Introduzione/HOWTO a Lex e YACC

PowerDNS BV (bert hubert <bert@powerdns.com>)
v0.8 $Date: 2004/09/20 07:14:23 $

Questo documento cerca di aiutare chi inizia a usare Lex e YACC. Traduzione a cura di Ivan Bazzi, revisione a cura di Antonio Colombo. Per versioni aggiornate di questo documento, e per trovare altra documentazione in italiano sul software libero, visitare il sito dell' *ILDP* <http://it.ildp.org>

Indice

1 Introduzione

Benvenuto, gentile lettore.

Se avete già programmato, almeno un po', in ambiente UNIX, avrete incontrato i mistici programmi Lex & YACC, o come sono conosciuti dagli utenti di GNU/Linux di tutto il mondo, Flex & Bison, dove Flex è un'implementazione di Lex di Vern Paxson e Bison la versione GNU di YACC. Questi programmi saranno chiamati d'ora in poi Lex e YACC - le nuove versioni sono compatibili con le vecchie, quindi Flex e Bison possono essere usati per provare gli esempi qui forniti.

Questi programmi sono immensamente utili, ma come per il compilatore C, la loro pagina di manuale non spiega il linguaggio da usare con loro, e neppure come usarli. YACC è realmente impressionante se utilizzato in combinazione con Lex, ciò non di meno, la pagina di manuale di Bison non dice come integrare il codice generato da Lex con Bison.

1.1 Cosa NON è questo documento

Ci sono parecchi libri ben fatti che trattano di Lex & YACC. Siete invitati a leggerli se vi servissero maggiori informazioni. Contengono molte pi$ugrave; informazioni di quelle che si possono trovare qua. Vedere la sezione 'Letture ulteriori' in fondo a questo documento. Questo documento ha lo scopo di permettervi di iniziare a usare Lex & YACC in modo da permettervi di creare i vostri primi programmi.

La documentazione fornita insieme a Flex e BISON è eccellente ma non è un tutorial. Comunque completa molto bene il mio HOWTO. Potete trovare i riferimenti ad essa, sempre alla fine di questo documento.

Non sono per nulla un esperto di YACC/Lex. All'inizio della stesura di questo documento, avevo esattamente due giorni di esperienza. Il mio solo obiettivo è quello di rendere pi$ugrave; facili a voi quei due giorni.

Non ci si aspetti neppure che l'HOWTO possieda un buono stile YACC e Lex. Gli esempi sono stati mantenuti il pi$ugrave; possibile semplici e ci possono essere maniere migliori per scriverli. Se ne conoscete, siete pregati di comunicarmelo.

1.2 Scaricare materiale

Si noti che è possibile scaricare tutti gli esempi qui mostrati [quelli originali in inglese], che sono in formato leggibile dalle macchine. Si veda la *homepage* <http://ds9a.nl/lex-yacc> per i dettagli. Gli esempi in italiano tradotti in italiano si possono scaricare dal sito

ILDP <esempi-lex-yacc> .

1.3 Licenza

2 Cosa Lex e YACC possono fare per voi

Se usati in modo appropriato questi programmi permettono di analizzare linguaggi complessi. Questo è un grande vantaggio quando si volesse leggere un file di configurazione, o si volesse scrivere un compilatore per qualsiasi linguaggio voi (o altri) abbiate potuto inventare.

Con un po' di aiuto, che possibilmente verrà fornito da questo documento, si scoprirà che non avrete; più bisogno di scrivere un analizzatore sintattico a mano - Lex e YACC sono gli strumenti per farlo.

2.1 Che cosa fa ognuno dei programmi, di per sé

Per quanto questi programmi eccellano se usati insieme, ognuno di essi serve a uno scopo differente. Il prossimo capitolo spiegherà cosa fa ciascuno.

3 Lex

Il programma Lex genera un cosiddetto 'Lexer' o 'Analizzatore lessicale'. Questa è una funzione che ha come input un flusso di caratteri e che, quando riconosce che un gruppo di caratteri corrisponde a una chiave, esegue un'azione specificata. Un esempio molto semplice:

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

%%
stop    printf("Ricevuto comando Stop\n");
start   printf("Ricevuto comando Start\n");
%%
```

La prima sezione tra la coppia %{ e %} è inclusa direttamente nell'output del programma. È necessario inserirla perché in seguito si usa printf il quale è definito in stdio.h.

Le sezioni si separano usando '%%', quindi la prima linea della seconda sezione comincia con la chiave 'stop'. Ogni volta che viene incontrata in input la chiave 'stop' viene eseguito il resto della linea (una chiamata a printf()).

Oltre a 'stop', si è definita anche 'start', il cui comportamento è analogo a quello di 'stop'.

La sezione di codice termina ancora con '%%'.

Per compilare Esempio 1, fare questo:

```
lex esempio1.l
cc lex.yy.c -o esempio1 -ll

NOTA: se si sta usando flex, invece che lex, è probabile sia necessario cambiare '-ll'
in '-lfl' negli script di compilazione.  In RedHat 6.x e SuSE è necessario questo
cambiamento anche se si invoca 'flex' come 'lex'!
```

Ciò genera il file 'esempio1'. Se lo si esegue, attenderà la scrittura in input di qualche carattere. Quando viene inserito in input qualcosa che non corrisponde a nessuna delle chiavi definite (cioè 'stop' e 'start') esso viene scritto in output nuovamente. Se venisse immesso 'stop' verrebbe stampato in uscita 'Ricevuto comando stop';

Chiudere il programma con un EOF (^D) [CONTROL-d].

Ci si potrebbe chiedere come mai il programma funzioni, visto che non è stata definita una funzione main(). Questa funzione viene definita implicitamente per voi in libl (liblex) che è stata inclusa nella compilazione con l'opzione -ll.

3.1 Espressioni regolari in corrispondenze

Questo esempio non è particolarmente utile in sé stesso, e neppure il prossimo. Comunque mostrerà come usare le espressioni regolari in Lex che risulteranno essere molto utili in seguito.

Esempio 2:

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

%%
[0123456789]+           printf("NUMERO\n");
[a-zA-Z][a-zA-Z0-9]*    printf("PAROLA\n");
%%
```

Questo file Lex descrive due tipi di corrispondenze (categorie): di tipo PAROLA e di tipo NUMERO. Le espressioni regolari possono essere scoraggianti, ma con un piccolo sforzo è facile capirle. Esaminiamo la combinazione del tipo NUMERO:

[0123456789]+

Questo dice: una sequenza di uno o più caratteri fra quelli del gruppo 0123456789. Avremmo potuto scriverli in maniera più breve:

[0-9]+

Ora, la combinazione del tipo PAROLA è in qualche modo più complicata:

[a-zA-Z][a-zA-Z0-9]*

La prima parte corrisponde a 1 e 1 solo carattere compreso tra 'a' e 'z', o tra 'A' e 'Z'. In altre parole, una lettera. Questa lettera iniziale deve poi essere seguita da zero o più caratteri che possono essere o una lettera o una cifra. Perché usare qui un asterisco? Il '+' significa 1 o più corrispondenze, ma una PAROLA potrebbe

consistere benissimo di un solo carattere, che è già stato considerato. Quindi la seconda parte potrebbe avere zero corrispondenze, ed è per questo che si scrive un '*'.

In questo modo, si è imitato il comportamento di molti linguaggi di programmazione che richiedono che il nome di una variabile *debba* cominciare con una lettera, ma possa poi contenere anche delle cifre. In altre parole, 'temperatura1' è un nome valido, ma '1temperatura' non lo è.

Si cerchi di compilare Esempio 2, allo stesso modo di Esempio 1, e si inserisca del testo. Ecco un esempio di sessione:

```
$ ./esempio2
foo
PAROLA

bar
PAROLA

123
NUMERO

bar123
PAROLA

123bar
NUMERO
PAROLA
```

Ci si può chiedere da dove vengano tutti questi spazi bianchi nell'output. La ragione è semplice: erano nell'input, e non si è stabilita alcuna corrispondenza per essi in nessun luogo, quindi vengono restituiti ancora (come sono).

La manpage di Flex descrive le sue espressioni regolari in dettaglio. Sono in molti a ritenere che la manpage delle espressioni regolari di perl (perlre) sia molto utile, anche se Flex non implementa tutto quello che si può fare con perl.

Si deve stare attenti a non creare corrispondenze di lunghezza zero come '[0-9]*' - altrimenti l'Analizzatore lessicale potrebbe andare in confusione cominciando a riconoscere ripetutamente stringhe vuote.

3.2 Un esempio più complicato con sintassi simile a quella del C

Mettiamo si voglia analizzare un file di questo tipo:

```
logging {
        category lame-servers { null; };
        category cname { null; };
};

zone "." {
        type hint;
        file "/etc/bind/db.root";
};
```

Si vedono chiaramente delle categorie (token) in questo file:

- PAROLA, tipo 'zone' e 'type'
- NOMEFILE, tipo '/etc/bind/db.root'
- DOPPOAPICE, come quelli che circondano il nome del file
- APERTAGRAFFA, {
- CHIUSAGRAFFA, }
- PUNTOEVIRGOLA, ;

Il file corrispondente per Lex è Esempio 3:

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

%%
[a-zA-Z][a-zA-Z0-9]*    printf("PAROLA ");
[a-zA-Z0-9\/.-]+        printf("NOMEFILE ");
\"                      printf("DOPPIOAPICE ");
\{                      printf("APERTAGRAFFA ");
\}                      printf("CHIUSAGRAFFA ");
;                       printf("PUNTOEVIRGOLA ");
\n                      printf("\n");
[ \t]+                  /* ignora spazi bianchi */;
%%
```

Quando si dà come input al programma generato da questo file per Lex il nostro file (usando esempio3.compile), si ottiene:

```
PAROLA APERTAGRAFFA
PAROLA NOMEFILE APERTAGRAFFA PAROLA PUNTOEVIRGOLA CHIUSAGRAFFA PUNTOEVIRGOLA
PAROLA PAROLA APERTAGRAFFA PAROLA PUNTOEVIRGOLA CHIUSAGRAFFA PUNTOEVIRGOLA
CHIUSAGRAFFA PUNTOEVIRGOLA

PAROLA DOPPIOAPICE NOMEFILE DOPPIOAPICE APERTAGRAFFA
PAROLA PAROLA PUNTOEVIRGOLA
PAROLA DOPPIOAPICE NOMEFILE DOPPIOAPICE PUNTOEVIRGOLA
CHIUSAGRAFFA PUNTOEVIRGOLA
```

Quando paragonato al file di configurazione di cui sopra, è chiaro che lo si è nitidamente 'categorizzato'. Ogni parte del file di configurazione è stato associato e convertito in una categoria (token).

E questo è esattamente quel che serve per poter utilizzare bene YACC.

3.3 Che cosa si è visto

Si è visto che Lex è in grado di leggere input arbitrari, e di determinare che cosa sia ogni parte dell'input. Questo è chiamato 'categorizzazione' (tokenizing).

4 YACC

YACC può analizzare flussi in input consistenti in categorie (token) con certi valori. Si vede qui chiaramente la relazione fra YACC e Lex, YACC non ha alcuna idea di cosa i 'flussi in input' siano, ha necessità di categorie preprocessate. Anche se è possibile scrivere un proprio analizzatore di categorie, qui si lascerà questo compito esclusivamente a Lex.

Una nota sulle grammatiche e sugli analizzatori sintattici. Quando YACC vide la luce, era uno strumento usato per analizzare i file in input ai compilatori: i programmi. I programmi scritti in un linguaggio di programmazione per computer sono tipicamente *non* ambigui - hanno un significato univoco. Per questo motivo, YACC non riesce a fare fronte ad ambiguità e si lamenterà di conflitti spostamento/riduzione o riduzione/riduzione. Qualcosa di più a riguardo di ambiguità e di problemi di YACC può essere trovato nel capitolo 'Conflitti'.

4.1 Un semplice controllo termostatico

Supponiamo di avere un termostato che si vuole controllare usando un semplice linguaggio. Una sessione con il termostato potrebbe essere di questo tipo:

```
riscaldamento acceso
        Riscaldamento acceso!
riscaldamento spento
        Riscaldamento spento!
obiettivo temperatura 22
        Nuova temperatura impostata!
```

Le categorie (token) che dobbiamo essere in grado di riconoscere sono: riscaldamento, acceso/spento (STATO), obiettivo, temperatura, NUMERO.

L'individuazione dei simboli di Lex avviene (Esempio 4) secondo:

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%%
[0-9]+                  return NUMERO;
riscaldamento           return TOKRISCALDAMENTO;
acceso|spento           return STATO;
obiettivo               return TOKOBIETTIVO;
temperatura             return TOKTEMPERATURA;
\n                      /* ignora fine linea */;
[ \t]+                  /* ignora spazi bianchi */;
%%
```

Si notano due cambiamenti importanti. Primo, viene incluso il file 'y.tab.h', e in secondo luogo, non viene stampato nulla, vengono restituiti i nomi delle categorie. Questo cambiamento è necessario perché si passerà tutto a YACC che si disinteressa di quello che viene stampato sullo schermo. y.tab.h contiene le definizioni di queste categorie.

Ma da dove proviene y.tab.h? Viene generato da YACC dal file di grammatica che si è in procinto di creare. Visto che il linguaggio è molto semplice, lo stesso vale per la grammatica:

```
comands: /* vuoto */
        | comandi comando
        ;

comando:
        interruttore_riscaldamento
        |
        imposta_obiettivo
        ;

interruttore_riscaldamento:
        TOKRISCALDAMENTO STATO
        {
                printf("\tRiscaldamento acceso o spento\n");
        }
        ;

imposta_obiettivo:
        TOKOBIETTIVO TOKTEMPERATURA NUMERO
        {
                printf("\tTemperatura impostata\n");
        }
        ;
```

La prima parte è quella che io chiamo la 'radice' ('root'). Stabilisce che ci sono dei 'comandi', e questi comandi sono composti da comandi singoli, ognuno dei quali è un 'comando'. Come si vede questa regola è molto ricorsiva, perché contiene anche la parola 'comandi'. Il che significa che il programma è in grado di ridurre una serie di comandi uno a uno. Per dettagli importanti sulla ricorsività leggere il capitolo 'Come lavorano internamente Lex e YACC'.

La seconda regola definisce che cosa è un comando. Noi supportiamo solo due tipi di comandi, un 'interruttore_riscaldamento' e un 'imposta_obiettivo'. Questo è quello che significa il simbolo | (or) - 'un comando consiste o di un interruttore_riscaldamento o di un imposta_obiettivo'.

Un interruttore_riscaldamento consiste nella categoria RISCALDAMENTO, che è semplicemente la parola 'riscaldamento', seguita da uno stato (che si è definito nel file di Lex come 'acceso' e 'spento').

In qualche modo più complicato è imposta_obiettivo, che consiste della categoria OBIETTIVO (la parola 'obiettivo'), la categoria TEMPERATURA (la parola 'temperatura') e un numero.

4.1.1 Un file YACC completo

La sezione precedente ha mostrato solo la parte di grammatica del file per YACC, ma c'è di più. È la testata che abbiamo omesso:

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

```
void yyerror(const char *str)
{
        fprintf(stderr,"errore: %s\n",str);
}

int yywrap()
{
        return 1;
}

main()
{
        yyparse();
}

%}

%token NUMERO TOKRISCALDAMENTO STATO TOKOBIETTIVO TOKTEMPERATURA

%%
```

La funzione yyerror() è chiamata da YACC in caso di errore. Qui si stampa semplicemente il messaggio passato, ma si può fare di meglio. Vedere la sezione 'Ulteriori letture' alla fine del documento.

La funzione yywrap() può essere usata per continuare a leggere da un ulteriore file. Viene richiamata alla fine di un file e si può allora aprire un altro file e restituire 0. Oppure si può restituire 1, indicando così che questa è effettivamente la fine di tutti i file di input. Per maggiori informazioni su questo, vedere il capitolo 'Come lavorano internamente Lex e YACC'.

Infine c'è la funzione main(), che non fa nulla se non mettere tutto in movimento.

L'ultima linea definisce semplicemente le categorie che verranno usate. Queste vengono prodotte nel file di output y.tab.h se YACC è richiamato con l'opzione '-d'.

4.1.2 Compilare ed eseguire il controllo termostatico

```
lex esempio4.l
yacc -d esempio4.y
cc lex.yy.c y.tab.c -o esempio4
```

Noterete alcune differenze. Ora si invoca anche YACC per compilare la nostra grammatica, il che crea y.tab.c e y.tab.h. Lex si richiama come al solito. Quando si compila, non serve più il flag -ll: essendoci ora una nostra funzione main() non serve quella fornita da libl.

```
NOTA: nel caso otteniate un errore legato al fatto che il compilatore non è in grado
di trovare 'yylval' aggiungere in esempio4.l, appena sotto #include <y.tab.h>, questa
riga:

extern YYSTYPE yylval;
```

```
Per spiegazioni, vedere la sezione 'Come lavorano internamente Lex e YACC'.
```

Una sessione di esempio:

```
$ ./esempio4
riscaldamento acceso
        Riscaldamento acceso o spento
riscaldamento aspento
        Riscaldamento acceso o spento
obiettivo temperatura 10
        Temperatura impostata
obiettivo umiditâ 20
errore: syntax error
$
```

Non è proprio quello che si voleva ottenere, ma per mantenere abbordabile la curva di apprendimento non si possono presentare tutte in una volta le belle cose che si possono fare.

4.2 Miglioramento del termostato per gestire parametri

Come si è visto, ora i comandi del termostato vengono analizzati correttamente e pure gli errori vengono individuati in modo appropriato. Ma come si può indovinare dalla formulazione ambigua, il programma non ha alcuna idea di quello che dovrebbe fare, non gli viene passato nessuno dei valori inseriti.

Cominciamo con aggiungergli la capacità di leggere il nuovo obiettivo di temperatura. Per fare questo, si deve insegnare alla corrispondenza NUMERO nell'analizzatore sintattico (Lexer) come convertirsi in un valore intero che possa poi essere letto da YACC.

Quando Lex trova una corrispondenza dell'obiettivo mette il testo della corrispondenza trovata nella stringa di caratteri 'yytext'. YACC a sua volta si aspetta di trovare un valore nella variabile 'yylval'. Nell'Esempio 5, vediamo l'ovvia soluzione:

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%%
[0-9]+                  yylval=atoi(yytext); return NUMERO;
riscaldamento           return TOKRISCALDAMENTO;
acceso|spento           yylval=!strcmp(yytext,"acceso"); return STATO;
obiettivo               return TOKOBIETTIVO;
temperatura             return TOKTEMPERATURA;
\n                      /* ignora fine linea */;
[ \t]+                  /* ignora spazi bianchi */;
%%
```

Come si vede, si esegue un atoi() su yytext, e si mette il risultato in yylval, da dove YACC può utilizzarlo. Si fa qualcosa di molto simile per la corrispondenza di STATO, che viene paragonato a 'acceso', e si imposta yylval a 1 se viene trovato uguale. Va notato che separando le corrispondenze di 'acceso' e 'spento' Lex

produrrebbe un codice più veloce, ma si vuole qui mostrare una regola e una azione più complicata, tanto per cambiare.

Si deve insegnare a YACC come comportarsi in questo caso. Quello che viene chiamato 'yylval' in Lex ha un nome differente in YACC. Esaminiamo la regola che imposta il nuovo obiettivo di temperatura:

```
imposta_obiettivo:
        TOKOBIETTIVO TOKTEMPERATURA NUMERO
        {
                printf("\tTemperatura impostata a %d\n",$3);
        }
        ;
```

Per accedere al valore della terza parte della regola (cioè NUMERO), è necessario usare $3. Al termine dell'esecuzione di yylex(), i contenuti di yylval vengono resi disponibili al terminale, e a quei valori si può accedere con il costrutto $.

Per spiegare ulteriormente la cosa si osservi la nuova regola 'interruttore_riscaldamento':

```
interruttore_riscaldamento:
        TOKRISCALDAMENTO STATO
        {
                if($2)
                        printf("\tRiscaldamento acceso\n");
                else
                        printf("\tRiscaldamento spento\n");
        }
        ;
```

Se ora si esegue esempio5 darà in output correttamente quello che si inserisce.

4.3 Analizzare un file di configurazione

Rivediamo una parte del file di configurazione usato precedentemente:

```
zone "." {
        type hint;
        file "/etc/bind/db.root";
};
```

Si ricordi che si è già scritto un Analizzatore lessicale (Lexer) per questo file. Tutto quello che ora è necessario fare è scrivere una grammatica per YACC, e modificare l'Analizzatore lessicale così che restituisca valori in un formato adatto a YACC.

Nell'Analizzatore lessicale dall'Esempio 6 si vede:

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
```

```
%%

zone                    return ZONETOK;
file                    return FILETOK;
[a-zA-Z][a-zA-Z0-9]*    yylval=strdup(yytext); return PAROLA;
[a-zA-Z0-9\/.-]+        yylval=strdup(yytext); return NOMEFILE;
\"                      return DOPPIOAPICE;
\{                      return APERTAGRAFFA;
\}                      return CHIUSAGRAFFA;
;                       return PUNTOEVIRGOLA;
\n                      /* ignora fine linea */;
[ \t]+                  /* ignora spazi bianchi */;
%%
```

Se si osserva attentamente si può vedere che yylval è cambiato! Non ci si aspetta più che sia un numero intero, ma in effetti si assume che sia un char *. Per amor di semplicità si è invocato strdup sprecando molta memoria. Si noti che ciò può non essere un problema in molti casi dove è solo necessario analizzare un file una volta e poi uscire.

Si vogliono memorizzare stringhe di caratteri perché qui si ha a che fare principalmente con nomi: nomi di file e nomi di zone. In un capitolo successivo si spiegherà come trattare dati di tipi differenti.

Per informare YACC riguardo al nuovo tipo di yylval, va aggiunta questa linea alla testata della grammatica di YACC:

```
#define YYSTYPE char *
```

La grammatica in sé è ancora più complicata. La vediamo a piccole dosi per renderla più facilmente digeribile.

```
comandi:
        |
        comandi comando PUNTOEVIRGOLA
        ;


comando:
        imposta_zone
        ;

imposta_zone:
        ZONETOK nomefradoppiapici contenutozone
        {
                printf("Zone completa trovata per '%s'\n",$2);
        }
        ;
```

Questa è l'introduzione, inclusa la 'radice' ricorsiva già menzionata più sopra. Notare che si è specificato che i comandi sono terminati (e separati) da ;. Si è definito un tipo di comando, 'imposta_zone'. Consiste della categoria ZONETOK (la parola 'zone'), seguita da un nome racchiuso tra doppi apici e 'contenutozone'. Questo contenutozone comincia in modo abbastanza semplice:

```
contenutozone:
        APERTAGRAFFA dichiarazionizone CHIUSAGRAFFA
```

È necessario che cominci con una APERTAGRAFFA, una {. Poi segue una dichiarazionizone, seguita da una CHIUSAGRAFFA, }.

```
nomefradoppiapici:
        DOPPIOAPICE NOMEFILE DOPPIOAPICE
        {
                $$=$2;
        }
```

Questa sezione definisce cosa sia un 'nomefradoppiapici': è un NOMEFILE tra due DOPPIOAPICE. Poi dice qualcosa di speciale: il valore di un simbolo di tipo nomefradoppiapici è il valore del NOMEFILE. Questo significa che nomefradoppiapici ha come suo valore il valore di filename senza i doppi apici.

Questo è quello che il comando magico '$$=$2;' fa. Dice: il mio valore è il valore della mia parte seconda. Quando si fa riferimento a nomefradoppiapici in altre regole e si accede al suo valore con il costrutto $, si vede il valore che si è impostato qui con $$=$2.

```
NOTA: questa grammatica s'inceppa con nomi di file che non contengono né un '.'  né una
'/'.

dichiarazionizone:
        |
        dichiarazionizone dichiarazionezona PUNTOEVIRGOLA
        ;

dichiarazionezona:
        dichiarazioni
        |
        FILETOK nomefradoppiapici
        {
                printf("Nome di file zone '%s' trovato\n",$2);
        }
        ;
```

Questa è una dichiarazione generica che intercetta tutti i tipi di dichiarazione all'interno del blocco 'zone'. Notiamo ancora la ricorsività.

```
blocco:
        APERTAGRAFFA dichiarazionizone CHIUSAGRAFFA PUNTOEVIRGOLA
        ;

dichiarazioni:
        | dichiarazioni dichiarazione
        ;

dichiarazione: PAROLA | blocco | nomefradoppiapici
```

Questo definisce un blocco e le 'dichiarazione' che vi si possono trovare.

Quando viene eseguito il risultato in uscita appare come:

```
$ ./esempio6
zone "." {
        type hint;
        file "/etc/bind/db.root";
        type hint;
};
Nome di file zone '/etc/bind/db.root' trovato
Zone completa trovata per '.'
```

5 Fare un analizzatore in C++

Per quanto Lex e YACC preceda il C++, è possibile generare un analizzatore sintattico C++. Anche se Flex include un'opzione per generare un Analizzatore lessicale in C++, non verrà usata qua, visto che YACC non sa come gestirla direttamente.

Il modo che preferisco per fare un analizzatore sintattico per C++ è di fare generare a Lex un file in C, e lasciare a YACC la generazione del codice C++. Quando poi si effettua il link dell'applicazione, si potrebbero incontrare dei problemi perché, senza indicazioni, il codice C++ non è in grado di trovare le funzioni C, a meno che non gli si dica che quelle funzioni sono extern C.

Per fare ciò, si fa una testata C in YACC come questa:

```
extern "C"
{
        int yyparse(void);
        int yylex(void);
        int yywrap()
        {
                return 1;
        }

}
```

Se si volesse dichiarare o cambiare yydebug, si deve fare ora in questo modo:

```
extern int yydebug;

main()
{
        yydebug=1;
        yyparse();
}
```

Questo a causa della regola del C++ che esige un'unica definizione, ciò che impedisce definizioni multiple di yydebug.

Può anche capitare che sia necessario ripetere la #define di YYSTYPE nel file per Lex, a causa di una maggiore rigidità nel controllo del tipo in C++.

Per compilare, fare qualcosa di simile a quel che segue:

```
lex bindconfig2.l
yacc --verbose --debug -d bindconfig2.y -o bindconfig2.cc
cc -c lex.yy.c -o lex.yy.o
c++ lex.yy.o bindconfig2.cc -o bindconfig2
```

A causa dell'asserzione -o y.tab.h è ora chiamato bindconfig2.cc.h, quindi tenetene conto.

Per riassumere: non prendetevi la briga di compilare l'analizzatore sintattico (Lexer) in C++, lasciatelo in C. Scrivete il vostro analizzatore (Parser) in C++ e spiegate al vostro compilatore che alcune funzioni sono funzioni C con dichiarazioni extern C.

6 Come lavorano internamente Lex e YACC

Nel file per YACC, siete voi a scrivere la vostra funzione main() che ad un certo punto chiama yyparse(). La funzione yyparse() viene creata automaticamente da YACC, e finisce in y.tab.c.

yyparse() legge un flusso di coppie simbolo/valore provenienti da yylex() che è necessario fornire. Si può scrivere questa funzione per conto proprio o lasciare questo compito a Lex. Nei nostri esempi si è scelto di lasciare questo compito a Lex.

La yylex() come scritta da Lex legge i caratteri da un puntatore a file FILE * chiamato yyin. Se non viene specificato yyin Lex usa lo standard input. L'output va in yyout, e se non specificato finisce sullo stdout. Si può anche modificare yyin nella funzione yywrap() che viene chiamata a fine file. Questa funzione permette di aprire un altro file, e continuare nell'analisi.

In questo caso va fatto in modo che restituisca 0. Nel caso si volesse far finire l'analisi dopo questo file gli si faccia restituire 1.

Ogni chiamata a yylex() restituisce un valore intero che rappresenta un tipo di categoria. Questo dice a YACC che genere di categoria ha letto. Il simbolo può eventualmente avere un valore, che dovrà essere messo nella variabile yylval.

Per default yylval è di tipo int ma si può cambiarlo dal file per YACC ridefinendo YYSTYPE con #define.

È necessario che l'Analizzatore lessicale (Lexer) sia in grado di accedere a yylval. Per questo la si deve dichiarare nell'ambito dell'Analizzatore lessicale (Lexer) come una variabile extern. Lo YACC originale tralascia di fare questo per voi, per cui si dovrebbe aggiungere il seguente codice al proprio Analizzatore lessicale (Lexer), giusto sotto #include <y.tab.h>:

```
extern YYSTYPE yylval;
```

Bison, che è utilizzato di questi tempi dalla maggioranza della gente, lo fa automaticamente per l'utilizzatore.

6.1 Valori di categoria (Token values)

Come detto precedentemente yylex() deve restituire quale tipo di categoria incontra e mettere il suo valore in yylval. Quando queste categorie vengono definite con il comando %token ad esse vengono assegnati id [identificativi] numerici a partire da 256.

Grazie a questo, è possibile avere tutti i caratteri ASCII come categorie. Diciamo che si voglia scrivere un calcolatore, sino ad ora avremmo scritto l'Analizzatore lessicale come segue:

```
[0-9]+          yylval=atoi(yytext); return NUMERO;
[ \n]+          /* ignora spazi bianchi */;
-               return MENO;
\*              return PER;
\+              return PIU;
...
```

La grammatica di YACC conterrebbe allora:

```
exp:    NUMERO
        |
        exp PIU exp
        |
        exp MENO exp
        |
        exp PER exp
```

Questo è inutilmente complicato. Usando caratteri come abbreviazioni degli id dei simboli numerici, si può riscrivere l'Analizzatore lessicale come:

```
[0-9]+          yylval=atoi(yytext); return NUMERO;
[ \n]+          /* ignora spazi bianchi */;
.               return (int) yytext[0];
```

Quest'ultimo punto mette in corrispondenza tutti i singoli caratteri per i quali non è stata trovata una corrispondenza.

La grammatica per YACC sarebbe allora:

```
exp:    NUMERO
        |
        exp '+' exp
        |
        exp '-' exp
        |
        exp '*' exp
```

Questo è molto più stringato e anche molto più chiaro. Non è necessario dichiarare questi simboli ascii con %token nella testata, sono definiti implicitamente.

Un'altra cosa molto utile a proposito di questo costrutto è che ora Lex non cercherà una corrispondenza a tutto quello che gli spediamo - evitando il comportamento di default per il quale tutto ciò che in input non

trova una corrispondenza viene inviato in output identico. Se l'utente di questo calcolatore usa un ^, per esempio, ora darà un errore di analisi, invece di venir riflesso nello standard output.

6.2 Ricorsione: 'a destra è sbagliata' (right is wrong)

La ricorsione è un aspetto vitale di YACC. Senza di essa non si può specificare che un file consista di una sequenza di comandi indipendenti o asserzioni. Di per sé stesso, YACC è interessato solo alla prima regola, o a quella che si designa essere, con il simbolo '%start', la regola di partenza.

La ricorsione appare in YACC di due tipi: a destra e a sinistra. La ricorsione a sinistra, quella che si dovrebbe usare la maggior parte delle volte, assomiglia a quella che segue:

```
commands: /* vuoto */
        |
        commands command
```

Questa dice: un comando può o essere vuoto o consistere di più comandi seguiti da un comando. Per come lavora YACC, ciò significa che YACC può ora ritagliare gruppi individuali di comandi facilmente (dall'inizio) via via riducendoli.

Paragoniamola con la ricorsione a destra, che abbastanza stranamente a molte persone pare migliore:

```
commands: /* vuoto */
        |
        command commands
```

Ma questa è costosa. Se usata come regola %start, richiede a YACC di tenere tutti i comandi del vostro file nello stack, il che può richiedere parecchia memoria. Quindi si raccomanda caldamente di usare la ricorsione a sinistra quando si analizzino lunghe sequenze di comandi, come interi file. Alcune volte è difficile evitare la ricorsione a destra ma se le sequenze di comandi non sono troppo lunghe non è necessario fare i salti mortali per usare la ricorsione a sinistra.

Se si ha qualcosa che delimita (e quindi separa) i comandi, la ricorsione a destra appare molto naturale, ma è comunque costosa:

```
commands: /* vuoto */
        |
        command PUNTOEVIRGOLA commands
```

Il modo corretto per codificare in questo caso è usando la ricorsione a sinistra (non sono stato io ad inventarmi neppure questo):

```
commands: /* vuoto */
        |
        commands command PUNTOEVIRGOLA
```

Una versione precedente di questo HOWTO erroneamente usava la ricorsione a destra. Markus Triska gentilmente ce l'ha fatto notare.

6.3 Advanced yylval: %union

Attualmente, si deve definire *il* tipo di yylval. Questo però non va sempre bene. Ci sono situazioni in cui dobbiamo essere in grado di gestire tipi di dati multipli. Tornando al nostro ipotetico termostato, magari si vuole essere in grado di scegliere quale stufa si vuole controllare, così:

```
stufa edificioprincipale
        Selected 'edificioprincipale' stufa
obiettivo temperatura 23
        'edificioprincipale' stufa obiettivo temperatura adesso 23
```

Quello che si richiede qui è che yylval sia una 'union' che può contenere sia stringhe che numeri interi - ma non simultaneamente.

Si ricordi che in precedenza si era detto a YACC che tipo ci si aspettava che yylval fosse definendo YYSTYPE. Si può ragionevolmente definire YYSTYPE in modo che sia una 'union' nello stesso modo, ma YACC ha un metodo più semplice per fare questo: il comando %union.

Basandoci sull'esempio 4, si può ora scrivere la grammatica per YACC dell'Esempio 7. Prima l'introduzione:

```
%token TOKSTUFA TOKRISCALDAMENTO TOKOBIETTIVO TOKTEMPERATURA

%union
{
        int numero;
        char *stringa;
}

%token <numero> STATO
%token <numero> NUMERO
%token <stringa> PAROLA
```

Si è definita la 'union', che contiene soltanto un numero e una stringa. Poi usando una sintassi estesa di %token si è spiegato a YACC a quale parte della 'union' ciascuna categoria dovrebbe accedere.

In questo caso, si è permesso alla categoria STATO di usare un intero, come prima. La stessa cosa per il simbolo NUMERO che viene usato per leggere la temperatura.

Nuovo invece è il simbolo PAROLA, che è dichiarato necessitare una stringa.

Anche il file dell'Analizzatore lessicale (Lexer) cambia un po':

```
%{
#include <stdio.h>
#include <string.h>
#include "y.tab.h"
%}
%%
[0-9]+              yylval.numero=atoi(yytext); return NUMERO;
stufa               return TOKSTUFA;
riscaldamento       return TOKRISCALDAMENTO;
acceso|spento       yylval.numero=!strcmp(yytext,"acceso"); return STATO;
```

```
obiettivo           return TOKOBIETTIVO;
temperatura         return TOKTEMPERATURA;
[a-z0-9]+           yylval.stringa=strdup(yytext);return PAROLA;
\n                  /* ignora fine linea */;
[ \t]+              /* ignora spazi bianchi */;
%%
```

Come si può vedere, non si accede più direttamente a yylval, si aggiunge un suffisso che indica a quale parte di esso si vuole accedere. Non si ha bisogno di farlo nella grammatica per YACC però, visto che YACC compie la magia da sé:

```
scegli_stufa:
        TOKSTUFA PAROLA
        {
                printf("\tScelta stufa '%s'\n",$2);
                stufa=$2;
        }
        ;
```

A seguito della dichiarazione del %token di cui sopra, YACC sceglie automaticamente il membro 'stringa' dalla nostra 'union'. Si noti pure che si è memorizzata una copia di $2, che successivamente viene usata per dire all'utente a quale stufa sta mandando comandi:

```
target_set:
        TOKOBIETTIVO TOKTEMPERATURA NUMERO
        {
                printf("\tPer stufa '%s' temperatura impostata a %d\n",stufa,$3);
        }
        ;
```

Per maggiori dettagli leggere esempio7.y.

7 Debugging

Specialmente quando si sta imparando è importante avere strumenti per il debugging. Fortunatamente YACC può dare molte informazioni. Queste informazioni richiedono alcune attività addizionali, così è necessario fornire qualche opzione per abilitarle.

Quando si compila la grammatica si deve aggiungere –debug e –verbose alla linea di comando di YACC. Nella testata C della grammatica aggiungere la seguente definizione:

int yydebug=1;

Questo genererà il file 'y.output' che spiega la macchina degli stati che è stata creata.

Quando si lancia il binario generato, verrà mostrato *molto* di quello che sta succendendo, incluso a quale stato la macchina degli stati si trova attualmente, e quale categoria si sta leggendo.

Peter Jinks ha scritto una pagina in *debugging* <http://www.cs.man.ac.uk/~pjj/cs2121/debug.html> che contiene alcuni errori comuni e come risolverli.

7.1 La macchina degli stati

Internamente, l'Analizzatore lessicale di YACC esegue una cosiddetta 'macchina degli stati'. Come il nome suggerisce è una macchina che può assumere molti stati. Ci sono poi regole che governano i passaggi da uno stato a un altro. Tutto parte con la cosiddetta regola che ho chiamato 'radice' menzionata precedentemente.

Per citare dall'uscita dell'Esempio 6 y.output:

```
stato 0

    0 $accept: . comandi $end

    $default        riduzione con la regola 1 (comandi)

    comandi         prosecuzione allo stato 1

stato 1

    0 $accept: comandi . $end
    2 comandi: comandi . comando PUNTOEVIRGOLA

    $end     shift e prosecuzione allo stato 2
    ZONETOK  shift e prosecuzione allo stato 3

    comando       prosecuzione allo stato 4
    imposta_zone  prosecuzione allo stato 5
```

Per default, questo stato riduce l'uso della regola 'comandi'. Questa è la regola ricorsiva menzionata prima che definisce 'comandi' come insieme di dichiarazioni di comandi individuali, seguita da un punto e virgola, seguito eventualmente da altri comandi.

Questo stato continua a ridurre la frase sino a quando non incontra qualcosa che è in grado di capire, in questo caso, un ZONETOK, cioè, la parola 'zone'. Va quindi allo stato 3, che ha ulteriormente a che fare con un comando di zona:

```
state 3

    4 imposta_zone: ZONETOK . nomefradoppiapici contenutozone

    DOPPIOAPICE  shift e prosecuzione allo stato 6

    nomefradoppiapici  prosecuzione allo stato 7
```

La prima linea contiene un '.' per indicare dove si è: si è appena visto un ZONETOK e si è in cerca di un 'nomefradoppiapici'. Apparentemente un nomefradoppiapici comincia con un DOPPIOAPICE, che manda la macchina allo stato 6.

Per andare oltre con l'esempio, utilizzare l'Esempio 6 che contiene le opzioni citate nella sezione Debugging, oppure aggiungete le opzioni all'Esempio 7, ricompilate, eseguite, ed esaminate y.output.

7.2 Conflitti: 'sposta/riduci', 'riduci/riduci'

Quando YACC avvisa di un conflitto, è probabile ci si trovi nei guai. La soluzione di questi conflitti sembra essere in qualche modo una forma d'arte che può insegnare molto a proposito del linguaggio in definizione. Più di quanto non si sarebbe voluto sapere.

I problemi girano intorno al modo d'interpretare una sequenza di categorie. Si supponga di definire un linguaggio che deve essere in grado di accettare entrambi questi comandi:

```
cancella stufa tutte
cancella stufa numero
```

Per farlo si definisce la grammatica:

```
cancella_stufe:
        TOKCANCELLA TOKSTUFA modo
        {
                cancellastufe($3);
        }

modo:   PAROLA

cancella_una_stufa:
        TOKCANCELLA TOKSTUFA PAROLA
        {
                cancella($3);
        }
```

Si annusa già odore di guai. La macchina degli stati comincia leggendo la parola 'delete' e deve poi decidere dove andare basandosi sulla successiva categoria. Questa prossima categoria può essere un modo che specifica come cancellare le stufe, oppure il nome di una stufa da cancellare.

Il problema è che per entrambi i comandi la prossima categoria sarà una PAROLA. YACC quindi non ha la minima idea su cosa fare. Questo porta ad un avviso di 'riduzione/riduzione', e a un ulteriore avviso che il nodo 'cancella_una_stufa' non verrà mai raggiunto.

In questo caso il conflitto è risolto facilmente (cioè rinominando il primo comando in 'cancella_tutte_le_stufe', o facendo diventare una categoria separata 'tutte'), ma qualche volta è più difficile venirne fuori. Il file y.output generato quando si passa a yacc l'opzione –verbose può essere di grandissimo aiuto.

8 Ulteriori letture

GNU YACC (Bison) viene fornito con un info-file (.info) molto bello che documenta la sintassi di YACC molto bene. Menziona Lex solo una volta, ma per altri versi è molto buono. Si possono leggere i file .info con Emacs o con quello strumento molto bello che è 'pinfo'. È disponibile anche sul sito GNU:

BISON Manual <http://www.gnu.org/manual/bison/> .

Flex viene fornito con una buona manpage che è molto utile se si ha già una comprensione a grandi linee di cosa faccia Flex. Il

Manuale Flex <http://www.gnu.org/manual/flex/> è anche disponibile online.

Dopo l'introduzione di Lex e YACC è probabile ci si accorga che si ha necessità di maggiori informazioni. Io non ho ancora letto alcuno di questi libri, ma sono buoni:

Bison-The Yacc-Compatible Parser Generator

Di Charles Donnelly e Richard Stallman. Un utente di *Amazon* <http://www.amazon.com/exec/obidos/ASIN/0595100325/qid=989165194/sr=1-2/ref=sc_b_3/002-7737249-1404015> lo ha trovato utile.

Lex & Yacc

Di John R. Levine, Tony Mason e Doug Brown. È considerato il lavoro standard sull'argomento, anche se è un po' datato. Recensioni su di esso su *Amazon* <http://www.amazon.com/exec/obidos/ASIN/1565920007/ref=sim_books/002-7737249-1404015> .

Compilers : Principles, Techniques, and Tools

Di Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Il 'Dragon Book'. È uscito nel 1985 e lo continuano a ristampare. Considerato il lavoro standard sulla costruzione di compilatori.

Amazon <http://www.amazon.com/exec/obidos/ASIN/0201100886/ref=sim_books/002-7737249-1404015>

Thomas Niemann ha scritto un documento che spiega come scrivere compilatori e calcolatori con Lex e YACC. Lo si può trovare *qui* <http://epaperpress.com/lexandyacc/index.html> .

Il newsgroup moderato comp.compilers può essere utile ma occorre tenere a mente che le persone non sono un helpdesk dedicato agli analizzatori! Prima di postare si legga la loro interessante *pagina* <http://compilers.iecc.com/> e specialmente la *FAQ* <http://compilers.iecc.com/faq.txt> .

Lex - A Lexical Analyzer Generator by M. E. Lesk e E. Schmidt è uno dei documenti di riferimento originali per Lex. Si può trovare

qui <http://www.cs.utexas.edu/users/novak/lexpaper.htm> .

Yacc: Yet Another Compiler-Compiler di Stephen C. Johnson è uno dei documenti di riferimento originali per YACC. Si può trovare

qui <http://www.cs.utexas.edu/users/novak/yaccpaper.htm> . Contiene utili suggerimenti di stile.

9 Riconoscimenti e ringraziamenti

- Pete Jinks <pjj%cs.man.ac.uk>
- Chris Lattner <sabre%nondot.org>
- John W. Millaway <johnmillaway%yahoo.com>
- Martin Neitzel <neitzel%gaertner.de>
- Sumit Pandaya <sumit%elitecore.com>
- Esmond Pitt <esmond.pitt%bigpond.com>

- Eric S. Raymond
- Bob Schmertz <schmertz%wam.umd.edu>
- Adam Sulmicki <adam%cfar.umd.edu>
- Markus Triska <triska%gmx.at>
- Erik Verbruggen <erik%road-warrior.cs.kun.nl>
- Gary V. Vaughan <gary%gnu.org> (leggete il suo splendido *Autobook* <http://sources.redhat.com/autobook>)
- *Ivo van der Wijk* <http://vanderwijk.info> (
  Amaze Internet <http://www.amaze.nl>)