Corso: Fondamenti, Linguaggi e Traduttori 2

Paola Giannini

Parsing Top-Down

Derivazione left-most (o sinistra)

- Data una grammatica $G = (V, \Sigma, P, S)$, **una stringa $w \in \Sigma^*$ è nel linguaggio generato da G** ($w \in L(G)$), se $S \rightarrow^* w$.
- Il processo di applicare le produzioni per derivare stringhe è semplice, ad esempio: data la grammatica

$$
\begin{array}{ll}
0. & S \rightarrow E\ \$ \\
1. & E \rightarrow Pr\ (\ E\ ) \\
2. & E \rightarrow \mathsf{v}\ Tl \\
3. & Pr \rightarrow \mathsf{f} \\
4. & Pr \rightarrow \epsilon \\
5. & Tl \rightarrow +\ E \\
6. & Tl \rightarrow \epsilon
\end{array}
$$

deriviamo la stringa

$$\mathsf{f}\ (\ \mathsf{v}\ +\ (\ \mathsf{v}\ )\ )\ \$$$

con una derivazione left-most (o sinistra) (cioè rimpiazzando sempre il non terminale più a sinistra):

$$
\begin{aligned}
&\boxed{S} \xrightarrow{0} \boxed{E}\ \$ \xrightarrow{1} \boxed{Pr}\ (\ E\ )\ \$ \xrightarrow{3} \mathsf{f}\ (\ \boxed{E}\ )\ \$ \xrightarrow{2} \mathsf{f}\ (\ \mathsf{v}\ \boxed{Tl}\ )\ \$ \xrightarrow{5} \\
&\mathsf{f}\ (\ \mathsf{v}\ +\ \boxed{E}\ )\ \$ \xrightarrow{1} \mathsf{f}\ (\ \mathsf{v}\ +\ \boxed{Pr}\ (\ E\ )\ )\ \$ \xrightarrow{4} \mathsf{f}\ (\ \mathsf{v}\ +\ (\ \boxed{E}\ )\ )\ \$ \xrightarrow{2} \\
&\mathsf{f}\ (\ \mathsf{v}\ +\ (\ \mathsf{v}\ \boxed{Tl}\ )\ )\ \$ \xrightarrow{6} \mathsf{f}\ (\ \mathsf{v}\ +\ (\ \mathsf{v}\ )\ )\ \$
\end{aligned}
$$

Parsing e Parsing Top-Down

- Il processo di applicare le produzioni per derivare stringhe è semplice, mentre rovesciare il processo, non lo è.
- Il **parsing** è il processo inverso alla derivazione, cioè, data una stringa ricostruire, se possibile, le produzioni che sono state applicate partendo dal simbolo iniziale per ottenere la stringa.

  Ad esempio data la stringa

  $$\mathsf{f}\ (\ \mathsf{v}\ +\ (\ \mathsf{v}\ )\ )\ \$$$

  e la grammatica della pagina precedente il parsing ci deve dire la sequenza delle produzioni che sono state applicate!

  Il parsing deve anche dire quando una stringa NON è derivabile da S. Ad esempio:

  $$\mathsf{f}\ (\ \mathsf{v}\ )\ +\ \mathsf{v}\ \$$$

- La tecnica di parsing top-down cerca di ricostruire la derivazione left-most (sinistra) e fallisce se la stringa NON fa parte del linguaggio.

  Questa tecnica è nota con diversi nomi:
  - **Parsing top-down**, perchè il parser comincia dal simbolo iniziale della grammatica e fa crescere l'albero di parsing dalla radice alle foglie
  - **Predittivo**, perchè il parser deve predire qual'è la prossima derivazione che deve essere applicata.
  - **LL(k)**, perchè l'input è scandito **da sinistra a destra** (il primo L), produce la **derivazione left-most** (il secondo L) e usa k simboli di "lookahead" (noi ci limiteremo a LL(1))

Implementazioni del Parsing top-down

- Parsing a Discesa Ricorsiva (quello che implementeremo noi). Il programma viene scritto a mano a partire dalla grammatica. Le funzioni/metodi che definiscono il parsing sono mutuamente ricorsive.
- Parsing LL basato su tabelle (top della pila e simbolo in input). Questo è simile all'implementazione del riconoscimento dei linguaggi regolari attraverso l'automa, ma in questo caso si ha un programma che simula un automa a pila. Il suo funzionamento è diverso rispetto all'automa shift-reduce che avete visto per il parsing LR.
- Entrambe le implementazioni partono dalla tabella PREDICT che ci deve dire per ogni produzione quali simboli dell'input ne predicono l'uso. Questa tabella si basa sul calcolo del FIRST e del FOLLOW che avete già visto (ma ora lo rivediamo). Fare questa tabella ci permetterà anche di dire se la grammatica e' LL(1) o non lo è.

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

Il PREDICT

- Il PREDICT di una produzione P ci dice quale è insieme di simboli terminali che sono prodotti all'inizio di una stringa generata dalla produzione P;
- Abbiamo visto che per ogni produzione P c'è un FIRST che è un insieme di simboli terminali che sono prodotti all'inizio di una stringa generata dalla produzione P;
- Ma come facciamo a sapere l' insieme di simboli terminali che sono prodotti all'inizio di una stringa generata una produzione che genera ϵ, ad esempio $A \rightarrow \epsilon$, ma anche $A \rightarrow B\ C$ e sia B che C derivano ϵ?

  per questo useremo il FOLLOW!

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

Algoritmo PREDICT

- Dobbiamo completare la tabella della grammatica aggiungendo per ogni produzione il suo insieme `Predict` che è calcolato usando il `First` e `Follow` delle produzioni.

$$
\begin{aligned}
&\texttt{Predict}(p : A \to X_1 \cdots X_n)\\
&\quad \texttt{ris} = \texttt{First}(X_1 \cdots X_n)\\
&\quad \texttt{if DerEmpty(p) then ris} = \texttt{ris} \cup \texttt{Follow}(A)\\
&\quad \texttt{return ris}
\end{aligned}
$$

- Abbiamo calcolato `First` e `Follow`, ora calcoliamo il `Predict` per la grammatica di pag. 3

Parsing LL(1)

Supponiamo di avere un insieme di simboli di tokens Tokens (questi sono i nostri simboli terminali).
La stringa di input di cui vogliamo fare il parsing è:

$$w\ a\ w'$$

dove $w, w' \in \texttt{Tokens}^*$ e $a \in \texttt{Tokens}$ e che il parser ha costruito (fino ad ora) la derivazione left-most

$$S \to^* w\ A\ X_1 \cdots X_n$$

Supponendo che $Q = \{p : A \to \alpha \in P \mid a \in \texttt{Predict}(p)\}$
Si possono verificare i seguenti casi:

- Q **è vuoto**, per cui nessuna produzione per A può generare il token a. Questo è un **errore sintattico**, (le produzioni per A potrebbero aiutare a capire che tipo di errore!)
- Q **contiene più di una produzione**. In questo caso il **parsing è non deterministico**. Questo produrrebbe un parsing inefficiente (che necessita di backtracking). La grammatica deve essere resa deterministica.
- Q **contiene esattamente una produzione**. In questo caso si procede applicandola.

Ad esempio rispetto al nostro esempio potremo avere

$$\texttt{f (v + (v)) \$} \quad \text{e} \quad S \to^* \texttt{f (v } Tl \texttt{) \$}$$

in questo caso $Q = \{Tl \to + E\}$, mentre

$$\texttt{f (v + (v)) \$} \quad \text{e} \quad S \to^* \texttt{f (v + Pr (} E \texttt{)) \$}$$

$Q = \{\text{Pr} \to \epsilon\}$.

Definizione di Grammatica LL(1) e Linguaggio LL(1)

- Una grammatica è LL(1), se per ogni simbolo non-terminale A, un token predice al piú una produzione.
- Cioè una volta fatta la tabella `Predict` per ogni simbolo non terminale A
  - se le produzioni, p_1, ...,p_n associate ad A e
  - $Pred_1$, ...,$Pred_n$ sono gli insiemi predict associati a p_1, ...,p_n,
  - allora $Pred_i \cap Pred_j = \emptyset$ per tutti $1 \leq i \neq j \leq n$
- Un linguaggio è LL(1), se ha una grammatica LL(1) che lo genera.

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

Esempi

- La grammatica di pag. 3 è LL(1)?
- La seguente grammatica

0. $S \rightarrow E\ \$$
1. $E \rightarrow T\ E'$
2. $E' \rightarrow -\ E$
3. $E' \rightarrow \epsilon$
4. $T \rightarrow F\ T'$
5. $T' \rightarrow /\ T$
6. $T' \rightarrow \epsilon$
7. $F \rightarrow \texttt{int}$
8. $F \rightarrow (\ E\ )$

è LL(1)?

Parsing a discesa ricorsiva

L'input del parsing è la sequenza di token generata dallo scanner.

- A ogni non terminale, A, **è associata una funzione**.
- La funzione associata con A fa un passo di riduzione, scegliendo una delle produzione associate ad A.
- Il parser sceglie la produzione da applicare ispezionando i prossimi k token dell'input. Per questo viene definito l'insieme di token `Predict` per ogni produzione $p \in P$.
- I token ispezionati sono il **lookahead**.

Interfaccia fra Parser e Scanner

Assumiamo di avere implementato uno Scanner, e di aver implementato i Token nel modo più semplice cioè con un'unica classe e un tipo enumerato che ci dice cosa rappresenta il token.
Supponiamo di avere a disposizione i due metodi seguenti dallo Scanner:

1. `Token nextToken()` che modifica lo stream (cioè la successiva chiamata a `nextToken()` ritorna il token successivo, e
2. `Token peekToken()` che non consuma consuma il token ritornato (cioè la successiva chiamata a `nextToken()` o `peekToken()` ritornerà lo stesso token).

Domanda: un modo facile di implementare il metodo `Token peekToken()` a partire dal vostro metodo `Token nextToken()` nella classe `Scanner`?

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

La funzione match

La funzione/metodo `match(TokenType type)` controlla che il prossimo token dello stream abbia uno specifico tipo, nel qual caso lo consuma e lo ritorna, mentre non ha lo stesso tipo `match` produce un ERRORE.

```
Token match(TokenType type) }
    if type==peekToken().getType())
      then return nextToken()
      else ErroreSintattico
```

Implementazione Non Terminali

Per ogni non terminale A, a cui sono associate le produzioni p_1, ..., p_n, scriviamo una funzione/metodo, del tipo seguente:

```
parseA()
  Token nextTk = peekToken()
  se nextTk ∈ Predict(p1) allora      //codice per p1
          ⋮
  se nextTk ∈ Predict(pn) allora      //codice per pn
  altrimenti ErroreSintattico
```

supponiamo che $p : A \to X_1 \cdots X_n$ il codice per p è la sequenza dei codici per X_i dove:

- se X_i é il non terminale B allora chiamiamo parseB(),
- se X_i é un token (cioè un terminale) allora chiamiamo match(token.getType()).

Esempio

- Il parsing inizierà con la chiamata della funzione associata con il simbolo iniziale, parseS.
- Vogliamo che alla fine sia stato consumato tutto l'input, cioè si trovi il token $ che denota la fine del programma.
- Per il momento consideriamo che il parsing riconosca le stringhe del linguaggio generato, cioè ci basta non produrre un ErroreSintattico chiamando la funzione parseS. Possiamo considerare Alternativamente fate ritornare un booleano che sia true se avete fatto il parsing corretto!

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

Funzioni parseS e parseE

```
//FUN e' il tipo del token per "f", PARA per "(" VAL per "v" PARC per ")" PLUS per "+"

parseS(){
  Token token=peekToken()
  switch (token.getType()) {
   case TokenType.FUN:
   case TokenType.PARA:
   case TokenType.VAL: // produzione  S -> E $
      parseE()
      match(TokenType.EOF)   // EOF e' il tipo del token per "$"
   }
   ErroreSintattico
}

parseE(){
   Token token=peekToken()
   switch (token.getType()) {
   case TokenType.FUN:
   case TokenType.PARA: //  produzione  E -> Pr ( E )
      parsePr()
      match(TokenType.PARA)
      parseE()
      match(TokenType.PARC) //
      return
   case TokenType.VAL:  //  produzione  E -> v Tl
      match(TokenType.VAL)
      parseTl()
      return
   }
   ErroreSintattico
}
```

Funzioni parsePr e parseTl

```
parsePr(){
   Token token=peekToken()
   switch (token.getType()) {
   case TokenType.FUN:  //  produzione  Pr -> f
      match(TokenType.FUN)
      return
   case TokenType.PARA:  //  produzione  Pr -> eps
      return
   }
   ErroreSintattico
}

parseTl(){
   Token token=peekToken()
   switch (token.getType()) {
   case TokenType.PLUS: //  produzione  Tl -> + E
      match(TokenType.PLUS)
      parseE()
      return
   case TokenType.PARC:
   case TokenType.EOF:   //  produzione  Tl -> eps
      return
   }
   ErroreSintattico
}
```

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

Parser basato su tabelle

- Come alternativa all'implementazione con insieme di funzioni ricorsive, il parser può essere implementato simulando un Automa a Pila (diverso dall'automa a pila shift-reduce usato per l'analisi LR).
- Lo **stack** contiene simboli terminali e non terminali del linguaggio,
  - **inizialmente ho** S cioè il non terminale iniziale della grammatica,
  - **alla fine** mi aspetto di essere alla fine della stringa e avere lo **stack vuoto** (se la stringa è stata riconosciuta).
- La **tabella di parsing**: array bidimensionale $M : V \times \Sigma \cup \{\$\} \rightarrow P$, che associa ad un simbolo non terminale A e un simbolo terminale a (che potrebbe anche essere \$), una produzione $p \in P$.

Le azioni del parser

Se X è il simbolo top della pila e a il simbolo di ingresso, ci sono 4 mosse possibili:

1. $X = \$$ e $a = \$$, $\Rightarrow$ il parsing termina con **successo**
2. $X = a$ (cioè il top della pila è uguale al simbolo letto), $\Rightarrow$ fare **pop** della pila e **nextToken()** (avanzare di un token l'input)
3. $X \in V$ (X è un simbolo non terminale) $\Rightarrow$ se $M(X, a) = p$ e $p : X \rightarrow X_1 \cdots X_n$ si eseguono le mosse seguenti
   1. **pop** della pila (cioè rimuovere X)
   2. **push** di $X_n \cdots X_1$ sulla pila
   3. output la produzione $p : X \rightarrow X_1 \cdots X_n$ che indica che abbiamo usato questa produzione
4. nessuno dei casi precedenti $\Rightarrow$ **ErroreSintattico** (in particolare se $M(X, a)$ è una casella vuota!)

Esercizio

Costruiamo la tabella per il nostro esempio e poi simuliamo l'esecuzione del parser sulla stringa

$$\mathtt{f}\ (\mathtt{v} + \mathtt{v})\ \$$$

Per costruire la tabella usiamo `Predict` delle produzioni:

Produzione	0. S	1. E	2. E	3. Pr	4. Pr	5. Tl	6. Tl
Predict	{ f ,v ,(}	{ f ,(}	{ v }	{ f }	{ (}	{ + }	{),$ }

Tabella

	f	v	(	)	+	$
S	0.	0.	0.			
E	1.	2.	1.			
Pr	3.		4.			
Tl				6.	5.	6.

Parsing con Automa PushDown

0. $S \rightarrow E\ \$$
1. $E \rightarrow Pr\ (\ E\ )$
2. $E \rightarrow \mathsf{V}\ Tl$
3. $Pr \rightarrow \mathsf{f}$
4. $Pr \rightarrow \epsilon$
5. $Tl \rightarrow +\ E$
6. $Tl \rightarrow \epsilon$

	f	V	(	)	+	$
S	0.	0.	0.			
E	1.	2.	1.			
Pr	3.		4.			
Tl				6.	5.	6.

Stringa di input

`f(v+v) $`

Input	Pila
f(v+v) $	S
f(v+v) $	E $
f(v+v) $	Pr (E) $
f(v+v) $	f (E) $
(v+v) $	(v Tl) $
v+v) $	v Tl) $
+v) $	Tl) $
+v) $	+ E) $
v) $	E) $
v) $	v Tl) $
) $	Tl) $
) $	) $
$	$

Grammatiche NON LL(1)

- Una grammatica **ricorsiva sinistra** non può essere LL(1). Ad esempio $E \to E \ + \ T \mid E \ - \ T \mid \mathsf{v}$

  Perché? Assumi $A \to A\ \alpha \mid \beta$

  - se $a \in \texttt{First}(\beta)$ allora $a \in \texttt{First}(A\ \alpha)$, perchè $A \Rightarrow A\ \alpha \Rightarrow \beta\ \alpha$
  - se $\beta = \epsilon$ allora $a \in \texttt{Follow}(A)$ se $a \in \texttt{First}(\alpha)$ e quindi anche $a \in \texttt{First}(A\ \alpha)$, perchè $A \Rightarrow A\ \alpha \Rightarrow \alpha$

- Una grammatica con **prefissi comuni**, cioè 2 o più produzioni per lo stesso non terminale hanno la stessa parte iniziale, non può essere LL(1).
  Ad esempio $S \to \texttt{if}\ E\ \texttt{then}\ E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E$.

  Perché?

  - se $A \to \alpha\ \beta_1 \mid \alpha\ \beta_2$, se $a \in \texttt{First}(\alpha\ \beta_1)$ allora $a \in \texttt{First}(\alpha\ \beta_2)$, e viceversa.

- Ma il linguaggio generato può essere LL(1), se troviamo un'altra grammatica LL(1) che lo genera!

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

Trasformazioni per grammatiche

- Rimozione della ricorsione sinistra (lo avete visto per la trasformazioni in forma normale di Greiback!)
- Fattorizzazione

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

Rimozione ricorsione sinistra

Le produzioni

$$A \to A\ \beta_1 \mid A\ \beta_2 \mid \cdots \mid A\ \beta_n \qquad n > 0$$
$$A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_m \qquad m > 0$$

vengono rimpiazzate da

$$A \to \alpha_1\ A' \mid \alpha_2\ A' \mid \cdots \mid \alpha_m\ A' \qquad m > 0$$
$$A' \to \beta_1\ A' \mid \beta_2\ A' \mid \cdots \mid \beta_n\ A' \mid \epsilon \qquad n > 0$$

dove A' è un **nuovo non-terminale**.

Fattorizzazione

Per ogni non terminale A consideriamo le produzioni

$A \to \alpha\ \beta_1 \mid \alpha\ \beta_2 \mid \cdots \mid \alpha\ \beta_n \quad n > 1$ dove α è il prefisso comune più lungo

Rimpiazziamo queste produzioni con

$$A \to \alpha\ A'$$
$$A' \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

dove A' è un **nuovo non-terminale**.

Esempio

Definire una grammatica equivalente che sia LL(1).

$$
\begin{array}{lll}
St & \rightarrow & \texttt{if}\ \ Exp\ \ \texttt{then}\ \ Sts\ \ \texttt{endif} \\
St & \rightarrow & \texttt{if}\ \ Exp\ \ \texttt{then}\ \ Sts\ \ \texttt{else}\ \ Sts\ \ \texttt{endif} \\
Sts & \rightarrow & Sts\ St\ \texttt{;} \\
Sts & \rightarrow & St\ \texttt{;} \\
Exp & \rightarrow & \texttt{var}\ \texttt{+}\ Exp \\
Exp & \rightarrow & \texttt{var}
\end{array}
$$

Trasformazioni

Fattorizzazione (delle produzioni per *St* e *Exp*)

$$
\begin{array}{lll}
St & \rightarrow & \texttt{if}\ Exp\ \texttt{then}\ Sts\ St' \\
St' & \rightarrow & \texttt{endif} \\
St' & \rightarrow & \texttt{else}\ Sts\ \texttt{endif} \\
Sts & \rightarrow & Sts\ St\texttt{;} \\
Sts & \rightarrow & St\texttt{;} \\
Exp & \rightarrow & \texttt{var}\ Exp' \\
Exp' & \rightarrow & \texttt{+}\ Exp \\
Exp' & \rightarrow & \epsilon
\end{array}
$$

Rimozione ricorsione sinistra dalle produzioni per *Sts*

$$A \rightarrow A\ \beta \mid \alpha \implies A \rightarrow \alpha\ A' \quad A' \rightarrow \beta\ A' \mid \epsilon$$

$$
\begin{array}{lll}
St & \rightarrow & \texttt{if}\ Exp\ \texttt{then}\ Sts\ St' \\
St' & \rightarrow & \texttt{endif} \\
St' & \rightarrow & \texttt{else}\ Sts\ \texttt{endif} \\
Sts & \rightarrow & St\texttt{;}\ Sts' \\
Sts' & \rightarrow & St\texttt{;}\ Sts' \\
Sts' & \rightarrow & \epsilon \\
Exp & \rightarrow & \texttt{var}\ Exp' \\
Exp' & \rightarrow & \texttt{+}\ Exp \\
Exp' & \rightarrow & \epsilon
\end{array}
$$

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

Segnalare gli errori

Nel parsing top-down è facile segnalare gli errori sintattici.
Consideriamo il parser a discesa ricorsiva, falliamo quando:

- il `match` non trova il token del tipo giusto nell'input
- il token dell'input non è generato da una produzione del non-terminale che ci aspettavamo

In entrambi i casi possiamo segnalare il token se cui si è manifestato l'errore. Per questo nei token dovrebbe essere memorizzata la riga del programma sorgente nella quale è il token.

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

Recupero dagli errori

- Uno dei meccanismi usati è il panic mode, cioè, trovato l'errore il parser scorre i token fino a trovare un delimitatore frequente, ad esempio ; e chiama parseA() dove A è un non terminale che che deriva una stringa che segue il delimitatore. Ad esempio nel linguaggio **ac** ricomincerebbe a fare il parsing da un non terminale da cui si derivano statements.
- Un altro meccanismo si basa sul raccogliere durante il parsing quei token che seguono la chiamata del parsing di un certo simbolo non terminale e cercare di completare la chiamata corrente scorrendo i token fino a trovarne uno in quell'insieme.
- Io non vi chiederò di fare il recupero dagli errori, ma sei volete potete farlo.

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE