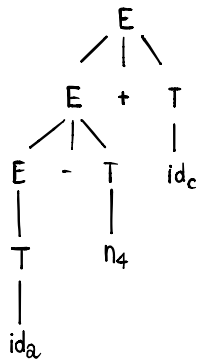
Grammatica:

```
E -> E+T | E-T | T
T -> (E) | id | n
```

Stringa in input: `a-4+c`

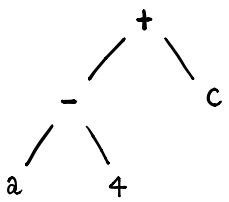
Riusciamo a trovare l'albero di derivazione? Sì: quindi la stringa in input appartiene al linguaggio della grammatica.

ALBERO DI DERIVAZIONE
(O PARSE TREE)



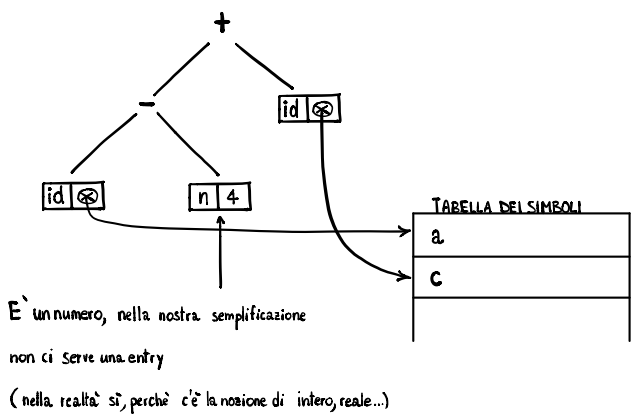
Ma l'utente vorrebbe l'abstract syntax tree, la cui struttura è molto più sintetica:

ABSTRACT SYNTAX TREE

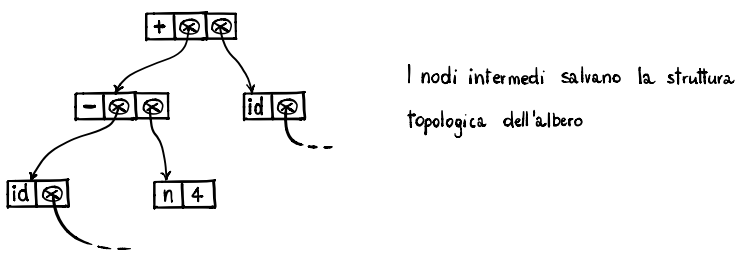


Come rappresentiamo l'abstract syntax tree in memoria? In memoria avremo un record per ogni nodo (o foglia).

Vediamo le foglie:



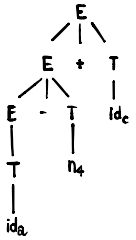
Vediamo i nodi intermedi:



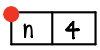
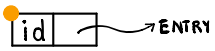
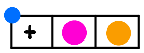
L'abstract syntax tree appena visto è una struttura che vogliamo generare mentre facciamo l'analisi sintattica.
Possiamo conseguire ciò specificando delle regole semantiche (azioni).

Supponiamo di avere due funzioni che allocano lo spazio per generare un nodo o una foglia: `new node (...)` e `new leaf (...)`

```
T -> n      { T.node = new leaf (n, n.lexval) }
T -> id     { T.node = new leaf (id, id.entry) }
T -> (E)    { T.node = E.node }
E -> T      { E.node = T.node }
E -> E1-T   { E.node = new node ('-', E1.node, T.node) }
E -> E1+T   { E.node = new node ('+', E1.node, T.node) }
```



Supponiamo di avere in input la stringa `a-4+c`. Applichiamo lo shift-reduce algorithm (supponendo di avere già la tabella di parsing) e vediamo come questa grammatica attribuita ci permette di ottenere un parse tree.
In corrispondenza delle direttive di reduce vengono eseguite le regole semantiche. "Coloriamo" i puntatori (indirizzi):

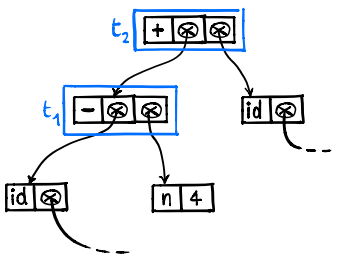
Direttiva	**Effetto della regola semantica**
reduce `T -> id`$_a$	
reduce `E -> T`	
reduce `T -> n`$_4$	
reduce `E -> E`$_1$`-T`	
reduce `T -> id`$_c$	
reduce `E -> E`$_1$`+T`	

L'abstract syntax tree è importante, perché permette di fare molti controlli, che altrimenti dovrebbero essere fatti direttamente sul parse tree, ma mettendo delle regole semantiche più complesse.

Ora, dall'abstract syntax tree, è possibile generare del codice intermedio, che, in questo caso, sarebbe:

```
t1 = a - 4
t2 = t1 + c
```

Ci sono arrivato associando ad ogni nodo nodo intermedio un temporaneo:



Ma siamo sicuri che, per generare il codice intermedio, devo necessariamente passare prima dall'abstract syntax tree?
No. È possibile conseguire l'ottenimento del codice intermedio aggiungendo alla grammatica degli attributi opportuni!

Supponiamo di avere la stessa grammatica di prima, con una produzione in più, `S -> id := E`, e con i seguenti attributi:

```
T -> n      { T.at = n.lexval }
T -> id     { T.at = table.get(id.lexval) }
T -> (E)    { T.at = E.at }
E -> T      { E.at = T.at }
E -> E1-T   { E.at = new Temp(), generate(E.at '=' E1.at '-' T.at) }
E -> E1+T   { E.at = new Temp(), generate(E.at '=' E1.at '+' T.at) }
S -> id:=E  { generate(table.get(id.lexval) '=' E.at) }
```

(dove `at` = attributo)

Supponiamo di avere in input la stringa `z := a - 4 + c`.
Applichiamo lo shift-reduce algorithm (supponendo di avere già la tabella di parsing).
In corrispondenza delle direttive di reduce vengono eseguite le regole semantiche.

Direttiva	**Effetto della regola semantica**
reduce `T -> id`$_a$	`T.at = a`
reduce `E -> T`	`E.at = a`
reduce `T -> n`$_4$	`T.at = 4`
reduce `E -> E`$_1$`-T`	`E.at =` t_1`, "`t_1` = a - 4"`
reduce `T -> id`$_c$	`T.at = c`
reduce `E -> E`$_1$`+T`	`E.at =` t_2`, "`t_2` =` t_1` + c"`
reduce `S -> id:=E`	`"z =` t_2`"`