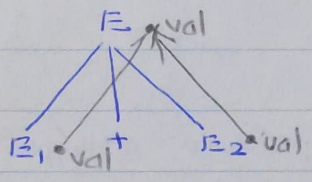
- The dependency graph for a given parse tree is constructed as follows:

  for each node 'n' in the parse tree do
    for each attribute 'a' of grammar symbol at n do
      construct a node in the dependency graph for 'a';
  for each node 'n' in the parse tree do
    for each semantic rule ~~in the parse tree etc~~ $b = f(c_1, c_2, \cdots, c_k)$
      associated with the production used at n do
        for $i = 1$ to k do
          construct an edge from the node for c_i to the node for b;

- Example

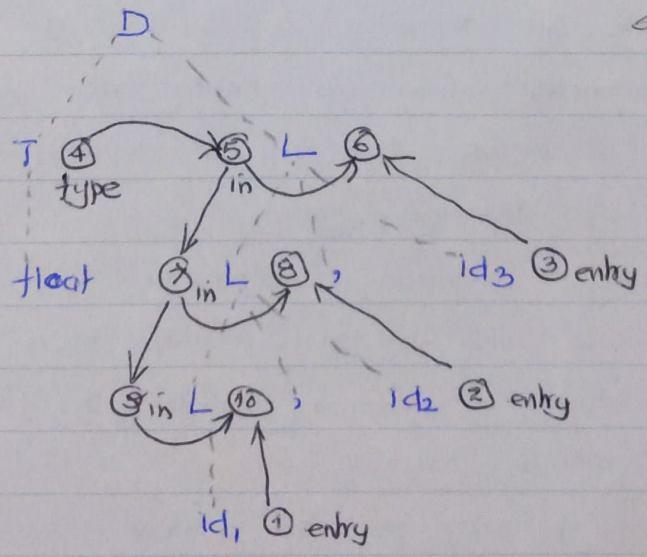
$E \rightarrow E_1 + E_2 \qquad E.val = E_1.val + E_2.val$



- Figure below shows the dependency graph for the parse tree for

  float id_1, id_2, id_3

  (Any topological sort of the dependency graph gives an evaluation order for the semantic rules.)



Nodes 6, 8, and 10 are for the dummy attribute of entry (id.entry, L.in)

TYPE CHECKING

- A compiler must check that the source program follows both the syntactic and semantic conventions of the source language.
- This checking called "static checking", ensures that certain kinds of programming errors will be detected and reported.
- Examples:
  1. Type checks: A compiler should report an error if an operator is applied to an incompatible operand; for example, if an array variable and a function variable are added together.
  2. Uniqueness checks: There are situations in which an object must be defined exactly once; for example, labels in a case statement must be distinct.
- A type checker verifies that the type of a construct matches that expected by its context. For example, the built-in arithmetic operator 'mod' requires integer operands, so a type checker must verify that the operands of 'mod' have type 'integer'.
- Also, the type checker must verify that
  - dereferencing is done only to a pointer,
  - indexing is done only on an array, and
  - a user-defined function is applied to the correct number and type of arguments.

Type Systems

- The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs.
- Each expression has a type associated with it. Furthermore, types have structure; the type 'pointer to integer' is constructed from integer type.

- In both Pascal and C, types are either basic or constructed. Basic types are the atomic types with no internal structure as far as the programmer is concerned. Eg. integer, float, and char.
- Programming languages allow a programmer to construct types from basic types and other constructed types, with arrays, records, and sets being examples. In addition, pointers and functions can also be treated as constructed types.

Type Expressions

- The type of a language construct will be denoted by a "type expression".
- Informally, a type expression is either a basic type or is formed by applying an operator called a "type constructor" to other type expressions.
- The sets of basic types and constructors depends on the language to be checked.
- Definition for type expressions.
  1. A basic type is a type expression. Among the basic types are integer, float, and char. A special basic type, "type-error", will signal an error during type checking. A basic type "void" denoting 'the absence of a value' allows statements to be checked.
  2. Since type expressions may be named, a type name is a type expression.
  3. A Type constructor applied to a type expression is a type expression. Constructors include:
     (a) Arrays: If T is a type expression, then "array (I, T)" is a type expression denoting the type of an array with elements of type T and index set I.
     For example, the Pascal declaration

     var A : array [1..10] of integer;

     associates the type expression "array (1..10, integer)" with A.

(b) Products : If T_1 and T_2 are type expressions, then their Cartesian product $T_1 \times T_2$ is a type expression.

(c) Records : The difference between a record and a product is that the fields of a record have names. The 'record' type constructor will be applied to a tuple formed from field names and field types.

(d) Pointers : If T is a type expression, then "pointer T" is a type expression denoting the type "pointer to an object of type T."

(e) Functions : We may treat functions in programming languages as mapping a "domain type D" to a "range type R". The type of such a function will be denoted by the type expression "D → R".

- For example, the Pascal declaration

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

  is denoted by the type expression

  char x char → pointer (integer)

4. A type expression may contain variables whose values are type expressions.

- A convenient way to represent a type expression is to use a graph.
  - Example : The type expression

    char x char → pointer (integer)

    can be represented as a graph, given below: