POLITECNICO DI TORINO

Laboratorio di Compilatori

Corso di Linguaggi e Traduttori

Esercitazione 3

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

a.a 2010 / 2011

Uso avanzato di Cup

- Grammatiche ambigue
- Le liste
- Precedenze degli operatori
- Gestione degli errori sintattici

Ambiguità e conflitti in Cup

- Se la grammatica è ambigua possono verificarsi dei conflitti.
- Ciò significa che l'analizzatore deve scegliere tra più azioni alternative plausibili.
- Il problema viene, di solito, risolto modificando la grammatica per renderla non ambigua oppure fornendo indicazioni a Cup su come comportarsi in caso di ambiguità.
- La seconda ipotesi richiede una comprensione adeguata dell'algoritmo di analisi, per evitare di generare comportamenti scorretti.

Grammatica Ambigua

- Una grammatica si dice ambigua se esiste almeno una sequenza di simboli del linguaggio per cui esistono due o più alberi di derivazione distinti.
- Esercizio: trovare gli alberi di derivazione per
  `if (i==1) if (j=2) a=0; else a=1;`

data la grammatica:

- S ::= M ;
- M ::= 'if' C M ;
- M ::= 'if' C M 'else' M ;
- M ::= ID '=' NUM ';' | VAR '=' VAR ';' ;
- C ::= '(' ID '==' NUM ')' ;

Esempio grammatica non ambigua: Costrutto if-then-else

- La grammatica per i costrutti del tipo if-then-else può essere resa non ambigua come segue:
  - S ::= M | U ;
  - U ::= 'if' C S ;
  - U ::= 'if' C M 'else' U ;
  - M ::= 'if' C M 'else' M ;
  - M ::= VAR '=' NUM ';' | VAR '=' VAR ';' ;
  - C ::= '(' VAR '==' NUM ')' ;
- **`if (i==1) if (j=2) a=0; else a=1;`**

Esempio grammatica non ambigua: Espressioni algebriche

- Una grammatica non ambigua per descrive le espressioni algebriche è la seguente:

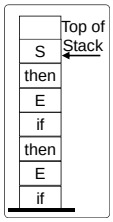
```
S  ::=  E
E  ::=  E '+' T
E  ::=  E '-' T
E  ::=  T
T  ::=  T '*' F
T  ::=  T '/' F
T  ::=  F
F  ::=  '(' E ')'
F  ::=  NUM
```

- I simboli T e F della grammatica algebrica servono a togliere l'ambiguità sulla priorità degli operatori '+' e '-' rispetto agli operatori '*' e '/'.

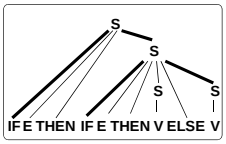
Ambiguità e conflitti in Cup: Conflitti shift-reduce (I)

1) S ::= if E then S
2) S ::= if E then S else S
3) S ::= V

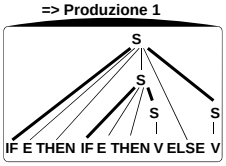
- Input: IF E THEN IF E THEN V (*) ELSE V
- Il prossimo simbolo in ingresso è '**ELSE**'
- 2 possibili azioni:



- **Fare lo SHIFT di 'else' nello Stack => Produzione 2**



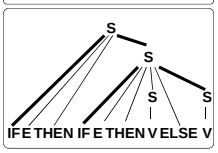
- **RIDURRE le prime 4 voci dello Stack => Produzione 1**



Ambiguità e conflitti in Cup: Conflitti shift-reduce (II)

1) S ::= if E then S
2) S ::= if E then S else S
3) S ::= V

Input
IF E THEN IF E THEN V ELSE V



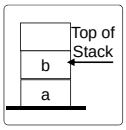
*** Shift/Reduce conflict found in state #8
between S ::= IF E THEN S (*)
and S ::= IF E THEN S (*) ELSE S
under symbol ELSE

Resolved in favor of shifting.

Cup decide di eseguire lo ***shift***.

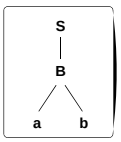
Ambiguità e conflitti in Cup: Conflitti reduce-reduce (I)

Input
a b

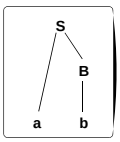


1) S ::= a B
2) S ::= B
3) B ::= a b
4) B ::= b

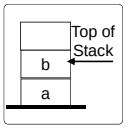
- Il prossimo simbolo in ingresso è '$'
- 2 possibili azioni:
  - **RIDURRE le prime 2 voci dello Stack => Produzione 3**



  - **RIDURRE la prima voce dello Stack => Produzione 4**



Ambiguità e conflitti in Cup: Conflitti reduce-reduce (II)



1) S ::= a B
2) S ::= B
3) B ::= a b
4) B ::= b

*** Reduce/Reduce conflict found in state #7
between B ::= b (*)
and B ::= a b (*)
under symbols: {EOF}

Resolved in favor of the second production.

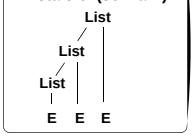
Cup decide di ridurre la regola che è stata definita per prima (3) .

Le Liste (I)

- Una lista, eventualmente vuota, di elementi
- Esempi di liste:
  - Lista di almeno un elemento E, separati da virgola V:

List ::= List E | E ; //senza V
List ::= List V E | E ;

Albero derivazione Lista 3 el (senza V)

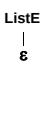


  - Lista di elementi, eventualmente vuota (primo esempio):

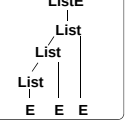
ListE ::= ε | List ;
List ::= List E | E ;

Albero derivazione

Lista vuota



Lista 3 el



Le Liste (II)

- Esempi di liste:
  - Lista di elementi, eventualmente vuota (secondo esempio):

List ::= List E | ε ;

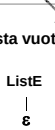
  - Lista di elementi, eventualmente vuota (esempio SBAGLIATO):

List ::= List E | E | ε ;

Stessa sequenza di simboli in ingresso, 2 alberi di derivazione differenti => GRAMMATICA AMBIGUA

Albero derivazione

Lista vuota



Lista 3 el (I)

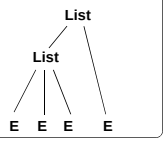
Lista 3 el (II)

Le Liste (III)

- Esempi di liste:
  - Lista di almeno 3 elementi:

List ::= List E | E E E ;

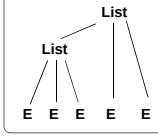
Albero derivazione Lista 4 el



  - Lista di almeno 3 elementi in numero dispari:

List ::= List E E | E E E;

Albero derivazione Lista 5 el



Sezione delle Precedenze: Grammatiche volutamente ambigue

- In certi casi la grammatica può essere resa volutamente ambigua al fine di limitare il numero delle regole.
- È necessario però fornire indicazioni sulla risoluzione delle ambiguità.
- Un caso tipico è dato dalle espressioni algebriche:

```
Grammatica non ambigua
S ::= E
E ::= E '+' T
E ::= E '-' T
E ::= T
T ::= T '*' F
T ::= T '/' F
T ::= F
F ::= '(' E ')'
F ::= NUM
```

```
Grammatica Fortemente
 ambigua
E ::= E '+' E
E ::= E '-' E
E ::= E '*' E
E ::= E '/' E
E ::= '(' E ')'
E ::= NUM
```


Associatività

- Operatore associativo a sinistra (E ::= E '+' E)
  - 1+2+3+4 → 3+3+4 → 6+4 → 10
- Operatore associativo a destra (E ::= E '+' E)
  - 1+2+3+4 → 1+2+7 → 1+9 → 10
- L'operatore '=' in un assegnazione è un operatore associativo a destra:
  - a = b = 3
  - Un altro operatore associativo a destra è l'elevazione a potenza
  - 3^2^2 → 3^4 → 81

Sezione delle Precedenze: Operatori

1) E → E '+' E
2) E → E '*' E
3) E → '(' E ')'
4) E → NUM

- La regola 1 (così come la 2) è ambigua in quanto non specifica l'associatività dell'operatore '+' ('*').
- Inoltre le regole 1 e 2 non specificano la precedenza tra gli operatori '+' e '*'.
- E' possibile suggerire a Cup come comportarsi aggiungendo come risolvere precedenze e associatività nella sezione delle precedenze.
- La parola chiave precedence left introduce un operatore associativo a sinistra, precedence right introduce un operatore associativo a destra, precedence nonassoc introduce un operatore non associativo.
- L'ordine con cui gli operatori sono dichiarati è inverso alla loro priorità.

Sezione delle Precedenze: Regole di risoluzione dell'ambiguità

- Ad ogni regola che contiene almeno un terminale definito come operatore, Cup associa la precedenza e l'associatività dell'operatore più a destra.
- Se la regola è seguita dalla parola chiave **%prec**, la precedenza e l'associatività sono quelle dell'operatore specificato.
- In caso di conflitto shift-reduce, viene favorita l'azione della regola con la precedenza maggiore.
- Se la precedenza è la stessa, si usa l'associatività: sinistra implica reduce, destra shift.

Sezione delle Precedenze: Esempio

```
terminal uminus;

precedence left '+', '-'; /* Bassa priorità */
precedence left '*', '/';
precedence left uminus;   /* Alta priorità */

start with E;

E ::=  E '+' E
     | E '-' E
     | E '*' E
     | E '/' E
     | '-' E    %prec uminus
     | '(' E ')'
     | NUM
;
```

Codice utente (I)

- Esistono alcune direttive che permettono di inserire codice utente direttamente nel parser
- Servono per:
  - Personalizzare il comportamento del parser
  - Aggiungere codice direttamente all'interno della classe che realizza il parser
  - Utilizzare uno scanner diverso da quello di default (JFlex)
- Sono:
  - init with {: … :}
    - Il codice viene eseguito prima di ogni chiamata allo scanner, quindi prima che qualsiasi simbolo terminale venga fornito al parser
    - Utilizzata per inizializzare variabili o per inizializzare lo scanner nel caso non si utilizzi JFlex

Codice utente (II)

- scan with {: … :}
  - Indica al parser quale procedura utilizzare per richiedere il successivo terminale allo scanner
  - Deve restituire un oggetto di tipo java_cup.runtime.Symbol
  - Anche questa direttiva serve per l'utilizzo di scanner diversi da quello di default (JFlex)
  - scan with {: return scanner.next_token(); :};
- Quando CUP genera il file java che realizza il parser, al suo interno vengono generate due classi:
  - public class parser extends java_cup.runtime.lr_parser
    - parser è la classe che realizza il parser ed eredita diversi metodi dalla classe java_cup.runtime.lr_parser
  - class CUP$parser$actions
    - CUP$parser$actions è la classe in cui vengono riportate le regole grammaticali del parser in sintassi java e le relative azioni semantiche (codice java associato ad ogni regola)

Codice utente (III)

- La classe java_cup.runtime.lr_parser è realizzata nel file java_cup/runtime/lr_parser.java che si trova nella directory di installazione di CUP
- parser code {: … :}
  - Il codice riportato all'interno viene ricopiato nella classe parser
  - Dichiarazioni di metodi e classi visibili da altri oggetti java ma soprattutto per eseguire l'overriding di funzioni del parser (Ad es. riscrivere le funzioni di gestione degli errori)
- action code {: … :}
  - Il codice riportato all'interno viene ricopiato nella classe CUP$parser$actions
  - Tale codice è accessibile solo all'interno delle azioni semantiche associate alle regole grammaticali
  - Procedure e variabili da utilizzare all'interno delle azioni associate alle regole della grammatica

Errori: stampa linea e colonna

scanner.flex

```
import java_cup.runtime.*;
…
%%
%cup
%line
%column

%{
   private Symbol symbol(int type){
         return new Symbol(type, yyline, yycolumn);
   }
   private Symbol symbol(int type, Object value){        //Per la gestione semantica
         return new Symbol(type, yyline,yycolumn,value);
   }
%}
…
%%
[a-z]          {return symbol(sym.EL); }
','            {return symbol(sym.VIRGOLA); }
```

Costuttori di Symbol:
public Symbol(int sym_id)
public Symbol(int sym_id, int left, int right)
public Symbol(int sym_id, Object o)
public Symbol(int sym_id, int left, int right, Object o)

Errori: stampa linea e colonna

parser.cup

```
import java_cup.runtime.*;

parser code {:
   public void report_error(String message, Object info) {
      StringBuffer m = new StringBuffer(message);
      if (info instanceof Symbol) {
         if (((Symbol)info).left != -1 && ((Symbol)info).right != -1) {
            int line = (((Symbol)info).left)+1;
            int column = (((Symbol)info).right)+1;
            m.append(" (linea "+line+", colonna "+column+")");
         }
      }
      System.err.println(m);
   }
:}
```


Gestione degli errori sintattici (I)

- In genere quando un parser incontra un errore non dovrebbe terminare brutalmente l'esecuzione
  - Un compilatore in genere cerca di provvedere alla situazione per poter analizzare il resto del file, in modo da segnalare il maggior numero possibile di errori
- Per default, un parser generato da CUP, quando incontra un errore:
  - Segnala chiamando la funzione public void syntax_error(Symbol cur_token) definita nella classe java_cup.runtime.lr_parser un errore sintattico scrivendo a schermo "Syntax error".
  - Se tale errore non viene recuperato dal parser attraveso un opportuno simbolo error, il parser lo segnala richiamando la funzione public void unrecovered_syntax_error(Symbol cur_token), anch'essa definita in java_cup.runtime.lr_parser. Tale funzione, dopo aver segnalato l'errore "Couldn't repair and continue parse" blocca l'esecuzione del parser.

Gestione degli errori sintattici (II)

Analizzando le due funzioni:

- public void syntax_error(Symbol cur_token)
  - Chiama la funzione report_error con i seguenti parametri report_error("Syntax error", cur_token);
    - Dove cur_token è l'attuale simbolo di looahead in cui si è verificato l'errore
- public void unrecovered_syntax_error(Symbol cur_token)
  - Chiama la funzione report_fatal_error, con i seguenti paramenti report_fatal_error("Couldn't repair and continue parse", cur_token);
  - La funzione report_fatal_error chiama la funzione report_error con gli stessi parametri e lancia un'eccezione che provoca la fine del parsing
- Un'opportuna redifinizione delle funzioni precedentemente elencate in parser code {: … :}, permette di personalizzare la gestione degli errori

Simbolo predefinito 'error'

- Il simbolo predefinito '**`error`**' indica una condizione di errore. Esso può essere usato all'interno della grammatica per consentire al parser di riprendere l'esecuzione dopo un eventuale errore.

Esempio:

```
ass ::= ID UGUALE exp PV
        |ID UGUALE error PV
;
```


Come viene gestito il simbolo 'error' da Cup?

- Quando il parser generato da Cup incontra un errore, comincia a svuotare lo stack fino a che incontra uno stato in cui il simbolo '**`error`**' è lecito.
  - Nell'esempio di prima viene svuotato lo stack da exp scorrette (cioè da quella sequenza di simboli che non riesce a ridursi in exp), fino a quando in cima allo stack viene trovato il simbolo UGUALE
- Fa lo *shift* del simbolo **`error`**.
- Se il simbolo successivo è accettabile il parser procede nell'analisi.
- Altrimenti il parser continua a leggere e scartare simboli finché non ne trova uno accettabile
  - Nell'esempio precedente il parser leggerà e scarterà simboli finché non trova un simbolo di PV

Alcune regole generali

- Una semplice strategia per la gestione degli errori è quella di saltare lo *statement* corrente:

```
stmt ::= error ';'
```

- A volte può essere utile recuperare un delimitatore di chiusura corrispondente ad uno di apertura:

```
expr    ::= '(' expr ')'
          | '(' error ')'
```

- NB: Per limitare la proliferazione di errori spuri, dopo il riconoscimento di un errore, la segnalazione è sospesa finché non vengono *shiftati* almeno tre simboli consecutivi

Grammatica

```
file   ::= funcs
;

funcs  ::= /* empty */
       | funcs func
;

func   ::= ID '(' ')'
           compound
;

compound ::= '{' stmts '}'
;
```

```
stmts  :: /* empty */
       | stmts stmt
;

stmt   ::= exp ';'
    | compound
;

exp    ::= NUM
    | exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | exp '/' exp
    | '-' exp %prec uminus
    | '(' exp ')'
;
```


Statement ed espressioni

```
stmt ::= exp ';'
       | compound
       | error ';'    {: System.err.println("Syntax error in statement"); :}
;

compound ::= '{' stmts '}'
           | '{' stmts error '}'  {: System.err.println("Missing ; before '}'"); :}
           | '{' error '}'  {: System.err.println("Missing ; before '}'"); :}
;

exp ::= …
       | '(' error ')'  {: System.err.println("Syntax error in expression\n"); :}
```