Unit-3

Evaluation Orders of SDD

SDT schemes

Three address codes types - quadruples

triples, indirect triples

Switch statements

Intermediate code for procedures.

Evaluation orders of SDD:-

→ Syntax Directed Definition.

synthesized / inherited

Dependency graphs - determining an evaluation order for the attribute instances in a parse tree.

Dependency graphs:-

node attribute node attribute

X.a X.b

their attribute will have some dependency node.

* for each parse-tree node X the graph has a node for each attribute associated with a node 'X'.

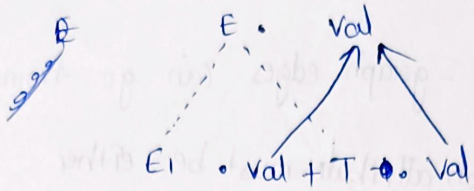
* A semantic rule associated ~~as~~ a with a production defines the value of synthesized attribute A.b in terms of X.c the dependency graph has an edge from X.c to A.b

* A semantic rule associated with a production defines the value of inherited attribute B.c in terms of X.a, then the dependency graph has an edge from X.a to B.c

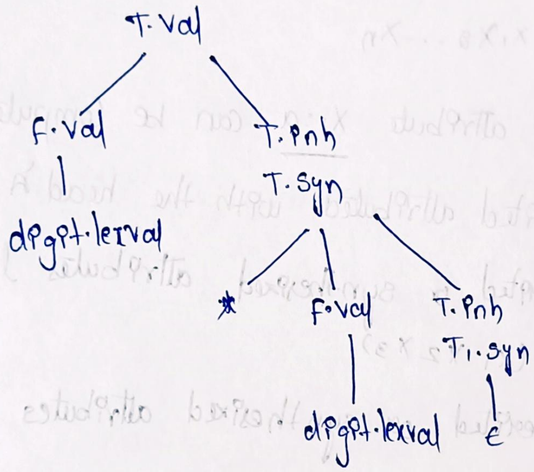
Partial dependency graph.

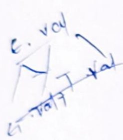
Synthesized attribute $E.val$

Production	Semantic value
$E \rightarrow E_1 + T$	$E.val \rightarrow E_1.val + T.val$



Complete depency graph





M
↓
N.

Ordering the evaluation of Attribute:

If the dependency graph has an edge from node M to node N, then the attribute of 'M' must be evaluated before the attribute of 'N'.

no entering edges

It follows the Topological sort of the graph, if there are no cycle.

S-attributed Definitions :-

An SDD is S-attributed if every attribute is synthesized.

Attributes are evaluated in any bottom-up order of the nodes of parse tree. It is simple to evaluate the attributes by post order parse tree.

L-Attribute Definitions

The dependency-graph edges can go from left to right, but not right to left. Each attribute must be either

1. synthesized (or)
2. Inherited

eg: $A \rightarrow X_1 X_2 \ldots X_n$

Then the inherited attribute $X_j.a$ can be computed

a) from inherited attributed with the head 'A'. $(X_4.a)$

b) from inherited or synthesized attributes located to the left of X_i $(X_1 \cdot X_2 X_3)$

c) from inherited or synthesized attributes associated with X_i itself.

example

Production	Sematic Rule
$T \rightarrow FT'$	$T'.inh = F.val$
$T' \rightarrow * F T_1'$	$T_1'.inh = T'.inh \times F.val$

Syntax Directed Translation schema :-

SDT = Grammer + Semantic rules

* The syntax Directed Translation scheme is a context free grammer.

* It evaluates the order of semantic rules.

* In translation scheme, the semantic rules are embedded within the right side of the production

* The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production

E → { }

Example

production	Semantic Rules / Actions
$S \rightarrow E\$$	{ print E.value }
$E \rightarrow E + E$	{ E.val := E.val + E.value }
$E \rightarrow E * E$	{ E.val := E.val * E.value }
$E \rightarrow (E)$	{ E.val := E.val }
$E \rightarrow I$	{ E.val := I.val }
$I \rightarrow I \cdot digit$	{ I.val = 10 × I.val + lexval }
$I \rightarrow digit$	{ I.val := lexval }

Implementation of syntax directed translation

* SDT is implemented by constructing a parse tree and performing the actions in a left to right depth first order.

a + b * c

* SDT is implementing by parse the input and a parse tree as a result.

Example:-

$E \longrightarrow E+T$ { E.val := E.val + T.val }

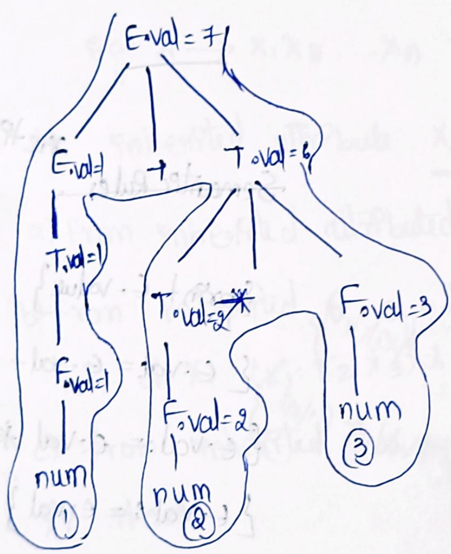
$E \longrightarrow T$ { E.val := T.val }

$T \longrightarrow T * F$ { T.val := T.val * F.val }

$T \longrightarrow F$ { T.val := F.val }

$F \longrightarrow num$ { F.val := num.lexval }

1 + 2 * 3



Top down

left to right

Three address codes types - quadruples

* It is an intermediate code. It is used by optimizing compilers.
* In three address code, the given expression is broken down into several seperate instructions. These instruction can easily translate into Assembly language.

* Each three address code instruction has at most three operands. It is a combination of assignment and a binary operator

In TAC, there is at most one operator on the right side of an instruction

eg:-

$x + y * z$

$t_1 = y * z$ t_1 & t_2 are compiler generated temporary names.

$t_2 = x + t_1$

eg:- $a + a * (b-c) + d * (b-c)$

~~$t_1 = b - c$~~

~~$t_2 = a * t_1$~~

~~$t_3 = d * t_1$~~

~~$t_4 = t_2 + t_3$~~

~~$t_5 = a + t_2 + t_3$~~

$x + y * z$

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = d * t_1$

$t_5 = t_3 + t_4$

① quatriples ⇒ 4 fields — operator, source 1, source 2, destination

② Triples ⇒ 3 fields — operator, source 1, source 2

eg:- $a := -b * c + d$

TAC:-

$t_1 = -b * c$

$t_2 = t_1 + d$

$a := t_2$

(or)

$t_1 = -b$

$t_2 = c + d$

$t_3 = t_1 * t_2$

$a := t_3$

Quatruples

	operator	source1	source2	destination
(0)	minus	b		t_1
(1)	plus	c	d	t_2
(2)	into	t_1	t_2	t_3
(4)	equal to	t_3	—	a

eg:- $a := -b * c + d$

TAC:-

$t_1 = -b$

$t_2 = c + d$

$t_3 = t_1 * t_2$

$a := t_3$

triples

	operator	source1	source2
(0)	minus	b	-
(1)	plus	c	d
(2)	into	(0)	(1)
(3)	equalto	(2)	-

Switch-statement:-

type of selection control statement that allows the value of a variable to change the control flow of program execution

Eg:-

```
switch(E) {
    case V1: S1
    case V2: S2
      :
      :
    case Vn-1 : Sn-1
    default: Sn
}
```

Translation of switch statements:-

* Evaluate the Expression E
* Find the value Vj in the list of cases
* Execute the statement.

step(2) - implemented as

* Sequence of conditional jumps
  ↳ Create table of pairs(value, label)
* Hash table for the values.

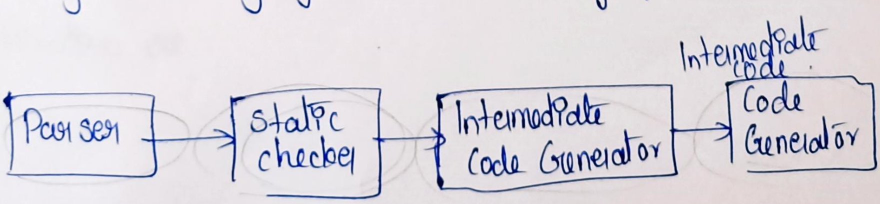
Syntax-Directed Translation of switch-statements-

```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        ...
Ln-1:   code for Sn-1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn-1 goto Ln-1
        goto Ln
```

~~Another translation of a~~ switch statements:-

Intermediate code

It is used to translate the source code into machine code. It lies between high level language and machine language



* If the compiler directly translates source code into machine code without generating intermediate code then a full native compiler is required for

each new machine.

* Intermediate code generator receives input from its predecessor phase and semantic analyzer phase. It takes input in the form of annoted syntax tree

* Using intermediate code, the second phase of the compiler is changed according to the machine.

Intermediate code can be represented in two ways

High level Intermediate code

It can represented as Source code

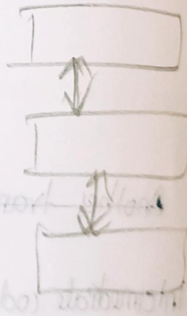
low level Intermediate code

* It is close to the target machine

It is used for machine independent optimization.

The different forms of Intermediate code

1) Abstract syntax tree
2) polish notation
3) Three Address code.



Unit - IV

1) stack allocation
2) acces to non local data
3) Heap management
4) Basic blocks and flow graphs
5) optimization of basic blocks
6) peephole optimization
7) Register allocation and assignment.

1) Storage Allocation:- — static storage allocation
- stack " "
- Heap " "

Static storage Allocation:-

* In static allocation, names are bound to storage locations.
* If memory is created at compile time then the memory will be created in static area and only once

Stack allocation:-

Storage is organised as stack  last in first out

local values are deleted after pop

Activation records are pushed and popped

Activation re