Breve introduzione a lex e yacc

mobytrick

23 agosto 2014

Sommario

Nella presente nota si fa riferimento a lex e yacc ma si tenga presente che già da tempo i due comandi fanno riferimento ai loro ai rispettivi successori, flex per quanto riguarda lex e bison per yacc.
lex significa **lex**ical analizer. yacc è una sigla che sta per **y**et **a**nother **c**ompiler **c**ompiler.
flex sta per **f**ast **lex**. Visto che yacc foneticamente si pronuncia come yak, un ruminante, pure il successore porta il nome di un animale di quel tipo.

lex

lex è un'utility pilotata da uno script[1] in cui sono codificate delle *regole* composte da una Regular Expression cui sono agganciate delle azioni. lex legge da standard input e confronta il materiale con le Regular Expressions presenti, nell'ordine con cui compaiono. Se c'è corrispondenza esegue l'azione correlata che consiste di statement scritti nel cosiddetto linguaggio *ospite*, in pratica C, e poi si ritorna alla fase di lettura. Se non c'è corrispondenza si passa alla Regular Expression successiva e via a seguire. L'input non intercettato da alcuna Regular Expression viene stampato per default sullo standard output. Il materiale intercettato viene deposto in una stringa apposita (`yytext`), la cui lunghezza viene riportata nella variabile `yyleng`. Entrambe le entità sono a disposizione del programmatore.
Una precisazione riguardante l'*azione*. lex **non** è un'utility operativa. Il suo compito è quello di tradurre Regular Expressions ed azioni in linguaggio C scrivendo le istruzioni nella funzione `yylex` nel file `lex.yy.c` (nome di default). La funzione è di tipo `int`, non richiede parametri e restituisce 1 (uno)

[1] lo script si trova in un file il cui nome deve terminare con i caratteri `.l` (punto elle minuscola)

in caso d'errore, altrimenti 0 (zero). Corredata da un main fornito dalla libreria lex oppure codificato a mano, viene compilata con la creazione dell'eseguibile. L'azione specificata diventa operativa quando l'eseguibile viene fatto girare.

Porre la massima attenzione all'ordine con cui compaiono le regole perché talvolta potrebbe accadere che l'input venga intercettato da una regola diversa da quella voluta.

Scendendo in dettaglio, lo script lex si compone di 3 parti e come separatore si usa una riga composta da due volte il carattere `%`. La prima e la terza parte sono opzionali. Le caratteristiche sono sommarizzate nella seguente tabella:

definizioni	opzionale
`%%`	**richiesta**
regole	opzionale
`%%`	opzionale
user's routines	opzionale

Il fatto che la seconda sezione sia opzionale è puramente teorico. Non specificare alcuna regola equivale ad usare quella di default. Ovvero, lex viene degradato ad un programma di copia alquanto laborioso. Stabilito che le regole devono esserci, queste, a loro volta, hanno la seguente sintassi:

Regular Expression azione

Tra Regular Expression ed azione **deve** esserci almeno 1 tabulatore orizzontale pena il non funzionamento. Copiare pedissequamente uno script lex non è buona cosa perché il carattere tabulatore non ha corrispondenza grafica e viene reso con un numero di spazi che dipende dalle circostanze.
Se l'azione è composta da un numero di statement tale da essere scritti su più righe, allora è necessario far intervenire una coppia di parentesi graffe. Sommarizzando:

Regular Expression	`statement C; statement C;`

Regular Expression	`{`
	`   statement C;`
	`   statement C;`
	`}`

Alle volte l'azione è la medesima per Regular Expressions diverse. Anziché ripetere l'azione, si usa l'operatore di alternanza (il carattere `|`) con la seguente sintassi:

<table><tr><td>Regular Expression$_1$ | Regular Expression$_2$</td><td>azione</td></tr></table>

Si tenga presente che in ambito lex il metacarattere . (punto) ha un comportamento leggermente differente da quello standard. In quest'ultimo il punto intercetta qualsiasi carattere, quale che esso sia. In ambito lex il punto intercetta qualsiasi carattere, fatta eccezione per il newline (\n). Questo va intercettato con una regola ad hoc:

Regular Expression	*azione*
\n	statement C

In lex esistono ulteriori metacaratteri: <> e /. La coppia di parentesi angolari serve per definire la cosiddetta *start condition*, qui non trattata poiché va oltre l'intento di fornire un'introduzione a lex e yacc. Il carattere / (barra) viene usato con la seguente sintassi, ove $\mathtt{RE}_x$ denota una Regular Expression:

$\mathtt{RE}_1$/$\mathtt{RE}_2$

/ intercetta $\mathtt{RE}_1$ **se e solo se** $\mathtt{RE}_1$ è seguita da $\mathtt{RE}_2$. In yytext vengono deposti i soli caratteri intercettati dalla prima Regular Expression e non anche quelli intercettati dalla seconda:

Oltre ai metacaratteri, in lex esistono pure i cosiddetti *caratteri di controllo*. Uno è il % (percento), usato nella sezione delle definizioni (nello script lex) per delimitare i segmenti di definizioni da riportare nel file lex.yy.c. Un altro carattere di controllo è il " (apice doppio). lex accetta i caratteri seguenti sino all'incontro di un altro ", purché sulla stessa riga.

Nella parte *definizioni* ci possono essere delle *macro testuali*, usate poi nella parte regole, e definizioni che vengono accluse alla funzione yylex. Per operare la necessaria distinzione, tutto il materiale compreso tra %{ e %} viene riportato pari pari all'inizio del file lex.yy.c ove verrà scritta la funzione yylex, prima della funzione stessa. In genere sono definizioni di variabili globali e #include. Si tenga presente che le eventuali routines accessorie (3^ parte dello script lex) scritte dall'utente vengono riportate pari pari alla fine del file di cui sopra, dopo la funzione. Esiste pure un'ulteriore possibilità: definire delle variabili -precedute da almeno un tabulatore- prima delle regole, ma sempre nella seconda sezione dello script (un esempio lo si trova nel listato che inizia a pag. 18. Tali variabili verranno poste tra le dichiarazioni della funzione yylex. Il tutto è sommarizzato nella figura 1 a pag. 4.
Le macro testuali facilitano la stesura di script lex facili da comprendere. Al posto di Regular Expression lunghe/complesse conviene scrivere un nome

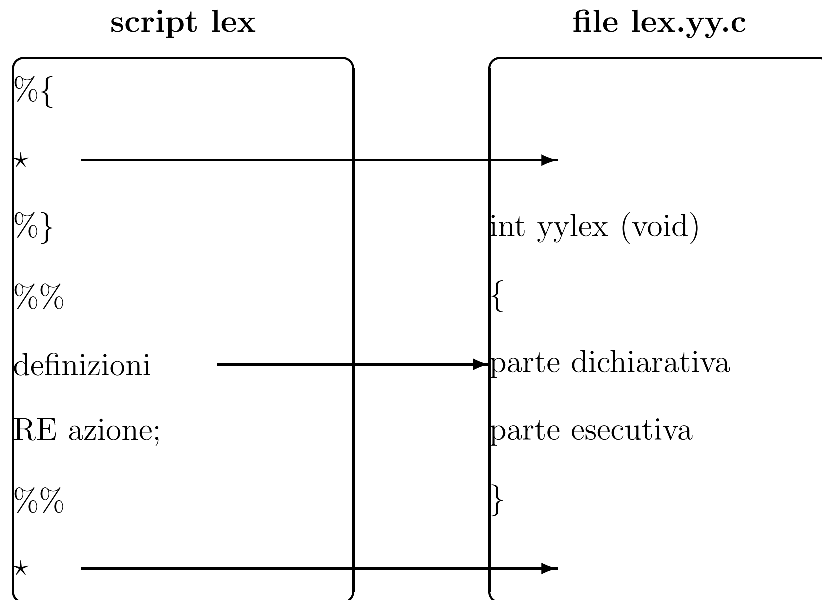

Figura 1: Trasferimenti

convenzionale corto. Altre volte si sostituisce una Regular Expression con un nome più mnemonico. Le macro sono composte da:

1. nome
2. corpo

lex di sua iniziativa sostituirà il nome della macro, purché racchiuso tra parentesi graffe, con la definizione vera e propria. La sostituzione parte dalla riga successiva quella della definizione e può interessare eventuali macro successive. Un esempio per fissare le idee:

```
OCT         1[0-9]{2}|2[0-4][0-9]|25[0-5]
IPADDR      ({OCT}\.){3}{OCT}
%%
{IPADDR}           ...
```

Un indirizzo Internet `IPV4` è formato da 4 *ottetti* separati dal carattere punto. Ciascun ottetto è un numero formato da max. 3 cifre e di valore non superiore a 255. `OCT` è una macro testuale a cui corrisponde una Regular Expression piuttosato complessa (per i più curiosi: questa Regular Expression intercetta un qualsisasi numero -formato da 1 a tre cifre- e di valore inferiore a 256).

Avendo descritto in termini di Regular Expression un ottetto, è facile comporre, sempre in tale ambito, l'indirizzo IP. In lex con 2 macro la faccenda è risolta in maniera concisa senza per questo scadere nella cripticità. Nell'esempio proposto la macro `OCT` viene subito usata e per ben 2 volte nella riga successiva la sua definizione. A sua volta la macro `IPADDR` viene sostituita nella parte regole dello script.
Convenzione vuole che le macro siano scritte in maiuscolo, così da risultare più evidenti.

lex legge dallo *stream* (gergo C) `yyin`, per default agganciato allo standard input. E scrive sullo *stream* `yyout`, per default agganciato allo standard output. È possibile cambiare l'uno oppure l'altro oppure tutti e due i default scrivendo di proprio pugno la funzione main opportuna. Supposto di voler leggere dal file `input_lex` si procede nel modo di seguito indicato (viene riportata la sezione terza dello script lex, quella dove vengono collocate le routines scritte dall'utente):

```
%%
int main (void)
{
  FILE *f;
  f = fopen ("input_lex", "r");
  if (f == NULL)
    yyerror ("errore apertura file input_lex");
  else
    {
      yyin = f;
      (void) yylex ();
      fclose (f);
    }
  exit (EXIT_SUCCESS);
}
```

Il testo che appare con sfondo grigio costituisce una falsariga per creare un proprio main, se le esigenze lo rendessero necessario.

Nell'esempio sopra riportato il nome del file di input è stato codificato all'interno del programma. Si poteva scegliere di farlo richiedere dal programma stesso oppure arrangiare le cose in modo da passarlo in linea. La funzione `yyerror` non viene fornita dalla libreria lex, bensì da quella yacc che va quindi linkata quando si procede alla compilazione del file `lex.yy.c`. Ma nulla vieta di scriversela da sé. Questa è una falsariga:

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

Vista la presenza di funzioni che effettuano l'input/output e di macro di sistema, è necessario pure un intervento nella prima sezione -quella delle definizioni- dello script lex per effettuare alcuni `#include` necessari per una corretta compilazione. Di seguito, quindi, l'inizio della prima sezione:

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

Esempio (completo) lex

In un programma C gli identificatori hanno la sintassi rappresentata dal diagramma sintattico della figura 2 a pag. 6.

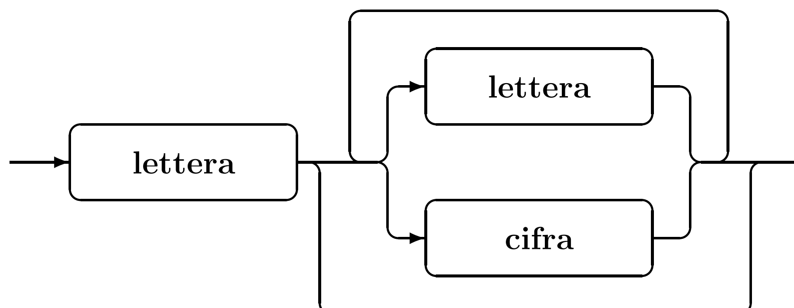

Figura 2: Identificatore

Per `cifra` si intendono le 10 cifre arabe, date per note; per `lettera` si intendono le 26 lettere dell'alfabeto internazionale, sia minuscole che maiuscole, ed il carattere `_` (underscore). È bene ricordare in maniera esplicita che lettere minuscole e maiuscole sono **differenti**. Il seguente script:

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

LETTER  [_a-zA-Z]
```

```
DIGIT   [0-9]
IDENTIF {LETTER}({LETTER}|{DIGIT})*

%%

\/\*.+\*\/      ;       /* salta i commenti         */
\"[^\"]+\"      ;       /* salta le stringhe */
{IDENTIF}       (void) printf ("%s\n", yytext); /*print ident*/
.|\n            ;       /* elimina tutto il resto    */
```

estrae gli identificatori **dopo** aver eliminato i commenti (purché su una riga ce ne sia 1 solo ed interamente compreso in essa) e le stringhe (purché non siano spezzate su più righe). Lo script considera identificatori anche le *keyword* del linguaggio C.
Se lo script si trova nel file `identif.l`, per ottenere l'eseguibile si opera nel seguente modo:

1. `lex identif.l`
   viene generato il file `lex.yy.c`

2. `gcc -ansi -pedantic -O2 lex.yy.c -o identif -s -ll`
   il file viene compilato, con intervento esplicito della libreria lex (parametro: `-ll`, due volte la lettere elle minuscola).

yacc

yacc è un'utility pilotata da uno script[2] scritto in maniera tale da eseguire delle determinate operazioni. yacc però non è autonomo. Non è stato progettato per leggere da input. La lettura dei dati (termine **molto** generico) ed una loro pre-elaborazione è demandata ad altre entità: lex oppure programmi ad hoc. Nettamente preferita la prima soluzione. Né yacc è operativo. Il suo compito si estrinseca nello scrivere una funzione -`yyparse`- in un file che di default si chiama `y.tab.c`[3]. La funzione è di tipo `int`, non richiede parametri e restituisce 1 (uno) in caso d'errore, altrimenti 0 (zero). Corredata da un main fornito dalla libreria yacc oppure codificato a mano, viene compilata con la creazione dell'eseguibile. L'operatività si estrinseca quando l'eseguibile viene fatto girare.

[2]lo script si trova in un file il cui nome deve terminare con i caratteri `.y`

[3]rigurado il nome, c'è una differenza tra yacc e bison. Quest'ultimo, a differenza di yacc, partendo dallo script *filename.y* genera il file *filename.tab.y*

Lo script yacc si compone di tre parti e come separatore si usa una riga che contiene due volte il carattere %. La seguente tabella sommarizza le caratteristiche:

dichiarazioni	opzionale
%%	**richiesta**
grammatica	**richiesta**
%%	opzionale
user's routines	opzionale

Il cuore dello script è costituito dalla grammatica di tipo *context-free*. Vengono descritte, tramite il formalismo Backus-Naur, le relazioni esistenti tra i vari componenti. La descrizione si avvale di *regole di produzione*, composte da una regola e dalla sua definizione. Per la definizione ci si avvale di *simboli non terminali* e di *simboli terminali*. Questi sono chiamati così perché non sono derivabili da altri. In altre parole sono i mattoni. Servono, assieme alle regole di produzione, a formare -meglio **produrre**- altri simboli, detti appunto non terminali.

Il precedente paragrafo è essenzialmente teorico e di difficile comprensione, a meno di non avere una solida conoscenza nel campo. Vediamo un esempio banale, quale è la definizione di numero decimale. A parole è semplice: una serie di cifre, seguita dal carattere . (punto), seguito da un'altra serie di cifre. Schematizzando formalmente:

numero decimale	::=	lista-cifre . lista_cifre
lista_cifre	::=	lista_cifre | cifra
cifra	::=	0 | 1 ... | 8 | 9

Il carattere . (punto) e le cifre arabe sono simboli terminali, in quanto non ulteriormente derivabili da altri. `cifra`, `lista_cifre`, `numero` sono non terminali in quanto componibili. Nel caso della `lista` si usano simboli terminali (le cifre) ed il modo con cui produrla. Il simbolo | (barra verticale, tecnicamente nota come *union operator* oppure *alternanza*) va interpretato come "*oppure*". In altre parole la `cifra` è prodotta (`::=`) prendendo 1 cifra, scelta tra le ben note 10. `cifra` può essere ad esempio il 5, oppure l'1 e così via. Il seguente diagramma sintattico di figura 3 a pag. 9 semplifica, forse e sperabilmente, la comprensione del concetto:
Stabilita la cifra, si sale di un livello. La `lista_cifre` è composta o da sé stessa oppure da una cifra. C'è di mezzo la ricorsione perché a priori non si può conoscere il numero esatto degli elementi (qui le cifre) coinvolti. Il concetto risulta più evidente ricorrendo alla grafica. La `lista_cifre` viene

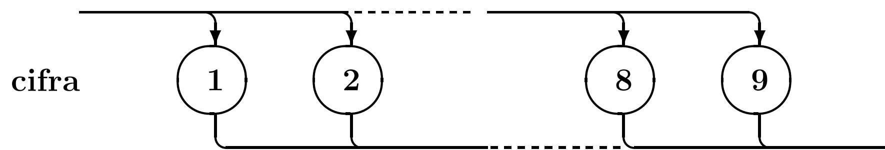

Figura 3: Cifra

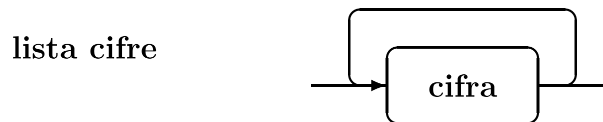

Figura 4: Lista cifre

rappresentata dal seguente diagramma sintattico in figura 4 a pag. 9 in cui è sufficiente seguire le frecce.
Resta da compiere l'ultimo passo, la produzione di numero. Si usano sia simboli non terminali (`lista_cifre`) che terminali . (punto). La regola di produzione si interpreta come concatenazione di una lista_cifre con un . (punto) ed un'altra lista_cifre. Graficamente[4]:

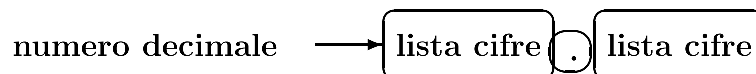

Figura 5: Numero decimale

Il procedimento è del tipo *bottom-up* ovvero si procede dai simboli terminali di base verso l'alto. Il modello è piramidale: al vertice c'è 1 solo simbolo.

Come precedentemente scritto, il cuore di uno script yacc è costituito dalle regole di produzione che descrivono la grammatica che a sua volta formalizza un qualsiasi processo che potrebbe non aver alcuna attinenza con l'informatica. Le regole di produzione hanno la seguente sintassi:

```
simbolo: definizioni
  {
    azione
```

[4]esiste una convenzione grafica: i simboli non terminali sono racchiusi entro rettangoli, quelli terminali entro cerchi

```
  }
  ;
```

Per quanto detto, `simbolo` è di tipo non terminale. Obbligatoria la presenza del carattere : (doppio punto) seguito da almeno 1 tabulatore orizzontale. L'`azione`, in pratica statement C, determina che cosa fare. Va racchiusa tra parentesi graffe e seguita dal carattere `;` (punto-e-virgola). L'azione, graffe comprese, potrebbe anche mancare. Rimane però il `;` (punto-e-virgola) che nel linguaggio C è lo statement che non fa nulla.
Le `definizioni` sono simboli terminali[5] e non. I primi sono trasmessi a yacc o da uno script lex (caso frequente) oppure da un programma scritto alla bisogna (caso più raro). La loro presenza deve essere preannunciata nella prima parte dello script yacc, ove vengono poste le definizioni.
Per convenzione, i simboli non terminali sono scritti in minuscolo, quelli terminali in maiuscolo.

Supponiamo di avere una sveglia software. La sveglia può essere attivata col comando `"set on"` ed ovviamente disattivata con `"set off"`. Inoltre è possibile posizionare l'ora col comando `"time hh:mm"`. Lo script yacc potrebbe essere il presente:

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

%token CLOCK STATE

%%
comandi:  /* nessun comando */
        | comandi comando
        ;
comando: alarm | set
        ;
alarm:  STATE
        {
          (void) printf ("sveglia %s\n", $1 ? "On" : "Off");
        }
        ;
set:    CLOCK
```

[5] in ambiente yacc però il termine *simbolo terminale* non viene usato in quanto più appropriatamente si parla di *token*

```
        {
          (void) printf ("ora allarme: %02d:%02dn",
$1 / 60, $1 % 60);
        }
        ;
```

Visto che le azioni prevedono l'uso della funzione `printf` è necessario provvedere all'include dello header `stdio.h`. La direttiva di pre-processing C `#include` viene posta tra `%{` e `%}` nella prima sezione dello script così da assicurarsi la sua presenza prima della funzione `yyparse`. In maniera analoga a quanto succede negli script lex, il materiale nell'ultima parte dello script viene copiato pari pari nel file costruito da yacc, ma dopo la funzione `yyparse` analogamente a quanto succede per lex secondo lo schema a pag. 4.

La direttiva `%token` elenca i simboli terminali usati dalla grammatica. Per default sono di tipo `int` anche se è possibile operare delle modifiche.

Le prime due regole stabiliscono che in input al programma o non viene passato alcunché oppure può essere passato ripetutamente 1 comando. Questo o posiziona la sveglia (`alarm`) oppure l'ora della sveglia (`set`).

Il primo comando parte da 1 token (simbolo terminale): `STATE`. Per accedere al valore dei token yacc mette in relazione il primo simbolo -in questo caso `STATE`- con `$1`, il secondo -se c'è- con `$2` e via discorrendo. Ovvero: i simboli terminali non vengono non vengono **mai** riferiti direttamente, ma sempre e solo attraverso il *meccanismo del $*. La regola di produzione è molto semplice: viene stampato il valore del simbolo terminale `STATE`, riferito come `$1`.

Pure nel secondo comando, quello che *produce* l'ora della sveglia, viene usato 1 token -`CLOCK`- al quale si accede col meccanismo del `$`. Anche in questo caso la regola di produzione è molto semplice: il valore del token -riferito con `$1`- viene scisso aritmeticamente nelle componenti ore e minuti che poi vengono stampate in maniera appropriata.

lex e yacc

Molto spesso le utility lex e yacc vengono usate congiuntamente. Entrambe leggono il rispettivo script e generano una funzione C. Delle 2 yacc è la più importante perché stabilisce quali sono i simboli terminali di cui ha bisogno e sa come utilizzarli. Il compito di procurarli è demandato a lex. Tramite la

coppia "Regular Expression - azione" lex individua i token richiesti e li passa al richiedente. Un qualsiasi token è caratterizzato da:

1. nome
2. valore

Il nome corrisponde ad uno di quelli citati nella direttiva `%token` presente nello script yacc ed a cui viene assegnato progressivamente un numero a partire da 258. Il valore, che per default è di tipo `int`, deve essere calcolato da lex e posto nella variabile `yylval`. Questa è definita da yacc e può essere usata la lex che la riconosce con classe di memoria `extern`. Si suppone che in uno script yacc ci sia:

```
%token INTEGER
```

perché è previsto un simbolo terminale contenente un valore intero. yacc trasforma la stringa `INTEGER` in un numero tramite la direttiva di pre-processing `#define`. Nello script lex le cose possono essere arrangiate nel seguente modo:

```
%{
#include <stdlib.h>
%}
NUMBER      [0-9]+
%%
{NUMBER}    {
              yylval = atoi (yytext);
              return (INTEGER);
            }
```

qui `NUMBER` è una macro testuale che intercetta una qualsiasi stringa costituita da cifre. Il materiale intercettato viene posto nella stringa `yytext`. Le cifre lette, quindi, sono trattate come caratteri, non come entità numerica. Per la conversione da stringa a numero si può utilizzare la funzione `atoi` il cui prototipo è definito nello header `stdlib.h`, che va incluso. Il risultato della conversione va posto in `yylval`. A yacc bisogna far sapere che è stato individuato un token `INTEGER` il cui valore si trova in `yylval`. A lex bisogna far sapere che `INTEGER` deve essere sostituito con 258 o più in generale col numero che gli viene assegnato da yacc[6]. Esistono 2 modi per ottenere la cooperazione:

[6] attenzione a non confondere il numero che contraddistingue ciascun token col valore del token!

Metodo 1.

viene dapprima invocato yacc con l'opzione `-d`. yacc trasforma le keyword `%token` in direttive di C pre-processing `#define` e le scrive nel file di tipo *header* `y.tab.h`. Viene generata pure la funzione `yyparse` scritta nel file `y.tab.c`. Poi si lancia lex nel cui script viene esplicitamente incluso lo header creato da yacc. lex produce il file `lex.yy.c` contenente la funzione `yylex`. Per costruire l'eseguibile si compilano entrambi i file C (`y.tab.c` e `lex.yy.c`) facendo attenzione all'ordine con cui si fanno intervenire le librerie accessorie di yacc (per prima) e lex (per seconda). L'ordine di citazione delle librerie è **tassativo**. Visto che entrambe le utility sono in grado di generare la funzione main, è necessario che venga generata per prima quella fornita da yacc[7]. Infatti, il main fornito da lex chiama la funzione `yylex`, mentre quello fornito da yacc chiama la funzione `yyparse` che a sua volta richiama `yylex`: ciò che in definitiva serve.

Ricapitolando: si vuole creare un eseguibile, supponiamo `esempio`, partendo da uno script lex (`esempio.l`) e da uno script yacc (`esempio.y`). La trafila dei comandi è la seguente:

- `yacc -d esempio.y`
  vengono generati il file `y.tab.c` che contiene la funzione `yyparse` e lo header `y.tab.h`
- `lex esempio.l`
  viene generato il file `lex.yy.c` che include esplicitamente lo header `y.tab.h`
- `gcc -O2 y.tab.c lex.yy.c -o esempio -s -ly -ll`
  generazione dell'eseguibile. Se non c'è una funzione main né in `lex.yy.c` né in `y.tab.c` il main viene supplito dalla libreria yacc

Metodo 2.

prima viene invocato lex che produce la funzione `yylex` ponendola in un file -nome di default: `lex.yy.c`- che poi viene incluso esplicitamente (tramite direttiva `#include`) nello script yacc. Quindi si invoca yacc che crea la funzione `yyparse` ponendola in un ulteriore file -nome di default: `y.tab.c`- che va fatto compilare. Devono essere usate le libreria di yacc `-ly` per prima e di lex `-ll` poi, come spiegato in precedenza.

[7] in caso di *object* file omonimi, viene usato quello incontrato per primo

Ricapitolando: si vuole creare un eseguibile, supponiamo `esempio`, partendo da uno script lex (`esempio.l`) e da uno script yacc (`esempio.y`). La trafila dei comandi è la seguente:

- `lex esempio.l`
  viene generato il file `lex.yy.c` che contiene la funzione `yylex`
- `yacc esempio.y`
  viene generato il file `y.tab.c`. Lo script yacc include esplicitamente il file `lex.yy.c`
- `gcc -O2 y.tab.c -o esempio -s -ly -ll`
  generazione dell'eseguibile. Se non c'è una funzione main né in `lex.yy.c` né in `y.tab.c` il main viene supplito dalla libreria yacc

Visto che sono presenti due possibilità equivalenti, quale scegliere? Non c'è nessun motivo per preferire l'una all'altra. Gli eseguibili generati con le due metodologie sono funzionalmente equivalenti anche se hanno dimensioni che differiscono di qualche byte.

Spesso è necessario scrivere la funzione main di proprio pugno. In tal caso si deve aggiungere la funzione `yyerror`, chiamata in caso d'errore. Anche in questo caso o si usa quella fornita dalla libreria lex (che quindi va linkata durante la creazione dell'eseguibile) oppure la si codifica in proprio secondo la falsariga che si trova a pag. 5.

In pratica

Per chiarire le idee, si immagini di avere l'ipotetica sveglia software già usata in precedenza. La sveglia opera in ambiente *case sensitive* ed è in grado di comprendere le seguenti operazioni:

1. alarm off
2. alarm on
3. alarm show
4. set `=` hh`[`.mm`]`
5. set `[-+]` hh`[`.mm`]`
6. quit

Le prime tre operazioni sono considerate autoesplicative. La quarta posiziona l'ora di attivazione della sveglia in maniera assoluta. La successiva posiziona la sveglia in maniera relativa: l'ora viene incrementata algebricamente della quantità indicata. In realtà la sveglia è *gentile*: `alarm` e `set` non servono. Ovvero: rendono più esplicito il comando, ma possono essere omessi del tutto. `quit` serve ovviamente per uscire.
Seguono alcuni esempi esplicativi di posizionamento della sveglia:

comando	*azione*
set = 17	la sveglia viene posizionata alle ore 17:00
set = 0.5	la sveglia viene posizionata alle ore 00:05
set +4	la sveglia viene posizionata in avanti di 4 ore
set -2.20	ls sveglia viene retrocessa di 2h 20'

Inizialmente la sveglia non è attivata. Prima di posizionare una qualsiasi ora è necessario attivarla e l'allarme è posto convenzionalmente alle ore 00:00. Pure la disattivazione pone l'ora alle ore 00:00. Si tenga presente che vengono diagnosticati i tentativi di attivare la sveglia già attivata, così come la disattivazione di una sveglia già posta in stato di off. Nell'impostazione dell'ora, le ore ed i minuti devono essere compresi nell'intervallo `[0-23]`, rispettivamente `[0-59]`.
In fondo al documento sono riportati i sorgenti dello script lex, di quello yacc e dello header di progetto. Tra le due modalità di cooperazione è scelta la seconda. Quindi, per ottenere l'eseguibile i passi da eseguire sono:

```
lex alarm.l
yacc alarm.y
gcc -ansi -pedantic y.tab.c -o alarm -s -ly -ll
```

Ottenuto l'eseguibile, non resta che usarlo. Enjoy with it!

Sorgente del file alarm.y

```
%{
#include <stdio.h>
#include "alarm.h"
void avanti (int);
void indietro (int);
void posiziona (int);
int alarm_state = not_set, alarm_time, hour, minute;
%}
```

```
%token STATE TIME
%%
comands:
        | comands comand
        ;
comand: alarm | set;
alarm:
        STATE
        {
          switch ($1)
            {
              case show:
                if (alarm_state == not_set)
                  (void) printf ("\talarm not yet setted\n");
                else
                  (void) printf ("\talarm is %s\n",
alarm_state ? "On" : "Off");
                  if (alarm_state == on)
                    (void) printf ("\talarm time setted to "
"%02d:%02d\n", alarm_time / 60, alarm_time % 60);
                break;
              case off:
              case on:
                if (alarm_state == $1)
                  (void) printf ("\talarm already %s\n",
alarm_state ? "On" : "Off");
                else
                  {
                    alarm_state = $1;
                    (void) printf ("\talarm set to %s\n",
alarm_state ? "On" : "Off");
                    if (alarm_state)
                      (void) printf ("\talarm time setted to "
"%02d:%02d\n", alarm_time / 60, alarm_time % 60);
                    else
                      alarm_time = 0;
                  }
                break;
            }
        }
        ;
```

```
set:
        TIME { (void) printf ("operazione non specificata\n");}
        | '+' TIME { avanti ($2);}
        | '-' TIME { indietro ($2);}
        | '=' TIME { posiziona ($2);}
        ;
%%
void avanti (int delta)
{
  int   tmp;
  if (alarm_state != on)
    (void) printf ("\talarm_state is neither setted nor On\n");
  else
    {
      tmp = (delta / 100) * 60 + delta % 100;
      alarm_time = (alarm_time + tmp) % 1440;
      (void) printf ("\talarm time setted to %02d:%02d\n",
alarm_time / 60, alarm_time % 60);
    }
}
void indietro (int delta)
{
  int   tmp;
  if (alarm_state != on)
    (void) printf ("\talarm_state is neither setted nor On\n");
  else
    {
      tmp = (delta / 100) * 60 + delta % 100;
      alarm_time -= tmp;
      if (alarm_time < 0)
        alarm_time += 1440;
      (void) printf ("\talarm time setted to %02d:%02d\n",
alarm_time / 60, alarm_time % 60);
    }
}
void posiziona (int delta)
{
  if (alarm_state != on)
    (void) printf ("\talarm_state is neither setted nor On\n");
  else
    {
```

```
      alarm_time = (delta / 100) * 60 + delta % 100;
      (void) printf ("\talarm time setted to %02d:%02d\n",
alarm_time / 60, alarm_time % 60);
    }
}
#include "lex.yy.c"
```

Sorgente del file alarm.l

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "alarm.h"
int out_of_range (int val, int inf, int sup)
{
  if (val < inf || val > sup)
    return (1);
  else
    return (0);
}
%}
DG      [0-9]{1,2}
DG2     [0-9]{2}
%%
        int    hour, minute, nv;
on|off|show     {
                  if (! strcmp (yytext, "on"))
                    yylval = on;
                  else
                    if (! strcmp (yytext, "off"))
                      yylval = off;
                    else
                      yylval = show;
                  return (STATE);
                }
[-+=]           return (yytext [0]);
{DG}(:{DG2})?   {
                  nv = sscanf (yytext, "%d:%d", &hour, &minute);
                  if (out_of_range (hour, 0, 23))
```

```
                    {
                      yyerror ("valore ore incongruo");
                      exit (EXIT_FAILURE);
                    }
                  if (nv == 2)
                    {
                      if (out_of_range (minute, 0, 59))
                        {
                          yyerror ("valore minuti incongruo");
                          exit (EXIT_FAILURE);
                        }
                    }
                  else
                    minute = 0;
                  yylval = hour * 100 + minute;
                  return (TIME);
                }
quit            return (0);
.|\n            ;
```

Sorgente del file alarm.h

```
#ifndef  ALARM_H
#  define  ALARM_H
   enum {not_set = -1, off = 0, on, show};
#endif
```

Approfondimento

Come è noto, la cooperazione tra lex e yacc si basa sul fatto che yacc determina i token di cui abbisogna e stabilisce per ciascuno di essi un numero univoco. lex, tramite le Regular Expressions, filtra i dati in ingresso e restituisce per il token pertinente il valore desunto dall'input, che però ha una grossa limitazione: deve essere di tipo `int`.
Un esempio semplice per chiarire meglio le idee. Supponiamo che yacc richieda un token il cui nome simbolico è `EVEN` ed a cui deve corrispondere un numero pari. Nello script yacc troviamo:

```
%token EVEN
```

La direttiva `EVEN` viene trasformata in una direttiva di pre-processing (terminologia del linguaggio C) del tipo:

```
#define  EVEN  †
```

ove † è del tutto arbitrario (ad ogni modo è maggiore di 256).

Nello script lex troviamo:

```
[0-9]*[02468]   yylval = atoi (yytext); return (EVEN);
```

Tramite i metodi per la cooperazione di yacc con lex, visti in precedenza, è possibile far sapere a lex di sostituire `EVEN` con †.

Ma alle volte il token deve avere una tipizzazione diversa da quella default. Ci sono percorsi alternativi da seguire. La scelta dipende dalle circostanze. Se ad esempio il valore di tutti i token è di un determinato tipo, allora nella prima sezione dello script yacc, quella delle `"definizioni"`, basta inserire:

```
#define YYSTYPE †
```

ove † è il tipo desiderato. Ciò comporta attenzione nella stesura dello script lex. Se ad esempio i token devono essere di tipo `double` la funzione `atoi` va sostituita con `atof`. Ma questo è solo un piccolo avvertimento.
Alle volte è necessario restituire a yacc token tipologicamente differenti. Si ricorre allora ad una direttiva lex chiamata `%union` che mima il comportamento dell'omonimo costrutto del linguaggio C. Questa permette di condividere un'area di memoria tra diversi dati tipologicamente diversi, ma senza intersezioni. Ovvero, se in una union memorizzo un dato `double` poi devo utilizzarlo come tale. Successivemente posso memorizzarci un dato di tipo `char` ed utilizzarlo per quello che è adesso e non più come double. Se ad esempio è necessario restituire un valore intero (token `INTEGER`) oppure una stringa di soli caratteri (token `STRINGA`) si procede nel seguente modo. Nello script yacc si crea una union tramite l'omonima direttiva lex `%union`. Sarà costituita da 2 membri: un intero e da una stringa di caratteri di lunghezza adeguata:

```
%union
  {
    char    parola [40];
    int     numero;
  }
```

In tal modo `yylval` non è di tipo `int`, bensì una union. C'è un impatto pure sulla direttiva `token`. Non basta citare i nomi dei token. È necessario agganciare al nome del token i membri della union. La sintassi prevista è:

```
%token <nome_membro> nome_token
```

e quindi

```
%token <parola> STRINGA
%token <numero> INTEGER
```

poi però ai token si fa riferimento nel solito modo (metodo del $, spiegato a pag. 11).
Nello script lex si deve tener conto che `yylval` ora è una union. Il valore tratto dall'input va posto nei membri e quindi si usa la notazione `yylval.numero` e `yylval.parola` rispettivamente. Quindi, per restituire a yacc i token si procede nel seguente modo:

```
[a-zA-Z]+       {
                  (void) strcpy (yylval.parola, yytext);
                  return (STRINGA);
                }

[0-9]+          {
                  (void) sscanf (yytext, "%d", &yylval.numero);
                  return (INTEGER);
                }
```

A titolo d'esempio, vengono riportati i sorgenti degli script yacc e lex con i quali al primo vengono passati o una stringa di soli caratteri oppure un valore intero. Gli script hanno valenza didattica (i token sono semplicemente stampati).

Sorgente del file mix.y

```
%{
#include <stdio.h>
%}
%union
  {
    char  parola [40];
```

```
    int   numero;
  }
%token <numero> INTEGER
%token <parola> STRINGA
%%
righe:  |
        righe riga
        ;
riga:   testo | intero
        ;
testo:  STRINGA
        {
          (void) printf ("%s\n", $1);
        }
        ;
intero: INTEGER
        {
          (void) printf ("%d\n", $1);
        }
        ;
%%
#include "lex.yy.c"
```

Sorgente del file mix.l

```
%{
#include <stdio.h>
#include <string.h>
int min (int, int);
%}
%%
[0-9]+          {
                  (void) sscanf (yytext, "%d", &yylval.numero);
                  return (INTEGER);
                }
[a-zA-Z]+       {
                  memset (yylval.parola, (int) '\0',
sizeof (yylval.parola));
                  (void) strncpy (yylval.parola, yytext,
min (yyleng, sizeof (yylval.parola)));
```

```
                  return (STRINGA);
                }
.|\n            ;
%%
int min (int alfa, int omega)
{
  if (alfa <= omega)
    return (alfa);
  else
    return (omega);
}
```

In una `%union` possono essere inseriti anche dati tipologicamente complessi come, ad esempio, vettori e strutture (il costrutto `struct` del linguaggio C).

Nel primo caso (vettori), che è anche quello più semplice, si tenga presente che si debbono usare gli indici. Qui di seguito sono riportati gli script yacc e lex, necessariamente di tipo scolastico (= semplice), in cui yacc richiede un token composto da sole cifre e gli vengono passati indietro non solo il dato numerico desunto dall'input ma anche il numero delle cifre intercettate.

Sorgente del file array.y

```
%{
#include <stdio.h>
#include "beauty_number.c"
%}
%union
  {
    int  vect [2];
  }
%token <vect> INTERO
%%
cmds:
        | cmds cmd
        ;
cmd:    INTERO
```

```
        {
          (void) printf ("%d cifre: ", $1 [1]);
          beauty_number ($1 [0], $1 [1]);
        }
        ;
```

Sorgente del file array.l

```
%{
#include <stdlib.h>
#include "y.tab.h"
%}
%%
[0-9]+  {
          yylval.vect [0] = atoi (yytext);
          yylval.vect [1] = yyleng;
          return (INTERO);
        }
q|quit  return (0);
.|\n    ;
```

La funzione `beauty_number` stampa il numero inserendo i punti per aumentare la leggibilità. Non è riportata, poiché si tratta di una mera funzione C in cui non sono presenti spunti od agganci a lex né tantomeno a yacc.

Più complessa l'inserzione di una struttura (il costrutto `struct` del linguaggio C). La definizione della struct non è ammessa nel corpo della direttiva `%union` e va quindi posta in un apposito file che deve essere incluso sia nello script yacc che in quello lex. In quest'ultimo, poi, l'inclusione deve **precedere** quella eventuale del file `y.tab.h`. Se questa non c'è allora c'è quella di `lex.tab.c` nel file prodotto da yacc. Ma così facendo, ci sono 2 inclusioni del file che va quindi congegnato in modo da non creare doppioni. Qui di seguito sono riportati gli script yacc e lex nonché lo header che contiene la definzione della struttura. In questo esempio yacc chiede a lex un'ora (composta da ore,: (doppio punto), minuti). lex restituisce indietro le due entità separate che yacc poi stampa in maniera appropriata.

Sorgente del file struct.y

```
%{
#include <stdio.h>
#include "struct.h"
%}
%union
  {
    Tempo  adesso;
  }
%token <adesso> INTERO
%%
cmds:
        | cmds cmd
        ;
cmd:    INTERO
        {
          (void) printf ("%02dh %02d\'\n", $1.ora, $1.minuto);
        }
        ;
```

Sorgente del file struct.l

```
%{
#include <stdlib.h>
#include "struct.h"
#include "y.tab.h"
%}
%%
[0-9]{1,2}:[0-9]{1,2}   {
                          (void) sscanf (yytext, "%d:%d",
&yylval.adesso.ora, &yylval.adesso.minuto);
                          return (INTERO);
                        }
q|quit                  return (0);
.|\n                    ;
```

Sorgente del file struct.h

```
#ifndef STRUCT_DEF
#  define STRUCT_DEF
```

```
   typedef
     struct
       {
         int  ora;
         int  minuto;
       }
     Tempo;
#endif
```