EXTENDED DCG FOR EBNF PARSING AND AST GENERATION IN TUPROLOG

Versione 1.0

04/02/2011

Paolo Piagnani

1. INTRODUZIONE:

Lo scopo del progetto presentato in questa relazione è quello di estendere la libreria DCG di tuProlog con funzionalità aggiuntive, quali il parsing di grammatiche EBNF e la seguente generazione di un Abstract Syntax Tree.
La libreria DCG offre all'utente la possibilità di rappresentare una grammatica tramite Definite Clause Grammar, un'estensione delle grammatiche context-free. La versione attuale consente di utilizzare solo la notazione in forma di Backus-Naur, d'ora in poi abbreviata BNF, per descrivere la sintassi di linguaggi formali. Si vuole dunque estendere le funzionalità delle notazioni BNF, cioè fornendo gli strumenti opzionali delle Extended BNF che diano all'utente un maggior potere espressivo nella definizione delle proprie grammatiche. Il progetto si basa quindi sulla realizzazione di un metainterprete Prolog per la costruzione di parser per grammatiche EBNF, e sul relativo generatore AST.

2. DESCRIZIONE EBNF:

EBNF non è altro che un'estensione della notazione basica BNF realizzata per migliorarne l'espressività tramite l'uso di simboli aggiuntivi. I seguenti sono i principali:

- "**(**X**)**" indica esattamente una occorrenza di X
- "**[**X**]**" indica zero o una occorrenza di X
- "**{**X**}**" indica zero, una, o più occorrenze di X

Inoltre, EBNF include una notazione per le potenze, migliorando la leggibilità rispetto ad una scrittura BNF. Insomma, grazie all'ausilio EBNF si riesce a definire una grammatica più concisa e intuitiva. E' importante evidenziare il fatto che una notazione EBNF non è più potente in termini di quanti e quali linguaggi può definire, è semplicemente più conveniente perchè rende la scrittura più compatta. Ciascuna produzione EBNF può essere tradotta in un insieme equivalente di produzioni BNF (più difficile e generalmente impossibile tuttavia il caso contrario), dove per ogni coppia di parentesi rimossa vengono aggiunti un nuovo simbolo non terminale e una nuova produzione.
Grammatiche EBNF senza notazione di potenza descrivono sempre linguaggi di tipo 3 (con riferimento alla classificazione di Chomsky); al contrario, quelle con notazione di potenza non descrivono mai linguaggi regolari, salvo casi particolari.
Come le grammatiche BNF, anche le EBNF hanno quattro tipi di componenti:

- Simboli Terminali: E' un insieme di simboli che appaiono esplicitamente nel linguaggio specificato.
- Simboli Nonterminali: Agiscono come "segnaposti" per altri simboli che descrivono il linguaggio. La specifica di linguaggio deve contenere una regola di produzione che definisca cosa un simbolo nonterminale rappresenti.
- Regole di Produzione: Definiscono un simbolo nonterminale in termini di altri simboli nonterminali e simboli terminali.
- Simboli di Partenza: E' un designato simbolo nonterminale. Tutte le istanze valide per la sintassi del linguaggio specificato iniziano con questo simbolo.

3. INTERPRETE DCG:

Data la definizione di una certa grammatica e di un testo (elaborato magari da un tokenizzatore), il parser riconosce se tale stringa è una frase valida per la grammatica e, se questa è opportunamente definita, restituisce l'AST per tale sintassi. L'Abstract Syntax Tree è una rappresentazione ad albero della struttura sintattica astratta del codice passato dal tokenizzatore e riconosciuto dal parser.
La libreria DCG fornisce all'utente gli strumenti necessari per creare un parser di tipo top-down (LL): partendo da un simbolo nonterminale detto di "start", e dato uno stream di input sottoforma di lista, viene fatta un'espansione di tipo top-down delle regole grammaticali definite in modo formale, e i token in input vengono "consumati" da sinistra verso destra. Un parser LL(K) è un parser dove K

è il numero di token da controllare per scegliere senza ambiguità la giusta produzione della grammatica (il numero di lookahead necessari).
I predicati per il parsing DCG forniti dalla libreria sono due, e sono quelli richiamati dall'utente per iniziare il riconoscimento delle frasi della grammatica specificata (per approfondimenti si veda direttamente la documentazione ufficiale di tuProlog):

- phrase/2: il predicato phrase(+ Category, ? List) è vero sse la lista List può essere riconosciuta come una frase di tipo Category. Ovvero, List è la lista di token passata in ingresso (è possibile che sia vuota), e Category è il simbolo di partenza della grammatica in questione.
- phrase/3: il predicato phrase(+ Category, ? List, ? Rest) è vero sse il segmento tra l'inizio della lista List e l'inizio della lista Rest può essere riconosciuto come una frase di tipo Category. Ovvero, Rest è la parte rimanente della lista che non si intende "parsare".

L'interprete DCG riconosce grammatiche definite con opportuna notazione BNF, specificate nel seguente modo tramite regole di produzione:

x --> **y**

dove **x** è un simbolo non terminale definito da altri simboli terminali e non terminali **y**. Un simbolo non terminale può essere un qualunque termine Prolog, tranne una variabile o un numero. Un simbolo terminale può essere un qualunque termine. '-->' è semplicemente un operatore.
Affinchè il parser generi anche l'AST relativo alla grammatica specificata, si aggiungono opportune variabili alle regole di produzione:

```
x(A) --> y(A).
y(A) --> [A], {...}.
```

Qui i simboli terminali sono un qualsiasi termine Prolog, e il parsing non fa altro che unificarli. Tra parentesi graffe si può includere del codice Prolog, al fine di definire ad esempio proprio l'entità dei simboli terminali. Partendo dunque dall'interprete DCG esistente vengono poi aggiunte le funzionalità che permettono all'utente finale di definire la propria grammatica tramite notazione BNF.

4. REALIZZAZIONE DEL PROGETTO:

Il funzionamento dell'interprete DCG è rimasto invariato, sono state semplicemente aggiunte le necessarie funzionalità per permettere la definizione di un parsing e la generazione AST di una grammatica EBNF.

Operatore	Sintassi EBNF	Descrizione
;	X --> A;B	X produce A oppure B
*	X --> *A	X produce zero o più occorrenze di A
+	X --> +A	X produce una o più occorrenze di A
?	X --> ?A	X produce zero o una occorrenza di A
^	X --> ^(A,N)	X produce N occorrenze di A

Tab.1

Con l'utilizzo dei cinque operatori descritti nella Tab.1, e servendosi delle potenzialità del linguaggio Prolog, è possibile specificare in notazione EBNF diversi tipi di regole di produzione che ne semplificano la scrittura rispetto a quella BNF. La maggior parte delle notazioni EBNF in uso da diversi testi pongono l'operatore unario dopo il simbolo a cui riferiscono (es. A* invece che *A), ma per come viene generato l'albero delle soluzioni dall'interprete Prolog, si rende necessario invece anteporlo al simbolo relativo.
In seguito si presenta un'analisi dettagliata del codice Prolog utilizzato per implementare le nuove funzionalità, e di come servirsene per ottenere un alto livello di espressività nella realizzazione della

propria grammatica.

- **Operatore ";"** : dcg_parse(A;B, Tokens) :- dcg_parse(A, Tokens);dcg_parse(B, Tokens).
  C'è poco da commentare qui: semplicemente, il parsing di una produzione del tipo X --> A;B dà esito positivo se dà esito positivo il parsing di A oppure, in caso contario, dà esito positivo almeno il parsing di B. E' l'equivalente del simbolo "|" (pipe) più comunemente utilizzato per indicare appunto una scelta alternativa. Si è scelto di cambiare il simbolo in uso dato che "|" è già utilizzato dall'interprete Prolog per indicare una concatenazione nelle liste; si è dunque preferito optare per ";", in uso dal Prolog per indicare soluzioni disgiunte.

- **Operatore "?"** : dcg_parse(?A, LI \ LO) :- dcg_parse(A, LI \ LO);LI=LO.
  Il parsing di una produzione del tipo X --> ?(A) dà esito positivo se dà esito positivo il parsing di A, dove A può occorrere zero o una volta (opzionale). Molti testi utilizzano la notazione tra parentesi quadre, [A], ma in Prolog tale scrittura serve già a definire una lista, e quindi per motivi pratici si preferisce utilizzare un altro simbolo.
  Per la generazione d'albero è necessario aggiungere del codice Prolog se si vuole avere un output anche in caso in cui la produzione A sia vuota; lo si può fare manualmente oppure utilizzare l'apposito operatore. Si guardi il seguente esempio:

  ```
  x(M) --> ?(n(N),N,null,M).
  n(N) --> [N], {number(N)}.
  ```

  Il parsing dà esito positivo se il token in ingresso è [N] oppure [], dove N è un numero. Nel caso in cui la produzione non vi sia, viene generato comunque qualcosa nell'AST (in questo caso "null", ma si può mettere anche altro).

- **Operatore "*"**:
  ```
  dcg_parse(*A, LI \ LO) :- (dcg_parse(A,LI \ L1), LI\=L1,
                                                    dcg_parse(*A,L1 \ LO));LI=LO.
  ```
  *A indica zero o più produzioni di A. Viene fatto quindi il parsing di A, con LI\=L1 necessario ad evitare un loop infinito, e poi ricorsivamente di nuovo il parsing di *(A), finchè non è vera LI=LO (fine delle occorrenze). In molti testi viene adottata la notazione fra parentesi graffe, {A}, per indicare lo stesso tipo di produzione: tuttavia, in molte implementazioni le parentesi graffe sono già utilizzate per altri fini, e risulta più comodo usufruire invece del simbolo *. Nella libreria DCG che si è andati ad estendere infatti {A} sta ad indicare che A è un qualsiasi tipo di codice Prolog, quindi per convenzione si è preferito cambiare simbolo per indicare la tipologia di produzione descritta.
  Per generare l'AST è stato necessario aggiungere un predicato che leggesse una produzione di questo tipo: P --> *(A,X,L), dove X è la produzione relativa ad A in uscita dal generatore d'albero, ed L la lista composta dalla concatenazione di tutte le produzioni X appena generate.

  ```
  dcg_parse(*(A,_,[]),LO \ LO).
  dcg_parse(*(A,X,[X|L]), LI \ LO) :- dcg_parse(A, LI \ L1),
                                                    dcg_parse(*(A,X,L),L1 \ LO).
  ```

  Ad esempio, data la grammatica:

  ```
  x(L) --> *(a(N),num(N),L).
  a(A) --> [A], {number(A)}.
  ```

  e una lista [1,1,1], il parser con generatore AST produrrà la lista [num(1),num(1),num(1)] (è un principio analogo a quello del predicato findall/3).
  Si possono avere anche più di due simboli non terminali nella parte destra di una regola di produzione. Infatti sarebbe possibile persino definire tutta la grammatica in una regola di produzione unica (naturalemente solo se ritenuto opportuno). Basta unire le funzionalità offerte dalla nuova libreria DCG con l'espressività di Prolog, inserendo cioè nella grammatica del codice Prolog (tra parentesi graffe).
  Ad esempio, si supponga di voler racchiudere, in un'unica regola di produzione (simboli

terminali a parte), la definizione di una grammatica composta da una serie di n letterali, seguita da una serie di m numeri. Sia x il simbolo di partenza della seguente grammatica EBNF:

```
x(L) --> *(a(A),lit(A),L1), *(n(N),num(N),L2), {append(L1,L2,L)}.
a(A) --> [A], {atom(A)}.
n(N) --> [N], {number(N)}.
```

Si ha quindi una scrittura più concisa rispetto alla equivalente in notazione BNF.

- **Operatore "+"**:
```
dcg_parse(+(A,X,[X|L]), LI \ LO) :- dcg_parse(A, LI \ L1),
                                    dcg_parse(*(A,X,L),L1 \ LO).
dcg_parse(+A, LI \ LO) :- (dcg_parse(A,LI \ L1), LI\=L1,
                           dcg_parse(*A,L1 \ LO)).
```

Le considerazioni su questo tipo di produzioni sono analoghe a quelle per il simbolo *, ad eccezione del fatto che +A indica almeno una o più occorrenze di A. Una regola di produzione X --> +(A) è semplicemente l'equivalente di X --> (A),*(A), ma per ridurre la lunghezza della definizione della grammatica si è ritenuto opportuno definire l'operatore direttamente nel codice dell'interprete.

- **Operatore "^" / "#":**
```
dcg_parse(^(A,N), LI \ LO) :- power_parse(^(A,N,0),LI \ LO).
power_parse(^(A,N,N),LO \ LO).
power_parse(^(A,N,M), LI \ LO) :- M1 is M+1, !,dcg_parse(A, LI \ L1),
                                  power_parse(^(A,N,M1),L1 \ LO).
```

Le produzioni del tipo X --> ^(A,N) indicano che vi sono esattamente N occorrenze di A. Come per i due precedenti operatori, per generare l'AST è necessario specificare anche la produzione in uscita dal generatore d'albero relativa ad A, e la lista L composta dalla concatenazione di tutte le produzioni X appena generate. Si è scelto di cambiare però l'operatore utilizzato, in quanto l'interprete riscontrava problemi nel riconoscere le corrette produzioni:

```
dcg_parse(#(A,N,X,L), LI \ LO) :- dcg_power(#(A,N,0,X,L),LI \ LO).
dcg_power(#(A,N,N,_,[]),LO \ LO).
dcg_power(#(A,N,M,X,[X|L]), LI \ LO) :- M1 is M+1, !, dcg_parse(A, LI \
                                        L1), dcg_power(#(A,N,M1,X,L),L1 \ LO).
```

Con l'ausilio di opportuno codice Prolog è possibile imporre diversi tipi di vincoli sui valori della variabile N. Di seguito sono riportati alcuni esempi:

Es. 1:

```
a(A) --> [A], {atom(A)}.
n(N) --> [N], {number(N)}.
x(L) --> #(a(A),N,lit(A),L1), #(n(B),M,num(B),L2), {N<M},{append(L1,L2,L)}.
parse(L,T):-phrase(x(T),L),!.
```

```
?-parse([a,a,1,1,1],T). yes. T / [lit(a),lit(a),num(1),num(1),num(1)].
?-parse([a,a,a,1,1,1],T). no.
```

Es. 2:

```
a(A) --> [A], {atom(A)}.
n(N) --> [N], {number(N)}.
x(L) --> #(a(A),N1,lit(A),L1), #(n(B),N2,num(B),L2), #(a(C),N3,lit(C),L3),{N3 is N1+N2},
        {append(L1,L2,LT), append(LT,L3,L)}.
parse(L,T):-phrase(x(T),L),!.
```

```
?-parse([a,1,1,b,b,b],T). yes. T / [lit(a),num(1),num(1),lit(b),lit(b),lit(b)].
```

?-parse([a,1,1,b,b],T). no.

Non è stata necessaria nessuna specifica per i simboli tra parentesi tonde, ad esempio per una regola di produzione del tipo X --> (A), che indica cioè esattamente una sola occorrenza di A, in quanto è una proprietà implicitamente soddisfatta, e non necessita nemmeno dell'utilizzo di parentesi tonde. Ovvero, X --> (A) equivale a X --> A. Le parentesi tonde sono lasciate ad utilizzo dell'utente a scopo di raggruppamento per gestire le precedenze nella propria grammatica.

5. CONCLUSIONI

Lo scopo del lavoro svolto è stato quello di estendere la libreria DCG di tuProlog con le funzionalità della Extended BNF, ottenendo un interprete per costruire parser in grado di leggere grammatiche definite con tale notazione, e di generarne l'AST relativo.
L'esecuzione del progetto si è suddivisa in diverse fasi: ricerca di materiale utile, sia su Internet che su materiale didattico, uno studio della struttura della libreria DCG di tuProlog già esistente, la scrittura del codice aggiuntivo, una revisione per cercare di ottimizzarlo, e alcuni test sulle frasi esprimibili tramite la nuova sintassi.
Il progetto è stato fatto partendo dal codice già esistente dell'interprete DCG, estendendolo al fine di poter riconoscere produzioni EBNF: l'interprete può riconoscere, data una regola di produzione A --> B, un'occorrenza opzionale ?(B), una o piu occorrenze +(B), zero o più occorrenze *(B), un'occorrenza alternativa (A;B). Inoltre, l'interprete può gestire anche notazioni di potenza ^(B,N), dove N è il numero di occorrenze di B. Infine, le potenzialità del nuovo interprete DCG sono ulteriormente arricchite dalla possibiltà di combinare codice Prolog A tramite la notazione {A} (già presente comunque nella versione precedente della libreria DCG).
L'interprete è stato esteso mantenendo lo standard di utilizzo di ricorsioni "tail", ovvero ogni volta che vi è una chiamata ricorsiva, questa è l'ultima operazione ad essere eseguita, ed il risultato è quindi subito disponibile per essere restituito. Sostanzialmente, si è ottimizzato il codice Prolog trasformando le ricorsioni in iterazioni e, anche se i predicati possono risultare un pò meno intuitivi, la loro esecuzione è più efficiente rispetto ad una soluzione con ricorsione "non tail".
Dal progetto realizzato si è dunque ottenuto un interprete DCG in grado di leggere una notazione EBNF usata per definire delle grammatiche. Nonostante tale notazione non sia più potente rispetto ad una BNF in termini di quanti e quali grammatiche possa definire, permette di scriverle in modo più compatto, aumentandone quindi l'espressività grazie al potere di rappresentare direttamente opzioni e ripetizioni con tanto di specifica del numero esatto.

6. BIBLIOGRAFIA

1. Mirko Viroli, "4a Formal Grammars", Linguaggi e Modelli Computazionali, A.A. 2009/2010.
2. Mirko Viroli, "4b Parsing", Linguaggi e Modelli Computazionali, A.A. 2009/2010.
3. Matteo Casadei, "Practical Session E6 Exercises on Parsing", Linguaggi e Modelli Computazionali, A.A. 2009/2010.
4. W3C, "Extensible Markup Language (XML) 1.0 (Fifth Edition)", http://www.w3.org/TR/REC-xml/#sec-notation , Novembre 2008.
5. Wikipedia, "Extended Backus-Naur Form", http://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form , Gennaio 2011.
6. Lars Marius Garshol, "BNF and EBNF: What are they and how do they work?", http://www.garshol.priv.no/download/text/bnf.html#id2.3. , Agosto 2008.
7. "tuProlog Guide", Alma Mater Studiorum, Università di Bologna a Cesena, Aprile 2007.