Introduzione alla Teoria dei Linguaggi
ed alle Espressioni Regolari

Antonio Bonifati

Compendio

Questo testo introduce alla teoria dei linguaggi con particolare riferimento alla classe più semplice di linguaggi, quelli regolari, ed ai formalismi che li rappresentano, tra cui le espressioni regolari.

Indice

1. Introduzione ai linguaggi formali

I linguaggi sono un'aspetto fondamentale e onnipresente in informatica. Linguaggi formali vengono usati continuamente dai programmatori per la descrizione non ambigua delle elaborazioni, e sono i cosiddetti linguaggi di programmazione, come il linguaggio C o il più recente Java.

Sono linguaggi anche i formalismi usati per la rappresentazione dell'informazione, come i linguaggi HTML e XML usati sul World Wide Web, o come i linguaggi per la composizione dei testi, come Troff, Tex e Postscript. Il testo presente è stato composto programmando manualmente in linguaggio Troff, mentre le immagini sono state realizzate col linguaggio Pic. Altri linguaggi vengono utilizzati per la manipolazione dei dati, come SQL.

Un problema fondamentale dell'informatica è quello della traduzione automatica da un linguaggio ad un altro. Un programma che effettui questa traduzione è detto compilatore. All'atto pratico l'interesse principale sta nel tradurre automaticamente la descrizione di un programma scritta in un linguaggio ad alto livello in un linguaggio di basso livello.

I *linguaggi di basso livello* o *linguaggi macchina* consistono di semplici istruzioni di calcolo aritmetico e logico, istruzioni per la manipolazione della memoria in vari modi e istruzioni per il controllo del flusso di esecuzione del programma. Sono detti linguaggi macchina perché la loro semantica è cablata direttamente nella logica dei dispositivi che li comprendono. Con il termine *memoria* usato sopra si intendono vari tipi di supporto per la memorizzazione dei dati, che possono essere variamente letti e scritti: memoria centrale, memoria periferica, cioè unità di input/output come tastiere, monitor e dischi, e registri veloci all'interno del processore.

Combinando variamente tutte queste istruzioni di macchina in lunghissime sequenze si possono risolvere automaticamente molti problemi matematici, di elaborazione del testo, crittografia, compressione dati, simulazione di sistemi fisici, gestione delle telecomunicazioni, giochi ecc...

Tuttavia il lavoro umano coinvolto nel combinare queste lunghe sequenze, quando si tratta di risolvere problemi non banali, è notevole. La programmazione in linguaggio macchina è molto difficile, perché bisogna specificare molti dettagli: un piccolo errore compromette il funzionamento dell'intero programma. Per rendere più semplice e veloce l'attività di programmazione, sono stati inventati molti linguaggi formali, che permettono di scrivere sintetiche istruzioni che, una volta tradotte, corrispondono a molti comandi in linguaggio macchina. Ad esempio in un linguaggio a basso livello come l'assembly 386 Intel, un semplice programma in un linguaggio ad alto livello come il C che calcola l'espressione aritmetica:

$$\frac{n(n+1)}{2}$$

la formula di Gauss che è pari alla somma dei primi n numeri interi, assume il seguente aspetto †:

file gauss.c

```
#include <stdio.h>
main()
{
  int n, s;

  scanf("%d", &n);
  s=n*(n+1)/2;

  printf("%d", s);

  exit(0);
}
```

† La sintassi usata per l'assembly è quella di AT&T System V/386. Il sorgente in assembly è stato ottenuto con lo switch *-S* di *gcc*.

file gauss.s

```
	.file	"gauss.c"
	.version	"01.01"
gcc2_compiled.:
	.section	.rodata
.LC0:
	.byte	0x25,0x64,0x0
.text
	.p2align 2,0x90
.globl main
		.type		main,@function
main:
	pushl %ebp
	movl %esp,%ebp
	subl $24,%esp
	addl $-8,%esp
	leal -4(%ebp),%eax
	pushl %eax
	pushl $.LC0
	call scanf
	addl $16,%esp
	movl -4(%ebp),%edx
	incl %edx
	movl %edx,%eax
	imull -4(%ebp),%eax
	cltd
	movl %edx,%ecx
	shrl $31,%ecx
	addl %ecx,%eax
	movl %eax,%edx
	sarl $1,%edx
	movl %edx,-8(%ebp)
	addl $-8,%esp
	movl -8(%ebp),%eax
	pushl %eax
	pushl $.LC0
	call printf
	addl $16,%esp
	addl $-12,%esp
	pushl $0
	call exit
	addl $16,%esp
	.p2align 2,0x90
.L6:
	leave
	ret
.Lfe1:
		.size		main,.Lfe1-main
	.ident "GCC: (GNU) c 2.95.3 20010315 (release) [FreeBSD]"
```

Da un punto di vista teorico i linguaggi ad alto livello offrono le stesse possibilità elaborative dei linguaggi macchina, ma in pratica i programmi ad alto livello sono più facili per l’uomo da scrivere, leggere e modificare. La cosa interessante è che si può fare la traduzione in automatico: si può scrivere un programma in linguaggio macchina che effettua la traduzione da un linguaggio ad alto livello in linguaggio macchina stesso, detto compilatore del linguaggio ad alto livello in esame. Una volta fatto questo grosso lavoro si può usare il compilatore per scrivere, senza più ricorrere al linguaggio macchina ma solo al linguaggio ad alto livello capito dal compilatore, un altro linguaggio ad alto livello, anche abbastanza diverso, ad esempio con un paradigma di programmazione differente. Così ad esempio si può usare il linguaggio C e il relativo compilatore funzionante su una determinata macchina per creare un traduttore per un linguaggio con un paradigma programmativo logico anziché imperativo, come il Prolog.

In generale un compilatore deve essere un programma molto complicato ed intelligente per generare codice macchina ottimale corrispondente al codice ad alto livello. Alcune volte si riesce a fare molto di meglio a mano se si è molto esperti di linguaggio macchina. Ma la maggior parte delle volte non è più necessario ricorrere al linguaggio macchina e in realtà diventa sempre meno necessario con il miglioramento delle capacità di calcolo, specialmente se si scrive software applicativo, piuttosto che software di base.

Inoltre è possibile che per uno stesso linguaggio esista un compilatore in grado di generare codice per diversi tipi di macchine, oppure più compilatori dello stesso linguaggio che generano codici macchina diversi. I codici macchina variano infatti da produttore a produttore, mentre la maggior parte dei linguaggi ad alto livello sono stati standardizzati. Così un programma scritto in linguaggio ad alto livello può essere facilmente portato da un'architettura di macchina ad un'altra.

Oggigiorno i calcolatori sono diventati così veloci che spesso non si ricorre più nemmeno alla traduzione dei programmi in linguaggio ad alto livello. E' spesso sufficientemente per l'utente la velocità di esecuzione raggiunta da un programma che riconosce sintatticamente il linguaggio ad alto livello e man mano che lo riconosce esegue le istruzioni in linguaggio macchina corrispondenti alla semantica delle istruzioni, o tuttalpiù lo traduce in una forma intermedia che viene poi interpretata più facilmente e velocemente, come una specie di misto tra compilazione ed interpretazione, ma senza il prodotto finale della compilazione che è il codice macchina che gira su un determinato tipo di computer. In questo modo il programma sorgente non viene mai tradotto completamente in linguaggio macchina.

Il codice effettivamente eseguito dalla macchina quando un programma viene interpretato anziché compilato, pur corretto, è spesso meno ottimale, tuttavia la messa a punto di grossi programmi risulta facilitata perché il programmatore non deve attendere i tempi spesso lunghi di compilazione per poter testare ogni minima modifica apportata e il debugging è più semplice. Un linguaggio open-source molto popolare e potente che di solito viene interpretato è il Perl.

Teoricamente ogni linguaggio che può essere interpretato può essere anche compilato e viceversa, ma spesso questi linguaggi interpretati offrono delle funzionalità avanzate che non è affatto agevole fornire a tempo di compilazione, come ad es. poter riferire una variabile il cui nome è a sua volta una variabile stringa o poter fare cambiamenti di tipo di una stessa variabile, perciò rimangono molto legati al loro interprete, percui a compilare un linguaggio del genere non si guadagna molto.

E' molto utile cimentarsi nella scrittura di interpreti e compilatori, anche per linguaggi molto semplici. Si imparano e si capiscono molte cose sul funzionamento dei calcolatori. Se si conoscono un po' di tecniche di analisi lessicale e sintattica e un po' di linguaggio di una determinata macchina, nonché un linguaggio di programmazione *general-purpose* come il C, con un po' di metodo non è molto difficile scrivere un compilatore di qualche linguaggio semplice. Per l'interprete non serve nemmeno conoscere il linguaggio macchina, in quanto ci si può appoggiare ai servizi offerti dal C.

Un altro genere di programmi interessanti sono i preprocessori: questi, similmente al preprocessore C, operano sui sorgenti ed effettuano operazioni quali inclusioni (condizionate o non) e sostituzioni di macro definizioni (senza parametri, oppure quelle con parametri, dette anche macro funzioni). In genere vengono usati per convertire sorgenti in altri sorgenti che di solito devono poi essere interpretati o compilati, esattamente come avviene col preprocessore C.

Un preprocessore general-purpose scritto dagli stessi autori del C, Brian Kernighan e Dennis Ritchie, generalmente disponibile sui sistemi UNIX è M4, descritto in [1]. Ne esiste anche una versione libera del progetto GNU della Fondazione per il Software Libero (Free Software Foundation) istituita da Richard M. Stallman.

Da pochi anni a questa parte molte applicazioni offrono dei meccanismi molto semplici, ma anche molto limitati di comunicazione con l'utente: sone le GUI, Graphical User Interface, un paradigma grafico di uso del calcolatore che lo ha reso così semplice da usare come un impianto stereo. Ciò nonostante i linguaggi restano il modo più flessibile e potente per comunicare con il calcolatore, a causa della struttura stessa dei calcolatori, che non sono altro che degli interpreti hardware di linguaggi macchina. Per questa ragione molti sistemi operativi o applicazioni uniscono il meglio dei due mondi e offrono entrambi i paradigmi di comunicazione umana col calcolatore: quello basato sulle GUI e quello basato sui linguaggi di comandi, come il linguaggio della shell Bash delle versioni moderne di UNIX o il linguaggio SQL per la descrizione e la manipolazione dei dati offerto da molti RDBMS.

Così i linguaggi formali usati in informatica assumono notevole interesse, non solo per i progettisti di linguaggi, ma anche per i programmatori applicativi e persino per i semplici utenti, a meno che per quest'ultimi non sia sufficiente limitarsi ai meccanismi offerti dalle GUI per ottenere le elaborazioni desiderate dal proprio calcolatore. Per capire meglio anche solo come usare questi linguaggi, vale la pena di imparare un po' di teoria dei linguaggi, almeno saper leggere una grammatica formale.

In questo testo ho cercato di introdurre un po' di teoria dei linguaggi: imparerete ad usare formalismi quali le grammatiche, gli alberi sintattici, la sintassi BNF, gli automi e le espressioni regolari. Imparerete poi a scrivere analizzatori del lessico dei linguaggi, sia a scriverli manualmente che a generarli in automatico e cosa si intende col termine parsing. C'è ancora molto, ma questo dovrebbe essere sufficiente per una introduzione.

Ho cercato dove possibile di bilanciare teoria e pratica. Invece di proporre esempi o esercizi su linguaggi inutili, ho pensato di impostare la trattazione su pochi esempi, semplici ma reali, come il linguaggio delle espressioni aritmetiche.

Come prerequisito per la lettura dovrebbe essere sufficiente una discreta conoscenza del linguaggio C. Per poter provare gli esempi di codice o sperimentare per conto proprio, dovete avete disponibile un ambiente di sviluppo, consistente almeno di un compilatore C. Meglio ancora se avete a disposizione anche i tool Lex e Yacc. Le versioni libere di UNIX come GNU/Linux o FreeBSD hanno tutto quello che serve.

Se trovate errori vi prego di segnalarmeli all'indirizzo e-mail *antonio.bonifati@libero.it*. Il mio sito web è raggiungibile all'indirizzo:

http://go.to/ninuzzo

1.1. Simboli, vocabolari, linguaggi

Cercheremo di dare una definizione precisa dei termini *frase* e *linguaggio* e di altri termini correlati, ricorrendo ad un esempio di linguaggio piuttosto semplice, ma molto utile e che usiamo quotidianamente: il linguaggio dei numeri. Per semplicità dapprima ci limiteremo a considerare i soli numeri naturali.

In generale dato un insieme o vocabolario di *simboli* atomici, ad esempio cifre, lettere, numeri, parole, segni di punteggiatura, ecc..., si definisce *frase* o *stringa* una giustapposizione di un numero qualsiasi di simboli del vocabolario. Se non si prende alcun simbolo si ha la *stringa vuota* o *stringa nulla*, che nella stampa può essere meglio visualizzata usando un simbolo grafico non appartenente al vocabolario, ad esempio la lettera greca ε.

Come esempio sia V il vocabolario costituito dalle cifre numeriche arabe:

$$V = \{\ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\ \}$$

i seguenti sono esempi di frasi costruite su questo vocabolario:

$$\begin{aligned} S_1 &= \varepsilon \\ S_2 &= 0 \\ S_3 &= 5 \\ S_4 &= 34 \\ S_5 &= 81059032 \\ S_6 &= 000 \\ S_7 &= 00100 \end{aligned}$$

e a parte la stringa vuota S_1 si tratta di numeri naturali; nel caso di S_6 e S_7 ci sono un paio di zeri superflui in testa.

Le definizioni di simbolo, frase e vocabolario sono la base per la definizione formale di linguaggio. Un *linguaggio* è semplicemente un insieme, finito o infinito, di stringhe costruite su un vocabolario.

Continuando l'esempio del vocabolario delle cifre, possiamo definire il linguaggio L dei numeri naturali come l'insieme di tutte le possibili stringhe che si possono generare dal vocabolario V, qualsiasi sia la loro lunghezza. Si tratta chiaramente di un linguaggio infinito e possiamo decidere se includervi o meno la frase nulla ε.

Questo include naturalmente numeri che iniziano con zeri superflui. Se vogliamo escluderli, in modo da avere un'unica rappresentazione per ciascuna quantità, dobbiamo dare una definizione più complicata, specificando come è fatto un numero con zero superflui e che questo tipo di frase deve essere esclusa dal linguaggio. Per semplicità adottiamo la prima definizione.

Come altro esempio possiamo definire il linguaggio dei numeri pari L_{pari}, come l'insieme di tutte le possibili stringhe costruite su V che terminano col simbolo-cifra 0, 2, 4, 6 oppure 8. Analogamente si può definire il linguaggio dei numeri dispari $L_{dispari}$. E' ovvio per definizione che si tratta di due linguaggi disgiunti, a meno che non comprendano entrambi la stringa vuota e che, naturalmente:

$$L = L_{pari} \bigcup L_{dispari}$$

se L_1, L_2 e L non comprendono la stringa vuota, oppure se L e almeno uno tra L_1 e L_2 la comprende.

Ai linguaggi si applicano quindi le usuali operazioni insiemistiche, perché non sono altro che insiemi di frasi.

Sia L_{pari}, $L_{dispari}$, sia L sono linguaggi infiniti. Il fatto che un linguaggio sia infinito naturalmente può renderlo molto potente, specie se il linguaggio non è banale, ma possiamo definire utilmente anche linguaggi finiti, cioè formati da un numero misurabile di frasi. Banalmente possiamo definire il linguaggio $L(V)$ basato direttamente sui simboli del vocabolario delle cifre V, nel senso che tutte le stringhe del linguaggio sono formate da un solo simbolo o, si dice, sono di lunghezza 1:

$$L(V) = \{\ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\ \}$$

questo linguaggio è finito e il numero di stringhe possibili è 10.

Tra i linguaggi definiti finora: L, L_{pari}, $L_{dispari}$, $L(V)$ ... più molti altri che si potrebbero definire su V ..., L è l'unico che li comprende tutti, nel senso insiemistico:

$$L \supset L_{pari},\ L \supset L_{dispari},\ L \supset L(V),\ \cdots$$

$\supset$ è il simbolo di inclusione stretta. Per questo motivo L è detto *linguaggio universale* su V. Si tratta infatti di un insieme universo, un insieme che comprende tutti i linguaggi definiti su V:

$$L = L_{pari} \bigcup L_{dispari} \bigcup L(V) \bigcup \cdots$$

Possiamo considerare molti altri esempi di linguaggi: il più importante è forse il linguaggio naturale. Le lingue come l'italiano, potrebbero essere definite con rigore, con un grandissimo lavoro, perchè sono molto complicate. Anche la definizione del solo vocabolario non è cosa da poco. Così dobbiamo accontentarci di studiare nel dettaglio solo linguaggi più semplici. Proseguiamo l'esempio aritmetico visto sopra definendo i numeri relativi detti anche numeri interi con segno. In tal caso il vocabolario comprende due simboli in più:

$$V = \{\ +, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\ \} \qquad [1.1\text{-}1]$$

e in una definizione a parole potremmo dire che il linguaggio dei numeri relativi è l'insieme di tutte le possibili stringhe che si possono generare dal vocabolario V sopra definito, dove però i simboli + e − possono comparire solo come primo simbolo della stringa, ma non devono per forza comparire.

1.2. Grammatiche e alberi sintattici

Questa e altre definizioni possono essere scritte in modo più preciso e agevole, se anziché usare il linguaggio naturale, si introduce un formalismo che permetta di definire il linguaggio e che risulti utile anche per leggere o scrivere il linguaggio. Ad esempio nel caso dei numeri relativi possiamo formalizzare così la descrizione del linguaggio †:

regole [1.2-1]

$Z \rightarrow N$	un numero relativo Z è un numero naturale senza indicazione esplicita di segno
$Z \rightarrow +N$	oppure un numero naturale preceduto dal segno +
$Z \rightarrow -N$	oppure un numero naturale preceduto dal segno −
$N \rightarrow C$	e un numero naturale è una cifra
$N \rightarrow CN$	oppure è un altro numero naturale preceduto da una cifra giustapposta
$C \rightarrow 0$	una cifra è lo 0
$C \rightarrow 1$	oppure l'1
$C \rightarrow 2$	oppure il 2
$C \rightarrow 3$	oppure il 3
$C \rightarrow 4$	oppure il 4
$C \rightarrow 5$	oppure il 5
$C \rightarrow 6$	oppure il 6
$C \rightarrow 7$	oppure il 7
$C \rightarrow 8$	oppure il 8
$C \rightarrow 9$	oppure il 9

ora qualsiasi numero relativo potrà essere generato seguendo quelle regole grammaticali, tramite un meccanismo generativo. Vediamo di che si tratta con un esempio, il numero relativo −13:

(1) parto dal non terminale Z

$$Z$$

(2) trasformo il non terminale di partenza scegliendo la regola che più assomiglia alla frase che voglio generare:

$$Z \rightarrow -N$$

(3) poi usando un'altra regola di sostituzione, quella che dice che $N \rightarrow CN$, ho:

$$Z \rightarrow -N \rightarrow -CN$$

(4) poi se uso la regola $C \rightarrow 1$ ho:

$$Z \rightarrow -N \rightarrow -CN \rightarrow -1N$$

(5) quindi uso la regola $N \rightarrow C$:

$$Z \rightarrow -N \rightarrow -CN \rightarrow -1N \rightarrow -1C$$

(6) infine uso la regola $C \rightarrow 3$:

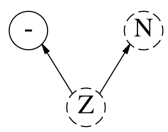
$$Z \rightarrow -N \rightarrow -CN \rightarrow -1N \rightarrow -1C \rightarrow -13$$

Tutto il processo si chiama *derivazione* o *generazione* e può essere rappresentato graficamente con un grafo chiamato *albero sintattico* ††:

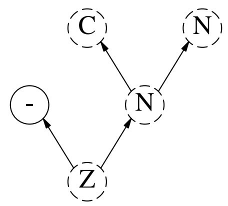
$$Z \qquad (1)$$

† Se si vuole generare anche la stringa vuota, occorre introdurre nella grammatica una regola del tipo $Z \rightarrow \varepsilon$.

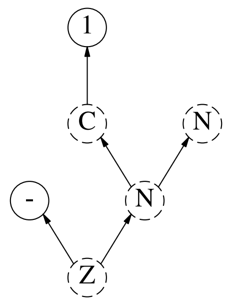
†† Di solito questo albero viene disegnato a rovescio perché la direzione usuale di scrittura è dall'alto verso il basso, altrimenti bisognerebbe prevedere quanto spazio lasciare in alto, poiché il disegno parte dalla radice. Siccome questo computo lo fa automaticamente pic(1), ho deciso di rompere con la tradizione, dandogli l'aspetto di un vero albero, con la radice sotto e le foglie sopra.



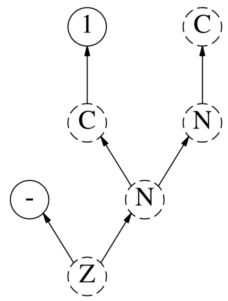
$$Z \rightarrow -N \tag{2}$$



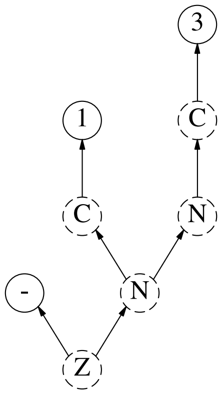
$$Z \rightarrow -N \rightarrow -CN \tag{3}$$



$$Z \rightarrow -N \rightarrow -CN \rightarrow -1N \tag{4}$$



$$Z \rightarrow -N \rightarrow -CN \rightarrow -1N \rightarrow -1C \tag{5}$$



$$Z \to -N \to -CN \to -1N \to -1C \to -13 \tag{6}$$

Ho scelto di rappresentare i simboli non terminali con cerchi tratteggiati, mentre con cerchi solidi ho rappresentato le foglie, cioè i simboli terminali. Un nodo foglia non può avere ulteriori figli.

Per impratichirvi scegliete dei numeri relativi a caso e poi provate a derivarli dalle regole e a disegnare nel frattempo il relativo albero sintattico, aggiungendo un ramo per ogni regola che applicate.

Un po' di terminologia: considerate l'insieme V definito in [1.1-1], più l'insieme delle regole [1.2-1], più l'insieme dei non terminali Z, N, C, più la regola che il non terminale da cui partire per poter generare tutti e soli gli elementi del linguaggio è Z. Tutte queste definizioni messe insieme costituiscono quella che si chiama la *grammatica* del linguaggio.

L'insieme V è detto *insieme dei simboli terminali* del linguaggio. Questi sono i simboli che giustapposti costituiscono le frasi legali nel linguaggio e sono sempre nodi foglia nella rappresentazione ad albero, a differenza dei simboli Z, N, C che non compaiono mai nelle frasi finite del linguaggio. Quest'ultimi sono quindi dei *non terminali*, cioè simboli usati per la definizione del linguaggio. Non compaiono nelle frasi finite perché vengono sostituiti con altri non terminali e/o simboli terminali durante il processo di generazione delle frasi del linguaggio secondo quanto stabilito dalle regole della grammatica †, finché non rimangono più non terminali.

Anche i non terminali si possono raggruppare in un insieme, l'*insieme dei simboli non terminali* della grammatica. Nell'albero sintattico di una frase i non terminali non sono mai foglie, ma nodi intermedi. Il nodo radice dell'albero deve essere sempre un non terminale, altrimenti non ci sarebbe molto da generare e l'albero sarebbe ridotto ad un solo nodo foglia.

Sfortunatamente non c'è un unico modo di scrivere una grammatica, anche se si descrive esattamente lo stesso linguaggio. Possono esserci più grammatiche del tutto equivalenti, ma che differiscono per la scelta dei non terminali, per il numero e la struttura delle regole.

Anche nel caso semplice della grammatica dei numeri relativi vista prima, le cui regole sono date dalle [1.2-1], se cambio la regola $N \to CN$ con un'altra regola $N \to NC$, ottengo una grammatica equivalente ma diversa. Equivalente perché non ha importanza se nel processo di generazione di un numero naturale parta da una cifra e continui ad aggiungere cifre alla sua destra o alla sua sinistra; riesco comunque a generare qualsiasi numero naturale.

Questo fatto che lo stesso linguaggio può essere generato da grammatiche diverse ci complica la vita perché non tutte le grammatiche equivalenti hanno le stesse proprietà, anche se generano lo stesso linguaggio. Ad esempio alcune grammatiche consentono di generare la stessa frase tramite alberi sintattici topologicamente diversi. Questo significa che la struttura sintattica di una frase, come definita dalla grammatica, non è unica, ma si presta a diverse interpretazioni, situazione di solito non desiderata. In tal

† I non terminali rappresentano la struttura sintattica delle frasi. Nei libri di grammatica della lingua italiana, anche se la grammatica viene purtroppo definita in modo informale e incompleto, vengono spesso mostrate delle produzioni dove sono introdotti dei nomi per i non terminali: soggetto, verbo, complemento, aggettivo, ecc...

caso si dice che la grammatica è ambigua.

Per esempio la grammatica che segue, dove per il simbolo n si intendono tutti i numeri, ad es. reali, cosicché V_T è un insieme infinito, definisce correttamente la struttura delle espressioni aritmetiche contenenti le quattro operazioni +, –, *, / † e le parentesi tonde:

grammatica [1.2-2]

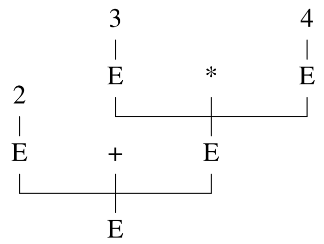
$$V_T = \{+, -, *, /,), (, n\}$$
$$V_N = \{E\}$$
$$R = \left\{ \begin{array}{l} E \to E + E \\ E \to E - E \\ E \to E * E \\ E \to E \;/\; E \\ E \to (E) \\ E \to n \end{array} \right\}$$
$$\textit{simbolo iniziale} = E$$

tuttavia risulta difficile da implementare in un *analizzatore semantico* (*parser*) perchè è ambigua. Se ad esempio ho l'espressione senza parentesi:

$$2 + 3 * 4$$

è facile verificare, tramite un processo di derivazione testuale o grafico usando gli alberi sintattici, che questa può essere interpretata, già dal punto di vista sintattico, senza parlare ancora di aspetti semantici, in due modi differenti. Infatti posso pensare al 2 come la prima espressione e al $3 * 4$ come la seconda espressione da sommare alla prima, secondo la seguente produzione:

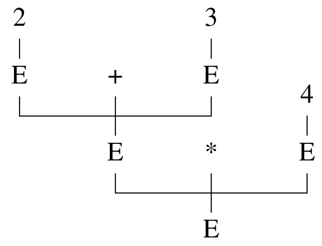
$$E \to E + E \to 2 + E \to 2 + E * E \to 2 + 3 * E \to 2 + 3 * 4 \qquad [1.2\text{-}3]$$



oppure posso pensare che la prima espressione sia $2 + 3$ e che vada moltiplicata per la seconda espressione che è semplicemente un 4 e che l'intera espressione sia generata tramite la seguente produzione:

$$E \to E * E \to E + E * E \to 2 + E * E \to 2 + 3 * E \to 2 + 3 * 4 \qquad [1.2\text{-}4]$$

† Il simbolo / in informatica viene comunemente utilizzato per indicare la divisione al posto del : e qui ho scelto di mantenere questa convenzione. E' chiaro, naturalmente, che l'ambiguità di questa grammatica è strutturale e prescinde dal significato attribuito al linguaggio generato.



Con la prima produzione [1.2-3] abbiamo individuato una struttura della frase che dà precedenza al simbolo * rispetto al simbolo +, nonostante il simbolo + venga prima nella frase. Con la seconda produzione [1.2-4] abbiamo assegnato alla frase una struttura che sembra dare una priorità superiore al simbolo + rispetto al simbolo *. Nei due casi abbiamo ottenuto due alberi sintattici topologicamente diversi.

Qui sto usando la parola simbolo invece di operatore, proprio per mettere in evidenza che non è rilevante il fatto che si tratti di espressioni aritmetiche: qualsiasi siano i simboli terminali e non terminali e il loro significato, se la struttura delle regole è la stessa, l'ambiguità rimane, perché dipende dalla struttura delle regole, non dal significato che attribuiamo ai simboli terminali o non terminali.

Se invece avessimo considerato le produzioni $(2 + 3) * 4$ oppure $2 + (3 * 4)$, allora l'ambiguità della grammatica non si sarebbe rivelata, perché entrambe queste frasi possono essere prodotte da un solo albero sintattico. La grammatica rimane comunque ambigua, perché basta che ci sia anche solo una produzione che può essere fatta in modi diversi a renderla ambigua secondo la definizione.

Se consideriamo per un attimo il significato algebrico della stringa generata $2 + 3 * 4$ vediamo che nel caso della produzione [1.2-3] abbiamo dato precedenza all'operatore * rispetto all'operatore +, mentre nella produzione [1.2-4] è successo l'inverso. Se poi considerate la produzione $2 + 3 + 4$, vedete che anche qui il problema dell'ambiguità si ripresenta. Potreste obiettare che $(2 + 3) + 4$ è la stessa cosa di $2 + (3 + 4)$, ma questo in realtà vale solo quando si considera il significato del simbolo +. In generale, quando al posto del + inserite qualche altro simbolo con un significato diverso, le due espressioni possono essere differenti e in generale l'ambiguità rimane. Ad esempio non è la stessa cosa fare $(2 - 3) - 4$ e fare $2 - (3 - 4)$, considerando il significato algebrico del simbolo $-$.

Se conveniamo di utilizzare sempre le parentesi, possiamo ignorare il problema dell'ambiguità, ma dobbiamo stare molto attenti a non scrivere espressioni ambigue, rendendoci ogni volta conto del problema e inserendo le parentesi opportunamente per forzare il significato che intendiamo. Probabilmente all'atto pratico è più conveniente stabilire una gerarchia tra i simboli, per non dover rendere obbligatorio l'uso delle parentesi o per evitare di dover usare troppe parentesi.

Ad esempio per gli operatori aritmetici potremmo dire che * e / sono prioritari rispetto a + e $-$ (potremmo anche fare diversamente ma per convenzione si è scelto così in aritmetica). Inoltre stabiliamo che nel caso di operatori con la stessa priorità, questi vengano valutati da sinistra verso destra, che per noi è la usuale direzione di scrittura. In generale anche con altri tipi di simboli possiamo fare le nostre scelte. Si tratta di convenzioni.

Naturalmente la grammatica scritta prima per le espressioni non tiene conto né delle priorità, né delle facoltà associative che abbiamo scelto arbitrariamente. Da sé rimane ambigua, ma se vi aggiungiamo la tabella delle priorità e le indicazioni di associatività come cose separate dalla grammatica stessa, abbiamo dei criteri per scartare le produzioni che producono ambiguità. Così scarteremo la produzione [1.2-4] perché, anche se conforme alle regole della grammatica, non rispetta le regole di priorità e analogamente potremmo scartare delle produzioni che non rispettano le regole di associatività. All'atto pratico esistono metodi di parsing per implementare anche grammatiche ambigue come questa e lo strumento [4] le mette a disposizione in un modo persino facile da usare, nonostante siano complicate da affrontare manualmente.

Quindi se aggiungiamo le regole sulla priorità e l'associatività possiamo pensare di scrivere un programma interprete o compilatore delle espressioni aritmetiche basato sulla grammatica. Perché dobbiamo aggiungere anche le regole? Perché un programma deve avere sempre un comportamente determinato cablato nel codice del programma stesso, quindi deve sapere se fare $(2-3)-4$ oppure $2-(3-4)$ quando gli viene fornito in input qualcosa di sintatticamente ambiguo come $2-3-4$. In pratica non ci sarà mai ambiguità in un programma; un algoritmo è per definizione qualcosa di determinato, perciò il programma per poter essere scritto dovrà implementare delle regole di priorità e associatività degli operatori.

Abbiamo quindi visto come, in generale, le regole riguardanti la priorità e l'associatività dei simboli terminali possano essere aggiunte ad una grammatica per disambiguarla. Tuttavia è anche possibile cablare queste regole all'interno della grammatica stessa. Se si complica leggermente la grammatica, aggiungendo qualche altro simbolo non terminale, e qualche regola in più, il rispetto delle regole sulla precedenza e l'associatività possono essere forzate durante il meccanismo di generazione. Questo si traduce in una notevole semplificazione del nostro programma interprete delle espressioni, perché ora è basato su una grammatica non ambigua. In tal caso possiamo usare una tecnica di parsing molto semplice da farsi manualmente.

Per esemplificare ecco una versione non ambigua della grammatica delle espressioni che tiene conto delle usuali convenzioni sulla precedenza e l'associatività:

<u>grammatica [1.2-5]</u>

$$V_T = \{+, -, *, /,), (, n\}$$
$$V_N = \{E, T, F\}$$
$$R = \begin{Bmatrix} E \to E+T \\ E \to E-T \\ E \to T \\ T \to T * F \\ T \to T \,/\, F \\ T \to F \\ F \to (E) \\ F \to n \end{Bmatrix}$$
$$\textit{simbolo iniziale} = E$$

qui E è il non terminale di espressione, T è il non terminale di termine e F è il non terminale che rappresenta un intero fattore. Le prime tre regole dicono che una espressione è una somma algebrica di termini e dicono anche che le somme algebriche sono associative a sinistra. Questo cabla nella grammatica un paio di regole di associatività, quelle degli operatori + e –. Poi abbiamo altre tre regole che dicono cos'è un termine: un termine è una produttoria di fattori. Notate che queste regole sono simili alle prime tre e in effetti se considerate forme di frasi del tipo $F_1 * F_2 \,/\, F_3 * F_4$ vedrete che la loro generazione tramite queste tre regole forza l'associatività a sinistra anche per gli operatori * e /.

Infine le ultime due regole dicono cos'è un fattore: è un numero n oppure una espressione tra parantesi. In effetti è facile convincersi che anche una intera espressione tra parentesi deve essere trattata come un fattore.

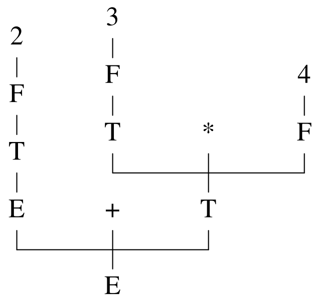
La regola $E \to T$ dice che una intera espressione può essere costituita da un solo termine, mentre $T \to F$ assicura che un termine possa essere costituito da un solo fattore. Queste regole sono molto importanti nel processo di generazione. Se ad es. mancasse la regola $E \to T$, non riusciremmo mai a terminare una somma algebrica di termini, perché resterebbe sempre una E all'inizio della somma:

$$E \to E+T \to E+T+T \to E+T+T+T \to E+T+T+T+T \to \cdots$$

Analogamente se mancasse la regola $T \to F$.

Notate come l'aver introdotto i non terminali T e F e le produzioni relative ha eliminato l'ambiguità derivante dal diverso modo di dare precedenza agli operatori + e *: adesso * assume necessariamente precedenza rispetto a +. Ora ad esempio tutte le produzioni di $2+3*4$ seguono lo stesso albero sintattico, quindi secondo la nuova grammatica questa frase non è più ambigua. Qui di seguito mostriamo solo due di queste produzioni: nella prima ad ogni passo viene espanso sempre il simbolo non terminale più a sinistra e si aggiungono rami all'albero sempre da sinistra verso destra e per tal regolarità questa derivazione viene detta *derivazione canonica sinistra*; la seconda viene chiamata *derivazione canonica destra* perché ad ogni passo si espande sempre il non terminale più a destra della forma di frase corrente e si aggiungono rami all'albero sempre da destra verso sinistra:

$$E \to E+T \to T+T \to F+T \to 2+T \to 2+T*F \to 2+F*F \to 2+3*F \to 2+3*4$$
$$E \to E+T \to E+T*F \to E+T*4 \to E+F*4 \to E+3*4 \to T+3*4 \to F+3*4 \to 2+3*4$$



in entrambi i casi il numero di passi o numero di frecce o se preferite il numero dei rami dell'albero è 8.

C'è però un problema con entrambe le grammatiche [1.2-2] e [1.2-5]. Così come sono formulate queste grammatiche non riconoscono gli operatori + e − in forma unaria, tranne davanti ad un fattore numerico. Così ad esempio le espressioni +(3 − 2) e −(3 − 2) non sono generabili. E' chiaro che dal punto di vista semantico queste espressioni possono essere riscritte come $0+(3-2)$ e $0-(3-2)$ oppure $2-3$, forme che sono sintassicamente corrette secondo le grammatiche [1.2-2] e [1.2-5], sicché gli operatori + e − unari non sono necessari per comporre le espressioni. Risulta comunque opportuno modificare le grammatiche per permettere anche le sintassi con + e − monadico, visto che questi operatori sono comunemente usati. Nel caso della grammatica [1.2-5] basta aggiungere un paio di regole per operare questa estensione del linguaggio generato:

<u>grammatica [1.2-6]</u>

$V_T = \{+, -, *, /,), (, n\}$ *con n positivo senza segno*
$V_N = \{E, T, F\}$

$$R = \begin{Bmatrix} E \to E+T \\ E \to E-T \\ E \to T \\ E \to +T \\ E \to -T \\ T \to T*F \\ T \to T/F \\ T \to F \\ F \to (E) \\ F \to n \end{Bmatrix}$$

simbolo iniziale = E

Dopo questa modifica la grammatica risulta ancora non ambigua? Se *n* include il segno no, perché se ad esempio abbiamo l'espressione −2 * 3, questa potrebbe essere interpretata in due modi diversi, ad es. secondo queste due produzioni canoniche sinistre:

$$E \to -T \to -T*F \to -F*F \to -2*F \to -2*3$$
$$E \to T \to T*F \to F*F \to -2*F \to -2*3$$

questo è un guaio da cui per fortuna possiamo uscire se ridefiniamo V_T in modo che contenga tutti numeri positivi senza indicazione esplicita del segno.

Nonostante tutti i nostri sforzi la grammatica [1.2-6] non è ancora pronta per essere implementata con la tecnica di parsing più semplice da implementare che, a dispetto della sua semplicità, è chiamata con un nome molto complicato: *analisi top-down discendente ricorsiva deterministica*. Il problema sta nel fatto che ci sono regole del tipo $X \to X\ldots$, dove i $\cdots$ stanno per una qualsiasi stringa di terminali e non terminali, mentre X è un simbolo non terminale. Si dice che la grammatica è *ricorsiva a sinistra* o *ricorsiva sinistra*.

Vediamo cosa succede: per l'esempio basta considerare una espressione minimale come 1. Il parser top-down parte dal simbolo iniziale E e decide di provare ad espanderlo con la prima regola $E \to E+T$. Al passo successivo cercherà di espandere la E della parte destra $E+T$ con la prima regola, secondo la produzione $E \to E+T+T$. Non ci vuol molto a capire che siamo incappati in un loop infinito e che il parser si pianterà †.

Per risolvere questo problema riscriviamo la grammatica eliminando le regole incriminate e sostituendole con altre che non danno problemi con l'analisi top-down. Per fare questo notiamo le seguenti equivalenze di regole:

$$\left\{\begin{array}{l} E \to E+T \\ E \to E-T \\ E \to T \\ E \to +T \\ E \to -T \end{array}\right\} = \left\{\begin{array}{l} E \to TE' \\ E \to +TE' \\ E \to -TE' \\ E' \to +TE' \\ E' \to -TE' \\ E' \to \varepsilon \end{array}\right\}$$

$$\left\{\begin{array}{l} T \to T * F \\ T \to T\ /\ F \\ T \to F \end{array}\right\} = \left\{\begin{array}{l} T \to F\ T' \\ T' \to *F\ T' \\ T' \to /\ F\ T' \\ T' \to \varepsilon \end{array}\right\}$$

ancora una volta il costo è di aver introdotto regole e non terminali in più. Operando le sostituzioni di questi due gruppi di regole la grammatica che ne risulta è la seguente:

grammatica [1.2-7]

$V_T = \{+, -, *, /,), (, n\}$ *con n positivo senza segno*
$V_N = \{E, T, F\}$

$$R = \left\{\begin{array}{l} E \to TE' \\ E \to +TE' \\ E \to -TE' \\ E' \to +TE' \\ E' \to -TE' \\ E' \to \varepsilon \\ T \to F\ T' \\ T' \to *F\ T' \\ T' \to /\ F\ T' \\ T' \to \varepsilon \\ F \to (E) \\ F \to n \end{array}\right\}$$

simbolo iniziale $= E$

† Almeno finché non esaurisce la memoria, cioè lo spazio sullo stack. Poi il sistema operativo lo farà terminare per violazione di memoria (*Segmentantion fault*).

e nonostante sembri complicata, questa grammatica è molto semplice da implementare in un parser di tipo top-down, perché non è necessario fare del backtracking, in quanto il parser è deterministico, vale a dire che ad ogni passo guardando anche solo il carattere successivo il parser è in grado di scegliere la giusta produzione con cui proseguire l'analisi. Il modo più semplice di scrivere un parser di questo tipo è fare uso della ricorsione; un'alternativa più complicata consiste nel fare uso di uno stack e di una tabella di guida espliciti, di fatto effettuando una conversione da ricorsione ad iterazione. Non mostrerò qui come scrivere un parser top-down ricorsivo per le espressioni basato su questa grammatica; vi rimando ad un qualsiasi testo sulla teoria dei linguaggi e dei traduttori, oppure all'ottimo tutorial pratico [2].

Per concludere il discorso sui problemi che possono presentare le grammatiche, farò menzione del fatto che possono esserci anche regole inutili o ridondanti o regole che non possono essere mai usate. Ad esempio se aggiungete alle regole [1.2-1] una regola del tipo $Z \to Z$ sarebbe una banale ridondanza. Esempio di regole non raggiungibili: sempre nella grammatica dei numeri relativi le cui regole sono date dalle [1.2-1], se definiamo N come simbolo iniziale, tutte le regole che hanno Z alla parte sinistra non possono essere mai utilizzate.

Per aiutare il progettista a definire grammatiche di buona qualità esistono algoritmi per scoprire questi problemi e risolverli per cui vi rimando a testi più approfonditi. Purtroppo però per il problema dell'ambiguità è stato dimostrato che non esiste un algoritmo che data una grammatica possa stabilire in un tempo finito se è ambigua oppure no. Esistono solo condizioni sufficienti, ma non necessarie, per la non-ambiguità.

Per mia buona fortuna la grammatica dei numeri relativi non soffre di nessuno di tali problemi e si può dire ridotta e non ambigua, come si potrebbe dimostrare. Ho anche verificato che lo stesso vale per la grammatica [1.2-7].

1.3. BNF e EBNF

La grammatica delle espressioni aritmetiche [1.2-7] a cui siamo giunti nel paragrafo precedente era composta da ben 12 regole. Certamente non è agevole leggere e scrivere tutte queste regole a mano o in tipografia, tanto più quanto cresce la complessità delle grammatiche. *BNF* che sta per *forma di Backus-Naur* † non aggiunge niente di nuovo sul piano concettuale, rappresenta solo una notazione abbreviata.

Secondo questa notazione tutte le regole che hanno la stessa parte sinistra vengono raggruppate in una sola riga, con le parti destre pure raggruppate su una riga e separate da un altro metasimbolo separatore che si aggiunge al metasimbolo di espansione $\to$ (oppure ::=). Solitamente per tale separatore si usa il metasimbolo |. Ad esempio ecco la grammatica [1.2-7] riscritta con la sintassi BNF:

grammatica [1.3-1]

$$
\begin{array}{l}
V_T = \{+, -, *, /,), (, n\} \textit{ con n positivo senza segno} \\
V_N = \{E, T, F\} \\
R = \left\{ \begin{array}{l}
E \to TE' \mid +TE' \mid -TE' \\
E' \to +TE' \mid -TE' \mid \varepsilon \\
T \to F\,T' \\
T' \to *F\,T' \mid /\,F\,T' \mid \varepsilon \\
F \to (E) \mid n
\end{array} \right\} \\
\textit{simbolo iniziale} = E
\end{array}
$$

Cosa succede se il simbolo | è un terminale della grammatica che si vuole scrivere in BNF? E' chiaro che in tal caso si ha ambiguità col metasimbolo | di BNF. Per evitare di fare confusione e non capire se | è un metasimbolo di BNF oppure un terminale che fa parte del linguaggio che si vuole descrivere, si conviene di racchiuderlo tra apici singoli o doppi quando è un terminale: '|' "|".

Continuando ad aggiungere zucchero sintattico abbiamo una notazione sintetica più sofisticata che si

† Infatti BNF è l'acronimo di *Backus-Naur Form*.

chiama EBNF †† o BNF estesa.

Per esempio ecco la grammatica BNF [1.3-1] riscritta in forma ancora più compatta EBNF:

grammatica [1.3-2]

$V_T = \{+, -, *, /, ')', '(', n\}$ *con n positivo senza segno*
$V_N = \{E, T, F\}$

$$R = \left\{ \begin{array}{l} E \rightarrow [+ \mid -]\ T\{(+ \mid -)T\} \\ T \rightarrow F\ \{(* \mid /)F\} \\ F \rightarrow '('E')' \ \mid\ n \end{array} \right\}$$

simbolo iniziale = E

In questo caso la notazione permette una notevole sinteticità, perché permette di eliminare molte regole intermedie e relativi non terminali. Notate infatti che non compaiono più esplicitamente i non terminali E', T' e nemmeno il non terminale di stringa vuota ε e le regole EBNF si sono ora ridotte a 3, contro le 5 del corrispondente BNF della [1-3.1] e le 12 regole della [1.2-7].

In EBNF i metasimboli [e] racchiudono stringhe di terminali e non terminali opzionali e ciò permette di risparmiare qualche regola; i metasimboli di ripetizione *{* e *}* indicano che la stringa racchiusa è opzionale ed è possibile ripeterla quante volte si desidera e anche questo permette di ridurre il numero delle regole esprimendo la ricorsione tramite la ripetizione; infine (e) indicano che la stringa racchiusa è obbligatoria e vengono usate per evitare ambiguità quando si raggruppano forme di frase alternative tramite fattorizzazione sinistra o destra. Anzi sarebbe meglio dire che alterano l'ordine degli operatori di EBNF perché l'operatore | ha la massima priorità. Se il linguaggio che si sta descrivendo contiene i simboli terminali [,], *{*, *}*, (o), | questi vanno racchiusi tra apici doppi o singoli onde evitare ambiguità, cioè poter distinguere tra terminali del linguaggio EBNF e terminali del linguaggio descritto. La situazione è analoga a quella che si aveva con BNF, solo che lì il metasimbolo BNF che può creare ambiguità era solo |. Notate che nella [1.3-2] le parentesi sono state quotate per questo motivo.

Notate come sono leggibili le [1.3-2]: corrispondono a dire che una espressione è formata da una sommatoria di termini, eventualmente ridotta ad un solo termine senza alcun operatore di somma algebrica che lo segua. Il primo termine può essere preceduto da un operatore unario che ne indica il segno. Ogni termine è poi una produttoria di fattori e un fattore può essere un numero positivo senza segno oppure un'intera espressione anche molto complessa racchiusa tra parentesi.

Se ci pensate su un po', anche quando scrivete delle grammatiche state usando un altro linguaggio. Si può usare un linguaggio per descrivere un linguaggio. Il primo linguaggio viene detto allora metalinguaggio. Così BNF, EBNF, ma anche il linguaggio più semplice utilizzato per scrivere le regole della [1.2-7] sono tutti metalinguaggi, linguaggi usati per la descrizione di linguaggio ed hanno naturalmente simboli terminali e non terminali, proprio come i linguaggi che descrivono.

Sorge spontanea una domanda: se EBNF descrive la sintassi di altri linguaggi, chi descriverà con precisione la sintassi del linguaggio EBNF? La risposta è il linguaggio EBNF stesso! Infatti è possibile utilizzare EBNF per definire la struttura di una regola di EBNF stesso:

grammatica di una regola EBNF in EBNF

$V_T = \{nt, t, ' \rightarrow ', ')', '(', '\{', '\}', '[', ']', '|'\}$
$V_N = \{R, D, T\}$

$$R = \left\{ \begin{array}{l} R \rightarrow nt\ ' \rightarrow '\ D \\ D \rightarrow T\{'|'T\} \\ T \rightarrow t \mid nt \mid '('D')' \mid '\{'D'\}' \mid '['D']' \end{array} \right\}$$

simbolo iniziale = R

---

†† EBNF sta per *Extended Backus-Naur Form*.

qui abbiamo indicato con *nt* e *t* rispettivamente un simbolo non terminale e terminale del linguaggio descritto da EBNF. Entrambi sono comunque dei simboli terminali per il linguaggio EBNF, così come lo sono $\rightarrow$, $\{$, $)$, ecc... Tuttavia questi ultimi non sono né terminali né non terminali del linguaggio descritto e per tale ragione vengono chiamati *metasimboli* rispetto a questo. I simboli non terminali del linguaggio EBNF sono stati indicati con le lettere maiuscole R, D, T che stanno rispettivamente per regola, parte destra della regola, e termine della parte destra. Così abbiamo formalizzato anche la sintassi di EBNF prima descritta a parole mentre ne spiegavamo la semantica.

Nella maggior parte dei libri che parlano di un determinato linguaggio di programmazione sono spesso presenti le produzioni EBNF di quel linguaggio, cioè le regole della grammatica che consentono di generare frasi legali nel linguaggio espresse secondo la grammatica del metalinguaggio EBNF. Queste regole vengono utilizzate dai progettisti di interpreti e compilatori per implementare il linguaggio che descrivono, ma anche un programmatore può trovarle utili per risolvere eventuali dubbi che sorgano sulla sintassi del linguaggio. Ora sapete come leggere queste regole. Spesso vengono scritte con convenzioni diverse da quelle usate da noi, o addirittura sotto forma di grafi orientati, ma si tratta sempre di EBNF.

1.4. Grammatiche regolari. Grammatiche libere dal contesto.

Probabilmente vi sarete chiesti perché non abbiamo fuso ad esempio le [1.2-7] con le [1.2-1] in un'unica grammatica, la grammatica delle espressioni costruite con numeri relativi. E' naturalmente possibile farlo, ma noi non lo abbiamo fatto perché le due grammatiche, quella delle espressioni scritte nella usuale notazione infissa, cioè con gli operatori binari che sono scritti in mezzo tra un operando e l'altro, e quella dei numeri relativi sono di classi diverse e così pure i linguaggi da loro generati.

Infatti all'atto pratico, quando si scrive ad esempio un programma interprete o compilatore di espressioni, per ragioni di efficienza e semplicità, conviene che siano trattate da moduli diversi: il *parser* o *analizzatore sintattico* costruirà l'albero sintattico delle espressioni, implicitamente o esplicitamente, memorizzandolo per intero o mantenendo solo l'informazione necessaria che li serve per proseguire nella generazione e man mano che riconosce alcuni costrutti sintattici, ad es. una somma, una divisione, ecc... potrà eseguire le azioni semantiche desiderate. Nel caso del compilatore le azioni semantiche consistono nella generazione di istruzioni in linguaggio macchina che quando verranno eseguite, in un momento successivo, cioè a traduzione terminata, calcoreranno il valore dell'espressione. Nel caso di un interprete le azioni semantiche consistono nella valutazione della espressione man mano che viene riconosciuta, utilizzando qualche struttura a stack esplicita o implicita. L'altro modulo invece effettua un parsing molto più semplice, basato sulla grammatica dei numeri relativi (o reali se volete fare una cosa più estesa) e viene chiamato *analizzatore lessicale*. Come comunicano i due moduli? Semplicemente l'analizzatore sintassico utilizza i servizi messi a disposizione dell'analizzatore lessicale, cioè ogni volta che ne ha bisogno gli richiede di analizzare l'input e fornire il prossimo simbolo terminale (dal punto di vista del parser), che sarà uno dei simboli dell'insime V_T della grammatica [1.2-7].

Del resto lo stesso fatto che esistono linguaggi più complicati come EBNF che possono essere usati per la descrizione di linguaggi più semplici di loro, ma non di linguaggi più complicati, indica che è possibile stilare una classificazione dei linguaggi e quindi parallelamente anche delle grammatiche.

Una tale classificazione, basata sulla struttura delle regole, fu fatta dal linguista Noam Chomsky. Non daremo la classificazione completa perché per i nostri scopi è utile conoscerne solo gli ultimi due gradini, che sono le grammatiche regolari e le grammatiche non contestuali.

Una grammatica e il relativo linguaggio che genera è *regolare* se è *regolare destra* oppure se è *regolare sinistra*.

Una grammatica è *regolare destra* se le sue regole sono tutte del tipo $X \rightarrow tY$ oppure del tipo $X \rightarrow t$, dove X e Y sono simboli non terminali, mentre t è una qualsiasi stringa costituita da soli terminali, eventualmente anche la stringa nulla.

Analogamente una grammatica è *regolare sinistra* se le sue regole sono tutte del tipo $X \rightarrow Yt$ oppure del tipo $X \rightarrow t$, dove X e Y sono non terminali della grammatica, mentre t è una stringa di soli terminali della grammatica, anche di lunghezza zero.

La grammatica dei numeri relativi le cui regole sono date dalle [1.2-1] è regolare? La risposta è no, perché c'è la regola $N \rightarrow CN$ che a destra contiene due non terminali, e quindi non è né del tipo $X \rightarrow tY$

né del tipo $X \to Yt$. Però possiamo riscrivere le [1.2-1] come segue:

regole [1.4-1]

$$
\begin{array}{l}
Z \to N \\
Z \to +N \\
Z \to -N \\
N \to 0 \\
N \to 1 \\
N \to 2 \\
N \to 3 \\
N \to 4 \\
N \to 5 \\
N \to 6 \\
N \to 7 \\
N \to 8 \\
N \to 9 \\
N \to 0N \\
N \to 1N \\
N \to 2N \\
N \to 3N \\
N \to 4N \\
N \to 5N \\
N \to 6N \\
N \to 7N \\
N \to 8N \\
N \to 9N
\end{array}
$$

questa grammatica equivalente è regolare destra. Abbiamo semplicemente eliminato il non terminale C, replicando per 10 volte le regole che lo usavano con tutti i valori terminali che C può assumere, cioè le cifre da 0 a 9. Se poi scriviamo:

$$Z \to [+|-](0|1|2|3|4|5|6|7|8|9)\{0|1|2|3|4|5|6|7|8|9\}$$

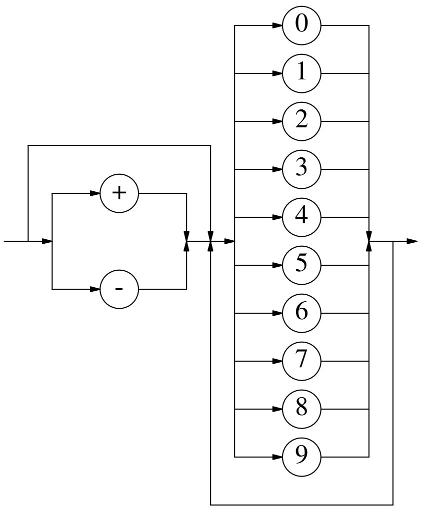
questa grammatica equivalente è regolare destra e per brevità abbiamo usato la sintassi EBNF. Ci accorgiamo che se aggiungiamo un altro di tipo di parentesi, per indicare la ripetizione di una stringa 1 o più volte, ad es. le parentesi angolari, possiamo scriverla in maniera ancora più compatta:

$$Z \to [+|-]\langle 0|1|2|3|4|5|6|7|8|9 \rangle$$

oppure, senza introdurre nuovi tipi di parentesi, possiamo fare la stessa dichiarazione in due tempi successivi, al costo di un non terminale in più, che chiameremo D, che sta per l'inglese digit, cioè cifra:

$$
\begin{array}{l}
D \to 0|1|2|3|4|5|6|7|8|9 \\
Z \to [+|-]D\{D\}
\end{array}
$$

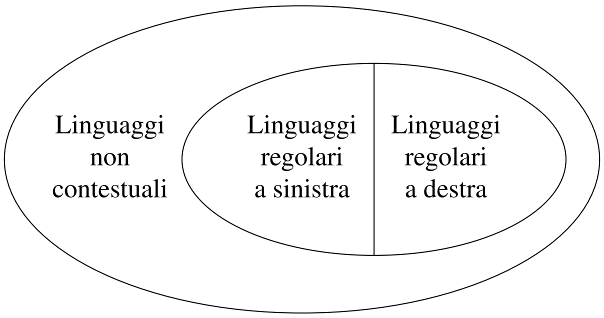
ed ecco poi la sua rappresentazione grafica:



Questa delle grammatiche regolari è la classe più semplice di grammatiche individuata da Chomsky e difatti i linguaggi che genera hanno una espressività abbastanza limitata. Ad esempio non è nemmeno capace di generare le espressioni aritmetiche nelle usuale forma infissa, ad es. $(1 + 2) * 3$ perché non permette di bilanciare le parentesi. Per bilanciare le parentesi a qualsiasi livello di annidamento occorrono derivazioni di autoinclusione, come $E \rightarrow (E)$ che abbiamo visto nella grammatica [1.2-2] oppure $F \rightarrow (E)$ come abbiamo visto nella corrisponde grammatica non ambigua [1.2-5], che non sono permesse dalle grammatiche regolari. Infatti le regole del tipo $E \rightarrow (E)$ oppure $F \rightarrow (E)$ non sono né del tipo $X \rightarrow tY$ né del tipo $X \rightarrow Yt$, ma sono del tipo $X \rightarrow t_1\ Y\ t_2$.

Così una grammatica regolare non sarebbe sufficiente per generare tutte e sole le frasi legali di un linguaggio di programmazione come il C, che contiene espressioni aritmetiche nella forma infissa. Per questo occorrono grammatiche di livello immediatamente superiore, che sono dette *libere dal contesto* o *non-contestuali* e hanno regole in cui nella parte sinistra compare sempre solo un non-terminale, come nel caso delle regolari, ma nella parte destra può comparire una qualsiasi stringa formata da terminali e non terminali in numero a piacere. Le grammatiche [1.2-1], [1.2-2], [1.2-5], [1.2-6], [1.2-7], [1.3-1], [1.3-2] e quella che descrive le regole EBNF, sono tutte di questo tipo, così come quelle che consideremo in futuro. Infatti in questo testo introduttivo non prenderemo mai in esame grammatiche di ordine superiore a quelle non contestuali.

E' chiaro naturalmente che le grammatiche non contestuali sono una estensione di quelle regolari, sicché ogni grammatica regolare sarà anche una grammatica non contestuale, ma non viceversa! Alcune volte una grammatica non contestuale può essere riscritta in forma regolare, come abbiamo fatto sopra con le [1.2-1], ma non sempre ovviamente ciò è possibile. Vale a dire che l'insieme infinito di tutte le possibili grammatiche libere dal contesto contiene l'insieme, pure infinito, di tutte le possibili grammatiche regolari:



1.5. Automi

In questo paragrafo parleremo di *automi a stati finiti* (*finite state automaton*) e della loro relazione con i linguaggi regolari e le loro grammatiche.

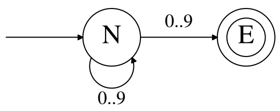
Per descrivere i linguaggi sia da un punto di vista generativo, cioè come si dice comunemente della scrittura del linguaggio, sia da un punto di vista riconoscitivo, cioè dal punto di vista di un lettore, risulta molto utile, come abbiamo visto il meccanismo formale della grammatica derivativa, che è generale e si applica, almeno in teoria, anche a linguaggi di ordine superiore: basta complicare un po' la struttura delle regole.

Limitatamente ai linguaggi regolari esiste anche un altro strumento equivalente alle grammatiche e in grado di essere usato ugualmente bene per leggere e scrivere un linguaggio.

Questo viene detto automa a stati finiti e in particolare permette agevolmente di risolvere automaticamente il problema di riconoscere se una frase appartiene o no al linguaggio regolare che l'automa rappresenta.

Anche qui, come con le grammatiche, è possibile definire in diversi modi l'automa che definisce uno stesso linguaggio, nel senso che tra questi automi equivalenti ci sono differenze non banali, topologiche. Questo rappresenta una complicazione con cui, ahimé, bisogna imparare a convivere.

Per esempio ecco un automa con un numero finito di stati che descrive il linguaggio dei numeri naturali:



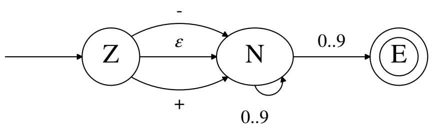
Questo automa è stato ottenuto usando un sottoinsieme delle regole [1.4-1]. Per ogni regola della grammatica è stato tracciato un arco dell'automa secondo la seguente tabella:

regola	arco
$N \to 0$	$N,0 \to E$
$N \to 1$	$N,1 \to E$
$N \to 2$	$N,2 \to E$
$N \to 3$	$N,3 \to E$
$N \to 4$	$N,4 \to E$
$N \to 5$	$N,5 \to E$
$N \to 6$	$N,6 \to E$
$N \to 7$	$N,7 \to E$
$N \to 8$	$N,8 \to E$
$N \to 9$	$N,9 \to E$
$N \to 0N$	$N,0 \to N$
$N \to 1N$	$N,1 \to N$
$N \to 2N$	$N,2 \to N$
$N \to 3N$	$N,3 \to N$
$N \to 4N$	$N,4 \to N$
$N \to 5N$	$N,5 \to N$
$N \to 6N$	$N,6 \to N$
$N \to 7N$	$N,7 \to N$
$N \to 8N$	$N,8 \to N$
$N \to 9N$	$N,9 \to N$

Il formato con cui sono stati indicati gli archi è semplice: un arco che congiunge il nodo N_1 con il nodo N_2, etichettato con il terminale t è stato indicato con la notazione $N_1, t \to N_2$. Si intende che la direzione, cioè la freccia è da N_1 ad N_2. Lo stato finale dell'automa è stato nominato E dalla parola inglese *end* che significa appunto fine. Lo stato iniziale è quello su cui arriva una freccia che non ha alcuna origine, N nell'esempio.

Usiamo la convenzione grafica di indicare lo stato finale con una doppia circonferenza o tramite riempimento e per facilitare il disegno riuniremo più archi che hanno la stessa origine e destinazione in uno solo, listando le etichette. Così i 10 archi che vanno da N a sé stesso sono stati riuniti in uno solo etichettato con la dicitura 0..9, che si intende come un formato compatto per la lista 0,1,2,3,4,5,6,7,8,9. Analogamente per gli altri 10 archi che vanno da N a E.

Questo è invece l'automa relativo al linguaggio nei numeri relativi che, come accadeva per le grammatiche, è una estensione di quello dei numeri naturali, seguito dalla tabella che indica le corrispondenze tra regola della grammatica ed arco dell'automa, che pure è una estensione di quella precedente:



regola	arco
$Z \to N$	$Z, \varepsilon \to N$
$Z \to +N$	$Z, + \to N$
$Z \to -N$	$Z, - \to N$
$N \to 0$	$N, 0 \to E$
$N \to 1$	$N, 1 \to E$
$N \to 2$	$N, 2 \to E$
$N \to 3$	$N, 3 \to E$
$N \to 4$	$N, 4 \to E$
$N \to 5$	$N, 5 \to E$
$N \to 6$	$N, 6 \to E$
$N \to 7$	$N, 7 \to E$
$N \to 8$	$N, 8 \to E$
$N \to 9$	$N, 9 \to E$
$N \to 0N$	$N, 0 \to N$
$N \to 1N$	$N, 1 \to N$
$N \to 2N$	$N, 2 \to N$
$N \to 3N$	$N, 3 \to N$
$N \to 4N$	$N, 4 \to N$
$N \to 5N$	$N, 5 \to N$
$N \to 6N$	$N, 6 \to N$
$N \to 7N$	$N, 7 \to N$
$N \to 8N$	$N, 8 \to N$
$N \to 9N$	$N, 9 \to N$

Un altro modo equivalente di indicare quali simboli terminali e non terminali ci sono e quali archi ci sono, consiste nel fare uso della *tabella di transizione*. Questa tabella non descrive in modo esaustivo l'automa. Bisogna infatti aggiungere l'informazione su quale è lo stato iniziale e qual è lo stato finale, o meglio quali sono *gli* stati finali. Infatti lo stato iniziale è sempre uno, ma in generale un automa può avere anche più di uno stato finale. Almeno uno stato finale deve essere definito, naturalmente. Ad esempio la tabella di transizione dell'automa dei numeri relativi visto prima è la seguente.

	ε	+	-	0..9
Z	{N}	{N}	{N}	
N				{N,E}
E				

Nella prima riga ci sono tutti i possibili simboli terminali che si possono presentare, mentre nella prima colonna ci sono tutti i simboli non terminali, cioè le etichette degli stati dell'automa. Ad es. l'incrocio tra la riga N e la colonna $0..9$ indica che quando l'automa è nello stato N e in input si presenta una cifra, allora può transitare in uno degli stati appartenenti all'insieme N, E, vale a dire che può o rimanere nello stato N o transitare nel nuovo stato E. Quando l'automa è nello stato E non può più transitare in nessun altro stato, perché nessun arco si dirama dal cerchio etichettato con E e quindi la riga corrispondente della tabella di transizione è vuota. Notate che se aggiungiamo l'informazione che Z è lo stato iniziale ed E è quello finale abbiamo descritto completamente l'automa in una forma tabellare anziché grafica.

Esiste una specie di dualismo tra grammatiche ed automi nel caso dei linguaggi regolari: data una grammatica regolare (destra o sinistra), esiste un algoritmo molto semplice per costruire l'automa a stati finiti corrispondente e viceversa si può passare dall'automa alla grammatica. In questa sede ci limiteremo a descrivere l'algoritmo per passare da grammatica regolare ad automa, che probabilmente è il più importante all'atto pratico. A dire il vero per il passaggio *grammatica* $\rightarrow$ *automa* gli algoritmi sono due, a seconda che la grammatica sia regolare destra o sinistra, tuttavia sono molto simili tra loro. Ci limitiamo ulteriormente al caso di grammatica regolare destra, come quella dei numeri relativi. Ecco quindi l'algoritmo generale, descritto a parole:

(1) Aggiungere un nodo che sarà quello finale dell'automa. Per convenzione lo chiameremo E, a meno che questo non sia il nome di un altro non-terminale della grammatica, perché naturalmente bisogna evitare di fare confusione, cioè non si possono avere due nodi con lo stesso nome.

(2) Aggiungere il nodo iniziale dell'automa che deve corrispondere sempre al simbolo iniziale della grammatica. Questo nodo è contraddistinto da una freccia entrante, ma non uscente da nessun altro nodo.

(3) Per ogni regola della grammatica disegnare un arco dell'automa secondo le corrispondenze dettate dalla seguente tabella:

regola	arco
$X \rightarrow \varepsilon$	$X, \varepsilon \rightarrow E$
$X \rightarrow t$	$X, t \rightarrow E$
$X \rightarrow Y$	$X, \varepsilon -> Y$
$X \rightarrow tY$	$X, t \rightarrow Y$

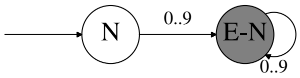
dove X, Y e T sono simboli non terminali, mentre t è un simbolo terminale.

Verificate che proprio queste regole sono state applicate per ottenere gli automi riconoscitori dei numeri naturali e dei numeri relativi prima introdotti a scopo di esempio.

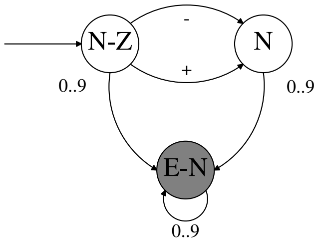
Sfortunamente gli automi che si ricavano dalle grammatiche regolari secondo algoritmi come quello prededente sono di scarsa utilità pratica. Perché? Perché sono *non deterministici*. Cosa significa ed implica questo? Supponete di avere il numero naturale 101 e di voler utilizzare l'automa visto prima per leggerlo. Supponete anche di essere un programma automatico, senza particolare buon senso e intelligenza. Si parte dal primo nodo che è N. Poi scegliete l'arco etichettato con 1 che porta in N stesso. Poi quello etichettato 0 che porta in N, e quindi ancora restate nello stato N. Infine di nuovo quello etichettato 1 verso N. La stringa in ingresso è finita, ma la lettura ancora no, perché lo stato finale non è E, ma ancora N. Il programma che implementa l'automa concluderà quindi che 101 non è un numero naturale? Se scrivete il programma in modo che si fermi al primo percorso di riconoscimento sì. Naturalmente il risultato è sbagliato, perché 101 è un normalissimo numero naturale, mentre il programma ha concluso che non lo è! Un programma corretto non dovrebbe terminare subito. Così come quando state cercando di ritrovare la strada in un labirinto, il programma dovrebbe tornare indietro, e provare tutti i percorsi di riconoscimento possibili attraverso l'automa prima di concludere che una data stringa non è un numero (cioè che nel labirinto non c'è via d'uscita). Solo quando il programma trova un percorso che esaurisce tutta la stringa di ingresso e termina con lo stato finale E, può dare subito il responso che la stringa è un numero. A questo punto infatti è inutile provare altri percorsi, anche se potrebbero esserci altri percorsi differenti che fanno uscire dal labirinto. Se invece dopo aver provato tutti i percorsi possibili, l'analisi della stringa di ingresso non termina sullo stato finale E, allora e solo allora, dopo averli provati tutti, il programma potrà affermare che la stringa di ingresso non è un numero.

Quando un automa presenta archi etichettati col simbolo ε, che sta ad indicare una transizione (ossia passaggio da uno stato all'altro) dell'automa che non comporta il consumo di nessun simbolo terminale dalla stringa di ingresso, oppure presenta più archi che escono dallo stesso nodo e vanno in nodi diversi ma etichettati allo stesso modo, allora si dice *non deterministico*. E come abbiamo visto nell'esempio questo comporta che può esserci più di un percorso che porta al riconoscimento di una determinata stringa e che bisogna scrivere un programma abbastanza complicato per implementare l'automa. Questo programma deve impiegare una tecnica chiamata backtracking, che è la stessa che si può utilizzare per risolvere un labirinto. Nonostante questa tecnica non sia molto difficile da implementare una volta fatta un po' di pratica, possiamo evitarla.

Infatti è possibile dimostrare che è sempre possibile convertire un automa non deterministico in uno *deterministico*, che cioè non presenta ambiguità di scelta alcuna nelle transizioni. Rispetto all'automa non deterministico corrispondente, l'automa deterministico può presentare un maggior numero di stati, ma la sua implementazione risulta enormemente semplificata. Il tempo di riconoscimento risulta proporzionale alla lunghezza della stringa da riconoscere per un automa deterministico, mentre per uno non deterministico non è valutabile a priori dipendendo dalla stringa particolare. La dimostrazione è del tipo costruttiva, nel senso che esiste un algoritmo ben preciso per fare questa conversione, che noi per non annoiar troppo non illustreremo. Ecco però gli automi deterministici riconoscitori dei linguaggi dei numeri naturali e relativi, ottenuti tramite le regole generali di conversione e le loro tabelle di transizione:



	0..9
N	E-N
E-N	E-N

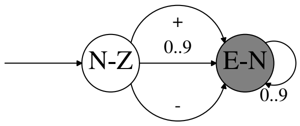


	+	-	0..9
N-Z	N	N	E-N
N			E-N
E-N			E-N

in questo caso il numero degli stati nel caso dell'automa dei numeri naturali è rimasto 2, e quello dei numeri relativi è rimasto ancora 3, ma in generale ci sono casi in cui il numero degli stati di un automa deterministico è molto maggiore rispetto al corrispondente non deterministico.

Infatti ad ogni stato dell'automa deterministico corrisponde un insieme di stati dell'automa non deterministico, come risulta dall'algoritmo di conversione che qui non abbiamo esposto. Difatti abbiamo utilizzato per gli stati dell'automa deterministico dei nomi che riflettono gli stati dell'automa non deterministico che vi corrispondono. Il nome dello stato dell'automa deterministico è stato ottenuto ordinando alfabeticamente la lista degli stati dell'automa non deterministico che vi corrispondono e ottenendo dalla lista ordinata una stringa per concatenazione dei suoi elementi, usando il trattino come separatore per ragioni di leggibilità. Poiché il numero di possibili sottoinsiemi di un insieme di n elementi, incluso l'insieme vuoto, è 2^n, tanti quanti i possibili numeri binari di n cifre, se il numero di stati di un automa non deterministico è n, il numero di stati di un automa deterministico che vi corrisponde può essere al più 2^n escluso, cioè strettamente minore di 2^n, perché ad ogni stato dell'automa deterministico deve corrispondere un insieme non vuoto di stati dell'automa non deterministico e un automa non può avere due o più stati con lo stesso nome.

Notate infine che l'automa deterministico bistatico che segue per i numeri relativi è sbagliato, perché consente di fermarsi dopo la generazione di un solo segno meno o più:



1.6. Analizzatori Lessicali

Nel paragrafo 1.4 abbiamo visto come le grammatiche regolari non siano sufficienti per generare ad esempio il linguaggio delle espressioni aritmetiche in forma infissa e quindi ovviamente nemmeno quello dei linguaggi di programmazione di cui tali espressioni fanno parte, perché permetterebbero di annidare le parentesi, ma non di bilanciarle, in quanto sono privi di regole di autoinclusione.

Tuttavia esistono opportune grammatiche regolari che permettono di generare un linguaggio più vasto rispetto al linguaggio delle espressioni aritmetiche o del C. Cosa ce ne facciamo di un linguaggio più vasto, che contiene sì tutte le frasi legali del linguaggio che ci interessa, ma anche alcune frasi che non sono legali, come per esempio l'espressione $5 * (2 + (3 - 1)$ dove le parentesi sono annidate ma non bilanciate?

Come abbiamo accennato ancora nel paragrafo 1.4, i linguaggi regolari risultano utili nella tecnologia dei compilatori, per ragioni che hanno a che fare con la struttura modulare di un compilatore stesso. Un compilatore è un grosso programma, che per la sua complessità non può essere scritto agevolmente senza prevedere una certa divisione dei compiti. Così il primo blocco funzionale di un compilatore comporta il parsing di un linguaggio regolare.

Un compilatore o un interprete di espressioni, è un buon esempio che per schematizzare molte delle caratteristiche di compilatori di linguaggi più complicati. Come i suoi fratelli maggiori è costituito da due moduli: il primo, detto *analizzatore lessicale* o *scanner*, basandosi sulla specifica di un linguaggio regolare che contiene il linguaggio delle espressioni, controlla se l'espressione in input appartiene o no a questo linguaggio. In questa fase emergerebbero errori piuttosto banali, quali avere usato un carattere spurio come @ all'interno dell'espressione, cioè un terminale non previsto dal linguaggio, oppure avere usato per un numero decimale la virgola quando la sintassi prevede il punto, ma non emergerebbero errori più sottili, come essersi dimenticati di specificare un operatore come in $4(2 + 3)$ o aver dimenticato di chiudere una parentesi, quindi l'espressione $5 * (2 + (3 - 1)$ passerebbe come corretta, perché in effetti appartiene al linguaggio regolare che lo scanner riconosce. Passerebbe come corretta attraverso il solo scanner anche qualcosa del tipo $56 + *4 - -$. Sarà il parser a controllare la struttura grammaticale di una espressione e a individuare errori di questo tipo.

Perciò abbiamo due categorie di errori: gli errori lessicali, individuati dallo scanner e quelli sintattici, individuati dal parser. Questo non esaurisce tutti i possibili errori che si possono verificare. Così come in italiano una frase può essere sintatticamente corretta ma non avere senso compiuto, così pure possono esserci espressioni sintatticamente corrette, ma non avere significato. Ad es. 5/0 non è semanticamente corretta. Il nostro programma deve produrre un errore quando incontra una divisione per zero.

In poche parole prima riconosciamo un linguaggio più vasto di quello che in realtà vogliamo riconoscere e poi restringiamo il riconoscimento a quest'ultimo. Invece di una sola grammatica, dividiamo la grammatica delle espressioni in due parti, similmente a come abbiamo visto al paragrafo 1.4. Una delle due grammatiche è regolare ed è quella utilizzata nel processo di scanning della stringa di ingresso, mentre la seconda è non contestuale e viene utilizzata per il parsing vero e proprio.

Il compito dello scanner è di fornire al parser dei non terminali costituiti da più caratteri. Infatti mentre il punto di vista sull'input di uno scanner è a carattere, quello di un parser è a *token*, ossia a simboli non terminali costituiti dalla giustapposizione di più caratteri secondo le regole lessicali. Ad es. nel lessico delle espressioni sono token modificatori come (e), operatori costituiti da un solo carattere come +, *, da due caratteri come <=, numeri costituiti da cifre e altri caratteri secondo particolari notazioni come 4.56 o $10E - 5$ e identificatori di variabili, se il nostro risolutore di espressioni le supporta, come *AREA*, *Perimetro*.

Ora perché gli automi riconoscitori che sono stati introdotti al paragrafo 1.5 sono così importanti? Primo perché il loro formalismo è più semplice da usare sia per il riconoscimento che per la generazione delle grammatiche regolari corrispondenti. Basta partire dallo stato iniziale, seguire gli archi e ricordarsi man man in quale stato ci si trova. In altri termini, ecco un altro caso in cui un grafico val più di mille parole o mille formule. Secondo perché è molto semplice e addirittura automatizzabile, una volta disegnato l'automa o data la corrispondente tabella di transizione e l'indicazione dello stato iniziale e degli stati finali, o equivalente informazione sul linguaggio regolare in esame, tradurlo in un programma che, evolvendo la sua esecuzione attraverso gli stati dell'automa, esplicitamente o implicitamente, effettui il lavoro di riconoscimento del linguaggio regolare. In altri termini i concetti di stato e tabella di transizione sono facili da implementare. Tuttavia anche il parsing diretto della grammatica regolare non è meno difficile da implementare manualmente. In questo paragrafo imparerete a scrivere scanner ed useremo entrambi gli approcci, cioè sia con indicazione esplicita sia implicita degli stati e delle transizioni.

Quello che segue è il programma riconoscitore o analizzatore lessicale di numeri relativi basato sull'automa deterministico disegnato nel paragrafo 1.5 e scritto in linguaggio ANSI-C. Perché conviene usare l'automa deterministico l'abbiamo spiegato nel paragrafo 1.5: c'è la stessa differenza quando, per raggiungere una data destinazione, si può scegliere se percorrere una strada con segnali e indicazioni o se percorrere un labirinto. E' ovvio che è molto più facile percorrere una via certa anziché un labirinto! Scegliamo quindi l'automa deterministico per evitare di dover "percorrere un labirinto", cioè di dover usare il backtracking. Il backtracking rimane un'ottima tecnica per risolvere labirinti e problemi simili e vale la pena di conoscerla. Ma visto che possiamo, adesso la evitiamo in favore di una codifica molto più semplice.

```
/* semplice scanner per numeri relativi in ANSI-C */

#include <stdlib.h>
#include <stdio.h>

typedef enum {FALSE, TRUE} bool;

/* codici di errore */
typedef enum {NO_SIGN_OR_DIGIT, NO_DIGIT, EOF_UNEXP} err_t;

int c,        /* carattere in input */
  p=0;        /* posizione di colonna nell'input */
bool err=FALSE; /* stato di errore */

/* tipi di token */
typedef enum {END_OF_FILE, NUMBER} token_t;

typedef struct
{
  token_t type;
  union      /* unione estendibile */
  {
    int ival; /* valore intero */
  } u;
} token;

void readchar(void)
{
  c=getchar();
  if (c=='\n')
    p=0;
  else
    p++;
}
```

```
void initscan(void)
{
  readchar();
}

void printerror(char s[])
{
  fprintf(stderr, "%*sErrore: %s\n", p, "^", s);
}

void error(err_t cod)
{
  err=TRUE;    /* setta il flag di errore */

  switch(cod)
  {
    case NO_SIGN_OR_DIGIT:
      printerror("Atteso un segno o una cifra");
    break;

    case NO_DIGIT:
      printerror("Attesa una cifra");
    break;

    case EOF_UNEXP:
      printerror("Fine file inattesa");
    break;

    default:
      printerror("sconosciuto");
    break;
  }
}

bool scan(token *t)
{
  typedef enum {N_Z, N, E_N} stato;    /* definizione degli stati */
  stato s;     /* stato corrente */
  int sign;    /* segno di un numero */

  s=N_Z;       /* stato iniziale */

  err=FALSE;   /* resetta flag di errore */

  /* azione iniziale */
  sign=+1;     /* segno implicito */
  /******************/

  do
  {
    switch(s)
    {
      case N_Z:
```

```
switch(c)
{
  case ' ':
  case '\t':
  case '\n':
    /* azione */
    /* niente da fare, i bianchi si ignorano */
    /**********/

    /* transizione */
    /* niente da fare, permane nello stato N_Z */
    /***************/
  break;

  case '+':
    /* azione */
    /* niente da fare, sign vale già +1 */
    /**********/

    /* transizione */
    s=N;
    /***************/
  break;

  case '-':
    /* azione */
    sign=-1;
    /**********/

    /* transizione */
    s=N;
    /***************/
  break;

  case '0':
  case '1':
  case '2':
  case '3':
  case '4':
  case '5':
  case '6':
  case '7':
  case '8':
  case '9':
    /* azione */
    t->u.ival=c-'0';
    /**********/

    /* transizione */
    s=E_N;
    /***************/
  break;
```

```
    case EOF:
      /* azione finale */
      t->type=END_OF_FILE;
      /**********/

      /* fine scanning */
      return err;
    break;

    default:
      /* errore lessicale */
      error(NO_SIGN_OR_DIGIT);
    break;
  }
break;

case N:
  switch(c)
  {
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
      /* azione */
      t->u.ival=c-'0';
      /**********/

      /* transizione */
      s=E_N;
      /***************/
    break;

    case EOF:
      /* errore lessicale */
      error(EOF_UNEXP);

      /* azione finale */
      t->type=END_OF_FILE;
      /**********/

      /* fine scanning */
      return err;
    break;

    default:
      /* errore lessicale */
      error(NO_DIGIT);
    break;
```

```
        }
      break;

      case E_N:
        switch(c)
        {
          case '0':
          case '1':
          case '2':
          case '3':
          case '4':
          case '5':
          case '6':
          case '7':
          case '8':
          case '9':
            /* azione */
            t->u.ival=t->u.ival*10+(c-'0');
            /**********/

            /* transizione */
            /* niente da fare, permane nello stato E_N */
            /***************/
          break;

          default:
            /* azione finale */
            t->u.ival*=sign;
            t->type=NUMBER;
            /**********/

            /* fine scanning */
            return err;
        }
      break;
    }

    readchar();     /* legge prossimo carattere */

  } while (1);      /* ciclo infinito */
}

int main()
{
  token t;

  initscan();
  scan(&t);
  if (err) return EXIT_FAILURE;
  if (t.type==NUMBER)
    printf("num=%d\n", t.u.ival);

  return EXIT_SUCCESS;
}
```

Notate che il programma funziona leggendo un carattere in anticipo e poi decidendo in base alla tabella di transizione come elaborarlo, quindi leggendo il prossimo carattere, ecc... Il punto di vista sull'input è a carattere. Questo non penalizza eccessivamente le prestazioni in C perché la funzione *getchar*() è in realtà una macro-funzione, quindi viene evitato l'overhead del richiamo di una funzione per ogni carattere, e inoltre la libreria di I/O standard del C mantiene dei buffer, quindi la maggior parte delle chiamate di *getchar*() pescheranno caratteri dal buffer. Il buffer viene caricato a blocchi, ad esempio linee.

Di fronte agli errori ci si può comportare in vari modi: interrompere subito l'esecuzione stampando un messaggio, oppure fare dell'*error-recovery*, visualizzando un messaggio di errore esplicativo, ma continuando l'analisi, il cui risultato già si sa sarà negativo perché se una frase presenta errori lessicali non può far parte del linguaggio. In tal caso viene settata la variabile booleana globale *err* e la funzione *scan*() ne ritorna il valore *TRUE*.

Nel primo caso di può pensare che l'automa transiti in uno stato di errore da cui poi non esce più, pur continuando a masticare tutto l'input fino alla fine. Comunque non è necessario codificare esplicitamente tale stato, né scandire tutto l'input fino alla fine, basta interrompere l'esecuzione. Nell'esempio ho scelto di fare l'error-recovery, tanto per illustrare come si fa.

Inoltre per rendere più realistica l'analisi, ho associato delle azioni ad ogni transizione. C'è anche un'*azione iniziale* che viene eseguita quando si entra nello stato iniziale *N_Z* † e alcune *azioni finali* che vengono eseguite quando l'automa si trova su uno stato finale e il prossimo carattere in input non è su nessun arco che si diparte da questo stato, quindi poco prima che la funzione *scan*() ritorni.

Lo scopo di alcune di queste azioni è convertire la stringa in input in un numero nella rappresentazione binaria interna. Siccome le cifre arrivano dalla più significativa alla meno significativa, si utilizza per la conversione lo sviluppo di Horner, che permette di esprimere un qualsiasi numero positivo tramite sole somme e prodotti, senza potenze di 10 con esponente maggiore di 1. Ad es:

$$4321 = 4\ 10^3 + 3\ 10^2 + 2\ 10^1 + 1\ 10^0 = ((4\ 10 + 3)\ 10 + 2)\ 10 + 1$$

Il segno viene memorizzato sotto forma di fattore +1 o −1 nella variabile *sign* e applicato come azione finale con una moltiplicazione.

Questo analizzatore lessicale è molto semplice, l'unico tipo di token che riconosce è un numero (relativo). Tuttavia la logica degli analizzatori lessicali di espressioni, o di linguaggi come il C non è differente da questa. Provate ad estendere l'analizzatore in modo che gestisca numeri decimali oltre che numeri relativi e poi in modo che possa gestire espressioni aritmetiche. La struttura non cambia, ci sono solo più stati, più transizioni, più codici di errore e una informazione maggiormente strutturata che la funzione *scan*() deve passare al chiamante. Per controllare bene il funzionamento della nuova versione modificate il *main*() in modo che vengano effettuate chiamate ripetute alla funzione *scan*(), fino alla fine dell'input. Ad es. nel programma precedente la versione ciclica di main() è questa:

```
int main()
{
  token t;

  initscan();
  do
  {
    scan(&t);
    if (t.type==NUMBER)
      printf("num=%d\n", t.u.ival);
  } while(t.type!=END_OF_FILE);

  return EXIT_SUCCESS;
}
```

† Poichè *N-Z* significherebbe *N* meno *Z* in C, ho sostituito il trattino con l'underscore.

Sarò onesto. Dirò quello che molti libri non dicono. Si può sempre evitare di implementare direttamente un automa a stati finiti. Ad esempio lo scanning di un numero relativo può essere fatto con del codice guidato direttamente dalla sintassi, nel senso che la struttura del programma riflette direttamente il formato dell'input e così pure analizzatori sintattici più complicati, come quello per le espressioni aritmetiche o per un intero linguaggio di programmazione come il C. E' in questo modo che è stata implementata la funzione C *atoi*(3). Ecco l'analizzatore lessicale visto prima implementato in questo modo:

```
/* semplice scanner per numeri relativi in ANSI-C */

#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

typedef enum {FALSE, TRUE} bool;

/* codici di errore */
typedef enum {NO_SIGN_OR_DIGIT, NO_DIGIT, EOF_UNEXP} err_t;

int c,        /* carattere in input */
  p=0;        /* posizione di colonna nell'input */
bool err=FALSE, /* stato di errore */
  eof=FALSE;     /* incontrato EOF */

void printerror(char s[])
{
  fprintf(stderr, "%*sErrore: %s\n", p, "^", s);
}

void error(err_t cod)
{
  err=TRUE;   /* setta il flag di errore */

  switch(cod)
  {
    case NO_SIGN_OR_DIGIT:
      printerror("Atteso un segno o una cifra");
    break;

    case NO_DIGIT:
      printerror("Attesa una cifra");
    break;

    case EOF_UNEXP:
      printerror("Fine file inattesa");
    break;

    default:
      printerror("sconosciuto");
    break;
  }
}

void readchar(void)
{
  c=getchar();
```

```
  if (c=='\n')
    p=0;
  else
    p++;
}

void skipwhite(void)
{
  while (isspace(c))
    readchar();
}

void init(void)
{
  readchar();
  skipwhite();
}

int getnum(void)
{
  int num=0, sign=+1;

  err=FALSE;   /* resetta flag di errore */

  while (!isdigit(c) && c!='+' && c!='-')
  {
    if (c==EOF)
    {
      eof=TRUE;
      return num;
    }
    error(NO_SIGN_OR_DIGIT);
    readchar();
    skipwhite();
  }

  if (c=='+')
    readchar();
  else if (c=='-')
  {
    readchar();
    sign=-1;
  }

  while (!isdigit(c))
  {
    if (c==EOF)
    {
      error(EOF_UNEXP);
      eof=TRUE;
      return num;
    }
    error(NO_DIGIT);
    readchar();
```

```
  }

  do
  {
    num=num*10+(c-'0');
    readchar();
  }
  while (isdigit(c));

  skipwhite();

  return sign*num;
}

int main()
{
  int num;

  init();
  do
  {
    num=getnum();
    if (eof) break;
    printf("num=%d\n", num);
  }
  while (1);

  return EXIT_SUCCESS;
}
```

Dove sono finiti gli stati? L'indicazione di quale sia lo stato corrente c'è ed è implicita nella posizione del codice in cui ci troviamo. Ad esempio siamo nello stato iniziale N-Z durante il primo ciclo *while* della funzione *getnum*(), in cui stiamo facendo dell'error-recovery. Quando usciamo da questo ciclo, a parte il caso di *EOF*, se il prossimo carattere in input è un segno, lo gestiamo e andiamo implicitamente nello stato N. Da questo stato ci aspettiamo poi che venga in input una cifra. Se così non è il prossimo ciclo *while* farà dell'error-recovery, finché non otteniamo una cifra e transitiamo nello stato E-N, in cui permaniamo durante tutto il ciclo *do*. . . *while*. Se uscendo dallo stato N-Z, cioè dal primo *while* della funzione *getnum*(), il prossimo carattere in input è una cifra † entriamo nel ciclo *do*. . . *while*, ossia nello stato finale E-N.

La funzionalità è la stessa, il codice è meno prolisso e ugualmente estendibile. Per ogni tipo di token basta aggiungere una funzione che lo riconosca e lo ritorni. E sarebbe ancora molto più semplice se decidessimo di fermarci al primo errore senza fare dell'error-recovery. Ma allora perché abbiamo fatto tutta questa teoria? Il formalismo degli automi a stati finiti può essere comunque utile come specifica anche se usate questo metodo, quindi non è male esserne a conoscenza. Inoltre il metodo di implementazione diretta degli automi è l'unico che si presta ad essere utilizzato per la generazione automatica di analizzatori lessicali a partire dalla specifica degli automi a stati finiti o qualche altro formalismo equivalente, quindi è utile ed interessante conoscere anche questo.

Se provate a scrivere i due programmi precedenti senza fare dell'error-recovery vedrete che si semplificano e in particolare si semplica la versione con stati impliciti. L'error-recovery complica molto l'analisi lessicale e analogamente complica di molto anche l'analisi sintattica. Il testo [2] mostra come è semplice scrivere un compilatore a mano, senza usare tool quali Lex e Yacc, se ci si ferma al primo errore e si rinuncia all'error-recovery, e si usa la tecnica di parsing più semplice, e si procede per raffinamenti

† Non può essere altrimenti, vista la struttura della condizione del primo *while* della funzione *getnum*().

successivi e altri piccoli trucchetti. In particolare [2] mostra come sia semplice l'analisi lessicale con stati impliciti e senza error-recovery.

1.7. Espressioni Regolari

Oltre alle grammatiche regolari introdotte nel paragrafo 1.4 e agli automi a stati finiti del paragrafo 1.5, esiste ancora un altro formalismo per la descrizione dei linguaggi regolari. Si tratta delle cosiddette *espressioni regolari*.

Questo formalismo offre il notevole vantaggio della sinteticità. Introduciamolo attraverso degli esempi. Consideriamo il linguaggio dei numeri naturali. Come abbiamo visto la notazione più compatta per la sua grammatica che è regolare destra o sinistra è quella EBNF:

$$D \rightarrow 0|1|2|3|4|5|6|7|8|9$$
$$N \rightarrow D\{D\}$$

se introduciamo anche un operatore che permetta di esprimere la ripetizione 1 o più volte. Se ad esempio usiamo le parentesi angolari con questo significato possiamo scrivere un'unica regola molto sintetica:

$$N \rightarrow \langle 0|1|2|3|4|5|6|7|8|9 \rangle$$

similmente per i numeri relativi:

$$Z \rightarrow [+ \mid -]\langle 0|1|2|3|4|5|6|7|8|9 \rangle$$

Anche il lessico delle espressioni aritmetiche tra numeri relativi può essere espresso con una sola regola EBNF. Un token di una tale espressione è definibile in EBNF come segue:

$$T \rightarrow + \mid - \mid * \mid / \mid '(' \mid ')' \mid \langle 0|1|2|3|4|5|6|7|8|9 \rangle$$

In generale si può dimostrare formalmente che una grammatica regolare destra o regolare sinistra può sempre essere espressa tramite una sola regola di tipo EBNF, cioè senza utilizzare alcun simbolo non terminale, tranne quello iniziale, che a questo punto da solo non è più parte essenziale della notazione. Questo ci suggerisce di semplificare un po' la sintassi. Ad esempio nel caso dei numeri naturali, per prima cosa omettiamo il simbolo iniziale:

$$\langle 0|1|2|3|4|5|6|7|8|9 \rangle$$

invece di utilizzare diversi tipi di parentesi per raggruppare stringhe di terminali da ripetere, usiamo solo le parentesi tonde per fare i raggruppamenti e le faremo seguire dal simbolo * per indicare la ripetizione 0, 1 o più volte e + per indicare la ripetizione 1 o più volte:

$$(0|1|2|3|4|5|6|7|8|9)+$$

quella scritta sopra è l'espressione regolare che descrive un numero naturale. Manteniamo la stessa sintassi di EBNF per indicare l'opzionalità e abbiamo subito anche l'espressione regolare per i numeri relativi:

$$['+' \mid -](0|1|2|3|4|5|6|7|8|9)+$$

mentre quella che descrive il lessico delle espressioni aritmetiche, vale a dire l'insieme infinito di stringhe V_T della grammatica [1.3-2] è questa:

$$'+' \mid - \mid '*' \mid / \mid '(' \mid ')' \mid (0|1|2|3|4|5|6|7|8|9)+$$

notate che abbiamo dovuto quotare anche gli operatori + e *, visto che ora sono dei terminali del linguaggio delle espressioni regolari, e dei metasimboli rispetto al linguaggio delle espressioni aritmetiche. Questo è un caso sfortunato, poiché i due linguaggi, quello delle espressioni aritmetiche e quello delle espressioni regolari, usano molti simboli graficamente uguali come operatori, perciò abbiamo dovuto ricorrere spesso al quoting.

Dobbiamo stabilire completamente qual'è l'insieme dei simboli terminali delle espressioni regolari. Siccome vogliamo usarle per descrivere tutti i linguaggi regolari senza dover fornire ulteriori indicazioni su quale sia l'insieme dei simboli terminali, cioè vogliamo che sia sufficiente fornire solo l'espressione regolare stessa, stabiliamo che un terminale di un'espressione regolare è un singolo carattere. Questo implica ad esempio che nell'espressione regolare:

$$ab+$$

l'operatore +, che si riferisce sempre al solo terminale, si applica solo a b. Quindi il linguaggio generato è il seguente:

$$L = \{ab, abb, abbb, abbbb, \cdots\}$$

e *non* questo:

$$L = \{ab, abab, ababab, abababab, \cdots\}$$

Se volete generale quest'ultimo linguaggio, allora occorre usare le parentesi nell'espressione regolare corrispondente:

$$(ab)+$$

col significato che stavolta il + si riferisce all'intera espressione regolare che lo precede, racchiusa tra parentesi. Facciamo un po' di pratica con questa nuova sintassi, definendo qualche altro linguaggio regolare utile in pratica, piuttosto che insistere con esempi astratti come questo.

Supponete di voler definire il lessico † delle parole, per semplicità genericamente, nel senso che anche parole come *xyziggy* o 1*foImNqw* fanno parte del lessico. La nostra lasca definizione di parola è la seguente:

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

ossia una parola è per noi una qualsiasi successione delle 26 lettere dell'alfabeto latino (maiuscole o minuscole) e/o delle 10 cifre arabe, di lunghezza minima 1 carattere.

Come ulteriore esercizio consideriamo i primi dieci numeri romani:

arabi	romani
1	I
2	II
3	III
4	IV
5	V
6	VI
7	VII
8	VIII
9	IX
10	X

abbiamo considerato solo i primi 10 perché altrimenti l'esercizio sarebbe molto difficile. L'esercizio consiste nello scrivere una opportuna espressione regolare che generi il linguaggio dei primi 10 numeri romani (stringa vuota esclusa). Una soluzione banale è quella che consiste di 10 alternative:

$$I \mid II \mid III \mid IV \mid V \mid VI \mid VII \mid VIII \mid IX \mid X$$

un'altra soluzione, che fa un po' di fattorizzazioni, è questa:

$$I[I[I] \mid V \mid X] \mid V[I[I[I]]] \mid X$$

un altro dei tanti altri modi di fattorizzare:

$$[V][I][I]I \mid [I]X \mid [I]V$$

Così anche col formalismo delle espressioni regolari purtroppo ci sono vari modi spesso molto diversi di rappresentare lo stesso linguaggio. Ma ormai abbiamo imparato a convivere con questa complicazione.

Accenniamo al fatto che esiste un algoritmo per passare da espressione regolare ad automa a stati finiti, ma sfortunatamente genera automi a stati finiti non deterministici. L'automa generato può sempre

† Ricordate che con il termine *lessico* si intende un *linguaggio regolare*.

essere trasformato in uno deterministico equivalente, ma occorre applicare un altro complicato algoritmo.

Formalizziamo la sintassi delle espressioni regolari prima descritta a parole tramite EBNF:

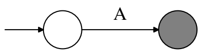
<espression> ::= <term> ["|" <term>]*
<term> ::= <factor> [<factor>]*
<factor> ::= [terminal character | (E)] [* | +] | "[" E "]"

Al paragrafo 1.3 abbiamo spiegato la sintassi del metalinguaggio EBNF. Qui, tanto per mostrare una variazione, abbiamo usato una sintassi leggermente diversa per scrivere le produzioni EBNF ma altrettanto comune.

A parole: una espressione regolare è formata da un termine, seguito eventualmente da altri termini separati tramite il metacarattere |. Un termine è formato dalla concatenazione di uno o più fattori, senza alcun operatore esplicito. Un fattore è un carattere terminale appartenente ad un certo alfabeto, ad es. l'ASCII, oppure un'altra espressione regolare racchiusa tra parentesi tonde. Ciascuno dei due *pu*ò essere seguito dal metacarattere * o +. Alternativamente un fattore può essere un'espressione regolare racchiusa tra parentesi quadre. Se *terminal character* è uno tra | ( ) [ ] * +, occorre un meccanismo di quoting, ad esempio farlo precedere da backslash (\). Se si opta per lo backslash, lo backslash stesso deve essere quotato, ossia scritto come \\.

Per esercizio proviamo a scrivere l'automa a stati finiti deterministico che corrisponde all'espressione regolare $(A|B)*A$. Invece di seguire l'algoritmo preciso, che non è spiegato in questo testo, procediamo in modo incrementale, usando un po' di quell'intuito umano di cui il buon Dio ci ha dotati e fors'anche un po' di fortuna nelle scelte che facciamo. Comunque se per caso siamo sfortunati e dovessimo fare delle scelte che non ci possano portare al risultato torniamo indietro e le cambiamo. Questo è quello che si dice procedere per tentativi, ma non senza princìpi. Cercheremo ad ogni passo di introdurre quanti meno stati possibili, una specie di principio di economia (degli stati) o principio del minimalista e inoltre di mantenere deterministico l'automa disegnato, altrimenti tutto il ragionamento non servirebbe a niente.

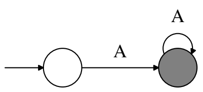
Se considerate una frase appartenente al linguaggio generato è ovvio che questa deve per forza terminare con una A. Perciò disegniamo dapprima lo stato finale e un arco etichettato con A che, partendo da uno stato di partenza, raggiunge lo stato finale:



Per semplicità non stiamo assegnando dei nomi agli stati, ma si intende che ogni stato ha un nome unico a piacere. Naturalmente dovrà esserci qualche altro stato, perché questo è solo l'automa corrispondente all'espressione regolare A, che è solo la parte finale dell'espressione regolare data $(A|B)*A$.

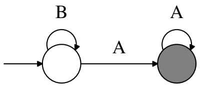
Ovviamente, poiché vogliamo che l'automa sia deterministico, non possiamo aggiungere un altro arco etichettato con A che vada dallo stato iniziale verso un nuovo nodo, perché avremmo due archi con la stessa etichetta che escono dallo stesso nodo (quello iniziale), quindi l'automa sarebbe non deterministico. Ad ogni passo cerchiamo di aggiungere archi e nodi in modo che la proprietà del determinismo rimanga! Inoltre in accordo col principio dell'economia, cerchiamo di non introdurre nuovi nodi se non ne abbiamo realmente bisogno!

Facciamo un passo avanti aggiungendo allo stato finale un *autoanello* etichettato con A, che non ci costa alcun nuovo stato:

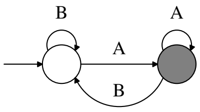


in questo modo abbiamo l'automa deterministico corrispondente all'espressione regolare $A*A$ o se preferite $AA*$ e ci siamo avvicinati di più al linguaggio definito da $(A|B)*A$. Sembra che stiamo facendo le scelte giuste. Adesso ci manca solo l'alternativa con B per la ripetizione! Abbiamo fatto la scelta di aggiungere questo autoanello dall'osservazione che se abbiamo incontrato una A e siamo nello stato finale, eventuali A successive in numero a piacere non guastano per il riconoscimento della frase come corretta.

Proviamo ad aggiungere un arco etichettato con B al nodo iniziale, perché una produzione di $(A|B)*A$ può anche iniziare con B. Dove faremo andare questo arco? Nel tentativo di usare quanti meno stati possibili lo facciamo andare nello stato iniziale stesso. Infatti se lo facciamo andare nello stato finale allora una produzione può terminare con B, mentre noi vogliamo che debba terminare per forza con A e non vogliamo introdurre nuovi stati. Così abbiamo questo automa deterministico:



che ancora non corrisponde a $(A|B)*A$, bensì a $B*A*A$ o $B*AA*$ se preferite. Pare che ci siamo avvicinati ancora un po' nella giusta direzione. Adesso notiamo che lo stato di partenza ha sia l'arco etichettato con A che quello etichettato con B, mentre lo stato finale ha solo l'arco etichettato con A. Allora proviamo ad aggiungere un arco etichettato con B allo stato finale e vediamo cosa esce fuori. Dove lo facciamo andare? Proviamo senza introdurre nuovi archi: possiamo renderlo un autoanello o farlo andare sullo stato iniziale. La prima ipotesi la scartiamo per evitare di ammettere produzioni che terminimo con B e quindi abbiamo il seguente automa deterministico:



non ci vuole tanto a convircersi che questo è l'automa che equivale all'espressione regolare $(A|B)*A$. E' anche chiaro che il nostro risultato utilizza il numero minimo di stati, perché con un solo stato non si può obbligare la produzione contemporaneamente a terminare con A e contenere anche il simbolo B.

Con la stessa tecnica cercate di ricavare l'automa a stati finiti corrispondente all'espressione regolare $(A|B)*AB$.

1.8. Il pattern matching

A voler essere minimalisti gli unici operatori strettamente necessari per poter definire tramite una espressione regolare qualsiasi linguaggio regolare sono solo tre. Uno è quello di concatenazione, che è implicito, nel senso che se ad esempio scriviamo *ab* intendiamo il carattere *a* seguito dal carattere *b* senza scrivere alcun operatore di concatenazione. Un'altro è quello di unione, ad es. l'espressione regolare $a|b$ indica l'unione del linguaggio costituito dalla sola stringa *a* con quello costituito dalla sola stringa *b*. Infine abbiamo l'operatore di chiusura, rappresentato con un asterisco. Ad esempio $a*$ indica la chiusura del linguaggio costituito dalla sola stringa *a*, che significa questo banale linguaggio infinito:

$$L = \{\varepsilon, a, aa, aaa, aaaa, \cdots\}$$

notate che la stringa vuota vi appartiene. L'operatore + in teoria non è strettamente necessario, perché può essere sostituito dall'operatore *, replicando due volte il terminale o espressione regolare tra parentesi che lo precede. Ecco qualche esempio:

$$a+ = aa*$$
$$(0|1|2|3|4|5|6|7|8|9)+ = (0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*$$
$$ab+ = abb*$$
$$(ab)+ = ab(ab)*$$

Anche l'operatore [ ] che esprime l'opzionalità della sottoespressione regolare contenuta non è strettamente necessario, potendo disporre delle parentesi e del metasimbolo di alternativa. Ad esempio:

$$['+' \mid -](0|1|2|3|4|5|6|7|8|9)+ = (\varepsilon \mid '+' \mid -)(0|1|2|3|4|5|6|7|8|9)+$$

Del resto anche le parentesi che si usano per la fattorizzazione non sono strettamente necessarie. Distribuendo la concatenazione si può ottenere sempre una espressione regolare priva di parentesi:

$$(\varepsilon \mid '+' \mid -)(0|1|2|3|4|5|6|7|8|9)+ = (0|1|2|3|4|5|6|7|8|9)+ \mid '+'(0|1|2|3|4|5|6|7|8|9)+ \mid -(0|1|2|3|4|5|6|7|8|9)+$$

Per semplicità non ho continuato l'espansione delle parentesi che racchiudono le alternative per le cifre, ma dovreste aver capito come fare.

E’ chiaro dagli esempi che questi operatori in più sono stati introdotti per ragioni di convenienza. Infatti permettono di compattare ulteriormente la notazione, rendendo il linguaggio più sintetico. Le implementazioni pratiche del formalismo delle espressioni regolari modificano ulteriormente la sintassi rispetto a quella vista finora, per far diventare le espressioni regolari ancora più sintetiche ed aggiungono anche delle funzionalità in più.

Cosa servono in pratica le espressioni regolari? Nel paragrafo 1.10 vedremo che esiste un generatore automatico di scanner che accetta la specifica del lessico tramite una espressione regolare. Questa espressione è costituita da una serie di sottoespressioni separate dall’operatore | e ogni volta che lo scanner rileva in input un token corrispondente ad una delle alternative, verrà eseguito un frammento di codice C specificato dal programmatore, che può compiere qualsiasi azione si desideri.

Un altro problema che è praticamente lo stesso dell’analisi sintattica, ma ha uno scopo leggermente diverso di quello di segmentare l’input in token, è quello del *pattern matching* che in italiano si traduce con *ricerca delle corrispondenze*.

Nella sua forma più semplice questo problema consiste banalmente nel trovare le occorrenze di una sottostringa in una stringa, lo stesso lavoro effettuato dalla funzione C *strstr*(3) oppure *strchr*(3) se ancora più banalmente la sottostringa è costituita da un solo carattere. Ma nella sua forma più generale il problema del *pattern matching* consiste nel trovare, se esistono, le occorrenze delle stringhe del linguaggio generato da una espressione regolare data all’interno della stringa di ingresso. L’espressione regolare viene chiamata in questo caso *pattern*. Quando viene trovata una sottostringa che appartiene al linguaggio generato dall’espressione regolare si dice che il pattern ha fatto corrispondenza. Usando una parola inglese si dice che ha fatto *match*.

La funzione di ricerca del vostro editor effettua del *pattern matching*. Se il vostro editor è abbastanza sofisticato dovrebbe accettare delle espressioni regolari nel campo di ricerca, in modo che possiate specificare delle ricerche di pattern più complicate di quelle costituite da soli terminali, usando operatori funzionalmente simili a quelli che abbiamo visto prima. Gli editor di testo *ed*(1) e *vi*(1) disponibili in tutti i sistemi UNIX accettano espressioni regolari la cui sintassi descriveremo nel prossimo paragrafo.

pattern matching and substitution cioè *ricerca e sostituzione delle corrispondenze* si riferisce invece alla funzionalità che utilizza il *pattern matching* per localizzare delle particolari sottostringhe e poi le sostituisce interamente con altre sottostringhe, formate da caratteri costanti eventualmente concatenati a sottostringhe della sottostringa stessa che è stata individuata.

Qual’è la differenza tra il *lexical scanning* e il *pattern matching* ? Sono praticamente la stessa cosa, cambia solo qualche dettaglio. Nel caso dell’analisi lessicale lo scopo è suddividere l’input in token. Si può fare dell’error-recovery saltando alcuni caratteri in input finché non si trova un carattere con cui inizia un token correttamente formato o si incontra la fine dell’input. Nel caso del pattern matching lo scopo è sempre quello di trovare dei token, ma i caratteri in input con cui nessun token può iniziare vengono semplicemente saltati senza che questo costituisca alcun errore. Lo scopo è indicare la posizione in cui inizia un dato token nella stringa di ingresso, oppure sostituire questo token con un’altro. Ci si può fermare alla prima occorrenza (se la si trova) o proseguire con la ricerca.

Se sapete scrivere un analizzatore lessicale, sapete anche come risolvere il problema del pattern matching, quando il pattern è una qualsiasi espressione regolare costante. Se l’espressione regolare è variabile, nel senso che viene fornita in input dall’utente insieme al testo da cercare, il problema si complica ed assomiglia a quello di scrivere un generatore di analizzatori lessicali, anziché un particolare analizzatore lessicale. Si può procedere così:

(1) Trasformare l’espressione regolare in input in un automa a stati finiti. Esiste un algoritmo per questo. In generale purtroppo se ne ottiene uno non deterministico.

(2) Trasformare l’automa non deterministico ottenuto al punto (1) in uno deterministico con un altro algoritmo.

(3) Interpretare l’automa deterministico ottenuto.

1.9. Espressioni Regolari di UNIX

Come abbiamo visto nel paragrafo precedente le espressioni regolari sono un metalinguaggio molto sintetico per la descrizione di linguaggi regolari.

Il concetto di espressione regolare è fondamentale nel sistema operativo UNIX, perché le espressioni regolari si ritrovano in molti utili programmi che fanno parte del sistema. Il punto di forza di UNIX è di fornire molti strumenti, fondamentalmente semplici e dedicati ad un unico task, e poi fornire dei linguaggi colla, come i linguaggi di shell, mediante i quali tali strumenti possano essere combinati insieme per costruirne altri che possano effettuare elaborazioni più sofisticate.

Questo non è un testo su UNIX, tuttavia descriverò la sintassi delle espressioni regolari di UNIX sia perché è utile imparare un po' di UNIX e sia perché la conoscenza di questa sintassi è un prerequisito per l'utilizzo del generatore di scanner Lex che descriveremo nel paragrafo successivo.

Lo scopo delle espressioni regolari in UNIX è sempre quello di specificare lo schema per la ricerca di corrispondenze o nel caso ad esempio di Lex, la struttura lessicale di un token di un linguaggio. Nell'editor di linea *ed*(1) risulta molto utile l'uso di pattern composti da espressioni regolari per ricercare le righe che corrispondono ad un determinato schema. Si può poi applicare a tale righe uno dei comandi di *ed*, per la semplice stampa, la cancellazione, la modifica o la sostituzione di pattern. Ormai quasi nessuno usa più l'editor di linea *ed*, perché esistono degli editor a video, detti in inglese *screen editor* che al contrario di *ed*, sfruttando le caratteristiche dei terminali moderni, permettono di controllare immediatamente le variazioni effettuate a video. Uno di questi editor è *vi*(1), che in una versione chiamata *vim*(1) è stato notevolmente migliorato da Bram Moolenaar e altri. *vi* sta per *visual interface*, intefaccia visuale su *ed*. E' quindi basato su *ed* e permette di passare in modalità *ed* per poter sfruttare le sue espressioni regolari o per eseguire lavori che *ed* fa meglio di ogni altro editor visuale.

Le espressioni regolari di *ed* sono usate anche in altri programmi più specializzati, come *grep*(1), che ricerca e stampa le linee di un testo che fanno match con un pattern, e *sed*(1) che è un efficiente editor di flusso, cioè un editor non interattivo. *sed* è in effetti un interprete di un semplice linguaggio di scripting per l'editing, per molti versi simile ad *ed* stesso.

In una espressione regolare di *ed* ogni carattere non speciale trova corrispondenza con sé stesso. Per eliminare l'eventuale significato speciale di un carattere, al posto delle virgolette si usa una sintassi più semplice che consiste nel farlo precedere dalla barra rovesciata (*backslash*), ad es. * invece di ′*′. Se il carattere non ha un significato speciale allora il backslash è pleonastico, sicché *a* equivale ad *a*. Questo implica che per togliere il significato speciale al backslash stesso occorre scrivere \\.

Il carattere . (punto) è speciale e trova corrispondenza con qualsiasi carattere. Questa estensione è molto utile in pratica perché vi evita, quando volete far corrispondenza con qualsiasi carattere, compresi quelli che andrebbero quotati, di dover listare come alternativa l'intero alfabeto dei caratteri.

Le parentesi tonde che racchiudono una sottoespressione regolare devono essere quotate. Le parentesi tonde non quotate sono considerate caratteri ordinari.

Per le alternative tra singoli caratteri esiste una sintassi molto sintetica: si possono inserire i caratteri tra parentesi quadre e le sequenze alfabetiche possono essere indicate tramite un trattino. Se il primo carattere della lista è ^ (caret), il risultato logico del sottomatch viene invertito.

L'asterisco indica la ripetizione 0, 1 o più volte di un carattere ordinario, del punto, della sequenza [ ] o della sottoespressione regolare tra parentesi che lo precedono.

Le espressioni regolari di UNIX non possono trovare corrispondenza con il segno di accapo. Questo le limita alla ricerca di pattern contenuti all'interno di una sola linea di testo.

Per chiarire meglio la sintassi, cerchiamo di riscrivere in UNIX tutte le espressioni regolari del paragrafo 1.7. Iniziamo con quella che descrive il linguaggio dei numeri naturali:

[0-9][0-9]*

estendiamo con un segno opzionale all'inizio in modo da descrivere il linguaggio dei numeri relativi:

[-+][0-9][0-9]*

questa regex tuttavia non consente di rappresentare anche i numeri privi di segno. Se provate un po' vedrete che con i soli operatori descritti prima non è possibile. Le "vecchie" espressioni regolari di UNIX, quelle della prima edizione di UNIX datata 1971, in realtà non sono vere e proprie espressioni regolari, ma descrivono linguaggi ancora più limitati! Per tale ragione tale forma limitata di espressione regolare è stata estesa con nuovi operatori nelle successive edizioni, su cui torneremo tra poco. Approssimativamente possiamo scrivere:

[-+]*[0-9][0-9]*

ma questa fa passare anche segni ripetuti come in $-++53$, cioè abbiamo descritto un linguaggio più lasco. Questa può non essere una differenza significativa nel problema del pattern matching a video. L'editor si posizionerà sul primo meno anziché sull'ultimo + di $-++53$, quindi un paio di caratteri prima del pattern che volevamo cercare. Se poi non ci sono mai frasi come $-++53$, non ci sono differenze ai fini del pattern matching e del pattern substitution.

L'espressione regolare di UNIX che equivale a $ab+$ secondo la sintassi del paragrafo 1.7 è:

*abb**

mentre quella che equivale a $(ab)+$ è:

ab\\(*ab*\\) *

invece questa è per la definizione di parole, data sempre nel paragrafo 1.7:

[0-9*A*-*Za*-*z*][0-9*A*-*Za*-*z*] *

Queste espressioni regolari limitate di UNIX hanno il vantaggio di poter usare un algoritmo di pattern matching compatto e non deterministico, ma naturalmente sono limitate come abbiamo visto. Le troviamo in *ed* già nella prima edizione di UNIX, quando non esistevano ancora programmi come *grep* e *sed*. Questi sono stati introdotti nelle edizioni successive. *grep* si basa sulle stesse espressioni regolari limitate di *ed*, ma poi ne venne introdotta una versione chiamata *egrep* † che supporta finalmente vere espressioni regolari, in grado di definire un qualsiasi linguaggio regolare di stringhe. Con *egrep* viene introdotto l'operatore + che indica una o più ricorrenze della sottoespressione regolare che lo precede, l'operatore "or" rappresentato con | e l'operatore ? che indica zero o una ricorrenza. Una differenza con le espressioni di *ed* è che adesso le parentesi tonde non devono essere più quotate quando raggruppano espressioni regolari. Così ecco gli esempi del paragrafo 1.7 secondo le espressioni di *egrep*:

[0-9]+
[−+]?[0-9]+
\\+ | − | * | / | \\(| \\) | [0-9]+
ab +
(*ab*) +
[0-9*A*-*Za*-*z*] +

Esiste poi uno standard più recente per le espressioni regolari, il POSIX 1003.2, che introduce parecchie estensioni e definisce due forme di espressioni regolari: quello "moderne" che sono simili a quelle di *egrep*, ma non esattamente e quelle obsolete che sono simili a quelle di *ed*, anche qui però purtroppo non esattamente. Se volete saperne di più consultate le pagine di manuale *regex*(5) o *regex*(7).

Il Perl non aderisce pienamente allo standard POSIX e supporta ancora un altro tipo di espressioni regolari, che offrono molte funzioni in più rispetto allo standard. Tra l'altro ogni programma tenta di introdurre estensioni cercando di mantenere la compatibilità col passato, come fanno le versioni moderne di *ed*. Quindi come vedete il panorama è abbastanza confuso: c'è uno standard, ma ci sono parecchie deviazioni ed estensioni. Per evitare rotture di testa con una espressione regolare che non si comporta come dovrebbe, consultate di volta in volta la documentazione del programma prima di usare le sue espressioni regolari, per vedere qual'è la sintassi e che differenze ci sono rispetto ad altri schemi di espressioni regolari che conoscete.

---

† *egrep* sta per *extended grep* cioè *grep esteso*.

1.10. Lex

Nel paragrafo 1.6 abbiamo accennato al fatto che esistono dei generatori automatici di scanner o analizzatori lessicali, quei programmi che riconoscono dei token o pattern lessicali in un testo che abbiamo imparato a scrivere manualmente nello stesso paragrafo. In questo paragrafo accenneremo brevemente all'uso di uno dei più diffusi programmi del genere, specie in ambiente UNIX, che si chiama Lex ed è stato scritto da Mike Lesk nei lontani anni '70. Questo paragrafo deve essere considerato una introduzione a Lex. Per approfondimenti consultate [3].

L'input di Lex è costituito principalmente da una sequenza di coppie, dette anche *regole*. Ogni coppia è costituita da una espressione regolare e da un frammento di codice C. L'output di Lex è un file sorgente C, che deve essere compilato e poi linkato con la libreria Lex ed eventualmente con altri file oggetto per poter produrre un programma eseguibile. Il sorgente C generato da Lex mette a disposizione una funzione analoga alla *scan* vista nel paragrafo 1.6, il cui prototipo è il seguente:

```
int yylex(void);
```

Questa funzione ricerca attraverso lo standard input le occorrenze delle espressioni regolari. Quando trova una corrispondenza esegue il codice C che fa coppia con l'espressione regolare. Se necessario l'input e l'output possono essere rediretti verso altre sorgenti.

La funzione yylex() generata da Lex può essere utilizzata da un parser scritto a mano, ad esempio con tecnica discendente ricorsiva, oppure da quello generato da tool come lo Yacc. In questo testo introduttivo non descriveremo Yacc e nemmeno la tecnica di parsing LALR † su cui questo si basa. Se siete interessati a Yacc consultate [4]. Esiste anche una versione di pubblico dominio di Yacc chiamata Byacc (Berkeley Yacc).

Per prendere confidenza col meccanismo di generazione del codice offerto da Lex, creiamo il programma Lex più semplice e proviamo a compilarlo. Il programma più semplice è quello vuoto, che non contiene alcuna regola e nessuna altra dichiarazione opzionale. Esso consiste del solo delimitatore %% che marca l'inizio delle regole seguito poi da nessuna regola. Inserite questa linea in un file chiamato ad esempio *minimum.l*:

```
%%
```

Di solito si assegna ai file sorgente di Lex l'estensione *.l*. Questa convenzione non è imposta da Lex, ma conviene aderire, per esempio per poter sfruttare il fatto che il programma *make* ha una regola implicita che converte un file *nome.l* nel file oggetto corrispondente con nome *nome.o*, cosa che noi ora faremo a mano per far vedere come si fa. Si fa così: si processa il sorgente tramite Lex, il quale produrrà un programma sorgente C abbastanza prolisso di nome *lex.yy.c*:

```
# lex minimum.l
# ls -s minimun.l lex.yy.c
  36 lex.yy.c     4 minimun.l
```

qui il simbolo # indica il prompt e naturalmente può essere diverso sul vostro sistema. Se Lex o la sua incarnazione GNU chiamata Flex non è installato, chiedete all'amministratore di sistema di aggiungerlo. Poi si può compilare il file *lex.yy.c* e linkarlo con la libreria Lex in un solo passo col comando:

```
# cc lex.yy.c -ll
```

se il linker si lamenta che non può trovare la libreria Lex, probabilmente state usando Flex. In tal caso sostituite *−ll* con *−lfl* e dovrebbe andare. In questo caso viene utilizzata una funzione *main*() che non fa altro che chiamare *yylex*():

† LALR sta per Looking Ahead Left to Right, operazione di analisi da sinistra a destra con conoscenza di un solo simbolo in avanti.

```
int main()
{
  yylex();
  return 0;
}
```

Eseguite il programma *a.out* prodotto:

```
# ./a.out
```

e notate che questo semplicemente copia lo stdin su stout, similmente al programma di sistema *cat*(1).

Come esempio più interessante, scriviamo con Lex un analizzatore sintattico per il linguaggio delle espressioni aritmetiche, per semplicità dapprima di soli numeri interi, di cui abbiamo parlato molto nei paragrafi precedenti. Questo analizzatore sarà pronto per essere interfacciato con un parser e del tutto indipendente dalla struttura di quest'ultimo.

Iniziamo con lo specificare che vogliamo saltare tutti i caratteri bianchi: accapi, spazi e tabulazioni. Create questo file *lex.l*:

```
%%
[ \t\n]+          ;
```

per maggiore visibilità abbiamo indicato la tabulazione con la sequenza di escape *\t*. La sequenza di escape *\n* indica l'accapo ed è l'unico modo per indicarlo: battere *invio* nel mezzo di una espressione regolare la farebbe terminare e ci sarebbe un errore sintattico. Uno o più spazi o tabulazioni devono separare l'espressione regolare dal frammento di programma che viene eseguito quando viene riconosciuta in input l'espressione. In tal caso l'azione è un'istruzione C singola vuota, che secondo la sintassi del C è costituita da un solo punto e virgola. Compilate ed eseguite come prima:

```
# lex lex.l
# cc lex.y.cc -ll
#./a.out
```

notate che ora tutto l'input viene riportato in output, tranne i bianchi. Infatti Lex salta e stampa tutti i caratteri, finché ne trova uno con cui può iniziare una corrispondenza con una espressione regolare. In generale Lex può essere utilizzato anche per produrre semplici trasformazioni o per fare analisi o statistiche su un testo a livello lessicale, ad esempio contare le parole e le linee, o per cercare tutte le parole che contengono *th*. Noi però vogliamo utilizzarlo come un analizzatore lessicale, cioè per segmentare l'input in preparazione ad una fase di parsing e scegliamo di non voler produrre niente in output. In tal caso dobbiamo fornire delle regole in modo che qualunque sia la combinazione di caratteri in input ci sia sempre corrispondenza. Solo così Lex non stamperà mai alcun carattere dell'input.

Nella colonna destra, quella del codice, naturalmente possiamo aggiungere dei commenti con i delimitatori /* e */ del C, come parte del codice che sarà ignorata dal compilatore C. Questo è utile per commentare le espressioni regolari, che possono essere molto complicate e quindi difficili da leggere. Aggiungiamo anche una regola che fa match con un numero senza segno, cioè una sequenza di 1 o più cifre. In tal caso l'azione corrispondente consiste nel convertire la stringa di cifre estratta dall'input, fornita da Lex stesso in un array di caratteri esterno alla funzione *yylex*() e chiamato *yytext*, in un numero nella rappresentazione binaria interna e memorizzarlo in una variabile globale, che ho scelto arbitrariamente di chiamare *yylval*.

Dopo aver riconosciuto un numero, la funzione *yylex*() non continuerà l'analisi come fa di solito, perché abbiamo inserito una istruzione return che fa uscire dalla funzione *yylex*(). L'analisi continuerà dove era stata sospesa alla prossima chiamata di *yylex*(), proprio come succedeva con la nostra *scan*(). L'intero ritornato è un codice da noi definito che indica il tipo di token ritornato. Nel caso invece degli operatori $+, -, *, /$ e le parentesi (,), *yylex*() ne ritorna direttamente il codice nell'alfabeto usato dal sistema operativo, comunemente l'ASCII. Mi sembrava infatti inutile introdurre dei codici anche per questi operatori monocarattere. Lo stesso per ogni altro carattere, che naturalmente costituisce un errore. Quando *yylex*() ritorna 0 vuol dire che ha incontrato la fine dell'input. Naturalmente bisogna fare attenzione che le costanti simboliche come *NUMBER* non abbiano alcun valore ASCII stampabile. Per evitare problemi diamo a questa ed eventuali altre costanti simboliche che introdurremo dei valori tutti strettamente maggiori di 255, cioè a partire da 256.

Se l'azione è una istruzione C composta, oppure volete scriverla su più linee, dovete racchiuderla tra parentesi graffe. Per semplicità racchiuderemo tutte le istruzioni tra parentesi graffe, completa quella vuota, così non possiamo mai avere problemi:

```
%%
[ \t\n]+        { ; }       /* salta bianchi */
[0-9]+          {           /* numero naturale */
                  yylval=atoi(yytext);
                  return NUMBER;
                }
.               { return yytext[0]; }      /* operatori, parentesi o carattere spurio */
```

per la conversione da stringa a numero ci siamo affidati alla funzione standard *atoi*(3) dichiarata in <stdlib.h>. Inoltre onde evitare problemi quando si compila *lex.yy.c* la variabile *yylval* deve essere definita o almeno dichiarata prima che sia usata nella funzione generata da Lex *yylex*(). Lo stesso vale per *NUMBER*. Inseriamo queste dichiarazioni in un file header *expr.h* che viene incluso in *lex.l* tramite la direttiva #include del preprocessore C. Questa direttiva deve essere racchiusa tra %{ e %}, che indicano a Lex di copiare così come sono le righe contenute in *lex.yy.c* prima della definizione della funzione *yylex*():

```
%{
#include "expr.h"
%}
%%
[ \t\n]+        { ; }       /* salta bianchi */
[0-9]+          {           /* numero naturale */
                  yylval=atoi(yytext);
                  return NUMBER;
                }
.               { return yytext[0]; }      /* operatori, parentesi o carattere spurio */
```

questo invece è *expr.h*:

```
unsigned int yylval;
#define NUMBER 256
```

Forniamo anche un programma principale *expr.c* che chiami ripetutamente yylex() fino alla fine dell'input, e in base al valore di ritorno capisca quale è il prossimo token in ingresso, proprio come fa un vero parser:

```
#include <stdio.h>
#include "expr.h"

int main()
{
  int tt; /* single character token or token type */
  int yylex(void); /* scan function provided by Lex */

  while ((tt=yylex())!=0)
  {
    switch (tt)
    {
      case NUMBER:
        printf("%u\n", yylval);
      break;

      case '+':
      case '-':
```

```
        case '*':
        case '/':
        case '(':
        case ')':
          printf("%c\n", tt);
        break;

        default:
          fprintf(stderr, "Error: '%c' unexpected\n", tt);
        break;
      }
    }
  return 0;
}
```

adesso la compilazione separata dei due moduli e il linking finale per produrre un eseguibile chiamato *expr* si svolge attraverso i seguenti passi:

```
# lex lex.l
# cc -o lex.o -c lex.yy.c
# cc -c expr.c
# cc -o expr expr.o lex.o -ll
```

siccome è noioso e inefficiente dover digitare tutti questi comandi o tenere traccia di quali file sono stati modificati e vengono ricompilati ci viene in aiuto l'utility make(1) [5]. Predisponendo il seguente file *makefile*:

```
OBJS = expr.o lex.o

expr:  $(OBJS)
       cc -o expr $(OBJS) -ll

expr.o lex.o:        expr.h

clean:
       @rm -f expr $(OBJS)

pr:
       @pr lex.l expr.h expr.c makefile
```

dopo ogni modifica per ricompilare solo quello che necessario e ri-linkare l'eseguibile *expr*, basta fare:

```
# make
```

mentre

```
# make clean
```

rimove l'eseguibile *expr* e i due file oggetto *expr.o* e *lex.o*. Un'altra regola non necessaria per la compilazione, ma comunque utile da richiamare come regola iniziale è *pr*. I due comandi:

```
# make pr | more
# make pr | lpr
```

stampano rispettivamente su video e su stampante tutti i sorgenti che fanno parte del programma effettuando una utile paginazione.

Ecco poi per esempio il risultato di una esecuzione:

```
# ./expr
10*(10+1)              input
10                     output
*
(
10
+
1
)

1+2/(3+4/(5+6))        input
1                      output
+
2
/
(
3
+
4
/
(
5
+
6
)
)

3 (4   +*              input (espressione sbagliata)
3                      output (nessun errore lessicale)
(
4
+
*

3th -2                 input (con errori lessicali)
3                      output (notare l'error-recovery)
Error: 't' unexpected
Error: 'h' unexpected
-
2
battere ^D
```

E’ molto facile ora estendere il nostro analizzatore. Supponete di voler supportare i numeri a virgola mobile o le variabili definite dall’utente. Per la prima cosa dobbiamo stabilire un formato per i decimali. Per semplicità limitiamoci alla notazione a punto fisso †. L’espressione regolare più semplice ma in grado di rappresentare tutti i numeri decimali, compresi naturalmente quelli interi, è questa:

```
[0-9]+\.?[0-9]*
```

vale a dire che un numero decimale è composto da una o più cifre, seguito eventualmente da un punto separatore e da zero, una o più cifre.

Per implementare questa estensione, è sufficiente editare solo la riga della tabella di *lex.l* che definisce la struttura lessicale dei numeri e le azioni corrispondenti da interprendere una volta che venga riconosciuta nell’input. La nuova versione di *lex.l* è la seguente:

† Per convenzione usiamo il punto invece della virgola come separatore della parte intera da quella decimale.

```
%{
#include "expr.h"
%}
%%
[ \t\n]+            { ; }  /* salta bianchi */
[0-9]+\.?[0-9]*     {      /* numero decimale */
                      yylval=atof(yytext);
                      return NUMBER;
                    }
.                   { return yytext[0]; }       /* operatori, parentesi o carattere spurio */
```

Notate che ora usiamo la funzione *atof*(3) invece di *atoi*(3). *atof*() converte una stringa in un numero decimale in doppia precisione (tipo *double*). Modifiche minori devono essere apportate agli altri file del progetto. Nel file *expr.h* il tipo di *yylval* deve essere cambiato da *unsigned int* a *double*:

```
double yylval;
#define NUMBER 256
```

e in *expr.c* la stringa di formato della *printf*(3) deve essere sostituita per stampare correttamente un *double*. Riporto solo la linea modificata:

```
printf("%lf\n", yylval);
```

quindi fate un bel

```
# make
cc -Os -mcpu=i386 -fno-strength-reduce   -c -o expr.o expr.c
lex  -t lex.l > lex.c
cc -Os -mcpu=i386 -fno-strength-reduce   -c -o lex.o lex.c
cc -o expr expr.o lex.o -lfl
rm lex.c
```

e poi riprovate lo scanner. Notate che la sintassi permette di utilizzare il punto senza che sia seguito da nessuna cifra, ad es. 5. è lo stesso che 5. 0. Se invece avessimo scritto l'espressione regolare così:

```
[0-9]+(\.[0-9]+)?
```

questo non sarebbe stato possibile. Si potrebbe complicare ulteriormente la sintassi prevedendo la possibilità di omettere lo zero prima del punto come in . 5 invece che 0. 5 o introducendo la notazione esponenziale supportata da *atof*(3). Lasciato per esercizio.

Per quanto riguarda l'introduzione delle variabili, occorre stabilire la regola con la quale sono formati i loro nomi, spesso chiamati genericamente *identificatori*. A questo punto da quali caratteri potrebbe essere costituito un identificatore, eccetto naturalmente i bianchi? Gli operatori, le parentesi, le cifre e il punto che costituiscono i numeri dobbiamo includerli o escluderli nella definizione di identificatore?

Gli operatori e le parentesi sicuramente dobbiamo escluderli, perché altrimenti non li riconosceremmo mai in quanto tali. Se ad es. ho *PIPPO* + 3, se includo il + come carattere di identificatore, vedo l'identificatore *PIPPO* + seguito dal numero 3, che chiaramente è sbagliato perché ho fatto fuori un operatore, nel senso che ho proprio tolto l'operatore dal lessico.

Ma davvero se un identificatore inizia con una cifra non riusciamo con un automa a stati finiti deterministico a distinguerlo da una variabile? Falso, perché possiamo transitare dallo stato finale di numero allo stato finale di identificatore non appena incontriamo qualche altro carattere diverso da una cifra che può stare in un identificatore.

La maggior parte dei linguaggi prevede che gli identificatori non debbano iniziare con un numero, e limita molto le specie di caratteri che possono stare in un identificatore, di solito alle sole lettere maiuscole e minuscole, numeri e pochi altri caratteri come underscore (_). Anche se non comporta problemi di ambiguità, non piace avere identificatori tipo 3*CANI* e 999999*gatti* perché iniziando con un numero si confondono con le altre costanti numeriche presenti in un programma. Non si permette all'utente di usare molti caratteri di punteggiatura e separatori negli identificatori, perché, anche se non sono usati come operatori e separatori per altri scopi, rendono gli identificatori meno leggibili e se si decide di usarli come operatori in futuro tutti gli identificatori dei programmi esistenti dovranno essere cambiati.

Ci sono quindi ottime ragioni pratiche per introdurre queste restrizioni, tuttavia tanto per far vedere quali restrizioni non sono strettamente necessarie riscriviamo la grammatica in questo modo:

```
%{
#include "expr.h"
%}
%%
[ \t\n]+        { ; }           /* salta bianchi */
[0-9]+\.?[0-9]*     {       /* numero decimale */
                    yylval=atof(yytext);
                    return NUMBER;
                  }
[^-+*/()]+    { puts(yytext); }          /* identificatore */
.             { return yytext[0]; }       /* operatori, parentesi */
```

così posso usare identificatori che iniziano con una o più cifre o altri caratteri non altrimenti utilizzati come operatori e separatori! Lex infatti nelle situazioni di ambiguità in cui più di una regola può essere impiegata seleziona la regola lessicale che corrisponde alla stringa più lunga o, a parità di lunghezza, la regola che è definita per prima. Così ad es. 3*CANI* viene riconosciuto correttamente come identificatore perché la regola che riconosce il 3 come numero consuma solo 1 carattere, mentre quella per gli identificatori ne consuma 5 e quindi vince sulla prima.

Notare che non è necessario leggere due caratteri successivi: all'inizio un analizzatore potrebbe pensare che si tratti di un numero 3, ma poi può transitare nello stato di identificatore sapendo solo che il successivo carattere è una *C*. Notate inoltre che se invertissi l'ordine delle due regole che riconoscono numeri e identificatori, ogni numero verrebbe riconosciuto come identificatore, il che sarebbe sbagliato.

Per le ragioni dette prima, e per evitare che ogni carattere spurio venga considerato come parte di un identificatore, è preferibile introdurre restrizioni sul lessico degli identificatori. Una espressione regolare tipicamente usata che fa match con un identificatore è:

```
[a-zA-Z_][a-zA-Z0-9_]*
```

2. Alcune risorse gratuite sulla rete

[1] The M4 Macro Processor, Brian W. Kernighan and Dennis M. Ritchie, 1977

[2] LET'S BUILD A COMPILER!, Jack W. Crenshaw, 1988

[3] Lex - A Lexical Analyzer Generator, M. E. Lesk and E. Schmidt, 1975

[4] Yacc: Yet Another Compiler-Compiler, Stephen C. Johnson, 1975

[5] Make - A program for Maintaining Computer Programs, S. I. Feldman, 1979

[6] PARSING TECHNIQUES - A Practical Guide, Dick Grune and Ceriel Jacobs, 1990