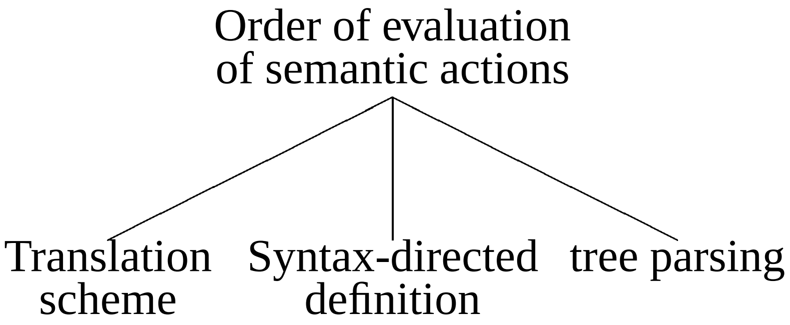
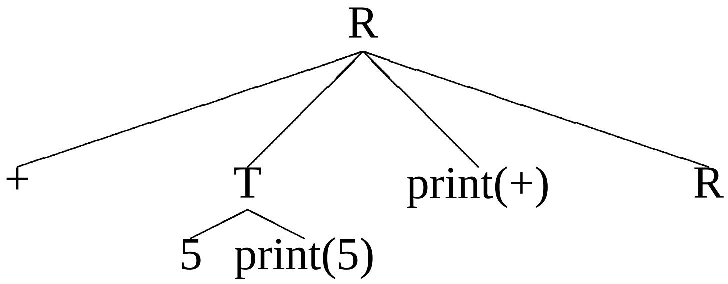


- Translation scheme: Order of semantic actions are explicitly shown within the RHS of a rule; their evaluation depends on the parsing strategy.

$$R \to \; +T\{\texttt{print('+')}\}R$$
$$T \to \; 5\{\texttt{print('5')}\}$$

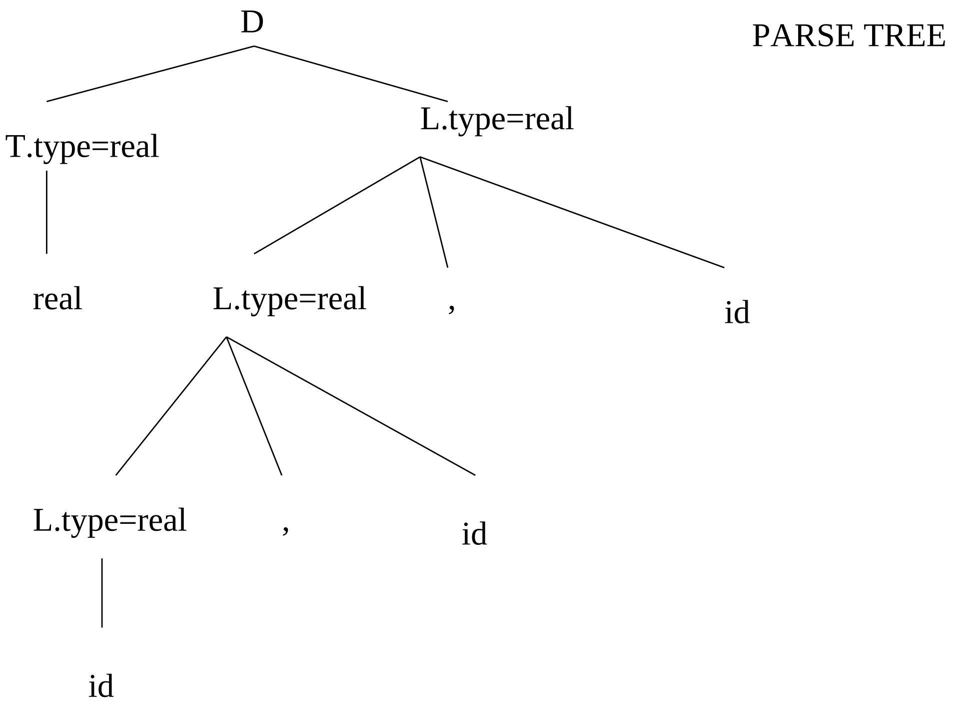


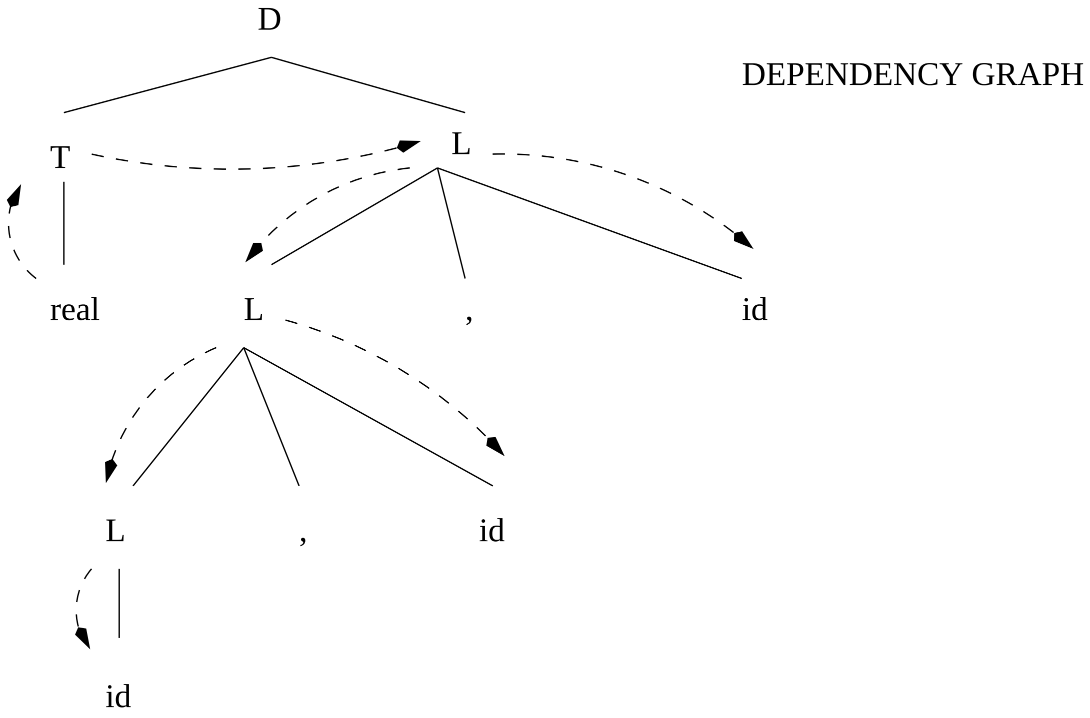
DFS prints `5+`. BFS prints `+` before 5.

- Syntax-directed definition: The semantic action is associated with the rule. Its order of execution depends on the dependencies among attributes (it is independent of the parsing strategy).

ex: declaring types of several vars

```
D -> T L {L.type=T.type}

T -> int {T.type=integer}

T -> real {T.type=real}

L -> L , id {id.type=L0.type;
             L1.type=L0.type }

L -> id     {id.type=L.type}
```


D
PARSE TREE
T.type=real
L.type=real
real
L.type=real
,
id
L.type=real
,
id
id


D
DEPENDENCY GRAPH
T
L
real
L
,
id
L
,
id
id

- Order of semantic actions:

  1. Topological sort of dependency graph. Earlier nodes are evaluated before dependent nodes.

  2. Set up heuristic rules before compiler construction.

  3. Order is dictated by parsing strategy, not by dependence among attributes (oblivious methods).

- Why `yacc` allows 1-1 pairing as in oblivious methods? If all attributes are synthesized, the dependency is always from right-to-left, hence bottom-up. Order of evalation is in lock step with the order of parsing.

- But, `yacc` can "simulate" a translation scheme by allowing limited kind of actions *within* the RHS. Since the scan is left-to-right, the symbols to the left in the RHS are recognized *before* symbols to the right-edge.

  ```
  x : a {print('a');} b     {print('b')};
  ```

  ```
  Yacc converts this to:
  ```

  ```
  x: a $ACT b    {print('b')};
  ```

  ```
  $ACT:   {print('a');};
  ```

  But this will cause shift/red conflicts.

  what about

  ```
  x : a {$3.f=$1.f} b     {print('b')};
  ```