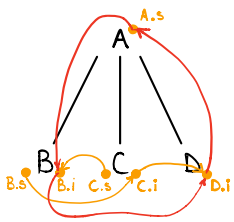
Come si **imposta** il dependency graph?
- Per ogni nodo del parse tree e per ogni attributo del simbolo rappresentato da quel nodo, impostare un nodo del dependency graph.
- Per ogni attributo `A.a` e per ogni attributo `X.x` usato per definire `A.a`, impostare un arco tra i nodi che rappresentano le coppie corrispondenti di `X.x` e `A.a`. L'arco va da `X.x` a `A.a` intendendo che `X.x` è necessario alla computazione di `A.a`

Come si **valuta** il dependency graph?
Numerare i nodi `N1`, ..., `Nk` in modo che, se c'è un arco da `Ni` ad `Nj` (`Ni -> Nj`), allora `i < j`.
Questo embedding è un sort topologico.
Lo SDD può essere valutato utilizzando un qualunque sort topologico, se esiste.

Osservazione:
Se c'è un ciclo nel dependency graph, allora non esiste alcun sort topologico dei suoi nodi, e lo SDD non può essere valutato.
Ad esempio, supponiamo di avere una grammatica
con la seguente produzione: `A -> BCD`,
e con le abbia le le seguenti regole semantiche associate alla produzione:

```
{   A.s = D.i
    B.i = A.s + C.s
    C.i = B.s
    D.i = B.i + C.i     }
```

Dato una stringa generabile da questa grammatica, quello che noi avremmo (in una qualche zona) del parse tree, sarebbe:



In che ordine li devo valutare?
C'è un sort topologico? No:
c'è pure un ciclo
Quindi non è valutabile.

Però, noi vorremmo avere delle garanzie. Non è che tutte le volte che cerchiamo di mettere degli attributi alla grammatica, vogliamo *anche* chiederci se poi mai riusciremo a valutarla.
Ci sono delle classi specifiche di syntax-directed definition per le quali è dimostrato che si può sempre trovare un ordine topologico e quindi sono sempre valutabili.
Si tratta di due classi di grammatiche attribuite: S-attribuite ed L-attribuite.

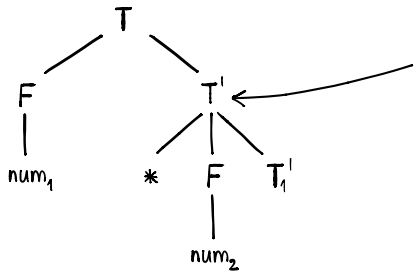
SDD **S-attribuite**: tutti gli attributi utilizzati sono di tipo sintetizzato.
La valutazione dello SDD può essere conseguita con una visita in **post-odine** del dependency graph.
Un esempio è `E -> E+T | T`.
In generale, tutte quante le grammatiche LALR, o che comunque si prestano al bottom-up parsing, se sono tali per cui possiamo scrivere gli attributi stando attenti che siano sintetizzati, possono essere svolte con questa tecnica. Questo vuol dire che risuciamo a valutare il dependency graph mentre facciamo il parsing bottom up.

SDD **L-attribuite**: gli attributi utilizzati sono:
- sintetizzati, oppure
- ereditati, ma con il vincolo che per ogni produzione `A -> X1...Xn` e per ogni `Xj.i` (attributo ereditato di `Xj`) usa solamente attributi ereditati di `A` (può ereditare dal padre) e attributi ereditati o sintetizzati di `X1...Xj-1` (attributi ereditati o sintetizzati dai fratelli a sinistra).

La grammatica per le espressioni aritmetiche di tipo LL che abbiamo visto ieri, è esattamente una SDD L-attribuita, in quanto ha esattamente questo tipo di schema. Rivediamola:

```
T -> FT'
T' -> *FT'1
T' -> ε
F -> num
```

Il problema che ci era sembrato difficile da gestire era questo: se prendiamo anche un semplice `num * num`,
l'albero di derivazione con il quale abbiamo a che fare ha questa tipologia:

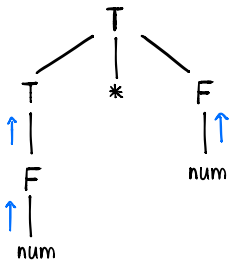


Nel momento in cui arriviamo in questa posizione, abbiamo conoscenza di `num2` (sappiamo che sarà un operando, un fattore, da coinvolgere nella moltiplicazione), ma ci manca completamente conoscenza dell'altro `num`.

Osserviamo che, con la grammatica di tipo bottom-up per lo stesso linguaggio, cioè

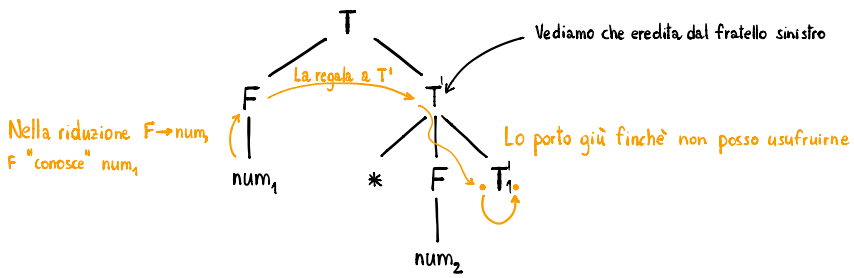
```
T -> T*F | F
F -> num
```

L'albero che si ottiene per la moltiplicazione fra due numeri è:



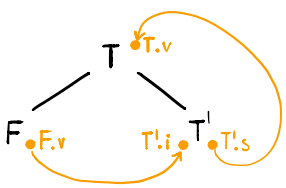
Qui, banalmente, si trasportano in su valori dei `num`, e quando si fa la riduzione, risulta anche possibile fare la moltiplicazione.

Quindi, abbiamo detto, la strategia che vogliamo utilizzare è di potarci il `num1` che abbiamo visto:

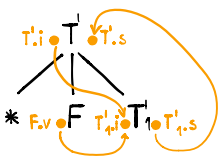


In particolare, vediamo cosa capita per le varie produzioni.

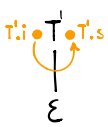
```
T -> FT' { T'.i = F.v, T.v = T.s }
```




```
T' -> *FT'1 { T'1.i = T'.i * F.v, T'.s = T'1.s }
```




```
T -> ε { T'.s = T'.i }
```




```
F -> num { F.v = num.lexval }
```

Questa è una SDD L-attribuita. Tutti e quanti gli attributi ereditati che stiamo utilizzando sono attributi che utilizzano attributi del padre e attributi dei fratelli sinistri.
E infatti, per questa qui, l'ordine topologico lo troviamo senza problemi.

Trovare la SDD data questa grammatica:

```
D -> TL
T -> int
T -> float
L -> L1, id
L -> id
```

Questo è lo spezzato della grammatica di un linguaggio di programmazione che permette di scrivere una cosa tipo:

```
int a, b
```

Per intendere che le variabili `a`, `b` sono di tipo intero.

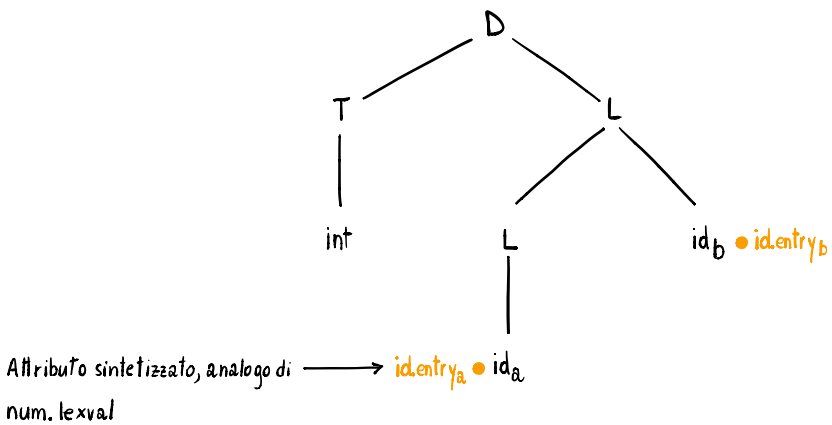
Grazie all'analisi lessicale, la tabella dei simboli contiene una entry per la `a`, una entry per la `b`, ecc. Ricordiamo che nella tabella dei simboli mettiamo tutte e quante le informazioni che ci servono sui particolari elementi che stiamo considerando. Ad esempio, di una dichiarazione di procedura, ci interesserebbe sapere il numero e il tipo dei parametri.
La tabella avrà questo aspetto:

	TIPO
a	int
b	int
c	int

In questo esercizio, immaginiamo di avere a disposizione una `insertType(id.entry, istanzaDiTipo)`, che possiamo usare successivamente come regola semantica.

Come organizzare una SDD per `int a, b`?

Ricordiamo che, quando usiamo `insertType`, ci serve sapere come si chiama l'identificatore, e che tipo ha. La storia è sempre quella: bisogna capire quando sono disponibili queste due informazioni.



Sta di fatto che, per invocare la `insertType` di questa cosa qua, dobbiamo aver visto sia l'`int`, che sta a sinistra, che `id.entry`, che stanno a destra, dall'altra parte. Dobbiamo attrezzarci con gli attributi.

Soluzione:

```
T -> int { T.s = integer }
D -> TL { L.i = T.s }
L -> L1, id { L1.i = L.i, insertType(id.entry, L.i) }
L -> id { insertType(id.entry, L.i) }
```