Dispensa YACC

1.1 YACC: generalità

Il tool Yacc (acronimo per *Yet Another Compiler Compiler*) è uno strumento software che a partire da una specifica grammaticale context free di un linguaggio scritta in un'appropriata sintassi genera automaticamente un parser LALR del linguaggio. Più precisamente, poiché alle produzioni della grammatica possono essere associate delle "azioni" (cioè frammenti di programmi scritti in linguaggio C), quello che Yacc genera è un traduttore input/output che oltre a fare l'analisi sintattica di costrutti input ne fa anche la traslazione in un determinato output come specificato dalle azioni.

L'architettura software che mostra il contesto d'uso di Yacc è la seguente. Nell'architettura i nomi dei files generati (*y.tab.c* e *a.out*) sono dipendenti da sistema, per esempio in questo caso si è considerato il sistema Unix.

<nome_file>.y ⇒ Yacc ⇒ y.tab.c

y.tab.c ⇒ Compilatore C ⇒ a.out

input ⇒ a.out ⇒ output

Nota: Invocando Yacc con l'opzione −v viene generato un file contenente una rappresentazione testuale degli items calcolati in accordo al metodo LALR. Compilando il file y.tab.c con l'opzione −ly viene caricato da libreria il parsing program LR.

Come si può notare dall'architettura, l'utente si deve concentrare essenzialmente sulla specifica sorgente Yacc (il file .y in cui è scritta la grammatica), tutto il resto, infatti, è implementato poi automaticamente. Il formato generale di un programma sorgente Yacc è costituito da tre sezioni ed ha la seguente struttura:

```
DICHIARAZIONI
%%
REGOLE DI TRASLAZIONE
%%
PROGRAMMI
```

La sezione "REGOLE DI TRASLAZIONE" è il cuore della specifica Yacc in quanto è in essa che è scritta la grammatica. I simboli speciali %% ne delimitano l'inizio e la fine. Nel seguito descriviamo queste tre sezioni.

1.2 La sezione "DICHIARAZIONI": sintassi

Questa sezione consiste di due parti:

- Nella prima parte vi è tutto il blocco di dichiarazioni C. Sostanzialmente vi vengono inserite le normali dichiarazioni di variabili che saranno usate nelle sezioni delle Regole e dei Programmi, e in

più vi possono essere direttive di compilazione del tipo #define oppure #include. Questa parte è delimitata dai simboli speciali %{ e %}

- Nella seconda parte si dichiarano i token della grammatica attraverso la direttiva Yacc %token seguita dalla lista dei token. (Si noti che nella sintassi Yacc è convenzione usare lettere maiuscole per i token e lettere minuscole per i non-terminali, al contrario di quanto si fa normalmente nella stesura di una grammatica)

Esempio: %token ID DIGIT

1.3 La sezione "REGOLE DI TRASLAZIONE": sintassi

In questa sezione vengono scritte le produzioni della grammatica. La sintassi è la seguente:

una produzione: <ParteSinistra> → <ParteDestra>
in formato Yacc è scritta così: <ParteSinistra> : <ParteDestra> ;

All'interno di una produzione, Yacc considera come terminali o stringhe di caratteri dichiarate come token nella sezione Dichiarazioni (es. ID oppure DIGIT) oppure singoli caratteri racchiusi tra apici (es. '+'). Questi ultimi tipi di terminali il parser se li aspetta così come sono nell'input (si pensi appunto al simbolo + che apparirà proprio così in una stringa sorgente del compilatore); invece i valori dei terminali dichiarati come token (si pensi ad esempio al token ID che non appare così nel sorgente ma attraverso i suoi valori, cioè i lessemi) saranno passati al parser dall'analizzatore lessicale.
Ogni stringa di caratteri che appare nelle produzioni e che non è stata dichiarata come token, viene assunta da Yacc essere un non-terminale.

Esempio:
la produzione: E → E + digit
in formato Yacc diventa: e : e '+' DIGIT ;

In essa DIGIT viene considerato token (se supponiamo che ce ne sia la dichiarazione nella prima sezione), + viene considerato come terminale (essendo un singolo carattere racchiuso tra apici), infine e viene considerato per esclusione un non-terminale.

Nota: per comunicare a Yacc chi è lo start symbol, ossia il non-terminale di partenza della grammatica, si può inserire la dichiarazione %start seguita dal nome del non-terminale nella seconda parte della sezione Dichiarazioni oppure, se questa dichiarazione si omette, Yacc assume per default come start symbol il non-terminale sinistro della prima produzione listata nella sezione delle Regole.

1.4 Azioni Semantiche

Il formato generale di una produzione Yacc è il seguente:

<ParteSinistra> : <ParteDestra> { AzioneSemantica} ;

dove un'azione semantica è una sequenza di istruzioni scritte in linguaggio C e racchiuse tra parentesi graffe.
Nelle produzioni Yacc i simboli grammaticali hanno attributi semantici, cioè valori, e le azioni servono proprio a gestire questi valori. Yacc usa la pseudo-variabile $$ per far riferimento al valore dell'attributo semantico del non-terminale sinistro di una produzione, e le pseudo-variabili $i per il valore dell'attributo dell'i-esimo simbolo grammaticale della parte destra (per es. $3 si riferisce al valore dell'attributo del terzo simbolo grammaticale nella parte destra).

Le azioni semantiche nelle produzioni Yacc oltre a contenere specifiche routines che implementano la particolare traduzione input/output richiesta, servono in particolare a calcolare $$ in funzione dei $i. Ad esempio, la produzione che specifica il costrutto dell'addizione vista prima può essere completata dalla seguente azione che permette di calcolare il valore parziale del risultato:

```
e : e '+' DIGIT { $$ = $1 + $3;};
```

In definitiva, i valori degli attributi dei non-terminali sono computati dalle azioni semantiche, mentre i valori degli attributi dei token (cioè i lessemi) sono comunicati a Yacc dall'analizzatore lessicale attraverso la pseudo-variabile yylval, come vedremo nel paragrafo 1.5.

Va anche considerato che:
- Se un'azione è inserita alla fine della parte destra di una produzione, allora essa sarà eseguita quando, durante il parsing, quella produzione sarà ridotta.
- Le azioni possono essere inserite anche all'interno della parte destra di produzioni; è chiaro che in questo caso nell'azione il computo di $$ può far riferimento solo ai $i dei simboli grammaticali alla sinistra dell'azione.
- Yacc ha un'azione di default che è { $$ = $1;}. Pertanto se si scrive una produzione senza azioni semantiche, comunque Yacc esegue la sua azione di default quando il parser ridurrà quella produzione.

1.5 La sezione "PROGRAMMI": sintassi

In questa sezione vi sono tutte le routines di supporto utili al corretto funzionamento del parser. In particolare, le tre routines più importanti e che devono necessariamente essere presenti in questa sezione sono: l'analizzatore lessicale yylex(), la funzione principale main() e la funzione di gestione errori yyerror().

- L'analizzatore lessicale

Yacc si aspetta un analizzatore lessicale di nome yylex() che fa lo scanning dell'input e ritorna al parser le tipologie di token dichiarati nella sezione Dichiarazioni comunicandone il valore (il lessema) tramite la variabile yylval. Pertanto il nucleo fondamentale di questa routine in un programma Yacc sarà sempre la coppia di istruzioni:

```
…
yylval = …
return (…)
…
```

L'analizzatore lessicale può essere implementato in due possibili modi: o si scrive esplicitamente all'interno della sezione Programmi una procedura yylex() in C oppure si usa un programma

separato Lex per generare l'analizzatore lessicale, e in questo caso si dovrà inserire nella prima parte della sezione Dichiarazioni del programma Yacc la direttiva #include "lex.yy.c" per stabilire l'aggancio fra Yacc e Lex.

- La funzione main()

La funzione principale main() fa partire il procedimento di lettura, parsing e traduzione I/O di un input. All'interno del programma Yacc si può usare semplicemente:

```
main() {
      yyparse(); }
```

dove yyparse() è il parser generato da Yacc che invoca ripetutamente yylex() innescando l'interazione parser – analizzatore lessicale.

- La funzione yyerror()

La sezione Programmi deve contenere una funzione che gestisce eventuali errori sintattici. Per semplicità, può essere usata la seguente funzione di base provvista da Yacc che stampa "syntax error" se viene riscontrato un errore:

```
yyerror(char *s) {
     { fprintf(stderr, "%s\n", s); }
```

1.6 I costrutti union e return(0)

Nel seguito accenniamo a due importanti problematiche: la gestione dei tipi di dato degli attributi semantici dei simboli grammaticali e la gestione della fine dell'input.

- il costrutto union

Yacc assume per default che i valori degli attributi dei simboli grammaticali (e quindi il tipo della variabile esterna yylval e dei $i e $$) siano di tipo integer. Pertanto nel caso i tipi siano interi non bisogna fare nulla, altrimenti la gestione di tipi diversi dagli interi è possibile grazie al costrutto %union { … } che va usato nella seconda parte della sezione Dichiarazioni e in cui si possono definire nuovi tipi di dato usando la sintassi C (es. char tipo).

- Quando si usa la 'union', i token che sono del nuovo tipo di dato vanno dichiarati nella sezione Dichiarazioni nella nuova forma:
%token <tipo> TOKEN1 TOKEN2 …
- Anche il tipo di un non-terminale, se usato, va dichiarato con una direttiva (sempre nella sezione Dichiarazioni) della forma:
%type <tipo> NONTERMINALE1 NONTERMINALE2 …

Nota importante: Se un token ha attributi di tipo non intero (ossia è stata usata la 'union'), allora l'analizzatore lessicale ne dovrà restituire il valore attraverso l'estensione yylval.tipo

- *il costrutto return(0)*

Per terminare correttamente la sua esecuzione (ossia l'interazione parsing – analisi lessicale fra `yyparse()` e `yylex()`), Yacc si aspetta l'istruzione `return(0)` che indichi quando l'input è stato completamente esaminato. Infatti, nella sintassi Yacc '0' è un token riservato corrispondente all'end-marker $. Per semplicità, si può assumere che questa istruzione sia posizionata come ultima azione a destra della produzione iniziale nella sezione Regole in modo che essa sia sicuramente eseguita a fine del processo di parsing.

1.7 Yacc e grammatiche ambigue

Yacc risolve i conflitti usando le seguenti due euristiche di default:

1) Un conflitto *reduce/reduce* è risolto scegliendo la produzione conflittuante listata prima nella specifica Yacc
2) Un conflitto *shift/reduce* è risolto in favore dello shift

La prima regola dà potere al programmatore, il quale dopo aver studiato la semantica delle azioni conflittuanti organizza appropriatamente le produzioni nell'ordine desiderato, la seconda regola è adatta a risolvere correttamente l'ambiguità in molti costrutti pratici quali ad esempio il *dangling else*.
Se queste regole non sono ciò che il programmatore vuole, Yacc offre un meccanismo più generale per risolvere i conflitti *shift/reduce* assegnando associatività e precedenze ai terminali e alle produzioni della grammatica, e quindi scegliendo tra le azioni conflittuanti quella con precedenza più alta.

- *Associatività ai terminali*

Per assegnare associatività ai terminali della grammatica si usano le direttive Yacc (nella seconda parte della sezione Dichiarazioni):

`%left`	assegna associatività sinistra
`%right`	assegna associatività destra
`%nonassoc`	forza un operatore a essere non associativo (ossia è scorretta la sua combinazione)

- *Precedenze ai terminali*

Per assegnare precedenze ai terminali della grammatica si usano queste regole:

i) Quando assegniamo le associatività si ha che tutti i terminali sulla stessa linea hanno lo stesso livello di precedenza
ii) Le linee sono listate in ordine di precedenza maggiore dal basso

Per esempio, se si considera il seguente frammento di programma Yacc:

```
…
%left A B
%right C
%left D
…
```

possiamo dedurre che i token A, B e D sono associativi sinistri mentre il token C è associativo destro; inoltre D è il token con precedenza più alta, poi C e poi A e B che hanno la stessa precedenza.

- *Associatività e precedenze alle produzioni*

Per default Yacc assegna a ogni produzione della grammatica una associatività e una precedenza che sono le stesse del suo terminale più a destra. Questa regola è adeguata in molti casi pratici, tuttavia se il terminale più a destra non fornisce un livello di precedenza adatto la regola si può forzare attaccando nella specifica a fine produzione la direttiva Yacc:

```
%prec <nome terminale>
```

in modo da assegnare alla produzione lo stesso livello di precedenza e la stessa associatività del terminale specificato.

In conclusione, possiamo riassumere la metodologia generale che Yacc usa per risolvere i conflitti:

- Se ci sono conflitti *shift/reduce* e/o *reduce/reduce* e né i terminali né le produzioni della grammatica hanno assegnate associatività e precedenze, allora vengono usate le due euristiche Yacc di default
- Se c'è un conflitto *shift/reduce* e i terminali e le produzioni della grammatica hanno assegnate associatività e precedenze, allora il conflitto è risolto in favore dell'azione (shift o reduce) che ha precedenza più alta
- Se le precedenze sono le stesse allora si sceglie:
    - l'azione di reduce se l'associatività della produzione è sinistra
    - l'azione di shift se l'associatività della produzione è destra

  se invece la produzione è non associativa si ha errore

Esercitazioni YACC

Di seguito vengono provvisti alcuni esercizi (con possibili soluzioni) sullo sviluppo di semplici programmi Yacc. Si provi anche a risolvere qualche esercizio, senza vederne subito la soluzione. Il suggerimento per lo svolgimento di esercizi è (dopo aver ben compreso la traccia e la particolare traduzione I/O richiesta) di concentrarsi anzitutto sulla specifica grammaticale, codificando in formato Yacc le produzioni della grammatica, poi sulle azioni semantiche da inserire nelle produzioni per implementare la particolare traduzione I/O richiesta, poi sull'analizzatore lessicale (si noti ad esempio l'uso della funzione `getchar()`, funzione C che legge un carattere alla volta da standard input, per implementare lo scanning dell'input all'interno dell'analizzatore lessicale scritto in C) e le altre routines della sezione Programmi sulla base degli spunti forniti nelle precedenti sezioni, e solo alla fine sulla parte Dichiarazioni quando si ha una visione completa di tutte le dichiarazioni e le direttive che servono (si noti ad esempio l'uso delle librerie `<ctype.h>` e `<stdio.h>` che contengono la funzione `printf`, il predicato `isdigit` etc.)

Esempio introduttivo

Implementare una piccola calcolatrice di espressioni aritmetiche su digit che legge un'espressione, la valuta e ne stampa il valore numerico, a partire dalla seguente grammatica sorgente G = {N, T, S, P}:

N = {S, E, T, F}
T = {digit, (,), +, *}
S = S
P = {
S → E
E → E + T
E → T
T → T * F
T → F
F → (E)
F → digit
}

Possibile soluzione:

```
%{ #include <ctype.h>
   #include <stdio.h>
%}
%token DIGIT
%start s
%%
s : e {printf("%d", $1); return(0);};
e : e '+' t {$$=$1+ $3;};
```

```
e : t {$$=$1;};        /* nota: quest'azione potrebbe essere anche omessa perché è quella di default
t : t '*' f {$$=$1*$3;};
t : f {$$=$1;};
f : '(' e ')' {$$=$2;};
f : DIGIT {$$=$1;};
%%
yylex()
{ int c;
c=getchar();
if isdigit(c)
   {yylval=c;
    return(DIGIT); }
else return(c); }
yyerror(char *s)  {
      { fprintf(stderr, "%s\n", s); }
main()
       { yyparse(); }
```

Esercizio 1 (estratto da traccia d'esame)
Si consideri la seguente grammatica G = {N, T, S, P} con:

N = {S, X}
T = {(,), pippo}
S = S
P = {
S → (X)
X → X pippo
X → pippo
}

dove 'pippo' è un token con lessemi in {a, b}.
Scrivere un programma YACC per generare un traduttore I/O che stampa il numero di 'a' presenti nell'input.

Esempio:
Input: (abaa)
Output: 3

Possibile soluzione:

```
%{ #include <ctype.h>
    #include <stdio.h>
```

```
    int contatore;
%}
%union {char tipo;}
%token <tipo> PIPPO
%start s
%%
s : '(' x ')' {printf("%d", contatore); return(0);};
x : x PIPPO {if $2=='a'  contatore=contatore+ 1;};
x : PIPPO {if $1=='a'  contatore=contatore+ 1;};
%%
yylex()
{ char c;
c=getchar();
if (c=='a' || c=='b')
   {yylval.tipo=c;
    return(PIPPO); }
else return(c); }
yyerror(char *s)  {
      { fprintf(stderr, "%s\n", s); }
main()
       { contatore=0;
         yyparse(); }
```

Se si preferisce usare Lex per generare l'analizzatore lessicale, allora un possibile sorgente Lex per questo esercizio è il seguente:

```
%%
[ab]    { yylval.tipo=yytext[0];
          return(PIPPO); }
.|\n     { return(yytext[0]; }
```

In questo caso, nel programma Yacc precedente non deve essere inserita la routine yylex() e nella prima parte della sezione Dichiarazioni deve essere inserita la direttiva #include "lex.yy.c"

Esercizio 2 (estratto da traccia d'esame)

Si consideri la seguente grammatica context free il cui linguaggio consiste di stringhe tra parentesi che rappresentano semplici addizioni o sottrazioni tra digit:

N = {S, A}

T = {(,), *op* , *digit*}

S = S

P = {

$S \rightarrow (A\ op\ A)$

A → *digit*

}

dove *digit* è un token con lessemi in {0, 1, 2, ... , 9} e *op* è un token con lessemi in {+, -}

Scrivere un programma YACC per generare un traduttore input-output che stampa in output il risultato dell'operazione specificata nella stringa input.

Esempio:
Input: (6+3)
Output: 9

Possibile soluzione:

```
%{ #include <ctype.h>
    #include <stdio.h>
%}
%union {char tipo;}
%token DIGIT
%token <tipo> OP
%start s
%%
s : '(' a OP a ')' {if $3=='+' printf("%d", $2+$4);
                        else printf("%d", $2-$4); return(0);};
a : DIGIT {$$=$1;};
%%
yylex()
{ char c;
c=getchar();
if isdigit(c)
  {yylval=c;
   return(DIGIT); }
else if (c=='+'||c=='-')
   {yylval.tipo=c;
    return(OP); }
else return(c); }
yyerror(char *s)  {
      { fprintf(stderr, "%s\n", s); }
main()
       { yyparse(); }
```

Se si preferisce usare Lex per generare l'analizzatore lessicale, allora un possibile sorgente Lex per questo esercizio è il seguente:

```
%%
[0-9]    { yylval=yytext[0];
           return(DIGIT); }
[+\-]    { yylval.tipo=yytext[0];
           return(OP); }
.|\n     { return(yytext[0]; }
```

In questo caso, nel programma Yacc precedente non deve essere inserita la routine yylex() e nella prima parte della sezione Dichiarazioni deve essere inserita la direttiva #include "lex.yy.c"

Esercizio 3 (estratto da traccia d'esame)
Si consideri la seguente grammatica context free G = {N, T, S, P} con:

N = {S, B}
T = {**op**, **bit**}
S = S
P = {
S → **op** B
B → **bit**
}

dove **op** è un token con lessemi in {+, -} e **bit** è un token con lessemi in {0, 1}.

Scrivere un programma YACC per generare un traduttore I/O che stampa in output il bit input o il suo negato a seconda che il valore di **op** sia '+' o '-', rispettivamente.

Esempi: per l'input `+0` deve essere stampato `0`
per l'input `-1` deve essere stampato `0`

Possibile soluzione:

```
%{ #include <ctype.h>
   #include <stdio.h>
%}
%union {char tipo;}
%token BIT
%token <tipo> OP
%start s
%%
s : OP b {if $1=='+' && $2=='0' printf("0");
          if $1=='+' && $2=='1' printf("1");
          if $1=='-' && $2=='0' printf("1");
```

```
            if $1=='-' && $2=='1' printf("0"); return(0);};
b : BIT {$$=$1;};
%%
yylex()
{ char c;
c=getchar();
if (c=='0'||c=='1')
   {yylval=c;
    return(BIT); }
else if (c=='+'||c=='-')
    {yylval.tipo=c;
     return(OP); }
else return(c); }
yyerror(char *s)  {
      { fprintf(stderr, "%s\n", s); }
main()
       { yyparse(); }
```

Se si preferisce usare Lex per generare l'analizzatore lessicale, allora un possibile sorgente Lex per questo esercizio è il seguente:

```
%%
[01]     { yylval=yytext[0];
           return(BIT); }
[+\-]    { yylval.tipo=yytext[0];
           return(OP); }
.|\n     { return(yytext[0]; }
```

In questo caso, nel programma Yacc precedente non deve essere inserita la routine yylex() e nella prima parte della sezione Dichiarazioni deve essere inserita la direttiva #include "lex.yy.c"

Esercizio 4 (estratto da traccia d'esame)

Si consideri la seguente grammatica G = {N, T, S, P} con:

N = {S, A, B}

T = {**x**, **y**}

S = S

P = {

S → A B

A → **x** A

A → **x**

B → **y** B

B → **y** }

dove **x** è un token con lessemi in {0, 1, …, 9} e **y** è un token con lessemi in {a, b, …, z}.

Scrivere un programma YACC per generare un traduttore I/O che stampa il più grande tra il numero di cifre e il numero di lettere presenti nell'input.

Esempio:
Input: 233477802bduula
Output: 9

Possibile soluzione:

```
%{ #include <ctype.h>
    #include <stdio.h>
   int contatoreC;
   int contatoreL;
%}
%union {char tipo;}
%token X
%token <tipo> Y
%start s
%%
s : a b {if (contatoreC > contatoreL) printf("%d", contatoreC);
                        else printf("%d", contatoreL); return(0);};
a : X a {contatoreC = contatoreC+ 1;};
a : X   {contatoreC = contatoreC+ 1;};
b : Y b {contatoreL = contatoreL+ 1;};
b : Y   {contatoreL = contatoreL+ 1;};
%%
yylex()
{ char c;
c=getchar();
if isdigit(c)
  {yylval=c;
   return(X); }
else if isletter(c)
   {yylval.tipo=c;
    return(Y); }
else return(c); }
yyerror(char *s)  {
      { fprintf(stderr, "%s\n", s); }
main()
       { contatoreC=0;
         contatoreL=0;
          yyparse(); }
```

Se si preferisce usare Lex per generare l'analizzatore lessicale, allora un possibile sorgente Lex per questo esercizio è il seguente:

```
%%
[0-9]    { yylval=yytext[0];
           return(X); }
[a-z]    { yylval.tipo=yytext[0];
           return(Y); }
.|\n     { return(yytext[0]; }
```

In questo caso, nel programma Yacc precedente non deve essere inserita la routine yylex() e nella prima parte della sezione Dichiarazioni deve essere inserita la direttiva #include "lex.yy.c"