Syntax Directed Translation

→ we Associate information with the programming language constructs by attaching attributes to grammar symbols.

→ Value of this attributes are evaluated by semantic Rules associated with the production rules.

→ Evaluation of these semantic rules
  - → may generate intermidiate code
  - → may put information into the symbol table.
  - → may perform type checking.

→ An attribute may hold almost anything, a string, number, memory loc., complex record, etc.

Syntax Directed Definition & Translation Schemes

→ when we associate semantic rules with productions, we use two notations

(i) Syntax Directed Definition

→ give high level specification for the translation

→ hide many implementation detail such as order of evaluation of semantic actions.

→ we associate production rules with a set of semantic actions, & do not say when they will be evaluated.

(ii) Translation Schemes

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

Possible Call Sequence

→ The caller evaluates actual

→ The caller stores a return address & the old value of top-sp in the callee's activation record. The caller then increments top-sp & moves past caller's local data & temps & the callee's parameter & status field.

→ The callee saves register values & other status information

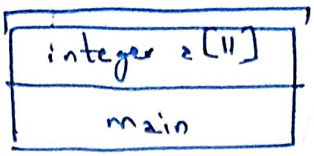
→ The callee initializes it's local data & begin execution.

A possible return sequence

→ The callee places a return value next to the activation record of caller

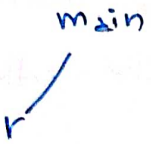
→ Using the information in the status field, the callee restores top-sp & other registers & branches to a return address in caller's code.

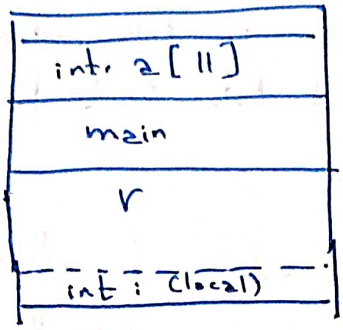
Eg. (I) main
(wrt quicksort)



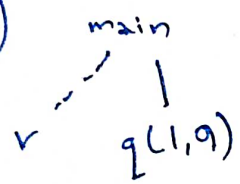
(Downward growing stack of the activation record)

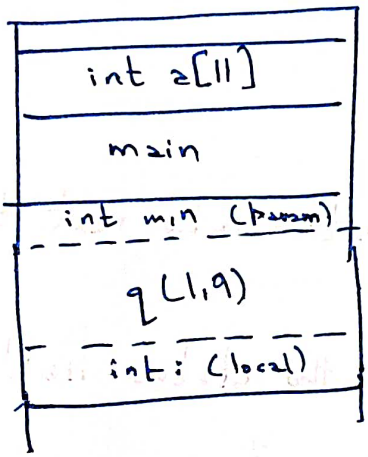
(II)



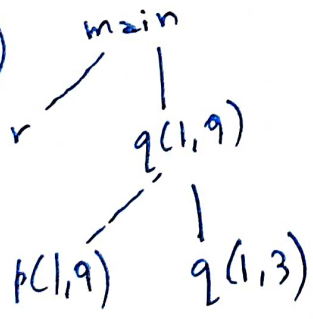


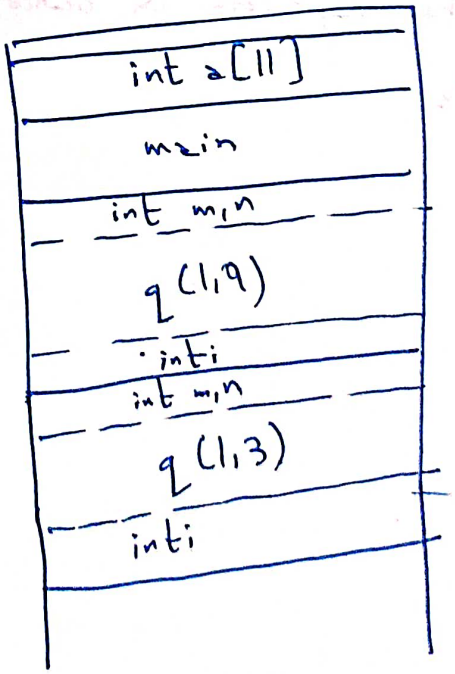
(III)





(IV)



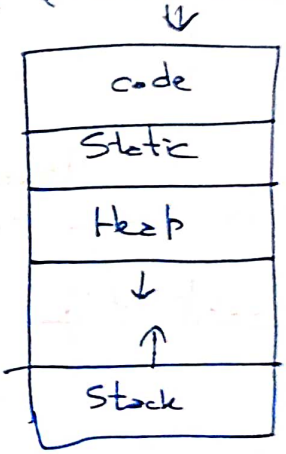


Storage Organization

→ Suppose that the compiler obtains a block of storage from OS for compiled program to run in.

→ Runtime storage might be subdivided to hold
  → generated target code
  → data objects &
  → a counter part of control stack to keep track of procedure activation

→ Typical Subdivision of run-time memory into code & data area



Activation Records

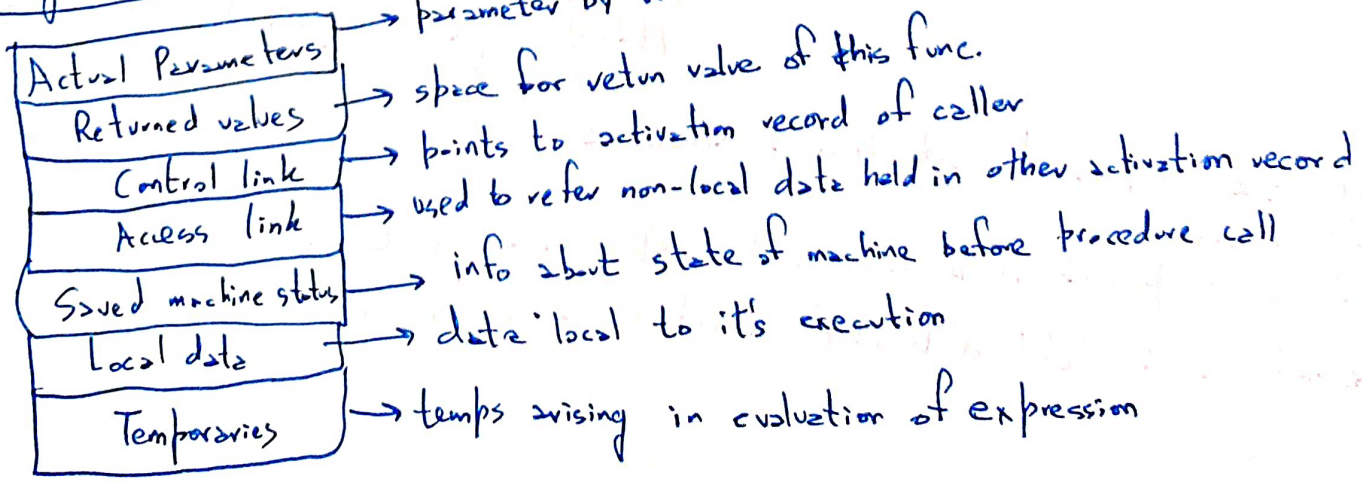
→ Procedure calls & returns are usually managed by a run time stack called the control stack

→ Each live activation has an active record (sometime called frame)

→ The root of activation tree is at bottom of stack

→ The current execution path specifies the content of the stack with the last activation has record in the top of the stack

A general Activation Record

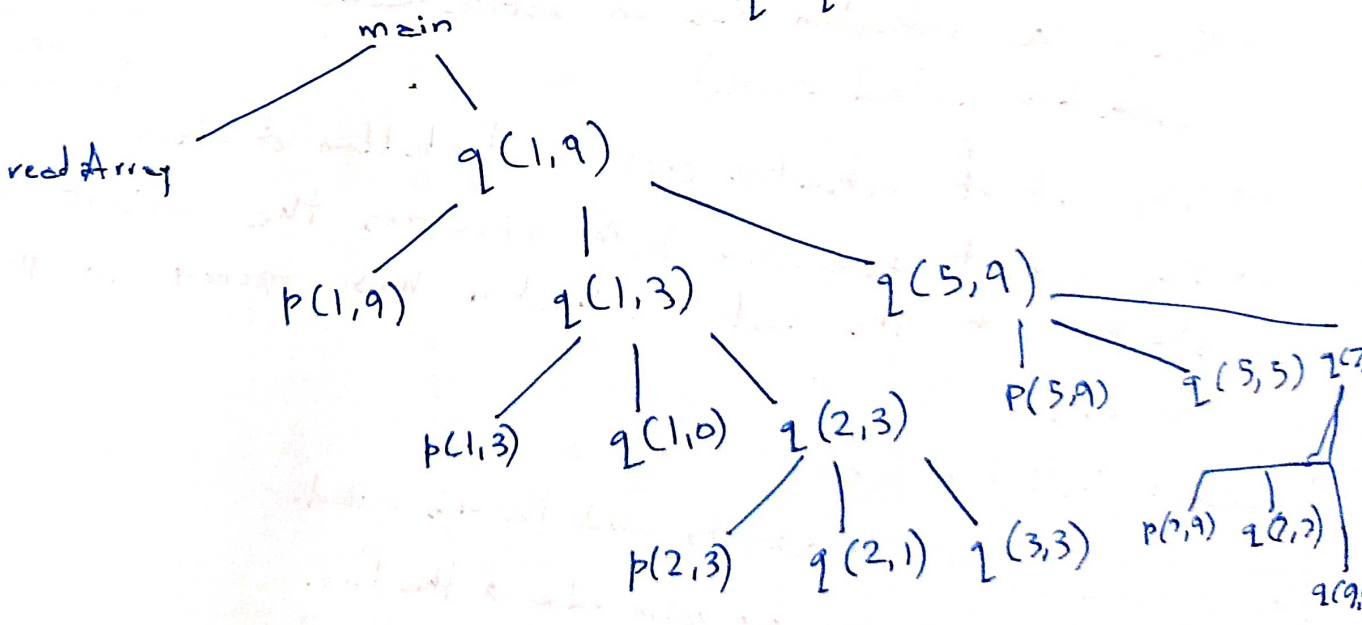


Eg. Activations for quicksort algo on an array of 10 elements (0th & 11th element not used)

enter main ()
  enter readArray ()
  leave readArray ()
  enter quicksort (1,9)
    enter partition (1,9)
    leave partition (1,9)
    enter quicksor (1,3)
      !
    leave quicksort (1,3)
    enter quicksort (5,9)
      ⋮
    leave quicksort (5,9)
  leave quicksort (1,9)
leave main ()

Coressponding activation tree

p → partition
q → quicksort



→ each node represents activation of procedure
→ root → main
→ node a is parent of b iff control flows from a to b
→ node a is left of b iff lifetime of a occurs before b

Runtime Enviornments

→ Compiler must do storage allocation & provide access to variables & data

→ The allocation & deallocation of data objects is managed by runtime support package, consisting of routines loaded with the generated target code.

→ Each execution of a procedure is referred to as an activation of procedure

→ If the procedure is recursive, several of it's activation may be alive at the same time.

Activation Tree

→ we make the following assumptions about the flow of control among procedures during execution of a program.

→ Control flows sequentially; that is the execution of a program consists of a ~~program~~ sequence of steps, with control being at some specific point in the program at each step.

→ each execution of a procedure starts at the beginning of procedure body & eventually returns the control to the point immediately following the place where procedure was called.

Lifetime: refers to a concecutive sequence of steps during the execution of program.

Loop Optimizations

(i) **Code Motion**
→ moves code ~~not~~ outside the loop (if possible)

(ii) **Induction Variable Elimination**
→ apply to eliminate i & j from inner loops B_2 & B_3

(iii) **Reduction in Strength**
→ replace expensive operation by cheaper one (such as multiplication by an addition.

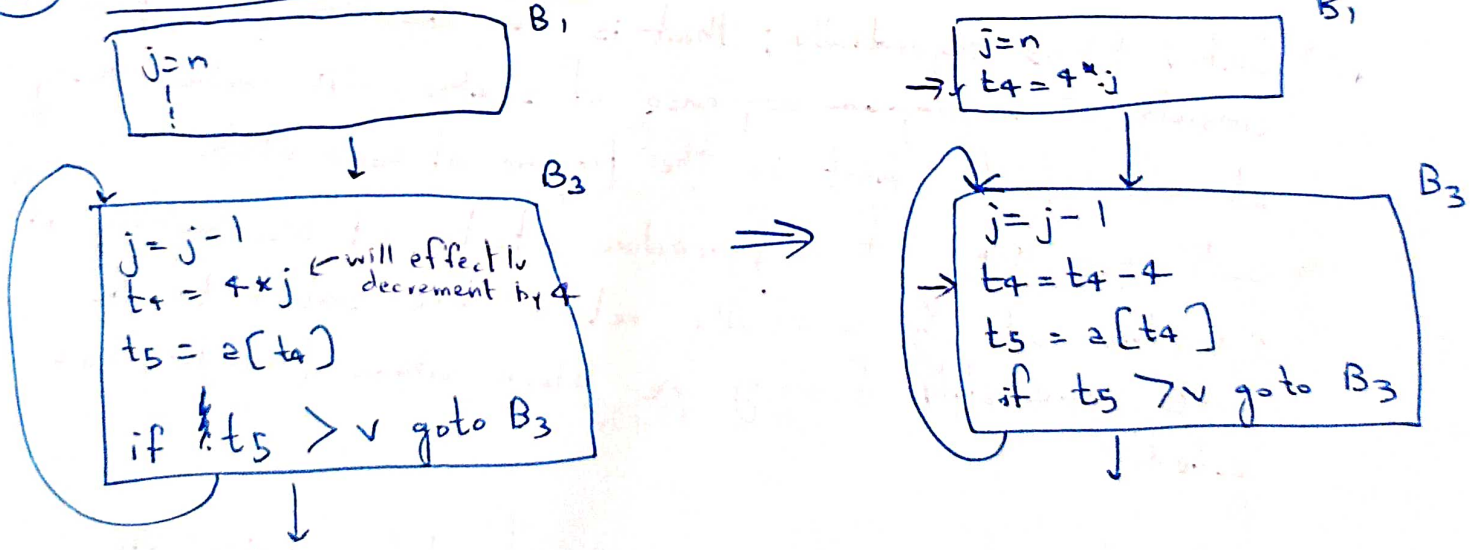
(I) code motion

```
while (i <= limit-2) {        // limit-2 calculated every time
    //limit does not change
}
```

⇒

```
t = limit-2
while (i <= t) {
}
```

(II) Induction variable & reduction in strength



Optimization : common subexpression

B_5

```
t6 = 4 * i
n = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = n
goto B2
```

remove redundancy

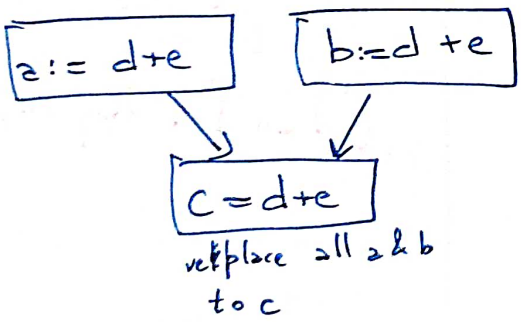
B_5

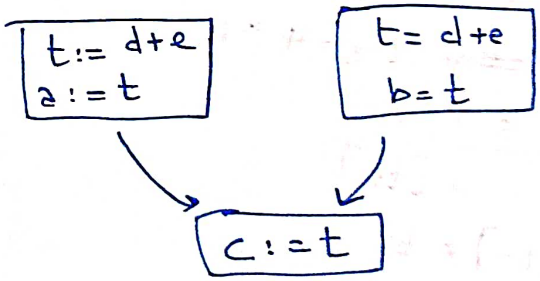
```
t6 = 4 * i
n = a[t6]
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
a[t8] = n
goto B2
```

→ now computation of t_6 & t_8 is also already done at B_3 using t_2 & t_4 & can also be removed

---

Copy Propagation

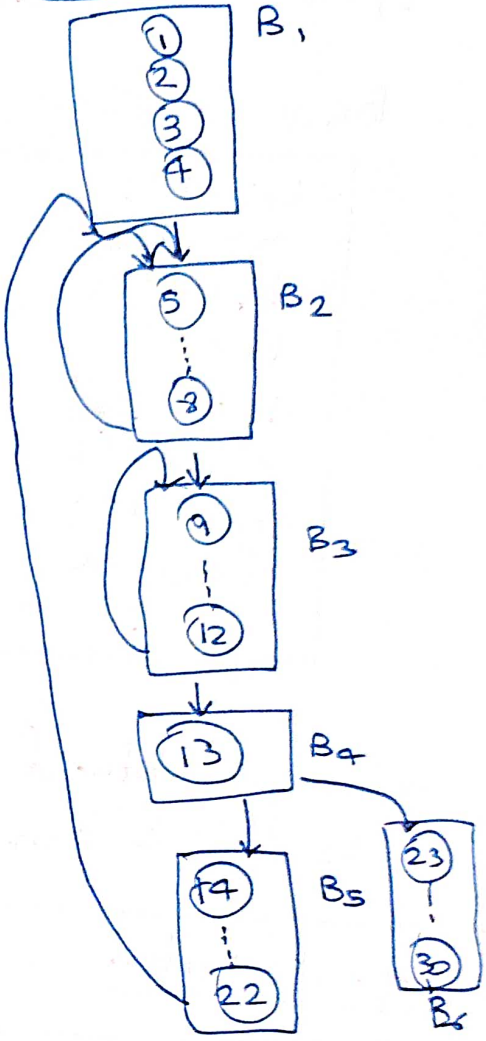




coress ponding three address code for the fragment

(1) i := m - 1 ← leader
(2) j := n
(3) $t_1 := 4 * n$
(4) v := a[t_1]
(5) i := i+1 ← leader
(6) $t_2 := 4 * i$
(7) $t_3 := a[t_2]$
(8) if $t_3 < v$ goto (5)
(9) j := j-1 ← leader
(10) $t_4 = 4 * j$
(11) $t_5 = a[t_4]$
(12) if $t_5 > v$ goto (9)
(13) if i >= j goto (23) ← leader
(14) $t_6 = 4 * i$ ← leader.
(15) x = a[t_6]
(16) $t_7 = 4 * i$
(17) $t_8 = 4 * n$
(18) $t_9 = a[t_8]$
(19) $a[t_7] = t_9$
(20) $t_{10} = 4 * n$
(21) $a[t_{10}] = x$
(22) goto (5)
(23) $t_{11} = 4 * i$ ← leader.
(24) x = a[t_{11}]
(25) $t_{12} = 4 * i$
(26) $t_{13} = 4 * n$
(27) $t_{14} = a[t_{13}]$
(28) $a[t_{12}] = t_{14}$
(29) $t_{15} = 4 * n$
(30) $a[t_{15}] = x$

Flow Graph

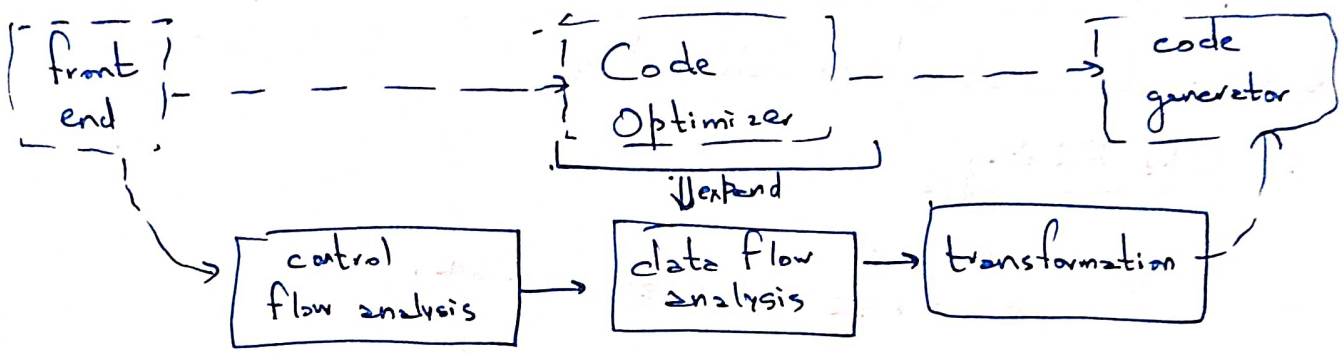


(draw full diagram)

Peephole Optimization Limitations

→ Local in Nature
→ Pattern Driven
→ Limited by size of window

Optimizing Compiler (Code Optimizer)



Eg: C code for quicksort

```
void quicksort (int m, int n) {
    int i,j;
    int v,x;
    if (n <= m) return;
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
    quicksort (m,j); quicksort (i+1, n);
}
```

fragment

Peephole Optimization

→ examining a short sequence of target instructions & replacing those by faster sequences

→ Peephole is a small moving window on the target program

Characteristics of Peephole optimization

(I) Redundant-instruction elimination

(i) Copy Folding

Eg: `n = 32;`
`n = n + 32;` ⟹ `n = 64;`

(ii) Unreachable code

Eg: `goto L2;`
`n = n + 1;` ⟹ `goto L2;`

(II) flow-of-control optimizations

`goto L1;`
⋮
`L1: goto L2;` ⟹

`goto L2;`
⋮
`L1: goto L2;`

(III) Algebraic Simplifications

(i) `n = n + 0` ← unneeded

(ii) Dead Code

Eg `n = 32` ⟹ `n = 32 + 4`
`n = n + 4`
~~(Control flow forward)~~

(iii) Reduction in Strength ⇒ replace expensive operation by cheaper one

Eg: `n = n * 2` ⟹ `n = n + n`

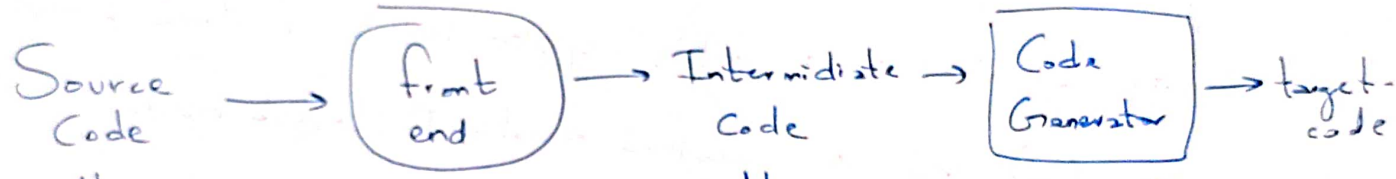
(IV) Use of machine idioms

Code Optimization

→ A transformation to a program to make it run faster and/or take up less space

→ Optimization should be safe, & preserve the meaning of a program

Getting Better Performance



User can
profile program, change algo, transform loop

Compiler can
improve loops, procedure call, address calculation

Compiler can
use registers, select instrc, do peephole transform

Levels

1. Window (Peephole Optimization)
2. Procedural → Global ((Control Flow Graph))

The function getreg()

→ getreg() returns the location L to hold the value of x for the assignment (x = y op z)

(1) If the name y is in a register that holds the value of no other names & If the y is not live & has no next use after execution of (x = y op z) then return the register of y for L. Update the address descriptor of y such that y is no longer in L

(2) Failing (1), return an empty register for L if there is one

(3) Failing (2), find an occupied register R, to be get freed.

(4) If x is not used in a block, or no suitable occupied register can be found, select memory location of x as L

A Code Generation Algorithm

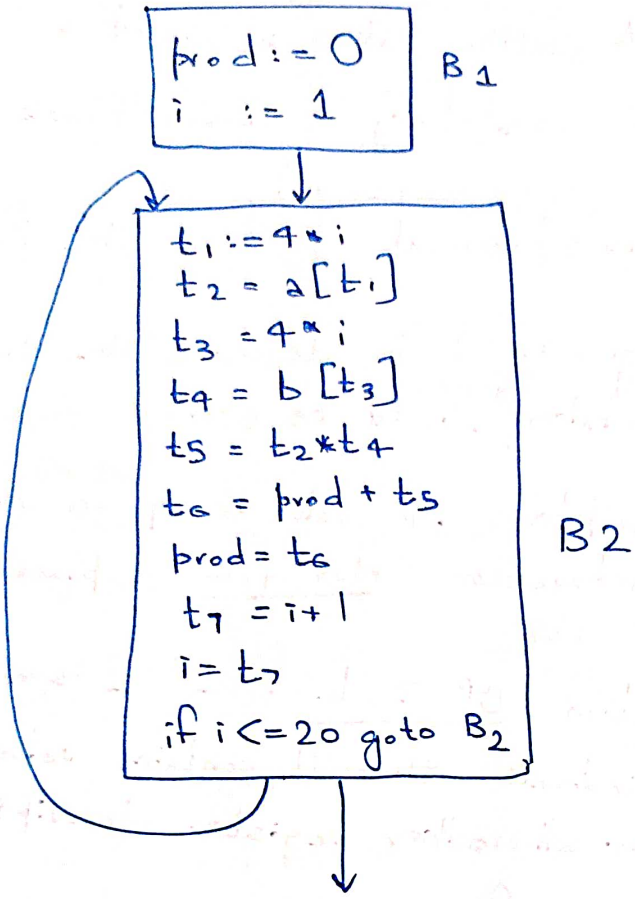
→ The Code Generation Algorithm takes as input a sequence of three address statements constituting a basic block

→ For each three address statement ($x = y \text{ op } z$), perform the following:

1. Invoke a function getreg() to determine the location L where result of computation ($y \text{ op } z$) should be stored
2. Consult address descriptor. If value of y is not already in L generate instruction MOV y, L to place a copy of y in L
3. Generate the instruction OP z, L. If L is a register, update descriptor to indicate that it contains value of x, and remove x from all other register descriptor.
4. If the current values of y and/or z have no next uses, they are not live on exit from the block. Alter the register descriptor.

*

Coressponding flow graph



Register & Address Descriptor

→ A register descriptor keeps track of what is currently in each register. It is consulted whenever a new register is needed.

→ An address descriptor keeps track of the location where the current value of any variable name can be found at runtime

Algorithm : Partition into Basic Block

Input: A Sequence of Three Address Statement
Output: A list of Basic Blocks with each TAC in exactly one block

Method:

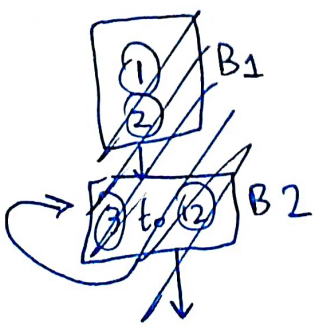
1) Determine the leaders, the first statements of Basic Blocks
   - (i) The first statement is a leader
   - (ii) any statement that is the target of a conditional or unconditional goto is a leader
   - (iii) Any statement which immidiately follows a goto or conditional goto statement is a leader
2) For each leader, it's basic block consists of the leader and all statements upto but not including the next leader or the end of program.

Eg: (dot product of two vectors a & b of size 20)

```
begin
    prod := 0;
    i := 1;
    do begin
        prod := prod + a[i] * b[i];
        i := i+1;
    end
    while i <= 20
end
```

TAC ⟹

(1) prod := 0 → lead
(2) i := 1
(3) t_1 = 4 * i → lead
(4) t_2 = a[t_1]
(5) t_3 = 4 * i
(6) t_4 = b[t_4]
(7) t_5 = t_2 * t_4
(8) t_6 = prod + t_5
(9) prod = t_6
(10) t_7 = i+1
(11) i = t_7
(12) if i <= 20 goto (3)



flow graph

Basic Block & Flow Graphs

→ A graph representation of TAC is called a flow graph

→ Nodes in flow graphs represent computation

→ Edges represent flow of control

→ Some register assignment algorithm use flow graphs to find the inner loop where a program is expected to spend most of it's time.

~~Basic Block~~

→ A Basic block is a sequence of consecutive statements in which flow of control enters at the beginning & leaves at the end without halt or possibility of branching except at the end.

→ A name in a basic block is said to be live at a given point if it's value is used after that point in the program, perhaps in another basic block.

~~Algorithm : Partition into Basic Blocks~~

Code Sequences for Conditional Statement

→ for branch instruction, set a condition code to indicate whether a last quantity computed or loaded into a register is -ve / zero / +ve

→ Compare instruction (CMP in our case) has the desirable property that it set the condition code without actually computing a value

→ That is CMP n, y sets a condition code to +ve if n > y & so on. A conditional-jump machine instruction makes the jump if a designated condition <, =, >, ≤, ≠, ≥ is met

Eg: if n < y goto z ⟹ CMP n, y
CJ< z

condition jump if < is satisfied (n<y here)

Eg: n = y + z
if n < 0 goto z ⟹

```
MOV  y, R0    (condition = y)
ADD  z, R0    (condition = y+z = n)
MOV  R0, n
CJ<  z
```

Code Sequences of Indexed Assigment

→ Indexing operation in Three Address Code are handled in the same manner as binary operations

Statement	i in Register R_i Code	i in memory M_i code
a = b [i]	MOV b(Ri) , R MOV R , a	MOV Mi , R MOV b(R) , R MOV R, a
a[i] = b	MOV b , a(Ri)	MOV Mi , R MOV b , a(R)

Code Sequences for pointer Assigment

Statement	p in register R_p code	p in memory M_p code
a = *p	MOV *Rp , a	MOV Mp , R MOV *R , a
*p = a	MOV a , *Rp	MOV Mp , R MOV a , *R

Address Modes

Mode	Form	Address
absolute	M	M
register	R	R
indexed	c(R)	c + contents(R)
indirect registers	*R	contents(R)
indirect indexed	*c(R)	contents(c + contents(R))

→ contents(a) denotes the content of register or memory loc. represented by a.

Eg: $d = (a - b) + (a - c) + (a - c)$

⇓ Three Addr Code

```
t = a - b
u = a - c
v = t + u
d = v + u
```

⇓ Generated Code

Statements	Code Generated	Register Desc.	Address Desc.
		Register empty	
t = a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u = a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
v = t + u	ADD R1, R0	R0 contains v R1 contains u	~~t in R~~ v in R0 u in R1
d = v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 & memory ~~d in~~

Instruction Selection

~~→ Every~~

Eg: $n = y + z$ ⟹

```
MOV  y, R0   → load y in register R0
ADD  z, R0   → add z to R0
MOV  R0, n   → store R0 to n
```

Eg: $a = b + c$
$d = a + e$ ⟹

```
MOV  b, R0
ADD  c, R0
MOV  R0, a
MOV  a, R0
~~MOV~~ ADD  e, R0
MOV  R0, d
```

A Simple Target Machine

→ Byte addressable machine with four bytes to a word

→ n general purpose registers R0, R1, ---- R_{n-1}

→ It has address instruction of the form

op src dest

op ↓ opcode

src ↓ source

dest ↓ destination

source, destination → data field

Eg of opcode:

```
MOV (move source to dest)
ADD (add src to dest)
SUB (sub src to dest)
 |
otc
```

Code Generation

→ The final phase of a compiler is code generation

→ It receives intermidiate representation (IR) with supplementary information in symbol table

→ Produces semantically equivalent target program

→ Code generator main tasks

(i) Instruction Selection

(ii) Register Allocation & Selection

(iii) Instruction Ordering.

Code Generation Issues

→ The most important criterion is that it produces correct target code.

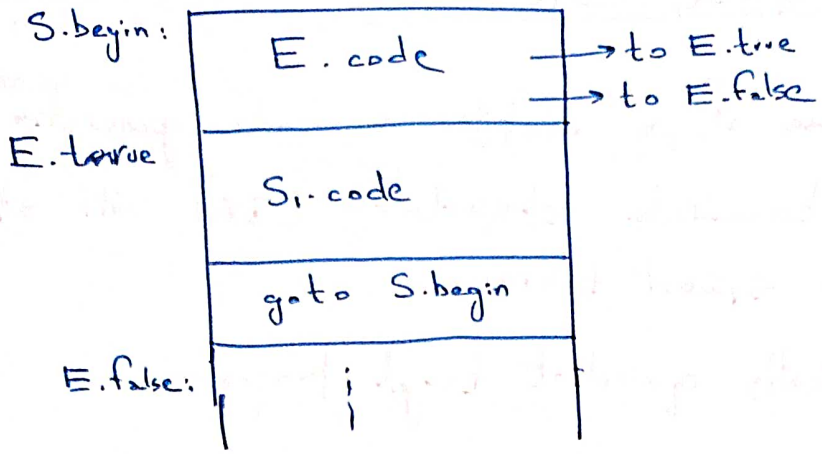
→ Input to Code Generator

→ IR + Symbol Table

→ We assume that front-end produces low level IR, i.e values in it can be directly manipulated by the machine instructions

→ The Target Program

Prerequisite: target machine & it's instruction set



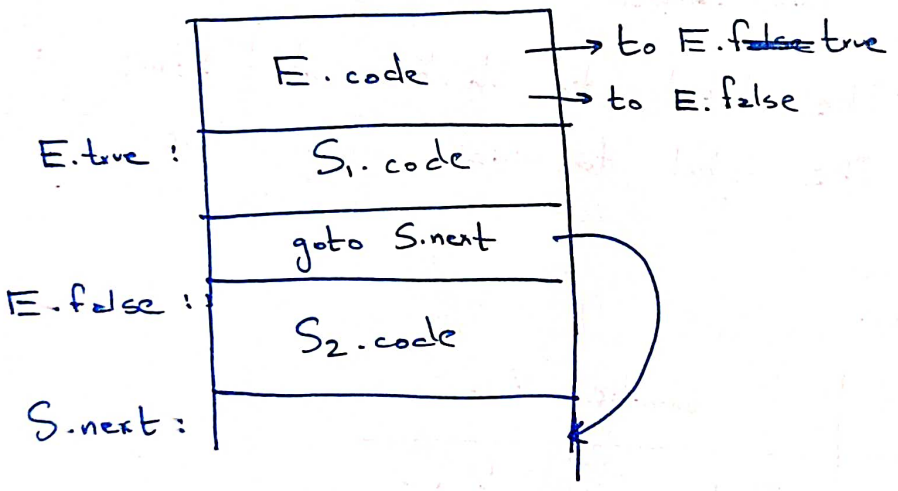
SDD for boolean expression

[here E_1 & E_2 ~~get~~ code will have gotos of their own]

Production	Semantic Rule
$E \to E_1$ or E_2	E_1.true = E.true ; E_1.false = ~~E.fe~~ newlabel() ; E_2.true = E.true ; E_2.false = E.false ; E.code = E_1.code || gen(E_1.~~true~~false, ':') || E_2.code
$E \to E_1$ and E_2	E_1.true = newlabel() ; E_2.false = E.false ; E_2.true = E.true ; E_2.false = E.false E.code = E_1.code || gen(E_1.true, ':') || E_2.code ;
$E \to$ not E_1	E_1.true = E_2.false ; E_1.false = E.true ; E.code = E_1.code ;
$E \to (E_1)$	E_1.true = E.true ; E_1.false = E.false ; E.code = E_1.code ;
$E \to id_1$ relop id_2	E.code = gen('if', id_1.place, relop.op, id_2.place, 'goto', E.true) || gen('goto', E.false)
$E \to$ true	E.code = gen(~~E~~ 'goto', E.true)
$E \to$ false	E.code = gen('goto', E.false)

(II) If - then - else

Production	Semantic Rule
$S \rightarrow$ if E then S_1 else S_2	E.true = newlabel(); E.false = newlabel(); S_1.next = S.next; S_2.next = S.next; S.code = E.code || gen(E.true, ':') || S_1.code || gen('goto', S.next) || gen(E.false, ':') || S_2.code



(III) while - do

Production	Semantic Rules
$S \rightarrow$ while E do S_1	S.begin = newlabel(); E.true = newlabel(); E.false = S.next; S_1.next = S.begin; S.code = gen(S.begin, ':') || E.code || gen(E.true, ':') || S_1.code || gen('goto', S.begin)

Semantic Rule generating three address code for a flow of control statements

$S \rightarrow$ if E then S_1
| if E then S_1 else S_2
| while E do S_1

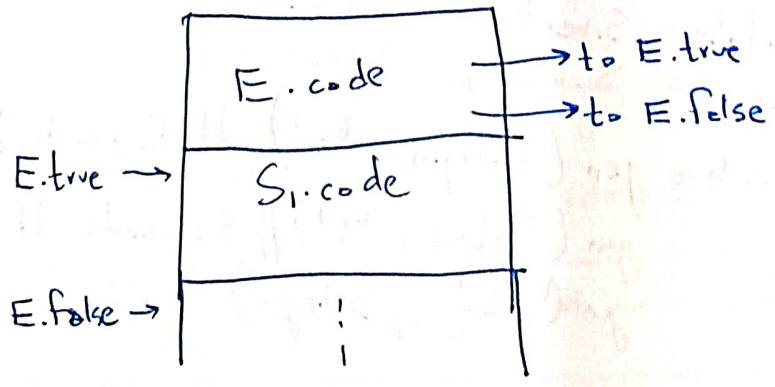
→ we assume that a three address statement can be symbolically labelled and the function newlabel() return a new symbolic label each time it is called

→ we associate two labels

E.true ⇒ The label to which control flows if E is true
E.false ⇒ The label to which control flows if E is false

(I) If – then

Production	Semantic Rules
$S \rightarrow$ if E then S_1	E.true = newlabel ; E.false = S.next ; S_1.next = S.next ; S.code = E.code || gen(E.true, ':') || S_1.code



Indirect Triples

→ Listing of pointers to Triples is maintained by a separate structure

Eg: $a = b^{*} - c + b^{*} - c$

Intermidiate Code	ref	op	arg 1	arg 2
$t_1 = -c$	(14)	uminus	c	
$t_2 = b * t_1$	(15)	*	b	(14)
$t_3 = -c$	(16)	uminus	c	
$t_4 = b * t_3$	(17)	*	b	(16)
$t_5 = t_2 + t_4$	(18)	+	(15)	(17)
$a = t_5$	(19)	=	a	(18)

another table ⇒

actual address	indirect address
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

→ useful due to OS virtualization.

Eg: $a = b^{a} - c + b^{a} - c$

Intermidiate code	op	arg 1	arg 2	res
$t_1 = -c$	uminus	c		t_1
$t_2 = b * t_1$	*	b	t_1	t_2
$t_3 = -c$	uminus	c		t_3
$t_4 = b * t_3$	*	b	t_3	t_4
$t_5 = t_2 + t_4$	+	t_2	t_4	t_5
$a = t_5$	=	t_5		a

(II) Triples

→ It is a record structure with three fields (op, arg 1, arg 2)

→ arg 1, arg 2 → either pointers to symbol table or pointer into Triple Structure

Eg: $a = b^{a} - c + b^{a} - c$

Intermidiate code	refer	op	arg 1	arg 2
$t_1 = -c$	(0)	uminus	c	
$t_2 = b * t_1$	(1)	*	b	(0)
$t_3 = -c$	(2)	uminus	c	
$t_4 = b * t_3$	(3)	*	b	(2)
$t_5 = t_2 + t_4$	(4)	+	(1)	(3)
$a = t_5$	(5)	assign	a	(4)

Implementation of Three-Address Statements

→ A Three Address Code is an abstract form of intermidiate code

→ This can be implemented in the form of records with fields for the operator & operands

→ Three such representation are as follows

(i) Quadruples
(ii) Triples
(iii) Indirect Triples

(I) Quadruples

→ It is a record structure with four fields (op, arg1, arg2, res)

$x = y \; op \; z \Rightarrow (op, y, z, x)$

Eg: $x = -y$ or $x = y \Rightarrow$ we do not use arg2

→ The fields arg1 or arg2 or result are pointers to the symbol table.

Eg:

Q) convert to three address code

(i) a or b and not c

A) t_1 = not c

t_2 = b and t_1

t_3 = a or t_2

(ii) if a < b or c < d and e < f ~~goto~~ then 1 else 0

~~t_1 = a < b~~ ~~c < d~~ e < f

t_2 = c < d

t_3 = t_1 and t_2

t_4 = a < b

~~t_5 = t_4 or t_3~~ t_5 = t_4 or t_3

~~if t_4 or t_5~~

if t_5 == true goto L

t_6 = 1

~~goto Lnext~~

L: t_6 = 0

Lnext:

---

if a < b goto L true

goto L1

t1: if c < d goto L2

goto Lfalse

L2: if e < f goto L true

goto Lfalse

Ltrue: t = 1

goto Lnext

Lfalse: t = 0

Lnext: ;

Syntax Directed Translation into Three Address Code

Production	Semantic Rule
$S \rightarrow id = E$	$S.code = E.code \mid\mid gen(id.place, '=', E.place)$
$E \rightarrow E_1 + E_2$	$E.place = newtemp$ $E.code = E_1.code \mid\mid E_2.code \mid\mid gen(E.place, '=', E_1.place, '+', E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place = newtemp$ $E.code = E_1.code \mid\mid E_2.code \mid\mid gen(E.place, '=', E_1.place, '*', E_2.place)$
$E \rightarrow -E_1$	$E.place = newtemp$ $E.code = E_1.code \mid\mid gen(E.place, '=', 'uminus', E_1.place)$
$E \rightarrow (E_1)$	$E.place = E_1.place$ $E.code = E_1.code$
$E \rightarrow id$	$E.place = id.place$ $E.code = ()$ (initialize)

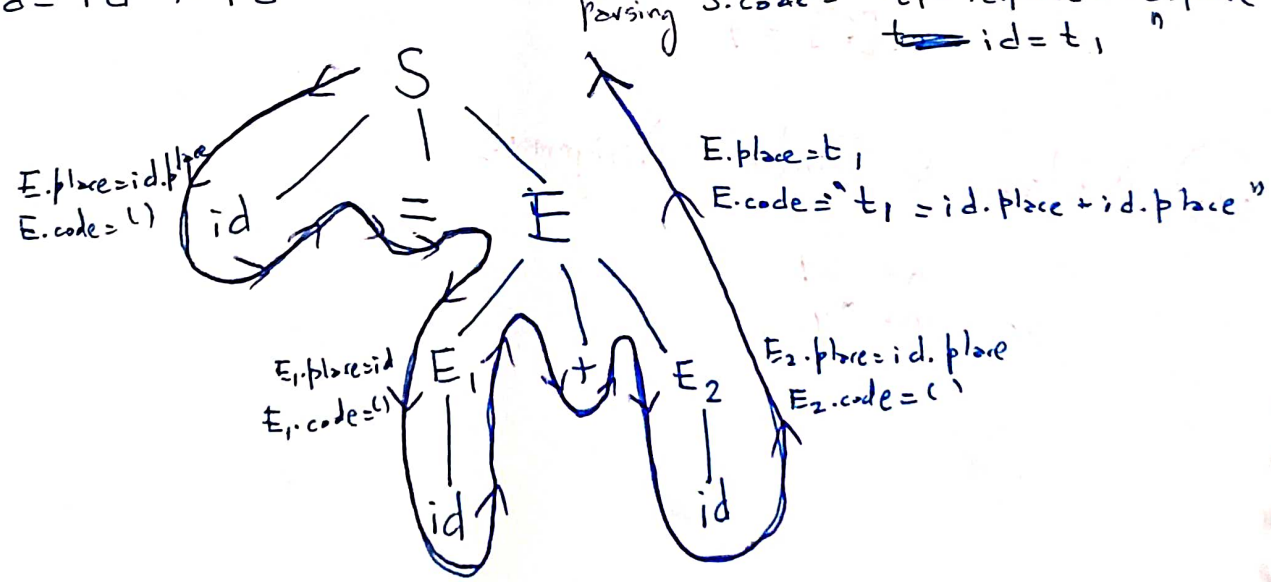
Data Structure where intermediate code being stored

concat

Symbol Table loc

generate string

Eg: id = id + id



(vi) Statement for procedure Call

→ param n → set a parameter for procedure call

→ call p, n → call procedure p with n parameters

→ return y → return from a procedure with return value y (optional)

Eg: ~~call:~~ calling $p(n_1, n_2, \ldots n_n)$

⇓

param n1,
param n2
⋮
param nn
y = call p, n

↖ label in intermidiate code

(vii) Indexed Assignment

n = y[i]

or n[i] = y

(viii) Address & Pointer Assignments

n = &y

or n = *y

(v) if x rel_op y goto L

Eg: if a<b then 1 else 0

⇓

```
100: if a<b goto 103
101: t=0
102: goto 104
103: t=1
104: ;
```

Eg: while a < b do
if c<d then
x = y+z
else
x = y-z

⇓

```
L1 : if a<b goto L2
     goto end_loop
L2: if c<d goto L3
    goto L4
L3: t1 = y+z
    x = t1
    goto L1
L4: t2 = y-z
    x = t2
    goto L1
end_loop: ;
```

Three Address Code

→ Three Address Code is a sequence of statements of general form

$$X = Y \ op \ Z$$

where X, Y, Z → variable / name / const / compiler generated temporary

op → operator

Eg: $x + y * z$

$t_1 = y * z$

$t_2 = x + t_1$

→ Three Address Code is a linearized Representation of Syntax Tree or DAG

General forms of Three Address Statements

(i) $x = y \ op \ z$ (op → binary or logical operator)

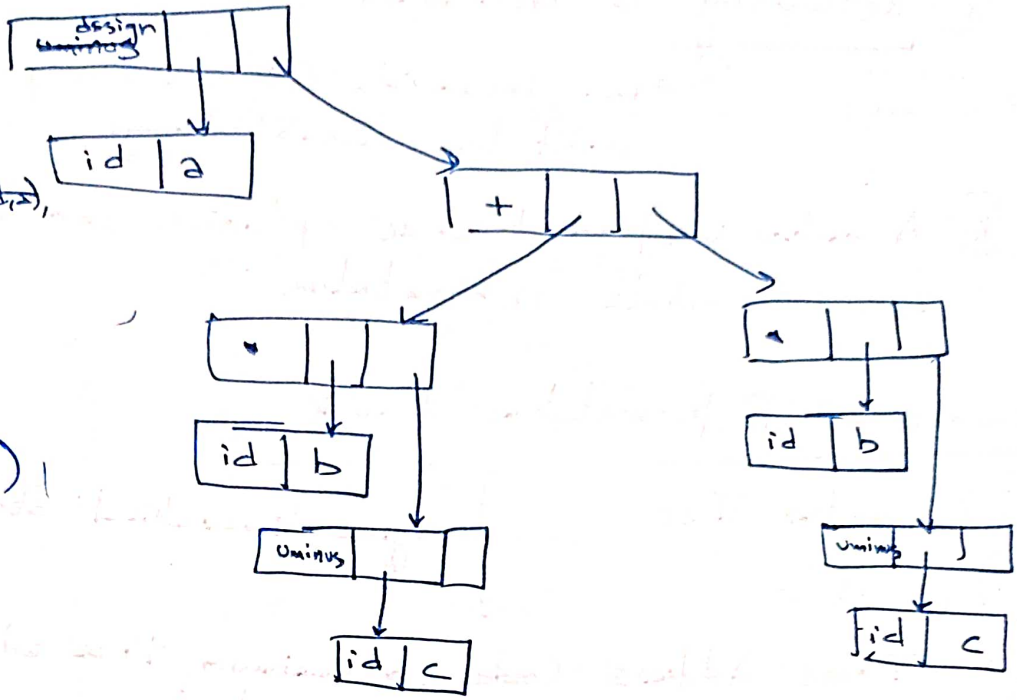
(ii) $x = op \ z$ (op → unary operator)

(iii) $x = y$ (value of y is assigned to x)

(iv) goto L (unconditional Jump)

Eg: $a = b * -c + b * -c$

(I) Syntax Tree



~~$p_1 = mknode(assign, mkleaf(id,a),$~~

$p_1 = mkleaf(id, c)$

$p_2 = mkunode(uminus, p_1)$

$p_3 = mkleaf(id, b)$

$p_4 = mknode(*, p_3, p_2)$

$p_5 = mkleaf(id, c)$

$p_6 = mkunode(uminus, p_5)$

$p_7 = mkleaf(id, b)$

$p_8 = mknode(*, p_7, p_6)$

$p_9 = mknode(+, p_4, p_8)$

$p_{10} = mknode(assign, mkleaf(id,a), p_9)$

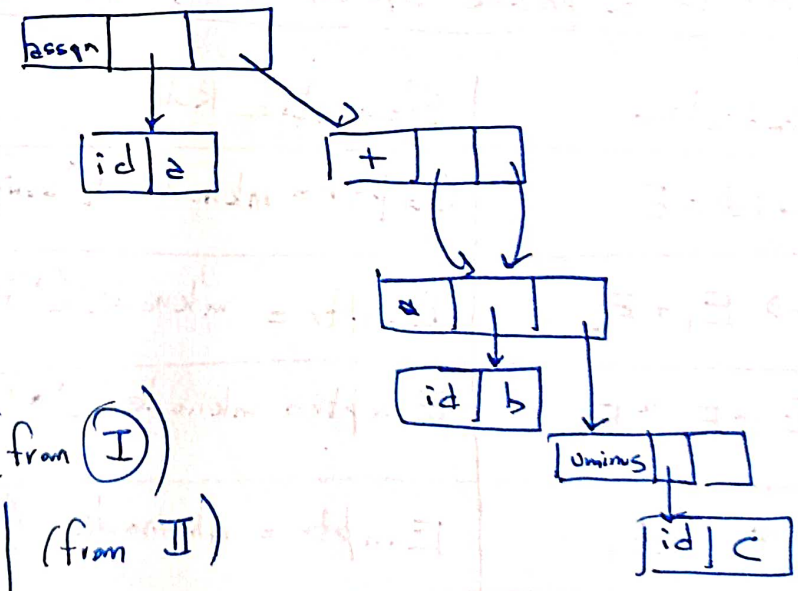
(II) DAG

$p_1 = p_5$

$p_2 = p_6$

$p_3 = p_7$

$p_4 = p_8$



(III) Three Address Code (from (I))

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

(from II)

$t_1 = -c$

$t_2 = b * t_1$

$t_5 = t_2 + t_2$ ~~(t5 to a)~~

$a = t_5$

(Notice Code Reduction)

Intermidiate Code Generation

Benefits of using machine-independant intermidiate form:

(i) Retargeting is facilitated
   → from Intermidiate Code, we can generate different architecture assembly language

(ii) A machine independant code optimizer can be applied to the intermidiate representation.

Intermidiate Representations of code

(i) Syntax Tree
(ii) DAG
   → gives hierarchical structure of the code

(iii) Three Address Code → maximum three addresses used at an instruction
   hence max three registers will be used at a time

(iv) post-fix notation (DSA)

Eg: SDD, to produce Syntax Trees for assignment Statements

Production	Semantic Rule
$S \to id = E$	S.nptr = mknode ("assign", mkleaf(id, id.place), E.nptr)
$E \to E_1 + E_2$	E.nptr = mknode('+', E_1.nptr, E_2.nptr)
$E \to E_1 * E_2$	E.nptr = mknode('*', E_1.nptr, E_2.nptr)
$E \to -E_1$	E.nptr = mkunode('uminus', E_1.nptr)
$E \to (E_1)$	E.nptr = E_1.nptr
$E \to id$	E.nptr = mkleaf(id, id.place)

Type Checking of Statements (=, if else, while) & funct

Productions	Associated ~~type~~ rules for types
$S \rightarrow id = E$	{ S.type = if id.type == E.type then void else type_error }
$S \rightarrow$ if E then S_1	{ S.type = if E.type == bool then S_1.type else type_error }
$S \rightarrow$ while E do S_1	{ S.type = if E.type == bool then S_1.type else type_error }
$S \rightarrow S_1 ; S_2$	{ S.type = if S_1.type == void && S_2.type == void, then void else type_error }
$T \rightarrow T_1 ' \rightarrow ' T_2$	{ T.type = T_1.type $\rightarrow$ T_2.type }
$E \rightarrow E_1 (E_2)$	{ E.type = if E_2.type = s && E_1.type == s $\rightarrow$ t then t else type_error }

Productions	Associated rule for types
$P \rightarrow D ; E$	
$D \rightarrow D ; D$	
$D \rightarrow id : T$	{ add type (id.entry, T) }
$T \rightarrow char$	{ T.type = char }
$T \rightarrow integer$	{ T.type = integer }
$T \rightarrow \uparrow T_1$	{ T.type = pointer (T_1.type) }
$T \rightarrow array [num] of T_1$	{ T.type = array (1 ... num.val, T_1.type) }
$E \rightarrow literal$	{ E.type = char }
$E \rightarrow num$	{ E.type = integer }
$E \rightarrow id$	{ E.type = lookup (id.entry) }
$E \rightarrow E_1 mod E_2$	{ E.type = if E_1.type = integer and E_2.type = integer ~~then integer~~ then integer else type_error }
$E \rightarrow E_1 [E_2]$	{ E.type = if E_2.type = integer and E_1.type = array (s, t) then t else type_error }
$E \rightarrow E_1 \uparrow$	{ E.type = if E_1.type = pointer (t) then t else type_error. }

Functions

→ Mathematically, a function maps elements of one set, the domain, to another set, the range.

→ We may treat functions in a programming language as mapping a domain type D to a range type R

→ The type of such function will be denoted by D → R

Eg: function f (a,b : char) : ↑ integer ;

⇓

type of f ⇒ char x char ⟶ ~~integer~~ pointer (integer)

Specification of a simple Type checker

P → nonterminal
D → declaration
E → expression

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↑

(II) Products

→ If T_1 & T_2 are type expressions, their Cartesian product $T_1 \times T_2$ is also a type expression

(III) Records

→ The record type constructor will be applied to a tuple formed from field name & field types

Eg: type row = record
address : integer;
lexeme : array[1 .. 15] of char
end;

var table: array[1 ... 10] of row

⇓

(i) declares ~~row~~ the name row representing type expression

record ((address x integer) x (lexeme x array(1 ... 15, char)))

(ii) The variable table to be an array of records of this type

~~(IV Fun~~

(IV) Pointer

→ If T is a type expression, then pointer(T) is a type expression denoting the type "pointer to an object of type T"

Eg: var p = ↑ row declareds variable p to have type pointer(row)

Type Expressions

→ The type of a language construct will be denoted by a "type expression"

→ few basic type expressions
  → boolean, char, int, real
  → A special basic type: type_error will signal an error during type checking
  → void : absence of value
  → Type expression may be named, a "type_name" is a type expression
  → A type constructor applied to type expression is also a type expression. Constructors include:
    → Arrays
    → Products
    → Records
    → Pointers
    → Functions

(I) Arrays

→ If T is a type expression, than array (I, T) is a type expression denoting type of an array with elements of type T & index set I. ~~I is often 0 to~~

Eg: ~~int A[]~~ =
var A: array [1 ... 10] of integer (code)

↓

array (1 ... 10, integer) (type expression)

Static Type Checking

→ Static Type checking is done at compile Time. The information type checker needs is declared via declarations & stored in a master symbol table

→ after this information is collected, the types involved in each operation are checked.

Eg:
```
int a = 1e10;
int b = 1e10;
int c = a * b;   ——→ overflow error not detected in compile time
int d = 20;
int e = b / d;   ——→ divide by 0 error not detected at compile time.
```

Dynamic Type Checking

→ Dynamic Type Checking is implemented by including type information for each data location at runtime

Eg: a variable of type double would contain both the actual double value & some kind of tag indicating "double type"

→ The execution of any operation begins by first checking these type tags.

→ The operation is performed only if everything checks out. Otherwise, a type error occurs. & usually halts execution.

Type Checking

→ Type Checking is the process of verifying that each operation executed in a program respects the type system in our language

→ This generally means all operands in any expression are of appropriate types & numbers.

→ Mostly what we do in semantic analysis phase is type checking.

Designing a Type Checker

→ When designing a type checker for a compiler, we

(i) Identify the type that are available in the language

(ii) Identify the language construct that have types associated with them

(iii) Identify Semantic rules for the language.

(iv) If problem found (Eg: char + double), we encounter type error.

→ A language is considered strongly-typed if each & every type error is detected during compilation

→ Type checking can be done in compile time or in execution time.

Eg: " 3*5+4n "

Input	State	val	Production used
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
	En	19	
	L	19	L → En

Bottom-up evaluation of Inherited Attributes

→ A translator for an S-attribute definition can often be implemented with the help of an LR parser

→ from an S-attribute definition, the parser generator can construct a translator that evaluates attributes as it parses input

→ We put the values of the synthesized attributes of the grammar symbols on a stack that has extra fields to hold the values of attributes.

Eg: LR parser calculator

production	Code Fragment
L → En	print (val[top]);
E → E₁ + T	val[ntop] = val[top-2] + val[top]
E → T	
T → T₁ * F	val[ntop] = val[top-2] * val[top]
T → F	
F → (E)	val[ntop] = val[top-1]
F → digit	

Directed Acyclic Graph for Expression

→ In DAG, common subexpression can have more than one parent

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

(I) using prev Syntax Directed Definition for making syntax tree

p_1 = mkleaf (id, a)

p_2 = mkleaf (id, a)

p_3 = mkleaf (id, b)

p_4 = mkleaf (id, c)

p_5 = mknode ('-', p_3, p_4)

p_6 = mknode ('*', p_2, p_5)

p_7 = mknode ('+', p_1, p_6)

p_8 = mkleaf (id, b)

p_9 = mkleaf (id, c)

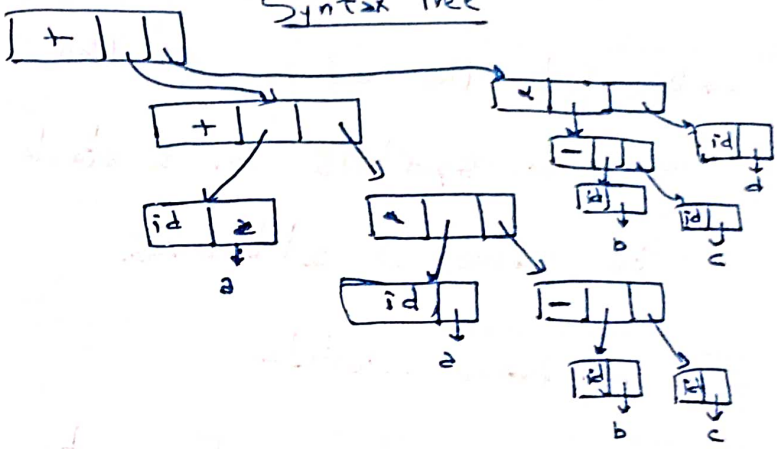
p_{10} = mknode ('-', p_8, p_9)

p_{11} = mkleaf (id, d)

p_{12} = mknode ('*', p_{10}, p_{11})

p_{13} = mknode ('+', p_7, p_{12})

Syntax Tree



(II) Reusability

$p_1 = p_2$

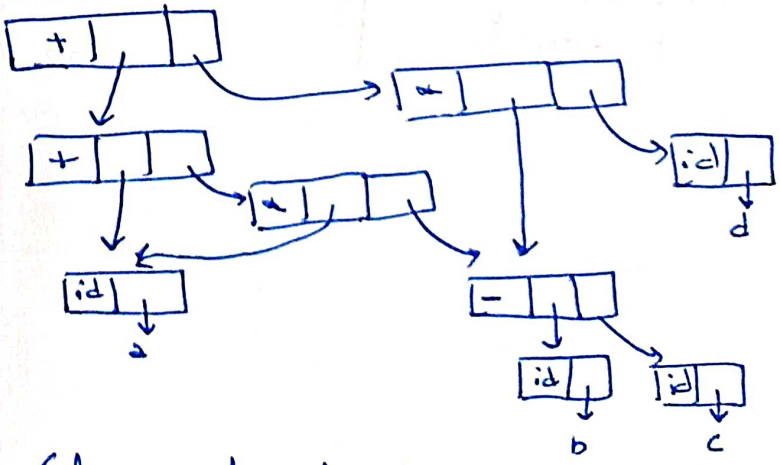
$p_7 = p_3$

$p_4 = p_9$

$p_5 = p_{10}$

⇒ DAG

DAG



(less nodes due to memory reuse)

→ Functions used to create nodes of syntax tree for expressions with binary operator

(i) mknode (op, left, right)
↳ operator

(ii) mkleaf (id, entry)
↳ entry of symbol table
↳ id name

(iii) mkleaf (num, val)
↳ const value
↳ const type

each fn returns a pointer to newly created node.

Eg: a - 4 + c

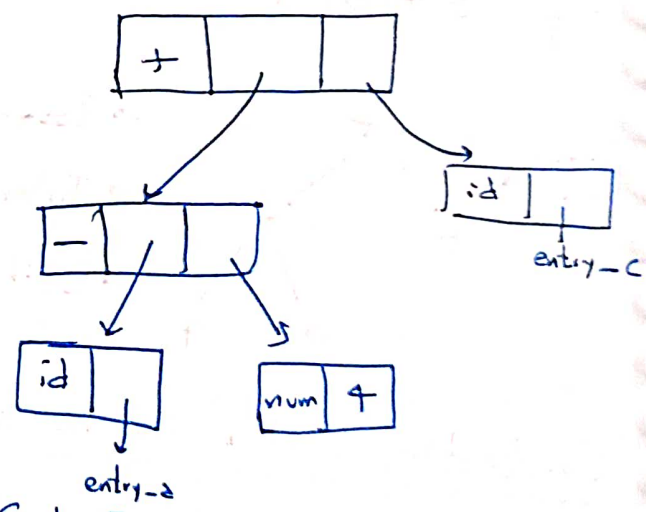
p_1 = mkleaf (id, entry_a);

p_2 = mkleaf (num, 4);

p_3 = ~~mkleaf~~ mknode ('-', p_1, p_2);

p_4 = mkleaf (id, entry_c);

p_5 = mknode ('+', p_3, p_4);



Syntax - Directed Definition for Constructing Syntax Tree

Production	Semantic Rule
$E \rightarrow E_1 + T$	E.nptr = mknode ('+', E_1.nptr, T.nptr)
$E \rightarrow E_1 - T$	E.nptr = mknode ('-', E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr = T.nptr
$T \rightarrow (E)$	T.nptr = E.nptr
$T \rightarrow id$	T.nptr = mkleaf (id, id.entry)
$T \rightarrow num$	T.nptr = mkleaf (num, num.val)

Syntax Tree

→ An intermidiate representation of the compiler's input

→ A condensed form of parse tree

→ Syntax Tree shows syntactic structure of program while omitting the irrelevant details

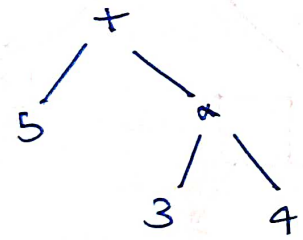
→ operator or keywords are associated with interior nodes

→ Chains of simple productions are collapsed

✱ Syntax Directed Translation can be based on syntax tree as well as parse tree.

Eg "5 + 3 * 4"

Parse Tree →



→ leaves → identifiers or const.

→ Internal node → operations

→ children of node (operator) are operands.

Constructing Syntax Tree for Expression

→ Each node can be implemented as a record with several fields

→ Operator node → one field identifies the operator (called label)
→ remaining field contains pointer to operand.

→ node may also contain fields to hold the values (pointer to values) of attribute attached to node

Evaluation Order of Semantic Rules

(I) Parse Tree Method

→ At compile time evaluation order obtained from dependency graph constructed from the parse tree

→ Fails if dependency graph contains a cycle

(II) Rule Based Method

→ Semantic Rules analyzed by hand or by specialized tools at compiler construction time

→ Order of evaluation of attributes associated with a production is pre-determined at compiler construction time.

(III) Oblivious Methods

→ Evaluation Order is chosen without considering the semantic rules

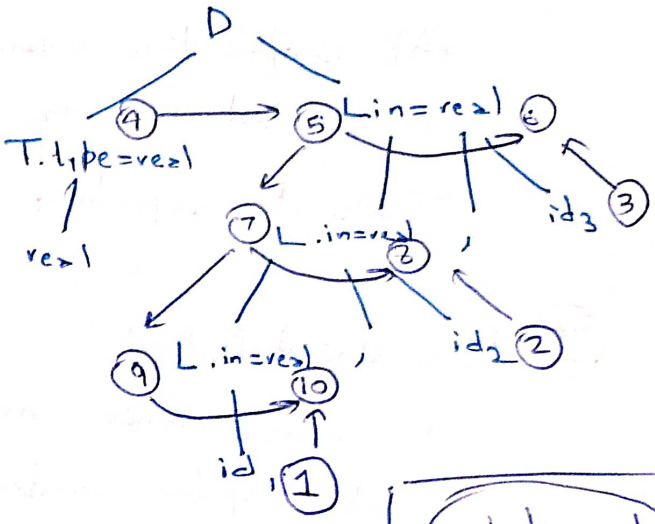
→ Restricts the class of syntax directed definitions that can be implemented.

→ Order of evaluation is forced by parsing method.

Eg (Inherited) "real, id_1, id_2, id_3"

Production	Semantic Rule
$D \rightarrow TL$	L.in = T.type
$T \rightarrow int$	T.type = int
$T \rightarrow real$	T.type = real
$L \rightarrow L_1, id$	L_1.in = L.in dummy = addtype(id.entry, L.in)
$L \rightarrow id$	dummy = addtype(id.entry, L.in)

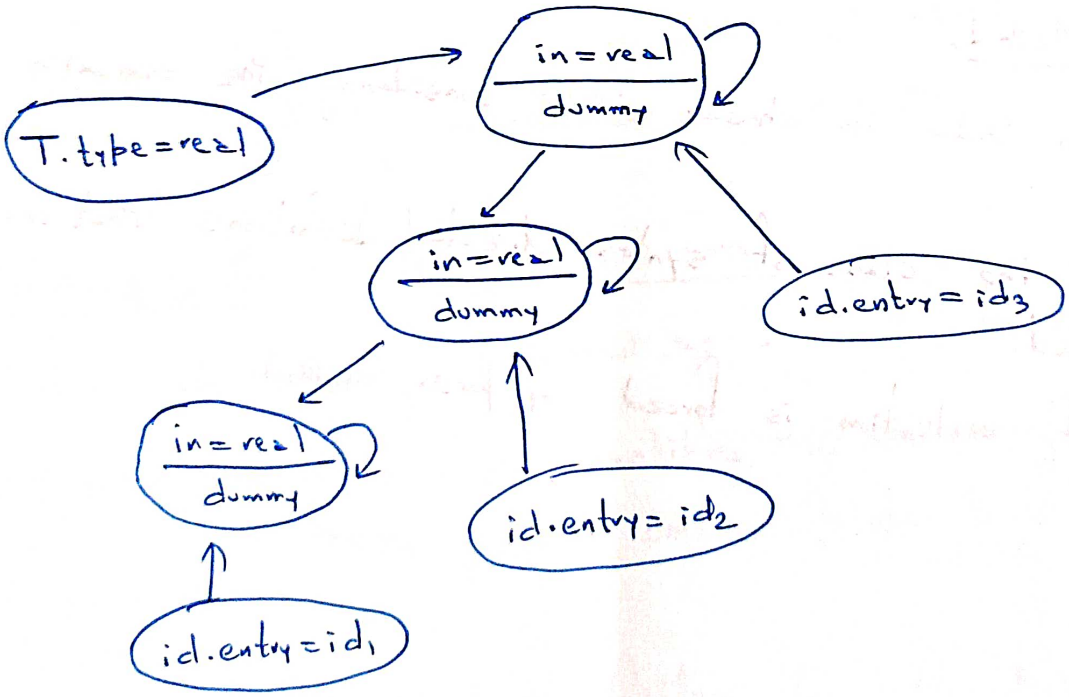
Annotated Parse Tree



evaluation order

a_4 = real
a_5 = a_4
addtype(id_3, a_5)
a_7 = a_5
addtype(id_2, a_7)
a_9 = a_7
addtype(id_3, a_9)

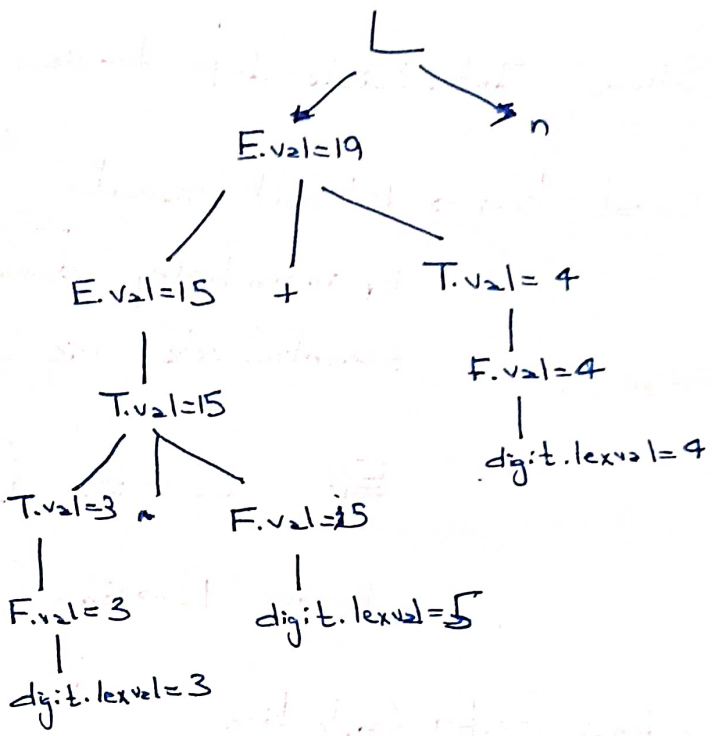
Dependency Graph



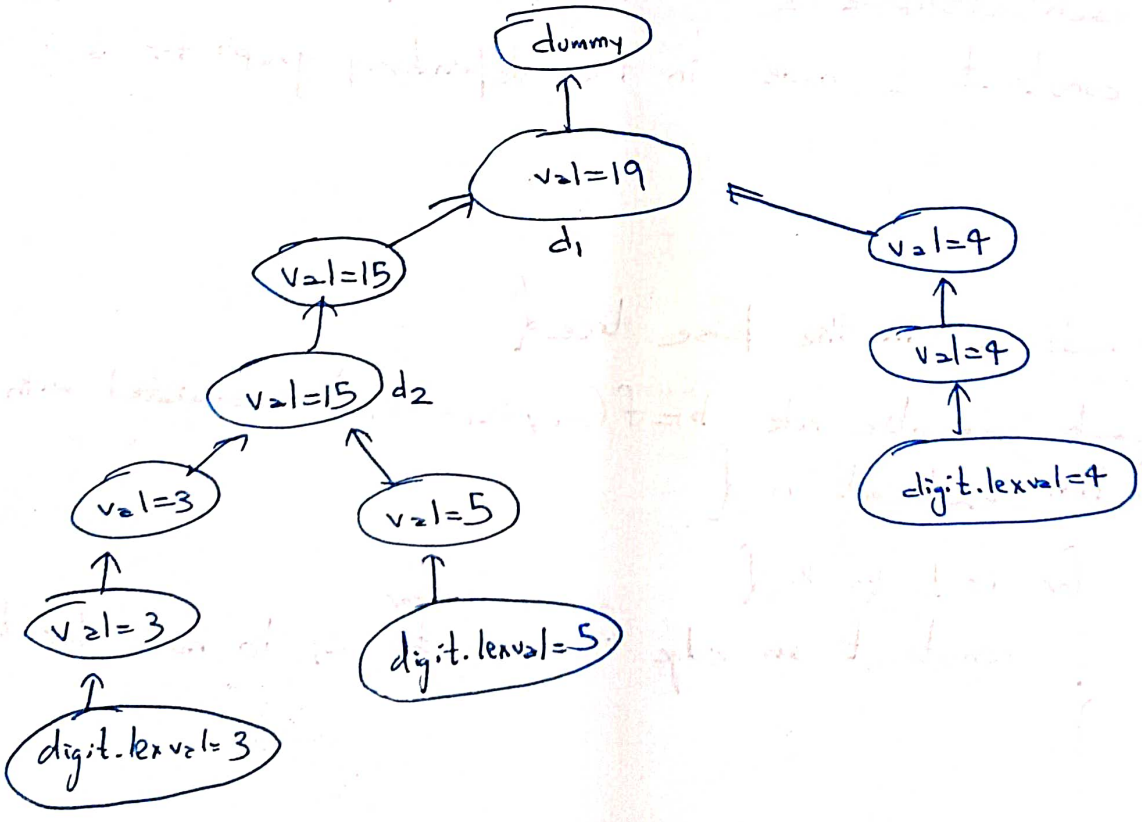
Eg: (Synthesized) "3*5+4n"

Production	Semantic Rules
L → En	print("E.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

Annot Parse Tree



Dependency Graph



Dependency Graph

→ Directed Graph

→ Shows Intermidiate dependencies b/w attributes

→ Construction → Put each semantic rule into the form $b = f(c_1, c_2, \cdots c_k)$ by introducing dummy synthesized attribute b for every semantic rule the consists of a procedure call

Eg: $L \to En$ { print ("E.val") }

becomes $L \to En$ { dummy = print ("E.val"); }

Dependency Graph Construction

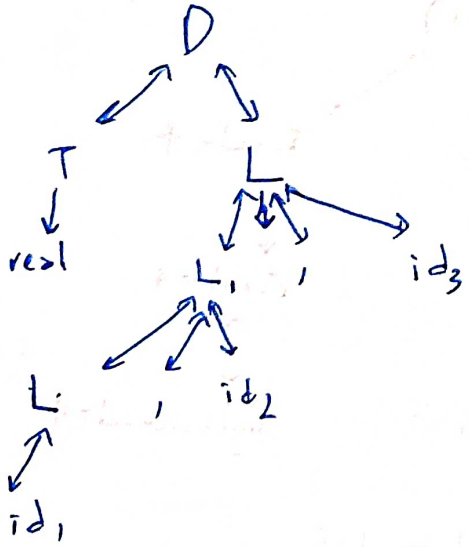
```
for each node n in the parse tree {
    for each attribute a of the grammar symbol at node n {
        construct a node in the dependency graph for a ;
    }
}

for each node n in the parse tree {
    for each semantic rule b = f(c1, c2, ... ck) associated with the
    production used at n {
        for i = 1 to k {
            construct an edge from node for ci to node for b;
        }
    }
}
```

Eg: Inherited (Grammar for int/real $id_1, id_2, \ldots$)

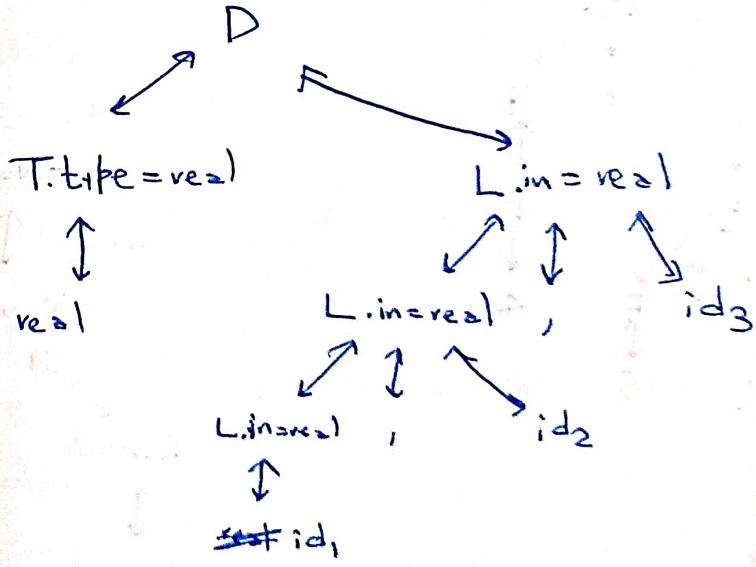
Production	Semantic Rules
$D \rightarrow TL$	$L.in = T.type$ (inherit)
$T \rightarrow int$	$T.type = int$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in$ (L & L' are same) addtype (id.entry, L.in) entry in symbol table
$L \rightarrow id$	addtype (id.entry, L.in)

Eg " real id_1, id_2, id_3 "

parse tree



Annotated Parse Tree



Eg:

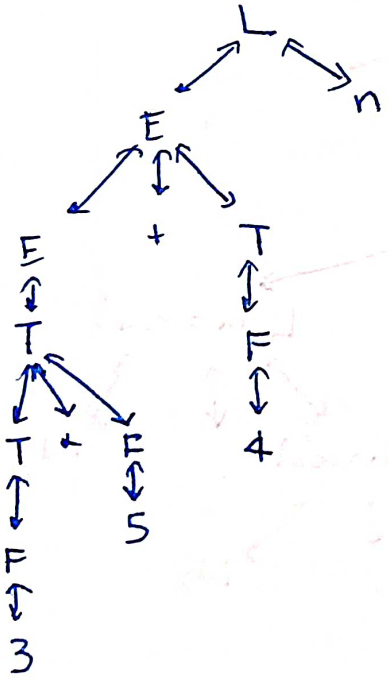
(Synthesized)

Productions	Semantic Rules
L → En	print ("E.val");
E → E+T	E.val = E.val + T.val
E → T	E.val = T.val
T → T*F	T.val = T.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit.lexval

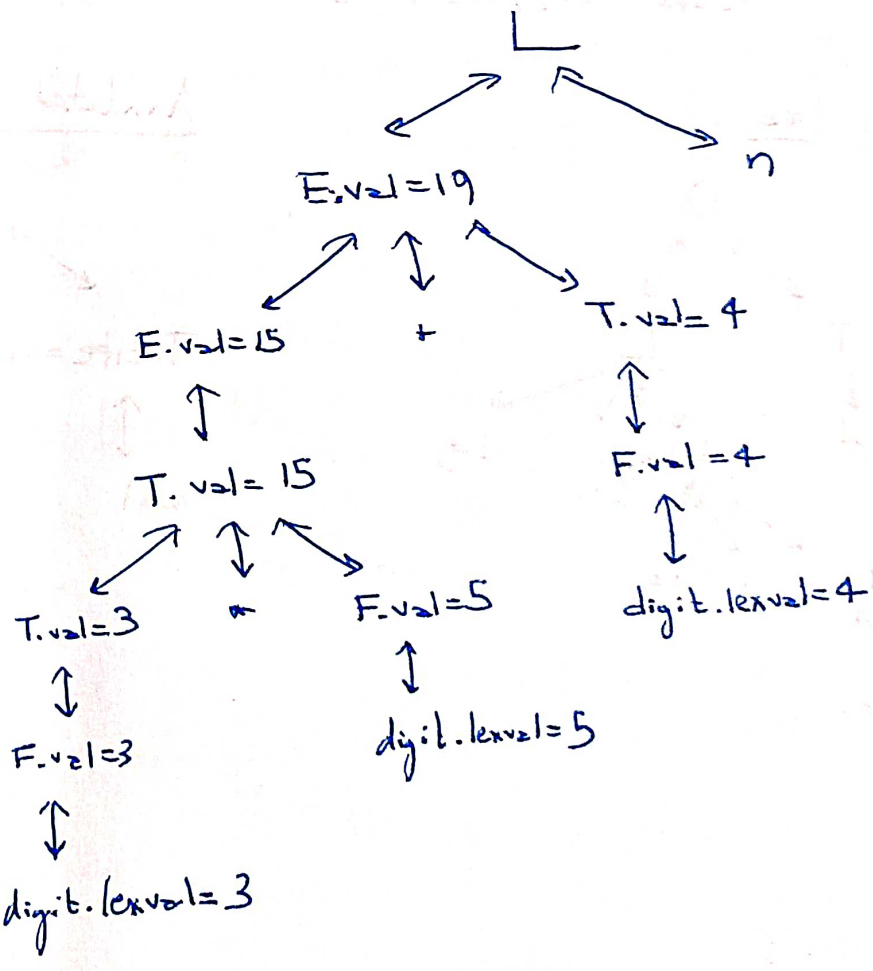
lexval → provided by lexical analyzer

Eg: 3 * 5 + 4n

Parse Tree



Annotated Parse Tree



Annotated Parse Tree

→ A parse tree showing the values of attributes at each node is called an annotated parse tree

→ Values of attribute in nodes of annotated parse tree are either
  (i) initialized to constant value or by the lexical analyzer
  (ii) determined by semantic rules

→ The process of computing the attributes values at the nodes is called annotating or decorating of parse tree.

→ The order of these computations depends on the dependency graph induced by the semantic rule

Syntax Directed Definition

→ In a syntax directed definition, each production $A \to \alpha$ is associated with a set of semantic rule of the form

$$b = f(c_1, c_2, \dots c_k) \quad [f \text{ is a function}]$$

→ b can be one of the following:
  (i) b is a synthesized attribute of A & $c_1, c_2 \dots c_k$ are attribute of the grammar symbol in α
  (ii) b is an inherited attribute of one of the grammar symbols on the right side of production rule $A \to \alpha$ & $c_1, c_2, \dots c_k$ are attributes of grammar symbol in $\{A, \alpha\}$

→ 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 sibling & parents in that node of the parse tree.

Eg: (i) Synthesized Attributes

$E \rightarrow E_1 + E_2 \quad \{ E.val = E_1.val + E_2.val \}$

(Parent val using children)

(ii) Inherited Attributes

$A \rightarrow X\ Y\ Z \quad \{ Y.val = 2 * A.val \}$

(child val. using parent)

→ Semantic Rules Setup & dependencies between attributes can be represented by a dependency graph

→ Dependency graph determines the evaluation order of these semantic rules

→ Evaluation of a semantic rule defines the value of an attribute. A semantic rule may also have some side effect like printing a value

Syntax Directed Translations

→ Conceptually with both Syntax Directed Definition & Translation Schemes, we

(i) parse the input token stream

(ii) Build Parse Tree

(iii) Traverse the tree to evaluate the semantic rules at the parse tree nodes.

Input Stream → Parse Tree → Dependency Graph → Evaluation Order for Semantic Rule

Syntax Directed Definition

→ A Syntax Directed Definition is a generalization of CFG in which:

- → Each Grammar symbol is associated with a set of attribute
- → This set of attributes can be classified as
  - (i) Synthesized Attributes
  - (ii) Inherited Attributes
- → 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