Flex

Enea Zaffanella

enea.zaffanella@unipr.it

29 settembre 2020

Linguaggi, interpreti e compilatori
Laurea Magistrale in Scienze informatiche

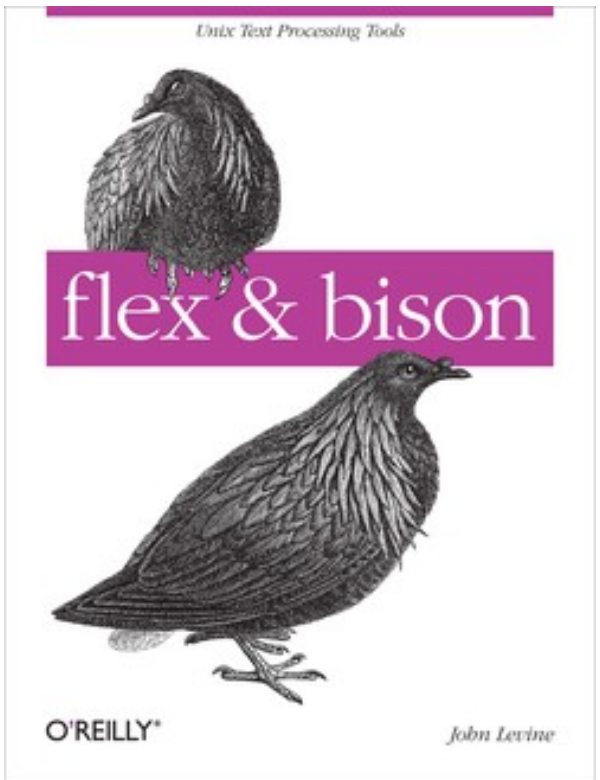
Sommario

Sommario

Lo strumento Flex

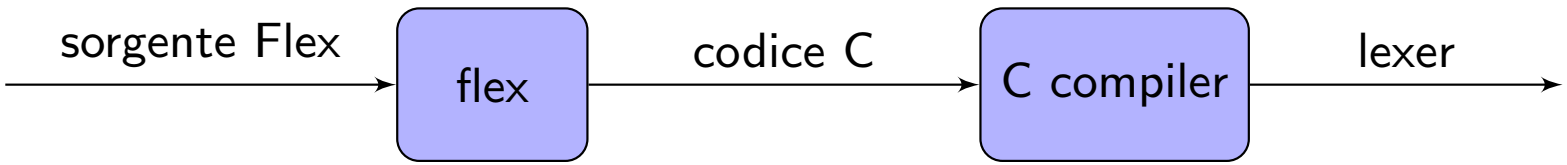
- Generatore di analizzatori lessicali (aka lexer/scanner)
- Versione free di **Lex** (1975)
- Produce codice C (oppure C++)
- Varianti per altri linguaggi (e.g., **JLex** e **JFlex** per Java)

RTFM: Flex (and Bison)



John Levine
flex & bison
O'Reilly, 2009

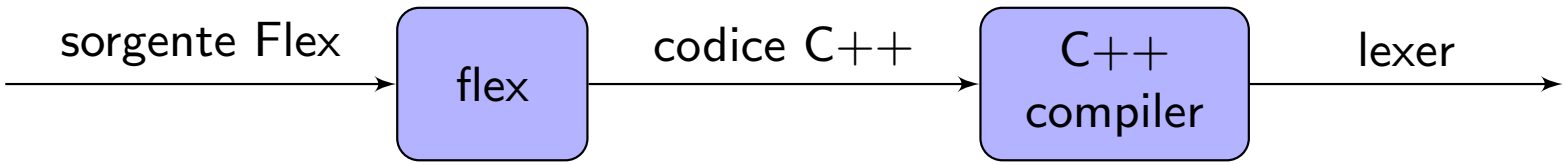
Generazione di un lexer con Flex



Uso tipico

- `flex -o lexer.c lexer.ll`
- `gcc -Wall -Wextra -o lexer lexer.c`
- `lexer < input_scanner`

Supporto per il C++ sperimentale/errato



Se funzionasse, si farebbe così

- `flex --c++ -o lexer.cc lexer.ll`
- `g++ -Wall -Wextra -o lexer lexer.cc`
- `lexer < input_scanner`

Don't try this at home!

Commento inserito nel sorgente generato

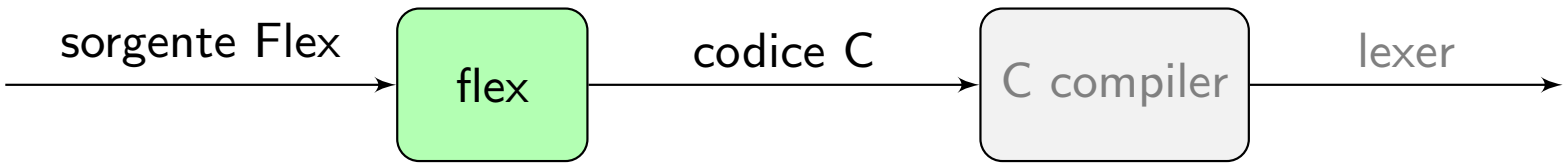
```
/* The c++ scanner is a mess. [...]
   We get reports that it breaks inheritance.
   We will address this in a future release of flex,
   or omit the C++ scanner altogether. */
```

Approccio alternativo

- codice C può essere compilato come C++
- nel sorgente Flex, dichiarazioni **pure** di funzioni C
- definirle in sorgente C++ (compilato separatamente)
- usare le linkage specification:

```
extern "C" void foo(int i) { /* code */ }
```

Nota bene: Flex è un compilatore



Compilatore da L verso M

- codice sorgente L = linguaggio Flex
- codice "macchina" M = linguaggio C

Sommario

Il linguaggio sorgente Flex

Struttura del file sorgente: 3 sezioni

```
  /* Sezione delle definizioni */
%%
  /* Sezione delle regole */
%%
  /* Sezione del codice utente */
```

Sommario

La sezione delle regole (i)

A cosa serve?

- Fornire la **definizione** della funzione `yylex()`
- La funzione `int yylex()` deve:
  - leggere un lessema dall'input
  - "restituire" il token corrispondente al chiamante

Cosa contiene la sezione?

- le regole lessicali per riconoscere i token
- codice C aggiuntivo (opzionale)

La sezione delle regole (ii)

Le regole lessicali

- formato: *pattern* *codice*
- *pattern*: l'**espressione regolare** che specifica il lessema
- *codice*: codice che "calcola" la **categoria lessicale**
- il **lessema** è individuato dalle variabili globali `yytext` (puntatore al primo carattere) e `yylength` (lunghezza)
- Nota Bene:
  - il pattern deve essere specificato a **inizio riga**
  - il codice deve iniziare nella stessa riga del pattern
  - è possibile andare a capo nel codice se lo si racchiude in un blocco: {*codice*}
  - è possibile andare a capo con pattern disgiuntivi usando | al posto del *codice*
  - l'ordine delle regole ne stabilisce la **priorità**

Esempio di regole (i)

Una keyword e gli identificatori

```
  /* omissis: definizione di KW_FOR e IDENT */

%%

  /* regola per keyword for */
for                     { return KW_FOR; }
  /* regola per identificatori */
[a-zA-Z][a-zA-Z0-9]*    { return IDENT; }

%%
```

Esempio di regole (ii)

Esempi di token riconosciuti dalle regole

lessema	token	note
`i`	`<IDENT, i>`	singolo match
`forza`	`<IDENT, forza>`	6 match, preferenza lessema lungo
`for`	`<KW_FOR, for>`	4 match, uso priorità

Come specificare i pattern in Flex

pattern	**significato**
c	carattere **non** speciale sta per se stesso
\c	carattere di escape (per i caratteri speciali)
(*pattern*)	parentesi (per specificare precedenze)
$pattern_1 pattern_2$	concatenazione
$pattern_1 \mid pattern_2$	alternanza
*pattern**	iterazione di Kleene (zero o più occorrenze)
pattern+	iterazione positiva (una o più occorrenze)
pattern?	opzionalità (zero o una occorrenza)
pattern{*m*,*M*}	iterazione limitata
.	qualsiasi carattere *singolo* **tranne newline**
[*chars*]	classe di caratteri (match singolo)
[^*chars*]	complemento di classe di caratteri
"*string*"	match letterale di *string*
{*name*}	uso di pattern tramite nome
$pattern_1 / pattern_2$	trailing context: $pattern_1$ solo se seguito da $pattern_2$
^*pattern*	start-of-line context (se primo carattere del pattern)
pattern$	end-of-line context (se ultimo carattere del pattern)

La sezione delle regole (iii)

Cose che **non** sono regole

- Una riga che inizia con **whitespace** è considerato codice
- Viene inserito **verbatim** nella definizione di `yylex`
- Ha senso solo in due casi:
  - commenti (come nell'esempio precedente)
  - un **lexical block**, i.e., blocco di codice racchiuso tra `%{` e `%}`, **prima di tutte le regole**: viene eseguito ogni volta che si invoca `yylex`

Sommario

La sezione definizioni

Cosa può contenere?

- “literal block”
- definizioni di pattern con nome
- opzioni per flex
- “start states”

Il literal block

Literal block

- un blocco di codice C racchiuso da %{ e %} (a inizio riga)
- viene copiato **verbatim** (i.e., letteralmente) nella parte iniziale del sorgente generato da Flex
- tipicamente può contenere:
  - definizioni costanti per categorie lessicali
  - dichiarazioni di variabili (usate nelle regole)
  - dichiarazioni di funzioni (invocate nelle regole)
  - definizione di funzioni **inline**

Esempio di literal block

Definizione delle costanti per le categorie lessicali

```
%{
enum Categorie {
  /*...*/, KW_FOR, IDENT, /*...*/
};
%}

%%

  /* regola per keyword for */
for                     { return KW_FOR; }
  /* regola per identificatori */
[a-zA-Z][a-zA-Z0-9]*    { return IDENT; }

%%
```

Note: costanti categorie lessicali

- sono normali costanti intere
- per es., si possono usare `#define` al posto delle costanti di enumerazione
- attenzione a non creare sovrapposizioni
- **NON** usare il valore costante zero
  (usata per segnalare il token speciale `<<EOF>>`)

Pattern con nome

Schema

NOME_1 *pattern*$_1$
$\ldots$ $\ldots$
NOME_n *pattern*$_n$

- i pattern **NON** hanno una azione (codice) associata
- dopo avere introdotto (il pattern per) il NOME_i, lo si può usare nei pattern successivi usando la sintassi $\{\mathrm{NOME}_i\}$
- scopo: migliorare **leggibilità** dei pattern nelle regole

Esempio di sezione definizioni (ii)

Esempi di (uso di) pattern con nome

```
DIGIT    [0-9]
LETTER   [a-zA-Z]
STARS    ("*")+

%%

  /* regola per keyword for */
for                          { return KW_FOR; }
  /* regola per identificatori */
{LETTER}({LETTER}|{DIGIT})* { return IDENT; }

%%
```

Come specificare i pattern in Flex (ii)

Suggerimenti

- usare le **virgolette** per simboli non alfanumerici e le **parentesi** (anche ridondanti) per aumentare leggibilità
  - cattivo stile: `/\*([^*]|\*+[^/*])*\*+/`
  - un po' meglio: `("/*")([^*]|("*")+[^*/])*(("*")+"/")`
- usare i **nomi di pattern** per evitare ripetizioni
  - ancora meglio:
    `STARS ("*")+` *(nella sezione definizioni)*
    `("/*")([^*]|{STARS}[^*/])*({STARS}"/")`

Opzioni per flex: due da usare sempre

Disabilitazione di `yywrap`

- `%option noyywrap`
- evita la generazione della funzione `yywrap()` e della sua chiamata a fine input

Disabilitazione della regola di default

- `%option nodefault`
- evita la generazione della *regola catch-all* (`. ECHO;`), che causa la stampa dei token non riconosciuti

Opzioni per flex: due da usare quando utile

Abilitazione conteggio linee

- `%option yylineno`
- definisce variabile intera `yylineno` che mantiene il numero di riga della posizione corrente (la **fine** del lessema)
- usare l'opzione causa una perdita di efficienza

Pattern case-insensitive

- `%option case-insensitive`
- rende case-insensitive i **pattern**
- **non** modifica il file di input (i **lessemi** riconosciuti rimangono case-sensitive)

Start states (aka start conditions)

Servono a limitare l'applicabilità di alcune regole

- le regole che abbiamo visto si applicano quando il lexer è nello stato/condition `INITIAL`
- possiamo definire altri stati/condition nella sezione delle definizioni, con la sintassi:
  `%x NOMESTATO`
- `%x` indica che si tratta di uno stato **esclusivo**: significa che quando il lexer entra in questo stato deve uscire dagli altri stati
- `%s` definirebbe uno stato **shared**, consentendo al lexer di essere contemporaneamente in più stati (complicato!)

Start states (ii)

Nella sezione delle regole posso:

- entrare in uno stato (uscendo dagli altri se esclusivo):
  pattern `{ BEGIN NOMESTATO; }`
- definire regole valide quando il lexer è nello stato:
  `<NOMESTATO>`*pattern* *codice*
- ritornare allo stato iniziale:
  `<NOMESTATO>`*pattern* `{ BEGIN INITIAL; }`

Esempio di uso di uno start state

Commento multilinea (C/C++/Java/SQL/...)

- pattern monolitico: `/\*([^*]|\*+[^/*])*\*+/`
- **sconsigliato**: potrebbe esaurire il buffer di lettura

Commenti usando lo start state

- nella sezione definizioni:
  - `%x COMMENT`
- nella sezione regole:
  - `"/*"  { BEGIN COMMENT; }`
  - `<COMMENT>"*/"  { BEGIN INITIAL; }`
  - `<COMMENT>([^*]|\n)+|.  /* skip */`
  - `<COMMENT><<EOF>>  { error(); return 0; }`

Sommario

La sezione del codice utente

- inizia dopo il secondo marker %%
- può contenere codice utente arbitrario, inserito **verbatim** dopo la definizione di `yylex`
- tipicamente:
  - definizione delle funzioni ausiliarie precedentemente dichiarate (nella sezione delle definizioni)
  - la funzione `main` (non usuale)
- **best practice**: non mettere le definizioni delle funzioni, usare piuttosto un'altra unità di traduzione

Esempio sezione codice utente

```
  /* ... */
%%
  /* ... */
%%

int main() {
  int token;
  while (1) {
    token = yylex();
    if (token == 0)
      break;
    if (token == ERROR)
      exit(1);
  }
  return 0;
}
```