R.M.K
GROUP OF INSTITUTIONS

Please read this disclaimer before proceeding:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.

CS8602
Compiler Design

Department: CSE
Batch/Year: 2020-24 / III

Created by:
Dr.P.EZHUMALAI, Prof & Head/RMDEC
Dr. A. K. JAITHUNBI, Associate Prof/RMDEC
V. SHARMILA , AP/CSE

Date: 12.03.2023

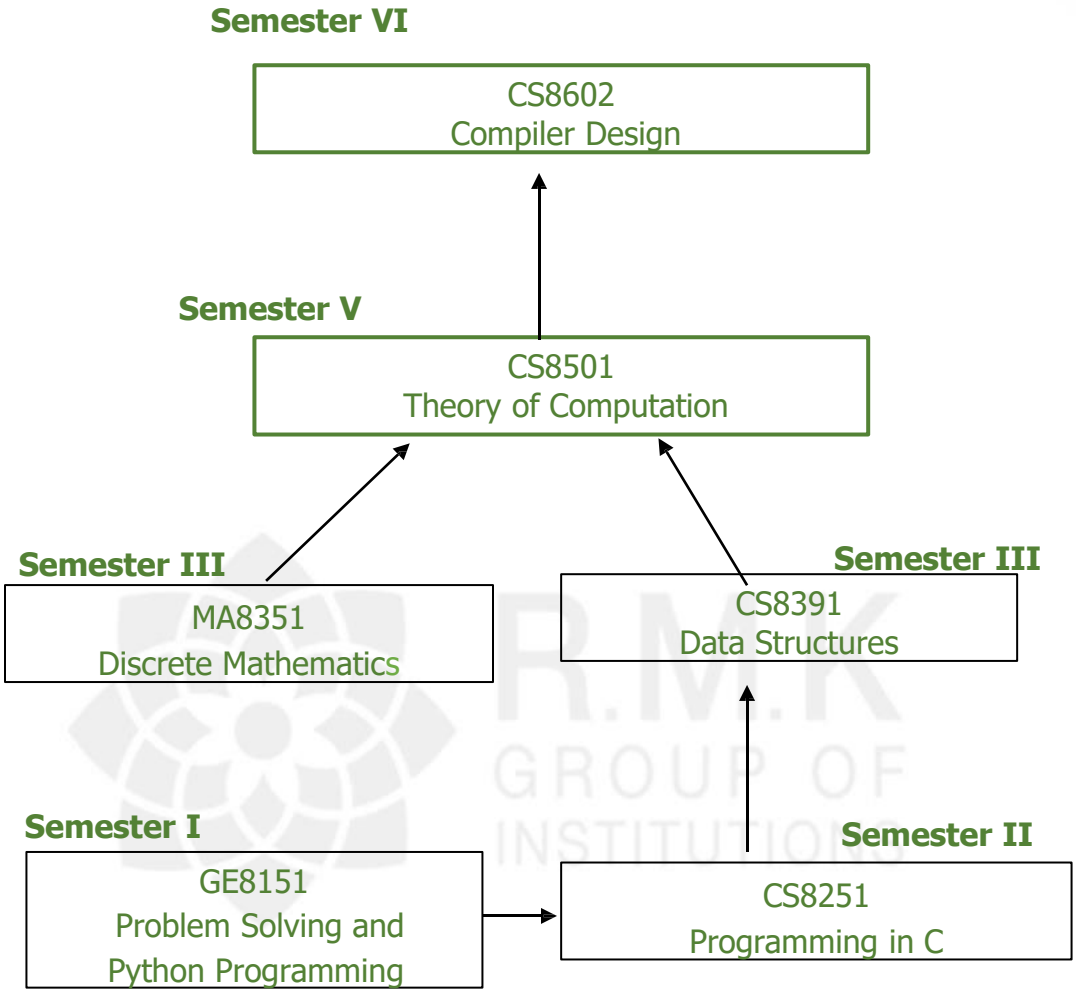
Table of Contents

4

1. COURSE OBJECTIVES

- To learn the various phases of compiler.
- To learn the various parsing techniques.
- To understand intermediate code generation and run-time environment.
- To learn to implement front-end of the compiler.
- To learn to implement code generator.

2. PRE REQUISITES



3. SYLLABUS

CS8602 COMPILER DESIGN

L T P C
3 0 2 4

OBJECTIVES

•To learn the various phases of compiler.

•To learn the various parsing techniques.

•To understand intermediate code generation and run-time environment.

•To learn to implement front-end of the compiler.

•To learn to implement code generator.

UNIT I INTRODUCTION TO COMPILERS 9

Structure of a compiler – Lexical Analysis – Role of Lexical Analyzer – Input Buffering – Specification of Tokens – Recognition of Tokens – Lex – Finite Automata – Regular Expressions to Automata – Minimizing DFA.

UNIT II SYNTAX ANALYSIS 12

Role of Parser – Grammars – Error Handling – Context-free grammars – Writing a grammar – 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.

UNIT III INTERMEDIATE CODE GENERATION 8

Syntax Directed Definitions, Evaluation Orders for Syntax Directed Definitions, Intermediate Languages: Syntax Tree, Three Address Code, Types and Declarations, Translation of Expressions, Type Checking.

UNIT IV RUN-TIME ENVIRONMENT AND CODEGENERATION 8

Storage Organization, Stack Allocation Space, Access to Non-local Data on the Stack, Heap Management - Issues in Code Generation - Design of a simple Code Generator.

UNIT V CODE OPTIMIZATION 8

Principal Sources of Optimization – Peep-hole optimization - DAG- Optimization of Basic Blocks- Global Data Flow Analysis - Efficient Data Flow Algorithm.

Theory - 40 periods + practicals - 35 periods

TOTAL :75 PERIODS

3. Syllabus -contd

LIST OF EXPERIMENTS:

1. Develop a lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments, operators etc.). Create a symbol table, while recognizing identifiers.
2. Implement a Lexical Analyzer using Lex Tool.
3. Implement an Arithmetic Calculator using LEX and YACC.
4. Generate three address code for a simple program using LEX and YACC.
5. Implement simple code optimization techniques (Constant folding, Strength reduction and Algebraic transformation).
6. Implement back-end of the compiler for which the three address code is given as input and the 8086 assembly language code is produced as output.

4. COURSE OUTCOME

Upon completion of the course, the students will be able to:

C313.1 Illustrate a lexical analyzer for a sample language

C313.2 Explain in different parsing algorithms to develop the parsers for a given grammar

C313.3 Understand syntax-directed translation and run-time environment

C313.4 Apply code optimization techniques for programming construct

C313.5 Apply code optimization technique for programming construct

C313.6 Develop a scanner and a parser using LEX and YACC tools

9

5. CO- PO/PSO MAPPING

Course OutComes	Level of CO	Program Outcomes												Program Specific Outcomes		
		K3	K4	K5	K5	K3 K5 K6	A3	A2	A3	A3	A3	A3	A2	PSO1	PSO2	PSO3
		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12			
CO 313.1	K2	2	1	1	1	-	-	-	-	-	-	-	-	2	-	-
CO 313.2	K3	3	2	1	1	-	-	-	-	-	-	-	-	2	-	-
CO 313.3	K3	3	2	1	1	-	-	-	-	-	-	-	-	2	-	-
CO 313.4	K3	3	2	1	1	-	-	-	-	-	-	-	-	2	-	-
CO 313.5	K3	3	2	1	1	-	-	-	-	-	-	-	-	2	-	-
CO 313.6	K3	3	2	1	1	-	-	-	-	-	-	-	-	2	-	-

- **Correlation Level -**
  1. **Slight (Low)**
  2. **Moderate (Medium)**
  3. **Substantial (High)**

If there is no correlation, put "-".

10

UNIT - III

INTERMEDIATE CODE GENERATION

R.M.K GROUP OF INSTITUTIONS

6. LECTURE PLAN

UNIT – III INTERMEDIATE CODE GENERATION

S. No	Proposed Lecture Period	Topic	Actual Lecture Period	Pertaining CO(s)	Highest Cognitive Level	Mode of Delivery	Delivery Resources	After successful completion of the course, the students should be able to (LU Outcomes)	Remarks
1		Syntax Directed Definitions		CO 313.3	K2	MD1	T1	create SDD for various programming stmt	
2		Evaluation Orders for SDD		CO 313.3	K3	MD1 MD5	T1	understand evaluation orders of SDD	
3		Syntax tree construction		CO 313.3	K2	MD1	T1	construct syntax tree	
4		Intermediate Languages		CO 313.3	K3	MD1 MD2	T1	represent in intermediate lang	
5		Three Address Code		CO 313.3	K2	MD1 MD2	T1	create 3-addr code	
6		Types and Declarations		CO 313.3	K3	MD1	T1	write SDD for various data types and declaration stmts	
7		Translation of Expressions		CO 313.3	K2	MD1	T1	understand the translations of expressions	
8		Type Checking		CO 313.3	K2	MD1	T1	perform type checking	

12

7. ACTIVITY BASED LEARNING

1. **Quiz - Exercise to understand the basic definition and concept of CFG:**

https://forms.gle/vgGLZwQR6LbqYhMA8

2. Hands-on Assignment

1. Give annotated parse trees for the following expressions:
   a) (3+4)∗(5+6)n.(3+4)∗(5+6)n.
   b) 1∗2∗3∗(4+5)n.1∗2∗3∗(4+5)n.
   c) (9+8∗(7+6)+5)∗4n.(9+8∗(7+6)+5)∗4n

2. Write semantic rules for the grammar with the attributes of both synthesized as well as inherited to computes 3 * 5 * 7. Give dependency graph in animated form in what order the rules are evaluated?

3. Give three-address code implementation for
   a) a[i] = b*c - b*d
   b) ((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))
   c) a+a+(a+a+a+(a+a+a+a))

4.1. Consider the SDD given below

Production	Semantic rule
A → B C	A.s = B.b; B.i = C.c + A.s

Answer the following?

a) What does this grammar generate?

b) Design L-attributed SDD to compute S.val, the decimal value of an input string.

c) For instance, 101.101 should output 5.625.

Idea: Use an inherited attribute L.side that tells which side (left or right) of the decimal point a bit is on.

13

8. LECTURE NOTES

3.1 Syntax-directed translation

Syntax-directed translation (SDT) refers to a method of compiler implementation where the source language translation is completely driven by the parser. The parsing process and parse trees are used to direct semantic analysis and the translation of the source program. We can augment grammar with information to control the semantic analysis and translation. Such grammars are called attribute grammars.

- Associate attributes with each grammar symbol that describes its properties.
- An attribute has a name and an associated value with each production in a grammar, give semantic rules or actions.
- The general approach to syntax-directed translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order.

There are two ways to represent the semantic rules associated with grammar symbols.

· Syntax-Directed Definitions (SDD)

· Syntax-Directed Translation Schemes (SDT)

· 3.2 **Syntax-Directed Definitions:**

- give high-level specifications for translations
- hide many implementation details such as order of evaluation of semantic actions.
- We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.

· **Translation Schemes:**

- indicate the order of evaluation of semantic actions associated with a production rule.
- In other words, translation schemes give a little bit information about implementation details.

14

- A syntax-directed definition is a generalization of a context-free grammar in which:
  - Each grammar symbol is associated with a set of attributes.
  - This set of attributes for a grammar symbol is partitioned into two subsets called
    - **synthesized** and
    - **inherited** attributes of that grammar symbol.
  - Each production rule is associated with a set of semantic rules.
  - The value of an attribute at a parse tree node is defined by the semantic rule associated with a production at that node.
- The value of a **synthesized attribute** at a node is computed from the values of attributes at the children in that node of the parse tree
- The value of an **inherited attribute** at a node is computed from the values of attributes at the siblings and parent of that node of the parse tree.

3.3 Annotated Parse Tree

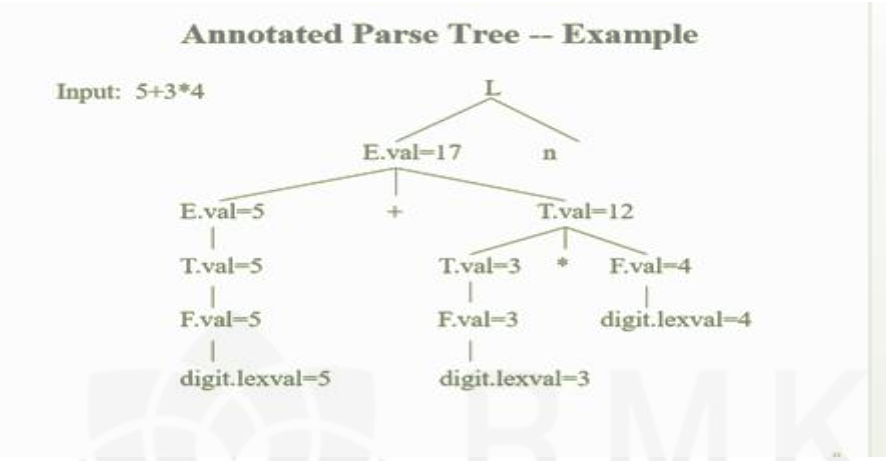
- A parse tree showing the values of attributes at each node is called an annotated parse tree.
- Values of Attributes in nodes of annotated parse-tree are either,
  - initialized to constant values or by the lexical analyzer.
  - determined by the semantic-rules.
- The process of computing the attributes values at the nodes is called annotating (or decorating) of the parse tree.
- Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

Example:

Production	Semantic Rules
$L \rightarrow E\ n$	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	E.val = T.val
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow digit$	F.val = digit.lexval

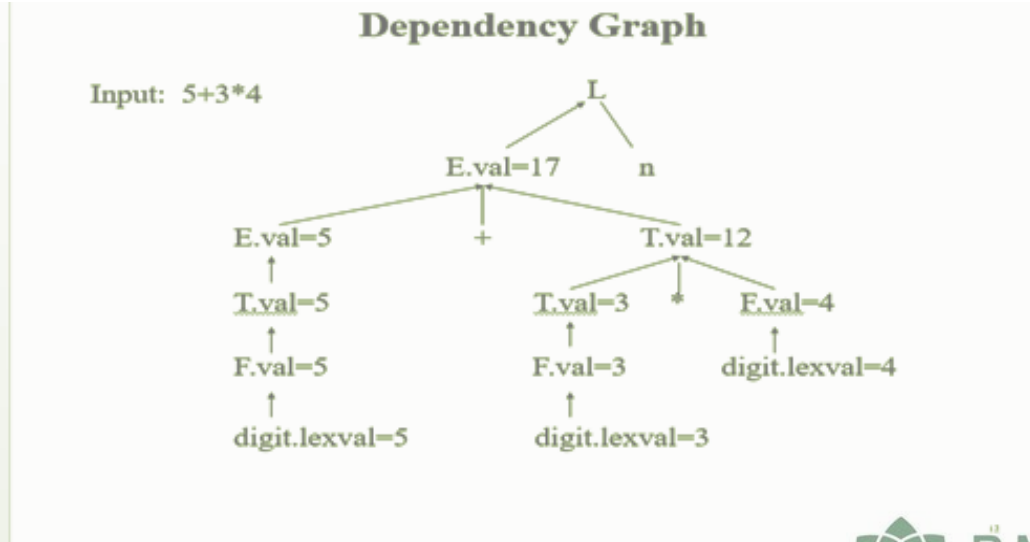
15

1. Symbols E, T, and F are associated with a synthesized attribute *val*.
2. The token digit has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).
3. Terminals are assumed to have synthesized attributes only. Values for attributes of terminals are usually supplied by the lexical analyzer.
4. The start symbol does not have any inherited attribute unless otherwise stated.



3.4 DEPENDENCY GRAPH

- Directed Graph
- Shows interdependencies between attributes.
- If an attribute b at a node depends on an attribute c, then the semantic rule for b at that node must be evaluated after the semantic rule that defines c.
- The graph has a node for each attribute and an edge to the node for b from the node for c if attribute b depends on attribute c.

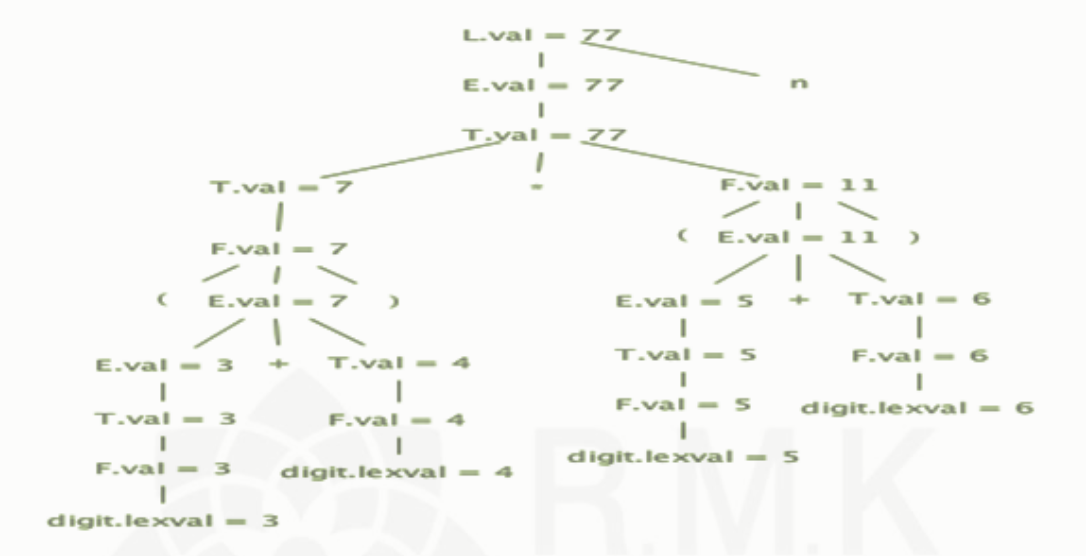


16

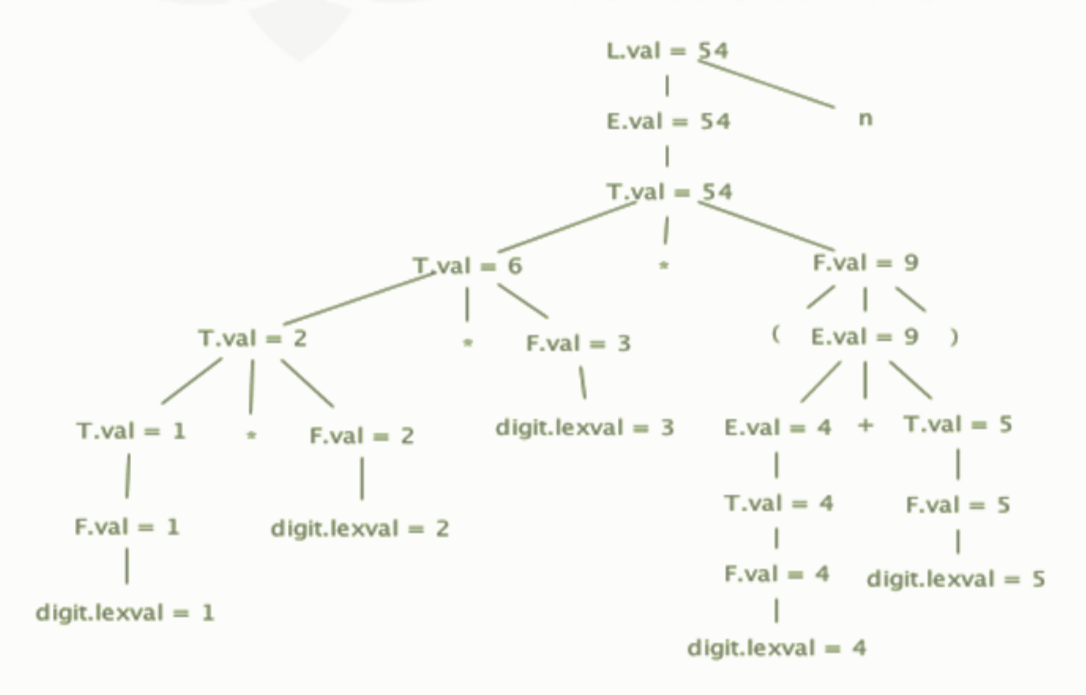
3.5 Evaluating an SDD at the Nodes of a Parse Tree

- A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree.
- With synthesized attributes, evaluate attributes in bottom-up order.

Example 1



Example 2

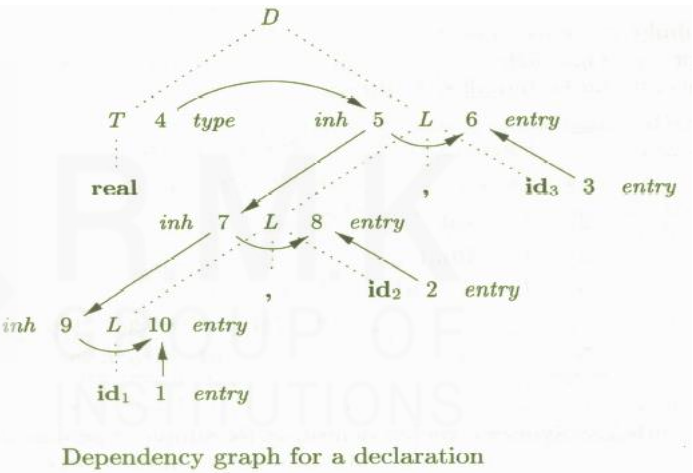


17

3.6 Inherited attributes

- An inherited value at a node in a parse tree is defined in terms of attributes at the parent and/or siblings of the node.
- Convenient way for expressing the dependency of a programming language construct on the context in which it appears.
- We can use inherited attributes to keep track of whether an identifier appears on the left or right side of an assignment to decide whether the address or value of the assignment is needed.
- **Example:** The inherited attribute distributes type information to the various identifiers in a declaration.

$D \rightarrow T\ L$	L.in = T.type
$T \rightarrow int$	T.type = integer
$T \rightarrow real$	T.type = real
$L \rightarrow L_1\ id$	L_1.in = L.in, addtype(id.entry,L.in)
$L \rightarrow id$	addtype(id.entry,L.in)



Dependency graph for a declaration

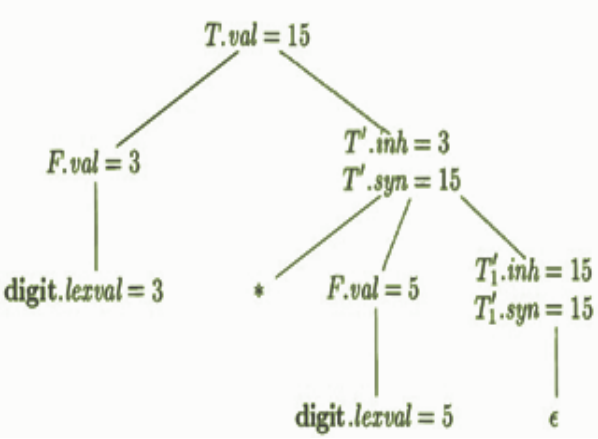
SDD for expression grammar with inherited attributes

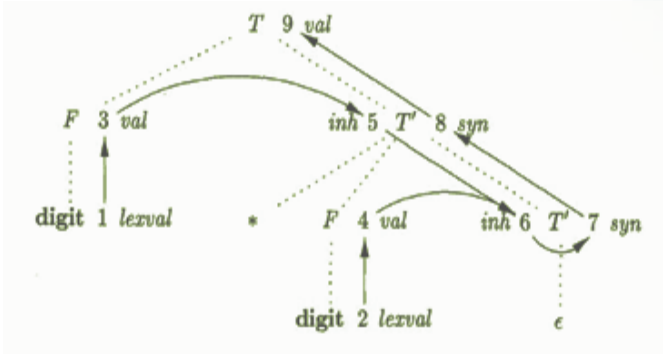
Production	Semantic rules
T-> FT′	T′.inh=F.val T.val=T′.syn
T′->*F T1′	T′1.inch=T′.inh * F.val T′.syn=T′1.syn
T′->ε	T′.syn = T′.inh
F->digit	F.val=digit.lexval

18

Annotated parse tree

Dependency graph

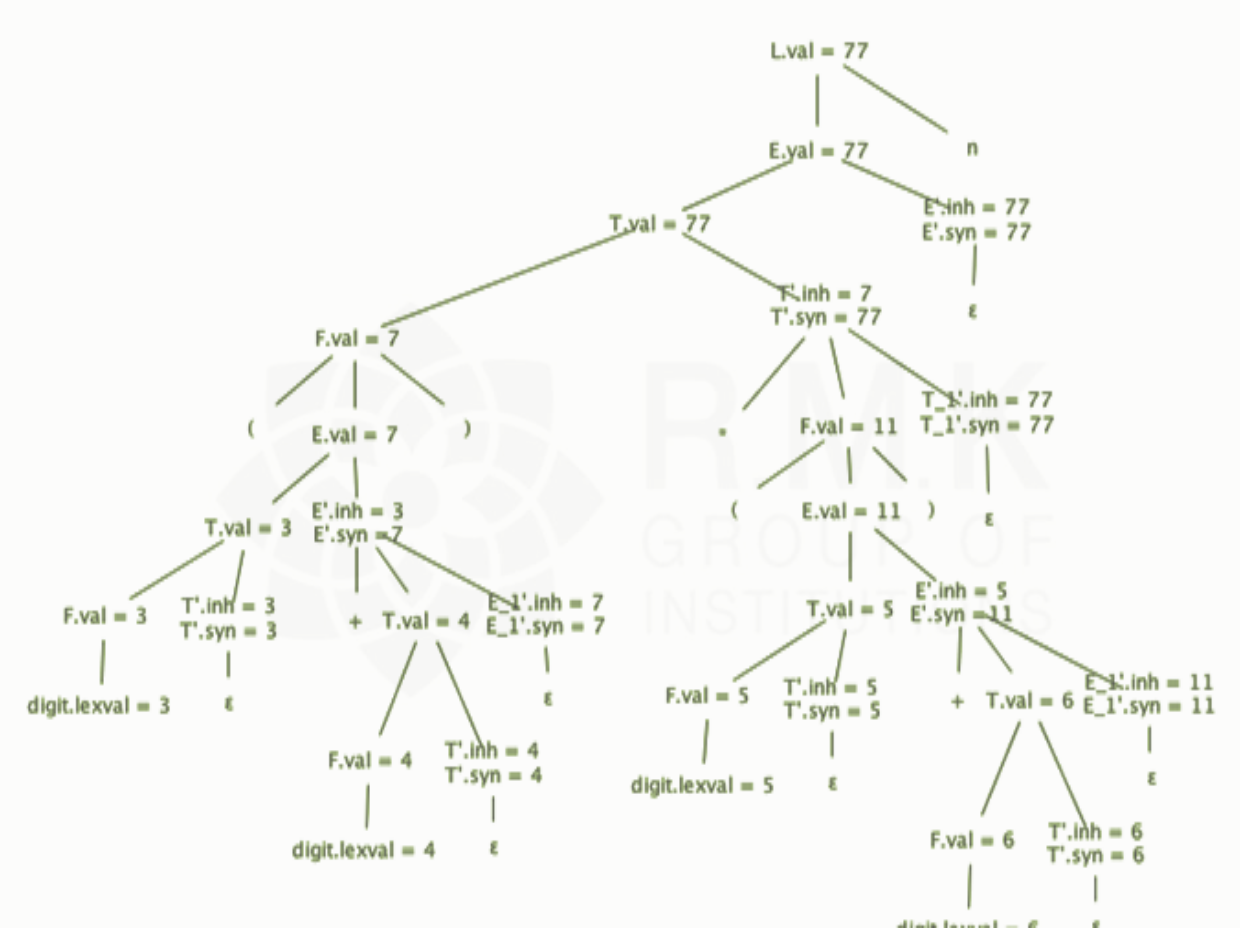




1)	L -> En	L.val = E.val
2)	E -> TE'	E'.inh = T.val E.val = E'.syn
3)	E' -> +TE_1'	E_1'.inh = E'.inh + T.val E'.syn = E_1'.syn
4)	E' -> ε	E'.syn = E'.inh
5)	T -> FT'	T'.inh = F.val T.val = T'.syn
6)	T' -> *FT_1'	T_1'.inh = T'.inh * F.val T'.syn = T_1'.syn
7)	T' -> ε	T'.syn = T'.inh
8)	F -> (E)	F.val = E.val
9)	F -> digit	F.val = digit.lexval

19

Example:1 (3+4)*(5+6)



20

Example:2 **1*2*3*(4+5)**

21

3.7 SDD with circular dependencies no guarantee in the order of evaluation.

e.g.

Production	Semantic Rules
$A \rightarrow B$	$A.s = B.i$
	$B.i = A.s + 1$

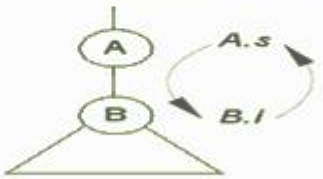


Fig 5.2: The circular dependency of *A.s* and *B.i* on one another

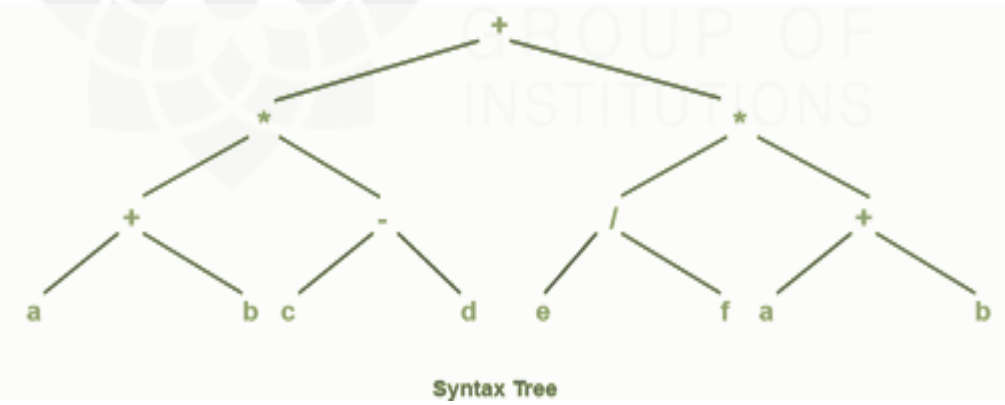
3.8 Evaluation Orders for SDD's

- Dependency graphs is a tool for determining an evaluation order for the attribute instances in a given parse tree.
- Annotated parse tree shows the values of attributes whereas a dependency graph helps to determine how those values can be computed.
- A dependency graph characterizes the possible order in which we can evaluate the attributes at various nodes of a parse tree.
- If there is an edge from node M to N, then attribute corresponding to M first be evaluated before evaluating N.
- Thus the allowable orders of evaluation are N1, N2, ..., Nk such that if there is an edge from Ni to Nj then i < j.
- Such an ordering embeds a directed graph into a linear order, and is called a *topological sort of the graph.*
- If there is any cycle in the graph, then there are no topological sorts

22

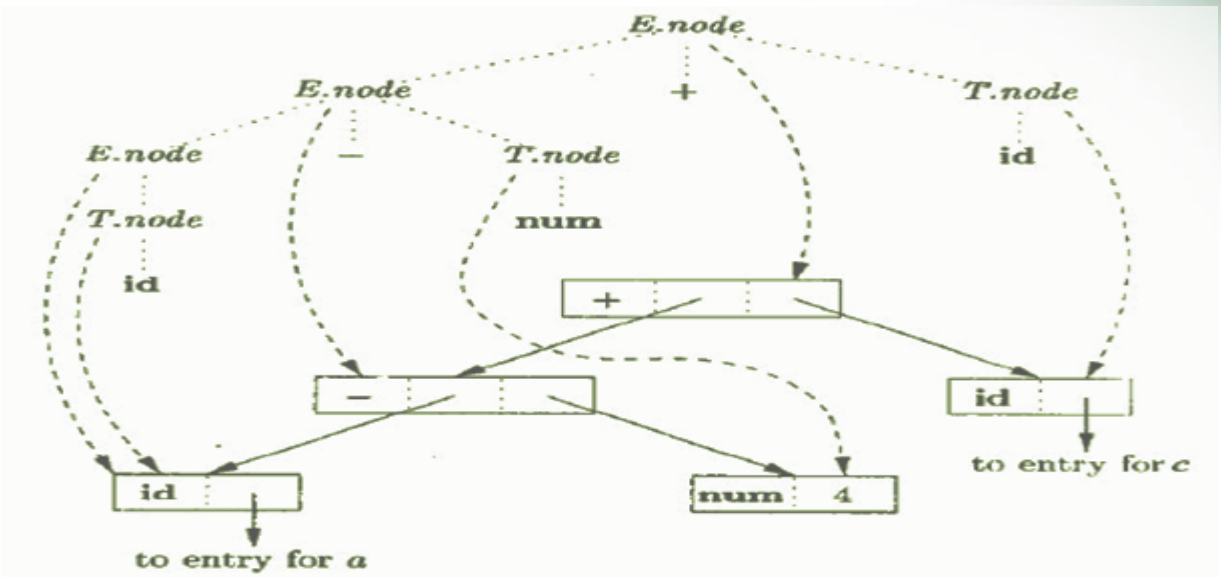
3.9 Intermediate Languages - Syntax trees

- Construction of Syntax Trees
- Syntax trees are useful for representing programming language constructs like expressions and statements.
- Each node of a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.
  - e.g. a syntax-tree node representing an expression E1 + E2 has label + and two children representing the sub expressions E1 and E2
- Each node is implemented by objects with suitable number of fields; each object will have an op field that is the label of the node with additional fields as follows:
- If the node is a leaf, an additional field holds the lexical value for the leaf . This is created by function Leaf(op, val)
- If the node is an interior node, there are as many fields as the node has children in the syntax tree. This is created by function Node(op, c1, 2,...,ck) .



Syntax Tree

23



3.10 Bottom-Up Evaluation of S-Attributed Definitions

- A translator for an S-attributed definition can often be implemented with the help of an LR parser.
- From an S-attributed definition the parser generator can construct a translator that evaluates attributes as it parses the input.
- We put the values of the synthesized attributes of the grammar symbols a stack that has extra fields to hold the values of attributes.
    - The stack is implemented by a pair of arrays *val & state*
    - If the i^{th} state symbol is A the val[i] will hold the value of the attribute associated with the parse tree node corresponding to this A.

Production	Semantic Rules
$L \rightarrow E\ n$	print(val[top-1])
$E \rightarrow E_1 + T$	val[ntop] = val[top-2] + val[top]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	val[ntop] = val[top-2] * val[top]
$T \rightarrow F$	
$F \rightarrow (E)$	val[ntop] = val[top-1]
$F \rightarrow digit$	

1. At each shift of digit, we also push digit.lexval into *val-stack*.
2. 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*).

24

3.11 Intermediate code Representation:

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code.It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code. A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples and triples

General representation –

a = b op c

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator

t1:=b op c

a:=t1

25

Example-1: Convert the expression a * – (b + c) into three address code.

t1=b+c

t2=-(t1)

t3=a*t2

Example-2: Write three address code for following code

```
for(i = 1; i<=10; i++)
{
  a[i] = x * 5;
 }
```

Sol:

i=1

L: t1=x*5

t2=&a

t3=sizeof(int)

t4=t3*i

t5=t2+t4

*t5=t1

i=i+1

If i<=10 goto L

Types of Three-Address Code

▪Assignment statement	x := y op z
▪Assignment statement	x := op y
▪Copy statement	x := y
▪Unconditional jump	goto L
▪Conditional jump	if x relop y goto L
▪Procedural call	param x call p return y

26

Assignment Statement

▪Assignment statements can be in the following two forms

1. x:=op y

2. x:=y op z

First statement op is a unary operation. Essential unary operations are unary minus, logical negation, shift operators and conversion operators.

Second statement op is a binary arithmetic or logical operator

T**hree-Address Statements**

A popular form of intermediate code used in optimizing compilers is three-address statements.

Source statement:

x = a + b* c + d

Three address statements with temporaries t1 and t2:

t1 = b* c

t2 = a + t1

x = t2 + d

Jump Statements

source statement like if-then-else and while-do cause jump in the control flow through three address code so any statement in three address code can be given label to make it the target of a jump.

The statement goto L

Cause an unconditional jump to the statement with label L. the statement

if x relop y goto L

Causes a jump to L condition if and only if

Boolean condition is true.

This instruction applies relational operator relop (>,=,<, etc.)

to x and y, and executes statement L next of x statement x relop y. If not, the three address statement following if x relop y goto L is executed next, as in the usual sequence.

27

Procedure Call / Return

A procedure call like P(A1,A2, A3,…….An) may have to many addresses for one statement in three-address code so it is shown as a sequence of n +1 statements'

Param A1

Param A2

M

Param An

Call p,n

Where P is the name of the procedure and and n is a integer indicating the number of actual parameters in the call.

This information is redundant, as n can be computed by counting the number of par am statements.

It is a convenience to have n available with the call statement.

Indexed Assignment

Indexed assignment of the form A:=B[I] and A[I]:=B.

the first statement sets A to the value in the location I memory units beyond location B.

In the later statement A [I]:=B, sets the location I units beyond A to the value of B.

In Both instructions ,A, B, and I are assumed to refer data objects and will represented by pointers to the symbol table.

Address and Pointer Assignment

Address and pointer assignment x := &y x := *y *x := y

First statement, sets the value of x to be the location of y.

In x := *y, here y is a pointer or temporary whose r-value is a location. The r-value of x is made equal to the contents of that location.

*x := y sets the r-value of the object pointed to by a to the r-value of y.

28

3.12 Implementation of Three Address Code

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple

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

Advantage

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

Disadvantage

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

2. Triples

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

Disadvantage

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

3. Indirect Triples

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

29

Problems: Three address code Representations

1. Translate the following expression to quadruple, triple and indirect triple:

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

Solution:

3-address code:

$_1$ = a + b

t_2 = c + d

t_3 = t_1 * t_2

t_4= a + b

t_5 = t_4 + c

t_6 = t_3 - t_5

Quadruples

	Operation	op1	op2	Result
1	+	a	b	t_1
2	+	c	d	t_2
3	*	t_1	t_2	t_3
4	+	a	b	t_4
5	+	t_4	c	t_5
6	-	t_3	t_5	t_6

30

Quadruple Representation

Location	Op	Arg1	Arg2	Result
(1)	uminus	c		T1
(2)	x	b	T1	T2
(3)	uminus	c		T3
(4)	x	b	T3	T4
(5)	+	T2	T4	T5
(6)	=	T5		a

Triple Representation

Location	Op	Arg1	Arg2
(1)	uminus	c	
(2)	x	b	(1)
(3)	uminus	c	
(4)	x	b	(3)
(5)	+	(2)	(4)

Three address code for various control structures

Problem 1 :

```
fact(x)
{
    int f = 1;
    for (i = 2; i <= x; i++)
        f = f * i;
    return f;
}
```


31

Three Address Code of the above C code:

1. f = 1;
2. i = 2;
3. if (i > x) goto 9
4. t1 = f * i;
5. f = t1;
6. t2 = i + 1;
7. i = t2;
8. goto(3)
9. goto calling program

Problem 2 : Generate three address code for the following code-

```
c = 0
do
{
if (a < b) then
x++;
else
x–;
c++;
} while (c < 5)
```

Solution: Three address code for the given code is

1. c = 0
2. if (a < b) goto (4)
3. goto (7)
4. T1 = x + 1
5. x = T1
6. goto (9)
7. T2 = x – 1
8. x = T2
9. T3 = c + 1
10. c = T3
11. if (c < 5) goto (2)

32

Problem-03: Generate three address code for the following code

```
while (A < C and B > D) do
if A = 1 then C = C + 1
else
while A <= D
do A = A + B
```

Solution: Three address code for the given code is

1. if (A < C) goto (3)
2. goto (15)
3. if (B > D) goto (5)
4. goto (15)
5. if (A = 1) goto (7)
6. goto (10)
7. T1 = c + 1
8. c = T1
9. goto (1)
10. if (A <= D) goto (12)
11. goto (1)
12. T2 = A + B
13. A = T2
14. goto (10)

Problem-04:

Generate three address code for the following code-

```
switch (ch)
{
case 1 : c = a + b;
break;
case 2 : c = a – b;
break;
}
```


33

Solution: Three address code for the given code is

```
 if ch = 1 goto L1
if ch = 2 goto L2
L1:
T1 = a + b
c = T1
goto Last
L2:
T1 = a – b
c = T2
goto Last
Last:
```

Problem-05: Generate three address code for the following for loop: (TA-three address)

Solution:

For loop code	

Solution
FOR LOOP
a=3;
b=4;
for(i=0;i<n;i++)
{
a=b+1;
a=a*a;
}
c=a;
	in 3 TA code
	a=3;
	b=4;
	i=0;
	L1:
	VAR1=i<n;
	if(VAR1) goto L2;
	goto L3;
	L4: i++;
	goto L1;
	L2:
	VAR2=b+1;
	a=VAR2;
	VAR3=a*a;
	a=VAR3;
	goto L4;
	L3:
	c=a;

34

Problem-06: Generate three address code for the following while loop

For loop code	Solution

WHILE LOOP
a=3;
b=4;
for(i=0;i<n;i++)
while(i<n) {
a=b+1;
a=a*a;
i++;
}
c=a;
	in 3 TA code
	a=3;
	b=4;
	i=0;
	L1:
	VAR1=i<n;
	if(VAR1) goto L2;
	goto L3;
	L2:
	VAR2=b+1;
	a=VAR2;
	VAR3=a*a;
	a=VAR3;
	i++;
	goto L1

Problem-07:

Generate three address code for the following do-while loop

For loop code	Solution

DO WHILE LOOP
a=3;
b=4;
i=0;
do
{
a=a+1;
a=a*a;
i++;
} while(i<n);
c=a;
	in 3 TA code
	a=3;
	b=4;
	i=0;
	L1:
	VAR2=b+1;
	a=VAR2;
	VAR3=a*a;
	a=VAR3;
	i++;
	VAR1=i<n;
	if(VAR1) goto L1;
	goto L2;
	L2:
	c=a;

35

3.13 Translation of Expressions

1. Translation of Assignment Statements

In the syntax directed translation, assignment statement is mainly deals with expressions. The expression can be of type real, integer, array and records.

The translation scheme of above grammar is given below:

Consider the grammar

1. S → id := E
2. E → E1 + E2
3. E → E1 * E2
4. E → (E1)
5. E → id

Production rule	Semantic actions

S → id :=E	{p = look_up(id.name);
	If p ≠ nil then
	Emit (p = E.place)
	Else
	Error;
	}

E → E1 + E2	{E.place = newtemp(); Emit (E.place = E1.place '+' E2.place) }
E → E1 * E2	{E.place = newtemp(); Emit (E.place = E1.place '*' E2.place) }

36

E → (E1)	{E.place = E1.place}

E → id	{p = look_up(id.name);
	If p ≠ nil then
	Emit (p = E.place)
	Else
	Error;
	}

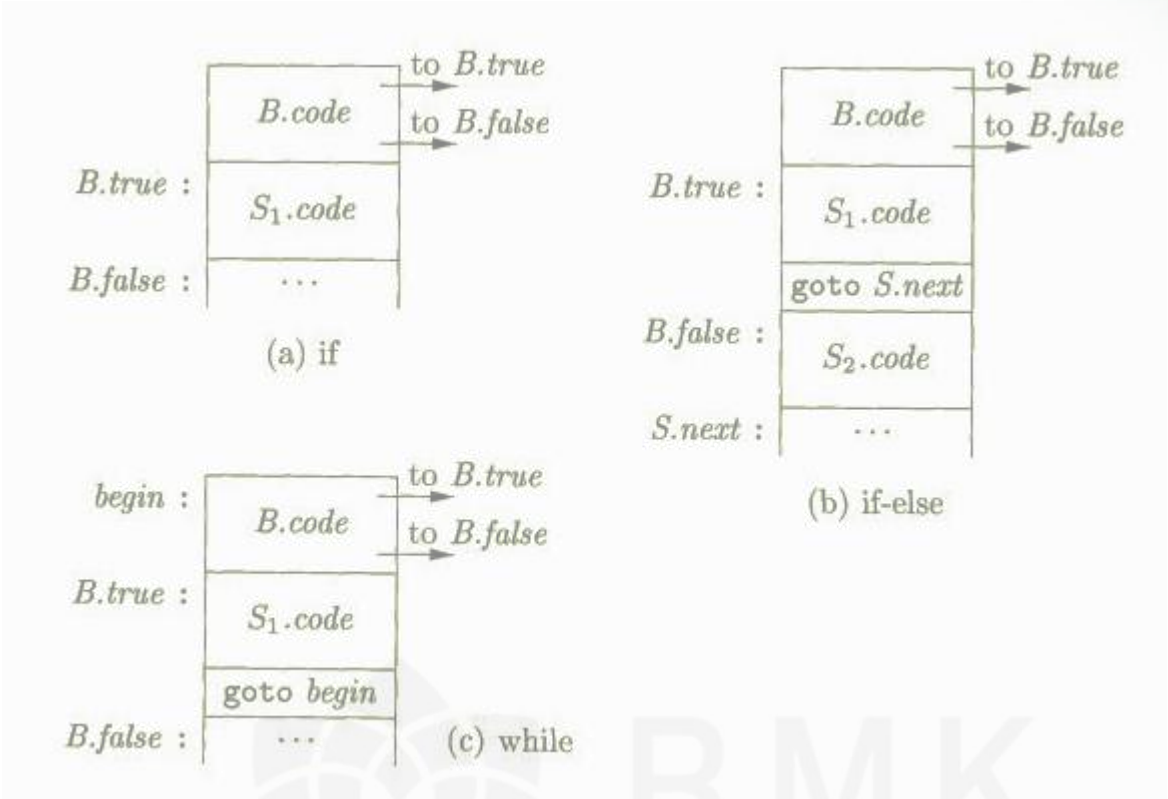
- The p returns the entry for id.name in the symbol table.
- The Emit function is used for appending the three address code to the output file. Otherwise it will report an error.
- The newtemp() is a function used to generate new temporary variables.
- E.place holds the value of E.

37

2. Translation of Control stmts:

Production	semantic rule
S --> if E then S1	E.true := newlabel E.false := S.next S1.next := S.next S.code := E.code || gen(E.true ':') || S1.code
S --> if E then S1 else S2	E.true := newlabel E.false := newlabel S1.next := S.next S2.next := S.next S.code := E.code || gen(E.true ':') || S1.code || gen('goto' S.next) || gen(E.false ':') || S2.code
S --> while E do S1	S.begin := newlabel E.true := newlabel E.false := S.next S1.next := S.begin S.code := gen(S.begin ':') || E.code || gen(E.true ':') || S1.code || gen('goto' S.begin)

38



3. Boolean expressions

Boolean expressions have two primary purposes. They are used for computing the logical values. They are also used as conditional expression using if-then-else or while-do.

Consider the grammar

1. E → E OR E
2. E → E AND E
3. E → NOT E
4. E → (E)
5. E → id relop id
6. E → TRUE
7. E → FALSE

The relop is denoted by <, >, <, >.

The AND and OR are left associated. NOT has the higher precedence then AND and lastly OR

39

Production rule	Semantic actions
E → E1 OR E2	{E.place = newtemp(); Emit (E.place ':=' E1.place 'OR' E2.place) }
E → E1 + E2	{E.place = newtemp(); Emit (E.place ':=' E1.place 'AND' E2.place) }
E → NOT E1	{E.place = newtemp(); Emit (E.place ':=' 'NOT' E1.place) }
E → (E1)	{E.place = E1.place}
E → id relop id2	{E.place = newtemp(); Emit ('if' id1.place relop.op id2.place 'goto' nextstar + 3); EMIT (E.place ':=' '0') EMIT ('goto' nextstat + 2) EMIT (E.place ':=' '1') }
E → TRUE	{E.place := newtemp(); Emit (E.place ':=' '1') }
E → FALSE	{E.place := newtemp(); Emit (E.place ':=' '0') }

40

The EMIT function is used to generate the three address code and the newtemp( ) function is used to generate the temporary variables.

The E → id relop id2 contains the next_state and it gives the index of next three address statements in the output sequence.

Example : **Boolean stmt :** p>q AND r<s OR u>r

3-addr code :

1. 100: if p>q goto 103
2. 101: t1:=0
3. 102: goto 104
4. 103: t1:=1
5. 104: if r>s goto 107
6. 105: t2:=0
7. 106: goto 108
8. 107: t2:=1
9. 108: if u>v goto 111
10. 109: t3:=0
11. 110: goto 112
12. 111: t3:= 1
13. 112: t4:= t1 AND t2
14. 113: t5:= t4 OR t3

41

4. Procedures call

Procedure is an important and frequently used programming construct for a compiler. It is used to generate good code for procedure calls and returns.

Calling sequence:

The translation for a call includes a sequence of actions taken on entry and exit from each procedure. Following actions take place in a calling sequence:

- When a procedure call occurs then space is allocated for activation record.
- Evaluate the argument of the called procedure.
- Establish the environment pointers to enable the called procedure to access data in enclosing blocks.
- Save the state of the calling procedure so that it can resume execution after the call.
- Also save the return address. It is the address of the location to which the called routine must transfer after it is finished.
- Finally generate a jump to the beginning of the code for the called procedure.

Let us consider a grammar for a simple procedure call statement

1. S → call id(Elist)
2. Elist → Elist, E
3. Elist → E

A suitable transition scheme for procedure call would be:

Production Rule	Semantic Action
S → call id(Elist)	for each item p on QUEUE do GEN (param p) GEN (call id.PLACE)
Elist → Elist, E	append E.PLACE to the end of QUEUE
Elist → E	initialize QUEUE to contain only E.PLACE

42

5. Case Statements

Switch and case statement is available in a variety of languages. The syntax of case statement is as follows:

```
1.      switch E
2.              begin
3.                      case V1: S1
4.                      case V2: S2
5.                      .
6.                      .
7.                      .
8.  case Vn-1: Sn-1
9.  default: Sn
10.             end
```

The translation scheme for this shown below:

Code to evaluate E into T

```
1.  goto TEST
2.              L1:     code for S1
3.                      goto NEXT
4.              L2:     code for S2
5.                      goto NEXT
6.                      .
7.                      .
8.                      .
9.              Ln-1:   code for Sn-1
10.                     goto NEXT
11.             Ln:     code for Sn
12. goto NEXT
13.             TEST:   if T = V1 goto L1
14.                     if T = V2goto L2
15.                     .
16.                     .
17.                     .
18.                     if T = Vn-1 goto Ln-1
19.                     goto
20. NEXT:
```

- When switch keyword is seen then a new temporary T and two new labels test and next are generated.
- When the case keyword occurs then for each case keyword, a new label Li is created and entered into the symbol table. The value of Vi of each case constant and a pointer to this symbol-table entry are placed on a stack.

3.14 Type checking

o Type checking is the process of verifying that each operation executed in a program respects the type system of the language.

o This generally means that all operands in any expression are of appropriate types and number.

o Much of what we do in the semantic analysis phase is type checking

Static checking

• Refers to the compile-time checking of programs in order to ensure that the semantic conditions of the language are being followed

Examples of static checks include:

- Type checks
- Flow-of-control checks
- Uniqueness checks
- Name-related checks

Flow-of-control checks:

Statements that cause flow of control to leave a construct must have some place where control can be transferred; e.g., break statements in C.

Uniqueness checks:

A language may dictate that in some contexts, an entity can be defined exactly once; e.g., identifier declarations, labels, values in case expressions.

Name-related checks:

Sometimes the same name must appear two or more times; e.g., in Ada a loop or block can have a name that must then appear both at the beginning and at the end.

Types and Type checking

• A type is a set of values together with a set of operations that can be performed on them

• The purpose of type checking is to verify that operations performed on a value are in fact permissible

44

- The type of an identifier is typically available from declarations, but we may have to keep track of the type of intermediate expressions.

Type Expression and Type constructors

A language usually provides a set of base types that it supports together with ways to construct other types using type constructors.

Type Expressions

• A base type is a type expression

• A type name (e.g., a record name) is a type expression

• A type constructor applied to type expressions is a type expression. E.g., – arrays: If T is a type expression and I is a range of integers, then array(I,T) is a type expression – records: If T1, …, Tn are type expressions and f1, …, fn are field names, then record((f1,T1),…,(fn,Tn)) is a type expression – pointers: If T is a type expression, then pointer(T) is a type expression – functions: If T1, …, Tn, and T are type expressions, then so is (T1,…,Tn) →T.

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.

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.

Type Checking and Type Inference

Type Checking is the process of verifying fully typed programs.

Type Inference is the process of filling in missing type information.

Type Checking Proofs

- Type checking proves facts e: T
  - Proof is on the structure of the AST
  - Proof has the shape of the AST
  - One type rule is used for each kind of AST node
- In the type rule used for a node e:
  - Hypotheses are the proofs of types of e's subexpressions
  - Conclusion is the type of e
- Types are computed in a bottom-up pass over the AST.

Type Checking of Statements

Assignment Semantic Rules: S → Lval := Rval
{check_types(Lval.type,Rval.type)}

- Note that in general Lval can be a variable or it may be a more complicated expression, e.g., a dereferenced pointer, an array element, a record field, etc.
- Type checking involves ensuring that: – Lval is a type that can be assigned to, e.g. it is not a function or a procedure – the types of Lval and Rval are "compatible", i.e, that the language rules provide for coercion of the type of Rval to the type of Lval.

Type Checking of Statements

Loops, Conditionals Semantic Rules:
Loop → while E do S {check_types(E.type,bool)}
Cond → if E then S1 else S2 {check_types(E.type,bool)}

46

3.15 SPECIFICATION OF A SIMPLE TYPE CHECKER

A type checker for a simple language checks the type of each identifier. The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. The type checker can handle arrays, pointers, statements and 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.

.

48

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

2. Conditional statement: S→if E then S1

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 → 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 }

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 }

9. ASSIGNMENTS

S.No	Questions	K Level	CO Level
1.	1.Construct annotated parse tree for a.(9+8*(7+6)+5)*4 b. 4+(3*7)-(5/3+4)+6	K2	CO3
2.	Give Syntax directed translation scheme for Boolean Expression and generate three address code for the following a. Boolean stmt : p>q AND r<s OR u>r b.control stmt: if (a < b) then x++; else x–; c++;	K2	CO3
3.	Consider the regular expression below which can be used as part of a specification of the definition of exponents in floating-point numbers. Assume that the alphabet consists of numeric digits (‘0’ through ‘9’) and alphanumeric characters (‘a’ through ‘z’ and ‘A’ through ‘Z’) with the addition of a selected small set of punctuation and special characters (say in this example only the characters ‘+’ and ‘-‘ are relevant). Also, in this representation of regular expressions the character ‘.’ denotes concatenation. Exponent = (+ | - | ε) . (E | e) . (digit)+	K2	CO3
4.	Consider the Context-Free Grammar (CFG) depicted below where “begin”, “end” and “x” are all terminal symbols of the grammar and Stat is considered the starting symbol for this grammar. Productions are numbered in parenthesis and you can abbreviate “begin” to “b” and “end” to “e” respectively. (1) Stat → Block (2) Block → begin Block end (3) Block → Body (4) Body → x	K2	CO3

50

10 . PART - A Q & A

S.No	Questions	K Level	CO Level
1.	**What are the benefits of intermediate code generation?** A Compiler for different machines can be created by attaching different back end to the existing front ends of each machine. A Compiler for different source languages can be created by proving different front ends for corresponding source languages t existing back end. A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation.	K2	CO3
2.	**What are the various types of intermediate code representation?** There are mainly three types of intermediate code representations. Syntax tree Postfix Three address code	K2	CO3
3.	**Define backpatching.** Backpatching is the activity of filling up unspecified information of labels using appropriate semantic actions in during the code generation process.In the semantic actions the functions used are mklist(i),merge_list(p1,p2) and backpatch(p,i)	K2	CO3
4.	**Mention the functions that are used in backpatching.** mklist(i) creates the new list. The index i is passed as an argument to this function where I is an index to the array of quadruple. merge_list(p1,p2) this function concatenates two lists pointed by p1 and p2. It returns the pointer to the concatenated list. backpatch(p,i) inserts i as target label for the statement pointed by pointer p.	K2	CO3

10. PART - A Q & A

S.No	Questions	K Level	CO Level
5.	**What is the intermediate code representation for the expression a or b and not c?** The intermediate code representation for the expression a or b and not c is the three address sequence t1 := not c t2 := b and t1 t3 := a or t2	K2	CO3
6.	**What are the various methods of implementing three address statements?** The three address statements can be implemented using the following methods. Quadruple : a structure with atmost four fields such as operator(OP),arg1,arg2,result. Triples : the use of temporary variables is avoided by referring the pointers in the symbol table. Indirect triples : the listing of triples has been done and listing pointers are used instead of using statements.	K2	CO3
7.	**Give the syntax-directed definition for if-else statement.** **1. S-> if E then S1** E.true:=new_label() E.false:=S.next S1.next :=S.next S.code:=E.code||gen_code(E.code',')||S1.code **1. S->if Ethen S1 else S2** E.true:=new_label() E.false:=new_label() S1.next:=S.next S2.next :=S.next S.code:=E.code | | gen_code(E.true ': ') | | S1.code| | gen_code('go to',S.next) ||gen_code(E.false ':') | | S2.code	K3	CO3

10 . PART - A Q & A

S.No	Questions	K Level	CO Level
8.	**Define a 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.	K2	CO3
9.	**Define a 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	K2	CO3
10.	**Give the 2 attributes of syntax directed translation into 3-addr code?** i)E.place, the name that will hold the value of E and ii)E.code , the sequence of 3-addr statements evaluating E.	K2	CO3
11.	**What are the advantages of generating an intermediate representation?** i) Ease of conversion from the source program to the intermediate code. ii) Ease with which subsequent processing can be performed from the intermediate code.	K2	CO3
12.	**Define annotated parse tree** A parse tree showing the values of attributes at each node is called an annotated parse tree. The process of computing an attribute values at the nodes is called annotating parse tree. Example: an annotated parse tree for the input 3*5+4n.	K2	CO3

53

10. PART - A Q & A

S.No	Questions	K Level	CO Level
13.	**Define dependency graph.** The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph is called a dependency graph. Example: Production Eà E1 + E2 Semantic Rule E.val:= E1.va; + E2.val	K2	CO3
14.	**What are the functions for constructing syntax tree 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.	K2	CO3
15.	**Define DAG. Give an example.** DAG is a directed acyclic graph for an expression identifies the common sub expression in the expression. Example: DAG for the expression a- 4 *c P1 = mkleaf(id,a) P2 = mknum(num,4) P3 = mkleaf(id,c) P4 = mknode('*',p2,p3) P5 = mknode('-',p1,p4)	K3	CO3
16.	**Define postfix notation?** Postfix notation is a linearized representation of a syntax tree. It is a list of the nodes of the tree in which a node appears immediately after its children. The syntax tree is, a: = b*-c The postfix notation for the syntax tree is, abc-*c	K2	CO3
17.	**Define three address code?** Three address code is a sequence of statements of the form x: = y op z. where x, y, z are names, constants, or compiler generated temporaries, op stand for any type of operator. Since a statement involves not more than three references it is called three-address statement, and hence a sequence of such statement is called three address codes.	K2	CO3

54

10. PART - A Q & A

S.No	Questions	K Level	CO Level
18.	**What are the methods of translating boolean expressions?** There are two principal methods of representing the value of a Boolean expression. a) Encode true and false numerically and to evaluate a Boolean expression analogous to an arithmetic expression. b) Flow-of –control. Represent the value of a Boolean expression by a position reached in a program.	K3	CO3
19.	**What are the two purposes of boolean expressions?** a) They are used to compute logical expressions. b) Often they are used as condition expression in statements that alter the flow of control, such as if-then, if-then-else, or while-do statements.	K2	CO3
20.	**Define quadruple. Give an example.** A quadruple is a record structure with four fields: op, arg1, arg2 and result. The op field contains an internal code for the operator. Example: x: =y op z	K2	CO3
21.	**Give the advantages of quadruple?** i)Can perform peephole optimization. ii) The contents of field's arg1, arg2 and result are normally pointers iii)The symbol-table entries for the names represented by these fields. iv)If So, temporary names must be entered into the symbol table as they Are created.	K2	CO3
22.	**Define triple. Give an example?** Triple is a record structure with three fields: op, arg1 and arg2. The fields arg1 and arg2 are either pointes to the symbol-table or pointers into the triple structure. This method is used to avoid temporary names into the symbol table.	K2	CO3

55

10 . PART - A Q & A

S.No	Questions	K Level	CO Level
23.	**Define indirect triples. Give an example.** Listing pointers to triples rather than listing the triples themselves are called indirect triples. Advantages: it can save some space compared with quadruples, if the same temporary value is used more than once.	K2	CO3
24.	**Define translation scheme?** A translation scheme is a CFG in which program fragments called semantic action are embedded within the right sides of productions. A translation scheme is like a syntax-directed definition, except that the order of evaluation of the semantic rules is explicitly shown.	K2	CO3
25.	**What are the three address code for a or b and not c?** The three address sequence is T1:= not c T2:= b and T1 T3:= a or T2.	K3	CO3
26.	**Write a three address code for the expression a< b or c<d ?** 100: if a<b goto 103 101: t1:=0 102: goto 104 103: t1:=1 104: if c<d goto 107 105: t2:=0 106: goto 108 107: t2:=1 108: t3:=t1 or t2	K3	CO3
27.	**What is the purpose of DAG?** i) A label for each node. For leaves the label is an identifier and for interior nodes, an operator symbol. ii) For each node a list of attached identifiers.	K2	CO3

56

10 . PART - A Q & A

S.No	Questions	K Level	CO Level
28.	**Define short circuit code?** Translate the Boolean expression into three-address code without generating code for any of the Boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes is called short-circuit or jumping code.	K2	CO3
29.	**Give the syntax of case statements?** Switch expression Begin Case value: statement Case value: statement ------- Case value: statement Default : statement End	K3	CO3
30.	**Write three address code for the statements a = b*-c+b*-c ?** Three address codes are: a=b*-c + b*-c T1 = -c T2 = b*T1 T3 = -c T4 = b*T3 T5 = T2+T4 a:= T5.	K3	CO3

11 . PART -B Q & A

S.No	Questions	K Level	CO Level
1.	Evaluate the expressions for the SDD annotated parse tree for the follow expressions. a) 3 * 5 + 4n b) (3 + 4) * (5 + 6)	K3	CO3
2.	(i)Analyze the grammar and syntax-directed translation for desk calculator (ii) Explain how the procedure of constructing syntax tree from SDD	K3	CO3
3.	What is three address codes. Mention the types. How would you implement the three address statement with e.g.	K2	CO3
4.	Explain a type checker which can handle expressions, statements and functions.	K2	CO3
6.	How would you generate the intermediate code for the flow of control statements in detail with example?	K2	CO3
7.	How would you generate the intermediate code for an declaration statements.	K2	CO3
8.	Explain in detail about the specification of a simple type checker.	K2	CO3
9.	Construct a syntax directed definition for constructing a syntax tree for assignment statements S -> id: = E E -> E1 + E2 E -> E1 * E2 E -> - E1 E -> (E1) E -> id	K3	CO3

58

11 . PART -B Q & A

S.No	Questions	K Level	CO Level
10.	Write about Bottom-Up evaluation S-Attributed definitions.	K2	CO3
11.	What is L-attributed definition? Give some example.	K2	CO3
12.	Explain synthesized attribute and inherited attribute with suitable examples.	K2	CO3
13.	Generate an intermediate code for the following code segment with the required syntax-directed translation scheme. if (a > b) x = a + b else x = a – b	K3	CO3
14.	Analyze the grammar and syntax-directed translation fordesk calculator and show the annotated parse tree for expression (3 + 4) * (5 + 6).	K3	CO3

59

11 . PART -C Q & A

S.No	Questions	K Level	CO Level
1.	A syntax-directed translation scheme that takes strs of a's,b's and c's as input and produces as output the number of substs in the input str that correspond to the pattern a(a | b)*c + (a | b)*b. for example the transaltion of the input st "abbcabcababc"is 3". (a).Write a context-free grammar that generate all strs of a's,b's and c's. (b). Apply the attribute definition & Give the semantic attributes for the grammar symbols. (c).For each production of the grammar present a set of rules for evaluation of the semantic attributes.	K3	CO3

2.	Generate an intermediate code for the following code segment with the required syntax-directed translation scheme.
	if (a > b)
	x = a + b
	else
	x = a - b
		K3	CO3

3.	**Apply** the S-attributed definition and constructs syntax trees for a simple expression grammar involve only the binary operators + and -. As usual, these operators are at the same precedence level and are jointly left associative. All nonterminal have one synthesized attribute node, which represents a node of the syntax tree. Production: E→E+ T, E →T, T→ (E), T→ id/ num.	K3	CO3

60

12. Supportive online Certification courses

NPTEL : https://nptel.ac.in/courses/106/105/106105190/

Swayam : https://www.classcentral.com/course/swayam-compiler-design-12926

coursera : https://www.coursera.org/learn/nand2tetris2

Udemy : https://www.udemy.com/course/introduction-to-compiler-construction-and-design/

Mooc : https://www.mooc-list.com/course/compilers-coursera

edx : https://www.edx.org/course/compilers

61

13. Real time Applications in day to day life and to Industry

1. Semantic analysis plays an essential sub-task of Natural Language Processing (NLP) and finds **machine learning applications** like chatbots, search engines, and text analysis.

1. Semantic analysis-driven tools provide the companies automatically extract meaningful information from **unstructured data**, such as emails, support tickets, and customer feedback.

1. Lexical semantics plays an important role in semantic analysis, allowing machines to understand relationships between **lexical items** (words, phrasal verbs, etc.)

1. Semantic analysis also considers **signs and symbols** (semiotics) and collocations (words that often go together).

1. Automated semantic analysis helps with machine learning algorithms which would train machines with samples of text to make **accurate predictions** based on past observations.

1. semantic-based approaches are useful for the task like **word sense disambiguation and relationship extraction**

62

14. Contents beyond the Syllabus

Natural Language Processing - Semantic Analysis

The purpose of semantic analysis is to draw exact meaning, or you can say dictionary meaning from the text. The work of semantic analyzer is to check the text for meaningfulness.

Elements of Semantic Analysis

Followings are some important elements of semantic analysis –

Hyponymy

It may be defined as the relationship between a generic term and instances of that generic term. Here the generic term is called hypernym and its instances are called hyponyms. For example, the word color is hypernym and the color blue, yellow etc. are hyponyms.

Homonymy

It may be defined as the words having same spelling or same form but having different and unrelated meaning. For example, the word "Bat" is a homonymy word because bat can be an implement to hit a ball or bat is a nocturnal flying mammal also.

Polysemy

Polysemy is a Greek word, which means "many signs". It is a word or phrase with different but related sense. In other words, we can say that polysemy has the same spelling but different and related meaning. For example, the word "bank" is a polysemy word having the following meanings –

- A financial institution.
- The building in which such an institution is located.
- A synonym for "to rely on".

Semantic analysis uses the following approaches for the representation of meaning –

- First order predicate logic (FOPL)
- Semantic Nets
- Frames
- Conceptual dependency (CD)

Rule-based architecture

Case Grammar

Conceptual Graphs

Lexical Semantics

The first part of semantic analysis, studying the meaning of individual words is called lexical semantics. It includes words, sub-words, affixes (sub-units), compound words and phrases also. All the words, sub-words, etc. are collectively called lexical items. In other words, we can say that lexical semantics is the relationship between lexical items, meaning of sentences and syntax of sentence.

Following are the steps involved in lexical semantics –

· Classification of lexical items like words, sub-words, affixes, etc. is performed in lexical semantics.

· Decomposition of lexical items like words, sub-words, affixes, etc. is performed in lexical semantics.

· Differences as well as similarities between various lexical semantic structures is also analyzed.

We understand that words have different meanings based on the context of its usage in the sentence. If we talk about human languages, then they are ambiguous too because many words can be interpreted in multiple ways depending upon the context of their occurrence.

Word sense disambiguation, in natural language processing (NLP), may be defined as the ability to determine which meaning of word is activated by the use of word in a particular context. Lexical ambiguity, syntactic or semantic, is one of the very first problem that any NLP system faces. Part-of-speech (POS) taggers with high level of accuracy can solve Word's syntactic ambiguity. On the other hand, the problem of resolving semantic ambiguity is called WSD (word sense disambiguation). Resolving semantic ambiguity is harder than resolving syntactic ambiguity.

For example, consider the two examples of the distinct sense that exist for the word ***"bass"*** –

64

I can hear bass sound.

- He likes to eat grilled bass.

The occurrence of the word **bass** clearly denotes the distinct meaning. In first sentence, it means **frequency** and in second, it means **fish**. Hence, if it would be disambiguated by WSD then the correct meaning to the above sentences can be assigned as follows –

- I can hear bass/frequency sound.
- He likes to eat grilled bass/fish.

Evaluation of WSD

The evaluation of WSD requires the following two inputs –

A Dictionary

The very first input for evaluation of WSD is dictionary, which is used to specify the senses to be disambiguated.

Test Corpus

Another input required by WSD is the high-annotated test corpus that has the target or correct-senses. The test corpora can be of two types &minsu;

- **Lexical sample** – This kind of corpora is used in the system, where it is required to disambiguate a small sample of words.
- **All-words** – This kind of corpora is used in the system, where it is expected to disambiguate all the words in a piece of running text.

65

Contents beyond the Syllabus

Approaches and Methods to Word Sense Disambiguation (WSD)

Approaches and methods to WSD are classified according to the source of knowledge used in word disambiguation.

Let us now see the four conventional methods to WSD –

- **Dictionary-based or Knowledge-based Methods**
- **Supervised Methods**
- **Semi-supervised Methods**
- **Unsupervised Methods**

Applications of Word Sense Disambiguation (WSD)

Word sense disambiguation (WSD) is applied in almost every application of language technology.

Let us now see the scope of WSD –

- **Machine Translation**
- **Information Retrieval (IR)**
- **Text Mining and Information Extraction (IE)**
- **Lexicography**

66

15. ASSESSMENT SCHEDULE

- **Tentative schedule for the Assessment**

S.NO	Name of the Assessment	Start Date	End Date	Portion
1	Unit Test 1	7.2.23	14.2.23	UNIT 1
2	IAT 1	27.2.23	4.3.23	UNIT 1 & 2
3	Unit Test 2	1.4.23	10.4.23	UNIT 3
4	IAT 2	18.4.23	25.4.23	UNIT 3 & 4
5	Revision 1	28.4.23	29.4.23	UNIT 5 , 1 & 2
6	Revision 2	2.5.23	4.5.23	UNIT 3 & 4
7	Model	11.5.23	20.5.23	ALL 5 UNITS

16. Prescribed Textbooks & Reference Books

TEXTBOOK:

Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles, Techniques and Tools‖, Second Edition, Pearson Education, 2009.

REFERENCES:

1. Randy Allen, Ken Kennedy, Optimizing Compilers for Modern Architectures: A Dependence based Approach, Morgan Kaufmann Publishers, 2002.

1. Steven S. Muchnick, Advanced Compiler Design and Implementation‖, Morgan Kaufmann Publishers - Elsevier Science, India, Indian Reprint 2003.

1. Keith D Cooper and Linda Torczon, Engineering a Compiler‖, Morgan Kaufmann Publishers Elsevier Science, 2004.

1. V. Raghavan, Principles of Compiler Design‖, Tata McGraw Hill Education Publishers,2010.

1. Allen I. Holub, Compiler Design in C‖, Prentice-Hall Software Series, 1993.

68

17. Mini Project suggestions

1. Develop a project to **implement the Analysis Phase (Frontend**) for a basic C compiler. Design a module to perform the following tasks

   a) generate lexemes and tokens

   b) check the syntactic structure of the statements

   c) check the semantic rules

   d)generate intermediate code in the form of 3-addr code

1. Implement and design an interpreter to convert source code in C language into Java source code.

1. Write a code to **generate quadruple code** for the given statement .

1. Write a code to **verify the correctness of the code** given below

```
int main( )
{
    string x;
    if (false)
        x = 137;
}
```

5. **Create a C compiler with semantic analyzer** to check the following constraints

   Multiple declarations

   Undeclared variables

   Type mismatch in variables

   Scope of variables

69

Thank you

Disclaimer:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.

70