# 04-analisi-semantica



# Analisi semantica

Per generare codice abbiamo bisogno di capire il suo significato, quindi il compilatore ha bisogno di porsi tante domande, ad esempio:

- "x" è uno scalare, un array o una funzione? "x" è dichiarata?
- Ci sono nomi che non sono stati dichiarati? Magari dichiarati e non usati? etc.
  Queste domande fanno parte dell'analisi context-sensitive, cioè che hanno bisogno di un contesto per avere senso.
  Come possiamo dunque rispondere a queste domande?
- Usando metodi formali
  - Grammatiche context-sensitive: consentono di definire regole in cui la produzione di un simbolo dipende dai simboli circostanti (difficili da implementare e poco efficienti).
  - Grammatiche attribuite: ai simboli vengono associati attributi (valori o informazioni aggiuntive) e regole semantiche per calcolare questi attributi (molto utili).
- Tecniche ad-hoc
  - Tabelle dei simboli: strutture dati utilizzate per tenere traccia delle informazioni sulle entità del programma, come variabili, funzioni, classi e scope.
  - Codice ad-hoc (action routines): frammenti di codice specifici (o funzioni) eseguiti durante la compilazione per risolvere problemi contestuali.

> Nel parsing (analisi sintattica) vincono le grammatiche libere dal contesto, mentre nell'analisi semantica (context-sensitive) le tecniche ad-hoc dominano la pratica.

# Grammatiche attribuite

Le grammatiche attribuite combinano la struttura di una grammatica sintattica con regole semantiche per arricchire i simboli del linguaggio con **attributi**, che rappresentano informazioni aggiuntive come il tipo di una variabile o il valore di un'espressione, e le regole semantiche permettono di **calcolare e verificare** questi attributi.
**PROBLEMI**:

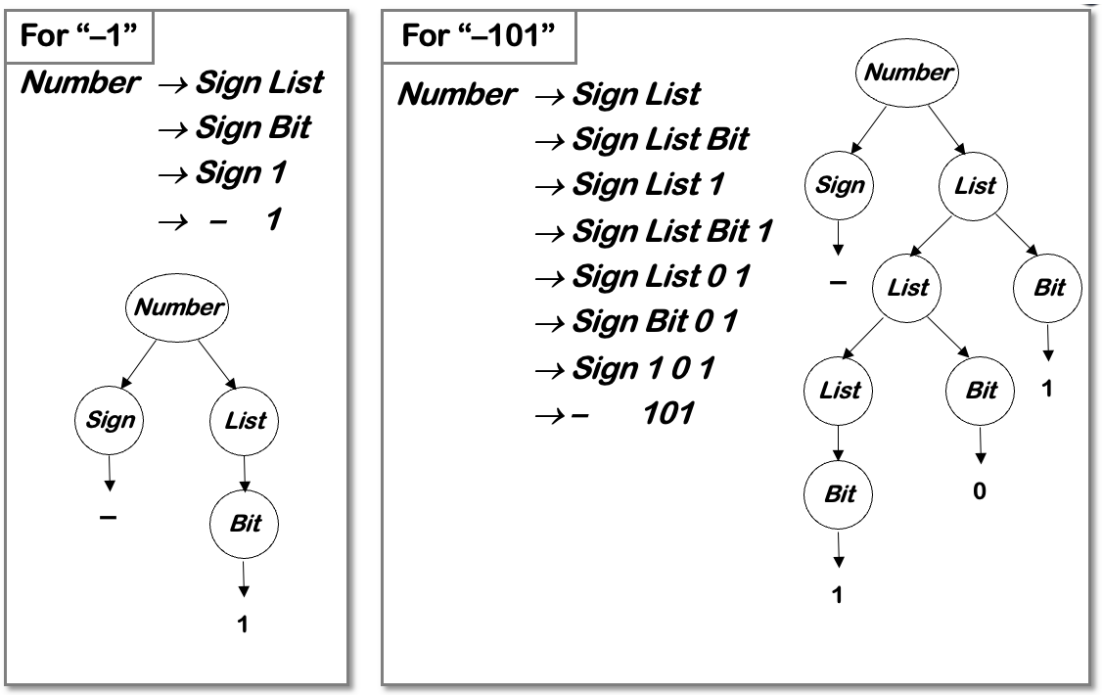
- calcoli non locali: se un attributo in un punto del codice dipende da informazioni lontane (come l'assegnazione di una variabile definita in un altro blocco), risulta difficile gestirlo in una grammatica attribuita.
- informazioni centralizzate: per molte analisi, serve una tabella centrale delle informazioni (tabella dei simboli) per tenere traccia delle variabili, funzioni e tipi le grammatiche attribuite non sono pensate per gestire direttamente strutture centralizzate, risultando quindi poco pratiche.
  Arriviamo dunque a prediligere le **Tecniche ad-hoc**, più flessibili e pratiche per problemi complessi, per ora però capiamo come funzionano queste.

Gli elementi principali sono:

- **Simboli e Attributi**: ogni simbolo della grammatica (sia terminale che non terminale) è arricchito con un insieme di attributi, quest'ultimi sono valori associati ai simboli, che possono contenere informazioni semantiche come il tipo, il valore o altre proprietà rilevanti, esistono due tipi principali di attributi:
  1. Attributi sintetizzati: calcolati dalle regole semantiche sulla base dei figli del simbolo nell'albero sintattico
  2. Attributi ereditati: derivano da informazioni che provengono dai genitori o dai fratelli del simbolo nell'albero.
- **Regole di attribuzione**: per ogni produzione della grammatica vengono definite regole che descrivono come calcolare gli attributi. Le regole sono funzionali, cioè determinano univocamente il valore di ogni attributo basandosi sugli attributi disponibili.
- **Funzioni semantiche**: le regole di attribuzione spesso usano funzioni definite dall'utente, che calcolano gli attributi in base a valori disponibili o logica specifica.
  Vediamo un esempio, la seguente grammatica descrive i numeri binari con segno:

| | | | |
|---|---|---|---|
| 1 | *Number* | → | *Sign List* |
| 2 | *Sign* | → | + |
| 3 | | \| | - |
| 4 | *List* | → | *List Bit* |
| 5 | | \| | *Bit* |
| 6 | *Bit* | → | 0 |
| 7 | | \| | 1 |

Formiamo due AST per due input differenti (-1 e -101):

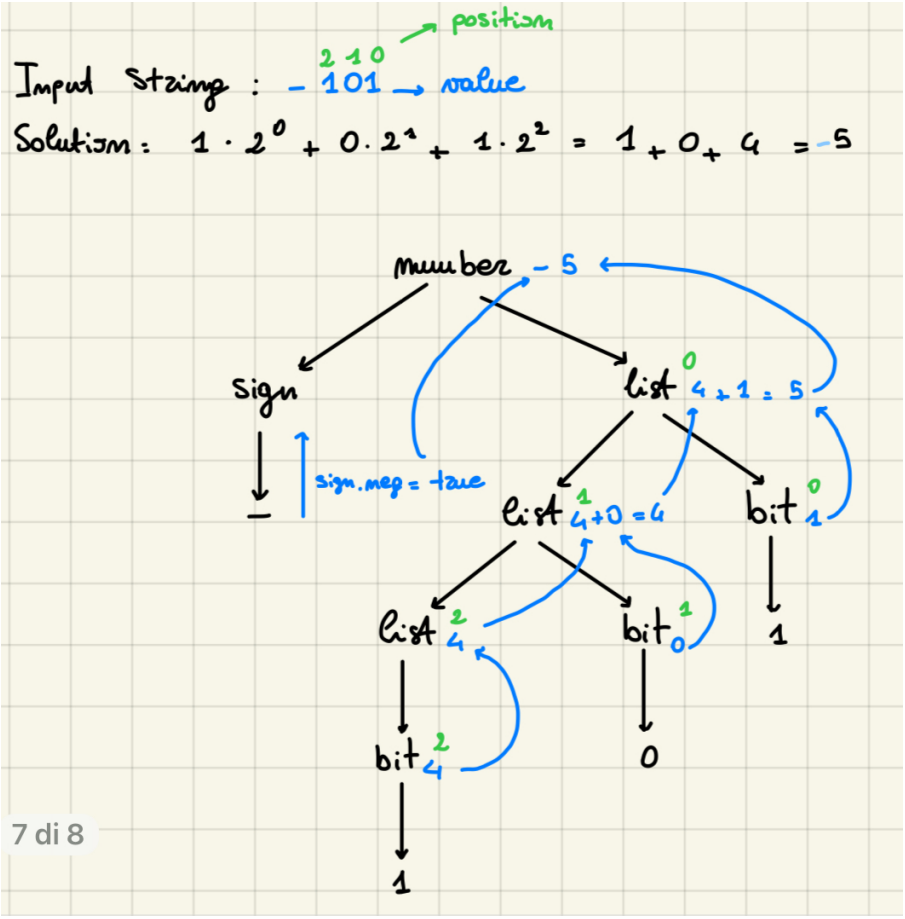



Vogliamo dunque calcolare il valore decimale, dobbiamo quindi aggiungere delle regole per farlo:

| Productions | | | Attribution Rules |
|---|---|---|---|
| *Number* | → | *Sign List* | *List.pos* ← 0<br>*if Sign.neg*<br>*then Number.val* ← - *List.val*<br>*else Number.val* ← *List.val* |
| *Sign* | → | + | *Sign.neg* ← *false* |
| | \| | - | *Sign.neg* ←*true* |
| $List_0$ | → | $List_1$ *Bit* | $List_1.pos$ ← $List_0.pos$ + 1<br>*Bit.pos* ← $List_0.pos$<br>$List_0.val$ ← $List_1.val$ + *Bit.val* |
| | \| | *Bit* | *Bit.pos* ← *List.pos*<br>*List.val* ← *Bit.val* |
| *Bit* | → | 0 | *Bit.val* ← 0 |
| | \| | 1 | *Bit.val* ← $2^{Bit.pos}$ |

| Symbol | Attributes |
|---|---|
| *Number* | val |
| *Sign* | neg |
| *List* | pos, val |
| *Bit* | pos, val |

Possiamo dedurre che, dall'immagine sopra, abbiamo degli attributi per ogni tipo di produzione, ad esempio banalmente per il "Bit" e "List" abbiamo due attributi necessari: posizione e valore. Andiamo con ordine vediamo come valutare un AST con gli attributi:




**Spiegazione**

In input abbiamo la stringa -101, che dovrà essere trasformata in decimale tramite una grammatica con attributi.

Partiamo a sinistra dove abbiamo il segno -, questo porterà la produzione di Sign con l'attributo neg a true e lo "assegna" a number.
Ora spostiamoci a destra, il primo List che incontriamo avrà l'attributo position uguale a 0 il secondo list attributo position uguale ad 1 e così via anche per i bit, il primo bit che incontriamo avrà position = 0 etc.
Ora risaliamo l'albero dalle foglie quindi iniziamo dal primo bit più a sinistra che è uguale ad 1, assegnamo quindi alla sua produzione "bit" il valore 4 dato dal calcolo di 2^bit.pos, riporto ora il valore alla produzione superiore "list", nel mentre guardo anche il bit 0, che ha come valore 0 dato dalla condizione che se bit.val = 0 allora 0.
Arriviamo ora alla produzione "List" con posizione 1, che valore avrà? Devo semplicemente riportare il valore dei due figli sommato, dunque abbiamo la list in posizione 2 con valore 4 e il bit in posizione 1 con valore 0 sommando troviamo che list in posizione 1 vale 4, questo meccanismo è applicato ricorsivamente a tutte le produzioni dell'AST trovando quindi in automatico che number avrà valore finale -5.

## Tipologie di Attributi

- **Attributi sintetizzati**: sono calcolati a partire dagli attributi dei figli e da eventuali costanti o informazioni esterne (dal basso verso l'alto). Le caratteristiche principali sono:
  - Gli attributi sintetizzati di un nodo dipendono dai valori dei nodi figli (o terminali associati) e possono includere costanti.
  - Se una grammatica utilizza solo attributi sintetizzati, si dice **S-attributed** (compatibile con il parsing LR, bottom-up)
  - Facile da implementare durante il parsing, il parsing LR costruisce l'albero sintattico dal basso verso l'alto, calcolare gli attributi sintetizzati durante il parsing è diretto e non richiede trasformazioni particolari.
- **Attributi ereditati**: sono calcolati usando valori provenienti dal genitore del nodo nell'albero sintattico, informazioni provenienti dai fratelli e tramite costanti o informazione esterne. Le caratteristiche principali sono:
  - Gli attributi ereditati sono particolarmente utili per rappresentare il **contesto** del nodo corrente.
  - Gli attributi ereditati spesso richiedono che le informazioni siano calcolate prima che il nodo sia visitato (non è facilmente compatibile con il parsing LR).
  - Spesso, un'analisi basata su attributi ereditati può essere riscritta per evitare questi attributi, trasformandoli in sintetizzati o utilizzando strutture di supporto.
  - Nonostante le difficoltà pratiche, gli attributi ereditati sono ritenuti più "naturali" per esprimere certe relazioni semantiche, poiché modellano esplicitamente il contesto.

## Metodi per calcolare gli attributi

**Metodi dinamici basati su dipendenze**: questi metodi si basano sulle **dipendenze** tra gli attributi, rappresentate in un grafo, per determinare l'ordine di calcolo degli attributi in tempo reale.
**Procedura**:

1. Costruzione dell'albero sintattico (AST)
2. Costruzione del grafo delle dipendenze: ogni attributo è rappresentato come un nodo del grafo, le dipendenze vengono rappresentate come archi diretti
3. Ordinamento topologico: si effettua un ordinamento topologico del grafo che garantisce che gli attributi siano calcolati nell'ordine corretto.
4. Calcolo degli attributi: vengono valutati in base all'ordine derivato dall'ordinamento topologico.
   **Vantaggi**: funziona con qualsiasi grammatica attribuita.
   **Svantaggi**: overhead, la costruzione e l'elaborazione del grafo delle dipendenze può essere costoso in termini di tempo e memoria.

**Metodi basati su regole (Treewalk)**: l'ordine di calcolo degli attributi è determinato **in anticipo**, durante la generazione del compilatore, analizzando le regole semantiche associate alla grammatica.
**Procedure**:

1. Analisi delle regole: durante la fase di generazione del compilatore, le regole di attribuzione sono analizzate per identificare le dipendenze tra gli attributi.
2. Si determina un **ordine fisso** per il calcolo degli attributi, basato su una strategia che garantisce che tutti gli attributi necessari per un calcolo siano disponibili al momento giusto.
3. Durante l'elaborazione dell'albero sintattico, i nodi vengono visitati in un ordine specifico (determinato in precedenza) e gli attributi vengono calcolati di conseguenza.
   **Vantaggi**: L'ordine statico elimina il bisogno di costruire grafi di dipendenze a runtime, una volta determinato l'ordine, il processo è diretto.
   **Svantaggi**: Non tutte le grammatiche attribuite possono essere risolte facilmente con un ordine statico. Alcune richiedono trasformazioni o modifiche.

**Metodi oblivious (Passes, Dataflow)**
Questi metodi ignorano le regole semantiche dell'albero sintattico, e invece scelgono un ordine predeterminato (in fase di progettazione del compilatore) per valutare gli attributi.
**Procedure**:

1. Gli sviluppatori decidono un ordine fisso di calcolo degli attributi, questo è basato su ipotesi pratiche e non richiede l'analisi delle regole semantiche.

2. Gli attributi sono calcolati in **più passaggi** sull'albero sintattico, questo approccio può essere simile a un'analisi dataflow: si propagano le informazioni attraverso i nodi fino a quando tutti gli attributi non sono calcolati.
   **Vantaggi**: Non richiede analisi sofisticate di dipendenze o costruzione di grafi, può gestire attributi complessi con un numero sufficiente di passaggi.
   **Svantaggi**: L'approccio può richiedere più passaggi rispetto ai metodi dinamici o basati su regole, con un aumento dei tempi di esecuzione.

> Il grafo delle dipendenze dev'essere aciclico.

## Circolarità nelle grammatiche attribuite

**Circolarità** nelle grammatiche si riferisce alla presenza di **dipendenze circolari** tra gli attributi. Se le regole di valutazione creano dipendenze cicliche, ovvero un attributo dipende da un altro che, a sua volta, dipende dal primo, si ha una situazione di **circolarità**, che è problematica per la valutazione, soprattutto in un compilatore.

Esistono **grammatiche non circolari** che non presentano questa problematica. La più grande classe di grammatiche che non genera dipendenze circolari è rappresentata dalle **grammatiche fortemente non circolari (SNC)**. La buona notizia è che **testare se una grammatica è SNC** può essere fatto in **tempo polinomiale**, quindi è possibile farlo in modo efficiente.

Il testo fornisce un esempio di grammatica in cui tutti gli attributi sono **sintetizzati**, quindi si tratta di una **grammatica S-attribuita**

Un possibile miglioramento è il **tracciamento dei carichi** (cioè, la gestione delle variabili o valori già caricati) nelle produzioni, per evitare di caricare un valore più volte.

- È necessario introdurre dei **set Before e After** per ogni produzione, che permettano di tracciare quali valori sono stati già caricati e quali no. Questo richiede una gestione più complessa degli attributi, ma è utile per migliorare l'efficienza.
- L'aggiunta di questi set aumenta la **complessità** della grammatica, poiché per ogni produzione bisogna aggiungere delle **regole di copia** per propagare i valori tra i vari set.

Un ulteriore passo nell'evoluzione del modello è la gestione di un **set di registri finito**, cioè un numero limitato di variabili che possono essere utilizzate per tracciare i valori.

- Questo complica la produzione **Factor → Identifier**, perché è necessario tener traccia dei registri disponibili e fare un'allocazione adeguata per ciascun identificatore. Anche se non sono necessari cambiamenti sostanziali, questa gestione richiede un'**inizializzazione più complessa**.

> La difficoltà di alcune modifiche dipende dal fatto che il tracciamento dei carichi comporta l'introduzione di molte **regole di copia**, mentre la gestione di un set di registri finiti è un cambiamento relativamente semplice, una volta che il tracciamento dei carichi è già stato implementato.

**Conclusione finale**

- Il **tracciamento dei carichi** e l'introduzione di set Before e After aumenta notevolmente la **complessità della grammatica**, mentre la gestione di un set di registri finiti è una modifica relativamente **più semplice** in quanto si basa su un meccanismo già introdotto.
- La **complessità aumenta con l'introduzione di regole di copia**, che devono essere scritte per ogni produzione e che comportano un grande aumento del numero di regole nella grammatica.

# Dalle grammatiche attribuite ai metodi ad-hoc

## Problemi Grammatiche Attribuite

- Le **regole di copia** (copy rules) aumentano la **difficoltà cognitiva** di comprensione e manutenzione del codice.
- Aumentano anche i **requisiti di spazio**, poiché bisogna copiare gli attributi tra i vari nodi dell'albero sintattico, e l'uso di **puntatori** può peggiorare ulteriormente la difficoltà cognitiva.
- Il risultato finale è un **albero attribuito**, in cui le informazioni semantiche sono assegnate ai nodi dell'albero. Questo comporta due possibili soluzioni:
  - **Costruire l'albero di parsing** e poi cercare le risposte all'interno di esso.
  - **Copiare le informazioni nei nodi** e fare riferimento a queste informazioni tramite il nodo radice dell'albero.

## Traduzione Sintattica Diretta Ad-Hoc

Un altro approccio più flessibile è l'uso della **traduzione sintattica diretta ad-hoc** (Ad-hoc Syntax-Directed Translation, SDT), che si basa su un parser bottom-up:

- **Associazione di frammenti di codice con ogni produzione**: ad ogni riduzione della produzione, viene eseguito il frammento di codice associato.
- La **flessibilità completa** è offerta dalla possibilità di includere codice arbitrario (anche codice che potrebbe essere problematico), il che permette di adattare il compilatore a vari scenari.

Per fare funzionare questo approccio:

- È necessario **dare un nome agli attributi** di ciascun simbolo a sinistra e a destra della produzione.
  - Yacc, per esempio, usa i simboli `$$`, `$1`, `$2`, ..., `$n` per fare riferimento agli attributi dei simboli nelle produzioni.
- È necessario avere uno **schema di valutazione** che si inserisce bene nell'algoritmo LR(1).

La maggior parte dei parser utilizza questo **approccio ad-hoc** per l'analisi semantica, con vari vantaggi e svantaggi:

- **Vantaggi**:
  - Supera le limitazioni del paradigma delle grammatiche attribuite.
  - È più **efficiente** e **flessibile**, poiché consente l'esecuzione di codice arbitrario in qualsiasi punto della produzione.
- **Svantaggi**:
  - È necessario scrivere il codice manualmente, senza l'assistenza di strumenti.
  - Il programmatore si occupa direttamente dei dettagli complessi.
  Molti generatori di parser, come Yacc, supportano una **notazione simile a Yacc**, che facilita questo approccio.

**Usi Tipici**

- **Costruire una tabella dei simboli**:
  - Si possono inserire le informazioni di dichiarazione durante il parsing.
  - Alla fine della sintassi della dichiarazione, si può fare un post-processing per verificare errori.
- **Controllo degli errori e verifica dei tipi**:
  - **Definire prima dell'uso** e **controllare al momento del riferimento** (ad esempio, verificando la dimensione, il tipo o la compatibilità di tipo di un'espressione).
  - Si può eseguire un controllo di tipo **bottom-up** o **verifica dell'interfaccia di una funzione**.

**Questa è davvero "Ad-hoc"?**
Il confronto tra **pratica** e **grammatiche attribuite** rivela:

- **Somiglianze**:
  - Entrambi gli approcci associano regole o azioni alle produzioni.
  - L'ordine di applicazione delle regole è determinato dagli **strumenti** (parser), non dall'autore.
  - I nomi dei simboli sono **astratti**.
- **Differenze**:

- Le azioni sono applicate come un'unità nel caso della traduzione sintattica diretta, mentre nelle grammatiche attribuite le regole sono applicate in modo **funzionale**.
- Nel caso delle azioni ad-hoc, **qualsiasi cosa è permessa**, mentre nelle grammatiche attribuite le regole sono più strutturate e formali.
- Le grammatiche attribuite sono più **astratte** rispetto alle azioni ad-hoc.

## Limitazioni

L'approccio ad-hoc richiede che le azioni vengano eseguite in un **ordine rigido** (post-ordine, da sinistra a destra o bottom-up), con alcune implicazioni:

- **Dichiarazioni prima dell'uso**.
- Non è possibile passare informazioni contestuali **dal basso verso l'alto**.
- La soluzione potrebbe necessitare di **variabili globali**, che richiedono una gestione più complessa.

## Come adattare la traduzione sintattica ad-hoc in un parser LR(1)?

- Per adattare il codice in un parser LR(1), è necessario:
  - Memorizzare gli attributi nella **pila** insieme allo stato e al simbolo.
  - Usare uno **schema di nominazione** per accedere agli attributi, come `$n` che fa riferimento alla posizione nella pila.
  - Sequenziare l'applicazione delle regole in base alla riduzione, aggiungendo una **grande dichiarazione** `case` al parser.

## Alternativa: Passeggiata sull'Albero Sintattico Astratto (AST)

Quando le azioni necessarie non si adattano bene alla traduzione sintattica diretta ad-hoc, è possibile utilizzare una **strategia basata sull'albero sintattico astratto (AST)**:

1. **Costruire l'AST**:
   - Invece di eseguire azioni durante il parsing, si costruisce un **albero sintattico astratto** che rappresenta la struttura della grammatica in modo più compatto e astratto rispetto all'albero di derivazione.
   - Ad esempio:
     - Un nodo per ogni operazione (es. somma, moltiplicazione).
     - Foglie per ogni valore (es. numeri o variabili).
   - Ogni nodo dell'albero contiene i dati e le regole necessarie per la successiva analisi o elaborazione.

2. **Eseguire azioni durante le passeggiate sull'AST**:
    - Una volta costruito l'AST, le azioni semantiche vengono eseguite tramite una o più **visite dell'albero** (**tree walks**).
    - Questo approccio è particolarmente comune nei linguaggi orientati agli oggetti, dove si utilizza il **pattern Visitor**:
        - Ogni nodo dell'albero implementa un metodo che definisce come elaborare quel nodo.
        - Un **Visitor** percorre l'albero e richiama i metodi corrispondenti ai nodi.
3. **Passaggi multipli per maggiore flessibilità**:
    - L'AST consente di effettuare più passaggi (o **tree walks**) per attività diverse:
        - Primo passaggio: verifica e calcolo degli attributi.
        - Secondo passaggio: generazione di codice intermedio.
        - Terzo passaggio: ottimizzazioni.
    - Questo approccio offre una maggiore **flessibilità**, consentendo di separare le fasi di analisi e generazione del codice.