Progetto di Linguaggi di programmazione
2015-2016
Terza e Quarta parte del Progetto

Filè Gilberto

10 dicembre 2015

1 Una grammatica LL(1) per LispKit

Nella parte 2 del progetto era stata proposta la seguente grammatica libera da contesto per il LispKit con la richiesta di costruirne la tabella di parsing. Era anche stato anticipato il fatto che questa grammatica non sarebbe risultata LL(1).

```
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
```

Diamo ora un piccolo aiuto per determinare quale sia il problema che impedisce a questa grammatica di essere LL(1). Questa grammatica può generare un'espressione come la seguente: `A( B (C))` che starebbe per un'invocazione della funzione `A` con una lista di parametri attuali che contiene un'ambiguità: questa lista potrebbe consistere di un solo parametro che sarebbe l'invocazione `B(C)`, oppure di 2 parametri `B` e `(C)`, cioè di 2 espressioni che consistono di una variabile ciascuna (la seconda racchiusa tra parentesi). Questa ambiguità rende impossibile decidere se il nonterminale Y, con '(' in input, debba continuare con la produzione 10.1 oppure con la 10.2. A voi ora

di modificare la grammatica in modo da eliminare il problema. In fondo si tratta di trovare un modo per far si che gli elementi di una lista di parametri siano separati tra loro in modo che non sia possibile che non sia più possibile interpretare la sequenza $var\ (...)$ come un unico parametro (un'invocazione della funzione var) e anche come 2 parametri, in cui il primo parametro è un'espressione che consiste della variabile var ed il secondo parametro è un'espressione tra parentesi.

Nel seguito assumiamo di avere modificato la precedente grammatica in modo che sia LL(1) e chiameremo G_{LK} questa nuova grammatica. Si chiede di costruire la sua tabella di parsing in modo da assicurarsi che sia LL(1).

Per ottenere una grammatica G_{LK} che sia LL(1) basta aggiungere un nuovo simbolo terminale (per separare i parametri attuali), un nuovo nonterminale (capace di generare il separatore) e 2 nuove produzioni relative al nuovo nonterminale. Sarà anche necessario modificare una delle produzioni originali in modo da far cooperare il nuovo nonterminale con quelli già presenti. **Fate attenzione al fatto che l'aggiunta di un nuovo simbolo terminale al linguaggio LispKit sicuramente richiede una modifica dell'analizzatore lessicale che dovrà gestire correttamente il nuovo terminale.**

2 Parte 3: il parser predittivo

Una volta trovata G_{LK} e calcolata la sua tabella di parsing, si richiede di costruire il corrispondente parser predittivo. La descrizione generale di un tale parser è stata delineata nella parte 2 del progetto. Per aiutarvi nella costruzione del parser per G_{LK}, vi viene dato un parser parziale scritto in Haskell. Si tratta di completarlo.

L'analizzatore sintattico (parziale) dato, segue la programmazione basata sulla nozione di monade. Questo tipo di programmazione permette di gestire i casi di errore senza usare eccezioni e restando quindi nella programmazione dichiarativa. Inoltre il programma per ciascun caso d'errore produce opportune stringhe di commenti che spiegano chiaramente cosa succede. Quindi la purezza del programma non va a discapito della sua espressività a livello di gestione degli errori rilevati durante il parsing.

La parte del programma che viene data contiene diverse cose significative. In particolare, contiene la monade che usiamo, che è Exc, che abbiamo visto a lezione (vedi slide della lezione 12). Il programma contiene anche le funzioni che corrispondono a molti nonterminali della grammatica, quali, `Prog`,`Exp`, `Bind`, e i diversi nonterminali usati nella grammatica per produrre i diversi tipi di espressioni. Sono anche date molte funzioni che servono a cercare in cima alla lista di Token un Token particolare. Vengono lasciate da fare le funzioni relative alla produzione di liste di argomenti, cioè quelle che contengono la porzione di grammatica che la Parte 2 del progetto richiedeva di modificare. Praticamente tutte le funzioni importanti hanno tipo: `[Token] -> Exc [Token]` e in questo modo è facile sfruttare la monade Exc per gestire i casi di errore del parsing.

3 Il linguaggio LKC

L'analizzatore sintattico predittivo della Sezione 2, durante l'analisi sintattica di un programma LispKit, attraverso le invocazioni delle funzioni, costruisce l'albero sintattico del programma analizzato, ma non mantiene memoria dell'albero costruito. Potremmo dire che esso percorre l'albero per controllare che un tale albero esista, ma che, non appena ha la prova della sua esistenza, se ne dimentica. Questo è un vero peccato. Infatti l'albero sintattico che corrisponde ad un programma costituisce un'informazione sulla struttura del programma che sarebbe molto utile per tradurre il programma in altri linguaggi come quello della macchina virtuale SECD, cosa che ci proponiamo di fare nell'ultima parte del progetto. Quindi, la quarta parte del progetto consiste nell'estendere il parser predittivo della Sezione 2 in modo che, oltre al controllo della correttezza sintattica dei programmi LispKit, esso sia capace di produrre una forma astratta (semplificata) dell'albero di derivazione dei programmi che analizza.

Cerchiamo di essere pratici. Un albero di derivazione astratto è per noi un valore del tipo Haskell LKC (che sta per LispKit Concreto). Quindi, in pratica, vogliamo scrivere un parser che abbia il seguente tipo: `[Token] -> Exc ([Token]* LKC)`, in cui LKC in `Exc ([Token]* LKC)` rappresenta l'albero sintattico del programma contenuto nella sequenza di `Token` in input. Il tipo Haskell LKC è definito nel modo seguente:

```
data LKC = ETY   | --segnala epsilon productions
           VAR String | NUM Integer |STRI String | BOO Bool |
           NIL  |   ADD LKC LKC  | SUB LKC LKC  | MULT LKC LKC  |
           REM LKC LKC |  DIV LKC LKC  | EQC LKC LKC | LEQC LKC LKC |
           CARC LKC   | CDRC LKC |  CONSC LKC LKC | ATOMC LKC  |
           IFC LKC LKC LKC | LAMBDAC [LKC]  LKC | CALL LKC [LKC] |
           LETC LKC  [(LKC,LKC)] | LETRECC  LKC  [(LKC, LKC)]
           deriving(Show, Eq)
```

Consideriamo un semplice programma LispKit e come esso viene tradotto in un corrispondente valore di tipo LKC.

```
let x=5 and y= 6 in x + y * 2 end
```

La sua traduzione in LKC è some segue:

```
LET(ADD(VAR "x", MULT(VAR "y", NUM 2)),
    [(VAR "x", NUM 5), (VAR "y", NUM 6)])
```

Da questo esempio si possono capire varie cose sul tipo LKC:

1. I costrutti del LispKit, come `let`, vengono rappresentati in LKC con dei costruttori (come`LET`) che permettono di raccogliere nei loro parametri le diverse parti del costrutto del LispKit che essi rappresentano, ovviamente tradotte in LKC. In riferimento ai parametri del `LET` facciamo le seguenti osservazioni:

2. Il corpo del `let`, cioè `x+y*2` è il primo parametro del costruttore `LET`. L'espressione viene rappresentata in LKC sostituendo al simbolo `+` il costruttore `ADD` e sostituendo `MULT` a `*`. Le costanti intere vengono rappresentate con il costruttore `NUM` che ha come parametro il valore intero rappresentato, come `NUM 5` e `NUM 6` nell'esempio. Le variabili `x` e `y` sono rappresentate con il costruttore `VAR` di LKC che ha come parametro la stringa `''x''` e `''y''`, rispettivamente.

3. La lista dei `Bind` del `let` viene tradotta nella lista:

   ```
   [(VAR ''x'',NUM 5),(VAR ''y'',NUM 6)]
   ```

   in cui ciascuna coppia rappresenta la parte sinistra e destra di ciascun binder. La lista è il secondo parametro del costruttore `LET`.

4. la traduzione in LKC è un valore (un albero) che è una rappresentazione del programma LispKit di partenza, particolarmente adatta a rendere semplici operazioni sul programma stesso, quali per esempio la sua compilazione in un altro linguaggio. Questa è esattamente il motivo per cui vogliamo che il nostro secondo parser produca la traduzione in LKC.

Dovrebbe essere facile capire dal nome usato lo scopo di ciascuno dei costruttori LKC. In alcuni casi è stata aggiunta una 'C' finale, come in `LAMBDAC`, per evitare di ripetere nomi già usati in altri data type precedenti. Vale la pena di segnalare il costruttore `CALL` che serve a rappresentare le invocazioni di funzione nel modo seguente: il primo parametro di `CALL` rappresenta la funzione che viene invocata (il nome della funzione in LKC, cioè, per esempio `VAR ''f''`), mentre il secondo parametro è la lista dei parametri attuali dell'invocazione (tradotti in LKC).

4 Con gli attributi il parser diventa traduttore

La tecnica che useremo per trasformare l'analizzatore sintattico della Sezione 2 in un traduttore è molto generale e si chiama tecnica degli **attributi semantici**. In generale, questa tecnica consiste nell'aggiungere attributi e regole semantiche alle produzioni della grammatica libera da contesto usata per il parsing. L'idea generale è che un parser che *percorre* un albero sintattico, contemporaneamente calcoli anche i valori degli attributi dei nodi di questo albero usando delle associate regole semantiche. I valori di alcuni attributi servono per calcolare quello di altri attributi, finché tutti gli attributi dell'albero sono calcolati. In pratica, nel nostro parser predittivo, gli attributi sono o dei parametri o dei risultati delle funzioni che compongono il parser. Nel seguito assumeremo che ogni nonterminale possieda (almeno) un attributo `Trad`. Dimenticandoci per il momento della monade Exc, quanto supposto significa che la funzione associata ad ogni nonterminale restituisce un risultato di tipo `[Token]*Trad`. Attributi che sono calcolati come risultati delle funzioni di parsing sono chiamati **sintetizzati**. Sono invece **ereditati** gli attributi che sono calcolati come parametri attuali di invocazioni di funzioni di parsing. La maggior parte degli attributi che useremo sono sintetizzati. Questi attributi vengono calcolati secondo un ordine bottom-up sull'albero di derivazione che spiegheremo con un esempio. Consideriamo la produzione di G_{LK} che introduce il `let`:

`1.1 Prog::= let Bind in Exp end`

I nonterminali `Bind` ed `Exp`, che occorrono nella parte destra della produzione 1.1, possiedono l'attributo sintetizzato `Trad`. I due atributi devono assumere i seguenti valori: `Trad(Bind)` deve assumere come valore la lista delle coppie `[(Var ''x1'', LKC(Exp1)),...,(Var ''xn'', LKC(Expn))]:LKC*LKC list` che è la traduzione in LKC della lista dei binders generati da `Bind`. Invece `Trad(Exp)` deve assumere come valore la traduzione in LKC dell'espressione generata da `Exp` (il corpo del `let`). La produzione 1.1 possiede una regola semantica che ha il compito di definire l'attributo `Trad` di `Prog` nel modo seguente: `Trad(Prog) = LET(Trad(Exp),Trad(Bind))`. Diciamo che questa regola semantica segue lo schema bottom up perché essa calcola l'attributo `Trad` del nonterminale che si trova nella parte sinistra della produzione 1.1, in funzione del valore degli attributi `Trad` dei nonterminali della parte destra di 1.1. Se osservassimo un'occorrenza di 1.1 in un albero sintattico, vedremmo che questa regola semantica calcola i valori degli attributi dal basso verso l'alto dell'albero.

Se invece considerassimo la faccenda dal punto di vista delle funzioni di parsing coinvolte, allora potremmo osservare che la funzione di parsing che corrisponde al nonterminale `Prog`, calcola `Trad(Bind)` e `Trad(Exp)` al suo interno e usa questi valori per calcolare `Trad(Prog)` che è (parte del) risultato della funzione `prog`.

La maggior parte delle produzioni di G_{LK} possiede solo attributi sintetizzati che seguono lo schema bottom-up appena descritto. Le sole eccezioni sono le produzioni 5, 6, 7 e 8 che necessitano anche di attributi **ereditati**. Vediamo perché.

Si considerino per esempio le produzioni 5 e 6:

```
5  ExpA::= T E1
6  E1_a::= OPA T E1_b | epsilon

// gli indici a e b delle 2 occorrenze di  E1
// servono solo a distinguerle
```

e si assuma che la 6.1 venga usata (in qualche albero di derivazione) per espandere il nonterminale `E1` della parte destra della 5.1. e che `OPA` generi `+`. In questa situazione è facile osservare che la produzione 6.1 *sa* che sta applicando la somma, ma *ha a disposizione* solo il secondo operando della somma (quello prodotto da `T` della parte destra della 6.1), mentre il primo operando è generato dal `T` della parte destra della 5.1. Quindi la 6.1 deve avere una regola semantica che produce un valore di tipo `LKC` come il seguente: `ADD( ?, Trad(T))` in cui il punto interrogativo indica il primo operando che manca. Come farlo arrivare nella produzione 6.1? La risposta è semplice: basta che `E1_a` possieda anche un attributo ereditato (chiamiamolo `PO` per "Primo Operando") attraverso il quale il nonterminale `T` della 5.1 gli fa arrivare il suo attributo `Trad(T)` che appunto è l'operando mancante. In pratica, per ottenere questo effetto, basta che la produzione 5.1 abbia una regola semantica come questa: `PO(E1)=Trad(T)`.

Quindi nella 6.1 potremo costruire il valore `LKC` desiderato nel modo seguente: `ADD(OP(E1_a), Trad(T))`. Resta ora da capire cosa fare con questo valore

LKC. In un primo momento si potrebbe pensare che essa debba diventare il valore dell'attributo sintetizzato Trad(E1_a), ma questo è vero solo nel caso che l'espressione generata da E1_a termini qui, cioè nel caso in cui E1_b generi la stringa vuota ϵ. Se invece E1_b genera un ulteriore pezzo di espressione, allora dobbiamo avere una regola semantica che gli passi il valore LKC appena costruito, cioè: OP(E1_b)=ADD(OP(E1_a),Trad(T)) mentre il valore di Trad(E1_b) sarà la traduzione in LKC dell'intera espressione generata da ExpA della produzione 5.1 e quindi la produzione 6.1 dovrà possedere anche la regola semantica: Trad(E1_a)=Trad(E1_b) e la 5.1 avrà un'analoga regola semantica, Trad(ExpA)=Trad(E1). È importante ricordare che in questo esempio stiamo considerando una porzione di albero sintattico in cui il nonterminale E1 di 5.1 e E1_a di 6.1 sono lo stesso nonterminale e quindi le due regole semantiche appena descritte hanno lo scopo di ricopiare il valore di Trad(E1_b) di 6.1 in ExpA di 5.1.

Ragionando in termini delle funzioni di parsing che sono coinvolte per processare le produzioni 5 e 6, dovrebbe essere chiaro che la funzione fune1 (che corrisponde al nonterminale E1) deve possedere un parametro formale aggiuntivo che realizza l'attributo ereditato PO. Da parte sua, la funzione expa, che invoca fune1, avrà cura di invocarla, passando Trad(T) come valore attuale del parametro PO.

Un ragionamento assolutamente simile a quello appena esposto, vale per le produzioni 7 e 8 di G_{LK} e quindi regole semantiche analoghe a quelle appena descritte dovranno venire associate a queste produzioni.

Va osservato che quanto descritto ora non è possibile per ogni grammatica libera da contesto e sistema di attributi semantici definito su quella grammatica. Infatti non è sempre possibile calcolare gli attributi semantici contemporaneamente ad un unico percorso dell'albero sintattico. Questo è possibile solo quando le dipendenze tra gli attributi (che determinano l'ordine di calcolo degli attributi) seguono lo stesso ordine seguito dal parser nel percorrere l'albero sintattico, cioè l'ordine in profondità da sinistra a destra nel caso di parser predittivo. Questo è il caso del sistema di attributi appena descritto sulla grammatica G_{LK}.

5 Parte 4

Questa parte del progetto consiste nel modificare l'analizzatore sintattico della Sezione 2 in modo che produca la traduzione in LKC del programma LispKit analizzato. Nell'eseguire questo compito, è richiesto di seguire le direttive date nella Sezione 4.