Compilatori

Corso di Laurea in Informatica

Mauro Leoncini

A.A. 2024/2025

Linguaggi e compilatori

1. Analisi sintattica (PARTE PRIMA)
   - Generalità sul parsing
   - Parser a discesa ricorsiva
   - Parser predittivi

Compilatori

1. Analisi sintattica (PARTE PRIMA)
   - Generalità sul parsing
   - Parser a discesa ricorsiva
   - Parser predittivi

L'input per il parser

- Nel contesto della compilazione l'input per il parser è costituito da una stringa di token generata dall'analizzatore lessicale.
- Per semplicità di linguaggio ignoreremo la presenza dello scanner e supporremo che il parser legga direttamente i token dallo stream di input.
- Per individuare la "fine" della stringa di input, supporremo che la stringa stessa sia terminata da uno speciale simbolo
- L'unico requisito (ovvio) è che tale simbolo speciale non sia anche un token del linguaggio
- È convenzione diffusa utilizzare il simbolo del dollaro: \$.

L'input per il parser

- Se $\mathcal{S}$ è l'assioma iniziale della grammatica, per tenere conto del simbolo di terminazione si introduce "formalmente" un nuovo assioma $\mathcal{S}'$, con la sola produzione $\mathcal{S}' \rightarrow \mathcal{S}\$$.
- In questo modo, $\mathcal{S} \stackrel{*}{\Rightarrow} \theta$ se e solo se $\mathcal{S}' \stackrel{*}{\Rightarrow} \theta\$$, dove θ indica naturalmente la stringa di token
- Nel seguito lasceremo implicita questa "aggiunta" alla grammatica, a meno che non risulti importante considerarla per comprendere meglio qualche altro concetto

Tipi di parser

- Una prima classificazione suddivide il parsing in accordo all'ordine di costruzione del parse tree per θ.
- Nel parsing *top-down* l'albero viene costruito a partire dalla radice, corrispondente all'assioma iniziale.
- Equivalentemente, possiamo dire che nel parsing top-down si cerca una derivazione canonica sinistra per $\theta\$$ partendo da $\mathcal{S}'$.
- Si noti che la costruzione top-down dell'albero di derivazione corrisponde in modo naturale al riconoscimento di variabili sintattiche (es, un comando o una espressione) in termini delle parti costituenti.

Tipi di parser

- Nel parsing *bottom-up* il parse tree per θ viene costruito procedendo dalle foglie verso la radice.
- Equivalentemente, possiamo dire che nel parsing bottom-up si cerca una derivazione canonica (destra) per la stringa $\theta\$$ applicando *riduzioni* successive.
- Una riduzione non è nient'altro che l'applicazione "in senso opposto" di una produzione della grammatica.
- Si noti che la costruzione bottom-up dell'albero di derivazione corrisponde in modo naturale al riconoscimento di singole porzioni di un programma e alla loro composizione in parti più complesse.

Tipi di parser (continua)

- I tipi di parser più diffusi includono:
  - parser a *discesa ricorsiva* con backtracking,
  - parser a discesa ricorsiva senza backtracking (parsing *predittivi*),
  - parser di tipo *shift-reduce*.
- I parser a discesa ricorsiva sono di tipo top-down, mentre i parser shift-reduce sono di tipo bottom-up.
- Considereremo sottoinsiemi di grammatiche libere per cui si possono costruire parser efficienti a discesa ricorsiva (grammatiche $LL(1)$) o di tipo shift-reduce (grammatiche $LR(1)$)

Scelta della produzione

- Al generico passo, di derivazione (parser top-down) o di riduzione (parser bottom-up), il parser deve decidere quale produzione utilizzare
- Tale scelta viene effettuata in funzione dello *stato interno* del parser e della "prossima" *porzione di input*
- Lo stato interno del parser (almeno nei parser *LR)*)è tipicamente costituito dall'informazione memorizzata nella cima di una struttura dati stack.
- Il numero di token considerati per prendere la decisione è noto invece con il termine di *lookahead*
- In generale l'interesse è per valori di lookhaed limitati, 0 o 1 token

Compilatori

Algoritmo generico

- Un parser a discesa ricorsiva (d.r.) costruisce il parse tree (eventualmente non in modo esplicito) a partire dall'assioma ed esaminando progressivamente l'input.
- Al generico passo, l'algoritmo è idealmente "posizionato" su un nodo x dell'albero.
- Se il nodo è una foglia etichettata con un simbolo terminale a, l'algoritmo controlla se il prossimo simbolo in input coincide con a.
- In caso affermativo fa avanzare il puntatore di input, altrimenti (nel caso più semplice) dichiara errore.
- Se invece il nodo è un simbolo non terminale A, sceglie una produzione $A \rightarrow X_1 X_2 \ldots X_k$, crea i nodi (figli di A) etichettandoli con $X_1, X_2, \ldots, X_k$, e passa ricorsivamente ad esaminare tali nodi, da sinistra verso destra.

Algoritmo generico

- Da un punto di vista implementativo, un parser a d.r. può essere realizzato come una collezione di procedure mutuamente ricorsive, una per ogni simbolo non terminale della grammatica.
- Il problema fondamentale consiste nella "scelta" della produzione da applicare, nel caso in cui (per un dato non terminale) ne esista più d'una.
- Lo pseudocodice nella diapositiva seguente lascia aperto questo problema, che andremo successivamente a risolvere in almeno due modi diversi.

Procedura per il generico non terminale A

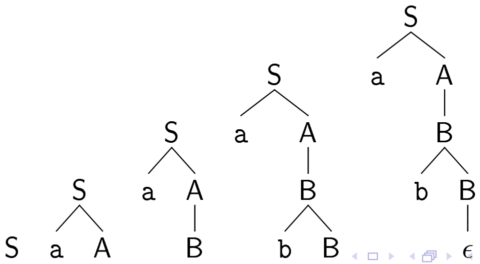
Scegli "opportunamente" una produzione $A \to X_1 X_2 \dots X_k$ $(X_j \in \mathcal{V})$
for $j = 1, \dots, k$ **do**
    if $X_j \in \mathcal{N}$ **then**
        $X_j()$
    else
        $x \leftarrow$ read()
        if $X_j \neq x$ **then**
            error() {Include il caso $x =$ EOF}

Esempio

- Consideriamo la "solita" grammatica $G_{n,m}$, che genera il linguaggio $\{\mathtt{a}^n\mathtt{b}^m : n > 0, m \geq 0\}$:

$$
\begin{array}{rcl}
S & \rightarrow & \mathtt{a}A \\
A & \rightarrow & \mathtt{a}A \mid B \\
B & \rightarrow & \mathtt{b}B \mid \epsilon
\end{array}
$$

- Su input ab un parser a d.r. nondeterministico produce il parse tree attraverso la sequenza di costruzioni elencate di seguito:



Implementazione concreta

- Per eliminare il non determinismo insito nel codice precedente, una prima soluzione consiste nell'esplorare tutte le possibili produzioni relative al generico non terminale A prima eventualmente di dichiarare errore.
- Se una particolare produzione fallisce, ma prima del fallimento sono stati letti simboli di input, è necessario operare un *backtracking* sull'input stesso.
- Per i parser a discesa ricorsiva ciò può essere sufficientemente agevole (dal punto di vista dell'implementatore), anche se computazionalmente pesante.
- Questa prima variante è mostrata nella diapositiva successiva.

Procedura con backtracking per A

```
saveInputPointer()
for all production A → X_1 X_2 ... X_k (X_j ∈ 𝒱) do
    fail ← False
    for j ← 1 ... k do
        if X_j ∈ 𝒩 and X_j() then
            continue
        if X_j ∈ 𝒯 then
            x ← read()
            if X_j = x then
                continue
        restoreInputPointer()
        fail ← True
        break
    if not fail then
        return True
return False
```

Procedura con backtracking per $\mathcal{S}$

- La procedura precedente va modificata nel caso dell'assioma $\mathcal{S}$ che deve dichiarare il successo complessivo o il fallimento
- Per questo è sufficiente rimpiazzare la riga 16 con il seguente codice:

16: **if not** eof() **then**
17:     Fail()
18: **else**
19:     Success()

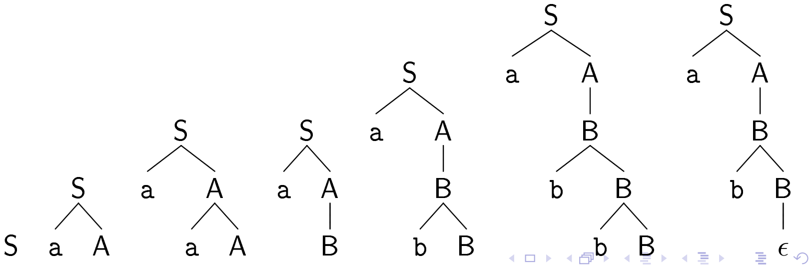
dove Fail() e Success() sono due opportune procedure che "informano" il main program sull'esito del parsing

Esempio

- Riconsideriamo la grammatica

$$\begin{array}{rcl} S & \rightarrow & \mathtt{a}A \\ A & \rightarrow & \mathtt{a}A \mid B \\ B & \rightarrow & \mathtt{b}B \mid \epsilon \end{array}$$

- Su input ab un parser a d.r. con backtracking potrebbe produrre la seguente derivazione:



Parsing a discesa ricorsiva

- Se analizziamo attentamente il codice del parser a d.r., comprendiamo perché una grammatica con ricorsioni a sinistra sia inadatta al parsing a discesa ricorsiva.
- Supponiamo, infatti, che ad un determinato passo il parser sia "posizionato" su un nodo (etichettato con il non terminale) A dell'albero.
- Supponiamo inoltre che la prima produzione che viene (tentativamente) applicata abbia una ricorsione a sinistra, sia cioè del tipo $A \to A\alpha$ (dove α è una qualunque stringa di terminali e/o non terminali).
- Accade allora che il codice relativo al non terminale A:
  - consideri il primo simbolo della parte sinistra della produzione, che è ancora A;
  - chiami ricorsivamente la procedura per A, innescando così un ciclo infinito.

Esempio

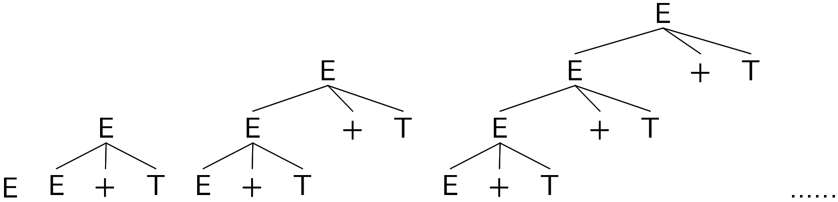
- Per la grammatica G_{expr_2} con precedenza di operatori (che abbiamo già incontrato):

$$\begin{array}{rcl} E & \rightarrow & E+T \mid T \\ T & \rightarrow & T \times F \mid F \\ F & \rightarrow & \textbf{number} \mid (E) \end{array}$$

  le procedure per i non terminali E e T innescano potenzialmente un ciclo infinito.

- Ad esempio, su input **number** $+$ **number**, la produzione corretta da applicare inizialmente è $E \rightarrow E+E$ (se si applica $E \rightarrow T$ la derivazione fallisce e bisogna operare backtracking sull'input), ma questa innesca un ciclo infinito.

Esempio



Eliminazione delle ricorsioni sinistre

- Supponiamo che la grammatica in input non contenga riscritture del tipo $A \to \epsilon$ e che non ammetta cicli, ovvero derivazioni del tipo $A \stackrel{*}{\Rightarrow} A$, per nessun non terminale A.
- Il primo requisito può sembrare eccessivo, perché se non esistono produzioni $A \to \epsilon$ allora il linguaggio non può contenere la stringa vuota.
- Tuttavia, in questo caso, è possibile aggiungere "alla fine" (cioè dopo aver manipolato la grammatica) la sola produzione $S \to \epsilon$.

Eliminazione delle ricorsioni sinistre

- Sia $A_1, \ldots, A_n$ un ordinamento (arbitrario) dei simboli non terminali.
- L'idea dell'algoritmo è di ottenere una grammatica in cui, per ogni riscrittura del tipo $A_i \rightarrow A_j\alpha$ si abbia $i < j$.
- Nella spiegazione che segue, ci farà comodo poter fare riferimento a produzioni del tipo $A_i \rightarrow A_j\alpha$ distinguendo i casi $i < j$ e $i > j$
- Ci riferiremo alle prime come a *forward production* (produzioni *in avanti*) e alle seconde come a *backward production* (produzioni *all'indietro*)
- L'algoritmo consta di due cicli annidati:
  1. Per ogni valore di $i = 1, 2, \ldots, n$,
     - per ogni valore di $j = 1, \ldots, i-1$ si eliminano le backward production dl tipo $A_i \rightarrow A_j\alpha$
  2. Si eliminano le eventuali produzioni del tipo $A_i \rightarrow A_i\alpha$

Eliminazione delle ricorsioni "dirette"

- Supponiamo che, per il non terminale A, siano presenti le produzioni:

$$A \rightarrow A\alpha_1 | \ldots | A\alpha_t | \beta_1 | \ldots | \beta_m$$

in cui nessuna stringa β_s, $s = 1, \ldots, m$, inizia per A

- L'eliminazione di A in una derivazione richiede che "prima o poi" si usi una produzione $A \rightarrow \beta_s$ (per questo ne deve esistere almeno una)
- Alla luce di quest'ultima osservazione, le produzioni per A possono essere eliminate in una derivazione prevedendo prima le sostituzioni con le sequenze β_s e poi inserendo un numero arbitrario di sequenze α_r utilizzando un nuovo non terminale (che chiameremo A')

$$\begin{array}{rcl} A & \rightarrow & \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A' \\ A' & \rightarrow & \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_t A' \mid \epsilon \end{array}$$

Eliminazione delle “backward” production

- L’eliminazione di tali produzioni si basa su un ragionamento ricorsivo
- Se abbiamo ordinato i non terminali, possiamo innanzitutto osservare che, per il non terminale A_1, una backward production non può esserci
- Semmai può esserci una ricorsione diretta, che però possiamo eliminare nel modo indicato nella slide precedente
- Questa è dunque la “base” del ragionamento ricorsivo
- L’ipotesi induttiva è quindi che, per un dato indice $i \geq 2$, per i primi $i-1$ non terminali non ci siano backward production
- Consideriamo ora il non terminale A_i e supponiamo che una delle sue produzioni sia di tipo “backward”

$$A_i \rightarrow A_j\alpha \mid \ldots \qquad j < i$$

Eliminazione delle "backward" production (2)

- La produzione $A_i \to A_j\alpha$ può essere "manualmente" eliminata inserendo esplicitamente, al posto di A_j, una ad una tutte le parti destre delle sua produzioni
- Se cioè avessimo $A_j \to \beta_1|\dots|\beta_m$, allora A_i avrebbe le produzioni

$$A_i \to \beta_1\alpha \mid \beta_2\alpha \mid \dots \mid \beta_m\alpha$$

- Si noti che, per ipotesi induttiva, le produzioni $A_j \to \beta_1 \mid \dots \mid \beta_m$ sono tutte di tipo forward, ma lo sono appunto rispetto a A_j e non rispetto al non terminale "attuale" A_i
- Non si può cioè escludere che un qualche β_s inizi con un non terminale A_k che ancora precede A_i
- Abbiamo però fatto comunque un passo avanti perché $k > j$ e dunque in al più $i - j$ passi le backward production vengono eliminate

Esempio 1

- Consideriamo ancora la grammatica G_{expr_1} per le espressioni

$$E \rightarrow E + E \mid E * E \mid (E) \mid \textbf{number}$$

  che contiene solo ricorsioni immediate (anche perché include un solo non terminale).
- Le produzioni vengono sostituite nel modo seguente, con l'introduzione di un non terminale E'

$$\begin{array}{rcl} E & \rightarrow & (E)\,E' \mid \textbf{number}\,E' \\ E' & \rightarrow & +\,E\,E' \mid *\,E\,E' \mid \epsilon \end{array}$$

- Un parser a discesa ricorsiva con backtracking per questa grammatica è presente nel repository condiviso su `gdrive`

Esempio 2

- Consideriamo la grammatica così definita:

$$\begin{array}{rcl} A & \rightarrow & B\mathtt{b} \mid \mathtt{a} \\ B & \rightarrow & B\mathtt{b} \mid A\mathtt{c} \end{array}$$

  che consente la derivazione $A \stackrel{+}{\Rightarrow} A\mathtt{cb}$.
- Se nell'ordinamento dei non terminali A si fa precedere a B diviene necessario dapprima eliminare la produzione $B \rightarrow A\mathtt{c}$.
- Questo comporta l'introduzione delle due produzioni $B \rightarrow B\mathtt{bc} \mid \mathtt{ac}$.
- La successiva eliminazione delle ricorsioni immediate ($B \rightarrow B\mathtt{b} \mid B\mathtt{bc}$) porta alla seguente grammatica modificata

$$\begin{array}{rcl} A & \rightarrow & B\mathtt{b} \mid \mathtt{a} \\ B & \rightarrow & \mathtt{ac}B' \\ B' & \rightarrow & \mathtt{b}B' \mid \mathtt{bc}B' \mid \epsilon \end{array}$$

Compilatori

Parsing top-down con scelta diretta della produzione

- Una soluzione migliore rispetto all'esplorazione esaustiva delle possibili derivazioni consiste nell'usare un certo numero di caratteri di lookahead per decidere la prossima produzione da utilizzare.
- Naturalmente, se questa strada possa essere percorsa con successo dipende dalla grammatica.
- Consideriamo la semplice grammatica:

$$\begin{array}{rcl} A & \rightarrow & \mathtt{a}A \mid \mathtt{b}B \\ B & \rightarrow & \epsilon \mid \mathtt{b}B \end{array}$$

- Per entrambi i non terminali, la scelta della produzione da usare può essere fatta guardando solo il prossimo simbolo x in input.
  - Per A: se $x = \mathtt{a}$, usa la produzione $A \rightarrow \mathtt{a}A$; se $x = \mathtt{b}$, usa la produzione $A \rightarrow \mathtt{b}B$.
  - Per B. se $x = \$$ (end of input), usa la produzione $B \rightarrow \epsilon$; se $x = \mathtt{b}$, usa la produzione $B \rightarrow \mathtt{b}B$;

Grammatiche $LL(1)$

- Un parser predittivo può essere realizzato agevolmente nel caso di grammatiche cosiddette $LL(1)$.
- La doppia L sta per Left-Left, ad indicare che l'input è letto da sinistra verso destra e che la derivazione prodotta è canonica sinistra.
- Il "parametro" 1 indica che un carattere di lookahead è sufficiente per decidere correttamente la produzione da utilizzare.
- Più in generale, si possono considerare grammatiche $LL(k)$, dove sono sufficienti (e, in generale, necessari) k caratteri di lookahead.

Grammatiche "non" $LL(k)$

- Nessuna grammatica con ricorsioni a sinistra può essere $LL(k)$, per nessun k.
- Anche nel caso in cui esistano produzioni con prefissi comuni la quantità di lookahead necessaria non è limitabile a priori.
- Un esempio grammatica con prefissi comuni è data dal già esaminato caso del "dangling else".

$$\begin{array}{rcl} S & \rightarrow & I \mid A \\ I & \rightarrow & \textbf{if } B \textbf{ then } S \mid \textbf{if } B \textbf{ then } S \textbf{ else } S \\ A & \rightarrow & \textbf{a}, \quad B \rightarrow \textbf{b} \end{array}$$

- Poiché la quantità di codice presente fra le keyword **then** e **else** può essere arbitrariamente lunga (come numero di token) il valore del lookahead non risulta limitabile

Grammatiche "non" $LL(k)$

- Un altro esempio è dato dalla grammatica:

$$\begin{array}{rcl} A & \rightarrow & \mathtt{a}B \mid \mathtt{a}C \\ B & \rightarrow & \mathtt{a}B \mid \mathtt{b} \\ C & \rightarrow & \mathtt{a}C \mid \mathtt{c} \end{array}$$

- Su input $\mathtt{a}^n\mathtt{b}$ è necessario un lookahead di $n+1$ caratteri per decidere inizialmente che la produzione corretta è $A \rightarrow \mathtt{a}B$.

Grammatiche $LL(1)$

- Studiamo ora le condizioni che devono essere soddisfatte perché una grammatica sia $LL(1)$, ricordando che consideriamo derivazioni canoniche sinistre
- Supponiamo di aver già effettuato i riscritture e ottenuto una forma di frase α_i in cui i primi k simboli sono terminali che "coincidono" (match) con i primi k simboli di input
- α_i può quindi essere scritta nel modo seguente

$$\alpha_i = a_1 a_2 \ldots a_k A \beta$$

  che evidenzia il primo non terminale che deve essere riscritto, cioè A
- Poiché stiamo supponendo che il parsing non sia (ancora) fallito, la stringa di input "deve" necessariamente essere $a_1 a_2 \ldots a_k a_{k+1} \ldots a_n$ e dunque $a = a_{k+1}$ è il simbolo "puntato" dal puntatore di input (sia perdonato il gioco di parole)
- β, cioè la stringa a destra di A nella forma di frase attuale, può naturalmente contenere terminali e non terminali

Grammatiche $LL(1)$

- Nella condizione generica in cui ci siamo messi, è evidente che il parsing <u>fallisce</u> se A è la testa di (almeno) due produzioni, poniamo,

$$A \rightarrow \gamma \mid \delta$$

  tali che sia da γ che da δ si può derivare una stringa che inizia per a (il prossimo simbolo di input)

- Più formalmente, se vale

$$\gamma \overset{*}{\Rightarrow} \mathsf{a}\gamma' \qquad \text{e} \qquad \delta \overset{*}{\Rightarrow} \mathsf{a}\delta'$$

  allora G non è $LL(1)$

Esempi

- Il caso più semplice si ha naturalmente quando δ e γ iniziano proprio con lo stesso simbolo terminale.
- Ad esempio, una grammatica che contenesse le produzioni $S \rightarrow \mathsf{a}A \mid \mathsf{a}B$ non potrebbe essere $LL(1)$.
- Il seguente esempio mostra come la situazione possa non essere di immediata verificabilità

$$
\begin{array}{lcl}
S & \rightarrow & \mathsf{a}A \mid B \\
A & \rightarrow & \mathsf{a}A \mid \epsilon \\
B & \rightarrow & \mathsf{b} \mid C \\
C & \rightarrow & \mathsf{a}B \mid \mathsf{c}
\end{array}
$$

- Questa grammatica non è $LL(1)$ perché $B \Rightarrow C \Rightarrow \mathsf{a}B$ e dunque, su input che inizia per a, non è possibile decidere quale produzione usare ($S \rightarrow \mathsf{a}A$ oppure $S \rightarrow B$) guardando solo il primo simbolo di input.

Grammatiche $LL(1)$

- C'è un'altra condizione che deve essere rispettata affinché la grammatica sia $LL(1)$.
- Riconsideriamo la stringa $\alpha_i = a_1 a_2 \ldots a_k A\beta$ che descrive lo stato del parser dopo i applicazioni di regole di riscrittura; come in precedenza, supponiamo che per il non terminale A da riscrivere ci siano almeno due produzioni

$$A \quad \rightarrow \quad \gamma \mid \delta$$

- Se accade che

$$\gamma \stackrel{*}{\Rightarrow} \epsilon \qquad \text{e} \qquad \delta \stackrel{*}{\Rightarrow} \mathsf{a}\delta'$$

allora la grammatica non è $LL(1)$ se il primo simbolo in β è a.

- In tal caso infatti entrambe le produzioni portano al riconoscimento del prossimo simbolo di input, ma non necessariamente al riconoscimento completo dell'input

Esempio

- Si consideri la grammatica

$$\begin{array}{rcl} S & \rightarrow & A\mathtt{a} \\ A & \rightarrow & \mathtt{a}A \mid \epsilon \end{array}$$

- Con input aa, dopo la prima riscrittura $S \Rightarrow A\mathtt{a}$, la scelta giusta per il parser sarebbe di applicare $A \rightarrow \mathtt{a}A$; infatti

$$S \Rightarrow A\mathtt{a} \Rightarrow \mathtt{a}A\mathtt{a} \Rightarrow \mathtt{aa}$$

- Tuttavia anche l'applicazione della regola $A \rightarrow \epsilon$ porterebbe al riconoscimento del primo carattere

$$S \Rightarrow A\mathtt{a} \Rightarrow \mathtt{a}$$

- Con un solo carattere di lookahead il parser non può decidere correttamente; si noti, infatti, che su input a la situazione è rovesciata

FIRST e FOLLOW

- Per esprimere in modo compatto le condizioni che qualificano una data grammatica come $LL(1)$, introduciamo le funzioni $FIRST$ e $FOLLOW$.
- Data $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ e data una stringa $\alpha \in (\mathcal{N} \cup \mathcal{T})^*$, si definisce $FIRST(\alpha)$ l'insieme dei simboli terminali con cui può iniziare una frase derivabile da α, più eventualmente ϵ se $\alpha \Rightarrow^* \epsilon$:

$$\begin{aligned} FIRST(\alpha) &= \{x \in \mathcal{T} | \alpha \Rightarrow^* x\beta, \beta \in \mathcal{T}^*\} \\ &\quad \cup \{\epsilon\}, \text{ se } \alpha \Rightarrow^* \epsilon. \end{aligned}$$

- Per ogni non terminale $A \in \mathcal{N}$, $FOLLOW(A)$ è l'insieme dei terminali che si possono trovare immediatamente alla destra di A in una forma di frase di una qualche derivazione canonica (destra o sinistra):
$x \in FOLLOW(A)$ se $\mathcal{S} \Rightarrow^* \alpha A x \beta$, con $\alpha, \beta \in \mathcal{V}^*$.

Calcolo di $FIRST(\alpha)$

- Definiamo innanzitutto come si calcola $FIRST(\alpha)$ nel caso in cui α sia un singolo simbolo della grammatica, cioè $\alpha = X$ con $X \in \mathcal{N} \cup \mathcal{T}$.
  - Se X è un terminale, si pone naturalmente $FIRST(X) = \{X\}$;
  - se X è un non terminale il calcolo procede per passi, con l'inizializzazione $FIRST(X) = \{\}$.
    - Se esiste la produzione $X \rightarrow X_1 \ldots X_n$, e risulta $\epsilon \in FIRST(X_j)$, $j = 1, \ldots, k-1$, poniamo $FIRST(X) = FIRST(X) \cup \{x\}$ per ogni simbolo $x \in FIRST(X_k)$.
    - Infine, se esiste la produzione $X \rightarrow \epsilon$ oppure $\epsilon \in FIRST(X_j)$, $j = 1, \ldots, k$, poniamo $FIRST(X) = FIRST(X) \cup \{\epsilon\}$.

Esempi

- Si riconsideri la grammatica per le espressioni che "forza" la precedenza di operatori:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T \times F \mid F \\
F & \rightarrow & \textbf{number} \mid (E)
\end{array}
$$

- Per questa grammatica risulta
  - $FIRST(F) = \{(\, , \textbf{number}\}$;
  - $FIRST(T) = FIRST(F)$;
  - $FIRST(E) = FIRST(T)$.

Esempi

- La seguente grammatica genera lo stesso linguaggio della precedente

$$\begin{array}{rcl} E & \rightarrow & (E)\, E' \mid \textbf{number}\, E' \\ E' & \rightarrow & +\, E\, E' \mid \times\, E\, E' \mid \epsilon \end{array}$$

- Risulta
  - $FIRST(E) = \{(\,, \textbf{number}\}$
  - $FIRST(E') = \{+, \times, \epsilon\}$
- Si consideri ora la grammatica per $\mathtt{a}^n\mathtt{b}^m\mathtt{c}^k$

$$\begin{array}{rcl} A & \rightarrow & \mathtt{a}A \mid BC \\ B & \rightarrow & \mathtt{b}B \mid \epsilon \\ C & \rightarrow & \mathtt{c}C \mid \epsilon \end{array}$$

- Per questa grammatica risulta
  - $FIRST(C) = \{\mathtt{c}, \epsilon\}$
  - $FIRST(B) = \{\mathtt{b}, \epsilon\}$
  - $FIRST(A) = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \epsilon\}$

Calcolo di $FIRST(\alpha)$

- Il calcolo di $FIRST(\alpha)$, dove $\alpha = X_1 \dots X_n$ è una generica stringa di terminali e nonterminali, può ora essere svolto nel modo seguente
- $a \in FIRST(\alpha)$ se e solo se, per qualche indice $k \in 1, \dots n$, risulta $a \in FIRST(X_k)$ e $\epsilon \in FIRST(X_j)$, $j = 1, \dots, k-1$ (si suppone sempre $\epsilon \in FIRST(X_0)$).
- Se $\epsilon \in FIRST(X_j)$, $j = 1, \dots, n$, allora $\epsilon \in FIRST(\alpha)$.
- Ad esempio, nel caso della seconda grammatica del lucido precedente

$$
\begin{array}{rcl}
A & \rightarrow & \mathsf{a}A \mid BC \\
B & \rightarrow & \mathsf{b}B \mid \epsilon \\
C & \rightarrow & \mathsf{c}C \mid \epsilon
\end{array}
$$

risulta: $FIRST(\mathsf{a}A) = \{\mathsf{a}\}$ e $FIRST(BC) = \{\mathsf{b}, \mathsf{c}, \epsilon\}$.

Calcolo di $FOLLOW(A)$

- Il calcolo di $FOLLOW(A)$, per un generico non terminale A, può essere svolto in questo modo.
- Se esiste la produzione $B \to \alpha A \beta$, tutti i terminali in $FIRST(\beta)$ si inseriscono in $FOLLOW(A)$.
- In particolare, poiché (almeno implicitamente) esiste sempre la produzione $\mathcal{S}' \to \mathcal{S}\$$, il simbolo speciale \$ sta sempre nel $FOLLOW(\mathcal{S})$.

Calcolo di $FOLLOW(A)$

- Se esiste la produzione $B \to \alpha A$, tutti i terminali che stanno in $FOLLOW(B)$ si inseriscono in $FOLLOW(A)$.
- Infatti, se esiste una derivazione $\mathcal{S} \Rightarrow^* \beta B \gamma$, allora usando la produzione $B \to \alpha A$ abbiamo anche:

$$\mathcal{S} \Rightarrow^* \beta B \gamma \Rightarrow \beta \alpha A \gamma$$

- Dunque ciò che segue B in una forma di frase (cioè il $FIRST(\gamma)$) può anche seguire A.
- Si arriva alla stessa conclusione anche nel caso in cui $B \to \alpha A \delta$ e $\epsilon \in FIRST(\delta)$.
- Infatti:

$$\mathcal{S} \Rightarrow^* \beta B \gamma \Rightarrow \beta \alpha A \delta \gamma \Rightarrow^* \beta \alpha A \gamma$$

Esempio

- Consideriamo ancora la grammatica

$$\begin{array}{rcl} E & \rightarrow & (E)\,E' \mid \textbf{number}\,E' \\ E' & \rightarrow & +\,E\,E' \mid \times\,E\,E' \mid \epsilon \end{array}$$

- Possiamo subito stabilire che $FOLLOW(E)$ include il simbolo \$ e il simbolo); inoltre contiene i simboli in $FIRST(E')$ (eccetto ϵ) e cioè $+$ e $\times$.
- $FOLLOW(E')$ include $FOLLOW(E)$, a causa (ad esempio) della produzione $E \rightarrow$ **number**E'.
- La produzione $E' \rightarrow +EE'$, unitamente a $E' \rightarrow \epsilon$, stabilisce che vale anche il contrario, e cioè che $FOLLOW(E)$ include $FOLLOW(E')$.
- Mettendo tutto insieme si ottiene $FOLLOW(E) = FOLLOW(E') = \{\$, \,),+,\times\}$.

Esempio

- Per la grammatica

$$\begin{array}{lcl} A & \rightarrow & \mathtt{a}A \mid BC \mid \epsilon \\ B & \rightarrow & \mathtt{b}B \mid \epsilon \\ C & \rightarrow & \mathtt{c}C \mid \epsilon \end{array}$$

risulta $FOLLOW(A) = FOLLOW(C) = \{\$\}$ e $FOLLOW(B) = \{\mathtt{c}, \$\}$.

Esempio

- Per la grammatica con precedenza di operatori:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T \times F \mid F \\
F & \rightarrow & \textbf{number} \mid (E)
\end{array}
$$

risulta:

- $FOLLOW(E) = \{\$, +,)\}$;
- $FOLLOW(T) = FOLLOW(E) \cup \{\times\}$;
- $FOLLOW(F) = FOLLOW(T)$.

Grammatiche $LL(1)$ (continua)

- Possiamo ora rivedere in modo più compatto la nozione di grammatica $LL(1)$.
- Una grammatica è $LL(1)$ se, qualora esistano due produzioni $A \rightarrow \alpha$ e $A \rightarrow \beta$, risulta:

$$FIRST(\alpha) \cap FIRST(\beta) = \{\},$$

- Inoltre, se $\alpha \stackrel{*}{\Rightarrow} \epsilon$, deve valere $FOLLOW(A) \cap FIRST(\beta) = \{\}$.
- Analogamente, se $\beta \stackrel{*}{\Rightarrow} \epsilon$, deve valere $FOLLOW(A) \cap FIRST(\alpha) = \{\}$.

Esempio

- Si consideri la grammatica

$$\begin{array}{rcl} A & \rightarrow & B\,E \\ B & \rightarrow & C \mid D \\ C & \rightarrow & \epsilon \mid \mathtt{cc} \\ D & \rightarrow & \epsilon \mid \mathtt{dd} \\ E & \rightarrow & \mathtt{c} \mid \mathtt{d} \end{array}$$

- Poiché risulta $FIRST(C) \cap FIRST(D) = \{\epsilon\}$, la grammatica non è $LL(1)$.
- Infatti, supponiamo che la stringa in input inizi con c. Dopo la riscrittura dell'assioma il parser verrebbe a trovarsi con la forma di frase BE e il carattere c in input.
- A questo punto potrebbe essere corretto derivare tanto CE (se l'input fosse, ad esempio, ccd) quanto DE (se l'input fosse c).

Esempio

- Si modifichi la precedente grammatica nel modo seguente

$$\begin{array}{rcl} A & \rightarrow & B\,E \\ B & \rightarrow & C \mid D \\ C & \rightarrow & \epsilon \mid \mathtt{cc} \\ D & \rightarrow & \mathtt{dd} \\ E & \rightarrow & \mathtt{c} \mid \mathtt{d} \end{array}$$

  (cancellando cioè la produzione $D \rightarrow \epsilon$).
- Poiché risulta $FIRST(D) \cap FOLLOW(B) = \{\mathtt{d}\}$, la grammatica non è $LL(1)$.
- Il problema si verifica con input che inizia con d, perché potrebbe essere corretto (dopo la derivazione iniziale) derivare tanto CE (se l'input fosse d) quanto DE (se l'input fosse, ad esempio ddc).

Esempio

- Consideriamo ancora la grammatica con precedenza di operatori:

$$\begin{array}{rcl} E & \rightarrow & E+T \mid T \\ T & \rightarrow & T \times F \mid F \\ F & \rightarrow & \textbf{number} \mid (E) \end{array}$$

- Sappiamo già che tale grammatica non è $LL(1)$ perché contiene ricorsioni a sinistra.
- Possiamo anche verificare, ad esempio, che $FIRST(E) \cap FIRST(T) \supseteq \{\textbf{number}\}$.
- Eliminando la left-recursion si può però ottenere una grammatica equivalente che è $LL(1)$

Esempio (continua)

- Per la grammatica modificata

$$\begin{array}{rcl} E & \to & TE' \\ E' & \to & +TE' \mid \epsilon \\ T & \to & FT' \\ T' & \to & \times FT' \mid \epsilon \\ F & \to & \textbf{number} \mid (E) \end{array}$$

  dobbiamo solo verificare che $FOLLOW(E')$ non contenga il simbolo $+$ e che $FOLLOW(T')$ non contenga il simbolo $\times$.

- È facile verificare che risulta:
  - $FOLLOW(E) = FOLLOW(E') = \{\$,)\}$;
  - $FIRST(E') = \{+, \epsilon\}$;
  - $FOLLOW(T) = (FIRST(E') \setminus \{\epsilon\}) \cup FOLLOW(E') = \{+, \$,)\}$;
  - $FOLLOW(T') = FOLLOW(T) = \{+, \$,)\}$;

Tabella di parsing $LL(1)$

- Per una grammatica $LL(1)$ è possibile costruire (utilizzando le funzioni $FIRST$ e $FOLLOW$) una tabella, detta *tabella di parsing*, che, per ogni non terminale A e ogni terminale x, prescrive il comportamento del parser.
- Indicheremo con M_G la tabella di parsing relativa alla grammatica G (o semplicemente con M, se la grammatica è evidente).
- La generica entry $M_G[A, x]$ della tabella può contenere una produzione $A \rightarrow \alpha$ di G, oppure essere vuota, ad indicare che si è verificato un errore.
- Disponendo di M_G, la prima riga dell'algoritmo a discesa ricorsiva (cioè la scelta della produzione) viene sostituita da un lookup alla tabella $M_G[A, x]$.

Costruzione della tabella di parsing

- L'algoritmo di costruzione della parsing table è molto semplice ed è formato da un ciclo principale nel quale si prendono in considerazione tutte le produzioni.
- Per ogni produzione $A \rightarrow \alpha$:
  - per ogni simbolo x in $FIRST(\alpha)$ si pone $M[A,x]$='$A \rightarrow \alpha$';
  - se $\epsilon \in FIRST(\alpha)$, per ogni simbolo y in $FOLLOW(A)$ si pone $M[A,y]$='$A \rightarrow \alpha$'.
- Tutte le altre entry della tabella vengono lasciate vuote (ad indicare l'occorrenza di un errore).
- Se la grammatica è $LL(1)$, nessuna entry della tabella viene riempita con più di una produzione.

Esempio

- Consideriamo ancora la grammatica

$$\begin{array}{rcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & \times FT' \mid \epsilon \\ F & \rightarrow & \textbf{number} \mid (E) \end{array}$$

- Il calcolo completo degli insiemi $FIRST$ e $FOLLOW$ produce:
  - $FIRST(F) = FIRST(T) = FIRST(E) = \{\textbf{number}, (\,\}$;
  - $FIRST(E') = \{+, \epsilon\}$, $FIRST(T') = \{\times, \epsilon\}$;
  - $FOLLOW(E) = FOLLOW(E') = \{\$, \,)\}$;
  - $FOLLOW(T) = (FIRST(E') \setminus \{\epsilon\}) \cup FOLLOW(E') = \{+, \$, \,)\}$;
  - $FOLLOW(T') = FOLLOW(T) = \{+, \$, \,)\}$;
  - $FOLLOW(F) = (FIRST(T') \setminus \{\epsilon\}) \cup FOLLOW(T') = \{\times, +, \$, \,)\}$.

Esempio (continua)

- L'algoritmo prima delineato produce quindi la seguente tabella di parsing

N.T.	Simbolo di input					
	number	(	)	$+$	$\times$	**$**
E	$E \rightarrow TE'$	$E \rightarrow TE'$				
E'			$E' \rightarrow \epsilon$	$E' \rightarrow +TE'$		$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	$T \rightarrow FT'$				
T'			$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$	$T' \rightarrow \times FT'$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ **number**	$F \rightarrow (E)$				

in cui le entry vuote corrispondono ad una situazione di errore.

Esempio

- Se tentassimo di produrre la tabella di parsing per la grammatica

$$\begin{array}{rcl} E & \rightarrow & (E)\,E' \mid \textbf{number}\,E' \\ E' & \rightarrow & +\,E\,E' \mid \times\,E\,E' \mid \epsilon \end{array}$$

otterremmo

N.T.	Simbolo di input					
	number	(	)	$+$	$\times$	$
E	$E \rightarrow$ **number** E'	$E \rightarrow (E)\,E'$				
E'			$E' \rightarrow \epsilon$	$E' \rightarrow +\,E\,E'$ $E' \rightarrow \epsilon$	$E' \rightarrow \times\,E\,E'$ $E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$

- Con in input il carattere $+$ o il carattere $\times$, il parser non saprebbe quindi come procedere.
- Il non soddisfacimento delle proprietà $LL(1)$ è in questo caso una conseguenza dell'ambiguità della grammatica.

Esempio (continua)

- A volte è possibile risolvere il conflitto presente in una entry della tabella scegliendo opportunamente la produzione da applicare (fra quelle in conflitto).
- Naturalmente la scelta non deve compromettere la capacità di riconoscere il linguaggio generato dalla grammatica.
- Nel caso dell'esempio, si deve optare in favore delle produzioni $E' \rightarrow +EE'$ e $E' \rightarrow \times EE'$, anziché $E' \rightarrow \epsilon$ (si provi, ad esempio, a riconoscere la stringa **number** $+$ **number**).
- Si noti tuttavia che, pur avendo risolto l'ambiguità, l'interpretazione delle espressioni che deriva dall'albero di parsing non è quella "naturale" (non viene soddisfatta la precedenza naturale degli operatori).
- Al riguardo, si provi a costruire l'albero di parsing per la stringa **number** $\times$ **number** $+$ **number**.

Esempio

- Consideriamo la seguente grammatica (che esprime, usando simbologia diversa, il problema del *dangling else* che abbiamo già esaminato in precedenza):

$$\begin{array}{rcl} S & \rightarrow & \mathbf{i}\,E\,\mathbf{t}\,S\,\mathbf{e}\,S \mid \mathbf{i}\,E\,\mathbf{t}\,S \mid \mathbf{a} \\ E & \rightarrow & \mathbf{b} \end{array}$$

- Tale grammatica presenta produzioni con prefisso comune e dunque non è idonea al parsing a d.r.
- È possibile eliminare i prefissi comuni, ottenendo:

$$\begin{array}{rcl} S & \rightarrow & \mathbf{i}\,E\,\mathbf{t}\,S\,S' \mid \mathbf{a} \\ S' & \rightarrow & \mathbf{e}\,S \mid \epsilon \\ E & \rightarrow & \mathbf{b} \end{array}$$

  e risulta
  - $FIRST(S) = \{\mathbf{i}, \mathbf{a}\}$; $FIRST(S') = \{\mathbf{e}, \epsilon\}$; $FIRST(E) = \{\mathbf{b}\}$;
  - $FOLLOW(S) = FOLLOW(S') = \{\$, \mathbf{e}\}$.

Esempio (continua)

- La grammatica modificata non è ancora $LL(1)$ in quanto $FIRST(\mathbf{e}S) \cap FOLLOW(S') = \{\mathbf{e}\}$.
- Ciò si riflette in una definizione multipla per una entry della tabella di parsing:

N.T.	Simbolo di input					
	i	**t**	**e**	**a**	**b**	**$**
S	$S \rightarrow \mathbf{i}E\mathbf{t}SS'$			$S \rightarrow \mathbf{a}$		
S'			$S' \rightarrow \mathbf{e}S$ $S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
E					$E \rightarrow \mathbf{b}$	

- Tuttavia se, con in input il carattere **e**, il parser venisse "istruito" a scegliere sempre la produzione $S' \rightarrow \mathbf{e}S$, l'ambiguità si risolverebbe (e pure con l'intepretazione "naturale", che associa ogni **else** al **then** più vicino).

Implementazione non ricorsiva

- È possibile dare un'implementazione non ricorsiva di un parser predittivo (cioè che non richiede backtracking) utilizzando esplicitamente una pila.
- La pila serve per memorizzare i simboli della parte destra della produzione scelta ad ogni passo.
- Tali simboli verranno poi "confrontati" con l'input (se terminali) o ulteriormente riscritti (se non terminali).
- Il comportamento del parser è illustrato nella seguente diapositiva.

Implementazione non ricorsiva (continua)

- Inizialmente, sullo stack sono contenuti (partendo dal fondo) i simboli \$ ed $\mathcal{S}$, mentre la variabile z punta al primo carattere di input.
- Al generico passo, il parser controlla il simbolo X sulla testa dello stack;
  - se X è un non terminale e $M[X, z] =$ '$X \rightarrow X_1 \ldots X_k$', esegue una pop dallo stack (rimuove cioè X) seguita da k push dei simboli $X_k, \ldots, X_1$, nell'ordine;
  - se X è un non terminale e $M[X, z] =$ '$error$', segnala una condizione di errore;
  - se X è un terminale e $X = z$, esegue una pop e fa avanzare z;
  - se X è un terminale e $X \neq z$, segnala una condizione di errore.
- Le operazioni terminano quando $X = \$$ e la stringa è accettata se $z = \$$.

Esempio

- Consideriamo nuovamente la grammatica delle espressioni con precedenza di operatore, della quale ricordiamo la tabella di parsing:

N.T.	Simbolo di input					
	number	(	)	$+$	$\times$	\$
E	$E \rightarrow TE'$	$E \rightarrow TE'$				
E'			$E' \rightarrow \epsilon$	$E' \rightarrow +TE'$		$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	$T \rightarrow FT'$				
T'			$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$	$T' \rightarrow \times FT'$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ **number**	$F \rightarrow (E)$				

- Supponendo di avere la stringa **number** $+$ **number** in input, la seguente tabella illustra il progressivo contenuto dello stack e dell'input.

Esempio (continua)

Stack	Input
$\$E$	**number** $+$ **number**$\$$
$\$E'T$	**number** $+$ **number**$\$$
$\$E'T'F$	**number** $+$ **number**$\$$
$\$E'T'$**number**	**number** $+$ **number**$\$$
$\$E'T'$	$+$**number**$\$$
$\$E'$	$+$**number**$\$$
$\$E'T+$	$+$**number**$\$$
$\$E'T$	**number**$\$$
$\$E'T'F$	**number**$\$$
$\$E'T'$**number**	**number**$\$$
$\$E'T'$	$\$$
$\$E'$	$\$$
$\$$	$\$$

Esercizi proposti

- Si calcolini gli insiemi $FIRST$ e $FOLLOW$ per la seguente grammatica:

$$
\begin{array}{rcl}
S & \rightarrow & \mathbf{c} \mid AS \mid BS \\
A & \rightarrow & \mathbf{a}B \mid \epsilon \\
B & \rightarrow & \mathbf{b}A \mid \epsilon
\end{array}
$$

  e si costruisca la relativa tabella di parsing per un parser a discesa ricorsiva.

- Si calcolino gli insiemi $FIRST$ e $FOLLOW$ per la grammatica G_2, che descrive le espressioni regolari su $\{0, 1\}$, dopo averla modificata in modo da eliminare i prefissi comuni. Se ne costruisca quindi la tabella di parsing per un parser a discesa ricorsiva.