Linguaggi

Un dizionario definisce il linguaggio come *"l'insieme di parole e metodi di combinazione di parole usate e comprese da una comunità di persone"*. Tuttavia questa definizione non è sufficientemente precisa per un linguaggio di programmazione, in quanto non consente di caratterizzare i linguaggi in modo da superare le ambiguità tipiche dei linguaggi naturali o in modo da comprendere cosa voglia dire denotare processi computazionali meccanizzabili e stabilire proprietà su cui poter effettuare ragionamenti. Occorre pertanto caratterizzare il linguaggio come un particolare sistema matematico che consenta di dare risposta a domande come:

1. Quali sono le frasi lecite di un linguaggio?
2. Data una frase, è possibile stabilire se essa appartiene a un linguaggio?
3. Come si stabilisce il significato di una frase?
4. Quali elementi linguistici primitivi occorrono per poter esprimere la soluzione a un qualunque problema (risolvibile)?

Sintassi e semantica

La descrizione dei linguaggi di programmazione viene effettuata introducendo notazioni formali diverse per la descrizione della struttura delle frasi (**sintassi**) e per la descrizione del significato di una frase (**semantica**).

La struttura delle frasi lecite di un linguaggio può essere descritta attraverso un dispositivo formale denominato **grammatica**.

Una **Grammatica** è definita da una quadrupla di enti (V_N, V_T, S, P) ove:

- V_N = insieme di *meta-simboli* detti anche simboli *non- terminali* o *variabili*. I meta-simboli rappresentano categorie sintattiche.
- V_T = insieme dei *simboli terminali*.
  - Sono stringhe su un alfabeto A, definito come un insieme finito e non vuoto di simboli atomici.
  - Il simbolo A^* denota la *chiusura* dell'alfabeto, cioè tutte le stringhe che si possono comporre con esso: $A^* = A_0 \cup A_1 \cup A_2 \ldots$
  - Il simbolo A^+ denota la *chiusura positiva* dell'alfabeto, che non comprende la stringa vuota (di solito denotata con #); ergo, $A^+ = A^* - \#$

    - Gli insiemi VT e VN devono essere *disgiunti*, cioè la loro intersezione deve essere vuota (∅).
- L'insieme V = VT U VN si dice *vocabolario* della grammatica.
- S = *scopo* della grammatica: è un simbolo particolare in VN, detto anche *start-symbol*.
- P = insieme finito di *produzioni*, ossia regole di riscrittura, della forma a ::= b , con a in V+, b in V* .

Una grammatica stabilisce le regole di un "gioco", le cui mosse sono costituite dalle produzioni: tutte le "partite" hanno origine dallo scopo S e consistono nell'applicare le produzioni per generare le frasi del linguaggio che la grammatica descrive.

Iniziando dallo scopo S ed applicando una o più regole di produzione, si ottengono via via diverse *riscritture*, dette *forme di frase (sentential forms)*. La sequenza di passi necessari per produrre una certa forma di frase a partire dallo scopo S si chiama *sequenza di derivazione*. Una forma di frase costituita *di soli simboli terminali* si dice *frase* del linguaggio.

Si dice Linguaggio *L(G)* generato dalla grammatica G l'insieme delle frasi formate da soli simboli terminali raggiungibili dallo scopo S di G tramite una sequenza di derivazione.

Un requisito essenziale è che la grammatica di un linguaggio sia definita in modo da poter risolvere il problema della *decidibilità* del linguaggio, ossia il problema di decidere se una certa stringa sia o meno una frase che appartiene al linguaggio.

Sul piano pragmatico, un linguaggio artificiale è effettivamente utilizzabile solo nel caso in cui possa esista un *automa riconoscitore* computazionalmente efficiente. Questo problema, essenziale per la costruzione di traduttori ed interpreti, è oggi risolto attraverso l'uso di grammatiche generative libere da contesto (si veda la sezione *"La classificazione di Chomsky delle grammatiche"*).

Pur se meno diffusi per via della loro relativa maggior complessità, sono disponibili anche vari metodi formali per la descrizione della *semantica* di un linguaggio, che stabilisce un secondo, più interessante, rapporto con il concetto di automa. Il rapporto tra la semantica di un linguaggio di programmazione e l'idea di *automa per la risoluzione di problemi* è a sua volta collegato al quesito fondazionale dell'informatica: *quali elementi occorre introdurre per denotare processi computazionali meccanizzabili e quali proprietà di conseguenza possono essere associate a tali processi in virtù dei sistemi di denotazione adottati?*

La risposta a questo quesito trova le sue radici nella logica matematica ed è un risultato di studi compiuti a partire dagli anni 1920-30. Per il moderno ingegnere del software questa parte della storia dell'informatica costituisce un ottimo esempio del rapporto tra analisi, progetto e realizzazione nel contesto di un'impresa che mirava a costruire una "macchina" capace di costituire un *elaboratore "universale"* di informazione.

La notazione BNF

La notazione BNF (*Bakus-Naur Form*) prende il nome da coloro che la proposero per descrivere la sintassi del linguaggio *Algol60*, che può essere considerato il progenitore di tutta la famiglia dei linguaggi imperativi di tipo *Pascal*. Rispetto alle notazione precedenti si introducono le seguenti nuove convenzioni:

- i simboli non terminali sono racchiusi tra le parentesi `< >` oppure rappresentati da simboli con lettere maiuscole;
- per semplificare la scrittura di più produzioni per uno stesso non-terminale si introduce il meta-simbolo `|` per indicare una alternativa.

Ad esempio, le produzioni:

```
<letter> ::= A
<letter> ::= B
<letter> ::= C
```

si possono compattare in quella che segue:

```
<letter> ::= A | B | C
```

Frasi ricorsive come:

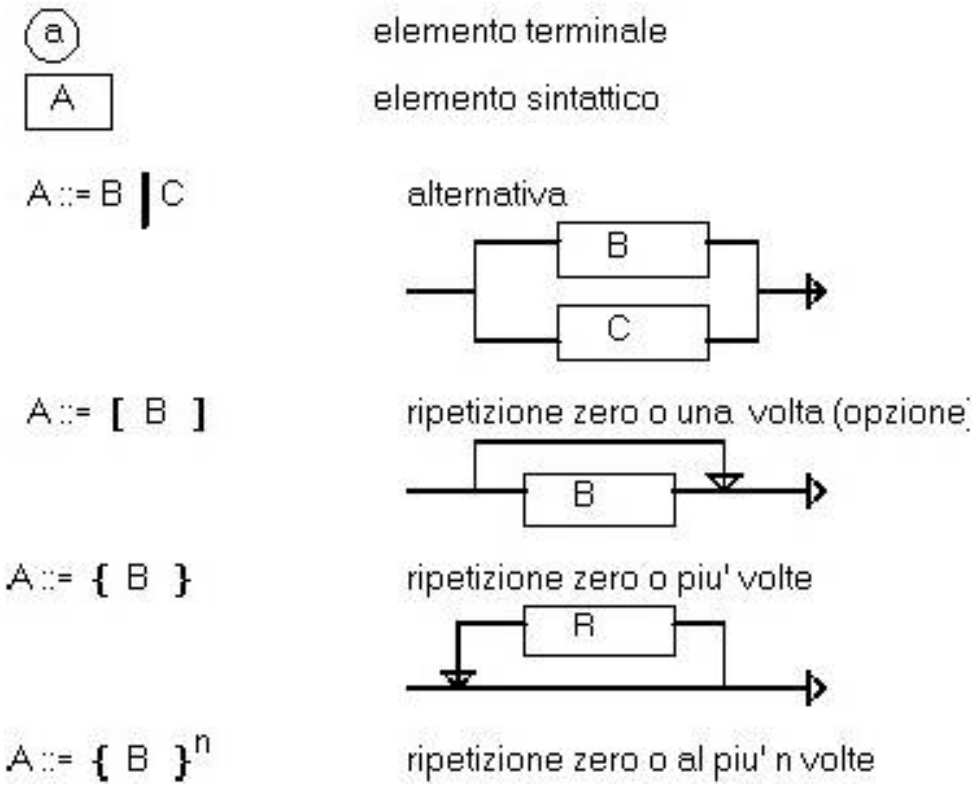
```
<alfa> ::= <letter> | <alfa> <letter>
```

possono essere riscritte includendo tra `{ }` le forme ripetibili zero o più volte:

```
<alfa> ::= <letter> { <letter> }
```

Per rendere più facilmente leggibili le regole di produzione, si fa inoltre uso di una rappresentazione grafica delle produzioni, detta diagramma sintattico, come riassunto nella figura:

Figura2. Notazioni BNF



Complessivamente, queste notazioni costituiscono un meta-linguaggio con cui descrivere le regole di produzione.

Esempio (identificatori).

E' molto comune che i linguaggi di alto livello introducano identificatori per denotare i vari oggetti del proprio dominio, nel modo che segue:

```
<letter> ::= A | B | ... | Z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<ident> ::= <letter> | <ident> <letter> | <ident> <digit>
```

Questa sintassi ammette identificatori di qualunque lunghezza. Per stabilire un limite massimo di 10 simboli significativi (come facevano un tempo molti compilatori ed interpreti) si può scrivere:

```
<ident> ::= <letter> | <letter> { <idrest> }9
<idrest> ::= <letter> | <digit>
```

La classificazione di Chomsky delle grammatiche

Nel formalismo delle produzioni grammaticali il parametro più importante è costituito dalla froma delle produzioni. *Noam Chomsky*

ha introdotto una classificazioe ormai classica articolata su 4 livelli: 0,1,2,3.

Le grammatiche di tipo 3, (dette *regolari*i) sono basate su produzioni del tipo (con `A, B in VN`, `a in VT`):

```
A ::= aB
A ::= a
```

dette *lineari a destra* oppure

```
A ::= Ba
A ::= a
```

dette *lineari a sinistra*.

I linguaggi di tipo 3 (regolari) sono i linguaggi generati da grammatiche di tipo 3. Ad esempio, il linguaggio $\{a^n b \mid n>=0\}$ è di tipo 3 in quanto può essere generato dalle produzioni:

```
S ::= aS
S ::= b
```

Le grammatiche di tipo 2 (dette *context free* o libere da contesto) sono basate su produzioni del tipo (`A in VN`, `alfa in V+`):

```
A ::= alfa
```

I linguaggi di tipo 2 (context free o non contestuali) sono i linguaggi generati da grammatiche di tipo 2. Ad esempio, il linguaggio $\{a^n b^n \mid n>=1\}$ è di tipo 2, infatti può essere generato dalla grammatica con le produzioni:

```
S ::= aSb
S ::= ab
```

Le grammatiche di tipo 1, (dette *context sensitive* o contestuali) sono basate su produzioni del tipo (con `A in VN`, `x,y in V*` , `alfa,beta in V+`, `alfa != ∅`):

```
x A y :== x alfa y
```

oppure

```
alfa :== beta con |alfa| <= |beta|
```

ove `|s|` denota la lunghezza della stringa `s`.

I linguaggi di tipo 1 (context sensitive o contestuali) sono i linguaggi generati da grammatiche di tipo 1. Ad esempio, il linguaggio $\{a^n b^n c^n \mid n>=1\}$ è di tipo 2

```
S  ::= aSBC | aBC
CB ::= BC
aB ::= ab
bB ::= bb
bC ::= bc
cC ::= cc
```

Le grammatiche di tipo 0, sono basate sulle produzioni più generali, del tipo (con `alfa in V+` , `beta in V*`):

```
alfa :== beta
```

Le grammatiche di tipo 0 ammettono derivazioni senza alcuna restrizione, tra cui derivazioni che accorciano stringhe.

Relazioni tra i tipi di grammatiche

I diversi tipi di grammatiche sono in relazione gerarchica tra loro. In particoare ogni grammatica regolare (*regular*) è anche libera da contesto (*context-free*), ogni grammatica context-free è anche dipendente da contesto (*context-sensitive*) e ogni grammatica context-sensitive è anche di tipo 0.

Un linguaggio `L(G)` si dice context-sensitive, context-free e regular quando può essere generato rispettivamente da una grammatica G context-sensitive, context-free o regular. Si dice lineare quando può essere generato da una grammatica lineare.

Il fatto che una grammatica `G` generi un linguaggio, non significa che questo sia necessariamente dello stesso tipo della grammatica.

Un linguaggio regolare può essere generato da grammatiche anche di tipo 2,1,0. Ad esempio, il linguaggio context-free `{anbn | n>=1}` può anche essere generato dalle produzioni di tipo 1:

```
S ::= aSBC                 CB ::= BC
SB ::= bF                  FB ::= bF
FC ::= cG                  GC ::= cG
G ::= #
```

Due grammatiche che generano lo stesso linguaggio possono essere definite equivalenti. Va però subito detto che una grammatica può essere preferibile ad un'altra dal punto di vista della analisi sintattica e del processo di compilazione. Il concetto di equivalenza tra grammatiche va dunque considerato con attenzione.

La stringa vuota

La stringa vuota non fa parte dei linguaggi generati dalle produzioni introdotte in precedenza. Quelle produzioni possono però essere

estese ammettendo produzioni della forma:

```
S  ::= #
```

a patto che S sia lo scopo della grammatica e che S non compaia nella parte destra di alcuna produzione.

In questo modo la produzione precedente può essere usata solo al primo passo di una derivazione e le stringhe non possono più accorciarsi. Si può dimostrare (si veda [Hopcroft-Ullman]) che se L è un linguaggio di tipo 0,1,2 o 3, allora anche L U {} e L - {} sono dello stesso tipo. Ad esempio, le produzioni:

```
S1 ::= #
S1 ::= a S b
S ::= a S b
S ::= a b
```

definiscono il linguaggio context-free L5 U {} cioè { anbn | n >= 0 }.

Sempre in [Hopcroft-Ullman] si può trovare la dimostrazione al seguente teorema.

Teorema.

Se G =<VN,VT,S,P> è una grammatica context-free in cui ogni produzione è della forma: A ::= alfa, con alfa in V* (alfa può essere #) allora esiste una grammatica context-free G1 che genera lo stesso linguaggio L(G) in cui ogni produzione è della forma

A ::= alfa, con alfa in V+

oppure S ::= #

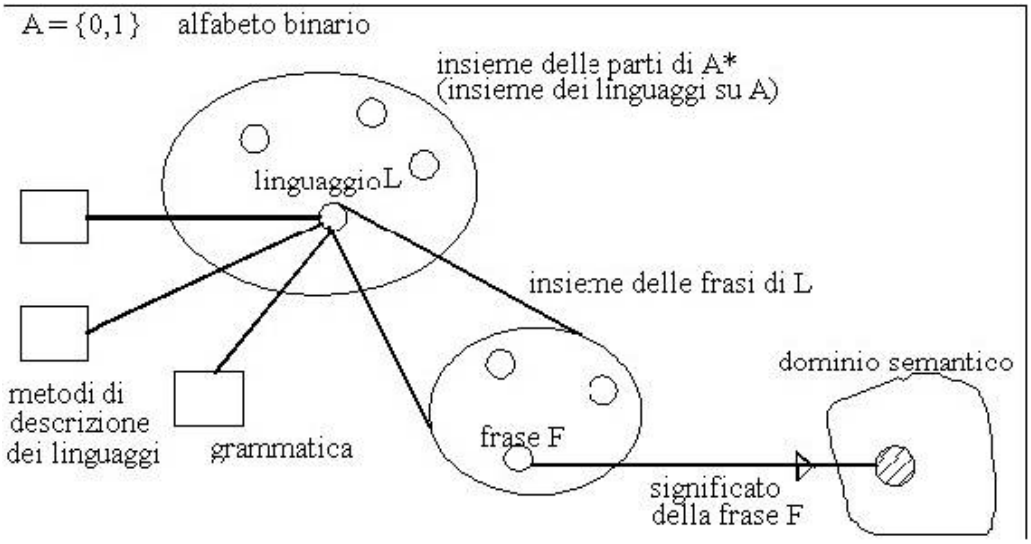
ed S non compare nella parte destra di alcuna produzione.

Il teorema assicura che l'unica differenza di una grammatica context-free con #-rules rispetto ad una senza tali produzioni, è che la prima può generare un linguaggio che include la stringa vuota.

Il problema del riconoscimento di un linguaggio

La figura che segue riassume in modo schematico il modo con si caratterizza il concetto di linguaggio.

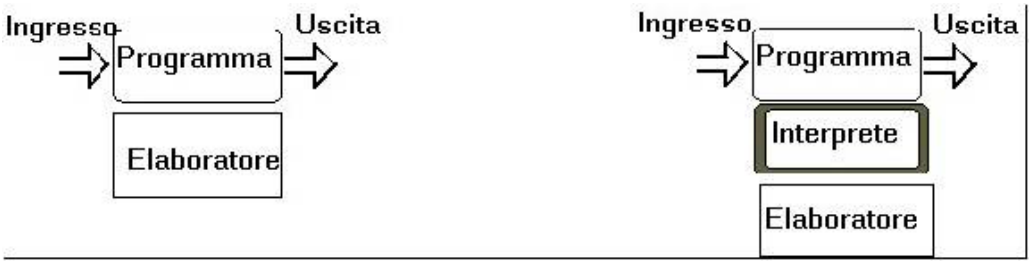
Figura3. Linguaggi



Interpreti

Un interprete per il linguaggio L è un programma che accetta come ingresso le frasi di L, eseguendole una alla volta. L'uscita di un interprete è quindi la valutazione di una frase di L.

Figura4. Interpreti



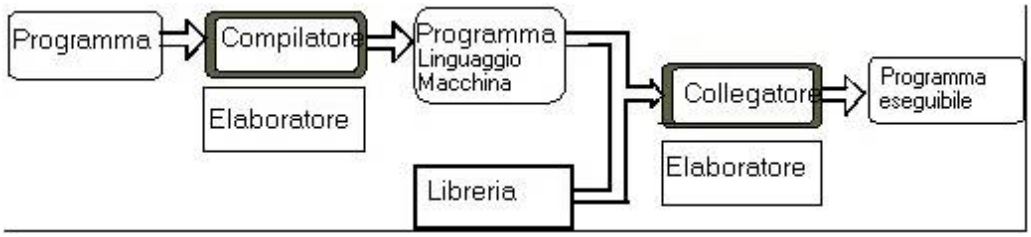
Compilatori

Il compilatore per un linguaggio L è un programma che accetta come ingresso un programma scritto in L. L'uscita del compilatore è una riscrittura dell'intero programma in un altro linguaggio (di solito nel linguaggio macchina di uno specifico elaboratore) se questo risulta sintatticamente e semanticamente corretto.

Normalmente un linguaggio di alto livello consente di esprimere una collezione di operazioni già rese disponibili dall'elaboratore o dal

sistema operativo. Il complesso di queste operazioni viene di solito incluso in una libreria di programmi specifica per il particolare elaboratore usato, che il linguaggio in qualche modo "eredita" attraverso il suo collegamento col programma tradotto.

Figura5. Compilatori



Analisi di un linaggio

L'analisi lessicale

L'analisi lessicale consiste nella individuazione delle parole individuali (dette in gerco tecnico **token**) che compongono una frase. I token che possono comparire nelle frasi di un linguaggio vengono stabiliti dalle regole grammaticali che descrivono il linguaggio.

Un *analizzatore lessicale* è un componente che, data una frase espressa da una sequenza di caratteri, restituisce la sequenza ordinata dei token rappresentati dai nomi, parole-chiave, simboli di punteggiatura, etc. che compaiono in quella frase.

L'analisi sintattica

L'analisi sintattica consiste nella verifica che una frase può essere costruita in base alle regole grammaticali che descrivono il linguaggio. Un analizzatore sintattico è un componente che, data la sequenza di token prodotti dall'analizzatore lessicale, produce di solito una rappresentazione interna della frase, in forma di albero.

L'analisi semantica

L'analisi semantica consiste nel "calcolo" del significato del linguaggio. Un analizzatore semantico è un componente che, data la rappresentazione intermedia prodotta dall'analizzatore sintattico, controlla la corenza logica interna del programma (ad esempio se le variabili sono usate dopo essere state definite, se sono rispettate le

regole di compatibilità in tipo, etc). Nel caso di un compilatore, l'analizzatore semantico può trasformare la rappresentazione delle frasi in modo più adatto per la generazione di codice.

Sequenza di derivazione

Si dice **sequenza di derivazione** la sequenza di passi necessari per produrre una forma di frase `s` a partire dallo scopo `S` di una grammatica `G` mediante applicazione di una o più regole di produzione. Si usano le seguenti notazioni:

`S => s`: la forma di frase `s` deriva da `S` in una sola applicazione di produzioni (un passo)

`S =+> s`: la forma di frase `s` deriva da `S` in una o più applicazioni di produzioni

`S =*> s` : la forma di frase `s` deriva da `S` applicando zero o più applicazioni di produzioni

Si dice **frase** una forma di frase costituita da soli simboli terminali.

Si dice **Linguaggio** `L(G)` generato dalla grammatica `G` l'insieme delle frasi `s` tali che: `s in VT*` e `S =*> s`

Ad esempio,il linguaggio `L5 = { anbn | n > 0 }` può essere descritto dalla grammatica `G5=(VN, VT, S, P)` con `VT={a,b}`, `VN={F}`, `S=F` e le regole di produzione:

```
F ::= a F b
F ::= a b
```

La prima regola stabilisce che il meta-simbolo `F` (scopo della grammatica) può essere riscritto come la frase `aFb`. La applicazione di questa produzione consente la effettuazione di un ulteriore passo generativo, a causa della presenza di `F` nella nuova frase. La seconda regola stabilisce una alternativa di riscrittura di `F`, scegliendo la quale non si provoca (nè si consente) altra generazione. Ogni stringa prodotta applicando le regole precedenti che non contenga il meta-simbolo `F` è una frase del linguaggio generato dalla grammatica.

Le regole di produzione sono frasi di tipo dichiarativo che esprimono che `L5` è costitituito da frasi composte da un ugual numero (al minimo 1) di caratteri `a` e `b`, in cui tutti i caratteri `a` precedono i caratteri `b`, fornendo contemporaneamente un modo per generare queste frasi. La sequenza:

```
F => aFb => aaFbb => aaabbb
```

costituisce la sequenza di derivazione della frase `aaabbb` di `L5`.

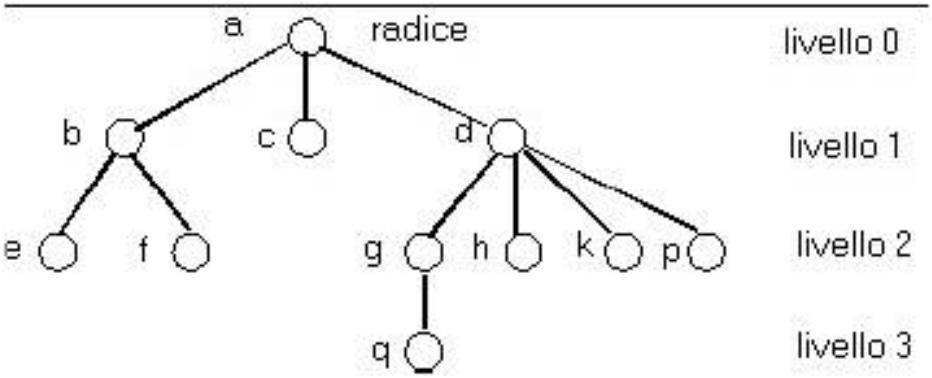
Alberi di derivazione per grammatiche context free

Per descrivere una derivazione è molto utile usare una forma particolare di grafo aciclico, detta *albero*.

Sia `A` un insieme finito e `T` una relazione su `A`. `T` si dice *albero* se esiste un unico elemento `r0` (detto *radice*) per cui vi è un unico cammino (collezione di nodi connessi) in `T` da `r0` ad ogni altro elemento in `A` e non esiste alcun arco entrante in `r0`. Dato un albero `T` di radice `r0`, si dice:

- insieme dei **discendenti** diretti (figli) di un nodo `m` di `T`: l'insieme dei nodi connessi dagli archi che si diramano da `m`;
- **discendente** di un nodo `m`: un nodo `n` tale che esiste una sequenza di nodi $n_1, n_2, \ldots n_k$ per cui $n_1=m$, $n_k=n$ e n_{i+1} è un discendente diretto di n_i per ($1 < i < n$);
- **livello** di un nodo `m`: lunghezza del cammino (numero di nodi) dalla radice `r0` al nodo `m`;
- **grado** di un nodo `m`: il numero dei figli di `m`
- **foglia**: un nodo senza figli (cioè di grado `0`)

Figura6. Albero



Con riferimento all'abero della figura qui sopra, i nodi `e`,`f` sono figli di `b` e discendenti di `a`. Il grado del nodo `a` è `3`, quello del nodo `d` è `4`. I nodi `e,f,c,q,h,k,p` sono foglie Tra i figli di ogni nodo esiste una relazione d'ordine che distingue tra il primo, secondo, etc. figlio (di norma disegnati da sinistra a destra).

Se `G=(VN, VT, S, P)` è una grammatica context-free, un albero è un albero di derivazione per `G` se:

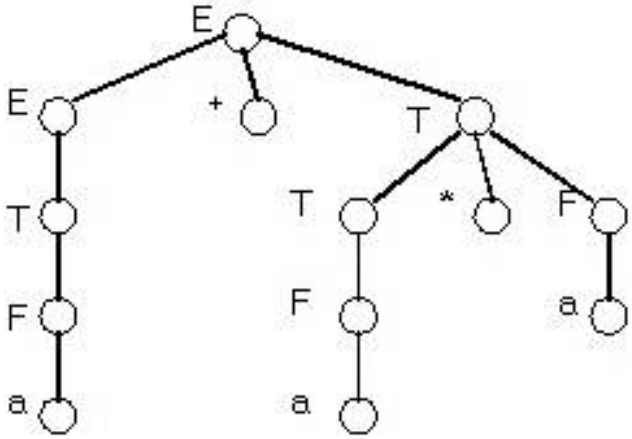
- ciascun nodo è associato ad un simbolo di V
- la radice è lo scopo S della grammatica
- se i nodi $A_1, A_2, \ldots A_k$ sono i figli ordinati di un nodo di etichetta A, allora $A ::= A_1\ A_2, \ldots\ A_k$ è una produzione in P.

Si consideri ad esempio ad la grammatica G = (VN,VT,S,P) con:

```
VN = {E,T,F} , VT = {a,+,*,(,)} , S = E,
P = {
E ::= E + T | T
T ::= T * F | F
F ::= ( E ) | a }
```

L' albero di derivazione della frase a+a*a è il seguente:

Figura7. Albero di derivazione dell'espressione a+a*a



L'albero di derivazione di una frase esprime la molteplicità delle singole sequenze di derivazione ottenibili attraverso la applicazione delle produzioni.

Derivazione canoniche

Supponendo di adottare un metodo di riscrittura che sostituisce sempre la variabile più a sinistra, (la così detta derivazione canonica sinistra), nel caso dell'esempio si avrebbe:

```
E => E+T => T + T => F + T => a + T => a + T * F => a + F * F =>
a+a * F => a + a * a
```

Sostituendo invece sempre la variabile più a destra (derivazione canonica destra), si otterrebbe la sequenza:

```
E => E+T => E+T*F => E+T*a => E+F*a => E+a*a => T+a*a =>
F+a*a => a + a * a
```

In [Hopcroft, Ullman] si può trovare la dimostrazione che S=*>s se e solo se esiste un albero di derivazione per la frase s.

Decidibilità dei linguaggi

Ogni frase di un linguaggio context-free può essere generata mediante una derivazione canonica (sinistra) (per la dimostrazione si veda [Hopcroft, Ullman]).

I linguaggi dipendenti dal contesto (e quindi anche quelli liberi da contesto e regolari) sono decidibili. I linguaggi di tipo 0 invece non è sempre detto siano decidibili.

La decidbilità, cioè il fatto che per linguaggi di tipo 1 (e quindi 2 e 3) esista un un procedimento meccanizzabile (algoritmo) capace di stabilire in un tempo finito se una qualunque frase costituita da simboli dell'alfabeto appartiene o meno al linguaggio è evidentemente un requisito essenziale per ogni linguaggio di programmazione.

In particolare si comprende perchè la sintassi dei linguaggi di programmazione viene generalmente descritta mediante grammatiche libere da contesto, o meglio da classi speciali di grammatiche context-free. Infatti le produzioni di questo tipo di grammatiche hanno una struttura tale da assicurare la definizione di riconoscitori e traduttori molto più efficienti di quanto non sia possibile nel caso di linguaggi generati da grammatiche dipendenti dal contesto. Le grammatiche regolari sono molto usate per decrivere alcune sottoparti di un linguaggio, come ad esempio identificatori e numeri.

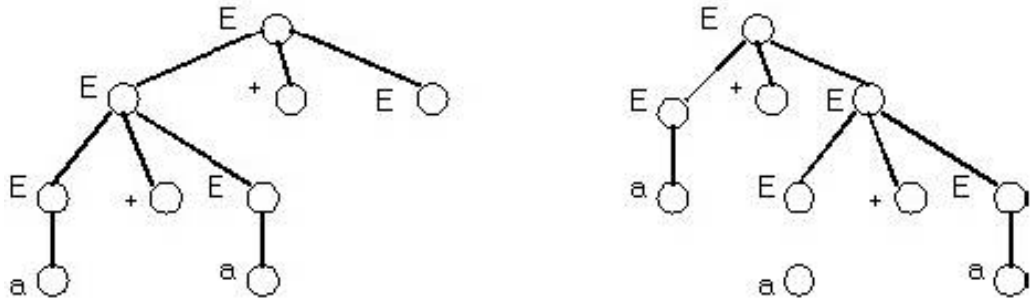
Ambiguita di grammatiche libere da contesto

Una grammatica context-free `G` si dice *ambigua* se esiste una frase `L(G)` che ammette due o più derivazioni canoniche sinistre distinte, o per cui vi sono almeno due alberi sintattici. Supponiamo ad esempio di riscrivere le produzioni precedenti come segue:

```
E ::= E + E
E ::= E * E
E ::= ( E )
E ::= a
```

Se consideriamo la stringa `a + a + a`, possiamo costruire due alberi di derivazione:

Figura8. Amibguità



Ovviamente l'ambiguità di una grammatica non è una caratteristica desiderabile e va eviata.

Il progetto dei riconoscitori

Il progetto e la costruzione di sistemi software volti alla analisi e riconoscimento di frasi di un linguaggio costituisce uno dei temi fondamentali e ricorrenti nella storia dell'informatica. Infatti, fin dalla nascita dei primi elaboratori (non solo elettronici) fu chiara la necessità di definire in modo rigoroso uno o più linguaggi di alto livello con cui esprimere i comandi da dare ad una macchina affinchè questa potesse risolvere in modo automatico un dato problema.

Il progetto e la realizzazione di un sistema per il riconoscimento delle frasi di un linguaggio si articola in una successione di passi ormai consolidata:

- Definizione formale della sintassi del linguaggio attraverso la specifica di una grammatica.
- Collocamento della grammatica nella classificazione di Chomsky e deduzione della macchina astratta richiesta per il riconoscimento delle frasi del linguaggio.
- Enucleazione, a partire dalla grammatica, del lessico del linguaggio.
- Impostazione della macchina astratta (di solito un automa a stati finiti) volta al riconoscimento delle unità lessicali (*token*) del linguaggio.
- Impostazione della macchina astratta (di solito un automa a stati finiti più stack) volta al riconoscimento delle unità sintattiche del linguaggio.

Per quanto riguarda la valutazione delle frasi riconosciute sintatticamente corrette, il punto di partenza dovrebbe essere, analogamente alla parte sintattica, la definizione formale della semantica del linguaggio e la progettazione, guidata da questa specifica, di un automa valutatore. Per questa parte va detto però che la specifica formale della semantica, pur possibile, è spesso tanto

complessa da essere impraticabile.

E' quindi consuetudine impostare la descrizione della semantica in linguaggio naturale riducendo al minimo indispensabile il formalismo matematico. Questo modo di procedere, se pur pragmaticamente valido, introduce tuttavia spesso specifiche non rigorose e anche contraddittorie. Per un esempio di una possibile definizione formale di Java e della JVM si può consultare *Java and the Java Virtual Machine: definzione, verification, validation* di Robert Stark, Joachim Scmid, Egon Borger, Springer 1998 ISBN 3-540-42088-6.

Data una frase appartenente ad un linguaggio, vi possono essere molte diverse forme di valutazione. Ad esempio, una valutazione potrebbe consistere nella riscrittura della frase, evidenziando in colori diversi le parti di cui essa si compone; oppure potrebbe consistere nella visualizzazione della frase in forma grafica, ad albero. Infine l'idea di valutazione potrebbe coincidere con l'idea di sostituire alla frase una "frase equivalente", idea che è alla base del concetto di calcolo. Ad esempio, la frase `2+3*5` è una espressione matematica la cui valutazione può produrre la "frase equivalente" `17` oppure la frase `diciassettte` oppure la frase `XVII` oppure la frase `11X`, etc.

Macchine astratte

Il rapporto tra la semantica di un linguaggio (di programmazione) e l'idea di *automa per la risoluzione di problemi* è collegato ad uno dei quesiti fondazionale dell'informatica: *quali elementi occorre introdurre per denotare processi computazionali meccanizzabili e quali proprietà di conseguenza possono essere associate a tali processi in virtù dei sistemi di denotazione adottati?*

Per affrontare il problema gli studiosi hanno inizialmente focalizzato l'attenzione su *cosa dovesse fare* una macchina del genere e solo in un secondo tempo su *come lo dovesse fare*.

Il concetto di **automa** (per elaborare informazione) viene introdotto come un *dispositivo concettuale* che stabilisce una precisa *relazione* tra un dato di ingresso e un dato di uscita, soddisfacendo i seguenti vincoli di realizzabilità fisica:

- se l'automa è fatto di parti, queste sono in *numero finito*;
- l'ingresso e l'uscita sono denotabili attraverso un *insieme finito* di simboli.

In questo modo non si considera alcun aspetto "fisico" o tecnologico specifico: l'automa può essere realizzato da un insieme di dispositivi elettronici digitali, come pure da dispositivi meccanici o biologici. L'obiettivo è appunto astrarre dai singoli, specifici casi concreti enucleando le caratteristiche essenziali.

Un primo rapporto tra il concetto di linguaggio di programmazione e il concetto di *automa risolutore di problemi*, è legato alla necessità di usare linguaggi per cui esista un (efficiente) *automa riconoscitore* ossia, appunto, un procedimento *meccanizzabile* per distinguere l'insieme delle frasi lecite (cioè appartenenti al linguaggio) da quelle sintatticamente scorrette.

Ma il rapporto più significativo, che ha segnato la storia stessa dell'informatica, è legato alla semantica del linguaggio; da questo punto di vista l'informatica (basandosi sulla matematica) ha seguito un "*processo di progettazione indipendente dalle tecnologie*", che ha prodotto come risultato la definizione di *modelli matematici* caratterizzati da precise proprietà. Questi modelli, detti *sistemi formali*, hanno definito di fatto il concetto stesso di *computabilità* e sono stati alla base dello sviluppo tecnologico che ha caratterizzato l'era dell'informazione.

Tra i più noti sistemi formali introdotti per caratterizzare il concetto di computabilità vi è una gerarchia di macchine astratte che parte dagli automi a stati finiti (**ASF**) e termina alla macchina di Turing (**TM**). Automi a capacità computazionale intermedia sono gli *automi a stati finiti con stack* (**PDA**: *Push Down Automata*) e la *macchina di Turing a nastro limitato*.

Vi è una importante corrispondenza tra la classificazione delle macchine astratte e quella delle grammatiche di Chomsky: ogni tipo di linguaggio nella classificazione di Chomsky può venire posto in corrispondenza con una specifica classe di macchine astratte, ciascuna delle quali caratterizza la struttura logica di un automa che riconosce le frasi del linguaggio. In particolare:

- i linguaggi di tipo 3 sono riconosciuti da automi a stati finiti;
- i linguaggi di tipo 2 sono riconosciuti da Push Down Automata;
- i linguaggi di tipo 1 sono riconosciuti da macchina di Turing (TM) con nastro limitato;
- i linguaggi di tipo 0 non sono decidibili in generale, ma, se lo sono, sono riconosciuti da TM.

Automi a stati finiti

Un ASF è definito da una quintupla: (`I, O, S, sfn, s`), ove:

```
I = alfabeto (insieme dei simboli) di ingresso
O = alfabeto (insieme dei simboli) di uscita
S = insieme degli stati
s = un particolare elemento di S (detto stato iniziale)
mfn: I x S -> O  (machine function)
sfn: I x S -> S  (state function)
```

L'uscita dipende sia dall'ingresso sia dallo stato interno.

Macchina di Turing

La macchina di Turing (TM) è formalmente definita dalla quintupla (`A,S, sfn,mfn,dfn`) ove:

```
A = insieme finito dei simboli di ingresso e uscita
S = insieme finito di stati (uno dei quali è halt)
mfn: A x S -> A          (machine function)
sfn: A x S -> S          (state function)
dfn: A x S -> {L,R,N} (direction function)
```

Questo automa è caratterizzato dalla capacità di leggere un simbolo dal nastro, di emettere una uscita, scrivendo sul nastro il simbolo specificato dalla funzione mfn, di transitare in un nuovo stato interno come specificato dalla sfn e di spostare una ideale testina di lettura-scrittura del nastro come specificato dalla dfn (R significa spostamento di una posizione a destra, L di una posizione a sinistra e N nessun spostamento). Quando raggiunge lo stato halt, la macchina si ferma.

Per risolvere un problema mediante una TM si può pensare di procedere in due fasi:

- definire una opportuna rappresentazione dei dati di ingresso sul nastro;
- definire la parte di controllo in modo da rendere disponibile su nastro, una volta che la TM entra nello stato halt, la rappresentazione della risposta.

Macchina di Turing Universale

Invece di costruire ex novo un automa diverso per ogni specifico problema, può risultare molto più conveniente, se possibile, costruire un **unico** automa capace da fungere da elaboratore universale di informazione. L'idea fu inizialmente formulata considerando *elaboratore universale di informazione* un qualunque ente capace di effettuare un insieme di mosse computazionalmente equivalenti a

quelle di una TM ed organizzato in modo da ricevere dall'esterno una *descrizione* a partire dalla quale *simulare* il comportamento di una specifica TM capace di risolvere il problema dato.

Ne consegue un modello logico articolato su due livelli orizzontali: il primo livello, invariante, realizza l'interprete di un linguaggio (il così detto *linguaggio macchina*); il secondo rappresenta un insieme di frasi scritte in linguaggio macchina: cambiando le frasi cambia il comportamento.

La concretizzazione del modello costituisce ancora una macchina astratta: la *Macchina di Turing Universale* (**UTM**) di cui l'elaboratore di Von Neumann può essere considerata una realizzazione concreta, in tecnologia elettronica. Storicamente, si fa discendere da questa impostazione la famiglia di linguaggi *imperativi*.

L'avvento dell'elaboratore elettronico come oggi lo conosciamo non ha ovviamente concluso l'iter di questo progetto, che continua a svolgersi lungo le volute di un tipico processo a spirale, su "dimensione storica". Infatti la progettazione degli elaboratori (intesi come *hardware* più *software di sistema*) costituisce un processo iterativo che produce continui raffinamenti innovativi anno dopo anno, sotto la spinta di nuovi scenari e di nuovi requisiti. Tra le modifiche più rilevanti degli ultimi anni vi è il fatto che il concetto di automa di elaborazione ha superato i limiti del singolo dispositivo, a favore della visione di una *rete* di elaboratori in cui il concetto stesso di elaborazione come pura trasformazione di ingressi in uscita è oggetto di discussione e revisione.

Stili computazionali

Sul piano del modello teorico, un elaboratore di informazione nasce dunque come un puro e cieco *manipolatore di segni*. Anche se oggi ogni elaboratore concreto è capace di riconoscere e gestire diversi tipi di informazione, non vi è alcuna necessità che a questi segni l'elaboratore dia una interpretazione predefinita: l'interpretazione rimane tutta nella mente di *chi usa* l'elaboratore.

Questo concetto, che sembra molto radicale e per certi versi irrealistico, costituisce una della basi della *Computer Science* e traspare evidente nel contesto dei sistemi formali e in parte anche dai linguaggi scaturiti da formalismi alternativi alla macchina di Turing.

Tra questi formalismi alternativi,l'*approccio funzionale* dei formalismi di *Hilbert, Church, Kleene* è fondato sul concetto di funzione

matematica e ha come obiettivo caratterizzare il concetto di *funzione computabile*. Questi formalismi sono alla base della famiglia dei *linguaggi di programmazione funzionali*.

I sistemi di produzione di *Thue, Post, Markov* partono invece dall'idea di automa come insieme di *regole di riscrittura* (dette anche produzioni o regole di inferenza) che trasformano frasi (insiemi di simboli) in altre frasi. Essi sono alla base della famiglia dei *linguaggi di programmazione logici*.

Secondo la *tesi di Church-Turing* non vi è alcun formalismo capace di risolvere una classe più ampia di problemi della macchina di Turing. Questa affermazione non può essere formalmente dimostrata o confutata. Tuttavia essa è sostenuta dal fatto che i vari formalismi introdotti allo scopo di caratterizzare il concetto di computabilità si sono rivelati tutti equivalenti a quello di Turing e quindi l'uno all'altro. I diversi formalismi si differenziano invece radicalmente per il modo con cui giungono ad esprimere la soluzione ad un problema.

La prevalenza dello stile imperativo sugli stili funzionale e logico è da ricondurre alla scarsa capacità di elaborazione e memoria dei primi elaboratori, che rendeva troppo forte la distanza tra il livello dei linguaggio macchina e il livello di astrazione connesso ai formalismi non imperativi. Oggi queste limitazioni sono ampiamente superate e ciò che va colmato è la lacuna culturale che si è nel frattempo formata nelle menti dei progettisti e dei loro formatori.

Problemi non risolubili

La formalizzazione del concetto di computabilità ha permesso di chiarire che possono esistere problemi ben formulati per cui non è possibile individuare alcun algoritmo che li risolve. L'esempio canonico è quello ormai famoso dell'**halt della macchina di Turing:** *dato un qualunque programma* `P` *e una qualunque configurazione di ingresso* `D`*, stabilire se* `P` *con ingresso* `D` *termina o meno*.

La non risolubilità di questo problema è relativa alla impossibilità di costruire un algoritmo valido per tutte le coppie `(P,D)`. Ciò non esclude però che il problema possa essere risolto per coppie `(P,D)` specifiche o per classi particolari di programmi. Una delle tecniche più diffuse per stabilire che un problema non è risolubile è dimostrare che esso è riconducibile al problema dell'halt di Turing.

Il riconoscimento delle espressioni

Nel seguito introdurremo i punti salienti di un possibile processo di analisi, progetto e realizzazione di un riconoscitore e valutatore di un linguaggio volto ad esprimere espressioni aritmetiche definite dalla seguente grammatica:

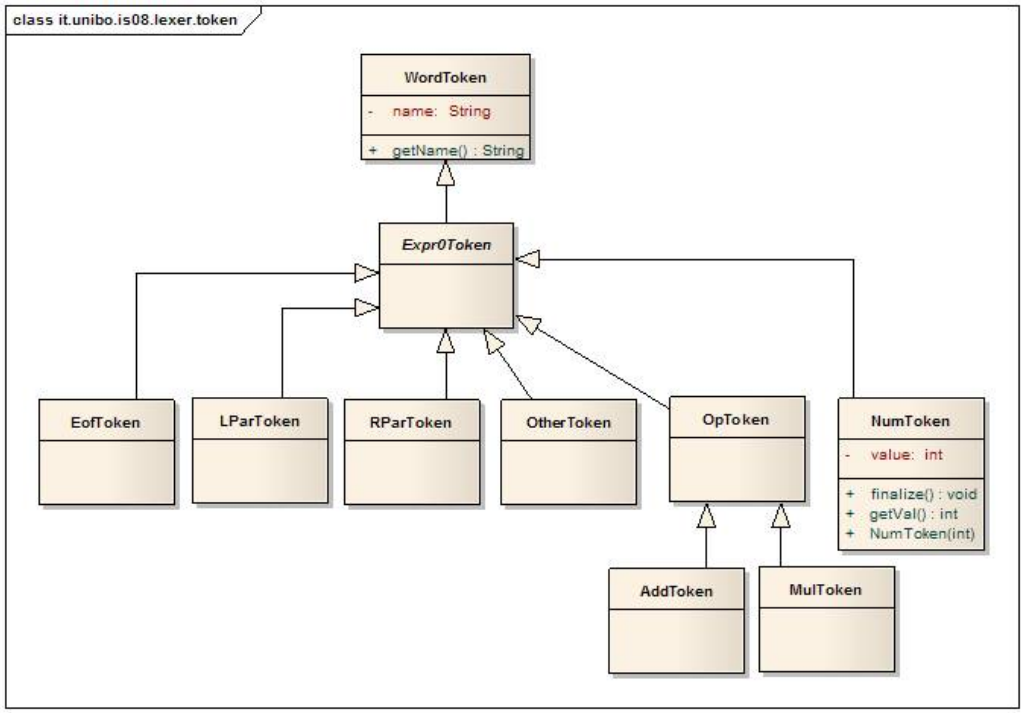
```
E ::= T | E + T | E - T
T ::= F | T * F | T / F
F ::= N | ( E )
N ::= D | N D
D ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

`E,T,F,N,D` denotano simboli non terminali e `+ - * / ( ) 1 2 3 4 5 6 7 8 9` denotano simboli terminali. I simboli `+ - * /` rappresentano gli usuali operatori di somma, sottrazione, prodotto, divisione e quoziente; il linguaggio generato dal metasimbolo `N` denota la rappresentazione in base `10` di un numero intero.

Analisi dei requisiti

Durante la fase di analisi dei requisiti, il committente ha chiarito che desidera la realizzazione di un analizzatore lessicale (*lexer*) e di un analizzatiore sintattico (*parser*) relativo al linguaggio generato dalla grammatica. Scopo del lexer è fornire la sequenza dei `token` presenti in una frase. Nel caso della delle espressioni artimetiche, è prassi comune considerare come token gli elementi terminali e i numeri (produzione `N`). Sulla base della grammatica data e interagendo con il committente, l'analista dei requisiti ha definito la nozione di `token` attraverso il seguente modello `UML`:

Figura9. Modello dei token



Ogni token è una entità atomica dotata dell'attributo `name` che verrà usato per distinguere i token di una stessa classe. Ad esempio il simbolo `+` può essere rappresentato da un `AddToken` con `name` uguale a `"+"`. I token che rappresentano numeri naturali (`NumToken`) posseggono anche l'attibuto `value` che ne denota il valore.

In base a questo modello è possibile anche stabilire piani di collaudo come quelli che seguono:

```
public final void testOpToken(){
        WordToken fixture = new MulToken("*");
        assertEquals("testOpToken", fixture.getName(), "*" );
}

public final void testNumToken(){
        NumToken fixture = new NumToken(12);
        assertTrue("testNumToken", fixture.getVal()==12 );
}
```

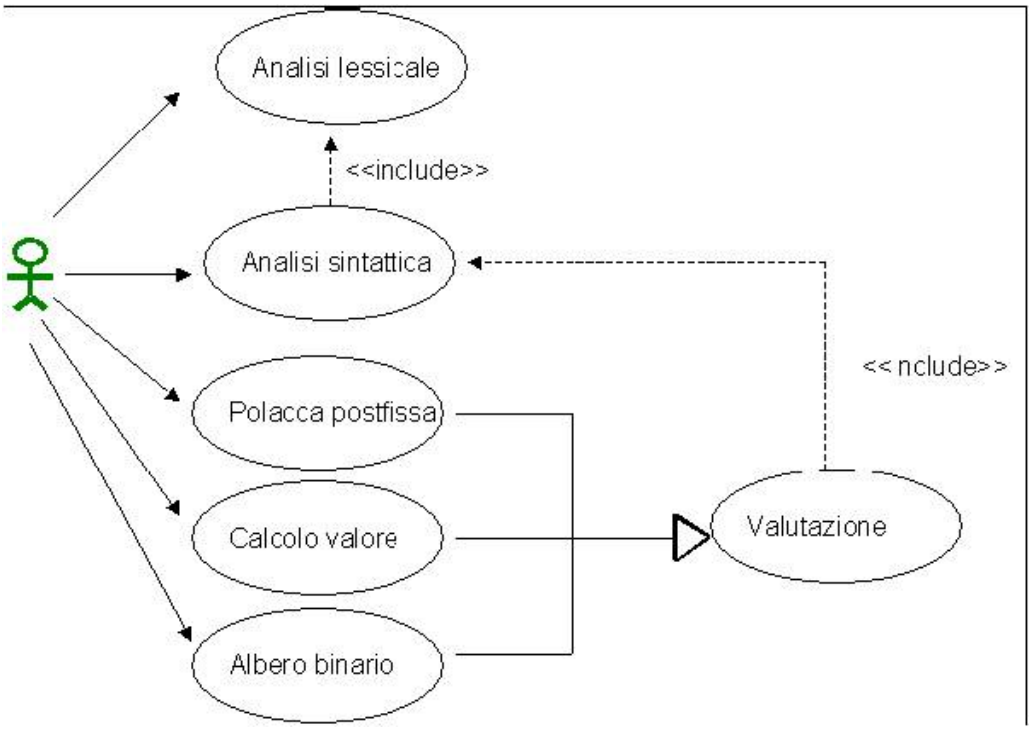
Questo piano di collaudo prefigura anche una precisa impostazione di costruttori.

Analisi del problema

L'analisi del problema è fortunatamente nota, essendo il concetto di espressione aritmetica oggetto dello studio della matematica e dell'informatica teorica. Per quanto riguarda il processo di costruzione del software, ricordiamo che obiettivi fondamentali dell'analisi sono:

- Documentazione dei requisiti mediante use cases e scenari. Ad esempio:

Figura10. Casi d'uso



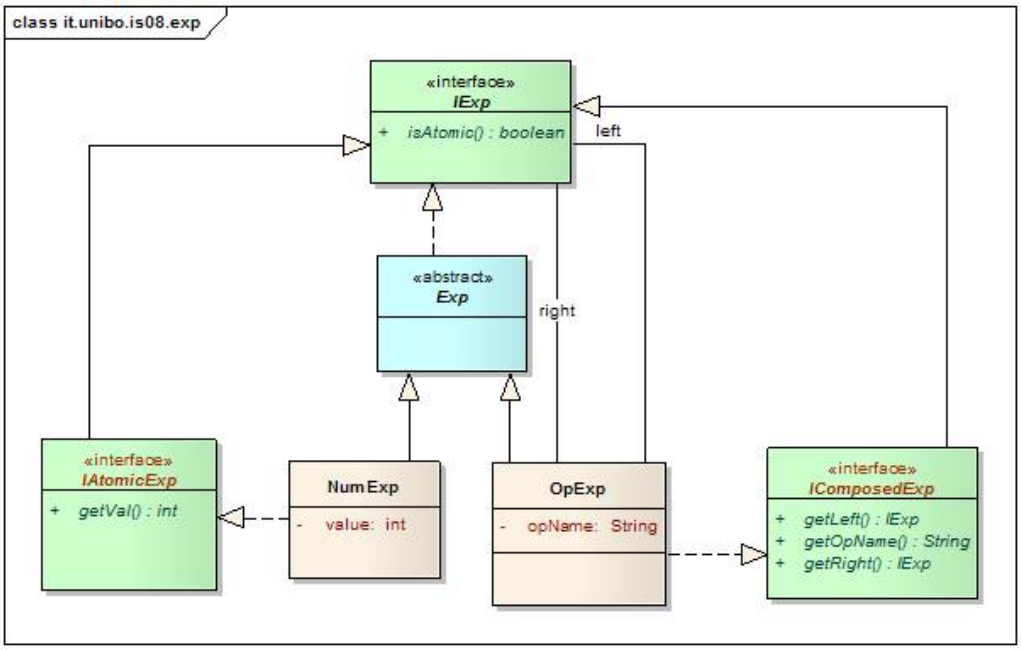
- Individuazione delle entità caratterizzanti il dominio applicativo e definizione di un *vocabolario* del dominico.

Nel caso specifico l'analista può impostare un modello logico delle espressioni osservando che le regole grammaticali possono produrre:

- epressioni atomiche formate da singoli fattori (`F`) in forma di numeri; esempi di espressione di queto tipo sono `"237"`, `"(21)"`, `"((413))"`. In pratica le espressioni atomiche sono formate da singoli numeri.
- epressioni composte formate da più fattori e da operatori binari; un esempio di espressione di queto tipo è `"2 + 3*(5 - 4)"`.

Su queste basi il modello delle espressioni può essere definito facendo riferimento al pattern Composite on page 88 :

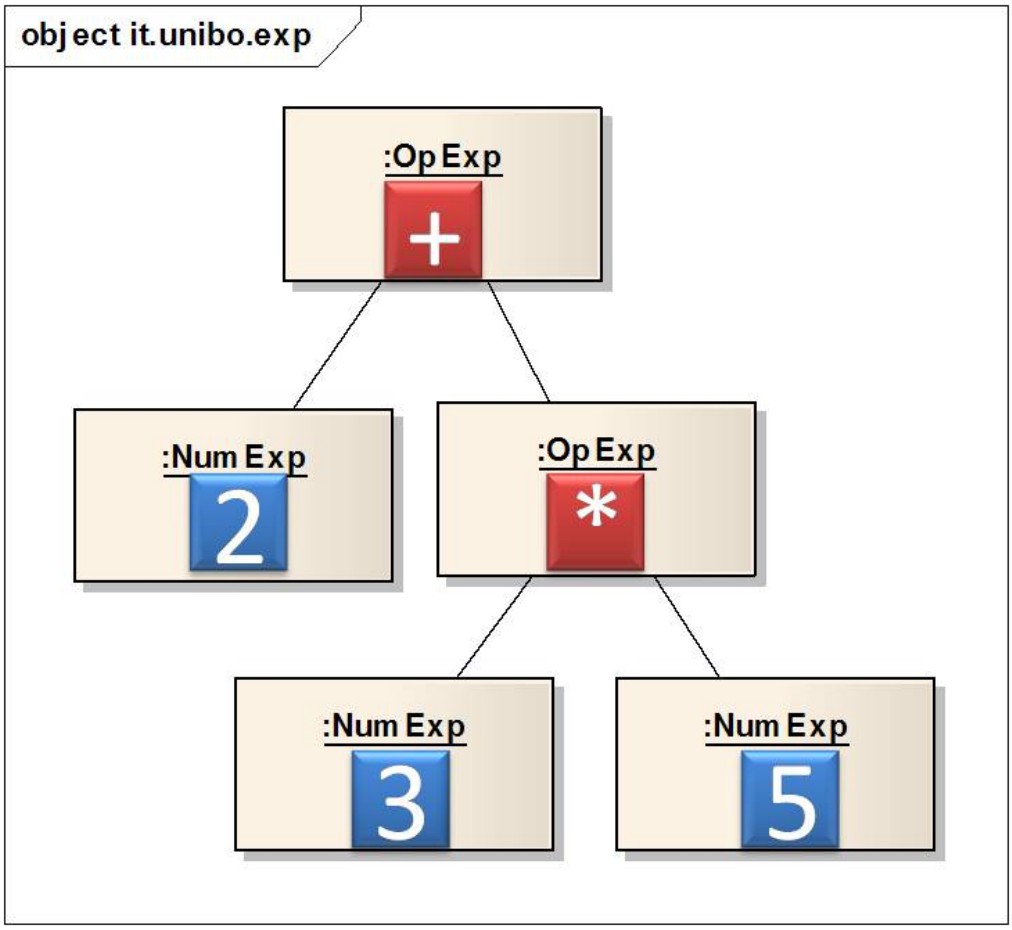
Figura11. Modello delle espressioni



La classe (astratta) `Exp` rappresenta l'entità alla radice del pattern; la classe `NumExp` rappresenta una foglia, mentre la classe `OpExp` rappresenta un elemento composto da due entità selezionabili attraverso le associazioni *left* e *right*.

Questo modello potrà costituire il punto di riferimento per la costruzione di una rappresentazione ad oggetti delle espressioni da parte di un analizzatore sintattico (parser). Poichè la rappresentazione interna prodotta dal parser è detta *Abstract Parse Tree* (`APT`), gli elementi (nodi) di questo albero saranno istanze delle classi definite dal modello. Ad esempio all'espressione `2 + 3 * 5` corrisponderà il seguente `APT`:

Figura12. APT di 2+3*5



In base a questo modello è possibile impostare anche le operazioni di costruzione relative alle classi concrete:

`NumExp`	Per costruire una espressione atomica `NumExp` basta disporre di un valore intero.
`OpExp`	Per costruire una espressione composta `OpExp` occorre disporre di una tripla di valori: una oggetto (ad es. una stringa) che denota un operatore e due oggetti di tipo `IExp` che denotano gli operandi.

Supponendo che la piattaforma operativa sia Java, queste specifiche logiche si possono tradurre nel codice che segue:

```
public abstract class Exp implements IExp {
        public boolean isAtomic(){
                return
this.getClass().getInterfaces()[0].getName().equals(
                        IAtomicExp.class.getName());
        }
}

public class NumExp extends Exp implements IAtomicExp {
private int value;
```

```
        public NumExp(int value){
                this.value = value;
        }
         ...
 }

public class OpExp extends Exp implements IComposedExp {
private String opName;
private IExp left;
private IExp right;
        public OpExp(String opName, IExp left, IExp right ){
                this.left = left;
                this.right = right;
                this.opName = opName;
        }
        ...
}
```

La definizione di questo codice non è necessaria a questo punto dello sviluppo; tuttavia l'introduzione dei costruttori ci permetterà di comprendere in modo più concreto l'importanze e il ruolo del modello di dominio e di definire in modo completo i piani di collaudo. Ad esempio:

```
public class OpExpTest extends junit.framework.TestCase {
private IComposedExp fixture;
...
protected void setUp() throws Exception{
        super.setUp();
        fixture = new OpExp( "+", new NumExp(10), new NumExp(20)
);
}

public final void testGetLeft(){
        assertTrue("testGetLeft", fixture.getLeft().isAtomic());
}

public final void testGetOpName(){
        assertTrue("testGetOpName",
fixture.getOpName().equals("+"));
}

public final void testGetRight(){
        assertTrue("testGetRight", fixture.getRight().isAtomic());
}
}
```

Ricordando che il codice del collaudo va scritto in un package distinto da quello dell'applicazione, si può pensare di introdurre a questo punto la seguente suite `JUnit`:

```
import junit.framework.Test;
import junit.framework.TestSuite;
import junit.textui.TestRunner;

 public class ExpAllTest extends TestSuite{

        public ExpAllTest(String name) {
                super(name);
        }

        public static Test suite() {
                TestSuite suite = new ExpAllTest("Expr Tests");
                suite.addTestSuite(AtomicExpTest.class);
                suite.addTestSuite(OpExpTest.class);
                suite.addTestSuite(ExpTest.class);
                return suite;
        }

        public static void main(String[] args) {
```

```
                TestRunner.run(suite());
        }
}
```

In questa suite è stato anche introdotto un main che permette l'attivazione del collaudo come una normale applicazione Java sotto il controllo della utility `junit.textui.TestRunner` che può essere anche lanciata (posizionandosi nella directory del progetto che contiente la directory `bin` del bytecode) attraverso un comando del tipo:

```
java -cp ./bin;C:/repo/junit/junit/3.8.1/junit-3.8.1.jar
        junit.swingui.TestRunner
java -cp ./bin;C:/repo/junit/junit/3.8.1/junit-3.8.1.jar
        junit.textui.TestRunner it.unibo.exp.exp.test.ExpAllTest
```

Sintassi astratta

Il modello delle espressioni sviluppato fino a questo punto fornisce la sintassi astratta delle espressioni generate dalla grammatica data, che può essere utilizzata per costruire applicazioni anche in assenza di un parser e di un lexer. Ad esempio l'espressione:

```
2 * 3 - (4 + 1 )
```

può essere costruita come segue:

```
IExp e1  = new OpExp( "*", new NumExp(2), new NumExp(3) );
IExp e2  = new OpExp( "+", new NumExp(4), new NumExp(1) );
IExp exp = new OpExp( "-", e1, e2 );
```

Sulla base delle interfacce definite è anche possibile impostare algoritmi di riscrittura e di valutazione delle espressioni senza averne completato l'implementazione. Le sezioni che seguono presenteranno qualche esempio in proposito.

Riscrittura in forma polacca

Un'espressione come `2 * 3 - (4 + 1 )` è scritta in *forma infissa*, cioè in una sintassi concreta in cui gli operatori binari compaiono tra i due rispettivi operandi. Un'espressione può tuttavia essere scritta anche in forma (polacca) prefissa, in cui gli operatori precedono gli gli operandi (`- 2 3 * 4 1 +`) o (polacca) postfissa, in cui gli operatori seguono gli gli operandi (`2 3 * 4 1 + -`).

Data un'espressione in forma interna ad albero binario, la riscrittura in forma polacca può essere effettuta impostando una visita all'abero stesso; ad esempio:

```
public String visitPostfissa(IExp exp){
        if( exp.isAtomic() ) return ""+((IAtomicExp)exp).getVal();
```

```
        else {
                String lS = visitPostfissa(
                        ((IComposedExp)exp).getLeft() );
                String rS = visitPostfissa(
                        ((IComposedExp)exp).getRight() );
                return lS + " " + " " + rS + " "+
                        ((IComposedExp)exp).getOpName();
        }
}
```

Allo stesso modo si può impostare la valutazione di un'espressione:

```
public int eval(IExp exp){
int v=0;
        if( exp.isAtomic() ) return ((IAtomicExp)exp).getVal();
        else {
                int v1 = eval( ((IComposedExp)exp).getLeft() );
                int v2 = eval( ((IComposedExp)exp).getRight() );
                char op =
                        ((IComposedExp)exp).getOpName().charAt(0);
                switch( op ){
                        case '+' : v = v1 + v2;break;
                        case '-' : v = v1 - v2;break;
                        case '*' : v = v1 * v2;break;
                        case '/' : v = v1 / v2;break;
                }
                return v;
        }
}
```

Il pattern Visitor

Osservando il codice precedente vediamo che siamo chiamati ad eseguire operazioni specifiche in funzione del tipo di elemento che forma un oggetto di tipo `Exp`. Per evitare l'uso del controllo esplicito di tipo e del type-casting sarebbe necessario introdurre nuove operazioni nelle classi `NumExp` e `OpExp` per supportare le esigenze di visita e di valutazione. D'altra parte le necessità che possono sorgere da parte di chi utilizza esprssioni possono essere molteplici; ci si chiede quindi se sia possibile estendere le operazioni relative alla gerarchia di classi di radice `Exp` senza dover modificare ogni volta l'insieme di operazioni definite da ciascuna classe.

La risposta a questa domanda è fornita dal pattern Visitor on page 98 che realizza l'idea di *double dispatch* per cui l'operazione eseguita da un oggetto dipende non solo dal nome della richiesta e dal tipo dell'oggetto ma anche dal tipo di un secondo oggetto. Questo secondo oggetto è detto *visitor* ed incpasula la logica dell'operazione da eseguire.

Il meccanismo è il seguente: ogni classe della gerarchia definisce una operazione (di nome `accept`) che riceve come argomento di ingresso un visitor; tale operazione delega a questo visitor la responsabilità di eseguire un'operazione (di nome standard `visit`) sull'oggetto corrente trasferito come argomento secondo il seguente schema di

comportamento:

```
public void accept( Visitor v){
        v.visit(this);
```

Nel caso del nostro modello delle espressioni un *Visitor* potrebbe realizzare la seguente interfaccia:

```
 public interface IExpVisitor {
        public  void visit( IAtomicExp e );
        public  void visit( IComposedExp e );
        public  Object doJob();
}
```

Il metodo `doJob` restituisce l'oggetto che rappresenta il risultato dell'applicazione dell'operazione realizzata dal visitor. Il modello di dominio deve essere esteso come segue:

```
public interface IExp {
        public void accept(IExpVisitor e);
}
```

Su queste basi possiamo pianificare il seguente piano di collaudo:

```
private IExp fixture ;

protected void setUp() {
//          String frase = "2 * 3 - (4 + 1 )";
        IExp e1 = new OpExp( "*",
                                new NumExp(2), new NumExp(3) );
        IExp e2 = new OpExp( "+",
                                new NumExp(4), new NumExp(1) );
        fixture = new OpExp( "-", e1, e2 );
}//setUp

public void testPolacca() {
        IExpVisitor visitor = new RewriteVisitor();
        String resultExpected = "2 3 * 4 1 + -";
        String result =  visitor.doJob(fixture).toString();
        assertTrue("testPolacca", result.equals(resultExpected));
}//testPolacca

public void testEval() {
        IExpVisitor visitor = new EvalVisitor();
        String resultExpected = "1";
        fixture.accept(visitor);
        String result =  visitor.doJob(fixture).toString();
        assertTrue("testEval", result.equals(resultExpected));
}//testEval
```

Il visitor che realizza la riscrittura in forma polacca postfissa può essere definito come segue:

```
package it.unibo.exp.expVisitor;
import java.util.Stack;

public class RewriteVisitor implements IExpVisitor{
Stack<String>  stack = new Stack<String>();

        public void visit(IAtomicExp e) {
                stack.push(""+e.getVal());
        }

        public void visit(IComposedExp e) {
                e.getLeft().accept(this);
```

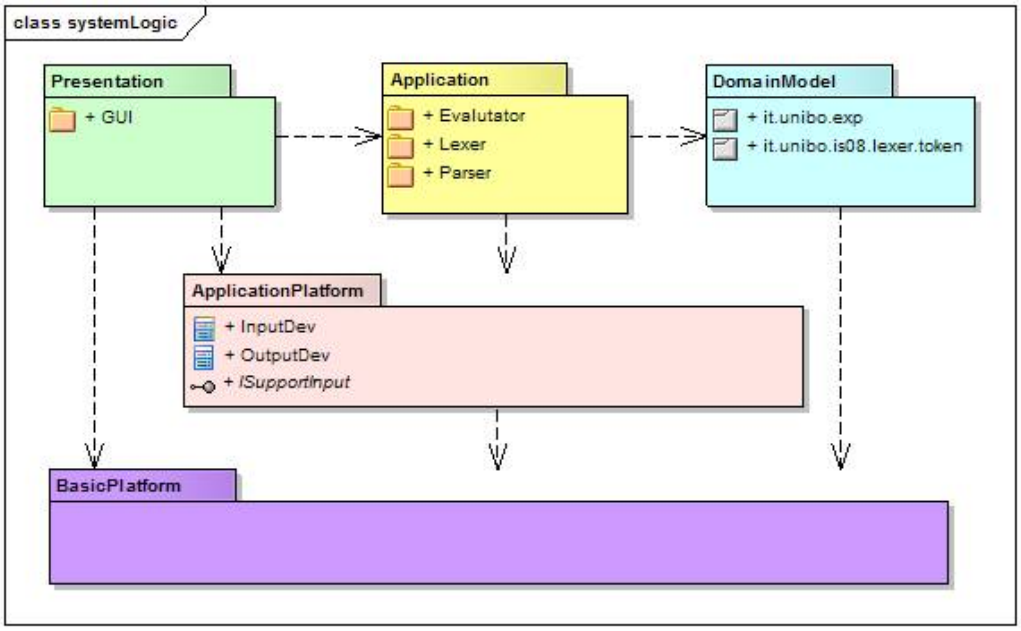
```
                String leftSon = (String) stack.pop();
                e.getRight().accept(this);
                String rightSon = (String) stack.pop();
                stack.push(
                  leftSon+" " + rightSon + " " + e.getOpName());
        }

        public String doJob(IExp e){
                e.accept(this);
                if( !stack.empty()) return stack.pop();
                else return null;
        }
}
```

La definizione del visitor che efettua la valutazione dell'espressione viene lasciata come esercizio al lettore.

Architettura logica

Un sistema per il riconoscimento e la valutazione di espressioni aritmetiche può essere basato sulla architettura logica rappresentata nella figura che segue:

Figura13. Architettura Logica del sistema



Questo diagramma strutturale dice che l'analista ha:

- Riconosciuto il fatto che il sistema richiede la definizione di tre sottosistemi (*three tier*):
  - il sottosistema che definisce il modello dei dati del dominio.
  - il sottosistema di presentazione che si occupa della interazione (in ingresso e in uscita) con l'utente (*interfaccia grafica*);
  - il sottosistema di elaborazione che realizza le attività che danno valore all'applicazione (*riconoscimento e valutazione di frasi*). Questo sottosistema comprende anche entità capaci

di configurare (*configurator*) il sistema e di gestire le interazioni (*controller*) tra il sottosistema di presentazione e quello di elaborazione.

- La separazione delle attività interne di elaborazione dai dispositivi di I/O.
- L'articolazione del sottosistema di riconoscimento e valutazione in tre parti: analizzatore lessicale (*Lexer*) analizzatore sintattico (*Parser*) e uno o più valutatori (*Evaluator*).

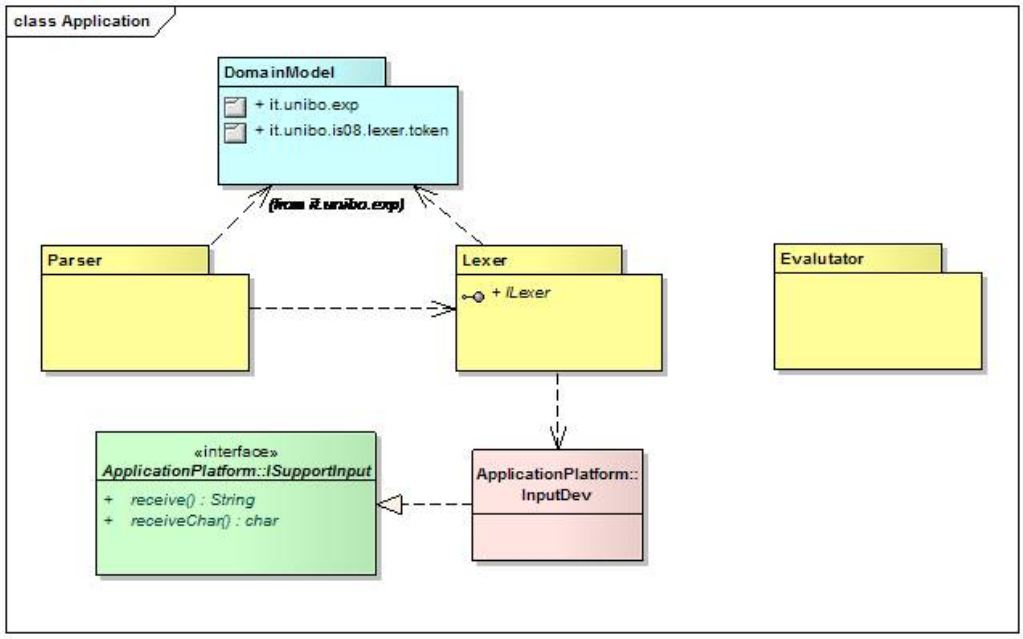
Naturalmente questa architettura va intesa come una prima decomposizione del sistema, legata ai requisiti funzionali e non funzionali che si vogliono raggiungere. Essa lascia ancora indefinite molte parti, quali ad esempio l'effettivo rappporto tra *Lexer*, *Parser* ed *Evaluator* tra il controller e gli altri componenti, etc..

Per quanto riguarda il sottosistema applicativo, l'analista può osservare che la frase da riconoscere e valutare può essere resa disponibile in diverse forme: su file, come stream di dati via rete, come campo di un database, etc. Per evitare dipendenze dirette da queste diverse tecnologie di supporto alle frasi, l'analista indica come opportuno pensare alla frase da analizzare come il contenuto informativo di un *dispositivo logico* di ingresso capace di permettere l'accesso sequenziale ai singoli caratteri. Questo dispositivo deve essere logicamente reso disponibile al *Lexer* attraverso un'interfaccia quale ad esempio:

```
public interface ISupportInput {
  public String receive() throws Exception;
  public char receiveChar() throws Exception;
}
```

Sulla base di queste considerazioni la parte strutturale dell'architettura logica della parte applicativa può essere rappresentata come segue:

Figura14. Architettura Logica della parte applicativa



L'analista ha detto che, sul piano logico, il *Parser* dipende dal *Lexer*; per essere più precisi ha detto qualcosa del tipo: *il (sottosistema)* `Parser` *deve avvelersi di* ***informazioni*** *prodotte dal (sottosistema)* `Lexer` . Non ha detto in alcun modo che, per ottenere queste informazioni, il (sottosistema) `Parser` deve "chiamare" (una operazione de) il `Lexer`; l'informazione potrebbe essere anche "iniettata" dal `Lexer` (o da altro sottosistema) entro il `Parser`.

Per quanto riguarda la parte di valutazione delle espressioni L'analista può osservare che un valutatore può essere posto in esecuzione in due modi diversi:

- procedendo passo-passo sotto la guida del `Parser`;
- agendo, al termine della attività del `Parser`, sull'`APT` da questi prodotto.

La figura mostra che l'analista non pone vincoli in merito a questo punto.

Piano di collaudo e di lavoro

La fase di analisi del problema è di strategica importanza all'interno del processo di produzione in quanto fornisce informazioni fondamentali per organizzare il lavoro, per identificare e valutare i rischi e per comprendere quali e quante risorse umane e tecnolgiche siano necessarie per affrontare il progetto e la realizzazione del sistema.

Nel caso specifico l'analista è fortunato, in quanto la letterature informatica riporta approfondite analisi sul tema del ricoscimento dei linguaggi. La struttura della grammatica permette di prefigurare la necessità di impostare il progetto e la realizzazione di un `PDA` per il sottosistema) `Parser` e (come di norma accade sempre) il progetto e la realizzazione di un `ASF` per il sottosistema) `Lexer`.

Progetto

Lo scopo della fase di progetto è raffinare l'architettura logica in modo da definire un insieme di parti (funzioni, classi, oggetti, componenti) e un insieme di relazioni tra queste parti in modo che il tutto possa essere realizzato con un conveniente rapporto costo-prestazioni avendo come riferimento una specifica tecnologia (Java, COM, C++, Web, etc).

Una possibile successione di attività da svolgere in questa fase è:

- Utilizzo dei pattern architetturali e di progetto per definizione l'architettura del sistema tenendo conto in modo sistematico delle forze in gioco
- Raffinamento dei piani di collaudo per ciascun componente e per ciascun sottosistema, prima separatamente e poi in modo integrato.
- Individuazione della tecnologia o delle tecnologie di riferimento, in relazione ai requisiti funzionali e non funzionali.
- Impostazione di un primo prototipo, da presentare rapidamente al committente il fine di procedere con più sicurezza alla revisione e/o conferma dei requisiti.

La fase di progettazione saraà discussa in modo approfondito nelle sezioni a seguire.

Progetto di un lexer

Iniziamo il progetto dell'analizzatore lessicale (`lexer`) partendo dalla definizione delle sue operazioni, assumendo come punto di vista quello della *interazione* tra un `lexer` (che potrebbe essere concepito come un oggetto, come un servizio, come un agente, etc) e i suoi utilizzatori (si veda Interazione on page 69).

Interazioni object oriented

Un primo approccio consiste nel definire la seguente interfaccia:

```
public interface ILexer {
  public Expr0Token next();
}
```

Questa interfaccia riflette l'idea di un lexer come una macchina con stato, capace di restituire al chiamante dell'operazione `next` un token diverso della sequenza, in ordine, per ogni diversa chiamata. La interazione è sincrona e di tipo *request-response*.

L'assenza di argomenti in `next` implica che il progettista intende associare al lexer la stringa (`sentence`) da analizzare; l'assenza di ogni altra operazione nell'interfaccia implica che l'associazione lexer-stringa dovrà avvenire al momento della costruzione del lexer.

Le ipotesi precedenti sono confermate dal seguente piano di collaudo:

```
private String sentence = "2+3*5=7";
private  ILexer fixture;

public void testNext(){
        Vector<Expr0Token> answer = createAnswer();
        fixture = createFixture(sentence);
        Expr0Token curToken = fixture.next();
        int i=0;
        while( ! (curToken instanceof EofToken ) ){
              assertEquals("testNext", curToken.toString(),
                 answer.elementAt(i).toString() );
           i++;
           curToken = fixture.next();
        }
}
```

Nel piano di collaudo l'operazione `createAnswer` provvede a creare la sequenza dei token attesi per la sentence assunta come ingresso.

```
protected Vector<Expr0Token> createAnswer(){
        Vector<Expr0Token> answer = new Vector<Expr0Token>();
        answer.add( new NumToken(2) );
        answer.add( new AddToken("+") );
        answer.add( new NumToken(3) );
        answer.add( new MulToken("*") );
        answer.add( new NumToken(5) );
        answer.add( new OtherToken("=") );
        answer.add( new NumToken(7) );
        return answer;
}
```

Una seconda operazione relativa al `lexer` può essere definita come segue:

```
public static Vector<Expr0Token> getAllTokens(String sentence)
```

Questa operazione può essere definita in una libreria; essa restituisce al chiamante tutta la lista di `token` presente nella frase di ingresso. Si noti che definire l'interfaccia

```
public interface ILexerWrong {
  public Expr0Token next();
  public Vector<Expr0Token> getAllTokens(String sentence)
}
```

avrebbe gettato i presupposti per la costruzione di un oggetto dalla semantica non chiara, che può introdurre notevoli problemi per un suo corretto utilizzo o molta complessità a livello implementativo.

Interazioni basate su messaggi

Una ulteriore interfaccia relativa al `lexer` può essere definita come segue:

```
public interface ILexerArtifact {
  public void produceTokens(String sentence, Key signature);
}
```

L'assenza di un valore in uscita implica che il progettista ha in mente un componente/servizio che restituisce l'informazione non al chiamante (direttamente) ma a qualcuno/qualcosa intrinsecamente connesso al servizio stesso. La interazione è di tipo asincrono; il cliente dovrebbe inviare un *message* (si veda Interazione on page 69) di nome `produceTokens` (con due argomenti, come specificato) inteso ad attivare il servizio che implementa `ILexerArtifact`; tale servizio deve acquisire il messaggio ed attivare azioni per emettere le informazioni attese dal cliente.

L'argomento di ingresso `signature` serve per "marcare" le risposte con una "chiave" indicata dal cliente, in modo che questi possa individuare le risposte pertinenti alla propria richiesta nel caso in cui il luogo usato dal servizio per il deposito delle risposte sia uno spazio condiviso.

Interazioni publish-subscribe

Per fornire al cliente le informazioni attese, il progettista potrebbe pensare di ricorrere al pattern observer on page 94 : [GHJV95] modificando l'interfaccia come segue:

```
public interface ILexerArtifact extends IObservable{
        ...
}

public interface IObservable{
        public void register( IObserver obj);
        public void unregister( IObserver obj);
}
```

Il cliente dovrebbe a sua volta comportarsi come un *observer*

realizzando una interfaccia del tipo:

```
public interface IObserver{
        public void update( Exp0Token value ) ;
}
```

Questa impostazione prevede che il cliente (l'observer, detto anche *subscriber*) si debba registare presso il `lexer` (l'observable, detto anche *publisher*) prima di utilizzarlo. In assenza di altre indicazioni, questa impostazione implica però che:

- più clienti possano registrarsi presso uno stesso `lexer`. Ciò può essere un vantaggio, a meno che non vi sono vincoli di privacy sulle informazioni emesse dal `lexer`; l'argomento `key` non è utile a questo scopo;
- il metodo `update` implica una interazione di tipo sincrono tra il `lexer` i suoi clienti-osservatori; se questo metodo entra in un loop infinto, il `lexer` risulta di fatto bloccato per il fatto di dover aggiornare gli osservatori;
- la implementazione può risultare non banale nel caso il client e il `lexer` siano allocati su due nodi di elaborazione diversi.

Interazioni attraverso mediator

Invece di ricorrere a un pattern object oriented, il progettista potrebbe fare riferimento ad uno *sile architetturale* in cui tutti i componenti/servizi hanno accesso a uno spazio logico di interazione condiviso in accordo al pattern pattern mediator on page 93 ; questo spazio condiviso potrebbe concretamente assumere due forme:

- uno spazio di memoria comune (`shared space`)
- un servizio in rete come *Java Message Service* (`JMS`)

Nel seguito approfondiremo l'uso di spazi di interazione di entrambi i tipi.

Concludiamo questa sezione osservando cho lo spazio condiviso può essere il luogo in cui depositare i diversi tipi di informazione citati in Interazione on page 69 (segnali, messaggi, etc.). Nel caso specifico l'interfaccia `ILexerArtifact` dovrebbe dichiarare in modo esplicito il tipo di informazione "emessa" da una operazione; ad esempio:

```
public interface ILexerArtifact {
  public void produceTokens(String sentence, Key signature)
                        emits message(Key,Expr0Token)
}
```

La mancanza di capacità espressiva a questo riguardo può essere

stimolo per la definizione di una estensione linguistica nella linea di quanto presentato nella sezione *Un linguaggio agenti-artefatti*.

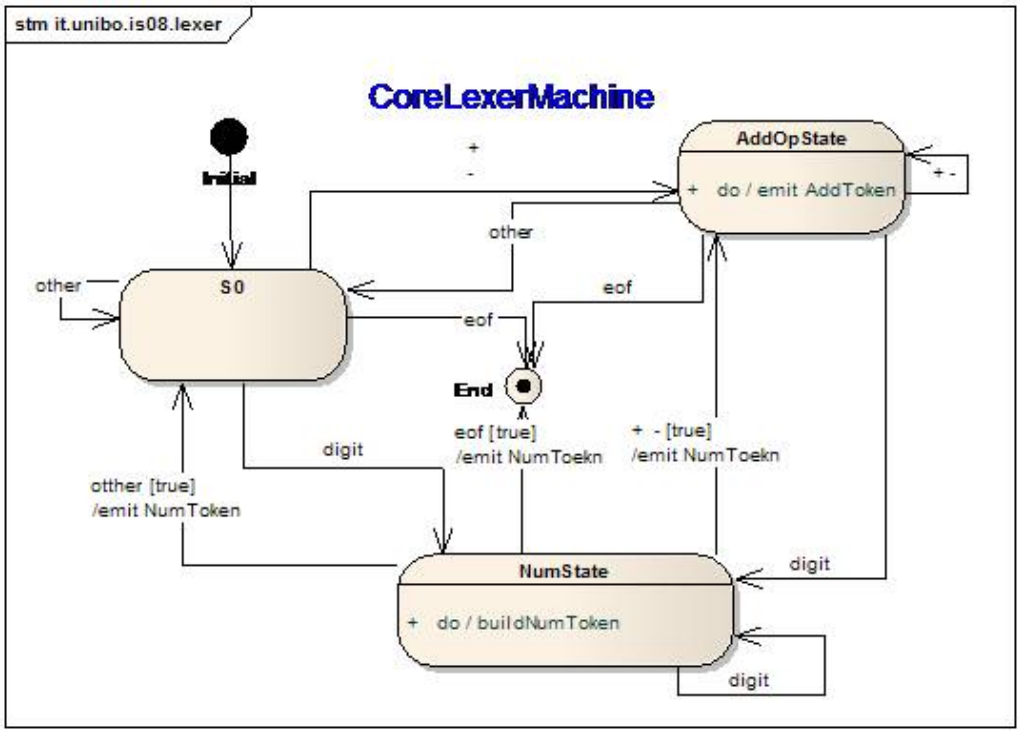
Lexer: struttura e comportamento

Per impostare il progetto della struttura e del comportamento di un analizzatore lessicale iniziamo dalla visione classica del lexer come automa a stati finiti (ASF) e quindi facciamo riferimento alla prima interfaccia d'uso introdotta:

```
public interface ILexer {
  public Expr0Token next();
}
```

La struttura della grammatica potrebbe portare alla definizione di un ASF come quello rappresentato dallo state diagram UML che segue:

Figura15. Il lexer come ASF



Il diagramma utilizza il termine `digit` per indicare una cifra decimale e il termine `other` per denotare qualunque simbolo (carattere) non compreso tra i simboli terminali della grammatica.

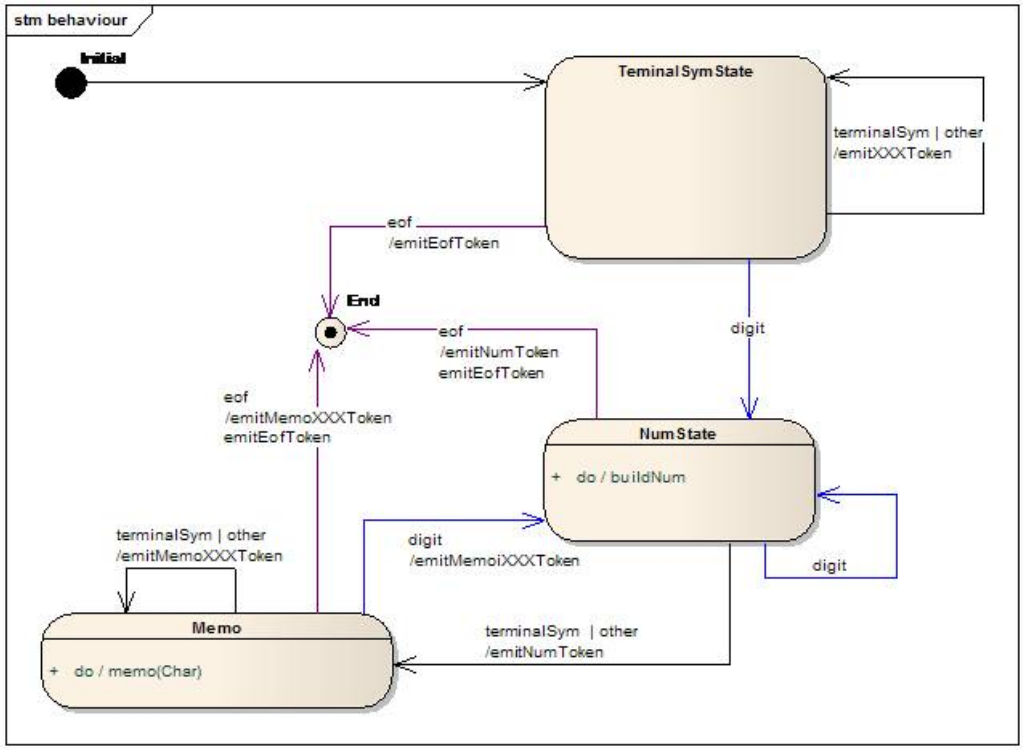
Il diagramma non mostra tutti gli stati: quelli mancanti, relativi al riconoscimento di operatori moltiplicativi e delle parentesi, sono simili allo stato `AddOpState` relativo al riconoscimento degli operatori

"+" o "-".

Questo automa emette le proprie uscite sia in corrispondenza agli stati, sia in corrispondenza a transizioni; esso è quindi un automa sia di Moore sia di Mealy. Si noti che al verificarsi della transizione da `NumState` a `AddOpSate` con ingresso `+` o `-` l'automa enette due uscite: un `NumToken` durante la transizione e un `AddToken` appena raggiunto il nuovo stato.

La versione mostrata nel diagramma che segue riduce il numero degli stati fattorizzando nello stato `TerminalSymState` la gestione dei simboli terminali (indicati da `terminalSym`) e dedicando lo stato `NumState` alla gestione dei `digit`.

Figura16. Il lexer come `ASF` di Mealy



L'automa è ora impostato come un ASF di Mealy che emette sempre solo un token tranne in alcune transizioni verso lo stato finale `End`, in cui pososno essere emessi due `token`, il secondo dei quali è sempre di classe `EofToken`.

Lo stato `Memo` rappresenta una memoria interna all'automa che tiene traccia di un token già riconosciuto ma non ancora emesso in uscita.

Dal modello dell'ASF al codice

L'ASF riconoscitore/emettitore di token rappresenta la soluzione (logica, astratta) al problema del riconoscimento lessicale. Questo automa può essere tradotto in codice in modo sistematico, creando una corrispondenza biunivoca tra gli elementi dell'automa e diverse parti del codice. Vi possono essere al proposito diverse strategie; nel seguito useremo i seguenti criteri:

1. Definizione di funzioni per la categorizzazione degli ingressi:

```
boolean isDigit( char n ){
  return (n >= '0' && n <= '9');
}

boolean isEof( char n ){
  return ( n == 65535 );
}

boolean isTerminal( char n ){
  return (isBraket(n) || isAddOp(n) || isMulOp(n) );
}

boolean isAddOp( char n ){
  return (n == '+' || n == '-');
}

boolean isMulOp( char n ){
  return (n == '*'|| n == '/');
}

boolean isBraket( char n ){
  return (n == '(' || n == ')');
}
```

2. Definizione dell'insieme degli stati mediante un tipo enumerativo:

```
enum State {
  TeminalSymState, NumState, MemoState, EndState, ErrorState
};
```

3. Definizione dello stato corrente (con inizializzazione):

```
State curState = State.TeminalSymState;
```

4. Definizione della funzione caratteristica di stato (*state function*) dell'automa:

```
void sfn(char n){
    switch (curState) {
  case TeminalSymState: TeminalSymState(n); break;
  case NumState: NumState(n); break;
  case MemoState: MemoState(n); break;
  case EndState: ErrorState(n); break;
  case ErrorState: ErrorState(n); break;
  default: ErrorState(n);
    }//switch
}
```

La *state function* `sfn` è definita ipotizzando che il simbolo di ingresso del'automa sia rappresentato da un singolo carattere. Il costrutto `switch` delega i dettagli del comportamento relativo a

ciascun stato a una funzione specifica per quello stato; in questo modo è facile modificare il codice per aggiungere / rimuovere stati.

Riportiamo qui di seguito la funzione relativa allo stato `TeminalSymState` sottolineando come la struttura interna di questa funzione rifletta le informazioni contenute nello state diagram; in particolare essa invoca una diversa funzione di transizione (si veda il punto 6) per ciascuna categoria di ingressi. Le funzioni relative agli altri stati sono definibili in modo analogo.

```
void TeminalSymState(char n){
  if ( isEof(n) ){
    transitionToEndState( n );
    return;
  }
  if( isDigit(n) ){
    transitionToNumState( n );
    return;
  }
  if( isTerminal(n) ){
    transitionMealy(State.TeminalSymState, n);
    return;
  }
  transitionMealy(State.TeminalSymState,n);
}
```

5. Definizione della funzione caratteristica di uscita (*machine function*) dell'automa:

```
public void mfn(State newState, char n){
switch (newState) {
case TeminalSymState:
    if( curState == State.MemoState)
      emitTheToken( memoCh );
    emitTheToken( n );
    break;
case NumState:  if( curState == State.MemoState)
      emitTheToken( memoCh );
    break;
case MemoState: if( curState == State.NumState )
      emitNumToken();
    else emitTheToken( memoCh );
    break;
case EndState:  if( curState == State.NumState )
      emitNumToken();
    else
    if( curState == State.MemoState)
      emitTheToken( memoCh );
    emitToken( new EofToken( ) );
    break;
case ErrorState: if( curState == State.NumState )
      emitNumToken();
    else
    if( curState == State.MemoState)
      emitTheToken( memoCh );
    emitToken( new ErrorToken( ) );
    break;
default:
  if( curState == State.MemoState)
    emitToken( new ErrorToken( ) );
  emitToken(new ErrorToken());
}//switch
}

void emitTheToken( char n ){
```

```
Expr0Token token;
   switch (n) {
      case '(':   token = new LParToken( ); break;
      case ')':   token = new RParToken( ); break;
      case '+':   token = new AddToken( "+" ); break;
      case '-':    token = new AddToken( "-" ); break;
      case '*':   token = new MulToken( "*" ); break;
      case '/':   token = new MulToken( "/" ); break;
      default:     token = new OtherToken( );
   }//switch
   emitToken( token );
}
```

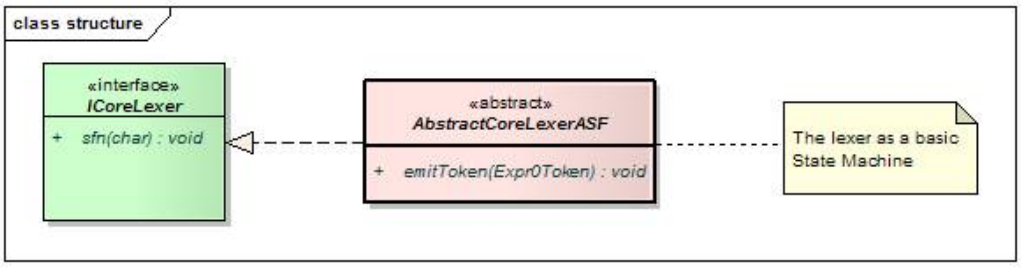
La funzione `emitToken` incapsula i dettagli in cui avviene l'emissione verso il mondo esterno di un `token`

6. Definizione di tante funzioni quante i diversi tipi di transizione dell'automa:

```
void transitionToEndState( char n ){
  mfn( State.EndState, n );
  curState = State.EndState;
}

void transitionMealy( State newState, char n){
  mfn( newState,n );
  curState = newState;
}

void transitionToNumState( char n ){
  curNum = curNum*10+(n-'0');
  mfn( State.NumState, n );
  curState = State.NumState;
}

void transitionToMemoState( char n ){
  memoCh = n;
  mfn( State.MemoState, n );
  curState = State.MemoState;
}
```

Per amore di chiarezza e modularità possiamo incaspulare il codice precedente in una classe, come rappresentato nel diagramma `UML` che segue:

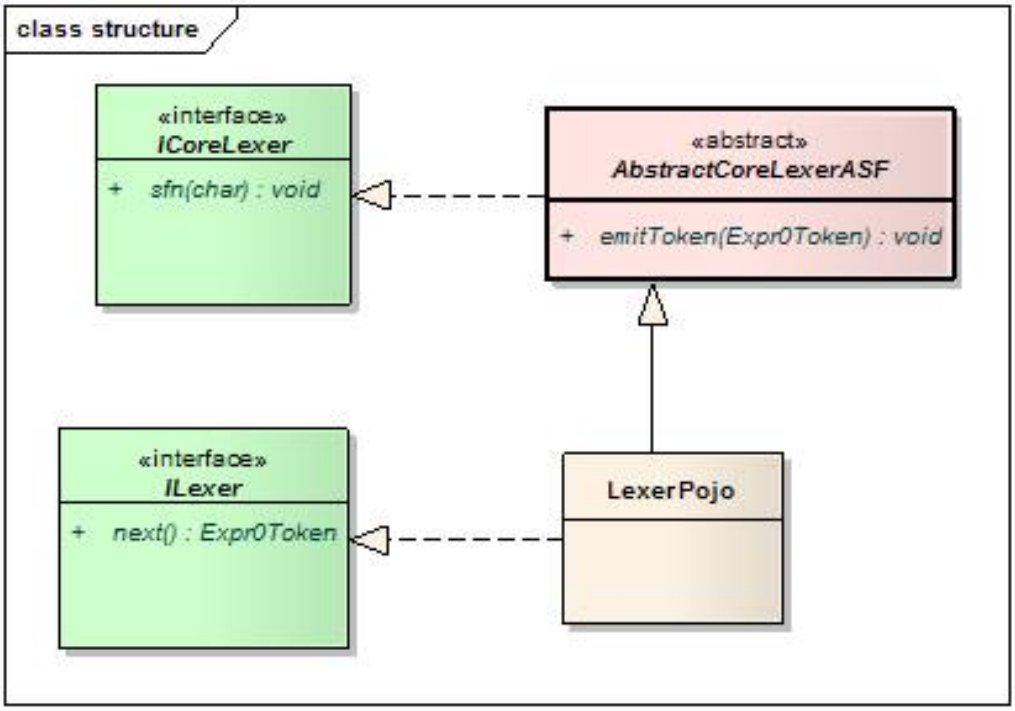
Figura17. Lexer core-machine



La classe astratta `AbstractCoreLexerASF` implementa l'operazione `sfn` lasciando non specificata l'operazione `emitToken`. E' stato cioè

applicato il pattern TemplateMethod on page 98 : [GHJV95] per evitare di cablare entro questa "macchina" i dettagli relativi alla emissione delle uscite; questi dettagli dovranno essere definiti da una versione specializzata della classe; ad esempio:

Figura18. Lexer come specializzazione della core-machine



`LexerPojo` è una specializzazione della classe `AbstractCoreLexerASF` che realizza in quasto caso un `POJO` cioè un *Plain Old Java Object*.

Interazioni con i dispositivi

Per evitare di dover modificare il codice al variare della sorgente-dati, è opportuno incapsulare in un oggetto di supporto la dipendenza dal dispositivo di ingresso della classe che implementa l'interfaccia `ILexer`. Al momento possiamo vicolare il supporto al "contratto" rappresentato dalla seguente interfaccia (su cui torneremo in seguito):

```
public interface ISupportInput {
  public String receive() throws Exception;
  public char receiveChar() throws Exception;
}
```

Le informazioni di ingresso possono essere disponibili in diverse forme: su stringa, su file, come stream di dati via rete, etc. Per ciascuna di queste forme si può introdurre una classe di

implementaizone dell'interfaccia `ISupportInput`; ad esempio, nel caso in cui l'ingresso sia disponibile in forma di stringa:

```
public class StringInputSupport implements ISupportInput {
 public StringInputSupport( String sentence ){
      ...//TODO
 }
  //TODO
}
```

Per evitare la dipendenza del codice dalla classe di implementazione dei dispositivi è opportuno l'uso del pattern FactoryMethod on page 90 :

```
public class SupportIOFactory {

  public static ISupportInput createStringSupport(
                                              String sentence){
          return new StringInputSupport( sentence );
  }

  public static ISupportInput createFileSupport(
                                              String fileName ){
          return null; //TODO
  }

}
```

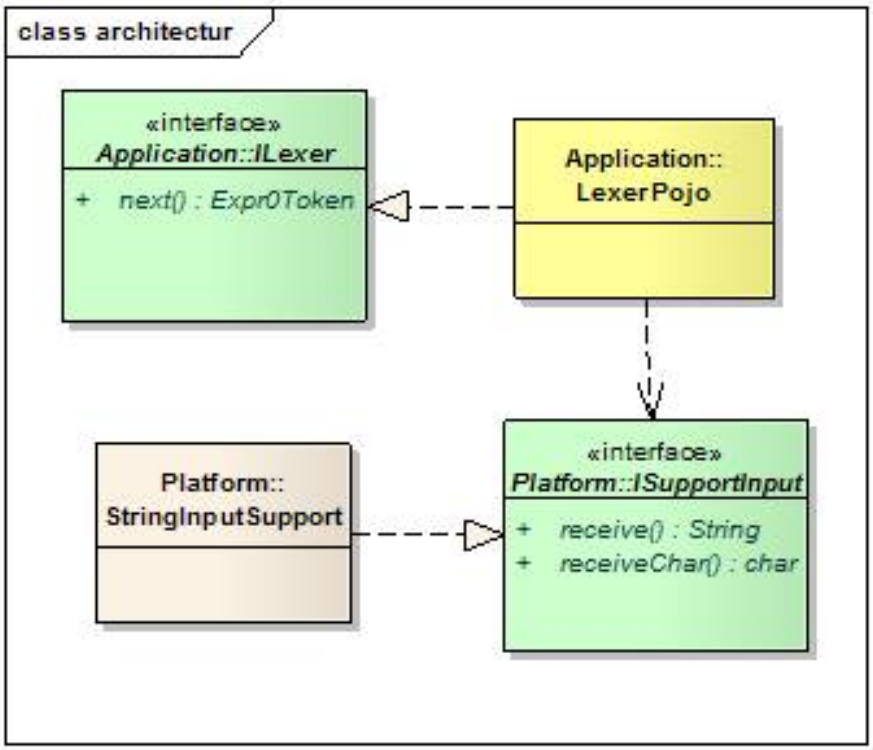
Il piano di collaudo mostra l'uso della factory e del dispositivo:

```
public final void testMoreChar(){
String str = "1 2 +-     H";
fixture = SupportIOFactory.createStringSupport(str);
try {
  for( int i=0; i<str.length();i++){
  char curCh = fixture.receiveChar();
  assertEquals("testMoreChar", curCh, str.charAt(i) );
  }
} catch (Exception e) {
  fail("testMoreChar " + e);
}
}
```

Layers

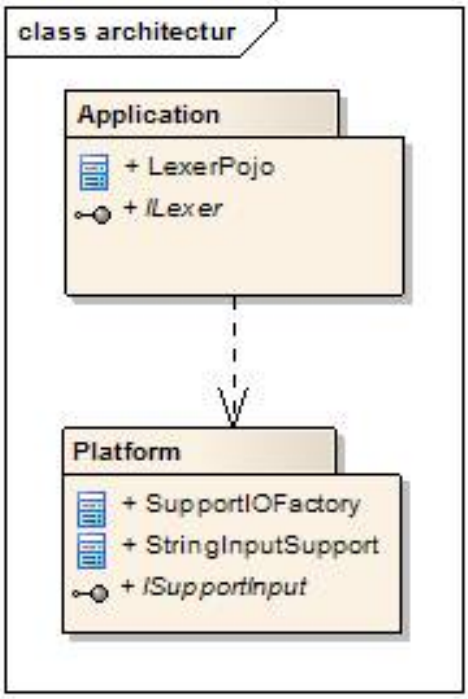
Il diagramma `UML` che segue mostra le relazione logica tra il lexer e il supporto di ingresso:

Figura19. Dipendenza dai dispositivi di input



Poichè i supporti di ingresso sono logicamente distinti dal codice applicativo (in questo caso costituito dal lexer) e possono costituire una risorsa riusabile in altre applicazioni, può essere conveniente incapsulare le classi di implementazione in un ambiente separato da quello che incapsula il codice del lexer. Il modo più semplice per ottenere lo scopo è avvalersi dell'entità `package`; ad esempio:

Figura20. I layer



Questo diagramma UML:

- rappresenta una tipica struttura a livelli, in cui i package più in alto rappresenta il layer applicativo e quello più in basso la "piattaforma" di supporto;
- include la definizione della interfaccia `ISupportInput` a livello piattaforma.

Il secondo punto solleva una questione importante: quale debba essere il livello "che comanda". Nel caso attuale è evidente che la piattaforma di supporto costituisce una sorta di implementazione per esigenze espresse dal layer applicativo. Tuttavia, il fatto che sia il package `Platform` a definire ed esporre il contratto rappresentato dall'interfaccia `ISupportInput` può indurre a pensare che debba essere il livello applicativo a doversi adeguare al modo con cui la piattaforma definisce l'uso dei suoi servizi e non la piattaforma a doversi adeguare alle esigenze del livello applicativo. D'altra parte, spostare la definizione dell'interfaccia `ISupportInput` nel package `Application` renderebbe il package `Platform` incapace di esporre i propri "servizi" in forma astratta

Sciogliere un dilemma di questo tipo (si veda Componenti e interfacce on page) costituisce una decisione molto importante per il progettista (se non per l'analista stesso) nel caso in cui i vari layer siano veri e propri componenti software.

Analisi sintattica

Il primo prototipo di analizzatore sintattico può essere costruito seguendo la stessa sequenza di azioni impostata per il prototipo del lexer. L'interfaccia del *Parser* può essere così definita:

```
public interface IParser{
   /**
   * Effettua l'analisi sintattica della stringa s,
   * restituendo l'Abstract Parse Tree dell'espressione.
   */
   public Exp parse( String frase );
}//IParser
```

Il piano di collaudo avrà una forma del tipo che segue:

```
public class TestParser extends TestCase {
private IParser parser;

 public TestParser(String name) {
    super(name);
 }

 protected void setUp() {
    parser = ParserFactory.create( );
 }//setUp

 public static Test suite() {
        TestSuite ts = new TestSuite();
        ts.addTest( new TestParser("testFraseOk") );
        ts.addTest( new TestParser("testFraseKo") );
        return ts;
 }

 public void testFraseOk() {
 String frase = "5 - 3 - 2" ;
 IExp resultExpected =
          new OpExp ('*' ,
                new OpExp('+',
                        new NumExp(1), new NumExp(2) ),
                 new NumExp(3) );
         IExp result = parser.parse( frase );
         checkExp( result, resultExpected );
  }//testFraseOk

 public void testFraseKo() {
 String frase = "(1+2)*)";
 String resultExpected = null;
 IExp result = parser.parse( frase );
         checkExp( result, resultExpected );
 }//testFraseKo

}//TestParser
```

Durante la fase di pianificazione del collaudo emergono alcuni casi notevoli:

- l'analisi della frase `5-2-3a` cosa restituisce?
- l'analisi della frase `(1+2)*3-)` cosa restituisce?
- l'analisi della frase `(1+2` cosa restituisce?

Per rispondere oseerviamo che un parser può essere impostato secondo diversi pattern di progettazione. Uno dei più diffusi e

semplici da realizzare si fonda su schemi di *analisi ricorsiva discendente* e prevede l'introduzione di tante procedure di analisi sintattica quante le produzioni grammaticali. In questo modo si stabilisce (come già avvenuto nel progetto del lexer) una corrispondenza sistematica tra la struttura della grammatica che definisce il linguaggio e la struttura del codice del riconoscitore, agevolando la modifica del prototipo in caso di modifica della grammatica.

Da questa impostazione scaturisce una tecnica di analisi ricorsiva discendente che consiste nell'associare ad ogni produzione `P` della grammatica una procedura di riconoscimento con una signature del tipo:

```
IExp ricP( Source )
```

ove `Source` denota (un dispositivo che rappresenta) la frase di ingresso, e `IExp` l'albero sintattico che fornisce il risultato dell'analisi. Di solito si assegna alla procedura il compito di determinare il *più lungo prefisso* in `Source` che costituisce una frase del linguaggio generato dalla produzione `P`.

Nel seguito le procedure di riconoscimento verranno definite associando all'argomento `Source` l'analizzatore lessicale (`Lexer`) definito in precedenza. In tal modo il `Parser` potrà avvantaggiarsi del lavoro svolto dal `Lexer` analizzando una sequenza di *token* e non una sequenza di caratteri.

Con queste premesse:

- l'analisi della frase `5-2-3a` restituisce l'albero che rappresenta l'espressione `5-2-3` lasciando come primo token "non consumato" il token relativo al carattere `a`;
- l'analisi della frase `(1+2)*3-)` potrebbe restituire `(1+2)*3` lasciando come primo token "non consumato" il token relativo al carattere `-`;
- l'analisi della frase `(1+2` restituisce `null` in quanto non riesce a trovare il token necessario a completare con successo l'analisi.

Nel caso della frase `(1+2)*3-)` la tecnica dell'analisi ricorsiva che adotteremo per il primo prototipo indurrà il parser alla consumazione (potremmo dire in modo "eager") del token relativo al carattere `-` in quanto esso rsulta accettabile in quel punto della frase; solo in seguito il parser si potrà accorgere che l'analisi della frase non può essere

completata con successo. Nel nostro piano di collaudo prefigureremo quindi che anche per questa frase il parser restituisca il valore `null` ad indicare la sua incapacità a completare con successo il riconoscimento, anche se un riconoscimento parziale ha avuto luogo.

Ricnoscimento dei fattori

Il codice che segue definisce una possibile realizzazione di un metodo dedicato al riconoscimento dei fattori, cioeè del linguaggio generato dalle produzioni che hanno come meta-simbolo `F`:

```
F :=  N
F ::= ( E )
```

Il metodo viene costruito sulla base del seguente ragionamento: *il metodo `F` deve "consumare" - partendo dal token corrente non ancora analizzato - tutti i token che si possono presentare in base alle regole che riscrivono il meta-simbolo `F`* .

Ogni frase prodotta dal meta-simbolo `F` ha come token iniziale o un `NumToken` o un `LParToken` relativo alla a `(` . Nel secondo caso il token `LParToken` deve essere seguito da una sequenza di token generabile dal meta-simbolo `E` a sua volta seguita dal token `RParToken` relativo alla `)`.

In base a questo ragionamento possiamo impostare la struttura del metodo riconoscitore anche se non abbiamo ancora scritto il codice del metodo dedicato al riconoscimento del linguaggio generato dal meta-simbolo `E`. Supponendo che la variabile non-locale `curToken` di tipo `Exp0Token` denoti il primo token non ancora analizzato, il codice può essere scritto come segue:

```
public class Parser implements IParser{

protected ILexer   lexer;
protected Exp0Token curToken;

...
        public IExp F() throws Exception {
        IExp myExp = null; //espressione riconosciuta da F
                 if( curToken instanceof NumToken ){
                //Produzione F := N
                        NumToken num = (NumToken)curToken;
                        curToken     = lexer.next();
                        return new NumExp( num.getVal() ) ;
                }
                if( curToken instanceof LParToken ){
                //Produzione F := ( E )
                        curToken = lexer.next();
                        myExp = E();
                        if( myExp == null ) return null;
                        if( curToken instanceof RParToken ){
                                curToken  = lexer.next();
                                return myExp;
                        }else return null;
                }
                else return null;
        }//F
```

Si noti che, dopo avere riconosciuto un token, il metodo modifica `curToken` in modo che esso referenzi il successivo token da analizzare.

Nel caso il metodo `F` sia chiamato ad operare in un momento in cui il valore di `curToken` non sia nè un `NumToken` nè un `LParToken`, il valore restituito è `null` in accordo a quanto discusso in fase di pianificazione del collaudo.

Ricnoscimento dei termini

In modo analogo a quanto fatto per i fattori è possibile definire in modo sistematico un metodo per il riconoscimento dei termini, cioè del linguaggio generato dalle produzioni che hanno come meta-simbolo `T`:

```
T :=  F
T ::= T * F
T ::= T / F
```

In questo caso *il metodo `T` deve "consumare" - partendo dal token corrente non ancora analizzato - tutti i token che si possono presentare in base alla regole che riscrivono il meta-simbolo `T`* . Per mettere in luce quale sia la sequenza di token lecita è opportuno eliminare le ricorsioni sinistre dalle ultime due regole, riscrivendole in notazione `BNF` come segue:

```
T :=  F
T ::= F { * F }
T ::= F { / F }
```

In questa notazione le riscritture contenute entro i meta-simbli `{ }` possono essere ripetute *0 o più volte*.

```
public class Parser implements IParser{

protected ILexer   lexer;
protected Exp0Token curToken;

...
        public IExp T() throws Exception{
        Expr0Token myOp;
        IExp myExp,fact2;
          //Produzione T := F
           myExp = F();
           if( myExp == null ) return null;
           while( curToken instanceof MulToken ) {
           //Produzioni T := F { * F } | F { / F }
                   myOp = curToken; //push the op
                   curToken = lexer.next();
                   fact2 = F();
                   if( fact2 == null ) return null;
                   myExp = new OpExp( myOp.getName(), myExp, fact2
);
           }//while
           return myExp;
        }
```

Si noti che:

- Il ciclo `while` esprime il concetto di ripetizione dell'attività di riconoscimento *0 o più volte*;
- il metodo di riconoscimento `F` viene invocato nella speranza che la sequenza di token sia corretta; nel caso in cui ciò non accada il metoodo `F` restuirà `null` come segno di mancato riconoscimento, il che indurra il metodo `T` a restituire `null` a sua volta;
- il metodo `T`, come ogni altro metodo di riconoscimento lascia nella variabile non locale `curToken` il primo token "non consumato";
- l'assegnamento `myOp = curToken;` alla variabile `myOp` locale alla procedura equivale ad una operazione di `push` (immissione in uno stack) del token che rappresenta l'operatore. Lo stack è in questo caso quello usato dalla macchina virtuale Java per gestire i record di attivazione delle procedure;
- il metodo `T` accumula nella variabile locale `myExp` la parte di frase che riesce a riconoscere applicando un criterio di associatività a sinistra, per cui una frase del tipo `12 / 3 / 2` viene considerata (tenendo conto delle regole ricorsive iniziali) equivalente a `(12 / 3) / 2` anzichè a `12 / (3 / 2)` .

Ricnoscimento delle espressioni

Il linguaggio geenrato dallo scopo `E` della grammatica può venire riconosciuto da un metodo organizzato in modo del tutto analogo a quanto fatto nel caso dei termini:

```
public class Parser implements IParser{
protected ILexer   lexer;
protected Exp0Token curToken;

 public Parser(String s){
  lexer = LexerFactory.createLexerPojo(s);
 }

 public Parser(ILexer lexer){
  this.lexer = lexer;
 }

 public IExp parse( ) throws Exception{
  curToken = lexer.next();
  return E();
 }

 public IExp E() throws Exception{
 Expr0Token myOp;
 IExp myExp,term2;
 //Produzione E := T
 myExp = T();
 if( myExp == null ) return null;
        while( curToken instanceof AddToken ) {
        //Produzioni T := T { + T } | T { - T }
         myOp = curToken; //push the op
         curToken = lexer.next();
         term2 = T();
         if( term2 == null ) return null;
         myExp = new OpExp( myOp.getName(), myExp, term2 );
        }//while
        return myExp;
```

```
 }
        ...
}
```

Si noti che il codice del `Parser` definisce anche due costruttori:

- un costruttore che riceve dall'esterno il `Lexer` relativo alla frase da nalizzare;
- un costruttore che riceve dall'esterno una stringa che rappresenta la frase da analizzare e provvede alla costruzione di un `Lexer` attraverso una classe factory `LexerFactory`.

Il secondo costruttore facilita il collaudo del parser.