

UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II



Corso di Laurea Magistrale in Ingegneria Informatica

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE
DELL'INFORMAZIONE

ELABORATO FINALE IN
ALGORITMI E STRUTTURE DATI

ANALIZZATORI LESSICALI E SINTATTICI

Professore:
Stefano Avallone

Candidato:
Benfenati Domenico M63001165

ANNO ACCADEMICO 2020/2021

Indice

1	Introduzione	2
2	L'analisi lessicale	3
2.1	Esempio di analizzatore lessicale	3
2.2	Traduzione dell'automa in codice	6
2.3	Analizzatori Lessicali con FLEX	7
2.4	Struttura di un programma Lex	8
2.4.1	Sezione delle definizioni	8
2.4.2	Sezione delle regole	8
2.4.3	Sezione del codice utente	9
2.5	Esempio di analizzatore lessicale in Lex	10
2.5.1	Test del programma Lex	11
2.6	Esempio di analizzatore lessicale in C	12
2.6.1	Test del programma di riconoscimento	16
3	L'analisi sintattica	17
3.1	Grammatiche context-free	18
3.2	YACC: generatore di analizzatori sintattici	18
3.2.1	Struttura del sorgente YACC	19
3.2.2	Regole di produzione in YACC	20
3.3	Integrazione tra Lex e YACC	20
4	Un analizzatore sintattico: Calcolatrice	23
4.1	File per l'analisi lessicale	23
4.2	File per l'analisi sintattica	24
4.3	Prova di esecuzione	25

Capitolo 1

Introduzione

Con il termine *analisi lessicale* si va ad indicare una pratica che permette di prendere in ingresso una sequenza di caratteri e produrre in uscita una sequenza di elementi, che vengono detti **token**.

Generalmente l'analisi lessicale fa parte della compilazione di un programma, e lo strumento che attua l'analisi lessicale viene detto analizzatore, o più comunemente **Scanner** o *lexer*.^[1]

L'analizzatore lessicale va ad effettuare la validazione di un token, la quale avviene tramite delle regole lessicali, che variano a seconda del linguaggio che viene usato.

Lo stream di token che viene fuori da un analizzatore lessicale può essere utilizzato per effettuare un'analisi sintattica per definire in che modo i token sono composti all'interno dello stream per comporre frasi compiute. Lo strumento che ci permette di effettuare questo tipo di analisi è detto *analizzatore sintattico* o **parser**, ed è un programma in grado di trasformare una sequenza di token in un albero che ne descrive la struttura sintattica.^[3]

A posteriore all'analisi sintattica, un ultimo processo che è possibile effettuare è il processo di analisi semantica: effettuare un *analisi semantica* vuol dire assegnare un significato, e quindi un senso ad una struttura sintattica, che si ottiene dal processo di analisi sintattica. I significati che si vanno ad attribuire, ovvero i *sensi*, sono rappresentati mediante collezioni di sinonimi, detti anche **synset**. Attribuendo ai vari synset dei codici univoci, si possono classificare i concetti in una struttura con le relazioni associate ai vari synset, detta **ontologia**, ed arrivare ad una traduzione automatica tale da permettere facilmente di passare da un insieme di sinonimi ad un altro.^[2]

Capitolo 2

L'analisi lessicale

2.1 Esempio di analizzatore lessicale

Consideriamo il seguente problema:

Si vogliono riconoscere, data in ingresso una sequenza di caratteri terminata dal carattere di fine-file, le parole costituite da sequenze di lettere dell'alfabeto inglese eventualmente seguite da un accento acuto (') o grave (`), ma solo nel caso la parola termini con una vocale. Ogni parola può cominciare anche subito dopo un accento

Come prima cosa, andiamo a studiare il problema che ci viene posto:

- nel caso in cui un accento debba essere posposto ad una consonante, significa che quell'accento non deve far parte della parola che stiamo considerando;
- è importante la distinzione tra il riconoscimento di una vocale o di una consonante, dal momento che se ho una vocale devo andare a verificare se successivamente ad essa è presente un accento;
- il fatto che una parola può cominciare anche dopo un accento, mi dice che dopo aver avuto un accento, devo accertarmi che qualsiasi carattere letterale a seguito di esso è l'inizio di una parola nuova.

Per andare a modellare una situazione del genere, è comodo andare ad effettuare quello che viene detto *automa a stati finiti*.

L'automa a stati finiti che rappresenta il nostro problema consta di uno stato di partenza (stato iniziale) nel quale mi troverò nel momento di analisi del carattere in ingresso, uno stato finale che indica il completamento del riconoscimento di un token, e di relative transizioni (archi), etichettate con i caratteri che possono occorrere nella sequenza di ingresso.

Prima di dare una rappresentazione grafica dell'automa andiamo a descrivere che cosa succede in quelli che saranno gli stati dell'automa:

- inizialmente vado a saltare gli eventuali caratteri che non sono lettere, che troviamo prima della prima lettera che sarà una potenziale parola nel nostro testo;
- appena riconosco una lettera devo distinguere se essa è una consonante o una vocale, dal momento che la differenza sta sul fatto che dopo una vocale potrei avere un accento che andrebbe ugualmente a fare parte della mia parola;
- riconosciuta la prima lettera, resto nello stato relativo al riconoscimento di una vocale se la lettera successiva è ancora una vocale, stessa per le consonanti. Nel caso in cui una delle due succeda all'altra devo necessariamente cambiare stato e tenere traccia dell'ultima lettera ricevuta;
- nel caso ricevo un carattere differente dopo una consonante, qualsiasi esso sia, significa che ho appena terminato una parola ed essa è stata riconosciuta; se invece l'ultima lettera era una vocale è importante distinguere se dopo essa succede un accento o un carattere differente: nel caso alla vocale succeda un accento riconosco la parola dopo di esso, altrimenti la parola termina con una vocale.

Andando quindi a tradurre graficamente le considerazioni fatte si ottiene una prima versione dell'automa.

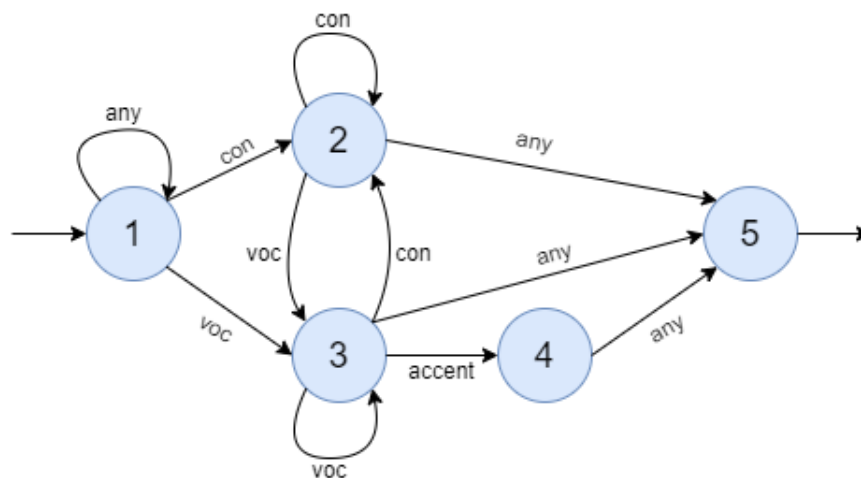


Figura 2.1: Prima versione dell'automa

L'etichetta **any** assegnata ad alcuni archi indica un carattere qualsiasi, e la transizione su quell'arco avviene per un qualunque carattere in ingresso, fatta eccezione per i caratteri specificati in altri archi uscenti dal nodo.

Guardando l'automa però ci sono alcune considerazioni che possono essere fatte, per gestire i possibili errori che si possono avere.

Infatti, un'alternativa può essere quella di saltare gli eventuali caratteri prima del riconoscimento di una lettera, successivamente al riconoscimento del token precedente. Ciò comporta che ci sia un ciclo sullo stato 5, e dallo stato 5 ci si sposti in un nuovo stato di riconoscimento della parola solo quando sono terminati i caratteri separatori.

Inoltre, quello che è possibile vedere è che se mi trovo sull'ultima lettera di una parola, o alternativamente su un accento, cambiando stato vado a *consumare* il carattere successivo, cioè vado ad avanzare lo stream che sto analizzando. Così facendo incorro nell'errore che potrei consumare un carattere che fa parte di una nuova parola.

Infine, è opportuno gestire la condizione di terminazione del file, che può avvenire in ogni stato dell'automa stesso, e farmi terminare l'analisi.

Andando a tradurre questi accorgimenti all'interno del diagramma, l'automa diventa il seguente.

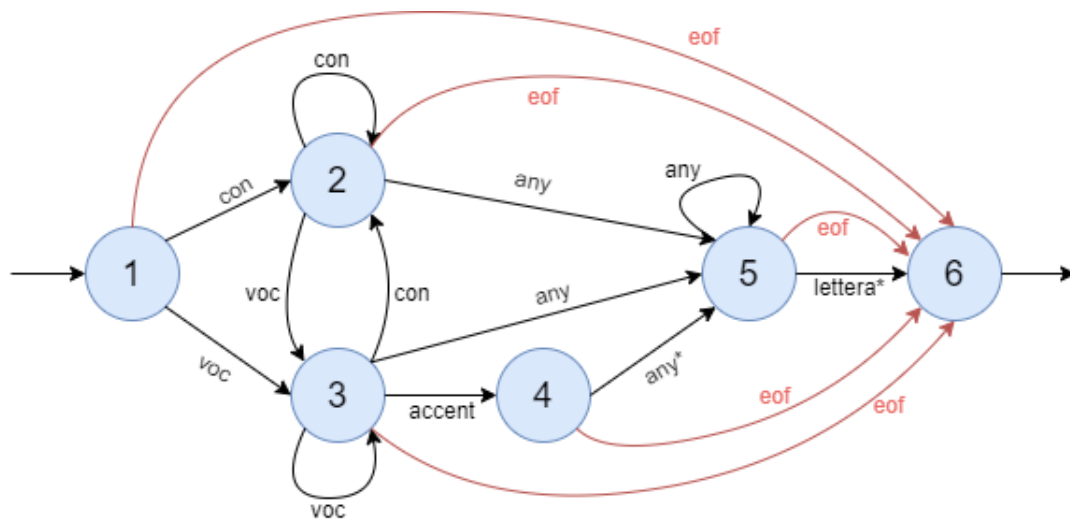


Figura 2.2: Prima versione dell'automa

L'asterisco con cui vengono etichettati alcuni archi sta ad indicare che percorrendo quell'arco non si andrà a consumare il carattere, ma verrà solo verificato che corrisponda al carattere giusto per fare la transizione da uno stato all'altro.

2.2 Traduzione dell'automa in codice

Grazie alla scrittura dell'automa, risulta estremamente semplice andare a stilare il codice a cui corrisponde l'automa stesso.

Andiamo a fare qualche considerazione e qualche definizione prima di procedere a scrivere il codice:

- La procedura, che chiameremo **Scanner-Parola**, dovrà restituire il token presente in ingresso, opportunamente rappresentato: nel caso delle parole di un testo, il token da restituire potrebbe essere un vettore contenente i caratteri riconosciuti e facenti parte della parola stessa. Un'idea è quella di andare a costruire un vettore che potrebbe chiamarsi *parola*, e per ogni carattere in ingresso che appartenga ad una specifica parola, poi andare ad aggiungere quel carattere all'array *parola*;
- Bisogna opportunamente conservare una variabile che rappresenta lo stato, in cui l'automa si trova, in modo da fare sì che si possano differenziare le operazioni da eseguire nei differenti stati;
- Dovendo scannerizzare tutti i caratteri presenti all'interno della sequenza di ingresso, le operazioni devono essere eseguite ciclicamente fino alla fine dello stream.

Fatte queste considerazioni, il codice corrispondente che ne deriva è il seguente:

```
SCANNER-PAROLA (T, parola)
{ riconosce e restituisce la parola successiva in T }
stato ← 1
inizializza l'array parola come stringa vuota
while stato ≠ 6
do case stato of
  1 : if next[T] ∈ CONSONANTI
      then stato ← 2
          aggiungi next[T] all'array parola
          GET(T)
      else if next[T] ∈ VOCALI
          then stato ← 3
              aggiungi next[T] all'array parola
              GET(T)
  2 : if next[T] ∈ CONSONANTI
      then stato ← 2
          aggiungi next[T] all'array parola
          GET(T)
      else if next[T] ∈ VOCALI
          then stato ← 3
              aggiungi next[T] all'array parola
              GET(T)
          else { next[T] = any }
              stato ← 5
              GET(T)
```

```

3:  if next[T] ∈ CONSONANTI
    then stato ← 2
        aggiungi next[T] all'array parola
        GET(T)
    else if next[T] ∈ VOCALI
        then stato ← 3
            aggiungi next[T] all'array parola
            GET(T)
        else if next[T] ∈ ACCENTI
            then stato ← 4
                aggiungi next[T] all'array parola
                GET(T)
            else { next[T] = any }
                stato ← 5
                GET(T)

4:  { transizione asteriscata }
    stato ← 5

5:  if next[T] ∈ LETTERE
    then { transizione asteriscata }
        stato ← 6
    else { next[T] = any }
        stato ← 5
        GET(T)

```

2.3 Analizzatori Lessicali con FLEX

Dal momento che scrivere un programma che analizza il lessico di una sequenza di caratteri in ingresso potrebbe essere estremamente complicato, specialmente quando si parla di aree tematiche che devono disporre di un vocabolario estremamente ampio, quello che spesso si fa è affidare la scrittura di un analizzatore lessicale a dei programmi automatici, che dato in ingresso le caratteristiche di ciò che lo scanner dovrà riconoscere, essi costruiscono un analizzatore lessicale ad-hoc.

Un esempio di strumento per la generazione di analizzatori lessicali è **FLEX** (*Fast Lexical Analyzer Generator*). Grazie al tool Flex, è possibile andare a definire alcune regole per la scrittura del mio programma attraverso una sintassi particolare.

Lex infatti, è in grado di costruire uno scanner in linguaggio C partendo da una sequenza di espressioni regolari che definiscono i token e un insieme corrispondente di azioni espresse come frammenti di programmi in linguaggio C.

Il funzionamento del tool è estremamente semplice:

- **Passo 1:** si dà in input al compilatore un file, scritto in linguaggio Lex, e con estensione .l che descrive il funzionamento del mio analizzatore. Il compilatore Lex trasforma il file in linguaggio Lex in un file in linguaggio C.
- **Passo 2:** il compilatore C compila un file particolare detto `lex.yy.c` e genera un file `a.out` eseguibile.
- **Passo 3:** al file eseguibile viene dato in pasto la sequenza di caratteri che deve essere analizzata, ed esso produce l'insieme di token corrispondente.

I passi che vengono effettuati sono schematizzabili tramite il seguente schema di funzionamento.

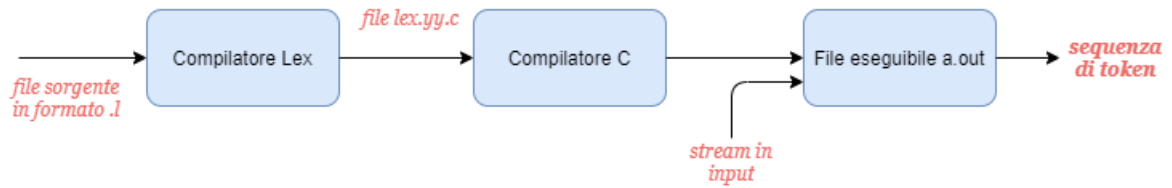


Figura 2.3: Schema di funzionamento di Flex

2.4 Struttura di un programma Lex

Un programma scritto in linguaggio Lex, ha una precisa sintassi e una determinata struttura. Esso infatti si compone di tre sezioni fondamentali: **sezione delle definizioni**, **sezione delle regole** e **sezione del codice utente**.

2.4.1 Sezione delle definizioni

La prima parte del file .l è dedicata alla dichiarazione delle variabili, alle definizioni generali e alla definizione di eventuali costanti. La sintassi prevede di racchiudere questa sezione tra `%{...%}`.

Tutto ciò che viene indicato all'interno di questa sezione, viene trasferito interamente nel file generato in linguaggio C.

La sintassi quindi è la seguente:

```
%{  
    // Definizioni  
%}
```

2.4.2 Sezione delle regole

Questa sezione contiene le regole che devono essere invocate all'atto della scannerizzazione dei caratteri, e sono nella forma `PATTERN AZIONE` priva di indentazione, dove per *pattern* si indica il carattere, o l'insieme di caratteri che l'analizzatore è in grado di riconoscere.

La sezione delle regole è racchiusa tra gli identificatori `%%...%%`, e per ogni pattern l'elenco di azioni ad esso associate è indicato tra parentesi `{...}` sullo stesso rigo del pattern a cui esse fanno riferimento.

La sintassi corretta è quindi la seguente:

```
%%
Pattern Azione
Pattern {Azioni}
%%
```

Nella tabella di seguito possiamo vedere alcuni pattern principali con i relativi elementi che essi vanno a identificare.

Pattern	Elementi rappresentati
[0-9]	indico tutte le cifre da 0 a 9
[0+9]	indico o i caratteri '0','+' oppure il carattere '9'
[0-9]+	indico una o più cifre tra 0 e 9
[^a]	indico tutti i caratteri eccetto 'a'
[^A-Z]	indico tutti i caratteri eccetto quelli da 'A' a 'Z' maiuscole
.	indico tutti i caratteri eccetto il carattere di fine riga
a{2,4}	indico da 2 a 4 occorrenze successive di 'a': 'aa', 'aaa' oppure 'aaaa'
a*	indico zero o più occorrenze del carattere 'a'
a{2,}	indico 2 o più occorrenze di 'a'
[a-zA-Z]	indico tutte le lettere alfabetiche, maiuscole e minuscole
w(x y)z	indico o la sequenza 'wxz' oppure la sequenza 'wyz'

2.4.3 Sezione del codice utente

Di seguito alla sezione che riguarda le regole, è possibile andare a specificare una ulteriore sezione, che riguarda delle funzioni e delle istruzioni in linguaggio C aggiuntive, che possono essere alternativamente compilate separatamente e caricate all'interno dell'analizzatore. Questa sezione non richiede alcun delimitatore specifico, tutto quello che viene rilevato non essere all'interno della sezione relative alle regole, viene considerato come appartenente alla sezione del codice.

Mettendo insieme le tre sezioni, si va a delineare quella che è la struttura del file .1, che viene indicata di seguito.

```
%{
    Definizione 1
    Definizione 2
    ...
}%
```

```

%%
Pattern Azione
Pattern {Azioni}
%%

codice C

```

2.5 Esempio di analizzatore lessicale in Lex

Vediamo ora un esempio di analizzatore lessicale in linguaggio Lex. Il seguente analizzatore effettua il conteggio delle parole che fanno parte di uno stream di caratteri in ingresso.

Il programma è stato scritto usando un file di testo, con estensione `.l`, e successivamente compilato sfruttando FLEX, e il compilatore `gcc` per la traduzione del file in uscita dal compilatore FLEX.

```

/*programma LEX che conta quante parole sono presenti
nello stream in ingresso*/
// SEZIONE DELLE DICHIARAZIONI
%{
#include<stdio.h>
#include<string.h>
int i = 0;
}%

// SEZIONE DELLE REGOLE
%%
([a-zA-Z0-9])*      { i++;}  /* regola di conteggio
                             delle parole*/

"\n" { printf("%d\n", i); i = 0;}
%%

// SEZIONE DEL CODICE UTENTE
int yywrap(void){}

int main()
{

```

```

        /* funzione che avvia l'analisi richiamando
        il compilatore Lex*/
        yylex();

        return 0;
    }

```

Come si può vedere, la regola che conta il numero di caratteri che sono appartenenti ad una singola parola, è possibile sintetizzarla in questo modo:

"incrementa la variabile i, se il carattere che analizzi è tra 'a' e 'z', oppure tra 'A' e 'Z', oppure è una cifra da 0 a 9, anche in più occorrente se si tratta di doppie"

L'altra regola presente all'interno del file invece serve a gestire il carattere di fine riga. Possiamo descrivere anche questa regola testualmente:

"stampa il valore di i accumulato fino ad ora, e azzera la variabile i nel caso il carattere che stai analizzando è un terminatore di riga"

La chiamata alla funzione `yylex()` rappresenta il punto di accesso al file che esce dal compilatore Lex. Quando la funzione `yylex` viene invocata, essa richiama la funzione `input()`, che legge il carattere dello stream successivo. Se essa incontra il carattere di fine file, viene invocata la funzione `yywrap()`, che, se restituisce 1, conferma che il file di input è terminato, mentre se restituisce 0 indica che ci sono ancora altri caratteri da analizzare.

2.5.1 Test del programma Lex

Per effettuare il test del programma, è necessario avere installato sulla macchina il compilatore Lex e un compilatore C, come può essere ad esempio il compilatore *gcc*.

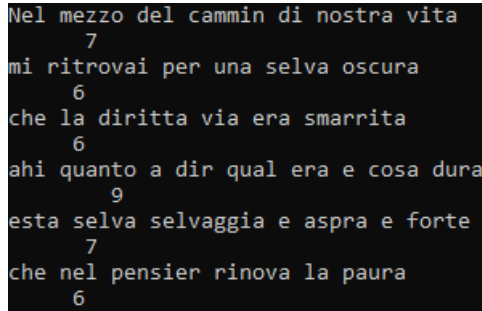
Per effettuare la compilazione Lex del file `.l`, tramite command line andiamo a impartire il comando

```
flex nomefile.l
```

Dopo aver effettuato la compilazione del file scritto in Lex, il compilatore stesso andrà a generare un nuovo file, come detto, scritto in linguaggio C, che deve essere altrettanto compilato, stavolta utilizzando un compilatore C. Il file che viene generato dal compilatore è un file di default, indicato con `lex.yy.c`, e per compilarlo utilizzando il compilatore *gcc* è sufficiente impartire il comando

```
gcc lex.yy.c
```

A questo punto il compilatore C crea il file `.exe`, che è possibile eseguire sia a riga di comando che tramite un click sul file stesso. Vediamo alcune esecuzioni del programma: così come è stato scritto il programma, esso prevede di far inserire la stringa di caratteri all'utente, e a seguito di un invio esso calcola il numero di parole che la stringa contiene per poi estrarne il numero.



```

Nel mezzo del cammin di nostra vita
7
mi ritrovai per una selva oscura
6
che la diritta via era smarrita
6
Oh! quanto a dir qual era e cosa dura
9
esta selva selvaggia e aspra e forte
7
che nel pensier rinova la paura
6

```

Figura 2.4: esecuzione del programma Lex di conteggio delle parole

2.6 Esempio di analizzatore lessicale in C

Oltre al linguaggio LEX, ovviamente, un analizzatore lessicale può essere scritto in un qualunque linguaggio, come ad esempio il linguaggio C. Nel seguente esempio è stato descritto il funzionamento di un analizzatore che riconosce dei token riguardo una frase in linguaggio C. Prima di vedere l'implementazione, andiamo a verificare quali sono gli elementi che si possono incontrare andando ad analizzare uno statement scritto in linguaggio C:

- Un primo elemento sono le parole chiave, che nell'esempio chiameremo **KEYWORD**: queste sono tutte le parole che sono specifiche del linguaggio, come `"if"`, `"then"`, `"for"`, `"int"`, `"short"`, `"return"`, e altri ancora;
- Un altro elemento valido sono alcuni caratteri speciali, che possono essere definiti **DELIMITATORI**: questi possono essere parentesi di ogni genere, `";"`, `","`, e altri elementi simili;
- Aggiungendo all'insieme dei delimitatori, anche caratteri numerici come `"0"`, `"1"`, `"2"`, eccetera, diremo che l'insieme così formato va a determinare quelli che indichiamo come **IDENTIFICATORI VALIDI**;
- Infine, tra gli elementi che possiamo riconoscere ci sono quelli che indichiamo come **OPERATORI**: di questo insieme fanno parte i classici operatori matematici, e operatori di confronto come `"<"`, `">"`, eccetera.

Andiamo dunque a vedere come può essere il codice del riconoscitore che supporta tutte le caratteristiche suddette. Come prima cosa, andiamo ad identificare delle funzioni booleane

che vanno a stabilire se un carattere è all'interno di una delle quattro categorie che si sono delineate, e cioè se esso è un delimitatore, un operatore, un identificatore valido o una parola chiave. Ognuna delle quattro funzioni prenderà in ingresso una stringa, e restituirà un booleano. Le funzionalità suddette sono descritte di seguito.

```

6 // Ritorna 'true' se il carattere è un DELIMITATORE.
7 bool isDelimiter(char ch) {
8     if (ch == '.' || ch == '+' || ch == '-' || ch == '*' ||
9         ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
10        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
11        ch == '[' || ch == ']' || ch == '{' || ch == '}')
12        return (true);
13    return (false);
14 }

```

Figura 2.5: funzione di ricerca di un delimitatore

```

16 // Ritorna 'true' se il carattere è un OPERATORE.
17 bool isOperator(char ch) {
18     if (ch == '+' || ch == '-' || ch == '*' ||
19         ch == '/' || ch == '>' || ch == '<' ||
20         ch == '=')
21         return (true);
22     return (false);
23 }

```

Figura 2.6: funzione di ricerca di un operatore

```

25 // Ritorna 'true' se il carattere è un IDENTIFICATORE VALIDO.
26 bool validIdentifier(char* str) {
27     if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
28         str[0] == '3' || str[0] == '4' || str[0] == '5' ||
29         str[0] == '6' || str[0] == '7' || str[0] == '8' ||
30         str[0] == '9' || isDelimiter(str[0]) == true)
31         return (false);
32     return (true);
33 }

```

Figura 2.7: funzione di ricerca di un identificatore valido

```

35 // Ritorna 'true' se il carattere è una KEYWORD.
36 bool isKeyword(char* str) {
37     if (!strcmp(str, "if") || !strcmp(str, "else") ||
38         !strcmp(str, "while") || !strcmp(str, "do") ||
39         !strcmp(str, "break") ||
40         !strcmp(str, "continue") || !strcmp(str, "int")
41         || !strcmp(str, "double") || !strcmp(str, "float")
42         || !strcmp(str, "return") || !strcmp(str, "char")
43         || !strcmp(str, "case") || !strcmp(str, "char")
44         || !strcmp(str, "sizeof") || !strcmp(str, "long")
45         || !strcmp(str, "short") || !strcmp(str, "typedef")
46         || !strcmp(str, "switch") || !strcmp(str, "unsigned")
47         || !strcmp(str, "void") || !strcmp(str, "static")
48         || !strcmp(str, "struct") || !strcmp(str, "goto"))
49         return (true);
50     return (false);
51 }

```

Figura 2.8: funzione di ricerca di una keyword

Oltre alle funzioni utili per effettuare i riconoscimenti all'interno della sequenza di caratteri, è possibile implementare alcune funzioni per andare a specificare ad esempio se il carattere che sto considerando fa parte di un intero o di un numero reale decimale. Per fare ciò implementiamo altre due funzionalità, che sono descritte in seguito.

```
53 // Ritorna 'true' se il carattere è un INTEGER.
54 bool isInteger(char* str) {
55     int i, len = strlen(str);
56
57     if (len == 0)
58         return (false);
59     for (i = 0; i < len; i++) {
60         if (str[i] != '0' && str[i] != '1' && str[i] != '2'
61             && str[i] != '3' && str[i] != '4' && str[i] != '5'
62             && str[i] != '6' && str[i] != '7' && str[i] != '8'
63             && str[i] != '9' || (str[i] == '-' && i > 0))
64             return (false);
65     }
66     return (true);
67 }
```

Figura 2.9: funzione di identificazione di un intero

```
69 // Ritorna 'true' se il carattere è un NUMERO REALE.
70 bool isRealNumber(char* str) {
71     int i, len = strlen(str);
72     bool hasDecimal = false;
73
74     if (len == 0)
75         return (false);
76     for (i = 0; i < len; i++) {
77         if (str[i] != '0' && str[i] != '1' && str[i] != '2'
78             && str[i] != '3' && str[i] != '4' && str[i] != '5'
79             && str[i] != '6' && str[i] != '7' && str[i] != '8'
80             && str[i] != '9' && str[i] != '.' ||
81             (str[i] == '-' && i > 0))
82             return (false);
83         if (str[i] == '.')
84             hasDecimal = true;
85     }
86     return (hasDecimal);
87 }
```

Figura 2.10: funzione di identificazione di un numero reale

Prima di descrivere l'algoritmo vero e proprio, è utile andare ad implementare una funzione che vada ad estrarre una sottoparte di una stringa in ingresso. Questo perchè ciò che la funzione generale farà sarà lavorare prendendo in ingresso l'intera stringa contenente tutti gli eventuali token presenti all'interno dello stream in ingresso e analizzandola. Ciò è possibile farlo grazie alla funzione `substring`, che è descritta qui di seguito.

```

89 // Estrazione di una SOTTOSTRINGA.
90 char* subString(char* str, int left, int right) {
91     int i;
92     char* subStr = (char*)malloc(
93         sizeof(char) * (right - left + 2));
94
95     for (i = left; i <= right; i++)
96         subStr[i - left] = str[i];
97     subStr[right - left + 1] = '\0';
98     return (subStr);
99 }

```

Figura 2.11: funzione di estrazione di una sottostringa

Fatto ciò andiamo a descrivere la funzione che effettua il riconoscimento.

```

102 void parse(char* str) {
103     int left = 0, right = 0;
104     int len = strlen(str);
105
106     while (right <= len && left <= right) {
107         if (isDelimiter(str[right]) == false)
108             right++;
109
110         if (isDelimiter(str[right]) == true && left == right) {
111             if (isOperator(str[right]) == true)
112                 printf("%c' IS AN OPERATOR\n", str[right]);
113
114             right++;
115             left = right;
116         } else if (isDelimiter(str[right]) == true && left != right
117             || (right == len && left != right)) {
118             char* subStr = subString(str, left, right - 1);
119
120             if (isKeyword(subStr) == true)
121                 printf("%s' IS A KEYWORD\n", subStr);
122
123             else if (isInteger(subStr) == true)
124                 printf("%s' IS AN INTEGER\n", subStr);
125
126             else if (isRealNumber(subStr) == true)
127                 printf("%s' IS A REAL NUMBER\n", subStr);
128
129             else if (validIdentifier(subStr) == true
130                 && isDelimiter(str[right - 1]) == false)
131                 printf("%s' IS A VALID IDENTIFIER\n", subStr);
132
133             else if (validIdentifier(subStr) == false
134                 && isDelimiter(str[right - 1]) == false)
135                 printf("%s' IS NOT A VALID IDENTIFIER\n", subStr);
136             left = right;
137         }
138     }
139     return;
140 }

```

Figura 2.12: funzione di riconoscimento di un token

2.6.1 Test del programma di riconoscimento

Per effettuare il test del riconoscitore, creiamo un programma `main` che vada a definire la stringa di caratteri che devo analizzare, e che chiami su quella stringa la funzione di `parse` che abbiamo descritto. Come output ci si aspetta una descrizione delle parole che l'analizzatore riconosce come tali. Di seguito è riportato un esempio di funzionamento del programma.

```
142 // FUNZIONE DI TEST
143 int main()
144 {
145     // dichiarazione di una stringa di prova
146     char str[100] = "int a = (12 * b)+5.3*c;";
147
148     parse(str); // analisi della stringa
149
150     return (0);
151 }
```

(a)

```
'int' IS A KEYWORD
'a' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'12' IS AN INTEGER
'*' IS AN OPERATOR
'b' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'5.3' IS A REAL NUMBER
'*' IS AN OPERATOR
'c' IS A VALID IDENTIFIER
-----
```

(b)

Figura 2.13: programma di test (a) e rispettivo output ottenuto (b)

Capitolo 3

L'analisi sintattica

Dopo aver effettuato l'analisi lessicale di un testo e aver determinato la sequenza di token corrispondenti ai caratteri che sono stati analizzati, c'è da capire se la sequenza di token rispetta le regole della grammatica secondo la quale è stato prodotto l'insieme di caratteri che sono stati scannerizzati, dette anche **regole di produzione**.

Per fare questo, ai token ottenuti, si applica un ulteriore processo di analisi, che viene detta **analisi sintattica**, che serve a determinare se la sequenza di token che mi ha restituito l'analizzatore lessicale è effettivamente una sequenza lecita per il linguaggio che si tiene in considerazione.

L'analisi sintattica è un processo, quindi, che restituisce un cosiddetto **albero sintattico** (*parse tree*), detto anche **albero di derivazione**, della sequenza di token.

L'albero di derivazione di una specifica sequenza di token è univoco: data una sequenza di token s possono esistere diversi alberi sintattici, ma dato un albero sintattico T è univocamente determinata la sequenza di token a cui esso corrisponde. Nel caso in cui, data una sequenza s di token, essa non dovesse rispettare le regole grammaticali che stabilisce il linguaggio, l'analizzatore sintattico dovrà restituire un errore, detto **errore sintattico**.

Di seguito viene specificato uno schema di funzionamento di un analizzatore sintattico.

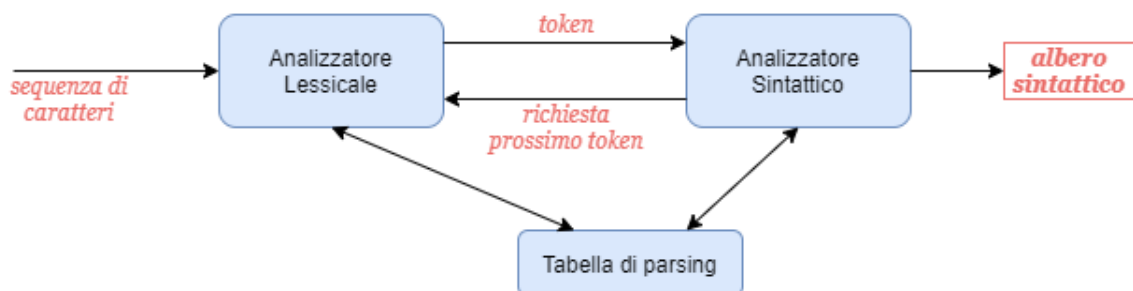


Figura 3.1: funzionamento di un analizzatore lessicale e sintattico

3.1 Grammatiche context-free

Una grammatica **context-free** (*CFG Context-Free Grammar*) è un particolare tipo di grammatica, utilizzata molto dagli analizzatori sintattici, che è possibile esprimere come una quadrupla $G = (T, N, P, S)$ dove:

- **T** rappresenta l'elenco di token, che in analisi sintattica assumono il nome di *simboli terminali*;
- **N** è l'insieme contenente tutti i token che hanno bisogno di ulteriori specificazioni, che vengono detti *simboli non terminali*;
- **P** include tutte le regole di produzione utili a specificare i simboli non terminali;
- **S** è un particolare simbolo non terminale, detto *simbolo iniziale*, a partire dal quale si vanno a specificare tutti i simboli.

Questo tipo di grammatica permette di derivare sequenze di token assumendo inizialmente come sequenza valida il solo simbolo iniziale, e sostituendo poi ciascun simbolo non terminale presente nella sequenza corrente sfruttando una regola di produzione definita per quel simbolo. Inoltre, questo tipo di grammatiche ha il vantaggio di essere molto semplice, di produrre delle tabelle di parsing compatte, e di poter produrre algoritmi associati a questi tipo di grammatiche molto facili e veloci. Di contro però, si ha che non tutte le regole appartenenti al linguaggio che si considera possono essere espresse.

3.2 YACC: generatore di analizzatori sintattici

Così come visto per gli analizzatori lessicali, è possibile sfruttare dei compilatori particolari, che grazie alla compilazioni di un particolare file sorgente tirano fuori un file in linguaggio C, che compilato sviluppa il relativo eseguibile.

Per l'analisi sintattica, un tool che permette di fare questo è **YACC** *Yet Another Compiler Compiler*: esso permette di creare un file in linguaggio C attraverso un file sorgente con estensione *.y* nel quale si descrivono le regole sintattiche che fanno parte della grammatica che si vuole utilizzare.

Il file *.y* permette a YACC di produrre un parser che è di tipo **LALR(1)**. Questo tipo di parser analizza l'input da sinistra a destra, sfrutta un unico simbolo di lookahead per la caratterizzazione dei token, e effettua un tipo di derivazione che viene detto **rightmost**: questo tipo di derivazione permette di avere la riscrittura all'interno delle regole sintattiche corrispondente sempre al simbolo che nella regola compare più a destra.

YACC funziona quindi come indicato nella figura che segue.



Figura 3.2: schema di funzionamento di YACC

3.2.1 Struttura del sorgente YACC

Come visto per FLEX, anche per YACC c'è bisogno di avere un file sorgente scritto con una certa sintassi, e di un compilatore apposito YACC per effettuare la traduzione del file scritto per YACC, in linguaggio C.

Nel caso di YACC, il file sorgente ha estensione `.y` e presenta la seguente sintassi:

```

%{ Prologo %}
Definizioni
%%
Regole
%%
Funzioni aggiuntive
  
```

Così come visto per il file sorgente per FLEX, anche il sorgente che sfrutta YACC si compone di tre parti:

- **Prologo:** in questa sezione vengono inserite macro, inclusioni di file di libreria, dichiarazioni di variabili e di funzioni sfruttate nelle successive sezioni;
- **Definizioni:** in questa sezione sono incluse le definizioni dei token a cui sono associate le regole, descrizione della precedenza e dell'associatività degli operatori e il simbolo di start. Se assente, come simbolo di start viene preso il primo left-value della prima regola di produzione descritta all'interno del file;
- **Regole:** qui vengono descritte tutte le regole di produzione, seguendo una particolare sintassi che sarà descritta in seguito;
- **Funzioni aggiuntive:** in questa sezione compaiono alcune funzioni esterne utili al corretto funzionamento dell'analizzatore, e alcune funzioni obbligatorie, come `main()` che avvia l'analizzatore tramite la funzione specifica `yyparse()`, e la funzione `yyerror()` che viene invocata qualora l'analizzatore riscontrasse problemi durante l'analisi.

3.2.2 Regole di produzione in YACC

Partendo da una regola di produzione scritta in linguaggio naturale, è estremamente semplice andare a costruire la regola stessa, in maniera sintatticamente corretta all'interno del file. In generale la traduzione di una regola del tipo

$$\text{SimboloNonTerminale} \rightarrow \text{termine1} \mid \text{termine2} \mid \dots \mid \text{termineN}$$

viene espressa nel seguente modo:

```
SimboloNonTerminale : termine1 {azioneSemantica 1}
                     | termine2 {azioneSemantica 2}
                     ...
                     | termineN {azioneSemantica N}
                     ;
```

Regole quindi, che hanno stessa testa vengono raggruppate in un solo blocco, e separate tramite un `|`. Alcune regole possono presentare anche un corpo vuoto, cioè privo di termini. Ogni regola è corredata da un insieme di azioni, dette **azioni semantiche**: un'azione semantica è uno stralcio di codice C racchiuso tra `{` e `}` che il parser generato da Yacc deve eseguire tutte le volte che esegue una riduzione secondo la produzione associata. Grazie ad un'azione semantica, spesso è possibile andare ad associare anche delle operazioni di calcolo della testa della regola, rappresentata con il simbolo `$$`, o di un valore *i*-esimo terminale, che si indica con il simbolo `$i`. Facciamo un esempio: la seguente regola di produzione

$$E \rightarrow E + \text{number}$$

tradotta in linguaggio YACC diventa:

```
e : e '+' NUMBER
```

Nell'esempio, `NUMBER` viene considerato token (se supponiamo che ce ne sia la dichiarazione nel prologo), `+` viene considerato come terminale, infine `e` viene considerato per esclusione un simbolo non terminale.

La produzione che specifica il costrutto dell'addizione può essere completata dalla seguente azione che ne calcola il valore parziale del risultato, che possiamo descrivere come segue:

```
e : e '+' NUMBER { $$ = $1 + $3; };
```

3.3 Integrazione tra Lex e YACC

Siccome YACC deve analizzare sequenze di token provenienti da un analizzatore lessicale, quello che si può fare è pensare di far comunicare un analizzatore lessicale come Lex, o Flex, con lo

strumento YACC.

La comunicazione può avvenire molto semplicemente; basta infatti che il programma YACC includa il file elaborato dal compilatore Lex, in modo da andare a visualizzare correttamente le sequenze di token per effettuare la caratterizzazione.

Routine obbligatoria da aggiungere a file .y per far sì che avvenga la comunicazione, è la routine **yylval()** che può essere generata automaticamente da Lex, attraverso la seguente operazione di include all'interno del file YACC:

```
#include "lex.yy.c"
```

Parallelamente, nella sezione Definizioni di un programma Lex, quando lo scanner viene usato in combinazione con il parser, va inserita l'operazione

```
#include "y.tab.h"
```

cioè il file generato da Yacc insieme al file *.tab.c che contiene la definizione dei token. Uno dei generatori di parser che viene usato in particolare insieme al generatore FLEX, è il generatore **BISON**, che genera un parser LALR in linguaggio C, sfruttando un file YACC come sorgente. Nello schema che segue ne è descritto il funzionamento

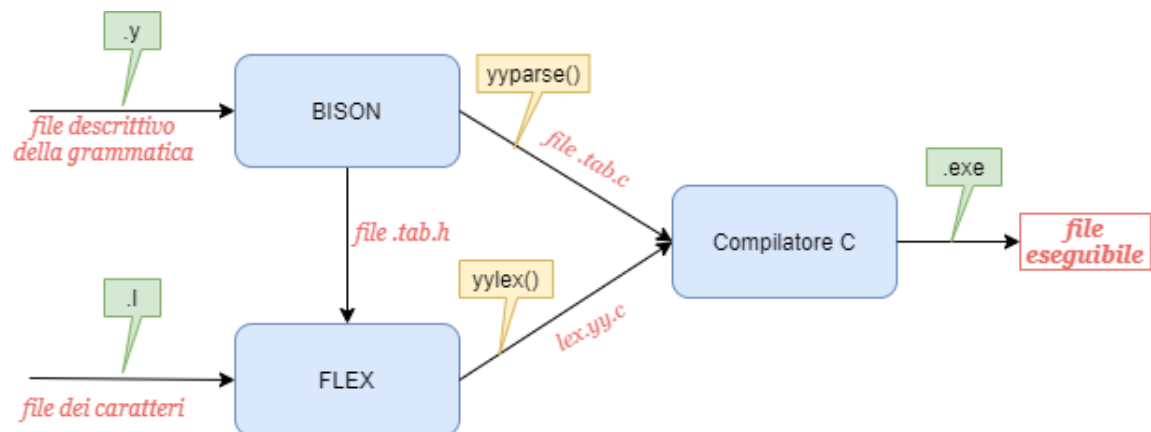


Figura 3.3: schema di funzionamento combinato di BISON e FLEX

Per capire meglio come lavorano i due strumenti in combinata, andiamo a fare un esempio: supponiamo di dover analizzare la seguente sequenza di caratteri

a = b + c * d

Quello che ne deriva dall'elaborazione è descritto nello schema

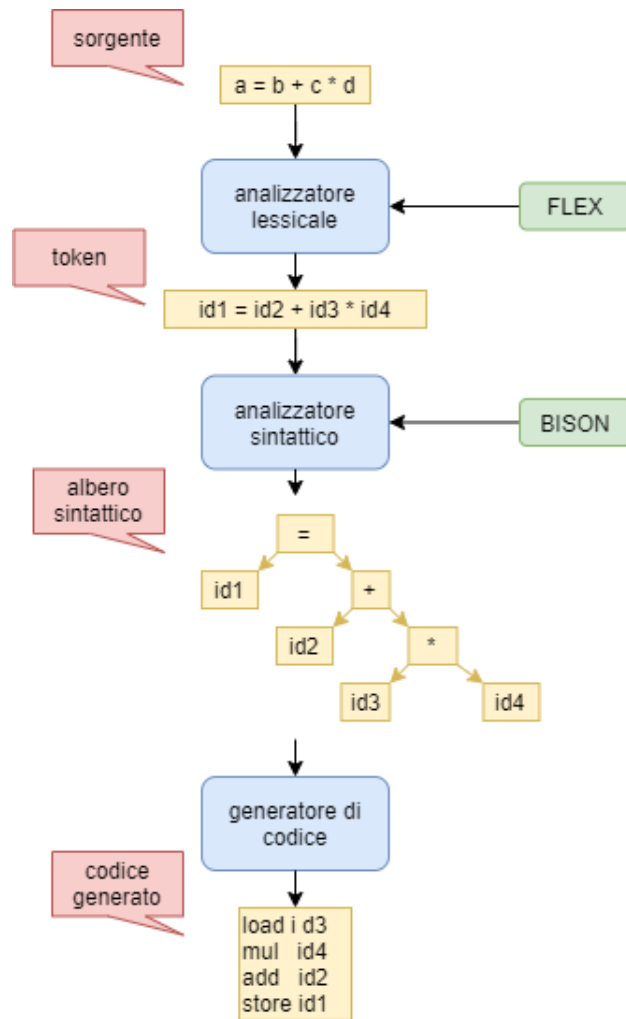


Figura 3.4: esempio di catena tra BISON e FLEX

Capitolo 4

Un analizzatore sintattico: Calcolatrice

4.1 File per l'analisi lessicale

```
1  /*CALCOLATRICE*/
2  %{
3      #include "y.tab.h"
4      #include <stdlib.h>
5  %}
6
7  %%
8
9  [0-9.]+      {
10                 yylval.dval = atof(yytext); //conversione stringa
11                 //in double
12                 return (NUM); //ritorno del token del parser
13             }
14  [-+()*=*/^.\n] {return *yytext;}
15  [\t]         ; //gestione del backspace
16
17  %%
18
19  int yywrap(void) {
20      return 1;
21  }
```


4.2 File per l'analisi sintattica

```
1  %{
2      #include <stdio.h>
3      #include <math.h>      //per usare pow()
4      void yyerror(char *);
5      int yylex(void);
6      double atof(char *);
7  %}
8  %union {double dval; }
9  %token <dval> NUM          //token rappresentante un numero reale (double)
10 %type <dval> expression
11 %left '+' '-'              //precedenza tra gli operatori + e -
12 %left '*' '/'
13 %left NEG
14 %right '^'                 //operatore di elevamento a potenza
15 %%
16
17 program:
18     program statement '\n'
19     | /* NULL */
20     ;
21
22 statement:
23     expression      {printf("%.10g\n", $1); /*stampa il risultato*/}
24     ;
25
26 expression:
27     NUM
28     | expression '+' expression      { $$ = $1 + $3; }
29     | expression '-' expression      { $$ = $1 - $3; }
30     | expression '*' expression      { $$ = $1 * $3; }
31     | expression '/' expression      { $$ = $1 / $3; }
32     | '-' expression %prec NEG       { $$ = -$2; }
33     | expression '^' expression      { $$ = pow($1, $3); }
34     | '(' expression ')'              { $$ = $2; }
35     ;
36
37 %%
38 void yyerror(char *s){          //stampa errori sintattici
39     fprintf(stderr, "%s\n", s);
40 }
41
42 int main(void) {
43     yyparse();
44 }
```

4.3 Prova di esecuzione

```
>bison -y -d calc2.y           //l'opzione -y genera y.tab.c, mentre
                                //-d genera il file tab.h
>flex calc2.l                  //genero il file lex.yy.c
>gcc -c y.tab.c lex.yy.c       //compilo i file C prodotti
                                //da bison e flex
>gcc y.tab.o lex.yy.o -o calcolatrice.exe //creo l'eseguibile usando
                                //i file .o che ha prodotto gcc
>./calcolatrice.exe           //eseguo il file "calcolatrice"
>
>8+2*2                        //comando
>12                            //-->risultato
>5.5*2-10                     //comando
>1                             //-->risultato
>2^5                          //comando
>32                           //-->risultato
```

Bibliografia

- [1] Wikipedia. *Analisi lessicale* - Wikipedia, L'enciclopedia libera. 2018. URL: it.wikipedia.org/w/index.php?title=Analisi_lessicale&oldid=100166786.
- [2] Wikipedia. *Analisi semantica* - Wikipedia, L'enciclopedia libera. 2019. URL: it.wikipedia.org/w/index.php?title=Analisi_semantica&oldid=109141334.
- [3] Wikipedia. *Parsing* - Wikipedia, L'enciclopedia libera. 2019. URL: it.wikipedia.org/w/index.php?title=Parsing&oldid=107124809.