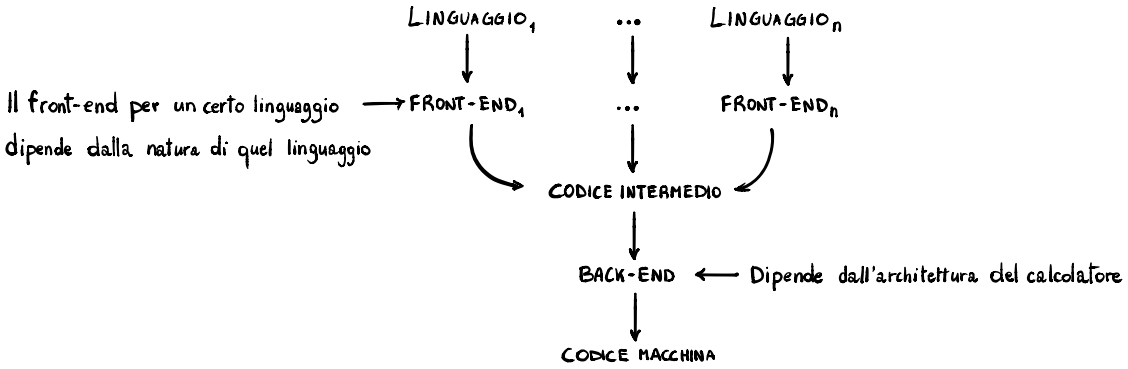
La compilazione è composta tipicamente da due fasi principali: il front-end e il back-end.

- Il front-end del compilatore si occupa del passaggio dal codice sorgente ad un opportuno codice intermedio. In questa fase hanno luogo l'analisi lessicale, l'analisi sintattica e l'analisi semantica.
- Il back-end del compilatore si occupa del passaggio dal codice intermedio al codice macchina, o codice eseguibile.

Questo schema consente di riutilizzare il medesimo back-end per front-end diversi:



Mentre i linguaggi di programmazione sono numerosissimi, ci sono solo due alternative per la rappresentazione intermedia, che vanno per la maggiore: il *three-address code* e l'*abstract syntax tree*.

Il *three-address code* è il codice per un'architettura astratta di calcolatore.
Il calcolatore astratto è in grado di eseguire semplici istruzioni, caratterizzate da un codice operativo e al più tre indirizzi per gli operandi.

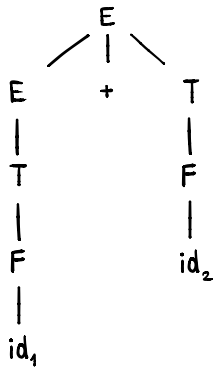
L'*abstract syntax tree*, invece, è la rappresentazione intermedia più usata e più interessante, nonché quella che studieremo.

Anticipazione: Abstract syntax tree

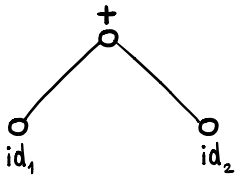
A dispetto del nome *tree*, è un particolare grafo, che deriva direttamente dall'albero di derivazione.
Facciamoci subito un'idea.
Prendiamo la seguente grammatica, che genera il linguaggio delle espressioni aritmetiche:

```
E -> E+T | T
T -> T*F | F
F -> (E) | id
```

Data una certa espressione aritmetica, quello che ci interessa sapere, alla fine, è il valore della stessa.
Quali sono l'albero sintattico dell'espressione `id1+id2`?



Come detto, quello che ci interessa sapere, alla fine, è la somma fra i due valori `id1` e `id2`.
Quindi, perché mai avere tutti questi nodi in un grafo? Infatti, ad un albero di derivazione come questo, viene fatto corrispondere un albero di derivazione astratto con un certo risparmio di nodi e di archi:



In seguito vedremo che questo grafo (in questo caso, albero) può essere computato *mentre* facciamo l'analisi sintattica.

Introduciamo ora le syntax-directed definitions

Le syntax-directed definitions si possono agganciare a ogni genere di grammatica, quindi sia a quelle analizzabili in top-down, che quelle analizzabili in bottom-up; motivo per cui è essenziale capire come funzionano le tecniche di parsing, e specialmente in quale momento dell'analisi vengono fatte le scelte di espansione o di riduzione, che ci consentono di derivare l'albero sintattico.

Una **syntax-directed definition** è data da una grammatica context-free **arricchita**
da attributi associati ai simboli della grammatica e regole associate alle produzioni della grammatica.
Un attribuito può essere: un tipo, un puntatore, una entry in una tabella, una funzione...
Più avanti, vedremo come questi attributi possono essere classificati in due categorie principali.

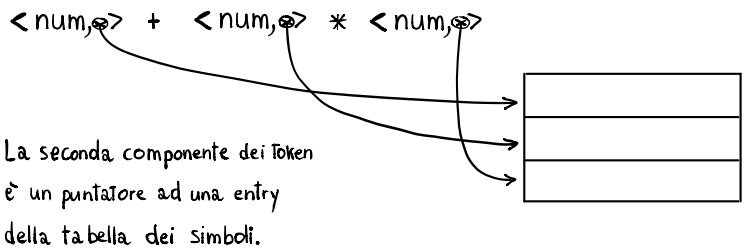
Nel nostro studio, per rendere le cose comprensibili, utilizzeremo questa grammatica LALR per le le espressioni aritmetiche:

```
E -> E+T
E -> T
T -> T*F
T -> F
F -> id
```

Grammatica nello stile bottom-up
Non va bene per il top-down perchè mostra ricorsione a sinistra (quindi non è LL-1)

Un possibile input è dato dalla stringa `3+4*5`.

La fase di analisi lessicale produce una sequenza di token:



La seconda componente dei token è un puntatore ad una entry della tabella dei simboli.

Nella tabella dei simboli mettiamo tutte e quante le informazioni che si riescono a capire man mano nelle varie fasi.

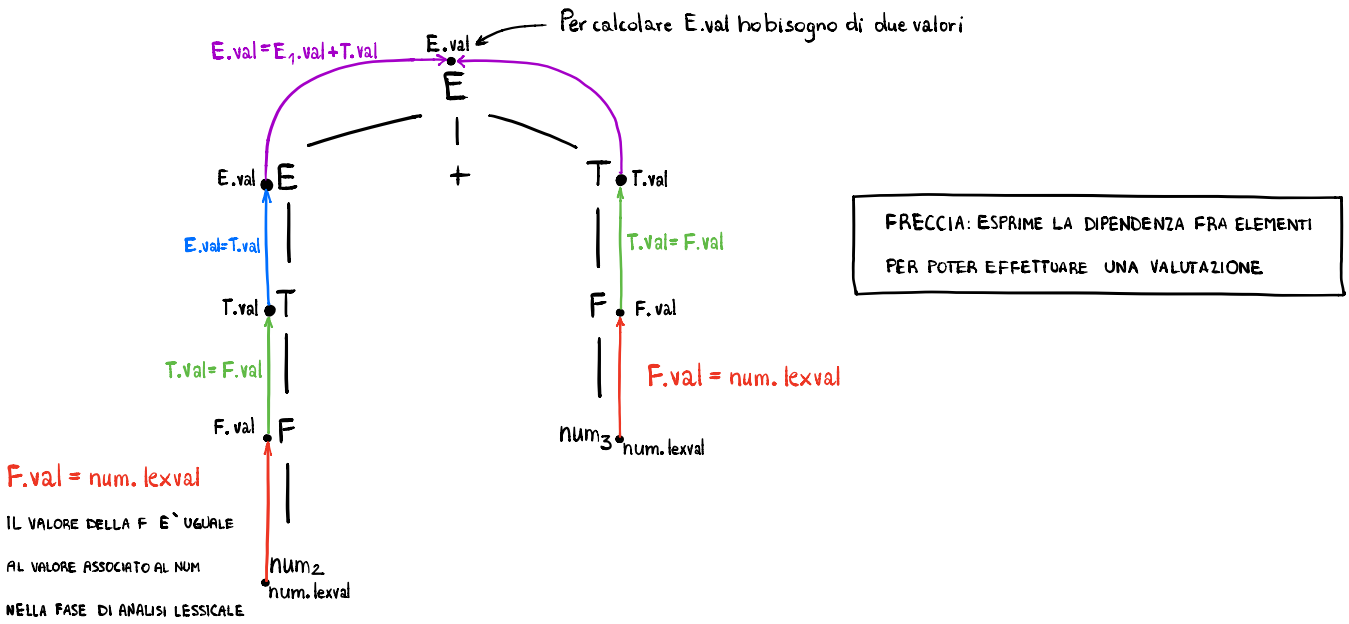
A questo punto, comincia l'analisi sintattica.
Quello che facciamo è considerare la stringa come `num+num*num` e capire se appartiene al linguaggio della grammatica.
Se l'analisi sintattica ha successo, dovremmo essere in grado di computare il valore dell'espressione.
Come dicevamo, possiamo fare questa operazione nello stesso momento in cui facciamo l'analisi sintattica. E come si fa?

Vediamo intanto come si scrive.
Si arricchisce la grammatica, facendola diventare una syntax-directed definition:

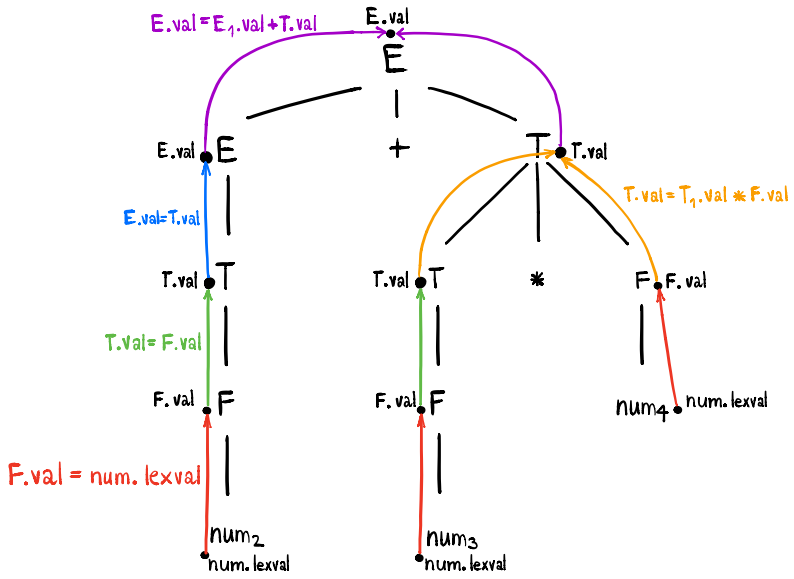
```
E -> E1 + T { E.val = E1.val + T.val }
E -> T      { E.val = T.val }
T -> T1 * F { T.val = T1.val * F.val }
T -> F      { T.val = F.val }
F -> num    {F.val = num.lexval }
```

Abbiamo messo queste regole affinchè ci consentano di valutare il valore delle espressioni in ingresso.

Per capire, vediamo subito un esempio: `2+3`. Disegnamp l'albero di derivazione in nero e l'albero annotato a colori:



Proviamo con un altro esempio: `2+3*4`. Gli alberi sono:



Le dipendenze vanno, come dire, all'insù, e questo ci suggerisce che stiamo usando degli attributi **sintetizzati**,
cioè l'attributo del padre (l'attributo del driver della produzione) è espresso in funzione
degli attributi dei figli (gli attributi dei simboli che stanno nel body della produzione).

Dal momento che le dipendenze vanno all'insù, la computazione dell'espressione può essere fatta direttamente mentre si fa l'analisi sintattica.

In altre parole:
Quando effettuiamo una riduzione, sfruttiamo la regola associata alla produzione per la quale stiamo riducendo.
Se, in questo processo, i valori sono di volta in volta noti, nessuno vieta di arrivare in fondo con l'albero e anche il risultato.

Ciò detto, è necessario osservare che è molto importante che la grammatica e le regole siano ben organizzate.
Se una certa regola necessita di informazioni che ancora non sono disponibili, la computazione non può funzionare.
Il risultato in assoluto migliore che possiamo ottenere è organizzare grammatica e attributi in modo tale che riusciamo a fare tutto nella medesima passata (calcolo di albero e attributi).

Tipologie di attributi:
Attributi sintetizzati: per la produzione `A -> α`, si dice sintetizzato l'attributo `A.at` (`at` = attributo),
definito come funzione di attributi dei simboli che occorrono in `α`.
Attributi ereditati: per la produzione `B -> αAβ` si si dice ereditato l'attributo `A.at`,
definito come funzione degli attributi di `B` e dei simboli che occorrono in `α` e in `β`.

Siccome i terminali non hanno figli:
I simboli terminali della grammatica hanno solo attributi sintetizzati provenienti dall'analisi lessicale.
Non possono esserci regole per computarli.

La grammatica che abbiamo visto prima aveva tutti attributi sintetizzati.
La definizione è proprio "valore di padre dipende da valore di figlio".

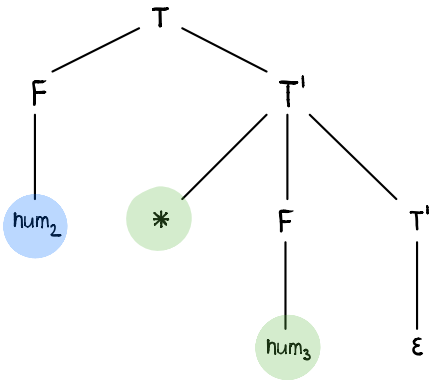
Le grammatiche *s-attribuite*, cioè quelle in cui tutti gli attributi sono sintetizzati, sono ideali per l'analisi bottom-up.
Visto che la direzione delle dipendenze fra attributi va dalle foglie alla radice, una qualunque visita in post-ordine
dell'albero denotato fa computare il valore associato alla radice. (Nb: post-ordine = depth-first).

Vediamo ora una frazione della grammatica LL delle espressioni aritmetiche, in cui abbiamo eliminato la ricorsione a sinistra, le parentesi e il +.

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

Grammatica nello stile bottom-up

Sia l'input `2*3`. L'albero di derivazione è:

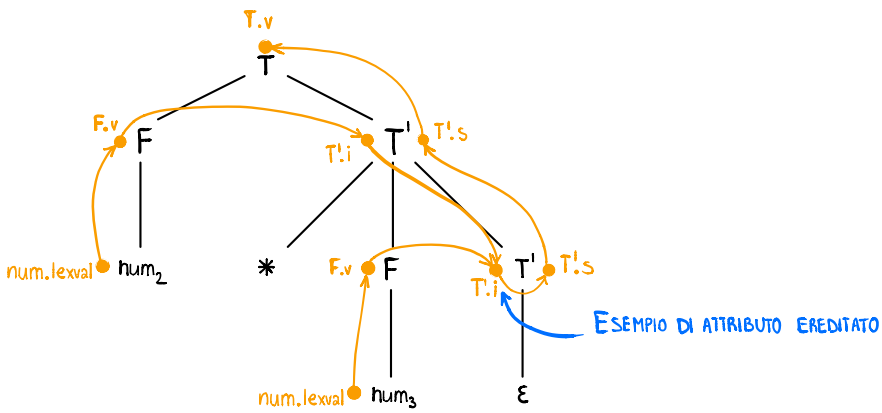


Questo albero ci mette in difficoltà, in quanto il `*` e `num3` vengono da una parte, mentre `num2` viene dall'altra.
Qua vengono in soccorso gli **attributi ereditati**, che permettono di portare le informazioni di qua e di là nell'albero.

La grammatica arricchita è:

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

L'albero di derivazione annotato è:



Gli attributi ereditati sono quelli espressi come funzione degli attributi del padre o dei fratelli.

Notiamo che, con la grammatica precedente, avevamo una struttura estremamente semplice,
in cui bastava una visita dell'albero annotato in post-ordine.

Con quest'ultima grammatica, invece, una visita dell'albero annotato in post-ordine non è in grado di valutarlo.
Quello che ci serve, in questo caso, è una individuare un ordinamento **topologico**.