10/05/22.

Sessional 2

COMPILER DESIGN (CEN603)

YASH VINAYVANSHI
19BCS081

Q1. The concept of Attributes.

→ Semantics or MEANING of a program can be viewed as computation of value of attributes associated with the non terminals in grammar.

→ Grammar symbols are associated with attributes (records of temporary variables in Memory) to associate information with programming language constructs that they represent

→ The values of these attributes are evaluated by semantic rules associated with the productions of grammar in syntax directed translation.

→ An attribute associating to variable can hold multiple types of information about that variable.

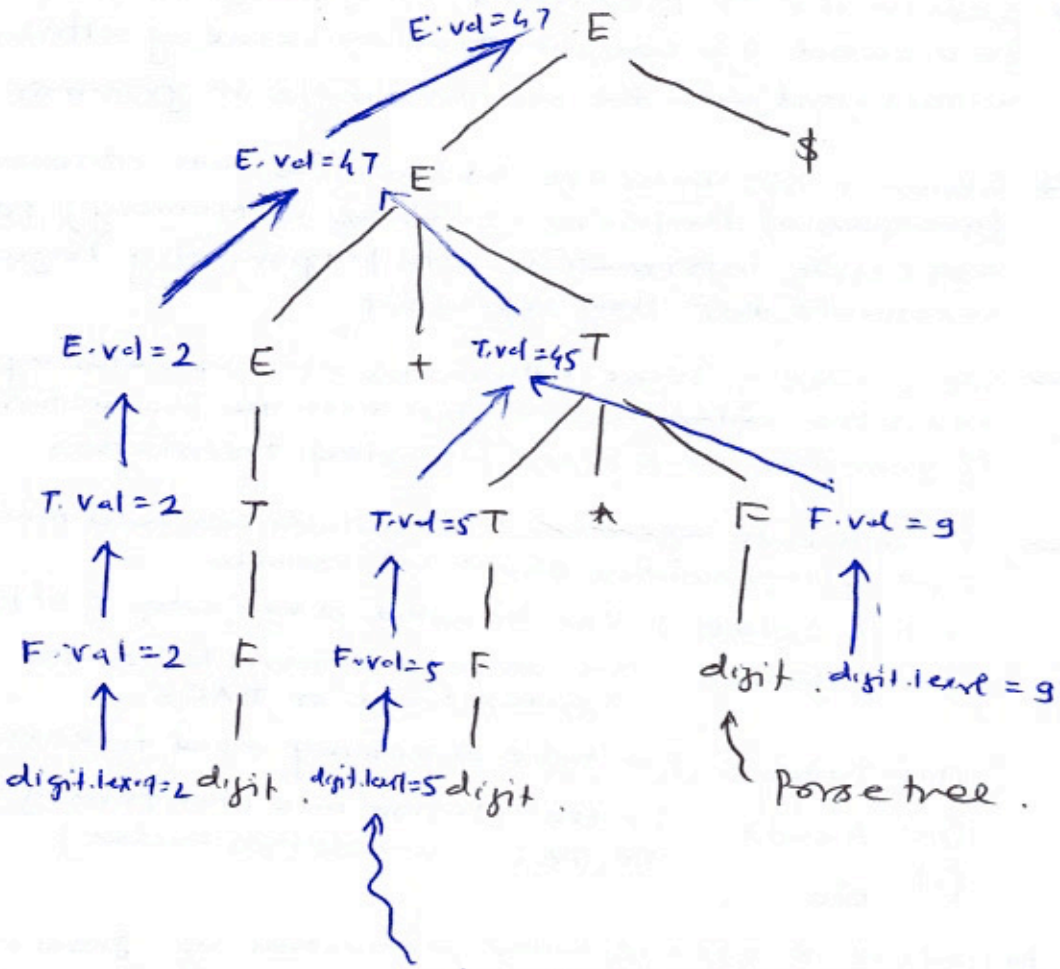
(i) A value : can be used in evaluation of expression
(ii) A pointer : can be used in translation to syntax trees or DAGs.
(iii) Type : For type checking and typecasting
(iv) A code : To translation into other form of code like TAC (3-address codes)
(v) etc.

Eg. In following syntax directed definition of grammar for desk calculator.

PRODUCTION	SEMANTIC RULES
$L \rightarrow E\$$	print (E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

Alongside parsing, we can also evaluate the input expression simultaneously by applying semantic rules at each stage and <u>data flow among attributes at nodes in the parse tree.</u>

eg Input : 2 + 5 * 9.



dataflow graph among attributes associated to variables in parse tree for simultaneous evaluation of expression along with parsing

S-Attributed definitions

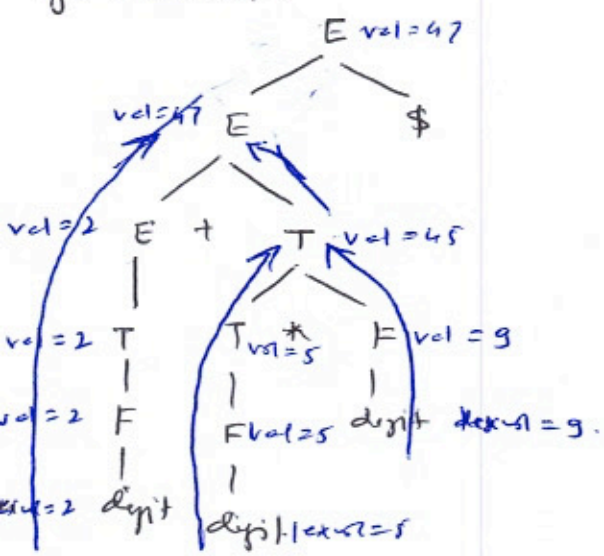
→ Based on synthesized attributes only.
(data flows from children to parent only)

→ use bottom up parsing

→ Example

$L \to E\$$	print(E.val)
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to digit$	$F.val = digit.lexval$

eg. 2+5*9$



Bottom up flow only.

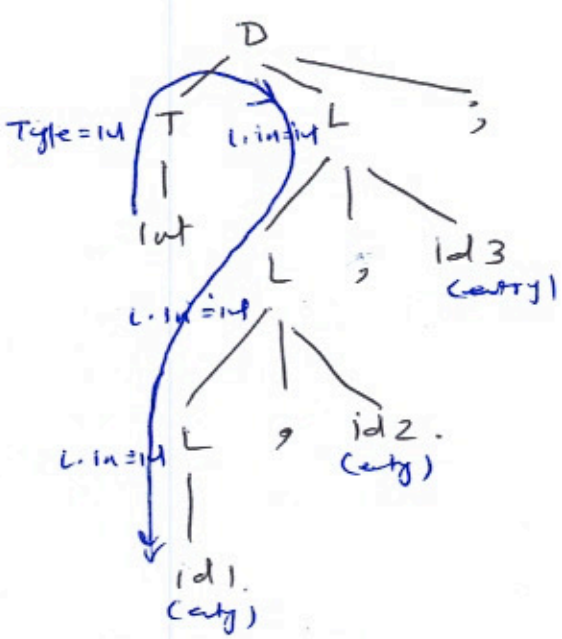
L-Attributed definitions

→ based on synthesized attributes as well as inherited attributes.
(data flows from children to parent
or
data flows from parent to children
data flows from sibling to sibling
in the flow graph)

→ Arbitrary traversals or top down parsing required to establish flow.

→ Example

$D \to TL;$	$L.in = T.type$
$T \to int$	$T.type = integer$
$T \to real$	$T.type = real$
$T \to char$	$T.type = character$
$L \to L_1, id$	$L.in = L.in$ addtype(id.entry, L.in)
$L \to id$	addtype(id.entry, L.in)



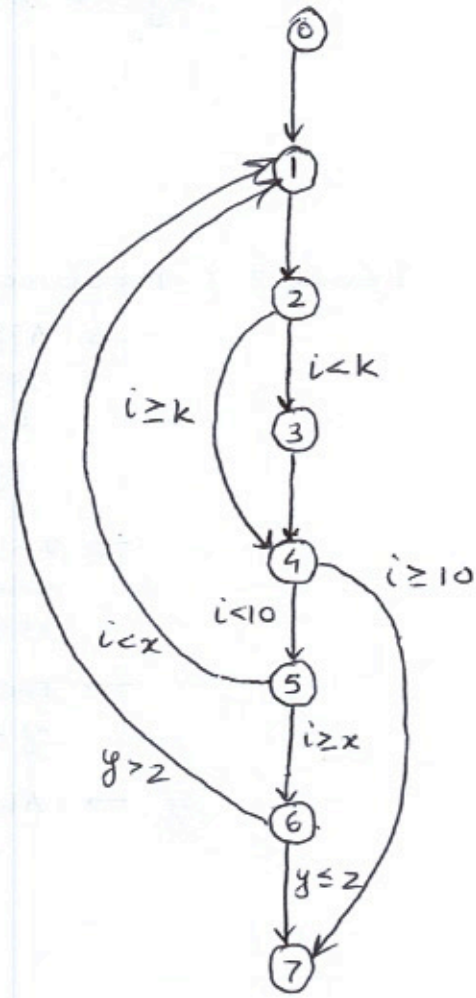
flow can be left to right
Top to bottom
or Bottom up.

Q2.

```
x = y * z;
i = 0;
do {
    k = x + i;
    i++;
    if (i < k) {
        a = a * b / k
    }
} while (i<10 && (i<x || y>z));
```

Three address code for above snippet. (Non optimized)

```
100 : t1 = y + z
101 : x = t1.
102 : i = 0
103 : t2 = x + i
104 : k = t2.
105 : t4 = i + 1
106 : i = t4.
107 : if i < k goto 109.
108 : goto 112
109 : t5 = b / k
110 : t6 = a * t5
111 : a = t6.
112 : if i < 10 goto 114
113 : goto 118
114 : if i < x goto 103.
115 : goto 116.
116 : if y > z goto 103.
117 : goto 118
118 : ———
```



Q3. various levels on compiler on which code can be optimized.

Level 1 : Machine independent optimizations.

→ These can be applied in frontend of compiler. ie LA, SA, SemA, ICG.

→ These optimizations are inefficient to apply in LA, SA or SemA phases and thus only attempted in intermediate code generation phase.

→ The part of intemediate code which is improved here doesnt involve any CPU registers or absolute memory locations.

eg. → Broadly these optimizn are classified into
local optimizations, (peephole optimizns)
Loop optimizations.
Global optimizations.

Level 2 : Machine dependent optimizations

→ Applied in backend of compiler after target code is generated and it is being transformed into assembly code of target machine.

→ It involves CPU registers and may have absolute memory references rather than relative references.

→ These try to achieve maximum advantage of the memory hierarchy

eg. → Aho Sethi Algorithm for reduction of spilling

Types of optimizations at ICG level.

1. Function preserving transformations.

1.1. common subexpression elimination.

common
$t1 = i+1$
$t2 = b[t1]$.
$t3 = i+1$
$a[t3] = t2$

→

$t1 = i+1$
$t2 = b[t1]$
$a[t1] = t2$.

1.2 copy propagation.

1.3 dead code elimination
remove statements to which control never reaches.

if (false) ← never executes
→ if & its body can be removed

1.4. Constant folding.

$A[i+1] = B[i+1]$ → $j = i+1$
$A[j] = B[j]$.

2. Loop optimizations

2.1 Code motion.
- Loop invariants placed ~~what~~ before loop.

90% - 10% rule
10% of code have 90% of execution time due to loops.

for i = 1 to 1000
j = 100 invariant
k = i * j
print k

→

j = 100
for i = 1 to 1000
k = i * j
print k

2.2. Elimination of induction variables.
induction variable : one variable increasing in AP series other variable increasing in GP series

2.3 Strength reduction
replacement of multiplication by a subtraction or addition might be possible when induction variables are modified in a loop.

$j = j - 1$
$t2 = 4 * j$
$t3 = a[t2]$.

$j = j - 1$.
$t = t2 - 4$
$t3 = a[t2]$

global optimizations

→ Building Directed acyclic data flow graphs.
to reuse common nodes.

$x = a + b$.
$y = a + c$