8. Parsing bottom-up

In generale, il parsing ha come obiettivo la costruzione di una **derivazione**, ovvero di una serie di passi di riscrittura del tipo

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{frase}$$

In cui, per ogni γ_i:

- Se γ_i è composta unicamente da simboli terminali, essa rappresenta una frase all'interno del linguaggio $L(G)$
- Quando γ_i include uno o più non-terminali, si tratta di una **forma sentenziale**

In particolare, per passare da γ_{i-1} a γ_i , si espande un non-terminale $A \in \gamma_{i-1}$ utilizzando la produzione $A \to \beta$, sostituendo l'occorrenza di A con β per ottenere γ_i . Ovviamente, nel caso di una **derivazione leftmost**, si espande sempre il **primo** non-terminale A di γ_{i-1}.

Nota. Una forma sentenziale leftmost si riferisce a una derivazione leftmost, mentre una forma sentenziale rightmost si riferisce a una derivazione rightmost.

I **parser bottom-up** operano in maniera inversa rispetto alla derivazione rightmost. Infatti, un parser bottom-up lavora partendo dalla frase di input fino al simbolo iniziale S, seguendo una serie di riscritture del tipo

$$\text{frase} \Rightarrow \gamma_n \Rightarrow \gamma_{n-1} \Rightarrow \cdots \Rightarrow \gamma_2 \Rightarrow \gamma_1 \Rightarrow \gamma_0 \Rightarrow S$$

Per trasformare γ_i in γ_{i-1}, il parser confronta un lato destro β con γ_i, e poi sostituisce β con il lato sinistro corrispondente, A (utilizzando la produzione $A \to \beta$).

In termini di **parse tree**, il parser procede dalle foglie verso la radice. I nodi senza genitori in un albero parziale formano la **frangia superiore**, e ogni sostituzione di β con A riduce la frangia. Per questo motivo, questo fenomeno che prende il nome di **riduzione**.

Nota. Una **derivazione rightmost inversa** analizza la frase da sinistra a destra.

L'albero di parsing non deve necessariamente essere costruito esplicitamente, ma può essere simulato. Il numero di nodi nell'albero di parsing è dato dalla somma dei simboli terminali e delle riduzioni:

$$|\text{nodi dell'albero di parsing}| = |\text{simboli terminali}| + |\text{riduzioni}|$$

Il cuore del parsing consiste nello scansionare l'input per individuare la prossima riduzione, un processo che deve essere eseguito in modo efficiente. Un **handle**, in questo contesto, è

una sottostringa particolare. Formalmente, un handle di una forma sentenziale (rightmost) γ è una coppia $\langle A \to \beta, k \rangle$, dove:

- $A \to \beta \in P$
- k è la posizione nella stringa γ dell'ultimo simbolo di β

Se $\langle A \to \beta, k \rangle$ è un handle, sostituire β in k con A produce la forma sentenziale rightmost da cui γ è derivata.

Poiché γ è una forma sentenziale rightmost, la sottostringa a destra di un handle contiene solo simboli terminali, il che implica che il parser non ha bisogno di scansionare molto oltre l'handle per procedere con la riduzione.

Abstract View

Un parser **bottom-up** individua ripetutamente un handle nella forma sentenziale destra corrente e sostituisce β con A per ridurlo. Il procedimento si basa sul seguente algoritmo concettuale.

Per $i \leftarrow n$ a 1, decrementando di 1:

- Trova l'handle $\langle A_i \to \beta_i, k_i \rangle$ in γ_i
- Sostituisci β_i con A_i per ottenere γ_{i-1}

Questo richiede un totale di $2n$ passaggi.

La struttura della frangia superiore dell'albero di parsing parzialmente completo è $(NT|T)^* \, T^*$, dove l'handle compare sempre con la sua estremità destra nella giunzione tra $(NT|T)^*$ e T^* (questo punto sarà fondamentale per il parsing **LR**, ed è detto **hot spot**)

Il parser mantiene il prefisso della frangia superiore dell'albero di parsing su uno stack, il che rende irrilevante la posizione esatta degli elementi. Gli handle, infatti, compaiono in cima allo stack, e tutte le informazioni necessarie per prendere decisioni si trovano nell'hot spot:

- La prossima parola nel flusso di input
- Il **non terminale** più a destra nella frangia e i suoi vicini immediati a sinistra

In un parser **LR**, informazioni aggiuntive sono fornite da uno "stato"

Teorema. Se una grammatica G è non ambigua, allora ogni forma sentenziale destra ha un handle univoco

Shift-reduce parsing

Per implementare un parser **bottom-up**, si adotta il paradigma **shift-reduce**. Un parser **shift-reduce** funziona come un automa a stack, con quattro azioni principali che governano il processo di parsing:

1. **Shift** — La prossima parola dall'input viene spostata sullo stack.
2. **Reduce** — Viene individuata l'estremità destra dell'handle, che si trova in cima allo stack. Una volta identificato:
   - Si individua l'estremità sinistra dell'handle all'interno dello stack.
   - L'handle viene rimosso dallo stack e sostituito con il lato sinistro (lhs) della produzione corrispondente.
3. **Accept** — Il parsing termina con successo.
4. **Error** — Viene richiamata una routine di gestione e recupero degli errori.

Le azioni **Accept** e **Error** sono semplici da implementare. L'azione **Shift** consiste in un'operazione di push della parola corrente sullo stack e in una chiamata al componente scanner. L'azione **Reduce** è più complessa, poiché richiede di rimuovere dallo stack un numero di simboli pari alla lunghezza del lato destro (rhs) della produzione e di inserire il lato sinistro (lhs), con un totale di $|rhs|$ operazioni di pop e una di push.

```
push INVALID
token ← next_token( )
repeat until (top of stack = Goal and token = EOF)
    if the top of the stack is a handle A→β
        then     // reduce β to A
            pop |β| symbols off the stack
            push A onto the stack
        else if (token ≠ EOF)
            then // shift
                push token
                token ← next_token( )
         else    // need to shift, but out of input
            report an error
```

Un handle, come noto, deve essere una sottostringa di una forma sentenziale γ che soddisfi due condizioni:

1. Deve corrispondere al lato destro β di una produzione $A \to \beta$.
2. Deve esistere una derivazione rightmost che produca la forma sentenziale γ, con $A \to \beta$ come ultima produzione applicata.

Tuttavia, semplicemente cercare corrispondenze tra stringhe e lati destri delle produzioni non è sufficiente per individuare gli handle.

La sfida nell'individuare un handle

Il problema principale è capire quando un handle è stato trovato, senza generare derivazioni inutili. La soluzione consiste nell'utilizzare il **contesto a sinistra**, che è codificato nello stato del parser e nella forma sentenziale corrente. Inoltre, il parser può fare affidamento su un **lookahead**, ossia l'analisi della prossima parola in input, che si trova subito dopo l'handle.

Gli stati del parser sono costruiti attraverso un'analisi della **raggiungibilità** sulla grammatica, ossia un'analisi che determina quali produzioni sono applicabili in quale punto del parsing. Tutte queste informazioni vengono utilizzate per costruire un **DFA** (deterministic finite automaton, o automa a stati finiti deterministico) che è in grado di riconoscere gli handle in modo efficiente, guidando così il processo di riduzione durante il parsing.

In questo modo, il **shift-reduce parser** sfrutta un'analisi del contesto per gestire correttamente la transizione tra gli stati del parsing, identificando gli handle e riducendo l'input fino a ottenere la frase finale corretta o segnalare errori se il parsing non può essere completato con successo

Parser LR(1)

I parser **LR(1)** sono una classe di parser **shift-reduce** che utilizzano tabelle e un contesto destro limitato a un solo token (**1 lookahead**) per il riconoscimento degli handle. La classe di grammatiche che questi parser sono in grado di riconoscere viene chiamata **set di grammatiche LR(1)**.

Una grammatica è definita **LR(1)** se, data una derivazione rightmost del tipo:

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{frase}$$

è possibile:

1. Isolare l'handle di ogni forma sentenziale destra γ_i.
2. Determinare quale produzione applicare per ridurre, scansionando γ_i da sinistra a destra e osservando al massimo un simbolo oltre l'estremità destra dell'handle, cioè il **lookahead**.

In altre parole, un parser **LR(1)** elabora l'input da sinistra a destra (**L**) e costruisce una derivazione rightmost in ordine inverso (**R**). Il **1** nel nome **LR(1)** indica che il parser fa affidamento su un solo simbolo di **lookahead** per decidere le riduzioni. Questo significa che il parser guarda un simbolo in avanti nell'input per identificare con certezza quale produzione applicare per ridurre la forma sentenziale corrente.

Da handle a parser

Consideriamo le forme sentenziali durante una derivazione, rappresentate come

$$NT^* \ (NT|T)^* \ T^*$$

Queste forme corrispondono alla parte superiore dell'albero sintattico parzialmente completato. Il suffisso T^* rappresenta, in ogni fase, l'input non ancora letto.

Il nostro parser *shift-reduce* funziona mantenendo la parte $NT^* \ (NT|T)^*$ su uno stack. Il simbolo più a sinistra si trova in fondo allo stack, mentre quello più a destra è in cima. Il parser esegue quindi una ricerca degli handles") dallo stack top (in alto) verso lo stack bottom (in basso). Se la ricerca fallisce, il parser esegue uno *shift*, spostando un altro terminale nello stack, continuando il processo di ricerca della maniglia corretta per completare la riduzione e costruire la struttura sintattica finale.

Di seguito è mostrato uno shift-reduce parser concettuale

```
push INVALID
word ← NextWord( )
repeat until (top of stack = Goal and word = EOF)
    if the top of the stack is a handle A→β
        then     // reduce β to A
            pop |β| symbols off the stack
            push A onto the stack
        else if (word ≠ EOF)
            then // shift
                push word
                word ← NextWord( )
         else    // need to shift, but out of input
            report an error
```

Il funzionamento del parsing LR(1) si basa sull'idea di una grammatica non ambigua, che garantisce una derivazione destra unica. Durante il processo, il parser mantiene la parte superiore della derivazione su uno stack, dove ogni maniglia attiva include la parte superiore dello stack (TOS - Top of Stack). Il parser esegue degli *shift* sull'input finché il TOS non diventa l'estremità destra di una maniglia.

Poiché il linguaggio delle maniglie è regolare e finito, si costruisce un DFA (automa a stati finiti deterministico) in grado di riconoscere le maniglie e controllare il riconoscitore basato sullo stack. Le tabelle **ACTION** e **GOTO** codificano il comportamento del DFA. Quando il parser riconosce un sottotermine, invoca il DFA in modo ricorsivo, lasciando lo stato corrente del vecchio DFA sullo stack e proseguendo.

Quando il DFA raggiunge uno stato finale, ciò indica che deve essere eseguita una riduzione. A questo punto, il parser estrae il lato destro della produzione dallo stack, rivelando lo stato che ha invocato la riduzione. Questo stato è quello in cui "sarebbe stato legale riconoscere un x, e lo abbiamo fatto".

Una volta completata la riduzione, il parser prende un nuovo stato dalla tabella GOTO, usando lo stato rivelato e il simbolo non terminale (lhs) appena ridotto. Il DFA quindi effettua una transizione sul nuovo NT (il simbolo non terminale) e il parser continua il processo, costruendo progressivamente la derivazione.

Costruzione delle tabelle $LR(1)$

Per generare le tabelle ACTION e GOTO, utilizziamo la grammatica per costruire un modello dell'automa di controllo (Control DFA) che gestirà il parsing. Il processo prevede la codifica delle azioni e delle transizioni del DFA nelle tabelle ACTION e GOTO. Se la costruzione di queste tabelle ha successo, significa che la grammatica è LR(1), e "successo" significa che ogni voce delle tabelle è definita in modo univoco, senza ambiguità.

In termini generali, il processo inizia modellando lo stato del parser, che viene rappresentato da una serie di stati DFA e dalle transizioni tra questi. Per costruire questi stati e le funzioni di transizione, utilizziamo due funzioni principali: `goto(s, X)` e `closure(s)`.

La funzione `goto()` è analoga alla funzione `move()` usata nella costruzione dei sottoinsiemi, e determina come lo stato del parser cambia quando si "muove" attraverso un simbolo di input o non terminale *X* a partire da uno stato *s*.

La funzione `closure()` completa lo stato corrente, aggiungendo informazioni su quali altre produzioni potrebbero essere pertinenti in quel punto del parsing. In altre parole, espande lo stato aggiungendo tutte le possibili derivazioni da esplorare.

Una volta costruiti gli stati e le funzioni di transizione del DFA, utilizziamo queste informazioni per riempire le tabelle ACTION e GOTO. ACTION contiene le decisioni su *shift*, *reduce*, *accept*, o *error* per ciascun terminale e stato, mentre GOTO specifica le transizioni verso nuovi stati quando si processano simboli non terminali.

LR(k) item

Gli *LR(k) items* rappresentano una configurazione valida di un parser LR(1) utilizzata nell'algoritmo di costruzione della tabella LR(1). Un item LR(k) è definito come una coppia $[P, \delta]$, dove:

- P è una produzione del tipo $A \to \beta$ con un simbolo • in una posizione qualsiasi nella parte destra della produzione.
- δ è una stringa di lookahead di lunghezza $\leq k$, che può essere costituita da parole o dal simbolo EOF.

Il simbolo • in un item indica la posizione della cima dello stack. Gli item possono essere interpretati come segue:

- $[A \to \bullet\ \beta\gamma, a]$ significa che l'input visto finora è coerente con l'uso della produzione $A \to \beta\gamma$ immediatamente dopo il simbolo che si trova in cima allo stack.
- $[A \to \beta\ \bullet \gamma, a]$ significa che l'input visto finora è coerente con l'uso della produzione $A \to \beta\gamma$ in questo punto del parsing, e che il parser ha già riconosciuto β, il che implica che β si trova in cima allo stack.
- $[A \to \beta\gamma\ \bullet, a]$ significa che il parser ha già visto $\beta\gamma$ e che il simbolo di lookahead a è coerente con la riduzione verso A.

In pratica, gli item LR(k) vengono utilizzati per tracciare lo stato di avanzamento del parser all'interno delle produzioni e per determinare le azioni di *shift*, *reduce* o *accept* basate sul contesto fornito dall'input e dalla struttura della grammatica.

LR(1) item

La produzione $A \to \beta$, dove $\beta = B_1B_2B_3$ con lookahead a, può generare 4 elementi: $[A \to \bullet\ B_1B_2B_3, a]$, $[A \to B_1 \bullet\ B_2B_3, a]$, $[A \to B_1B_2\ \bullet\ B_3, a]$, e $[A \to B_1B_2B_3\ \bullet, a]$. L'insieme degli elementi LR(1) per una grammatica è finito.

Qual è lo scopo di tutti questi simboli di lookahead? Sono portati avanti per aiutare a scegliere la riduzione corretta. I lookahead sono principalmente un modo di tenere traccia delle informazioni, a meno che l'elemento non abbia il simbolo • alla fine destra. Non hanno uso diretto in $[A \to \beta\ \bullet\ \gamma, a]$, ma in $[A \to \beta\ \bullet, a]$, un lookahead di a implica una riduzione tramite $A \to \beta$.

Per l'insieme di elementi $\{[A \to \beta\ \bullet, a], [B \to \gamma\ \bullet\ \delta, b]\}$, se $a \Rightarrow$ riduzione a A, allora $FIRST(\delta) \Rightarrow$ shift. Un contesto limitato a destra è sufficiente per determinare le azioni da intraprendere.

Costruzione delle tabelle LR(1)

Per costruire la collezione canonica di insiemi di elementi LR(1), indicata come I, si inizia con uno stato appropriato, s_0, ad esempio $[S' \to \bullet\ S, EOF]$, insieme a qualsiasi elemento equivalente. Si derivano gli elementi equivalenti calcolando la chiusura di s_0, ovvero

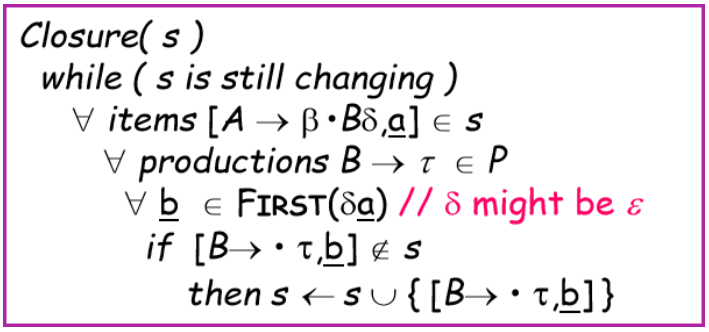
$closure(s_0)$. Successivamente, si calcola ripetutamente, per ogni stato s_k e ogni simbolo X, l'operazione $goto(s_k, X)$. Se l'insieme risultante non è già presente nella collezione, viene aggiunto.

Durante questo processo, si registrano tutte le transizioni create dall'operazione $goto()$. Alla fine, questo processo raggiunge un punto fisso, dove non si generano più nuovi insiemi. A questo punto, si completa la tabella utilizzando la collezione di insiemi di elementi LR(1) ottenuti.

Calcolo delle chiusure

La funzione di chiusura $Closure(s)$ aggiunge tutti gli elementi implicati dagli elementi già presenti in s. Un elemento del tipo $[A \to \beta \bullet B\delta, a]$ implica un elemento del tipo $[B \to \bullet \tau, x]$ per ciascuna produzione con B sul lato sinistro (lhs) e per ciascun $x \in FIRST(\delta a)$.

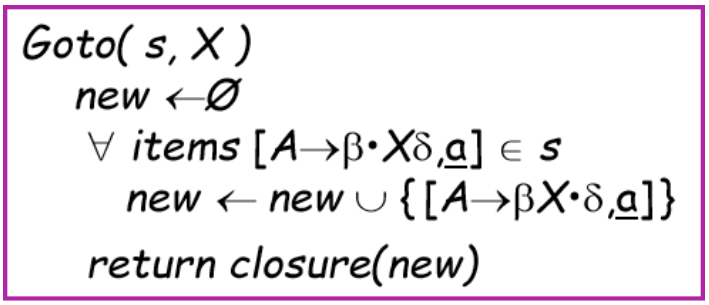
Poiché $\beta B\delta$ è valido, qualsiasi modo per derivare $\beta B\delta$ è anch'esso valido. La chiusura consente di espandere gli elementi inclusi nello stato, considerando tutte le produzioni possibili per i simboli successivi al punto • .



Calcolo delle goto

L'operazione $Goto(s, x)$ calcola lo stato che il parser raggiungerebbe se riconoscesse un simbolo x mentre si trova nello stato s. In termini pratici, $Goto(\{[A \to \beta \bullet X\delta, a]\}, X)$ produce l'elemento $[A \to \beta X \bullet \delta, a]$.

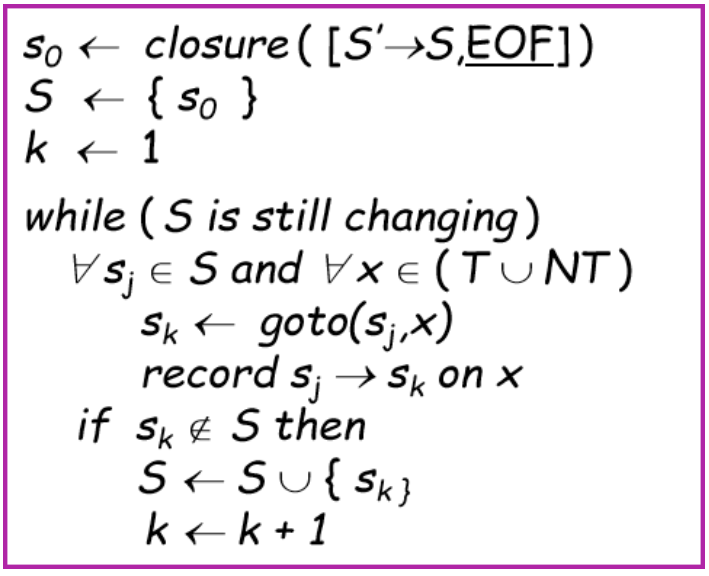
L'operazione individua tutti gli elementi di questo tipo e utilizza la funzione $closure()$ per completare lo stato, aggiungendo gli elementi implicati dalle nuove configurazioni raggiunte.



Costruzione della collezione canonica

Si inizia con lo stato $s_0 = closure([S' \rightarrow \ \bullet \ S, EOF])$. A partire da questo stato, si costruiscono ripetutamente nuovi stati utilizzando l'operazione $Goto()$ per ogni simbolo possibile, fino a quando tutti gli stati possibili sono stati trovati.

Ogni volta che un nuovo stato viene generato tramite $Goto(s_k, X)$, viene calcolata la sua chiusura. Se questo nuovo stato non è già presente nella collezione, viene aggiunto. Questo processo continua fino a quando non si scoprono più nuovi stati, raggiungendo così una collezione canonica completa di insiemi di elementi LR(1).



Riempire le tabelle $ACTION$ e $GOTO$

```
∀ set Sx ∈ S
  ∀ item i ∈ Sx
    if  i is [A→β•aδ,b] and goto(Sx,a) = Sk , a ∈ T
       then ACTION[x,a] ← "shift k"
    else if  i is [S'→S •,EOF]
         then ACTION[x ,EOF] ← "accept"
    else if  i is [A→β•,a]
         then ACTION[x,a] ← "reduce A→β"
  ∀ n ∈ NT
    if  goto(Sx ,n) = Sk
       then GOTO[x,n] ← k
```

Cosa può andare storto?

Supponiamo che l'insieme s contenga gli elementi $[A \to \beta \cdot a\gamma, b]$ e $[B \to \beta \cdot, a]$. Il primo elemento genera un'azione di "shift", mentre il secondo genera un'azione di "reduce". Entrambi definiscono $ACTION[s, a]$, ma non è possibile eseguire entrambe le azioni contemporaneamente. Questa è una ambiguità fondamentale, chiamata errore di shift/reduce. Per risolverlo, è necessario modificare la grammatica per eliminarlo (ad esempio, nei costrutti *if-then-else*). Spesso, lo *shift* risolve correttamente il conflitto, ma non sempre.

Se invece l'insieme s contiene $[A \to \gamma \cdot, a]$ e $[B \to \gamma \cdot, a]$, entrambi gli elementi generano un'azione di "reduce", ma con produzioni differenti. Anche in questo caso, entrambe le produzioni definiscono $ACTION[s, a]$, ma non è possibile eseguire entrambe le riduzioni. Questa ambiguità è chiamata errore di reduce/reduce, e anche in questo caso bisogna modificare la grammatica per eliminarlo (come l'overloading di *PL/I* con il simbolo *(...)*).

In entrambi i casi, la grammatica non è di tipo LR(1).

Riduzione delle tabelle

Esistono tre opzioni principali per ridurre le dimensioni delle tabelle di parsing:

1. **Combinare terminali simili**, come numeri e identificatori, o operatori come $+$ e $-$, $*$ e $/$. Questo approccio rimuove direttamente una colonna dalla tabella e, in alcuni casi, può rimuovere anche una riga. Per esempio, nella grammatica delle espressioni, si può passare da 384 a 198 voci nella tabella.
2. **Combinare righe o colonne** della tabella. Questo metodo consiste nell'implementare una sola volta le righe o colonne identiche e rimappare gli stati. Tuttavia, richiede un'ulteriore indirezione per ogni ricerca nella tabella. Si può usare una mappatura separata per l'operazione $ACTION$ e per l'operazione $GOTO$.

3. **Usare un altro algoritmo di costruzione**. Algoritmi come LALR(1) e SLR(1) producono tabelle più piccole rispetto all'algoritmo LR(1) canonico. Implementazioni di questi algoritmi sono facilmente disponibili e spesso preferite quando lo spazio delle tabelle è una considerazione importante.