Linguaggi e compilatori

Corso di Laurea in Informatica

Mauro Leoncini

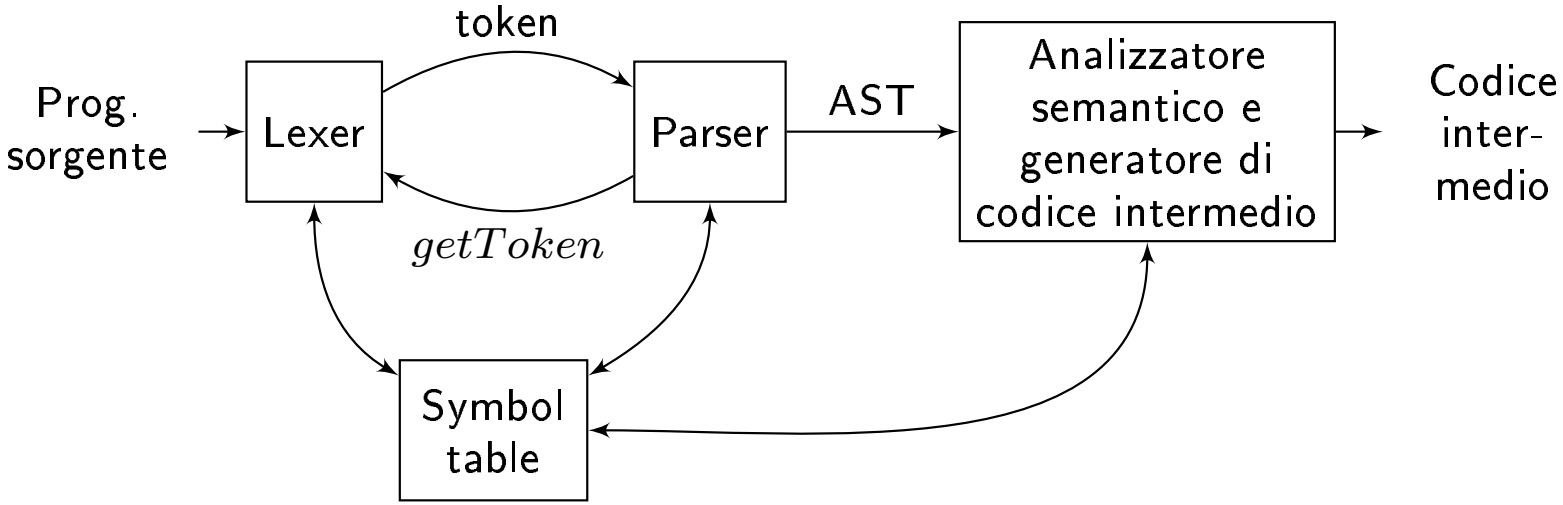
A.A. 2023/2024

1. Analizzatore lessicale (lexer)
   - Riconoscimento di token
   - Altri compiti del lexer
   - Il lexical analyzer Lex

Linguaggi e compilatori

Lo schema del front-end

- Il compito principale del lexer è di trasformare la sequenza di caratteri che costituisce l'input del compilatore (cioè il programma) in una sequenza di *token*
- Riconsideriamo lo schema del front-end (già visto a suo tempo)



- Il lexer agisce come subroutine del parser e, ad ogni chiamata, restituisce a quest'ultimo un *token*
- Ma che cos'è esattamente un token?

Token

- Un token è un *oggetto astratto*, che rappresenta un elemento significativo per l'analisi sintattica
- Esempi di token sono i *numeri*, gli *identificatori* e le *parole riservate* di un linguaggio di programmazione.
- Ci sono altre due nozioni da introdurre allo scopo di comprendere bene che cosa si intenda quando si parla di token
  - La prima è quella di *lessema*, cioè una sotto-stringa dell'input che è istanza concreta di un token: ad esempio `x` e `somma` possono essere due lessemi riconosciuti come token di tipo *identificatore*
  - La seconda è una descrizione formale (nella terminologia inglese si usa il termine *pattern*) che consente allo scanner di riconoscere i lessemi come istanze di particolari token
- È facile immaginare, anche da quanto detto nel precedente set di slide, che le descrizioni formali dei lessemi da riconoscere come token sono proprio le *espressioni regolari*

Token

- Possiamo ora essere più precisi nel definire i token
- Un token è un oggetto caratterizzato da due attributi, un *nome* obbligatorio e un *valore* opzionale.
- Il *token name* è un identificatore, che può essere arbitrario (anche se è opportuno che sia anche significativo); ciò che è cruciale è che uno stesso nome abbia lo stesso senso per lexer e parser
- Indicheremo i token name usando il grassetto
- Tipici nomi per i token sono **id**, **number**, **literal**, ...
- Il valore non è presente sempre, ma è necessario per dentificatori e numeri.
- Nel caso degli identificatori, il valore è proprio il corrispondente lessema, mentre nel caso dei numeri è il valore numerico del lessema

Esempio

- Supponiamo che la porzione di input da analizzare abbia come prefisso

  `somma = 0`

- Alle richieste del parser, il lexer restituisce in sequenza i token
  - (**id**, "somma")
  - **assignment**
  - (**number**, 0)
- Si noti la differenza fra i due valori presenti: nel caso di **id** il valore è una stringa (lo stesso lessema presente nell'input) mentre per **number** il valore è il numero 0
- Abbiamo cercato di evidenziare questo fatto usando gli apici (oltre che il font typewriter usato sempre per le stringhe)
- Nel parsing, come vedremo, il valore di un token non è importante e dunque ci riferiremo ai token con il solo token name
- Il valore di un token serve ovviamente nella generazione del codice

Pattern, lessemi e (nomi di) token

IMPORTANTE !

- La seguente tabella riassume, mediante alcuni esempi, i concetti che abbiamo presentato

Token name	Pattern	Esempio di lessema
id	[:alpha:][:alnum:]*	pippo1
number	[+-][1-9][:digit:]*	-3.14
comparison	< | > | <= | >= | = | !=	<
literal	[:*alpha*:]	"Pi greco"
if	if	if
while	while	while

- Solo per ragioni di spazio, nella precedente tabella il pattern che descrive il token **number** non include la notazione scientifica.

Linguaggi e compilatori

Scanning e altre funzioni

- L'analizzatore lessicale è l'unico modulo del compilatore che legge l'input testuale
- Il termine *scanner* viene a volte utilizzato per riferirsi all'analizzatore lessicale nel suo insieme
- Tuttavia esso è propriamente il modulo separato che effettua la lettura del testo, utilizzando opportune tecniche di buffering
- L'analizzatore lessicale vero e proprio, nel processo di tokenizzazione, procede anche a riconoscere e "filtrare" commenti, spazi e altri caratteri di separazione
- Deve inoltre associare gli eventuali errori trovati da altri moduli del compilatore (in particolare dal parser) alle posizioni (righe di codice) dove tali errori si sono verificati allo scopo di emettere appropriati messaggi diagnostici

Progetto di un analizzatore lessicale

- Procederemo ora nel modo seguente
- Dapprima introdurremo uno strumento per la generazione di analizzatori lessicali
- Impiegheremo poi lo strumento per realizzare il modulo di riconoscimento dei token del linguaggio XXX
- Solo in un secondo momento vedremo i principi interni di funzionamento di un lexer, cioè come sia possibile, a partire dalle espressioni regolari, realizzare in modo automatico il riconoscimento di token definiti da quelle espressioni regolari

Linguaggi e compilatori

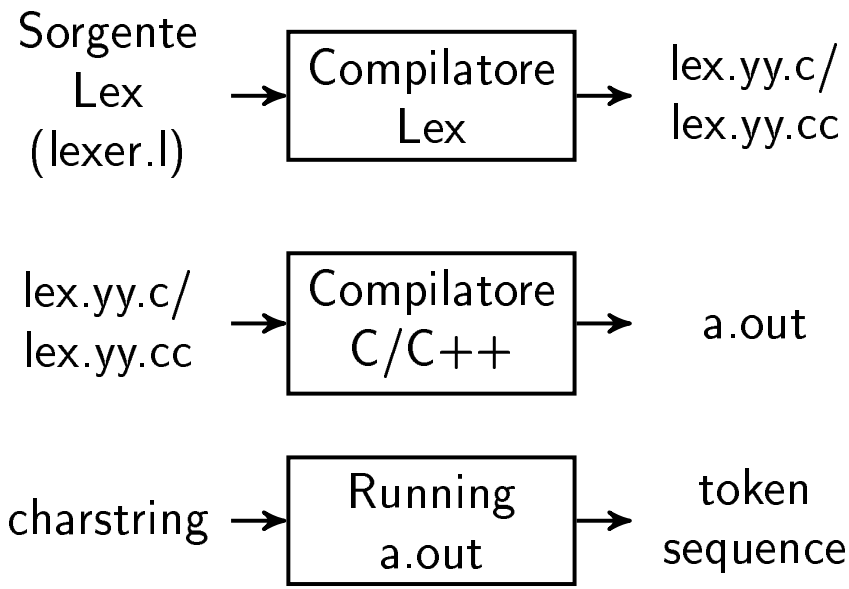
Che cosa è Lex

- Lex è un *generatore di analizzatori lessicali*.
- Si tratta cioè di un software in grado di generare automaticamente un altro programma che riconosce stringhe di un linguaggio regolare.
- Non solo, il software generato da Lex ha "capacità di scanning", cioè di acquisire le stringhe da analizzare in sequenza (da file o standard input) e di passare l'output ad un altro programma, tipicamente un parser.
- Lex può quindi essere uno strumento prezioso nella realizzazione di un compilatore.

Come funziona Lex

- L'input per un programma Lex è costituito "essenzialmente" da un insieme di espressioni regolari/pattern da riconoscere.
- Inoltre, ad ogni espressione regolare $\mathcal{E}$, l'utente associa un'*azione*, espressa sotto forma di codice in linguaggio `C`.
- Tale codice andrà in esecuzione quando l'analizzatore lessicale prodotto da Lex avrà riconosciuto un lessema descritto da $\mathcal{E}$
- NOTA. Il programma Lex originale era utilizzabile solo con il linguaggio `C`. La versione oggi disponibile in ambiente `Linux` (denominata `Flex`) può lavorare anche con il `C++`

Schema d'uso di Lex



Struttura generale di un programma Lex

- Un generico programma Lex contiene tre sezioni, separate dal dalla sequenza %%

```
Dichiarazioni
%%
Regole di traduzione
%%
Funzioni ausiliarie
```

- La sezione "Dichiarazioni" può contenere definizione di costanti/variabili, specifica di header file e, soprattutto, *definizioni regolari*, cioè espressioni regolari "con un nome"
- La sezione intermedia specifica le *regole di traduzione*, ovvero le descrizioni delle azioni che devono essere eseguite a seguito del riconoscimento dell'istanza di un pattern

Struttura generale di un programma Lex (continua)

- L'ultima sezione può contenere funzioni aggiuntive (che vengono tipicamente invocate nella parte relativa alle regole di traduzione)
- Se lo scanner non è utilizzato come routine del parser o di altro programma, quest'ultima sezione contiene anche il main program.
- Se presente, il main dovrà ovviamente contenere la chiamata allo scanner prodotto automaticamente dal compilatore Lex
- Nel caso del `C`, l'entry point dello scanner è la funzione `yylex`
- Nel caso del `C++`, Lex produce una classe `FlexLexer` e lo scanner si invoca chiamando il "metodo" `yylex`
- Il lessema corrispondente al token riconosciuto viene registrato dallo scanner nella variabile globale `yytext`

Un primo, semplicissimo, "tokenizer"

```
%{
#include <iostream>
using namespace std;
%}

DIG     [0-9]
DIG1    [1-9]

/* read only one input file */
%option noyywrap C++

%%
"+"             { cout << "   operatore <" << yytext[0] << ">" << endl;}
"-"             { cout << "   operatore <" << yytext[0] << ">" << endl;}
"="             { cout << "   operatore <" << yytext[0] << ">" << endl;}
{DIG1}{DIG}*    { cout << "      numero <" << yytext    << ">" << endl;}
.               { cout << "Altro token <" << yytext[0] << ">" << endl;}
%%
int main(int argc, char** argv) {
    FlexLexer* lexer = new yyFlexLexer;
    lexer->yylex();
    return 0;
}
```

Queste vengono inserite nel programma compilato

Espressioni regolari

Impostazione

Analizzare più di un file

- Al termine dell'esecuzione, `yylex` invoca il metodo `yywrap`.
- Questo può quindi essere utilizzato per aprire un secondo file di input e chiamare nuovamente `yylex`
- Se viene utilizzata l'opzione `noyywrap`, il metodo `yywrap` non viene invocato e non deve essere ridefinito.
- Il file `wrap.l` nell cartella condivisa su `gdrive` illustra un esempio di utilizzo di `yywrap`, valido per `flex` con l'opzione -+ (si veda la slide successiva).

Compilazione ed esecuzione

- Il programma Lex (file di norma con estensione `.l`) viene compilato con il seguente comando

  ```
  flex -+ -o <source>.cpp <filename>.l
  ```

  dove `<filename>` e `<source>` indicano i nomi rispettivamente del file Lex e del programma C++ generato.
- L'opzione `-+` indica proprio che il target è il linguaggio C++
- Se si omette l'opzione `-o`, il file viene creato con il nome default `lex.yy.cc`
- Il file C++ può essere compilato nel modo noto

Applicazioni "standalone"

- Uno strumento come Lex può essere utilmente impiegato anche per la creazione di applicazioni *standalone*
- Ad esempio, il programma Lex riportato nelle slide seguenti "emula" l'utility `wc` di Linux; conta cioè caratteri, parole e linee presenti in un file
- Del programma daremo in realtà due versioni, di fatto identiche ma organizzate in modo diverso, la prima in un file singolo e la seconda su due file, che andremo a compilare separatamente.
- I programmi mettono in evidenza l'esistenza di un'altra variabile globale, `yyleng`, che memorizza la lunghezza del lessema che ha provocato il match

Versione di wc realizzata con Lex (file singolo)

```
%{
#include <iostream>
using namespace std;

unsigned long charCount = 0, wordCount = 0, lineCount = 0;
%}

word [^ \t\n]+
eol \n
%option noyywrap C++
%%
{word}  {wordCount++; charCount += yyleng; }
{eol}   {charCount++; lineCount++;}
.       charCount++;

%%
int main( int argc, char** argv) {
   FlexLexer* lexer = new yyFlexLexer;
   lexer->yylex();
   cout << lineCount << " " << wordCount << " "
        << charCount << endl;
   return 0;
}
```

File Lex, senza main, per wc (wcsep.cpp)

```
%{
#include <iostream>
using namespace std;

unsigned long charCount = 0, wordCount = 0, lineCount = 0;
%}

word [^ \t\n]+
eol \n
%option noyywrap C++

%%
{word}  {wordCount++; charCount += yyleng; }
{eol}   {charCount++; lineCount++;}
.       charCount++;

%%
```

- Il main program (prossima slide) deve importare l'header file `FlexLexer.h`

Main program per l'applicazione wc (wcmain.cpp)

```
#include <iostream>
#include <FlexLexer.h>
using namespace std;
extern unsigned long charCount, wordCount, lineCount;

int main() {}
    FlexLexer* lexer = new yyFlexLexer;
    lexer->yylex();
    cout << lineCount << " " << wordCount << " "
    << charCount << endl;
    return 0;
}
```

- Se supponiamo che questo file abbia nome `wcmain.cpp`, la compilazione dell'applicazione può procedere nel modo seguente

```
g++ -c wcsep.cpp
g++ -c wcmain.cpp
g++ -o wc wcsep.o wcmain.o
```

Il linguaggio Kaleidoscope

- Utilizzeremo questo semplice linguaggio per studiare su un caso concreto i concetti teorici analizzati a lezione
- Seguiremo abbastanza fedelmente il tutorial `https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html`
- Introdurremo però alcune importanti varianti, soprattutto nell'implemetazione di lexer e parser
- Esattamente come nel tutorial procederemo arricchendo per gradi il linguaggio
- Nella prima versione, Kaleidoscope permette solo la definizione di funzioni e la scrittura di espressioni aritmetiche
- NOTA: Nei riferimenti al tutorial, la scrittura `<tutorial>` sarà da considerare una sorta di "macro" per l'url sopra riportato

Il linguaggio e il suo lessico

- In questa prima versione, Kaleidoscope accetta soltanto tre tipi di frasi, che presentiamo attraverso tre esempi
  1. `def g(x y z) x*y+z;`
  2. `extern f(x y);`
  3. `3*f(3,4)-x;`
- La prima frase descrive la forma delle definizioni di funzione
- La seconda l'utilizzo di funzioni esterne
- La terza suggerisce infine la possibilità di utilizzare espressioni aritmetiche, che includano le quattro operazioni fondamentali e l'utilizzo di (chiamate di) funzioni
- Tutto ciò ci consente di comprendere quale sia (a questo punto) il lessico del linguaggio

Il lessico di Kaleidoscope (prima versione)

- Fanno parte del lessico
  - Gli operatori aritmetici +, -, * (che denota la moltiplicazione) e /
  - Le parentesi tonde e la virgola
  - Il punto e virgola (separatore delle espressioni)
  - Le due parole chiave `def` e `extern`
  - I numeri interi e decimali
  - Gli identificatori
- Andremo a breve a definire le espressioni regolari che descrivono i possibili lessemi delle categorie lessicali appena introdotte
- Prima però vogliamo brevemente procedere con esempi di riconoscimento di token utilizzando il programma disponibile nel tutorial

Il Lexer di Kaleidoscope

- Il codice per il Lexer è scaricabile all'url

  `<tutorial>/LangImpl02.html#full-code-listing`

- Il codice in questione effettua anche parsing e costruzione dell'AST.
- Per ora la nostra attenzione è solo sull'analisi lessicale e dunque il lexer è stato "scorporato" e inserito in un file a parte
- Abbiamo anche predisposto un main per testare il lexer
- Tutto il software si trova nella cartella condivisa GDRIVE (sotto-cartella *Kaleidoscope1*).
- Compilazione ed esecuzione (usando clang):

```
> clang++ -c lexer.cpp
> clang++ -c main.cpp
> clang++ -o klex lexer.o main.o
> echo "3*x-2;" | ./klex
```

Il lexer realizzato con Lex

- Il codice che abbiamo appena descritto è stato realizzato (dagli autori del tutorial) in modo diretto, senza ausilio di strumenti automatici
- In questa fase del nostro percorso, vogliamo semplicemente ottenere lo stesso risultato (ovvero il riconoscimento di un sottoinsieme dei token) usando `lex`
- Gli esempi già presentati sono sufficienti per svolgere questo primo esercizio.
- L'aspetto cruciale consiste, come è ovvio, nella definizione delle espressioni regolari
- A riguardo vanno distinti
  - identificatori e parole chiave
  - numeri interi e decimali
  - operatori aritmetici e confronto (solo $<$)
  - separatori
  - eventuali token formati da un singolo carattere