Progetto di Linguaggi di programmazione 2014-15
Seconda parte
Grammatica LL(1) per LispKit

G.Filè

10 novembre 2014

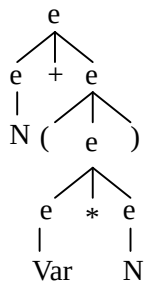

Figura 1: Esempio di un albero di parsing

1 Introduzione

La nozione di **grammatica libera da contesto** (GLC) viene data per conosciuta e quindi rinfrescheremo solo la relativa terminologia con qualche esempio. Una GLC che specifica la sintassi di espressioni aritmetiche è la seguente che chiameremo G_e:

```
e::= e + e | e * e | (e) | N | Var
```

G_e ha un solo **nonterminale** (e) mentre gli altri simboli sono **terminali**. Il simbolo | serve a separare le diverse **produzioni** di G_e. Partendo dal nonterminale iniziale della grammatica (in questo caso anche l'unico nonterminale) e applicando le produzioni, si derivano le strighe del **linguaggio** generato da G_e. Una **derivazione** è la seguente:

```
e -> e + e -> N + e -> N + (e) -> N + (e * e) ->
-> N + (Var * e) -> N + (Var * N)
```

I terminali N e Var indicano, rispettivamente, un qualsiasi numero ed una qualsiasi variabile. Sarebbe facile trasformarli in nonterminali che con opportune produzioni possano generare, rispettivamente, numeri e stringhe che inizino con una lettera. Questo non è necessario in quanto la grammatica G_e così com'è è sufficiente per spiegare quello che ci interessa in questa Sezione. La stringa finale N + (Var * N) della derivazione appena mostrata è composta da soli terminali, mentre le stringhe intermedie, come per esempio N + (e * e), contengono sia terminali che nonterminali e sono dette **forme sentenziali**. Ogni derivazione è rappresentata da un **albero di derivazione o di parsing**. La Figura 1 mostra l'albero di parsing della derivazione precedente. La stringa terminale che risiede sulla foglie di questo albero (da sinistra a destra), cioè N + (Var * N), è detta la sua **frontiera**.

Nel seguito avremo bisogno di una GLC che generi espressioni, come G_e, da inserire nella grammatica che genera i programmi LispKit. G_e è troppo semplice per questo scopo perché è **ambigua**, cioè ammette diversi alberi di derivazione con uguale frontiera. Un esempio di ambiguità è la stringa N + N * N che è frontiera dei due alberi di parsing di Figura 2.

Intuitivamente l'albero di parsing a sinistra nella Figura 2 corrisponde al seguente ordine di valutazione dell'espressione, N + (N * N), mentre quello a destra

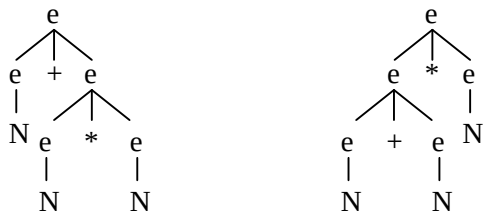

Figura 2: Due alberi di parsing con uguale frontiera: N + N * N

corrisponde all'ordine (N + N) * N. Visto che la moltiplicazione nei linguaggi di programmazione ha precedenza rispetto alla somma, l'ordine N + (N * N) è quello coerente con la precedenza. Questa osservazione è importante perché per eliminare l'ambiguità da G_e dovremo trasformare G_e in una grammatica capace di definire uno solo dei 2 alberi di parsing di Figura 2 e sceglieremo di mantenere l'albero a sinistra della Figura, cioè quello che è coerente con la precedenza degli operatori. La nuova grammatica per generare espressioni che risolve il problema dell'ambiguità è la seguente G_{ok}:

```
e::= e + t | e - t | t
t::= t * v | t / v | v
v::= N | Var | (e)
```

Si osservi che in G_{ok} sono stati aggiunti gli operatori - e / per renderla maggiormente espressiva. Per capire che G_{ok} non è ambigua, osserviamo come essa genera N + N * N. La sola derivazione che esiste è:

```
e-> e + t -> t + t -> v + t -> N + t ->
-> N + t * v -> N + v * v -> N + N * N
```

che corrisponde all'albero di parsing a sinistra della Figura 2. In generale, le derivazioni di G_{ok} funzionano così: i + sono generati da destra a sinistra e dopo i nonterminali t tra un + e il successivo generano i * da destra a sinistra. Che i + sono generati prima (più vicini alla radice dell'albero) dei *, garantisce che la precedenza sia rispettata. Inoltre, il fatto che i + (e anche i *) siano generati da destra a sinistra, è coerente con il fatto che + (e anche *) associa a sinistra. Per capire meglio questa osservazione, consideriamo l'espressione v + v - v[1]. L'associatività a sinistra richiede che l'espressione sia valutata in questo ordine, (v + v) - v, ed infatti, l'unico albero di parsing di G_{ok} per questa espressione è quello di Figura 3 che corrisponde a quest'ordine di valutazione.

Nel seguito vedremo che G_{ok} non è LL(1) e quindi la dovremo trasformare per renderla LL(1).

[1]Per semplicità ignoriamo i passi che riscrivono i nonterminali v.

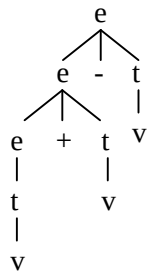

Figura 3: L'unico albero di parsing di G_{ok} per v + v - v

2 Grammatiche LL(1)

L'analisi sintattica o parsing di una GLC G consiste in una programma che riceve in input una stringa s e che cerca di costruire un albero di parsing di G per la stringa s. Se l'albero di parsing esiste, allora s appartiene al linguaggio generato da G, altrimenti non lo è. Nel contesto del parsing compreso in un **compilatore**, la stringa s è un programma in qualche linguaggio di programmazione (per esempio in LispKit) e G è la GLC che descrive la sintassi del linguaggio. Se il parser costruisce un albero di parsing per il programma s, allora s è sintatticamente corretto.

Esiste un parser per ogni GLC (anche ambigua), ma nel caso dei linguaggi di programmazione si desidera che il parsing sia efficiente. A questo fine sono state definite varie sottoclassi di GLC per le quali il parsing sia semplice. La classe più semplice tra queste è la classe LL(1) per la quale è possibile fare il parsing percorrendo il programma s da analizzare da sinistra a destra una sola volta ed esaminando un simbolo del programma alla volta. Intuitivamente basta esaminare un solo simbolo alla volta del programma in input per sapere come continuare il parsing con la certezza di non dover mai tornare indietro per seguire una strada diversa da quella scelta in precedenza.

Il test per decidere se una data GLC è LL(1) o no, consiste nella costruzione di una tabella, detta **tabella di parsing**. Se la tabella di parsing ha una forma particolarmente semplice, allora la grammatica è LL(1), altrimenti non lo è. Per costruire la tabella di parsing per una GLC G è necessario costruire 2 funzioni che si chiamano FIRST e FOLLOW per G. I tipi di FIRST e FOLLOW per G sono come segue: sia GN, GT e GFS, rispettivamente, l'insieme dei nonterminali, dei terminali e delle forme sentenziali di G, sia inoltre POWER(S) l'insieme di tutti i sottoinsiemi di S, allora FIRST: GFS $\rightarrow$ POWER(GT$\cup\{\epsilon\}$), FOLLOW: GN $\rightarrow$ POWER(GT). Come i nomi indicano intuitivamente, FIRST(w) per una forma sentenziale w, è l'insieme di tutti i primi simboli terminali delle stringhe di terminali che sono derivabili da w. FIRST(w) può contenere ϵ qualora w derivi ϵ. FOLLOW(A) con A nonterminale è l'insieme di tutti i terminali b tali che in qualche forma sentenziale generabile da G compaia A seguito immediatamente da b. Nel seguito descriveremo gli algoritmi per calcolare queste 2 funzioni, iniziando da FIRST.

Definizione 1 *Sia G una GLC e siano GP, GN, GT, GFS gli insiemi, rispettivamente, delle produzioni, dei nonterminali, dei terminali e delle forme sentenziali di G. La funzione FIRST:GN→ POWER(GT∪{ϵ}) viene calcolata in modo iterativo partendo da FIRST$_0$ che assegna $\emptyset$ ad ogni nonterminale e poi calcolando FIRST$_1$, FIRST$_2$, ... come descritto nel seguito. L'iterazione termina non appena FIRST$_i$=FIRST$_{i-1}$. Il resto della definizione spiega come calcolare FIRST$_i$ da FIRST$_{i-1}$:*

- *inizialmente FIRST$_i$ = FIRST$_{i-1}$*
- *si consideri ogni nonterminale X di G:*
- *si consideri ogni produzione X::=w $\in$ GP,*
- *sia w=$A_1 \dots A_n$, con $n \geq 0$ e dove ciascun A_i può essere o un terminale o un nonterminale. Calcoliamo First($A_1 \dots A_n$) come segue e lo aggiungiamo a FIRST$_i$(X):*

  1. *se $n = 0$ allora w=ϵ e quindi First(ϵ)=$\{\epsilon\}$ e si salta direttamente al punto (6);*
  2. *Altrimenti, se $n > 0$, sia inizialmente First($A_1 \dots A_n$)=$\emptyset$ e $j = 1$ e si eseguano i seguenti passi:*
  3. *fino a che $j < n$ si esegua il seguente punto (4), altrimenti, se $j = n$ si vada a (5):*
  4. – *se $A_j \in$ è un terminale, allora si aggiunga A_j a First($A_1 \dots A_n$) e si termini saltando al punto (6),*
     – *se $A_j \in$ è un nonterminale, tale che FIRST$_{i-1}$(A_j)=$\emptyset$, allora si termini saltando al punto (6),*
     – *altrimenti, se FIRST$_{i-1}$(A_j)$\neq \emptyset$, si aggiunge FIRST$_{i-1}$(A_j)$\backslash\{\epsilon\}$ a First($A_1 \dots A_n$) e*
     – *se $\epsilon \notin$FIRST$_{i-1}$(A_j), allora si termina saltando al punto (6), e altrimenti si torni al punto (3) con $j = j + 1$;*
  5. *$j = n$,*
     – *se A_n è un terminale, allora si aggiunge A_n a First($A_1 \dots A_n$) e si salta a (6),*
     – *altrimenti, se A_n è un nonterminale tale che FIRST$_{i-1}$(A_n)=$\emptyset$, allora si salta a (6),*
     – *se invece A_n è un nonterminale tale che FIRST$_{i-1}$(A_n)$\neq \emptyset$, allora si aggiunge FIRST$_{i-1}$(A_n) a First($A_1 \dots A_n$) e si salta al punto (6);*
  6. *STOP: First($A_1 \dots A_n$) va aggiunto a FIRST$_i$(X).*

Mettiamo immediatamente all'opera l'algoritmo per calcolare FIRST della grammatica G$_{ok}$ della sezione precedente.

`FIRST(e)=FIRST(t)=FIRST(v)={N,Var,(}`

La conoscenza di FIRST è essenziale per calcolare FOLLOW. L'algoritmo per calcolarlo è contenuto nella prossima Definizione. Nel seguito assumeremo che ogni stringa di un linguaggio termina con una sentinella, cioè un simbolo terminale usato solo per questo scopo. Come sentinella useremo il simbolo $. Per tenere conto di questo fatto, nella prossima definizione, metteremo immediatamente il $ nel FOLLOW del nonterminale iniziale. L'ipotesi di avere la sentinella sarà utile per realizzare l'analizzatore sintattico che analizza queste stringhe.

Definizione 2 *Nel seguito G è una GLC e GP e GN hanno il consueto significato. In aggiunta S è il nonterminale iniziale.*

1. *inizialmente FOLLOW$_0$(S)=$\{\$\}$, mentre per ogni X $\in$ GN$\backslash\{S\}$, FOLLOW$_0$=$\emptyset$, si fissa $i = 1$ e si ripete quanto segue fino alla terminazione:*

2. *si ponga FOLLOW$_i$=FOLLOW$_{i-1}$ e si ripeta*

   - *per ogni nonterminale X$\in$GN e per ogni produzione in GP della forma Y::=$w_1 X w_2$ si considerano i seguenti punti:*
     - *si aggiunge a FOLLOW$_i$(X), FIRST(w_2)$\backslash\{\epsilon\}$*
     - *se $\epsilon \in$ FIRST(w_2) allora si aggiunge a FOLLOW$_i$(X), FOLLOW$_{i-1}$(Y);*

3. *se FOLLOW$_i$=FOLLOW$_{i-1}$ allora l'algoritmo termina e FOLLOW=FOLLOW$_{i-1}$, altrimenti si aumenta i di 1 e si ritorna al punto precedente.*

Come esempio, calcoliamo FOLLOW per G_{ok}.

```
FOLLOW(e)={$,+,)}
FOLLOW(t)={$,+,*,)}
FOLLOW(v)={$,+,*,)}
```

Per concludere la Sezione, mostriamo come si costruisce la tabella di parsing di una GLC.

Definizione 3 *Sia G una GLC e GP, GN e GT gli insiemi delle produzioni, dei nonterminali e dei terminali di G. La tabella di parsing di G è una matrice M di dimensione $|GN| \times |GT|$ e M[X][a] contiene la produzione X::=w quando a$\in$ FIRST(w) oppure anche quando $\epsilon \in$FIRST(w) e a$\in$ FOLLOW(X).*

Vediamo com'è fatta la tabella di parsing di G_{ok} che denotiamo con M_{ok}. Visto che M_{ok} contiene delle produzioni di G_{ok}, per semplicità, abbiamo numerato queste produzioni. La grammatica consiste di 3 triple di produzioni che sono indicate quindi con 1.1, 1.2, 1.3, 2.1, 2.2, ... eccetera. Per risparmiare spazio abbiamo evitato di considerare i simboli * e / le cui colonne sarebbero comunque vuote, come quelle di + e *.

	$	(	)	Var	N	+	*
e		1.1,1.2,1.3		1.1,1.2,1.3	1.1,1.2,1.3		
t		2.1,2.2,2.3		2.1,2.2,2.3	2.1,2.2,2.3		
v		3.3		3.2	3.1		

Questa tabella di parsing mostra che G_{ok} non è LL(1). Infatti la sua tabella contiene entrate in cui figurano più di una produzione. Per esempio M_{ok}[e][(] contiene 3 produzioni. Intuitivamente, la presenza di più produzioni in un'entrata della tabella di parsing, significa che non siamo sicuri su come procedere nel parsing. Possiamo *provare* una delle produzioni, ma potremmo dover tornare indietro per provarne un'altra. Questa maniera di procedere si chiama **backtrack** e ovviamente fare backtrack costa e quindi è meglio avere grammatiche che non lo richiedono, come le grammatiche LL(1). É possibile modificare G_{ok} in modo che diventi LL(1)? La risposta è positiva e la nuova grammatica, che denoteremo $G_{LL(1)}$, segue. Si osservi che, per facilità di scrittura, si è indicata la stringa vuota ϵ con la stringa $''$`epsilon`$''$.

```
1)e::= t e'
2)e'::= + t e' | - t e' | epsilon
3)t::=f t'
4)t'::= * f t' | \ f t' | epsilon
5)f::= N | Var | (e)
```

Lasciamo il calcolo di FIRST e FOLLOW di $G_{LL(1)}$ come esercizio e mostriamo immediatamente la sua tabella di parsing che indica che questa grammatica è LL(1).

	$	(	)	Var	N	+	-	*	\
e		1.1		1.1	1.1				
e'	2.3		2.3			2.1	2.2		
t		3.1		3.1	3.1				
t'	4.3		4.3			4.3	4.3	4.1	4.2
f		5.3		5.2	5.1				

Da una tabella di parsing come questa è molto facile realizzare un parser per $G_{LL(1)}$. Il parser è costituito da un insieme di funzioni mutuamente ricorsive nel quale c'è una funzione per ogni nonterminale, cioè per ogni riga della tabella. La funzione che corrisponde ad un certo nonterminale consiste di un grande `switch` che considera tutti i possibili simboli terminali in input (le colonne della tabella) e per ognuno di essi, la tabella specifica quale produzione è applicata a quel punto e quindi quali altre funzioni devono venire invocate per continuare correttamente il parsing. Per esempio, consideriamo la precedente tabella di parsing per $G_{LL(1)}$ e consideriamo in particolare il nonterminale e' e delineiamo quello che deve fare la funzione che corrisponde ad e'. La tabella specifica che il prossimo terminale in input deve essere o il $ o la parentesi) o il + o il -. Quindi qualsiasi altro input significherebbe che la stringa analizzata non è sintatticamente corretta. In questo

caso la funzione corrispondente a e' dovrebbe sollevare un'eccezione. Se invece l'input fosse +, la tabella di parsing mostra che si deve applicare la produzione 2.1, cioè `e'::=+ t e'`, e quindi la funzione per e', in questo caso, deve eseguire i seguenti passi:

1. *consumare* il + in input;
2. invocare la funzione che corrisponde al nonterminale t passandole l'input corrente;
3. al ritorno dall'invocazione della funzione che corrisponde a t (che dovrà restituire quello che resta dell'input) deve invocare se stessa con l'input rimasto.

Vale la pena di notare che l'input, che viene consumato nella descrizione intuitiva appena data, è la sequenza di tokens che viene prodotta dall'analizzatore lessicale della prima parte del progetto.

3 Cosa c'è da fare per la parte 2 del progetto

Ora vogliamo applicare la teoria sviluppata nelle Sezioni precedenti al linguaggio LispKit. Per prima cosa ci serve una GLC che sia capace di specificare la sintassi LispKit. Proponiamo inizialmente la seguente grammatica G_{LK_1}. Si osservi che le produzioni dalla 5 alla 9 sono (praticamente) la grammatica $G_{LL(1)}$, vista prima[2].

```
1  Prog::= let  Bind in Exp end | letrec Bind in Exp end
2  Bind::= var = Exp X
3  X::= and Bind | epsilon
4  Exp ::= Prog | lambda(Seq_Var) Exp  | ExpA | OPP(Seq_Exp) |
if Exp then Exp else Exp
5  ExpA::= T E1
6  E1::= OPA T E1 | epsilon
7  T::= F T1
8  T1::= OPM F T1 | epsilon
9  F::= var Y |exp_const | (ExpA)
10 Y :: = (Seq_Exp) | epsilon
11 OPA::= + | -
12 OPM::= * | /
13 OPP::= cons | car | cdr | eq | leq | atom
14 Seq_Exp::= Exp Seq_Exp |epsilon
15 Seq_Var ::= var Seq_var | epsilon
```

Si noti che nella precedente grammatica i nonterminali iniziano sempre con una lettera maiuscola e i terminali con una minuscola (o sono caratteri di punteggiatura, operatori ecc.). La stringa vuota ϵ è scritta come $'$`epsilon`$'$. Inoltre, si noti che

[2]Si osservi che F::= var Y generalizza f::= Var di $G_{LL(1)}$ in modo che il tra i fattori F si possa rappresentare anche l'invocazione di una funzione

`var` e `exp_const` sono terminali che stanno per una qualsiasi variabile ed una qualsiasi costante (per esempio un numero, un booleano, una stringa costante, e nil). Sarebbe stato inutile aggiungere a G_{LK_1} produzioni che dettagliassero queste diverse possibilità, infatti l'analizzatore lessicale già si prende cura di rappresentare in modo distinto i diversi casi. G_{LK_1} ha 3 tipi di operatori, `OPA` e `OPM` sono quelli infissi, additivi e moltiplicativi, rispettivamente, e `OPP` sono quelli prefissi. Il significato di questi ultimi operatori è descritto nella prima parte del progetto che introduce il LispKit. Si noti anche che ogni riga della grammatica è numerata e le diverse scelte per la riga n sono numerate $n.1, n.2, \ldots$.

La seconda parte del progetto richiede di verificare se G_{LK_1} è LL(1). Anticipiamo che la risposta è negativa. Sta a voi individuare il *piccolo* errore che causa questo fatto e proporre un rimedio. Questo esercizio, oltre a permettervi di mettere in pratica la teoria studiata nelle Sezioni precedenti, mette in luce un fatto interessante e cioè che chi disegna un linguaggio di programmazione spesso inserisce nella sintassi del suo linguaggio dei simboli di separazione (tipicamente caratteri di punteggiatura e parentesi) allo scopo di rendere l'analisi sintattica particolarmente semplice ed efficiente. Insomma la sintassi dei linguaggi di programmazione è spesso determinata da motivazioni di ordine pratico.

Per testare se G_{LK_1} è LL(1) occorre costruire la sua tabella di parsing che, a sua volta, necessita della costruzione di FIRST e FOLLOW.

Come per la prima parte del progetto, anche per questa parte **non c'è nulla da consegnare**. Si deve invece calcolare FIRST e FOLLOW di G_{LK_1} e costruire la corrispondente tabella di parsing che dovrebbe mostrare una violazione della proprietà di LL(1). Si deve quindi capire da cosa è causata questa violazione e qual'è la modifica della grammatica che risolve il problema rendendo la (nuova) grammatica LL(1). Si deve inoltre ri-calcolare la tabella di parsing della nuova grammatica in modo da mostrare che essa è effettivamente LL(1). All'orale si dovrà essere in grado di illustrare tutto questo percorso al docente.