DATE: / /
PAGE NO.:

UNIT-4

INTERMEDIATE CODE GENERATOR :

ADD R_1, R_2 ADD → Binary operator

ADD R_1, R_2, R_3 → ×

ADD R_1, M_1, M_2 → M2, M3 added and store result in R1

ADD M1, M2, M3

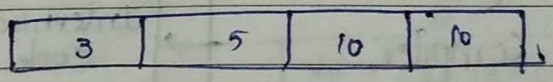
ADD M1, M2

we write ADD RM } use different opcodes for different type of addition operation

ADD M

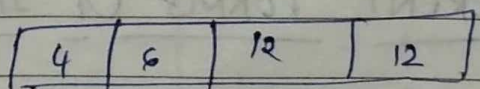
① ADDR
② ADD RM
③ SUB R
④ SUBRM
⑤ MOV
⑥ LOAD
⑦
⑧

A machine architecture containing
8 diff op ⇒ 3 bits
32 diff regist ⇒ 5 bits
1024 diff locations ⇒ 10 bits



Maximum instruction length can be 28 bits.

Another Machine
15 diff operations ⇒ 4 bits
64 diff registers ⇒ 6 bits
4k memory size ⇒ 12 bits



34 bits

Compiler design so depends on architectures

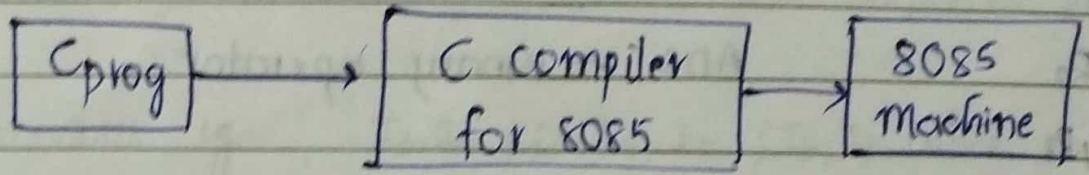
Teacher's Signature

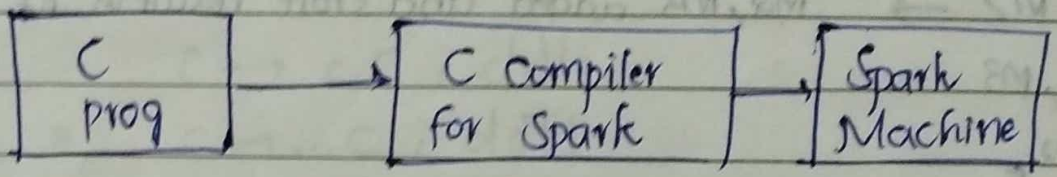
Importance of ICG

DATE: / /
PAGE NO.:

* If for any language, a compiler is designed and if it is to be run on different machines, then the entire compiler has to be reconstructed for the other machine.

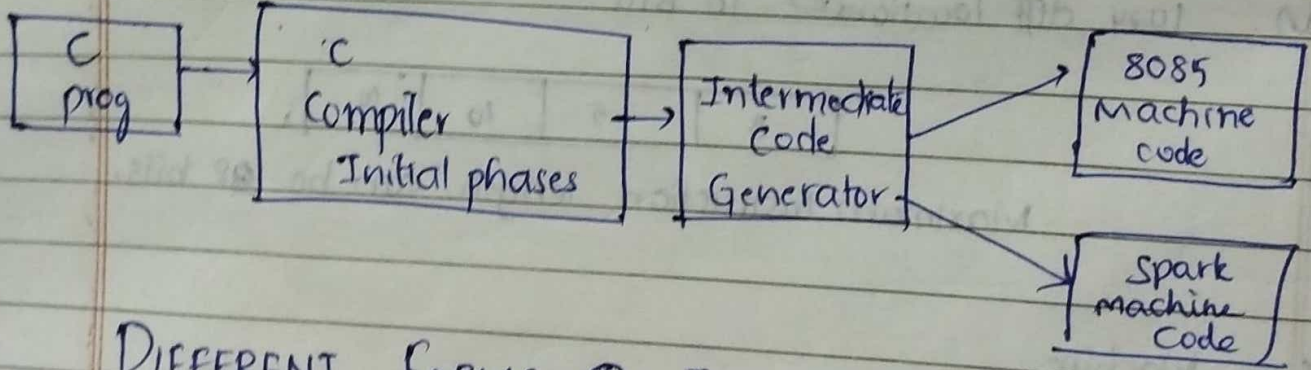
Eg:





* If intermediate code generator is introduced, then, there is no need of re-construction of the compiler with all the phases for a new machine

* The code generated in Intermediate code is suitable for any machine and so, the portability of the compiler is increased.



DIFFERENT FORMS OF INTERMEDIATE CODE GENERATION

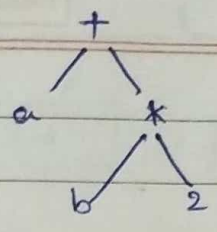
1. Abstract Syntax Tree - similar to parse tree
2. Reverse Polish notation (postfix)
3. Three address code

Eg: a+b * 2

Eg:2 : (a+b) * (c+d)

DATE: / /
PAGE NO.:

parse tree:



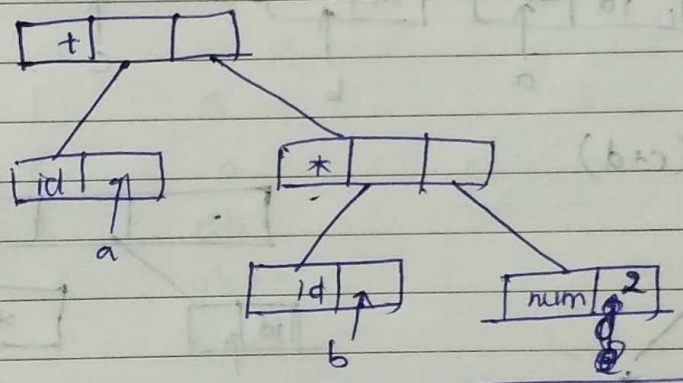
Functions : mknode (op, left, child)

↓ operator ↓ left subtree

mkleaf (id, entry)

or mkleaf (num or const, value)

a+b *2 → binary op



2) postfix expression :

ch	Stack	Postfix exp
a		a
+	+	a
b	+	ab
*	+*	ab*
2	+*	ab*2
	+	ab*2*+

(a+b)*(c+d)

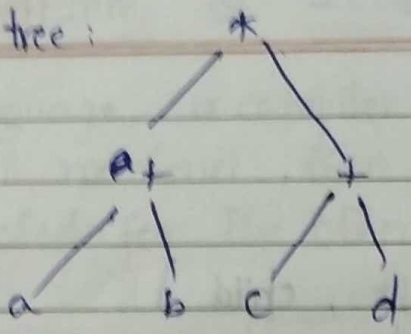
ch	stack	Postfix
(	(	
a	(	a
+	(+	a
b	(+	ab
)		ab+
*	*	ab+
(	*(	ab+
c	*(	ab+c
+	*(+	ab+c
d	*(+	ab+cd
)	*	ab+cd+
		ab+cd+*

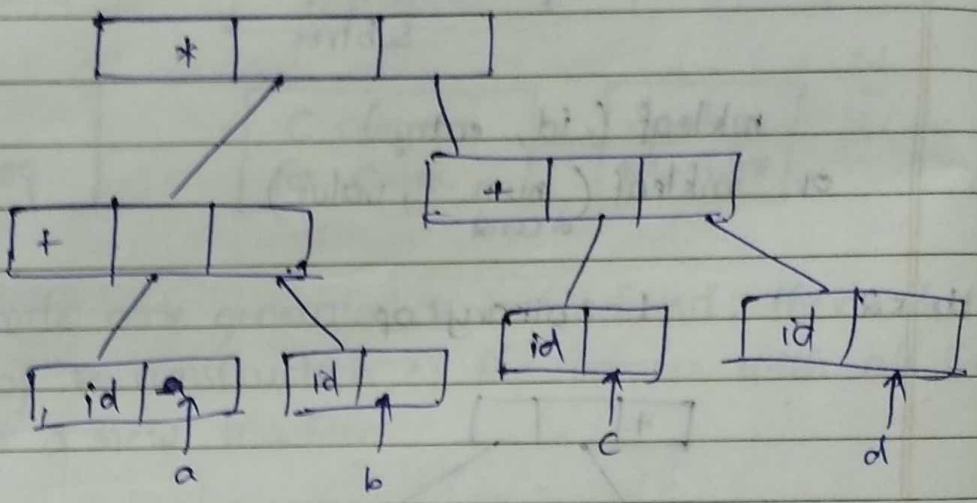
Teacher's Signature

(a+b) * (c+d)

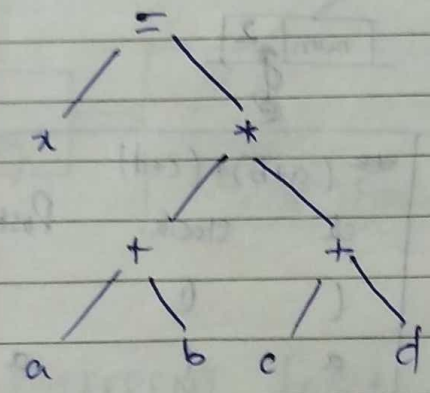
DATE: / /
PAGE NO.:

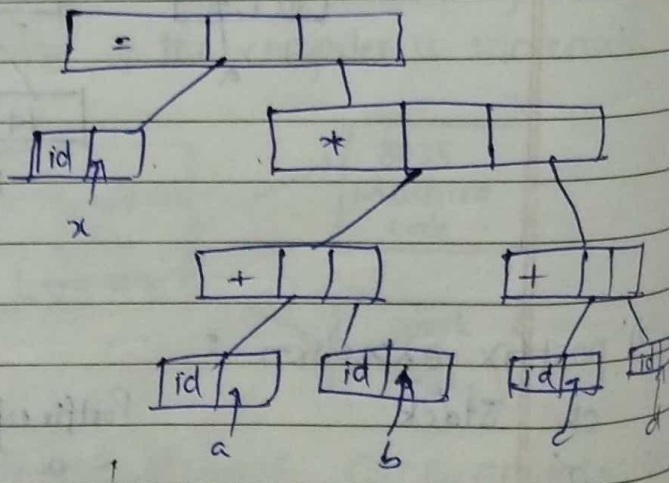
Parse tree:





x = (a+b)*(c+d)





③ x = ((-a) * b) + ((-c)+d)

* Unary has highest priority

④ a = (b+c)*(d/e)

⑤ ~~d = (a/b)~~

⑤ d = (a & b) & c ó ; (a|b)&c

⑥ (a|b)& (!c | !d)

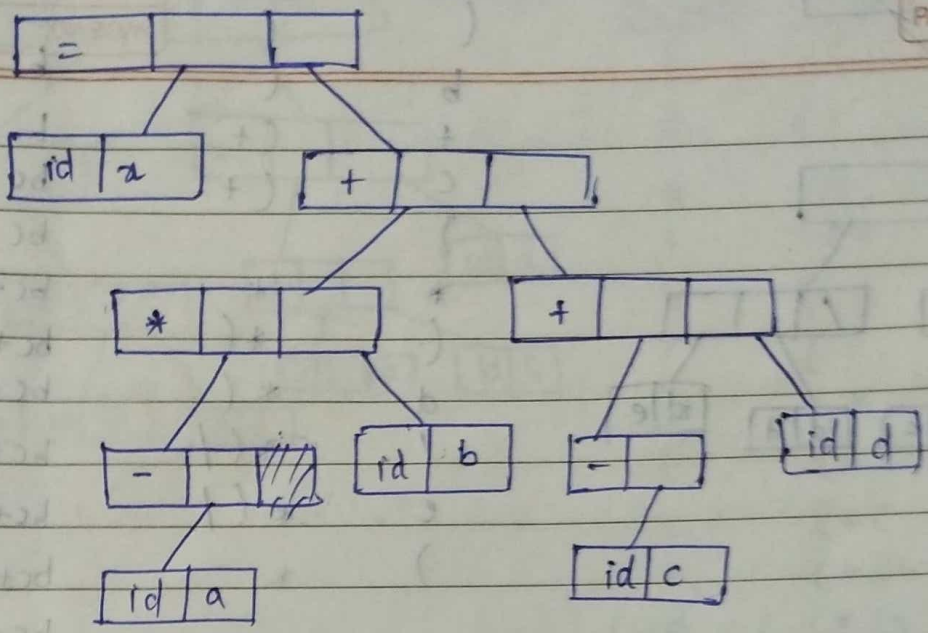
⑦ ((a+b) * (c+d)) - ((a+b)*d)

⑧ x = (a * (b+c+(-a)))

⑨ x = (a*b+c+d) * b-d

Teacher's Signature

3) $x = (-a * b) + (-c * d)$

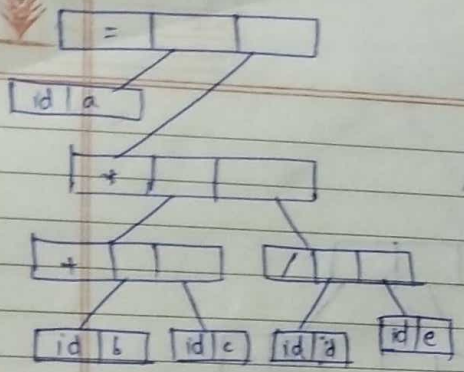


Reverse Polish Notation :

character	stack	expression
(	(	
-	(-	
a	(-	a
*	(- *	a
b	(- *	ab
)	(	ab*-
+	+	ab*-
(	+(	ab*-
-	+(-	ab*-
c	+(-	ab*-c
*	+(-*	ab*-c
d	+(-*	ab*-cd
)	+	ab*-cd*-
		ab*-cd*-+

Teacher's Signature

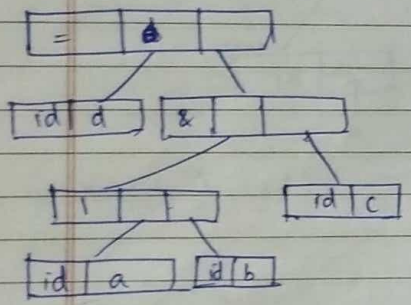
4) a = (b + c) * (d / e)



Reverse polish Notation

Character	Stack	Expression
(	(	
b	(	b
+	(+	b
c	(+	bc
)		bc+
*	*	bc+
(	*(	bc+
d	*(	bc+d
/	*(/	bc+d
e	*(/	bc+de
)	*	bc+de/
		bc+de/*

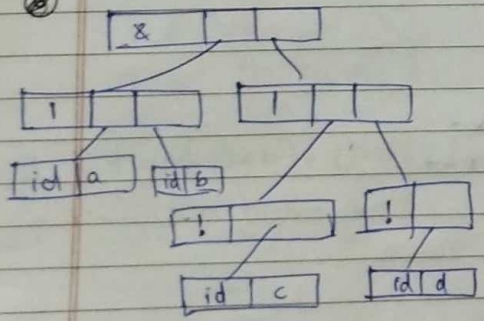
5) d = (a|b) & c



Reverse polish Notation

Character	stack	Expression
(	(	
a	(	a
|	(|	a
b	(|	ab
)		ab|
&	&	ab|
c	&	ab|c
		ab|c&

6) (a|b) & (!c | !d)

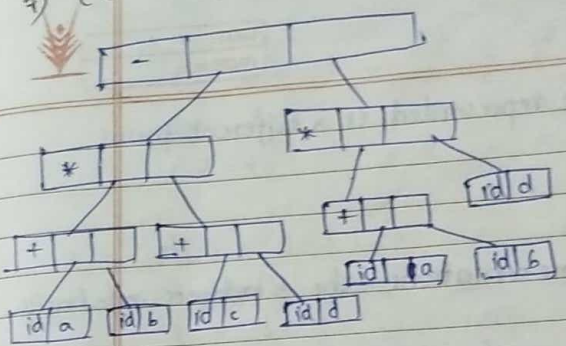


Reverse polish Notation

Character	Stack	Expression
(	(	
a	(	a
|	(|	a
b	(|	ab
)		ab|
&	&	ab|
(	&(	ab|
!	&(!	ab|
c	&(	ab|c!
|	&(|	ab|c!
!	&(|!	ab|c!
d	&(|	ab|c!d!
)	&	ab|c!d!&

Teacher's Signature

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



Reverse Polish Notation

Character	stack	Expression
(	(	
(	((	
a	((	a
+	((+	a
b	((+	ab
)	(	ab+
*	(*	ab+
(	(*(	ab+
c	(*(	ab+c
+	(*(+	ab+c
d	(*(+	ab+cd
)	(*	ab+cd+
)		ab+cd+*
-	-	ab+cd+*
(	-(	ab+cd+*
(	-((	ab+cd+**
a	-((	ab+cd+*a
+	-((+	ab+cd+*a
b	-((+	ab+cd+*ab
)	-(	ab+cd+*ab

Teacher's Signature

12/05/22 Thursday

THREE ADDRESS CODES

DATE: / /
PAGE NO.:

* Three address code can be represented in 3 different forms
  → 1. Quadruples
  → 2. Triples
  → 3. Indirect triples.
* for any of these 3 representations, the 3 address code has to be generated initially
* In the 3 address code, each statement contains atmost 3 address locations.

$x = (a+b)*(c+d)$

TAC

$t_1 = a+b$
$t_2 = c+d$
$t_3 = t_1 * t_2$
$x = t_3$

Quadruple Representation (4 fields)

S.no	op	arg1	arg2	result
0	+	a	b	t_1
1	+	c	d	t_2
2	*	t_1	t_2	t_3
3	=	t_3		x

3) $x = (-a*b) + (-c*d)$

TAC

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

Quadruple (u- ⇒ unary -)

S.no	op	arg1	arg2	result
0	u-	a		t_1
1	u-	c		t_2
2	*	t_1	b	t_3
3	*	t_2	d	t_4
4	+	t_3	t_4	t_5
5	=	t_5		x

4) $x = (b+c)*(d/e)$

TAC:

$t_1 = b+c$
$t_2 = d/e$
$t_3 = t_1 * t_2$
$x = t_3$

S.no	op	arg1	arg2	result
0	+	b	c	t_1
1	/	d	e	t_2
2	*	t_1	t_2	t_3
3	=	t_3		x

Teacher's Signature

5) $d = (a|b) \& c$

TAC:

$t_1 = a|b$
$t_2 = t_1 \& c$
$d = t_2$

Quadruple

S.no	op	arg1	arg2	result
0	|	a	b	t_1
1	&	t_1	c	t_2
2	=	t_2		d

6) $(a|b) \& (!c | !d)$

TAC:

$t_1 = a|b$
$t_2 = !c$
$t_3 = !d$
$t_4 = t_2 | t_3$
$t_5 = t_1 \& t_4$

Quadruple

S.no	op	arg1	arg2	result
0	|	a	b	t_1
1	!	c		t_2
2	!	d		t_3
3	|	t_2	t_3	t_4
4	&	t_1	t_4	t_5

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

TAC:

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

Quadruple

S.no	op	arg1	arg2	result
0	+	a	b	t_1
1	+	c	d	t_2
2	*	t_1	t_2	t_3
3	*	t_1	d	t_4
4	-	t_3	t_4	t_5

8) $x = (a * (b+c+-a))$

TAC:

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

Quadruple

S.no	op	arg1	arg2	result
0	-	a		t_1
1	+	b	c	t_2
2	+	t_1	t_2	t_3
3	*	a	t_3	t_4
4	=	t_4		x

9) ~~$x = (a*$~~

Teacher's Signature

9) $x = (a*b+c+d) * b-d$

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

Quadruple

S.no	op	arg1	arg2	result
(0)	*	a	b	t_1
(1)	+	c	d	t_2
(2)	+	t_1	t_2	t_3
(3)	*	b	t_3	t_4
(4)	-	t_4	d	t_5
(5)	=	t_5		x

13/05/22 Friday

TRIPLES

$x = (a+b) * (c+d)$

S.no	op	arg1	arg2
(0)	+	a	b
(1)	+	c	d
(2)	*	(0)	(1)
(3)	=	x	(2)

INDIRECT TRIPLES

S.no	op	arg1	arg2
(0)	+	a	b
(1)	+	c	d
(2)	*	(10)	(11)
(3)	=	x	(12)

Indirect table
(0) → (10)
(1) → (11)
(2) → (12)

2) $x = (-a * b) + (-c * d)$

TRIPLES

S.no	op	arg1	arg2
(0)	u-	a	
(1)	u-	c	
(2)	*	(0)	b
(3)	*	(1)	d
(4)	+	(2)	(3)
(5)	=	x	(4)

INDIRECT TRIPLES

S.no	op	arg1	arg2
(0)	u-	a	
(1)	u-	c	
(2)	*	(10)	b
(3)	*	(11)	d
(4)	+	(12)	(13)
(5)	=	x	(14)

Indirect table
(0) → (10)
(1) → (11)
(2) → (12)
(3) → (13)
(4) → (14)

4) $x = (b+c) * (d/e)$

TRIPLES

S.no	op	arg1	arg2
(0)	+	b	c
(1)	/	d	e
(2)	*	(0)	(1)
(3)	=	x	(2)

INDIRECT TRIPLES

S.no	op	arg1	arg2
(0)	+	b	c
(1)	/	d	e
(2)	*	(10)	(11)
(3)	=	x	(12)

Indirect table
(0) → (10)
(1) → (11)
(2) → (12)

5) $d = (a|b)\&c$

TRIPLES

S.no	op	arg1	arg2
(0)	|	a	b
(1)	&	c	(0)
(2)	=	d	(1)

INDIRECT TRIPLES

S.no	op	arg1	arg2
(0)	|	a	b
(1)	&	c	(10)
(2)	=	d	(11)

Indirect table
(0) → (10)
(1) → (11)

6) $(a|b) \& (!c \,|\, !d)$

TRIPLES

S.no	op	arg1	arg2
(0)	!	a	b
(1)	!	c	
(2)	!	d	
(3)	|	(0)	(2)
(4)	&	(0)	(3)

INDIRECT TRIPLES

S.no	op	arg1	arg2
(0)	!	a	b
(1)	!	c	
(2)	!	d	
(3)	|	(11)	(12)
(4)	&	(10)	(13)

Indirect triples
(0) → (11)
(2) → (12)
(10) → (10)
(3) → (13)

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

TRIPLES

S.no	op	arg1	arg2
(0)	+	a	b
(1)	+	c	d
(2)	*	(0)	(1)
(3)	*	(0)	d
(4)	-	(2)	(3)

INDIRECT TRIPLES

S.no	op	arg1	arg2
(0)	+	a	b
(1)	+	c	d
(2)	*	(10)	(11)
(3)	*	(10)	d
(4)	-	(12)	(13)

Indirect triples
(0) → (10)
(1) → (11)
(2) → (12)
(3) → (13)

8) $x = (a * (b+c+-a))$

TRIPLES

S.no	op	arg1	arg2
(0)	-	a	
(1)	+	b	c
(2)	+	(0)	(1)
(3)	*	a	(2)
(4)	=	x	(3)

INDIRECT TRIPLES

S.no	op	arg1	arg2
(0)	-	a	
(1)	+	b	c
(2)	+	(10)	(11)
(3)	*	a	(12)
(4)	=	x	(13)

Indirect triples
(0) → (10)
(1) → (11)
(2) → (12)

9) $x = (a * b + c + d) * b - d$

TRIPLES

S.no	op	arg1	arg2
(0)	*	a	b
(1)	+	c	d
(2)	+	(0)	(1)
(3)	*	(b)	(2)
(4)	-	(3)	d
(5)	=	x	(4)

INDIRECT TRIPLES

S.no	op	arg1	arg2	Indirect triple
(0)	*	a	b	(0) → (0)
(1)	+	c	d	(1) → (11)
(2)	+	(10)	(11)	(2) → (12)
(3)	*	b	(12)	(3) → (13)
(4)	-	(13)	d	(4) → (14)
(5)	=	x	(14)	

16/05/22 Monday

16/05/22 Monday

UNIT-3

Syntax Directed

* The value of an attribute at any node of parse tree is defined by a semantic rule associated with the production used as the node.
There are 2 different attributes.
i) Synthesized ii) Inherited attributes
* The value of a Synthesized attribute at any node is computed from the values of the attribute at the child nodes of that node in the parse tree
*

Attributed Grammar with Synthesized attribute

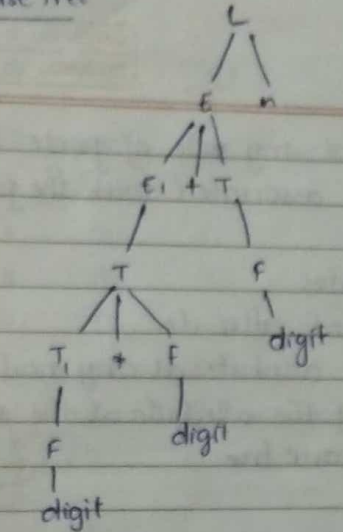
Eg:

	SDD / Attributed Grammar
$L \rightarrow En$	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexvalue$

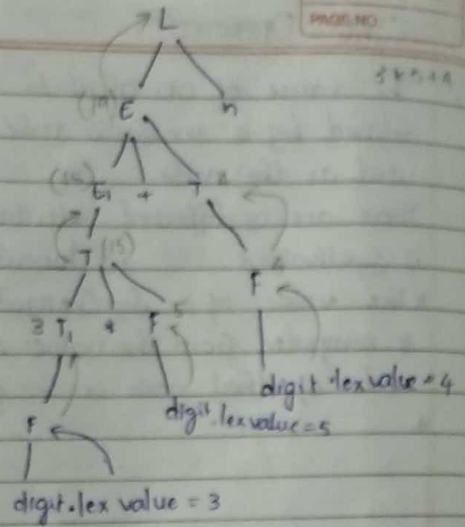
NOTE: A parse tree representing the values of its attributes ~~is known~~ using SDD is known as Annotated parse tree.

Q. Construct annotated parse tree for the string 3*5+4 using the above given grammar.

Parse Tree

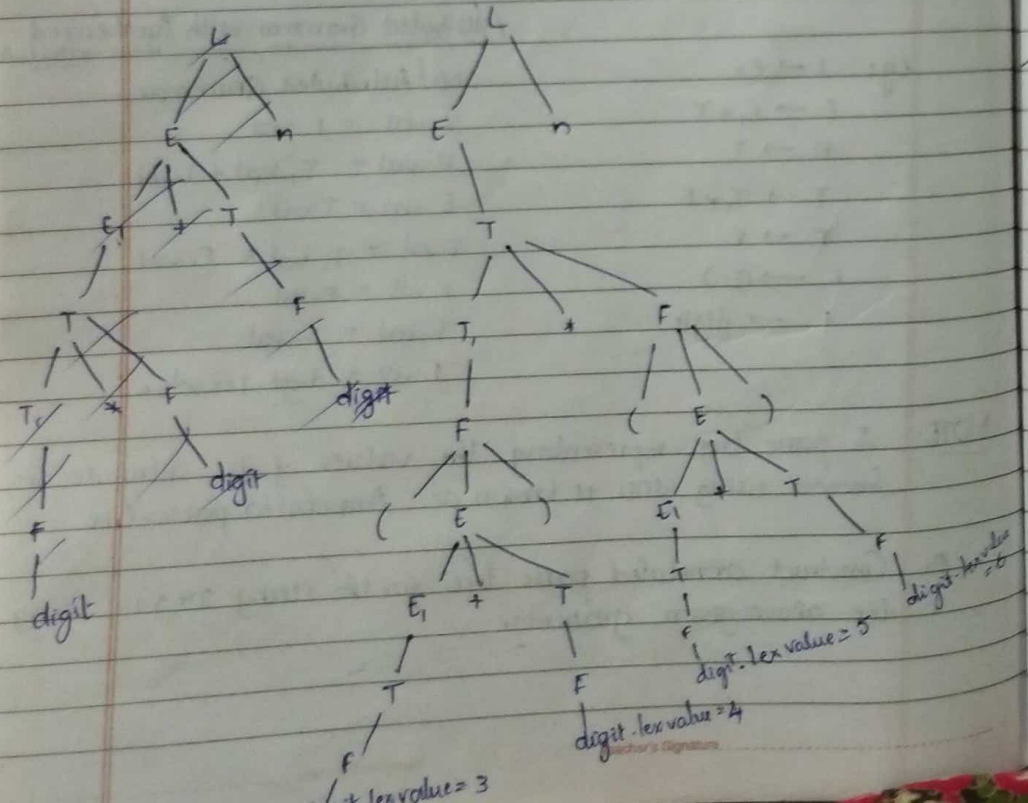


Annotated Parse Tree

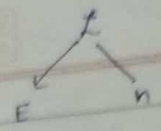


strings : 1) (3+4) * (5+6) n

2) 1 * 2 * 3 * (4+5) n



2) 1 * 2 * 3 * (4+5) n



Inherited Attributes :

17/05/22 Tuesday

Eg: D → TL

T → int

T → real

L → L1 , id

L → id

Attributed Grammar with inherited attributes

L.in = T.type

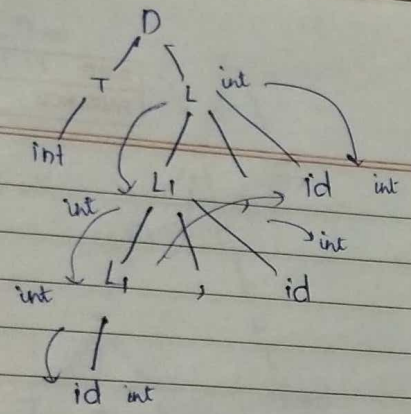
T.type = int

T.type = real

L1.in = L.in addtype (id.entry, L.in)

addtype (id.entry, L.in

int a1, a2, a3



DATE: / /
PAGE NO.:

20/5/22
Friday

Dependency Graph

NOTE: The interdependencies among the inherited and synthesized attributes can be represented in a parse tree with the help of the directed graph known as Dependency Graph.

Eg:

$E \rightarrow E_1 + T$
$E \rightarrow E_1 - T$
$E \rightarrow T$
$T \rightarrow (E)$
$T \rightarrow id$
$T \rightarrow num$

Q) Construct SDD for the given CFG and also construct the syntax tree for the expression a-4+c

Syntax trees a-4+c

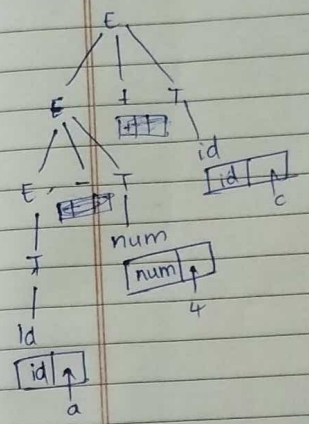
p_1 = mkleaf (id, entry a)
p_2 = mkleaf (num, 4)
p_3 = mknode (-, p_1, p_2)
p_4 = mkleaf (id, entry c)
p_5 = mknode (+, p_3, p_4)

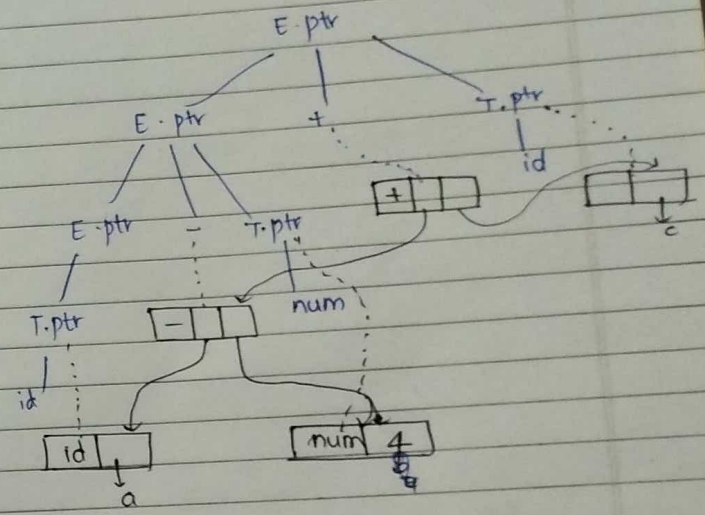
E.ptr = mknode ('+', E_1.ptr, T.ptr)
E_1.ptr = mknode ('-', E_1.ptr, T.ptr)
E.ptr = T.ptr
T.ptr = E.ptr
T.ptr = mkleaf (id, id.entry)
T.ptr = mkleaf (num, value)

SDD for CFG to construct parse tree

DATE: / /
PAGE NO.:

Parse Tree





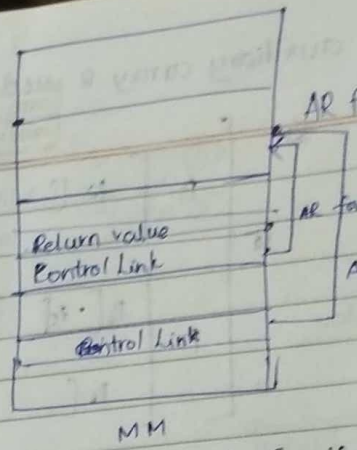
14/06/22 Tuesday

Temporaries

* Return value is the value returned by the procedure
* It consists of the values of the actual parameters
* It Points to the activation record of the calling procedure
* Access Link: It refers to the non-local data in other activation records.
* Saved Machine status: Before the procedure is called, some control information will be stored in the registers
* Local Variables: It holds all the local variables of a procedure.
* Temporaries: All the temporary variables generated by the compiler inorder to execute the procedure are stored in this field.

Eg:

```
main ()                    float sum(x,y)
{                          {
  =                          =
  a = sum(b,c);            }
  =                        int mul(p,q)
  b = mul(e,f);            {
  =                          =
}                          }
```

MM

Q. Represent activation records for the recursive function factorial (3)

Access to Non-Local Names:

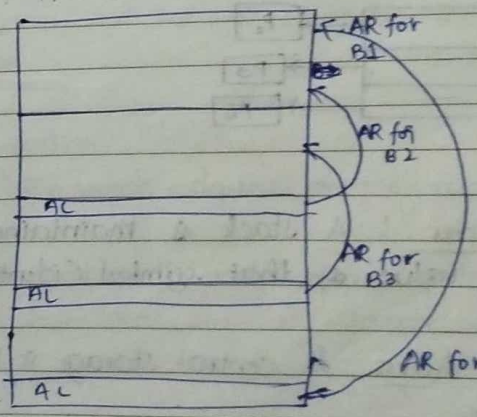
1. Static Scope (or) Lexical Scope:
   Two types of representations 1. Access Link 2. Display
2. Dynamic Scope:
   1 Deep Access 2. Shallow access.

1. Access Link:
In access link, the activation record is represented using the access links.

Eg:

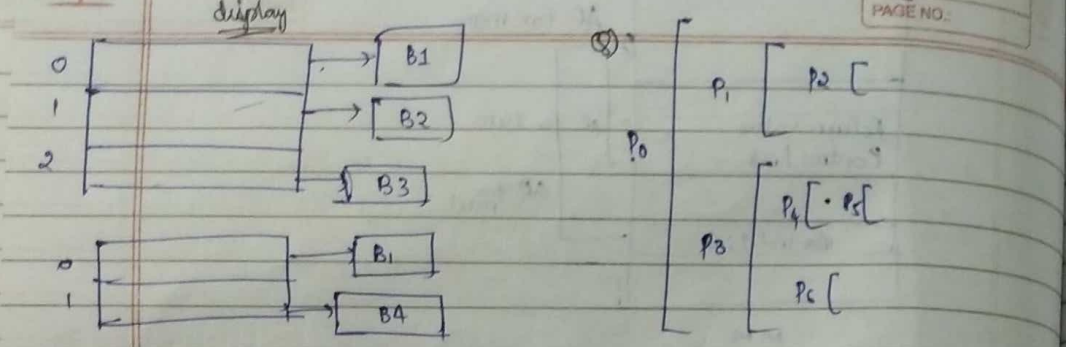
```
B1 ()
{
  =
  B2 ()
  {
    =
    B3 ()
    {
      =
    }
  }
  B4 ()
  {
    =
  }
}
```

Access Link Representation of static scope.



2. Display:

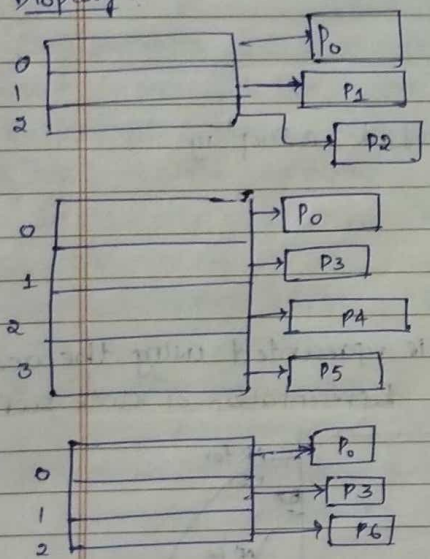
In this representation, an auxiliary array is used



Represent the access link and display representations for the given procedures

Display:

Access link

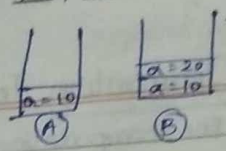


1) Deep Access: A stack is maintained in order to retrieve the most recent value of that symbol (identifier)

2) Shallow Access: A central storage is kept with one slot for every variable

```
B1()
{
  int a = 15
  Pf(a);  ---> (A)
  B2();   ---> (B)
  =
}
B2()
{
  int a = 20;
  Pf(a);
}
```

Deep Access



Shallow Access

10 — value of a in B1

20 — value of a in B2

UNIT - V

16/06/22 Thursday

CODE OPTIMIZATION

* Intermediate code is the input for this phase.
* Optimised Intermediate code is the output.
* The main tasks of code optimizer are
  1. The semantic equivalence of the program should not be changed.
  2. It should produce efficient object code.
* The main Challenges for Code Optimizer
  1. The optimising algorithm must be efficient enough in such a way that the target code should not become time and space inefficient.
  2. The algorithm should preserve its semantic equivalence

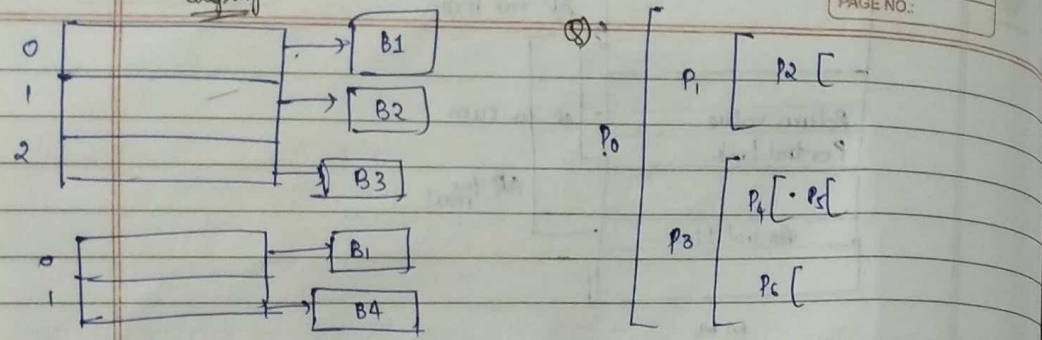
Types of Code Optimization

1. Machine Dependent Optimization
2. Machine Independent Optimization - depends on programming lang

1. Machine Dependent : It depends on the architecture of the machine i.e the number of registers, storage

2. Display:

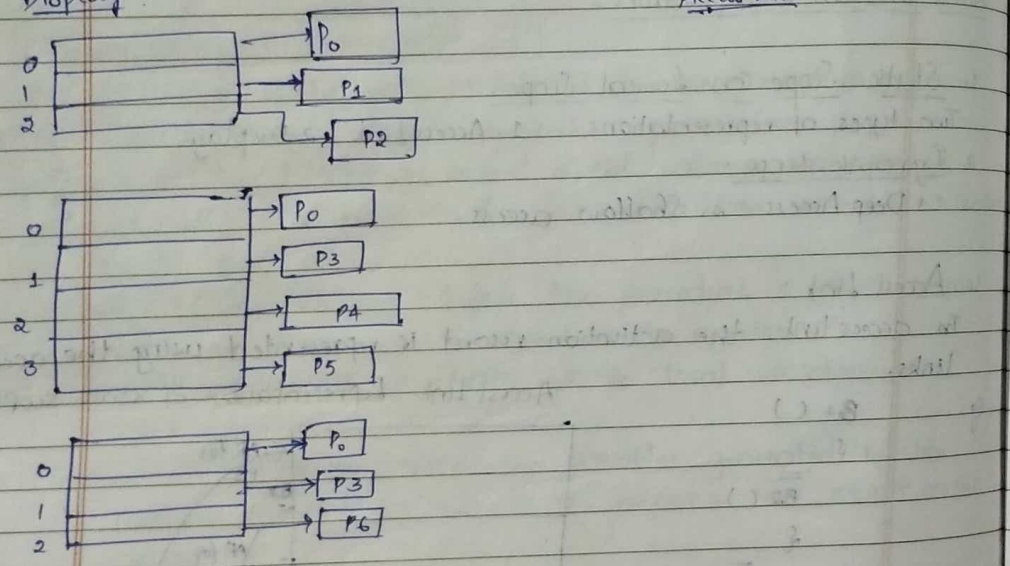
In this representation, an auxiliary array is used



Represent the access link and display representations for the given procedures

Display:

Access Link



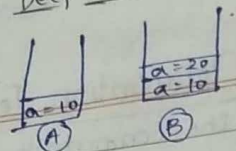
1) Deep Access: A stack is maintained in order to retrieve the most recent value of that symbol (identifier)

2) Shallow Access: A central storage is kept with one slot for every variable

Teacher's Signature

```
B1()
{
  int a = 15
  Pf(a);  ---> (A)
  B2();   ---> (B)
  =
}
B2()
{
  int a = 20;
  Pf(a);
}
```

Deep Access



Shallow Access

10 — value of a in B1

20 — value of a in B2

UNIT - V

16/06/22 Thursday

CODE OPTIMIZATION

* Intermediate code is the input for this phase.
* Optimised Intermediate code is the output.
* The main tasks of code optimizer are
  1. The semantic equivalence of the program should not be changed.
  2. It should produce efficient object code.
* The main Challenges for Code Optimizer
  1. The optimising algorithm must be efficient enough in such a way that the target code should not become time and space inefficient
  2. The algorithm should preserve its semantic equivalence

Types of Code Optimization

1. Machine Dependent Optimization
2. Machine Independent Optimization - depends on programming lang

1. Machine Dependent : It depends on the architecture of the machine i.e the number of registers storage

Teacher's Signature

Types of Optimization Techniques :

1. **Folding** : The computation of a constant can be performed at compile time rather than at execution time.

Eg: `a = 15/4 * p`

15/4 is computed at compile time using Folding Optimization technique.

2. **Constant Propogation** : The value of a variable is replaced and the expression is evaluated at compilation time.

Eg: `r = 5 , a = b*r*r`

→ `a = b*25` 5*5 is executed in compile time

3. **Common Sub-Expression Elimination** : An expression which is computed previously can be eliminated if it appears further in the program.

Eg:
```
t1 = 4*i                t1 = 4*i
t2 = a(t1)      →       t2 = a(t1)
t3 = 4*i                t4 = b(t1)
t4 = b(t3)
```

4. **Variable Propogation** : There is no need of using two different variables for one single value

Eg:
```
x = pi                  x = 3.14
pi = 3.14       →       a = x*r*r
a = x*r*r               ↓
                        Constant propogation
```

5. **Code Movement** : Some portion of the code can be moved out of the loop in order to reduce time and space.

Eg:
```
for(i=0; i<10; i++) {
    x = y*10;
    z = y*10 + w;
}
```
→
```
Optimized Code
y = y*10
for(i=0; i<10; i++) {
    x = y;
    z = y + w;
}
```

NOTE: Code Movement is also known as Loop Invariant method

Eg:
```
while (i<= max-1) {
    ---
}
```
Optimized code
```
n = max-1
while (i<= n) {
    ---
}
```

6. **Strength Reduction or Induction Variable Elimination**:

Higher strength operators like * and / can be replaced by + and - respectively.

> *, / are said to be higher strength operator as they require more no. of seconds compared to + and - respectively

```
for (i=1; i<10; i++) {
    c = i*5;
}
```

Optimized Code
```
t = 5
for( i=1; i<10; i++) {
    c = t;
    t = t+5;
}
```

Here t is known as Induction variable

Strength Reduction

Eg: `b = 2*a`

→ `b = a+a`

7. **Dead Code Elimination** : A variable is said to be dead if its value is never used in the program

Eg.1)
```
i = 2
j = 5
x = i+k ....
```
Code optimization
```
i = 2
x = i+k
```

Eg.2)
```
i = 1
if (i == 0)
```

8. **Loop Unrolling** : The number of tests and jumps can be reduced by writing the loop for two or more timing

```
int i = 1;
while(i<=200) {
    a[i] = b[i] + c[i];
    i = i+1;
}
```

Continue from only 100 - 200

a(i) = b(i) + c(i);
i++;
}

i = 1;
while(i <= 100)
{ a(i) = b(i) + c(i);
i++;
a[i] = b[i] + c[i];
i++;
}

2) Loop Fusion: Several loops can be merged to a single loop

Eg: for(i=1 to n)
for(j=j to m)
a[i,j] = 20

for(i=0 to n*m)
a[i] = 20

17/06/22 Friday

BASIC BLOCKS

* To construct a basic block, the following rules are applied.
1. Identify the leader statements.
2. A basic block is formed starting at the leader and ending just before the leader statements.

Rules to find the Leader:
1. The first statement is a leader statement
2. Any conditional or unconditional goto's target statement is a leader
3. Any statement that immediately follows a goto is a leader

Eg: ① s=0
② i=1
③ t1 = 4*i
④ t2 = a(t1)
⑤ t3 = 4*i
⑥ t4 = b(t3)
⑦ t5 = t2*t4
⑧ s = s+t5
⑨ i = i+1
⑩ if(i<=10) goto ③

Leader statements: ① and ③
① ② - one block, ③ to ⑩ - another block

Block 1
s=0
i=1

Teacher's Signature

OPTIMISATION OF BASIC BLOCKS:

* Basic blocks can be optimized in 2 different ways.
1. Structure Preserving Optimization
2. Use of Algebraic identities

1. Structure Preserving Optimization:
* It can be achieved by applying common sub-expression elimination, dead code elimination, Variable propogation, Constant propogation and code movement.

2 Use of Algebraic Identities:

a+0 = a	1. These identities can be used for optimization
a*1 = a	2. Strength reduction can also be applied to optimize
a/1 = a	the given code

Eg: 2*a can be replaced with a+a

3. Constant folding technique can be applied:
a = 2*(2.3) + b
2*2.3 will be evaluated at compile time

4. Common Sub-expression elimination can also be performed.

Eg:
x = y*z
t = z*w*y

Optimized Code
x = y*z
t = w*x

PEEP HOLE OPTIMIZATION TECHNIQUES:

1. Redundant Instruction Elimination: Redundant loads and stores can be eliminates

Eg: 1) MOV R0, x
MOV x, R0 → Redundant statement - can be eliminated

Eg: 2) int sum(a,b){
c = a+b
return c
printf(c){

Eg: 3 s=0
if(s>1){
}
= Redundant block of code

DATE: / /
PAGE NO.:

2. Flow of Control Optimization: Unneccessary jumps or ju
can be eliminated.

Eg: go to l1;
~~goto l2~~
l1:
goto l2
=
l2:
=

Optimized code

goto l2;
l1:
go to l2
=
l2:
=

3. Algebraic Simplification
$x = x+0$ or $x = x*1$ or $x = x/1$ can be reduced to
by using algebraic identities.

4. Strength Reduction:
'*' can be replaced with '+' and '/' can be replaced by

5. Machine Idioms: increment or decrement instructions can
used instead of executing the binary operations.
Eg: $i = i+1$ can be replaced by ~~inc i~~ INCR i