Generatori di analizzatori lessicali: Lex

Analizzatori lessicali

- Analisi lessicale: riconoscere nella stringa di ingresso gruppi di simboli che corrispondono a specifiche categorie sintattiche.
- La stringa di ingresso è trasformata in una sequenza di simboli astratti: i **token**, che sono passati all'analizzatore sintattico.
- Token: coppia
  - nome: simbolo astratto che rappresenta una categoria sintattica
  - valore: una sequenza di simboli del testo di ingresso

Token	⟨Nome,Valore⟩	⟨`IDE`, **x1**⟩
Nome	Informazione che identifica una classe di token	`IDE`
Valore	Informazione che identifica uno specifico token	**x1**
Pattern	Descrizione generale della forma dei valori di un token (espressione regolare)	(**x**|**y**)(**x**|**y**|0|1)*
Lessema	Una stringa **istanza** di un pattern	**x1**

Analizzatori lessicali

- Esempi di token
  - keywords (IF,…)
  - segni di punteggiatura (;…)
  - operatori costituiti di caratteri singoli e multipli (=, ==, <=, …)
  - identificatori (stringhe che iniziano con una lettera e continuano con lettere e cifre: token IDE)
  - numeri (interi, reali)
  - commenti
- Nella stringa C

```
if (x==0) printf("Zero")
```

potrebbero essere riconosciuti i token:

⟨`IF`⟩, ⟨`(`⟩, ⟨`IDE,x`⟩, ⟨`OPREL,==`⟩, ⟨`CONST-NUM,0`⟩, ⟨`)`⟩, ⟨`IDE,printf`⟩, ⟨`(`⟩, ⟨`CONST-STRING,Zero`⟩, ⟨`)`⟩

Generatori di analizzatori lessicali

- *Input*: insieme di pattern e corrispondente *azione* da eseguire
- *Output*: **programma** che riconosce i pattern nella sua stringa d'ingresso
- LEX: noto tool di Unix per la generazione di analizzatori lessicali (scanner)
- Lex costruisce uno scanner in C da un insieme di espressioni regolari che definiscono i token e un insieme corrispondente di azioni espresse come frammenti di programmi in linguaggio C

Lex: Sorgente

- Struttura
  Definizioni (opzionale)
  %%
  Regole (coppie: espressione regolare azione)
  %%
  Funzioni ausiliarie (opzionale: se la sezione è vuota, il precedente separatore '%%' viene omesso)

- La prima e la terza sezione contengono rispettivamente tutte le dichiarazioni e le procedure utili alle Regole e sono ricopiate nel file C generato da Lex

Lex: Sorgente - Definizioni

- **Definizioni regolari**
  l_1 espressione regolare 1
  l_2 espressione regolare 2
  …
  Il nome $\{l_k\}$ potrà essere usato successivamente per denotare la *k*-esima espressione regolare

- **Codice C delimitato dai simboli speciali `%{` e `%}`** dichiarazioni di variabili e costanti, **`#include`**, prototipi di funzioni, macro Lex

Lex: Sorgente – Espressioni regolari

- Sequenze di caratteri ASCII che utilizzano gli operatori:
  `"\[]^-?.*+|()$/{}%<>`
- Lettere e numeri del testo di ingresso sono descritti mediante loro stessi
- I caratteri non alfabetici sono descritti facendoli precedere dal carattere `\`
  - `xyz\+\+` rappresenta la sequenza 'x' 'y' 'z' '+' '+'
- Classi di caratteri: `[0123456789]`, `[0-9]`
- Negazione di classi di caratteri: `[^0-9]`
- Tutti i caratteri eccetto fine riga: `.`
- Fine riga: `\n`; tabulazione: `\t`

Ulteriori dettagli:

`info flex`

(da Unix command-line)

Lex: Sorgente – Espressioni regolari

- espressione precedente opzionale: `ab?c`
- espressione precedente ripetuta 0 o più volte: `ab*c`
- espressione precedente ripetuta 1 o più volte: `ab+c`
- espressione precedente ripetuta tra *n* e *m* volte: `a{2,5}`
- alternativa tra due espressioni: `ab|cd`
- priorità tra operatori (parentesi): `(ab|cd+)?ef` indica sequenze tipo **`ef`**, **`abef`**, **`cdddef`**
- espressione a inizio/fine linea: `^a` / `a$`
- **lookahead**: `ab/cd` indica la stringa "**`ab`**", ma solo se seguita da "**`cd`**"; "**`cd`**" però non farà parte del testo riconosciuto

Lex: Sorgente – Azioni

- Se il codice C delle azioni comprende più di una istruzione o occupa più di una linea deve essere racchiuso tra `{}`
- Le azioni devono iniziare sulla stessa riga in cui termina l'espressione regolare e ne sono separate tramite spazi o tabulazioni
- carattere `;` : azione nulla (ignora il testo)
- Il testo riconosciuto viene accumulato nella variabile **`yytext`**, array di caratteri di lunghezza **`yyleng`** (**`int`**)
- Il testo non descritto da nessuna espressione regolare viene ricopiato in uscita

Lex: Sorgente – Ambiguità lessicali

- Se la parte iniziale di una sequenza di caratteri riconosciuta da un'espressione regolare è riconosciuta anche da una seconda espressione regolare → Lex sceglie il **match più lungo**
- Se la stessa sequenza di caratteri è riconosciuta da due espressioni regolari distinte → Lex sceglie la **regola dichiarata per prima**

Lex: Output

- Il file **lex.yy.c** prodotto da Lex è privo di **`main()`** e il punto di accesso è dato dalla funzione **`int yylex()`**
- Implementa un automa a stati finiti deterministico, riconoscitore delle espressioni regolari definite nelle regole
- Al termine di ogni azione l'automa si ricolloca sullo stato iniziale pronto a riconoscere nuovi simboli
- Legge dal file **`yyin`** (di default **`stdin`**) e scrive sul file **`yyout`** (di default **`stdout`**); è possibile specificare file di input/output

Lex: Output

- Se non specificato diversamente nelle azioni (tramite l'istruzione **`return`**), tale funzione termina solo quando l'intero file di ingresso è stato analizzato
- **yylex()** chiama **input()** per leggere il (singolo) carattere di input successivo
- quando incontra `EOF`, chiama **yywrap()**
  - Se yywrap restituisce 1, yylex restituisce il token 0 per indicare end of file
  - Se yywrap restituisce 0, indica ulteriore input
    - yyin deve associarsi ad un altro file

FLEX (Fast LEXical analyzer generator)

- **LEX**: Sviluppato da M.E. Lesk e E. Schmidt presso AT&T Bell Labs
- **FLEX**: 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.

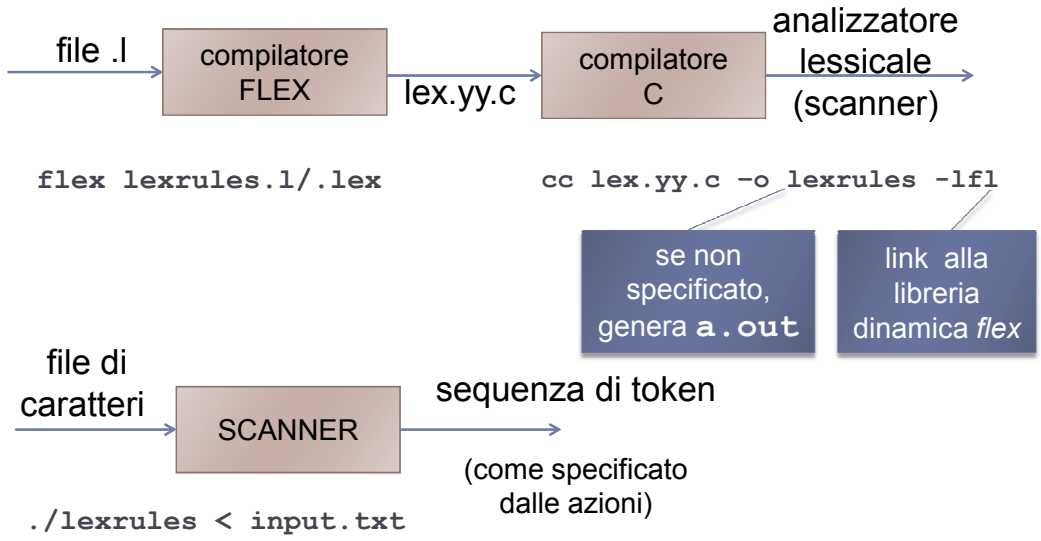
  http://flex.sourceforge.net/

- Nelle distribuzioni Debian:

  sudo apt-get install flex

- Flex++: scanner per C++

FLEX (Fast LEXical analyzer generator)



Lex: Esempio 1: eliminazione spazi

- Il seguente sorgente Lex

```
SPACES    [ \t]+
%%
{SPACES}  ;
```

definisce un pattern chiamato SPACES.

A questo è associata l'azione vuota

Lex: Esempio 2

- Il seguente sorgente Lex

```
%{
#include <stdio.h>
%}
%%
[0-9]          printf("*");
[a-z]+         printf("%c", yytext[yyleng-1]);
```

descrive 2 pattern a cui sono associate azioni di stampa. Lex genera un programma C `lex.yy.c` che, compilato ed eseguito sulla stringa d'ingresso

12caso45casa2

stampa …

Lex: Esempio 3 (ambiguità lessicale)

- Dato il file

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

e la stringa d'ingresso **`format`**, la funzione **`yylex()`** restituisce 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.

Esercizio 1

- Dato l'input:

```
pigreco := 3.14;                    (:= seguito da 1 spazio)
temp1   := pigreco*(temp1+2); (:= seguito da tab)
temp2   := 6.0;
```

- scrivere un programma Lex (Definizioni e Regole) che produca in output la lista di token:

```
<IDE,pigreco>_<:=>_<FRACT,3.14><;>_
<IDE,temp1>_<:=>_<IDE,pigreco><OP2,*>
<(><IDE,temp1><OP2,+><NUM,2><)><;>_
<IDE,temp2>_<:=>_<FRACT,6.0><;>
```


Esercizio 2

- Scrivere un programma Lex che elimina spazi o tab presenti *a fine riga*, e trasforma le rimanenti stringhe (interne) composte di più spazi o tab in un singolo spazio.

Esercizio 3

- Scrivere un programma Lex che conta il numero di caratteri e di linee dell'input e riporta in output solo i conteggi.
  - inserire la stampa di riepilogo dei conteggi nella funzione (ausiliaria) main(), dopo aver invocato yylex().

Esercizio 4

- Scrivere un programma Lex che produca in output le linee lette in input precedute dal numero d'ordine.

Esercizio 5

- Scrivere un programma Lex che traduca numeri in input da notazione decimale ad esadecimale e stampi il numero di sostituzioni *effettive.*