

Università degli studi di Perugia
Facoltà di Scienze MM.FF.NN.
Dipartimento di Informatica e Matematica

Corso di laurea triennale in informatica
Anno accademico: 2003/2004, II semestre

**Corso di Laboratorio
di linguaggi di programmazione e compilatori**

99
Tutte le dispense 2003-2004

v 1.0

Docente: Prof. Simone Brunozzi
telefono: non disponibile
email: labcompilatori@wedoit.us
www: www.wedoit.us/labcompilatori/

file di riferimento:
dispense_lab_compilatori-99_tutto__2004-06-28__v1.0

URI di riferimento:
non disponibile

Questo documento è rilasciato sotto licenza Creative Commons (www.creativecommons.org),
il testo della licenza è reperibile agli URI
<http://creativecommons.org/licenses/by-sa/1.0/>
<http://creativecommons.org/licenses/by-sa/1.0/legalcode>

This work is licensed under the Creative Commons Attribution-ShareAlike License.
To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/1.0/>
or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Attribution



Share-Alike



E' possibile reperire materiale licenziato con Creative Commons presso l'URI
<http://commoncontent.org/>

00 – Introduzione

- 00-A Introduzione al corso
- 00-B Syllabus
- 00-C Elenco studenti
- 00-D Registrazione audio/video delle lezioni
- 00-E Alcune parole sulla licenza d'uso

Log delle modifiche

1.1 .

00 – A Introduzione al corso

Il corso di **“Laboratorio di linguaggi di programmazione e compilatori”** si prefigge lo scopo di illustrare in maniera approfondita il funzionamento dei compilatori, unendo alcune nozioni di teoria ad esempi pratici.

L'utilizzo di linguaggi di programmazione è il tramite essenziale tra uomo e computer; per tale motivo, la **comprensione** di aspetti semantici e sintattici di tali linguaggi e l'utilizzo di alcuni strumenti software può apportare dei benefici a chi programma. Molti aspetti di questo corso, inoltre, **completano** la formazione degli studenti su aspetti che per ovvi motivi non vengono affrontati in maniera approfondita nei corsi di programmazione.

A differenza di molti altri settori della disciplina informatica, quello dei compilatori è ormai relativamente **“stabile”**, perciò quello che si apprende oggi sarà di buon uso, in questo ed altri settori, **anche tra alcuni anni**.

Il corso, pur essendo un **laboratorio** e impegnando gli studenti anche su aspetti pratici, non trascurerà la **base teorica** necessaria per poter affrontare le sessioni in laboratorio con cognizione di causa. Rispetto ad altri corsi di laboratorio gli studenti, perciò, incontreranno **più lezioni teoriche**.

Gli studenti sono invitati a partecipare attivamente alle lezioni, ad intervenire in caso di dubbi o richieste di approfondimenti, a segnalare eventuali errori nelle dispense. Il corso, tuttavia, **non richiede frequenza obbligatoria**.

Il docente potrebbe saltuariamente **contare le presenze in aula**: tale conteggio viene fatto solo **a scopi statistici** e per ottenere un **feedback** delle lezioni, il docente **garantisce** che tali presenze non influiranno minimamente nella valutazione dello studente **in sede di esame**.

Si richiede inoltre di rispettare l'orario delle lezioni ed **evitare**, quando possibile, l'ingresso o l'uscita dall'aula/laboratorio quando la lezione è in corso.

Il docente è sempre **rintracciabile tramite email**, o nei normali orari di consultazione qualora uno o più studenti **abbiano manifestato** la necessità della presenza del docente, possibilmente tramite comunicazione email o direttamente a voce.

Qualora **nessuna richiesta** di consultazione pervenga al docente entro **due ore** precedenti l'inizio della consultazione, il docente stesso si riserva o meno di essere presente per la consultazione stessa.

Il docente presuppone che i suoi allievi abbiano già assimilato determinate **competenze da corsi precedenti**; qualora questa presupposizione risulti scorretta, gli alunni sono invitati a **farlo notare** tempestivamente al docente.

00 – B Syllabus

Il **Syllabus** è una sorta di raccolta di tutte le informazioni utili per seguire il corso ed affrontare l'esame.

Docente:

Prof. Simone Brunozi (labcompilatori@wedoit.us)

telefono: non disponibile

www: non disponibile

Assistente:

nessuno

Orario delle lezioni:

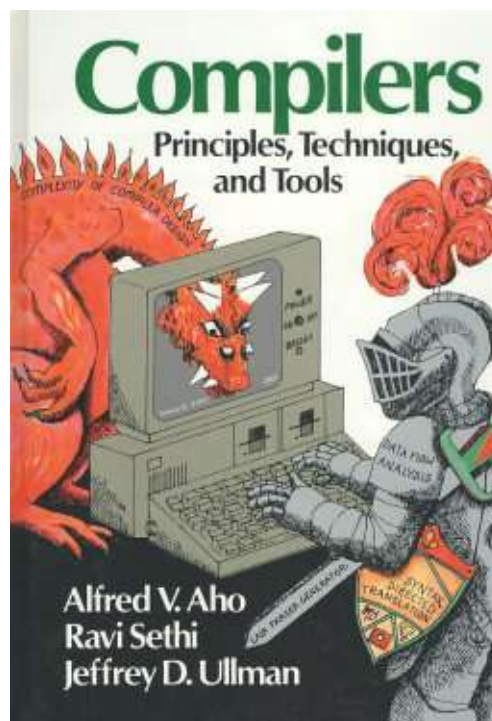
mercoledì dalle 15:00 alle 16:00 (aula A2 oppure in aula laboratorio computer)

venerdì dalle 11:00 alle 13:00 (aula A2)

Testi di riferimento:

- Dispense del corso (la versione del file indica eventuali aggiornamenti o correzioni successivi alla prima pubblicazione);

- testo: **“Compilers – Principles, Techniques, and Tools”** di Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman (il testo, **in lingua inglese**, è disponibile per la consultazione in biblioteca del dipartimento di Informatica e matematica, suggerisco agli studenti di **acquistarlo** solo se davvero necessario);



- testo: **“Linguaggi formali e compilatori”**, di Giorgio Bruno, ed. UTET, disponibile in **italiano** (questo testo è per eventuali approfondimenti personali).

- eventuali altri riferimenti bibliografici potranno essere comunicati a lezione.

Orario e luogo di consultazione:

dalle 14:00 alle 14:40, mercoledì, aula A2 (**non definitivo**)

Reperibilità delle dispense:

Da fotocopiare, in portineria del Dipartimento di Informatica e Matematica, possibilmente dal giorno stesso della presentazione delle dispense a lezione.

Su internet, temporaneamente all'indirizzo www.wedoit.us/labcompilatori/ .

Le **dispense in portineria** saranno presto disponibili in due diversi layout: una pagina per foglio, due pagine per foglio (per chi volesse limitare l'ingombro delle dispense stesse).

Invito gli utilizzatori a riconsegnarle **ordinate e pulite**, e di segnalare al docente eventuali ammanchi o manomissioni.

Il **primo foglio** nel mazzo delle dispense indica l'ultima data di aggiornamento delle versione cartacea delle stesse, ad opera del docente.

Mailing list del corso:

Home page: <http://it.groups.yahoo.com/group/labcompilatoripg/>

Invia messaggio: labcompilatoripg@yahoogroups.com

Iscriviti: labcompilatoripg-subscribe@yahoogroups.com

Annulla iscrizione: labcompilatoripg-unsubscribe@yahoogroups.com

Proprietario lista: labcompilatoripg-owner@yahoogroups.com

Regolamentazione: <http://it.docs.yahoo.com/info/utos.html>

Modalità di esame:

Esame scritto ed esame orale.

Gli studenti possono sostituire l'esame scritto con la realizzazione di un progetto, da discutere col docente.

Lo scritto consiste in esercizi rivolti soprattutto ad aspetti pratici, ed alcuni esercizi di livello più teorico.

L'esame orale serve invece a valutare la preparazione globale dello studente.

Programma del corso:

non ancora disponibile

Obiettivi del corso:

Comprensione degli aspetti formali sintattico/semantici dei linguaggi e dei meccanismi di supporto per interpreti/compileri. Capacità di utilizzare strumenti per la realizzazione di semplici interpreti/parser in ambiente linux.

Propedeuticità:

Corso di programmazione I

Crediti formativi:

tre

Prerequisiti:

Conoscenza dell'ambiente operativo linux, di un linguaggio di programmazione ad alto livello, dell'architettura dei sistemi di elaborazione.

00 – C Elenco studenti

Al fine di poter **comunicare efficacemente** tra di noi, o per condividere le dispense finché non sarà pronto il link su internet, desidero raccogliere **nome, cognome, matricola, indirizzo email e telefono cellulare** di ogni studente.

Per non confondere gli studenti con un'altra iniziativa (mailing list), ritengo utile nominare (forse impropriamente) quella di cui sopra **"newsletter"**.

Nel caso di **variazioni tempestive** degli orari di lezione, qualora i tempi siano troppo stretti per l'utilizzo della email, potrei avvalermi di messaggi **SMS** da inviare al cellulare, per evitare spiacevoli inconvenienti dovuti a cambi di orario o a cancellazione delle lezioni.

Invito pertanto ogni studente ad **inviare una email** all'indirizzo labcompilatori@wedoit.us indicando nel titolo "registrazione", e indicando nel corpo del messaggio **cognome, nome, numero di matricola, telefono cellulare, indirizzo email** (sì, è necessario inserirlo anche nel corpo del messaggio, non basta il mittente). Grazie.

Gli studenti sono **invitati ad unirsi alla mailing list del corso**, in cui sarà possibile discutere di tutto ciò che riguarda il corso.

La mailing list verrà utilizzata dal docente **anche** per comunicazioni ufficiali, che saranno comunque ripetute a lezione.

00 – D Registrazione audio/video delle lezioni

Questo corso partecipa ad una iniziativa del **Nucleo di progettazione universitaria**, che consiste nella **registrazione audio/video** delle lezioni tenute in aula.

A differenza dei più noti strumenti di didattica online, come il consorzio **Nettuno**, questa iniziativa **non punta a sostituire** le lezioni frontali, ma a fornire un **supporto** utile a chi non è in grado di essere fisicamente presente a lezione, o chi necessita di rivisitare una particolare lezione.

Essendo una **sperimentazione**, non sappiamo ancora bene in quale formato, o quale tipo di soluzione multimediale verrà utilizzata per riproporre queste lezioni.

Appena possibile, comunicherò le modalità di fruizione dei contenuti multimediali registrati.

Indicativamente, almeno per l'anno in corso, i contenuti più ingombranti potranno essere fruibili solo su **supporto ottico (CD, DVD)**, e non direttamente da scaricare online.

00 – E Alcune parole sulla licenza d'uso

Tutte le dispense e il materiale del corso fornito dal docente, **fatta eccezione** per tutto ciò la cui paternità e diritti vengano esplicitamente attribuiti ad altri, viene rilasciato sotto licenza **“Creative Commons Attribution-ShareAlike”**, disponibile all'URI <http://creativecommons.org/licenses/by-sa/1.0/>

Per motivi di **validità legale**, tale licenza **non può essere tradotta**, ma deve rimanere nella lingua originariamente utilizzata.

Per meglio comprenderla, tuttavia, sintetizzo in poche righe i punti salienti di questo tipo di licenza:

Chiunque è **libero di** copiare, distribuire, mostrare, ed utilizzare il materiale, realizzare materiale derivato da questo, o utilizzare questo materiale per scopi commerciali, **a patto che**:

- venga riconosciuta la paternità del lavoro all'autore originale;
- Nel caso in cui si alteri, trasformi, o si produca del materiale a partire da questo lavoro, il lavoro risultante venga rilasciato sotto una licenza identica a questa.

Per eventuali dubbi o chiarimenti riguardanti la licenza, siete pregati di contattarmi per email.

Nel caso in cui il mio lavoro venga riutilizzato o modificato, gradirei esserne **messo al corrente**.

01 – Introduzione ai compilatori

- 01-A Compilatori
- 01-B Il modello Analisi-Sintesi
- 01-C Analisi del programma sorgente
- 01-D Sintesi: dal codice intermedio al codice obiettivo

Log delle modifiche

1.1 .

01 – A Compilatori

Un **compilatore** è un software che legge un programma, scritto in **codice sorgente** (*source language*), e lo traduce in un programma equivalente in un altro **linguaggio obiettivo** (*target language*).

Nel corso della **compilazione**, il compilatore può comunicare eventuali **errori** riscontrati.

La varietà dei compilatori è enorme: esistono centinaia di linguaggi di programmazione, dai più noti **C++** e **Pascal** fino a linguaggi specializzati per task particolari; inoltre, il **linguaggio obiettivo** può essere un altro linguaggio di programmazione, oppure il linguaggio macchina di un qualsiasi tipo di computer.

I compilatori vengono spesso classificati in:

- 1) **Passata singola** (*Single-pass compiler*);
- 2) **Passata multipla** (*Multi-pass compiler*);
- 3) **Carica-e-vai** (*Load-and-go compiler*);
- 4) **Rimuovi-errori** (*Debugging compiler*);
- 5) **Ottimizzatore** (*Optimizing compiler*);

Dagli anni '50, in cui apparvero i primi rudimentali compilatori, sono stati fatti **enormi progressi** nella teoria retrostante e nelle tecniche di progettazione e implementazione dei compilatori.

Per tale motivo, **creare un compilatore base** è diventato un compito alla portata di molti, a differenza del primo **compilatore Fortran** che richiese ben diciotto anni-lavoro per essere completato!

La compilazione viene grossolanamente suddivisa in due parti, **analisi e sintesi**, a loro volta suddivise in sottofasi.

Più esplicitamente, le fasi di un compilatore possono essere individuate così:

- 1. Lexical analysis**
- 2. Syntax analysis**
- 3. Semantic analysis**
- 4. Generazione del codice intermedio**
- 5. Ottimizzazione**
- 6. Generazione del codice obiettivo**

Le fasi **1,2,3,4** dipendono dal linguaggio utilizzato (**front-end phases**).

La fase **6** dipende dall'architettura (**back-end phase**).

01 – B Il modello Analisi-Sintesi

La compilazione è suddivisa in **Analisi** e **Sintesi**.

L'analisi suddivide il codice sorgente in **elementi essenziali** e crea una **rappresentazione intermedia** dello stesso.

La **sintesi** costruisce il codice obiettivo desiderato a partire dalla rappresentazione intermedia del codice sorgente.

Delle due fasi, quella di sintesi ovviamente è la più **complessa**.

Nel corso dell'analisi, il codice sorgente viene utilizzato per costruire una **struttura gerarchica ad albero**, spesso chiamato **albero di sintassi** (*syntax tree*). Ogni nodo rappresenta una **operazione**, e il figlio di un nodo rappresenta gli **argomenti di una operazione**.

Ecco un esempio di albero di sintassi per la seguente istruzione:

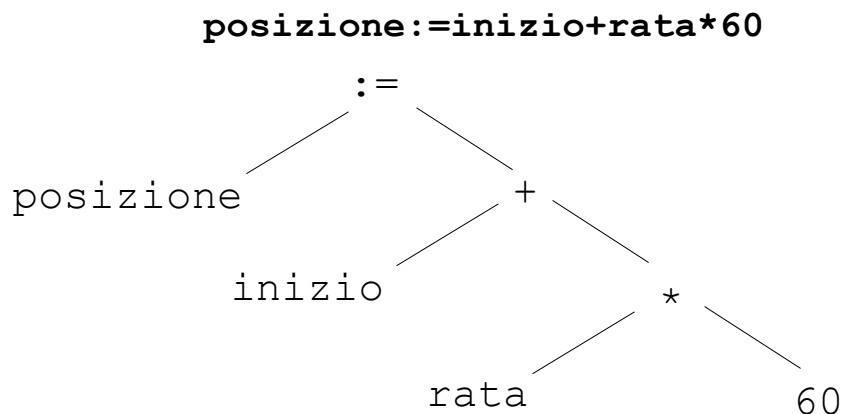


figura 01.01

Molti strumenti software che manipolano codici sorgente effettuano **inizialmente** alcuni tipi di analisi. Alcuni **esempi** di questi software includono *structure editors, pretty printers, static checkers, interpreters*.

Structure editor:

Riceve in input una sequenza di comandi per costruire un codice sorgente, analizza il testo del programma creando una opportuna struttura gerarchica del programma sorgente. Può verificare la **correttezza formale** dell'input o fornire automaticamente **parole chiave** del linguaggio durante la digitazione. L'output di questi tipi di editor è solitamente simile a quello della **fase analitica** del compilatore (*analysis phase*).

Pretty printer:

Analizza un programma e lo ripropone in una maniera in cui la struttura del programma sia **esplicitamente visibile**. Ad esempio i commenti possono apparire con un font speciale di un colore particolare, le dichiarazioni possono apparire con una indentazione proporzionale alla profondità delle stesse, etc.

Static checker:

Legge un programma, lo analizza, e tenta di scoprire potenziali errori senza farlo eseguire, ad esempio cercando parti di programma che non vengono mai eseguite, o quelle in cui una variabile può essere usata prima di essere stata definita. Evidenzia anche **errori logici** come ad esempio tentare di usare una variabile reale come un puntatore (*type-checking*).

Interpreter:

Invece di produrre un programma obiettivo (*target program*), esegue le operazioni indicate nel codice sorgente.

Per un comando di assegnamento, ad esempio, può costruire un albero di sintassi come quello in **figura 01.01**, e poi estrarre le operazioni dai nodi durante la visita dell'albero stesso.

Alla radice scoprirebbe di dover eseguire un assegnamento, perciò chiamerebbe una routine per valutare **l'espressione a destra**, e poi salvare il valore risultante nella variabile a sinistra.

Durante la visita al ramo destro della radice, scoprirebbe di dover eseguire la somma di due espressioni, e in tal modo andrebbe avanti ricorsivamente fino alle foglie dell'albero, e così via.

Gli interpreti sono utilizzati frequentemente per eseguire programmi ad alto livello come **APL** (Array Processing Language), oppure **comandi di script** (si pensi ad una shell di comando linux), poiché solitamente con un comando di script si invoca una routine complessa, come editor, compilatori o altro.

... ..

Solitamente si pensa ad un compilatore come ad un software che traduce un codice sorgente come **C++** nell'assembly di una particolare architettura, tuttavia per i compilatori ci sono **campi di utilizzo non immediatamente intuibili**, come ad esempio *text formatters*, *silicon compilers*, *query interpreters*.

Text formatter:

Prende in input uno stream di caratteri, costituito soprattutto da testo che deve essere formattato, ma anche da comandi che indicano paragrafi, figure, o strutture matematiche. Da questo input viene fatta una analisi, e la successiva sintesi produce un output che viene utilizzato per visualizzare correttamente il testo. Un esempio di text formatter è costituito dal sistema **TEX** (leggi "tec") e dal successivo **LATEX** (leggi "latec") di **Donald Knuth**.

Silicon compiler:

Prende un codice sorgente simile a quello dei comuni linguaggi di programmazione ad alto livello, tuttavia le variabili del linguaggio non rappresentano locazioni di memoria ma **segnali logici** (0,1) o gruppi di segnali in un circuito logico. L'output prodotto è il disegno di un circuito elettronico in un linguaggio specifico.

Query interpreter:

traduce un predicato contenente operatori relazionali o booleani in comandi per ricercare in un **database** dei record che soddisfino tale predicato. Ogni DBMS

(Data Base Management System), come Oracle, DB/2, MySQL, Postgresql, Informix, possiede al suo interno un query interpreter.

... ..

In aggiunta al compilatore, **molti altri programmi** possono essere necessari per creare un programma obiettivo eseguibile.

Un programma sorgente, ad esempio, può essere costituito da moduli suddivisi in file separati, e il compito di raccogliere i vari "spezzoni" del sorgente viene solitamente affidato ad un programma separato, chiamato **preprocessore** (*preprocessor*), che spesso si occupa anche di gestire le **macro** (una serie di comandi e istruzioni che vengono raggruppate insieme per essere eseguite come comando unico).

Il **loader** prende codice macchina rilocabile, modifica gli indirizzi rilocabili piazzando poi le istruzioni modificate nella giusta posizione di memoria.

Il **link editor** effettua il linking con librerie routine esterne.

Un esempio grafico delle varie fasi della compilazione è mostrato in **figura 01.02** (in grassetto i "tool"):

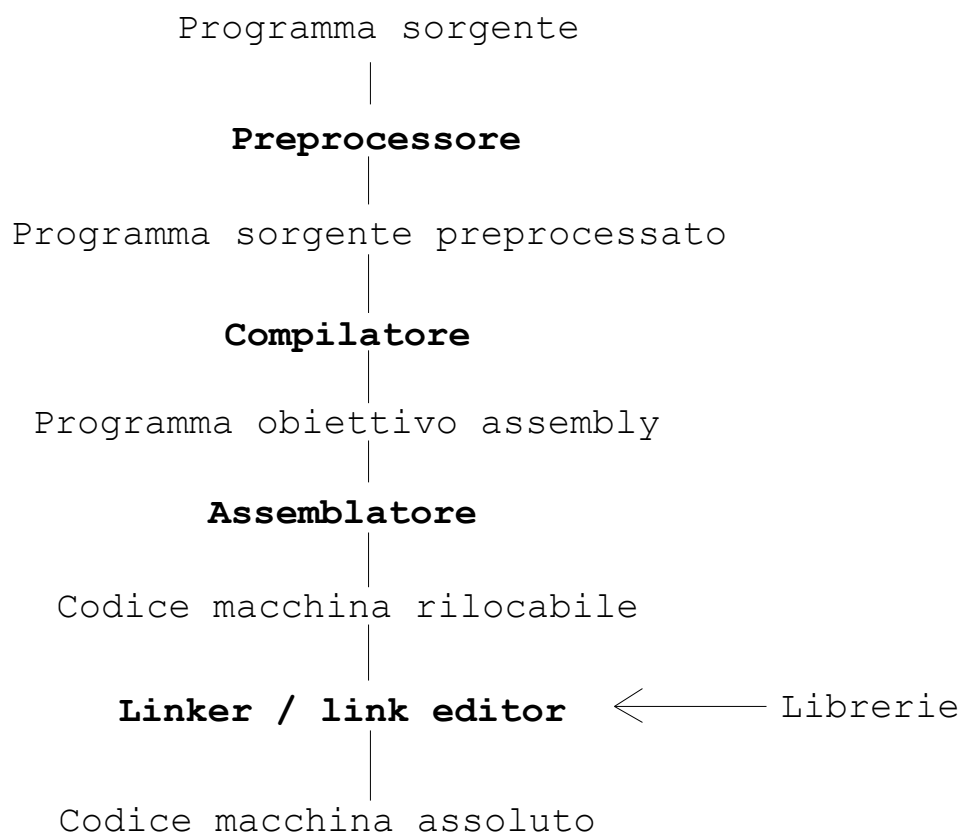


figura 01.02

01 – C Analisi del programma sorgente

L'analisi del programma sorgente si suddivide in **tre fasi** distinte: *linear analysis*, *hierarchical analysis*, *semantic analysis*.

Linear analysis (chiamata spesso **Lexical analysis** oppure **Scanning**):

Lo stream dei caratteri del programma sorgente viene letto da sinistra a destra e raggruppato in **token** (operazione di *tokenizing*), ovvero sequenza di caratteri con un significato collettivo (ad esempio, la parola chiave "WHILE" è composta dai caratteri "W" + "H" + "I" + "L" + "E", che assumono un significato particolare solo se raggruppati insieme). **Token** significa "gettone", "simbolo", "contrassegno".

I vari token, una volta isolati, vengono immagazzinati nella tabella dei simboli (**symbol table**), come nell'esempio seguente:

pos := val + rate * 60;

pos	:=	val	+	rate	*	60	;
ID.0	ASS_OP	ID.1	AR_OP	ID.2	AR_OP	NUM	TERM

ID: identificativo di una variabile
ASS_OP: operazione di assegnamento
AR_OP: operazione aritmetica
NUM: numero
TERM: simbolo terminale della istruzione

Symbol table:

identificativo	nome	tipo	posizione memoria
ID.0	"pos"	int	0x001E
ID.1	"val"	int	...
...

Hierarchical analysis (chiamata spesso **Parsing** oppure **Syntax analysis**):

I token vengono raggruppati gerarchicamente in collezioni annidate, in base al significato collettivo degli stessi. Il **syntax tree** può apparire così:

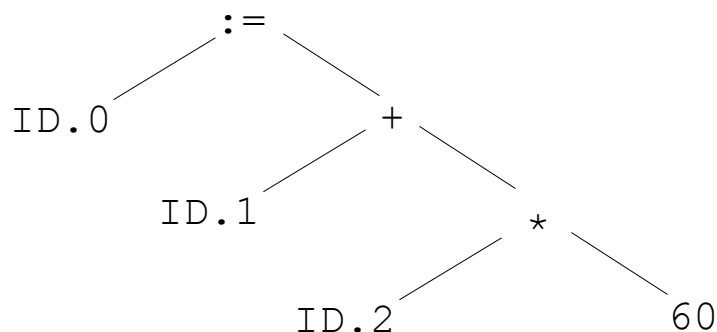


figura 01.03

Semantic analysis:

Vengono eseguiti dei controlli per verificare che i componenti del programma siano stati inseriti correttamente nel programma sorgente.

In particolare si controllano soprattutto i **tipi di dati**, gli **identificatori**, l'uso corretto degli **operatori**.

L'albero di sintassi potrebbe essere modificato in questa maniera, aggiungendo la conversione di tipo **inttoreal** poiché il **type checking** rivela che l'operatore "*" viene applicato alla variabile di tipo reale "rata" e al numero di tipo intero "60".

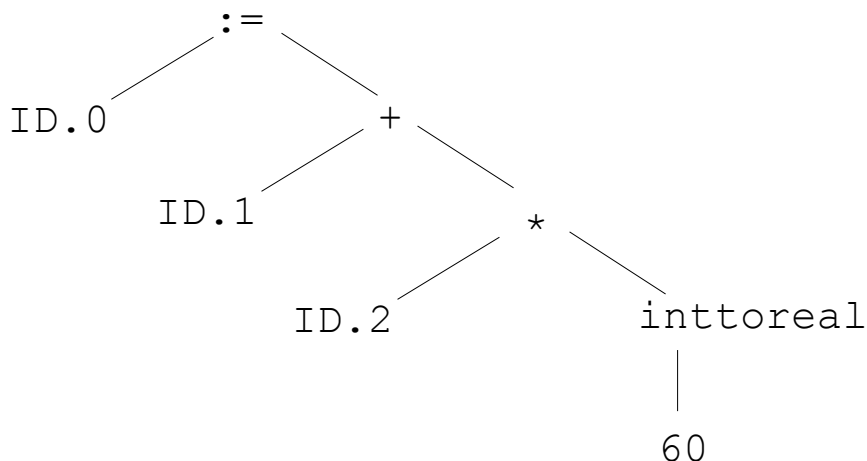


figura 01.04

Nei linguaggi interpretati, infine, l'analisi semantica è meno utilizzata.

01 – D Sintesi: dal codice intermedio al codice obiettivo

Generazione del codice intermedio:

Terminata la fase di sintesi, si passa alla **generazione del codice intermedio**, che non dipende dalla particolare architettura utilizzata.

Il modello utilizzato per rappresentare le informazioni in questa particolare fase viene spesso chiamato **execution model**.

Per essere più chiari, questo potrebbe essere un **esempio** di codice intermedio, relativo al codice sorgente utilizzato in precedenza:

```
temp0 := inttoreal (60)
temp1 := ID.2 * temp0
temp2 := ID.1 + temp1
ID.0 := temp2
```

Questo tipo di codifica viene chiamato **three-address code**, molto simile per impostazione al linguaggio **assembly**.

Ottimizzazione:

Il codice intermedio viene ottimizzato per rendere più efficiente la sua esecuzione, come ad esempio:

```
temp0 := ID.2 * 60.0
ID.0 := ID.1 + temp0
```

Generazione del codice obiettivo (target code):

Come fase finale, dal codice intermedio viene generato il codice obiettivo come nell'esempio seguente, un codice macchina assembly che utilizza due registri:

```
MOVF ID.2, R1
MULF #60.0, R1
MOVF ID.1, R0
ADDF R1, R0
MOVF R0, ID.0
```

la "F" nelle istruzioni indica che si sta operando in floating-point, il "#" indica che il numero va trattato come una costante.

02 – Grammatiche e Turing Machine

- 02-A Introduzione
- 02-B Cenni sulle grammatiche
- 02-C BNF: Backus-Naur Form
- 02-D Cenni sulla macchina di Turing
- 02-E Cenni sulla Gerarchia di Chomsky

Log delle modifiche

1.1 .

02 – A Introduzione

Dopo aver trattato a grandi linee le varie **fasi della compilazione**, spostiamo per un momento il focus su alcuni altri **aspetti teorici** che servono da **base** per poter comprendere le successive lezioni.

Molto presto parleremo delle **espressioni regolari**, attingendo ad alcune nozioni proprie della **teoria degli automi** e della **teoria dei linguaggi formali**.

Le **espressioni regolari** corrispondono alle **grammatiche di tipo 3** (type-3 grammars) della **Gerarchia di Chomsky**, e costituiscono una formulazione per la rappresentazione di **linguaggi regolari**.

Vengono spesso utilizzate per definire modelli di ricerca (**search patterns**), oppure la struttura lessicale dei linguaggi di programmazione.

Prima di addentrarci nei **dettagli** delle espressioni regolari, è bene **introdurre** a grandi linee (ovvero, senza lo stesso livello di dettaglio di un corso teorico):

le **grammatiche**;

le **Macchine di Turing**;

la **Gerarchia di Chomsky**.

02 – B Cenni sulle grammatiche

L'idea di base delle **grammatiche** è che si possono **generare stringhe** partendo con uno speciale **simbolo iniziale** e poi applicando regole che indicano come certe combinazioni di simboli possano essere **rimpiazzate** con altre combinazioni di simboli.

Chiariamo subito con un **esempio**:

Prendiamo come **alfabeto** le lettere "a" e "b", e indichiamo "S" come simbolo di partenza (**starting symbol**). Definiamo inoltre le seguenti regole di produzione (**production rules**):

1. **S --> aSb**
2. **S --> ba**

Partendo da **S** (il simbolo iniziale), applichiamo le seguenti sostituzioni:

S --> aSb --> aaSbb --> aababb

Il **linguaggio definito da questa grammatica**, perciò, consiste in tutte le stringhe che possono essere generate applicando queste due regole, ovvero:

ba, abab, aababb, aaababbb, ecc.

Definizione rigorosa di grammatica formale

Per essere più precisi, una **grammatica formale** è costituita da:

1. **insieme finito N di simboli nonterminali (sequenza di token);**
2. **insieme finito Σ (sigma) di simboli terminali (token) disgiunto da N;**
3. **insieme finito P di regole di produzione della forma:
stringa in $(\Sigma \cup N)^*$ --> stringa in $(\Sigma \cup N)^*$
(* è la chiusura di Kleene (**Kleene closure**), U è l'unione di insiemi)**
4. **un simbolo S contenuto in N che viene indicato come start symbol.**

Vale la restrizione che la **parte sinistra** di una regola deve contenere **almeno** un simbolo nonterminale.

La **chiusura di Kleene** può essere **informalmente** chiarita con due esempi, relativi a un **set di stringhe** e un **set di caratteri** (ϵ (epsilon) indica simbolo vuoto):

$\{"ab", "c"\}^*$ = $\{\epsilon, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "ababc", "abcbab", "abcc", "cabab", "cabcb", "ccab", "ccc", \dots\}$

$\{'a', 'b', 'c'\}^*$ = $\{\epsilon, "a", "b", "c", "aa", "ab", "ac", "ba", "bb", "bc", \dots\}$

Di solito una grammatica formale **G** viene simboleggiata come **G = (N, Σ , P, S)**.

E' possibile **sintetizzare in una unica riga** (utilizzando il pipe "|") le regole di produzione la cui parte sinistra coincide, ad esempio:

num --> 11
num --> 1001

diventa:

num --> 11 | 1001

Il **linguaggio L(G)** definito da questa grammatica è costituito da tutte le stringhe su Σ (insieme dei simboli terminali) che possono essere generate partendo da **S** e applicando le regole di produzione in **P** finchè **non sono più presenti** simboli nonterminali.

Questa "definizione rigorosa", che a prima vista può risultare davvero **difficile** da comprendere, verrà subito chiarita da un **esempio**:

Consideriamo la grammatica **G** in cui

N = {S, B} (simboli nonterminali, scritti in **maiuscolo**)

$\Sigma = \{a, b, c\}$ (simboli terminali, scritti in **minuscolo**)

P consiste nelle seguenti regole di produzione (*production rules*), in cui **S** è lo start symbol:

1. S --> aBSc

2. S --> abc

3. Ba --> aB

4. Bb --> bb

Ecco alcuni esempi di derivazioni di stringhe nel linguaggio **L(G)**, in cui **indichiamo tra parentesi** la regola di produzione utilizzata di volta in volta:

S --> (2) abc

S --> (1) aBSc --> (2) aBabcc --> (3) aaBbcc --> (4) aabbcc

S --> (1) aBSc --> (1) aBaBSc --> (2) aBaBabccc --> (3) aaBBabccc --> (3) aaBaBbccc --> (3) aaaBBbccc --> (4) aaaBbbccc --> (4) aaabbbccc

Dopo questi tre esempi di stringhe, potrebbe risultare chiaro che questa grammatica definisce il linguaggio

{ aⁿbⁿcⁿ | n > 0 }, (in questo caso "|" sta per "tale che")

in cui **aⁿ** denota una stringa composta da un numero **n** di simboli **a**.

Esercizio 02.01

Elencare i vari passaggi per ottenere la stringa **aaaabbbbcccc** .
Una delle soluzioni (in giallo il simbolo sul quale viene applicata la regola):

1,1,1,2,3,3,3,3,3,3,4,4,4

S	--> (1) aBSc	--> (1) aBaBScc
	--> (1) aBaBaBSccc	--> (2) aBaBaBabcccc
	--> (3) aBaBaaBbcccc	--> (3) aBaaBaBbcccc
	--> (3) aBaaaBBbcccc	--> (3) aaBaaBBbcccc
	--> (3) aaaBaBBbcccc	--> (3) aaaaBBBbcccc
	--> (4) aaaaBBbbcccc	--> (4) aaaaBbbbcccc
	--> (4) aaaabbbbcccc	

Esercizio 02.02 (per casa)

Dimostrare che tutte le **stringhe binarie** generate dalla seguente grammatica hanno valori divisibili per **tre**. (suggerimento: si può usare l'induzione sul numero dei nodi nell'albero di parsing)

N = {num}

Σ = {0, 1}

num --> 11 | 1001 | num 0 | num num

num è lo start symbol

Altra domanda: questa grammatica genera **tutte** le stringhe binarie i cui valori sono divisibili per tre?

Soluzione:

Il numero **num** può diventare **11** (3 in decimale), **1001** (9 in decimale), num moltiplicato per 2 (**num 0**), oppure **num num**.

3 e 9 sono divisibili per 3, quindi ogni **trasformazione** del singolo nonterminale **num** in simbolo terminale porta ad un numero divisibile per tre.

Un numero **num** qualsiasi, divisibile per tre, rimane divisibile per tre anche se raddoppiato (effetto dato da **num 0**).

Infine, un numero divisibile per tre, **concatenato** ad un altro numero divisibile per tre, dà ancora un numero divisibile per tre (prova del nove).

Questo significa che tutte le stringhe binarie generate sono divisibili per tre.

Vediamo se questa grammatica genera tutte le stringhe i cui valori sono divisibili per tre.

Per effettuare questa verifica, è sufficiente trovare **almeno una** stringa binaria, il cui equivalentemente decimale sia divisibile per 3, che **non sia generabile** da questa grammatica.

Una delle stringhe è quella del decimale **21** (**10101**), non ottenibile da questa grammatica.

Context-free grammars, regular grammars

Esistono due interessanti categorie di grammatiche, che sono **più restrittive** rispetto alle grammatiche **formali** appena incontrate: **context-free grammars**, e **regular grammars**.

Nelle grammatiche libere dal contesto (**context-free grammars**), la **parte sinistra** delle regole di produzione può essere formata solo da un simbolo nonterminale.

Il linguaggio definito dalla grammatica vista in precedenza ($\{ a^n b^n c^n \mid n > 0 \}$), **non è** context-free a causa delle regole **3 e 4**.

Questo linguaggio, invece, lo è:

$\{ a^n b^n \mid n > 0 \}$

Possiamo infatti definirlo con la grammatica **G2** in questo modo:

$N = \{ S \}$

$\Sigma = \{ a, b \}$

S come start symbol, e le seguenti regole di produzione:

1. $S \rightarrow aSb$

2. $S \rightarrow ab$

Da notare che le regole di produzione possono anche essere indicate con:

$S \rightarrow aSb \mid ab$

esempi di stringhe generate da questa grammatica sono:

$S \rightarrow (2) ab$

$S \rightarrow (1) aSb \rightarrow (2) aabb$

$S \rightarrow (1) aSb \rightarrow (1) aaSbb \rightarrow (2) aaabbb$

Nelle **regular grammars** la parte sinistra delle regole di produzione deve ancora essere un singolo simbolo nonterminale, ma come **ulteriore restrizione** la parte destra può essere:

1. nulla;

2. un singolo simbolo terminale;

3. un singolo simbolo terminale seguito da un simbolo nonterminale

Il linguaggio $\{ a^n b^n \mid n > 0 \}$ perciò, per entrambe le regole, non deriva da una regular grammar, mentre invece deriva da una grammatica regolare il linguaggio:

$\{ a^n b^m \mid m, n > 0 \}$

Infatti tale linguaggio può essere definito dalla seguente grammatica **G3**:

$N = \{S, A, B\}$

$\Sigma = \{a, b\}$

S è lo start symbol, e le regole di produzione sono le seguenti:

1. $S \rightarrow aA$

2. $A \rightarrow aA$

3. $A \rightarrow bB$

4. $B \rightarrow bB$

5. $B \rightarrow \epsilon$

Esercizio 02.03

Data la seguente grammatica libera dal contesto:

$S \rightarrow SS+ \mid SS^* \mid a$

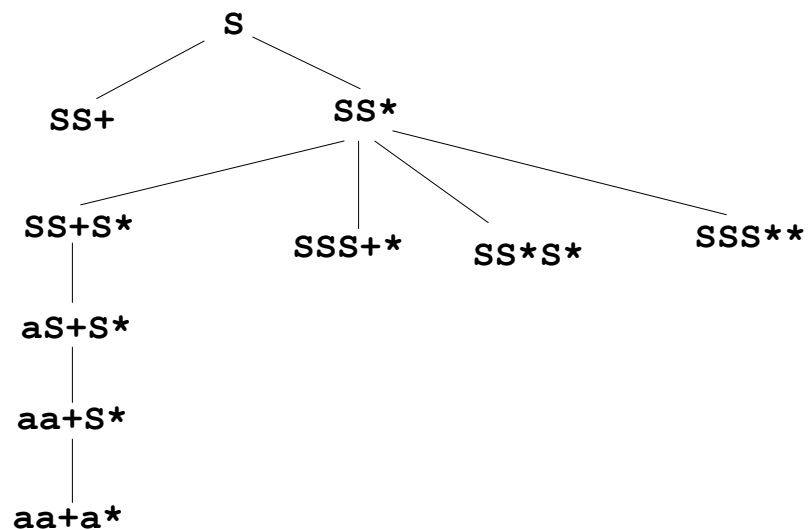
1. mostrare come la stringa **aa+a*** può essere generata;

2. costruire un albero di derivazione per questa stringa.

Soluzione:

$S \rightarrow SS^* \rightarrow SS+S^* \rightarrow aS+S^* \rightarrow aa+S^* \rightarrow aa+a^*$

Albero di derivazione:



Esercizio 02.04

Data la grammatica:

$N = \{S, A, B\}$; $\Sigma = \{a, b\}$

S è lo start symbol, e le regole di produzione sono le seguenti:

1. $S \rightarrow ABS$
2. $S \rightarrow \epsilon$ (ϵ è la stringa vuota)
3. $BA \rightarrow AB$
4. $BS \rightarrow b$
5. $Bb \rightarrow bb$
6. $Ab \rightarrow ab$
7. $Aa \rightarrow aa$

Quale linguaggio definisce?

(soluzione: $a^n b^n$)

Nota: se applico le seguenti regole:

$S \rightarrow (1) ABS \rightarrow (1) \rightarrow ABABS \rightarrow (4) ABAb \rightarrow (6) ABAb$

al termine ottengo una stringa composta da simboli nonterminali e terminali, alla quale **non si può più applicare alcuna regola**. Tale stringa, semplicemente, non fa parte del linguaggio.

02 – C BNF: Backus-Naur Form

La **Backus-Naur Form (BNF)** (anche conosciuta come Backus normal form) è una metasintassi usata per esprimere **context-free grammars**.

In altre parole, è un modo formale per descrivere linguaggi formali.

Viene largamente usata come notazione per le grammatiche dei linguaggi di programmazione, set di comandi e protocolli di comunicazione.

La **BNF** è chiamata così da John **Backus** e (su suggerimento di Donald Knuth) da Peter **Naur**, pionieri dell'informatica e del **compiler design** che introdussero questo formalismo mentre creavano le regole per il linguaggio **Algol 60**.

Una BNF è un insieme di regole di derivazione, scritte nella forma:

<simbolo> ::= <espressione con simboli>

in cui <simbolo> è un nonterminale, e <espressione con simboli> consiste in sequenze di simboli e/o sequenze separate dal carattere pipe "|" che indica una possibile scelta.

I simboli che non appaiono mai a sinistra sono **simboli terminali**.

I simboli all'interno di parentesi quadre (brackets) sono **opzionali**.

Consideriamo, come esempio, la **BNF** per un indirizzo postale statunitense:

<indirizzo> ::= <nome> <via> <parte CAP>

<personale> ::= <nome> | <iniziale> "."

<nome> ::= <personale> <cognome> [<jr>] <EOL> | <personale> <nome>

(la seconda opzione è per chi usa nomi o iniziali multipli)

<via> ::= [<interno>] <numero civico> <nome via> <EOL>

<parte CAP> ::= <città> ", " <provincia> <codice CAP> <EOL>

La Extended Backus-Naur form (**EBNF**) è una variazione della BNF con alcuni costrutti addizionali, introdotta da **Niklaus Wirth** mentre stava sviluppando il **Pascal**.

Da notare che il **W3C** usa una differente **EBNF** per specificare la sintassi **XML**.

(E notate, inoltre, come tante cose che nell'informatica, anche se a prima vista sembrano inutili o preistoriche, in realtà si ritrovano anche nelle tecnologie più moderne).

Esercizio 02.05

Scrivere due esempi di indirizzi postali corretti, e due esempi di indirizzi postali sbagliati, e motivare la correttezza o meno degli stessi.

02 – D Cenni sulla Macchina di Turing (TM)

La **Macchina di Turing**, abbreviata con **TM** (Turing Machine), è un interessante **modello di automa** (*automaton*), introdotto da Alan Turing negli anni '50. Non va confusa con il **test di Turing**, riguardante il settore dell'intelligenza artificiale.

Una **TM** consiste in:

1. Un nastro diviso in celle, una di seguito all'altra. Ogni cella contiene un simbolo di un dato alfabeto finito.

L'alfabeto contiene uno speciale simbolo vuoto, indicato con **'0'**, e uno o più **altri simboli**.

Si assume che il nastro sia **arbitrariamente lungo** a sinistra e a destra.

Le celle non ancora scritte dall'automata si assume che contengano il simbolo '0'.

2. Una testina in grado di leggere e scrivere simboli sul nastro e muoverlo a destra (R per *right*) o a sinistra (L per *left*).

3. Un registro di stato che immagazzina lo stato della **TM**. Il numero dei possibili diversi stati è finito. Esiste uno speciale stato di partenza col quale il registro è inizializzato.

4. Una tabella di azioni (o funzione di transizione), che dice alla macchina quale simbolo scrivere, in che direzione muovere la testina (L o R) e quale sarà lo stato successivo nel registro, basandosi sul simbolo appena letto sul nastro e sullo stato corrente del registro.

Se nella funzione di transizione non c'è una corrispondenza con la attuale combinazione di simbolo e stato, allora la macchina si ferma (**halt**).

Nota che ogni parte della TM è **finita**, ma la potenzialmente illimitata lunghezza del nastro permette un **ammontare di spazio di immagazzinamento** illimitato.

Qui sotto vediamo un esempio di rappresentazione di TM:

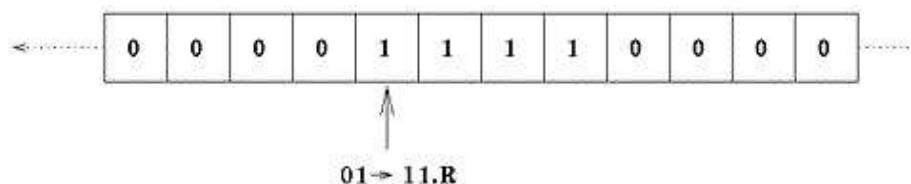


Figura 02.01

Una **realizzazione fisica** in piena regola della TM è ad opera del matematico tedesco **Karl Scherer**, che ne produsse una nel 1986; attualmente questa TM è esposta all'ingresso del Dipartimento di Informatica della Università di **Heidelberg**, in Germania.

Un esempio di Turing Machine

Anche se il nostro scopo è di avere solo una grossolana idea del funzionamento di una TM, un **esempio concreto** può aiutarci a capire di cosa stiamo parlando.

La seguente **TM** ha un alfabeto $\{ '0', '1' \}$, in cui **'0'** è il simbolo vuoto.

Si aspetta una serie di **'1'** sul nastro, con la testina posta inizialmente sul simbolo **'1'** più a sinistra, e **raddoppia gli '1' inserendo dapprima uno zero in mezzo**, ad esempio "111" diventa "1110111".

L'insieme degli stati è $\{ s1, s2, s3, s4, s5 \}$ e lo stato iniziale è **s1**.

Segue la **tabella delle azioni**:

stato preced.	simbolo letto		simbolo scritto	movim.	nuovo stato
s1	1	->	0	R	s2
s2	1	->	1	R	s2
s2	0	->	0	R	s3
s3	1	->	1	R	s3
s3	0	->	1	L	s4
s4	1	->	1	L	s4
s4	0	->	0	L	s5
s5	1	->	1	L	s5
s5	0	->	1	R	s1

Un **esempio di computazione** per questa TM può essere:

Passo	Stato	Nastro
1	s1	11000
2	s2	01000
3	s2	01000
4	s3	01000
5	s4	01010
6	s5	01010
7	s5	01010
8	s1	11010
9	s2	10010
10	s3	10010
11	s3	10010
12	s4	10011
13	s4	10011
14	s5	10011
15	s1	11011
16	-- halt --	

Esercizio 02.06

Verificare il comportamento della precedente TM con un input "111".

Soluzione:

stato preced.	simbolo letto		simbolo scritto	movim.	nuovo stato
s1	1	->	0	R	s2
s2	1	->	1	R	s2
s2	0	->	0	R	s3
s3	1	->	1	R	s3
s3	0	->	1	L	s4
s4	1	->	1	L	s4
s4	0	->	0	L	s5
s5	1	->	1	L	s5
s5	0	->	1	R	s1

La **computazione** per questa TM con nastro di input 111:

```
01  s1  1110000
02  s2  0110000
03  s2  0110000
04  s2  0110000
05  s3  0110000
06  s4  0110100
07  s5  0110100
08  s5  0110100
09  s5  0110100
10  s1  1110100
11  s2  1010100
12  s2  1010100
13  s3  1010100
14  s3  1010100
15  s4  1010110
16  s4  1010110
17  s5  1010110
18  s5  1010110
19  s1  1110110
20  s2  1100110
21  s3  1100110
22  s3  1100110
23  s3  1100110
24  s4  1100111
25  s4  1100111
26  s4  1100111
27  s5  1100111
28  s1  1110111
29  --halt--
```

02 – D Cenni sulla Gerarchia di Chomsky

La **Gerarchia di Chomsky** è una gerarchia di classi di grammatiche formali.

Una **grammatica formale**, come abbiamo visto poco fa, è un modo per descrivere un linguaggio formale.

Un **linguaggio formale** è un insieme di stringhe di lunghezza finita costruito su un alfabeto finito.

Un **automa a stati finiti** (finite-state automaton (**FSA**)) è una macchina astratta che ha solamente un ammontare di memoria finita e costante.

Un **automa pushdown** è simile ad un automa a stati finiti, con l'eccezione che ha accesso a un ammontare di memoria potenzialmente illimitato, nella forma di un singolo **stack** (**pila LIFO** (Last-In First-Out)).

Se diamo ad un automa a stati finiti accesso a **due stack** invece di uno, otteniamo qualcosa di molto più potente di un automa pushdown: l'equivalente di una **Turing Machine**.

In un **automa deterministico**, per dirla con semplicità, per ogni stato esiste al massimo una transizione per ogni possibile input.

In un **automa non-deterministico**, per ogni stato ci può essere più di una transizione per un dato input.

La **Gerarchia di Chomsky** (Chomsky Hierarchy) consiste nei seguenti **livelli**:

Type-0 grammars (unrestricted grammars):

Include tutte le grammatiche formali. Tali grammatiche generano esattamente tutti i linguaggi che possono essere riconosciuti da una macchina di Turing (**TM**). Tali linguaggi sono definiti come tutte le stringhe con le quali la **TM termina**.

Type-1 grammars (context-sensitive grammars):

Generano linguaggi context-sensitive.

Queste grammatiche hanno regole della forma:

aAb --> acb

in cui **A** è simbolo nonterminale e **a,b,c** sono stringhe di terminali e nonterminali. Le stringhe **a,b** possono essere vuote, ma **c** non può esserlo.

La regola **S --> ε** è permessa se **S** non appare mai al lato destro delle regole.

I linguaggi descritti da queste grammatiche sono esattamente tutti i linguaggi riconoscibili da una **TM non-deterministica** il cui nastro è delimitato da una costante moltiplicata per la lunghezza dell'input.

Type-2 grammars (context-free grammars):

Queste grammatiche generano i linguaggi context-free.

Tali linguaggi sono definiti da regole nella forma:

A --> **c**

in cui **A** è simbolo nonterminale e **c** è una stringa di terminali e nonterminali.

I linguaggi descritti da queste grammatiche sono esattamente tutti i linguaggi riconoscibili da un **automa pushdown non-deterministico**.

Inoltre, i linguaggi context-free sono la **base teorica** della sintassi della maggior parte dei linguaggi di programmazione.

Type-3 grammars (regular grammars):

Queste grammatiche generano i linguaggi regolari.

Le regole sono composte a sinistra da un **singolo** nonterminale, e a destra da un **singolo** terminale, eventualmente seguito da un **singolo** nonterminale.

La regola **S** --> ϵ è permessa se **S** non appare mai al lato destro delle regole.

I linguaggi descritti da queste grammatiche sono esattamente tutti i linguaggi riconoscibili da un **automa a stati finiti**.

In aggiunta, la famiglia dei linguaggi formali può essere ottenuta da **espressioni regolari**.

03 – Espressioni regolari

- 03-A Espressioni regolari
- 03-B Sintassi delle espressioni regolari
- 03-C storia di POSIX / SUS
- 03-D Sintassi avanzata delle espressioni regolari
- 03-E Esercizi

Log delle modifiche

1.1 .

03 – A Espressioni regolari

Una **espressione regolare** (abbreviata con regexp o **regex**) è una **stringa** che **descrive** un intero insieme di stringhe, in base a **definite** regole sintattiche.

Le regex sono usate da molti editor di testo e software vario (specialmente nei sistemi operativi Unix e Linux) per **ricercare** testi di una determinata struttura e, ad esempio, **sostituire** le stringhe trovate con altre stringhe.

Come sappiamo, le espressioni regolari corrispondono alle **type-3 grammars** della **Gerarchia di Chomsky**, e descrivono un **linguaggio regolare**.

Alcuni dettagli storici:

Le espressioni regolari hanno origine nella **teoria degli automi** e nella **teoria dei linguaggi formali**, campi che studiano modelli di computazione (**automi**) e modi per descrivere e classificare linguaggi formali.

Un linguaggio formale, infatti, non è altro che un **insieme di stringhe**.

Negli anni '40 **McCulloch e Pitts** descrissero il sistema nervoso umano considerando i neuroni come piccoli automi.

Il matematico **Kleene** (quello della omonima **chiusura**, per intenderci) descrisse successivamente questi modelli usando una propria notazione matematica, chiamata **“regular set”** (insieme regolare).

Ken Thompson (il creatore, con **Dennis Richie**, di **Unix**) inserì questa notazione nell'editor **qed** (leggi q-ed), e successivamente nell'editor Unix chiamato **ed**.

Il passo successivo fu la nascita del ben più noto **grep**.

Da allora le espressioni regolari sono state usate in una moltitudine di software e utility come: **expr, awk, Emacs, vim, lex, e Perl**.

Molti di questi tool usano una implementazione delle espressioni regolari costruita da **Henry Spencer**.

Prima di addentrarci nei formalismi necessari, proviamo a dare un **veloce esempio (informale) di espressione regolare**, cercando di ricordare quanto già visto per le grammatiche.

lettera (lettera | cifra) *

Questa **regex** definisce gli **identificatori nel linguaggio Pascal**: la *pipe* **“|”** indica un **OR**, l'asterisco **“*”** indica **zero o più istanze** della espressione tra parentesi (chiusura di Kleene, ricordate?), mentre la successione di **lettera** e di una espressione (in questo caso tra parentesi) indica **concatenazione**.

In sostanza, questa regex **identifica** tutte le stringhe che iniziano per una **lettera** e sono seguite da zero o più stringhe composte da **lettere o cifre**, ad esempio **a121, bello, g, a6t66a, eccetera, eccetera**.

Espressioni regolari nella teoria dei linguaggi formali

Passiamo ad alcune definizioni più **rigorose**.

Le espressioni regolari sono costituite da **costanti** (che denotano insiemi di stringhe) e **operatori** (che denotano operazioni su questi insiemi).

Ogni espressione regolare "**r**" denota un linguaggio **L(r)**.

Le **regole di definizione** specificano come **L(r)** è formato combinando in vari modi i linguaggi denotati dalle sottoespressioni di "**r**".

Dato un **alfabeto finito** Σ (sigma) di simboli terminali, vengono definite le seguenti **regole** su tale alfabeto:

1. ϵ è una regex che denota l'insieme $\{\epsilon\}$, contenente la stringa vuota.
2. se "**a**" è un simbolo in Σ , allora "**a**" è una regex che denota $\{a\}$.
3. supposto che "**r**" ed "**s**" siano regex e denotino i linguaggi **L(r)** e **L(s)**, seguono le operazioni:

- 3.1 **(r) | (s)** è una regex che denota **L(r) U L(s)**. (set union)
- 3.2 **(r)(s)** è una regex che denota **L(r)L(s)**. (concatenation)
- 3.3 **(r)*** è una regex che denota **(L(r))***. (Kleene star)
- 3.4 **(r)** è una regex che denota **L(r)**. (extra brackets)

Notare che la **regex** "**a**" è diversa dalla **stringa** "**a**" o dal **simbolo** "**a**", nonostante vengano **tutte e tre** indicate con la stessa notazione.

Notare, inoltre, che la operazione **3.4** specifica che è possibile aggiungere una **coppia di parentesi extra** intorno ad una regex.

Per evitare il più possibile l'uso di parentesi, è assunto che la **priorità** sia discendente da 3.3 a 3.2 a 3.1, e ovviamente, **quando non ci sia ambiguità**, è possibile **omettere** le parentesi, ad esempio:

(ab)c viene scritto come **abc** ; **a U (b(c*))** viene scritto come **a U bc***.

In alcuni casi viene aggiunto l'operatore "**~**", ovvero il **NOT**, anche se non necessario, poichè è ottenibile come **combinazione degli altri operatori**.

Valgono i seguenti **assiomi**, utili per maneggiare e convertire le regex (prese da pag. 96 del libro di testo):

1. **r|s = s|r** | è commutativa
2. **r|(s|t) = (r|s)|t** | è associativa
3. **(rs)t = r(st)** concatenazione è associativa
4. **r(s|t) = rs|rt** concatenazione distribuisce su |
(s|t)r = sr|tr
5. **εr = r** ε è elemento neutro per la concatenazione
rε = r
6. **r* = (r|ε)*** relazione tra * ed ε
7. **(r*)* = r*** * è idempotente

Definizioni regolari (regular definitions)

Per convenienza notazionale possiamo dare dei **nomi** ad alcune regex e definire altre regex usando questi nomi come fossero simboli.

Una **definizione regolare**, quindi, è una sequenza di definizioni nella forma:

$$\begin{aligned}d_1 &\rightarrow r_1 \\d_2 &\rightarrow r_2 \\&\dots \\d_n &\rightarrow r_n\end{aligned}$$

in cui ogni d_i è un nome distinto, e ogni r_i è una regex sui simboli nell'alfabeto:

$$\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$$

Riprendendo l'esempio di inizio paragrafo riguardante i nomi degli identificatori in Pascal, potremmo scrivere la loro **definizione regolare** (i puntini di sospensione non sono validi, ma li usiamo in questo contesto per evitare di scrivere troppe lettere):

$$\begin{aligned}\text{lettera} &\rightarrow A|B|C|D| \dots |Z|a|b|c|d| \dots |z \\ \text{cifra} &\rightarrow 0|1|2|3|4|5|6|7|8|9 \\ \text{id} &\rightarrow \text{lettera} (\text{lettera} | \text{cifra})^*\end{aligned}$$

Esempi di espressioni regolari

Sia $\Sigma = \{a,b\}$

1. $a|b$ denota $\{a,b\}$
2. $(a|b)(a|b)$ denota $\{aa,ab,ba,bb\}$
3. $aa|ab|ba|bb$ denota $\{aa,ab,ba,bb\}$
4. $(a|b)^*$ denota tutte le stringhe aventi una delle due lettere "a" oppure "b", il tutto ripetuto zero o più volte $\{a,aa,aaba,abba, bbaa, babababa, bbaab\dots\}$
5. $(a^*b^*)^*$ Secondo voi, denota la stessa cosa dell'esempio 4 ?
6. $a|a^*b$ denota l'insieme contenente la stringa "a" e tutte le stringhe fatte da zero o più "a" seguite da una "b", ad esempio $\{a,b,ab,aab,aaab,aaaab, \dots\}$

Esercizio 03.01

Trovare quattro stringhe di esempio per le seguenti regex:

1. $b^*(ab^*)^*$
2. $(bb | a(bb)^*aa | a(bb)^*(ab | ba)(bb)^*(ab | ba))^*$

03 – B Sintassi delle espressioni regolari

Le tradizionali regex di Unix

La sintassi base delle regex di Unix è ora definita come obsoleta dallo standard **POSIX**, ma è ancora largamente utilizzata per motivi di **retrocompatibilità** (backwards compatibility).

Molte utility di Unix (grep, sed, ecc.) usano ancora questa notazione, che potremmo informalmente chiamare "**Unix regex**".

Ricordiamo che in questa sintassi non esiste rappresentazione dell'operatore "**unione di insiemi**".

In questa sintassi un carattere **uguaglia** (*match*) se stesso (la regex "**a**" corrisponde al carattere "**a**", e così via).

Ovviamente lo **spazio** è considerato un separatore tra stringhe.

Le eccezioni vengono chiamate **metacaratteri**:

- . un qualsiasi singolo carattere.
- [] un singolo carattere tra quelli contenuti tra parentesi quadre.
Ad esempio, **[abc]** corrisponde al carattere "a", o "b", o "c", mentre **[a-z]** corrisponde a tutti i caratteri minuscoli dell'alfabeto italiano.
Il segno "-" tra le lettere viene chiamato "**hyphen**".
- [^] un singolo carattere NON contenuto tra parentesi quadre.
- ^ inizio della linea
- \$ termine della linea
- * zero o più occorrenze del carattere precedente.
- \ il carattere successivo alla barra viene trattato come normale
- \< \> corrisponde all'inizio, o fine, di una parola.
Ad esempio, "**\<uno**" corrisponde a "uno" nella stringa "dammene uno solo", ma non corrisponde a "uno" nella stringa "per me nessuno è perfetto".
- \{x,y\} almeno x e al massimo y occorrenze del blocco precedente.
Ad esempio, "**a\{3,5\}**" corrisponde a "aaa", "aaaa" or "aaaaa".
- | OR tra due condizioni.
- + Una o più occorrenze del carattere o regex precedente.
- ? zero o una occorrenza del carattere o regex precedente.

Esempi dell'uso di metacaratteri:

<code>.at</code>	qualsiasi parola di tre lettere che termina per "at".
<code>[hc]at</code>	corrisponde a "hat" o "cat".
<code>[^b]at</code>	qualsiasi parola di tre lettere che termina per "at" ma che non inizia per "b".
<code>^[hc]at</code>	corrisponde a "hat" o "cat" ma solo all'inizio della riga.
<code>[hc]at\$</code>	corrisponde a "hat" o "cat" ma solo alla fine della riga.
<code>[A-Za-z]</code>	una qualsiasi lettera, minuscola o maiuscola, dell'alfabeto italiano (26 lettere).
<code>B[0-9]{3}</code>	"B" seguito da esattamente tre cifre.
<code>\<alm</code>	stringa che inizia per "alm".
<code>to\></code>	stringa che finisce per "to".

Esercizio 03.02

Per ognuno degli esempi di metacaratteri di cui sopra, dato il seguente **alfabeto finito** Σ :

{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,0,1,2,3,4,5,6,7,8,9}

scrivere (ove possibile) almeno quattro stringhe individuabili da tali regex.

Esercizio 03.03

Per **esercitarsi** nelle espressioni regolari è possibile scaricare il software regex-coach, reperibile per Linux e Windows all'URI:

<http://www.weitz.de/regex-coach/>

Regex Coach utilizza interattivamente regex **Perl-compatibili**. Suggerisco di consultare la documentazione prima di utilizzarlo.

Vediamo qui sotto un esempio dell'interfaccia di regex-coach:

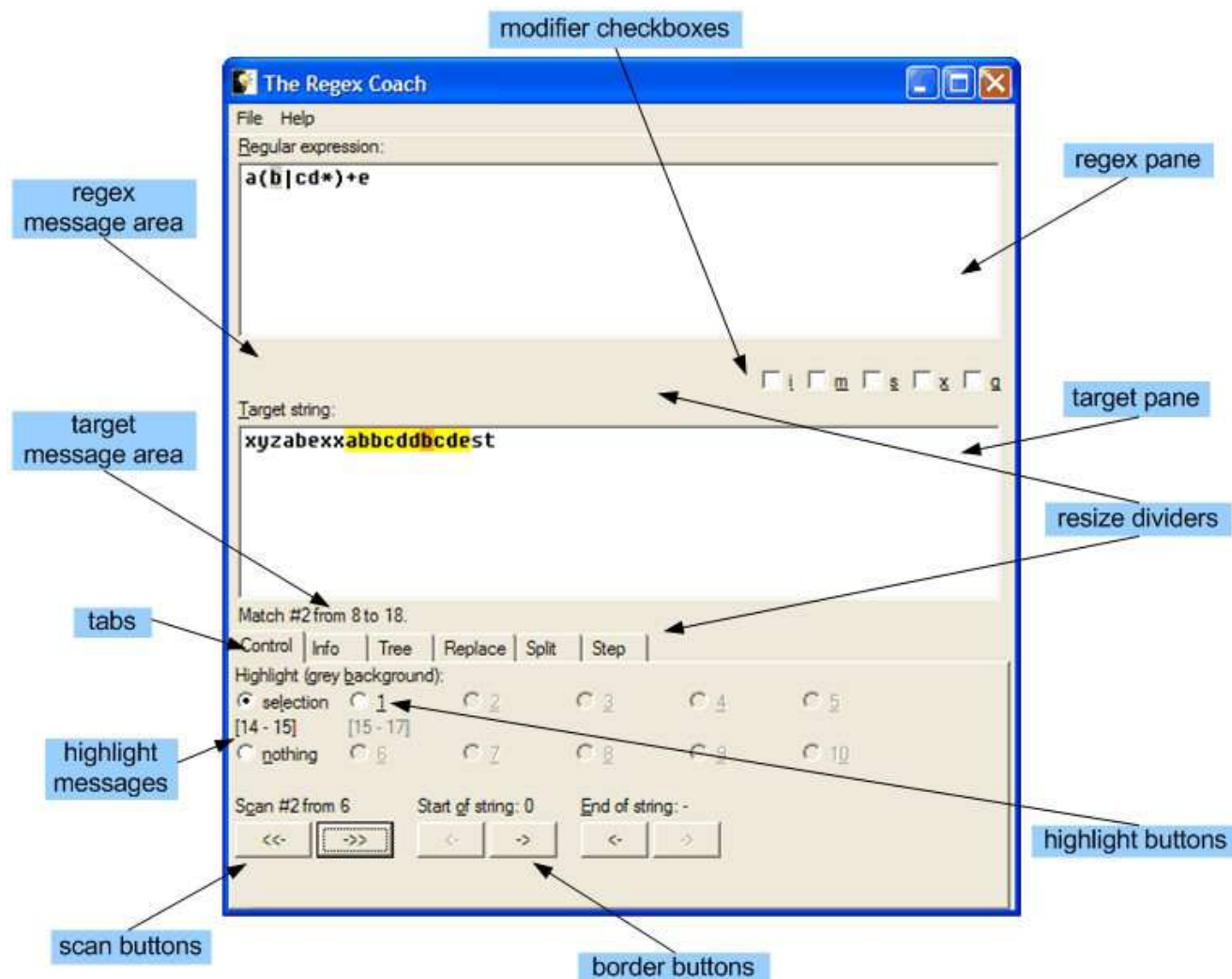


Figura 03.01

Sarebbe apprezzato **preparare un documento in lingua italiana** che spieghi succintamente le caratteristiche e l'utilizzo di tale software, unito ad alcuni esempi di regex.

Nel caso, prego i **volontari** di contattarmi prima di iniziare.

03 – C Storia di POSIX / SUS

Nel campo delle **regex** e, più in generale, negli standard **de iure** (dettati dalle regole) e **de facto** (dettati dal favore degli utenti e del mercato) riguardanti il settore **informatico**, regna una certa confusione, che cercherò di **chiarire** nelle prossime righe prima di affrontare i dettagli puramente **tecnici** dei più recenti standard riguardanti le regex.

Ritengo **utile** questo passaggio per essere in grado di capire bene in quale campo ci si sta muovendo, e a quale standard si sta dando credito e fiducia.

Negli anni '80 emerse la necessità di **potenziare** ulteriormente l'espressività delle regex, mantenendo la retrocompatibilità con le regole precedenti. Da questa necessità nacquero **SUS** e **POSIX**.

Il **SUS** (**S**ingle **U**nix **S**pecification) emerse da un progetto, iniziato nel **1985**, per standardizzare l'interfaccia di programmazione (**API**, **A**pplication **P**rogram **I**nterface) del software disegnato per girare su sistemi operativi **Unix**.

In quel periodo, ricco di tensioni a livello informatico, si rendeva necessario trovare un **dialogo su questioni ritenute importanti** (si parlava anche delle cosiddette **Unix Wars**, vedi http://livinginternet.com/i/iw_unix_war.htm).

Successivamente lo standard **SUS** (versione **1 e 2**) divenne il **IEEE POSIX**, anche se l'originario **SUS** non è stato mai accantonato (versione **3**).

Il **SUS** viene portato avanti da **The Open Group** (i creatori del SUS), un consorzio di industrie del calibro di IBM, Sun Microsystems, Hitachi, Hewlett-Packard e Fujitsu.

Il SUS era precedentemente conosciuto come **Open Software Foundation** (da non confondere con **Free Software Foundation**) e **X/Open Company**.

Dal 1998 la SUS versione **3** viene portata avanti da **Austin Group**.

Il termine **POSIX** è stato suggerito da **Richard Stallman** su richiesta dell'IEEE, ed è un acronimo che significa **P**ortable **O**perating **S**ystem **I**nterface, dove la **X** simboleggia la versione Unix delle API.

POSIX è conosciuto anche come **IEEE-1003**.

IEEE (pronunciato eye-triple-e) sta per **Institute of Electrical and Electronics Engineers**, la più numerosa organizzazione professionale non-profit del mondo, con sede negli Stati Uniti.

Fondata nel **1963**, si occupa di promuovere la conoscenza nel campo dell'ingegneria elettrica ed elettronica, e di stabilire standard per computer,

periferiche e comunicazioni. (www.ieee.org , <http://standards.ieee.org>).

Molti standard sono stati ufficializzati dalla IEEE, come ad esempio la porta parallela (**IEEE-1294**), la porta firewire (**IEEE-1394**), le reti di comunicazione Ethernet (**IEEE 802.1**) e Wireless (**IEEE 802.11**), eccetera.

Dato che **IEEE** ha sempre richiesto **prezzi salati** per la documentazione o la certificazione in standard POSIX, negli anni si è verificata la tendenza ad adeguarsi allo **standard SUS**, che è aperto, accetta suggerimenti da tutti, ed è **gratuitamente** disponibile su internet.

E' possibile reperire la **documentazione completa** riguardante lo standard SUS all'URI <http://www.opengroup.org/onlinepubs/007904975/> .

Segnalo anche questo sintetico documento sugli standard di Unix, scritto dal più che famoso Eric S. Raymond nel testo "The Art of Unix Programming":
<http://www.faqs.org/docs/artu/ch17s02.html>

Linux e SUS

La maggiorparte dei vendor di **Linux** non intendono **pagare** per certificare che le varie versioni delle loro distribuzioni siano compatibili con SUS; inoltre, Linux cambia così **rapidamente** che il processo di ricertificazione richiederebbe delle risorse economiche enormi.

Per tali motivi, per sistemi Linux, molte estensioni e standard "de facto", che in pratica aderiscono quasi perfettamente agli standard SUS, sono forniti dal consorzio Linux Standard Base (<http://www.linuxbase.org/>).

Due parole sulla compatibilità

In alcuni ambienti o regolamentazioni (Perl, Python, **PCRE** (Perl Compatible Regular Expression), ecc.) è possibile riscontrare un **comportamento non esattamente conforme allo standard** per alcune particolari regole delle regex; tale confusione è probabilmente originata dalla separazione degli standard POSIX e SUS.

Per evitare tali problemi è opportuno **consultare la documentazione** del software/linguaggio che si utilizza.

E allora... a chi diamo retta?

In linea di massima le regole per **regex POSIX** corrispondono a quelle per **regex SUS**.

La documentazione a disposizione su internet o sui testi è abbondante per quanto riguarda **SUS**, perciò presenteremo tali regole per espressioni regolari (**RE, Regular Expression**) basandoci su **SUS**.

Invito gli studenti ad effettuare una **ricerca** per conto proprio sull'argomento.

La notazione e costruzione di regole **BRE (Basic Regular Expression)** dovrebbe poter essere applicata a tutto il software che supporta le regex.

Alcuni software supportano anche le **ERE (Extended Regular Expression)**.

Le BRE non supportano alcuni metacaratteri, come ad esempio:

"?", "+", "{", "|", "(", ")"

A detta di **Open Group** (possiamo solo fidarci, ma non verificare direttamente), entrambe le BRE e ERE sono **supportate** dalla **Regular Expression Matching interface** nel volume **System Interfaces** dello standard **IEEE 1003.1-2001** (2001 indica l'anno in cui è stato specificato e redatto).

Nelle prossime dispense vedremo più in dettaglio le specifiche di tali espressioni regolari.

03 – D Sintassi avanzata delle espressioni regolari

Scendiamo più nel **dettaglio** cercando di coprire altre particolarità delle regex secondo lo standard definito da **Open Group**.

E' bene ricordare ancora che possono esserci sottili differenze tra le regex di ambienti e linguaggi diversi, la cui trattazione esula dagli scopi del corso.

Alcune definizioni

collating element: POSIX generalizza la nozione di **carattere** a quella di collating element (dall'inglese "**collate**" : comparare, confrontare).

Lo definisce come "una sequenza di uno o più byte definiti nella corrente *collating sequence* come unità di comparazione".

Questa definizione generalizza la nozione di carattere in **due modi**.

Un singolo carattere può "**mappare**" due o più collating element (ad esempio, il tedesco "es-zet" viene comparato come il collating element "s" seguito da un altro collating element "s").

Due o più caratteri possono "**mappare**" un unico collating element, ad esempio lo spagnolo "LL" viene comparato tra la "L" e la "M" dell'alfabeto. Poiché i collating element preservano l'idea di "carattere", possiamo **considerare le due cose identiche**, per i nostri scopi.

collating symbol: un collating element racchiuso tra [. e .] ;
ad esempio, [.] indica una parentesi quadra, e può essere usato all'interno di parentesi quadre ([**ae**[.] **44**] uguaglia uno dei caratteri seguenti: **ae** **44**).
Come altro esempio, [] [. -] **0**] uguaglia una parentesi quadra destra "]", oppure un collating element compreso tra "-" e "0".

equivalence class: racchiuso tra [= e =] , rappresenta l'insieme di collating element appartenenti alla classe; ad esempio [= **a** =] potrebbe equivalere ad **àâãä**. Questa categoria è comunque **poco chiara** nella documentazione.

character class: racchiuso tra [: e :] , indica un insieme di elementi.

Vediamo alcune character class:

alnum	digit	punct	alpha	graph	space
blank	lower	upper	cntrl	print	xdigit

[: **alnum** :], ad esempio, indica tutti i caratteri alfanumerici.

range operator: lo "hyphen" "-" nella regex [0-9], ovvero un particolare collating element

range expression: l'intervallo **0-9** nella regex [0-9]

Tabella di riferimento dettagliata

REGEX base

^	il campione (pattern) deve apparire all'inizio della stringa. ^ciao uguaglia (match) ogni stringa che inizia per ciao
\$	il campione (pattern) deve apparire alla fine della stringa. vederci\$ uguaglia (match) ogni stringa che finisce per vederci
.	uguaglia qualsiasi carattere
[]	uguaglia un qualsiasi carattere incluso all'interno
[^]	uguaglia un qualsiasi carattere NON incluso all'interno
[-]	uguaglia un qualsiasi carattere nel <i>range</i> indicato
?	zero o una istanza dell'oggetto precedente
+	una o più istanze dell'oggetto precedente
*	zero o più istanze dell'oggetto precedente
()	la regex all'interno viene considerata un oggetto unico. e(tyu)*t uguaglia etyut, etyutyut, etyutyutyut, et
{n}	n volte l'oggetto precedente. [0-9]{4} uguaglia 9405, 4403, 1212, 3402
{n,}	almeno n volte l'oggetto precedente. [0-9]{4,} uguaglia 9405, 4403, 12124, 340267
{n,m}	almeno n volte, al massimo m volte l'oggetto precedente. [0-9]{2,4} uguaglia 94, 440, 1212
 	una delle alternative a sinistra e destra del simbolo "pipe"

Classi di caratteri POSIX

[:alnum:]	un qualsiasi carattere alfanumerico
[:alpha:]	un qualsiasi carattere alfabetico, maiuscolo o minuscolo
[:blank:]	spazio, TAB
[:digit:]	un qualsiasi carattere numerico
[:lower:]	un qualsiasi carattere alfabetico minuscolo
[:upper:]	un qualsiasi carattere alfabetico maiuscolo
[:punct:]	carattere di punteggiatura (ad es. ! . : ;)
[:space:]	spazio, TAB, newline, carriage return

Metacaratteri PERL-compatibili

//	delimitatori di un campione (pattern) /colo?r/ uguaglia color, colour
i	accodato al pattern per una uguaglianza case-insensitive; /colo?r/i uguaglia color, colour, COLOR
\b	delimitazione di parola (word boundary) /\bfred\b/i uguaglia Fred ma non Alfred o Frederick
\B	non-delimitazione di parola (non-word boundary) /\Bfred\B/i uguaglia Frederick ma non Fred
\d	un singolo carattere numerico (digit)
\D	un singolo carattere non-numerico (non-digit)
\w	un singolo carattere alfanumerico, oppure il carattere underscore "_"
\W	un carattere non alfanumerico, e non il carattere underscore "_"
\n	il carattere di newline
\r	il carattere di carriage-return
\t	il TAB
\s	un singolo spazio
\S	un qualsiasi carattere eccetto lo spazio

03 – E Esercizi

Esercizio 03.04

Per ciascuna delle regex a-f, indicare quali delle seguenti stringhe vengono matchate:

- (a) a^*b^*
- (b) $(ab)^*$
- (c) $(a.b)^*$
- (d) $(a|b)^*$
- (e) $a^*b.+a+b.*$
- (f) $b^*a.+b.+a^*$

''	'a'	'b'	'ab'	'ba'
'aa'	'bb'	'aba'	'aaa'	'aab'
'bab'	'abba'	'abab'	'baab'	'aaaa'
'ababa'	'ababab'	'abbabb'	'abbabba'	

Esercizio 03.05 (per casa)

Ricerca su internet almeno sei ambiti applicativi delle regex, cercando di sintetizzare in un file di testo il tipo di ambito, e l'utilizzo che viene fatto delle regex.

Esercizio 03.06

Le seguenti regex vengono utilizzate frequentemente in vari ambiti; studiare tali regex e cercare di **capire** quale ne sia l'utilizzo, fornendo almeno **tre esempi** di stringhe valide e **tre esempi** di stringhe non valide.

Le regex troppo lunghe vengono continuate alla riga successiva.

Regex 1:

```
^[A-Za-z0-9]([_\.\\-]?[a-zA-Z0-9]+)*@([A-Za-z0-9]+)([_\.\\-]?[a-zA-Z0-9]+)*\.[A-Za-z]{2,4}$
```

Regex 2:

```
^(([0-2]*[0-9]+[0-9]+)\.([0-2]*[0-9]+[0-9]+)\.([0-2]*[0-9]+[0-9]+)\.([0-2]*[0-9]+[0-9]+))$
```

Regex 3:

```
^http:\/\/[a-zA-Z0-9\-\.\_]+\.[a-zA-Z]{2,3}(\/\S*)?$
```

Regex 4:

```
(\w[-_\\w]*\w@\w[-_\\w]*\w\.\w{2,3})
```

Regex 5:

```
^((4\d{3})|(5[1-5]\d{2})|(6011))-?\d{4}-?\d{4}-?\d{4}|3[4,7]\d{13}$
```

Regex 6:

```
^#?([a-f]|[A-F]|[0-9]){3}(([a-f]|[A-F]|[0-9]){3})?$
```

Regex 7:

```
((([0][1-9]|[12][0-9]|3[01])([-./])(0[13578]|10|12)([-./])(\d{4}))|((([0][1-9]|[12][0-9]|30)([-./])(0[469]|11)([-./])(\d{4}))|((0[1-9]|1[0-9]|2[0-8])([-./])(02)([-./])(\d{4}))|((29)(\.-|\/)(02)([-./])([02468][048]00))|((29)([-./])(02)([-./])([13579][26]00))|((29)([-./])(02)([-./])([0-9][0-9][0][48]))|((29)([-./])(02)([-./])([0-9][0-9][2468][048]))|((29)([-./])(02)([-./])([0-9][0-9][13579][26]))))
```

Nota: questa regex è molto complessa, e viene qui riportata per chi volesse cimentarsi con cose del genere. Si tratta di un validatore di data in formato italiano. Da notare il comportamento con il mese di febbraio negli anni bisestili.

Uguaglia:

29/02/2000 31/01/2000 30-01-2000

Non uguaglia:

29/02/2002 32/01/2002 10/2/2002

Soluzione regex 1:

validazione di un indirizzo di posta elettronica.

Uguaglia:

he_llo@worl.d.com hel.l-o@wor-ld.museum h1ello@123.com

Non uguaglia:

hi@worl_d.com he&llo@world.co1 .hello@wor#.co.uk hello@154.145.68.12

Soluzione regex 2:

uguaglia semplici indirizzi IP, versione Ipv4.

Uguaglia:

113.173.40.255 171.132.248.57 79.93.28.178

Non uguaglia:

189.57.135 14.190.193999 A.N.D.233

Soluzione regex 3:

Verifica la correttezza di URI (URL è una definizione obsoleta).

Uguaglia:

http://ciao.org http://www.blu.com/index.php

http://ciao.ciccio.net/alex/index.html

Non uguaglia:

ftp://psychopop.org http://www.edsroom/

http://un/pleasant.jarrin.net/markov/index.asp

Soluzione regex 4:

validazione di un indirizzo di posta elettronica, regex Perl-compatibile

Uguaglia:

foo@bar.com foobar@foobar.com.au

Non uguaglia:

foo@bar \$\$\$@bar.com

Soluzione regex 5:

Verifica la validità (ma non la checksum) delle principali carte di credito, incluse VISA (16 cifre, prefisso 4), Mastercard (16 cifre, prefissi 51-52-53-54-55), American Express (lunghezza 15, prefisso 34-37).

Uguaglia:

6011-1111-1111-1111 5423-1111-1111-1111 3411111111111111

Non uguaglia:

4111-111-111-111 3411-1111-1111-111 Visa

Soluzione regex 6:

Verifica la correttezza di codici colore HTML esadecimali.

Uguaglia:

#00ccff #039 fffcc

Non uguaglia:

blue 0x000000 #ff000

04 - Automi

- 04-A Automa a stati finiti (finite state automaton)
- 04-B Trasformazione da regex a NFSA
- 04-C Trasformazione da NFSA a DFSA
- 04-D Minimizzazione di un DFSA

Log delle modifiche

1.1 .

04 – A Automa a stati finiti (finite state automaton)

Un **Automa** (in inglese: *automaton*, plurale *automata*) è una macchina che opera per proprio conto; a volte la stessa parola viene usata come sinonimo di **robot**.

Giusto per **curiosità**, il primo disegno di un automa umanoide è accreditato a Leonardo da Vinci negli anni attorno al 1490: si tratta del famoso “**uomo vitruviano**”, facente parte del cosiddetto “**Canone delle proporzioni**”. L'automa che appare nei disegni di Leonardo sembrerebbe essere in grado di muovere le braccia, girare la testa, e sedersi.

Tornando a noi, una **macchina a stati finiti (FSM, finite state machine)**, detta anche **automa a stati finiti (FSA, finite state automaton)**, è una macchina astratta che ha un ammontare finito e costante di memoria. Gli **stati** interni della macchina **non contengono** alcuna struttura addizionale.

Un **FSA** può essere concettualizzato come un **grafo orientato** (directed graph).

Esiste un numero finito di **stati**, e ogni stato ha delle transizioni verso altri stati. Ad ogni **passo**, in base ad una stringa di input, viene decisa la **transizione** successiva (alcune di esse possono partire da uno stato e tornare a se stesso).

Esistono diversi tipi di automi a stati finiti:

Acceptor (accettore): risponde all'input con “si” o “no”;

Recognizer (riconoscitore): verifica l'input ricevuto secondo regole definite, e può fornire risposte di vario tipo;

Transducer (trasduttore): genera un output a partire da un dato input.

Gli **FSA** possono operare su **linguaggi di parole finite** (il caso standard), **parole infinite** (automi di Rabin, automi di Büchi), o diversi tipi di **alberi** (tree automata).

Come si lega tutto questo con ciò che abbiamo studiato finora?

Le espressioni regolari (**regex**) definiscono dei linguaggi regolari.

Un **riconoscitore (recognizer)** per un linguaggio è un programma che prende in input una stringa “**x**” e verifica che tale stringa appartenga al linguaggio.

Per **verificare** se una stringa appartiene ad un linguaggio, si utilizza una regex come base di partenza per ottenere un riconoscitore per un dato linguaggio.

Questa operazione si effettua costruendo un **diagramma di transizione** generalizzato, chiamato **automa a stati finiti (FSA)**.

Un automa a stati finiti può essere **deterministico** o **non-deterministico**.

Quello **non-deterministico** permette che più di una transizione uscente da uno stato possa essere sullo stesso simbolo di input.

Per fissare questo concetto vediamo un esempio di un semplice FSA, basato sulla espressione regolare:

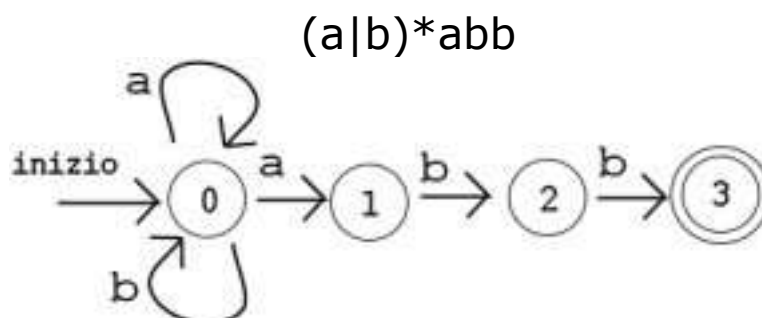


figura 04.01

La regex **$(a|b)^*abb$** uguaglia l'insieme di stringhe di "a" e "b" terminanti per "abb".

Per esercizio, provare il comportamento con la stringa **baababbabaabb**.

Automa a stati finiti nondeterministico

Un **NFSA** (**N**ondeterministic **F**inite **S**tate **A**utomaton) è un modello matematico che consiste in una quintupla (S, Σ, T, s, A) :

- un insieme **S** di stati;
- un insieme di simboli di input Σ (Σ è l'alfabeto dei simboli)
- una funzione di transizione **T** che **mappa** coppie stato-simbolo con insiemi di stati;
- uno stato iniziale **s**;
- un set di stati **A** definiti come stati finali, o stati di accettazione.

Un **NFSA** può essere rappresentato da un **grafo orientato etichettato**, chiamato **grafo di transizione**, in cui i **nodi** sono gli stati e gli **archi etichettati** rappresentano la funzione di transizione.

Lo stesso carattere può etichettare **più di una transizione uscente** da uno stesso stato, e gli archi possono essere etichettati con il simbolo speciale " ϵ ", il cui significato verrà presto chiarito.

Per il **NFSA** della figura **04.01**, l'insieme degli stati è **$\{0,1,2,3\}$** , mentre l'alfabeto Σ consiste in **$\{a,b\}$** .

Lo stato indicato con "0" è lo stato iniziale, quello indicato con "3" è lo stato finale, contraddistinto da un doppio cerchio.

Riepiloghiamo le caratteristiche del comportamento di una NFSA:

- La macchina parte dallo **stato iniziale** e legge una stringa di simboli facenti parte dell'alfabeto;
- La macchina usa la funzione di transizione **T** per determinare il successivo stato, basandosi sullo stato corrente e il simbolo appena letto;
- Se lo stato raggiunto è uno **stato finale**, si dice che la macchina **accetta** la stringa, altrimenti si dice che la macchina **rifiuta** la stringa.
In altre parole un NFSA accetta una data stringa di input **se e solo se** esiste un qualche percorso (**path**) nel grafo di transizione che, partendo dallo stato iniziale e terminando in uno stato finale, verifica una **esatta corrispondenza** tra la stringa data e l'etichettatura degli archi attraversati.
Una frase complicata per un concetto molto semplice ed intuitivo.
- L'insieme di stringhe che la macchina accetta forma un **linguaggio**, che è il linguaggio **riconosciuto** dal NFSA.

La funzione di transizione può essere rappresentata in una maniera intuitiva con una **tabella di transizione**:

Stato	simboli di input	
	a	b
0	{0,1}	{0}
1	-	{2}
2	-	{3}

Automa a stati finiti deterministico

Un **DFSA** (**D**eterministic **F**inite **S**tate **A**utomaton), rispetto al **NFSA**, pone due limitazioni:

- non esistono archi con etichetta "**ε**";
- non esistono due o più archi uscenti dallo stesso nodo e aventi la stessa etichetta.

Le importanti implicazioni di tali limitazioni saranno presto chiare.

Chiariamo innanzitutto il significato della etichetta "**ε**".

"**ε**" non corrisponde alla stringa vuota, ma permette di passare da un nodo ad un altro senza "spendere" un carattere della stringa in input.

L'esistenza di **due o più archi** uscenti dallo stesso nodo e aventi la stessa etichetta implica che non è possibile determinare esattamente il comportamento dell'automa (da qui la dicitura **nondeterministico**).

Esercizio 04.02

Il seguente esempio descrive un **DFSA** con un alfabeto binario. Analizzare le specifiche cercando di capire che tipo di verifica viene fatta da questo automa.

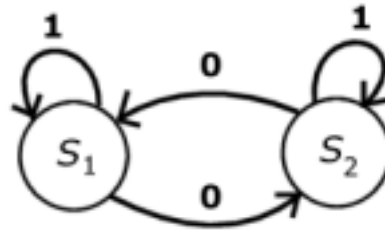


Figura 04.03

$M = (S, \Sigma, T, s, A)$
 $s = S1$

$\Sigma = \{0, 1\}$
 $A = \{S1\}$

$S = \{S1, S2\}$

Tabella di transizione:

Stato	simboli di input	
	0	1
s1	{s2}	{s1}
s2	{s1}	{s2}

Tabella di transizione alternativa:

$T(S1, 0) = S2$

$T(S1, 1) = S1$

$T(S2, 0) = S1$

$T(S2, 1) = S2$

(soluzione: la M verifica che l'input contenga un numero pari di zeri).

Esercizio 04.03

Mostrare **DFSA** che accettino i seguenti linguaggi su un alfabeto $\Sigma = \{0, 1\}$:

- l'insieme di stringhe con tre zeri consecutivi;
- l'insieme di stringhe in cui ogni insieme di cinque simboli consecutivi contenga almeno due zeri;
- l'insieme di tutte le stringhe inizianti per "1" tali che, quando interpretati come una rappresentazione binaria di un numero decimale, siano divisibili per cinque.

04 – B Trasformazione da regex a NFSA

Finora abbiamo introdotto sommariamente una serie di concetti relativi agli automi.

Ora si rende necessario vedere più **in dettaglio** il passaggio **da regex a DFSA** (paragrafo **04-C**), passando per la generazione di un **NFSA** (paragrafo corrente).

Per fare ciò ci avvarremo della cosiddetta "**costruzione di Thompson**", un algoritmo per passare da regex a NFSA, facilmente trasportabile verso un linguaggio di programmazione.

E' bene ricordare che esistono moltissimi algoritmi che compiono la stessa operazione, ognuno con i suoi vantaggi e svantaggi.

La **costruzione di Thompson** ha il grosso vantaggio di poter essere facilmente implementata in un linguaggio di programmazione; di contro, gli automi generati da questo algoritmo sono poco efficienti.

Tale algoritmo è **syntax-directed** (guidato dalla sintassi), ovvero utilizza la struttura sintattica della regex per guidare il processo di costruzione del NFSA.

Ora mostreremo come costruire automi per:

- riconoscere " **ϵ** ";
- riconoscere **qualsiasi simbolo** dell'alfabeto;
- riconoscere espressioni contenenti **OR, concatenazione, chiusura di Kleene**.

Costruzione di Thompson

1) Riconoscimento di **ϵ** :

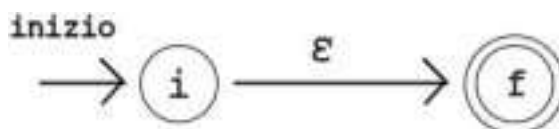


figura 04.04

2) Riconoscimento di **a** nell'alfabeto **Σ** :

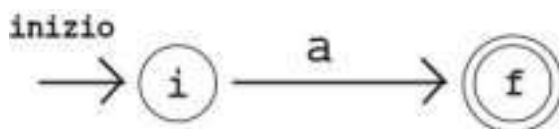


figura 04.05

3) Supposto che $N(s)$ e $N(t)$ siano **NFSA** per la regex $s|t$, si costruisce:

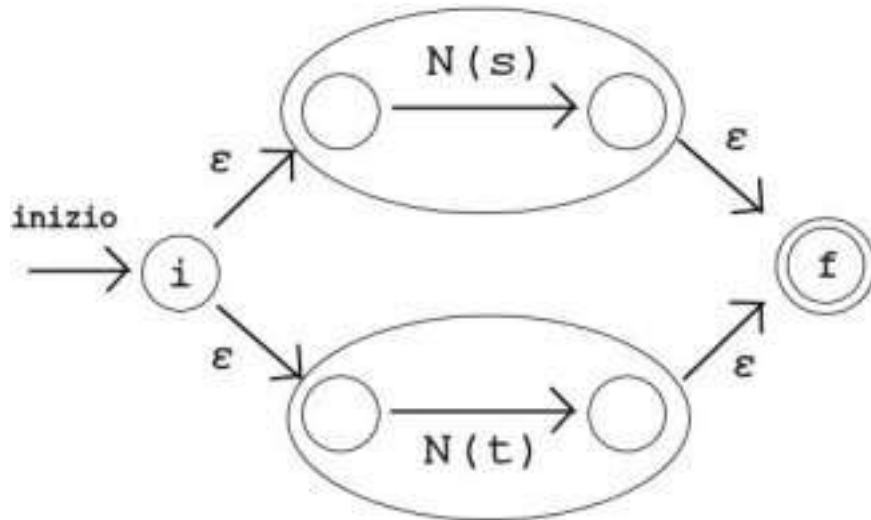


figura 04.06

4) Per la regex st (concatenazione) si costruisce:

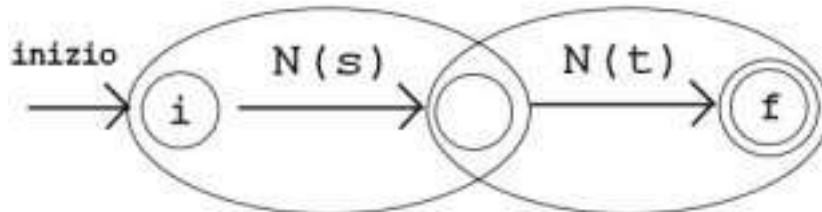


figura 04.07

5) Per la regex s^* si costruisce:

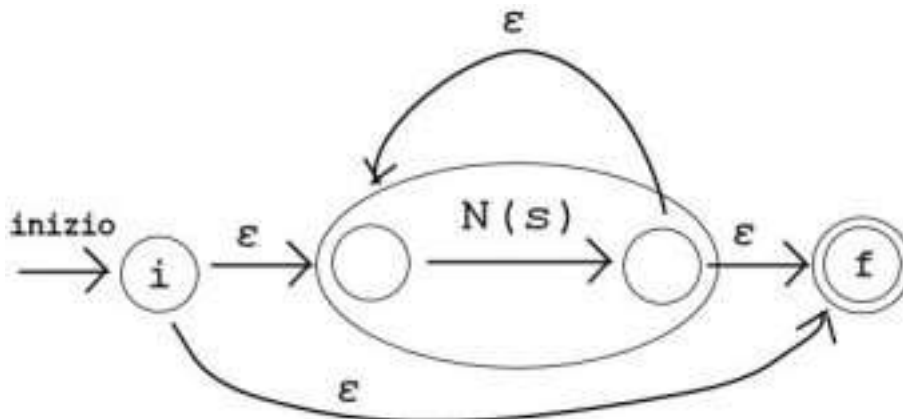


figura 04.08

Ora applicheremo le regole appena elencate ad un esempio concreto, con la regex $(a|b)^*abb$.

Partendo dalla regex costruiamo un albero di parsing:

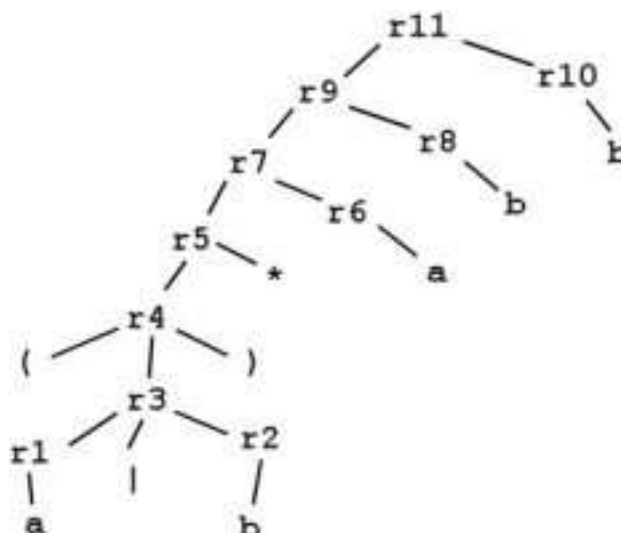


figura 04.09

E da qui applichiamo le regole appena viste per creare il NFSA:

1) Applico la regola 2 per il costituente **r1**:

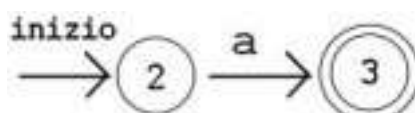


figura 04.10

2) Applico la regola 2 per il costituente **r2**:

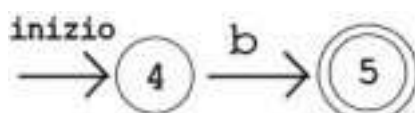


figura 04.11

3) Combinando i due automi di cui sopra e applicando la regola 3 otteniamo l'automa per **r1 | r2**:

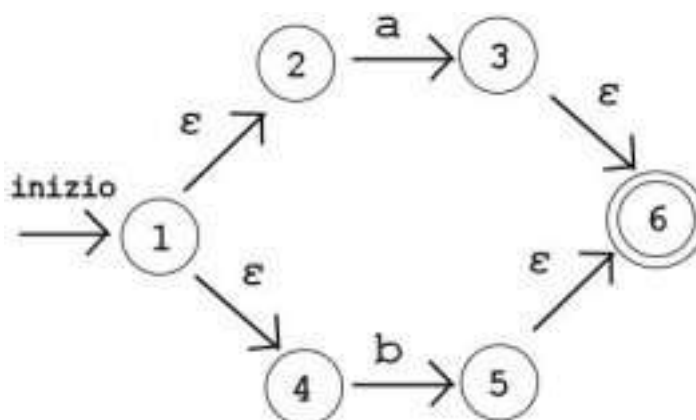


figura 04.12

4) Il NFSA per **(r3)** è identico a quello per **r3**.

5) Con la regola 5 costruisco il NFSA per **(r3)***, ovvero per **r5**:

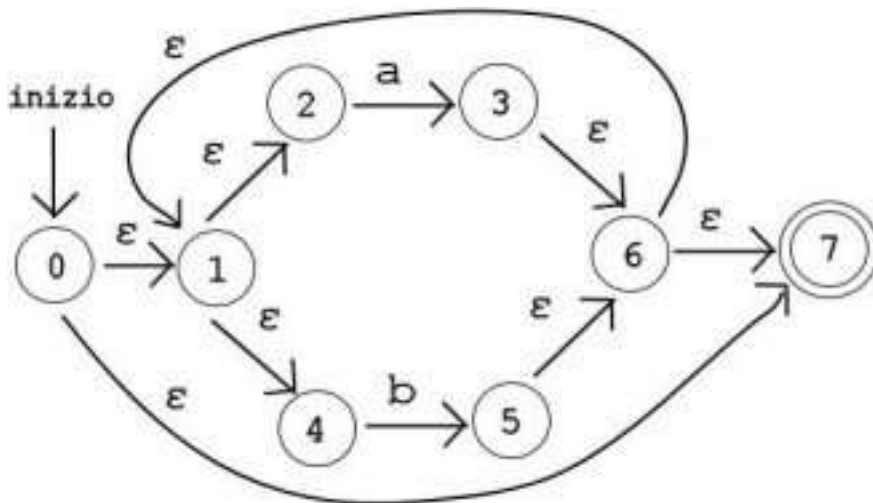


figura 04.13

6) Il NFSA per **r6** si costruisce così con la regola 2:

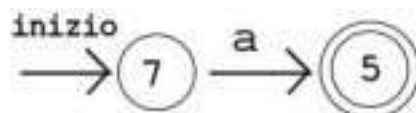


figura 04.14

7) Unisco i due con la regola 4 per formare un NFSA per l'espressione concatenata **r5r6**:

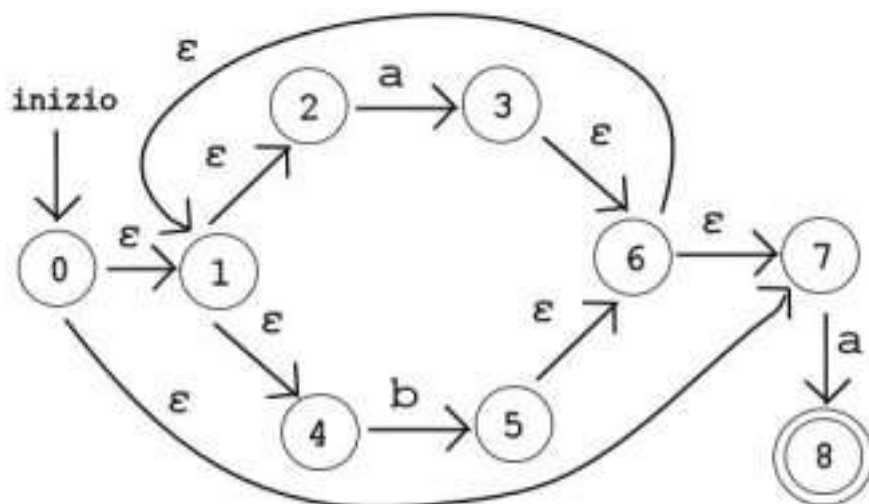


figura 04.15

8) Continuando con lo stesso procedimento e applicando le regole, si ottiene finalmente il **NFSA completo**:

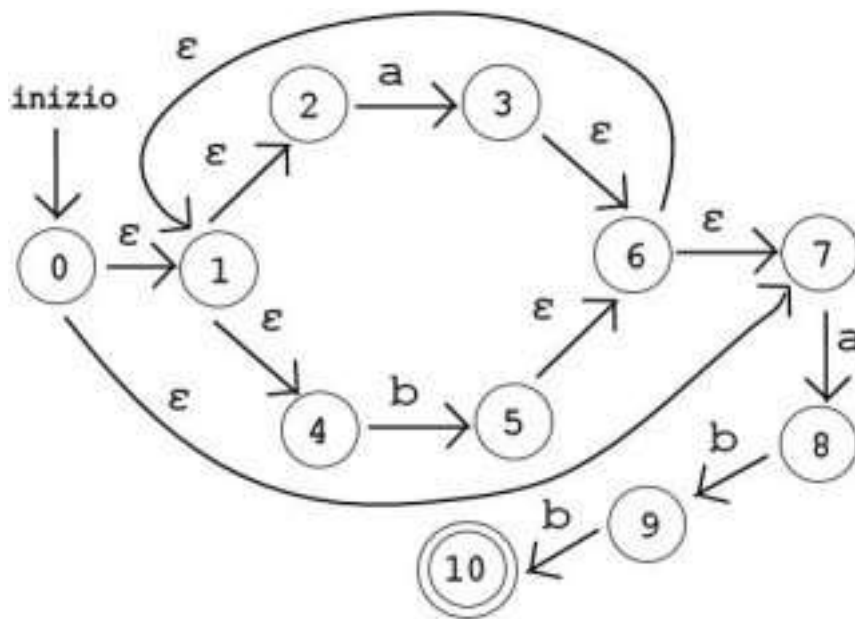


figura 04.16

Limiti del NFSA

Un **NFSA**, per riconoscere una stringa, impiega una quantità di tempo **sconosciuta**, poiché per asserire che una stringa data non sia riconosciuta dall'automa è necessario valutare tutti i possibili percorsi.

In tali casi la visita del grafo viene effettuata con tecniche di **backtracking**, che diminuiscono drasticamente i tempi necessari.

Il **backtracking** consiste nel visitare tutti i possibili **path** qualora non si trovi un percorso che permette all'automa di accettare la stringa; il nome deriva dal fatto che, giunti ad un "punto morto", si torna indietro e si esplorano altri percorsi non ancora analizzati.

Da tali considerazioni emerge spesso la **convenienza** (per stringhe lunghe, ad esempio) di trasformare un NFSA in DFSA per riconoscere lo stesso linguaggio.

Prima di addentrarci nei dettagli, è bene ricordare, inoltre, che in un NFSA è possibile che **la stessa stringa** venga uguagliata da **path** diversi.

04 – C Trasformazione da NFSA a DFSA

Vediamo ora un **algoritmo** per trasformare un NFSA nel DFSA corrispondente.

La definizione delle regole è **difficile da comprendere** immediatamente; seguirà un **esempio** che ci permetterà di fissare bene i concetti che stiamo per incontrare.

Algoritmo NFSA – DFSA (subset construction)

Ogni singolo nodo del **DFSA** corrisponde ad un insieme di nodi del **NFSA**.

Il DFSA usa un nodo per tenere traccia di tutti i possibili stati in cui il NFSA può essere dopo aver letto un qualsiasi simbolo di input.

L'algoritmo utilizza le seguenti **operazioni**:

op_1) ϵ -chiusura (s): insieme di stati del NFSA (incluso "s" stesso) raggiungibili dallo stato "s" del NFSA su archi etichettati con ϵ ;

op_2) ϵ -chiusura (T): insieme di stati raggiungibili da un insieme T di stati; in sostanza, questa operazione restituisce l'unione dell'operazione **ϵ -chiusura (s)** per ogni "s" dell'insieme considerato;

op_3) muovi (T,a): insieme di stati del NFSA in cui esiste una transizione sul simbolo di input "a" da un qualche stato "s" facente parte dell'insieme di stati T.

Definiamo "N" l'insieme degli stati del NFSA e "D" l'insieme degli stati del DFSA. Definiamo **N_k** un particolare nodo **k** dell'insieme **N**, e **D_k** un particolare nodo **k** dell'insieme **D**.

Il **nome** di un qualsiasi nodo del DFSA si ottiene concatenando i nomi dei nodi nel NFSA a lui corrispondenti secondo le operazioni di chiusura di cui sopra (presto vedremo come).

Vediamo i vari passi dell'algoritmo:

P1) viene generato **D₀**, il nodo di partenza del DFSA:

$$\mathbf{D_0 = \epsilon\text{-chiusura (N}_0\text{)} = D_{\epsilon 0}}$$

Questa regola è generalizzata così per un nodo **k**:

$$\mathbf{D_k = \epsilon\text{-chiusura (N}_k\text{)} = D_{\epsilon k}}$$

P2) Per ogni nodo **D_k** vengono effettuate le seguenti operazioni:

P2.1) Gli archi uscenti da **D_k** sono quelli uscenti da **D _{ϵ k}** escludendo quelli etichettati con ϵ ; non è ammesso più di un arco con la stessa etichetta;

P2.2) Il nodo nel quale termina un arco viene calcolato così:

- Definiamo $SR_{a,k}$ l'insieme degli stati del NFSA raggiungibili partendo da un qualunque stato appartenente a $D_{\epsilon,k}$ e percorrendo un qualsiasi arco etichettato con "a" oppure con " ϵ " (**ϵ -chiusura ($D_{\epsilon,k}$)**).
L'intera operazione corrisponde, in pratica, alla **op_3**, ovvero a **muovi ($D_{\epsilon,k},a$)**.

P3) una volta completato il passo **2** per ogni nodo, etichettiamo come "**finali**" quegli stati del DFSA in cui **almeno uno** dei nodi NFSA che ne compongono il nome sia uno stato finale nel NFSA.

Esempio NFSA - DFSA

Partiamo dal **NFSA** di figura 04.16 a pagina 12, e trasformiamolo in **DFSA**.

Il passo **P1** dell'algoritmo prevede la generazione di **D0** :

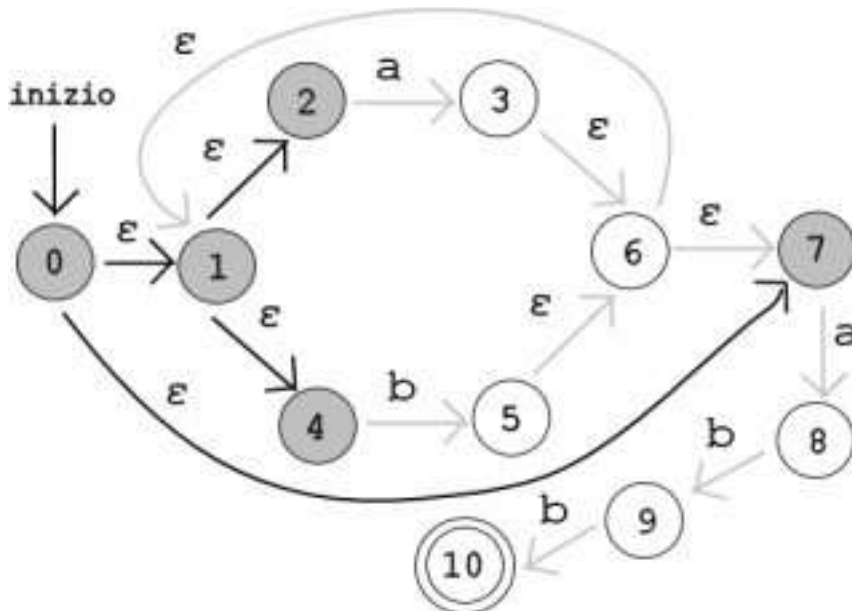


figura 04.17

I nodi con sfondo grigio possono essere raggiunti da **N0** su archi (lasciati in nero) etichettati con ϵ ; gli altri archi sono lasciati in grigio chiaro.

Da ciò concludiamo:

$D_0 = (0,1,2,4,7)$

Il passo **P2.1** consiste nel prendere **D0** e trovare gli archi uscenti, e i nodi nei quali quegli stessi archi terminano.

I due archi sono **a** e **b**, uscenti dai nodi 2 e 4 rispettivamente.

Ognuno di essi termina in un nodo DFSA la cui denominazione viene stabilita al passo **P2.2**.

Proseguendo poi lo stesso procedimento per i nodi che via via vengono generati,

si arriva finalmente al **DFSA completo**, visibile nella figura seguente:

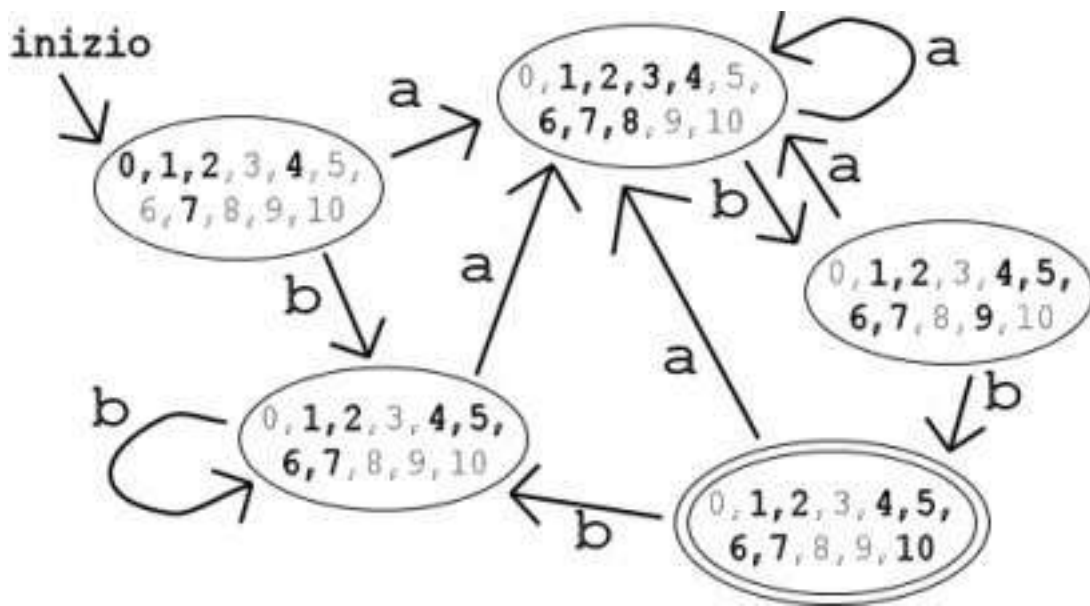


figura 04.18

Possiamo tranquillamente rinominare gli stati trovati con una notazione più comoda:

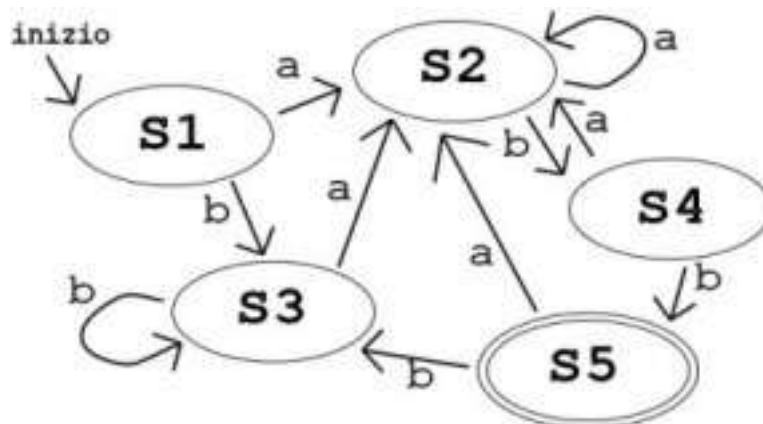


figura 04.19

Ora vedremo come è possibile **minimizzare** il DFA ottenuto.

04 – D Minimizzazione di un DFSA

In questa sezione mostriamo come trasformare un DFSA in un altro DFSA, equivalente al primo, ma con un numero di stati **minimo**. Questa operazione viene chiamata "minimizzazione di un DFSA".

Come al solito usufruiremo di esempi per meglio comprendere i concetti presentati.

Algoritmo di minimizzazione di un DFSA

Supponiamo di avere un **DFSA** che chiamiamo **Pluto**, con un insieme **S** di stati e un alfabeto Σ di simboli di input.

Assumiamo che ogni stato abbia una transazione su **ogni** simbolo di input. Nel caso in cui ciò non corrisponda al nostro **DFSA**, introduciamo un nuovo "stato morto" **M**, con transizione da **M** ad **M** su tutti gli input, e aggiungiamo una transizione da ogni stato **S** allo stato **M** sugli input **mancanti**, come nella figura di esempio qui sotto.

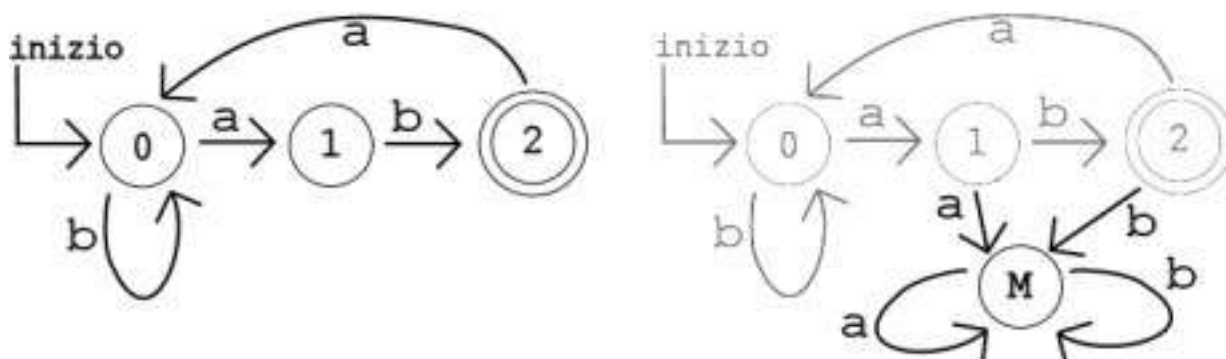


figura 04.20

Possiamo dire che la stringa **W distingue** lo stato **A** dallo stato **B** se, partendo dallo stato **A** di Pluto e leggendo un input **W**, finiamo in uno stato **finale** (accepting state), ma se partiamo dallo stato **B** di Pluto e leggiamo lo stesso input **W**, finiamo in uno stato **non finale** (non accepting state).

Il nostro algoritmo trova tutti i gruppi di stati che possono essere distinti da una qualche stringa di input.

Vediamo insieme i vari passi del nostro algoritmo.

Utilizziamo il DFSA visto in precedenza:

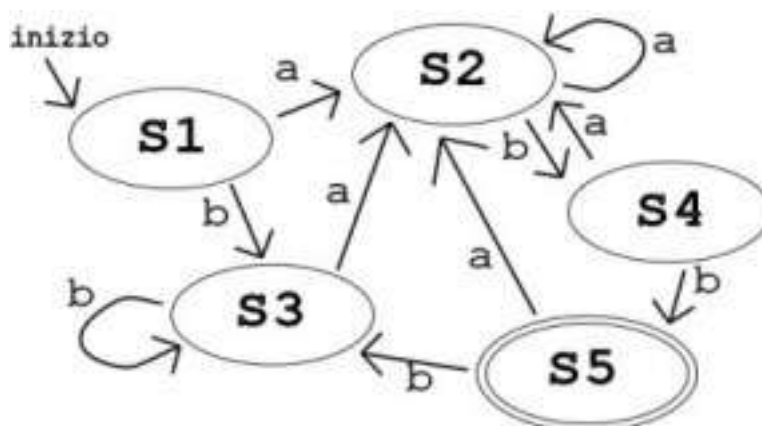


figura 04.21

Passo 1: costruisci una partizione iniziale degli stati, raggruppandoli in **F** (stati finali) e **N** (stati non finali). **F** e **N** sono due partizioni dell'insieme di tutti gli stati di Pluto. La **partizione P** di Pluto è pertanto **F** ed **N**.

Nel nostro caso, $F=S5$, $N=S1,S2,S3,S4$

Passo 2: Per entrambi i gruppi **F** ed **N**, partiziona il gruppo in sottogruppi, tali che due stati generici **Sx** e **Sy** sono nello stesso sottogruppo se e solo se per tutti i simboli di un generico input **a**, gli stati **Sx** e **Sy** hanno transizioni su **a** verso stati dello stesso gruppo **F** o dello stesso gruppo **N**.

Nel caso peggiore, uno stato farà parte da solo di un sottogruppo.

Fatto ciò, rimpiazzare il gruppo considerato (**F** o **N**) con l'insieme dei sottogruppi così creati nella partizione **Pnuova**.

Passo 3: se $P_{nuova}=P$, allora $P_{finale}=P$, e puoi continuare al **passo 4**.

Altrimenti, ripetere il passo 2 con **Pnuova** invece che con la **P** iniziale.

Passo 4: Scegli uno stato in ogni gruppo della partizione **Pfinale** come rappresentante per quel gruppo.

L'insieme dei rappresentanti formerà il nuovo DFSA **Pluto ridotto**.

Capiamo meglio la dinamica di tutto ciò.

Sia **Sx** un rappresentante, e supponiamo che su un input **a** ci sia una transizione di Pluto da **Sx** a **Sy**.

Sia **Sz** il rappresentante del gruppo di **Sy**; segue che **Pluto ridotto** ha una transizione da **Sx** a **Sz** su input **a**.

Passo 5: Se **Pluto ridotto** ha uno "stato morto" **M**, allora va rimosso. Vanno rimossi anche tutti gli stati non raggiungibili dallo stato iniziale.

Vediamo l'esecuzione di questo algoritmo per il DFSA di figura 04.21.

Esercizio 04.03

Minimizzazione di un DFSA.

Passo 1:

$F=(S5)$ $N=(S1,S2,S3,S4)$

Passo 2:

Consideriamo la partizione F , composta solo dallo stato $S5$; poiché tale partizione è costituita da un solo stato, non può essere ulteriormente partizionata. Avremo quindi P_{nuova} costituita dal gruppo $(S5)$.

L'algoritmo passa poi al gruppo $N=(S1,S2,S3,S4)$.

Sull'input a , ognuno di questi stati ha una transizione verso $S2$.

Sull'input b , invece, $(S1,S2,S3)$ vanno verso membri del proprio gruppo, mentre $(S4)$ va verso membri di un altro gruppo $(S5)$. Da ciò segue che in P_{nuova} il gruppo $(S1,S2,S3,S4)$ va diviso in due nuovi gruppi.

P_{nuova} perciò diventa $(S1,S2,S3)(S4)(S5)$.

Eseguendo di nuovo questo passo, non abbiamo un partizionamento su input a , ma sull'input b si verifica un partizionamento per il gruppo $(S1,S2,S3)$.

Il nuovo partizionamento di P_{nuova} diventa $(S1,S3)(S2)(S4)(S5)$.

Eseguiamo di nuovo questo passo ma questa volta non si verifica alcun partizionamento ulteriore, dato che l'unico gruppo a poter essere partizionato, $(S1,S3)$, ha transizioni comuni con qualsiasi input: su a entrambi gli stati vanno verso $S2$, su b vanno entrambi verso $S3$.

Passo 4:

Scegliamo $S1$ come rappresentante del gruppo $(S1,S3)$ e otteniamo il DFSA ridotto, rappresentato qui sotto:

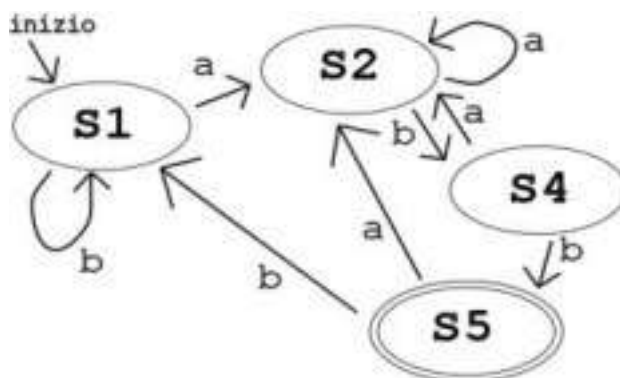


figura 04.22

Esercizio 04.04

Partendo dalla regex $(a|b)^*abb(a|b)^*$, creare l'albero di parsing, costruire il corrispondente NFSA, passare al DFSA e minimizzarlo.

05 – Vim e grep

- 05-A Introduzione
- 05-B Editor vim
- 05-C Comando grep

Log delle modifiche

1.1 .

05 – A Introduzione

Dopo tanta teoria e (per fortuna) un po' di esercizi pratici, è giunto il momento di mettere le mani sulla **tastiera** e fare un po' di esperimenti.

I due "strumenti" che presenteremo tra poco sono il comando **grep** e il software **flex**, preceduti da una breve introduzione all'editor testuale **vim**.

"**grep**" è un comando di **shell** (Unix / Linux) che in sostanza ricerca linee in un testo o in un file, tali che corrispondano ad un dato **pattern** (*campione*), stampando sullo standard output le righe trovate.

Ovviamente è poi possibile modificare il comportamento di **grep** con varie opzioni, che vedremo in dettaglio.

"**flex**" è un software rilasciato sotto licenza **GPL** (**GNU Public License**), il cui nome significa "**fast lexical analyzer generator**", si tratta cioè di un generatore di analizzatori lessicali.

Partendo da un file editato in una certa maniera e "compilandolo" con **flex**, si ottiene un **sorgente in linguaggio C** in grado, una volta compilato col solito compilatore C, di effettuare analisi lessicali.

Prima di affrontare i dettagli di **grep** e **flex**, però, è necessario dotarsi degli strumenti giusti per editare file: stiamo parlando di **vim**, un potente **editor testuale** la cui difficoltà e complessità vengono ben ripagate dalle sue caratteristiche.

Per il nostro corso ho preferito **vim** ad altri editor come **emacs** perchè ritengo importante il poter trovare **vim** su **OGNI** ambiente Unix / Linux, e il poterlo utilizzare anche con limitate risorse hardware. **Emacs**, inoltre, pur essendo davvero potente, è più complesso di **vim**, tanto che sarebbe necessario un intero corso a sé per imparare ad usarlo in maniera completa e produttiva.

Ciò non impedisce agli studenti già familiari con **emacs** di utilizzarlo al posto di **vim** per tutti gli esercizi che seguono; tuttavia, dato che **vim** viene trattato a lezione, è ragionevole aspettarsi che farà parte dei potenziali argomenti di esame.

Queste parte del corso, quindi, si divide in tre sottoparti: dapprima spiegheremo come utilizzare l'editor di testo **vim**, poi lo utilizzeremo per editare alcuni file e fare esperimenti con **grep**, infine (nelle dispense 06) passeremo a **flex** per creare degli analizzatori lessicali costruiti con regole nostre.

Tutto questo sarà poi messo in pratica nel laboratorio di computer.

05 – B Editor vim

Introduzione

Nel mondo **Unix / Linux** sono disponibili un certo numero di editor di testo; **vim** è uno dei migliori per i nostri scopi, è infatti disponibile praticamente in ogni versione di Linux e Unix, richiede poche risorse di sistema e consente di effettuare molte operazioni.

Inizialmente, nel mondo Unix, l'editor di testo per eccellenza era **ed**, che visualizzava solo una riga alla volta del file da editare; dopo **ed** venne implementato **ex**, che eliminava quest'ultima limitazione potendo infatti passare alla modalità a schermo intero... come? Digitando il comando **vi** (che sta per **visual mode**).

La modalità a schermo intero divenne ben presto la regola, così gli sviluppatori decisero di permettere agli utenti di far partire **ex** subito in visual mode: nacque così **vi**. E' possibile editare un testo con **vi** lanciando il comando:

```
sim@debian:~/lab-comp/vim-eserc$ vi prova.txt
```

Nei sistemi odierni non viene quasi mai utilizzato **vi**, ma una sua "versione" più aggiornata e potenziata: sotto slackware, ad esempio, è possibile lanciare **elvis**, sotto Red Hat o Debian o Aix o altri potete lanciare il comando **vim**.

In realtà si tratta di "cloni" di **vi**, che implementano tutti i comandi propri di **vi**, aggiungendo altre "feature" che li rendono ancora più potenti e flessibili.

Noi utilizzeremo **vim** (creato dall'olandese Bram Moolenaar), o **Vi Improved**.

Le principali caratteristiche di **vim** sono:

- compatibilità con **vi** al 99%;
- portabilità verso numerosissimi sistemi operativi e architetture;
- undo/redo ripetuto;
- multi-finestre;
- sintassi evidenziata;
- help online;
- modalità visuale (GUI);
- finestre separabili;
- editing da linea di comando;
- espressioni e script estesi;
- file editing remoto;

Per un sommario delle differenze tra **vi** e **vim**, una volta lanciato **vim** basta eseguire il comando (non sempre disponibile):

```
:help v_diff.txt
```

Noi illustreremo **solo** i comandi più importanti per poter poi lavorare con **grep** e **flex**; invito gli studenti ad approfondire per conto proprio l'utilizzo di **vim**.

Creiamo il nostro ambiente di lavoro

Una volta effettuata la login sul nostro computer, ci troviamo nella nostra **home directory**; ciò viene evidenziato dalla tilde `~` nel prompt.

Il simbolo `█` indica il nostro cursore.

E' possibile verificare il percorso in cui ci troviamo con il comando **pwd**; se vogliamo tornare alla nostra home dir possiamo farlo con il comando **"cd"** :

```
sim@debian:~/ $ pwd
/home/sim
sim@debian:~/ $ cd /var/www/informatica.unipg.it/pippo/
sim@debian:/var/www/informatica.unipg.it/pippo $ cd
sim@debian:~/ $ █
```

Creiamo una directory per il nostro corso, chiamandola ad esempio **lab-comp**; entriamo nella directory e creiamo la cartella **vim-eserc**, nella quale salveremo tutti i file per gli esercizi con **vim**; entriamo poi nella cartella, che ovviamente è ancora vuota, e lanciamo **vim** su un nuovo file, che chiamiamo **testo-01.txt** :

```
sim@debian:~/ $ mkdir lab-comp
sim@debian:~/ $ cd lab-comp/
sim@debian:~/lab-comp $ mkdir vim-eserc
sim@debian:~/lab-comp $ cd vim-eserc/
sim@debian:~/lab-comp/vim-eserc $ ls
sim@debian:~/lab-comp/vim-eserc $ vim testo-01.txt
```

Una volta lanciato questo comando, ci troveremo nell'ambiente di lavoro di **vim**. Il quadrato grigio `█` rappresenta il nostro cursore. La tilde `~` indica semplicemente che non ci sono righe.

```
█
~
~
~
"testo-01.txt" [New File]                0,0-1      All
```

Prima di procedere, dobbiamo spiegare che in **vim** è possibile muoversi in tre "modalità" (**mode**) diverse: **command**, **input**, **last line**.

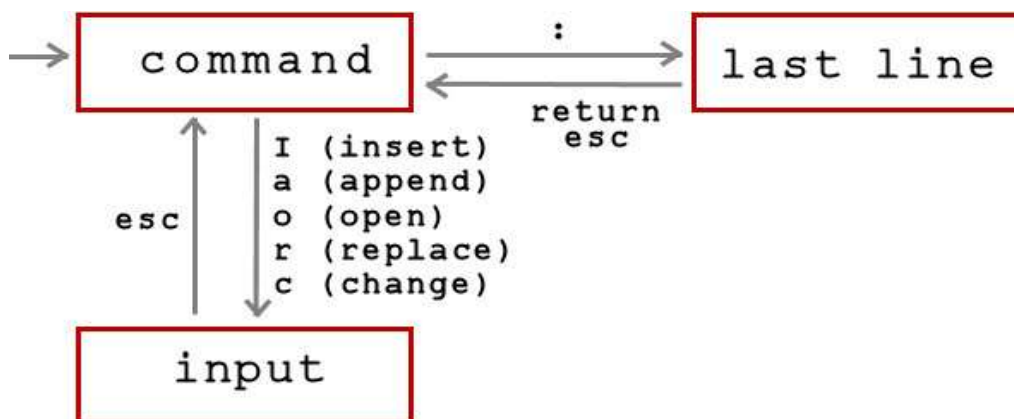


figura 05.01

Il comando "d" (delete) può essere usato in combinazione con numeri e frecce per darci più flessibilità nella cancellazione, ad esempio "d4" seguito dalla **freccia destra** indica di dover cancellare quattro caratteri verso destra a partire da quello in cui si trova il cursore.

Se passiamo all'**input mode** col comando "r" (replace), avremo l'effetto di poter modificare una sola lettera, dopodichè torneremo automaticamente al command mode.

Nell'esempio, da command mode, ci spostiamo col cursore sulla terza riga, quinto carattere:

```
viva
la pace nel mondo.
la pace nel mondo.
la pace nel mondo.
~
                                     3,5      All
```

poi pigiamo "r" (entrando in input mode) e pigiamo "e" (il carattere sostitutivo), tornando automaticamente in command mode:

```
viva
la pace nel mondo.
la pece nel mondo.
la pace nel mondo.
~
                                     3,5      All
```

Per poter cancellare un carattere per volta in command mode pigiare "x":

```
viva
la pace nel mondo.
la pce nel mondo.
la pace nel mondo.
~
                                     3,5      All
```

Notare che l'utilizzo di "x" comporta un inserimento nel buffer del carattere appena cancellato. Basta infatti pigiare "p" quattro volte per ottenere il seguente risultato:

```
viva
la pace nel mondo.
la pceeeeee nel mondo.
la pace nel mondo.
~
                                     3,5      All
```

Un'altra funzione utile è la **ricerca nel testo**, effettuata sempre da command mode.

Usando il comando **"/la"** cercheremo tutte le occorrenze di **"la"** nel testo, procedendo in avanti. Con **"?la"**, invece, cercheremo tali occorrenze all'indietro.

Nel caso di più occorrenze, è possibile scorrerle con i tasti **"n"** (avanti) e **"N"** (indietro).

Da sottolineare che dopo il comando di ricerca è possibile usare **sintassi regex**, ad esempio **"/v[aeiou]v*a"** uguaglia la parola **"viva"** nella riga 1.

Per concludere, presentiamo un ultimo comando per poter salvare in buffer più righe.

Posizionarsi su una riga, pigiare il tasto **"m"** (sta per mark) seguito da una lettera (ad es. **"a"**); questo marcherà quella riga con la lettera stessa.

Ora possiamo scendere di qualche riga e pigiare **"y"** seguito dalla lettera di cui sopra (ad es. **"a"**): l'effetto sarà di aver **"strattonato"** (**yanked**) le righe comprese tra queste due, che ovviamente verranno inserite nel buffer.

Al termine di questo comando **vim** restituisce un messaggio in last line:

```
viva
la pace nel mondo.
la pceeeee nel mondo.
la pace nel mondo.
~
~
3 lines yanked                3,5      All
```

Altri comandi utili sono:

- :13 in last line mode, per andare alla riga 13;
- u in command mode, per effettuare l'undo delle ultime azioni;
- 0 in command mode, per andare all'inizio della riga;
- \$ in command mode, per andare alla fine della riga;
- :\$ in last line mode, per andare all'ultima riga del file;

Il **vim** ha ovviamente molte altre potenzialità, ma per il momento quello che abbiamo appena visto ci è sufficiente per poter lavorare con un file di testo, imparando a conoscere il comando **grep**.

05 – C Comando grep

Grep è un **utility da riga di comando** originariamente offerta nei sistemi operativi Unix.

Il nome proviene da un comando nell'editor testuale **ed** che prende la forma di **g/re/p**, il cui significato è "search **g**lobally for a **r**egular **e**xpression and **p**rint lines where instances are found" (ricerca globalmente una espressione regolare e stampa linee in cui siano trovate istanze).

Esistono altre varianti di grep, come **agrep** per ricerche approssimate, **fgrep** per ricerche con campione fisso (fixed pattern), **egrep** per ricerche che coinvolgono regex più complesse; tuttavia, per il nostro corso, ci limiteremo alla versione standard di **grep**.

Grep può ricevere in input lo standard input, oppure un file.

Se vogliamo cercare le linee nel file testo-01.txt contenenti la parola "mond", basta lanciare il comando seguente per ottenere il risultato voluto:

```
sim@debian:~/lab-comp/vim-eserc$ grep "mond" testo-01.txt
la pace nel mondo.
la pceeeee nel mondo.
la pace nel mondo.
sim@debian:~/lab-comp/vim-eserc$
```

Possiamo anche usare il pipe "|" per ridirigere un qualsiasi output verso il **grep**, in questo modo ad esempio:

```
sim@debian:~/lab-comp$ mkdir grep-eserc
sim@debian:~/lab-comp$ cd grep-eserc
sim@debian:~/lab-comp/grep-eserc$ pwd
/home/sim/lab-comp/grep-eserc
sim@debian:~/lab-comp/grep-eserc$ history | tail | grep "gre"
 194 mkdir grep-eserc
 196 cd grep-eserc
 198 history | tail | grep "gre"
sim@debian:~/lab-comp/grep-eserc$
```

Alcune opzioni utili per lanciare il comando **grep** possono essere:

- c stampa solo il numero di linee che uguagliano il pattern.
- e PATTERN usa PATTERN come campione (per pattern che iniziano per -).
- f FILE ottiene i pattern dal FILE specificato.
- E regex estese.
- n prefissa ogni linea di output con il numero di linea.
- r legge ricorsivamente ogni file nella directory.
- v inverte la corrispondenza (in output le righe non uguagliate).
- w Restringe la ricerca alle parole intere.

- h Quando si ricerca in più di un file, di solito il risultato contiene il nome del file; questa opzione disabilita questo comportamento.
- i Viene ignorato il "case" (ovvero si ricercano stringhe sia uppercase che lowercase).

Per poter fare esempi più complessi prendiamo un file, **testo-02.txt**, contenente una poesia di Cecco Angiolieri, per poi utilizzare quanto appreso finora con le **regex** per poter effettuare ricerche nel file.

```
CECCO ANGIOLIERI - S'i' fosse foco, arderei 'l mondo.
```

```
S'i' fosse foco, arderei 'l mondo;  
s'i' fosse vento, lo tempestarei;  
s'i' fosse acqua, i' l'annegherei;  
s'i' fosse Dio, mandereil' en profondo;
```

```
s'i' fosse papa, allor serei giocondo,  
ché tutti cristiani imbrigarei;  
s'i' fosse 'mperator, ben lo farei:  
a tutti tagliarei lo capo a tondo.
```

```
S'i' fosse morte, andarei a mi' padre,  
s'i' fosse vita, non starei con lui:  
similmente faria da mi' madre.
```

```
S'i' fosse Cecco, com' i' sono e fui,  
torrei le donne giovani e leggiadre:  
le zoppe e vecchie lasserei altrui.
```

Ecco alcuni esempi di output:

```
sim@debian:~/lab-comp/grep-eserc$ grep -n "[lf]ui" testo-02.txt  
14:s'i' fosse vita, non starei con lui:  
17:S'i' fosse Cecco, com' i' sono e fui,
```

Senza l'opzione -E, le parentesi graffe vengono intese come pattern:

```
$ grep "[a-z]{8,9}" testo-02.txt
```

Qui invece vengono intese come intervallo:

```
$ grep -E "[a-z]{8,9}" testo-02.txt  
ché tutti cristiani imbrigarei;  
a tutti tagliarei lo capo a tondo.  
le zoppe e vecchie lasserei altrui.
```

Ecco un ultimo esempio:

```
$ grep -nE ",$" testo-02.txt  
8:s'i' fosse papa, allor serei giocondo,  
13:S'i' fosse morte, andarei a mi' padre,  
17:S'i' fosse Cecco, com' i' sono e fui,
```

Invito gli studenti ad esercitarsi con **grep** e inventare esercizi ed esempi.

06 - flex

- 06-A Introduzione
- 06-B GNU flex
- 06-C Alcuni usi avanzati di flex
- 06-D Approfondimenti su flex

Log delle modifiche

1.1 .

06 – A Introduzione

GNU flex è un software open source, ispirato al noto **Lex**, che genera **analizzatori lessicali** (i cosiddetti “**scanner**”).

Lex è stato originariamente scritto da **Mike Lesk** ed è divenuto l'analizzatore lessicale standard nei sistemi Unix, venendo incluso anche nello standard POSIX. Lex è ovviamente un software proprietario, a pagamento.

Flex, al contrario, è gratuito e a codice aperto, pertanto nel nostro corso useremo quest'ultimo per i nostri esempi.

Flex ha anche il notevole pregio di essere compatibile quasi al 100% con **Lex**.

Analisi lessicale

Come abbiamo già accennato, l'analisi lessicale:

- **riconosce campioni (pattern)**, ad esempio
 - “una sequenza di cifre”
 - “una sequenza di lettere”
 - una parola specifica, ad esempio “while”
 - un simbolo, ad esempio “<”
- **associa campioni a token**, ad esempio BEGIN, END, NUMERO.
- se necessario, **associa attributi a token**, ad esempio per il pattern 309 il valore 309 è associato al token NUMERO.

Cosa fa flex in dettaglio?

Flex legge in input un file che specifica le regole dell'analizzatore lessicale e produce in output un file che costituisce una implementazione in linguaggio C dell'analizzatore lessicale in questione.

Tale file può in seguito essere compilato con un normale compilatore C per ottenere un eseguibile in grado di comportarsi da analizzatore lessicale.

Vediamo una figura per capire meglio:

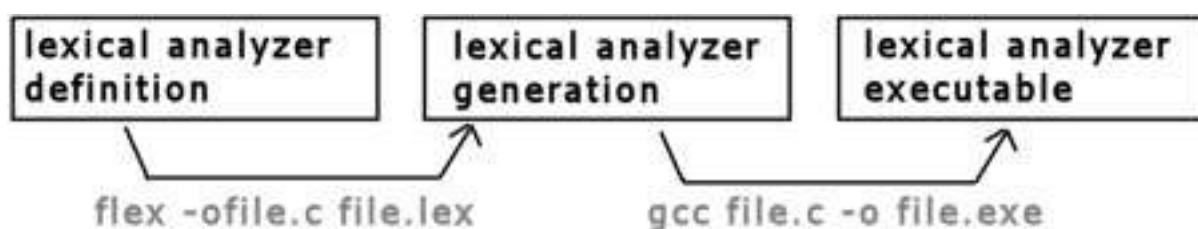


figura 06.01

06 – B GNU flex

Partiamo, come sempre, da alcune definizioni per poi chiarire tutto con un esempio.

Il file che utilizziamo per descrivere il nostro scanner può essere nominato con estensione **.flex**, ad esempio **file.flex** ; tale file è suddiviso in tre sezioni:

– **definition section:**

qui vengono definite macro utilizzando **regex**, e vengono importati file di header scritti in linguaggio C.

– **rules section:**

qui vengono associate regole a comandi in linguaggio C. Quando lex trova un pattern nel suo input che uguaglia una certa regola qui definita, esegue il codice C ad essa associato.

Le regole sono semplicemente delle espressioni regolari, eventualmente contenenti le macro definite nella sezione precedente.

– **C code section:**

Qui viene inserito il codice C dell'utente che viene copiato direttamente nel file generato da flex. Generalmente questa sezione contiene codice richiamato da alcune regole nella rules section.

In programmi di grandi dimensioni conviene inserire questo codice in un file separato e linkarlo in fase di compilazione.

Nelle prime due sezioni è possibile aggiungere direttamente codice C, inserendolo tra `"%{"` e `"%}"` oppure **indentandolo** (con un TAB, ad esempio).

E' ovviamente possibile inserire commenti con: `/* commento */` .

Attenzione: il modo di commentare (poco chiaro nel **man**) può variare con le varie versioni di flex; accertatevi del modo corretto di commentare prima di procedere.

In generale, nella **rules section** di flex è bene non posizionare i commenti ad inizio riga, mentre nelle sezioni in linguaggio C non dovrebbero esserci problemi.

Valgono le seguenti regole:

- 1) **longest match:** uguaglianze più lunghe hanno la precedenza; se ho ad esempio due pattern, `viva` e `vivaio`, quando trovo nell'input la parola `vivaio`, uguaglio il secondo pattern, non il primo.
- 2) **First rule:** a parità di precedenza, le regole inserite per prime vengono preferite.
- 3) **Standard action:** quando nessuna regola viene uguagliata, l'input viene passato direttamente in output.

Quando viene lanciato flex, è possibile usare l'opzione `"-l"` per forzare la piena compatibilità con lex, oppure `"-s"` per sopprimere la regola 3.

E' giunto il momento di passare ad un esempio.

Esempio 1:

```
/* file.flex */
/*//////////////////////////////////////
// definition section                                     */
%{
#define NPARI 1001
#define NDISPARI 1002
#define SALUTO 1010
%}
PARI [1-9]+[02468]
DISPARI [1-9]+[13579]
%option main
/*//////////////////////////////////////
// rules section                                       */

%%
[ ] ;
{PARI}      { printf("Trovato pari.\n") ; return(NPARI) ;}
{DISPARI}   { printf("Trovato dispari.\n") ; return(NDISPARI) ;}
"CIAO"|"AOO" { printf("Trovato saluto.\n") ; return(SALUTO) ;}
.           { printf("Riconoscimento fallito.\n") ; exit(1) ;}
%%
/*//////////////////////////////////////
// C code section                                     */
```

Nella definition section troviamo **%option main**, è il modo di aggiungere delle opzioni al comportamento di flex. **"main"** indica che flex deve provvedere una funzione **main()** in linguaggio C per lo scanner, che quindi non deve essere fornita da noi.

Nella rules section troviamo regole per lo spazio [] , per i numeri pari o dispari, oppure per le due stringhe CIAO e AOO; infine, ogni altro carattere "." produce la stampa a schermo dell'avvertimento corrispondente.

Nella rules section le **istruzioni C** devono essere racchiuse tra **parentesi graffe**, a meno che non si tratti di una azione unica, nel cui caso le graffe non servono.

Proviamo ad eseguire i vari passaggi per la creazione dell'eseguibile:

```

sim@debian:~/lab-comp/flex$ flex -ofile.c file.flex
sim@debian:~/lab-comp/flex$ gcc file.c -o scanner1.exe
sim@debian:~/lab-comp/flex$ ./scanner1.exe
56
Trovato numero pari.
sim@debian:~/lab-comp/flex$ ./scanner1.exe
rtt
Riconoscimento fallito.
sim@debian:~/lab-comp/flex$ ./scanner1.exe
A00
Trovato saluto.
sim@debian:~/lab-comp/flex$ ./scanner1.exe
Aoo
Riconoscimento fallito.
sim@debian:~/lab-comp/flex$

```

Nel file **.flex** è possibile "giocare" con tutta una serie di funzioni proprie di flex, qui vediamo subito un esempio:

```

/* file.flex */
/*//////////////////////////////////////
// definition section                                     */
%{
%}
%option main
/*//////////////////////////////////////
// rules section                                         */

%%
[ ] ;
"ciao" ECHO; ymore();
. { printf("Riconoscimento fallito.\n") ; exit(1) ;}
%%
/*//////////////////////////////////////
// C code section                                       */

```

Se compiliamo questo file come prima, una volta lanciato lo scanner otteniamo di vederci stampare a schermo la parola "ciao" due volte:

```

sim@debian:~/lab-comp/flex$ ./scanner1.exe
56
Riconoscimento fallito.
sim@debian:~/lab-comp/flex$ ./scanner1.exe
ciao
ciaociao
sim@debian:~/lab-comp/flex$

```

06 – C Alcuni usi avanzati di flex

Esempio 1

Questo esempio stampa a schermo il nome utente:

```
%option main
%%
[ ] ;
username { printf("%s", getlogin() );}
. { printf("Riconoscimento fallito.\n");}
%%
```

In fase di compilazione con **gcc** questo file flex dovrebbe restituire degli **warnings**, il che può capitare spesso.

Più in generale, nel caso di veri e propri **errori**, invece, se non è sufficiente sistemare il file flex, occorre mettere le mani sul sorgente C generato da flex stesso; quest'ultima operazione è spesso piuttosto **complicata**.

Vediamone l'esecuzione ([^C] indica la combinazione di tasti **Ctrl-C**):

```
sim@debian:~/lab-comp/flex$ flex -o2.c 2-username.flex
sim@debian:~/lab-comp/flex$ gcc 2.c -o 2.exe
sim@debian:~/lab-comp/flex$ ./2.exe
ciao
Riconoscimento fallito.
username
sim
[ ^C ]
sim@debian:~/lab-comp/flex$
```

Esempio 2

Vediamo un altro esempio, nel quale la funzione **main()** viene scritta da noi.

```
int n_linee = 0, n_caratteri = 0;
%option noyywrap
%%
\n      n_linee++;n_caratteri++;
.       n_caratteri++;
"esci"  return (1);
%%
main () {
    yylex ();
    printf ("linee:%d  caratteri:%d\n",n_linee, n_caratteri);
}
```

La opzione **noyywrap** indica che la corrispondente funzione **yywrap** non viene fornita dall'utente, ma deve essere creata da flex stesso.

Vediamone l'esecuzione:

```
sim@debian:~/lab-comp/flex$ flex -o3.c 3-contaparole.flex
sim@debian:~/lab-comp/flex$ gcc 3.c -o 3.exe
sim@debian:~/lab-comp/flex$ ./3.exe
ciao a tutti
bella giornata, vero?
Esci
linee: 2  caratteri: 35
sim@debian:~/lab-comp/flex$
```

Esempio 3

Ora passiamo ad un altro esempio in cui, invece che leggere da standard input, leggiamo da file:

```
%option noyywrap
%%

[aA]+      printf("e");
[eE]+      printf("i");
[iI]+      printf("o");
[oO]+      printf("u");
[uU]+      printf("a");
.          ;
%%

main (argc, argv)
    int argc;
    char **argv;
    {
    ++argv, --argc; /* così evitiamo di considerare il nome
                    del programma lanciato */
    if (argc > 0) {yyin = fopen (argv[0], "r");}
    else {printf ("Leggo da tastiera!\n"); yyin = stdin;}
    yylex ();
    }
```

Immaginiamo che il nostro file **4-esempio.txt** sia questo:

```
Una donna alta non e' mai banale...
sarà per lo sguardo necessariamente superiore.
[Flavio Giurato, 'Il tuffatore']

Le religioni sono come le lucciole:
per splendere hanno bisogno delle tenebre.
[A. Schopenhauer]
```

Vediamo ora la compilazione e l'esecuzione sia in presenza di un file passato su riga di comando, sia nel caso opposto:

Ecco la relativa esecuzione:

```
sim@debian:~/lab-comp/flex$ flex -o5.c 5-maleducato.flex
sim@debian:~/lab-comp/flex$ gcc 5.c -o 5.exe
sim@debian:~/lab-comp/flex$ ./5.exe
ciao!
ciao!
antonioooooo
antonioooooo
giampieroo
Che te rode?
maleducato
maleducato sarai tu!
banano!
banano sarai tu!!
ANDIAMO?
ANDIAMO? hmmm...?

sim@debian:~/lab-comp/flex$
```

Da notare che il punto esclamativo dopo la parola "banano" viene messo in fondo alla frase "banano sarai tu!".

Esempio 6

In flex possiamo leggere la variabile **yylen** che indica la lunghezza, in caratteri, dell'ultimo "match" (corrispondenza, uguaglianza). Possiamo anche accedere a **yytext** come vettore, e usare **yylen** per "giocare" un po' con l'ultima stringa uguagliata. Vediamo un esempio, e la relativa esecuzione:

```
%option main
%%
[a-zA-Z ]+ {
    int i;
    for (i = 1; i <= yylen; i++)
        printf ("%c", yytext[yylen-i]);
}
%%
```

```
sim@debian:~/lab-comp/flex$ flex -o6.c 6-yylen.flex
sim@debian:~/lab-comp/flex$ gcc 6.c -o 6.exe
sim@debian:~/lab-comp/flex$ ./6.exe
simone
enomis
brunozzi
izzonurb
maleducato
otacudelam

sim@debian:~/lab-comp/flex$
```

Esempio 7

Qualcosa di più consistente:

```
/* scanner per un linguaggio simil-Pascal */
%{
#include <math.h>
%}
DIGIT      [0-9]
ID         [a-z][a-z0-9]*
%option noyywrap
%%

{DIGIT}+   {
            printf( "Numero intero: %s (%d)\n", yytext,
                    atoi( yytext ) );
            }

{DIGIT}+"."{DIGIT}*   {
            printf( "Numero reale: %s (%g)\n", yytext,
                    atof( yytext ) );
            }

if|then|begin|end|procedure|function      {
            printf( "Parola chiave: %s\n", yytext );
            }

{ID}          printf( "Identificatore: %s\n", yytext );

"+"|"-"|"*"|"/"   printf( "Operatore: %s\n", yytext );

{"^[^]\n]*"}      /* elimina i commenti */

[ \t\n]+        /* elimina gli spazi inutili a fine riga */

.              printf( "Carattere sconosciuto: %s\n", yytext );

%%
main( argc, argv )
int argc;
char **argv;
{
++argv, --argc;
if ( argc > 0 )
    yyin = fopen( argv[0], "r" );
else
    yyin = stdin;

yylex();
}
```

Invito gli studenti a provare questo esempio in laboratorio.

06 – D Approfondimenti su flex

Il file di output generato da flex a partire dal sorgente da noi scritto contiene una routine chiamata **yylex()**, un certo numero di tabelle usate per uguagliare token, e un certo numero di routine ausiliarie.

Di default, **yylex()** viene dichiarato come segue:

```
int yylex()
{
    ... varie definizioni ed azioni ...
}
```

Quando **yylex()** viene invocata, esegue la scansione di token dal file globale di input chiamato **yyin**, che legge di default dallo standard input.

La funzione **yylex()** continua ad eseguire la scansione finchè non raggiunge un **EOF** (End Of File) oppure una delle azioni collegate ad un token letto eseguono un **return**.

Quando **yylex** raggiunge un **EOF**, eventuali successive chiamate ad essa sono indefinite, a meno che **yyin** non venga puntato ad un nuovo file di input, oppure venga invocata la funzione **yyrestart()**; tale funzione prende come argomento un puntatore a file, **'FILE *'**, e inizializza **yyin** per effettuare la scansione da quel file.

La maniera in cui lo scanner legge caratteri dall'input viene definita dalla macro **YY_INPUT**, che di default legge dal puntatore di file globale **yyin**.

Quando lo scanner riceve un **EOF** da **YY_INPUT**, esegue la funzione **yywrap()**; nel caso in cui essa restituisca **false** (zero), si assume che tale funzione abbia proseguito e fatto puntare **yyin** ad un altro file di input; in tal caso la scansione continua.

Se invece **yywrap()** restituisce **true** (non-zero), allora lo scanner termina la sua esecuzione.

Nel caso in cui non venga fornita una propria versione di **yywrap()**, è necessario aggiungere l'opzione seguente:

```
%option noyywrap
```

Vediamo ora di spiegare l'utilizzo delle start conditions.

Condizioni iniziali (start conditions)

In flex è possibile utilizzare un meccanismo per **attivare regole secondo certe condizioni**; tale meccanismo usufruisce delle start conditions.

L'utilità di questo meccanismo è data dal poter utilizzare **regole differenti a seconda della porzione di input analizzata**; un esempio banale è dato da un codice sorgente in un linguaggio di programmazione, che possiamo considerare suddiviso in **codice** e **commenti**.

Ogni **regola** il cui **pattern** venga preceduto da **<sc>** sarà attiva solo quando lo scanner è nella start condition denominata **sc**.

Le start conditions vengono dichiarate nella **definition section**, utilizzando linee senza indentazione che iniziano per **%s** oppure per **%x**, seguite da una lista di nomi. Il **%s** indica delle condizioni **inclusive**, il **%x** delle condizioni **esclusive** (nel senso di escludere).

Una start condition è attivata usando l'azione **BEGIN**; finchè la successiva azione **BEGIN** non viene eseguita, le regole con la start condition scelta sono **attive**, e quelle con altre start condition sono **inattive**.

Se la start condition è **inclusiva**, allora le regole **prive** di start condition saranno attive anch'esse; se la start condition è **esclusiva**, allora solo le regole qualificate con una start condition saranno attive.

Chiariamo subito con un esempio per capire la differenza tra **%s** e **%x**, mostrando due codici **equivalenti nel comportamento**:

```
%s condiz1
%%

<condiz1>ciao          printf("pippo");
bye                    printf("pluto");
```

```
%x condiz1
%%

<condiz1>ciao          printf("pippo");
<INITIAL,condiz1>bye  printf("pluto");
```

La parte **<INITIAL,condiz1>** è necessaria perchè, usando semplicemente **<condiz1>** per qualificare la stringa **"bye"**, le relative regole sarebbero attive solo nella condizione esempio, ma non in **INITIAL** (che indica lo stato all'inizio della scansione), mentre nel primo esempio le regole per **"bye"** sono attive sia all'inizio, sia nella condizione **condiz-1**.

Tale differenza è data dal fatto che, nel caso di **%s**, le regole prive di start conditions sono rese **attive**.

Manca un tassello per comprendere appieno l'utilizzo delle start condition, ovvero come utilizzare **BEGIN** per passare alle varie condizioni.

Nel seguente esempio lo scanner entrerà nella condizione **condiz1** a patto che la variabile **valore** sia diversa da zero:

```
int valore;

%x condiz1
%%
    if ( valore )
        BEGIN(condiz1);

<condiz1>eureka    printf("Ho trovato!");

...altre regole...
```

Detto ciò, mostriamo un esempio più complesso, in cui lo scanner può interpretare in **due maniere differenti** una stringa come "**123.456**", a seconda che sia preceduta o meno dalla stringa "**expect-floats**".

Invito gli studenti ad esercitarsi con tale esempio e se possibile modificarlo a piacere, inserendo ulteriori funzionalità.

```
{ #include <math.h> %}
%s expect
%%
expect-floats          BEGIN(expect);

<expect>[0-9]+ "." [0-9]+    {
                             printf( "trovato reale = %f\n",
                             atof( yytext ) ); }

<expect>\n                {
                             /* essendo fine riga, abbiamo bisogno di un
                             altro "expect-number" prima di riconoscere
                             ulteriori numeri */
                             BEGIN(INITIAL);
                             }

[0-9]+                    {
                             printf( "trovato intero = %d\n",
                             atoi( yytext ) );
                             }

"."                        printf( "trovato un punto\n" );
```

07 - Parsing

- 07-A Analisi sintattica / parsing
- 07-B bison
- 07-C flex e bison insieme
- 07-D VCG (Visualization of Compiler Graphs)
- 07-E conflitti
- 07-F esempi

Log delle modifiche

1.1 .

07 – A Analisi sintattica / parsing

Dopo aver affrontato gli argomenti riguardanti l'analisi lessicale, e aver sperimentato l'uso del software **flex**, passiamo alla fase successiva: l'**analisi sintattica**, comunemente detta **parsing**, che consiste, detto in parole povere, nel **determinare** se una stringa di **token** può essere generata da una **grammatica**.

Quando iniziamo l'analisi sintattica del nostro codice abbiamo già a disposizione i risultati dell'analisi lessicale (scanning), ovvero una suddivisione dei vari simboli incontrati, organizzata in una **symbol table**, come può essere questa:

```
pos := val + rate * 60 ;
```

pos	:=	val	+	rate	*	60	;
ID.0	ASS_OP	ID.1	AR_OP	ID.2	AR_OP	NUM	TERM

ID.n: identificativo di una variabile
ASS_OP: operazione di assegnamento
AR_OP: operazione aritmetica
NUM: numero
TERM: simbolo terminale della istruzione

La analisi sintattica consiste nel "**riconoscere**" delle sentenze (ad es. associare un valore ad una variabile, un ciclo if, una funzione), e associare tali sentenze ad un **parse tree**.

In parole più povere, l'analisi lessicale associa ai **token** il tipo ed il valore, mentre quella sintattica cerca di riconoscere la struttura di un determinato linguaggio, ad es. un linguaggio di programmazione.

Alcune considerazioni e definizioni

I linguaggi di programmazione hanno regole a cui sottende la struttura sintattica di programmi ben formati. Nell'linguaggio **C**, ad esempio, un programma è fatto da blocchi, un blocco è fatto da statement, uno statement è fatto da espressioni, le espressioni sono composte da token, e così via.

La sintassi dei costrutti dei linguaggi di programmazione può essere descritta da **context-free grammars**, o notazione **BNF** (Backus-Naur Form).

Le grammatiche offrono vantaggi significanti:

- una grammatica fornisce una precisa **specificata sintattica**;
- da alcune classi di grammatiche possiamo costruire un parser che determina se un programma sorgente è **ben formato**. Come beneficio addizionale, il processo di parsing rivela **ambiguità sintattiche** che potrebbero altrimenti non essere riconosciute;
- Se un linguaggio **evolve**, i suoi nuovi costrutti possono essere aggiunti con facilità.

I parser basati sugli algoritmi di **Cocke-Younger-Kasami**, o di **Earley**, possono "parsare" qualsiasi tipo di grammatica, ma sono talmente **inefficienti** in termini prestazionali da non essere interessanti in applicazioni pratiche; verranno perciò ignorati nel nostro corso.

Escludendo tale categoria possiamo perciò **suddividere** i parser in due grandi classi: **top-down** e **bottom-up**; tali definizioni si riferiscono all'ordine in cui i nodi del **parse tree** vengono costruiti.

Nel **top-down** la costruzione parte dalla radice dell'albero e procede verso le foglie; nel **bottom-up** la costruzione parte dalle foglie per arrivare alla radice.

I parser top-down o bottom-up più efficienti lavorano soltanto con alcune **classi di grammatiche**, come le **LL** e le **LR**, ma questa limitazione non impedisce di gestire la stragrande maggioranza dei costrutti, e in generale è comunque possibile applicarli ai moderni linguaggi di programmazione.

Un **LL-parser** effettua il parsing dell'input da sinistra (**Left**) a destra, e costruisce una derivazione **Leftmost** (si espande sempre il simbolo nonterminale più a sinistra) della sentenza.

Un **LR-parser** effettua il parsing dell'input da sinistra (**Left**) a destra, e costruisce una derivazione **Rightmost** (si espande sempre il simbolo nonterminale più a destra) della sentenza.

Vengono spesso usati i termini **LR(1)-parser**, oppure **LL(1)-parser**, dove il numero indica il numero di simboli di input "**look ahead**" (ovvero, esaminati in avanti) non consumati che vengono usati per prendere decisioni nella costruzione dell'albero di parsing.

Quando tale numero è **1** viene spesso omissso.

Questo schema sintetizza le interazioni tra **analisi lessicale** e **parsing**:

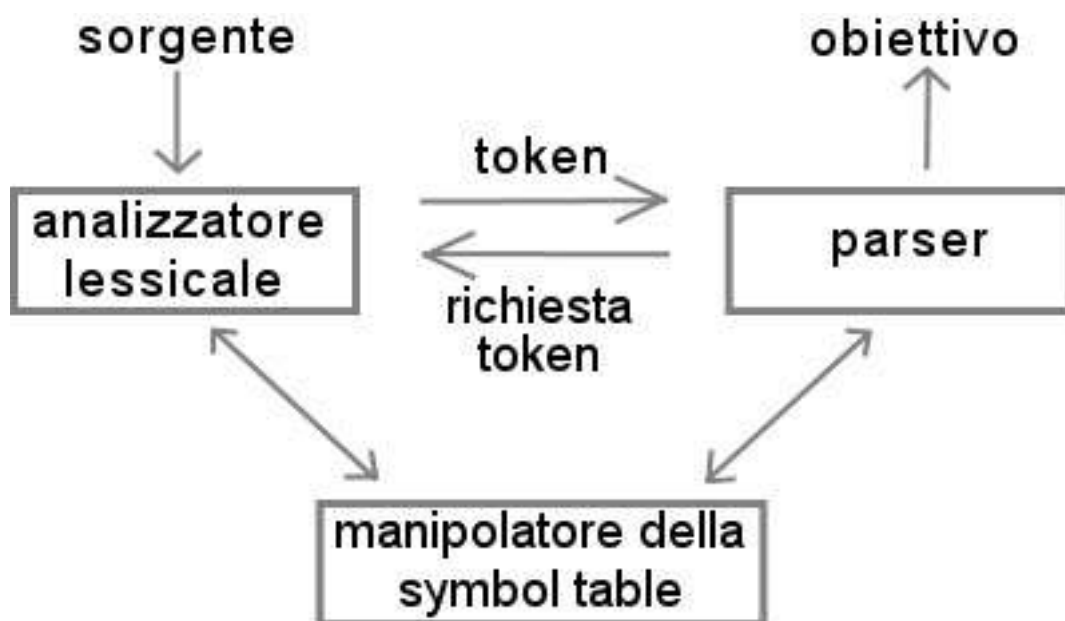


figura 07.01

Vediamo ora un esempio di grammatica e relativo parsing, sia LL che LR.

Grammatica:

S --> if E then S else S fi | while E do S od | skip
E --> num | id

esempio 1:

```
1   2   3
if num then
4       5   6   7       8
while id do skip od
9       10
else skip
11
fi
```

Vediamo il parsing con algoritmo **top-down**, che equivale in questo caso ad un parsing **LL(1)**:

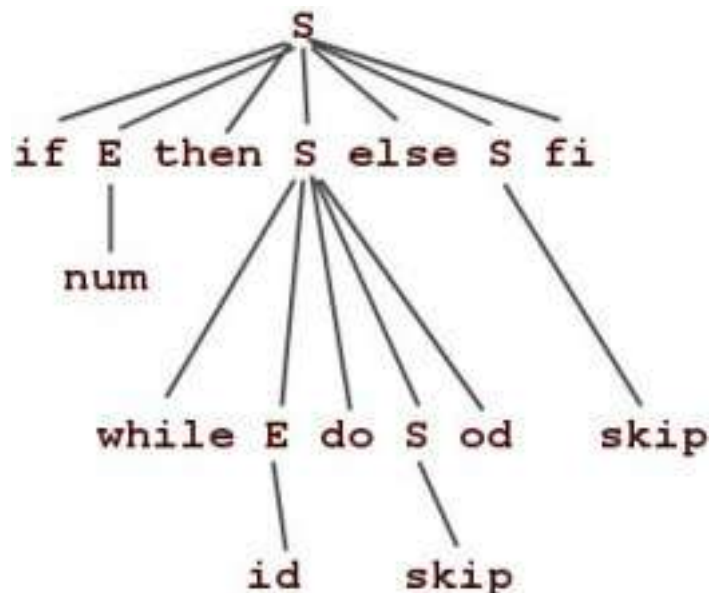


figura 07.02

Qui di seguito i vari passi, evidenziando in grassetto sottolineato l'applicazione delle regole grammaticali:

LL(1)-parsing, top-down:

```
S --> if E then S else S fi
--> if num then S else S fi
--> if num then while E do S od else S fi
--> if num then while id do S od else S fi
--> if num then while id do skip od else S fi
--> if num then while id do skip od else skip fi
```

Vediamo invece il parsing con algoritmo **bottom-up**, che in questo caso equivale ad un **LR**:

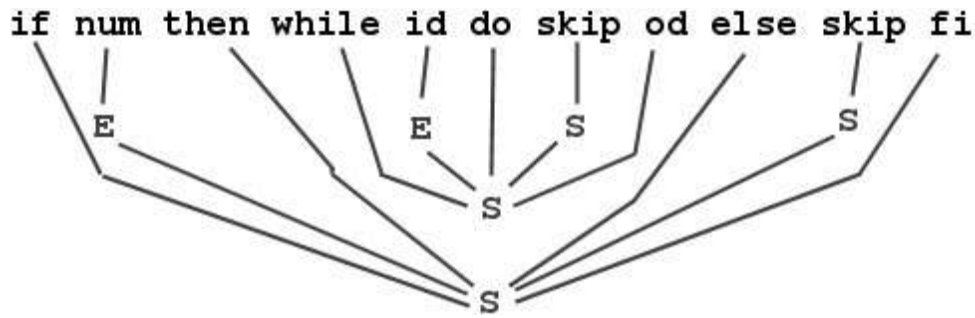


figura 07.03

Qui di seguito i vari passi, evidenziando in grassetto sottolineato ciò che cambia:

LR-parsing, con bottom-up:

```

if num then while id do skip od else skip fi <--
if E then while id do skip else skip fi <--
if E then while E do skip else skip fi <--
if E then while E do S else skip fi <--
if E then S else skip fi <--
if E then S else S fi <--
S

```

Le **grammatiche context-free** sono riconoscibili da **automi a pila**, nel caso specifico di algoritmo **top-down**.

L'algoritmo **top-down** utilizza una tabella che possiamo definire come **tabella di parsing**; eccone un esempio per la nostra grammatica di prima, in cui il simbolo **\$** indica il fondo della pila:

Token	S	E
id	-	E --> id
num	-	E --> num
if	S --> <u>if</u> E <u>then</u> S <u>else</u> S	-
then	-	-
else	-	-
fi	-	-
while	S --> <u>while</u> E <u>do</u> S <u>od</u>	-
do	-	-
od	-	-
skip	S --> <u>skip</u>	-
\$	-	-

figura 07.04

Mostriamo i movimenti della pila durante la fase di parsing in un **top-down**, evidenziando in **giallo** i **token** che vengono tirati fuori dalla pila (azione **pop**) e inseriti nel parse tree, e in **grigio** i simboli **nonterminali** che vengono espansi:

1.	\$	S							
2.	\$	fi	S	else	S	then	E	if	
3.	\$	fi	S	else	S	then	E		
4.	\$	fi	S	else	S	then	num		
5.	\$	fi	S	else	S	then			
6.	\$	fi	S	else	S				
7.	\$	fi	S	else	od	S	do	E	while
8.	\$	fi	S	else	od	S	do	E	
9.	\$	fi	S	else	od	S	do	num	
10.	\$	fi	S	else	od	S	do		
11.	\$	fi	S	else	od	S			
12.	\$	fi	S	else	od	skip			
13.	\$	fi	S	else	od				
14.	\$	fi	S	else					
15.	\$	fi	S						
16.	\$	fi	skip						
17.	\$	fi							
18.	\$								

figura 07.05

Proviamo ora a vedere il comportamento per un algoritmo **bottom-up**, facendo attenzione alle azioni di **shift** ("mettere il token nello stack") e alle azioni di **reduce** (sostituire un token con un simbolo, basandosi sulla grammatica).

(1) token	azione	Pila -->										
if	shift	if										
(2) token	azione	Pila -->										
num	shift	if	num									
(3) token	azione	Pila -->										
then	reduce	if	E									
(4) token	azione	Pila -->										
-	shift	if	E	then								
(5) token	azione	Pila -->										
while	shift	if	E	then	while							
(6) token	azione	Pila -->										
id	shift	if	E	then	while	id						
(7) token	azione	Pila -->										
do	reduce	if	E	then	while	E						
(8) token	azione	Pila -->										
-	shift	if	E	then	while	E	do					
(9) token	azione	Pila -->										
skip	shift	if	E	then	while	E	do	skip				
(10) token	azione	Pila -->										
od	reduce	if	E	then	while	E	do	S				
(11) token	azione	Pila -->										
-	shift	if	E	then	while	E	do	S	od			
(12) token	azione	Pila -->										
else	reduce	if	E	then	S							
(13) token	azione	Pila -->										
-	shift	if	E	then	S	else						
(14) token	azione	Pila -->										
skip	shift	if	E	then	S	else	skip					
(15) token	azione	Pila -->										
fi	reduce	if	E	then	S	else	S					
(16) token	azione	Pila -->										
-	shift	if	E	then	S	else	S	fi				
(17) token	azione	Pila -->										
-	reduce	S										

figura 07.06

07 – B bison

Bison è un software reso disponibile dal **GNU** project della **Free Software Foundation (FSF)**; consiste in un generatore di parser che di fatto converte una descrizione per una grammatica in un programma C in grado di effettuare il parsing di tale grammatica.

Nel caso particolare di Bison, si tratta di una grammatica **LALR**, ovvero Look-Ahead LR.

Bison è compatibile con **Yacc (Yet Another Compiler-Compiler)**, e generalmente è in grado di accettare senza ulteriori modifiche le grammatiche scritte per Yacc.

Il **bison source** (sorgente bison) è un file che definisce delle regole grammaticali; una volta "compilato" dal software **bison**, produce un file costituito da istruzioni C, che possiamo chiamare **bison parser**.

Nel **bison parser** è definita una funzione chiamata **yyparse** che implementa la grammatica definita nel bison source.

Tale funzione non è sufficiente, da sola, ad eseguire il task assegnato: dobbiamo preoccuparci noi di fornire un **analizzatore lessicale** (ad esempio con flex), una funzione che ci comunica eventuali **errori** in fase di parsing, e ovviamente la funzione **main**, la quale deve a sua volta invocare **yyparse**.

E' convenzione nominare i bison source con l'estensione **.y** (ad es. **prova.y**).

Vediamo l'aspetto di un tipico **bison source**:

```
%{  
C code  
%}
```

Bison declarations

```
%%  
%{  
C code  
%}
```

Grammar rules

```
%%
```

Additional C code

Vediamo ora un esempio concreto di un piccolo **bison source**, una calcolatrice con notazione polacca inversa:

```

* Calcolatrice con notazione polacca inversa */

%{
#define YYSTYPE double
#include <math.h>
#include <ctype.h>
#include <stdio.h>
%}
%token NUM
%%
input:      /* vuoto */
           | input line;
line:      '\n'
           | exp '\n'      {printf("\t%.10g\n", $1);};
exp:      NUM              {$$ = $1; }
           | exp exp '+'   {$$ = $1 + $2; }
           | exp exp '-'   {$$ = $1 - $2; }
           | exp exp '*'   {$$ = $1 * $2; }
           | exp exp '/'   {$$ = $1 / $2; };
%%
/* analizzatore lessicale */

int yylex (void)
{
    int c;

    /* elimina gli spazi vuoti*/
    while ( (c = getchar()) == ' ' || c == '\t' ) ;

    /* processa i numeri */
    if (c == '.' || isdigit (c) )
    {
        ungetc (c,stdin);
        scanf ("%lf",&yylval);
        return NUM;
    }

    /* restituisce la fine dell'input */
    if (c == EOF) return 0;

    /* restituisce un singolo carattere */
    return c;
}

int main (void)
{
    return yyparse();
}

int yyerror (const char *s)
{
    printf ("%s\n", s);
}

```

Vediamo come **compilare** un bison source per poi utilizzarlo. Una volta lanciato, bison crea un file con estensione **.tab.c**, il quale va poi compilato per ottenere un eseguibile.

```
sim@debian:~/ $ ls
rpcalc.y

sim@debian:~/ $ bison rpcalc.y
sim@debian:~/ $ ls
rpcalc.y  rpcalc.tab.c

sim@debian:~/ $ gcc rpcalc.tab.c -o rpcalc.exe
sim@debian:~/ $ ./rpcalc.exe
5 22 +
      27
13 4 -
      9
4 6 *
      24
15 3 /
      5
```

Invito gli studenti ad esercitarsi con questo esempio, e a modificarlo rendendo la calcolatrice più completa.

Precedenza degli operatori

Quando viene dichiarato un token si può utilizzare **%token**, oppure una parola chiave diversa per specificare l'associatività degli operatori, in particolare **%left**, **%right**, **%nonassoc**.

Tali parole chiave vengono chiamate "**precedence declarations**", dichiarazioni di precedenza.

La sintassi è identica a quella relativa a **%token**, ovvero:

%left symbols...

%left <type> symbols...

L'associatività di un operatore **op1** determina come vengono considerati usi ripetuti di tale operatore: l'espressione

x op1 y op1 z

viene analizzata (parsed) raggruppando dapprima **x con y** se l'operatore è definito con **%left**, oppure **y con z** se l'operatore è definito con **%right**.

%nonassoc specifica invece una non associatività, e nel nostro caso porterebbe ad un errore.

La associatività sinistra o destra di un operatore determina come esso si annidi con altri operatori.

Tutti i token dichiarati in una **singola dichiarazione di precedenza** hanno identica precedenza, e si annidano insieme in base alla loro associatività.

Quando invece due token, dichiarati in **diverse dichiarazioni di precedenza**, vengono associati, quello dichiarato per **ultimo** ha precedenza maggiore e quindi viene raggruppato per primo.

07 – C flex e bison insieme

Vediamo ora un esempio di utilizzo di bison in combinazione con flex.

Esempio 1: semplice calcolatrice

```
/* file: simple_calc.y */
%{
#define YYSTYPE int
%}

%token NUM PIU MENO PER DIVISO ACCAPO

%%
input:      /* empty */
          | input line ;

line:      ACCAPO
          | exp ACCAPO ;

exp:       NUM
          | exp exp PIU
          | exp exp MENO
          | exp exp DIVISO
          | exp exp PER ;

%%

int yyerror(char *s)
{
    extern char yytext[];
    printf("Al token \"%s\": %s\n", yytext, s);
}

int main() { yydebug = 1; yyparse(); }
```

```
/* file: simple_calc.flex */
%{
#include"simple_calc.tab.h"
%}
%option noyywrap
%%
[ \t\r\f]          ;
[\n]               { return(ACCAPO); }
"+"               { return(PIU); }
"-"               { return(MENO); }
"*"               { return(PER); }
"/"               { return(DIVISO); }
[1-9][0-9]*       {
                    int i;
                    yylval = sscanf(yytext,"%d",&i);
                    return(NUM);
                }
.                  { printf("Non riconosciuto.\n"); exit(1); }
%%
```

Da questi due file è possibile generare un eseguibile che risponde alle regole da noi specificate, che vediamo alla prossima pagina.

L'opzione **-l** usata con **flex** serve per attivare la massima compatibilità possibile con l'originale lex dello Unix di AT&T.

Vediamo le opzioni usate con **bison**:

-t : il parser viene eseguito in trace mode (modalità traccia)

-d : produce un file di header

-v : produce un file .output che descrive l'automa.

```
sim@debian:~/ $ ls
simple_calc.y          simple_calc.flex

sim@debian:~/ $ bison -t -d -v simple_calc.y
sim@debian:~/ $ ls
simple_calc.y          simple_calc.flex
simple_calc.tab.h      simple_calc.output
simple_calc.tab.c

sim@debian:~/ $ flex -osimple_calc.yy.c -l simple_calc.flex
sim@debian:~/ $ ls
simple_calc.y          simple_calc.flex
simple_calc.tab.h      simple_calc.output
simple_calc.tab.c      simple_calc.yy.c

sim@debian:~/ $ gcc simple_calc.tab.c simple_calc.yy.c -o simple.exe
sim@debian:~/ $ cat simple_calc.esempio
4 9 +

sim@debian:~/ $ ./simple.exe < simple_calc.esempio

Starting parse
Entering state 0
Reducing via rule 1 (line 8), -> input
state stack now 0
Entering state 1
Reading a token: Next token is 257 (NUM)
Shifting token 257 (NUM), Entering state 2
Reducing via rule 5 (line 14), NUM -> exp
state stack now 0 1
Entering state 5
Reading a token: Next token is 257 (NUM)
Shifting token 257 (NUM), Entering state 2
Reducing via rule 5 (line 14), NUM -> exp
state stack now 0 1 5
Entering state 7
Reading a token: Next token is 258 (PIU)
Shifting token 258 (PIU), Entering state 8
Reducing via rule 6 (line 15), exp exp PIU -> exp
state stack now 0 1
Entering state 5
Reading a token: Next token is 262 (ACCAPO)
Shifting token 262 (ACCAPO), Entering state 6
Reducing via rule 4 (line 12), exp ACCAPO -> line
state stack now 0 1
Entering state 4
Reducing via rule 2 (line 9), input line -> input
state stack now 0
Entering state 1
Reading a token: Now at end of input.
Shifting token 0 ($), Entering state 12
Now at end of input.
sim@debian:~/ $
```

Gli **stati** a cui si riferisce il nostro output (qui in debug mode, come indicato nel file .y) derivano dalla notazione usata nel file **simple_calc.output**, che qui vediamo.

Per generare tale file è sufficiente lanciare **bison** aggiungendo l'opzione **-v**.

```
/* file: simple_calc.flex */
Grammar
  Number, Line, Rule
  1   8 input -> /* empty */
  2   9 input -> input line
  3  11 line -> ACCAPO
  4  12 line -> exp ACCAPO
  5  14 exp -> NUM
  6  15 exp -> exp exp PIU
  7  16 exp -> exp exp MENO
  8  17 exp -> exp exp DIVISO
  9  18 exp -> exp exp PER

Terminals, with rules where they appear

$ (-1)
error (256)
NUM (257) 5
PIU (258) 6
MENO (259) 7
PER (260) 9
DIVISO (261) 8
ACCAPO (262) 3 4

Nonterminals, with rules where they appear
input (9)
  on left: 1 2, on right: 2
line (10)
  on left: 3 4, on right: 2
exp (11)
  on left: 5 6 7 8 9, on right: 4 6 7 8 9

state 0
  $default      reduce using rule 1 (input)
  input         go to state 1

state 1
  input -> input . line   (rule 2)
  $         go to state 12
  NUM      shift, and go to state 2
  ACCAPO   shift, and go to state 3
  line     go to state 4
  exp      go to state 5

state 2
  exp -> NUM .   (rule 5)
  $default      reduce using rule 5 (exp)

state 3
  line -> ACCAPO .   (rule 3)
  $default      reduce using rule 3 (line)

state 4
  input -> input line .   (rule 2)
  $default      reduce using rule 2 (input)
```

```

/* file: simple_calc.flex CONTINUA */

state 5
  line -> exp . ACCAPO      (rule 4)
  exp  -> exp . exp PIU     (rule 6)
  exp  -> exp . exp MENO    (rule 7)
  exp  -> exp . exp DIVISO  (rule 8)
  exp  -> exp . exp PER     (rule 9)
  NUM          shift, and go to state 2
  ACCAPO       shift, and go to state 6
  exp          go to state 7

state 6
  line -> exp ACCAPO .      (rule 4)
  $default   reduce using rule 4 (line)

state 7
  exp  -> exp . exp PIU     (rule 6)
  exp  -> exp exp . PIU     (rule 6)
  exp  -> exp . exp MENO    (rule 7)
  exp  -> exp exp . MENO    (rule 7)
  exp  -> exp . exp DIVISO  (rule 8)
  exp  -> exp exp . DIVISO  (rule 8)
  exp  -> exp . exp PER     (rule 9)
  exp  -> exp exp . PER     (rule 9)
  NUM          shift, and go to state 2
  PIU          shift, and go to state 8
  MENO         shift, and go to state 9
  PER          shift, and go to state 10
  DIVISO       shift, and go to state 11
  exp          go to state 7

state 8
  exp  -> exp exp PIU .     (rule 6)
  $default   reduce using rule 6 (exp)

state 9
  exp  -> exp exp MENO .    (rule 7)
  $default   reduce using rule 7 (exp)

state 10
  exp  -> exp exp PER .     (rule 9)
  $default   reduce using rule 9 (exp)

state 11
  exp  -> exp exp DIVISO .  (rule 8)
  $default   reduce using rule 8 (exp)

state 12
  $          go to state 13

state 13
  $default   accept

```

07 – D VCG (Visualization of Compiler Graphs)

Eseguendo bison con l'opzione **-g** si richiede la generazione di un file con estensione **vcg**: si tratta di un formato particolare per descrivere dei grafici.

E' possibile visualizzare tali grafici con tool appositi, come ad esempio **aiSee** per Windows, reperibile a questo URI:

<http://www.absint.com/aiSee/download/>

Suggerisco anche questo link, completo di molte informazioni anche per Linux:

<http://rw4.cs.uni-sb.de/~sander/html/gsvcg1.html#examples>

Una volta installato, bisogna aprire il file **vcg** da noi creato per poter visualizzare un grafico come questo:

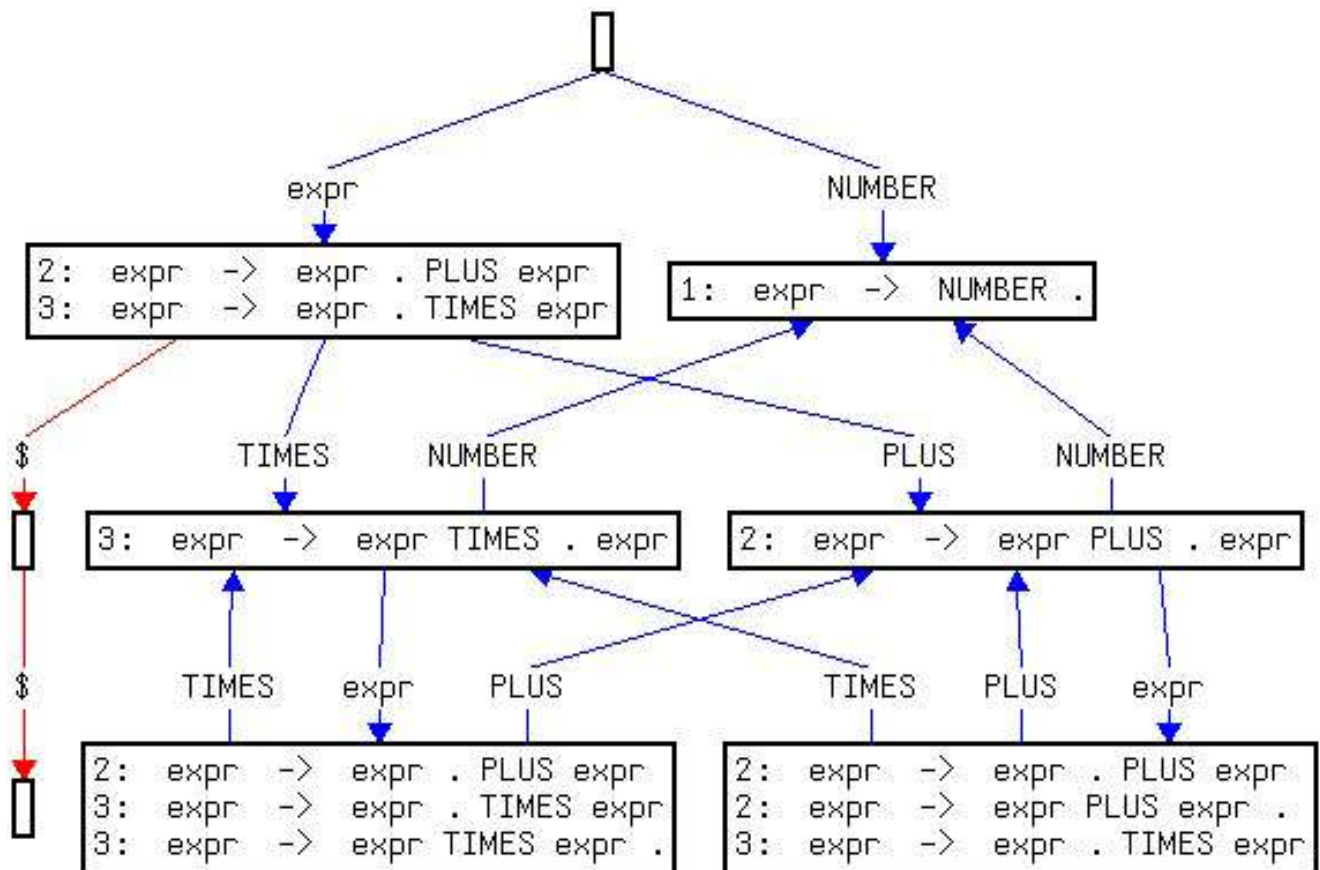


figura 07.07

Si tratta della rappresentazione di un automa, e questo schema dovrebbe permettere di capire il funzionamento della nostra grammatica.

Invito gli studenti a provare questa cosa con propri esempi, e cercare di capire bene il funzionamento della grammatica scelta.

07 – E Conflitti

Nel corso del parsing l'automa che si occupa di prelevare i token e trasformarli in base alle regole grammaticali può incontrare dei **conflitti**, quando ad esempio non sa se effettuare una operazione di **shift** oppure di **reduce**.

Si definiscono conflitti **shift-reduce** quando le due opzioni possibili sono shift e reduce; allo stesso modo si definiscono conflitti **reduce-reduce** quando le due opzioni possibili sono entrambe reduce, ma con espressioni differenti.

Tali ambiguità possono essere risolte in vari modi; generalmente gli **shift-reduce** si risolvono aggiungendo simboli nonterminali, mentre i **reduce-reduce** si possono risolvere utilizzando un'analisi lessicale più intelligente.

Esempio di shift-reduce conflict

```
...
%{
void yyerror(char *s);
%}
%union {
node_ptr node;
}
%token PLUS TIMES
%token <node> NUMBER
%type <node> expr
%start expr
%%
expr :      NUMBER |
        expr PLUS expr |
        expr TIMES expr ;
%%
...
```

Esempio di reduce-reduce conflict

```
...
%left PLUS TIMES LP RP
%token <node> NUMBER ID
%type <node> expr zeroaryfuncall unaryfuncall array
%start expr
%%
expr :      zeroaryfuncall |
        unaryfuncall |
        array |
        expr PLUS expr ;

zeroaryfuncall : ID LP RP ;
unaryfuncall : ID LP RP ;
array : ID LP RP ;
%%
...
```

Lo studente è invitato a provare i due esempi di cui sopra e risolvere i conflitti.

07 – F esempi

esempio 1: infix_calc

Questo è un esempio di una calcolatrice con notazione infissa, con un conflitto di **shift/reduce**.

Lanciando il programma da un input "4 * 2 + 5 * 3", ci accorgiamo che prima tutti i numeri vengono spostati nello stack e convertiti in espressioni, poi la riduzione degli operatori binari viene effettuata dalla destra dello stack, e quindi l'interpretazione di quella espressione diviene "4 * (2 + (5 * 3))", diversa da ciò che vorremmo. Qui i due sorgenti:

```
infix_calc.flex

%{
#include"infix_calc.tab.h"
%}
%option noyywrap
%%
[ \t\r\f]          ;
"+"               return(PLUS);
"*"               return(TIMES);
[1-9][0-9]*       return(NUMBER);
.                  { printf("Not recognized.\n"); exit(1); }
%%

infix_calc.y

%{
#define YYSTYPE int
void yyerror(char *s);
%}
%token PLUS TIMES NUMBER
%%
expr          : NUMBER          |
               expr PLUS expr   |
               expr TIMES expr ;
%%
void yyerror(char *s)
{
    extern char yytext[];
    printf("At token \"%s\": %s\n", yytext, s);
}
int main() { yydebug = 1; yyparse(); }
```

Ecco i comandi da lanciare (da notare che il bison comunica la presenza di conflitti shift/reduce), o con **file di input** o con **standard input**:

```
sim@debian:~/$ bison -d -v -t -g infix_calc.y
infix_calc.y contains 4 shift/reduce conflicts.
sim@debian:~/$ flex -oinfix_calc.yy.c -l infix_calc.flex
sim@debian:~/$ gcc infix_calc.tab.c infix_calc.yy.c -o infix_calc.exe
sim@debian:~/$ ./infix_calc.exe < FILE_DI_INPUT
...

sim@debian:~/$ ./infix_calc.exe
starting parse
Entering state 0
Reading a token: █
```



```

infix_calc_2.y

%{
#define YYSTYPE int
void yyerror(char *s);
%}
%token PLUS TIMES NUMBER
%%
expr      : expr PLUS factor |
          : factor ;
factor    : factor TIMES NUMBER |
          : NUMBER ;
%%
void yyerror(char *s)
{
extern char yytext[];
printf("At token \"%s\": %s\n", yytext, s);
}
int main() { yydebug = 1; yyparse(); }

```

Ecco i comandi:

```

sim@debian:~/$ bison -d -v -t -g infix_calc_2.y
sim@debian:~/$ flex -oinfix_calc_2.yy.c -l infix_calc_2.flex
sim@debian:~/$ gcc infix_calc_2.tab.c infix_calc_2.yy.c -o infix_calc_2.exe
sim@debian:~/$ ./infix_calc_2.exe < FILE_DI_INPUT
...

sim@debian:~/$ ./infix_calc_2.exe
starting parse
Entering state 0
Reading a token: █

```

Vediamo il grafico corrispondente:

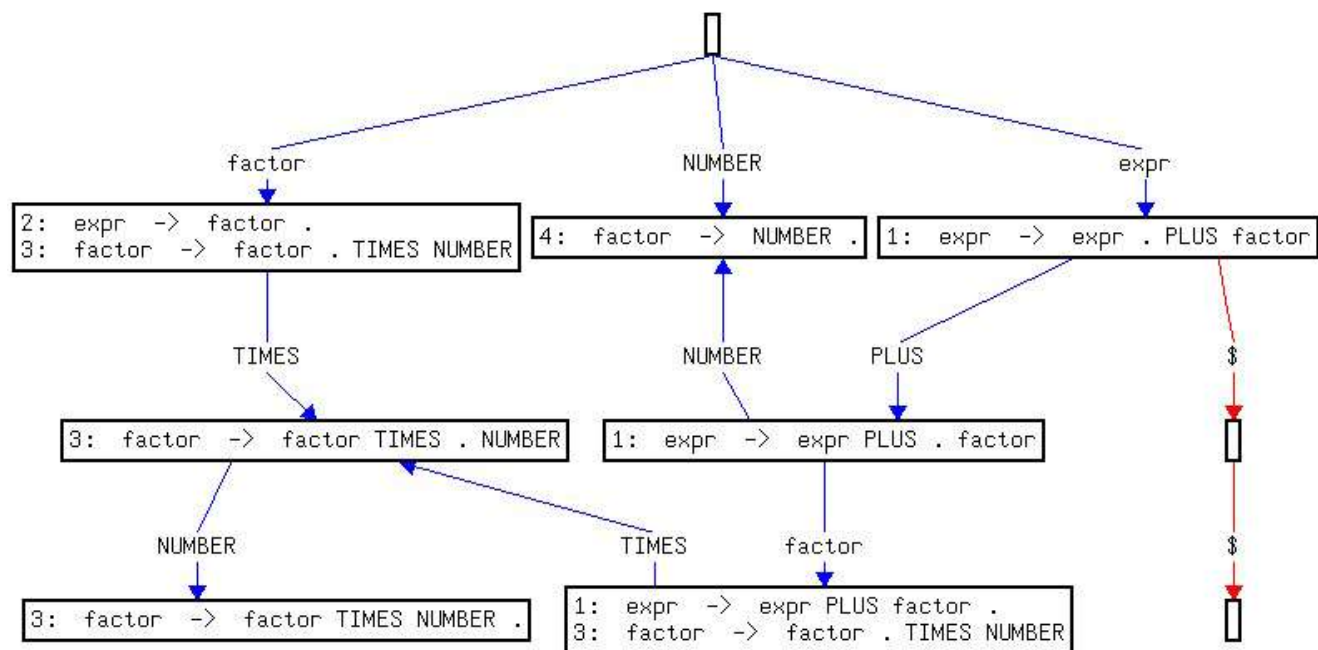


figura 07.09

esempio 3: infix_calc_3

Un altro semplice esempio di calcolatrice a notazione infissa, in cui il conflitto **shift/reduce** viene evitato usando le precedenze nel sorgente Bison.

I comandi da lanciare al prompt sono i soliti.

Come nel secondo esempio, una espressione come "4 * 2 + 5 * 3" viene correttamente interpretata come (4 * 2) + (5 * 3).

"6 + 4 + 7" viene interpretata come (6 + 4) + 7.

Ecco il sorgente bison:

infix_calc_3.y

```
%{
#define YYSTYPE int
void yyerror(char *s);
%}
%left PLUS
%left TIMES
%token NUMBER
%%
expr      : NUMBER          |
          | expr PLUS expr |
          | expr TIMES expr ;
%%

void yyerror(char *s)
{
    extern char yytext[];
    printf("At token \"%s\": %s\n", yytext, s);
}

int main() { yydebug = 1; yyparse(); }
```

Ecco il grafico **vcp**:

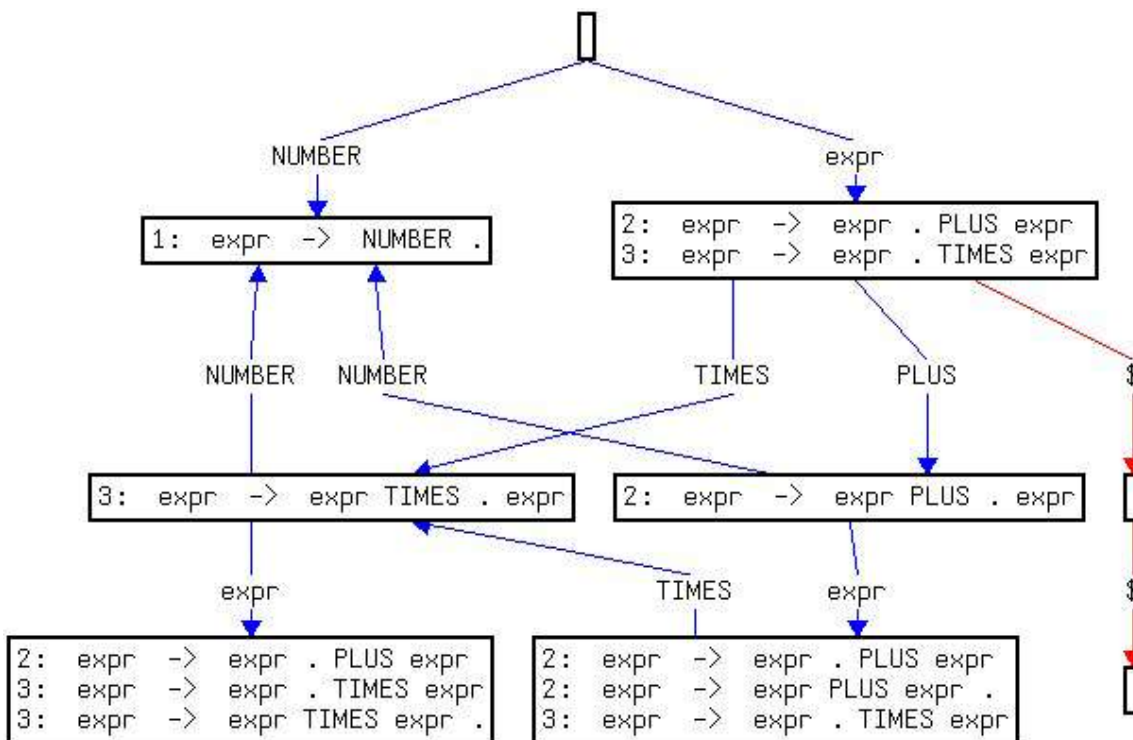


figura 07.10

esempio 4: IF-THEN-ELSE

In questo esempio vogliamo definire una grammatica che accetti stringhe contenenti il tipico costrutto **IF-THEN-ELSE** dei comuni linguaggi di programmazione.

Vediamo lo scanner:

```
%{
#include"ifthenelse.tab.h"
}%
%option noyywrap
%%
[ \n\t\r\f]          ;
"IF"                 return(IF);
"THEN"               return(THEN);
"ELSE"              return(ELSE);
"ISTR"              return(ISTR);
[a-zA-Z]*           return (ID);
.                   { printf("Not recognized.\n"); exit(1); }
%%
```

E il sorgente bison:

```
%{
void yyerror(char *s);
#define YYSTYPE int
}%
%token IF THEN ELSE
%token ID ISTR
%%
stat :      IF ID THEN stat ELSE stat |
        IF ID THEN stat |
        INSTR;
%%
void yyerror(char *s)
{
    extern char yytext[];
    printf("At token \"%s\": %s\n", yytext, s);
}
int main() { yydebug = 1; yyparse(); }
```

```
sim@debian:~/ $ bison -d -v -t -g ifthenelse.y
ifthenelse.y contains 1 shift/reduce conflict.
sim@debian:~/ $ flex -oifthenelse.yy.c -l ifthenelse.flex
sim@debian:~/ $ gcc ifthenelse.tab.c ifthenelse.yy.c -o ifthenelse.exe
sim@debian:~/ $ ./ifthenelse.exe < FILE_DI_INPUT
...
sim@debian:~/ $ ./ifthenelse.exe
starting parse
Entering state 0
Reading a token: █
```

Ecco il grafico **vcg**:

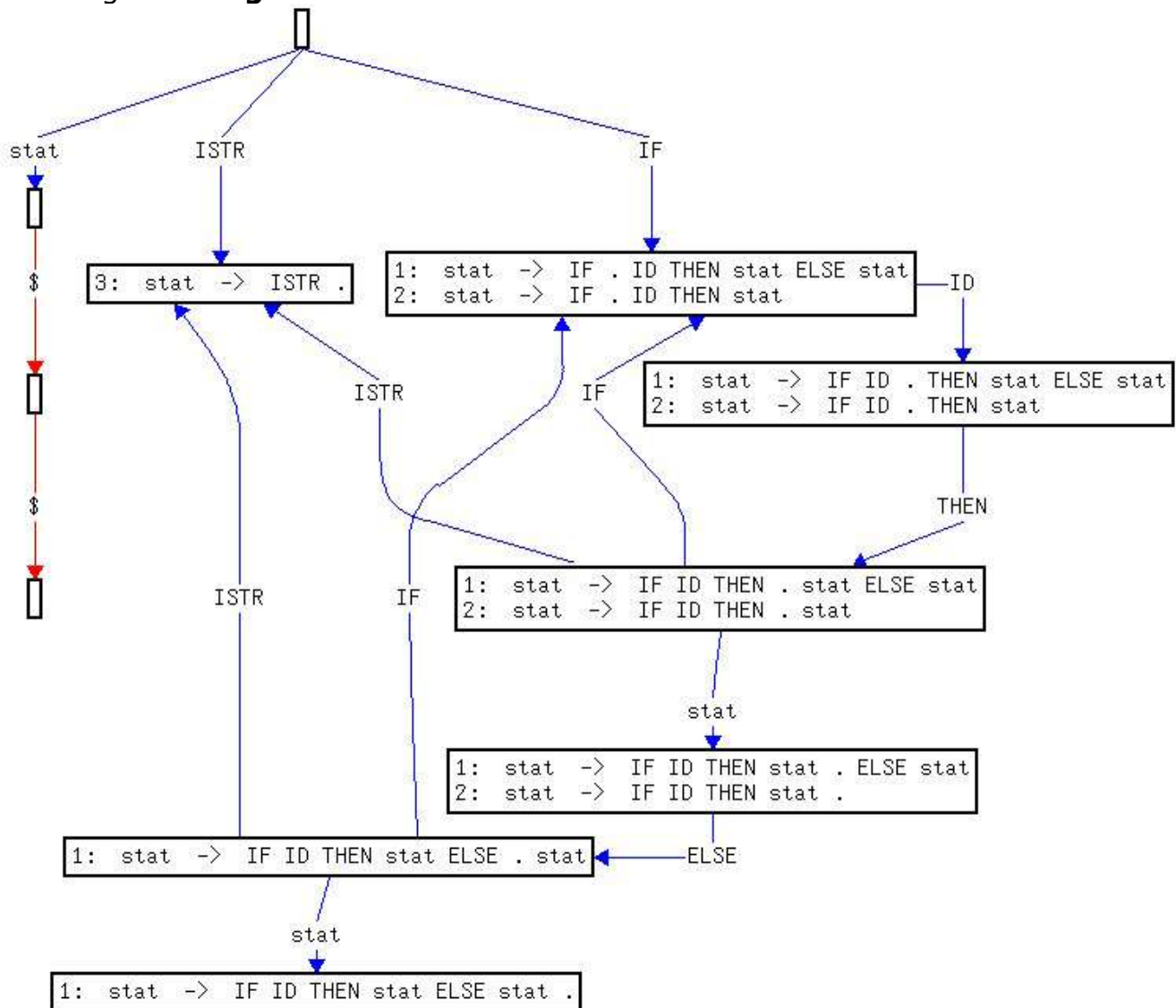


figura 07.11

esempio 5: IF-THEN-ELSE con conflitto risolto

```

%{
void yyerror(char *s);
#define YYSTYPE int
%}
%token IF THEN ELSE
%token ID INSTR
%%
stat      : balanced | unbalanced;
balanced : IF ID THEN balanced ELSE balanced |
          INSTR;
unbalanced : IF ID THEN stat |
            IF ID THEN balanced ELSE unbalanced;
%%
void yyerror(char *s)
{
    extern char yytext[];
    printf("At token \"%s\": %s\n", yytext, s);
}
int main() { yyparse(); }

```

Ecco i comandi:

```
sim@debian:~/ $ bison -d -v -t -g ifthenelse_2.y
sim@debian:~/ $ flex -oifthenelse_2.yy.c -l ifthenelse_2.flex
sim@debian:~/ $ gcc ifthenelse_2.tab.c ifthenelse_2.yy.c -o ifthenelse_2.exe
sim@debian:~/ $ ./ifthenelse_2.exe < FILE_DI_INPUT
...
sim@debian:~/ $ ./ifthenelse_2.exe
starting parse
Entering state 0
Reading a token: █
```

Ecco il grafico **vcg**:

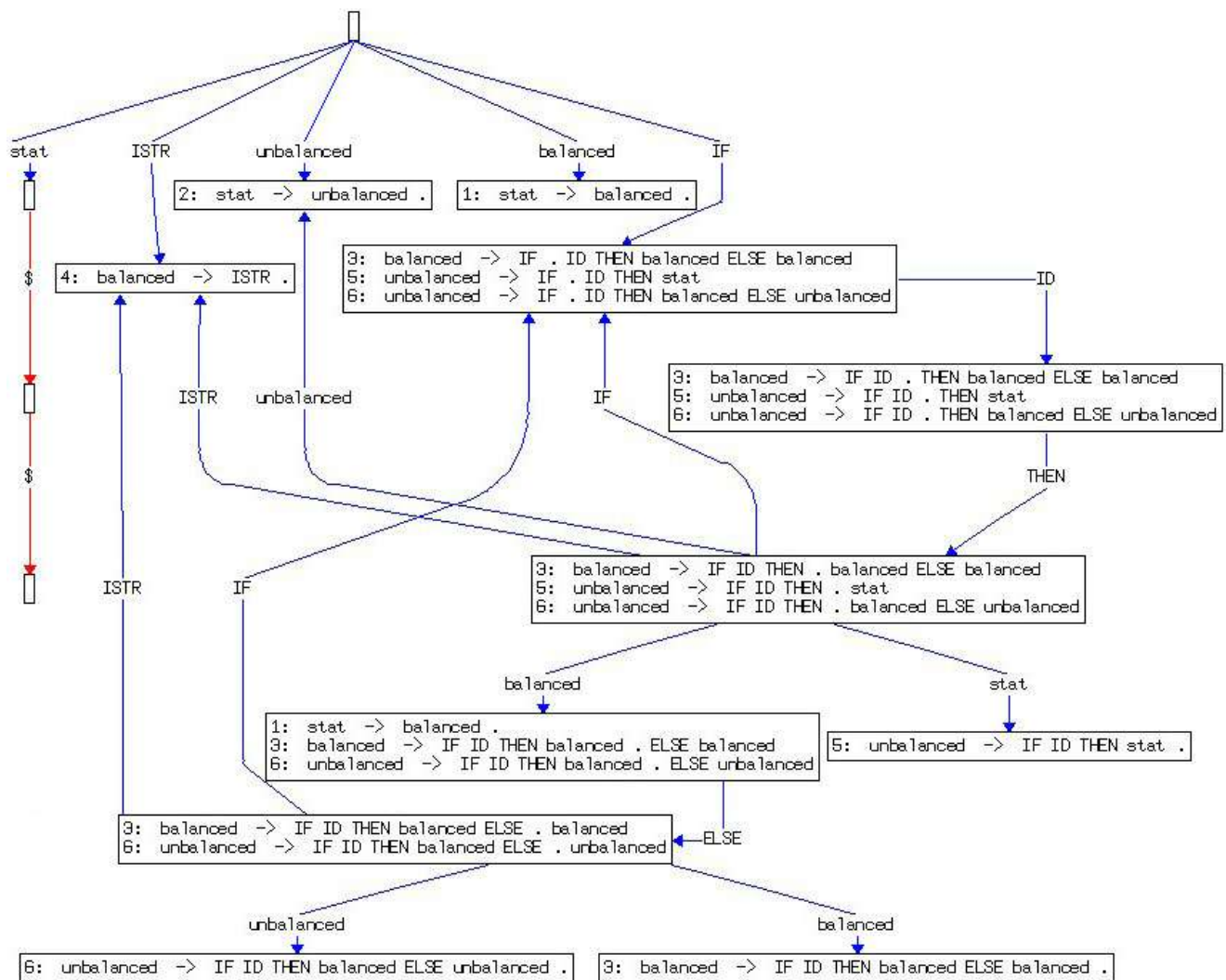


figura 07.12

Vediamo l'output generato dall'esecuzione del parser con la seguente stringa di input:

IF ciao THEN IF bye THEN ISTR ELSE ISTR ELSE ISTR

```

Starting parse
Entering state 0
Reading a token: IF
Next token is 257 (IF)
Shifting token 257 (IF), Entering state 1
Reading a token:
ciao
Next token is 260 (ID)
Shifting token 260 (ID), Entering state 5
Reading a token:
THEN
Next token is 258 (THEN)
Shifting token 258 (THEN), Entering state 6
Reading a token:
IF
Next token is 257 (IF)
Shifting token 257 (IF), Entering state 1
Reading a token:
bye
Next token is 260 (ID)
Shifting token 260 (ID), Entering state 5
Reading a token:
THEN
Next token is 258 (THEN)
Shifting token 258 (THEN), Entering state 6
Reading a token:
ISTR
Next token is 261 (ISTR)
Shifting token 261 (ISTR), Entering state 2
Reducing via rule 4 (line 10), ISTR -> balanced
state stack now 0 1 5 6 1 5 6
Entering state 8
Reading a token:
ELSE
Next token is 259 (ELSE)
Shifting token 259 (ELSE), Entering state 9
Reading a token:
ISTR
Next token is 261 (ISTR)
Shifting token 261 (ISTR), Entering state 2
Reducing via rule 4 (line 10), ISTR -> balanced
state stack now 0 1 5 6 1 5 6 8 9
Entering state 10
Reducing via rule 3 (line 10), IF ID THEN balanced ELSE balanced -> balanced
state stack now 0 1 5 6
Entering state 8
Reading a token:
ELSE
Next token is 259 (ELSE)
Shifting token 259 (ELSE), Entering state 9
Reading a token:
ISTR
Next token is 261 (ISTR)
Shifting token 261 (ISTR), Entering state 2
Reducing via rule 4 (line 10), ISTR -> balanced
state stack now 0 1 5 6 8 9
Entering state 10
Reducing via rule 3 (line 10), IF ID THEN balanced ELSE balanced -> balanced
state stack now 0
Entering state 3
Reducing via rule 1 (line 8), balanced -> stat
state stack now 0
Entering state 12
Reading a token:
.
non riconosciuto.

```

figura 07.13