Linguaggi formali e compilazione

Corso di Laurea in Informatica

A.A. 2012/2013

L'input per il parser

- Nel contesto della compilazione l'input per il parser è costituito da una stringa di token (che indicheremo genericamente con θ) generata dall'analizzatore lessicale.
- Per semplicità di linguaggio ignoreremo la presenza dello scanner e supporremo che il parser legga direttamente θ dallo stream di input.
- Per individuare la "fine" della stringa di input, negli algoritmi di parsing supporremo che la stringa stessa sia terminata da uno speciale simbolo, non presente fra i token del linguaggio, ad esempio \$.

L'input per il parser

- Se $\mathcal{S}$ è l'assioma iniziale della grammatica, per tenere conto del simbolo di terminazione introduciamo "formalmente" un nuovo assioma $\mathcal{S}'$, con la sola produzione $\mathcal{S}' \rightarrow \mathcal{S}\$$.
- In questo modo, $\mathcal{S} \Rightarrow^* \theta$ se e solo se $\mathcal{S}' \Rightarrow^* \theta\$$.
- 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 categorie sintattiche (es, un comando o una espressione) in termini delle parti costituenti.

Tipi di parser (continua)

- 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 da usare al generico passo

- In entrambi i tipi di parser (top-down o bottom-up) la scelta della produzione da utilizzare (rispettivamenti, in avanti o all'indietro) viene effettuata in funzione dello *stato* interno del parser e della prossima *porzione di input* (1 o più caratteri).
- Come vedremo, lo stato interno del parser sarà tipicamente espresso dall'informazione memorizzata nella cima di una struttura dati stack.
- Il numero di caratteri di input considerati per prendere la decisione è noto invece come *lookahead*.
- Noi saremo interessati di norma ad un lookahead di un carattere.

Algoritmo generale

- 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 generale (continua)

- 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 $A \to X_1X_2 \dots X_k$ $(X_j \in \mathcal{V})$

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**

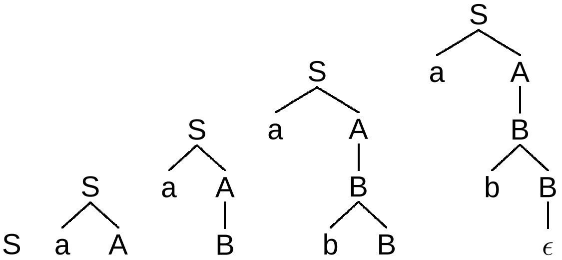
            ERROR() {Include il caso $x =$ EOF}

Esempio

- Si consideri la seguente grammatica, che genera il "solito" linguaggio $\{a^n b^m : n > 0, m \geq 0\}$:

$$
\begin{array}{rcl}
S & \rightarrow & aA \\
A & \rightarrow & aA \mid B \\
B & \rightarrow & bB \mid \epsilon
\end{array}
$$

- Su input ab un parser a d.r. nondeterministico produce la seguente derivazione:



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 per il generico non terminale A, con backtracking

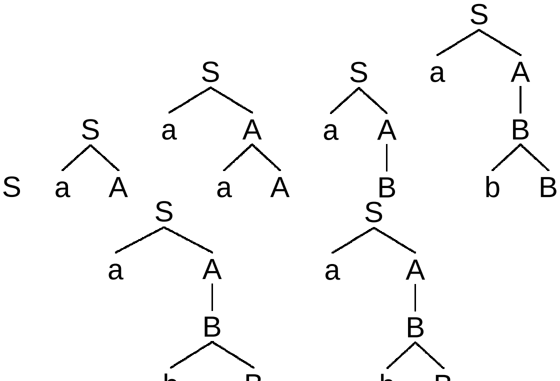
SAVEINPUTPOINTER()
for all produzione $A \to X_1X_2 \dots X_k$ $(X_j \in \mathcal{V})$ **do**
    for $j = 1, \dots, k$ **do**
        if $X_j \in \mathcal{N}$ **then**
            $X_j()$
        else
            $x \leftarrow$ READ()
            if $X_j \neq x$ **then**
                RESTOREINPUTPOINTER()
                break
{Solo nel codice per l'assioma iniziale}
if not EOF() **then**
    ERROR()

Esempio

- Riconsideriamo la grammatica

$$\begin{aligned} S &\rightarrow aA \\ A &\rightarrow aA \mid B \\ B &\rightarrow bB \mid \epsilon \end{aligned}$$

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



Parsing a discesa ricorsiva (continua)

- 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 $A \rightarrow A\alpha$ (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.

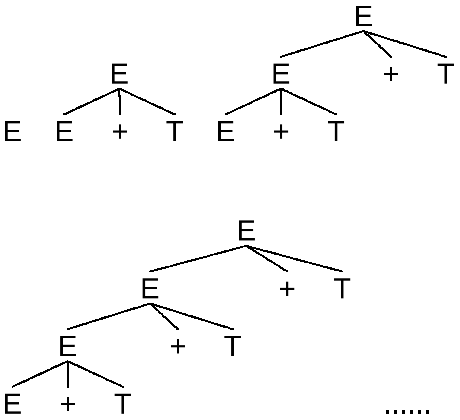
Esempio

- Per la grammatica con precedenza di operatori (che abbiamo già incontrato):

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

  le procedure per i non terminali E e T innescano potenzialmente un ciclo infinito.
- Ad esempio, su input $\mathbf{id} + \mathbf{id}$, la produzione corretta da applicare inizialmente è $E \rightarrow E + E$ (se si applica $E \rightarrow \mathbf{id}$ la derivazione fallisce e bisogna operare backtracking sull'input), ma questa innesca un ciclo infinito.

Esempio (continua)



Parsing a discesa ricorsiva (continua)

- 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:

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

- 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 $x = \mathrm{a}$, usa la produzione $A \rightarrow \mathrm{a}A$; se $x = \mathrm{b}$, usa la produzione $A \rightarrow \mathrm{b}B$.
  - Per B. se $x = \$$ (end of input), usa la produzione $B \rightarrow \epsilon$; se $x = \mathrm{b}$, usa la produzione $B \rightarrow \mathrm{b}B$;

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 più volte esaminato caso del “dangling else”.

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

- Infatti, la quantità di codice che può essere presente fra le keyword **then** e **else** non è limitabile a priori.

Grammatiche “non” $LL(k)$ (continua)

- Un altro esempio è dato dalla grammatica:

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

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

Grammatiche $LL(1)$

- Analizziamo ora in dettaglio le caratteritiche che deve avere una grammatica G per potersi qualificare $LL(1)$.
- Consideriamo un generico non terminale per il quale esistano almeno due produzioni, poniamo

$$A \rightarrow \alpha \mid \beta$$

- Se vale simultaneamente che $\alpha \stackrel{*}{\Rightarrow} \mathrm{a}\alpha'$ e $\beta \stackrel{*}{\Rightarrow} \mathrm{a}\beta'$, cioè se da α e da β si possono derivare stringhe che iniziano con lo stesso simbolo non terminale, allora G non è $LL(1)$.

Esempi

- Il caso più semplice è quando α e β iniziano con lo stesso simbolo terminale.
- Ad esempio, se la grammatica contiene le produzioni $S \rightarrow aA \mid aB$ non può essere $LL(1)$.
- Tuttavia il problema può non essere di così immediata evidenza.
- Ad esempio, la grammatica

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

non è $LL(1)$ perché $B \Rightarrow C \Rightarrow aB$ e dunque la procedura per il non terminale S, su input a... non può decidere quale produzione usare ($S \rightarrow aA$ oppure $S \rightarrow B$) guardando solo il primo simbolo di input.

Grammatiche $LL(1)$ (continua)

- La situazione appena descritta non è l'unica che deve essere evitata affinché la grammatica sia $LL(1)$.
- Un secondo vincolo è che, sempre considerando una coppia di produzioni $A \rightarrow \alpha \mid \beta$, se risulta $\alpha \stackrel{*}{\Rightarrow} \epsilon$ allora non può accadere che $\beta \stackrel{*}{\Rightarrow} \epsilon$ (e viceversa).
- Infatti, in caso contrario, qualunque sia il prossimo simbolo di input entrambe le produzioni potrebbero teoricamente essere valide.
- Anche questo non basta ancora; è infatti necessario che sia verificata un'ultima condizione.
- Se $\alpha \stackrel{*}{\Rightarrow} \epsilon$ e $\beta \stackrel{*}{\Rightarrow} \mathrm{a}\beta'$, allora non deve accadere che $\mathcal{S} \stackrel{*}{\Rightarrow} \gamma_1 A \mathrm{a} \gamma_2$, dove $\beta', \gamma_1, \gamma_2 \in \mathcal{V}^*$.
- In altri termini a non deve apparire subito dopo A in alcuna forma di frase.

Esempio

- Si consideri la grammatica

$$
\begin{aligned}
S &\rightarrow A \mid B \\
A &\rightarrow aA \mid \epsilon \\
B &\rightarrow Aab
\end{aligned}
$$

- Su input la stringa “a” la procedura per il non terminale S non può decidere correttamente la produzione da usare.
- Risulta infatti:

$$S \Rightarrow A \Rightarrow aA \Rightarrow a$$

$$S \Rightarrow A \Rightarrow aA \Rightarrow a$$

e

$$S \Rightarrow B \Rightarrow Aab \Rightarrow ab$$

FIRST e FOLLOW

- Le condizioni che abbiamo appena derivato possono essere espresse in modo più compatto utilizzando due funzioni, dette *FIRST* e *FOLLOW*, che definiscono insiemi di simboli terminali.
- 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 $A \in \mathcal{N}$ si definisce $FOLLOW(A)$ l'insieme dei terminali che si possono trovare immediatamente alla destra di A in una qualche forma di frase. In altri termini, $x \in FOLLOW(A)$ 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:

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

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

Esempi

- La seguente grammatica genera lo stesso linguaggio della precedente

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

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

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

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

Calcolo di $FIRST(\alpha)$ (continua)

- Il calcolo di $FIRST(\alpha)$, dove $\alpha = X_1 \ldots X_n$ 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 $k \in 1, \ldots n-1$, risulta $a \in X_k$ e $\epsilon \in X_j$, $j = 1, \ldots, k-1$ (si suppone sempre $\epsilon \in FIRST(X_0)$).
- Se $\epsilon \in FIRST(X_j)$, $j = 1, \ldots, n$, allora $\epsilon \in FIRST(\alpha)$.
- Ad esempio, nel caso della seconda grammatica del lucido precedente

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

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

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 $B \rightarrow \alpha A \beta$, tutti i terminali in $FIRST(\beta)$ si inseriscono in $FOLLOW(A)$.
- In particolare, poiché (almeno implicitamente) esiste sempre la produzione $\mathcal{S}' \rightarrow \mathcal{S}\$$, il simbolo speciale \$ sta sempre nel $FOLLOW(\mathcal{S})$.

Calcolo di $\mathit{FOLLOW}(A)$

- Se esiste la produzione $B \rightarrow \alpha A$, tutti i terminali che stanno in $\mathit{FOLLOW}(B)$ si inseriscono in $\mathit{FOLLOW}(A)$.
- Infatti, se esite una derivazione $\mathcal{S} \Rightarrow^* \beta B \gamma$, allora l'applicazione della produzione $B \rightarrow \alpha A$ rende evidente che esiste anche la derivazione $\mathcal{S} \Rightarrow^* \beta \alpha A \gamma$.
- Dunque ciò che segue B in una forma di frase (cioè il $\mathit{FIRST}(\gamma)$) può anche seguire A.
- Si arriva alla stessa conclusione anche nel caso in cui $B \rightarrow \alpha A \beta$ e $\epsilon \in \mathit{FIRST}(\beta)$.

Esempio

- Consideriamo ancora la grammatica

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

- 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 $E \rightarrow \mathbf{id}E'$.
- 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

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

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

Esempio

- Per la grammatica con precedenza di operatori:

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

  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 $A \rightarrow \beta$, deve risultare:

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

- 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

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

- 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

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

(cancellando cioè la produzione $D \rightarrow \epsilon$).

- Poiché risulta $FIRST(D) \cap FOLLOW(B) = \{\mathrm{d}\}$, 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:

$$\begin{array}{rcl} E & \to & E + T \mid T \\ T & \to & T \times F \mid F \\ F & \to & \mathbf{id} \mid (E) \end{array}$$

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

Esempio (continua)

- Per la grammatica modificata

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

  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

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

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

Esempio (continua)

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

N.T.	Simbolo di input					
	id	(	)	$+$	$\times$	\$
E	$E \to TE'$	$E \to TE'$				
E'			$E' \to \epsilon$	$E' \to +TE'$		$E' \to \epsilon$
T	$T \to FT'$	$T \to FT'$				
T'			$T' \to \epsilon$	$T' \to \epsilon$	$T' \to \times FT'$	$T' \to \epsilon$
F	$F \to \mathbf{id}$	$F \to (E)$				

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

Esempio

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

$$\begin{aligned} E &\rightarrow (E)E' \mid \mathbf{id}\, E' \\ E' &\rightarrow +E\,E' \mid \times E\,E' \mid \epsilon \end{aligned}$$

otterremmo

N.T.	Simbolo di input					
	id	(	)	$+$	$\times$	\$
E	$E \rightarrow \mathbf{id}E'$	$E \rightarrow (E)E'$				
E'			$E' \rightarrow \epsilon$	$E' \rightarrow +EE'$ $E' \rightarrow \epsilon$	$E' \rightarrow \times EE'$ $E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$

- 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 $E' \to +EE'$ e $E' \to \times EE'$, anziché $E' \to \epsilon$ (si provi, ad esempio, a riconoscere la stringa $\mathbf{id} + \mathbf{id}$).
- 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 $\mathbf{id} \times \mathbf{id} + \mathbf{id}$.

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	$S \to \mathbf{i}E\mathbf{t}SS'$			$S \to \mathbf{a}$		
S'			$S' \to \mathbf{e}S$ $S' \to \epsilon$			$S' \to \epsilon$
E					$E \to \mathbf{b}$	

- Tuttavia se, con in input il carattere **e**, il parser venisse "istruito" a scegliere sempre la produzione $S' \to \mathbf{e}S$, 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 $M[X, z] =$ '$X \to X_1 \ldots X_k$', 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					
	id	(	)	$+$	$\times$	\$
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	$F \rightarrow \mathbf{id}$	$F \rightarrow (E)$				

- Supponendo di avere la stringa $\mathbf{id} + \mathbf{id}$ in input, la seguente tabella illustra il progressivo contenuto dello stack e dell'input.

Esempio (continua)

Stack	Input
$\$E$	$\mathbf{id} + \mathbf{id}\$$
$\$E'T$	$\mathbf{id} + \mathbf{id}\$$
$\$E'T'F$	$\mathbf{id} + \mathbf{id}\$$
$\$E'T'\mathbf{id}$	$\mathbf{id} + \mathbf{id}\$$
$\$E'T'$	$+\mathbf{id}\$$
$\$E'$	$+\mathbf{id}\$$
$\$E'T+$	$+\mathbf{id}\$$
$\$E'T$	$\mathbf{id}\$$
$\$E'T'F$	$\mathbf{id}\$$
$\$E'T'\mathbf{id}$	$\mathbf{id}\$$
$\$E'T'$	$\$$
$\$E'$	$\$$
$\$$	$\$$

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.