Corso di Laurea Magistrale in Ingegneria Informatica
A.A. 2013-2014

Linguaggi Formali e Compilatori

LEX, FLEX, JLEX

Giacomo PISCITELLI

LEX/FLEX/JLEX

Poiché la trasformazione di espressioni regolari in automi a stati finiti deterministici e la implementazione di questi ultimi sono processi meccanici (e noiosi), spesso si utilizza un **generatore automatico** di analizzatori lessicali.

Per la generazione di analizzatori lessicali si utilizza spesso il tool **LEX**, che è un noto generatore di scanner di Unix, progettato specificamente per essere utilizzato insieme allo **YACC**. Infatti molte assunzioni del codice generato da Lex si sposano bene con quelle dello Yacc. Per esempio l'analizzatore lessicale prodotto da Lex è una funzione C chiamata `yylex()`, che è esattamente quella che si aspetta Yacc per l'analizzatore lessicale.

Lex fu sviluppato da M. E. Lesk and E. Schmidt degli AT&T Bell Laboratories e costruisce uno scanner in C da un insieme di espressioni regolari che definiscono i token.

L'input è un file contenente token definiti usando delle espressioni regolari. Lex produce un intero *scanner module* che può essere compilato e linkeditato con altri moduli del compilatore.

LEX/FLEX/JLEX

FLEX (**F**ast **LEX**ical analyzer generator), originariamente scritto in C da Vern Paxson nel 1987, è un free software alternativo a Lex e rappresenta una versione più recente e più veloce di Lex.

Esso è frequentemente usato con **Bison**, un **parser generator** alternativo a sua volta a Yacc.

Lex è distribuito con il sistema operativo Unix mentre Flex è un prodotto della Free Software Foundation, Inc.

JLEX è una versione Java di Lex. Si genera uno scanner Java (le espressioni regolari sono molto simili a quelle utilizzate da Lex e Flex). Spesso usato in coppia con **CUP** *(Constructor of Useful Parsers)* come alternativa a YACC/BISON.
www.cs.princeton.edu/~appel/modern/java/CUP/index.html **(attualmente mantenuto da Techn. Univ. of Munich)**
www.cs.princeton.edu/~appel/modern/java/JLex/

Lex, Flex e JLex sono in gran parte non procedurali. Non c'è bisogno di dire come gli strumenti devono eseguire la scansione. Tutto ciò che serve è dire ciò che si vuole scandire, dando la definizione dei token validi. Questo approccio semplifica notevolmente la costruzione di uno scanner, dal momento che la maggior parte dei dettagli di scansione (I/O, buffering, ecc.), sono gestiti automaticamente.

FLEX: descrizione e funzionamento

Flex è, quindi, un generatore che accetta in ingresso (in un file con suffisso **.l**, f.e. **prova.l**) un insieme di espressioni regolari (*rules*) e di azioni (*actions*) associate a ciascuna espressione (sotto forma di codice C), producendo in uscita una routine di riconoscimento lessicale **yylex()** (contenuta nel file **lex.yy.c**) in grado di identificare e restituire le singole "parole" che il linguaggio ammette.

Il file **lex.yy.c**, privo di **main()** (che viene definito implicitamente, se lo scanner funziona da solo, grazie all'opzione di compilazione `-lfl`), contiene la routine di scanning **yylex()** assieme ad altre routine ausiliari e macro.

Il file prodotto viene compilato e fuso con la libreria `-lfl` per generare un eseguibile.

```
lex prova.l
cc lex.yy.c -o prova -lfl
```

Quando il file eseguibile (**prova**) viene mandato in esecuzione, esso analizza il/i file in input per individuare le occorrenze di espressioni regolari conformi a quelle definite. Se ne riconosce una, esegue il codice C associato.

I link utili per FLEX

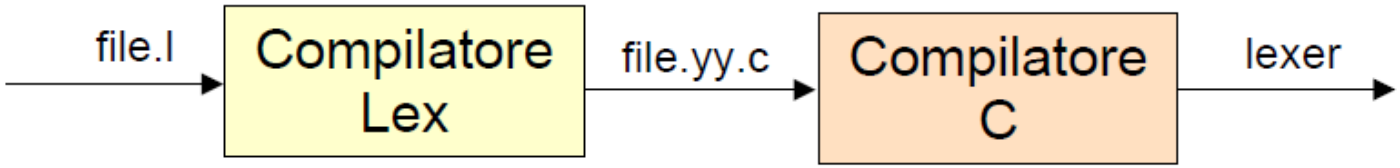
http://flex.sourceforge.net/ flex home page

http://www.gnu.org/software/flex/manual/html_node/flex_toc.html
flex manual

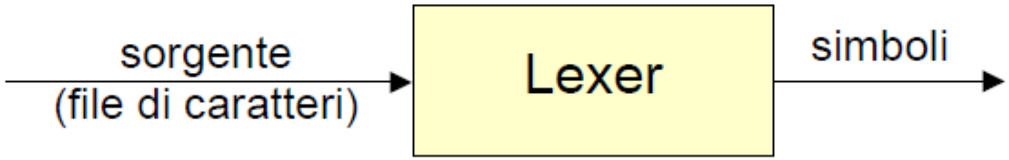
http://www.quut.com/c/ANSI-C-grammar-l-1998.html
ANSI C grammar (LEX specification)

FLEX: descrizione e funzionamento

- Doppia valenza < linguaggio / compilatore



- Teleologicamente:



FLEX: La struttura del programma generato

Flex produce un programma C, privo di **`main()`** il cui punto di accesso è dato dalla funzione **`int yylex()`**.

Tale funzione legge dal file **`yyin`** e ricopia sul file **`yyout`** il testo non riconosciuto.

Se non specificato diversamente nelle azioni (tramite l'istruzione **`return`**), tale funzione termina solo quando l'intero file di ingresso è stato analizzato.

Al termine di ogni azione l'automa si ricolloca sullo stato iniziale pronto a riconoscere nuovi simboli.

Per default, i file **`yyin`** e **`yyout`** sono inizializzati rispettivamente a **`stdin`** e **`stdout`**.

Il programmatore può alterare questa situazione re-inizializzando tali variabili globali.

Espressioni regolari in FLEX

Lex utilizza per la specifica dell'analizzatore lessicale le espressioni regolari, che sono un formalismo più efficiente delle grammatiche ma meno potente.

La distinzione tra grammatiche e espressioni regolari sta nel fatto che le espressioni regolari non sono in grado di riconoscere strutture sintattiche ricorsive, mentre questo non è un problema per le grammatiche.

Una struttura sintattica come le parentesi bilanciate, che richiede che le parentesi aperte siano nello stesso numero di quelle chiuse non può essere riconosciuta da un analizzatore lessicale, e per questo scopo si ricorre all'analizzatore sintattico.

Invece costanti numeriche, identificatori o keyword vengono normalmente riconosciute dall'analizzatore lessicale.

Espressioni regolari in FLEX

Le **espressioni regolari** descrivono sequenze di caratteri ASCII ed utilizzano un certo numero di operatori:

" \ [] ^ - ? . * + | () $ / {} % <>

Lettere e numeri del testo di ingresso sono descritti mediante se stessi:

l'espressione regolare **val1** rappresenta la sequenza **'v' 'a' 'l' '1'** nel testo di ingresso

I caratteri non alfabetici vengono rappresentati in Lex racchiudendoli tra doppi apici, per evitare ambiguità con gli operatori:

l'espressione **xyz"++"** rappresenta la sequenza **'x' 'y' 'z' '+' '+'** nel testo di ingresso

I caratteri non alfabetici possono essere anche descritti facendoli precedere dal carattere ****

l'espressione **xyz\+\+** rappresenta la sequenza **'x' 'y' 'z' '+' '+'** nel testo di ingresso**.**

Espressioni regolari in FLEX

Le classi di caratteri vengono descritte mediante gli operatori **[]**

l'espressione **[0123456789]** rappresenta una cifra nel testo di ingresso**.**

Nel descrivere classi di caratteri, il segno **-** indica una gamma di caratteri:

l'espressione **[0-9]** rappresenta una cifra nel testo di ingresso

Per includere il carattere **-** in una classe di caratteri, questo deve essere specificato come primo o ultimo della serie:

l'espressione **[-+0-9]** rappresenta una cifra o un segno nel testo d'ingresso**.**

Nelle descrizioni di classi di caratteri, il segno **^** posto all'inizio indica una gamma di caratteri da escludere:

l'espressione **[^0-9]** rappresenta un qualunque carattere che non sia una cifra nel testo di ingresso

L'insieme di tutti i caratteri eccetto il fine riga (new line) viene descritto mediante il simbolo "**.**"

Il carattere di fine riga viene descritto dal simbolo **\n**

Il carattere di tabulazione viene descritto dal simbolo **\t**

Espressioni regolari in FLEX

L'operatore **?** indica che l'espressione ***precedente*** è opzionale:

ab?c indica sia la sequenza **ac** che **abc**

L'operatore ***** indica che l'espressione ***precedente*** può essere ripetuta 0 o più volte:

ab*c indica tutte le sequenze che iniziano per **a**, terminano per **c** e hanno all'interno un numero qualsiasi di **b**

L'operatore **+** indica che l'espressione ***precedente*** può essere ripetuta 1 o più volte:

ab+c indica tutte le sequenze che iniziano per **a**, terminano per **c** e hanno all'interno almeno una **b**.

L'operatore **|** indica un'alternativa tra due espressioni:

ab|cd indica o la sequenza **ab** o la sequenza **cd**

Le parentesi tonde **( )** consentono di esprimere la priorità tra operatori:

(ab|cd+)?ef indica sequenze tipo **ef**, **abef**, **cdddef**.

FLEX: principali espressioni regolari supportate

`x`	il carattere 'x'
`.`	ogni carattere eccetto '\n'
`[xyz]`	una classe di caratteri; in questo caso, 'x', 'y' o 'z'
`[a-z]`	una classe con un range; ogni carattere compreso tra 'a' e 'z'
`[^A-Z]`	una classe negata: ogni carattere NON nella classe
`r*`	zero o più r, dove r è un'altra espressione regolare
`r+`	una o più r
`r?`	zero o una r
`r{2,5}`	tra due a cinque r
`r{2,}`	due o più r
`r{4}`	esattamente 4 r
`{nome}`	l'espansione della definizione di nome
`(r)`	r, parentesizzata per raggruppare
`rs`	concatenazione: r seguita da s
`r|s`	alternativa: r oppure s
`r/s`	restrizione: r ma solo se seguita da s
`^r`	r ma solo a inizio linea
`r$`	r ma solo a fine linea

Formato dell'input file di FLEX

Un file in ingresso a LEX/FLEX è composto di tre sezioni distinte, separate dal simbolo '%%'.

Sezione 1

```
%{
#include
constant definition
scanner macro
%}
basic definitions
```

Sezione 2

```
%%
Token definitions and actions
```

Sezione 3

```
%%
Support procedures C user code
```

Formato dell'input file di FLEX

La Sezione 1 (che può anche essere vuota) contiene:

- racchiuse tra i caratteri `%{` e `%}`, le **`#include`** di libreria, le definizioni di costanti e/o macro personalizzate del programma C che si vuole realizzare; infatti, questa parte di testo verrà letteralmente copiata nel programma C generato; in questa parte, quando lo scanner viene usato in combinazione con il parser, va inserita la **`#include y.tab.h`**, che è il file generato dal parser e che contiene la definizione dei token multi-caratteri ai fini dell'analisi sintattica;
- le definizioni di base usate nella seconda sezione per descrivere una espressione regolare.

La Sezione 2 contiene la definizione dei pattern con le relative azioni da intraprendere, sotto forma di coppie

pattern **azione**

Le azioni devono iniziare sulla stessa riga in cui termina l'espressione regolare e ne sono separate tramite spazi o tabulazioni.

La Sezione 3 (che può anche essere vuota) contiene le procedure di supporto di cui il programmatore intende servirsi per associarle alle azioni indicate nella seconda sezione: se è vuota, il separatore %% viene omesso.

Formato dell'input file di FLEX

Ogni riga della prima sezione il cui primo carattere non sia di spaziatura è una definizione:

```
numero [+-]?[0-9]+
```

L'espressione così definita può essere utilizzata nella seconda sezione racchiudendone il nome tra parentesi graffe:

```
{numero} printf("trovato numero\n");
```

Parti di codice possono essere inserite sia nella prima sezione (inserendole, come si è detto, tra `%{` e `%}`) che nella seconda (inserendolo tra `{ }` immediatamente dopo ogni espressione regolare che si vuole riconoscere), e vengono copiate integralmente nel file di output.

Se, in corrispondenza di un pattern, non viene indicate alcuna azione, quando un tale tipo di token viene riconosciuto nel testo scandito, esso viene scartato.

Le righe presenti nella sezione delle **procedure di supporto** (la terza sezione) del programma sorgente sono ricopiate nel file **`lex.yy.c`** generato da Lex.

Formato dell'input file di FLEX

Esempio 1 di contenuto del file di input

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

%%
[0-9]+                  printf("NUMERO INTERO\n");
[a-zA-Z][a-zA-Z0-9]*    printf("IDENTIFICATORE\n");
%%
```

Questo file descrive 2 pattern, cioè 2 particolari tipi di token: `[0-9]+` a cui è associata l'azione di stampa `NUMERO INTERO` e `[a-zA-Z][a-zA-Z0-9]*` a cui è associata la stampa `IDENTIFICATORE` .

Si noti la presenza, nella sezione 1, della direttiva `#include <stdio.h>` necessaria per consentire l'utilizzo dell'istruzione `printf` .

Questo semplice esempio ipotizza l'utilizzo di LEX/FLEX indipendente da YACC/BISON.

Formato dell'input file di FLEX

Esempio 2 di contenuto del file di input

```
        int num_lines = 0, num_chars = 0;
%%
\n      num_lines=num_lines+1; num_chars=num_chars+1;
.       num_chars=num_chars+1;
%%
main()
        {
        yylex();
        printf( "# of lines = %d, # of chars = %d\n",
                num_lines, num_chars );
        }
```

Questo scanner conta il numero di caratteri e il numero di linee presenti nel file di input e produce nel file di output il valore dei contatori indicati.

Si noti che la prima linea dichiara 2 variabili globali, accessibili sia alla funzione `yylex()` che al `main()` dichiarato dopo il secondo `%%`.

Ambiguità lessicali

Esistono due tipi di ambiguità lessicali:

1. la parte iniziale di una sequenza di caratteri riconosciuta da un'espressione regolare è riconosciuta anche da una seconda espressione regolare;
2. la stessa sequenza di caratteri è riconosciuta da due espressioni regolari distinte.

Nel primo caso verrà eseguita l'azione associata all'espressione regolare che ha riconosciuto la ***sequenza più lunga***.

Nel secondo caso sarà eseguita l'azione associata all'***espressione regolare dichiarata per prima*** nel file sorgente di LEX/FLEX.

ESEMPIO
Dato il file

```
%%
for {return FOR_CMD;}
format {return FORMAT_CMD;}
[a-z]+ {return GENERIC_ID;}
```

e la stringa di ingresso "format", la procedura `yylex` ritorna il valore FORMAT_CMD, preferendo la seconda regola alla prima - perché descrive una sequenza più lunga, e la seconda regola alla terza -perché definita prima nel file sorgente.

Risoluzione delle ambiguità lessicali

Date le regole di risoluzione dell'ambiguità, è necessario definire prima le regole per le parole chiave e poi quelle per gli identificatori.

Il principio di preferenza per le corrispondenze più lunghe può essere pericoloso:

```
'.*' {return QUOTED_STRING;}
```

cerca di riconoscere il secondo apice il più lontano possibile: cosi, dato il seguente ingresso

first ' quoted string here, 'second' here

riconoscerà 36 caratteri invece di 7

Una regola migliore è la seguente:

```
'[^'\n]+' {return QUOTED_STRING;}
```

Azioni associate alle espressioni regolari in LEX

Ad ogni espressione regolare è associata in LEX un'azione che viene eseguita all'atto del riconoscimento.

Le azioni sono espresse sotto forma di codice C: se tale codice comprende più di una istruzione o occupa più di una linea deve essere racchiuso tra parentesi graffe.

L'azione più semplice consiste nell'ignorare il testo riconosciuto: si esprime un'azione nulla con il carattere ';'.

Il testo riconosciuto viene accumulato nella variabile `yytext`, definita come puntatore a caratteri.

Operando su tale variabile, si possono definire azioni più complesse.

Il numero di caratteri riconosciuti viene memorizzato nella variabile yyleng, definita come intero.

Esiste un'azione di default che viene eseguita in corrispondenza del testo non descritto da nessuna espressione regolare: **il testo non riconosciuto viene ricopiato in uscita**, carattere per carattere.

Esempi di input a LEX

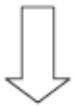
1. *Generazione di linee precedute dal numero d'ordine*:

```
%{
#include <stdio.h>
int l = 1;
%}
linea     .*\n
%%
{linea}   {printf("%d %s", l++, yytext);}
%%
main(){yylex(); return(0);}
```

definizione regolare

regola di traduzione

stringa lessicale



file compilabile ed eseguibile

Esempi di input a LEX

2. *Sostituzione dei numeri da notazione decimale ad esadecimale + stampa numero sostituzioni effettive*

```
%{
#include <stdio.h>
#include <stdlib.h>
int cont = 0;
%}
cifra     [0-9]
num       {cifra}+
%%
{num}   { int n = atoi(yytext);
          printf("%x", n);
          if (n > 9) cont++; }
%%
main()
{
  yylex();
  fprintf(stderr, "Tot sostituzioni = %d\n", cont);
  return(0);
}
```

Oss: Azione di default: quando un carattere (o una stringa di caratteri) non è parte di alcun simbolo

⇩

ECHO sull'output

Esempi di input a LEX

3. *Selezione delle linee che iniziano o terminano con il carattere a*

```
%{
#include <stdio.h>
%}
a_linea     a.*\n
linea_a     .*a\n
%%
{a_linea}   ECHO;
{linea_a}   ECHO;
.*\n        ;
%%
main()
{
  yylex();
  return(0);
}
```

azione vuota

Oss: Insieme di regole ambiguo (una stringa può corrispondere a diverse expreg (es: *a*)

⇩

Regole di priorità (built-in)

1. Principio del maximal munch.
2. Se ∃ più regole che fanno matching con la stringa → seleziona la prima specificata.

```
.*\n          ;
{a_linea}     ECHO;
{linea_a}     ECHO;
```

⇨ output vuoto!

```
{a_linea}     ECHO;
{linea_a}     ECHO;
```

⇨ output = input!

FLEX: uso combinato con BISON

La struttura del programma generato da FLEX

Come detto, eseguendo LEX/FLEX si ottiene come risultato il file **lex.yy.c**, un programma C, privo di **main()**, che contiene la routine di scanning **yylex()** assieme ad altre routine ausiliari e macro.

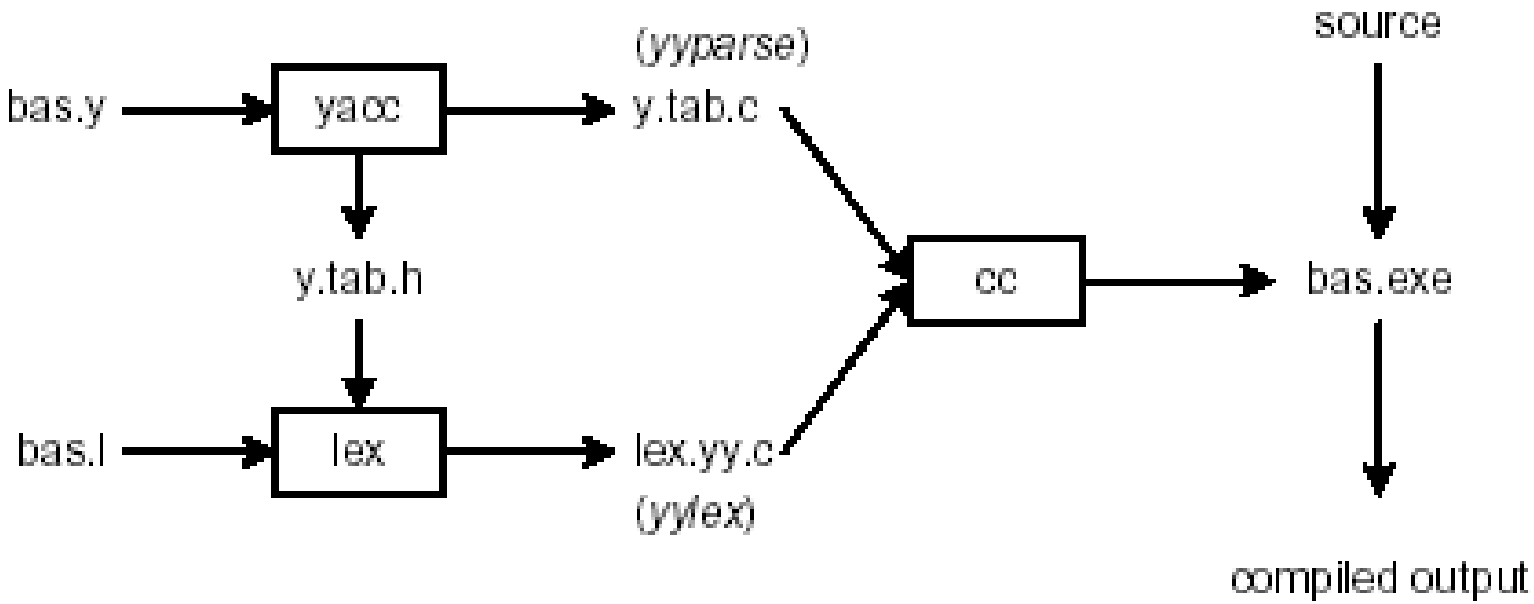
Per default, **yylex()** è dichiarata come:

```
int yylex()
    {
    ... various definitions and the actions in here ...
    }
```

Quando s'intende combinare l'analisi lessicale con quella sintattica, il file **lex.yy.c** (generato dall'esecuzione dello scanner), viene normalmente incluso (con una **#include**) nel sorgente generato da YACC. Infatti molte dichiarazioni, come i token e le strutture dati per comunicare con il parser, vengono dichiarate nel sorgente generato da YACC, **y.tab.c**. La funzione **yylex()** viene richiamata ripetutamente dalla funzione principale del parser, **yyparse()**.

FLEX: uso combinato con BISON

Esempio: compilatore BASIC



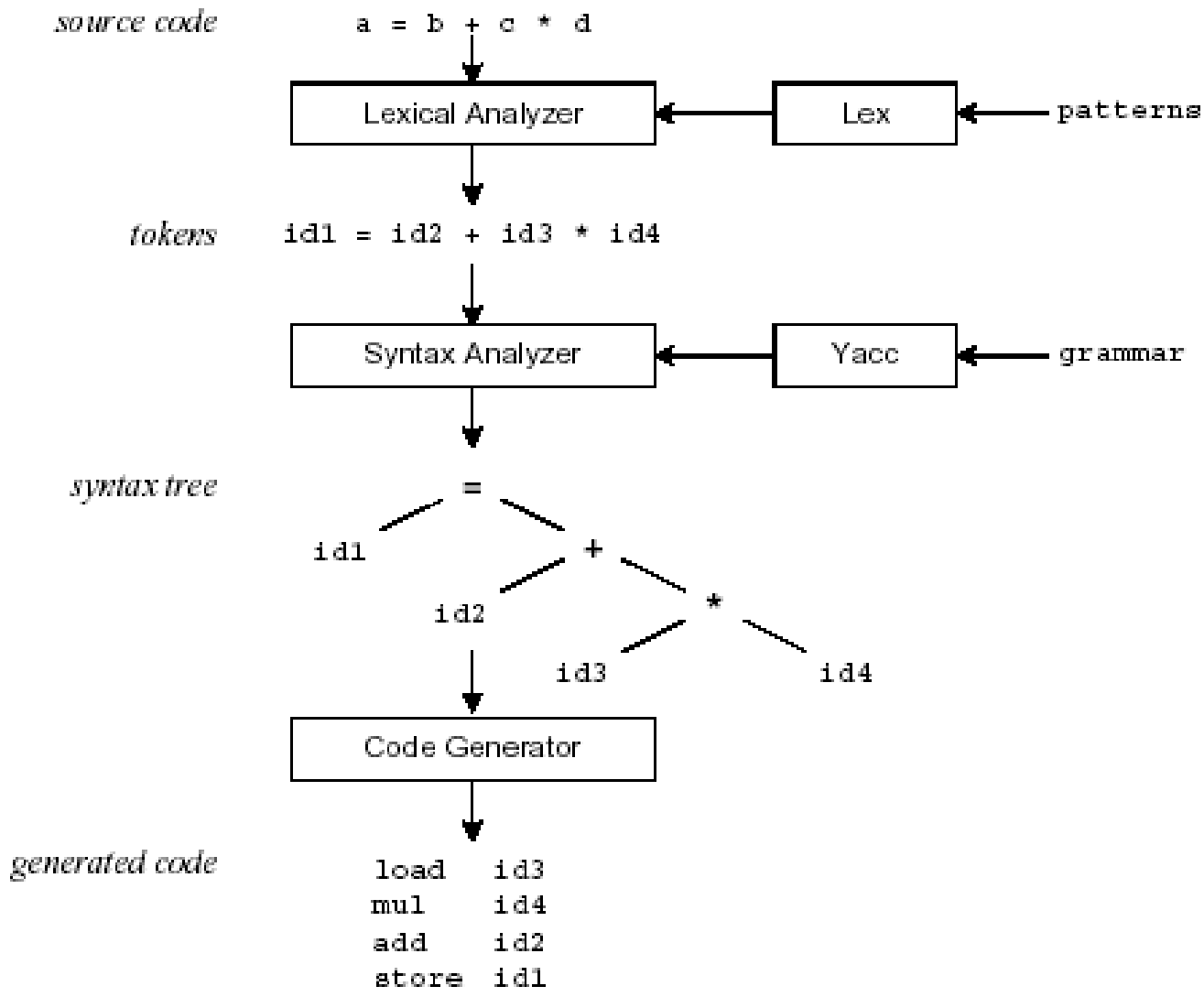
bas.l -> regole lessicali **bas.y** -> regole sintattiche con definizione tokens

CC -> comandi per creare il compilatore **bas.exe** -> compilatore

y.tab.h -> definizioni dei tokens per Lex

```
yacc -d bas.y # create y.tab.h, y.tab.c
lex bas.l # create lex.yy.c
cc lex.yy.c y.tab.c -obas.exe # compile/link
```

FLEX: uso combinato con BISON



FLEX: l'output

Quando viene eseguito lo scanner generato, esso analizza il suo input individuando le stringhe di caratteri che rispettano i ***pattern*** specificati.

Come già osservato, se viene trovato più di un possibile *match*, viene assunto quello che considera il maggior numero di caratteri. Nel caso di 2 match della stessa lunghezza, viene scelto quello che corrisponde alla regola che appare per prima nell'input file di FLEX.

Una volta determinato il match, il corrispondente testo (che rappresenta un ***token***) viene reso disponibile globalmente attraverso il puntatore a carattere (`char *yytext`), e la sua lunghezza viene globalmente riportata sotto forma dell'intero (`int yyleng`).

Viene quindi eseguita l'azione (***action***) corrispondente al pattern individuato, e si procede alla scansione del successivo testo.

FLEX: l'output

Una volta che un token viene riconosciuto, abbiamo varie possibilità: possiamo ignorarlo (come facciamo per gli spazi bianchi) e passare al token successivo; oppure possiamo ritornare il codice del token riconosciuto. Quando viene ritornato un token, la funzione **`yylex()`** termina ma sarà richiamata dal parser quando avrà bisogno di un altro token.

Notiamo che può essere necessario fare qualche azione supplementare oltre a ritornare un token o ignorarlo. Possiamo notare che quando incontriamo un newline, incrementiamo il contatore delle linee in input.

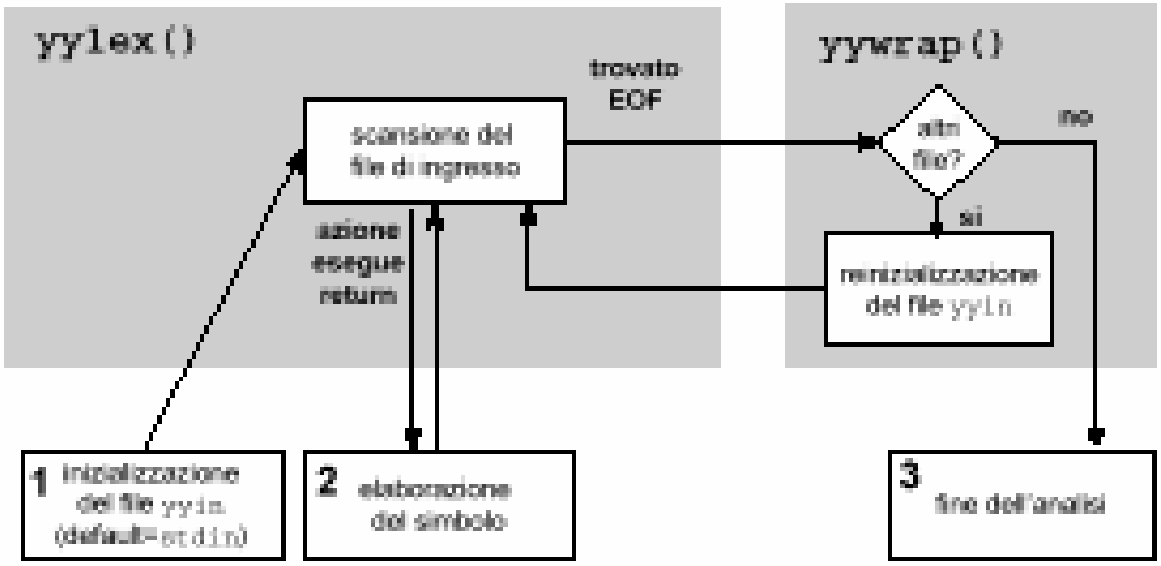
Molto più importante è il fatto che di certi token occorre conoscere qualcosa di più oltre al loro tipo. Per esempio, di una variabile non basta sapere che è una variabile: occorre sapere quale è.

Come già detto, Lex mantiene in un buffer il testo letto dall'input e riconosciuto; tale buffer è accessibile all'utente tramite le variabili **`char *yytext`** e **`int yyleng`**.

FLEX: l'output

FLEX chiama la function **yywrap()** al termine del suo input e fornisce la variabile globale **char *yytext**, che contiene i caratteri del token corrente e la variabile globale **int yyleng**, che contiene la lunghezza di tale stringa.

Se la function **yywrap()** ritorna il valore 0 (false), significa che la funzione prevede che si debba procedere e imposta **yyin** a puntare ad un altro input file, facendo sì che lo scanning proceda da tale nuovo file di input. Se la function **yywrap()** ritorna invece un valore diverso da 0 (true), lo scanner termina e restituisce il valore 0 alla funzione chiamante.



Esercizi esemplificativi

Scrivere un programma LEX che dato in ingresso un programma C, ne produca in uscita uno equivalente ma privo dei commenti.

Si modifichi il programma dell'esercizio precedente in modo che riconosca le direttive **`#include`** e, quando le incontra, segnali un errore e termini l'analisi.

JLEX: la versione JAVA di LEX

Per produrre lo scanner di applicazioni Java, si può far uso del *generatore d'analizzatori lessicali* conosciuto come **JLex**.

Questo software, scritto interamente in Java, produce in uscita delle *classi java* che implementano i *metodi* per effettuare l'analisi lessicale di una stringa in ingresso.

La classe principale che è prodotta è **Yylex**, la quale contiene il metodo **yylex()** che preleva e analizza il successivo token in arrivo.

Un altro metodo contenuto nella classe è **yytext()**, esso ritorna il testo riconosciuto da yylex().

Il JLex per funzionare, ha bisogno in ingresso di un file di s*pecifica*, contenente tutti i dettagli relativi all'analisi lessicale che si vuole realizzare.

Come già detto, spesso Jlex viene usato in coppia con **CUP** *(Constructor of Useful Parsers)* come alternativa a YACC/BYSON.