Progetto di Linguaggi di programmazione 2015-16
Prima parte
Analizzatore lessicale del Linguaggio LispKit

G.Filè

16 ottobre 2015

Il Linguaggio LispKit

Il LispKit è un linguaggio funzionale molto semplice, ma sufficientemente complesso da illustrare molte cose interessanti dei linguaggi di programmazione che vengono trattate durante il corso. La principale semplificazione del LispKit è che non ha tipi,ma può comunque manipolare valori diversi che sono i seguenti:

1. valori interi $12, 0, \sim 3 \ldots$
2. valori booleani true e false
3. stringhe costanti \″ecco una stringa\″
4. liste come cons(0 cons(true cons(\″abc\″ nil))).

Si deve subito osservare che le liste sono costruite con l'uso dell'operatore `cons` che corrisponde all'operatore `:` di Haskell. Quindi in LispKit non si possono scrivere liste con le parentesi quadrate come in Haskell, ma solo con espressioni che usano l'operatore `cons` che costruisce liste qualsiasi aggiungendo un elemento per ciascun `cons`. Questa scelta è volta a mantenere il linguaggio semplice senza perdere la possibilità di manipolare liste. Inoltre l'esempio 4 mostra che in LispKit le liste non sono omogenee, ma possono contenere oggetti di tipo completamente diverso. D'altra parte i tipi non esistono in LispKit. Oltre a questo, è importante osservare che i due parametri di `cons` sono separati semplicemente da uno o più spazi. Questo è vero per tutte le funzioni e operatori prefissi.

Ogni programma LispKit è costituito da una serie di dichiarazioni locali, contenute in un costrutto `let` o `letrec` seguite da una espressione che usa le dichiarazioni. Questa espressione può essere costituita da altre dichiarazioni locali seguite da un'espressione che le usa e così via con annidamento di profondità arbitraria. Vediamo qualche esempio di programma LispKit.

```
val D= "let x=cons(\"ab\" cons(\"cd\" nil))
in if true then cons(\"01\" x) else nil end $";
```

La variabile `D` ha come valore una stringa che è un programma LispKit che dichiara in `x` una lista che consiste di 2 stringhe costanti (\″ab\″ \″cd\″) che è seguita da un'espressione che aggiunge come primo elemento della lista la stringa costante \″01\″. In LispKIT le stringhe costanti sono iniziate e terminate da \″ per facilitarne il riconoscimento. Infatti in questo modo sono chiaramente diverse dalle keyword e dalle variabili. Inoltre l'esempio mostra nuovamente che il LispKit, per costruire liste, usa l'operatore `cons` che corrisponde al `:` di Haskell ed usa anche il costrutto condizionale `if_then_else`. Si osservi inoltre che, come per le liste, così anche i parametri formali e attuali delle funzioni sono separati solamente da spazi. LispKit non ha tipi. Sta al programmatore scrivere cose che mescolano valori solo in modo sensato. Ogni programma termina sempre con il simbolo speciale $. Il programma `D` è un possibile input dell'analizzatore lessicale che chiameremo

`lexi`. Quella che segue è la ben nota funzione ricorsiva `reverse` che rovescia una lista qualsiasi usando un secondo parametro per contenere in ogni momento la lista rovesciata prodotta fino a quel momento.

```
val R="letrec rev = lambda(x y) if eq(x nil)
then y else rev(cdr(x)  cons(car(x) y))
in  rev(cons(0 cons(1 cons(2 nil))) nil) end $";
```

Va notato che la dichiarazione locale inizia ora con la keyword `letrec` (anziché `let`) ad indicare che si sta per definire una funzione ricorsiva. L'esempio mostra che la keyword che in LispKit definisce le espressioni funzionali è `lambda`, anziché il \ di Haskell). Inoltre la sintassi prevede che i parametri formali seguano la keyword `lambda` e siano racchiusi tra parentesi tonde e siano separati da spazi, mentre il corpo della funzione semplicemente segue la lista dei parametri formali. Le operazioni `head` e `tail` in LispKit sono indicate col nome che queste operazioni hanno nel linguaggio Lisp e cioè, `car` e `cdr`, rispettivamente. Il test di uguaglianza del LispKit è indicato con la funzione binaria `eq`.

lambda indica la definizione di una funzione anonima

Esaminiamo ora un esempio più complesso con dichiarazioni di funzioni ricorsive e di ordine superiore.

```
val C="letrec  FACT = lambda (X) if  eq(X 0)
then 1 else  X*FACT(X-1)and
G = lambda (H L) if  eq (L  nil)
then L else cons(H(car(L)) G (H cdr (L)))
in G ( FACT cons(1 cons(2 cons(3 nil)))) end $";
```

Si noti nuovamente l'uso di `letrec` ad indicare che si stanno per dichiarare funzioni ricorsive. Nel `letrec` vengono dichiarate due funzioni ricorsive (separate dalla keyword `and`). La prima è la funzione fattoriale, mentre la seconda `G` è una funzione di ordine superiore in quanto aspetta una funzione `H` come parametro. Il fatto che il parametro `H` debba essere una funzione, lo si deduce dal suo uso nel corpo di `G`. Questo fatto viene anche confermato dalla successiva espressione che invoca `G` passandole proprio `FACT` come primo parametro.

L'analizzatore lessicale

L'analizzatore lessicale riceve come input un programma in LispKit, cioè una lista di caratteri e deve riconoscere le componenti elementari del linguaggio e deve metterle in una forma che sia semplice da manipolare nella successiva fase di analisi sintattica. Per esempio, deve riconoscere le costanti (per esempio i numeri interi oppure il valore true, eccetera), le parole chiave, gli identificatori, gli operatori ed i simboli di separazione.

Ognuna di queste componenti elementari viene rappresentata da un valore del seguente tipo Haskell `token`:

```
data Keyword_T = LET | IN | END | LETREC | AND | IF | THEN |
                 ELSE | LAMBDA
                 deriving (Show,Eq)

data Operator_T = EQ | LEQ | CAR | CDR | CONS | ATOM
    deriving (Show,Eq)

data Symbol_T = LPAREN | RPAREN | EQUALS | PLUS | MINUS |
                TIMES | DIVISION |VIRGOLA| DOLLAR
                deriving (Show,Eq)

data Token = Keyword Keyword_T | Operator Operator_T |
             Id String | Symbol Symbol_T | Number Integer |
             String String | Bool Bool | Nil
             deriving (Show,Eq)
```

Per semplicità nel seguito chiameremo **token** i valori del tipo `Token`. Vediamo alcuni esempi di token corrispondenti a componenti elementari di programmi LispKit:

- numeri interi: `Number n`, dove `n` è un intero;
- stringhe: `String s`, dove `s` è una stringa (senza \″);
- identificatori: `Id s`, dove `s` è la stringa corrispondente all'identificatore;
- corrispondentemente a ciascuna keyword K, verrà prodotto il token `Keyword K'`, dove `K'` è il valore `Keyword_T` corrispondente a K, per esempio, se K=let, allora il token corrisponente è `Keyword LET` e `Keyword THEN` rappresenta la keyword then e così via.
- nil: `Nil`;
- true e false : `Bool true` e `Bool false`;
- i simboli come `+,=,(` eccetera sono rappresentati da

  `Symbol PLUS, Symbol EQUALS, Symbol LPAREN` eccetera, dove `PLUS, EQUALS, LPAREN` sono costruttori del tipo `Symbol_T`.
- per gli operatori come `eq, car, cdr` eccetera, essi sono rappresentati da `Operator EQ`, `Operator CAR`, `Operator CDR`, eccetera, dove `EQ,CAR,CDR` sono costruttori del tipo `Operator_T`.

La segnatura della funzione Haskell `lexi` che realizza l'analizzatore è dunque la seguente:

`lexi:: [char] -> [token]`

Il tipo esprime che `lexi` deve trasformare una stringa che è un programma LispKit nella corrispondente lista di `token`. Questa funzione deve implementare un automa a stati finiti che funziona secondo i seguenti principi. Partendo da uno stato iniziale I considera un carattere alla volta della sua lista in input. Chiamiamo questo carattere `s`:

- se `s` è uno spazio, lo si salta;
- se `s` è il dollaro, l'analisi termina;
- se `s` è numerico o è il carattere tilde che indica il meno unario, allora si deve passare ad un nuovo stato N (per Numero) atto a riconoscere i numeri; in N si continueranno a leggere caratteri numerici fino a che non si legga un non numerico e alla fine si produrrà il token `Number x` dove x è l'intero corrispondente alla stringa di caratteri letta;
- se invece `s` è un simbolo di parentesi o di = allora si deve produrre la corrispondente coppia `Symbol LPAREN` o `Symbol RPAREN` o `Symbol EQUALS`;
- infine se `s` è un simbolo alfabetico allora si deve passare ad un nuovo stato S (per Stringa) che legge tutti i caratteri che seguono, fino ad un carattere che non può stare in una stringa (cioè né alfabetico né numerico) ed a quel punto dovrà "capire" se ha letto una keyword o una costante del LispKit (true, false e nil) oppure un identificatore del programma. Ovviamente le possibilità sono mutuamente esclusive: non ci possono essere variabili che sono anche keyword o costanti del LispKit. A seconda che si sia letta una keyword, o una costante o un identificatore, si produce il token corrispondente. Si osservi che una stringa come "falsetto" deve venire interpretata come la stringa "falsetto" e non la costante booleana "false" seguita da "tto". Per avere questo secondo risultato sarebbe necessario scrivere "false tto".
- se `s` è il carattere $'\backslash''$ che indica l'inizio di una stringa costante, allora si deve passare ad uno stato SC nel quale leggere tutti i caratteri che seguono fino al prossimo $\backslash''$ che chiude la stringa costante. Sia `s` la stringa racchiusa dai 2 simboli $\backslash''$, allora il token da produrre è `String s`;
- Una volta che si è prodotto un token si ritorna allo stato I e si ricomincia.

Il fatto che `lexi` implementi un automa a stati finiti non deve essere preso troppo alla lettera. Gli stati sono realizzati con funzioni (generalmente ricorsive per ripetere le operazioni di scansione della lista di caratteri) e la transizione dallo stato iniziale I verso un altro stato X è realizzata semplicemente da un'invocazione della funzione che realizza lo stato X. Il ritorno da X a I è realizzato semplicemente con il return alla fine della funzione che corrisponde a X. È importante osservare

che le funzioni S, SC ed N, che realizzano l'automa, *consumano* la lista di `char` e quindi quando ritornano ad I devono restituire a I quello che è rimasto della lista di `char`, cioè la parte della lista che deve ancora venire esaminata.