path existing is 1-2-4-5 or 1-2-3-5 which is going through node 2.

Node 3 dominates itself similarly node 4 dominates itself. The reason is that for going to remaining node 5 we have another path 1-4-5 which is not through 3 (hence 3 dominates itself). Similarly for going to node 5 there is another path 1-3-5 which is not through 4 (hence 4 dominates itself).

Node 5 dominates no node.

2. Natural loops

Loop in a flow graph can be denoted by n —> d such that d dom n. These edges are called back edges and for a loop there can be more than one back edge. If there is p —> q then q is a head and p is a tail. And head dominates tail.

For example :

The loops in the graph shown in Fig. 8.6.2 can be denoted by 4 --> 1 i.e. 1 dom 4. Similarly
5 —> 4 i.e. 4 dom 5.

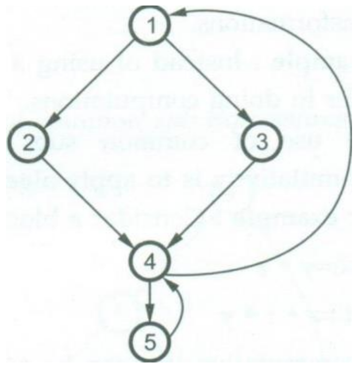


Fig. 8.6.2 Flow graph with loops

The natural loop can be defined by a back edge n --> d such that there exists a collection of all the nodes that can reach to n without going through d and at the same time d also can be added to this collection.

For example:

6 —> 1 is a natural loop because we can reach to all the remaining nodes from 6.

By observing all the predecessors of node 6 we can obtain a natural loop. Hence 2-3-4-5-6-1 is a collection in natural loop.

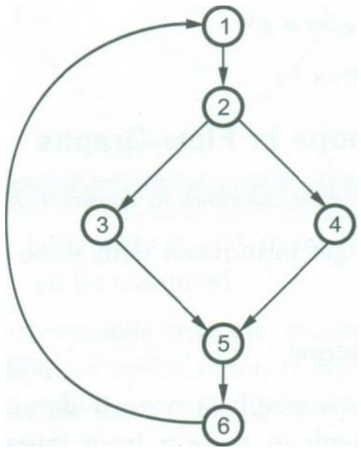


Fig. 8.6.3 Flow graph of natural loop

3. Inner loops

The inner loop is a loop that contains no other loop.

Here the inner loop is 4 —> 2 that means edge given by 2-3-4.

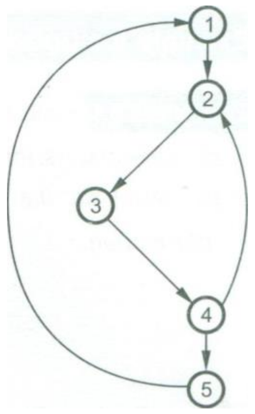


Fig. 8.6.4 Flow graph for inner loop

4. Pre-header

The pre-header is a new block created such that successor of this block is the header block. All the computations that can be made before the header block can be made before the pre-header block.

The pre-header can be as shown in Fig. 8.6.5

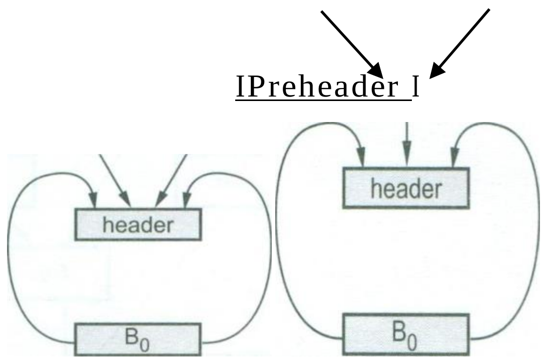


Fig. 8.6.5 Pre-header

5. Reducible flow graphs

The reducible graph is a flow graph in which there are two types of edges forward edges and backward edges. These edges have following properties,

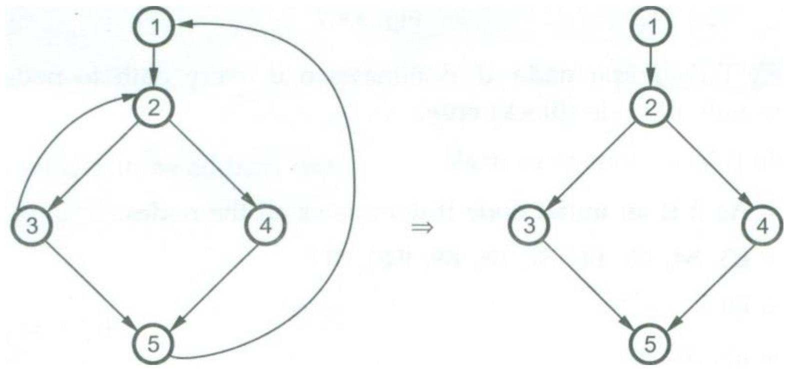
i) The forward edge forms an acyclic graph.

ii)The back edges are such edges whose head dominates their tail.

The flow graph is reducible as shown in Fig. 8.6.6. We can reduce this graph by removing the back edge from 3 to 2 edges. Similarly by removing the back edge from 5 to 1 we can reduce the above flow graph. And the resultant graph is a cyclic graph.

The program structure in which there is exclusive use of if-then, while-do or goto statements generates a flow graph which is always reducible.

Fig. 8.6.6 Flow graph



ANNA UNIVERSITY QUESTIONS

Nov/Dec 2016

PART A

1. Identify the constructs for optimization in basic block. **(Q.No:54)**
2. What are the characteristics of peephole optimization? **(Q.No:40)**

PART B

1. (i) Write an algorithm for constructing natural loop of a back edge.(8) **(Q.No:13)**
   (ii) Explain any four issues that crop up when designing a code generator. (8)**(Q.No:10)**
2. Explain global data flow analysis with necessary equations. (16) **(Q.No:8)**

MAY/JUNE 2016

PART A

1. What are the properties of optimizing compiler? **(Q.No:53)**
2. Write three address code sequences for the assignment statement

   D :=(a-b) + (a-c) + (a-c). **(Q.No:55)**

PART B

1. (a) Explain Principal sources of optimization with examples. (16) **(Q.No:1)**
   (b) (i) Explain various issues in the design of code generator.(8) **(Q.No:10)**
   (ii) Write note on Simple code generator. (8) **(Q.No:11)**

MAY/JUNE 2014

PART-A

1. What do you mean by Cross-Compiler? **(Ref.pg.no10 , Q.no.54)**
2. How would you represent the dummy blocks with no statements indicated in global data flow analysis? **(Ref.pg.no.26 , Q.no.5)**

PART-B

1. Discuss in detail the process of optimization of basic blocks. Give an example (16) **(Ref.pg.no 19, Q.no.3)**
2. What is data flow analysis? Explain data flow abstraction with examples. (16) **(Ref.pg.no28 , Q.no.6)**

NOV/DEC-2013

PART-A

1. Define loop unrolling with example. **(Ref.pg.no , Q.no.)**
2. What is an optimizing compiler? **(Ref.pg.no.2 , Q.no.2)**

PART-B

1.(i) Write in details about loop optimization(8) **(Ref.pg.no.42 , Q.no.9)**
(ii)Discuss the characteristics of peephole optimization.(8) **(Ref.pg.no.17 , Q.no.2)**
2.Discuss in detail about global data flow analysis.(16) **(Ref.pg.no.26 , Q.no.5)**

MAY/JUNE 2013

PART-A

1. What is constant folding? **(Ref.pg.no 9, Q.no.50)**
2. What are the properties of optimizing compilers? **(Ref.pg.no.10, Q.no.53)**

PART-B

1. Write the principle sources of optimization.(16) **(Ref.pg.no.12, Q.no.1)**
2. (i)Explain the data-flow analysis of structured programs(8) **(Ref.pg.no.28, Qno.6)**
(ii)Write global common sub expression elimination algorithm with example.(8) **(Ref.pg.no.32, Q.no.7)**

May/Jun-2012

Part-A

1.What is the use of algebraic identifiers in optimization of basic blocks **Ref.pg.no.6, Q.no.35)**
2. List out two properties of reducible flow graph **(Ref.pg.no.6, Q.no.34)**

Part-B

1. Explain the principal sources of optimization in details (8 marks) **(Ref.pg.no.12, Q.no.1)**
2. Discuss the various peephole optimization techniques in detail(8)R**ef.pg.no.17, Q.no.2)**

3. How to trace dataflow analysis of structured program? Discuss(6) **(Ref.pg.no.28, Qno.6)**
4. Explain the common sub expression elimation, copy propagation and transformations for
moving loop invariant computations in details(10 marks) **(Ref.pg.no.32, Q.no.7)**

Apr/May -2011

Part-A

1. Define dead-code elimination. **(Ref.pg.no.8, Q.no.42)**
2. What is loop optimization? **(Ref.pg.no.7, Q.no.37)**

Part-B

1. Write in detail about function-preserving transformations (marks) **(Ref.pg.no.12, Q.no.1)**
2. Discuss briefly about peephole optimization**(Ref.pg.no17, Q.no.2)**
3. Write an algorithm to construct the natural loop of a back edge (6)**(Ref.pg.no.23, Q.no.4)**
4. Explain in detail about code- improving transformations**(Ref.pg.no.32, Q.no.7)**

Nov/Dec -2011

Part-A

1. List out the criteria for code improving transformations**(Ref.pg.no.2, Q.no.3)**
2. When does dangling reference occur? **(Ref.pg.no.7, Q.no.38)**

Part-B

1. Explain in detail optimization of basic blocks with example.(8) **(Ref.pg.no.19, Q.no.3)**
2. Write about data flow analysis of structural programs(8.) **(Ref.pg.no.28, Qno.6)**
3. Describe in detail the principal sources of optimization (16) **(Ref.pg.no.12, Q.no.1)**

Nov/Dec -2010

Part-A

1. What is dead (useless) code elimination? **(Ref.pg.no.8, Q.no.42)**
2.How would you map names to values? **(Ref.pg.no.7, Q.no.39)**

Part-B

1. Explain the access to non local names (16 marks) **(Ref.pg.no, Q.no.)**
2. Discuss in detail the source language issues to be taken in to account (16)
(Ref.pg.no, Qno.)

Apr/May -2010

Part-A

1. Give any two examples for strength reduction. **(Ref.pg.no.8, Q.no.44)**
2. What are the fields available in activation record? **(Ref.pg.no.8, Q.no.43)**
3.What is a pee hole optimization? **(Ref.pg.no.52, Q.no.10)**

Part-B

1. For the following program segments construct the flow graph and apply the possible optimizations

```
                Sum ;=0
                i:=1
                do
                sum:= prod +a[i}*b[i]
                i=i+1
     while i<=20; ( Ref.pg.no37, Q.no.8)
```

May/Jun -2009

Part-A

1. What is meant by copy restore? **(Ref.pg.no.10, Q.no.51)**

Part-B

1.Explain with an example how basic blocks are optimized (16 markes) **(Ref.pg.no.20, Q.no.3)**

2.Write short notes on : peep hole optimization (8 Marks) **(Ref.pg.no17, Q.no.2)**

Apr/May -2005

Part-A

1. How would you map names to values? **(Ref.pg.no.7, Q.no.39)**

Part-B

1. Explain the principle sources of optimization (16 marks)
(Ref.pg.no.12, Q.no.1)

Apr/May -2004

Part-A

1.What is code motion? **(Ref.pg.no.5, Q.no.33)**

Part-B

1. Explain the peephole optimization (8 marks) **(Ref.pg.no17, Q.no.2)**
2. Explain the principle sources of optimization (16 marks) **(Ref.pg.no.12, Q.no.1)**

Nov/Dec -2004

Part-A

1. What is a basic block? **(Ref.pg.no.3, Q.no.11)**

Part-B

1. Explain the principle sources of optimization (16 marks) **(Ref.pg.no.12, Q.no.1)**

CS6660- COMPILER DESIGN – UNIT II

LEXICAL ANALYSIS

Need and Role of Lexical Analyzer-Lexical Errors-Expressing Tokens by Regular Expressions-Converting Regular Expression to DFA- Minimization of DFA-Language for Specifying Lexical Analyzers-LEX-Design of Lexical Analyzer for a sample Language.

TWO MARKS

1. Write a regular expression for an identifier and whitespace. (Nov/Dec 2013)

Identifier

An identifier is defined as a letter followed by zero or more letters or digits.

The regular expression for an identifier is given as

R.E=letter (letter | digit)*

white space

spaces and tabs are generally ignored by the compiler, expect to serve as delimiters in most language and are not put out as tokens.

Delim → blank|tab|newline

Ws → delim+

2. Mention the various notational shorthands for representing regular expressions.

- One or more instances (+)
- Zero or one instance (?)
- Character classes ([abc] where a,b,c are alphabet symbols denotes the regular expressions a | b | c.)
- Non regular sets

3. What is the function of a hierarchical analysis?

Hierarchical analysis is one in which the tokens are grouped hierarchically into nested collections with collective meaning.Also termed as Parsing.

4. What does a semantic analysis do?

Semantic analysis is one in which certain checks are performed to ensure that components of a program fit together meaningfully.

Mainly performs type checking.

5. List the various error recovery strategies for a lexical analysis. [May/ June 2012]

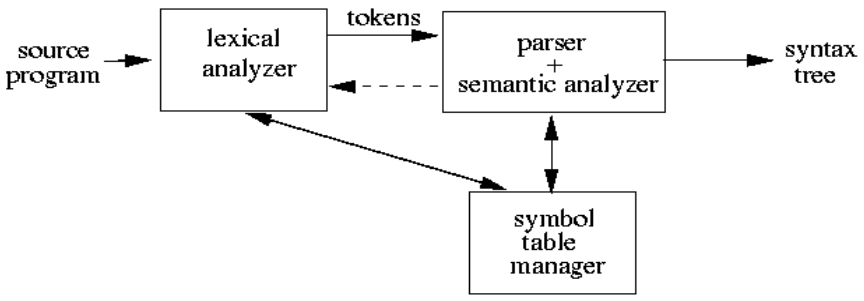
Possible error recovery actions are:

- Panic mode recovery
- Deleting an extraneous character
- Inserting a missing character
- Replacing an incorrect character by a correct character

- Transposing two adjacent characters

6. What are roles and tasks of a lexical analyzer? [Nov/Dec 2011]

<u>*Main Task*</u>: Take a token sequence from the scanner and verify that it is a syntactically correct program.



<u>*Secondary Tasks*</u>:

- Process declarations and set up symbol table information accordingly, in preparation for semantic analysis.
- Construct a syntax tree in preparation for intermediate code generation.

7. What is the purpose of scanner?

The lexical analyzer or ***scanner*** reads one character at a time from the source program and divides the sequence of characters into a logically cohesive element called ***tokens.***

8. Define sentence.

Collection of finite sequence of characters from the character set is a called a sentence.

9. Define terminal symbol and non-terminal symbol

Terminal symbol can be defined as a basic symbol in the language. For ex if, ; , + , switch, etc.,

Non-terminals are special symbols that denote set of strings.

10. What do you mean by LEX?

LEX is a software tool used for automatically writing Lexical Analyzers. LEX source is a specification of tokens in the form of a list of regular expressions together with an action for each regular expression.

Structure of LEX Program

declarations

%%

translation rules

%%

auxiliary functions

11. What do you mean by look ahead symbol?

The next input symbol for reduction/ derivation is called a look ahead symbol.

12. Define regular expression.

Regular expressions are the useful notations for defining the class of languages or regular sets. Regular expressions over an alphabet ∑ can be constructed from the following rules.

1) € is the regular expression denoting {€}, language containing only the empty string.
2) For each a in ∑ a is a regular expression denoting {a}, the language having the only string a.
3) If R and S are regular expressions denoting the languages L_R and L_S respectively, then
   i) (R) | (S) is a regular expression denoting L_R U L_S
   ii) (R) . (S) is a regular expression denoting $L_R . L_S$
   iii) $(R)^*$ is a regular expression denoting L_R^*

13. What are the tasks done by scanner other than token generation?

- Stripping out the comments
- Stripping out the white spaces
- Correlating error messages from the compiler with the source program
- Keep track of number of new line characters found, to provide a line number with an error message
- Making a copy of the source program with error messages marked in it

14. What are the reasons for separating the analysis phase to lexical analysis and parser? [May/June 2013] [May/June 2009]

- Simpler design
- Improving compiler efficiency
- Enhancing compiler portability.

15. Define attribute.

The information about the token is called an *attribute*. The attributes are available in the corresponding tables.

16. Define panic mode.

Panic mode error recovery is deleting the successive characters from the remaining input until the lexical analyzer can find a well formed token.

17. Write down the possible error recovery actions taken by lexical analyzer.

- Deleting an extraneous character.
- Inserting a missing character.
- Replacing an incorrect character by a correct character.
- Transposing two adjacent characters.

18. Define character class and language

Any finite set of symbols is called a *character class*.

Example: (i)[a-z] denotes the RE a|b|....|z

(ii)[abc] denotes the RE a|b|c

Language denotes any set of strings over some fixed alphabet.

Example: L={0^n1^n|n>0}

19. What are the operations supported by a string?

Prefix, suffix, sub string, proper prefix, proper suffix, proper sub string and subsequence.

20. What do you mean by subsequence?

Any string formed by deleting zero or more characters from any position (ie., not necessarily contiguous symbols) from S.

21. What is the purpose of the following notations ✳ , + and ?

These operators are unary postfix operators.

✳ Represents zero or more occurrences

+ Represents one or more occurrences

? Represents zero or one occurrences

22. Explain the notation useful for character class

The notation for representing a character class is [ ]. For example [a-z] denotes the regular expression a / b / c / d / e ……/z

23. What is the purpose of transition diagram?

Transition diagram describe the actions take place when analyzer is called by the parser to get the next token.

24. Define the purpose of gettoken( ) function.

This function identifies the identified word is a key word or not. If it is a keyword it returns the same word. Otherwise it returns the token.

25. Define the purpose of installid( ) function.

The installid function returns the attribute value of a token returned by gettoken function. If the lexeme is already available it will return a pointer. Otherwise it will install the lexeme, create a new pointer and return a pointer.

26. Define Look ahead operator.

The operator (normally ‘/’) used in the regular expressions to specify the lexical analyzers correctly is called look-ahead operator.

27. Define Finite automata.

The transition diagram constructed as a recognizer for a language is called as Finite Automata. The transition diagrams represent the regular expressions and accept any string x as input, answers ***yes*** if x is a sentence of the language generated by the regular expression or answers ***no***

28. Define NFA. May/June-14

A finite automata which has more than one transition from some state for one input symbol is termed as Non-determin‑‑tic finite automata.

29. Define DFA. May/June-14

A finite automata is said to be deterministic if

i) it has no transitions on input

ii) for each state s and input symbol ‘a’, there is at most one edge labeled ‘a' leaving s.

30. Write down the structure of LEX program.

A LEX source program consists Auxiliary Definitions and Translation Rules.

declarations

%%

translation rules

%%

auxiliary functions

Auxiliary definitions are the statements of the form

$D_1 = R_1$

$D_2 = R_2$

....

$D_n = R_n$ where each D_i is a distinct name and each R_i is a regular expression chosen from D_1 D_2 D_3 ….. D_{i-1} , ie the characters or previously defined names.

Translation rules are the statements of the form

P_1 { A_1 }

P_2 { A_2 }

....

P_n { A_n } where each P_i is a regular expression called pattern over the alphabet consisting of and the auxiliary definition names. Each A_i is a program fragment describing the action to be taken when token P_i is found.

31. Define concrete and abstract syntax with example. [May/June 2009]

- Concrete syntax is the surface level of a language (think, strings)
- Abstract syntax is the deep structure of a language (think, trees/terms)

- The abstract syntax defines the way the programs look like to the evaluator/compiler.
- Parsers convert concrete syntax into abstract syntax and have to deal with ambiguity
- It consists of a set of rules (productions) that define the way programs look like to the programmer

32. Give the transition diagram for an identifier. [Nov/Dec 2011]

$$\begin{tikzcd} {} \arrow[r] & q0 \arrow[r, "\text{letter}"] & \boxed{q1} \arrow[loop above, "\text{letter / digit}"] \end{tikzcd}$$

33. Why is buffering used in lexical analysis? What is the commonly used buffering methods? [May/June-14]

- Use a lexical analyzer generator, such as Lex compiler to produce the lexical analyzer from a regular expression based specification.
- Write the lexical analyzer in a conventional system-programming language, using the I/O facilities of that language to read the input.
- Write the lexical analyzer in assembly language and explicitly manage the reading of input
- the commonly used buffering methods are sentinel, buffer pairs.

34. Define LEXEME.(May/June 2014)

Collection or group of characters forming tokens is called Lexeme

35. Define tokens, patterns and lexemes. (Nov/Dec 2016)

Tokens

Sequence of characters that have a collective meaning.

A token is a string of characters, categorized according to the rules as a symbol (e.g., keywords, operators, identifiers, constants, literal strings, and punctuation symbols such as parentheses, commas, and semicolons.). The process of forming tokens from an input stream of characters is called **tokenization**.

LEXEME:

Collection or group of characters forming tokens is called Lexeme.

PATTERN:

A pattern is a description of the form that the lexemes of a token may take.

In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	**Const**	**const**

if	**if**	**if**
relation	**<,<=,=,<>,>,>=**	**< or <= or = or <> or >= or >**
id	**pi,count,D2**	**letter followed by letters and digits**
num	**3.1416,0,6.02E23**	**any numeric constant**
literal	**“core dumped”**	**any characters between “ and “ except”**

36. List the operations on languages. May/June2016

- **Union -** L U M ={s | s is in L or s is in M}
- **Concatenation** – LM ={st | s is in L and t is in M}
- **Kleene Closure –** L* (zero or more concatenations of L)
- **Positive Closure –** L+ (one or more concatenations of L)

37. Write a grammar for branching statements. May/June2016

A grammar for branching statements

$$\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \epsilon \\
expr & \rightarrow & term \textbf{ relop } term \\
 & | & term \\
term & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}$$

38. Divide lexical analysis phase.

1. Scanning Phase: This does simple tasks.
2. Lexical analysis phase: This does complex tasks.

39. What are the issues in lexical analysis?

i. Separation of phases
ii. Compiler efficiency is improved
iii. Compiler portability is enhanced
iv. The lexical analyzer has to recognize the longest possible string.
v. Skipping comments
vi. Symbol table interface.

40. Define tokens.

Token is a sequence of characters that can be treated as a single logical.

41. List 4 software tools that generate parser.

LEX,YACC, BISON, LEMON

41. Define Recognizer or Automata.

A recognizer for a language is a program that takes a string X, and answers “yes” if x is a sentence of that language, and “no” otherwise.

42. Algebraic properties of regular expressions.

R|S =S|R – Commutative

R|(S|T) = (R|S)T – Associative

(RS)T = R(ST) – Concatenation is associative

R** = R*-* is idempotent

43. What is Buffer Pair?

To reduce the amount of overhead required to process an input character, a buffer has been used. It is divided into 2 N-character halves where N is the number of characters on one disk block.

44. What is Sentinel?

The is a special character used to represent the end of the buffer which cannot be part of the source program.

: :E: : : M: *: eof	:C : : : : : eof

45. What are equivalent states?

Two states P ans Q are said to be equivlent if the transition from the state P on input on input X and the transitions from the state Q on input x are reaching the accepting state F.

46. Define String.

A language is a finite sequence of symbols drawn from that alphabet or letters.The strings are synonymously called as words.

The length of a string is denoted by |S|

The empty string can be denoted by ε.

The empty set od strings is denoted by Φ.

Example:banana

47. What is kleene’s closure?

The Kleene closure of a language L is wriiten as

L* = Zero or more occurence of language L.

SIXTEEN MARKS

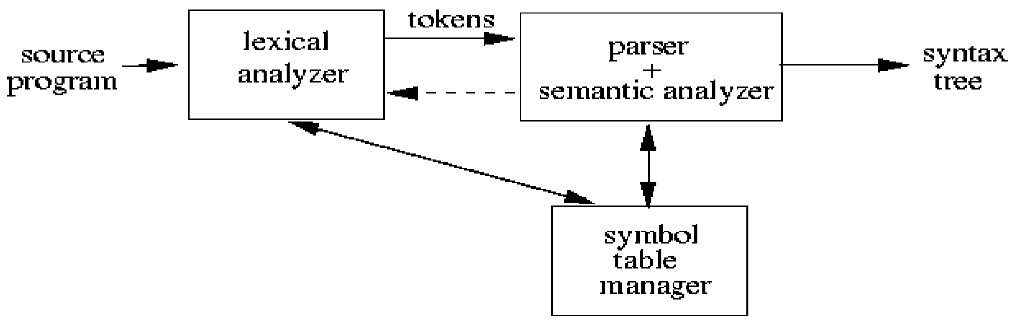
1. Explain in detail about the role of the lexical analyzer with the possible error recovery actions. [May/June 2013] [May/June 2012] [Apr/May 2011] [Nov/Dec 2007] [May/June-14] Or

Discuss the role of lexical analyzer in detail with necessary examples. (Nov/Dec 2016) (8)

LEXICAL ANALYSIS

THE ROLE OF THE LEXICAL ANALYZER

- The lexical analyzer is the first phase of compiler.
- Its main task is to read the input characters and produces output a sequence of tokens that the parser uses for syntax analysis.
- As in the figure, upon receiving a "get next token" command from the parser the lexical analyzer reads input characters until it can identify the next token.



- Since the lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface. One such task is **stripping out from the source program comments** and **white space in the form of blank, tab, and new line character**.
- Another is correlating error messages from the compiler with the source program.
- Sometimes lexical analyzers are divided into a cascade of two phases, the first called scanning" and the second "lexical analysis".

Issues in Lexical Analysis

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing.

1) Simpler design is the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases.

2) Compiler efficiency is improved.

3) Compiler portability is enhanced.

Tokens, Patterns and Lexemes.

Token:

A token is a string of characters, categorized according to the rules as a symbol (e.g., keywords, operators, identifiers, constants, literal strings, and punctuation symbols such as parentheses, commas, and semicolons.). The process of forming tokens from an input stream of characters is called **tokenization**.

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	Const	const
if	if	if
relation	<,<=,=,<>,>,>=	< or <= or = or <> or >= or >
id	pi,count,D2	letter followed by letters and digits
num	3.1416,0,6.02E23	any numeric constant
literal	“core dumped”	any characters between “ and “ except”

LEXEME:

Collection or group of characters forming tokens is called Lexeme.

PATTERN:

A pattern is a description of the form that the lexemes of a token may take.

In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

Attributes for Tokens

Some tokens have attributes that can be passed back to the parser. The lexical analyzer collects information about tokens into their associated attributes. The attributes influence the translation of tokens.

i) Constant : value of the constant

ii) Identifiers: pointer to the corresponding symbol table entry.

ERROR RECOVERY STRATEGIES IN LEXICAL ANALYSIS:

The following are the error-recovery actions in lexical analysis:

1) Deleting an extraneous character.

2) Inserting a missing character.

3) Replacing an incorrect character by a correct character.

4) Transforming two adjacent characters.

5) **Panic mode recovery**: Deletion of successive characters from the token until error is resolved.

2. **Differentiate between lexeme, token and pattern.(6) May/June2016**

   Tokens

   A token is a string of characters, categorized according to the rules as a symbol (e.g., keywords, operators, identifiers, constants, literal strings, and punctuation symbols such as parentheses, commas, and semicolons.). The process of forming tokens from an input stream of characters is called **tokenization**.

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	Const	const
if	if	if
relation	<,<=,=,<>,>,>=	< or <= or = or <> or >= or >
id	pi,count,D2	letter followed by letters and digits
num	3.1416,0,6.02E23	any numeric constant
literal	“core dumped”	any characters between “ and “ except”

LEXEME:

Collection or group of characters forming tokens is called Lexeme.

PATTERN:

A pattern is a description of the form that the lexemes of a token may take.

In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

Attributes for Tokens

Some tokens have attributes that can be passed back to the parser. The lexical analyzer collects information about tokens into their associated attributes. The attributes influence the translation of tokens.

i) Constant : value of the constant

ii) Identifiers: pointer to the corresponding symbol table entry.

3. **What are the issues in lexical analysis?(4) May/June2016**

Issues in Lexical Analysis

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing.

1) Simpler design is the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases.

2) Compiler efficiency is improved.

3) Compiler portability is enhanced.

4. **Explain breifly about specification of token.**

(or) Explain briefly about Expressing Tokens by Regular Expressions

SPECIFICATION OF TOKENS

There are 3 specifications of tokens:

1) Strings

2) Language

3) Regular expression

Strings and Languages

An **alphabet** or character class is a finite set of symbols.

A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.

A **language** is any countable set of strings over some fixed alphabet.

In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s, usually written |s|, is the number of occurrences of symbols in s.

For example, banana is a string of length six. The empty string, denoted ε, is the string of length zero.

Operations on strings

The following string-related terms are commonly used:

1. A **prefix** of string s is any string obtained by removing zero or more symbols from the end of string s.

For example, ban is a prefix of banana.

2. A **suffix** of string s is any string obtained by removing zero or more symbols from the beginning of s.

For example, nana is a suffix of banana.

3. A **substring** of s is obtained by deleting any prefix and any suffix from s.

For example, nan is a substring of banana.

4. The **proper prefixes, suffixes, and substrings** of a string s are those prefixes, suffixes, and substrings, respectively of s that are not ε or not equal to s itself.

5. A subsequence of s is any string formed by deleting zero or more not necessarily consecutive positions of s.

For example, baan is a subsequence of banana.

Operations on languages:

The following are the operations that can be applied to languages:

1.Union

2.Concatenation

3.Kleene closure

4.Positive closure

The following example shows the operations on strings:

Let L={0,1} and S={a,b,c}

1. Union : L U S={0,1,a,b,c}

2. Concatenation : L.S={0a,1a,0b,1b,0c,1c}

3. Kleene closure : L*={ ε,0,1,00….}

4. Positive closure : L+={0,1,00….}

Regular Expressions

Each regular expression r denotes a language L(r).

Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote:

1. ε is a regular expression, and L(ε) is { ε }, that is, the language whose sole member is the empty string.
2. If 'a' is a symbol in Σ, then 'a' is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with 'a' in its one position.
3. Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then,

a) (r)|(s) is a regular expression denoting the language L(r) U L(s).

b) (r)(s) is a regular expression denoting the language L(r)L(s).

c) (r)* is a regular expression denoting (L(r))*.

d) (r) is a regular expression denoting L(r).

4. The unary operator * has highest precedence and is left associative.
5. Concatenation has second highest precedence and is left associative.
6. | has lowest precedence and is left associative.

Regular set

A language that can be defined by a regular expression is called a regular set.

If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s.

There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms.

For instance, r|s = s|r is commutative; r|(s|t)=(r|s)|t is associative.

Regular Definitions

Giving names to regular expressions is referred to as a Regular definition. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

dl → r 1

d2 → r2

………

dn → rn

1. Each di is a distinct name.
2. Each ri is a regular expression over the alphabet Σ U {dl, d2,. . . , di-l}.

Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set:

letter → A | B | …. | Z | a | b | …. | z |

digit → 0 | 1 | …. | 9

id → letter (letter | digit) *

Shorthands

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

1. *One or more instances (+)*:

- The unary postfix operator + means " one or more instances of" .

- If r is a regular expression that denotes the language L(r), then (r)+ is a regular expression that denotes the language (L (r))+

- Thus the regular expression a+ denotes the set of all strings of one or more a's.

- The operator + has the same precedence and associativity as the operator *.

2. *Zero or one instance (?)*:

- The unary postfix operator ? means "zero or one instance of".

- The notation r? is a shorthand for r | ε.

- If 'r' is a regular expression, then (r)? is a regular expression that denotes the language L(r) U { ε }.

3. *Character Classes*:

- The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c.

- Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z.

- We can describe identifiers as being strings generated by the regular expression,

[A–Za–z][A–Za–z0–9]*

Non-regular Set

A language which cannot be described by any regular expression is a non-regular set.

Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

5. <u>How do you recognize a token?</u> [May/June 2013] [May/June 2012] [Apr/May 2011]

Or Discuss how finite automata is used to represent tokens and perform lexical analysis with examples. (Nov/Dec 2016)

RECOGNITION OF TOKENS

Consider the following grammar fragment:

$$\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \epsilon \\
expr & \rightarrow & term \textbf{ relop } term \\
 & | & term \\
term & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}$$

A grammar for branching statements

Example:

For relop, we use the comparison operators of languages like Pascal or SQL, where = is "equals" and <> is "not equals," because it presents an interesting structure of lexemes. The terminals of the grammar, which are if, then, else, relop , id, and number, are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using regular definitions, as in Fig. 3.11.

The patterns for id and number are similar

$$\begin{array}{rcl}
\textit{digit} & \rightarrow & \texttt{[0-9]} \\
\textit{digits} & \rightarrow & \textit{digit}^{+} \\
\textit{number} & \rightarrow & \textit{digits}\ (\,.\ \textit{digits})?\ (\ \texttt{E}\ [+-]?\ \textit{digits}\ )? \\
\textit{letter} & \rightarrow & \texttt{[A-Za-z]} \\
\textit{id} & \rightarrow & \textit{letter}\ (\ \textit{letter} \mid \textit{digit}\ )^{*} \\
\textit{if} & \rightarrow & \texttt{if} \\
\textit{then} & \rightarrow & \texttt{then} \\
\textit{else} & \rightarrow & \texttt{else} \\
\textit{relop} & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}$$

Patterns for tokens

For this language, the lexical analyzer will recognize the keywords if, then, and else, as well as lexemes that match the patterns for relop, id, and number. To simplify matters, we make the common assumption that keywords are also reserved words : that is , they are not identifiers, even though their lexemes match the pattern for identifiers.

In addition, we assign the lexical analyzer the job of stripping out whitespace, by recognizing the "token" ws defined by:

ws -t (blank | tab | newline)+

Here, blank, tab, and newline are abstract symbols that we use to express the ASCII characters of the same names. Token ws is different from the other tokens in that , when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace. It is the following token that gets returned to the parser.

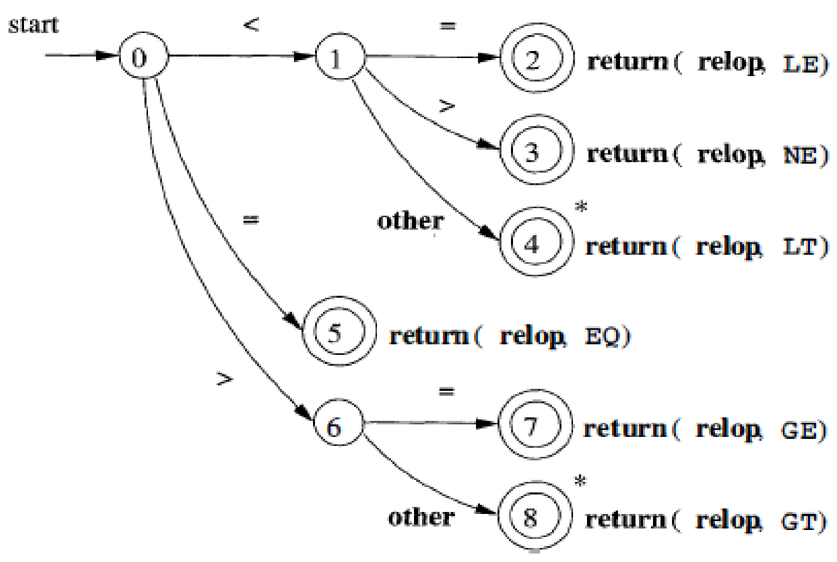
LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	**if**	–
then	**then**	–
else	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Tokens, their patterns, and attribute values

1. **Transition diagrams**

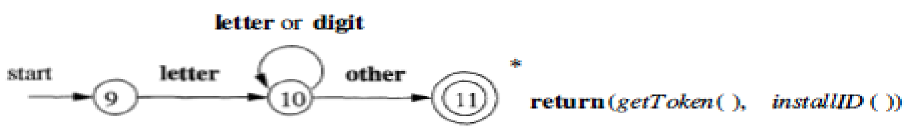
It is a diagrammatic representation to depict the action that will take place when a lexical analyzer is called by the parser to get the next token. It is used to keep track of information about the characters that are seen as the forward pointer scans the input.

Transition diagram for relop

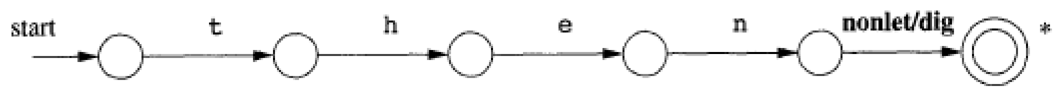


2. **Recognition of reserved words and Identifiers**

A transition diagram for id's and keywords



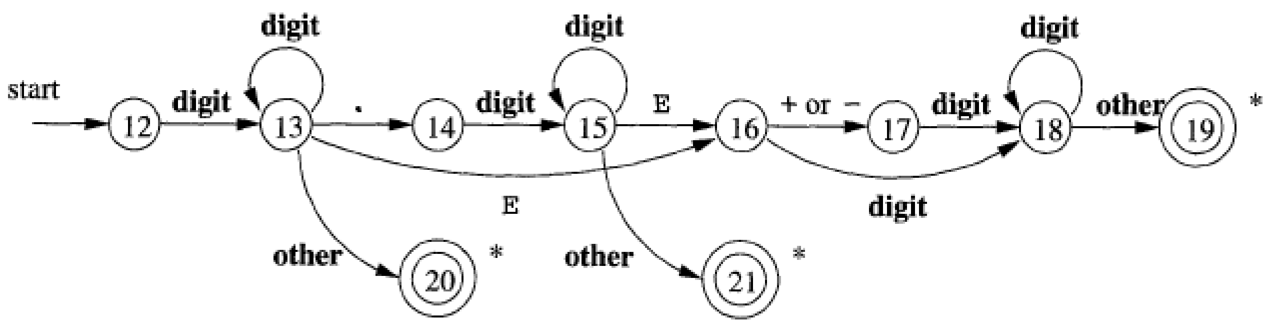
Hypothetical transition diagram for the keyword then



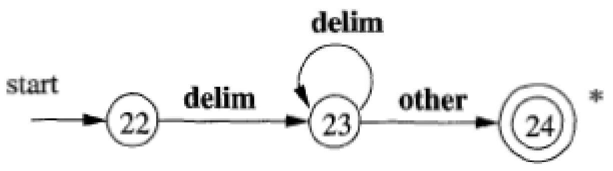
3. Completion of the Running Example

Beginning in state 12, if we see a digit, we go to state 13. In that state, we can read any number of additional digits. However, if we see anything but a digit or a dot, we have seen a number in the form of an integer; 123 is an example. That case is handled by entering state 20, where we return token number and a pointer to a table of constants where the found lexeme is entered. These mechanics are not shown on the diagram but are analogous to the way we handled identifiers

Transition diagram for unsigned numbers in pascal



A transition diagram for whitespace



4. Architecture of a Transition-Diagram-Based Lexical Analyzer

There are several ways that a collection of transition diagrams can be used to build a lexical analyzer. Regardless of the overall strategy, each state is represented by a piece of code. We may imagine a variable state holding the number of the current state for a transition diagram. A switch based on the value of state takes us to code for each of the possible states, where we find the action of that state. Often, the code for a state is itself a switch statement or multi way branch that determines the next state by reading and examining
the next input character.

6. **Write notes on regular expressions. (6) May/June2016**

Regular Expressions

Each regular expression r denotes a language L(r).

Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote:

1. ε is a regular expression, and L(ε) is { ε }, that is, the language whose sole member is the empty string.

2. If ‘a’ is a symbol in Σ, then ‘a’ is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with ‘a’ in its one position.

3. Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then,

a) (r)|(s) is a regular expression denoting the language L(r) U L(s).

b) (r)(s) is a regular expression denoting the language L(r)L(s).

c) (r)* is a regular expression denoting (L(r))*.

d) (r) is a regular expression denoting L(r).

4. The unary operator * has highest precedence and is left associative.

5. Concatenation has second highest precedence and is left associative.

6. | has lowest precedence and is left associative.

Regular set

A language that can be defined by a regular expression is called a regular set.

If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s.

There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms.

For instance, r|s = s|r is commutative; r|(s|t)=(r|s)|t is associative.

Regular Definitions

Giving names to regular expressions is referred to as a Regular definition. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

dl → r 1

d2 → r2

………

dn → rn

1. Each di is a distinct name.

2. Each ri is a regular expression over the alphabet Σ U {dl, d2,. . . , di-l}.

Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set:

letter → A | B | …. | Z | a | b | …. | z |

digit → 0 | 1 | …. | 9

id → letter (letter | digit) *

Shorthands

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

1. *One or more instances (+)*:

- The unary postfix operator + means " one or more instances of" .
- If r is a regular expression that denotes the language L(r), then (r)+ is a regular expression that denotes the language (L (r))+
- Thus the regular expression a+ denotes the set of all strings of one or more a's.
- The operator + has the same precedence and associativity as the operator *.

2. *Zero or one instance (?)*:

- The unary postfix operator ? means "zero or one instance of".
- The notation r? is a shorthand for r | ε.
- If 'r' is a regular expression, then (r)? is a regular expression that denotes the language L(r) U { ε }.

3. *Character Classes*:

- The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c.
- Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z.
- We can describe identifiers as being strings generated by the regular expression, [A–Za–z][A–Za–z0–9]*

Non-regular Set

A language which cannot be described by any regular expression is a non-regular set.

Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

A regular expression is a string that describes the whole set of strings according to certain syntax rules. Regular expressions ***serve as the input language for many systems that process strings.*** These expressions are used by many text editors and utilities to search bodies of text for certain patterns etc.

Regular expressions denote languages. **For Example -** The regular expression **01*+10*** denotes the language consisting of all strings that are either a single 0 followed by any number of 1's or a single 1 followed by any number of 0's.

The three operators involved in the Regular Expression are,

(i) The **Union** of two languages L and M, denoted by LUM, is the set of strings that are either in L or M, or both.

Example: If L = {001, 10, 111} are M = {Є, 001}, then LUM = {Є, 10, 001, 111}

(ii) The **Concatenation** of languages L and M is the set of strings that can be formed by taking any string in L and concatenating it with any string in M.

Example: If L= {001, 10, 111} and M = {Є, 001} then
L.M (Or) LM = {001, 10, 111, 001001, 10001, 111001}

(iii)The **closure** of a language L is denoted L* and represents the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions (i.e., the same string may be selected more than once) and concatenating all of them.

Example: If L = {0,1}, then L* is all strings of 0's and 1's
If L = {0,11}, then L* consists of those strings of 0's and 1's such that the 1's come in pairs, e.g., 011, 11110, and Є, but not 01011 or 101.

The regular expression involving the **Closure** operation and its types are,

- Kleene or star closure
- Positive closure

7. Explain Briefly about Finite Automata

Finite Automata

A recognizer for alanguage is a program that takes a sinput a string x and answers"yes" if x is a sentence of the lahnguage and "no" otherwise.

The generaliszed transition for regular expression is called finite automation. It is a labeled directed graph. Hence the nodes are the states and the labeled edges are transitions.

The automata can be grouped into two as given below:

1. Non-deterministic Finite Automata(NFA)
2. Deterministic Finite Automata(DFA)

"Non-determisitic" means that more than one transition outof a state may be possible on the same input symbol.

Both finite automata are capable of recognizing precisely the regular sets.

- Finite Automata are used as a model for:
  - Software for designing digital circuits
  - Lexical analyzer of a compiler
  - Searching for keywords in a file or on the web.

– Software for verifying finite state systems, such as communication protocols.

1. Non-Deterministic Finite Auotomata

A non-deterministic finite automaton NFA) is a mathematical model tha consists of

1. A set of state s.
2. A set of input symbols Σ (he input symbol alphabet).
3. A transition function move that maps state-symbol pairs to sets of states.
4. A state so that is distinguished as the start (or initial) state.
5. A set of states F distinguished as acecepting(or final states.

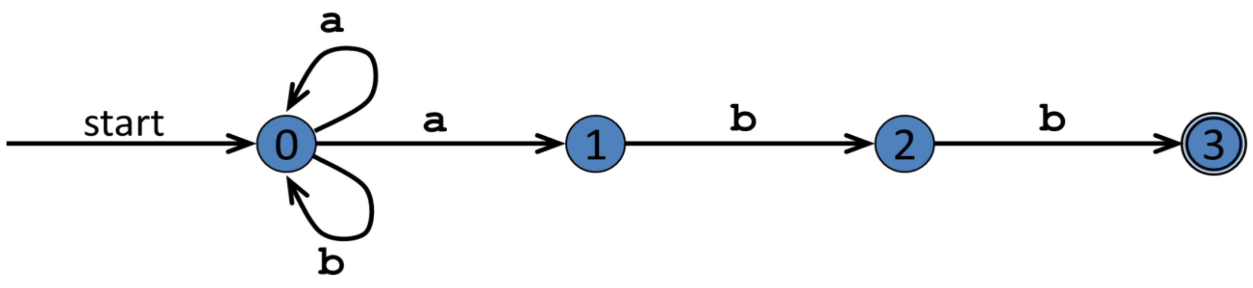
An NFA can be represented diagrammatically by a labeled directed graph called a transition graph, in which the nodes are the states and the labeled edges represent the transition function.

This graph looks like a transition diagram but the same character can labeled two or more transitions out of one state, and edges can be labeled by the special symbol ε as well as input symbols.

(eg) NFA that recognizes the laguage (a/b)*abb

1.1 Transition Graph

- **An NFA can be diagrammatically represented by a labeled directed graph called a *transition graph***



$S = \{0,1,2,3\}$

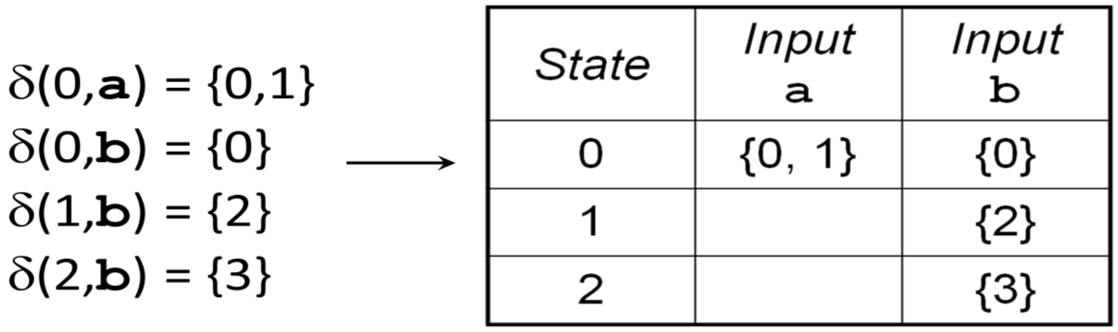
$\Sigma = \{\mathbf{a},\mathbf{b}\}$

$s_0 = 0$

$F = \{3\}$

1.2 Transition Table

- The mapping δ of an NFA can be represented in a *transition table*



$$\delta(0,\mathbf{a}) = \{0,1\}$$
$$\delta(0,\mathbf{b}) = \{0\}$$
$$\delta(1,\mathbf{b}) = \{2\}$$
$$\delta(2,\mathbf{b}) = \{3\}$$

State	*Input* a	*Input* b
0	{0, 1}	{0}
1		{2}
2		{3}

Transition Table for the NFA

1.3 The Language Defined by an NFA

- An NFA *accepts* an input string *x* if and only if there is some path with edges labeled with symbols from *x* in sequence from the start state to some accepting state in the transition graph
- A state transition from one state to another on the path is called a *move*
- The *language defined by* an NFA is the set of input strings it accepts, such as (**a**|**b**)***abb** for the example NFA

Advantage of NFA

It provides fast access to the transitions of a given character.

Dis advantage of NFA

Disadavantage of using transition table is that it can take up a lot of space when tha input alphabet is large and most transitions are to the empty set.

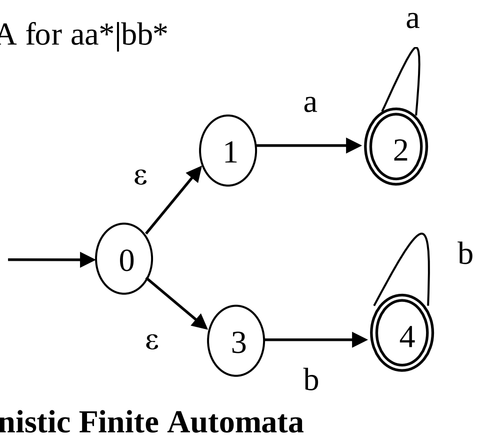
Accepting the string aabb

A path can be represented by sequence od state transitions called moves. The following diagram shows the moves made in accepting the input string aabb.

$$\begin{tikzcd} 0 \arrow[r, "a"] & 0 \arrow[r, "a"] & 1 \arrow[r, "b"] & 2 \arrow[r, "b"] & 3 \end{tikzcd}$$

Example:

Design NFA for aa*|bb*

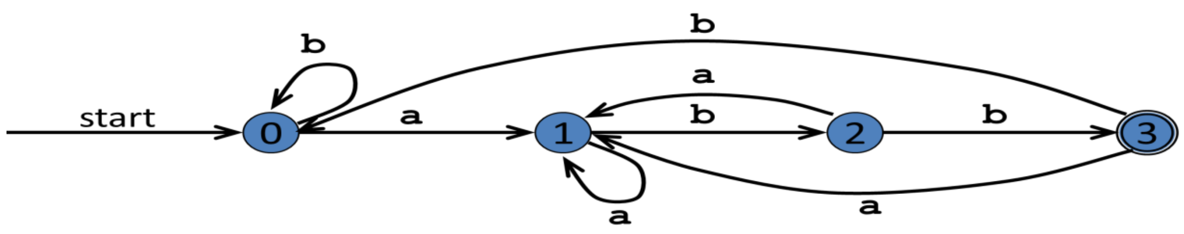


2. Deterministic Finite Automata

- A *deterministic finite automaton* is a special case of an NFA in which:
  - No state has an ε-transition
  - For each state *s* and input symbol *a* there is at most one edge labeled *a* leaving *s*
- Each entry in the transition table is a single state
  - At most one path exists to accept a string
  - Simulation algorithm is simple

Example DFA

A DFA that accepts $(a|b)^*abb$



8. Explain briefly about converting from a Regular Expressions to An NFA

This is one way to convert a regular expression into a NFA.

- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method.
- It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
- To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA.

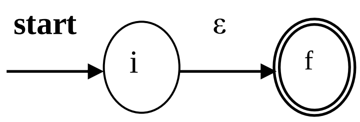
Algorithm: (Thompson's construction) An NFA from a regular expression.

Input: A regular expression r over an alphabet Σ.

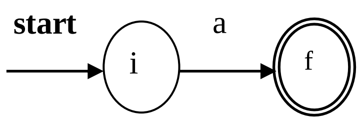
Output: An NFA N accepting L(r).

Method:

1. For ε, construct the NFA

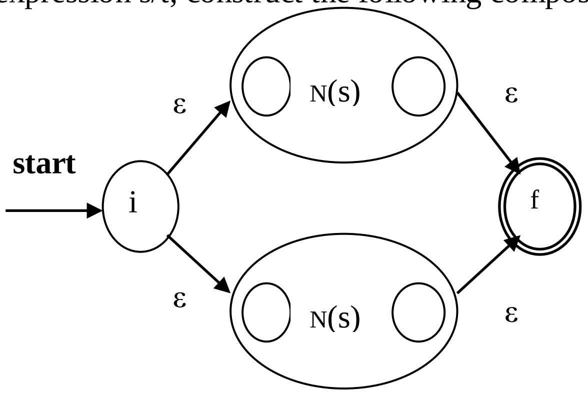


2. For a in Σ, construct the NFA

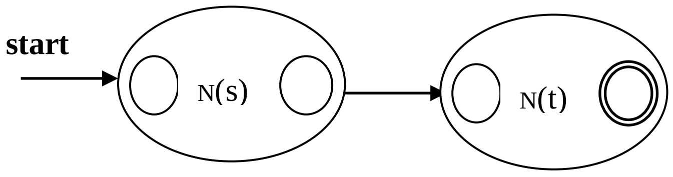


3. Suppose N(s) and N(t) are NFA's for regular expressions s and t.

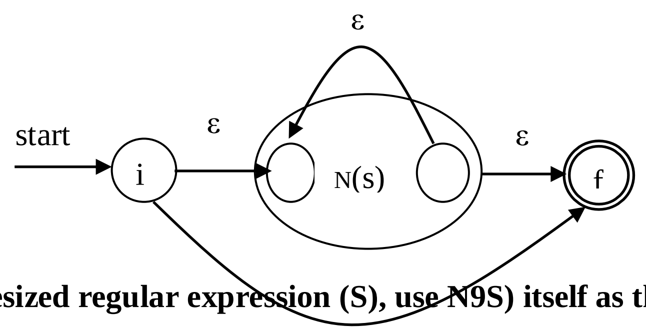
a) For the regular expression s/t, construct the following composite NFA N(s/t):



b) For the regular expression s.t,construct the composite NFA N(s.t)



c) For he regular expression S*,construct the composite NFA N(S*):



d) For the parenthesized regular expression (S), use N9S) itself as the NFA.

The construction produces an NFA) with the following proerties.

1. N(r) has the most twice as many states as the number of symbols and operators in r.
2. N(r) has exactly one start and one accepting state. The accepting state has no outgoing transitions.
3. Each state of N(r) has either one outgoing transition on a symbol in ε or at most two outgoing ε-transitions.

9. Explain briefly about converting of an NFA into a DFA.

Conversion of an NFA into a DFA

- The *subset construction algorithm* converts an NFA into a DFA using:

$\varepsilon\text{-}closure(s) = \{s\} \cup \{t \mid s \rightarrow_{\varepsilon} \ldots \rightarrow_{\varepsilon} t\}$

$\varepsilon\text{-}closure(T) = \cup_{s \in T}\, \varepsilon\text{-}closure(s)$

$move(T,a) = \{t \mid s \rightarrow_{a} t \text{ and } s \in T\}$

- The algorithm produces:

  Dstates is the set of states of the new DFA consisting of sets of states of the NFA

  Dtran is the transition table of the new DFA

Algorithm: (subset sonstruction)

```
Initially, ε-closure(s) is the only state in Dstates and it is unmarked;
While there ia an unmarked state T in Dstates do begin
        mark T;
        For each input symbol a do begin
                U:= ε-closure(move(T,a));
                If U is not in Dstates then
                        add U as an unmarked state to Dstate;
                Dtran[T,a]:= U
        End
End
```

The subset sonstruction

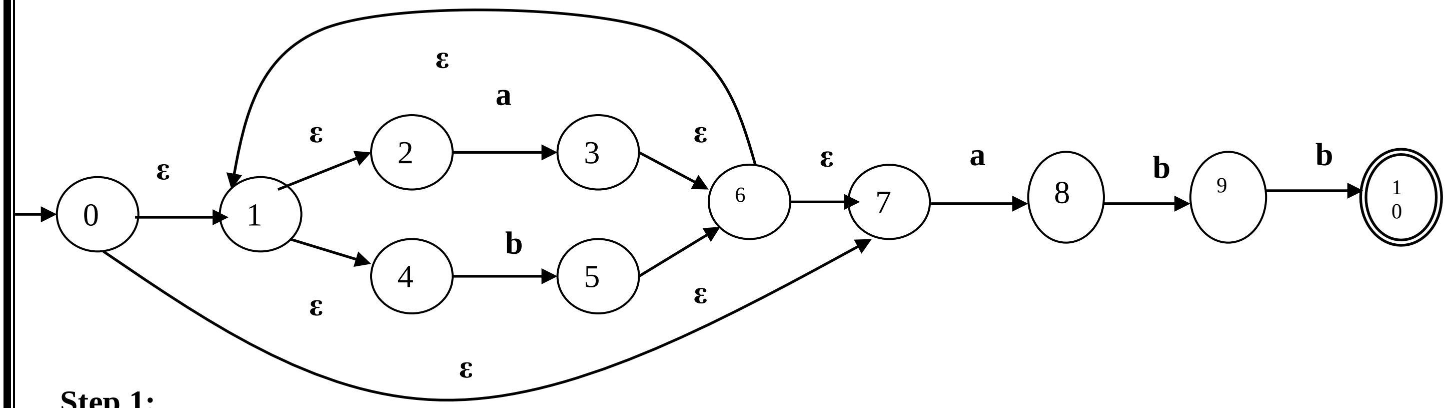
Computation of ε-closure:

```
push all states in T onto stack;
initialize ε-closure(T) to T;
while stack is not emptydo begin
        pop t, the top element,off of stack;
        for each state u with an edge from t to u labeled ε do
         if u is not in ε-closure(T) do begin
                add u to ε-closure(T);
                push u onto stack
        end
end
```

<u>**Subset Construction Example 1**</u>

Solution:

The NFA for the RE (a+b) * abb is shown below,



Step 1:

ε – closure (0) ={ 0,1,2,4,7} → A

Now process the state A with ‘a’ and ‘b’.

Step 2:

Move [A,a] = {3,8} (Similar to δ(A,a))

ε – closure (Move [A,a]) = { 1,2,3,4,6,7,8} → B

Move [A,b] = {5}

ε – closure (Move[A,b]) = { 1,2,4,5,6,7} → C

Step 3:

Move [B,a] = {3,8}

ε – closure (Move [B,a]) = {1,2,3,4,6,7,8} → B

Move [B,b] ={5}

ε – closure (Move [B,b]) = {1,2,4,5,6,7,9} → D

Step 4:

Move[C,a] = {3,8}

ε – closure (Move[C,a]) = {1,2,3,4,6,7,8} → B

Move [C,b] = {5}

ε – closure (Move [C,b]) = {1,2,4,5,6,7} → C

Step 5:

Move[D,a] = {3,8}

ε – closure (Move[D,a]) = {1,2,3,4,6,7,} → B

Move[D,b] ={5,10}

ε – closure (Move[D,b]) = {1,2,4,5,6,7,10} → E

Step 6:

Move [E,a] = {3,8}

ε – closure (Move [E,a]) = {1,2,3,4,6,7,8} → B

Move [E,b] = {5}

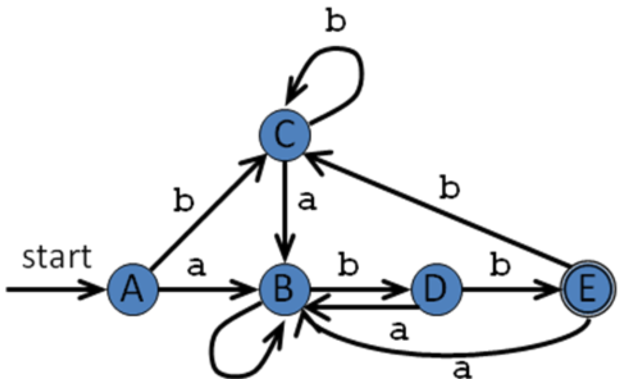
ε – closure (Move [E,b]) = { 1,2,4,5,6,7} → C

Dstates
A = {0,1,2,4,7}
B = {1,2,3,4,6,7,8}
C = {1,2,4,5,6,7}
D = {1,2,4,5,6,7,9}
E = {1,2,4,5,6,7,10}

The transition table for DFA is given below,

	a	**b**
A	**B**	**C**
B	**B**	**D**
C	**B**	**C**
D	**B**	**E**
***E**	**B**	**C**

DFA Daigram



10. Explain about converting From Regular Expression to DFA Directly

From Regular Expression to DFA Directly

- The *"important states"* of an NFA are those without an ε-transition, that is if *move*({*s*},*a*) ≠ ∅ for some *a* then *s* is an important state
- The subset construction algorithm uses only the important states when it determines ε-*closure*(*move*(*T*,*a*))

From Regular Expression to DFA Directly (Algorithm)

ALGORITHM : Construction of a DFA from a regular expression r.

INPUT : A regular expression r.

OUTPUT: A DFA D that recognizes L (r) .

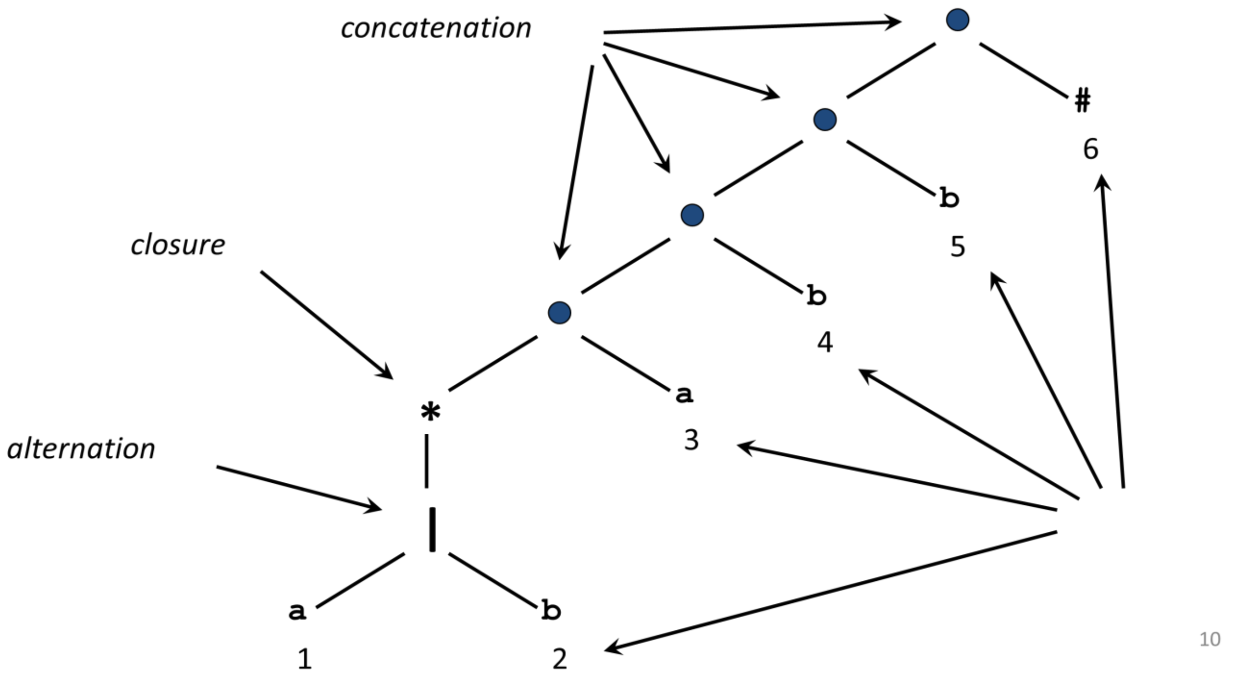
METHOD:

1 . Construct a syntax tree T from the augmented regular expression (r) #.

2. Compute nullable,firstpos, lastpos, and followpos for T

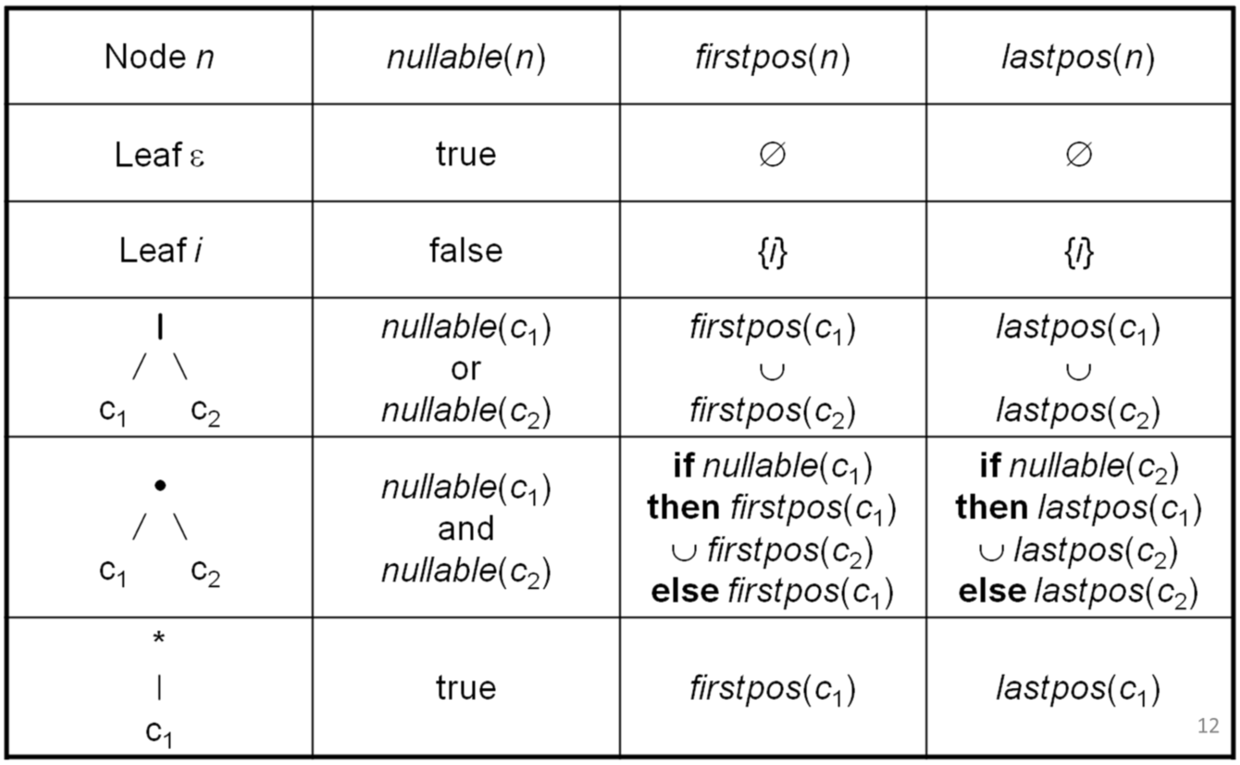
3. Construct Dstates, the set of states of DFA D , and Dtran, the transition function for D , by the procedure of Fig. 3.62. The states of D are sets of positions in T. Initially, each state is "unmarked," and a state becomes "marked" just before we consider its out-transitions. The start state ofD is jirstpos(no) , where node no is the root of T. The accepting states are those containing the position for the endmarker symbol #.

From Regular Expression to DFA Directly: Syntax Tree of (a|b)*abb#



From Regular Expression to DFA Directly: Annotating the Tree

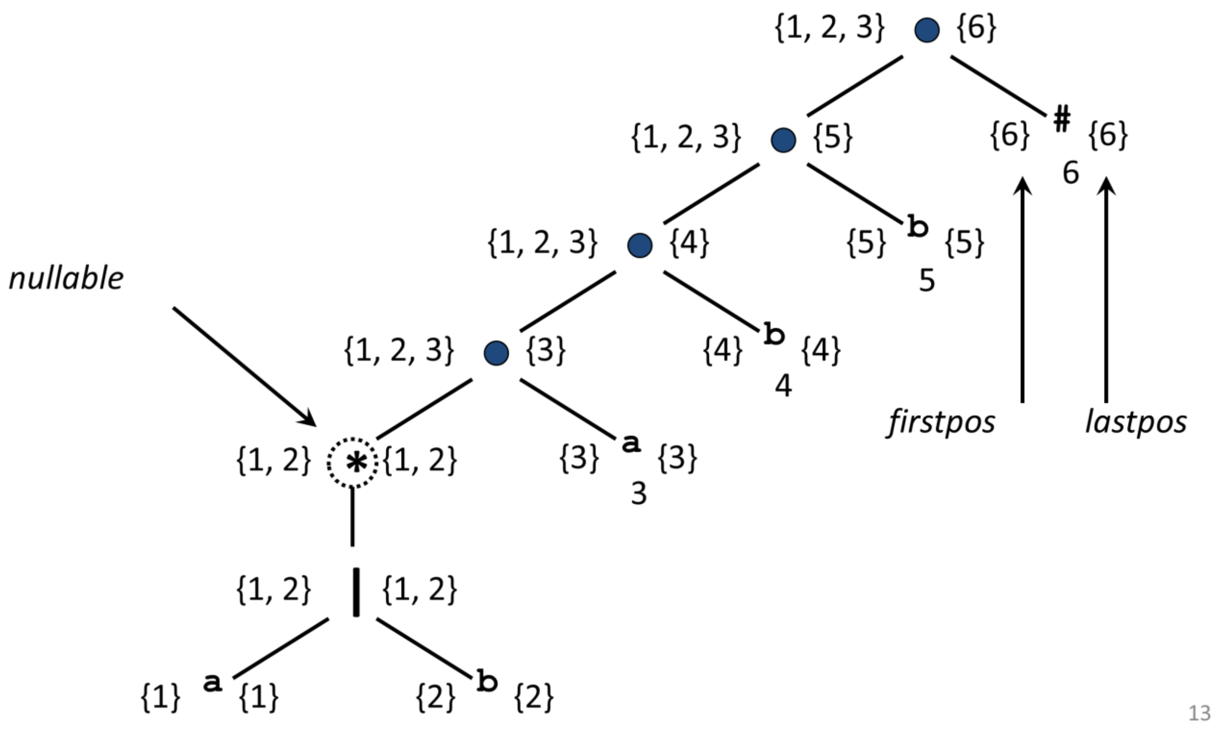
- *nullable*(*n*): the sub tree at node *n* generates languages including the empty string
- *firstpos*(*n*): set of positions that can match the first symbol of a string generated by the sub tree at node *n*
- *lastpos*(*n*): the set of positions that can match the last symbol of a string generated be the sub tree at node *n*
- *followpos*(*i*): the set of positions that can follow position *i* in the tree



Node *n*	*nullable*(*n*)	*firstpos*(*n*)	*lastpos*(*n*)
Leaf ε	true	∅	∅
Leaf *i*	false	{*i*}	{*i*}
$\mid$ / \ c_1 c_2	*nullable*(c_1) or *nullable*(c_2)	*firstpos*(c_1) ∪ *firstpos*(c_2)	*lastpos*(c_1) ∪ *lastpos*(c_2)
• / \ c_1 c_2	*nullable*(c_1) and *nullable*(c_2)	**if** *nullable*(c_1) **then** *firstpos*(c_1) ∪ *firstpos*(c_2) **else** *firstpos*(c_1)	**if** *nullable*(c_2) **then** *lastpos*(c_1) ∪ *lastpos*(c_2) **else** *lastpos*(c_2)
* | c_1	true	*firstpos*(c_1)	*lastpos*(c_1)

12

From Regular Expression to DFA Directly: Syntax Tree of (a|b)*abb#



From Regular Expression to DFA Directly: *followpos*

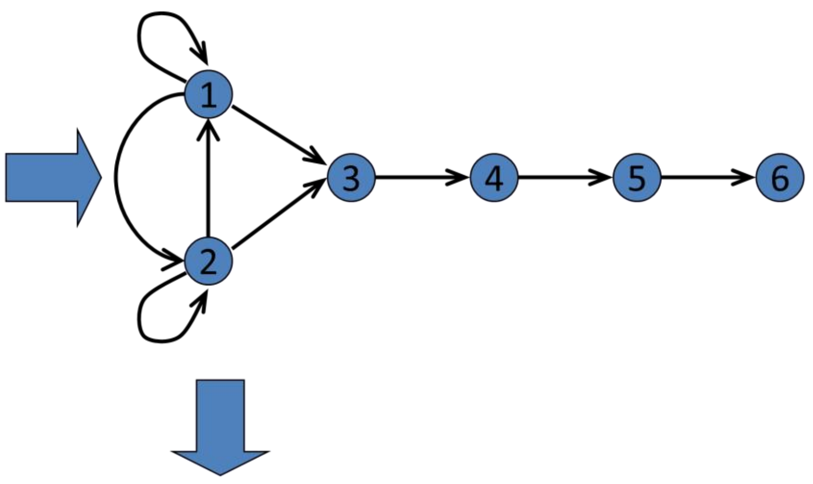
```
for each node $n$ in the tree do
    if $n$ is a cat-node with left child $c_1$ and right child $c_2$ then
        for each $i$ in $lastpos(c_1)$ do
            $followpos(i) := followpos(i) \cup firstpos(c_2)$
        end do
    else if $n$ is a star-node
        for each $i$ in $lastpos(n)$ do
            $followpos(i) := followpos(i) \cup firstpos(n)$
        end do
    end if
end do
```

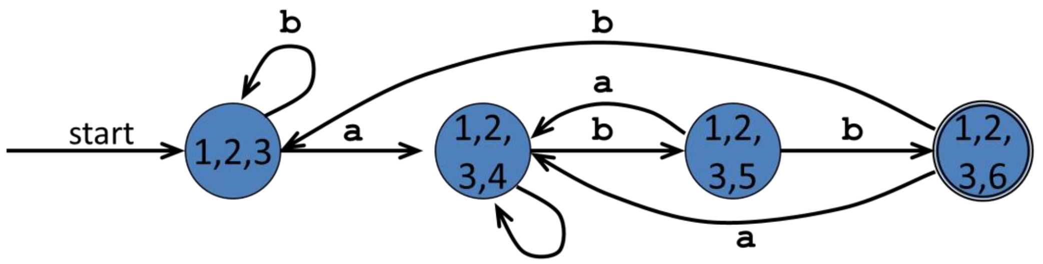
From Regular Expression to DFA Directly: Algorithm

```
$s_0 := firstpos(root)$ where $root$ is the root of the syntax tree
$Dstates := \{s_0\}$ and is unmarked
while there is an unmarked state $T$ in $Dstates$ do
    mark $T$
    for each input symbol $a \in \Sigma$ do
        let $U$ be the set of positions that are in $followpos(p)$
            for some position $p$ in $T$,
            such that the symbol at position $p$ is $a$
        if $U$ is not empty and not in $Dstates$ then
            add $U$ as an unmarked state to $Dstates$
        end if
        $Dtran[T,a] := U$
    end do
end do
```

From Regular Expression to DFA Directly: Example

Node	*followpos*
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	-





11. Write an algorithm for minimizing the number od states of a DFA.(8) Nov/Dec 2016

Minimization the number of states of a DFA

We can construct a minimum state DFA by reducing the number of states in a given DFA to the bare minimum without attacking the language that is being recognized.

The following steps are used to minimize the number of states of a DFA.

Partition the set of states into two groups.

- G_1 :Set of accepting states
- G_1 :Set of no-accepting states

For each new group G

- Partition G into subgroups such that S_1 and S_2 are in the same group iff for all input sympols a, states s and S_2 have transitions to state in the same group.

Start state of the minimized DFA is the group containing the start state of the original DFA.

Accepting states of the minimized DFA are the groups containing the accepting states of he original DFA.

Algorithm : Minimizing the number of states of a DFA.

Input: A DFA M with set of states s,set of input ε,transitions defined for all states and inputs start state S_0 and set of accepting states F.

Output: A DFA M^1 accepting the same languages as M and having as few states as possible.

Method:

1. Construct an initial partition Π of the se of states with two groups: the accepting F and nonaccepting states S-F.
2. Apply the procedure to construct a new partition Π_{new} ·

```
initially, let $\Pi_{new}$ = Π;
for ( each group G of Π ) {
        partition G into subgroups such that two states s and t
                are in the same subgroup if and only if for all
                input symbols a, states s and t have transitions on a
                to states in the same group of Π;
        /* at worst, a state will be in a subgroup by itself * /
        replace G in $\Pi_{new}$ by the set of all subgroups formed;
}
```

3. If Π_{new} = Π, let Π_{final} = Π and continue with step (4) . Otherwise, repeat step (2) with Π_{new} in place of Π.

4. Choose one state in each group of Π_{final} as the rep resentative for that group. The representatives will be the states of the minimum-state DFA M'.

Let S be a representative state, and suppose on input a there is a transition of M from s to t. Let r be the representative of t's group (r may be t). Then M' has a transition from s to r on a. Let the start state S_0 of M, and let the accepting states of M' be the reprenstatives that are in F.

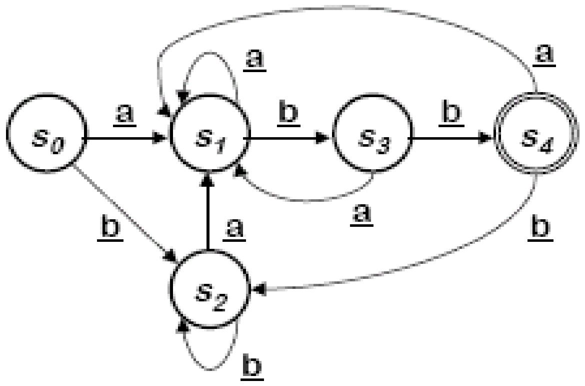
Note that each group of Π_{final} either consists only of staes in F or has no states in F.

8.If M' has a dead state, that is, a state d that is not accepting and that has transitions to itself on all input symbols, then remove d from M'. Also remove any states not reachable from the start state. Any trensitions to d from other states become undefined.

Procedure for Π_{new} construction

```
for ( each group G of Π ) {
        partition G into subgroups such that two states s and t
                are in the same subgroup if and only if for all
                input symbols a, states s and t have transitions on a
                to states in the same group of Π;
        /* at worst, a state will be in a subgroup by itself * /
        replace G in $\Pi_{new}$ by the set of all subgroups formed;
end
```

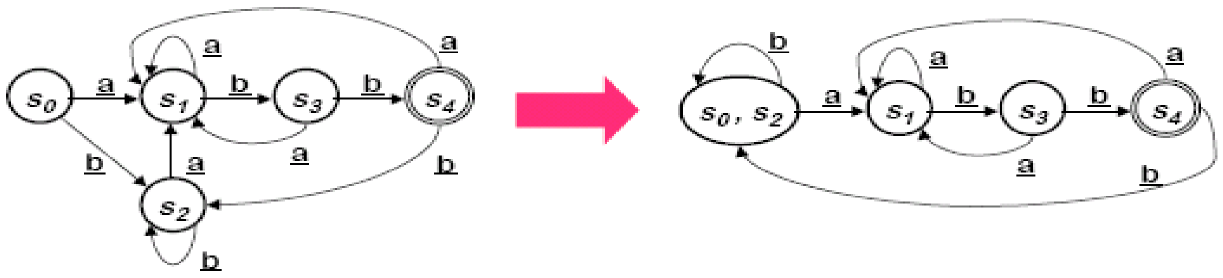
Example # 1



	Character	
State	a	b
s_0	s_1	s_2
s_1	s_1	s_3
s_2	s_1	s_2
s_3	s_1	s_4
s_4	s_1	s_2

Minimization of DFA

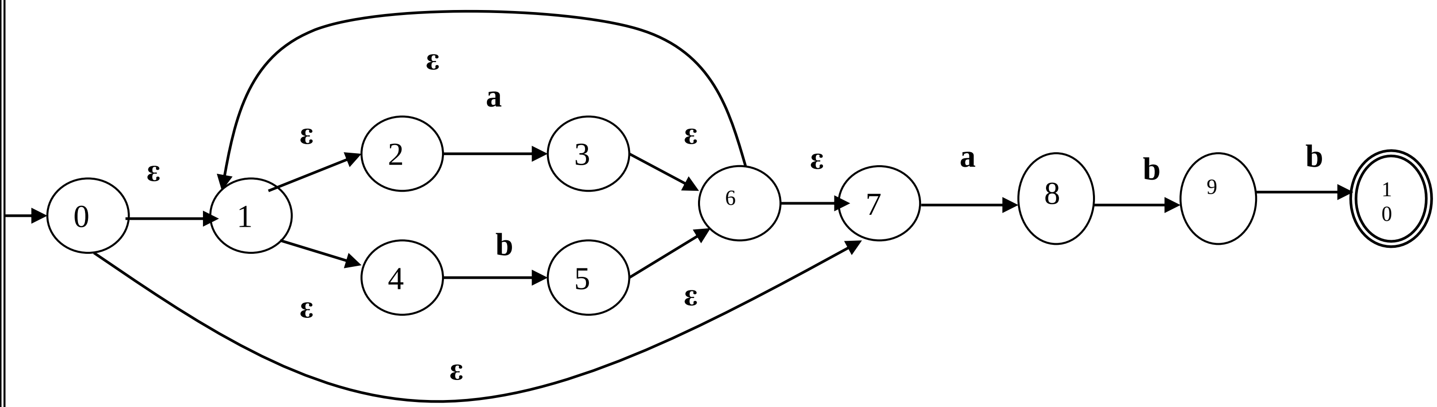
	Current Partition	Split on a	Split on b
P_0	$\{s_4\}\{s_0,s_1,s_2,s_3\}$	none	$\{s_3\}\{s_0,s_1,s_2\}$
P_1	$\{s_4\}\{s_3\}\{s_0,s_1,s_2\}$	none	$\{s_1\}\{s_0,s_2\}$
P_2	$\{s_4\}\{s_3\}\{s_1\}\{s_0,s_2\}$	none	none
P_2	$\{s_4\}\{s_3\}\{s_1\}\{s_0,s_2\}$	none	none



12.Convert the regular expression (a+b) * abb into NFA – ε and find the equivalent minimum state DFA. [NOV/DEC 2008]

Solution:

The NFA for the RE (a+b) * abb is shown below,



Step 1:

ε – closure (0) ={ 0,1,2,4,7} → A

Now process the state A with ‘a’ and ‘b’.

Step 2:

Move [A,a] = {3,8} (Similar to δ(A,a))

ε – closure (Move [A,a]) = { 1,2,3,4,6,7,8} → B

Move [A,b] = {5}

ε – closure (Move[A,b]) = { 1,2,4,5,6,7} → C

Step 3:

Move [B,a] = {3,8}

ε – closure (Move [B,a]) = {1,2,3,4,6,7,8} → B

Move [B,b] ={5}

ε – closure (Move [B,b]) = {1,2,4,5,6,7,9} → D

Step 4:

Move[C,a] = {3,8}

ε – closure (Move[C,a]) = {1,2,3,4,6,7,8} → B

Move [C,b] = {5}

ε – closure (Move [C,b]) = {1,2,4,5,6,7} → C

Step 5:

Move[D,a] = {3,8}

ε – closure (Move[D,a]) = {1,2,3,4,6,7,} → B

Move[D,b] ={5,10}

ε – closure (Move[D,b]) = {1,2,4,5,6,7,10} → E

Step 6:

Move [E,a] = {3,8}

ε – closure (Move [E,a]) = {1,2,3,4,6,7,8} → B

Move [E,b] = {5}

ε – closure (Move [E,b]) = { 1,2,4,5,6,7} → C

The transition table for DFA is given below,

	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
*E	B	C

To Find Minimized DFA

By using minimized algorithm the minimized DFA is as follows,

[A B C D E]

[A B C D] [E] { E is final state}

[A B C] [D] [E] { Since $\delta(D, b) = E$ }

[A C] [B] [D] [E]

A ≡ C , Eliminate C and replace ‘C’ by ‘A’

The transition table for minimum state DFA is given below,

	a	b
→ A	B	A
B	B	D
D	B	E
* E	B	A

13. Discuss about LEX tool.

A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER

1. Use of Lex

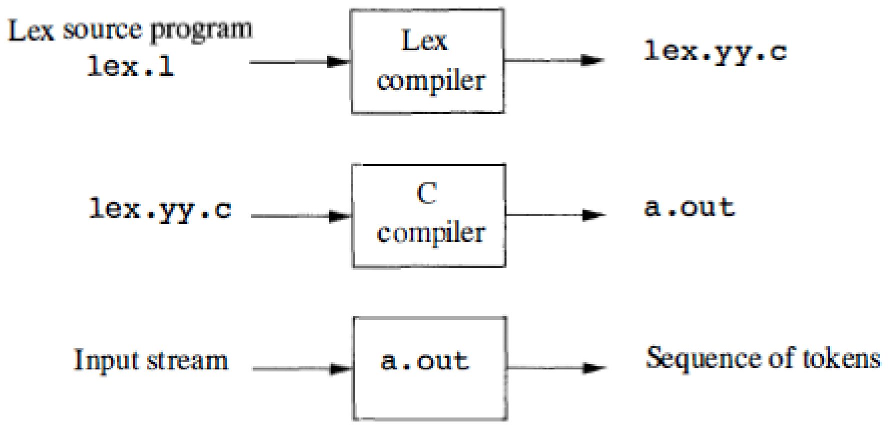
In this section, we introduce a tool called Lex, or in a more recent implementation Flex, that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens. The input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler. Behindthe scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex . yy . c, that simulates this transition diagram.

There is a wide range of tools for constructing lexical analyzers.

 Lex

LEX

Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc parser generator.



Creating a lexical analyzer with Lex

2. Structure of Lex programs

First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.

Finally, lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

lex.l lex.yy.c

lex.yy.c a.out

Lex Specification

A Lex program consists of three parts:

declarations

%%

translation rules

%%

auxiliary functions

Definitions or declarations include declarations of variables, constants, and regular definitions

Rules are statements of the form

p1 {action 1}

p2 {action 2}

…p

n {action n}

where pi is regular expression and action i describes what action the lexical analyzer should take when pattern pi matches a lexeme. Actions are written in C code.

 User subroutines or auxiliary functions are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

Example:

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}

%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Lex program for the tokens of Fig.

3. **Conflict Resolution in lex**

We have alluded to the two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:

1. Always prefer a longer prefix to a shorter prefix.
2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

4. **The Lookahead Operator**

Lex automatically reads one character ahead of the last character that forms the selected lexeme, and then retracts the input so only the lexeme itself is consumed from the input. However, sometimes, we want a certain pattern to be matched to the input only when it is followed by a certain other characters. If so, we may use the slash in a pattern to indicate the end of the part of the pattern that matches the lexeme. What follows / is additional pattern that must be matched before we can decide that the token in question was seen, but what matches this second pattern is not part of the lexeme.

14. Discuss about Lexical errors.

Detection of Errors

• To be useful, a compiler should detect all errors in the source code and report them to the user. • These errors culd be:

Compilation Errors

• Lexical errors: e.g., badly formed identifiers or constants, symbols which are not part of the language, badly formed comments, etc.

• Syntactic errors: chains of syntactic units that do not conform to the syntax of the source language.

• Semantic errors: e.g., operations conducted on incompatible types, undeclared variables, double declaration of variable, reference before assignment, etc.

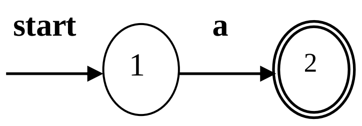
Execution Errors

• Run-time errors: errors detectable solely at run time, pointers with null value or whose value is outside allowed limits, or indexing of vectors with unsuitable indices, etc.

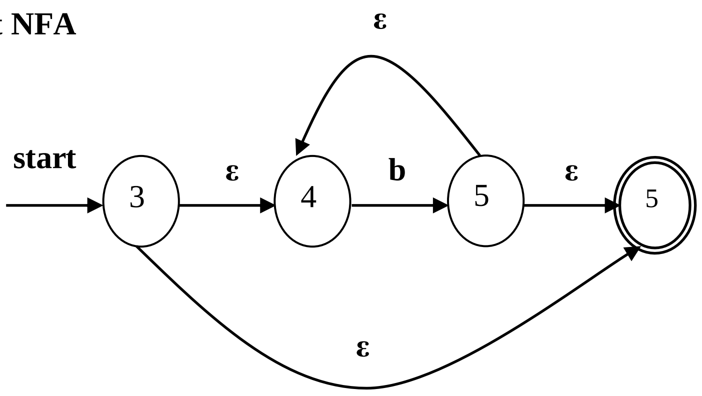
15. Draw NFA for the regular expression ab*/ab.[May/June-14]

Solution:

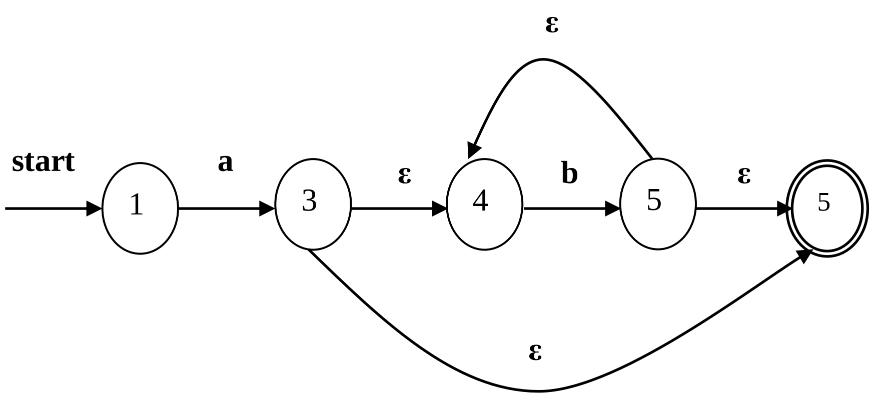
For $r_{1=}$ a construct NFA



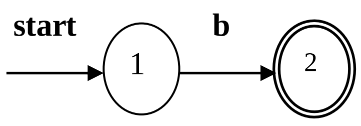
For $r_{=b*}$ construct NFA



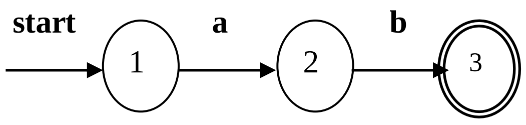
For r_3= r_1. r_2 construct NFA



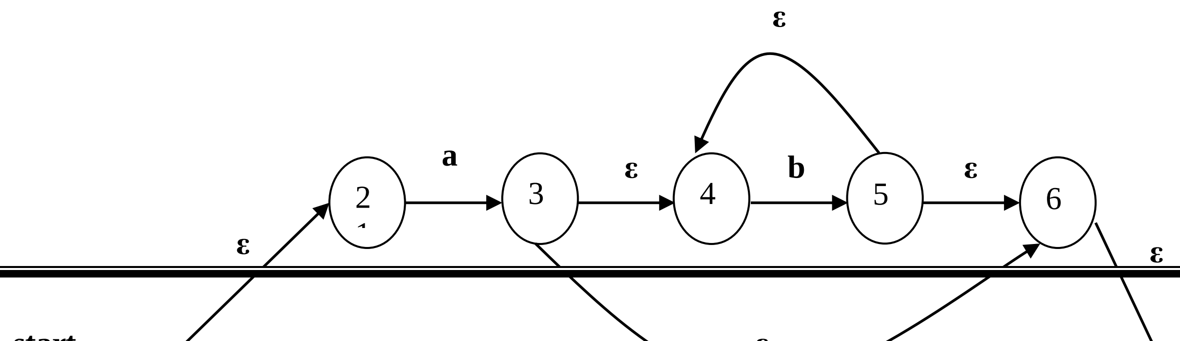
For r_4= b construct NFA



For r_5= r_1. r_4 construct NFA



For r_7= r_3/r_5 construct NFA



16. Explain briefly about Design of Lexical Analyzer for a sample Language.

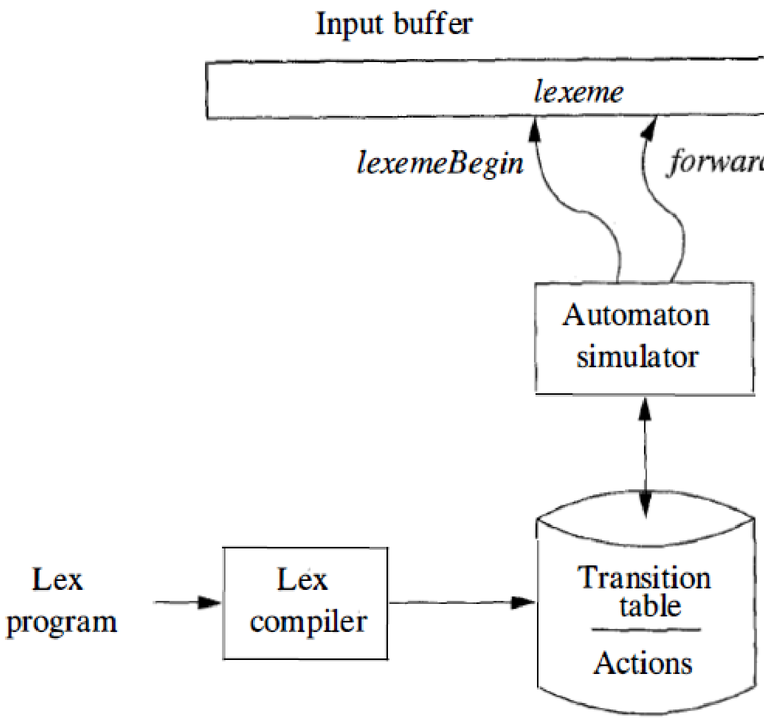
We assume that we have a specification of a lexical analyzer of the form

P1 { action1 }

P2 { action2 }

……

Pn { actionn }



where, each pattern pi is a regular expression and each action action$_i$ is a program fragment that is to be executed whenever a lexeme matched by p_i is found in the input.

Our problem is to construct a recognizer that looks for lexemes in the input buffer. If more than one pattern matches, the recognizer is to choose the longest lexeme matched. If there are two or more patterns that match the longest lexeme, the first-listed matching pattern is chosen.

A finite automaton is a natural model around which to build a lexical analyzer, and the one constructed by our Lex compiler has the form shown in Fig. above. There is an input buffer with two pointers to it, a lexeme beginning and a forward pointer. The Lex compiler constructs a transition table for a finite automaton from the regular expression patterns in the Lex specification. The lexical analyzer itself consists of a finite automaton simulator that uses this transition table to look for the regular expression patterns in the input buffer.

compiler can be based on either nondeterministic or deterministic automata, At the end of the last section we saw that the transition table of an NFA for a regular expression pattern can be considerably smaller than that of a DFA, but the DFA has the decided advantage of being able to recognize patterns faster than the NFA.

Pattern Matching Based on NFA'S

One method is to construct the transition table of a nondeterministic finite automaton *N* for the composite pattern $P_1| P_2|\ldots| P_n$. This can be done by first creating an NFA $N(p_i)$ for each pattern p_i using Algorithm, then adding a new start state s_o, and finally linking s_o to the start state of each *N (p* $_i$ *)* with an €-transit ion, *as* shown in Fig. below.

To simulate this NFA we can use a modification of Algorithm 3.4. The modification ensures that the combined NFA recognizes the longest prefix of the input that is matched by a pattern. In the combined NFA, there is an accepting state for each pattern *pi.* When we simulate the NFA using Algorithm 3.4, *we* construct the sequence of sets of states that the combined NFA can b in after seeing each input character. Even if we find a set of states that contains an accepting state, to find the longest match we must continue to simulate the NFA until it reaches *termination*, that is, a set of states from which there are no transitions on the current input symbol.

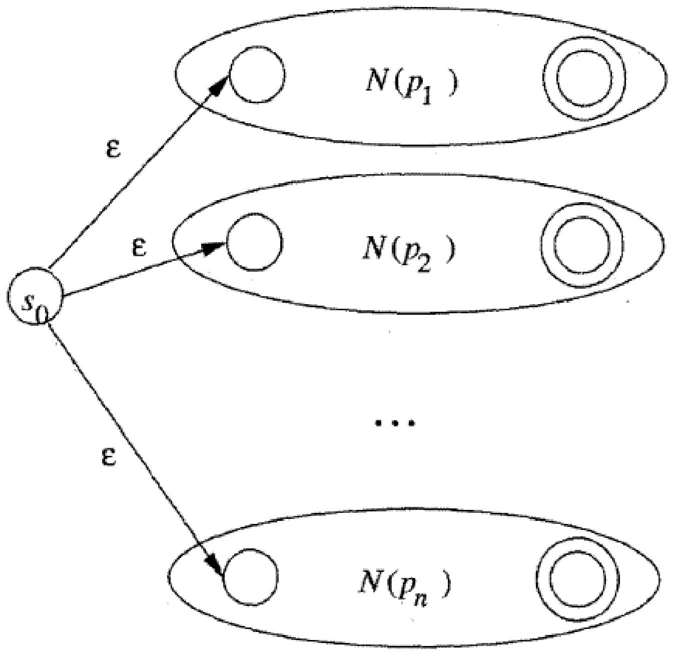


Figure 3.50: An NFA constructed from a Lex program

a	{ action A_1 for pattern p_1 }
abb	{ action A_2 for pattern p_2 }
$\mathbf{a}^*\mathbf{b}^+$	{ action A_3 for pattern p_3 }

We presume that the Lex specification is designed *so* that a valid source program cannot entirely fill the input buffer without having the NFA reach termination. For example, each compiler puts sow restriction on the length of an identifier, and violations of this limit will be detected when the input buffer overflows, if not sooner,

To find the correct match, we make two modifications to Algorithm. First, whenever we add an accepting state to the current set of stales, we record the cu rent input position and the pattern pi corresponding to this accepting state. If the current set of states already contains en accepting state, then only the pattern that appears first in the Lex specification is recorded.

Second, we continue making transitions until we reach termination. Upon termination, we retract the forward painter to the position at which the last match occurred. The pattern making this match identifies the token found, and the lexeme matched is the string between the lexeme-beginning and forward pointers.

Usually, the Lex specification is such that some pattern, possibly an error pattern, will always match. If no pattern matches, however, we have an error condition for which no provision was made, and the lexical analyzer should transfer control to some default error recovery routine.

Example 3.18

A simple example illustrates the above ideas. Suppose we have the following Lex program consisting of three regular expressions and no regular definitions,

```
a          { action A\  for pattern p\  }
abb        { action A,  for pattern p,  }
a*b+      { action As  for pattern p%  }
```

The three tokens above are recognized by the automata of Fig.*(a)*. We have simplified the third automaton somewhat from what would be produced by Algorithm . As indicated above, we can convert the NFA's of Fig.(a) into one combined NFA *W* shown in (b).

Let us now consider the behavior of *N* on the input string *aaba* using our modification of Algorithm. Figure 3,36 shows the sets of states and patterns that match as each character of the input aaba is processed. This figure shows that the initial: set of states is 10, l, 3, 7). States I , 3, and 7 each haw a transition on a, to states 2, 4, and 7, respectively. Since state 2 is the accepting state for the first pattern, we record the fact that the first pattern matches after reading the first a.

However, there is a transition from state 7 to state 7 on the second input character, so we must continue making transitions. There is a transition from state 7 to state 8 on the input character b. State 8 is the accepting state for the third pattern. Once we reach state 8, there arc no transitions possible on the next input character a so we have reached termination. Since the last match occurred after we read the third input character, we repor1 that the third pattern has matched the lexeme *aab*.

The role of action$_i$ associated with the pattern pi in the Lex specification is as follows. When an instance of pi is recognized, the lexical analyzer executes the associated program *action$_i$*. Note that *action;* is not executed just because the NFA enters a state that includes the accepting state for pi; *action$_i$* is only executed if *pi* turns out to be the pattern yielding the longest match.

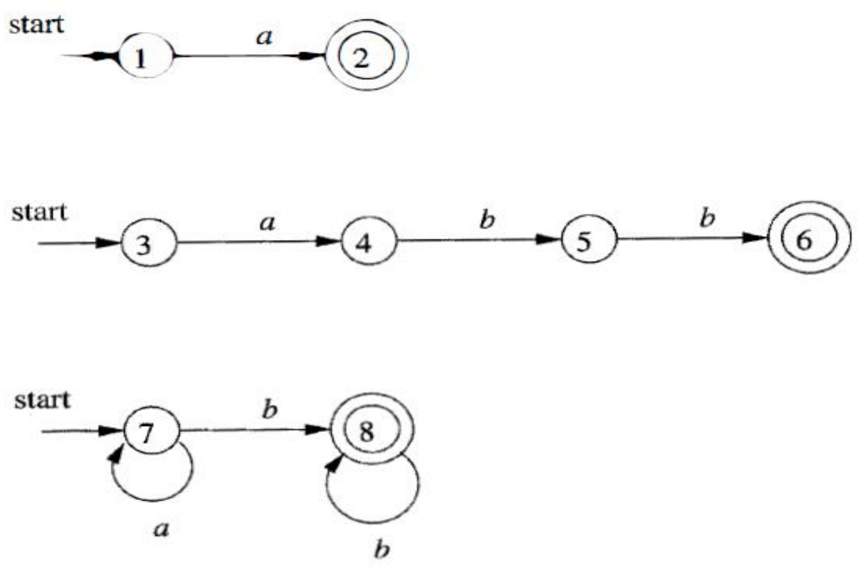


Figure 3.51: NFA's for **a**, **abb**, and **a*b**$^+$

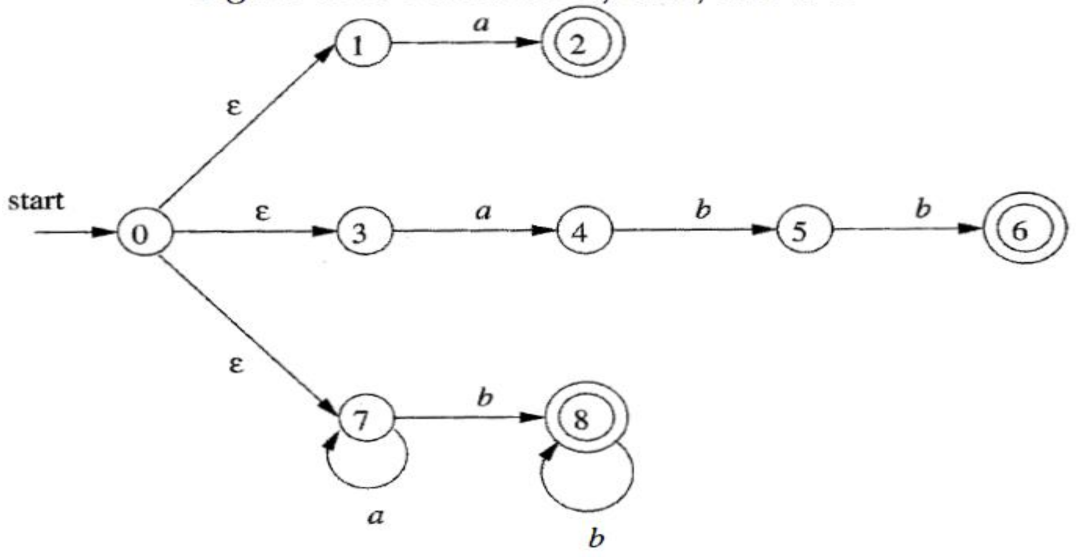
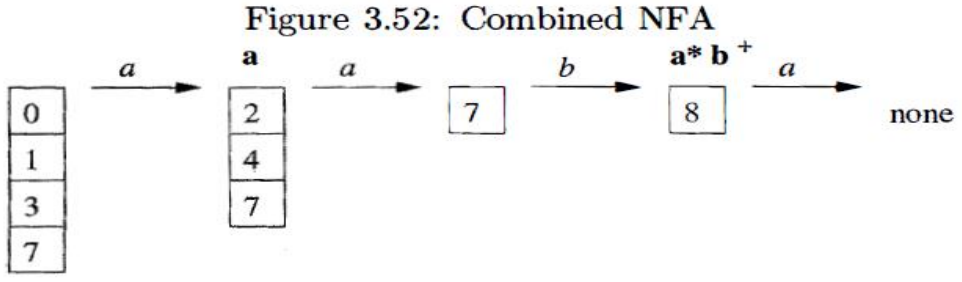


Figure 3.52: Combined NFA

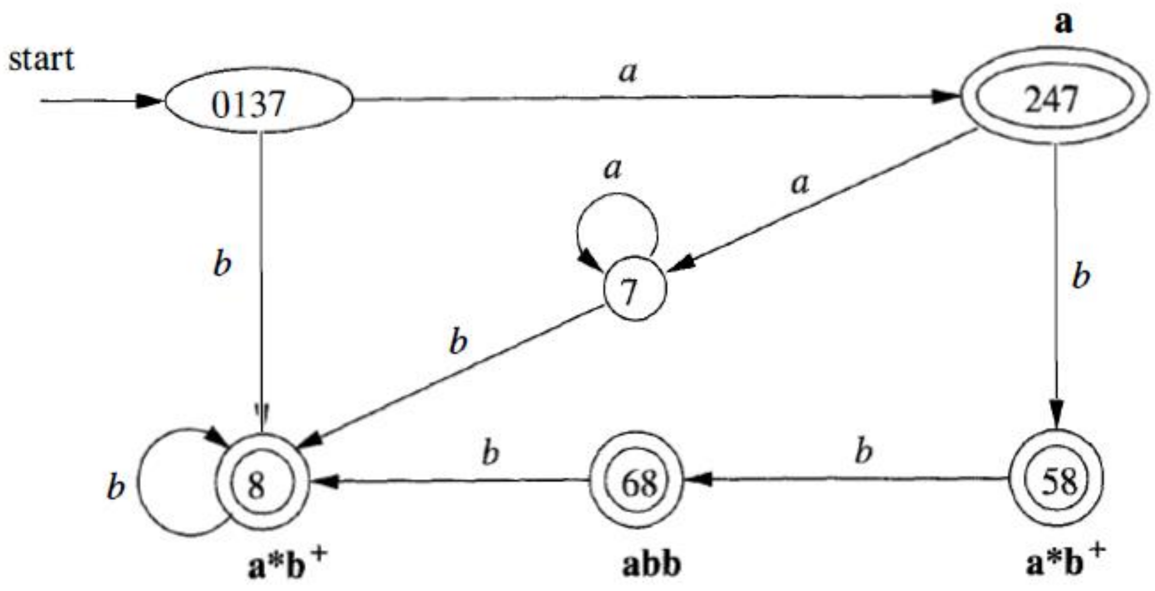


Sequence of sets of states entered when processing input *aaba*

DFA's fo r Lexical Analyzers

Another approach to the construction of a lexical analyzer from a Lex specification is to use 3 DFA to perform the pattern matching+ The only nuance is to make sure we find the proper pattern matches. The situation is completely analogous to the modified simulation of an NFA jug described. When we convert an NFA to a DFA using the subset construction Algorithm 3.2, there may be several accepting states in a given subset of nondeterministic states.

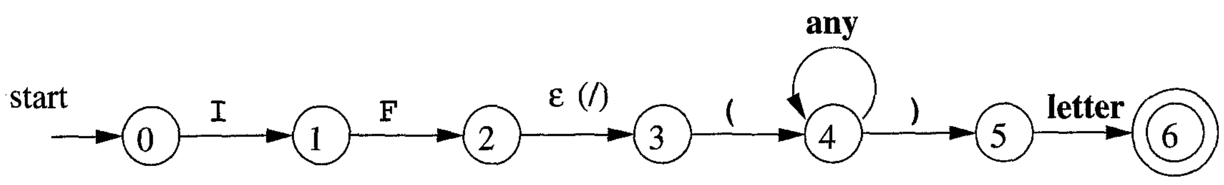
In such a situation, the accepting states corresponding to the pattern listed first in the Lex specification has priority. As in the NFA simu4ation, the only other modification we need to perform is to continue making state transitions until we reach a state with no next state(i.e., the state *0)* for the current input symbol, To find the lexeme matched, we return to the last input position at which the DFA entered an accepting state.



Transition graph for DFA handling the patterns **a**, **abb**, and **a*b**$^{+}$

Recall from Section 3.4 that the lookahead operator *1* is necessary in some situations, since the pattern that denotes a particular token may need to describe some trailing context for the actual lexeme. When converting a pattern with / to an NFA, we car, treat the / as if it were c, so that we do not actually look for / on the input, However, if a string denoted by this regular expression is recognized in the input buffer, the end of the lexeme is not the position of the NFA's accepting state. Rather the end occurrence of the state of this NFA having a transition on the (imaginary) /.

Example 3. The NFA recognizing the pattern for IF given in Example 3.12 i s shown in Fig. State *6* indicates the presence of keyword IF; however, we find the token IF by scanning backwards to the laa occurrence of state **2.**

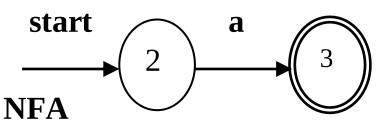


NFA recognizing the keyword IF

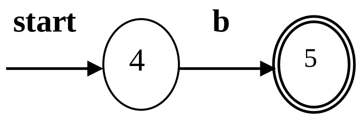
Refer the lex program that is question number 8.

17. Write notes on regular expression to NFA. Construct Regular expression to NFA for the sentence (a/b)*a. (10) May/June 2016

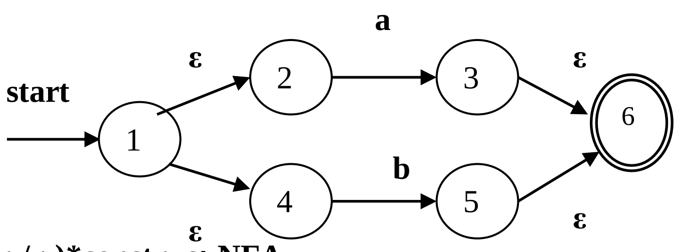
For r_1 construct NFA



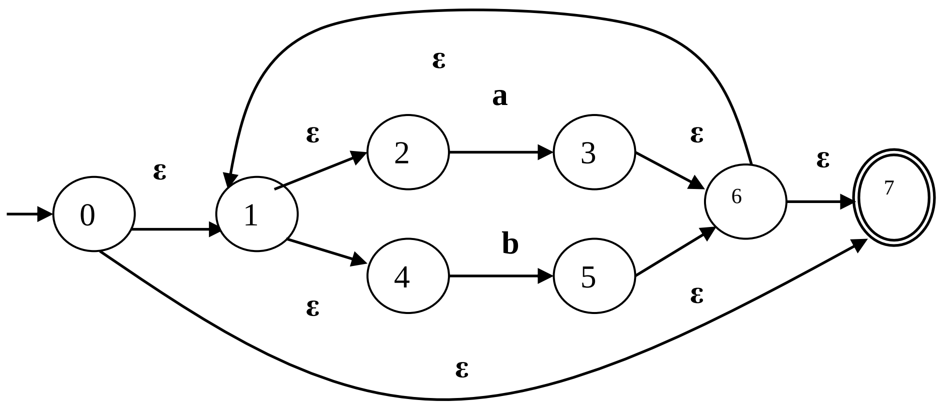
For r_2 construct NFA



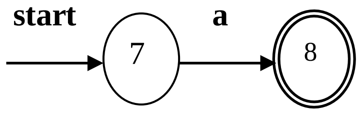
For r_3= r_1/ r_2 construct NFA



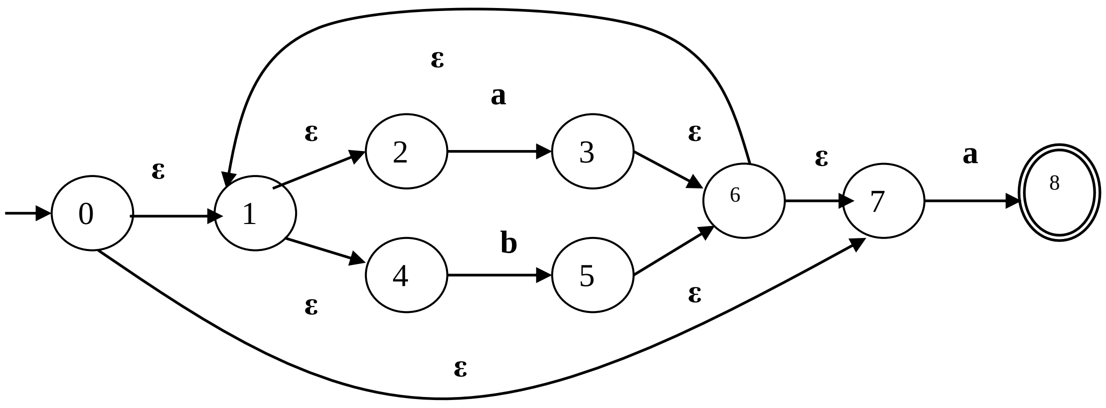
For r_3= (r_1/r_2)*construct NFA



For r_6= a construct NFA

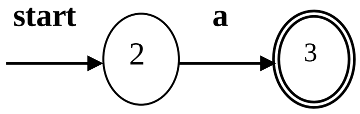


For r_7= r_5.r_6 construct NFA

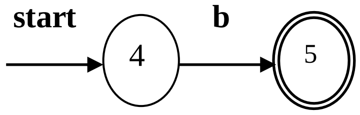


18. Conversion of regular expression (a/b)*abb to NFA. (8) (Nov/Dec 2016)

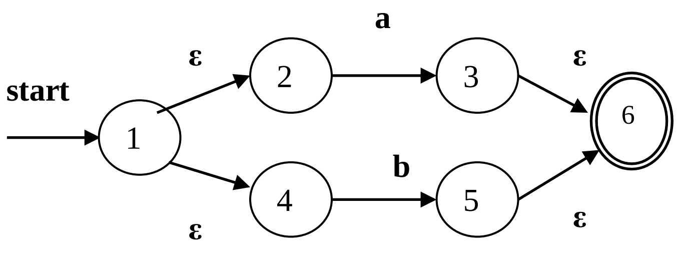
For r_1 construct NFA



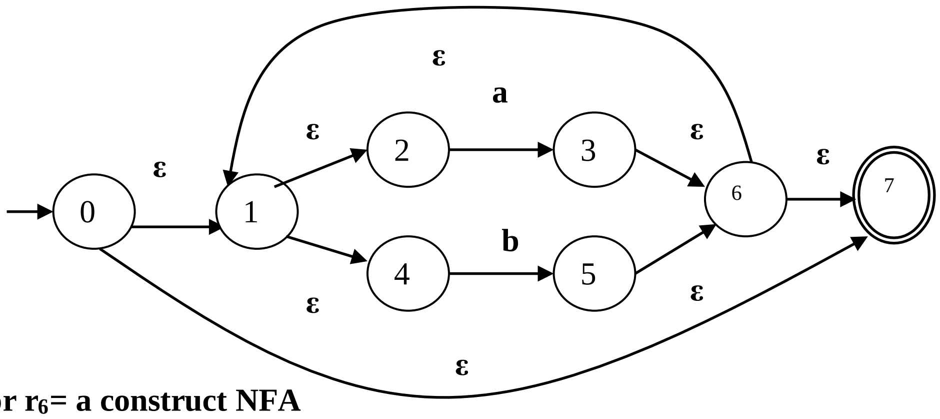
For r_2 construct NFA



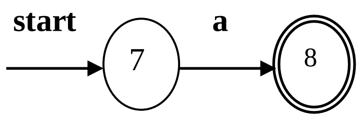
For r_3= r_1/ r_2 construct NFA



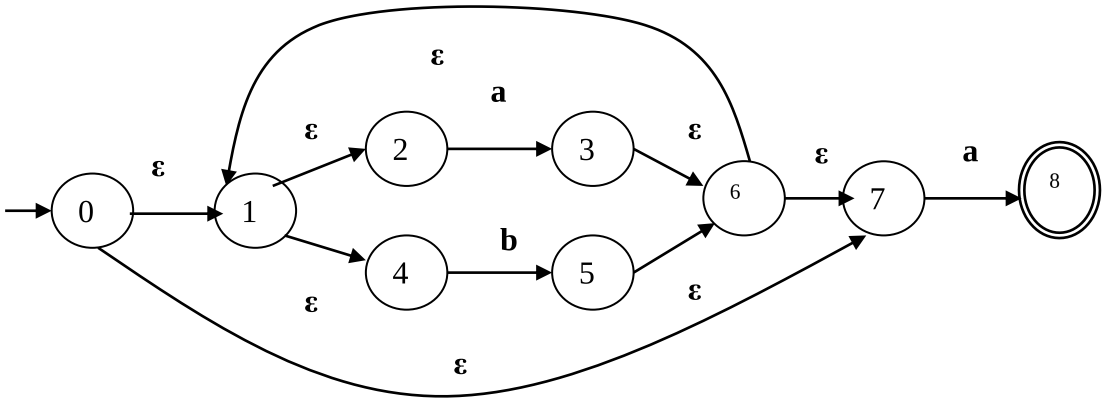
For r_3= (r_1/r_2)*construct NFA



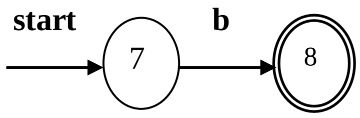
For r_6= a construct NFA



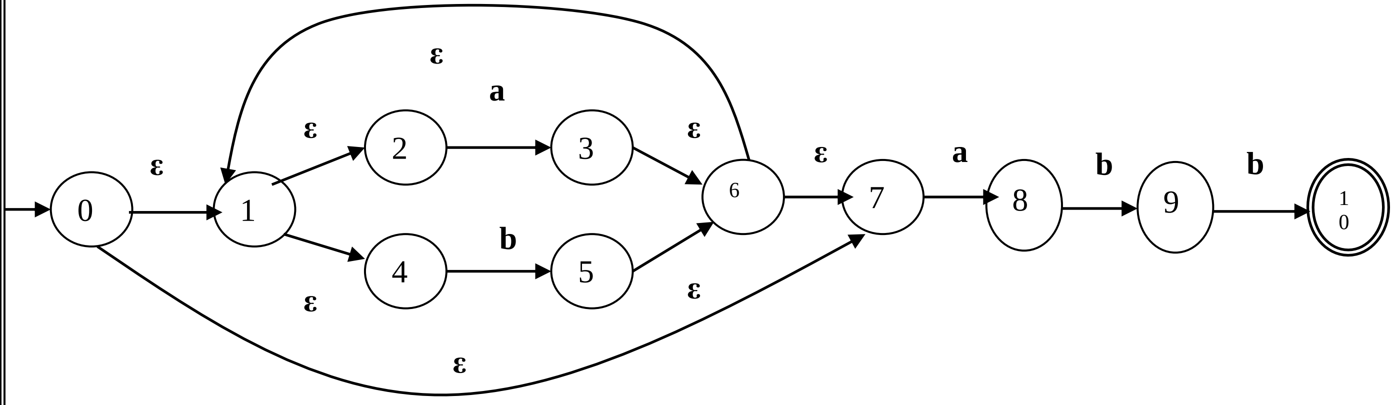
For r_7= r_5.r_6 construct NFA



For r_8= b construct NFA



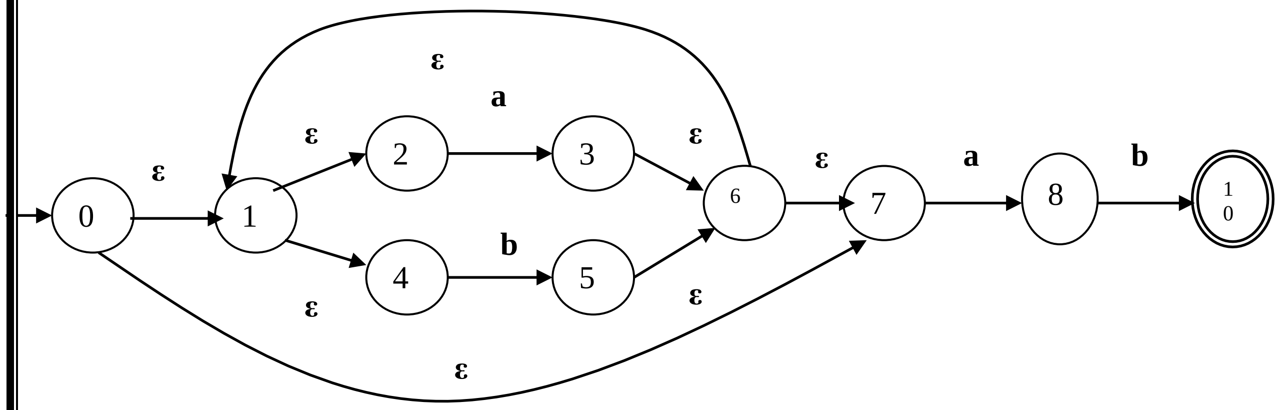
For r_9= r_7.r_8. r_8 construct NFA



19. Construct DFA to recognize the language (a/b)*ab. (6) (May/June 2016)

Solution:

The NFA for the RE (a/b) * ab is shown below,



Step 1:

ε – closure (0) ={ 0,1,2,4,7} → A

Now process the state A with 'a' and 'b'.

Step 2:

δ(A,a)= ε – closure (δ [A,a])

= ε – closure (δ[{ 0,1,2,4,7},a])

= ε – closure(3,8)

={ 3,6,7,1,2,4,8} → B

δ(A,b)= ε – closure (δ [A,b])

= ε – closure (δ[{ 0,1,2,4,7},b])

= ε – closure(5)

={ 5,6,7,1,2,4} → C

Step 3:

δ(B,a)= ε – closure (δ [B,a])

= ε – closure (δ[{ 3,6,7,1,2,4,8}a])

= ε – closure(3,8)

={ 3,6,7,1,2,4,8} → B

δ(B,b)= ε – closure (δ [B,b])

= ε – closure (δ[{ 3,6,7,1,2,4,8},b])

= ε – closure(5,9)

={ 5,6,7,1,2,4,9} → D

Step 4:

δ(C,a)= ε – closure (δ [C,a])

= ε – closure (δ[{ 5,6,7,1,2,4},a])

= ε – closure(3,8)

={ 3,6,7,1,2,4,8} → B

δ(C,b)= ε – closure (δ [C,b])

= ε – closure (δ[{ 5,6,7,1,2,4},b])

= ε – closure(5)

={ 5,6,7,1,2,4} → C

Step 5:

δ(D,a)= ε – closure (δ [D,a])

= ε – closure (δ[{ 5,6,7,1,2,4,9},a])

= ε – closure(3,8)

={ 3,6,7,1,2,4,8} → B

δ(D,b)= ε – closure (δ [D,b])

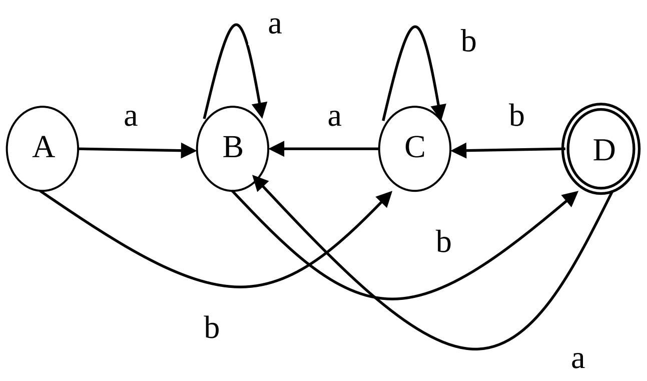
= ε – closure (δ[{ 5,6,7,1,2,4,9},b])

= ε – closure(5)

={ 5,6,7,1,2,4} → C

The transition table for DFA is given below,

	a	**b**
→ **A**	**B**	**C**
B	**B**	**D**
C	**B**	**C**
***D**	**B**	**C**



UNIVERSITY QUESTION PAPER

NOV/DEC 2016

PART A

1 . List the rules that form the BASIS.

2 . Differentiate tokens, patterns, lexeme:

PART B

1. (a) (i)Discuss the role of lexical analyzer in detail with necessary
Examples.(8)

(ii) Discuss how a finite automaton is used to represent tokens and perform lexical analysis with examples.(8)

2.(b) (i)Conversion of regular expression (a/b)*abb to NFA.(8)

(ii) Write an algorithm for minimizing the number of states of a DFA.(8)

MAY/JUNE-2016

PART A

1. Write a grammar for branching statements.
2. List the operations on languages.

PART B

1. (a) (i) Differentiate between lexeme, token and pattern.(6)

   (ii) What are the issues in lexical analysis?(4)

   (iii) Write notes on regular expressions. (6)

   Or

   (b) (i) Write notes on regular expression to NFA. Construct Regular expression to NFA for the sentence (a/b)*a. (10)

   (ii) Construct DFA to recognize the language (a/b)*ab. (6)

MAY/JUNE-2014

PART-A

1.Define Lexeme.

2..Compare the features of DFA and NFA.

PART B

1.a)i)Differentiate between lexeme,token and pattern.(6)

ii)What are the issues in lexical analysis?(4)

2. i)Draw NFA for the regular expression ab*/ab.

NOV/DEC-2013

PART A

1.Write the regular expresstion for identifier and white space.

PART-B

1. Draw the DFA for the augmented regular expression(a|b)*# directly using syntax tree.(12)

MAY/JUNE 2013

PART A

1. Define tokens, patterns and lexemes.
2. Mention the issues in a lexical analyzer.

PART-B

1. (ii) Explain in detail about the role of lexical analyzer with the possible error recovery action.

2. (ii) Elaborate specification of token.

MAY/JUNE 2012

PART A

1. What are the possible error recovery actions in lexical analyzer?

PART B

(b) (i) What are the issues in lexical analysis?

(ii)Elaborate in detail the recognition of tokens.

APRIL/MAY 2011

PART A

1. Define token and lexeme.

PART B

2. (i) Discuss the role of lexical analyzer in detail.

(ii) Draw the transition diagram for relational operators and unsigned numbers in Pascal.

NOV/DEC 2011

PART A

1. What is the role of lexical analyzer?

2. Give the transition diagram for an identifier.

PART B

1.(ii) Construct the minimized DFA for the regular expression (0+1)*(0+1) 10.

MAY/JUNE 2009

Part A

1. What are the issues to be considered in the design of lexical analyzer?

2. Define concrete and abstract syntax with example.

NOV/DEC 2007

PART-B

1.Explain in detail about the role of lexical analyzer with the possible error recovery actions.

CS6660- COMPILER DESIGN – UNIT I
INTRODUCTION TO COMPILERS

Translators-Compilation and Interpretation-Language processors -The Phases of Compiler-Errors Encountered in Different Phases-The Grouping of Phases-Compiler Construction Tools -Programming Language basics.

TWO MARKS

1. Define compiler.(Nov/Dec09)

A compiler is a program that reads a program written in one language –the **source language** and **translates** it into an equivalent program in another language-the **target language**. The compiler **reports** to its user the **presence of errors** in the source program. Example: Turbo C, common for both C & C++

2. What is the need for a Compiler?

We need Compilers, because the source program does not understand by the Computer. So, it has to convert into machine understandable language. So we use Compilers for this purpose.

We cannot use the same compiler for all computers. Because every HLL has its own syntaxes.

3. What are the two types of analysis of the source programs by compiler? (April/May 10)

Or What are the two parts of a compilation? Explain briefly. MAY/JUNE-2016

Analysis and **Synthesis** are the two parts of compilation.

The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.

The synthesis part constructs the desired target program from the intermediate representation.

4. What is a language processing system?(Nov/Dec 2008)

The program which translate the program written in a programming language by the user into an executable program is known as language processors.

5. List the subparts or phases of analysis part.

Analysis consists of three phases:

- Linear Analysis.
- Hierarchical Analysis.
- Semantic Analysis.

6. What is linear analysis?

Linear analysis is one in which the stream of characters **making up the source program** is read from left to right and grouped **into tokens** that are sequences of characters having a collective meaning.

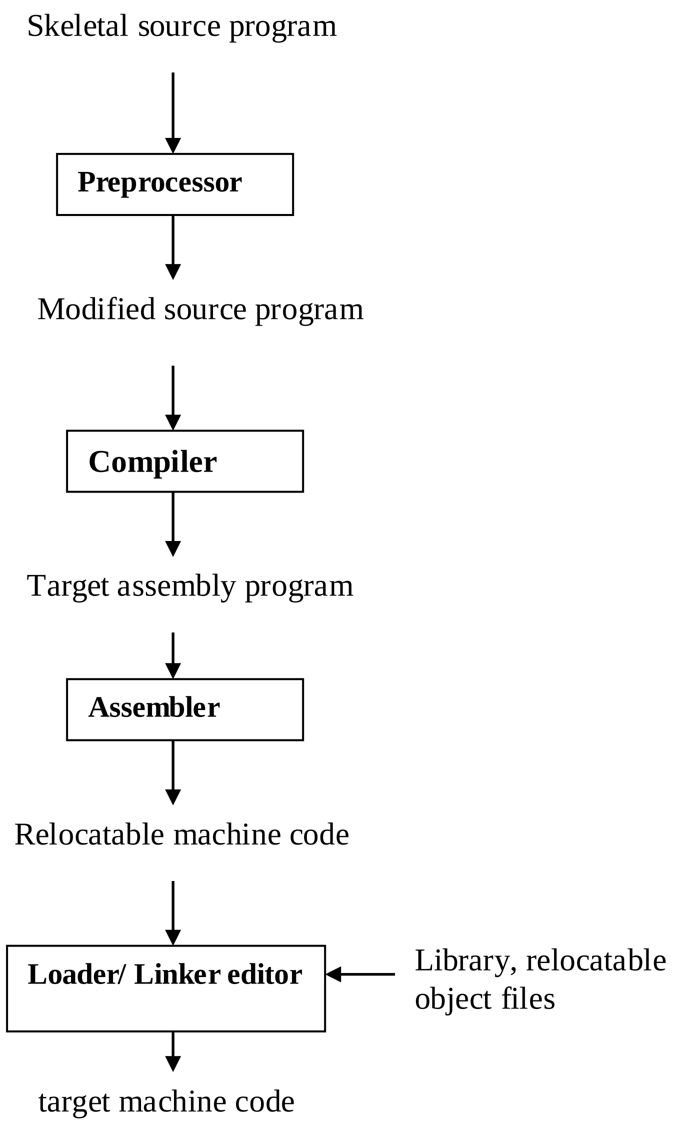
Also called lexical analysis or scanning.

7. Describe the error recovery schemes in a lexical phase of a compiler.(April/May 2015)

- Panic mode
- Statement mode

- Error productions
- Global correction

8. Illustrate diagrammatically how a language is processed. MAY/JUNE-2016



9. What are the phases of a compiler?

Phases of a Compiler

1. Lexical analysis (“scanning”)
    - Reads in program, groups characters into “tokens”
2. Syntax analysis (“parsing”)
    - Structures token sequence according to grammar rules of the language.
3. Semantic analysis
    - Checks semantic constraints of the language.
4. Intermediate code generation
    - Translates to “lower level” representation.
5. Program analysis and code optimization
    - Improves code quality.

6. Final code generation.

10. What is a symbol table? (Nov/Dec 2007) (Nov/Dec 2016)

A symbol table is a data structure containing a **record for each identifier**, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

Whenever an identifier is detected by a lexical analyzer, it is entered into the symbol table. The attributes of an identifier cannot be determined by the lexical analyzer.

11. Mention some of the cousins of a compiler. [May/June 2012]

Cousins of the compiler are:

- Preprocessors
- Assemblers
- Loaders and Link-Editors

12. What is grouping of phases?[Nov/Dec 2013][May/June-14]

Grouping of Phases

- *Front end* : machine independent phases
  - Lexical analysis
  - Syntax analysis
  - Semantic analysis
  - Intermediate code generation
  - Some code optimization
- *Back end* : machine dependent phases
  - Final code generation
  - Machine-dependent optimizations

13. List the phases that constitute the front end of a compiler.

The front end consists of those phases or parts of phases that depend primarily on the source language and are largely independent of the target machine. These include

- Lexical and Syntactic analysis
- The creation of symbol table
- Semantic analysis
- Generation of intermediate code

A certain amount of code optimization can be done by the front end as well. Also includes error handling that goes along with each of these phases.

14. Mention the back-end phases of a compiler.

The back end of compiler includes those portions that depend on the target machine and generally those portions do not depend on the source language, just the intermediate language. These include

 Code optimization

 Code generation, along with error handling and symbol- table operations.

15. Define compiler-compiler.

Systems to help with the compiler-writing process are often been referred to as compiler-compilers, compiler-generators or translator-writing systems.

Largely they are oriented around a particular model of languages, and they are suitable for generating compilers of languages similar model.

16. What are compiler construction tools? (Nov/Dec 04,05) (Nov/Dec 2016)

The following is a list of some compiler construction tools:

- Parser generators
- Scanner generators
- Syntax-directed translation engines
- Automatic code generators
- Data-flow engines

17. What does a semantic analysis do?

Semantic analysis is one in which certain checks are performed to ensure that components of a program fit together meaningfully.

Mainly performs type checking.

18. Define translator.

Translator converts the high level programming code to a machine understandable code.

19. What do you mean by assembler?

Assembler translates assembly language program, which uses mnemonic names for operation codes and data addresses into the machine language.

20. What do you mean by preprocessor? (May/June07)

A translator whose source language and object language are high- level language is termed as preprocessor.

21. Define linker.

A linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

22. Define loader.

A loader is the part of an operating system that is responsible for loading programs, one of the essential stages in the process of starting a program.

23. Define macro processor.

A macro processor is a system program that **replaces the macro calls** in an assembly language program **by the macro definition**. During this text replacement process, the formal arguments in the macro definition are converted into the actual arguments in the macro call.

24. Define phase.

A phase is a logically cohesive operation that takes as input one representation of source program and produces as output another representation of the source program.

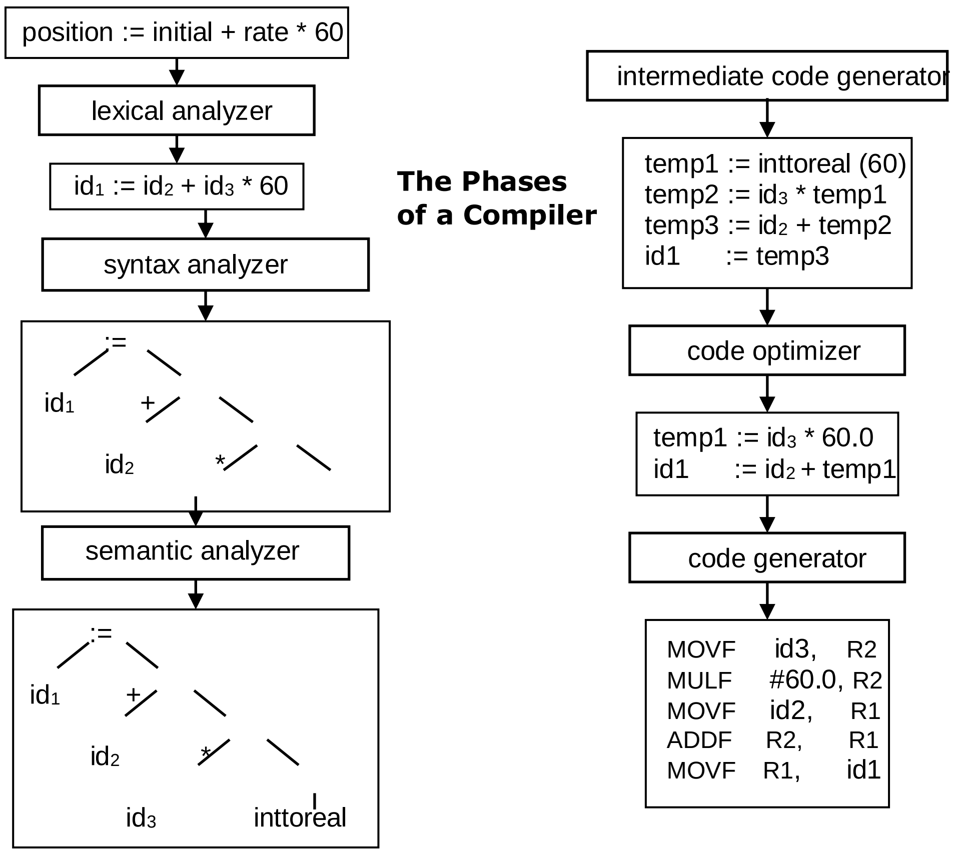
25. Differentiate compiler and interpreter. (May/June2008)

Compiler	**Interpreter**
Merit: In the process of compilation the program is analyzed only once and then the code is generated. Hence compiler is efficient than interpreter.	Demerit: The source program gets interpreted every time it is to be executed, and every time the source program is analyzed. Hence interpretation is less efficient than compiler.
The compiled produce object code.	The interpreters do not produce object code.
Demerit: The compiler has to be present on the host machine when particular program needs to be compiled. Demerit : The compiler is a complex program and it requires large amount of memory.	Merit: The interpreters can be made portal because they do not produce object code. Merit: Interpreters are simpler and give us improved debugging environment.

26. Define pretty printers?

A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible. For the comments may appear with an amount of indentation proportional to the depth of their nesting in the hierarchical organization of the statements.

27. Explain with diagram how a statement is compiled.



28. Write short notes on error handler?

The error handler is invoked when a flaw in the source program is detected. It must warn the programmer by issuing a diagnostic, and adjust the information being passed from phase to phase so that each phase can proceed. So that as many errors as possible can be detected in one compilation.

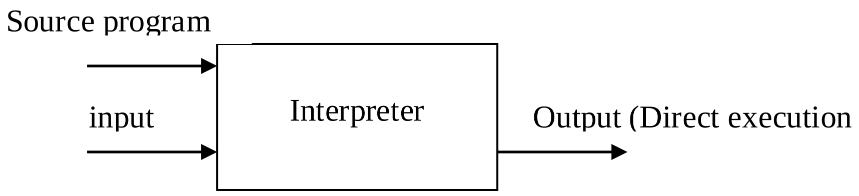
29. What do you meant by passes?

A pass reads the source program or the output of the previous pass, makes the transformations specified by its phases and writes output into an intermediate file, which may then be read by a subsequent pass. In an implementation of a compiler, portions of one or more phases are combined into a module called pass.

30. Define Interpreters. [April/May 2011]

Interpreters

- An interpreter is a kind of translator which produces the result directly when the source language and data is given to it as input.
- It does not produce the object code rather each time he program needs execution. The model for interpreter is as shown in figure



- Languages such as BASIC, SNOBOL, LISP can be translated using interpreters.JAVA also uses interpreter.
- The process of interpretation can be carried out in following phases.
  1. Lexical analysis
  2. Syntax analysis
  3. Semantic analysis
  4. Direct execution.

31. What are the functions of preprocessor? [Nov/Dec 2007]

Preprocessor perform the following functions

1. Macro processing
2. File Inclusion
3."Rational Preprocessors
4. Language extension

32. Define concrete and abstract syntax with example. [May/June 2009]

- Concrete syntax is the surface level of a language (think, strings)
- Abstract syntax is the deep structure of a language (think, trees/terms)
- The abstract syntax defines the way the programs look like to the evaluator/compiler.
- Parsers convert concrete syntax into abstract syntax and have to deal with ambiguity
- It consists of a set of rules (productions) that define the way programs look like to the programmer

33. Define panic mode.

Panic mode error recovery is deleting the successive characters from the remaining input until the lexical analyzer can find a well formed token.

34. What are the phases included in front end of a compiler? What does the front end produce? May -11

The phases of front end of compiler are lexical analysis, syntax analysis and semantic analysis. The front end produces an intermediate code.

35. Define the term cross compiler. Nov/Dec 05

There may be a compiler which run on one machine and produces the target code for another machine. Such a compiler is called cross compiler.

36. Difference between phase and pass.

Phase

The process of compiler is carried out in various steps.

These steps are referred as phases.

The phases of compilation are lexical analysis, syntax analysis, intermediate code generation, code generation code optimization.

Pass

Various phases are logically grouped together to form a pass.

The process of compilation can be carried out in single pass or in multiple passes.

37. What are the factors affecting number of passes in compiler?

Various factors affecting the number of passes in compiler are

1. Forward reference
2. Storage limitations
3. Optimization.

38. What are machine dependent and machine independent phases?

The Machine dependent phases are code generation and code optimization phases.

The Machine independent phases are lexical analyzers, syntax analyzers, semantic analyzers.

39**. What are the different parameter passing mechanisms?**

1) Call-by-value

2) Call-by-reference

40. Define Scope of declaration.

The scope of a declaration of a variable x is the region of the program in which the use of x is referred to this declaration.

41. Define Dynamic scope of a variable.

The dynamic scope refers to the fact anything to be known only during execution. A use of name x refers to the declaration of x in the most recently called procedure containing the delaration of x.

42. What are the classifications of a compiler?

Compilers are classified as:

1. Single-pass
2. Multi-pass
3. Load-and-go
4. Debugging or optimizing.

43. What are the Error Recovery actions in Lexical Analyzer? (Nov/Dec 2008)

1. Deleting an extraneous character

2. Inserting a missing character

3. Replacing an incorrect character by a correct character

4. transposing two adjacent characters.

44. State any two reasons as to why phases of compiler should be grouped. ? (May-June 2014)

Grouping of phases facilities efficient intermediate code generation used portability.

45. Why is buffering used in lexical analysis? What are the commonly used buffering methods? (May-June 2014)

Buffers are used to reduce the overhead required to process an input character.

46. What are the issues in lexical analysis?(May/June2007)

Simpler design will improve performance

- Compiler efficiency is improved
- Compiler portability is enhanced.

SIXTEEN MARKS

1. What is a compiler? Explain about the different types of software tools available for analyzing. (May/June-14)

TRANSLATOR

A translator is a program that takes as input a program **written in one language** and produces as output a **program in another language**. Beside program translation, the translator performs another very important role, the error-detection. Any violation of d HLL specification would be detected and reported to the programmers. Important role of translator are:

1. Translating the **HLL program input into an equivalent ML program.**
2. Providing diagnostic messages wherever the programmer violates specification of the HLL.

TYPE OF TRANSLATORS:-

- INTERPRETOR
- COMPILER
- PREPROSSESSOR

COMPILER (5 MARKS) :-

Compiler is a translator program that **translates a program written in (HLL)** the source program and **translates it into an equivalent program in (MLL)** the target program. As an important part of a compiler is error showing to the programmer.

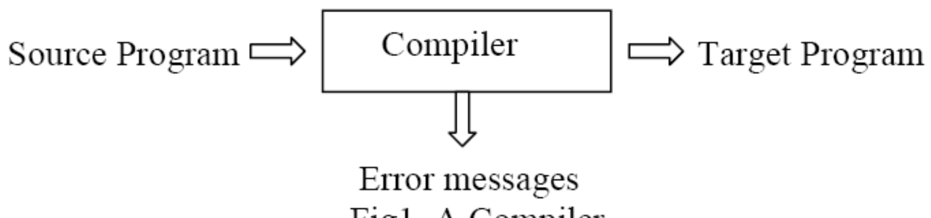


Fig1. A Compiler

Executing a program written in HLL programming language is basically of two parts. The source program must first be compiled translated into **object program**. Then the results object program is **loaded into a memory** executed.

Analysis – Synthesis Model of Compilation:

The process of compilation has two parts namely:

- Analysis
- Synthesis

Analysis:

The analysis part breaks up the source **program into constituent pieces** and creates an intermediate representation of the source program. The analysis part is often called the **front end** of the compiler

Synthesis:

The synthesis part **constructs the desired target** program from the intermediate representation. The synthesis part is the **back end** of the compiler.

Software Tools:

Many software tools that manipulate source programs first perform some kind of analysis. Some examples of such tools include:

- Structure Editors
- Pretty printers
- Static Checkers
- Interpreters

- **Structure Editors**
  - A structure editor takes as input a sequence of commands **to build a source program**.
  - The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program.
  - Example – while …. do and begin….. end.
- **Pretty printers**
  - A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible.
  - For example, comments may appear in a special font.
- **Static Checkers**
  - A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program.
  - For example, a static checker may detect that parts of the source program can never be executed.
- **Interpreters**
  - Translate from high level language (BASIC, FORTRAN, etc..) into assembly or machine language. Interpreters are frequently used to execute command language.
  - An interpreter might build a syntax tree and then carry out the operations at the nodes as it walks the tree.
  - Interpreters are frequently used to execute command language since each operator executed in a command language is usually an invocation of a complex routine such as an editor or complier.

Examples of Compiler:

The following examples are similar to that of a conventional complier.

- Text formatters
- Silicon Compiler
- Query interpreters

Text formatters:

It takes input as a stream of characters includes commands to indicate paragraphs, figures etc.

Silicon Compiler:

Variables represent logical signal 0 & 1 and output is a circuit design.

Query interpreters:

It translates a predicate containing relational & Boolean operators into commands to search a database.

2. <u>What advantages are there to a language processing sytem in which the complier produces assembly language rather than machine laguages. OR</u>

Explain language processing system with neat diagram. (8) May/June 2016

LANGUAGE PROCESSORS

<u>COMPILER :-</u>

Compiler is a translator program that **translates** a program written in **(HLL)** the source program and translates it into an equivalent program in **(MLL)** the target program. As an important part of a compiler is error showing to the programmer.

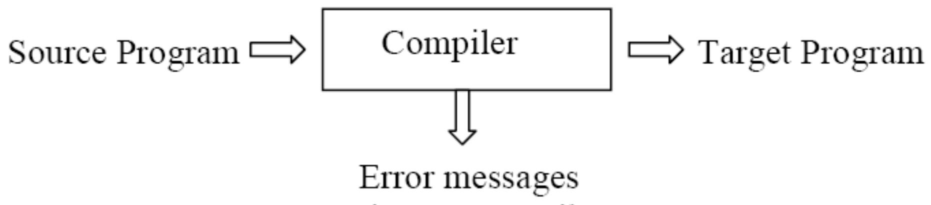


Fig1. A Compiler

Executing a program written in HLL programming language is basically of two parts. The source program must first be compiled translated into a **object program**. Then the results object program is **loaded into a memory executed**.

If the target program is an executable machine- language program. It can then be called by the user to process inputs and produce output see fig. below

input
Target program
output

Running the target program

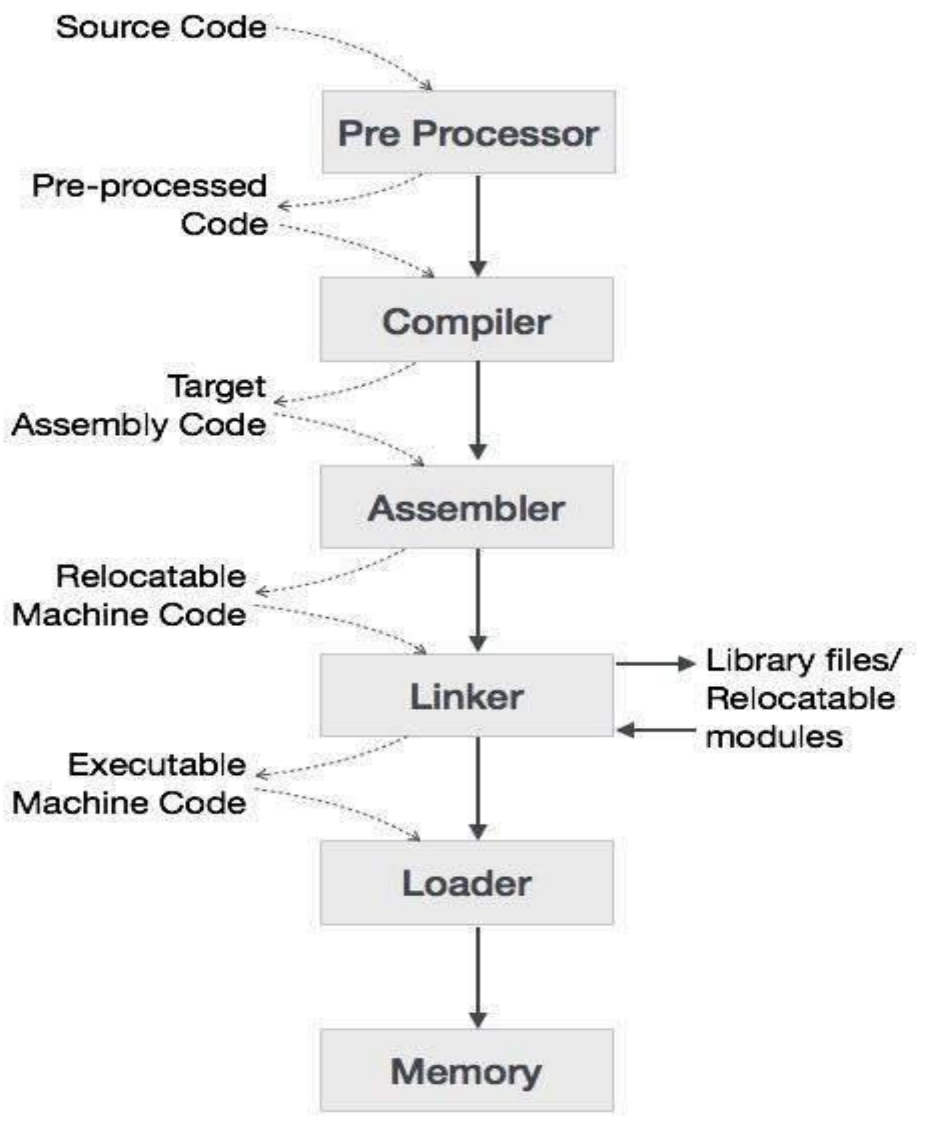
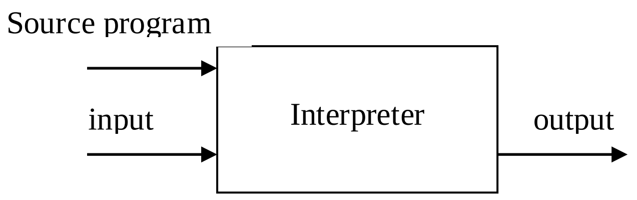


Fig. A language- processing System

INTERPRETOR

An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operation specified in the source program on inputs supplied by the user, see fig. below



An Interpreter

PREPROCESSOR:

A preprocessor is a program that processes its input data to produce output that is used as input to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers.

They may perform the following functions

1. Macro processing
2. File Inclusion
3."Rational Preprocessors
4. Language extension

ASSEMBLER

Assembler translates assembly language program, which uses mnemonic names for operation codes and data addresses into the machine language.

LINKERS AND LOADERS:

- The process of loading consists of taking relocatable machine code, altering the relocatable addresses and placing the altered instructions and data in memory at the proper locations.
- A linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

3.What is a compiler? Explain the various phases of compiler with a neat sketch.(Nov/Dec 2016)

(OR)Explain the structure of the compiler.

(OR)Discuss about the analysis-synthesis model of the compiler with neat diagram.

(OR)Describe the various phases of complier and trace the program segment 4 : ∗ + = c b a for all phases.

(OR)Explain the need for dividing the compilation process into various phases and explain its functions.

[May/June 2013] [May/June 2012] [Apr/May 2011] [Nov/Dec 2011] [May/June 2009] [Nov/Dec 2007] [May/June-14] Or

Describe the various phases of compiler and trace it with the program segment (position := initial + rate*60). (16)

STRUCTURE OF THE COMPILER DESIGN (10 MARKS):-

Compiler Definition

Compiler is a translator program that **translates** a program written in **(HLL)** the source program and translates it into an equivalent program in **(MLL)** the target program. As an important part of a compiler is error showing to the programmer

Compiler operates in phases, each of which transforms the source program from one representation into another. The following are the phases of the compiler:

Main phases:

1) Lexical analysis

2) Syntax analysis

3) Semantic analysis

4) Intermediate code generation

5) Code optimization

6) Code generation

Sub-Phases:

1) Symbol table management

2) Error handling

A pass reads the source program or the output of the previous pass, makes the transformations specified by its phases and writes output into an intermediate file, which may then be read by a subsequent pass. In an implementation of a compiler, portions of one or more phases are combined into a module called pass.

<u>**ANALYSIS PHASE**</u>

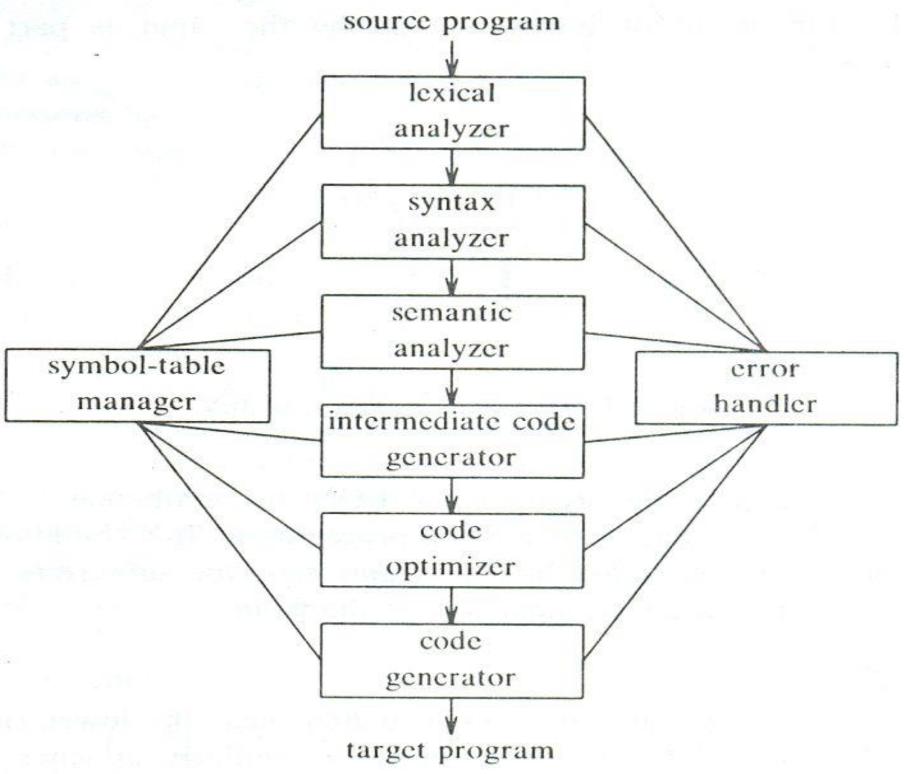


Fig. 1.9. Phases of a compiler.

LEXICAL ANALYSIS:

- It is the **first phase** of the compiler. It gets input from the source program and **produces tokens as output.**

- It reads the characters one by one, starting from left to right and forms the tokens.
- **Token** : It represents a logically cohesive sequence of characters such as keywords, operators, identifiers, special symbols etc.

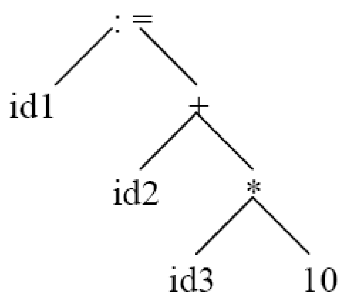
Consider the statement,

position := initial + rate * 10

- Group of characters forming a token is called the **Lexeme**.
- The representation of the statement given above after the lexical analysis would be:
    - **id1,: =, id2, + ,id3 ,* ,10**
- The lexical analyzer not only generates a token but also enters the lexeme into the symbol table if it is not already there.

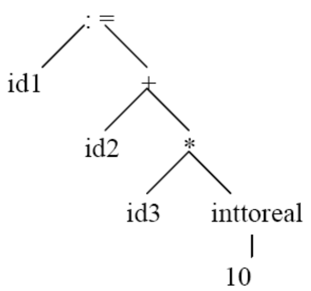
SYNTAX ANALYSIS:

- It is the **Second phase** of the compiler.
- The tokens from the lexical analyzer are grouped hierarchically into nested collections with collective meaning called **"Parse Tree"** followed by syntax tree as output.
- A Syntax Tree is a compressed representation of the parse tree in which the operators appears as interior nodes & the operands as child nodes.



SEMANTIC ANALYSIS:

- It is the **third phase** of the compiler.
- An important part of semantic analysis is **type checking**, where the compiler checks that each operator has matching operands.
- For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may **convert the integer into a floating-point number**.



INTERMEDIATE CODE GENERATION:

- It is the **fourth phase** of the compiler.
- It gets input from the semantic analysis and converts the input into **output as intermediate code** such as three-address code, postfix notation etc.
- The three-address code consists of a sequence of instructions, each of which has atmost three operands.
- **Example: t1=t2+t3**
- The intermediate representation should have two important properties:
  - it should be easy to produce and
  - it should be easy to translate into the target machine
- We consider an intermediate form called three-address code, which consists of a sequence of assembly like instructions with three operands per instruction.

Properties of three-address instructions:

1. Each three-address assignment instruction has at most one operator on the right side.
2. The compiler must generate a temporary name to hold the value computed by a three-address instruction.
3. Some "three-address instructions may have fewer than three operands

In three-address code, the source program might look like this,

temp1: = inttoreal (10)

temp2: = id3 * temp1

temp3: = id2 + temp2

id1: = temp3

<u>THE SYNTHESIS PHASE</u>

CODE OPTIMIZATION:

- It is the **fifth phase** of the compiler.
- It gets the intermediate code as input and produces optimized intermediate code as output.
- This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.
- During the code optimization, the result of the program is not affected.
- To improve the code generation, the optimization involves
  - deduction and removal of dead code (unreachable code).
  - calculation of constants in expressions and terms.
  - collapsing of repeated expression into temporary string.
  - loop unrolling.
  - moving code outside the loop.

- removal of unwanted temporary variables.

The output will look like this:

temp1: = id3 * 10.0

id1: = id2 + temp1

CODE GENERATION:

- It is the **final phase** of the compiler.
- The code generator takes as input an intermediate representation of the source program and maps it into the target language
- Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- The code generation involves
  - allocation of register and memory
  - generation of correct references
  - generation of correct data types
  - generation of missing code

If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.

Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

MOVF id3, R2

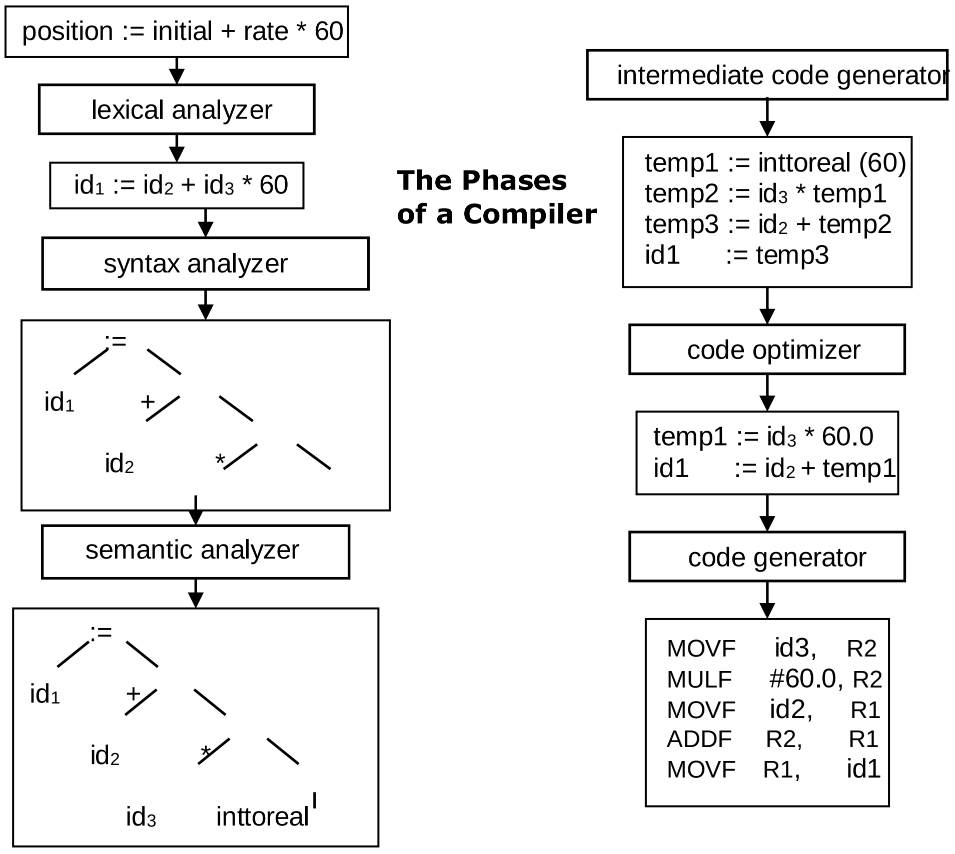
MULF #10.0, R2

MOVF id2 , R1

ADDF R2, R1

MOVF R1, id1

The figure shows the representation of this statement after each phase:



SYMBOL TABLE MANAGEMENT:

- Symbol table is used **to store all the information about identifiers** used in the program.
- It is a data structure containing a **record for each identifier**, with fields for the attributes of the identifier.
- It allows finding the record for each identifier quickly and to store or retrieve data from that record.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

ERROR HANDLING:

- Each phase can encounter errors. After **detecting an error**, a phase must handle the error so that compilation can proceed.
- In lexical analysis, errors occur in separation of tokens.
- In syntax analysis, errors occur during construction of syntax tree.
- In semantic analysis, errors occur when the compiler detects constructs with right syntactic structure but no meaning and during type conversion.
- In code optimization, errors occur when the result is affected by the optimization.
- In code generation, it shows error when code is missing etc.

- To illustrate the translation of source code through each phase, consider the statement Position : = initial + rate * 10.

3. Explain briefly about Errors Encountered in Different Phases.

Or explain various Errors encountered in different phases of compiler. (4) (8) May/June2016 (Nov/Dec 2016)

A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. A program may have the following kinds of errors at various stages:

- **Lexical** : name of some identifier typed incorrectly
- **Syntactical** : missing semicolon or unbalanced parenthesis
- **Semantical** : incompatible value assignment
- **Logical** : code not reachable, infinite loop

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

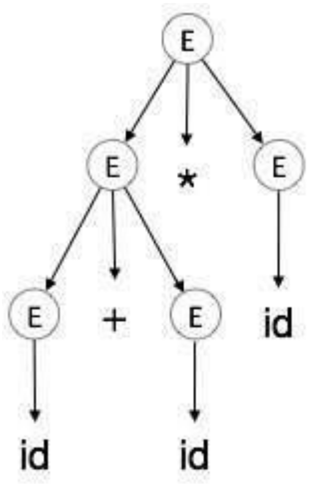
Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

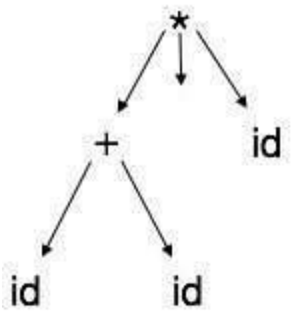
The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Abstract Syntax Trees

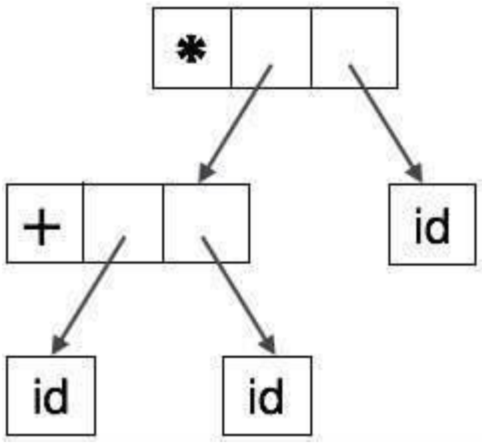
Parse tree representations are not easy to be parsed by the compiler, as they contain more details than actually needed. Take the following parse tree as an example:



If watched closely, we find most of the leaf nodes are single child to their parent nodes. This information can be eliminated before feeding it to the next phase. By hiding extra information, we can obtain a tree as shown below:



Abstract tree can be represented as:



ASTs are important data structures in a compiler with least unnecessary information. ASTs are more compact than a parse tree and can be easily used by a compiler.

5. **Write note on front and back end of compiler. Or**

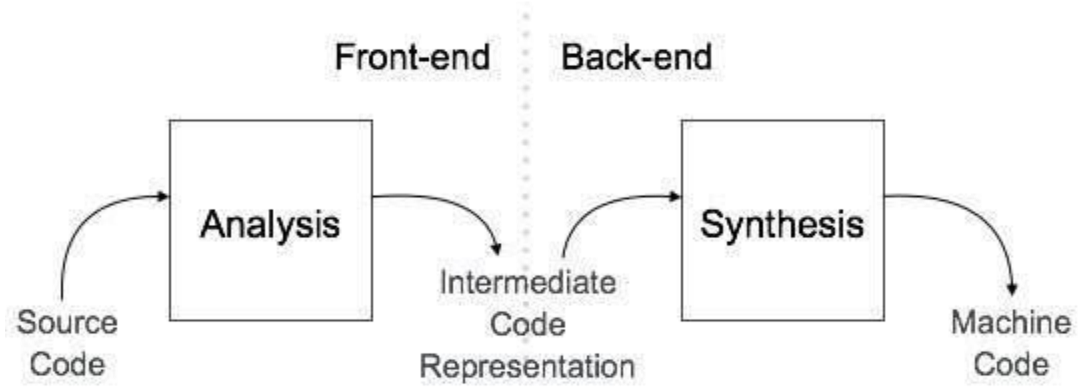
Explain the need for grouping of phases. (4) (8) (May/June 2016) (Nov/Dec 2016)

GROUPING OF PHASES

Definition:

Activities from more than one phase are often grouped together. The phases are collected into a front end and a back end

- **Front End**
- **Back End**



Front End: analysis (machine independent)

• The Front End consists of those phases or parts of phases that depends primarily on the source language and is largely **independent of target machine**.

• Lexical and syntactic analysis, symbol table, semantic analysis and the generation of intermediate code is included.

• Certain amount of code optimization can be done by the front end.

• It also includes error handling that goes along with each of these phases.

Back End: synthesis (machine dependent)

• The Back End includes those portions of the compiler that depend on the target machine and these portions **do not depend on the source language**.

• Find the aspects of code optimization phase, code generation along with necessary error handling and symbol table operations.

Compiler passes

- **Passes:** Several phases of compilation are usually implemented in a single pass consisting of reading an input file and writing an output file.
    - ❖ **Single pass**: usually requires everything to be defined before being used in source program.
    - ❖ **Multi pass**: compiler may have to keep entire program representation in memory.
- It is common for several phases to be grouped into one pass, and for the activity of these phases to be interleaved during the pass.

- Eg: Lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass. If so, the token stream after lexical analysis may be translated directly into intermediate code.
- **Reducing the number of passes:**
  - It is desirable to have relatively few passes, since it takes time to read and write intermediate files.
  - If we group several phases into one pass, we may forced to keep the entire program in memory, because one phase may need information in a different order than a previous phase produces it.

6. Explain construction tools in compiler. [May/June 2012] [Apr/May 2011] [Nov/Dec 2011][Nov/Dec2016]

COMPILER CONSTRUCTION TOOLS

Definition:

These tools have been developed for helping implement various phases of a compiler. These systems have often been referred to as compiler-compilers, compiler- generators or translator writing systems.

Some commonly used compiler-construction tools include

I. Parser generator
II. Scanner generator
III. Syntax-directed translation engine
IV. Data flow engine
V. Automatic code generator

I) Scanner generators:

$$\text{Input(Regular Expression)} \xrightarrow{\text{Generates}} \text{Output(Lexical Analyzer)}$$

- Automatically generates lexical analyzers from a specification based on regular expression.
- The basic organization of the resulting lexical analyzers is finite automation.

II) Parser generators:

$$\text{Input(Context Free Grammar)} \xrightarrow{\text{Generates}} \text{Output(Syntax Analyzer)}$$

- Produce syntax analyzers from input that is based on context-free grammar.
- Many parser generators utilize powerful parsing algorithms that are too complex to be carried out

by hand.

- It consumes a large fraction of the running time of a compiler.

  Example-YACC (Yet Another Compiler-Compiler).

III) Syntax-directed translation engines:

Generates

Input(Parse or Syntax Tree)--------------------- Output(Intermediate code)

- Produce collections of routines that walk a parse tree and generating intermediate code.
- The basic idea is that one or more "translations" are associated with each node of the parse tree.
- Each translation is defined in terms of translations at its neighbor nodes in the tree.

IV) Data-flow analysis engines:

Generates

Input(Intermediate code)--------------------- Output(Optimized code)

- Gathering of information about how values are transmitted from one part of a program to each other part.
- Data-flow analysis is a key part of code optimization from Intermediate codes.

V) Automatic Coder generators:

Generates

Input(Optimized code)--------------------- Output(Object Code)

- A tool takes a collection of rules that define the translation of each operation of the optimized language into the machine language for a target machine.
- The rules must include sufficient details that we can handle the different possible access methods for data.
- Eg. Variables may be in registers, in a fixed location in memory or may be allocated a position on a stack.
- The basic technique is "template matching". The intermediate code statements are replaced by templates.
- That template represents sequences of machine instructions.
- The assumptions about storage of variables match from template to template.

7.Explain briefly about Programming Language Basics

1. The Static/Dynamic Distinction

Among the most important issues that we face when designing a compiler for a language is what decisions can the compiler make about a program. If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a static policy or that the issue can be decided at compile time. On the other hand, a policy that only allows a decision to be made when we execute the program is said to be a dynamic policy or to require a decision at run time.
One issue on which we shall concentrate is the scope of declarations. The scope of a declaration of x is the region of the program in which uses of x refer to this declaration. A language uses static scope or lexical scope if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses dynamic scope. With dynamic scope, as the program runs, the same use of x could refer to any of several different declarations of x.
Most languages, such as C and Java, use static scope.

Example 1: As another example of the static/dynamic distinction, consider the use of the term "static" as it applies to data in a Java class declaration. In Java, a variable is a name for a location in memory used to hold a data value. Here, "static" refers not to the scope of the variable, but rather to the ability of the compiler to determine the location in memory where the declared variable can be found. A declaration like

public static int x ;

makes x a class variable and says that there is only one copy of x, no matter how many objects of this class are created. Moreover, the compiler can determine a location in memory where this integer x will be held. In contrast, had "static" been omitted from this declaration, then each object of the class would have its own location where x would be held, and the compiler could not determine all these places in advance of running the program

2. Environments and States

Another important distinction we must make when discussing programming languages is whether changes occurring as the program runs affect the values of data elements or affect the interpretation of names for that data. For example, the execution of an assignment such as x = y + 1 changes the value denoted by the name x. More specifically, the assignment changes the value in whatever location is denoted by x .

It may be less clear that the location denoted by x can change at run time. For instance, as we discussed in Example 1 .3, if x is not a static (or "class") variable, then every object of the class has its own location for an instance of variable x. In that case, the assignment to x can change any of those "instance" variables, depending on the object to which a method containing that assignment is applied.

$$\begin{tikzcd}\text{names} \arrow[r, bend left, "\text{environment}"] & \text{Locations (variables)} \arrow[r, bend left, "\text{state}"] & \text{values}\end{tikzcd}$$

Figure 1 .8: Two-stage mapping from names to values

The association of names with locations in memory (the store) and then with values can be described by two mappings that change as the program runs (see Fig. 1 .8) :

1. The environment is a mapping from names to locations in the store. Since variables refer to locations ("I-values" in the terminology of C) , we could alternatively define an environment as a mapping from names to variables.

2. The state is a mapping from locations in store to their values. That is, the state maps I-values to their corresponding r-values, in the terminology of C.

Environments change according t o the scope rules of a language.

3. **Static Scope and Block Structure**

Most languages, including C and its family, use static scope. The scope rules for C are based on program structure; the scope of a declaration is determined implicitly by where the declaration appears in the program. Later languages, such as C++, Java, and C#, also provide explicit control over scopes through the use of keywords like **public, private, and protected.**

In this section we consider static-scope rules for a language with blocks, where a block is a grouping of declarations and statements. C uses braces { and } to delimit a block; the alternative use of begin and end for the same purpose dates back to Algol.

4. **Explicit Access Control**

Classes and structures introduce a new scope for their members. If p is an object of a class with a field (member) x, then the use of x in p.x refers to field x in the class definition. In analogy with block structure, the scope of a member declaration x in a class C extends to any subclass C' , except if C" has a local declaration of the same name x .

Through the use of keywords like public, private, and protected, object oriented languages such as C++ or Java provide explicit control over access to member names in a superclass. These keywords support encapsulation by

restricting access. Thus, private names are purposely given a scope that includes only the method declarations and definitions associated with that class and any "friend" classes (the C++ term). Protected names are accessible to subclasses. Public names are accessible from outside the class.

In C++, a class definition may be separated from the definitions of some or all of its methods. Therefore, a name x associated with the class C may have a region of the code that is outside its scope, followed by another region (a method definition) that is within its scope. In fact, regions inside and outside the scope may alternate, until all the methods have been defined.

5. Dynamic Scope

Technically, any scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes. The term dynamic scope, however, usually refers to the following policy: a use of a name x refers to the declaration of x in the most recently called procedure with such a declaration. Dynamic scoping of this type appears only in special situations. We shall consider two examples of dynamic policies: macro expal1sion in the C preprocessor and method resolution in object-oriented programming.

6. Parameter Passing Mechanisms

All programming languages have a notion of a procedure, but they can differ in how these procedures get their arguments. In this section, we shall consider how the actual parameters (the parameters used in the call of a procedure) are associated with the formal parameters (those used in the procedure definition). Which mechanism is used determines how the calling-sequence code treats parameters. The great majority of languages use either "call-by-value," or "call-by-reference," or both. We shall explain these terms, and another method known as "call-by-name," that is primarily of historical interest .

Call- by-Reference

In call-by-reference, the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter. Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller. Changes to the formal parameter thus appear as changes to the actual parameter.

If the actual parameter is an expression, however, then the expression is evaluated before the call, and its value stored in a location of its own. Changes to the formal parameter change this location, but can have no effect on the data of the caller.

Call-by-reference IS used for "ref" parameters in C++ and is an option in many other languages. It is almost essential when the formal parameter is a large object, array, or structure. The reason is that strict call-by-value requires that the caller copy the entire actual parameter into the space belonging to the corresponding formal parameter. This copying gets expensive when the parameter is large. As we noted when discussing call-by-value, languages such as Java solve the problem of passing arrays, strings, or other objects by copying only a reference to those objects. The effect is that Java behaves as if it used call-by-reference for anything other than a basic type such as an integer or real.

Call-by-Name

A **third mechanism** - call-by-name - was used in the early programming language Algol 60. It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (**with renaming of local names in the called procedure**, to keep them distinct) . When the actual parameter is an expression rather than a variable, some unintuitive behaviors occur, which is one reason this mechanism is not favored today.

7. Aliasing

There is an interesting consequence of call-by-reference parameter passing or its simulation, as in Java, where references to objects are passed by value. It is possible that **two formal parameters can refer to the same location**; such variables are said to be aliases of one another. As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other, as well.

UNIVERSITY QUESTION PAPER

NOV/DEC 2016

PART A

1. What is a symbol table?
2. List the various compiler construction tools.

PART B

1.(a) (i)Explain the phases of compiler with a neat diagram. (10)
   (ii) Write notes on compiler Construction tools. (6)
2.(b) (i)Explain the need for grouping of phases. (8)
   (ii) Explain the various errors encountered in different phases of compiler. (8)

MAY/JUNE-2016

PART A

1. What are the two parts of a compiler? Explain briefly.
2. Illustrate diagrammatically how a language is processed.

PART B

1. (a)Describe the various phases of compiler and trace it with the program segment (position := initial + rate*60). (16)

   Or

   (b) (i) Explain language processing system with neat diagram. (8)
   (ii) Explain the need for grouping of phases. (4)
   (iii) Explain various Error encountered in different phases of compiler. (4)

MAY/JUNE-2015

PART A

1. Describe the error recovery schemes in the lexical phase of a compiler.

PART B

1. Mention any four compiler construction tools with their benefits and drawbacks.

MAY/JUNE-2014

PART A

1.State any two reasons as to why phases of compiler should be grouped. [Q.No.11]

PART B

1.a)i)Define the following terms :compiler, interpretor, Translator and differentiate between them. (6) [Q.No 1]

2.Explain in detail the process of compilation .Illustrate the output of each phase of compilation of the input "a=(b+c)*(b+c)*2".(16) [Q.No.3]

NOV/DEC-2013

PART A

1.How will you group the phases of compiler? [Q.No 11]

PART-B

1.a.(i) Explain the different phases of a compiler in details.(12) [Q.No 5]

MAY/JUNE 2013

PART-B

2.(i)What are the phases of the compiler? Explain the phases in detail. Write down the output of each phases for the expression a: = b + c*50. [Q.No. 3]

MAY/JUNE 2012

PART B

11. (a) (i) What are the various phases of the compiler? Explain each phase in detail. [Q.No. 3]

(ii) Briefly explain the compiler construction tools. [Q.No. 6]

APRIL/MAY 2011

PART A

1. What is an interpreter? [Q.No. 26]

PART B

11. (a) (i) Describe the various phases of complier and trace the program segment 4 : $\ast$ + = c b a for all phases. [Q.No. 3]

(ii) Explain in detail about compiler construction tools. [Q.No. 6]

NOV/DEC 2011

PART B

1. (a) (i) Describe the various phases of compiler and trace it with the program segment (position: = initial + rate * 60). (10) [Q.No. 3]

(ii) State the complier construction tools. Explain them. (6) [Q.No. 6]

<u>MAY/JUNE 2009</u>

PART B

11. (a) (i) Explain the need for dividing the compilation process into various phases and explain its functions. [Q.No. 3]

<u>NOV/DEC 2007</u>

PART-A

1. What are the functions of preprocessor? [Q.No. 27]
2. Define a symbol table. [Q.No. 9]

PART-B

1. What is a compiler? Explain the various phases of compiler in detail, with a neat sketch. [Q.No. 3]