Indice

1 — Linguaggi, macchine astratte, teoria della computabilità

Alcune definizioni

- **Algoritmo**: sequenza *finita* di istruzioni che risolve in un *tempo finito* una classe di problemi

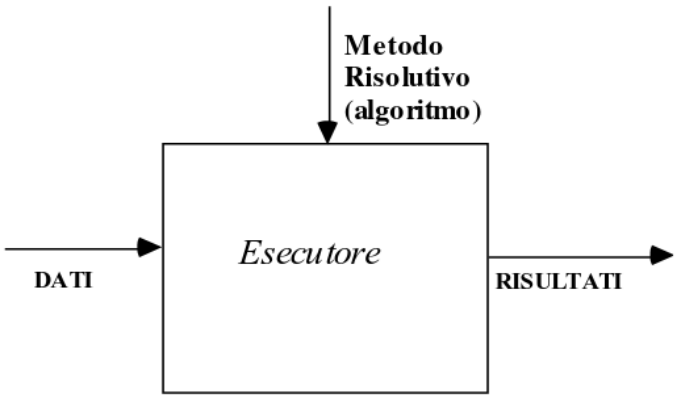
- **Codifica**: descrizione dell'algoritmo tramite un *insieme ordinato di frasi* di un linguaggio di programmazione, che specificano le *azioni* da svolgere

- **Programma**: testo scritto in accordo alla *sintassi* e alla *semantica* di un linguaggio di programmazione

1.0.1 Esecuzione

Un algoritmo esprime la soluzione a un problema, il programma è la formulazione testuale rigorosa in un dato linguaggio; l'esecuzione *ordinata* delle azioni specificate porta a ottenere la soluzione da un insieme di dati in ingresso.

Viene assunta l'esistenza di un **automa esecutore**, una macchina astratta capace di eseguire le azioni dell'algoritmo.



1.1 Macchine astratte

1.1.1 Automa esecutore

Un automa deve poter ricevere dall'esterno una *descrizione* dell'algoritmo, deve quindi essere in grado di *interpretare* un linguaggio, che verrà chiamato **linguaggio macchina**.

Realizzabilità fisica

- le parti che compongono l'automa devono essere in *numero finito*
- ingresso e uscita devono essere denotabili da un *insieme finito* di **simboli**

1.1.2 Gerarchia di macchine astratte

- macchina combinatoria
- automi a stati finiti
- ....
- macchina di Turing

Si individua una gerarchia perché diverse classi di macchina hanno diversa *capacità* di risolvere problemi. Se neanche la macchina più "potente" può risolvere un problema, questo potrebbe essere **non risolubile**.

1.1.3 Macchina base

La macchina base è definita formalmente dalla tripla:

$$\langle I, O, mfn \rangle$$

dove

- I = insieme finito dei simboli di *ingresso*
- O = insieme finito dei simboli di *uscita*
- $mfn : I \to O$ = funzione di macchina

Limiti

Utilizzare una di queste macchine per risolvere problemi comporta valutare tutte le possibili configurazioni di ingresso.

Esiste inoltre un altro limite, essendo puramente combinatoria, la macchina base non può risolvere problemi che richiedano di ricordare qualcosa dal passato, in quanto privo di memoria interna.

1.1.4 Automa a stati finiti

Nell'automa a stati finiti si introduce il concetto di memoria attraverso l'utilizzo di un numero *finito* di **stati interni**.

Un **automa a stati finiti** è definito dalla quintupla:

$$\langle I, O, S, mfn, sfn \rangle$$

dove

- I = insieme finito dei simboli di *ingresso*
- O = insieme finito dei simboli di *uscita*
- $mfn : I \times S \to O$ = funzione di macchina
- $sfn : I \times S \to S$ = funzione di **stato**

Limiti

Dato che lo stato funge da memoria interna, le risposte possono essere diverse a parità di dati d'ingresso.

Esistono varie categorie ASF come automi di Mealy o Moore, sincroni o asincroni, minimo numero di stati etc..

Esiste un limite computazionale che rende l'automa inadatto a risolvere problemi che non permette di limitare a priori la lunghezza delle sequenze, dovuto al fatto che la memoria è **finita**.

1.1.5 Macchina di Turing

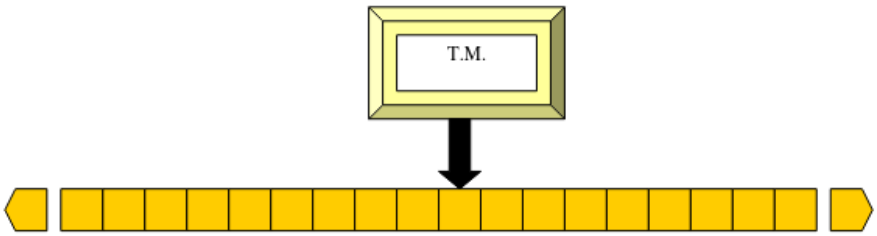
Per ovviare al limite della memoria finita, si introduce un *nastro* di memoria esterno, la **macchina di Turing** è definita dalla quintupla:

$$\langle A, S, mfn, sfn, dfn \rangle$$

dove

- A = insieme finito dei simboli di *ingresso* e *uscita*
- S = insieme finito degli *stati* (tra i quali $HALT$)
- $mfn : A \times S \to A$ = funzione di macchina
- $sfn : A \times S \to S$ = funzione di stato
- $dfn : A \times S \to D = \{Left, Right, None\}$, funzione di *direzione*

Il deposito dei dati è rappresentato da un natro illimitatamente espandibile



La macchina è dotata di una testina di lettura e scrittura che può:

- leggere un simbolo dal nastro
- scrivere sul nastro il simbolo specificato da $mfn()$
- passare a un nuovo stato interno specificato dalla $sfn()$
- spostarsi sul nastro di una posizione nella direzione indicata da $dfn()$

ipotizzando che quando viene raggiunto lo stato HALT, la macchina si ferma.

Per risolvere un problema con questa macchina, è necessario definire la *rappresentazione dei dati* di partenza (da porre sul nastro) e quelli di uscita (sempre sul nastro), e definire il *comportamento*, ovvero le tre funzioni $mfn()$, $sfn()$ e $dfn()$.

Non è certo che questa macchina arrivi allo stato di HALT.

Tesi di Church-Turing

Esiste una macchina più "potente" della MdT?

La tesi di Church-Turing afferma che

> Non esiste alcun formalismo capace di risolvere una classe di problemi più ampia di quella risolta dalla macchina di Turing.

1.1.6 Macchine universali

Nella macchina di Turing l'algoritmo è cablato nella macchina.

Si può espandere la MdT alla macchina di Turing **Universale** (UTM), macchina il cui programma è letto dalla macchina direttamente dal nastro.

Questo richiede poter descrivere l'algoritmo richiesto attraverso un *linguaggio* e avere una macchina che lo interpreti.

Le tre operazioni che la UTM effettua sono:

- **fetch**, ricerca delle istruzioni
- **decode**, interpretazione delle istruzioni
- **execute**, esecuzione le istruzioni

UTM vs Macchina di Von Neumann

Macchina di Turing

- leggere/scrivere simbolo da/su nastro
- transitare in un nuovo stato
- spostarsi sul nastro di x posizioni

Macchina di Von Neumann

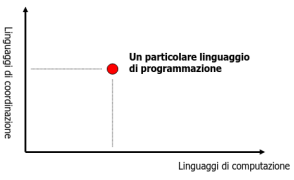
- lettura/scrittura da/su RAM/-ROM
- nuova configurazione registri CPU
- scelta cella di memoria su cui operare

La UTM non ha il concetto di "mondo esterno", ne nessuna istruzione di I/O, è puramente computazione senza la dimensione di *interazione*.

Computazione e interazione

Computazione e interazione sono dimensioni *ortogonali*, potenzialmente espressi da due linguaggi distinti:

- linguaggio di **computazione**
- linguaggio di **coordinazione**
  - linguaggio di **comunicazione**



1.2 Teoria della computabilità

Problemi irrisolubili

Secondo la tesi di Church-Turing, non esiste un formalismo più potente della macchina di Turing, quindi se la MdT non riesce a risolvere un problema, quel problema è **irrisolubile**.

Per irrisolubile si intende che la MdT non si ferma, quindi non ritorna un output definito.

Problemi risolubili

Un problema risolubile è un problema la cui soluzione può essere calcolata da una MdT o equivalenti.

1.3 Funzioni

Per valutare la risolubilità di un problema è necessario individuare una **funzione** che lo descrive.

1.3.1 Funzione caratteristica

Come prima cosa si definisce la funziona **caratteristica** di un problema.

> Dato un problema P e detti
>
> - X l'insieme dei suoi dati in ingresso
> - Y l'insieme delle risposte corrette
>
> si dice funzione caratteristica del problema P
>
> $$f_p : X \to Y$$

In questo modo, problema non risolubile equivale a una funzione caratteristica non computabile.

1.3.2 Funzione computabile

Si formalizza ora la classe di funzioni dette **computabili**.

> Una funzione $f : A \to B$ è detta *computabile* se esiste una macchina di Turing

che, data sul nastro una rappresentazione di $x \in A$, dopo un *numero finito* di passi produce sul nastro una rappresentazione di $f(x) \in B$.

1.3.3 Funzioni definibili vs funzioni computabili

É necessario capire se tutte le funzioni sono computabili, o se esistono funzioni definibili ma non computabili.

Funzione computabile su N

Per semplicità, considereremo funzioni sui *numeri naturali*

$$f : N \to N$$

Questa condizione non è limitativa, dato che ogni informazione è necessariamente finita, può quindi essere codificata con una collezione di numeri naturali, la quale a sua volta può essere espressa tramite un *unico* numero naturale (procedimento di Gödel.

Procedimento di Gödel

Data una collezione di numeri naturali, ottenere un unico numero naturale.

- siano $N_1, N_2, \ldots, N_k$ i numeri naturali dati
- siano $P_1, P_2, \ldots, P_k$ i primi k numeri primi

Si definisce R, un nuovo numero naturale così definito

$$R ::= P_1^{N_1} \cdot P_2^{N_2} \cdot \cdots \cdot P_k^{N_k}$$

R rappresenta unicamente la collezione originale $N_1, N_2, \ldots, N_k$.

Poiché che l'insieme $F = \{f : N \to N\}$ delle funzioni sui naturali non è *enumerabile*, e l'insieme delle funzioni computabili è enumerabile (dato che l'insieme dei simboli è finito ogni MdT può essere associata a un numero di Gödel), la gran parte delle funzioni definibili non è computabile.

Interessano però soltanto le funzioni definibili con un linguaggio basato su un alfabeto finito di simboli, sfortunamtamento esistono funzioni definibili con tale alfabeto ma *non computabili*.

1.3.4 Funzioni non computabili

Dire che esistono funzioni definibili ma non computabili, equivale a dire che esistono problemi irrisolubili, basta trovare uno solo di questi problemi.

Problema dell'HALT della macchina di Turing

> Stabilire se una data macchina di Turing T, con un generico ingresso X, si ferma oppure no.

Slide 49-55.

1.3.5 Generabilità e decidibilità

Poiché un linguaggio è un insieme di frasi, è utile indagare la **generabilità** e la **decidibilità** di un insieme.

Si introduce il concetto di *insieme ricorsivamente numerabile*, per decidere che l'insieme sia effettivamente generabile.

Insieme ricorsivamente numerabile

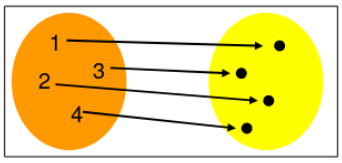
Data la definizione di un insieme numerabile, ovvero un insieme i cui elementi possono essere "contati" cioè che possiede una funzione biettiva:

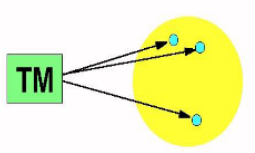
$$f : N \to S$$

che mette in corrispondenza i numeri naturali con gli elementi del sistema.

Si passa alla definizione di insieme **ricorsivamente numerabile** (*semi-decidibile*).

> Un insieme è ricorsivamente numerabile se la funzione biettiva $f : N \to S$ è *computabile*.





Tuttavia, il fatto che l'insieme possa essere costruito, non significa che si possa decidere se un certo elemeto vi appertiene o no.

Decidibilità

In generale, da un insieme ricorsivamente numerabile, potrebbe non essere trovabile un elemento, in questo caso la MdT può entrare in loop:

Un inisieme è **semi-decidibile** se è possibile stabilire se un elemento appartiene a un insieme ma non se un elemento *non appartiene*.

1.3.6 Insiemi decidibili

Occore un concetto più potente della semi-decidibilità.

Un insieme S è **decidibile** (o ricorsivo) se la sua funzione caratteristia è computabile.

$$f(x) = \begin{cases} 1, \text{se } x \in S \\ 0, \text{se } x \notin S \end{cases}$$

ovvero se esiste una macchina di Turing capace di rispondere sì o no, senza mai entrare in un *ciclo infinito*, alla domanda se un certo elemento appartiene all'insieme.

Teorema 1
Se un insieme è **decidibile** è anche **semi-decidibile**, ma non viceversa.

Teorema 2
Un insieme S è **decidibile** se e solo se

- S
- il suo complemento $N - S$

sono **semi-decidibili**.

2 — Linguaggi e grammatiche

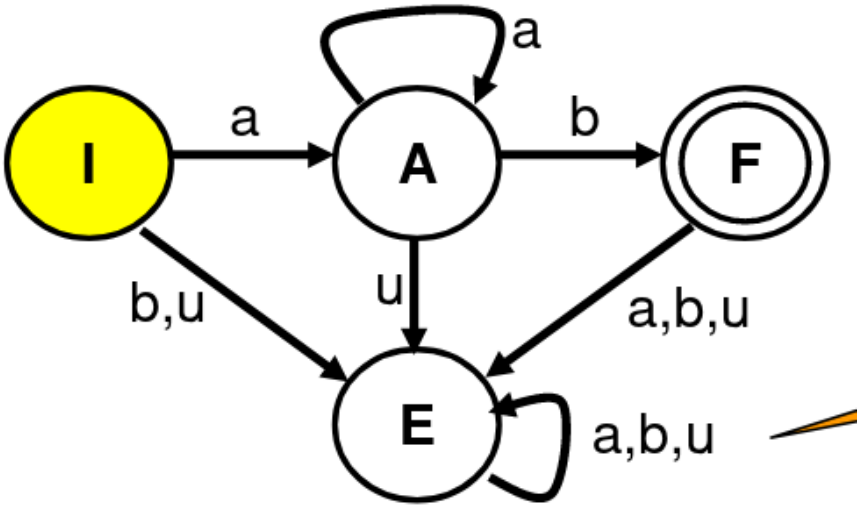
empty

3 — Riconoscitori a stati finiti

...

3.1 Dai riconoscitori ai generatori

Figura 3.1: Automa



Descrivendo a parole le transizioni utili dell'automa:

- nello stato I l'automa può accettare:
  - il simbolo a e portarsi nello stato A
- nello stato A l'automa può accettare:
  - il simbolo b e poi fermarsi
  - il simbolo a e riportarsi nello stato A stesso
- nello stato finale F l'automa puà accettare:
  - nessun simbolo

Se si sostituisce la parola accettare alla parola generare, si nota che l'automa può essere considerato come generatore.

Si può definire un **mapping** tra:

- stati $\longleftrightarrow$ simboli non terminali

- transizioni $\longleftrightarrow$ produzioni
- scopo $\longleftrightarrow$ uno stato particolare

É possibile automatizzare la costruzione di un RSF a partire dalla grammatica o viceversa.

Grammatica regolare a destra:

- scopo = stato iniziale: I
- stato finale: F
  - $S \to a\ A$
  - $A \to a\ A \mid b$

Grammatica regolare a sinistra:

- scopo = stato finale: F
- stato inziale: I
  - $S \to A\ b$
  - $A \to A\ a \mid a$

Lo stato finale F non si considera perché non ha archi uscenti.
Come arrivare a queste grammatiche?

3.1.1 Mapping RSF $\longleftrightarrow$ grammatica

Si immagini un osservatore che, stando in ogni stato, guardi da ogni stato:

- dove si va
  - la freccia col **simbolo terminale**
  - lo stato **successivo**
- da dove viene
  - lo stato **precendente**
  - la freccia col **simbolo terminale**

Mapping tra automa riconoscitore e grammatica

Tra la grammatica e il riconoscitori si riconoscono le seguenti corrispondenze:

- **stati** dell'automa $\longleftrightarrow$ **metasimboli** della grammatica
- **transizioni** dell'automa $\longleftrightarrow$ **produzioni** della grammatica
- **uno stato** dell'automa $\longleftrightarrow$ **scopo** della grammatica

Se la grammatica è regolare a destra si ottiene un automa top-down.
Se la grammatica è regolare a sinistra si ottiene un automa bottom-up.
In modo analogo si ottengono grammatiche destra/sinistra interpretando un RSF in modo top-down/bottom-up
Ad esempio:

In questo caso gli stati iniziale e finale sono anche stati di transito.

(a) Traduzione grammatica

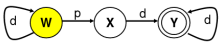
ESEMPIO di RSF

Sia:

- **A = { d, p, u }**
- **S = { W, X, Y, Z }**
- **S_0 = W**
- **F = { Y }**
- **sfn: A × S → S**

	d	p	u
W	W	X	Z
X	Y	Z	Z
Y	Y	Z	Z
Z	Z	Z	Z

(b) Riconoscitore



Per passare dall'automa alla grammatica si analizzano i collagamenti degli stati e si ricavano le trasformazioni:

$$W \to d\ W \mid p\ X \qquad Y \to X\ d \mid Y\ d$$
$$X \to d \mid d\ Y \qquad X \to p \mid W\ p$$
$$Y \to d \mid d\ Y \qquad W \to d \mid W\ d$$

3.1.2 Riconoscitori top down

Derivazione top-down

Si parte dallo scopo della grammatica e si tenta di coprire la frase data tramite produzioni successive.

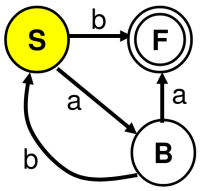
Data una grammatica regolare lineare a destra, il riconoscitore:

- ha tanti **stati** quanti i simboli non terminali
- ha come **stato iniziale** lo scopo S
- per ogni regola del tipo $X \to x\ Y$, l'automa con ingresso x passa dallo stato X allo stato Y
- per ogni regola del tipo $X \to x$, l'automa con ingresso x passa dallo stato X allo stato finale F

Esempio

Sia G una grammatica lineare a destra caratterizzata dalle seguenti produzioni:

- $S \to a\ B \mid b$
- $B \to b\ S \mid a$



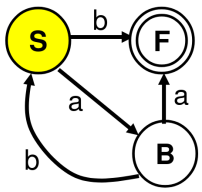
3.1.3 Riconoscitori bottom up

Data una grammatica regolare lineare a sinistra, il riconoscitore:

- ha tanti **stati** quanti i simboli non terminali
- ha come **stato finale** lo scopo S
- per ogni regola del tipo $X \rightarrow Y\ x$, l'automa con ingresso x passa dallo stato Y allo stato X
- per ogni regola del tipo $X \rightarrow x$, l'automa con ingresso x passa dallo stato iniziale I allo stato X

Esempio
Sia G una grammatica lineare a sinistra caratterizzata dalle seguenti produzioni:

- $S \rightarrow B\ a \mid b$
- $B \rightarrow S\ b \mid a$

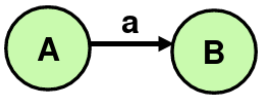


3.1.4 Dall'automa alle grammatiche

Dato un automa riconoscitore, se ne possono trarre sia una grammatica regolare a destra (interpretazione top-down) che una grammatica regolare a sinistra (interpretazione bottom-up).

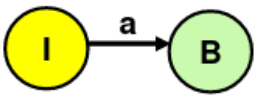
Caso generico

- top-down: $A \rightarrow a\ B$
- bottom-up: $A\ a \leftarrow B$



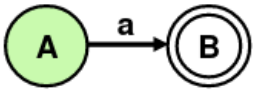
Caso iniziale

- top-down: $S \rightarrow a\ B$
- bottom-up: $I\ a \leftarrow B$

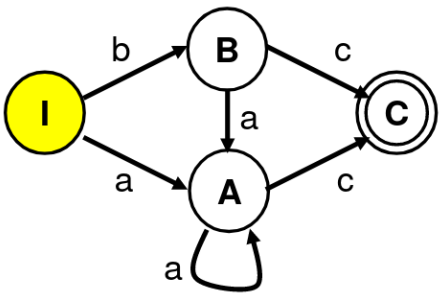


Caso finale

- top-down: $A \rightarrow a\ F$
- bottom-up: $A\ a \leftarrow S$



Esempio



Analisi top-down (C finale omesso):

- $I \rightarrow b\ B$
- $I \rightarrow a\ A$
- $B \rightarrow a\ A$
- $A \rightarrow a\ A$
- $B \rightarrow c$
- $A \rightarrow c$

Analisi bottom-up (C iniziale):

- $b \leftarrow B$
- $a \leftarrow A$
- $B\ a \leftarrow A$
- $A\ a \leftarrow A$
- $B\ c \leftarrow S$
- $A\ c \leftarrow S$

Grammatica G1 regolare a destra:

- $I \rightarrow b\ B \mid a\ A$
- $B \rightarrow a\ A \mid c$
- $A \rightarrow a\ A \mid c$

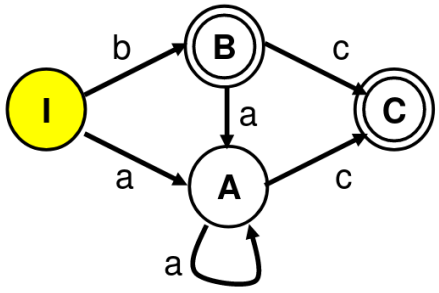
Grammatica G2 regolare a sinistra:

- $S \rightarrow B\ c \mid A\ c$
- $A \rightarrow B\ a \mid A\ a \mid a$
- $B \rightarrow b$

L(*G1*) =
b L(**B**) + **a** L(**A**) = (**ba+a**) L(**A**) + **bc** =
= (**ba+a**)**a*****c** + **bc** = **ba*****c** + **aa*****c** =
= (**b** + **a**) **a*** **c**

L(*G2*) =
L(**B**) **c** + L(**A**) **c** = (**b** + L(**A**)) **c** =
(**b** + (**b a** + **a**) **a***) **c** = (**b** + **a**) **a*** **c**
perché (b + b a a) = b a**

Esempio multipli stati finali



Analisi top-down (regola in più sulla I):

- $I \to b\ B \mid b$
- $I \to a\ A$
- $B \to a\ A$
- $A \to a\ A$
- $B \to c$
- $A \to c$

Analisi bottom-up (C iniziale):

- $b \leftarrow B$
- $a \leftarrow A$
- $B\ a \leftarrow A$
- $A\ a \leftarrow A$
- $B\ c \leftarrow C$
- $A\ c \leftarrow C$

L'analisi bottom-up deve scegliere quale scopo adottare, se B o C. Per risolvere questo problema si creano due grammatiche in cui si assumono a turno $C \equiv S$ e $B \equiv S$.

*Grammatica G2' (assume **C ≡ S**)*:
S → B c | A c
A → B a | A a | a
B → b
L(*G2'*) = (**b** + **a**) **a*** **c**

*Grammatica G2" (assume **B ≡ S**)*:
b ← S
le altre regole sono inutili, essendo i loro metasimboli irraggiungibili !
L(*G2"*) = **b**

Caso generale

Nel caso bottom-up, in presenza di più stati finali:

- si assume come (sotto)scopo S_k uno stato finale per volta
- si scrivono le regole bottom-up corrispondenti
- si esprime il linguaggio complessivo come unione dei vari sotto-linguaggi, definendo lo scopo globale come $S \to S_1 \mid S_2 \mid \cdots \mid S_n$

4 — Riconoscitori con PDA

Un RSF non può riconoscere un linguaggio di tipo 2, ha un limite intrinseco alla capacità di memorizzazione: non riesce a riconoscere frasi che richiedano di memorizzare una parte di lunghezza non nota a priori.

Esempio: bilanciamento delle parentesi $L = ({}^n c)^n, n \geq 0$, $G = S \to (S) \mid c$

In questo linguaggio il prefisso $({}^n$ non ha lunghezza limitabile a priori.

4.1 Push-Down Automaton (PDA)

Un PDA è un RSF con aggiunto uno stack, con stack non ci si riferisce a una struttura dati fisica ma a un suo modello astratto, ovvero una sequenza di simboli, definito in modo tale che si possa operare soltanto su quello in "cima".

Il PDA legge un simbolo d'ingresso e transita in un nuovo stato, in più a ogni passo altera lo stack, producendo una nuova configurazione.

Un PDA può prevedere ε-mosse, ovvero transizioni spontanee che manipolano lo stack senza consumare simboli in ingresso.

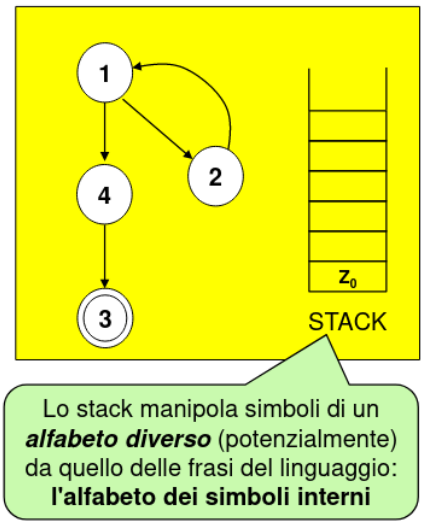
Un PDA è una sestupla:

$$\langle A,\ S,\ S_0,\ sfn,\ Z,\ Z_0 \rangle$$

dove:

- A = alfabeto
- S = insieme degli stati
- S_0 = stato iniziale $\in S$
- $sfn : (A \cup \varepsilon) \times S \times Z \to W$
- Z = alfabeto dei simboli interni
- $Z_0 \in Z$ = simbolo iniziale sullo stack

Figura 4.1: Struttura PDA



Il linguaggio accettatto da un PDA è definibile in 2 modi equivalenti:

- **Criterio dello stato finale**: il linguaggio accettato è l'insieme di tutte le stringhe di ingtesso che portano il PDA in uno degli stati finali.
- **Criterio dello stack vuoto**: il linguaggio accettato è definito come l'insieme di tutte le stringhe di ingresso che portano il PDA nella configurazione di *stack vuoto*.

La funzione sfn, dati:

- un *simbolo in ingresso* $a \in A$
- lo *stato attuale* $s \in S$
- il simbolo interno attualmente al *top dello stack* $z \in Z$

opera come segue:

- **consuma** il simbolo di ingresso a
- effettua il **POP** dello dallo stack, prelevando il simbolo z
- porta l'automa in stato futuro $s' = sfn(a, s, z)_{\prod S}$
- effettua una **PUSH** sullo stack di zero o più simboli interni $z' \in Z$
  $z' = sfn(a, s, z)_{\prod S}$

Si consideri il linguaggio generato da:

- $A = \{0, 1, c\}$

- $P = \{S \to 0\ S\ 0 \mid 1\ S\ 1 \mid c\}$
- $L = \{\texttt{word}\ c\ \texttt{word}^R\}$

Definiamo il PDA come segue:

- $A = \{0, 1, c\}$
- $S = \{Q1 = S_0, Q2\}$
- $Z = \{Zero, Uno, Centro\}$

A ∪ ε	S	Z	S × Z*
0	Q1	Centro	Q1 × CentroZero
1	Q1	Centro	Q1 × CentroUno
c	Q1	Centro	Q2 × Centro
0	Q1	Zero	Q1 × ZeroZero
1	Q1	Zero	Q1 × ZeroUno
c	Q1	Zero	Q2 × Zero
0	Q1	Uno	Q1 × UnoZero
1	Q1	Uno	Q1 × UnoUno
c	Q1	Uno	Q2 × Uno
0	Q2	Zero	Q2 × ε
1	Q2	Uno	Q2 × ε
ε	Q2	Centro	Q2 × ε

4.1.1 PDA non deterministici

Anche un PDA può essere non deterministico: in tal caso, la funzione sfn produce insiemi di elementi W (W sottoinsieme finito di $S \times Z^*$).

Ad esempio, il PDA tale che $sfn(Q_0, a, Z) = \{(Q_1, Z_1)(\cdots)(Q_k, Z_k)\}$ è **non deterministico** in quanto l'automa, nello stato Q_0, con simbolo interno in cima allo stack Z e ingresso a ha più di uno stato futuro possibile in base alla evoluzione cambia anche il se di simboli da porre nello stack.

> **Teorema**
> La classe dei linguaggi riconosciuti da un PDA non-deterministico coincide con la classe dei linguaggi context-free: perciò qualunque linguaggio context-free può sempre essere riconosciuto da un opportuno PDA.

É possibile rinunciare al determinismo?
In generale no:

Teorema
Esistono linguaggi context-free riconoscibili soltanto da PDA non-deterministici.

ma in molti casi di interesse pratico esistono linguaggi context-free riconoscibili da PDA deterministici: linguaggi context-free deterministici.

4.1.2 PDA deterministici

Cosa serve per ottenerlo?

Viste le condizioni precedenti, non deve succedere che l'automa, in dato stato Q_0, con simbolo in cima allo stack Z e ingresso x possa:

- portarsi in più stati futuri $sfn(Q_i, x, Z) = \{(Q_1, Z_1), (\cdots), (Q_k, Z_k)\}$
- optare se leggere o non leggere il simbolo di ingresso x a causa della presenza di entrambe le mosse $sfn(Q_i, x, Z)$ e $sfn(Q_i, \varepsilon, Z)$

Unendo, intersecando o concatenando linguaggi deterministici, non necessariamente si ottiene un linguaggio deterministico.

I complemento di un linguaggio è deterministico.

Con L linguaggio deterministico e R linguaggio regolare, il linguaggio quoziente L/R (insieme di stringhe di L private di suffisso regolare) è deterministico.

Con L linguaggio deterministico e R linguaggio regolare, il concatenamento $L.R$ (insieme di stringhe di L cun suffisso regolare) è deterministico.

Sottoclassi particolari

Per un PDA **deterministico**:

- il criterio dello stack vuoto risulta meno potente del criterio stati finali
- una limitazione sul numero di stati intenro o sul numero di configurazioni finali riduce l'insieme dei linguaggi riconoscibili
- l'assenza di ε-mosse riduce l'insieme dei linguaggi riconoscibili

4.1.3 Realizzazione di PDA deterministici

Possibilità di seguire la definzione, ma non molto pratico.

Si adotta un approccio che manipoli uno stack con la stessa logica di un PDA, dove lo stack è la vera differenza, ad esempio una macchina virtuale che abbia uno stack può essere fatta funzionare come PDA, opportunamente pilotata.

Si potrebbe controllare "a mano" lo stack, oppure in modo automatico attraverso le chiamate ricorsive di funzioni, dove sono già gestiti stack relativi alla ricorsione.

Top-Down Recursive-Descent Parsing

Con l'**analisi ricorsiva discendente** si introduce una funzione per ogni metasimbolo della grammatica, e la si chiama ogni volta che si icontra quel metasimbolo.

Ogni funzione copre le regole di quel metasimbolo, ossia riconosce il sotto-linguaggio corrispondente:

- termina normalmente (o segno di successo) se incontra simboli coerenti
- abortisce (o restituisce un segno di fallimento) se incontra simboli non coerenti

Esempio

Il solito linguaggio $L = \{\texttt{word}\ c\ \texttt{word}^n\}$, alfabeto $A = \{0,\ 1,\ c\}$ e regole $S \to 0\ S\ 0 \mid 1\ S\ 1 \mid c$.

- Introdurre tante funzioni quanti i metasimboli, qui una sola $S()$.
- Chiamare una funzione ogni volta che si incontra il suo metasimbolo
- Ogni funzione deve coprire le regole di quel metasimbolo
  - se il simbolo d'ingresso è $0 \to$ seguire la prima regola
  - se il simbolo d'ingresso è $1 \to$ seguire la seconda regola
  - se il simbolo d'ingresso è c $\to$ seguire la terza regola

Nel caso della prima regola, consumiamo il carattere di ingresso 0, invochiamo la funzione $S()$ e consumiamo un nuovo carattere d'ingresso e verifichiamo che sia 0.

Se la verifica ha esito positivo, significa che la funzone ha incontrato simboli coerenti con le proprie regole.

Se la verifica ha esito negativo, significa che la funzione ha incontrato simboli che non corrispondono alle sue regole.

4.1.4 Separare motore e grammatica

Applicare l'analisi ricorsiva discendente è un processo meccanico, che tuttavia introduce informazioni cablate nel codice, difficili da aggiornare.

É possibile separare il **motore**, invariante rispetto alle regole, dalle regole della grammatica; si presta a questo scopo la tabella di parsing, simile alla tabella delle transizioni di un RSF, indica però la prossima *produzione* da applicare.

Il motore prenderà singole decisioni consultando questa tabella.

Parsing tables - esempi deterministici

Linguaggio $L = \{\texttt{word}\ c\ \texttt{word}^n\}$

	0	1	c
S	S → 0 S 0	S → 1 S 1	S → c

Linguaggio $L = \{\texttt{if c then cmd (endif | else cmd) }\}$

Produzioni $S \rightarrow \texttt{if c then cmd X}$, $\texttt{X} \rightarrow \texttt{endif} \mid \texttt{else cmd}$

	if	c	then	endif	else	cmd
S	S → if c then cmd X	*error*	*error*	*error*	*error*	*error*
X	*error*	*error*	*error*	X → endif	X → else cmd	*error*

4.1.5 Analisi ricorsiva discendente - vantaggi e limiti

Vantaggi

- immediata scrittura del riconoscitore
- migliore leggibilità a modificabilità del codice
- facilitata inserzione di azioni nella fase di analisi

Svantaggi

- non sempre applicabile
- funzionale solo se non esistono ambiguità sulla regola da applicare

Si individua una sottoclasse di grammatiche context-free che garantisce il determinismo dell'analisi sintattica.

Per rendere deterministica l'analisi ricorsiva, si rende necessario avere una visione del *passato* dell'analisi (simboli consumati fino a quel punto) e una del *futuro*, generalmente un solo carattere in avanti.

4.2 Grammatiche $LL(k)$

Si definiscono *grammatica* $LL(k)$ quelle che sono analizzabili in modo deterministico:

- procedendo *left to right*
- applicando *left-most derivation*

- guardando avanti al più k simboli

Ricoprono un posto fondamentale le grammatiche LL(1), ovvero quelle in cui basta guardare un simbolo in avanti per operare in modo deterministico.

Esempio

Si consideri la grammatica

- VT = {p, q, a, b, d, x, y}
- VN = {S, X, Y}

Produzioni

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{p X} \quad | \texttt{q Y} \\ \texttt{X} &\rightarrow \texttt{a X b} \quad | \texttt{x} \\ \texttt{Y} &\rightarrow \texttt{a Y d} \quad | \texttt{y} \end{aligned}$$

Le parti **destre** delle produzioni di uno stesso meta simbolo, iniziano tutte con un simbolo terminale diverso, è sufficiente guardare avanti di un carattere per scegliere con certezza la produzione per scegliere con certezza la produzione con cui proseguire l'analisi.

Creando la parsing table, si nota che ogni cella contiene una sola produzione, quindi non si hanno ambiguità sulle prossime mosse da fare, è facile vedere che il parser è deterinistico.

	p	q	a	b	d	x	y
S	S → p X	S → q Y	*error*	*error*	*error*	*error*	*error*
X	*error*	*error*	X → a X b	*error*	*error*	X → x	*error*
Y	*error*	*error*	Y → a Y d	*error*	*error*	*error*	Y → y

La frase di inzio deve essere completa oppure può essere parziale? Nel caso in cui si voglia imporre che la frase deve essere finale occorre imporre una regola *top-level* che specifica che la frase deve terminare con $, carattere che rappresenta il fine stringa/linea/file del tipo Z → S $.

4.2.1 Starter Symbol Set

Spesso le parti destre delle produzioni di uno stesso metasimbolo non iniziano tutte con un simbolo terminale, non è chiaro quali siano gli input ammissibili.

Occorre ridefinire il concetto di simbolo iniziale → Starter Symbols Set.

Lo starter symbols set della riscrittura α è l'insieme

$$SS(\alpha) = \{a \in \texttt{VT} \mid \alpha \rightarrow^* a\ \beta\}, \quad \texttt{con}\ \ \alpha \in V^+\ \ \texttt{e}\ \ \beta \in V^*$$

In sostanza gli starter symbols sono le iniziali di una forma di frase α, ricavate applicando con più produzioni. L'operatore $*$ cattura il caso limite in cui $\alpha \in VT$, ossia già terminale, e non richiede di applicare derivazioni.

Per includere anche il caso $\alpha \to \varepsilon$, si introduce l'insieme

$$\texttt{FIRST}(\alpha) = \texttt{trunc}_1(\{x \in VT^* \mid \to^*\}), \ \texttt{con} \ \alpha \in V^*$$

dove $\texttt{trunc}_1$ denota il troncamento della stringa al primo elemento.

Generalizzando la regola precendete, condizione necessaria perché una grammatica sia $LL(1)$ è che gli *start symbols* relativi alle parti destre di uno stesso metasimbolo siano disgiunti.

É possibile capire in modo più rapido se una grammatica è $LL(1)$?

Due opzioni:

- agire sulla parsing table, formalizzando il concetto di **blocco annullabile** e integrando nella tabella l'informazione sulle stringhe che possono scomparire.
- ampliare la nozione di starte symbols set con i Director Symbols set (o Look-Ahed set)

4.2.2 Parsing table con blocchi annullabili

Un **blocco annullabile** è una stringa che può *degenerare* in ε.

In presenza di blocchi annullabili, un metasimbolo che non appare iniziale, può trovarsi a inizio frase, per tenere conto delle ε-rules è necessario considerare anche i simboli che possono *seguire* quellli annullabili, l'insieme **FOLLOW**.

Un blocco annullabile deve essere previsto dove la sua presenza non appare evidente, ovvero in corrispondenza dei terminali che possono *seguire* il blocco annullabile.

Esempio

$$\begin{aligned}
&\texttt{S} \to \texttt{A B} \\
&\texttt{A} \to \texttt{P Q} \mid \texttt{B C} \\
&\texttt{P} \to \texttt{p P} \mid \varepsilon \\
&\texttt{Q} \to \texttt{q Q} \mid \varepsilon \\
&\texttt{B} \to \texttt{b B} \mid \texttt{d} \\
&\texttt{C} \to \texttt{c C} \mid \texttt{f}
\end{aligned}$$

Il blocco `PQ` è annullabile, perché sia `P` che `Q` possono degenerare in ε.

La regola A → P Q va inclusa in tutte le colonne della riga A che possono seguire A perché potrebbero avere un PQ invisibile davanti.

	p	q	b	c	d	f
S	S → A B	S → A B	S → A B	*error*	S → A B	*error*
A	A → P Q	A → P Q	A → B C A → P Q	*error*	A → B C A → P Q	*error*
B	*error*	*error*	B → b B	*error*	B → d	*error*
C	*error*	*error*	*error*	C → c C	*error*	C → f
P	P → p P	P → ε	P → ε	*error*	P → ε	*error*
Q	*error*	Q → q Q	Q → ε	*error*	Q → ε	*error*

4.2.3 Director Symbols Set

L'idea è quella di ampliare la nozione di Starte Symbols, definendo un nuovo insieme che integri a priori l'effetto delle ε-rules.

Si crea un nuovo insieme caratterizzante, integrato dal nuovo insiem **Follow** quando qualche produzione genera la stringa vuota (o uguale allo SSS senza stringhe vuote).

Il *Director Symbols set* della produzione della $\mathtt{A} \to \alpha$ è l'unione di due insiemi:

- lo Start Symbols set
- il **Following Symbols set**

$$\mathtt{DS}(\mathtt{A} \to \alpha) = \mathtt{SS}(\alpha) \cup \mathtt{FOLLOW}(\mathtt{A}) \text{ se } \to^* \varepsilon$$

dove FOLLOW(A) denota l'insieme dei simboli che possono seguire la frase generate da A:

$$\mathtt{FOLLOW}(\mathtt{A}) = \{\mathtt{a} \in \mathtt{VT} \mid \mathtt{S} \to^* \gamma \mathtt{A} \mathtt{a} \beta\} \text{ con } \gamma, \beta \in \mathtt{V}^*$$

In pratica

$$\mathtt{DS}(\mathtt{A} \to \mathtt{a}) = \begin{cases} \mathtt{SS}(\alpha) \text{ se } \alpha \text{ non genera mai } \varepsilon \\ \mathtt{SS}(\alpha) \cup \mathtt{FOLLOW}(\alpha) \text{ se } \alpha \text{ può generare } \varepsilon \end{cases}$$

o anche:

$$\mathtt{DS}(\mathtt{A} \to \alpha) = \mathtt{trunc}_1(\mathtt{FIRST}(\alpha) \cdot \mathtt{FOLLOW}(\mathtt{A}))$$

ossia quello che si ottiene prendendo l'iniziale delle frasi ottenute concatenando FIRST(α) con ciò che segue A.

Questo insieme permette di formulare la condizione LL(1):

> Condizione necessaria e sufficiente perché una grammatica sia LL(1) è che i Director Symbols set relativi a produzione *alternativa* siano **disgiunti**.

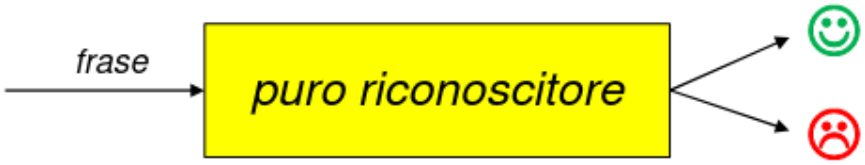
Questi set sono spesso chiamati **Look-Ahead set** perché servono per "guardare avanti" per guidare deterministicamente il parser.

5 — Dai riconoscitori agli interpreti

Dai puri riconoscitori...

Finora si sono considerati soltanto *puri riconoscitori*, che:

- accettano in ingresso una stringa di caratteri
- riconoscono se essa appartiene a un linguaggio



La risposta è sempre di tipo booleano, non ha importanza *come* si arriva a stabilire la correttezza.

...agli interpreti

Un **interprete** è più di un puro riconoscitore, riconosce una stringa ma esegue anche azioni in base al significato (*semantica*) della frase analizzata.

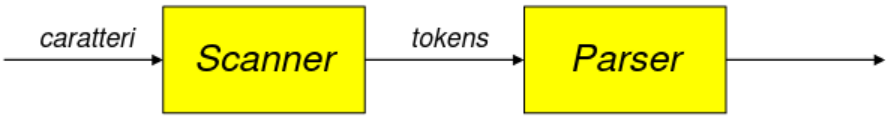
In questo caso diventa importante la sequenza di derivazione.

5.1 Interprete

5.1.1 Struttura

Un interpreste è solitamente basasto su due componenti:

- analizzatore lessicale, **scanner** o *lexer*
- analizzatore sintattico-semantico, **parser**



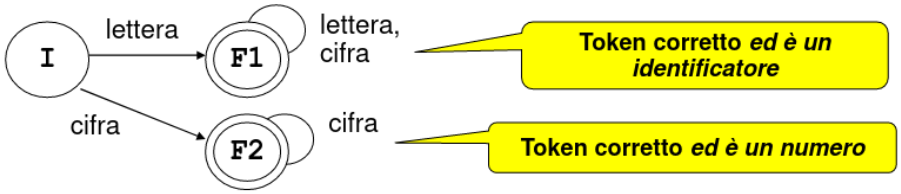
Lo scanner analizza le parti *regolari* dei linguaggi, fornendo al parser singoli token già aggregati.

Il parser riceve dallo scanner i token, considerati come elementi terminali del suo linguaggio per valutare la correttezza della loro sequenza: opera sulle parti context free del linguaggio.

5.1.2 Analisi lessicale

L'analisi lessicale è la fase in cui si individuano le singole parole (token) che compongono la frase. Questa azione viene svolta raggruppando i singoli caratteri dell'input secondo **produzioni regolari** associate alle diverse possibili **categorie** lessicali.

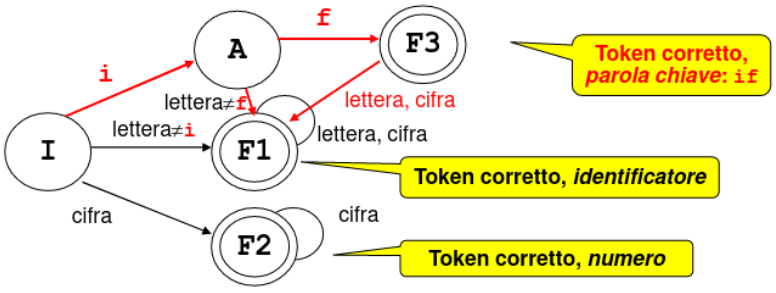
L'analizzatore lessicale categorizza i token osservando in quale stato finale del RSF si ritrova.



5.1.3 Parole chiave

Cablare ogni dettaglio nella struttura del RSF non è una buona strategia in quanto renderebbe estremamente complicato il riconoscitore.

Figura 5.1: Esempio riconoscimento identificatore if



5.1.4 Tabelle

Per evitare di cablare le parole chiave, simboli etc. nella struttura del RSF, conviene agire diversamente:

- categorizzare prima le parole chiave come identificatori
- ricategorizzare poi consultando opportune tabelle che incapsulano il dettaglio del linguaggio
  - tabella *parole chiave*
  - tabella *simboli*

5.1.5 Analisi sintattica top-down

In caso di grammatiche $LL(1)$, una tecnica semplice per costruire il riconoscitore è l'analisi top-down ricorsiva discendente.

Per passare da un puro riconoscitore a un interprete occorre propagare qualcosa di più di un si o no, come ad esempio un **albero**, per una valutazione differita.

5.2 Caso di studio - espressioni aritmetiche

Si supponga di voler riconoscere espressioni aritmetiche con le quattro operazioni + * - /.

```
3+4-5     3+4*5     9-4-1     9-4/2
```

Un puro riconoscitore deve solo dire se sono corrette, ma un interprete deve anche dire:

- se il dominio sono gli *interi* il risultato può essere un valore int
- se il dominio sono i *reali* il risultato può essere un valora double
- se l'obiettivo è la *valutazione differita*, il risultato può essere un opportuno oggetto (albero)

5.2.1 Analisi del dominio

Sintassi

La notazione classica insegnata identifica i quattro operatori con i seguenti simboli: $+$, $-$, $\times$, $:$.

Sono stati sostituiti dagli informatici con $+$, $-$, $*$, $/$, spesso vengono anche usate le parentesi per esprimere una priorità associativa.

Semantica

Nel dominio aritmetico usuale:

- i *valori numerici* si assumono espressi in notazione **posizionale** su base dieci
- il significato inteso dei quattro *operatori* è quello di somma, sottrazione, moltiplicazione, divisione
- si denotano le nozioni di priorità e associatività
  - **priorità** fra operatori diversi, moltiplicativi prioritari su quelli additivi
  - **associatività** tra operatori equiprioritari, solitamente si associa a sinistra

5.2.2 Grammatica per le espressioni

Consideriamo il linguaggio $E(G)$ relativo alla seguente grammatica per espressioni aritmetiche;

$\texttt{VN} = \{\texttt{EXP}\}$
$\texttt{VT} = \{+, *, -, :, \texttt{num}\}$
$S = \texttt{EXP}$
$P = \{$
$\quad \texttt{EXP} := \texttt{EXP} + \texttt{EXP}$
$\quad \texttt{EXP} := \texttt{EXP} - \texttt{EXP}$
$\quad \texttt{EXP} := \texttt{EXP} * \texttt{EXP}$
$\quad \texttt{EXP} := \texttt{EXP} : \texttt{EXP}$
$\quad \texttt{EXP} := \texttt{num}$
$\}$

É una grammatica ambigua, la semantica è informale: se `EXP` è un numero, l'espressione denota un intero e il valore dell'espression coincide con quello del numero.

5.2.3 Una grammatica a "strati"

É possibile dare una struttura *gerarchica* alle espressioni, esprimendo così intrinsicamente priorità e associatività degli operatori.

$\texttt{VN} = \{\texttt{EXP}, \texttt{TERM}, \texttt{FACTOR}\}$
$\texttt{VT} = \{+, *, -, :, (,), \texttt{num}\}$
$S = \texttt{EXP}$
$P = \{$
$\quad \texttt{EXP} := \texttt{TERM}$
$\quad \texttt{EXP} := \texttt{EXP} + \texttt{TERM}$
$\quad \texttt{EXP} := \texttt{EXP} - \texttt{TERM}$

TERM := FACTOR

TERM := TERM ∗ FACTOR

TERM := TERM : FACTOR

FACTOR := num

FACTOR := (EXP)

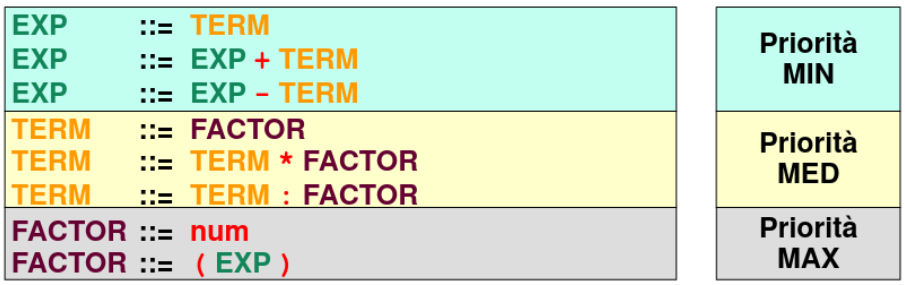
}

Ogni strato considera *terminali* gli elementi linguistici definiti in altri strati:

- EXP considera terminali +, − e TERM
- TERM considera terminali ∗, : e FACTOR
- FACTOR considera terminali num, (,), e EXP

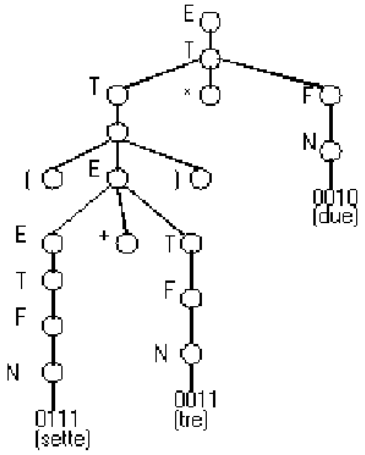
Le *somme* e *sottrazioni* aggregano i termini, le *moltiplicazioni* e *divisioni* aggregano i fattori e i *fattori* sono entità atomiche.

Figura 5.2: Priorità operatori

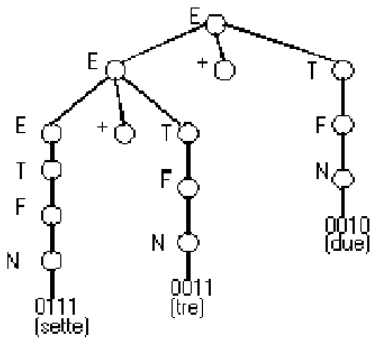


Esempi

Frase:
$(0111 + 0011) * 0010$
Albero di derivazione



Frase:
$0111 + 0011 + 0010$
Albero di derivazione



La grammatica individuata è *ricorsiva a sinistra* per esprimere la corretta associatività, tuttavia è incompatibile con l'analisi ricorsiva discendente, utile per la costruzione del parser.

5.2.4 Variante 1 - Associativa a destra

Se si riscrivono le regole in forma ricorsiva a destra, ovvero invertendo `TERM` e `EXP` e `FACTOR` e `TERM`, si otterrebbe una associatività inversa rispetto a quella usuale.

Ad esempio l'espressione $7 - 3 - 2$ verrebbe derivata come $7 - (3 - 2)$ anziché come $(7 - 3) - 2$.

$\texttt{VN} = \{\texttt{EXP}, \texttt{TERM}, \texttt{FACTOR}\}$

$\texttt{VT} = \{+, *, -, :, (,), \texttt{num}\}$

$P = \{$

$\texttt{EXP} := \texttt{TERM}$

$\texttt{EXP} := \texttt{TERM} + \texttt{EXP}$

$\texttt{EXP} := \texttt{TERM} - \texttt{EXP}$

$\texttt{TERM} := \texttt{FACTOR}$

$\texttt{TERM} := \texttt{FACTOR} * \texttt{TERM}$

$\texttt{TERM} := \texttt{FACTOR} : \texttt{TERM}$

$\texttt{FACTOR} := \texttt{num}$

$\texttt{FACTOR} := (\texttt{EXP})$

$\}$

5.2.5 Variante 2 - Non associativa

Si potrebbe anche fare a meno dell'associatività, mantenendo però lo stesso ordine di priorità. Queste regole non usano ricorsione, ne destra ne sinistra, risolvendo il problema *obbligando a usare le parentesi*.

$\texttt{VN} = \{\texttt{EXP}, \texttt{TERM}, \texttt{FACTOR}\}$

$\texttt{VT} = \{+, *, -, :, (,), \texttt{num}\}$

$P = \{$

$\texttt{EXP} := \texttt{TERM}$

$\texttt{EXP} := \texttt{TERM} + \texttt{TERM}$

$\texttt{EXP} := \texttt{TERM} - \texttt{TERM}$

$\texttt{TERM} := \texttt{FACTOR}$

$\texttt{TERM} := \texttt{FACTOR} * \texttt{FACTOR}$

$\texttt{TERM} := \texttt{FACTOR} : \texttt{FACTOR}$

$\texttt{FACTOR} := \texttt{num}$

$\texttt{FACTOR} := (\texttt{EXP})$

$\}$

5.3 Dalla grammtica al parser

5.3.1 Ricorsione sinista e analisi top-down

Il riconoscitore è un PDA, perché la grammatica ha self-embedding, il problema è che la sintassi delle espressioni include produzioni *ricorsive sinistra*, quindi questa non è una grammatica $LL(1)$.

Trasformazione della grammatica

Si procede riconsiderando i sotto-linguaggi generati dai diversi strati:

$\texttt{L(EXP)} = \texttt{TERM} \pm \texttt{TERM} \pm \texttt{TERM} \ldots$

$\texttt{L(TERM)} = {}^*/{:}\texttt{FACTOR}{}^*/{:}\texttt{FACTOR} \ldots$

$\texttt{L(FACTOR)} = \texttt{num} \mid \texttt{(EXP)}$

Continuiamo mappando le espressioni individuate seconda la notazione EBNF:

$\texttt{EXP} = \texttt{TERM}\ \{(+|-)\texttt{TERM}\}$

$\texttt{TERM} = \texttt{FACTOR}\ \{(*|\ :)\texttt{FACTOR}\}$

Queste regole non presentano più ricorsione esplicita, descrivono un processo computazionale iterativo.

Questa è una grammatica $LL(1)$, in quanto è analizzabile con una tecnica ricorsiva discendente e hanno start symbols set distinti.

Si può esplicitare inserendo un termine:

$\texttt{EXP} = \texttt{TERM AFTERTERM}$

$\texttt{AFTERTERM} = \varepsilon \mid +\texttt{EXP} \mid -\texttt{EXP}$

$\texttt{TERM} = \texttt{FACTOR AFTERFACTOR}$

$\texttt{AFTERFACTOR} = \varepsilon \mid *\texttt{TERM}|\ :\texttt{TERM}$

Questa è una grammatica $LL(1)$ ma è anche ricorsiva a destra, non verrà quindi utilizzata operativamente ma solo per la verifica $LL(1)$

Schema parser

Si effettua una analisi ricorsiva discendente:

- procedura o funzione per ogni simbolo non terminale
- invocazione ricorsiva solo per il caso con self-embedding

Tutte le funzioni restituiscono un *boolean*, nel caso di puro riconoscitore, oppure un valore o oggetto nel caso di parser completi con valutazione.

Ogni funzione dovrebbe terminare quando incontra un simbolo che non appartiene al sotto-linguaggio di sua pertinenza.

```
public boolean parseExp() {
    boolean t1 = parseTerm();
    while (currentToken != null) {
        if (currentToken.equals("+"))
```

inserisci codice slide 37-39

Una architettura di sistema

Si passa ora a definira una architettura di supporto per il parser, come ad esempio concretizzare il concetto di **token** e astrarre lo scanner per leggere la stringa e definire il modello di interazione tra parser e scanner.

Si potrebbe, ad esempio, utilizzare una variabile privata `currentToken`, una classe `MyScanner` per tokenizzare la stringa e una classe `Token` che incapsula una stringa.

inserire codice 42-46

5.4 Dal parser al valutatore

Finora abbiamo definito come:

- dato il *linguaggio* desiderato, trovare una *grammatica* adatta a descriverlo
- data la grammatica, scrivere un puro *riconoscitore* per tale linguaggio

manca:

- data la grammatica, scrivere un **parser** completo che effettui una **valutazione**

serve:

- la specifica della semantica che il parser dovrà applicare

5.4.1 Specificare la semantica

É necessario un modo sistematico e formale stabilire con precisione il **significato** di ogni possibile frase di un linguaggio: se il linguaggio è *infinito* serve una notazione finita applicabile a infinite frasi.

Un metodo può essere qello di definire una *funzione di interpretazione*:

- dominio, ovvero il linguaggio
- codominio, ovvero l'insieme dei possibili significati

5.4.2 Semantica denatoziale

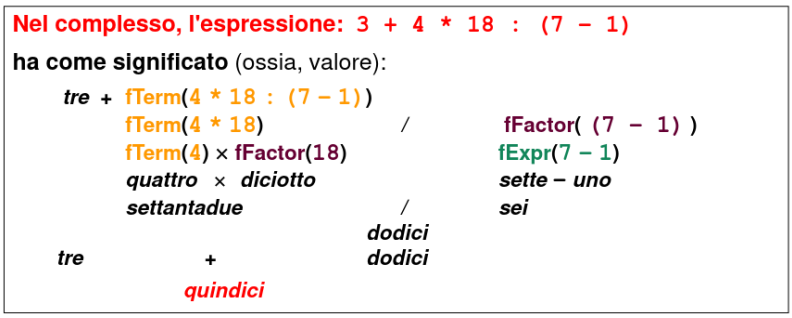
Quando la semantica di un lingauggio è espressa in questo modo si parla di **semantica denotazionale**.

L'idea principale è quella di associare a ogni *regola sintattica* una **regola semantica**.

Nel caso delle espressioni aritmetiche la sintassi prevede `Exp`, `Term` e `Factor`, quindi la semantica prevederà le funzioni `fExp`, `fTerm` e `fFactor`.

- `fExpr(s)`
  - `fTerm(s)` se `s` non contiene ne + ne −
  - `fExpr(s1)`+`fTerm(s2)`, se `s` ha la forma `s1+s2`
  - `fExpr(s1)`−`fTerm(s2)`, se `s` ha la forma `s1−s2`
- `fTerm(s)`
  - `fFactor(s)` se `s` non contiene ne ∗ ne :
  - `fTerm(s1)`×`fFactor(s2)` se `s` ha la forma `s1∗s2`
  - `fTerm(s1)`/`fFactor(s2)` se `s` ha la forma `s1:s2`
- `fFactor(s)`
  - `fFactor(innerExp)` se `s` ha la forma `(innerExp)`
  - `valueof(num)`, se `s` ha la forma `num`

Esempio



Schema interprete

Ogni funzione analizza il sotto-linguaggio di pertinenza:

- `parseExp` analizza `L(EXP)`, per il quale $+$, $-$ e `TERM` sono l'alfabeto terminale
- `parseTerm` analizza `L(TERM)`, per il quale $*$, : e `FACTOR` sono l'alfabeto terminale
- `parseFactor` analizza `L(FACTOR)`, il cui alfabeto terminale è costituito da (,) e `EXP`

inserisci codice slide 64-67

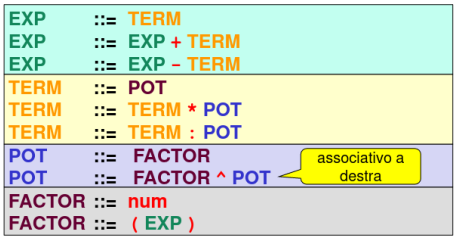
5.4.3 Elevamento a potenza

Se si volesse aggiungere l'elevamento a potenza?

Si introduce l'operatore `^`, che denota un elevamento a potenza: $3 * 4^2$ si scrive `3*4^2`

Questo operatore ha priorità maggiore di tutti gli altri e si considera la sua associatività è a destra, ovvero si interpreta 4^{2^3} o `4^2^3` come 4^8.

Si aggiunge un nuovo "strato" alla grammatica a strati definita in precedenza:



inserisci codice slide 70-72

5.5 Rappresentare le frasi

Manca soltanto una rappresentazione intermedia della frase interpretata per poter "risolverla" successivamente, qui entrano in gioco gi **alberi sintattici**.

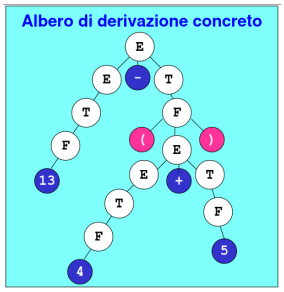
5.5.1 Alberi sintattici astratti

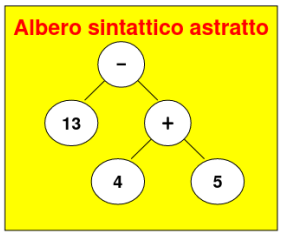
Si potrebbe usare dirattamente l'*albero di derivazione*, ma darebbe luogo a una rappresentazione ridondante.

Conviene adottare un albero più *compatto*, che contenga solo i nodi indispensabili, la soluzione è il **Abstract Syntax Tree** (AST).

Tipologie di nodi non indispensabili:

- i nodi terminali (foglie) non legati a niente di significativo sono ridondanti e possono essere ignorati, le foglie
- le foglie che esauriscono la loro funzione dopo il parser, nelle espressioni le parentesi
- le foglie che hanno un unico nodo figlio





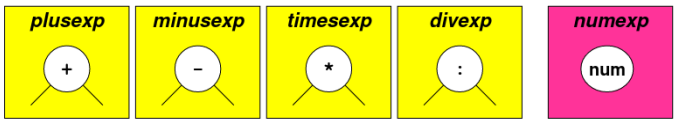
Esempio espressioni

Nel caso delle espressioni, l'AST è così definito

- ogni **operatore** è un nodo con due figli
  - figlio sinistro è il primo operando
  - figlio destro è il secondo operando
- i valori numerici sono le foglie

La rappresentazione è sempre univoca, quindi la struttura dell'albero fornisce intrinsicamente l'ordine corretto di valutazione.

Si tenta ora di estendere il parser per generare l'opportuno AST, per farlo è necessario stabilire quali e quanti tipi di nodo diversi si hanno.



5.5.2 Sintassi astratta

Per descrivere formalmente questi nodi si utilizza una **sintassi astratta**, ad esempio i nodi in considerazione possono essere rappresentanti con la seguente sintassi astratta

$$\texttt{EXP} = \texttt{EXP} + \texttt{EXP}$$
$$\texttt{EXP} = \texttt{EXP} - \texttt{EXP}$$
$$\texttt{EXP} = \texttt{EXP} * \texttt{EXP}$$
$$\texttt{EXP} = \texttt{EXP} : \texttt{EXP}$$
$$\texttt{EXP} = \texttt{num}$$

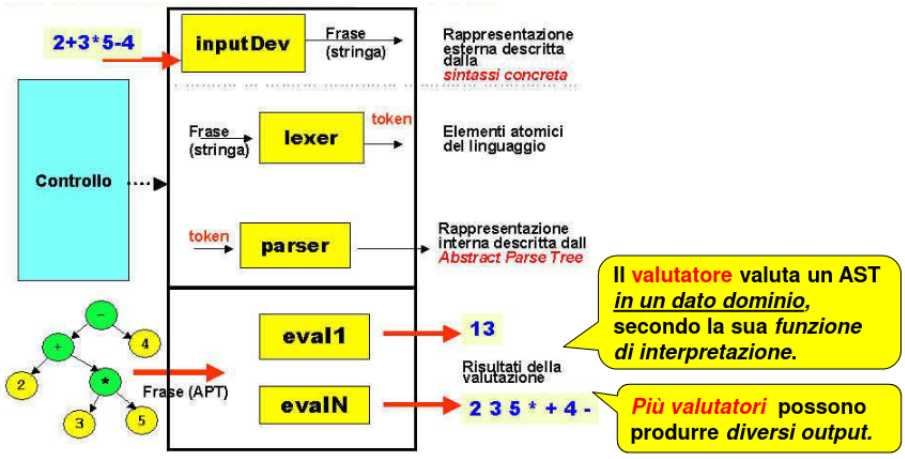
Una possibile implementazione dell'AST

inserisci codice slide 86-89

Si passa ora alla nuova implementazione del parser che include la nuova architettura ad AST.

inserisci codice slide 91-93

5.6 Architettura interprete



Si è arrivati al punto nel quale sono state definite strategie per:

- dato il linguaggio desiderato, trovare una *grammatica* adatta
- data la grammatica, scrivere il *puro riconoscitore* per il linguaggio

- data la grammatica, scrivere un *parser* completo che valuti e generi l'albero

manca da **valutare l'albero sintattico** dopo la sua generazione.

5.7 Valutazione di alberi

Esistono diverse modalità per analizzare una albero, la teoria degli alberi introduce il concetto di visita:

- *pre-order*, radice → figli (da sx a dx)
- *post-order*, figli (da sx a dx) →, radice
- *in-order*, figlio sx → radice → figlio dx

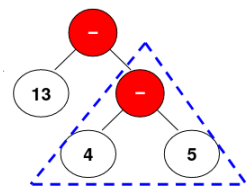
Nel nostro dominio delle espressioni aritmetiche, queste metodologie si traducono in:

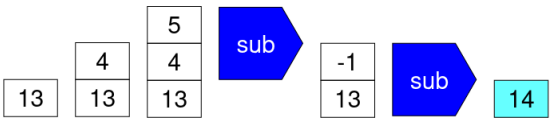
- notazione **prefissa**
- notazione **postfissa**
- notazione **infissa** classica

Con la notazione postfissa si presta perfettamente alla traduzione in codice per un elaboratore, in quanto fornisce prima gli operandi e successivamente cosa farne.

Idealmente si caricherebbero gli operandi su un registro, ma essendo questi limitati si utilizza lo **stack**.

Evoluzione dello stack:





5.8 Valutatore

Il valutatore incorpora la funziona di valutazione, che deve **visitare** l'albero applicando la corretta *semantica* a seconda del nodo visitato.

5.8.1 Implementazione

É necessario implementare un metodo `eval()` che calcola il valore di una espressione, conviene introdurre questo metodo nella classe astratta dell'AST, per poi specializzare la sua implementazione per ogni tipo di nodo.

```
public abstract class Exp {
   public abstract int eval();
}
```

Si utilizza quindi una metodologia completamente object oriented, senza creare una funzione piena di if..else per scegliere l'azione da eseguire, questo approccio comporta tuttavia la decentralizzazione della valutazione, con conseguente impatto in caso di necessità di modifiche.

Stile a oggetti

```
abstract class Exp {
    abstract public int eval();
}
...
class PlusExp extends OpExp {
    public PlusExp(Exp l, Exp r) {
        super(l,r);
    }
    public String myOp() {
        return "+";
    }
    public int eval() {
        return left.eval() + right.eval();
    }
}
...
class NumExp extends Exp {
    int val;
    public NumExp(int v) {
        val = v;
    }
    public int eval() {
        return val;
    }
```

}

Confronto

- Metodologia **funzionale**
  - *PRO*: facilitata l'introduzione di **nuove interpretazioni**, basta scrivere una nuova `eval2()`
  - *CONTRO*: rende più oneroso introdurre una **nuova produzione**, impatta tutte le funzioni di interpretazione
- Metodologia **object oriented**
  - *PRO*: facilita l'aggiunta di **nuove produzioni**, basta aggiungere nuova classe, con relative eval
  - *CONTRO*: rende oneroso introdurre **nuove intepretazioni**, impatta tutte le classi della tassonomia nell'introduzione del relativo metodo

Cosa scegliere

Solitamente un linguaggio di programmazione ha una *grammatica fissa* ma richiede *molteplici interpretazioni* delle frasi (per analisi della semantica, type checking, code generation etc.); per questo l'approccio funzionale sembrerebbe il più adatto.

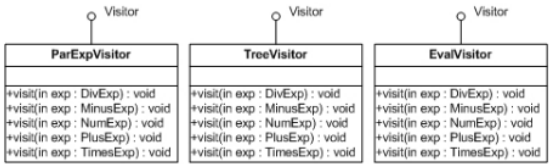
É necessario unire i pro delle due metodologie, a questo compito si presta perfettamente il **pattern Visitor**, che ci permette di incapsulare la *logica d'interpretazione* in una entità di livello più alto.

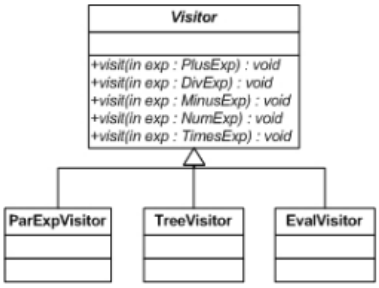
5.8.2 Visitor come interprete

Questo pattern realizza la logica di interpretazione in modo coerente all'approccio a oggetti: cattura **una** logica di interpretazione, centralizzandola in un unico posto.

Nel caso delle espressioni si possono realizzare

- una classe base astratta (o interfaccia) `Visitor`
- visitor per *stampare* le espressioni → `ParExpVisitor`
- visitor per *calcolare* le espressioni → `EvalVisitor`
- visitor per tradurre le espressioni sotto forma di *codice*
- ...





inserisci codice slide 162-173

6 — Estensione: assegnamenti, ambienti, sequenze

6.1 Assegnamento

Ogni linguaggio di programmazione introduce le nozioni di **variabili** e **assegnamento**.

L'assegnamento non è una uguaglianza o equazione, ma denota l'azione di prendere la destra dell'uguale e metterla all'interno della sinistra dell'uguale, considerando come valori le variabili sulla destra e contenitori le variabili sulla sinistra.

Si deduce che il simbolo di variabile ha significato diverso tra la destra e la sinistra dell'operatore.

6.1.1 L-Value vs R-Value

Si definisce quindi la distinzione tra l-value e r-value:

- **l-value**: il significato della variabile a sinistra è la variabile *in quanto tale*
- **r-value**: il significato della variabile a destra è il *contenuto* della variabile

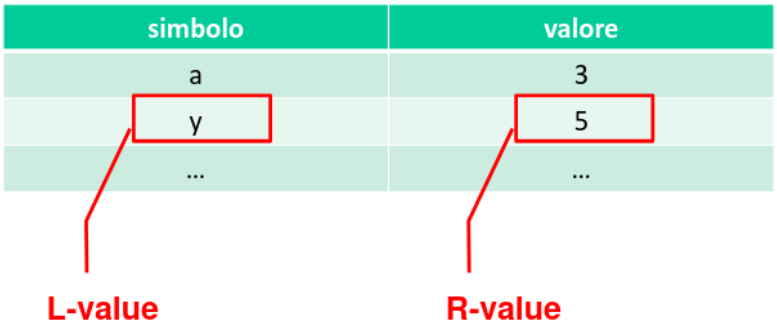
Sorge inoltre la questione dell'*assegnamento distruttivo/non distruttivo*, è possibile cambiare il valore associato in precedenza a un simbolo di variabile?

Nei linguaggi *imperativi* l'assegnameto è distruttivo, nei linguaggi *logici* si ha una trasparenza referenziale dove un simbolo ha sempre lo stesso valore.

6.2 Environment

Per esprimere al meglio la semantica dell'assegnamento, occorre introdurre il concetto di **environment**, inteso come *insieme di coppie* `(simbolo, valore)`, esprimibile tramite una tabella a due colonne (**map**).

Figura 6.1: Mappa ambiente



6.2.1 Semantica

Un assegnamento modifica l'environment causando un *effetto collaterale* secondo questa semantica

$$x = \text{valore}$$

- Se *non esiste* una coppia adatta, se ne inserisce una nuova
- se *esista* una coppia adatta, si denotano due possibilità
  - **assegnamento distruttivo**: viene sostituita la coppia esistente con la nuova
  - **singolo assegnamento**: viene mantenuta la coppia esistente, tentativi di assegnamento a x danno luogo a errori.

6.2.2 Environment multipli

Nei linguaggi imperativi l'assegnamento è distruttivo, produce effetti *collaterali* nell'environment.

Solitamente l'environment è suddiviso in sotto-ambienti collegati al *tempo di vita* delle strutture run-time:

- **environment globale**: coppie il cui tempo di vita è l'*intero programma*
- **environment locale**: coppie con tempo di vita diverso dall'intero programma, tipicamente relativo a *funzioni o altre strutture*.

Ogni modello computazionale deve specficiare il campo di *visibilità dei suoi* ***simboli***, ossia quali environment sono visibili in un certo punto del programma.

6.3 Scelta del tipo di assegnamento

É necessario valutare vari aspetti di un per l'introduzione di un sistema di assegnamento:

- la sintassi dei nomi delle variabili
- se l'assegnamento sia *distruttivo* o *meno*
- decidere se la scrittura di assegnamento `x = valore` sia
  - **istruzione**: effettua una azione senza denotare un valore
  - **espressione**: effettua una azione e denota il valore risultante

Assegnamento multiplo

Questa ultima considerazione dipende dal fatto che si voglia o meno supportare l'assegnamento multiplo, che consiste nel rendere valide espressioni come `x = y = z = valore`.

Normalmente l'assegnamento ha la natura di istruzione, in quanto causa un effetto collaterale nell'environment, tuttavia in questo modo si rende impossibile comporre assegnamenti multipli.

Consentire l'assegnamento multiplo implica interpretarlo come espressione, questo rende l'operatore di assegnamento *associativo a destra*, poice il valore è l'ultimo elemento.

```
x = (y = (z = valore))
```

6.3.1 Estensione dell'interprete

Per implementare l'assegnamento occorre aggiungere la nozione di:

- espressione di **assegnamento**, nuova `AssignExp` (produzione per EXP)
- **variabile** nelle sue due interpretazioni (L/R-value) che necessita di decidere se mantenenere la stessa sintassi per R e L-value, nuovo tipo di fattore (produzione per FACTOR)

Senza un simbolo che distingua R-value da L-value, la grammatica diventa LL(2).
Si aggiungono le produzioni:

$P = \{$

...

`EXP` := `ASSIGN`

`ASSIGN` := `IDENT` = `EXP`

```
        FACTOR := $IDENT
        IDENT := lettera
        ...
    }
```

Occorre inoltre estendere la sintassi astratta, si hanno tre nuovi tipi di nodo per l'AST:

- nodo *operazione assegnamento* → `AssignExp`
- nodo *L-value* (`IDENT` a sinsitra) → `LIdentExp`
- nodo *R-value* (`IDENT` a destra) → `RIdentExp`

In modo informale, si può definire il nodo L-value in modo da accedere al contenitore avente quel simbolo, e il nodo R-value in modo da ottenere il valore associato a quel simbolo nell'environment corrente.

Figura 6.2: Nuovi nodi

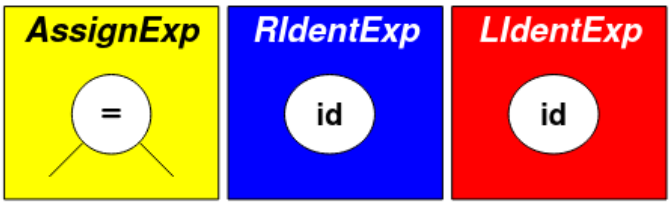
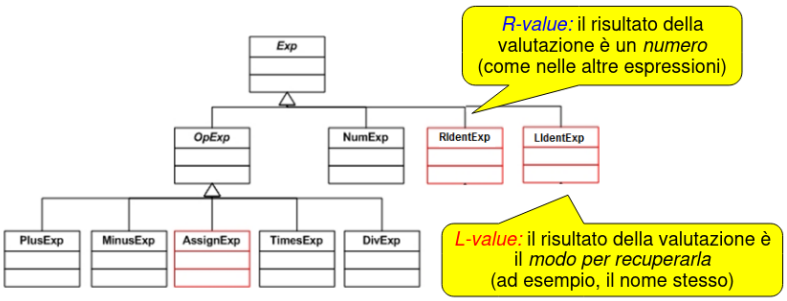


Figura 6.3: Nuova struttura delle classi



Nel parser si effettuano alcune modifiche per integrare comdodamente i nuovi nodi:

- classe `Token` ha metodo `isIdentifier()` che controlla la sintassi degli identificatori
- `parseExp` cattura la nuova espressione `Assign` intercettando la sequenza `IDENT` = `EXP`
- `parseFactor` cattura il nuovo fattore `RIdent`, intercettando la sequenza `$IDENT`

inserisci codice slide 26-41

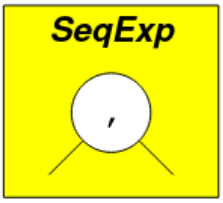
6.4 Espressioni sequenza

Il prossimo obiettivo è quello di aggiungere la possibilità di esprimere più espressioni in sequenza, ovvero espressioni concatenate da una virgola.

Ipotizzando che la prima espressione è sempre un assegnamento e i valore complessivo è quello dell'exp più a destra:

`EXP` := `ASSIGN`, `EXP`

Si indentifica il nuovo nodo `SeqExp`.



continua slide 47..

inserisci codice slide 49-55

7 — Multi-Paradigm Programming: parser espressioni in Prolog

Il motore Prolog contiene un suo scanner e un suo parser usato per interpretare la sua sintassi di regole.

L'obiettivo è quello di mappare la nostra sintassi "sopra" alla sua per riutilizzare le strutture già definite.

Cosa abbiamo già:

- tre dei quattro operatori sono già *operatori infissi* leciti in Prolog

Cosa manca:

- l'operatore : è diverso in quanto Prolog usa la barra /
- le parentesi tonde sono già intercettate dal parser di Prolog

7.0.1 Definizione nuovi operatori

Per definire un nuovo operatore infisso in Prolog basta porre all'inizio della teoria logica la dichiarazione

```
:- op(livelloprioritàà, associatività, operatore)
```

- livelloprioritàà: numero che esprime la priorità dell'operatore
- associatività: atomo che esprime se l'operatore è infisso o pre/postfisso e la sua associatività
- operatore: è il nuovo operatore da dichiarare

```
:- op(400, yfx, ':')
```

7.0.2 Problema delle parentesi tonde

Non si possono utilizzare parentesi tonde dato che sono utilizzate da Prolog, sostituiamo quindi le tonde con le parentesi quadre, anch'esse già utilizzate da prolog per le liste (già riconosciute e bilanciate).

Il prezzo sarà minimo, in quanto consiste solo nella differenza della forma delle parentesi, esistono inoltre tecniche per mascherarle.

7.0.3 Espressioni in Prolog

La semantica denotazionale definita esprime già tutte le regole "pattern oriented", sono direttamente trasferirle in Prolog.

```
fExpr(Term) :- fTerm(Term).
fExpr(Exp+Term) :- fExpr(Exp), fTerm(Term).
fExpr(Exp-Term) :- fExpr(Exp), fTerm(Term).
fTerm(Factor) :- fFactor(Factor).
fTerm(Term*Factor) :- fTerm(Term), fFactor(Factor).
fTerm(Term:Factor) :- fTerm(Term), fFactor(Factor).
fFactor([Exp]) :- fExpr(Exp).
fFactor(num) :- number(Num).
```

Le funzioni sono solo sintassi, per ora non hanno nessun significato.

7.0.4 Dal riconoscitore al valutatore

Per sintetizzare il valutatore è necessario:

- estendere la testa delle regole aggiungendo un *argomento per restituire il risultato*
- estendere il corpo delle regole, ricavando i *valori* e *combinandoli* in modo appropriato alla semantica

Per il trattamento dei numeri, Prolog utilizza soltanto il concetto di numeri reali, e attraverso il predicato `is` si può applicare la semantica dei numeri naturali.

```
evalExpr(Term, Value) :- evalTerm(Term, Value).
evalExpr(Exp+Term, Value) :- evalExpr(Exp, Value1),
                             evalTerm(Term, Value2),
                             Value is Value1 + Value2.
evalExpr(Exp-Term, Value) :- evalExpr(Exp, Value1),
                             evalTerm(Term, Value2),
                             Value is Value1 - Value2.
evalTerm(Factor, Value) :- evalFactor(Factor, Value).
evalTerm(Term*Factor, Value) :- evalTerm(Term, Value1),
```

```
                                evalFactor(Factor, Value2),
                                Value is Value1 * Value2.
evalTerm(Term:Factor, Value) :- evalTerm(Term, Value1),
                                evalFactor(Factor, Value2),
                                Value is Value1 / Value2.
evalFactor([Exp], Value) :- evalExpr(Exp, Value).
evalFactor(Num, Num) :- number(Num).
```

7.1 Parser ibrido con 2p-kt

Java si occupa di:

- visualizzare finestre di dialogo per richiedere stringa di input
- sostituire parentesi tonde con quadre
- *creare* un **motore Prolog**, configurandolo con l'opportuna **teoria** e **interrogarlo** con l'espressione
- recuperare e visualizzare il risultato della valutazione

tuProlog si occupa di

- interpretare e valutare l'espressione data

7.1.1 Primi passi per utilizzare il Prolog engine

Leggere la teoria logica (input o file)

```
ClausesReader theoryReader = ClausesReader.getWithDefaultOperators();
Theory theory = theoryReader.readTheory(input);
```

Ottenere una istanza di `SolverFactory`, per ottenere il un solver adeguato

```
SolverFactory solverFactory = ClassicSolverFactory.INSTANCE;
```

Costruire tramite la factory il solver, passando la teoria

```
Solver solver = solverFactory.solverWithDefaultBuiltins(theory);
```

7.1.2 TermParser

Il `TermParser` è il componente che effettua il parsing dei termini, deve quindi riconoscere tutti gli operatori, anche quelli definiti custom.

É possibile costruire un `TermParser` con i giusti operatori "a mano" oppure è possibile ottenere dal solver le definizioni trovate.

Metodo 1

```
OperatorSet myOpSet = OperatorSet.STANDARD.plus(new Operator(":", Specifier.YFX, 400));
TermParser termParser = TermParser.withOperators(myOpSet);
```

Metodo 2

```
TermParer termParser = TermParser.withOperators(solver.getOperators())
```

Per creare la query a partire dalla rappresentazione testuale

```
Struct query = termParser.parseStruct("evalExpr(" + queryText + ", Result)");
```

8 — Riconoscitori LR

Parsing LR vs Parsing LL

Costruendo l'albero top-down, la debolezza dell'analisi LL si traduce in:

- deve poter identificare la produzione giusta usando soltanto i *primi k simboli della parte destra*
- LL(1) è l'unico caso interessante
- LL(2) è utile in casi particolari

L'analisi LR invece, costruisce l'albero **bottom-up**:

- parte dalle foglie, aspettando di avere abbastanza informazione per decidere come interpretarle
- meno naturale ma superiore dal punto di vista teorico
- ogni grammatica LL(k) è anche LR(k)

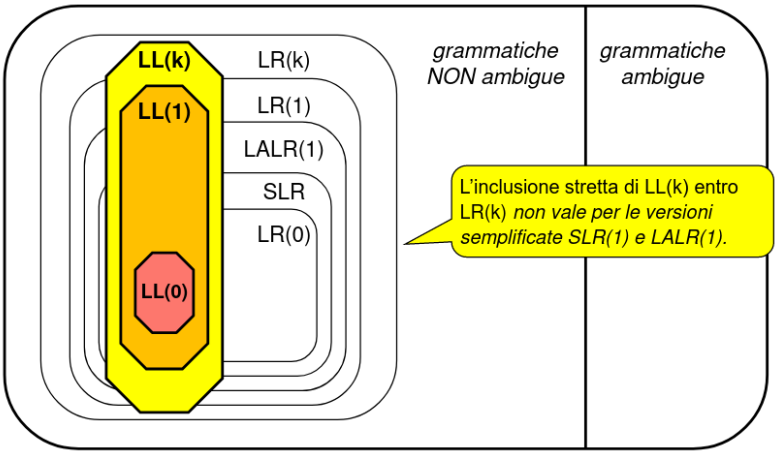
Tuttavia l'analisi LR è complessa da progettare e già il caso LR(1) spesso ingestibile per le grammatiche dei tipici linguaggi.

Sono state sviluppate *tecniche semplificate*:

- **SLR**: Simple LR
- **LALR**: Look-Ahead LR

Si utilizzano comunque sempre strumenti *automatici*.

Figura 8.1: Gerarchia LR



Tecniche LR

L'analisi LR procede BOTTOM UP, parte dalla frase e cerca di **riduarla** allo scopo S, ogni passo deve decidere se:

- proseguire la lettura da input → **SHIFT**
- costruire un pezzo di albero → **REDUCE**

L'ultima REDUCE conclude l'albero con successo, accetta la frase nella fase di **ACCEPT**.

Per questo ha sempre bisogno di *informazioni di contesto*.

8.1 Architettura

Il parser LR richiede un componenente, detto **ORACOLO**, che in base al contesto corrente, comunich se effettuare SHIFT o REDUCE al parser.

Un parser LR è quindi composto da:

- un **oracolo**, che comunica se fare SHIFT o REDUCE
- uno **stack**, in cui conservare lo stato corrente di input e albero
- un **controller** che governa i primi due

Informazione di contesto

L'oracolo sfrutta opportune *informazioni di contesto* per decidere se effettuare la lettura di un nuovo input o un passo di riduzione.

Questo componenente è un *riconoscitore di contesti*, e solo se riconosce un appropriato *contesto di riduzione*, ordina l'azione REDUCE appropriata per costruire senza ambiguità un certo pezzo di albero.

8.2 Caso LR(0)

Nell'analisi LR conviene studiare per prima LR(0), un sottoinsieme di LR nel quale è possibile scegliere la mossa da fare *senza dover guardare il prossimo simbolo di input*.

In LR questo non significa non avere informazioni in assoluto, si precludono quelle future ma si ritengono informazioni sul contesto passato.

8.2.1 Analisi LR(0)

Passi:

- calcolare il *contesto* LR(0) di ciascuna produzione
- identificare **collisioni** in contesti di produzioni diverse
  - **collisione**: stringa appartenente a un contesto è un **prefisso proprio** di una stringa in un altro contesto
  - **prefisso proprio**: una stringa è prefisso di un'altra se ciò che segue è un terminale (non metasimbolo)
- se non ci sono collisioni, si possono usare i contesti LR(0) per guidare l'analisi

Se sono presenti collisioni, i contesti LR(0) non sono sufficienti, è necessario testare con LR(1).

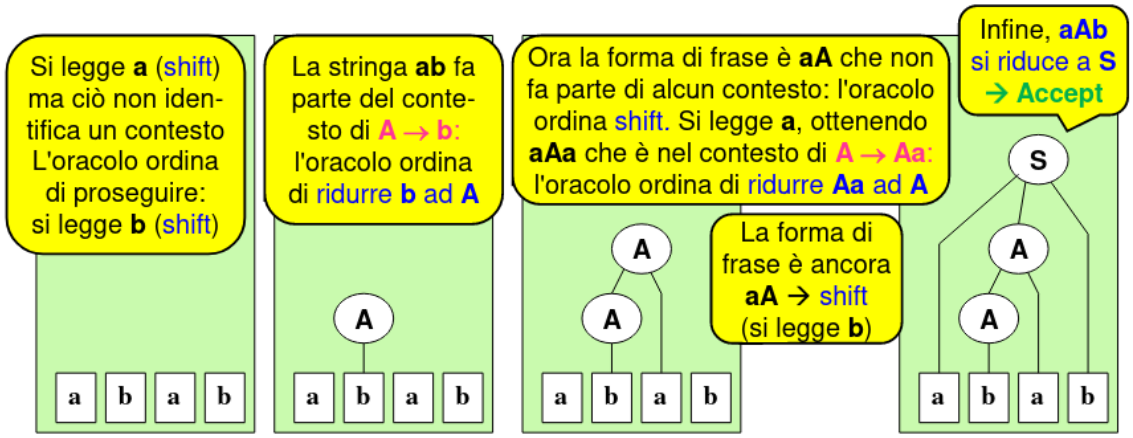
Esempio 1

Grammatica LR(0):

- `S` $\rightarrow$ `a A b` — contesto LR(0): {`aAb`}
- `S` $\rightarrow$ `a a B b a` — contesto LR(0): {`aaBba`}
- `A` $\rightarrow$ `A a` — contesto LR(0): {`aAa`}
- `A` $\rightarrow$ `b` — contesto LR(0): {`ab`}
- `B` $\rightarrow \varepsilon$ — contesto LR(0): {`aa`}

Da notare che non esiste collisione tra i contesti `aaBba` e `aa`, perché la stringa `aa`, pur essendo un prefisso di `aaBba` non è *prefisso proprio* di quest'ultima (`B` è un non terminale).

La frase `abab` è analizzata come segue:

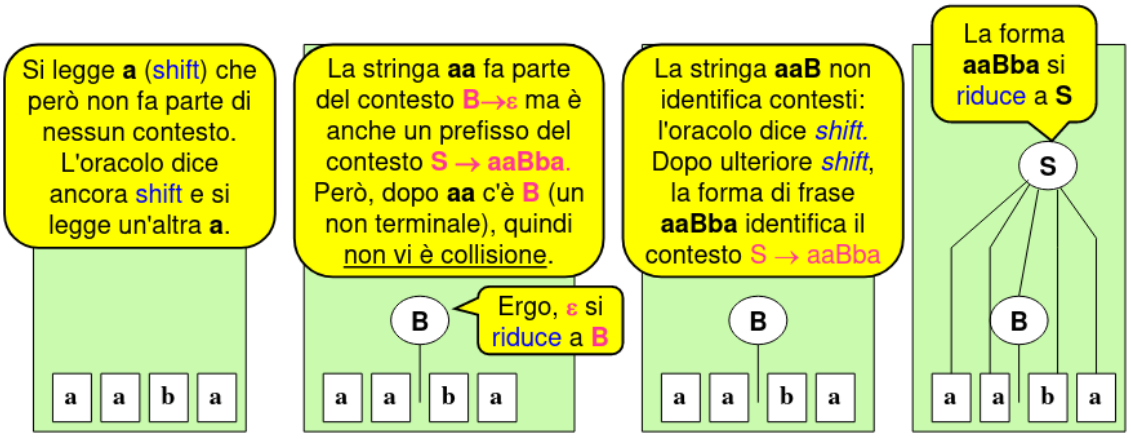


Per ovviare a un possibile caso critico, conviene assumere che esista una produzione di **top-level** `Z` $\rightarrow$ `S`, dove `Z` è il nuovo scopo.

Esempio 2

- `S` $\rightarrow$ `a A b` — contesto LR(0): {`aAb`}
- `S` $\rightarrow$ `a a B b a` — contesto LR(0): {`aaBba`}
- `A` $\rightarrow$ `A a` — contesto LR(0): {`aAa`}
- `A` $\rightarrow$ `b` — contesto LR(0): {`ab`}
- `B` $\rightarrow$ ε — contesto LR(0): {`aa`}

La frase aaba è analizzata come segue:



L'emulatore richiede che la grammatica sia estesa con la regola top-level `Z` $\rightarrow$ `S`.

Caso critico

Perché il calcolo dei contesti può dare una informazione fuorviante se lo scopo è riuscato nella parte destra di qualche produzione?

Si consideri l'esempio:

- `S` → `S a` contesto LR(0): {`Sa`}
- `S` → `a` contesto LR(0): {`a`}

Apparentemente i contesti sono diversi, tuttavia, dato che `S` può comparire in forme di frase intermedie, la "riduzione a `S`" non sempre denota il termine di una frase.

Aggiungere la regola top-level `Z` → `S` risolve l'ambiguità.

Supponendo di dover riconoscere la frase `aaa`:

- senza la top-level, nella prima mossa si riduce `a` in `S`, ma poi non è chiaro se terminare o meno:
  - se non ci sono altre `a`, si è già ridotto a `S`
  - se la stringa continua con altre `a`, si applicano altre riduzione da S`Sa` ad `S`
  - occorre tuttavia guardare avanti di un simbolo → no LR(0)
- riformulando con lo scopo `Z`
  - `Z` → `S` contesto LR(0): {`S`}
  - `S` → `S a` contesto LR(0): {`Sa`}
  - `S` → `a` contesto LR(0): {`a`}
  - si nota che i primi due contesti collidono

8.2.2 Contesti LR(0)

Il calcolo dei contesti LR(0) si basa sul fatto che essi sono definiti da una *opportuna grammatica*.

> La grammatica che definisce i contesti LR(0) è sempre **regolare** (a sinistra).

Ciò denota che il *riconoscimento del contesto corrente*, svolto dall'oracolo, si può ottenere con un *automa a stati finiti*.

Definizione

> Il contesto LR(0) di una produzione `A` → α
> LR(0) ctx(A → α) = $\{\gamma | \gamma = \beta\alpha, Z \Rightarrow \beta Aw \Rightarrow \beta\alpha w, w \in VT^*\}$

Tradotto:

> Il contesto LR(0) della produzione $\mathtt{A} \to \alpha$ è l'insieme di tutti i **prefissi** ($\beta\alpha$) di una forma di frase che usi la produzione $\mathtt{A} \to \alpha$ all'ultimo passo ($\beta\alpha\mathrm{w} \Rightarrow \beta\alpha\mathrm{w}$) di una derivazione canonica destra.

Conseguenza:

> Tutte le stringhe del contesto LR(0) della produzione $\mathtt{A} \to \alpha$ hanno la forma $\alpha\beta$ e *differiscono solo per il prefisso* β.

Calcolo

Dato che le stringhe del contesto LR(0) differiscono solo per il prefisso β, che dipende da A, si può esprimere il contesto come concatenazione tra insieme dei β e il suffisso α.

L'insieme dei β si chiama **contesto sinistro** di A.

Formalmente:

> Il contesto sinistro di un non terminale A:
>
> $$\mathrm{leftctx(A)} = \{\beta | \mathrm{Z} \Rightarrow \beta\mathrm{Aw}, \mathrm{w} \in \mathrm{VT}^*\}$$
>
> da cui il contesto LR(0) della produzione $\mathtt{A} \to \alpha$
>
> $$\mathrm{LR(0)\ ctx(A} \to \alpha) = \mathrm{leftctx(A)} \cdot \{\alpha\}$$

Calcolo dei contesti sinistri

Determinare leftctx(A), comporta trovare tutti i modi in cui può apparire il meta-simbolo A in una forma di frase.

Poiché lo scopo Z per definizione non compare mai nella parte destra di alcuna produzione, leftxctx(Z) = $\{\varepsilon\}$.

Data una produzione $\mathtt{B} \to \gamma\mathtt{A}\delta$, i prefissi che possono esserci davanti ad A sono quelli che potevano esserci davanti a B seguiti dalla stringa γ.

Un primo contributo a leftctx(A) è

$$\mathrm{leftctx(A)} \supseteq \mathrm{leftctx(B)} \cdot \{\gamma\}$$

Si analizzano poi tutte le altre produzioni nelle quali compare A nella parte destra.

Esempio

- $\mathtt{Z} \to \mathtt{S}$

- $\texttt{S} \to \texttt{a S A B} \mid \texttt{B A}$
- $\texttt{A} \to \texttt{a A} \mid \texttt{B}$
- $\texttt{B} \to \texttt{b}$

Considerando i due postulati:

$$\text{leftctx}(\texttt{Z}) = \{\varepsilon\}$$
$$\texttt{B} \to \gamma\texttt{A}\delta \Rightarrow \text{leftctx}(\texttt{A}) \supseteq \text{leftctx}(\texttt{B}) \cdot \{\gamma\}$$

Postulato 1 e 2 per produzione $\texttt{Z} \to \texttt{S}$

- $\text{leftctx}(\texttt{Z}) = \{\varepsilon\}$
- $\text{leftctx}(\texttt{S}) \supseteq \text{leftctx}(\texttt{Z}) \cdot\{\varepsilon\} = \text{leftctx}(\texttt{Z})$

Postulato 2 per la produzione $\texttt{S} \to \texttt{a S A B} \mid \texttt{B A}$

- $\text{leftctx}(\texttt{S}) \supseteq \text{leftctx}(\texttt{S}) \cdot\{\texttt{a}\}$ $(\texttt{S} \to \texttt{a S}\ \delta)$
- $\text{leftctx}(\texttt{A}) \supseteq \text{leftctx}(\texttt{S}) \cdot\{\texttt{aS}\}$ $(\texttt{S} \to \texttt{a S A}\ \delta)$
- $\text{leftctx}(\texttt{B}) \supseteq \text{leftctx}(\texttt{S}) \cdot\{\texttt{aSA}\}$ $(\texttt{S} \to \texttt{a S A B})$
- $\text{leftctx}(\texttt{B}) \supseteq \text{leftctx}(\texttt{S}) \cdot\{\varepsilon\}$ $(\texttt{S} \to \texttt{B}\ \delta)$
- $\text{leftctx}(\texttt{A}) \supseteq \text{leftctx}(\texttt{S}) \cdot\{\texttt{B}\}$ $(\texttt{S} \to \texttt{B A})$

Postulato 2 per la produzione $\texttt{A} \to \texttt{a A} \mid \texttt{B}$

- $\text{leftctx}(\texttt{A}) \supseteq \text{leftctx}(\texttt{A}) \cdot\{\texttt{a}\}$ $(\texttt{A} \to \texttt{a A})$
- $\text{leftctx}(\texttt{B}) \supseteq \text{leftctx}(\texttt{A}) \cdot\{\varepsilon\}$ $(\texttt{A} \to \texttt{B})$

Componendo i pezzi

- $\text{leftctx}(\texttt{Z}) = \{\varepsilon\}$ $\quad \text{leftctx}(\texttt{Z}) = \{\varepsilon\}$

- $\text{leftctx}(\texttt{S}) \supseteq \text{leftctx}(\texttt{Z})$
- $\text{leftctx}(\texttt{S}) \supseteq \text{leftctx}(\texttt{S}) \cdot \{\texttt{a}\}$

$\text{leftctx}(\texttt{S}) = \text{leftctx}(\texttt{Z}) \cup$
$\text{leftctx}(\texttt{S}) \cdot \{\texttt{a}\}$

- $\text{leftctx}(\texttt{A}) \supseteq \text{leftctx}(\texttt{S}) \cdot \{\texttt{aS}\}$
- $\text{leftctx}(\texttt{A}) \supseteq \text{leftctx}(\texttt{S}) \cdot \{\texttt{B}\}$
- $\text{leftctx}(\texttt{A}) \supseteq \text{leftctx}(\texttt{A}) \cdot \{\texttt{a}\}$

$\text{leftctx}(\texttt{A}) = \text{leftctx}(\texttt{S}) \cdot \{\texttt{aS}\} \cup$
$\text{leftctx}(\texttt{S}) \cdot \{\texttt{B}\} \cup$
$\text{leftctx}(\texttt{A}) \cdot \{\texttt{a}\}$

- $\text{leftctx}(\texttt{B}) \supseteq \text{leftctx}(\texttt{S}) \cdot \{\texttt{aSA}\}$
- $\text{leftctx}(\texttt{B}) \supseteq \text{leftctx}(\texttt{S})$
- $\text{leftctx}(\texttt{B}) \supseteq \text{leftctx}(\texttt{A})$

$\text{leftctx}(\texttt{B}) = \text{leftctx}(\texttt{S}) \cdot \{\texttt{aSA}\} \cup$
$\text{leftctx}(\texttt{S}) \cup$
$\text{leftctx}(\texttt{A})$

- leftctx(Z) = {ε} — ⟨LctxZ⟩ → {ε}

- leftctx(S) = leftctx(Z) ∪ — ⟨LctxZ⟩ → ⟨LctxZ⟩ |
- leftctx(S) · {a} — ⟨LctxS⟩ a

- leftctx(A) = leftctx(S) · {aS} ∪ — ⟨LctxS⟩ → ⟨LctxS⟩ aS |
- leftctx(S) · {B} ∪ — ⟨LctxS⟩ B |
- leftctx(A) · {a} — ⟨LctxA⟩ a

- leftctx(B) = leftctx(S) · {aSA} ∪ — ⟨LctxB⟩ → ⟨LctxS⟩ aSA |
- leftctx(S) ∪ — ⟨LctxS⟩ |
- leftctx(A) — ⟨LctxA⟩

Risolvendo le equazioni si possono trovare le espressioni regolari dei vari contesti:

- leftctx(Z) = {ε}
- leftctx(S) = {a*}
- leftctx(A) = {(a* a S + a* B) a*}
- leftctx(B) = {a* a S A + a* + (a* a S + a* B) a*}

Semplificando le espressioni regolari

- leftctx(Z) = {ε}
- leftctx(S) = {a*}
- leftctx(A) = {a*(a S + B) a*}
- leftctx(B) = {a* (a S A + ε) + a* (a S + B) a*}

e concatenando i contesti sinistri con i rispettivi suffissi

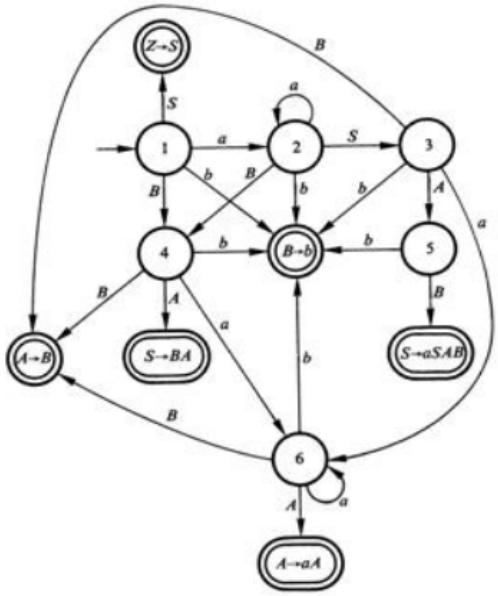
- LR(0)ctx(Z → S) = {ε} S = {S}
- LR(0)ctx(S → aSAB) = {a*aSAB}
- LR(0)ctx(S → BA) = {a*BA}
- LR(0)ctx(A → aA) = {a*(aS+B)a*aA}
- LR(0)ctx(A → B) = {a*(aS+B)a*B}
- LR(0)ctx(B → b) = {a*(aSA+ε)b+a*(aS+B)a*b}

si ottengono diversi linguaggi, è necessario costruire un RSF che riconosca l'unione di questi linguaggi: è necessario che risulti **deterministico**.

8.2.3 L'automa a stati ausiliario

Questo RSF è la **macchina caratteristica** della grammatica iniziale: ogni stato finale è etichettato con una produzione, usata dal parser per effettuare una REDUCE nel contesto.

Si parte ogni volta dallo stato iniziale e si percorre l'ASF per deciedere la mossa.



Algoritmo parsing LR(0)

L'analisi LR(0) di una frase si svolge sottoponendo all'automa carrateristico la forma di frase corrente.

- **SHIFT**: quando l'automa caratteristico raggiunge uno stato *non finale*, indica che le informazioni disponibili non sono sufficienti
- **REDUCE**: quando l'automa caratteristico raggiunge uno stato *finale*, si applica quindi la produzione raggiunta

La forma di frase corrente è mantenuta su uno stack:

- **SHIFT** pone in *cima* allo stack il nuovo simbolo terminale letto
- **REDUCE** *toglie* dallo stack tanti elementi quanti sono quelli nella *parte destra* della produzione da applicare e si pone in *cima* allo stack il simbolo non terminale corrispondente alla *parte sinistra* di tale riduzione.

ESEMPIO A SLIDE 40-43

Miglioramenti

Ogni volta che si effettua una REDUCE si riparte dallo stato iniziale, ma dato che la forma di frase cambia solo in fondo anche la prima parte del percorso non cambia.

Si può quindi ottimizzare il processo memorizzando il percorso fatto, ripartendo dall'ultimo stato utile; a questo scopo si utilizza uno *stack ausiliario degli stati*, dove si impilano gli stati attraversati e se ne rimuovono tanti quanti i simboli coinvolti in un passo di riduzione.

ESEMPIO SLIDE 45-51

8.2.4 Condizione LR(0)

Condizione sufficiente perché una grammatica sia LR(0) è che, date due produzioni $\mathtt{A} \to \alpha$ e $\mathtt{B} \to \omega$

- se $\theta \in$ LR(0)(ctx$\mathtt{A} \to \alpha$) dove $\theta \in (\mathtt{VT} \cup \mathtt{VN})*$
- e θw$\in$ LR(0)ctx($\mathtt{B} \to \omega$) dove w $\in \mathtt{VT}^*$

risulti

$$\mathrm{w} = \varepsilon, \mathrm{A} = \mathrm{B}, \alpha = \omega$$

Ogni stato di riduzione dell'automa ausiliario sia etichettato da una *produzione unica* e *non abbia archi di uscita* etichettati da terminali.

8.2.5 Automa caratteristico - approccio alternativo

Il metodo descritto è difficile da automatizzare e complesso, esiste tuttavia un metodo operativo facilmente meccanizzabile.

Procedimento operativo

Si parte dalla regola top-level $\mathtt{Z} \to \mathtt{S\$}$ e si analizzano le situazioni che si presentano:

- creare un rettangolo **LR item** per ogni "situazione"
- quando una regola ne usa un'altra, la includiamo nel corrispondente LR item
- si introduce l'astrazione di **cursore "."** che indica la posizione all'interno della regola dell'analisi

Si studiano le evoluzioni possibili di ogni LR item, costruendo l'automa dal basso, caso per caso.

Convenzioni

- ogni frase lecita è terminata da `$`
- il *cursore* `.` denota il *confine* tra la parte già analizzata e quella ancora da analizzare: $\mathtt{A} \to \mathtt{a.B}$, il prossimo input è `B`, dove
  - *input* è la forma di frase eventualmente già sottoposta a passi di riduzione (simboli terminali e non terminale della grammatica di partenza)
  - l'automa si appoggia allo stack di input: i simboli a sinistra del cursore sono già stati estratti, l'*input è il simbolo al top dello stack*

Facendo riferimento alla grammatica

- `Z` → `S`
- `S` → `a S A B | B A`
- `A` → `a A | B`
- `B` → `b`

Tutti gli input sono ancora da leggere e ogni frase lecita deriva dallo scopo Z:

- `Z` → `.S$`

Per descrivere appieno questo stato, occorre specificare anche tutte le produzioni di `S`

- `Z` → `.S$`
- `S` → `.aSAB | .BA`
- `B` → `.b`

Si include anche `B` poiché una delle produzioni di `S` può iniziare `B`.

Una volta individuato un stato, in questo caso stato 1, si procede a analizzare tutte le possibili evoluzioni spostando il cursore a *destra in tutti i modi possibili*:

- con input `S`, situazione `Z` → `S.$`, stato F
- con input `a`, situazione `S` → `a.SAB`, stato 2
- con input `B`, situazione `S` → `B.A`, stato 4
- con input `b`, situazione `B` → `b.`, stato R1

Si denotano due stati finali

- situazione `Z` → `S.$`, stato finale di accettazione F
- situazione `B` → `b.`, stato finale di riduzione R1

Nella situazione `S` → `a.SAB` (stato 2), l'input può iniziare per `S`, quindi vanno considerate tutte le produzioni relative a `S`, tra queste una può iniziare per `B` quindi si aggiunge anche `B` → `b`, si ottiene lo stato 2.

- `S` → `a.SAB`
- `S` → `.aSAB | .BA`
- `B` → `.b`

Analogamente nella situazione `S` → `B.A` (stato 4), l'input può iniziare per `A` vanno considerate anche le produzioni relative ad `A`, così come quelle relative a `B`, dato che tra quelle di `A` ne esiste una che può iniziare per `B`, si ottiene lo stato 4.

- `S` → `B.A`
- `A` → `.aA | .B`
- `S` → `.b`

Dallo stato 2

- con input S, situazione S → aS.AB, stato 3
- con input a, situazione S → a.SAB, stato 2
- con input B, situazione S → B.A, stato 4
- con input b, situazione B → b., stato R1

Dallo stato 4

- con input A, situazione S → BA., stato R3
- con input a, situazione A → a.A, stato 6
- con input B, situazione A → B., stato R2
- con input b, situazione B → b., stato R1

Nella situazione S → aS.AB (stato 3), l'input può iniziare per A, si aggiungono le produzoni di A e quelle di B per ottenere lo stato 3.

- S → aS.AB
- A → .aA | .B
- B → .b

Nella situazione A → a.A (stato 6), l'inputo può iniziare solo per A, quindi vanno aggiunte le produzioni di A e B per ottenere lo stato 6.

- A → a.A
- A → .aA | .B
- B → .b

Dallo stato 3

- con input A, situazione S → aSA.B, stato 5
- con input a, situazione A → a.A, stato 6
- con input B, situazione A → B., stato R2
- con input b, situazione B → b., stato R1

Dallo stato 6

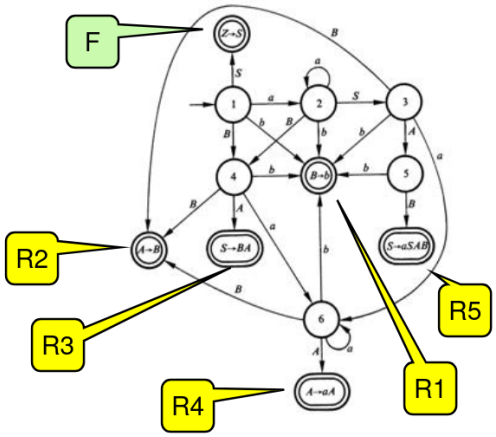
- con input A, situazione A → aA., stato R4
- con input a, situazione A → a.A, stato 6
- con input B, situazione A → B., stato R2
- con input b, situazione B → b., stato R1

Nella situazione `S` → `aSA.B`, l'input può iniziare solo con `B`, quindi va aggiunta solo la produzione `B` → `b.` per ottenere lo stato 5.

- `S` → `aSA.B`
- `B` → `.b`

Dallo stato 5 si ottengono solo situazioni già appartenenti agli stati finali R1 e R5.

Il risultato finale è lo stesso.



8.2.6 Costruzione parser LR(0)

Le mosse da compiere sono quattro:

- ogni arco corrisponde a uno SHIFT
  - SHIFT è l'arco corrispondente a un *terminale*, perché corrisponde a una vera *lettura da input*
  - se l'arco corrisponde a un metasimbolo, l'azione è chiamata **GOTO** e corrisponde a una lettura *interna dello stack*
- ogni stato finale corrisponde a una REDUCE
  - nel caso finale chiamata **ACCEPT**, si completa l'albero con lo scopo finale `Z` → `S.$`

Si procede alla costruzione della tabella di parsing LR, dove ogni cella contiene l'indicazione dell'azione da compiere tra le quattro possibili (SHIFT, GOTO, REDUCE, ACCEPT) e indicazioni accessorie necessarie.

Si utilizza la notazione

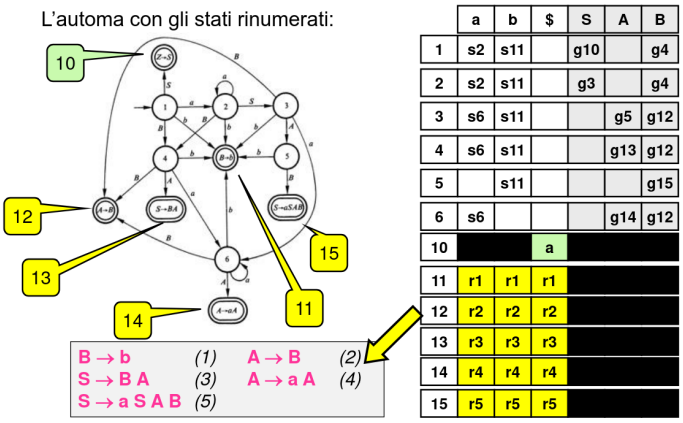
- `s / 5`, SHIFT e GOTO stato 5
- `g / 5`, GOTO stato 5 e consuma metasimbolo

- `r / 4`, REDUCE con produzione numero 4
- `a`, ACCEPT

Algoritmo compilazione tabella di parsing:

- per ogni arco `S1` → `S2` con input il simbolo terminale `a`, si inserisce in tabella alla posizione `(S1, a)` l'azione SHIFT to `S2`
- per ogni arco `S1` → `S2` con input da stack il metasimbolo `X`, si inserisce in tabella alla posizione `(S1, X)` l'azione GOTO `S2`
- per ogni stato `Si` associato alla regola R-esima `A` → α, si inserisce nella tabella l'azione REDUCE `R` in *tutta la riga* corrispondente allo stato `Si`
- per ogni stato `Si` contenente la produzione `Z` → `S.$`, si inserisce nella tabella alla posizione `(S, $)` l'azione ACCEPT

Applicato al caso precedente



	a	b	$	S	A	B
1	s2	s11		g10		g4
2	s2	s11		g3		g4
3	s6	s11			g5	g12
4	s6	s11			g13	g12
5		s11				g15
6	s6				g14	g12
10			a			
11	r1	r1	r1			
12	r2	r2	r2			
13	r3	r3	r3			
14	r4	r4	r4			
15	r5	r5	r5			

8.3 Verso l'analisi LR(1)

Di fatto, LR(0) non è utile nella pratica. L'analisi LR(k) opera analogamente a LR(0), guardando avanti di `k` simboli (**lookahead symbols**).

Tutte le riduzioni sono ritardate di k simboli; la definizione di contesto, e il processo operativo per trovarlo, vengono estesi considerando i k simboli successivi.

Contesto LR(k)

> Formalmente, il contesto LR(k) di una produzione `A` → α è così definito:
>
> $$\text{LR(k)ctx}(\text{A} \to \alpha) = \{\gamma | \gamma = \beta\alpha u, \text{Z} \Rightarrow \beta\text{A}u\text{w} \Rightarrow \beta\alpha u\text{w}, \text{w} \in \text{VT}^*, u \in \text{VT}^k\}$$

Tutte le stringe del contesto LR(k) della produzione `A` → α hanno la forma $\beta\alpha$u e differiscono per il prefisso β e per la stringa u.

La stringa u appartiene all'insieme $\text{FOLLOW}_k(\text{A})$ delle stringhe di lunghezza k che possono seguire il simbolo non terminale A

$$\text{FOLLOW}_k(\text{A}) = \{u \in \text{VT}^k | \text{S} \Rightarrow \gamma \text{A} u \beta\} \text{ con } \gamma, \beta \in \text{V}^*$$

$\text{FOLLOW}_k(\text{A})$ non è altro che l'insieme FOLLOW(A) dei Following Symbols già introdotto nella definizione di Director Symbols dell'analisi LL

Contesti sinistri LR(k)

Contesto sinistro di un non terminale A:

finisci definizione 83 fino 85

8.3.1 Approssimazione del parsing LR(1)

Questi metodi creano automi con molti stati e sono costosi dal punto di vista computazionale, esistono quindi alternative semplificate del parsing LR(1):

- SLR, Simple LR
- LALR(1), Look-Ahead LR

L'idea principale è quella di accorpare gli stati simili, queste due metodologie definiscono come fare.

8.3.2 Contesti SLR(k)

Il contesto SLR(k) di una produzione $\texttt{A} \to \alpha$ è così definito:

$$\text{SLR(k)ctx}(\text{A} \to \alpha) = \text{LR(0)ctx}(\text{A} \to \alpha) \cdot \text{FOLLOW}_k(\text{A})$$

Il contesto SLR(k) può quindi essere calcolato facilmente a partire dal contesto LR(0), il cui calcolo sappiamo già essere fattibile.

É possibile dimostrare che

$$\text{SLR(k)ctx}(\text{A} \to \alpha) \supseteq \text{LR(k)ctx}(\text{A} \to \alpha)$$

ovvero il contesto SLR è un po' più grande, e quindi più esposto a potenziali conflitti, del contesto LR completo.

esempio calcolo slide 110-128

8.3.3 Parser LALR(1)

Un approccio alternativo a SLR consiste nel collassare in un solo stato quegli stati che, nel paraser LR(1) completo, sono identici *a meno dei lookahead set*.

Il vantaggio è il fatto che è una trasformazione sempre possibile e spesso molto conveniente (meno stati), tuttavia possono apparire conflitti reduce / reduce, tipicamente gestibili.

Può essere implementanto calcolando prima LR(0), aggiungendo poi i lookahead set, calcolati separatamente.

Esempio 130-150

9 — Iterazione vs Ricorsione

Spesso si confonde un *costrutto linguistico* con il corrispondente *processo computazionale* sottostante.

L'obiettivo è concentrarsi sul **modello computazione** indipendentemente dalla sintassi, dal linguaggio e dal costrutto linguistico usati.

9.0.1 Processi computazionali iterativi

Nei linguaggi imperativi, il costrutto linguistico che esprime un processo computazionale iterativo è tipicamente il **ciclio**.

Al di la della sintassi, esistono caratteristiche comuni:

- presente un **accumulatore**
- inizializzazione e modifica della variabile
- al termine contiene il risultato

É un processo computazionale basato sulla *modifica di valori*: il nuovo valore socrascrive il precedente.

```
int fact = 1;

for (int i = 1; i <= n; i++) {
    fact = fact * i;
}
```

9.0.2 Processi computazionali ricorsivi

Un processo computazione **ricorsivo** è espresso da una *funzione ricorsiva*.

Al di la della sintassi, esistono caratteristiche comuni:

- non c'è alcun accumulatore

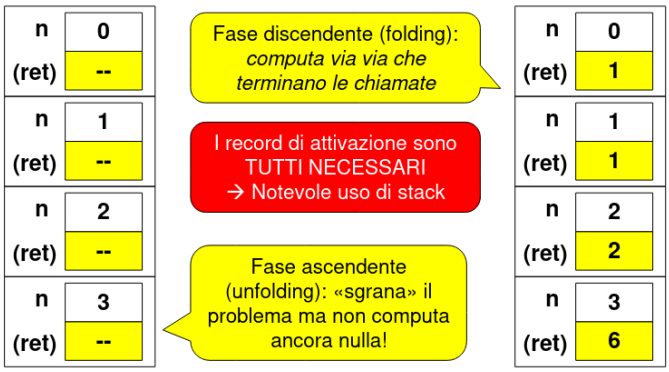
- la chiamata ricorsiva ottiene il risultato (n-1)-esimo
- il corpo della funzione opera sul risultato (n-1)-esimo per sintetizzare il risultato (n)-esimo desiderato

Durante le chiamate ricorsive non c'è alcun risultato parziale: si svolge solo il problema.

Il risultato è sintetizzato mentre le chiamate si *chiudono*, ovvero all'indietro.

Questo è un processo computazionale basato sulla *sintesi di nuovi valori* che non sovrascrivono i precedenti.

```
int fact(int n) {
    return n == 0 ? 1  : fact(n - 1) * n;
}
```



9.0.3 Processi computazionali iterativi espressi da costrutti ricorsivi

Non è sempre detto che una sintassi ricorsiva, dia luogo a un processo computazionale ricorsivo.

Un costrutto *sintatticamente e formalmente ricorsivo* può dare luogo a un *processo computazionale iterativo*, cioè che computa in avanti.

Ciò accade quanto la chiamata ricorsiva è l'*ultima istruzione della funzione* e il risultato parziale k-esimo viene *passato in avanti come argomento*.

```
int factIt(int acc, int n) {
    return n == 0 ? acc : factIt(n * acc, n - 1)
}
```

Come nel caso di un ciclo, al passo k, l'accumulatore contiene il risultato k-*esimo*.

É un processo computazionale basato sulla sintesi di *nuovi valori* che però possono sovrascrivere il precedenti.

Questa è la **ricorsione in coda** (tail recursion), è un costrutto sintatticamente ricorsivo ma computazionalmente iterativo.

I linguaggi funzionali e logici tendono a usare questa ricorsione al posto dei cicli, al contrario dei linguaggi imperativi.

Tail Recursion Optimization

La tail recursion, può essere ottimizzata, stabilendo di allocare il nuovo record di attivazione sopra il precedente.

In questo modo l'occupazione delle risorse è identica a quella del ciclo.

Nei linguaggi imperativi tradizionali (C, Java, C#), la tail recursion *non è ottimizzata*, in quanto sono linguaggi che offrono già una sintassi specifica per l'iterazione.

Nei linguaggi logici (Prolog) e funzionali (Lisp, Scheme) è sempre presente un sistema di TRO, così come in nuovi linguaggi *blended* (Scala, Kotlin).

10 — Basi di programmazione funzionale

Idee chiave del paradigma funzionale:

- distinzioni variabili / valori (**var** e **val**)
- costrutti come espressioni: *everything is an expression*
- collezione e oggetti immodificabili: *compute by synthesis*
- funzioni sono **first-class entitites**
  - chiusure, lambda expressions
  - **lazy** evaluation vs **eager** evaluation
- abolizione tipi primitivi: *everything is an object*
- abolizione parti statiche in favore di singleton *companion*

10.1 Uniformità

10.1.1 Everything is an object

I linguaggi a oggetti tradizionali, pur chiamandosi a oggetti, si basano su tipo primitivi che non sono oggetti.

Questa caratteristica causa disuniformità che necessita di trattamento:

- tipi primitivi passati per **valore** vs oggetti passati per **riferimento**
- collezioni di tipi primitivi: non ammesse o classi wrapper

I linguaggi blended ripuliscono lo spazio concettuale eliminando i tipi primitivi, si distingue però tra **classi valore** (entità immodificabili) e **classi riferimento** (classi classiche).

10.1.2 Don't be static

I linguaggi a oggetti tradizionali, pur chiamandosi oggetti, si basano sulle *classi*, che non sono oggetti.

I linguaggi blended cancellano le *parti statiche*: il loro ruolo è svolto da un singleton, il **comapanion object**:

- in Scala, il co ha lo stesso nome della classe
- in Kotlin, può anche avere nome diverso (ma nello stesso file)

10.2 Oggetti immodificabili

10.2.1 Distinzioni `var` vs `val`

Nel paradigma imperativo, basato sulla modifica dei valori, si generano programmi difficili da leggere e manutenere dato che i simboli cambiano continuamente significato.

Il paradigma funzionale si basa sul concetto opposto: i simboli *mantengono il loro significato*.

I moderni linguaggi blended uniscono i due mondi:

- `var` denota variabili
- `val` denota valori

Proprietà introdotte da `val` hanno solo getter e in certi contesti il suo utilizzo è semplificato o obbligatorio.

10.2.2 Null safety

L'introduzione del null mina alla base la correttezza dei programmi.

Moderni linguaggi introducono forme di *Null safety*:

- vietando che un riferimento sia `null`
- può essere etichettato esplicitamente come tale

10.2.3 Collezioni immodificabili

Completa

Expressions everywhere

I linguaggi imperativi distinguono tra *istruzioni* e *espressioni*

- `for, while, if, throw` sono istruzioni: non denotano un valore ma denotano azioni
-

La presenza di istruzioni induce a esprimere computazioni mettendole in sequenza che implica variabili di appoggio.

Nei linguaggi blended, tutte le istruzioni sono sostituite da espressioni, tutto produce valori.

10.3 Funzioni come first-class entities

Ogni linguaggi di programmazione offre la possibilità di creare funzioni, nei linguaggi blended le funzioni sono anche *first-class entities*.

Nei linguaggi blended, le funzioni hanno queste caratteristiche:

- possono essere *assegnate a variabili* (tipo funzione)
- possono essere *passate come argomento* a un'altra funzione
- possono essere *restituite da un'altra funzione*
- possono essere *definite e usate al volo* come valori di ogni altro tipo

Con questi concetti, è possibile definire *funzioni di ordine superiore*, che sono funzioni che manipolano/generano altre funzioni, e si può estendere il concetto a funzione di secondo, terzo ... ordine.

Assegnare e restiture funzioni implica la necessità di avere **tipi funzione** che *incapsulano il comportamento* delle funzioni.

Esempi in javascript:

```
// keyword function definisce oggetti-funzioni (anonimi)
var f = function (z) { return z * z; }
// funzione può riceverne un'altra come argomento
var ff = function (f, x) { return f(x); }
// si può passare una funzione come argomento a un'altra
var res = loop( function() { i++ }, 10 );
// si può restituire un risultato funzione
function fr() { return function(r) { return r + 10 } }
// si può creare una funzione da stringhe
square = new Function("x", "return x * x")
```

In javascript esiste anche una sintassi per definire funzioni senza la keyword *function*

```
var f = z => z * z

var ff = (f, x) => f(x)

var res = loop( () => i++, 10)

fr = () => r => r + 10
```

10.4 Variabili libere e chiusure

Se un linguaggi ammette la definizione di *funzioni con variabili libere* (non definite localmente), tali funzioni dipendono dal contesto circostante.

Sono necessari criteri per dare significato alle variabili libere:

- **chiusura lessicale**
- **chiusura dinamica**

10.4.1 Funzioni e chiusure

Se un linguaggi supporta funzioni come first-class entities, la presenza di variabili libere determina la nascita della nozione di **chiusura**:

- un *oggetto funzione* ottenuto chiudendo rispetto a un certo contesto una definizione di funzione che aveva variabili libere
- il prodotto finale dell'atto di chiudere le variabili libere rispetto a un certo contesto d'uso

Tempo di vita delle variabili

In presenza di chiusure, il tempo di vita delle variabili di una funzione *non coincide più necessariamente con quello della funzione che le contiene.*

Il modello computazionale deve tenere conto delle variabili libere, non più allocate semplicemente sullo stack.

Una variabile locale a una funzione *esterna* può essere indispensabile al funzionamento della funzione più *interna*, il tempo di vita delle variabili locali che fanno parte di una chiusura è pari a quello della chiusura, anche se la funzione che la definisce *termina prima.*

Queste variabili vengono allocate sull'*heap.*

Esempio

```
function ff(f, x) {
    return function(r) { return f(x) + r; }
}
```

Per poter operare, tale nuova funzione necessita di `f`, `x` (e `r`)

```
var f1 = ff( Math.sin, .8 )
var f2 = ff( function(q) { return q * q }, .8 )
```

Utilità delle chiusure

- rappresentare e gestire uno stato privato e nascosto
  - le variabili della funzione esterna, mantenute vive dalla chiusura, restano comunque invisibili fuori da essa
- creare una comunicazione nascosta
  - definendo più funzioni nella stessa chiusura, esse condividono uno stato nascosto, usabile per comunicare privatamente
- definire nuove strutture di controllo
  - la funzione esterna esprime il controllo, mentre quella ricevuta come argomento esprime il corpo da seguire
- riprogettare/semplificare API e framework di largo uso
  - metodi parametrici che ricevono comportamento come argomento

10.4.2 Criteri di chiusura

Occorre stabilire un criterio su *come e dove cercare* una definizione per le variabili libere da chiudere.

In presenza di funzioni innestate, si apre una questione:

- il testo del programma contiene una catena di *ambienti di definizione innestati*, **catena lessicale**
- l'attivazione delle funzioni crea a runtime una catena di *ambienti attivi*, **catena dinamica** che riflette l'ordine delle chiamate

Esempio

```
var x = 20;

function provaEnv(z) { return z + x }

function testEnv() {
    var x = -1;
    return provaEnv(18);
}
```

Quale `x` viene considerata?

Se si considera la catena lessicale si utilizza la `x = 20`, la definizione più vicina in senso *lessicale*.

Se si considera la catena dinamica si utilizza la `x = -1`, la definizione più vicina nel senso *dimanico*.

Stabilire il significato di `x` in `provaEnv` seguendo la *sequenza delle chiamate* o la *struttura del programma* non è la stessa cosa.

Se si segue la catena lessicale, il modello computazionale adotta la **chiusura lessicale**:

- CONTRO: vincola a priori un simbolo a un certo legame
- PRO: permette di leggere e capire un programma senza ricostruire la sequenza delle chiamate

Se si segue la catena dinamica, il modello computazionale adotta la **chiusura dinamica**:

- PRO: massima dinamicità, simboli legati alla specifica situazione
- CONTRO: difficoltà a tracciare il reale valore di una variabile

10.5 Modelli computazionali per la valutazione di funzioni

Ogni linguaggio che introduca funzioni deve prevedere un modello computazionale per la loro **valutazione**.

Tale modello deve specificare:

- quando si valutano i **parametri**
- cosa si **passa** alla funzione

- come si **attiva** la funzione

Il modello più usato è il **modello applicativo**

- si valutano *subito* i parametri
- si passa il *valore* (tranne certi tipi)
- tipicamente *chiamandola* e *cedendo il controllo* (modello sincrono)

Questo modello è molto usato perché efficiente e semplice da comprendere, tuttavia esistono alcune inefficienze che si possono creare:

- valutare sempre i parametri può essere ridondante se un parametro non serve realmente nella funzione
- se la valutazione dei parametri da errore questo causa un fallimento di tutta la funzione

10.5.1 Modello applicativo - fallimenti evitabili

```
class Esempio {
    static void printOne(boolean cond, double a, double b){
        if (cond) System.out.println("result = " + a);
        else System.out.println("result = " + b);
    }

    public static void main(String[] args) {
        int x=5, y=4;
        printOne(x>y, 3+1, 3/0); //ArithmeticException: div by zero
    }
}
```

In questo esempio la valutazione dei parametri attuali causa eccezione, ma la chiamata non sarebbe fallita se i parametri non fossero valutati a priori.

In linguaggi che adottano un modello diverso questo problema non si sarebbe verificato.

10.5.2 Modello Call-By-Name

Il **modello normale**, adotta un *meccanismo di valutazione* noto come **Call-By-Name** in cui:

- i parametri vengono valutati solo **al momento dell'uso**, quindi solo se servono
- a tal fine vengono passati non dei valori ma degli **oggetti computazionali**
- un parametro ricevuto *può non essere mai calcolato*

Svantaggi di call-by-name

Nonostante i vantaggi concettuali, è meno efficiente in casi normali e più comuni rispetto al modello applicativo:

- i parametri possono esser calcolati più volte
- richiede più risorse a runtime
- richiede una vm capace di *lazy runtime*

Simulare il modello call-by-name

Il modello call by name può essere facilmente *simulato* nei linguaggi con funzioni come first class entities.

Questo si può ottenere passando alla funzione *oggetti computazionali* che, quando eseguiti, producono i valori desiderati: si passano alle funzioni delle funzioni che ritornano il valore quando invocate, sostituendo tutti gli usi di un parametro con chiamate a funzione.

Esempo javascript in stile call-by-name

```
var f = function(flaf, x) {
    return (flag() < 0) ? 5 : x();
}
var b = f(function(){return 3}, function(){abort()});
document.write("result = " + b);
```

Ricostruzione in Java e C#

In Java ¡=7 non è possibile, in quanto le funzioni non sono entità di prima classe.

Da Java 8, le lambda expression catturano funzioni, ma le funzioni non costituiscono un tipo autonomo.

In C#, le lambda expression catturano funzioni e i delegati esprimono i veri *tipi funzione*.

In Java 7 la call-by-name può essere implementata definendo una interfaccia che definisce un metodo `invoke` per ottenere il valore.

In Java 8, con le lambda, si può emulare in modo migliore questo comportamento, ma comunque non in modo pulito come in linguaggi funzionali:

```
static void printTwo(boolean cond, IntSupplier a, IntSupplier b) {
    if (cond) System.out.println("result = " + a.getAsInt());
    else System.out.println("result = " + b.getAsInt());
}
```

```
public static void main(String[] args) {
    int x = 5, y = 4;
    printTwo(x > y, () -> 3+1, () -> 3/0);
}
```

Il miglioramento avviene soltanto lato chiamante, si hanno nomi di tipi particolari, ovvero le interfacce funzionali, come `IntSupplier`.

In C# i delegati completano l'opera

```
class Esempio {
    delegate double MyFunction(); //vero tipo funzione
    static void printTwo(bool cond, MyFunction a, MyFunction b) {
        if (cond) System.Console.WriteLine("result = " + a());
        else System.Console.WriteLine("result = " + b());
    }
    public static void Main(string[] args) {
        int x = 5, y = 4, a = 0; //var a = 0 per valutazione ottimizzata C\#
        printTwo(x > y, () => 3+1, () => 3/a);
```

Ricostruzione in Kotlin e Scala

In Kotlin e Scala si può ricostruire la call-by-name come in C#, ma in Scala non è necessario, in quanto il linguaggio supporta nativamente le call by name.

```
object CallByName {
    def printTwo(cond: Boolean, a: => Int, b: => Int): Unit = {
        if (cond) println("result = " + a);
        else println("result = " + b);
    }

    def main(args: Array[String]) {
        val x=5;
        val y=4;
        printTwo(x>y, 3+1, 3/0);
    }
}
```

11 — Multi-Paradigm Programming con JavaScript

11.1 Il lato funzionale

Le **funzioni** sono introdotte dalla keyword `function`:

- il nome è *opzionale*, possibili funzioni anonime
- sintassi alternativa: *lambda expression*
- *function expression* vs *function declaration*
- non si usa la keyword `void`

Parametri privi di dichiarazione di tipo

```
// funzione
function sum(a, b) { return a + b }
// procedura no return
function printSum(a, b) {
    document.write(a + b)
}
// funzione anonima
var sum = (a, b) => a + b
```

A differenza di molti altri linguaggi, i parametri *attuali* possono non corrispondere in numero ai parametri *formali*:

- se sono più, extra ignorati
- se sono meno, mancanti `undefined`

11.1.1 Funzioni come first class entities

In javascript, una funzione è una entità-oggetto, manipolabile come ogni altro tipo:

- assegnabile a variabili

```
var f = function(x) { return x / 10 }
var f = x => x / 10
```

- definita e usata al "volo" con un *function literal*

```
var z = function(y) { return y + 1 }(8)
var z = (y = > y + 1)(8)
```

- passata come argomento

```
var p = ff(f)
```

- restituita da funzione factory

```
var f = fgen(...)
```

11.1.2 Function expression vs function declaration

La **function expression** assegna una funzione a una variabile

- il nome della funzione non è essenziale
- lo scope del nome, se presente, è il *corpo della funzione* (utile in chiamate ricorsive)

```
var f = function g(x) { return x / 10 }
g(32) //errore, nome g non definito
```

La **function declaration** introduce una funzione senza assegnarla a una variabile

- il nome è essenziale
- lo scope è l'*ambiente di definizione* della funzione

```
function g(x) { return  x / 10 }
g(32) //nome g definito
```

11.1.3 Funzioni innestate e chiusure

Si possono definire funzioni dentro altre, nasce quindi la discussione delle **chiusure**, javascript segue la chiusura *lessicale*.

Una chiusura nasce quando una funzione innestata fa riferimento a *variabili della funzione esterna*.

11.1.4 Currying

Un caso interessante di chiusura è la possibilità di simulare una funzione a N argomenti con N funzioni a 1 argomento.

Si può esprimere una funzione come questa

```
function sum(a, b) { return a + b }
var result = sum(3, 4)
```

in una forma diversa, basata su una funzione "esterna" con argomento `a`, che restituisce una funzione "interna" con argomento `b`.

```
function sum(a) { return function(b) { return a + b } }
var result = sum(3)(4)
```

Questa possibilità si chiama **currying**.

É possibile utilizzare anche la definizione tramite lambda con la seguente sintassi

```
sum = a => b => a + b
var result = sum(3)(4)
```

Il currying è concettualmente interessante perché indica che l'unico "ingrediente" fondamentale per esprimere qualunque funzione sono le *funzioni a un argomento.*

11.1.5 Utilizzi chiusure

Rappresentare uno stato privato e nascosto

Si può ottenere una *proprietà privata* tramite una chiusura, mappando lo stato su un argomento della funzione "generatrice"

```
function incBy2From(x) {
    return function() { return x += 2 }
}
```

Da ogni invocazione di `incBy2From`, nasce un nuova funzione, che ha uno "stato" interno privato e utilizza come punto di partenza il parametro passato al generatore.

Altro esempio, generatore di contatori

```
function genContatore(){
    var contati=0;
    function tick() { return contati++; }
    function num() { return contati;}
    //metodo per restituire più funzioni di accesso
    return { num, tick };
}
```

Realizzare un canale di comunicazione privato

Si può ottenere un canale di comunicazione privato, mettendo in una chiusura sia lo **stato** che i due **metodi accessor**.

Si utilizza un array di due funzioni per restituirli

```
function myChannel() {
    var msg = "bla bla"
    return [ function(x) { msg = x; },    //set
             function() { return msg; } ]  //get
}

var channel = myChannel()
var msg1 = channel[1]() //recupera "bla bla"
channel[0]("bruuh")      //setta il nuovo msg
var msg2 = channel[1]() //recupera "bruuh"
```

Versione con restituzione di object literal con due accessor

```
function canale() {
    var msg = "";
    function set(m) { msg = m }
    function get() { return msg }
    return { set, get }
}

var ch = canale()

document.writeln(ch.set("hello")); // undefined
document.writeln(ch.set("world")); // undefined
document.writeln(ch.get());        // world
```

Realizzare nuove strutture di controllo

Una funzione di secondo ordine incapsula il controllo e gli argomenti delle funzioni rappresentano le azioni da svolgere.

Nuova struttura ciclica loop

```
function loop(statement, n) {
    var k = 0;
    return function iter() {
        if (k < ) { k++; statement(); iter(); }
    }
}

var i = 1
loop(function() { i++ }, 10)();
```

Variante currying

```
function loop(n) {
    return function iter(action) {
        if (n > 0) { action(); n--; iter(action); }
    }
}

loop(5)( function() { document.writeln("ciao") })
loop(5)( () => document.writeln("ciao") )
```

11.1.6 Chiusure e binding delle variabili

A una chiusura corrisponde una sola istanza delle sue variabili, concetto da considerare quando si creano ciclicamente più funzioni.

Ad esempio, creando un'array di funzioni che restituiscono numeri diversi in questo modo

```
function fillFunctionsArray(myarray) {
    for (var i = 4; i < 7; i++)
        myarray[i-4] = function() { return i; };
}
```

si ottiene un array di funzioni che ritornano tutte il valore finale di `i`, ovvero 7.

Per ovviare questo problema, è necessario creare una variabile ausiliaria per ogni funzione che si vuole generare, quindi si utilizza una funzione ausiliaria `aux()` che crea una nuova chiusura, mantenendo il valore della variabile.

```
function fillFunctionsArray(myarray) {
    function aux((j) { return function() { return j; } }
    for (var i = 4; i < 7; i++)
        myarray[i-4] = aux(i);
}
```

11.2 Il lato a oggetti

Javascript adotta un modello fondamentalmente **object-based** (non *oriented*), basato sulla nozione di **prototipo**.

In questo approccio, un oggetto:

- non è istanza di una classe, non esistono vere classi ma solo *simil-classi* come sovrastruttura sintattica
- è solo una *collezione di* ***proprietà pubbliche***, accessibili tramite dot notation

- è costruito tramite `new`, da un costruttore
- è associato a un **oggetto padre**, detto prototipo, di cui eredita le proprietà (**prototype-based inheritance**

11.2.1 Definizione di oggetti

Object literals

Un primo modo per specificare oggetti è l'**object literal**, elencando le proprietà in termini di coppie `nome:valore`.

```
p3 = { x: 10, y: 7 };
```

Si possono definire metodi e utilizzare *stringhe qualunque* per i nomi delle proprietà.

```
p3 = { x: 10, y: 7, getX: funtion() { return this.x; } }

p4 = { "for-me": 10, "for you": "bleah" };
```

Costruzione di oggetti

Un costruttore è una normale funzione, che specifica le proprietà e i metodi dell'oggetto tramite la parola chiave `this`

```
Point = funciton(i, j) {
    this.x = i; this.y = j;
    this.getX = function(){ return this.x; }
    this.getY = function(){ return this.y; }
}

p1 = new Point(3, 4);
```

11.2.2 Proprietà

Accesso proprietà

Dato che tutte le proprietà sono pubbliche, sono anche modificabili (eccetto alcuno proprietà di "sistema").

Per accedere a multiple proprietà di un oggetto si può usare il costrutto `with`:

```
with (p1) x = 22, y = 2
```

Aggiunta e rimozione di proprietà

Le proprietà definite nel costruttore, sono solo le proprietà iniziali dell'oggetto, alle quali se ne possono aggiungere dinamicamente delle nuove.

```
p1.z = -3;
```

É anche possibile rimuovere dinamicamente proprietà, tramite l'operatore `delete`.

```
delete p1.x
```

Ogni oggetto mantiene una proprietà `constructor` che identifica il costruttore.

Proprietà e metodi privati

In generale, le proprietà sono pubbliche, è possibile definirne di private tramite le *chiusure*.

```
Point = function(i,j){
    this.getX = function() { return i; }
    this.getY = function() { return j; }
}
```

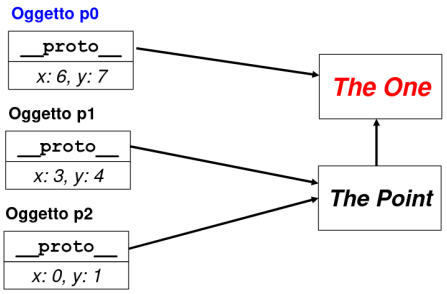
11.2.3 Prototipi di oggetti

Ogni oggetto referenzia il suo prototipo tramite una **proprietà nascosta**, chiamata `__proto__`.

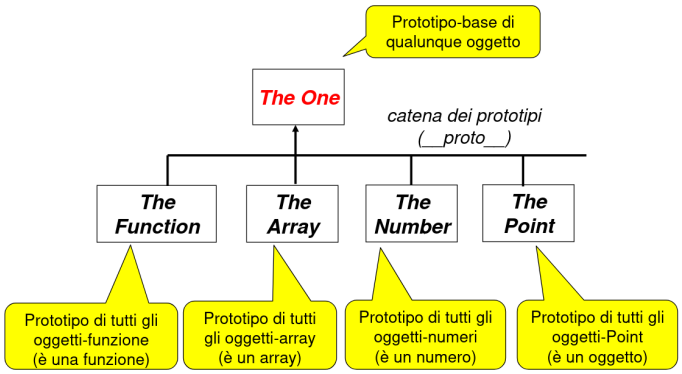
Esiste un *antenato comune* a tutti, le cui proprietà sono ereditate da ogni oggetto.

É antenato **diretto** di ogni *object literal* e antenato **indiretto** di ogni altro oggetto.

```
p0 = { x: 6, y: 7 };
p1 = new Point(3, 4);
p2 = new Point(0, 1);
```



Esiste un prototipo predefinito per ogni categoria di oggetti: funzioni, array, numeri... ma tutti questi prototipo condividono a loro volta l'antenato comune, nasce così la tassonomia di prototipi che esprime l'ereditarietà.



11.2.4 Prototipi di costruzione

Il prototipo di un oggetto dipende da *chi lo costruisce*. La proprietà che definisce quale prototipo viene assegnato agli oggetti creati è `prototype` (riservata ai costruttori).

Spesso questa proprietà è chiamata **prototipo di costruzione**.

Tramite la proprietà `prototype` è possibile referenziare (indirettamente) gli oggetti predefiniti.

11.2.5 Prototipi ed effetti a runtime

Le relazioni fra oggetti espresse dalle catene di prototipi possono essere *modificate a runtime*, in particolare è possibile

- *aggiungere/togliere* proprietà a un prototipo in uso
- *sostituire* un oggetto-prototipo con un altro

Type augmenting

Per **type augmenting** si intende la possibilità di aggiungere proprietà a un prototipo esistente, causando un immediato impatto sugli oggetti già esistenti: la struttura del sistema cambia a runtime.

Per implementare questa funzionalità, si manipola la propritetà prototype del corrispondente costruttore.

Creating e inheriting

In alternativa, è possibile *sostituire il prototipo di costruzione* con un altro, ciò non ha impatto sugli oggetti già esistenti ma avrà conseguenze sugli oggetti costruiti successivamente.

É il modo per definire un ereditarietà prototipale personalizzata, si reimposta la proprietà prototype agganciandola a un opportuno oggetto-prototipo.

11.2.6 Ereditarietà prototype-based

Per avere oggetti in relazione padre-figlio ("is a"), occorre *impostare i prototipi di costruzione* in modo da *riflettere la tassonomia desiderata*

Esempio

Esprimere che la categoria `Studente` *eredita* dalla categoria `Persona`.

Occorre

- definire il costruttore `Persona` con le relative proprietà
- definire il costruttore `Studente` con le relative proprietà

Per creare la gererchia, gli oggetti studente che saranno costruiti, dovranno avere come prototipo una persona.

```
Persona = function(nome, annoNascita){
    this.nome = nome; this.anno = annoNascita;
    this.toString = function() {
        return this.nome + " è nata nel " + this.anno }
}

Studente = function(nome, annoNascita, matricola){
    this.nome = nome; this.anno = annoNascita;
    this.matricola = matricola;
    this.toString = function() {
        return this.nome + " è nata nel " + this.anno +
        " e la sua matricola è " + this.matricola }
}
```

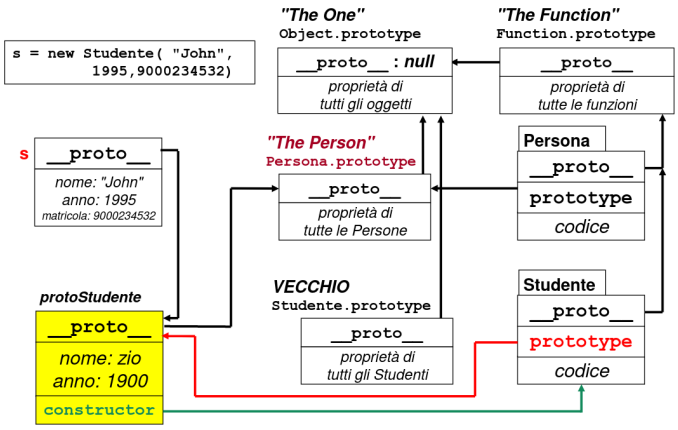
Si crea una specifica `Persona`, da usare come prototipo a tutti gli studenti.

```
protoStudente = new Persona("zio", 1900);
```

Si imposta il prototipo di `Studente` in modo che faccia riferimento a `protoStudente` e il suo costruttore a `Studente`.

```
Studente.prototype = protoStudente;

Studente.prototype.constructor = Studente;
```



Riferimenti alla classe base

Non avendo l'idea di classe, non riesce a riferire la classe base (keyword `super`); tuttavia, tramite il metodo `call`, è possibile chiamare un oggetto-funzione.

```
Studente = function(nome, annoNascita, matricola){
    Persona.call(this, nome, annoNascita);
    this.matricola = matricola;
    this.toString = function() {
        return Studente.prototype.toString.call(this) +
        " e la sua matricola è " + this.matricola }
}
```

11.2.7 Oggetto globale

Javascript prevede un **ambiente globale** che ospita funzioni e variabili definite fuori da ogni altro scope.

É un opportuno oggetto, avente:

- come metodi, tutte le funzioni predefinite
- più tutte le funzioni e le variabili globale definite dall'utente

Non è prestabilito, ma impostato dall'engine che esegue il codice (broswer → `window`, server → `response` ...).

11.2.8 Simil classi

Da ECMAScript 6, sono state introdotte le classi con la sintassi classica, che permette anche l'ereditarietà tramite keyword.

11.3 Dove i due lati si incontrano

11.3.1 Costruire oggetti funzione

Finore abbiamo utilizzato solo function literals, con il costruttore `Function` si possono costruire oggetti funzione dedicati a creare altri oggetti funzione: questo è un costrutto linguistico di **meta-livello**.

Come parametri prende soltanto stringhe, i primi N - 1 sono i nomi dei parametri, l'ultimo è il testo del corpo della funzione da creare.

```
square = new Function("x", "return x*x")
```

12 — Lambda calcolo

Definizione

$$L ::= \lambda x.L|x|LL$$
$$(\lambda x.L_b)L_a \to L_b[L_a/x]$$

L può esprime una qualsiasi **struttura dati** e qualsiasi **algoritmo**.
$\to$ rappresenta una **trasformazione atomica**, basata sull'idea della sostituzione di testo che risulta nel termine L_b con, al suo interno, x sostituite da L_a.

Sintassi

$$L, M, N ::= \lambda x.L|x|LL$$

Un lambda termine può essere:

- x, una *variabile* (λ e . sono simboli terminali)
- $\lambda x.L$, una *funzione*, dove L è il corpo (altro lambda-termine)
- LM, *applicazione di una funzione*

Ambiguità grammaticale

La grammatica di una lambda è *ambigua*, come si interpreta $\lambda x.xy$?

- come $(\lambda x.x)y$ (applicare y a funzione $\lambda x.x$)
- o come $\lambda x.(xy)$

Per disambiguare si utilizzano parentesi o associatività a sinistra, $\lambda x.\lambda y.xyx$ si legge come $\lambda x.(\lambda y.(xy)x)$

Semantica

La semantica è definita tramite *regole di trasformazione*:

$$(\lambda x.L)M \to L[M/x]$$

Si identifica un pattern del tipo $(\lambda x.L)M$ e lo si trasforma nel lambda-termine $L[M/x]$, sostituendo, nella funzione L, ogni occorrenza x con M.

Esempio

$(\lambda x.xx)(\lambda y.(\lambda z.yz))$ si riduce a $(\lambda y.(\lambda z.yz))(\lambda y.(\lambda z.yz))$, in quanto la lambda $(\lambda x.xx)$ duplica il suo parametro formale x, che in questo caso assume il valore di $(\lambda y.(\lambda z.yz))$.

Forme normalizzate

Un lambda termine è in **forma normale** se non si possono applicare altre riduzioni.

Si dice che un lambda termine è:

- fortemente normalizzabile, se qualunque sequenza di riduzioni porta a una forma normale
- *debolmente* normalizzabile, se esiste *almeno una* sequenza di riduzioni che porta a una forma normale
- *non* normalizzabile, se non si arriva mai a una forma normale

12.1 Teorema di Church-Rosser - Terminazione delle computazioni

> Ogni lambda ha al più **una** forma normale, questa può essere interpretata come risultato.
>
> Il lambda calcolo è deterministico.

12.2 Strategie di riduzione

L'idea di ridurre il più possibile, introduce il concetto di *strategia di riduzione*, scegliendo la strategia errata, le sostituzioni potrebbero non terminale.

12.2.1 Principi

Adottare una o l'altra strategia, si possono ottenere sostituzioni *infinite* o *finite*.

Due principi alternativi:

- **eager evaluation**: privilegia la valutazione dell'argomento, valuta l'argomento il prima possibile
- **lazy evaluation**: privilegia l'applicazione dell'argomento alla funzione, valuta l'argomento il più tardi possibile

Da questi principi sono state sviluppate varie strategia, usate nei vari linguaggi.

12.2.2 Panoramica

- strategie *eager* (strict):
  - applicative order (rightmost innermost)
    - prima riduce poi applica gli argomenti
    - può non trovare la forma normale
  - call by value
  - call by reference
  - call by copy-restore
- strategie *lazy* (non-strict)
  - normal order (leftmost outermost)
    - prima riduce la lambda più a sinistra
    - normale perché se esiste una forma normale, la trova
  - call by name
  - call by need
  - call by macro

12.2.3 Strategie eager

Applicative order

Prima valuta gli argomenti, poi li applica alle funzioni. Tenta di ridurre il più possibile i termini dentro la funzione prima di applicare gli argomenti. **Non normalizzante**.

Call by value

Non riduce i termini prima di applicare gli argomenti. É una famiglia di strategia che differiscono per l'ordine di valutazione degli argomenti (left to right, right to left, undefined). Se il valore è un puntatore, si ha lo stesso effetto di call-by-reference.

Call by reference

La funzione riceve un *riferimento implicito* all'argomento del chiamante, anziché una copia dello stesso.

Call by copy-restore

Caso speciale della call-by-reference adatto al multi-threading: ogni funzione riceve una copia privata dell'argomento e lo ricopia indietro alla fine, evita interferenze ma solo l'ultimo valore sopravvive.

12.2.4 Strategie lazy

Normal order

Applica gli argomenti alle funzioni prima di ridurli, riducendo per prima la lambda più a sinistra e considerando come corpo ciò che compare a destra.