

Indice

1	Linguaggi, macchine astratte, teoria della computabilità	5
1.0.1	Esecuzione	5
1.1	Macchine astratte	6
1.1.1	Automa esecutore	6
1.1.2	Gerarchia di macchine astratte	6
1.1.3	Macchina base	6
1.1.4	Automa a stati finiti	7
1.1.5	Macchina di Turing	7
1.1.6	Macchine universali	9
1.2	Teoria della computabilità	10
1.3	Funzioni	10
1.3.1	Funzione caratteristica	10
1.3.2	Funzione computabile	10
1.3.3	Funzioni definibili vs funzioni computabili	11
1.3.4	Funzioni non computabili	11
1.3.5	Generabilità e decidibilità	12
1.3.6	Insiemi decidibili	13
2	Linguaggi e grammatiche	15
3	Riconoscitori a stati finiti	17
3.1	Dai riconoscitori ai generatori	17
3.1.1	Mapping RSF \longleftrightarrow grammatica	18
3.1.2	Riconoscitori top down	19
3.1.3	Riconoscitori bottom up	19
3.1.4	Dall'automa alle grammatiche	20
4	Riconoscitori con PDA	23
4.1	Push-Down Automaton (PDA)	23
4.1.1	PDA non deterministici	25
4.1.2	PDA deterministici	26

4.1.3	Realizzazione di PDA deterministici	26
4.1.4	Separare motore e grammatica	27
4.1.5	Analisi ricorsiva discendente - vantaggi e limiti	28
4.2	Grammatiche $LL(k)$	28
4.2.1	Starter Symbol Set	29
4.2.2	Parsing table con blocchi annullabili	30
4.2.3	Director Symbols Set	31
5	Dai riconoscitori agli interpreti	33
5.1	Interprete	33
5.1.1	Struttura	33
5.1.2	Analisi lessicale	34
5.1.3	Parole chiave	34
5.1.4	Tabelle	34
5.1.5	Analisi sintattica top-down	35
5.2	Caso di studio - espressioni aritmetiche	35
5.2.1	Analisi del dominio	35
5.2.2	Grammatica per le espressioni	36
5.2.3	Una grammatica a "strati"	36
5.2.4	Variante 1 - Associativa a destra	38
5.2.5	Variante 2 - Non associativa	38
5.3	Dalla grammatica al parser	39
5.3.1	Ricorsione sinistra e analisi top-down	39
5.4	Dal parser al valutatore	40
5.4.1	Specificare la semantica	40
5.4.2	Semantica denotazionale	41
5.4.3	Elevamento a potenza	42
5.5	Rappresentare le frasi	43
5.5.1	Alberi sintattici astratti	43
5.5.2	Sintassi astratta	44
5.6	Architettura interprete	44
5.7	Valutazione di alberi	45
5.8	Valutatore	45
5.8.1	Implementazione	46
5.8.2	Visitor come interprete	47
6	Estensione: assegnamenti, ambienti, sequenze	49
6.1	Assegnamento	49
6.1.1	L-Value vs R-Value	49
6.2	Environment	49

6.2.1	Semantica	50
6.2.2	Environment multipli	50
6.3	Scelta del tipo di assegnamento	51
6.3.1	Estensione dell'interprete	51
6.4	Espressioni sequenza	53
7	Multi-Paradigm Programming: parser espressioni in Prolog	55
7.0.1	Definizione nuovi operatori	55
7.0.2	Problema delle parentesi tonde	56
7.0.3	Espressioni in Prolog	56
7.0.4	Dal riconoscitore al valutatore	56
7.1	Parser ibrido con 2p-kt	57
7.1.1	Primi passi per utilizzare il Prolog engine	57
7.1.2	TermParser	57
8	Strumenti per la generazione automatica di riconoscitori LL	59
8.1	Domain-Specific Languages	59

1 — Linguaggi, macchine astratte, teoria della computabilità

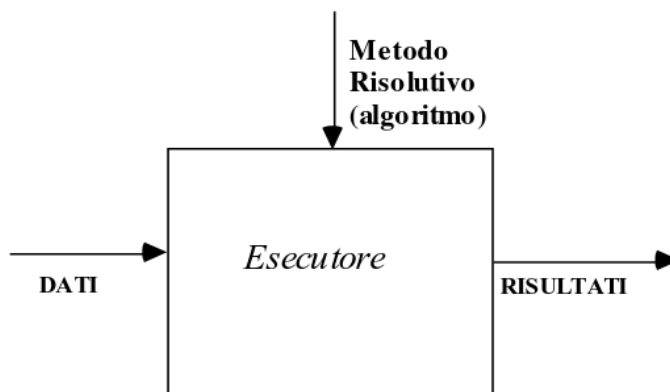
Alcune definizioni

- **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*
- $mfn : I \rightarrow O$ = 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*
- $mfn : I \times S \rightarrow O$ = funzione di macchina
- $sfn : I \times S \rightarrow S$ = 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

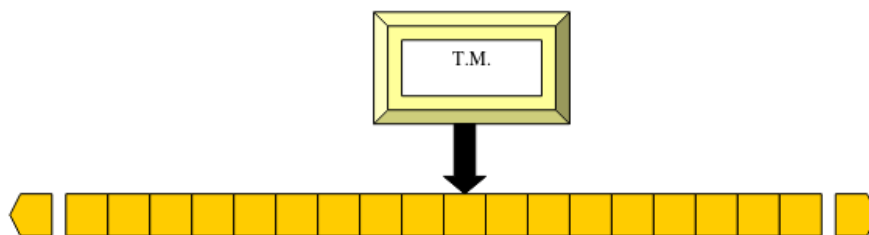
Per ovviare al limite della memoria finita, si introduce un *nastro* di memoria esterno, la **macchina di Turing** è definita dalla quintupla:

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

dove

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

Il deposito dei dati è rappresentato da un nastro 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 funzione **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

$$f_p : X \rightarrow Y$$

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 $f : A \rightarrow B$ 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

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

Funzione computabile su \mathbb{N}

Per semplicità, considereremo funzioni sui *numeri naturali*

$$f : \mathbb{N} \rightarrow \mathbb{N}$$

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, \dots, N_k i numeri naturali dati
- siano P_1, P_2, \dots, P_k i primi k numeri primi

Si definisce R , un nuovo numero naturale così definito

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

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

Poiché che l'insieme $F = \{f : \mathbb{N} \rightarrow \mathbb{N}\}$ 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, sfortunatamente 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 irrisolvibili, 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

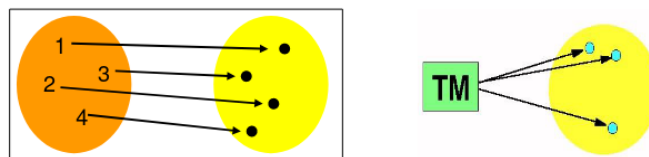
Data la definizione di un insieme numerabile, ovvero un insieme i cui elementi possono essere "contati" cioè che possiede una funzione biettiva:

$$f : N \rightarrow S$$

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 $f : N \rightarrow S$ è *computabile*.



Tuttavia, il fatto che l'insieme possa essere costruito, non significa che si possa decidere se un certo elemento vi appartiene 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 insieme 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

Occorre un concetto più potente della semi-decidibilità.

Un insieme S è **decidibile** (o ricorsivo) se la sua funzione caratteristica 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

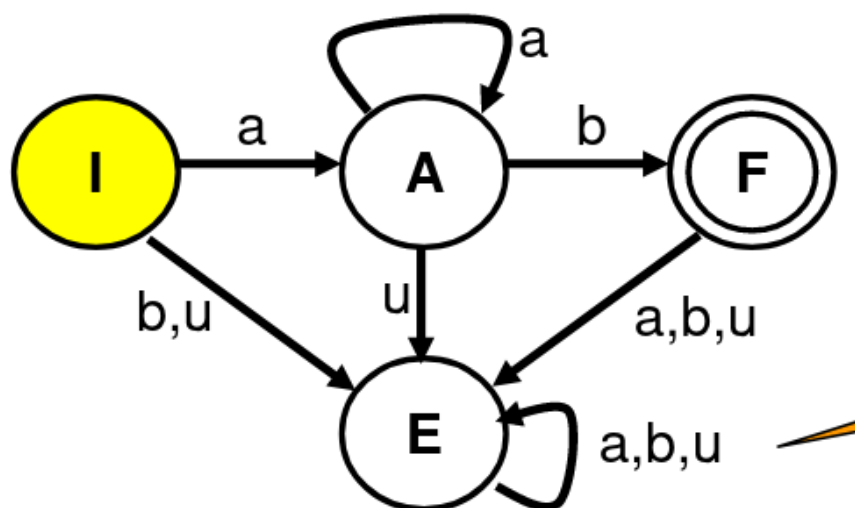
empty

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 può 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

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

Grammatica regolare a destra:

- scopo = stato iniziale: I
- stato finale: F

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

Grammatica regolare a sinistra:

- scopo = stato finale: F
- stato iniziale: I

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

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:

- | | |
|---|---|
| • <u>dove si va</u> | • <u>da dove viene</u> |
| – la freccia col simbolo terminale | – lo stato precedente |
| – lo stato successivo | – 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

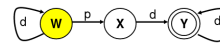
ESEMPIO di RSF

Sia:

- ♦ $A = \{d, p, u\}$
- ♦ $S = \{W, X, Y, Z\}$
- ♦ $S_0 = W$
- ♦ $F = \{Y\}$
- ♦ $sfn: A \times S \rightarrow S$

	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:

$$W \rightarrow d W \mid p X$$

$$X \rightarrow d \mid d Y$$

$$Y \rightarrow d \mid d Y$$

$$Y \rightarrow X d \mid Y d$$

$$X \rightarrow p \mid W p$$

$$W \rightarrow d \mid W d$$

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:

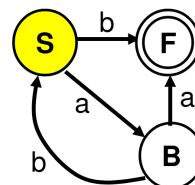
- ha tanti **stati** quanti i simboli non terminali
- ha come **stato iniziale** lo scopo S
- per ogni regola del tipo $X \rightarrow x Y$, l'automa con ingresso x passa dallo stato X allo stato Y
- per ogni regola del tipo $X \rightarrow x$, l'automa con ingresso x passa dallo stato X allo stato finale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

Esempio

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

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

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



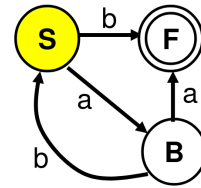
3.1.3 Riconoscitori bottom up

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

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

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

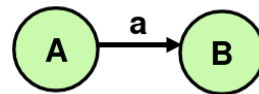


3.1.4 Dall'automa alle grammatiche

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).

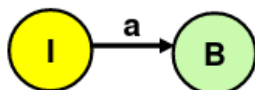
Caso generico

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



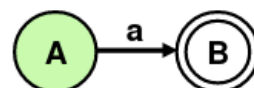
Caso iniziale

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

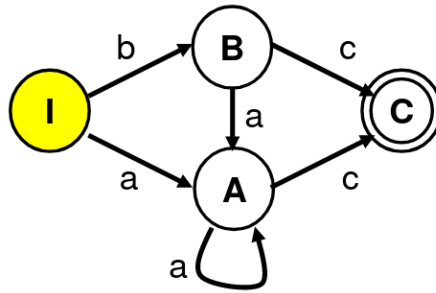


Caso finale

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



Esempio



Analisi top-down (C finale omissa):

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

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:

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

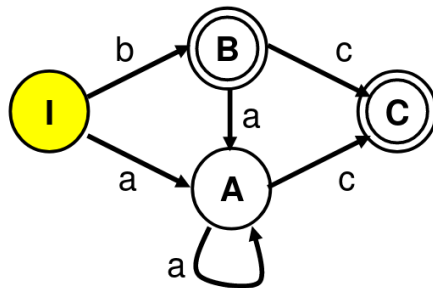
$$\begin{aligned}
 L(G1) &= \\
 & b L(B) + a L(A) = (ba+a) L(A) + bc = \\
 & = (ba+a)a^*c + bc = ba^*c + aa^*c = \\
 & = (b+a) a^* c
 \end{aligned}$$

Grammatica G2 regolare a sinistra:

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

$$\begin{aligned}
 L(G2) &= \\
 & L(B) c + L(A) c = (b + L(A)) c = \\
 & (b + (b a + a) a^*) c = (b + a) a^* c \\
 & \text{perché } (b + b a a^*) = b a^*
 \end{aligned}$$

Esempio multipli stati finali



Analisi top-down (regola in più sulla I): Analisi bottom-up (C iniziale):

- | | |
|--------------------------------|------------------------|
| • $I \rightarrow b \ B \mid b$ | • $b \leftarrow B$ |
| • $I \rightarrow a \ A$ | • $a \leftarrow A$ |
| • $B \rightarrow a \ A$ | • $B \ a \leftarrow A$ |
| • $A \rightarrow a \ A$ | • $A \ a \leftarrow A$ |
| • $B \rightarrow c$ | • $B \ c \leftarrow C$ |
| • $A \rightarrow c$ | • $A \ c \leftarrow C$ |

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 $C \equiv S$):

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

$L(G2') = (b + a) a^* c$

Grammatica $G2''$ (assume $B \equiv S$):

$b \leftarrow S$

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 $S \rightarrow S_1 \mid S_2 \mid \cdots \mid S_n$

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 $L = ({}^nc)^n, n \geq 0, G = S \rightarrow (S) \mid c$

In questo linguaggio il prefisso $({}^nc$ 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.

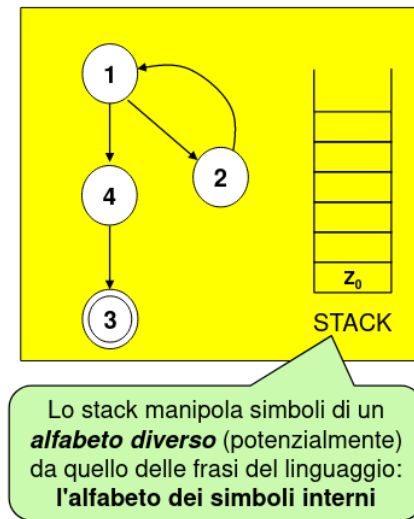
Un PDA è una sestupla:

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

dove:

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

Figura 4.1: Struttura PDA



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

- **Criterio dello stato finale:** il linguaggio accettato è l'insieme di tutte le stringhe di ingresso 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 $s' = \text{sfn}(a, s, z)_{\Pi S}$
- effettua una **PUSH** sullo stack di zero o più simboli interni $z' \in Z$
 $z' = \text{sfn}(a, s, z)_{\Pi S}$

Si consideri il linguaggio generato da:

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

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

Definiamo il PDA come segue:

- $A = \{0, 1, c\}$
- $S = \{Q_1 = S_0, Q_2\}$
- $Z = \{Zero, Uno, Centro\}$

$A \cup \varepsilon$	S	Z	$S \times Z^*$
0	Q1	Centro	Q1 \times CentroZero
1	Q1	Centro	Q1 \times CentroUno
c	Q1	Centro	Q2 \times Centro
0	Q1	Zero	Q1 \times ZeroZero
1	Q1	Zero	Q1 \times ZeroUno
c	Q1	Zero	Q2 \times Zero
0	Q1	Uno	Q1 \times UnoZero
1	Q1	Uno	Q1 \times UnoUno
c	Q1	Uno	Q2 \times Uno
0	Q2	Zero	Q2 $\times \varepsilon$
1	Q2	Uno	Q2 $\times \varepsilon$
ε	Q2	Centro	Q2 $\times \varepsilon$

4.1.1 PDA non deterministici

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

Ad esempio, il PDA tale che $sf n(Q_0, a, Z) = \{(Q_1, Z_1)(\dots)(Q_k, Z_k)\}$ 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 $sf_n(Q_i, x, Z) = \{(Q_1, Z_1), (\dots), (Q_k, Z_k)\}$
- optare se leggere o non leggere il simbolo di ingresso x a causa della presenza di entrambe le mosse $sf_n(Q_i, x, Z)$ e $sf_n(Q_i, \varepsilon, Z)$

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 con 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 inteno 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 definizione, 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 incontra 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 $L = \{\text{word } c \text{ word}^n\}$, alfabeto $A = \{0, 1, c\}$ e regole $S \rightarrow 0 S 0 \mid 1 S 1 \mid c$.

- 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 \rightarrow seguire la prima regola
 - se il simbolo d'ingresso è 1 \rightarrow seguire la seconda regola
 - se il simbolo d'ingresso è c \rightarrow 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 funzione 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 $L = \{\text{word } c \text{ word}^n\}$

	0	1	c
S	$S \rightarrow 0 S 0$	$S \rightarrow 1 S 1$	$S \rightarrow c$

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

Produzioni $S \rightarrow \text{if } c \text{ then cmd } X, X \rightarrow \text{endif | else cmd}$

	if	c	then	endif	else	cmd
S	$S \rightarrow \text{if } c \text{ then cmd } X$	<i>error</i>	<i>error</i>	<i>error</i>	<i>error</i>	<i>error</i>
X	<i>error</i>	<i>error</i>	<i>error</i>	$X \rightarrow \text{endif}$	$X \rightarrow \text{else cmd}$	<i>error</i>

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

$$\begin{array}{lcl} S \rightarrow p X & | & q Y \\ X \rightarrow a X b & | & x \\ Y \rightarrow a Y d & | & y \end{array}$$

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 deterministico.

	p	q	a	b	d	x	y
S	$S \rightarrow p X$	$S \rightarrow q Y$	<i>error</i>	<i>error</i>	<i>error</i>	<i>error</i>	<i>error</i>
X	<i>error</i>	<i>error</i>	$X \rightarrow a X b$	<i>error</i>	<i>error</i>	$X \rightarrow x$	<i>error</i>
Y	<i>error</i>	<i>error</i>	$Y \rightarrow a Y d$	<i>error</i>	<i>error</i>	<i>error</i>	$Y \rightarrow y$

La frase di inizio 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 $Z \rightarrow S \$$.

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 \rightarrow Starter Symbols Set.

Lo starter symbols set della riscrittura α è l'insieme

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

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 $\alpha \rightarrow \varepsilon$, si introduce l'insieme

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

dove trunc_1 denota il troncamento della stringa al primo elemento.

Generalizzando la regola precedente, 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 start symbols set con i Director Symbols set (o Look-Ahead 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* quelli 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

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

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

La regola $A \rightarrow P Q$ 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	$S \rightarrow A B$	$S \rightarrow A B$	$S \rightarrow A B$	error	$S \rightarrow A B$	error
A	$A \rightarrow P Q$	$A \rightarrow P Q$	$A \rightarrow B C$ $A \rightarrow P Q$	error	$A \rightarrow B C$ $A \rightarrow P Q$	error
B	error	error	$B \rightarrow b B$	error	$B \rightarrow d$	error
C	error	error	error	$C \rightarrow c C$	error	$C \rightarrow f$
P	$P \rightarrow p P$	$P \rightarrow \varepsilon$	$P \rightarrow \varepsilon$	error	$P \rightarrow \varepsilon$	error
Q	error	$Q \rightarrow q Q$	$Q \rightarrow \varepsilon$	error	$Q \rightarrow \varepsilon$	error

4.2.3 Director Symbols Set

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

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

Il *Director Symbols set* della produzione della $A \rightarrow \alpha$ è l'unione di due insiemi:

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

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

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

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

In pratica

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

o anche:

$$DS(A \rightarrow \alpha) = \text{trunc}_1(\text{FIRST}(\alpha) \cdot FOLLOW(A))$$

ossia quello che si ottiene prendendo l'iniziale delle frasi ottenute concatenando $\text{FIRST}(\alpha)$ 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**.

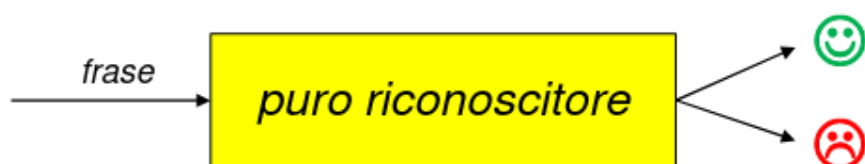
Questi set sono spesso chiamati **Look-Ahead set** perché servono per "guardare avanti" per guidare deterministicamente il parser.

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.

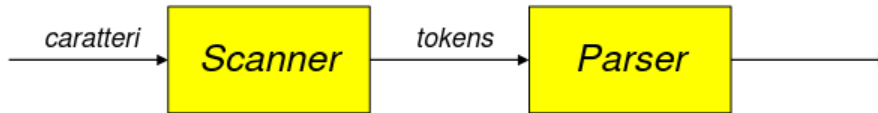
In questo caso diventa importante la sequenza di derivazione.

5.1 Interprete

5.1.1 Struttura

Un interprete è solitamente basato su due componenti:

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



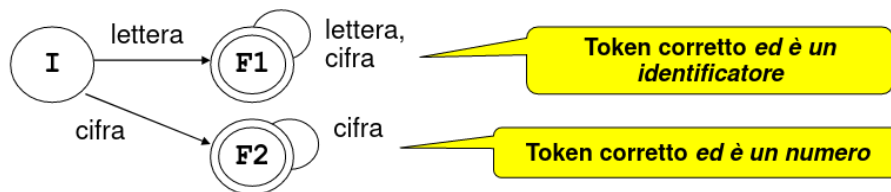
Lo scanner analizza le parti *regolari* dei linguaggi, fornendo al parser singoli token già aggregati.

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.

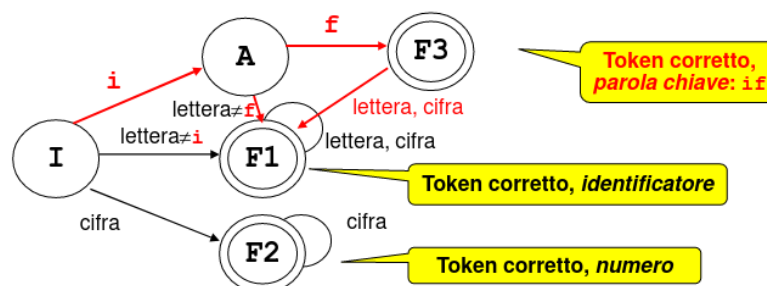
L'analizzatore lessicale categorizza i token osservando in quale stato finale del RSF si ritrova.



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 sì 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 valore 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 , \div .

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;

$$\begin{aligned} VN &= \{\text{EXP}\} \\ VT &= \{+, *, -, :, \text{num}\} \\ S &= \text{EXP} \\ P &= \{ \\ &\quad \text{EXP} := \text{EXP} + \text{EXP} \\ &\quad \text{EXP} := \text{EXP} - \text{EXP} \\ &\quad \text{EXP} := \text{EXP} * \text{EXP} \\ &\quad \text{EXP} := \text{EXP} : \text{EXP} \\ &\quad \text{EXP} := \text{num} \\ &\} \end{aligned}$$

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

5.2.3 Una grammatica a "strati"

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

$$\begin{aligned} VN &= \{\text{EXP}, \text{TERM}, \text{FACTOR}\} \\ VT &= \{+, *, -, :, (,), \text{num}\} \\ S &= \text{EXP} \\ P &= \{ \\ &\quad \text{EXP} := \text{TERM} \\ &\quad \text{EXP} := \text{EXP} + \text{TERM} \\ &\quad \text{EXP} := \text{EXP} - \text{TERM} \end{aligned}$$

```

    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

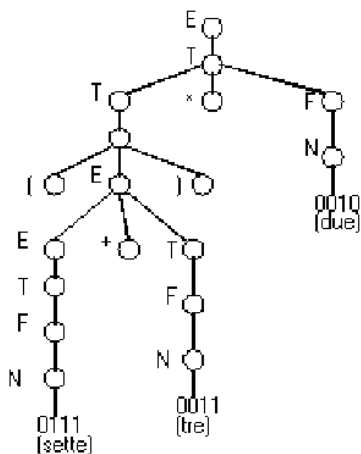
EXP ::= TERM	Priorità MIN
EXP ::= EXP + TERM	
EXP ::= EXP - TERM	
TERM ::= FACTOR	Priorità MED
TERM ::= TERM * FACTOR	
TERM ::= TERM : FACTOR	
FACTOR ::= num	Priorità MAX
FACTOR ::= (EXP)	

Esempi

Frase:

(0111 + 0011) * 0010

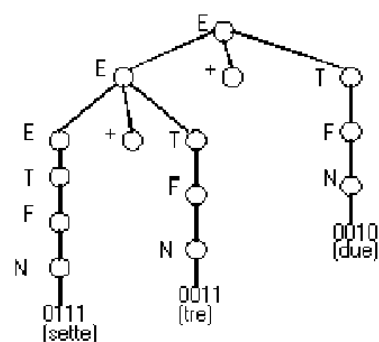
Albero di derivazione



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 $(7 - 3) - 2$.

$$\begin{aligned} \text{VN} &= \{\text{EXP}, \text{TERM}, \text{FACTOR}\} \\ \text{VT} &= \{+, *, -, :, (,), \text{num}\} \\ P &= \{ \\ &\quad \text{EXP} := \text{TERM} \\ &\quad \text{EXP} := \text{TERM} + \text{EXP} \\ &\quad \text{EXP} := \text{TERM} - \text{EXP} \\ &\quad \text{TERM} := \text{FACTOR} \\ &\quad \text{TERM} := \text{FACTOR} * \text{TERM} \\ &\quad \text{TERM} := \text{FACTOR} : \text{TERM} \\ &\quad \text{FACTOR} := \text{num} \\ &\quad \text{FACTOR} := (\text{EXP}) \\ &\} \end{aligned}$$

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, né destra né sinistra, risolvendo il problema *obbligando a usare le parentesi*.

$$\begin{aligned} \text{VN} &= \{\text{EXP}, \text{TERM}, \text{FACTOR}\} \\ \text{VT} &= \{+, *, -, :, (,), \text{num}\} \\ P &= \{ \\ &\quad \text{EXP} := \text{TERM} \\ &\quad \text{EXP} := \text{TERM} + \text{TERM} \\ &\quad \text{EXP} := \text{TERM} - \text{TERM} \\ &\quad \text{TERM} := \text{FACTOR} \\ &\quad \text{TERM} := \text{FACTOR} * \text{FACTOR} \\ &\quad \text{TERM} := \text{FACTOR} : \text{FACTOR} \\ &\quad \text{FACTOR} := \text{num} \\ &\quad \text{FACTOR} := (\text{EXP}) \\ &\} \end{aligned}$$

5.3 Dalla grammtica al parser

5.3.1 Ricorsione sinistra 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:

$$L(\text{EXP}) = \text{TERM} \pm \text{TERM} \pm \text{TERM} \dots$$

$$L(\text{TERM}) = * / : \text{FACTOR} * / : \text{FACTOR} \dots$$

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

Continuiamo mappando le espressioni individuate seconda la notazione EBNF:

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

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

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:

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

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

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

$$\text{AFTERFACTOR} = \varepsilon \mid * \text{TERM} \mid : \text{TERM}$$

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 definire 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

È 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 quello di definire una *funzione di interpretazione*:

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

5.4.2 Semantica denotazionale

Quando la semantica di un linguaggio 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**

```

fExpr (s) = fTerm (s)
fExpr (s1 + s2) =
    fExpr (s1) + fTerm (s2)
fExpr (s1 - s2) =
    fExpr (exp) - fTerm (s2)
fTerm (s) = fFactor (s)
fTerm (s1 * s2) =
    fTerm (s1) × fFactor (s2)
fTerm (s1 : s2) =
    fTerm (term) / fFactor (s2)
fFactor ( (s) ) = fExpr (s)
fFactor (num) = valueof (num)
  
```

In **rosso**: simboli
della grammatica

In **nero**: operazioni
"note" sul dominio Z

Esempio

Nel complesso, l'espressione: $3 + 4 * 18 : (7 - 1)$
ha come significato (ossia, valore):

<i>tre</i>	+	<i>fTerm(4 * 18 : (7 - 1))</i>		<i>fFactor(7 - 1)</i>
		<i>fTerm(4 * 18)</i>	/	<i>fExpr(7 - 1)</i>
		<i>fTerm(4) × fFactor(18)</i>		<i>sette - uno</i>
		<i>quattro × diciotto</i>		<i>sei</i>
		<i>settantadue</i>	/	
			<i>dodici</i>	
<i>tre</i>	+		<i>dodici</i>	
		<i>quindici</i>		

Schema interprete

Ogni funzione analizza il sotto-linguaggio di pertinenza:

- `parseExp` analizza $L(\text{EXP})$, per il quale $+$, $-$ e `TERM` sono l'alfabeto terminale
- `parseTerm` analizza $L(\text{TERM})$, per il quale $*$, $:$ e `FACTOR` sono l'alfabeto terminale
- `parseFactor` analizza $L(\text{FACTOR})$, 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 \wedge , che denota un elevamento a potenza: $3 * 4^2$ si scrive $3 * 4 \wedge 2$

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 $4 \wedge 2 \wedge 3$ come 4^8 .

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

EXP	::=	TERM
EXP	::=	EXP + TERM
EXP	::=	EXP - TERM
TERM	::=	POT
TERM	::=	TERM * POT
TERM	::=	TERM : POT
POT	::=	FACTOR
POT	::=	FACTOR \wedge POT
FACTOR	::=	num
FACTOR	::=	(EXP)

associativo a destra

-----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 gli **alberi sintattici**.

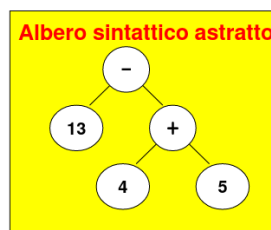
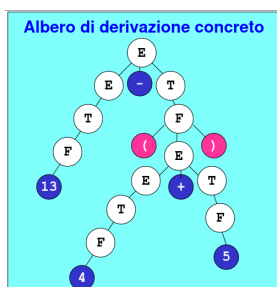
5.5.1 Alberi sintattici astratti

Si potrebbe usare direttamente 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 intrinsecamente 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.

- 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 \rightarrow figli (da sx a dx)
- *post-order*, figli (da sx a dx) \rightarrow , radice
- *in-order*, figlio sx \rightarrow radice \rightarrow figlio dx

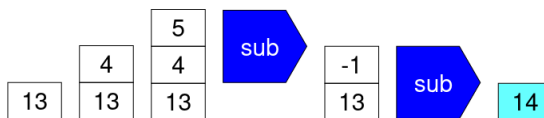
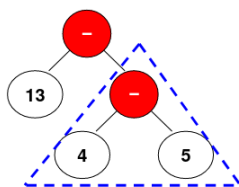
Nel nostro dominio delle espressioni aritmetiche, queste metodologie si traducono in:

- 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

È 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 interpretazioni**, 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.

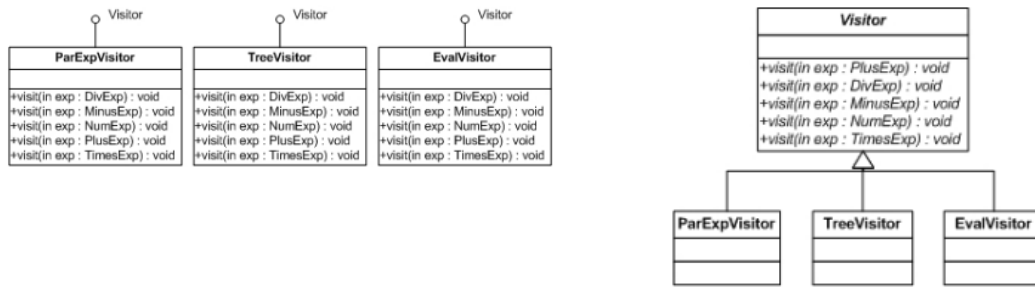
É necessario unire i pro delle due metodologie, a questo compito si presta perfettamente il **pattern Visitor**, che ci permette di incapsulare la *logica d'interpretazione* in una entità di livello più alto.

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 \rightarrow **ParExpVisitor**
- visitor per *calcolare* le espressioni \rightarrow **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'assegnamento 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

simbolo	valore
a	3
y	5
...	...

The diagram shows a table with two columns: 'simbolo' and 'valore'. The rows are 'a', 'y', and '...'. The row containing 'y' is highlighted with a red box. Below the table, there are two red labels: 'L-value' and 'R-value'. Red lines connect these labels to the 'y' row. The 'L-value' line starts from the 'y' in the 'simbolo' column and goes down. The 'R-value' line starts from the '5' in the 'valore' column and goes down.

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 specificare 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

È 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*, poicè il valore è l'ultimo elemento.

$$x = (y = (z = \text{valore}))$$

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 mantenere 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:

$$P = \{$$

$$\dots$$

$$\text{EXP} := \text{ASSIGN}$$

$$\text{ASSIGN} := \text{IDENT} = \text{EXP}$$

```

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

```

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

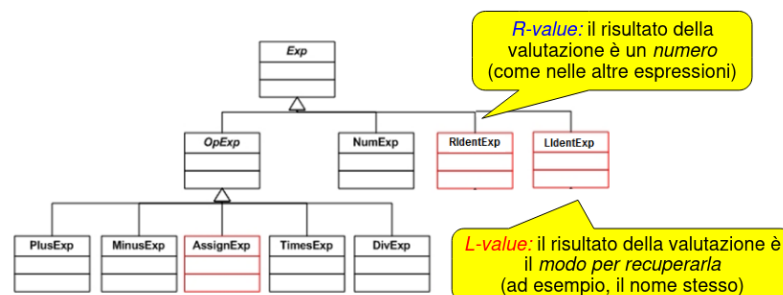
- nodo *operazione assegnamento* \rightarrow AssignExp
- nodo *L-value* (IDENT a sinistra) \rightarrow LIdentExp
- nodo *R-value* (IDENT a destra) \rightarrow RIdentExp

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 comodamente 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`

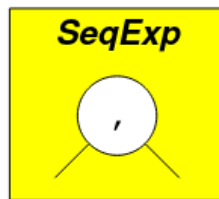
6.4 Espressioni sequenza

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 il valore complessivo è quello dell'exp più a destra:

$$\text{EXP} := \text{ASSIGN}, \text{EXP}$$

Si identifica 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(livellopriorità, associatività, operatore)
```

- `livellopriorità`: 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.getDefaultOperators();
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

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

Per creare la query a partire dalla rappresentazione testuale

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

8 — Strumenti per la generazione automatica di riconoscitori LL

Esistono moltissimi strumenti chiamati *parser generator*, o compiler compiler, questi, data una grammatica (annotata), producono automaticamente il riconoscitore in un linguaggio prescelto.

Alcuni seguono l'approccio LL, altri quello LR:

- LL è efficiente e facilita l'aggiunta di azioni semantiche
- LR è più potente (vedi ricorsione sinistra) ma più complesso

Oltre alla generazione del parser tuttavia, è necessario integrarsi con strumenti di sviluppo, come impostare validazione, segnalazioni di errori, content assists, quick fix e altro.

8.1 Domain-Specific Languages

Al contrario dei GPL, General Purpose Languages, i **DSL** sono realizzati per essere utilizzati in **specifici domini applicativi**, non mirano a *risolvere ogni problema* ma a offrire soluzioni efficaci per uno specifico ambito.

Esistono strumenti a supporto dei DSL, utili per la *generazione automatica* dei DSL e del supporto, il più diffuso è **Xtext**.