POLITECNICO DI TORINO

Laboratorio di Compilatori

Corso di Linguaggi e Traduttori

Esercitazione 5

Stefano Scanzio
mail: stefano.scanzio@polito.it
sito: http://www.skenz.it/traduttori

a.a 2010 / 2011

Attributi ereditati

- Sono utili per esprimere la dipendenza di costrutti di un linguaggio dal contesto in cui si trovano.
- a , b : int ;

D → L ':' T ';'	`L.type = T.type`
L → L_1 ',' id	`L1.type = L.type` `new_var(id.name,L.type)`
L → id	`new_var(id.name,L.type)`
T →'integer'	`T.type = type_int`

Attributi *L*

- L'ordine di valutazione degli attributi dipende dall'ordine con cui vengono creati o visitati i nodi dell'albero di derivazione.
- Comunemente i parser seguono l'ordine di una visita in profondità dell'albero.
- Una grammatica è detta ad *attributi L* se è possibile il calcolo dei valori degli attributi con una visita in profondità dell'albero di derivazione.
- In tali grammatiche si ha una propagazione delle informazioni da sinistra a destra (dell'albero di derivazione).
- La grammatica precedente non è una grammatica ad *attributi L*
  - Il passaggio delle informazioni avviene da destra a sinistra
  - CUP gestire solo grammatiche ad *attributi L*

Attributi ereditati di tipo L

- int a, b;

D → T L ';'	`L.type = T.type`
L → L_1 ',' id	`L1.type = L.type` `new_var(id.name,L.type)`
L → id	`new_var(id.name,L.type)`
T →'integer'	`T.type = type_int`

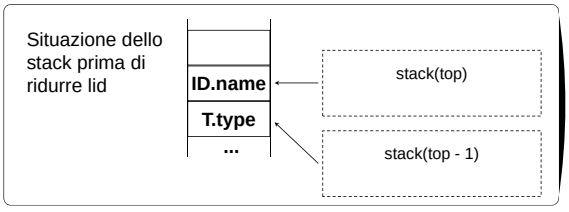
Calcolo di attributi ereditati

- In un riconoscitore bottom-up, non viene riservato spazio sullo stack semantico fino a che il corrispondente simbolo sintattico non è riconosciuto.
- Ciò rende problematica la gestione degli attributi ereditati.
- Se la grammatica è ad attributi L, e le regole semantiche non sono complesse è di solito possibile "aggirare l'ostacolo" in maniera semplice.
- Se le regole sono più complesse è possibile inserire dei *marker*, cioè dei non-terminali che si espandono in ε.

Calcolo di attributi ereditati

- Un semplice esempio:

D → T lid PV	`lid.type = T.type`
lid → ID	`var(ID.name,lid.type)`



Calcolo degli attributi ereditati (I)

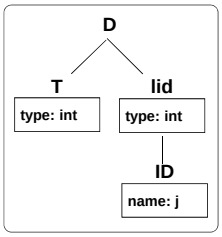
- Per accedere ai valori semantici memorizzati nello stack usare la funzione :

Object stack(int position)

```
parser code {:
……..
public Object stack ( int position){
   // returns the object at the specified  position
      from the top (tos) of the stack
      return(((Symbol)stack.
      elementAt(tos+position)).value);
   }
……..
:}
```

- *stack(0)* indica il valore semantico associato al simbolo in cima allo Stack;
- *stack(n)* indica il valore semantico associato al simbolo nella positione top-n nello Stack

Calcolo di attributi ereditati

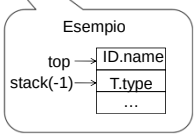


- Il simbolo 'lid' ha l'attributo ereditato 'type'.
- I valore di tale attributo è presente sullo stack semantico (nella posizione associata a 'T') prima cha venga creato il simbolo 'lid'.
- Tuttavia esso è al di fuori del contesto semantico della regola relativa a 'lid'.

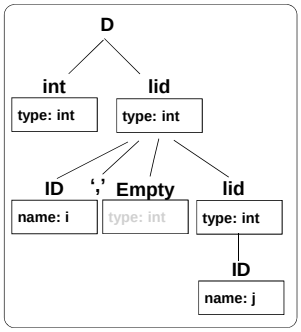
Calcolo degli attributi ereditati (II)

- Assumendo che il simbolo 'lid' sia sempre preceduto da un identificatore di tipo:

```
lid ::= ID:name {:
        String type = (String) parser.stack(-1);
        RESULT = new String (type);
        add_id(name, RESULT);
:} ;
```

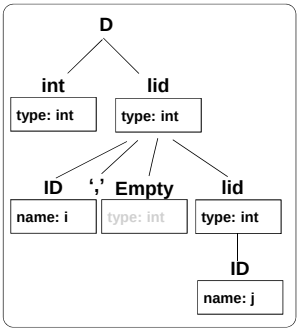


Calcolo di attributi ereditati mediante marker



- Aggiungendo la regola
  lid ::= ID VIR lid ;
  non è più vero che 'ID' è sempre preceduto da un identificatore di tipo.
- Nel caso della regola:
  lid ::= ID ;
  Il simbolo che precede 'ID' nello stack prima dell'operazione di reduce è 'VIR'

Calcolo di attributi ereditati mediante marker



- Si aggiunge una regola vuota (marker) che fa in modo che la regola lid::=ID sia sempre precedute da un valore semantico di tipo
  - Il marker sposta perciò un valore semantico nella posizione desiderata
- Si fa notare che, per avere una gestione semantica meno complessa, conviene utilizzare sempre liste ricorsive sinistre:
  - lid ::= ID | lid VIR ID;
  - Nonostante l'utilizzo di liste ricorsive sinistre, spesso è inevitabile l'utilizzo dei marker

Esempio:
Calcolo degli attributi ereditati mediante marker

```
lid ::= ID:name {:
      RESULT = (String) parser.stack(-1);
      add_id(name, RESULT);
:} ;

lid ::= ID:name VIR Empty lid {:
      RESULT = (String) parser.stack(-1);
      add_id(name, RESULT);
:} ;

Empty ::= {:
      RESULT = (String) parser.stack(-2);
:} ;
```

Azioni intermedie

- Si può evitare di introdurre esplicitamente un non terminale riscritto come stringa nulla, utilizzando, nella parte destra delle regole, le **azioni intermedie**.
- Esse vengono automaticamente sostituite da CUP con un simbolo non terminale, a sua volta riscritto come ε.
- Il codice relativo alle regole:

```
lid ::= ID:name VIR Empty lid ;
Empty ::= ;
```

- Potrà essere così riscritto:

```
lid ::= ID:name VIR {:
        RESULT = (String) parser.stack(-2);
    :}
    lid {:
        RESULT = (String) parser.stack(-1);
        add_id(name, RESULT);
    :} ;
```


Esempio marker

scanner.jflex

```
import java_cup.runtime.*;
%%
%cup
%unicode

nl    = \n | \r | \r\n
id    = [a-zA-Z][a-zA-Z0-9_]*
tipo  = int | float | char | double

%%

","             { return new Symbol(sym.VIR);}
";"             { return new Symbol(sym.PV);}

{tipo}          { return new Symbol( sym.TYPE, new String(yytext()) ); }

{id}            { return new Symbol(sym.ID, new String(yytext()) ); }

{nl} | " " | \t     { ; }
```


Esempio marker

parser.cup

```
import java_cup.runtime.*;

parser code {:
    // Return semantic value of symbol in position (position)
    public Object stack(int position) {
        return (((Symbol)stack.elementAt(tos+position)).value);
    }
};

terminal VIR, PV;
terminal String TYPE, ID;
non terminal goal, lista_dich;
non terminal String dich, lid;

start with goal;

goal ::= lista_dich    {: System.out.println("PARSER: GRAMMATICA RICONOSCIUTA!!");
};

lista_dich ::= | lista_dich dich;
```


Esempio marker

parser.cup

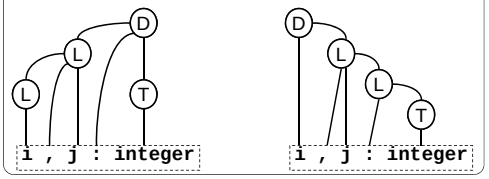
```
dich ::= TYPE lid:x PV {:
    System.out.println("PARSER: Riconosciuta dichiarazione di tipo: " + x);
:};

lid ::= ID:name VIR {:
        RESULT = (String) parser.stack(-2);
    :}
    lid {:
        RESULT = (String) parser.stack(-1);
        System.out.println("PARSER: var(" + name + ", " + RESULT + ")");
:} ;


lid ::= ID:name {:
        RESULT = (String) parser.stack(-1);
        System.out.println("PARSER: var(" + name + ", " + RESULT + ")");
:} ;
```


Trasformazione della grammatica

- È possibile evitare l'uso degli attributi ereditati trasformando la grammatica.

D → L ':' T
T → **integer** | **real**
L → L ',' **id** | **id**

D → **id** L
L → ',' **id** L | ':' T
T → **integer** | **real**



Gestione degli errori semantici

- La verifica degli errori semantici viene di solito effettuata all'interno delle azioni associate alle regole
- Di solito vengono verificati:
  - La giusta corrispondenza tra gli operandi
  - Se le variabili e le funzioni sono state dichiarate
  - La coerenza tra i prototipi delle funzioni e i parametri passati quando vengono utilizzate

Generazione di codice intermedio per i costrutti di tipo WHILE

- Come esempio di generazione di codice intermedio, ipotiziamo di voler trasformare il seguente codice che realizza un ciclo WHILE:

```
while_c ::= WHILE ( a > 0) { /* something */ }
                  cond      stmt
```

- Nel seguente codice intermedio:

```
L0:   EVAL cond
      GOTOF L1
      stmt
      GOTO L0
L1:
```

- Dove GOTOF è un'istruzione di salto incondizionato eseguita solo se il risultato della precedente operazione di EVAL è 0 (cioè FALSE)
- L0 e L1 sono delle etichette

Generazione di codice intermedio per i costrutti di tipo WHILE

- Una possibile soluzione al problema esposto è la seguente (che richiede l'utilizzo degli attributi ereditati:

```
wc ::= WHILE cond NT0:x stmt {: Integer[] l = x;
                                System.out.println( "GOTO L" +l[0]);
                                System.out.print( "L"+l[1]+":"); :};
NT0 ::= {: RESULT = new Integer[2];
        RESULT[0] = genLabel(); //L0:
        RESULT[1] = genLabel(); //L1:
        System.out.print( "L"+RESULT[0]+":");
        System.out.println( "EVAL"+parser.stack(0));
        System.out.println( "GOTOF L"+RESULT[1]); :};
```