



POLITECNICO DI MILANO

# Linguaggi formali e compilatori

---

Appunti

Stefano Invernizzi

Anno accademico 2010-2011



## Sommario

Introduzione ai linguaggi .....	5
Terminologia di base .....	5
Operazioni sulle stringhe .....	6
Sottostringhe .....	7
Operazioni sui linguaggi .....	7
Proprietà degli operatori tra linguaggi .....	9
Inizi, fini e digrammi .....	9
Espressioni regolari .....	10
Introduzione intuitiva .....	10
Terminologia di base .....	10
Derivazione .....	11
Linguaggi regolari .....	12
Ambiguità delle espressioni regolari .....	12
Astrazione linguistica: liste .....	13
Grammatiche libere .....	14
Le grammatiche generative .....	14
Derivazione .....	14
Linguaggio generato .....	15
Grammatiche libere e linguaggi liberi .....	15
Classificazione delle regole .....	16
Grammatica lineare .....	17
Eliminazione dei non terminali non raggiungibili .....	17
Eliminazione dei non terminali non ben definiti .....	18
Alberi sintattici .....	18
Derivazioni canoniche .....	19
Ambiguità di una grammatica .....	19
Eliminazione delle ambiguità .....	20
Equivalenza tra grammatiche .....	21
Forme normali .....	21
Trasformazioni elementari per ottenere le forme normali .....	22
Come ottenere le forme normali .....	23
Il linguaggio di Dyck .....	24
Grammatica delle espressioni aritmetiche .....	24
Confronto tra REG e LIB .....	25
Ottenere la grammatica libera dall'espressione regole .....	25
Le grammatiche unilineari: un formalismo equivalente alle e.r. ....	25
Proprietà di chiusura dei linguaggi regolari e liberi .....	26
Proprietà di chiusura di <i>REG</i> .....	26
Proprietà di chiusura di <i>LIB</i> .....	26
Classificazione delle grammatiche .....	28
Le classificazione di Chomsky .....	28
Grammatiche libere estese o EBNF .....	29
Le grammatiche libere estese (EBNF) .....	29
Automi finiti .....	30
Gli automi finiti .....	30
Automa pulito .....	31
Automa minimo .....	31
Automi non deterministici .....	32
Operazioni sugli automi non deterministici .....	33
Ambiguità degli automi .....	33
Dalla grammatica all'automa a stati finiti e viceversa .....	33
Trasformazione da AF non deterministico ad AF deterministico .....	35
Dall'automa all'espressione regolare: il metodo BMC .....	35
Dall'espressione regolare all'automa riconoscitore .....	36
Determinizzazione di automi mediante algoritmo BS .....	38
Automi a stati finiti e operazioni tra linguaggi .....	38

Automi a pila.....	40
Gli automi a pila .....	40
Proprietà e osservazioni .....	41
Dalla grammatica all'automa a pila .....	41
Linguaggi deterministici .....	42
Automa a pila riconoscitore dell'intersezione.....	42
Analisi sintattica discendente LL(k).....	43
L'analisi sintattica discendente e ascendente .....	43
Premessa per l'analisi sintattica LL(k): grammatica come rete di AF.....	43
Analisi sintattica discendente deterministica LL(1) .....	45
Analisi sintattica ascendente LR(k) .....	47
Analisi sintattica ascendente deterministica.....	47
Concetti preliminari per l'analisi LL(0): la macchina pilota .....	48
Analisi LR(k) .....	50
Analisi LALR(1) .....	52
Esempio .....	53
Analisi sintattica con il metodo di Early .....	56
Il metodo di Early .....	56
Il procedimento .....	56
Altre osservazioni .....	57
Esempio .....	57
Analisi statica dei programmi .....	58
L'analisi statica dei programmi.....	58
Grafo di controllo del flusso .....	58
Intervallo di vita delle variabili e definizioni inutili.....	59
Definizioni raggiungenti .....	60
Disponibilità di una variabile .....	60
Traduzione e analisi semantica.....	61
La traduzione sintattica .....	61
Grammatiche di traduzione .....	62
Traduzioni regolari .....	63
Grammatiche ad attributi.....	64
Analisi sintattica e semantica integrate .....	68
Laboratorio .....	69
Compilatore.....	69
L'analisi lessicale.....	70
L'analisi semantica .....	73
ACSE .....	76
Sorgenti ACSE .....	77

# Introduzione ai linguaggi

## Terminologia di base

Definizioni dei termini fondamentali usati all'interno del corso:

### **Alfabeto**

Un alfabeto, spesso indicato con la lettera greca  $\Sigma$  (sigma maiuscola), è un insieme di simboli.

### **Simbolo o terminale o carattere terminale di un alfabeto**

Un simbolo di un alfabeto, detto anche terminale o carattere terminale dell'alfabeto, è un elemento dell'alfabeto stesso.

### **Cardinalità dell'alfabeto**

La cardinalità dell'alfabeto  $\Sigma$ , indicata con  $|\Sigma|$ , è il numero di simboli dell'alfabeto stesso.

### **Stringa o parola o frase**

Una stringa (o parola o frase) dell'alfabeto  $\Sigma$  è una qualsiasi sequenza ordinata di simboli di  $\Sigma$ , eventualmente contenente simboli ripetuti.

### **Lunghezza di una stringa**

La lunghezza di una stringa  $x$  dell'alfabeto  $\Sigma$  è il numero di caratteri (comprese le eventuali ripetizioni) che costituiscono la stringa  $x$  stessa.

### **Stringa vuota**

Una stringa vuota è una stringa con lunghezza nulla. La stringa vuota viene in genere indicata con la lettera greca  $\varepsilon$  (epsilon minuscola).

### **Numero di ripetizioni del carattere $a$ nella stringa $x$**

Data una stringa  $x$  dell'alfabeto  $\Sigma$  e dato il simbolo  $a$  appartenente a  $\Sigma$ , il numero di ripetizioni di  $a$  nella stringa  $x$  è il numero di volte in cui tale carattere appare in  $x$  e si indica  $|x|_a$ .

### **Linguaggio**

Un linguaggio  $L$  definito sull'alfabeto  $\Sigma$  è un insieme di stringhe (eventualmente vuoto) dell'alfabeto  $\Sigma$ .

### **Cardinalità del linguaggio**

La cardinalità del linguaggio  $L$  definito sull'alfabeto  $\Sigma$  è il numero di stringhe che costituiscono  $L$ , e si indica con  $|L|$ .

### **Linguaggio finito o vocabolario**

Un linguaggio  $L$  viene detto vocabolario (o linguaggio finito) se ha cardinalità finita.

### **Linguaggio infinito**

Un linguaggio  $L$  viene detto infinito se ha cardinalità infinita.

### **Linguaggio vuoto**

Il linguaggio vuoto, indicato con  $\emptyset$ , è il linguaggio che non contiene alcuna stringa. La cardinalità del linguaggio vuoto è perciò nulla:  $|\emptyset| = 0$ .

### **Linguaggio universale**

Il linguaggio universale di alfabeto  $\Sigma$  è l'insieme di tutte e sole le stringhe dell'alfabeto  $\Sigma$ .

### **Linguaggio artificiale o formale**

Un linguaggio artificiale o formale è un linguaggio nel quale la forma delle frasi e il loro significato vengono definiti mediante strumenti matematici, in maniera precisa ed algoritmica. In tal modo è possibile costruire una procedura informatica che verifichi la correttezza grammaticale delle frasi e che ne calcoli il significato.

## Sintassi di un linguaggio

La sintassi di un linguaggio è l'insieme delle regole che ci consentono di definire la forma delle frasi del linguaggio stesso.

## Semantica di un linguaggio

La semantica di un linguaggio è l'insieme delle regole che ci consentono di ricavare il significato di una frase del linguaggio stesso.

## Parsificazione

La parsificazione è il processo di analisi sintattica di un linguaggio.

## Operazioni sulle stringhe

Le operazioni fondamentali definite su stringhe  $x$  ed  $y$  di alfabeto  $\Sigma$  sono:

### Concatenamento

#### Definizione

Il concatenamento tra la stringa  $x$  e la stringa  $y$  è la stringa  $xy$  (o  $x.y$ ), costituita da tutti i simboli di  $x$  seguiti da tutti i simboli di  $y$ , negli stessi ordini in cui compaiono nelle stringhe di partenza.

$$x = a_0 \dots a_n$$

$$y = b_0 \dots b_k$$

$$xy = x.y = a_0 \dots a_n b_0 \dots b_k$$

#### Proprietà

1. Il concatenamento è associativo a sinistra e a destra.
2. La cardinalità del concatenamento è la somma delle cardinalità delle stringhe di partenza.
3. La stringa vuota è l'elemento neutro per il concatenamento.

$$x(yz) = (xy)z = xyz$$

$$|xy| = |x| + |y|$$

$$\varepsilon x = x\varepsilon = x$$

### Riflessione

#### Definizione

Il riflesso della stringa  $x$  è la stringa  $x^R$ , costituita da tutti e soli i simboli di  $x$ , nell'ordine inverso:

$$x = a_0 a_1 \dots a_{n-1} a_n$$

$$x^R = a_n a_{n-1} \dots a_1 a_0$$

#### Proprietà

1. Il riflesso del riflesso di  $x$  è  $x$ .
2. Il riflesso della stringa vuota è la stringa vuota.
3. Il riflesso del concatenamento tra  $x$  e  $y$  è il concatenamento tra il riflesso di  $y$  e il riflesso di  $x$ .
4. La cardinalità del riflesso di  $x$  è uguale alla cardinalità di  $x$ .

$$(x^R)^R = x$$

$$\varepsilon^R = \varepsilon$$

$$(xy)^R = y^R x^R$$

$$|x^R| = |x|$$

### Potenza

#### Definizione

La potenza  $m$ -esima di una stringa  $x$ , dove  $m$  è un intero non negativo, è la stringa  $x^m$  ottenuta concatenando  $m$  con sé stessa per  $m$  volte; se  $m$  è zero,  $x^m = x^0 = \varepsilon$ .

$$x^m = \begin{cases} x \cdot x^{m-1} & \text{se } m > 0 \\ \varepsilon & \text{se } m = 0 \end{cases}$$

#### Proprietà

1. Il concatenamento di due potenze di  $x$  è la potenza di  $x$  avente come esponente la somma delle potenze di partenza.
2. Per ogni  $m$ , la potenza  $m$ -esima di  $\varepsilon$  è  $\varepsilon$ .
3. La cardinalità della potenza  $m$ -esima di  $x$  è  $m$  volte la cardinalità di  $x$ .
4. La potenza  $m$ -esima del riflesso di una stringa è il riflesso della potenza  $m$ -esima della stringa stessa.

$$x^n \cdot x^k = x^{n+k}$$

$$\varepsilon^m = \varepsilon$$

$$|x^m| = m \cdot |x|$$

$$(x^R)^m = (x^m)^R$$

Si noti che, come dato per scontato nei precedenti esempi e come di consueto anche nell'algebra, le operazioni unarie hanno sempre la priorità rispetto a quelle binarie (qui solo il concatenamento).

## Sottostringhe

Introduciamo adesso delle definizioni relative alle sottostringhe.

### Sottostringa

Data una stringa  $x$  che può essere scritta come concatenamento tra le stringhe  $u$ ,  $y$  e  $v$  (eventualmente vuote), diciamo che  $u$ ,  $y$  e  $v$  sono sottostringhe di  $x = u y v$ .

Siccome abbiamo ammesso che le stringhe possano essere vuote, sarebbe equivalente modificare la precedente definizione affermando che chiamiamo sottostringa di  $x$  la stringa  $y$  (senza dire che anche  $u$  e  $v$  sono sottostringhe).

### Sottostringa propria

Una sottostringa di  $x$  è propria se non coincide con  $x$ .

### Prefisso

Data la stringa  $x = u y v$ , diciamo che la sottostringa  $u$  di  $x$  è un prefisso di  $x$ .

### Suffisso

Data la stringa  $x = u y v$ , diciamo che la sottostringa  $v$  di  $x$  è un suffisso di  $x$ .

### Inizio di lunghezza $k$

Data la stringa  $x$ , l'inizio di lunghezza  $k$  di  $x$ , indicato con  $Ini_k(x)$  è il prefisso di  $x$  avente lunghezza  $k$  (dove  $k$  è un numero compreso tra 1 e la lunghezza di  $x$ ).

## Operazioni sui linguaggi

### Riflessione

Il linguaggio riflesso del linguaggio  $L$  è il linguaggio  $L^R$  contenente tutte e sole le stringhe di riflesse di tutte e sole le stringhe di  $L$ .

$$L^R = \{x^R | x \in L\}$$

### Concatenamento

Dati i linguaggi  $L_1$  ed  $L_2$ , il concatenamento tra  $L_1$  ed  $L_2$  è il linguaggio  $L_1 L_2$  contenente tutte e sole le stringhe ottenute concatenando una stringa di  $L_1$  e una di  $L_2$ .

$$L_1 L_2 = \{x.y | x \in L_1, y \in L_2\}$$

### Potenza

La potenza  $m$ -esima del linguaggio  $L$ , dove  $m$  è un intero non negativo, è il linguaggio  $L^m$  contenente tutte e sole le stringhe ottenute concatenando tra loro  $m$  stringhe di  $L$  (dove eventualmente la stessa stringa può comparire più volte tra le  $m$  considerate). Formalmente:

$$L^m = \begin{cases} L.L^{m-1} & m > 0 \\ \{\varepsilon\} & m = 0 \end{cases}$$

### Unione

L'unione di due linguaggi  $L_1$  e  $L_2$  è il linguaggio  $L_1 \cup L_2$  costituito da tutte e sole le stringhe che appartengono ad almeno uno dei linguaggi di partenza  $L_1$  ed  $L_2$ .

$$L_1 \cup L_2 = \{x | x \in L_1 \vee x \in L_2\}$$

### Intersezione

L'intersezione tra i linguaggi  $L_1$  e  $L_2$  è il linguaggio  $L_1 \cap L_2$  costituito da tutte e sole le stringhe che appartengono ad entrambi i linguaggi di partenza  $L_1$  ed  $L_2$ .

$$L_1 \cap L_2 = \{x | x \in L_1 \wedge x \in L_2\}$$

### Differenza

La differenza tra il linguaggio  $L_1$  e il linguaggio  $L_2$  è il linguaggio  $L_1 \setminus L_2$ , talvolta indicato anche con  $L_1 - L_2$ , costituito da tutte e sole le stringhe che appartengono  $L_1$  ma non a  $L_2$ .

$$L_1 \setminus L_2 = L_1 - L_2 = \{x | x \in L_1 \wedge x \notin L_2\}$$

### Complemento

Il complemento di un linguaggio  $L$  di alfabeto  $\Sigma$  è il linguaggio  $\neg L$ , costituito da tutte e sole le stringhe definite sull'alfabeto  $\Sigma$  ma non appartenenti a  $L$ . Perciò:

$$\neg L = L_{\text{universale}} \setminus L$$

### Stella di Kleene

La stella di Kleene di un linguaggio  $L$  è la chiusura riflessiva e transitiva del linguaggio stesso rispetto all'operatore di concatenamento, ovvero il linguaggio  $L^*$  contenente tutte e sole le stringhe ottenute concatenando tra loro  $m$  stringhe di  $L$ , per ogni  $m$  (dette *sequenze finite* di elementi di  $L$ ),

$$L^* = \bigcup_{h=0}^{+\infty} L^h$$

### Croce

La croce di un linguaggio  $L$  è la chiusura transitiva (ma non riflessiva)  $L^+$  del linguaggio stesso rispetto all'operatore di concatenamento.

$$L^+ = \bigcup_{h=1}^{+\infty} L^h$$

### Quoziente destro

Dati i linguaggi  $L_1$  e  $L_2$ , il quoziente destro  $L_1 /_D L_2$  è il linguaggio costituito da tutte e sole le stringhe ottenute togliendo da ogni stringa di  $L_1$  i relativi prefissi che coincidano con stringhe di  $L_2$ .

$$L_1 /_D L_2 = \{y | xy \in L_1 \wedge x \in L_2\}$$

### Quoziente sinistro

Dati i linguaggi  $L_1$  e  $L_2$ , il quoziente sinistro  $L_1 /_S L_2$  è il linguaggio costituito da tutte e sole le stringhe ottenute togliendo da ogni stringa di  $L_1$  i relativi suffissi che coincidano con stringhe di  $L_2$ .

$$L_1 /_S L_2 = \{y | yx \in L_1 \wedge x \in L_2\}$$

### Insieme dei prefissi propri

L'insieme dei prefissi propri di un linguaggio  $L$  è il linguaggio costituito da tutti i prefissi di stringhe appartenenti ad  $L$ .

$$\text{Prefissi}(L) = \{x | xy \in L \wedge y \neq \varepsilon\}$$

### Traslitterazione o omomorfismo alfabetico

È l'operazione che consiste nel passare da un linguaggio  $L_1$  di alfabeto  $\Sigma_1$  ad un linguaggio  $L_2$  di alfabeto  $\Sigma_2$  semplicemente per mezzo di un omomorfismo che associ ad ogni simbolo di  $\Sigma_1$  uno ed un solo simbolo di  $\Sigma_2$  (oppure la stringa vuota):

$$h: \Sigma_1 \rightarrow (\Sigma_2 \cup \{\varepsilon\})$$

Nel caso particolare in cui la traslitterazione sia del tipo:

$$h: \Sigma_1 \rightarrow (\Sigma_2 \cup \{\varepsilon\})$$

Si parla di *traslitterazione priva di cancellazioni*.

### Traslitterazione a parole

È l'operazione che consiste nel passare da un linguaggio  $L_1$  di alfabeto  $\Sigma_1$  ad un linguaggio  $L_2$  di alfabeto  $\Sigma_2$  semplicemente per mezzo di un omomorfismo che associ ad ogni simbolo di  $\Sigma_1$  una stringa di  $\Sigma_2$  (di lunghezza qualsiasi, quindi anche nulla o unitaria):

$$h: \Sigma_1 \rightarrow \Sigma_2^*$$

### Sostituzione

È l'operazione che consiste nel passare da un linguaggio  $L_1$  ad un linguaggio  $L_2$ , sostituendo ad ognuno dei caratteri  $a$  dell'alfabeto  $\Sigma_1$  di  $L_1$  una qualsiasi stringa di un opportuno linguaggio, definito su un alfabeto  $\Sigma_2$ . Quindi, la traslitterazione a parole è una particolare sostituzione nella quale il linguaggio immagine contiene una sola stringa.



## Proprietà degli operatori tra linguaggi

1. Il complemento del complemento di  $L$  è  $L$ .

$$\neg(\neg L) = L$$

2. Il complemento del linguaggio universale è il linguaggio vuoto.

$$\neg L_{\text{universale}} = \emptyset$$

3. La stella del linguaggio delle stringhe di lunghezza 1 di alfabeto  $\Sigma$  è il linguaggio universale su  $\Sigma$ .

$$\Sigma^* = L_{\text{universale}}$$

4.  $L$  è sempre incluso in  $L^+$ , che è incluso in  $L^*$ .

$$L \subseteq L^+ \subseteq L^*$$

5. La stella di Kleene del riflesso di  $L$  è il riflesso della stella di Kleene di  $L$ .

$$(L^R)^* = (L^*)^R$$

6. La stella di Kleene è idempotente.

$$(L^*)^* = L^*$$

7. Se 2 stringhe appartengono a  $L^*$ , il loro concatenamento appartiene a  $L^*$

$$(x \in L^* \text{ e } y \in L^*) \rightarrow xy \in L^*$$

8. La stringa vuota appartiene a  $L^+$  se e solo se appartiene a  $L$ .

$$\varepsilon \in L^+ \Leftrightarrow \varepsilon \in L$$

9.  $L^+$  è il concatenamento tra  $L$  e  $L^*$  (in qualsiasi ordine).

$$L^+ = LL^* = L^*L$$

10. Il linguaggio delle stringhe di lunghezza  $m$  sull'alfabeto  $\Sigma$  è  $\Sigma^m$ .

11. Il linguaggio delle stringhe di lunghezza diversa da  $m$  sull'alfabeto  $\Sigma$  è  $\neg(\Sigma^m)$ .

12. Se nessun prefisso proprio di  $L$  appartiene ad  $L$ ,  $L$  è privo di prefissi

$$\text{Pref}(L_{pp}) \cap L_{pp} = \emptyset$$

In realtà è stata usata una notazione impropria:  $\Sigma$  viene usato per indicare non un alfabeto, ma il linguaggio avente come stringhe tutte e sole le stringhe di lunghezza 1, contenenti un solo simbolo dell'alfabeto  $\Sigma$ .

## Inizi, fini e digrammi

### Insieme degli inizi

Dato un certo linguaggio  $L$  di alfabeto  $\Sigma$ , chiamiamo insieme degli inizi l'insieme di tutti i simboli appartenenti ad  $\Sigma$  che possono comparire come simbolo iniziale di una stringa appartenente ad  $L$ :

$$\text{Ini}(L) = \{a \in \Sigma \mid a\Sigma^* \cap L \neq \emptyset\}$$

### Insieme delle fini

Dato un certo linguaggio  $L$  di alfabeto  $\Sigma$ , chiamiamo insieme delle fini l'insieme di tutti i simboli appartenenti ad  $\Sigma$  che possono comparire come simbolo finale di una stringa appartenente ad  $L$ :

$$\text{Fin}(L) = \{a \in \Sigma \mid \Sigma^*a \cap L \neq \emptyset\}$$

### Insieme dei digrammi

Dato un certo linguaggio  $L$  di alfabeto  $\Sigma$ , chiamiamo insieme dei digrammi l'insieme di tutte le coppie di simboli appartenenti ad  $\Sigma$  che possono comparire consecutivamente all'interno di una stringa appartenente ad  $L$ :

$$\text{Dig}(L) = \{ab \mid a, b \in \Sigma, \Sigma^*ab\Sigma^* \cap L \neq \emptyset\}$$

# Espressioni regolari

## Introduzione intuitiva

Una espressione regolare può essere intuitivamente vista come una stringa, costituita dai simboli terminali di un certo alfabeto  $\Sigma$ , con l'aggiunta dei simboli corrispondenti agli operatori definiti sui linguaggi. L'espressione regolare consente di *denotare* un linguaggio, cioè è una sintassi che permette di descrivere un insieme di stringhe.

Intuitivamente, se ad esempio  $\Sigma = \{a, b\}$ , l'espressione regolare  $ab$  denota il linguaggio  $L_1 = \{ab\}$  contenente solo la stringa  $ab$ , mentre l'espressione regolare  $(ab)^*$  denota il linguaggio  $L_2 = \{ab\}^*$ .

Occorre però formalizzare il concetto di espressione regolare e quello di linguaggio derivato da un'espressione regolare.

## Terminologia di base

Vediamo alcune definizioni preliminari.

### Espressione regolare (ER)

Una espressione regolare (ER) è una stringa definita su un alfabeto contenente tutti i simboli di un alfabeto terminale  $\Sigma$ , con l'aggiunta dei metasimboli di unione, concatenamento, stella, insieme vuoto e delle parentesi tonde aperte e chiuse, che rispetti le seguenti regole ricorsive:

1. La stringa  $r = \emptyset$  è un'espressione regolare, che corrisponde al linguaggio vuoto.
2. La stringa  $r = a$ , dove  $a \in \Sigma$ , è un'espressione regolare.
3. La stringa  $r = (s \cup t)$  o, equivalentemente,  $r = s|t$ , dove  $s$  e  $t$  sono ER, è una ER.
4. La stringa  $r = (s.t)$  o, equivalentemente,  $r = st$ , dove  $s$  e  $t$  sono ER, è una ER.
5. La stringa  $r = s^*$ , dove  $s$  è un'espressione regolare, è a sua volta un'espressione regolare.

In genere sono ammessi nelle espressioni regolari anche i simboli  $\varepsilon$  (stringa vuota) e croce, che però non aggiungono potere espressivo alle espressioni regolari, in quanto:

$$\varepsilon = \emptyset^*$$

$$s^+ = s.s^*$$

### Interpretazione dell'espressione regolare

Da un punto di vista intuitivo, l'espressione regolare consente di denotare un linguaggio: gli operatori utilizzati rappresentano infatti i consueti operatori tra linguaggi. Inoltre, è intuitivo interpretare un'espressione regolare costituita solo da simboli dell'alfabeto  $\Sigma$  come il linguaggio contenente solo la stringa indicata. Ad esempio, l'ER  $aba$  corrisponde al linguaggio  $L = \{aba\}$ . A breve formalizzeremo tale concetto in maniera matematicamente più rigorosa.

### Espressione regolare estesa

Una espressione regolare estesa è un'espressione regolare nella quale, oltre agli operatori ammessi in ogni espressione regolare, si ammettono anche altri operatori derivati, che non aggiungono potenza espressiva ma consentono di scrivere l'espressione regolare in maniera più concisa. Gli operatori aggiuntivi sono:

- |   |                               |
|---|-------------------------------|
| 1. L'operatore di elevamento a potenza.                       | $s^n, n \geq 2$               |
| 2. L'operatore di ripetizione minimo $k$ , massimo $n$ volte. | $[s]_k^n, n > k$              |
| 3. L'operatore di opzionalità.                                | $[s]$                         |
| 4. L'operatore di intervallo ordinato.                        | Es.: $(A..Z), (0..9), (a..z)$ |
| 5. L'operatore di intersezione insiemistica.                  | $s \cap t$                    |
| 6. L'operatore di complemento insiemistico.                   | $\neg s$                      |
| 7. L'operatore di differenza insiemistica.                    | $s \setminus t$               |

### Sottoespressione (SE)

Una sottoespressione (SE) di un'espressione regolare  $e$  completamente parentizzata è una qualsiasi sottostringa ben parentizzata di  $e$ , contenuta direttamente entro le parentesi più esterne.

Se l'espressione regolare di partenza non è completamente parentizzata, le sottoespressioni di  $e$  sono semplicemente le SE della stringa ottenuta aggiungendo tutte le parentesi sottintese ad  $e$ .

Se l'espressione regolare contiene operatori associativi (come l'unione), le SE dell'ER di partenza sono tutte le SE di tutte le ER che si ottengono parentizzando  $e$  in tutti i modi possibili.

### Scelta

La definizione di scelta può essere data per casi:

1. Se è data l'espressione regolare  $e = e_1 | e_2 | \dots | e_k$ , allora  $e_1, e_2, \dots, e_k$  sono scelte di  $e$ .
2. Se è data l'espressione regolare  $e = e_1^*$ , allora  $\varepsilon$  ed  $e^n$  (per ogni  $n \geq 1$  intero) sono scelte di  $e$ .
3. Se è data l'espressione regolare  $e = e_1^+$ , allora le ER  $e^n$  (per ogni  $n \geq 1$  intero) sono scelte di  $e$ .

## Derivazione

### Relazione di derivazione

Data un'espressione regolare  $e$ , diciamo che  $e'$  è derivata da  $e$  ( $e \Rightarrow e'$ ) se è ottenuta da  $e$  sostituendo ad una sua sottoespressione  $\beta$  una scelta  $\delta$  di  $\beta$ .

$$e = \alpha\beta\gamma \qquad e' = \alpha\delta\gamma \qquad e \Rightarrow e'$$

Le scelte devono essere sempre fatte dall'esterno verso l'interno.

### Relazione di derivazione in zero o più passi

La relazione di derivazione in uno o più passi ( $\Rightarrow^*$ ) tra espressioni regolari è la chiusura riflessiva e transitiva della relazione di derivazione ( $\Rightarrow$ ). Quindi, data un'espressione regolare  $e$ , diciamo che  $e'$  è derivata da  $e$  in uno o più passi se  $e \Rightarrow e_1 \Rightarrow e_2 \Rightarrow \dots \Rightarrow e'$ .

### Derivazione in $n$ passi

Data un'espressione regolare  $e$ , l'espressione regolare  $e_n$  deriva da  $e$  in  $n$  passi ( $\Rightarrow^n$ ) se:

$$e \Rightarrow e_1 \Rightarrow e_2 \Rightarrow \dots \Rightarrow e_n$$

### Derivazione in più passi

Data un'espressione regolare  $e$ , l'espressione regolare  $e'$  deriva da  $e$  in più passi ( $\Rightarrow^+$ ) se esiste una sequenza di derivazioni con almeno un passo che porta da  $e$  a  $e'$ . Si tratta perciò della chiusura transitiva (ma non riflessiva) della relazione di derivazione ( $\Rightarrow$ ).

### Stringa terminale

Una stringa terminale è un'espressione regolare  $e'$  derivata da un'espressione regolare  $e$ , tale che  $e'$  sia costituita solo da simboli dell'alfabeto terminale  $\Sigma$ .

### Linguaggio definito da un'espressione regolare

Il linguaggio  $L(e)$  definito dall'espressione regolare  $e$  di alfabeto  $\Sigma$  è l'insieme di tutte e sole le stringhe terminali che si possono derivare a partire da  $e$  in zero o più passi.

$$L(e) = \{x \in \Sigma^* | e \Rightarrow^* x\}$$

### Espressioni regolari equivalenti

Due espressioni regolari sono equivalenti se definiscono lo stesso linguaggio.

## Linguaggi regolari

### Linguaggio regolare

Un linguaggio regolare  $L$  sull'alfabeto  $\Sigma = \{a_1, a_2, \dots, a_n\}$  è detto regolare se è denotato da un'espressione regolare, ovvero può essere espresso mediante le operazioni di concatenamento, unione e stella applicate a un numero finito di linguaggi unitari  $\{a_1\}, \{a_2\}, \dots, \{a_n\}$ .

### Famiglia dei linguaggi regolari (REG)

La famiglia dei linguaggi regolari (REG) è l'insieme di tutti e soli i linguaggi regolari, indipendentemente dal linguaggio sul quale i singoli linguaggi sono definiti.

### Famiglia dei linguaggi finiti (FIN)

Come abbiamo già affermato, un linguaggio finito è un linguaggio avente cardinalità finita. La famiglia dei linguaggi finiti è l'insieme di tutti e soli i linguaggi finiti e viene indicata con FIN.

Ogni linguaggio finito è anche regolare:

$$FIN \subseteq REG$$

### Famiglia dei linguaggi locali (LOC)

Un linguaggio locale è un linguaggio interamente caratterizzato dai tre insiemi  $Ini(L)$ ,  $Fin(L)$  e  $Dig(L)$ , ovvero è un linguaggio del tipo:

$$L = \{x \in \Sigma^* \mid Ini(x) \in Ini(L), Fin(x) \in Fin(L), Dig(x) \subseteq Dig(L)\}$$

In altri termini,  $L$  è l'insieme di tutte e sole le stringhe che iniziano per un simbolo di  $Ini(L)$ , finiscono con un simbolo di  $Fin(L)$  e nelle quali tutte le coppie di caratteri consecutivi appartengono a  $Dig(L)$ .

Equivalentemente, se indichiamo con  $\overline{Dig}(L)$  il complemento di  $Dig(L)$ , possiamo definirli come:

$$L = \{x \in \Sigma^* \mid Ini(x) \in Ini(L), Fin(x) \in Fin(L)\} \setminus (\Sigma^* \overline{Dig}(L) \Sigma^*)$$

I linguaggi locali sono un sottoinsieme proprio dei linguaggi regolari:

$$LOC \subset REG$$

## Ambiguità delle espressioni regolari

### Espressione regolare marcata

#### Definizione

Data un'espressione regolare  $e$ , chiamiamo espressione regolare marcata associata ad  $e$  l'ER  $e'$  ottenuta numerando con un pedice tutti i terminali presenti all'interno di  $e$ .

#### Esempio

$$e = (a|b)^* abba(a|b)^*$$

$$e' = (a_1|b_2)^* a_3 b_4 b_5 a_6 (a_7|b_8)^*$$

### Espressione regolare ambigua

#### Definizione

Un'espressione regolare è ambigua se e solo se, nel linguaggio definito dalla corrispondente espressione regolare marcata, esiste almeno una coppia di stringhe che differiscono tra loro solo per i pedici associati ai terminali (ovvero, cancellando i pedici le due stringhe sono uguali).

#### Esempio

L'espressione regolare:

$$e = (a|b)^* a(a|b)^*$$

È ambigua, perché la corrispondente espressione marcata:

$$e' = (a_1|b_2)^* a_3 (a_4|b_5)^*$$

Definisce un linguaggio al quale appartengono le stringhe:

$$a_1 a_3 a_4$$

$$a_1 a_1 a_3$$

Che, tolti i pedici, equivalgono entrambe alla stringa  $aaa$ .

#### Significato

Se una grammatica è ambigua, significa che la stessa stringa terminale può essere ottenuta con più derivazioni che differiscono strutturalmente tra loro, e non solo per l'ordine in cui vengono effettuate le scelte. Questo si traduce concretamente nell'impossibilità di interpretare in maniera univoca il significato (la semantica) della stringa ottenuta, che viene detta ambigua.

## Astrazione linguistica: liste

### Astrazione linguistica

L'astrazione linguistica è un processo che consente di sostituire ai terminali dell'alfabeto del linguaggio di partenza (detto *linguaggio sorgente*) le stringhe di un altro linguaggio, detto *linguaggio pozzo*.

In tal modo si rappresenta un'espressione regolare semplice, che denota il linguaggio sorgente, ma in realtà ad ogni simbolo di tale linguaggio è associato un linguaggio pozzo, e si rappresenta quindi in questo modo il linguaggio che contiene tutte le stringhe ottenute a partire da una stringa del linguaggio sorgente, sostituendo ad ogni suo terminale una stringa del corrispondente linguaggio pozzo.

### Uso dell'astrazione linguistica per rappresentare una lista con separatori

Una lista con separatori può essere rappresentata da una delle seguenti espressioni regolari:

$$e(se)^* \qquad (es)^*e$$

Dove  $e$  è il terminale al quale è associato come linguaggio pozzo il linguaggio degli elementi della lista, mentre ad  $s$  è associato il linguaggio pozzo dei separatori.

### Uso dell'astrazione linguistica per rappresentare una lista con separatori e marche

Una lista con separatori e marche di inizio e fine può essere rappresentata da una delle seguenti espressioni regolari:

$$ie(se)^*f \qquad i(es)^*ef$$

Dove  $e$  è il terminale al quale è associato come linguaggio pozzo il linguaggio degli elementi della lista, ad  $s$  è associato il linguaggio pozzo dei separatori, mentre ad  $i$  e  $f$  sono associati i linguaggi pozzo dei marcatori di inizio e di fine della lista.

### Uso dell'astrazione linguistica per rappresentare una lista a più livelli, con separatori e marche

Similmente a quanto finora fatto, possiamo rappresentare una lista a più livelli con separatori e marche mediante più espressioni regolari, come di seguito mostrato:

$$\begin{aligned} lista_1 &= i_1 lista_2 (s_1 lista_2)^* f_1 \\ lista_2 &= i_2 lista_3 (s_2 lista_3)^* f_2 \\ &\dots \\ lista_k &= i_k e (s_k e)^* f_k \end{aligned}$$

# Grammatiche libere

## Le grammatiche generative

### Grammatica generativa

Una grammatica  $G$  è una tupla del tipo:

$$G = \langle V, \Sigma, P, S \rangle$$

Dove:

- $V$  è l'alfabeto dei simboli non terminali;
- $\Sigma$  è l'alfabeto dei simboli terminali;
- $P$  è l'insieme delle regole di riscrittura (o produzioni della grammatica);
- $S$  è un particolare simbolo di  $V$ , detto simbolo iniziale o assioma della grammatica.

### Produzioni

Detto  $W = V \cup \Sigma$ , l'insieme delle produzioni  $P$  è un insieme del tipo:

$$P \subseteq V^+ \times W^*$$

Ogni produzione è perciò una coppia, il cui primo elemento è una sequenza non vuota di simboli non terminali della grammatica, e il cui secondo elemento è una stringa qualsiasi di terminali e non terminali della grammatica stessa.

Una produzione  $(A, bA)$  viene normalmente indicata mediante la notazione:

$$A \rightarrow bA$$

Il primo elemento della coppia viene detto *parte sinistra* e il secondo viene chiamato *parte destra*. Se si hanno produzioni con la stessa parte sinistra  $P_s$  e con diverse parti destre  $P_{d_i}$ , si scrive:

$$P_s \rightarrow P_{d_1} | P_{d_2} | \dots | P_{d_{n-1}} | P_{d_n}$$

## Derivazione

### Derivazione immediata

Data la grammatica  $G = \langle V, \Sigma, P, S \rangle$  e date le stringhe  $\beta, \gamma \in (V \cup \Sigma)^*$ , diciamo che la stringa  $\gamma$  deriva immediatamente da  $\beta$  per la grammatica  $G$  e scriviamo  $\beta \Rightarrow_G \gamma$  (oppure  $\beta \Rightarrow \gamma$ ) se si valgono tutte le condizioni di seguito elencate:

1. La stringa  $\beta$  è del tipo:  $\beta = \delta_1 A \delta_2$
2. La stringa  $\gamma$  è del tipo:  $\gamma = \delta_1 \alpha \delta_2$
3. Esiste in  $P$  la produzione  $A \rightarrow \alpha$ , ovvero:  $(A, \alpha) \in P$

### Derivazione in $n$ passi

Data la grammatica  $G = \langle V, \Sigma, P, S \rangle$  e date le stringhe  $\beta, \gamma \in (V \cup \Sigma)^*$ , diciamo che la stringa  $\gamma$  deriva da  $\beta$  in  $n$  passi per la grammatica  $G$  e scriviamo  $\beta \Rightarrow^n \gamma$  se esiste una sequenza di  $n$  derivazioni immediate che consente di ottenere la stringa  $\gamma$  a partire dalla stringa  $\beta$ .

### Derivazione in zero o più passi

Data la grammatica  $G = \langle V, \Sigma, P, S \rangle$  e date le stringhe  $\beta, \gamma \in (V \cup \Sigma)^*$ , diciamo che la stringa  $\gamma$  deriva da  $\beta$  in zero o più passi per la grammatica  $G$  e scriviamo  $\beta \Rightarrow^* \gamma$  se esiste una sequenza di zero o più derivazioni immediate che consente di ottenere la stringa  $\gamma$  a partire dalla stringa  $\beta$ . In sostanza quindi la relazione indicata con il simbolo  $\Rightarrow^*$  non è altro che la chiusura transitiva e riflessiva della relazione di derivazione.

### Derivazione in più passi

Data la grammatica  $G = \langle V, \Sigma, P, S \rangle$  e date le stringhe  $\beta, \gamma \in (V \cup \Sigma)^*$ , diciamo che la stringa  $\gamma$  deriva da  $\beta$  in più passi per la grammatica  $G$  e scriviamo  $\beta \Rightarrow^+ \gamma$  se esiste una sequenza di una o più derivazioni immediate che consente di ottenere la stringa  $\gamma$  a partire dalla stringa  $\beta$ . In sostanza quindi la relazione indicata con il simbolo  $\Rightarrow^+$  è la chiusura transitiva della relazione di derivazione.

## Linguaggio generato

### **Forma generata dalla grammatica**

Una stringa viene detta *forma generata* dalla grammatica  $G$  se essa deriva da un non terminale  $A$  di  $G$ .

### **Forma di frase della grammatica**

Una stringa viene detta *forma di frase* dalla grammatica  $G$  se essa deriva dall'assioma  $S$  di  $G$ .

### **Frase della grammatica**

Una stringa viene detta *frase* dalla grammatica  $G$  se essa è una forma di frase costituita solamente da simboli terminali del linguaggio.

### **Linguaggio generato dalla grammatica partendo da un non terminale $A$**

Data una grammatica  $G$  e dato un suo simbolo non terminale  $A$ , chiamiamo *linguaggio generato da  $G$  partendo da  $A$*  il linguaggio  $L_A(G)$  costituito da tutte e sole le stringhe terminali che derivano da  $A$  in uno o più passi.

### **Linguaggio generato dalla grammatica**

Data una grammatica  $G$ , chiamiamo *linguaggio generato da  $G$*  il linguaggio  $L(G)$  generato da  $G$  partendo dall'assioma  $S$ . In altri termini, il linguaggio generato dalla grammatica è l'insieme di tutte le frasi della grammatica.

### **Grammatiche equivalenti**

Due grammatiche  $G_1$  e  $G_2$  si dicono equivalenti se generano lo stesso linguaggio, ovvero:

$$L(G_1) = L(G_2)$$

## Grammatiche libere e linguaggi liberi

### **Definizione di grammatica libera**

Una grammatica (o sintassi) libera dal contesto  $G$ , detta anche semplicemente grammatica libera, non contestuale, del tipo 2 oppure BNF (da Backus Normal Form o Backus Naur Form) è una grammatica  $G$  del tipo:

$$G = \langle V, \Sigma, P, S \rangle$$

Dove ogni regola appartenente a  $P$  ha come parte sinistra un solo simbolo non terminale di  $G$ :

$$\forall (X, \alpha) \in P, \quad X \in V, \alpha \in (V \cup \Sigma)^*$$

### **Linguaggio libero (o libero dal contesto)**

Un linguaggio  $L$  viene detto *libero dal contesto* (o solamente *libero*) se esiste almeno una grammatica libera che generi  $L$ .

### **Famiglia dei linguaggi liberi**

La famiglia dei linguaggi liberi, indicata con  $LIB$ , è l'insieme di tutti e soli i linguaggi liberi. Tutti i linguaggi regolari sono anche liberi, perciò possiamo scrivere:

$$REG \subseteq LIB$$

## Classificazione delle regole

Data una regola  $A \rightarrow \alpha$ , dove  $A \in V$  e  $\alpha \in (V \cup \Sigma)^*$ , possiamo classificare la produzione stessa secondo le seguenti definizioni:

### Regola terminale

La produzione viene detta *regola terminale* se la sua parte destra contiene solamente simboli terminali della grammatica, ovvero:

$$\alpha \in \Sigma^*$$

### Regola vuota

La produzione viene detta *regola vuota* se la sua parte destra coincide con la stringa vuota:

$$\alpha = \varepsilon$$

### Regola iniziale

La produzione viene detta *regola iniziale* se la sua parte sinistra è l'assioma:

$$A = S$$

### Regola ricorsiva

La produzione viene detta *regola ricorsiva* se la sua parte sinistra compare anche nella parte destra:

$$\alpha = \delta_1 A \delta_2, \quad \delta_1, \delta_2 \in (V \cup \Sigma)^*$$

### Regola ricorsiva a sinistra

La produzione viene detta *regola ricorsiva a sinistra* se la sua parte sinistra compare come prefisso della parte destra:

$$\alpha = A \delta_1, \quad \delta_1 \in (V \cup \Sigma)^*$$

### Regola ricorsiva a destra

La produzione viene detta *regola ricorsiva a destra* se la sua parte sinistra compare come suffisso della parte destra:

$$\alpha = \delta_1 A, \quad \delta_1 \in (V \cup \Sigma)^*$$

### Regola ricorsiva a destra e a sinistra

La produzione viene detta *regola ricorsiva a destra e a sinistra* se la sua parte sinistra compare sia come prefisso che come suffisso della parte destra:

$$\alpha = A \delta_1 A, \quad \delta_1 \in (V \cup \Sigma)^*$$

### Regola di copiatura (o categorizzazione)

La produzione viene detta *regola di copiatura* se la sua parte destra è costituita da un solo simbolo non terminale della grammatica:

$$\alpha \in V$$

### Regola lineare

La produzione viene detta *regola lineare* se la sua parte destra non contiene più di un non terminale:

$$\alpha = \delta_1 B \delta_2 \text{ con } \delta_1, \delta_2 \in \Sigma^* \text{ e } B \in V \quad \text{oppure} \quad \alpha \in \Sigma^*$$

### Regola lineare a destra

La produzione viene detta *regola lineare a destra* se è lineare e l'unico eventuale non terminale appare come suffisso della parte destra della regola stessa.

$$\alpha = \delta_1 B \text{ con } \delta_1 \in \Sigma^* \text{ e } B \in V \quad \text{oppure} \quad \alpha \in \Sigma^*$$

### Regola lineare a sinistra

La produzione viene detta *regola lineare a sinistra* se è lineare e l'unico eventuale non terminale appare come prefisso della parte destra della regola stessa.

$$\alpha = B \delta_1 \text{ con } \delta_1 \in \Sigma^* \text{ e } B \in V \quad \text{oppure} \quad \alpha \in \Sigma^*$$



### **Regola normale di Chomsky**

La produzione viene detta *regola normale di Chomsky* se la sua parte destra è costituita da un solo simbolo terminale, oppure da due simboli non terminali:

$$\alpha \in V^2 \quad \text{oppure} \quad \alpha \in \Sigma$$

### **Regola normale di Greibach**

La produzione viene detta *regola normale di Greibach* se la sua parte destra è costituita da un solo simbolo terminale, eventualmente seguito da uno o più non terminali.

$$\alpha = x\delta, \quad x \in \Sigma, \delta \in V^*$$

### **Regola a operatori**

La produzione viene detta *regola a operatori* se la sua parte destra è costituita da due non terminali, separati da un simbolo terminale (operatore):

$$\alpha = BxC, \quad x \in \Sigma, B, C \in V$$

## **Grammatica lineare**

### **Grammatica lineare a destra**

Una grammatica lineare destra è una grammatica nella quale l'insieme delle produzioni contiene solamente regole lineari a destra.

### **Grammatica lineare a sinistra**

Una grammatica lineare sinistra è una grammatica nella quale l'insieme delle produzioni contiene solamente regole lineari a sinistra.

### **Linguaggi generati da grammatiche lineari**

Le grammatiche lineari a destra e le grammatiche lineari a sinistra hanno la stessa potenza espressiva: entrambe infatti sono equivalenti alle espressioni regolari. Ciò significa che un linguaggio è regolare se e solo se esiste almeno una grammatica lineare a destra (o, equivalentemente, a sinistra) che lo generi.

## **Eliminazione dei non terminali non raggiungibili**

### **Non terminale raggiungibile**

Data una grammatica  $G = \langle V, \Sigma, P, S \rangle$ , diciamo che il non terminale  $A \in V$  è raggiungibile se esiste da una derivazione da  $S$  in un numero qualsiasi di passi, nella quale compaia il non terminale dato  $A$ .

### **Non terminali non raggiungibili**

Naturalmente, un non terminale non raggiungibile è un non terminale che non soddisfa la definizione di non terminale raggiungibile. I non terminali raggiungibili sono del tutto inutili, e perciò vanno eliminati dalla grammatica.

### **Procedimento per l'eliminazione dei non terminali non raggiungibili**

Per eliminare i non terminali non raggiungibili:

1. Si costruisce il grafo della relazione binaria *produce*, definita sui non terminali di  $V$  in modo che:  

$$A \text{ produce } B \Leftrightarrow \exists (A \rightarrow \alpha B \beta) \in P, \alpha, \beta \in (V \cup \Sigma)^*$$
2. Si osserva poi il grafo così ottenuto: il non terminale  $A$  è non raggiungibile se e solo se non esiste alcun cammino da  $S$  ad  $A$  nel grafo della relazione *produce*.
3. Si eliminano da  $V$  tutti i non terminali non raggiungibili e si cancellano da  $P$  tutte le regole nelle quali compare almeno uno dei non terminali non raggiungibili.

## Eliminazione dei non terminali non ben definiti

### **Non terminale ben definito**

Data una grammatica  $G = \langle V, \Sigma, P, S \rangle$ , diciamo che il non terminale  $A \in V$  è ben definito se il linguaggio generato da  $G$  partendo da  $A$  non è vuoto, ovvero:

$$L_A(G) \neq \emptyset$$

### **Non terminale non ben definito**

Ovviamente, un non terminale non ben definito è un non terminale tale che il linguaggio generato dalla grammatica partendo da esso è vuoto.

I non terminali non ben definiti sono del tutto inutili, perciò è opportuno eliminarli dalla grammatica.

### **Eliminazione dei non terminali non ben definiti**

Per eliminare i non terminali non ben definiti:

1. Si individuano tutti i non terminali ben definiti:
  - 1.1. Si inizializza un insieme  $DEF$  in modo che contenga tutti i non terminali che compaiono nella parte sinistra di almeno una regola terminale.
  - 1.2. Si aggiungono a  $DEF$  tutti i non terminali che sono la parte sinistra di una regola nella cui parte destra tutti i non terminali siano già appartenenti a  $DEF$ , e si itera tale procedimento fino al raggiungimento di un punto fisso.
2. Si determina l'insieme dei non terminali non ben definiti, semplicemente come differenza insiemistica tra  $V$  e  $DEF$ :

$$NDEF = V / DEF$$

3. Si eliminando dalla grammatica  $V$  tutti i non terminali appartenenti a  $NDEF$  e da  $P$  tutte le regole nelle quali appare almeno un non terminale appartenente a  $NDEF$ .

## Alberi sintattici

### **Albero sintattico**

Data una grammatica  $G = \langle V, \Sigma, P, S \rangle$  e data una stringa appartenente a  $L(G)$ , possiamo costruire l'*albero sintattico* (o gli alberi sintattici) della stringa stessa; in particolare, l'albero sintattico è così definito:

1. La radice dell'albero è l'assioma  $S$ .
2. La frontiera dell'albero, ovvero la sequenza delle foglie lette ordinatamente da sinistra a destra, è la stringa della quale si sta costruendo l'albero.
3. Per costruire l'albero, si considerano tutte le regole che sono state usate per derivare la stringa data. Se si usa la regola  $A_0 \rightarrow A_1 \dots A_n$ , allora il nodo  $A_0$  avrà nell'albero i figli  $A_1, \dots, A_n$ , dove  $A_i$  può essere un terminale, un non terminale o la stringa vuota.

### **Albero sintattico scheletrico**

L'*albero sintattico scheletrico* viene ottenuto mediante l'eliminazione dall'albero sintattico di tutti i simboli non terminali: i nodi interni saranno perciò privi di etichetta.

### **Albero sintattico condensato**

L'*albero sintattico condensato* viene ottenuto a partire dall'albero sintattico scheletrico, fondendo tra loro tutti i nodi interni che sono su un cammino privo di biforcazioni.

### **Albero sintattico rappresentato mediante espressioni parentetiche**

Un albero sintattico è anche rappresentabile per mezzo di espressioni parentetiche, nelle quali

## Derivazioni canoniche

### Derivazione canonica destra

Data una grammatica  $G = \langle V, \Sigma, P, S \rangle$ , chiamiamo *derivazione canonica destra* di una stringa  $\beta_p$  da una stringa  $\beta_0$  una sequenza di derivazioni immediate nella quale ogni stringa intermedia viene ottenuta dalla precedente applicando la regola di derivazione al simbolo non terminale più a destra:

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \dots \Rightarrow \beta_p$$

Dove, per ogni  $i$ :

$$\beta_i = \eta_i A_i \delta_i \quad \beta_{i+1} = \eta_i \alpha_i \delta_i \quad \delta_i \in \Sigma^* \quad (A_i \rightarrow \alpha_i) \in P$$

### Derivazione canonica sinistra

Data una grammatica  $G = \langle V, \Sigma, P, S \rangle$ , chiamiamo *derivazione canonica sinistra* di una stringa  $\beta_p$  da una stringa  $\beta_0$  una sequenza di derivazioni immediate nella quale ogni stringa intermedia viene ottenuta dalla precedente applicando la regola di derivazione al simbolo non terminale più a sinistra:

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \dots \Rightarrow \beta_p$$

Dove, per ogni  $i$ :

$$\beta_i = \eta_i A_i \delta_i \quad \beta_{i+1} = \eta_i \alpha_i \delta_i \quad \eta_i \in \Sigma^* \quad (A_i \rightarrow \alpha_i) \in P$$

### Teorema

1. Ogni frase di una grammatica libera può essere generata mediante una derivazione sinistra.
2. Ogni frase di una grammatica libera può essere generata mediante una derivazione destra.
3. Fissato un albero sintattico per una stringa generata da una grammatica libera, esistono sempre una sola derivazione destra ed una sola derivazione sinistra corrispondenti a quell'albero sintattico.

## Ambiguità di una grammatica

### Frase ambigua

Data una grammatica  $G = \langle V, \Sigma, P, S \rangle$ , diciamo che una frase  $x \in L(G)$  è ambigua se ammette alberi sintattici distinti.

### Grammatica ambigua

Data una grammatica  $G = \langle V, \Sigma, P, S \rangle$ , diciamo che  $G$  è ambigua se esiste almeno una frase  $x \in L(G)$  che sia ambigua.

### Grado di ambiguità della grammatica

Data una grammatica  $G = \langle V, \Sigma, P, S \rangle$ , chiamiamo grado di ambiguità di  $G$  il massimo numero di alberi sintattici distinti con i quali è possibile generare una delle frasi di  $L(G)$ .

### Dimostrazione dell'ambiguità

Dimostrare se una grammatica è ambigua o no è un problema indecidibile, nel caso generale. Quindi:

1. Se si deve dimostrare l'ambiguità di una grammatica, si va semplicemente a ricercare un esempio di frase ad essa appartenente che ammetta più alberi sintattici.
2. Se si deve dimostrare la non ambiguità di una grammatica, si deve ragionare caso per caso.

### Linguaggio inerentemente ambiguo

Un linguaggio è inerentemente ambiguo se tutte le grammatiche che lo generano sono ambigue.

## Eliminazione delle ambiguità

### Eliminazione della ricorsione bilaterale

- Esempio n. 1

Data la grammatica:

$$E \rightarrow E + E|i$$

Possiamo eliminare l'ambiguità nel modo seguente:

$$E \rightarrow E + i|i$$

- Esempio n. 2

Data la grammatica:

$$A \rightarrow aA|Ab|c$$

Possiamo eliminare l'ambiguità nel modo seguente:

$$S \rightarrow AcB$$

$$A \rightarrow aA|\varepsilon$$

$$B \rightarrow bB|\varepsilon$$

### Altri esempio

- Esempio n. 1

Data la grammatica:

$$S \rightarrow DcD$$

$$D \rightarrow Dc|bD|\varepsilon$$

Possiamo eliminare l'ambiguità nel modo seguente:

$$S \rightarrow BcD$$

$$B \rightarrow bB|b|\varepsilon$$

$$D \rightarrow Dc|bD|\varepsilon$$

In questo modo impongo che la  $c$  indicata nella regola  $S$  sia la prima (nella precedente grammatica, poteva essere una qualunque, e questo dava origine all'ambiguità).

- Esempio n. 2

Data la grammatica:

$$S \rightarrow bSc|bbSc|\varepsilon$$

Possiamo eliminare l'ambiguità nel modo seguente:

$$S \rightarrow bSc|X$$

$$X \rightarrow bbXc|\varepsilon$$

- Esempio n. 3

Data la grammatica:

$$S \rightarrow aSbS|aS|c$$

È ambigua, perché la frase:

$$aaSbS$$

Può essere generata in due modi diversi:

$$S \Rightarrow aS \Rightarrow aaSbS$$

$$S \Rightarrow aSbS \Rightarrow aaSbS$$

Questo è il classico problema del *dangling else*: basti sostituire ad  $a$  il simbolo *if(cond)then* e a  $b$  il simbolo *else*, considerando  $S$  come un blocco di codice. Allora, possiamo risolvere il problema scrivendo la grammatica:

$$S \rightarrow S_1|S_2$$

$$S_1 \rightarrow aS_1bS_1|c$$

$$S_2 \rightarrow aS_1bS_2|aS$$

In questo modo, l'*else* corrisponde sempre all'*if* più interno.

## Equivalenza tra grammatiche

### Equivalenza debole

Due grammatiche sono debolmente equivalenti se generano lo stesso linguaggio.

### Equivalenza forte (o strutturale)

Due grammatiche  $G_1$  e  $G_2$  sono in relazione di equivalenza forte o strutturale se  $L(G_1) = L(G_2)$  e inoltre ogni frase del linguaggio da esse generato, viene ottenuta per mezzo dello stesso albero sintattico scheletrico condensato sia in  $G_1$  che in  $G_2$ . Si dice per la precisione che i due alberi sono sintatticamente simili.

### Equivalenza forte (o strutturale) in senso lato

Due grammatiche  $G_1$  e  $G_2$  sono in relazione di equivalenza forte o strutturale in senso lato se  $L(G_1) = L(G_2)$  e inoltre è possibile stabilire una corrispondenza biunivoca tra gli alberi scheletrici condensati attribuiti da  $G_1$  ad ogni stringa e gli alberi condensati attribuiti da  $G_2$  alla stessa stringa.

### Osservazioni

1. Se due grammatiche hanno equivalenza strutturale, sono debolmente equivalenti.
2. Due grammatiche debolmente equivalenti possono non avere equivalenza forte.
3. Stabilire se 2 grammatiche sono debolmente equivalenti è indecidibile.
4. Stabilire se 2 grammatiche hanno equivalenza forte è decidibile.

## Forme normali

### Le forme normali

Una grammatica si dice in forma normale se le regole che vi compaiono sono sottoposte ad opportuni vincoli, senza che questo riduca la classe dei linguaggi da essa generabili.

### Forma normale non annullabile

Una grammatica è in forma normale non annullabile se nessun non terminale, escluso al più l'assioma, è annullabile; un terminale si dice annullabile se è possibile derivare da esso la stringa vuota.

### Forma normale di Chomsky

Una grammatica è in forma normale di Chomsky se valgono entrambe le proprietà seguenti:

1. Tutte le sue regole sono di una delle due tipologie seguenti:
  - a) Regole omogenee binarie, cioè del tipo  $A \rightarrow BC$  dove  $B$  e  $C$  sono non terminali.
  - b) Regole terminali con parte sinistra unitaria cioè del tipo  $A \rightarrow \alpha$ , dove  $\alpha$  è un terminale.
2. Se il linguaggio generato contiene la stringa vuota, allora ammessa anche la regola  $S \rightarrow \varepsilon$ , ma l'assioma non compare mai nella parte destra di alcuna produzione.

### Forma normale in tempo reale

Una grammatica è in forma normale in tempo reale se la parte destra di tutte le sue regole inizia con un simbolo terminale.

### Forma normale di Greibach

Una grammatica è in forma normale di Greibach se la parte destra di tutte le sue regole inizia con un simbolo terminale, seguito da zero o più non terminali (cioè in ogni regola si ha uno ed un solo terminale, che deve essere il primo simbolo della parte destra).

## Trasformazioni elementari per ottenere le forme normali

Di seguito sono riportati gli algoritmi necessari per eseguire alcune trasformazioni che serviranno poi per portare una grammatica in una delle forme normali appena descritte.

### Eliminazione delle regole di copiatura

1. Si individuano quali sono i non-terminali *copia* di ognuno dei non terminali della grammatica. In particolare, per ogni non terminale  $A$  della grammatica:
  - 1.1. Si inizializza l'insieme  $Copia(A) = \{A\}$ .
  - 1.2. Si aggiungono a  $Copia(A)$  tutti i non terminali  $C$  per i quali esistono delle regole del tipo:

$$B \rightarrow C$$

Con  $B \in Copia(A)$ .

- 1.3. Si ripete il passo 2 fino al raggiungimento di un punto di stabilità.
2. Una volta individuate le copie di ogni non terminale, si sostituisce all'insieme  $P$  delle regole della grammatica l'insieme  $P'$  così ottenuto:
  - 2.1. Si inizializza  $P'$  in modo che contenga tutte le regole di  $P$ , escluse tutte e sole le regole di copiatura. Formalmente:

$$P' = P \setminus \{A \rightarrow B \mid A, B \in V\}$$

- 2.2. Per ogni regola in  $P$  del tipo:

$$B \rightarrow \alpha$$

Dove  $\alpha$  è una stringa qualsiasi (con terminali e/o non terminali):

$$\alpha \in (V \cup \Sigma)^*$$

E dove  $B \in Copia(A)$ , si aggiunge la regola:

$$A \rightarrow \alpha$$

### Trasformazione delle ricorsioni sinistre immediate in ricorsioni destre

Per farlo, si considerano tutte le regole del tipo:

$$A \rightarrow A\beta_1|A\beta_2|\dots|A\beta_h|\gamma_1|\gamma_2|\dots|\gamma_k \quad h, k \geq 1$$

Dove si sottintende che quelle indicate siano tutte e sole le alternative per il non terminale  $A$ ; si possono eliminare allora le s-ricorsioni (o ricorsioni sinistre) trasformando la regola in:

$$\begin{aligned} A &\rightarrow \gamma_1 A' |\gamma_2 A' | \dots |\gamma_k A' | \gamma_1 |\gamma_2 | \dots |\gamma_k \\ A' &\rightarrow \beta_1 A' |\beta_2 A' | \dots |\beta_h A' \end{aligned}$$

### Trasformazione delle ricorsioni sinistre non immediate in ricorsioni destre

Si eliminano tutte le ricorsioni non immediate. Per illustrare questo algoritmo, chiamiamo  $\{A_1, A_2, \dots, A_m\}$  l'insieme di tutti e soli i non terminali della grammatica.

1. Allora, per ognuno degli  $m$  non terminali della grammatica, indicando con  $A_i$  il non-terminale in analisi:

- 1.1. Si considerano uno alla volta tutti i non terminali precedenti,  $A_j$  con  $j < i$ .

- 1.1.1. Per ciascuno di questi, si sostituisce a ogni regola  $A_i \rightarrow A_j \alpha$  la regola:

$$A_i \rightarrow \gamma_1 \alpha |\gamma_2 \alpha | \dots |\gamma_k \alpha$$

Dove:

$$\gamma_1 |\gamma_2 | \dots |\gamma_k$$

Sono tutte e sole le alternative di  $A_j$ .

- 1.2. Si eliminano tutte le ricorsioni sinistre immediate mediante l'algoritmo apposito.

## Come ottenere le forme normali

### Forma normale non annullabile

Per eliminare i non terminali annullabili:

1. Si individua quali sono i terminali annullabili; per fare ciò:
  - 1.1. Si inizializza un insieme *Null* in modo che contenga tutti i non terminali che sono parte sinistra di una regola vuota.
  - 1.2. Si aggiungono a *Null* tutti i non terminali che sono parte sinistra di una regola nella cui parte destra compaiono solo non-terminali già appartenenti a *Null*.
  - 1.3. Si itera il passo 1.2 fino al raggiungimento di un punto di stabilità.
2. Si rendono non annullabili tutti i non terminali dell'insieme *Null*:
  - 1.1. Si eliminano tutte le regole

$$A \rightarrow A_1 \dots A_n$$

Nelle quali almeno un non terminale  $A_i$  che compare nella parte destra è appartenente a *Null*. A ciascuna di tali regole, si sostituisce un insieme di regole, ottenute sostituendo in tutte le possibili combinazioni, a tutti i non terminali annullabili che compaiono nella parte destra, tutte le "parti sinistre" corrispondenti in altre regole.

- 1.2. Si eliminano tutte le regole vuote.
- 1.3. Si pulisce la grammatica.

### Forma normale di Chomsky

Una grammatica in forma normale di Chomsky viene ottenuta nel modo seguente:

1. Si porta la grammatica in forma normale non annullabile.
2. Per ogni regola del tipo  $A \rightarrow A_1 \dots A_n$ , si ripete iterativamente il procedimento seguente, fino ad ottenere una grammatica nella quale siano verificate tutte le condizioni espresse dalla definizione di forma normale di Chomsky:
  - 1.1. Si elimina la regola  $A \rightarrow A_1 \dots A_n$ .
  - 1.2. Si aggiungono i non terminali  $\langle A_1 \rangle$  e  $\langle A_2 \dots A_n \rangle$ .
  - 1.3. Se  $A_1$  è un terminale, si aggiunge la regola  $\langle A_1 \rangle \rightarrow A_1$ .
  - 1.4. Si aggiunge la regola  $A \rightarrow \langle A_1 \rangle \langle A_2 \dots A_n \rangle$ .
  - 1.5. Si aggiunge la regola  $\langle A_2 \dots A_n \rangle \rightarrow A_2 \dots A_n$ .

### Forma normale in tempo reale

Per ottenere la grammatica in forma normale in tempo reale:

1. Si eliminano tutte le ricorsioni sinistre (mediante l'apposito algoritmo).
2. Si espandono con trasformazioni elementari tutti i non terminali eventualmente presenti all'inizio della parte destra. Ad esempio, se si ha:

$$S \rightarrow Bb$$

$$B \rightarrow bD$$

Si esegue semplicemente la trasformazione:

$$S \rightarrow bDb$$

### Forma normale di Greibach

Per ottenere la grammatica in forma normale di Greibach:

1. Si trasforma la grammatica in forma normale in tempo reale.
2. Si introducono dei nuovi non terminali per sostituire i non terminali che eventualmente si trovino in posizioni diverse dalla prima nella parte destra di qualche regola. Ad esempio:

$$S \rightarrow aBcD$$

Si trasforma in:

$$S \rightarrow aBCD$$

$$C \rightarrow c$$

## Il linguaggio di Dyck

### *Il linguaggio di Dyck*

Il linguaggio di Dyck è il linguaggio delle stringhe ben parentesizzate. Nel seguito, anziché utilizzare i convenzionali simboli di parentesi aperta e chiusa, utilizzeremo:

1.  $a$  per le parentesi tonde aperte e  $a'$  per le parentesi tonde chiuse.
2.  $b$  per le parentesi quadre aperte e  $b'$  per le parentesi quadre chiuse.

### *La grammatica libera che genera il linguaggio di Dyck*

Il linguaggio di Dyck può essere generato per mezzo di una grammatica libera del tipo:

$$S \rightarrow \varepsilon | SbSb' | SaSa'$$

O, equivalentemente:

$$S \rightarrow \varepsilon | bSb'S | aSa'S$$

## Grammatica delle espressioni aritmetiche

### *Il linguaggio delle espressioni aritmetiche*

Consideriamo adesso il linguaggio delle espressioni aritmetiche che possono contenere gli operatori  $+$  e  $*$  e che possono contenere delle parentesi tonde (ben parentesizzate). Indicheremo con  $i$  un generico numero intero, senza definire la grammatica corrispondente.

### *Esempio di grammatica ambigua*

Una grammatica che genera questo linguaggio è:

$$E \rightarrow E + E | E * E | (E) | i$$

Tale grammatica è ambigua: ad ogni frase possono corrispondere diversi alberi sintattici, e questo equivale all'impossibilità da parte del calcolatore di stabilire la precedenza tra gli operatori.

### *La grammatica non ambigua*

Una grammatica non ambigua che genera il linguaggio in analisi è invece:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow i$$

In questo modo si definisce la priorità dell'operatore  $*$  rispetto all'operatore  $+$ .



# Confronto tra REG e LIB

## Ottenere la grammatica libera dall'espressione regole

### Il concetto

La famiglia *REG* è un sottoinsieme della famiglia *LIB*, perciò il linguaggio generato da un'espressione regolare può sempre essere generato anche da una grammatica libera. Esiste un algoritmo per ottenere una grammatica libera a partire dall'espressione regolare.

### L'algoritmo

Si utilizzano banalmente le regole indicate in tabella:

Espressione regolare	Corrispondente regola nella grammatica
$r = r_1 r_2 \dots r_3$	$E \rightarrow E_1 E_2 \dots E_3$
$r = r_1 \cup r_2 \cup \dots \cup r_3$	$E \rightarrow E_1   E_2   \dots   E_3$
$r = (r_1)^*$	$E \rightarrow \varepsilon   E_1 E$ oppure $E \rightarrow \varepsilon   E E_1$
$r = (r_1)^+$	$E \rightarrow E_1   E_1 E$ oppure $E \rightarrow E_1   E E_1$
$r = b$	$E \rightarrow b$
$r = \varepsilon$	$E \rightarrow \varepsilon$

### Esempio

L'espressione regolare

$$r = (a|b)^* c (b|d)^+$$

Diventerà:

$$E \rightarrow E_1 E_2 E_3$$

$$E_2 \rightarrow c$$

$$E_4 \rightarrow a|b$$

$$E_1 \rightarrow \varepsilon | E_4 E_1$$

$$E_3 \rightarrow E_5 | E_5 E_3$$

$$E_5 \rightarrow b|d$$

### Osservazione

Se l'espressione regolare di partenza è ambigua, ottengo una grammatica ambigua.

## Le grammatiche unilineari: un formalismo equivalente alle e.r.

### Grammatica unilineare a destra

Una grammatica *G* si dice unilineare a destra se tutte le sue regole sono lineari a destra, ovvero contengono al più un non terminale nella parte destra e, se è presente, tale non terminale è il simbolo più a destra della parte destra della regola in cui compare:

$$B \rightarrow uA$$

$$u \in \Sigma^*$$

$$A \in V \cup \{\varepsilon\}$$

### Grammatica unilineare a sinistra

Una grammatica *G* si dice unilineare a sinistra se tutte le sue regole sono lineari a sinistra, ovvero contengono al più un non terminale nella parte destra e, se è presente, tale non terminale è il simbolo più a sinistra della parte destra della regola in cui compare:

$$B \rightarrow Au$$

$$u \in \Sigma^*$$

$$A \in V \cup \{\varepsilon\}$$

### Grammatica unilineare o di tipo 3

In generale, una grammatica di tipo 3 o unilineare è una grammatica unilineare a sinistra o a destra.

### Proprietà

La classe dei linguaggi generabili con grammatiche di tipo 3 è equivalente a *REG*.

### Grammatica strettamente unilineare

Una grammatica strettamente unilineare è una grammatica unilineare nella quale ogni regola contiene al più un terminale nella propria parte destra.

### Grammatica strettamente unilineare con regole terminali nulle

Una grammatica strettamente unilineare con regole terminali nulle è una grammatica unilineare nella quale le regole terminali sono tutte nulle (quindi non ci sono regole  $A \rightarrow a$ , dove *a* è un terminale).

# Proprietà di chiusura dei linguaggi regolari e liberi

## Proprietà di chiusura di *REG*

### *Operatori rispetto ai quali è chiusa*

La famiglia *REG* dei linguaggi regolari è chiusa rispetto ai seguenti operatori:

1. Concatenamento
2. Unione
3. Stella
4. Croce
5. Elevamento a potenza
6. Intersezione
7. Complemento
8. Riflessione
9. Traslitterazione a parole
10. Sostituzione di linguaggi della famiglia *REG*

### *Proprietà*

La famiglia *REG* è la più piccola famiglia di linguaggi che contiene tutti i linguaggi finiti e allo stesso tempo è chiusa rispetto agli operatori di concatenamento, unione e stella.

## Proprietà di chiusura di *LIB*

### *Operatori rispetto ai quali è chiusa*

La famiglia *LIB* dei linguaggi liberi dal contesto è chiusa rispetto ai seguenti operatori:

1. Unione
2. Concatenamento
3. Stella
4. Croce
5. Riflessione
6. Traslitterazione a parole
7. Sostituzione di linguaggi della famiglia *LIB*

### *Operatori rispetto ai quali non è chiusa*

La famiglia *LIB* dei linguaggi liberi dal contesto non è chiusa rispetto ai seguenti operatori:

1. Complemento
2. Intersezione

### *Intersezione tra un linguaggio libero ed uno regolare*

L'intersezione tra un linguaggio libero ed un linguaggio regolare è sempre un linguaggio libero:

$$(L_1 \cap R_1) \in LIB$$

### ***Costruire le grammatiche dei linguaggi ottenuti dall'operazione di operatori noti***

Date due grammatiche libere  $G_1$  e  $G_2$ :

$$G_1 = \langle V_1, \Sigma_1, P_1, S_1 \rangle$$

$$G_2 = \langle V_2, \Sigma_2, P_2, S_2 \rangle$$

possiamo introdurre dei semplici meccanismi per costruire la grammatica che genera un linguaggio ottenuto da  $L(G_1)$  e  $L(G_2)$  applicando gli operatori rispetto ai quali la famiglia *LIB* risulta essere chiusa:

1. Unione

$$G_3 = \langle V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \rightarrow S_1 | S_2\}, S \rangle \quad L(G_3) = L(G_1) \cup L(G_2)$$

Cioè semplicemente "uniamo" tutto, aggiungendo un non terminale  $S$  che rappresenta il nuovo assioma, ed inserendo la regola che porta dal nuovo assioma agli assiomi delle precedenti grammatiche.

Si noti che il linguaggio così generato è ambiguo se i due linguaggi di partenza sono non disgiunti: in tal caso, è opportuno prima renderli disgiunti, e poi applicare la regola appena esposta.

2. Concatenamento

$$G_3 = \langle V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\}, S \rangle \quad L(G_3) = L(G_1)L(G_2)$$

Il principio è analogo al precedente, con la sola differenza che la regola da aggiungere è il concatenamento tra i due precedenti assiomi.

3. Stella

$$G_3 = \langle V_1 \cup \{S\}, \Sigma_1, P_1 \cup \{S \rightarrow S_1 S_1 | \epsilon\}, S \rangle \quad L(G_3) = L(G_1)^*$$

In questo caso, si inserisce semplicemente un nuovo assioma, con la regola che porta dal nuovo assioma alla stringa vuota, oppure al vecchio assioma concatenato con sé stesso.

4. Croce

$$G_3 = \langle V_1 \cup \{S\}, \Sigma_1, P_1 \cup \{S \rightarrow S_1 | S_1 S_1\}, S \rangle \quad L(G_3) = L(G_1)^+$$

È del tutto analogo al caso precedente, ma qui cambia la regola inserita, perché non si può ottenere la stringa vuota (a meno che non appartenga al linguaggio generato da  $G_1$ ).

5. Riflessione

$$G_3 = \langle V_1, \Sigma_1, P_3, S_1 \rangle \quad L(G_3) = L(G_1)^R$$

Con:

$$P_3 = \{A \rightarrow \alpha^R | (A \rightarrow \alpha) \in P_1\}$$

Cioè basta semplicemente eseguire il riflesso della parte destra di tutte le regole della grammatica di partenza.

# Classificazione delle grammatiche

## Le classificazione di Chomsky

### Classificazione di Chomsky

Chomsky ha proposto una classificazione delle grammatiche, determinando così un ordine sulla base della loro generalità e definendo i "tipi" da 0 a 3, dove il tipo 3 rappresenta le grammatiche meno generali, il tipo 0 è il caso più generale.

### Grammatiche di tipo 0: famiglia dei linguaggi ricorsivamente enumerabili

Hanno come riconoscitore associato le macchine di Turing e generano i linguaggi ricorsivamente enumerabili. I linguaggi generati non sono in generale decidibili.

Le regole sono del tipo più generale possibile:

$$\beta \rightarrow \alpha \quad \alpha, \beta \in (\Sigma \cup V)^*, \quad \beta \neq \varepsilon$$

### Grammatiche di tipo 1: famiglia dei linguaggi contestuali (o dipendenti dal contesto)

Hanno come riconoscitore associato le macchine di Turing con nastro di lunghezza uguale a quella della stringa da riconoscere. Generano i linguaggi contestuali, che sono sempre decidibili.

Le regole sono del tipo:

$$\beta \rightarrow \alpha \quad \alpha, \beta \in (\Sigma \cup V)^*, \quad \beta \neq \varepsilon, \quad |\beta| < |\alpha|$$

Queste grammatiche sono dette contestuali, oppure dipendenti dal contesto, oppure ancora context-sensitive.

### Grammatiche di tipo 2: famiglia dei linguaggi liberi

Hanno come riconoscitore associato gli automi a pila non deterministici. Generano i linguaggi liberi, che sono sempre decidibili e che abbiamo già ampiamente analizzato. Si tratta quindi delle grammatiche libere, le cui regole sono del tipo:

$$A \rightarrow \alpha \quad \alpha \in (\Sigma \cup V)^*, \quad A \in V$$

### Grammatiche di tipo 3: famiglia dei linguaggi regolari

Hanno come riconoscitore associato gli automi a stati finiti. Generano i linguaggi regolari. Si tratta quindi delle grammatiche unilineari (a destra o a sinistra), le cui regole sono del tipo:

1. Se la grammatica è unilineare a destra:

$$A \rightarrow \alpha B \quad \alpha \in \Sigma^*, \quad A \in V, \quad B \in V \cup \{\varepsilon\}$$

2. Se la grammatica è unilineare a sinistra:

$$A \rightarrow Ba \quad \alpha \in \Sigma^*, \quad A \in V, \quad B \in V \cup \{\varepsilon\}$$

### Osservazione

Le famiglie di grammatiche sono legate da una relazione di contenimento stretto:

1. Tutte le grammatiche di tipo 3 sono anche di tipo 2, di tipo 1 e di tipo 0.
2. Tutte le grammatiche di tipo 2 sono anche di tipo 1 e di tipo 0.
3. Tutte le grammatiche di tipo 1 sono anche di tipo 0.

### Proprietà di chiusura

La tabella seguente mostra le proprietà di chiusura per le varie famiglie:

	Unione	Concatenamento	Stella	Riflesso	Intersezione con linguaggio regolare	Complemento
0	Sì	Sì	Sì	Sì	Sì	No
1	Sì	Sì	Sì	Sì	Sì	Sì
2	Sì	Sì	Sì	Sì	Sì	No
3	Sì	Sì	Sì	Sì	Sì	Sì

# Grammatiche libere estese o EBNF

## Le grammatiche libere estese (EBNF)

### Definizione di grammatiche libere estese o EBNF

Una grammatica libera estesa (o EBNF, Extended Backus Naur Form) è una particolare grammatica  $G = \langle V, \Sigma, P, S \rangle$  che contiene esattamente  $|V|$  regole (cioè una regola per ogni non terminale); ogni regola ha un diverso non terminale come parte sinistra ed ha come parte destra un'espressione regolare di alfabeto  $V \cup \Sigma$ , nella quale possono comparire anche gli operatori derivati croce, elevamento a potenza ed opzionalità.

### Derivazione

Data una grammatica EBNF  $G = \langle V, \Sigma, P, S \rangle$ , diciamo che dalla stringa  $\eta_1$  deriva la stringa  $\eta_2$  e scriviamo:

$$\eta_1 \Rightarrow \eta_2$$

$$\eta_1, \eta_2 \in (V \cup \Sigma)^*$$

Se:

$$\eta_1 = \alpha A \beta$$

$$\eta_2 = \alpha \gamma \beta$$

Ed esiste una produzione:

$$A \rightarrow e$$

$$e \Rightarrow^* \gamma$$

### Capacità generativa

Siccome la famiglia *LIB* è chiusa rispetto alle operazioni regolari, la capacità generativa delle grammatiche estese è uguale a quella delle grammatiche libere.

### Osservazione

Siccome la lunghezza della stringa che si ottiene con un solo passo di derivazione può essere anche molto più lunga di quella di partenza, in generale le grammatiche libere estese hanno alberi sintattici molto "più larghi" e meno profondi di quelle libere.

### Ambiguità

1. Se una grammatica estesa viene ottenuta da una grammatica libera ambigua, allora anche la grammatica EBNF sarà ambigua.
2. Se una grammatica estesa viene ottenuta da una grammatica libera non ambigua, è possibile che, a seguito dell'inserimento delle espressioni regolari, la grammatica EBNF sia ambigua.

# Automi finiti

## Gli automi finiti

Riprendiamo solo alcuni concetti elementari relativi agli automi finiti, già studiati in Informatica Teorica.

### Definizione formale

Un automa finito è una quintupla:

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle$$

Dove:

1.  $Q$  è l'insieme degli stati. Ha dimensione finita e non è vuoto.
2.  $\Sigma$  è l'alfabeto di ingresso, detto anche alfabeto terminale.
3.  $\delta$  è la funzione di transizione, che codifica le mosse dell'automa, e che è del tipo:  $\delta: Q \times \Sigma \rightarrow Q$ .
4.  $q_0$  è lo stato iniziale. Si deve avere:  $q_0 \in Q$ .
5.  $F$  è l'insieme degli stati finali. Si deve avere:  $F \subseteq Q$ .

### Funzione di transizione estesa

Si introduce il concetto di funzione di transizione estesa  $\delta^*: Q \times \Sigma^* \rightarrow Q$ , che è definita come:

$$\begin{cases} \delta^*(q, \varepsilon) = q \\ \delta^*(q, xa) = \delta^*(\delta^*(q, x), a), & a \in \Sigma, b \in \Sigma^* \end{cases}$$

Spesso la funzione di transizione estesa viene indicata con  $\delta$ .

### Stringa riconosciuta dall'automa a stati finiti

Una stringa  $x$  viene riconosciuta o accettata da un automa a stati finiti  $M$  se:

$$\delta(q_0, x) \in F$$

Cioè partendo dallo stato iniziale e dando come ingresso la stringa  $x$ , si raggiunge uno stato finale.

### Linguaggio riconosciuto dall'automa a stati finiti

Il linguaggio riconosciuto dall'AF  $M$  è l'insieme di tutte e sole le stringhe riconosciute da  $M$ :

$$L(M) = \{x | x \text{ viene riconosciuta da } M\}$$

### Osservazione

Si noti che, nel riconoscimento di una stringa, un AD ha sempre la massima efficienza in assoluto, in quanto la riconosce sempre in tempo reale, scandendola una sola volta da sinistra a destra.

### Completamento con stato posso

La funzione di transizione potrebbe non essere definita in corrispondenza di particolari coppie stato-ingresso. È possibile *completare* l'automa aggiungendo le transizioni mancanti tutte verso un nuovo stato, detto pozzo e indicato con  $q_{err}$ , dal quale non è definita alcuna transizione uscente: per ogni possibile ingresso, se si parte da  $q_{err}$  si rimane in  $q_{err}$ . Il linguaggio riconosciuto dall'automa completato è lo stesso che veniva riconosciuto dall'automa di partenza.

## Automa pulito

### Stato raggiungibile

Uno stato  $p$  è raggiungibile se esiste un calcolo, ovvero una sequenza di mosse, che porta l'automa da uno stato  $q$  allo stato  $p$ .

### Stato accessibile

Uno stato  $p$  è accessibile se è raggiungibile dallo stato iniziale  $q_0$ .

### Stato post-accessibile

Uno stato  $p$  è post-accessibile se da esso può essere raggiunto uno stato finale.

### Stato utile

Uno stato  $p$  è utile se è accessibile e post-accessibile.

### Automa pulito

Un automa è pulito se tutti i suoi stati sono utili.

### Proprietà

Ogni automa a stati finiti ammette un automa pulito equivalente (che riconosce lo stesso linguaggio): per ottenerlo, basta eliminare dall'automa di partenza tutti gli stati che non sono utili e tutti gli archi entranti e/o uscenti da tali stati.

## Automa minimo

### Stati indistinguibili

Diciamo che lo stato  $p$  è indistinguibile dallo stato  $q$  se e solo se, per ogni stringa  $x$ , vale una delle seguenti condizioni:

1.  $\delta(p, x)$  e  $\delta(q, x)$  appartengono entrambi ad  $F$ , cioè sono entrambi finali.
2. Né  $\delta(p, x)$  né  $\delta(q, x)$  appartengono ad  $F$ , cioè nessuno dei due è finale.
3. Scandendo tutta la stringa  $x$  da  $q$  e da  $p$  non è possibile che si giunga in due stati tali che uno sia finale e l'altro no.

Impossible to compute the indistinguishability relation directly from its definition one should consider the whole accepted language, which may be infinite  
we compute the indistinguishability relation through its complement: the distinguishability relation  
it can be computed through its inductive definition as follows

### Distinguibilità

Uno stato  $p$  è distinguibile da uno stato  $q$  se è vera almeno una delle seguenti condizioni:

1.  $p$  è uno stato finale e  $q$  è non finale, o viceversa.
2. Esiste un simbolo  $a \in \Sigma$  per il quale  $\delta(p, a)$  è distinguibile da  $\delta(q, a)$ .

Si nota quindi che la definizione data è una definizione ricorsiva.

$p$  is distinguishable from  $q$  (both assumed postaccessible)  
if the set of labels on arcs outgoing from  $p$  and  $q$  are different

### Automa minimo

L'automa minimo corrispondente ad un certo automa  $M$  dato è quell'automa equivalente ad  $M$  e che abbia il minimo numero possibile di stati.

### Minimizzazione

Per minimizzare  $M$  si compila una tabella del tipo (ipotizzando di avere stati da  $q_0$  a  $q_3$ ):

$q_1$			
$q_2$			
$q_3$			
	$q_0$	$q_1$	$q_2$

Si inserisce poi una  $X$  in ogni casella all'incrocio tra coppie di stati indistinguibili, oppure si indicano le coppie di stati dalle quali dipende la distinguibilità, e si procede così iterativamente fino a quando non è più possibile aggiungere relazioni di distinguibilità: le caselle rimaste senza il simbolo  $X$  sono quelle che identificano le coppie di stati indistinguibili. Tali stati possono poi essere *fusi* in un unico stato, in modo da ridurre il più possibile il numero complessivo di stati dell'automa.

Questo algoritmo è applicabile solo a patto che l'automa di partenza sia completo.

## Automi non deterministici

### Automa non deterministico

Un automa non deterministico è un automa nel quale è vera almeno una delle seguenti condizioni:

1. Si ha almeno uno stato per il quale sono definite più mosse in corrispondenza dello stesso ingresso;
2. Si ha almeno uno stato per il quale, oltre ad una o più mosse "tradizionali" è definita almeno una  $\varepsilon$ -mossa (mosse spontanea), che permette all'automa di cambiare stato senza consumare l'ingresso.
3. Esistono più stati iniziali.

### Potenza degli automi non deterministici

Gli automi a stati finiti non deterministici hanno la stessa potenza degli AF deterministici. L'unico vantaggio che comportano è una maggiore semplicità (in genere si ottengono automi con un numero di stati di molto inferiore, e che risultano molto più semplici da comprendere o da ottenere).

### Definizione formale

Un automa non deterministico a stati finiti  $N$  è definito dalla tupla:

$$N = \langle Q, \Sigma, \delta, I, F \rangle$$

Dove  $Q$  è l'insieme degli stati,  $\Sigma$  è l'alfabeto di ingresso,  $\delta$  è la funzione di transizione, definita come:

$$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$$

Mentre  $I$  è l'insieme degli stati iniziali e  $F$  è l'insieme degli stati finali.

### Calcolo

Un calcolo di origine  $q_0$  e di termine  $q_n$ , di lunghezza  $n$  e di etichette  $a_1, \dots, a_n$  è, nel caso di automi non deterministici, una sequenza del tipo:

$$q_0 \rightarrow_{a_1} q_1 \rightarrow_{a_2} \dots \rightarrow_{a_n} q_n$$

Scritta anche come:

$$q_0 \rightarrow_{a_1 \dots a_n} q_n$$

Tale che, per ogni  $i \in [0, n)$ , si abbia:

$$q_{i+1} \in \delta(q_i, a_{i+1})$$

Dove  $a_{i+1}$  è un simbolo dell'alfabeto, oppure la stringa vuota:

$$a_{i+1} \in \Sigma \cup \{\varepsilon\}$$

### Funzione di transizione estesa

Come nel caso deterministico, è possibile definire il concetto di funzione di transizione estesa; anche in questo caso, utilizzeremo in genere il simbolo  $\delta$  per indicare la funzione di transizione estesa anziché la funzione di transizione vera e propria, come sarebbe più opportuno fare.

$$\delta: Q \times (\Sigma \cup \{\varepsilon\})^* \rightarrow \mathcal{P}(Q)$$

Tale funzione è così definita:

$$\forall q \in Q, y \in \Sigma^* \quad \delta(q, y) = \{p \mid q \rightarrow_y p\}$$

In altri termini, la  $\delta(q, y)$  è l'insieme di tutti gli stati  $p$  per i quali esiste un calcolo di origine  $q$  e di termine  $p$  le cui etichette sono i simboli della stringa  $y$ .

### Linguaggio riconosciuto

Il linguaggio riconosciuto dall'automa non deterministico  $N$  è l'insieme di tutte e sole le stringhe  $x$  per le quali esiste almeno un calcolo che porta da uno stato iniziale ad uno stato finale:

$$L(N) = \{x \mid q \rightarrow_x r, q \in I, r \in F\}$$

Questo significa anche che:

$$L(N) = \{x \in \Sigma^* \mid \exists q \in I: \delta(q, x) \cap F \neq \emptyset\}$$



## Operazioni sugli automi non deterministici

### Ottenere l'automa che riconosca il riflesso

Dato un qualsiasi automa  $A$ , sia esso deterministico o indeterministico, è sempre possibile ottenere l'automa che riconosca il linguaggio  $[L(A)]^R$  semplicemente:

1. Scambiando gli stati iniziali con gli stati finali;
2. Calcolando la relazione inversa della funzione di transizione  $\delta$  (cioè, a livello pratico, invertendo il verso di tutte le frecce che indicano le transizioni dell'automa);

L'automa che si ottiene è, nel caso generale, un automa non deterministico.

### Eliminare gli stati iniziali multipli dell'automa

Dato un automa non deterministico  $A$  con più stati iniziali, possiamo ottenere un automa non deterministico  $A'$  ad esso equivalente ma con un solo stato iniziale semplicemente:

1. trasformando gli stati iniziali di  $A$  in stati non iniziali;
2. introducendo un nuovo stato  $Q_0$ , iniziale, dal quale si ha una  $\varepsilon$ -mossa verso ciascuno degli stati che in  $A$  erano iniziali.

## Ambiguità degli automi

### Definizione di automa ambigui

Un automa  $A$  si dice ambiguo se e solo se accetta una frase con più calcoli diversi. Ne consegue che gli automi deterministici sono tutti non ambigui, mentre gli automi non deterministici sono in genere ambigui.

### Proprietà

Un automa è ambiguo se e solo se la grammatica unilineare destra che si ottiene a partire da esso per mezzo dell'algoritmo che andremo a descrivere a breve è una grammatica ambigua.

## Dalla grammatica all'automa a stati finiti e viceversa

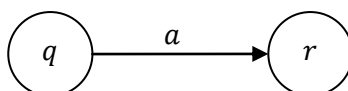
### Come ottenere la grammatica a partire dall'automa a stati finiti

Dato un automa a stati finiti che riconosca un linguaggio  $L(A)$ , è sempre possibile determinare una grammatica unilineare destra che generi il linguaggio stesso:

$$G = \langle V, \Sigma, P, S \rangle$$

E si avrà  $L(G) = L(A)$ . Per farlo, basta seguire alcune regole:

1. L'insieme dei non terminali di  $G$  è l'insieme degli stati di  $A$ .
2. L'assioma di  $G$  è il non terminale corrispondente allo stato iniziale di  $A$ .
3. Per ogni transizione:



Si aggiunge una regola:

$$q \rightarrow ar$$

4. Se lo stato di arrivo  $q$  è finale si aggiunge anche la regola:

$$q \rightarrow \varepsilon$$

### Come ottenere l'automa a stati finiti a partire dalla grammatica unilineare destra

Sia data una grammatica unilineare a destra con regole strettamente unilineari:

$$G = \langle V, \Sigma, P, S \rangle$$

Secondo quanto abbiamo già visto, la grammatica sarà costituita solo da regole del tipo:

$$A \rightarrow uB \quad u \in \Sigma \cup \{\varepsilon\}, \quad B \in V \cup \{\varepsilon\}$$

Allora possiamo costruire un automa a stati finiti (non deterministico)  $N$  che riconosca  $L(G)$ :

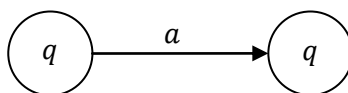
$$N = \langle Q, \Sigma, \delta, q_0, F \rangle$$

Dove:

1. L'alfabeto di ingresso di  $N$  è l'insieme dei terminali di  $G$ ;
2. L'insieme  $Q$  degli stati di  $N$  è l'insieme dei non terminali di  $G$ :  $Q = V$ .
3. Lo stato iniziale  $q_0$  di  $N$  è lo stato associato all'assioma  $S$  di  $G$ .
4. Per ogni regola:

$$p \rightarrow aq, \quad a \in \Sigma, \quad q \in V$$

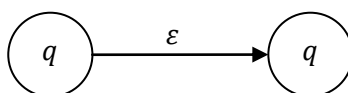
Si aggiunge in  $A$  una transizione del tipo:



5. Per ogni regola:

$$p \rightarrow q, \quad q \in V$$

Si aggiunge in  $A$  una  $\varepsilon$ -mossa, del tipo:



6. Se è presente una regola del tipo:

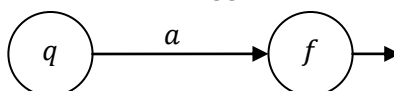
$$p \rightarrow \varepsilon$$

Allora si trasforma  $p$  in uno stato finale.

7. Se è presente una regola del tipo:

$$p \rightarrow a, \quad a \in \Sigma$$

Allora si inserisce una transizione verso uno stato aggiuntivo  $f$ , che deve essere terminale:



### Come ottenere l'automa a stati finiti a partire dalla grammatica unilineare sinistra

Se è data una grammatica strettamente unilineare a sinistra:

$$G = \langle V, \Sigma, P, S \rangle$$

Cioè costituita solo da regole del tipo:

$$A \rightarrow Bu \quad u \in \Sigma \cup \{\varepsilon\}, \quad B \in V \cup \{\varepsilon\}$$

Allora possiamo costruire un automa a stati finiti (non deterministico)  $N$  che riconosca  $L(G)$ :

$$N = \langle Q, \Sigma, \delta, q_0, F \rangle$$

Per farlo:

1. Costruiamo la grammatica  $G'$  che riconosca il linguaggio  $[L(G)]^R$ , semplicemente "ribaltando" la parte destra di tutte le produzioni di  $G$ .
2. Costruiamo l'automa  $A'$  che riconosca  $[L(G)]^R = L(G')$ , secondo il procedimento visto per le grammatiche strettamente unilineari a destra;
3. Costruiamo l'automa che riconosce il linguaggio  $[L(A')]^R = L(G)$ , secondo l'algoritmo precedentemente descritto.

## Trasformazione da AF non deterministico ad AF deterministico

### Passo 0: eliminazione degli stati iniziali multipli

Per prima cosa, ci si riconduce ad un automa con un solo stato iniziale, secondo la tecnica già descritta.

### Passo 1: eliminazione delle mosse spontanee

Per prima cosa, si eliminano le mosse spontanee dall'automa. Per farlo:

1. Si ottiene la grammatica unilineare destra associata all'automa a stati finiti dato, utilizzando l'algoritmo precedentemente descritto;
2. Si cancellano dalla grammatica così ottenuta tutte le regole di copiatura, usando l'algoritmo noto;
3. Si ottiene l'automa a stati finiti corrispondente alla grammatica così ottenuta, utilizzando ancora una volta uno degli algoritmi già studiati.

### Passo 2: determinizzazione con l'insieme delle parti finiti

A questo punto, si devono eliminare tutte le situazioni in cui da uno stato sono definite più mosse verso altri stati in corrispondenza dello stesso simbolo in ingresso. Si usa allora l'algoritmo di determinizzazione con l'insieme delle parti finite, che prevede i seguenti passi:

1. Per ogni stato e per ogni ingresso, si calcola l'insieme dei possibili stati di arrivo:

$$\forall q \in Q, \forall b \in \Sigma \quad \text{calcolo} \quad \delta(q, b)$$

e, se non esistente ancora, si crea un nuovo stato associato al sottoinsieme di  $Q$  così ottenuto.

2. Per ogni stato  $\{A_1, \dots, A_n\}$  aggiunto, si calcola quali dovranno essere le transizioni definite della funzione di transizione del nuovo automa; per farlo:

$$\forall b \in \Sigma \quad \text{calcolo} \quad \delta(\{A_1, \dots, A_n\}, b) = \delta(A_1, b) \cup \dots \cup \delta(A_n, b)$$

E, se ancora non esiste, si crea un nuovo stato associato al sottoinsieme di  $Q$  così ottenuto.

3. Si itera il procedimento fino a quando non si hanno più stati da aggiungere.

## Dall'automa all'espressione regolare: il metodo BMC

### Assunzioni iniziali

Assumiamo che sia dato un automa  $A = \langle Q, \Sigma, \delta, i, t \rangle$  nel quale:

1. Si ha un solo stato iniziale  $i$  (se così non fosse, abbiamo già visto come rendere unico lo stato iniziale dell'automa). Tale stato deve inoltre essere privo di archi entranti (anche questa condizione può eventualmente essere ottenuta aggiungendo un nuovo stato con opportune  $\varepsilon$ -mosse).
2. Si ha un solo stato finale  $t$ , privo di archi uscenti. Se così non fosse, si può ancora una volta trasformare l'automa di partenza aggiungendo un nuovo stato con opportune  $\varepsilon$ -mosse.

### Terminologia

1. *Stati interni*

Tutti gli stati dell'automa che non iniziali o finali, cioè gli stati dell'insieme  $Q \setminus \{i, t\}$ .

2. *Automa generalizzato*

Chiamiamo *automa generalizzato* un automa nel quale gli archi possono anche essere etichettati per mezzo di espressioni regolari, con il significato più ovvio ed intuitivo (cioè la transizione avviene se si riceve un ingresso che sia una sequenza di caratteri appartenente al linguaggio generato da quella espressione regolare).

### Il procedimento del metodo Brozowski e McCluskey

Il modo di procedere è basato sull'idea di eliminare uno alla volta tutti gli stati interni, aggiungendo delle mosse compensatorie che preservano il linguaggio riconosciuto, e che siano etichettate da espressioni regolari. Il procedimento viene portato avanti fino ad avere solamente i due stati  $i$  e  $t$ , con un solo arco da  $i$  a  $t$  etichettato da un'espressione regolare, che sarà quella cercata.

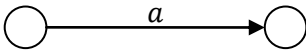
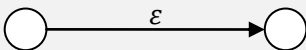
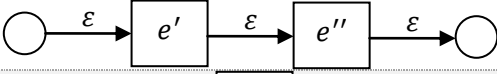
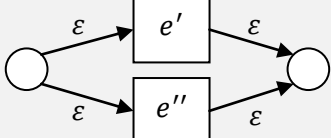
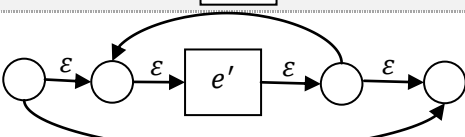
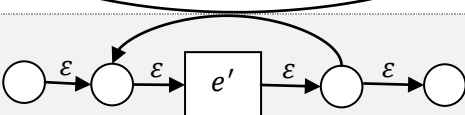
L'ordine di eliminazione degli stati interni è irrilevante.

## Dall'espressione regolare all'automa riconoscitore

### Il metodo di Thomson (o strutturale)

Il primo metodo per ottenere l'automa in grado di riconoscere una data espressione regolare è quello di Thomson; è molto semplice, ma si ottiene un automa non deterministico con  $\varepsilon$ -mosse.

Prevede semplicemente che si eseguano le seguenti trasformazioni:

Se si ha una sottoespressione del tipo	Si inserisce nell'automa una struttura del tipo
$a$	
$\varepsilon$	
$e'e''$	
$e' e''$	
$(e')^*$	
$(e')^+$	

### Algoritmo di Glushkov, McNaughton e Yamada (GMY)

Una seconda possibilità è quella di utilizzare l'algoritmo GMY, che permette di ottenere un automa non deterministico, ma privo di  $\varepsilon$ -mosse. Il metodo si basa sull'utilizzo dei linguaggi locali:

#### - Passo 21

Se il linguaggio di partenza non è locale, si costruisce si numerano tutti i simboli dell'espressione regolare: si ottiene così un'espressione regolare che genera certamente un linguaggio locale.

Ad esempio, l'espressione regolare:

$$aba^*c$$

Diventerà:

$$a_1b_2a_3^*c_4$$

Siccome ogni simbolo compare una sola volta, il calcolo di inizi e fini del linguaggio generato dalle espressioni numerati è in genere molto semplice.

#### - Passo 2

Si calcolano gli insiemi delle inizi, delle fini e dei digrammi:

##### a) *Calcolo di $Ini(L)$ dall'espressione regolare*

Basta applicare alcune semplici regole:

1.  $Ini(\emptyset) = \emptyset$
2.  $Ini(a) = \{a\}$
3.  $Ini(e_1|e_2) = Ini(e_1) \cup Ini(e_2)$
4. Se  $\varepsilon \in L(e_1)$ ,  $Ini(e_1e_2) = Ini(e_1) \cup Ini(e_2)$
5. Se  $\varepsilon \notin L(e_1)$ ,  $Ini(e_1e_2) = Ini(e_1)$
6.  $Ini(e^+) = Ini(e)$
7.  $Ini(e^*) = Ini(e)$

b) *Calcolo di  $Fin(L)$  dall'espressione regolare*

Basta applicare alcune semplici regole:

1.  $Fin(\emptyset) = \emptyset$
2.  $Fin(a) = \{a\}$
3.  $Fin(e_1|e_2) = Fin(e_1) \cup Fin(e_2)$
4. Se  $\varepsilon \in L(e_2)$ ,  $Ini(e_1e_2) = Fin(e_1) \cup Fin(e_2)$
5. Se  $\varepsilon \notin L(e_2)$ ,  $Ini(e_1e_2) = Fin(e_2)$
6.  $Fin(e^+) = Fin(e)$
7.  $Fin(e^*) = Fin(e)$

c) *Calcolo di  $Dig(L)$  dall'espressione regolare*

Basta applicare alcune semplici regole:

1.  $Dig(\emptyset) = \emptyset$
2.  $Dig(a) = \emptyset$
3.  $Dig(e_1|e_2) = Dig(e_1) \cup Dig(e_2)$
4.  $Dig(e_1e_2) = Dig(e_1) \cup Dig(e_2) \cup Fin(e_1).Ini(e_2)$
5.  $Dig(e^+) = Dig(e) \cup Fin(e).Ini(e)$
6.  $Dig(e^*) = Dig(e) \cup Fin(e).Ini(e)$

- Passo 3

Si costruisce l'automa riconoscitore del linguaggio locale:

1. Gli stati dell'automa sono i simboli terminali del linguaggio, con l'aggiunta di uno stato  $q_0$ .
2. Lo stato  $q_0$  è l'unico stato iniziale dell'automa.
3. Gli stati appartenenti a  $Fin(L)$  sono tutti e soli gli stati finali dell'automa.
4. Per ogni simbolo  $a \in Ini(L)$ , si definisce la transazione da  $q_0$  ad  $a$  con etichetta  $a$ .
5. Per ogni coppia  $ab \in Dig(L)$  si definisce la transazione dallo stato  $a$  a  $b$  con etichetta  $b$ .

- Passo 4

Se il linguaggio di partenza non era locale, si eliminano da tutte le etichette i pedici inseriti per rendere locale il linguaggio: l'automa così ottenuto riconosce il linguaggio di partenza.

**Algoritmo di Berry & Sethi (BS)**

Il terzo ed ultimo algoritmo possibile è quello di Berry & Sethi, che permette di ottenere direttamente un automa a stati finiti deterministico.

1. Si numera l'espressione regolare  $e$ , ottenendo così l'espressione regolare  $e'$  numerata.
2. Si inizializza l'insieme degli stati dell'automa in modo da avere  $Q = \{Ini(e' \rightarrow)\}$ .
3. Fino a quando esiste almeno uno stato in  $Q$  che non sia ancora stato visitato, si prende in analisi uno degli stati  $q \in Q$  ancora rimasti non analizzati e si eseguono le seguenti operazioni:
  - 3.1. Si considera ogni singolo carattere  $b$  dell'alfabeto  $\Sigma$  non numerato e, per ogni  $b$ :
    - 3.1.1. Si calcola lo stato  $q'$  che contiene l'unione dei seguiti di tutti i caratteri  $b'$  ottenuti aggiungendo un pedice qualsiasi a  $b$ , che appartengano allo stato  $q$  in analisi (gli stati sono definiti come insiemi di simboli numerati dell'alfabeto terminale di partenza):
 
$$q' = \bigcup_{i: b_i \in q} Seg(b_i)$$
    - 3.1.2. Se l'insieme  $q'$  non appartiene ancora a  $Q$  e non è vuoto, allora si pone:
 
$$Q = Q \cup \{q'\}$$
    - 3.1.3. Si aggiunge la mossa:
 
$$q \rightarrow_b q'$$
4. Lo stato iniziale è  $Ini(e' \rightarrow)$ .
5. Gli stati finali sono tutti quelli che contengono il simbolo  $\rightarrow$ .

## Determinizzazione di automi mediante algoritmo BS

### Il problema

L'algoritmo BS, che consente di ottenere un automa deterministico a partire da un'espressione regolare qualsiasi, può anche essere usato, con piccole modifiche, per determinizzare un automa a stati finiti non deterministico, eventualmente comprendente anche delle  $\varepsilon$ -mosse.

### L'algoritmo

1. Si numerano in modo distinto le etichette di tutti gli archi dell'automa dato che non corrispondono ad  $\varepsilon$ -mosse.
2. Si calcolano gli insiemi degli inizi, delle fini e dei seguiti del linguaggio generato dall'automa (le regole da usare derivano in modo ovvio da quelle relative all'espressione regolare).
3. Si applica l'algoritmo di BS esattamente come è stato descritto quando introdotto nel precedente paragrafo.

## Automi a stati finiti e operazioni tra linguaggi

### Automa riconoscore dell'unione

Dati due automi:

$$A_1 = \langle Q_1, \Sigma_1, \delta_1, q_{0_1}, F_1 \rangle$$

$$A_2 = \langle Q_2, \Sigma_2, \delta_2, q_{0_2}, F_2 \rangle$$

L'automa  $A_3$  che riconosce il linguaggio:

$$L(A_1) \cup L(A_2)$$

Viene così costruito:

1. L'alfabeto d'ingresso è l'unione degli alfabeti di ingresso  $\Sigma_1$  e  $\Sigma_2$ .
2. Lo stato iniziale  $q_{0_3}$  di  $A_3$  viene ottenuto semplicemente creando un nuovo stato che rappresenti la fusione tra  $q_{0_1}$  e  $q_{0_2}$ .
3. Tutti gli altri stati e le altre transizioni rimangono esattamente le stesse che si avevano nei singoli automi, ma risultano essere "fusi" in un automa unico.
4. Gli stati finali sono dati dall'unione degli stati finali dei due automi di partenza:

$$F_1 \cup F_2$$

5. Inoltre, se almeno uno dei due linguaggi di partenza contiene la stringa vuota,  $q_{0_3}$  è finale.

### Automa riconoscore del concatenamento

Dati due automi:

$$A_1 = \langle Q_1, \Sigma_1, \delta_1, q_{0_1}, F_1 \rangle$$

$$A_2 = \langle Q_2, \Sigma_2, \delta_2, q_{0_2}, F_2 \rangle$$

L'automa  $A_3$  che riconosce il linguaggio:

$$L(A_1).L(A_2)$$

Viene così costruito:

1. L'alfabeto d'ingresso è l'unione degli alfabeti di ingresso  $\Sigma_1$  e  $\Sigma_2$ .
2. Gli stati di  $A_3$  sono dati dall'unione degli stati di  $A_1$  e di  $A_2$ .
3. Lo stato iniziale di  $A_3$  è lo stato iniziale di  $A_1$ , ovvero  $q_{0_1}$ .
4. Le mosse di  $A_3$  sono date dall'unione delle mosse di  $A_1$  e delle mosse di  $A_2$ , escluse però le mosse che hanno origine dallo stato  $q_{0_2}$ . A queste si aggiungono le nuove mosse del tipo:

$$q_1 \rightarrow_a q_2$$

Tali che  $q_1 \in F_1$  e che sia definita da  $\delta_2$  la mossa del tipo:

$$q_{0_2} \rightarrow q_2$$

### Automa riconoscitore della stella

Dato l'automa:

$$A_1 = \langle Q_1, \Sigma_1, \delta_1, q_{0_1}, F_1 \rangle$$

L'automa  $A_2$  che riconosce il linguaggio:

$$L(A_1)^*$$

Viene ottenuto nel modo seguente:

1. Gli stati, l'alfabeto d'ingresso e lo stato iniziale rimangono invariati rispetto ad  $A_1$ .
2. Gli stati finali di  $A_2$  sono dati dagli stati finali di  $A_1$ , ai quali si aggiunge  $q_{0_2}$ :

$$F_2 = \{q_{0_2}\} \cup F_1$$

3. Le mosse di  $A_2$  sono tutte le mosse di  $A_1$ , con l'aggiunta di tutte le mosse del tipo:

$$q \xrightarrow{a} r$$

Dove  $q$  è uno stato finale ed esiste una transizione dallo stato iniziale ad  $r$ , con il carattere  $a$ , cioè:

$$q \in F_1$$

$$r \in \delta_1(q_{0_1}, a)$$

### Automa riconoscitore del complemento

Dato l'automa deterministico:

$$A_1 = \langle Q_1, \Sigma_1, \delta_1, q_{0_1}, F_1 \rangle$$

L'automa  $A_2$  che riconosce il linguaggio:

$$\overline{L(A_1)}$$

Viene ottenuto nel modo seguente:

1. Gli stati di  $A_2$  sono gli stessi stati di  $A_1$ ; analogamente, anche l'alfabeto di ingresso non cambia.
2. Lo stato iniziale di  $A_2$  è lo stato iniziale di  $A_1$ .
3. Gli stati finali di  $A_2$  sono tutti e soli gli stati non finali di  $A_1$ :

$$F_2 = Q \setminus F_1$$

Si noti che per poter applicare il procedimento illustrato è indispensabile che l'automa di partenza sia deterministico: in caso contrario, il risultato potrebbe non essere quello desiderato.

### Automa riconoscitore dell'intersezione

Dati due automi privi di mosse spontanee (ma non necessariamente deterministici):

$$A_1 = \langle Q_1, \Sigma_1, \delta_1, q_{0_1}, F_1 \rangle$$

$$A_2 = \langle Q_2, \Sigma_2, \delta_2, q_{0_2}, F_2 \rangle$$

L'automa  $A_3$  che riconosce il linguaggio:

$$L(A_1) \cap L(A_2)$$

Viene costruito calcolando il **prodotto cartesiano** tra i due automi di partenza, ovvero:

1. Gli stati di  $A_3$  sono tutte le possibili coppie di stati dei precedenti due:

$$Q_3 = Q_1 \times Q_2 = \{ \langle q_1, q_2 \rangle \mid q_1 \in Q_1, q_2 \in Q_2 \}$$

2. Le mosse di  $A_3$  sono tutte e sole le mosse del tipo:

$$\langle q'_1, q'_2 \rangle \xrightarrow{a} \langle q''_1, q''_2 \rangle$$

Tali che  $q'_1 \xrightarrow{a} q''_1$  è una mossa di  $A_1$  e allo stesso tempo  $q'_2 \xrightarrow{a} q''_2$  è una mossa di  $A_2$ .

3. Gli stati iniziali di  $A_3$  sono le coppie costituite da uno stato iniziale di  $A_1$  e uno stato iniziale di  $A_2$ :

$$I_3 = I_1 \times I_2 = \{ \langle q_1, q_2 \rangle \mid q_1 \in I_1, q_2 \in I_2 \}$$

4. Gli stati finali di  $A_3$  sono le coppie costituite da uno stato finale di  $A_1$  e uno stato finale di  $A_2$ :

$$F_3 = F_1 \times F_2 = \{ \langle q_1, q_2 \rangle \mid q_1 \in F_1, q_2 \in F_2 \}$$

# Automi a pila

## Gli automi a pila

### Definizione formale

Un automa a pila  $M$  (in generale non deterministico) è una settupla del tipo:

$$M = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$$

Dove:

1.  $Q$  è l'insieme degli stati di  $M$ .
2.  $\Sigma$  è l'alfabeto d'ingresso.
3.  $\Gamma$  è l'alfabeto della pila.
4.  $\delta$  è la **funzione di transizione**, definita come:

$$\delta: Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$$

Che **stabilisce sulla base dello stato attuale**, del **simbolo in ingresso** (ma sono possibili anche mosse spontanee) e del **simbolo in cima alla pila**, **quale** sarà lo **stato prossimo**, e **quali simboli aggiungere in cima alla pila**, al posto del simbolo appena letto (che viene cancellato).

5.  $q_0 \in Q$  è lo stato iniziale.
6.  $Z_0 \in \Gamma$  è il simbolo iniziale della pila.
7.  $F \subseteq Q$  è l'insieme degli stati finali.

### Configurazione istantanea

La configurazione istantanea della macchina  $M$  è data dalla tripla:

$$(q, \gamma, \eta) \in Q \times \Sigma^* \times \Gamma^*$$

Dove:

1.  $q$  è lo stato in cui  $M$  si trova in quel particolare istante.
2.  $\gamma$  è l'insieme dei simboli in ingresso ancora da leggere.
3.  $\eta$  è la stringa contenuta nella pila.

### Transizione

Una transizione del tipo:

$$(q_1, \gamma_1, \eta_1) \rightarrow (q_2, \gamma_2, \eta_2)$$

può avvenire in 2 casi:

1. Mossa con lettura

In questo caso, chiamando  $a$  il simbolo che viene letto, si deve avere:

$$\gamma_1 = a\gamma_2 \quad \eta_1 = \eta Z \quad \eta_2 = \eta \alpha \quad (q_2, \alpha) \in \delta(q_1, a, Z)$$

2. Mossa spontanea

In questo caso, si deve avere:

$$\gamma_1 = \gamma_2 \text{ non ho letto nulla} \quad \eta_1 = \eta Z \quad \eta_2 = \eta \alpha \quad (q_2, \alpha) \in \delta(q_1, \epsilon, Z)$$

### Riconoscimento

Una stringa  $x$  viene riconosciuta dall'automa a pila  $M$  se e solo se è definita una sequenza di transizioni dalla configurazione iniziale  $(q_0, x, Z_0)$  ad una configurazione finale, ovvero:

$$(q_0, x, Z_0) \rightarrow^* (q, \epsilon, \lambda)$$

Dove  $q \in F$ . In altri termini, è **sufficiente che si legga tutto l'ingresso e che ci si ritrovi in uno stato finale** (non viene posta **alcuna condizione** riguardante la **pila**, che può essere in uno stato qualsiasi).

Se si modificasse la condizione di terminazione in uno dei due modi seguenti:

1. Si riconosce la stringa quando la pila è vuota ed è stato letto tutto l'ingresso.
  2. Si riconosce la stringa quando la pila è vuota, si è in uno stato finale ed è stato letto tutto l'ingresso.
- la capacità di riconoscimento dei linguaggi resterebbe immutata.



## Proprietà e osservazioni

### Linguaggi riconosciuti

CF

La famiglia dei linguaggi riconosciuti dagli automi a pila è la famiglia *LIB*: la **potenza espressiva degli automi a pila è perciò equivalente a quella delle grammatiche libere**.

### Automa senza cicli spontanei

Se un automa a pila ha delle mosse spontanee, è possibile che si determinino dei cicli, nei quali l'automa continua per tempo indefinito ad eseguire mosse senza mai leggere caratteri di ingresso; è perciò opportuno eliminarli. In particolare, dato un qualsiasi automa a pila, è sempre possibile costruire un automa a pila privo di cicli spontanei ad esso equivalente.

### Automa in linea

Un automa a pila funziona in linea se è in grado di decidere se accettare oppure no la stringa in ingresso immediatamente dopo la lettura del suo ultimo carattere, senza cioè compiere alcuna mossa spontanea aggiuntiva dopo la lettura dell'ultimo carattere della stringa d'ingresso.

## Dalla grammatica all'automa a pila

Data una grammatica libera  $G$ , è possibile costruire un automa a pila che riconosca il linguaggio  $L(G)$  nel modo seguente:

1. L'alfabeto d'ingresso è l'insieme dei terminali di  $G$  e l'alfabeto della pila è dato dall'unione dei terminali e dei non terminali di  $G$ , con l'aggiunta del simbolo di inizio pila,  $Z_0$ . stack used as a list of future actions.
2. L'automa considerato ha due stati: uno stato iniziale  $q_0$  ed uno stato finale  $q_1$ .
3. Dallo stato finale non è definita alcuna mossa, e vi si arriva solamente con una  $\varepsilon$ -mossa dallo stato iniziale  $q_0$  in corrispondenza della situazione in cui sulla pila si abbia solo il simbolo  $Z_0$ :

$$(q_1, Z_0) \in \delta(q_0, \varepsilon, Z_0)$$

4. Si aggiunge poi una mossa per ogni carattere  $b$  dell'alfabeto d'ingresso, del tipo:

$$(q_0, \varepsilon) \in \delta(q_0, b, b)$$

Cioè se la testina del nastro in ingresso legge il carattere  $b$ , che è anche il carattere in cima alla pila, quest'ultimo viene cancellato dalla pila e la testina del nastro d'ingresso viene portata avanti di una posizione.

5. Tutte le altre mosse vengono definite sulla base delle regole della grammatica; in particolare, se indichiamo con  $A, B, A_1, \dots, A_n$  dei simboli non terminali e con  $b$  un generico terminale:

- a) Per ogni regola del tipo:

$$A \rightarrow BA_1 \dots A_n, \quad n \geq 0$$

Si aggiunge una mossa del tipo:

$$(q_0, A_n \dots A_1 B) \in \delta(q_0, \varepsilon, A)$$

- b) Per ogni regola del tipo:

$$A \rightarrow bA_1 \dots A_n, \quad n \geq 0$$

Si aggiunge una mossa del tipo:

$$(q_0, A_n \dots A_1) \in \delta(q_0, b, A)$$

- c) Per ogni regola del tipo:

$$A \rightarrow \varepsilon$$

Si aggiunge una mossa del tipo:

$$(q_0, \varepsilon) \in \delta(q_0, \varepsilon, A)$$

"daisy automaton"

L'automa che si ottiene è anche detto "a margherita", perché è costituito da uno stato principale, con tutta una serie di transizioni verso sé stesso, che appaiono appunto come i petali di una margherita.

## Linguaggi deterministici

### Linguaggi deterministici

La famiglia degli automi a pila deterministici ha potenza di riconoscimento inferiore rispetto alla famiglia degli automi a pila non deterministici; la famiglia dei linguaggi riconosciuti dagli automi a pila deterministici è detta *dei linguaggi deterministici* e viene indicata con *DET*. La famiglia *DET* quindi non coincide con la famiglia *LIB*.

N.B. Per ragioni di efficienza, nel design di un compilatore vengono considerati solo i linguaggi deterministici, quindi quelli accettati da un automa a pila deterministico

### Ambiguità

Dato un automa a pila deterministico, la grammatica che si ottiene a partire da esso non è ambigua.

### Chiusura dei linguaggi deterministici

I linguaggi deterministici sono chiusi rispetto alle operazioni seguenti:

1. Complemento
2. Unione con un linguaggio regolare (cioè l'unione tra un linguaggio regolare ed uno deterministico è un linguaggio deterministico)
3. Concatenamento con un linguaggio deterministico (cioè il concatenamento tra un linguaggio regolare ed uno deterministico è un linguaggio deterministico)
4. Intersezione

Non sono invece chiusi rispetto a:

1. Riflesso
2. Stella
3. Unione tra linguaggi deterministici
4. Concatenazione tra linguaggi deterministici

N.B. Linguaggi deterministici possono avere "spontaneous moves", infatti valgono le seguenti:

- Se è definita una mossa che comprende una lettura di un carattere d'ingresso, allora la relativa mossa spontanea non è definita.
- Se è definita una mossa spontanea allora non è definita la mossa con la lettura di un carattere.

Con tali accorgimenti si esclude non determinismo legato a mosse spontanee

## Automa a pila riconoscitore dell'intersezione

### Introduzione

Sappiamo che i linguaggi liberi non sono chiusi rispetto all'intersezione. Sappiamo però che l'intersezione tra un linguaggio libero ed un linguaggio regolare è un linguaggio libero. Possiamo allora introdurre un modo per determinare l'automa a pila  $A_3$  che riconosca l'intersezione tra il linguaggio generato da un automa a stati finiti  $AF$  e un automa a pila  $AP$ .

### Procedimento

Si esegue lo stesso procedimento che si eseguirebbe se le macchine in questione fossero due automi a stati finiti, ma in più si aggiungono tutte le operazioni sulla pila che vengono compiute dall'automa a pila  $AP$ .

L'automa ottenuto riconosce le stringhe con condizione congiunta (ci si deve cioè trovare in uno stato finale con pila vuota). Volendolo trasformare in un automa che riconosce solo a stato finale, è sufficiente aggiungere un nuovo stato, finale, rendendo non finali tutti gli altri, al quale si arrivi solo con delle  $\epsilon$ -mosse compiute dagli stati che in precedenza erano finali, in corrispondenza della situazione in cui in cima alla pila si abbia il simbolo  $Z_0$  (cioè la pila è vuota).

La macchina costruita:

- ha gli stati interni che sono il prodotto degli stati delle macchine che la compongono
- stati finali sono quelli che includono lo stato finale dell'automa a pila
- E' deterministico se lo sono gli automi di partenza
- Accetta esattamente le stringhe appartenenti all'intersezione dei due linguaggi

# Analisi sintattica discendente LL(k)

## L'analisi sintattica discendente e ascendente

Pacco: "9 - Pushdown automata and Context free language parsing.pdf"

### Il concetto di analisi sintattica

L'analisi sintattica è quel procedimento che consente di ottenere, a partire da una certa grammatica  $G$ , l'albero sintattico di una stringa appartenente a  $L(G)$ . Lo strumento utilizzato per eseguire l'analisi sintattica è detto *analizzatore sintattico* o *parsificatore*. (parser)

Sono possibili alcuni comportamenti particolari:

1. Se la stringa fornita al parsificatore non appartiene a  $L(G)$ , allora il parsificatore dovrà essere in grado di segnalarlo.
2. Se la stringa fornita al parsificatore è ambigua, quest'ultimo produrrà non un albero sintattico, ma un insieme di possibili alberi sintattici.

### L'analisi sintattica discendente top-down analysis

L'analisi sintattica si dice *discendente* se costruisce una derivazione sinistra della stringa data, procedendo dall'assioma verso le foglie.

In sostanza quindi si parte dall'assioma, e di volta in volta si sostituisce ad un non terminale  $A$  presente nella derivazione la parte destra di una regola che ha come parte sinistra il non terminale stesso  $A$ .

L'analisi sintattica termina quando si ottiene una stringa costituita solamente da simboli terminali.

Esempio slide 21 del pacco

### L'analisi sintattica ascendente bottom-up analysis

L'analisi sintattica si dice *ascendente* se costruisce una derivazione destra della stringa data, procedendo dalle foglie alla radice dell'albero, mediante una serie di riduzioni.

Questo significa che si analizza la stringa finale e si riduce di volta in volta una sottostringa  $a_1 \dots a_n$  di tale stringa in un solo non terminale  $A$ , a patto che esista la regola del tipo:

$$A \rightarrow a_1 \dots a_n$$

Il processo termina quando l'intera stringa si riduce all'assioma.

Esempio slide 22 del pacco sopracitato  
Slide 23 del pacco sopracitato presenta altre informazioni su questo tipo di analisi

## Premessa per l'analisi sintattica LL(k): grammatica come rete di AF

### Terminologia e simbologia

Sia data una grammatica  $G$  *estesa*, nella quale cioè ogni non terminale è parte sinistra di una e una sola regola, e la parte destra è un'espressione regolare sull'alfabeto dei terminali e non terminali della grammatica. Chiamiamo:

#### - Alfabeto terminale

L'insieme dei simboli dell'alfabeto  $\Sigma$  sul quale è definito  $L(G)$ .

#### - Alfabeto non terminale

L'insieme dei simboli di non terminali della grammatica  $G$ , ovvero  $V$ .

#### - Linguaggi regolari

Indichiamo con  $R_S, R_A, R_B, \dots$  i linguaggi regolari definiti dalle espressioni regolari  $\sigma, \alpha, \beta, \dots$ , dove:

$$S \rightarrow \sigma$$

$$A \rightarrow \alpha$$

$$B \rightarrow \beta$$

Questi linguaggi sono definiti sull'alfabeto  $\Sigma \cup V$ . (alfabeti su cui è definita la grammatica  $G$ )

#### - Macchine

Chiamiamo *macchine*  $M_S, M_A, M_B, \dots$ , gli automi finiti deterministici che riconoscono i linguaggi  $R_S, R_A, R_B, \dots$ .

#### - Rete di macchine

Chiamiamo *rete di macchine* l'insieme delle macchine  $M_S, M_A, M_B, \dots$ , ovvero:

$$\mathcal{M} = \{M_S, M_A, M_B, \dots\}$$

SLIDE 25: Ulteriore limitazioni per macchine corrispondenti a simboli non terminali: Lo stato iniziale  $q_A$  non dev'essere visitato dopo l'inizio della computazione. Nel caso questo requisito non sia soddisfatto, è possibile modificare la macchina aggiungendo uno stato (che sarà il nuovo stato iniziale)

e delle transizioni da esso. La nuova macchina non sarà minima ma avremo solo aggiunto un singolo stato. Gli automi che soddisfano tale requisito sono detti "normalized" or con stato iniziale "non recirculativo" or "non reentrant"

### Esempio di costruzione della rete di macchine

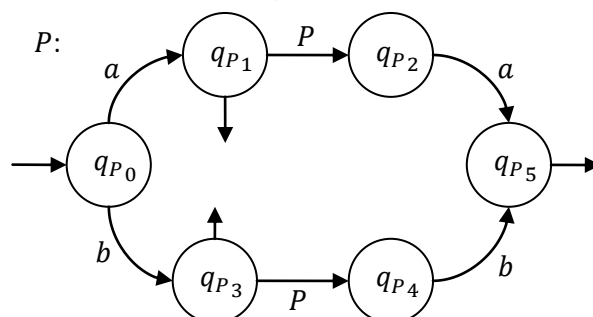
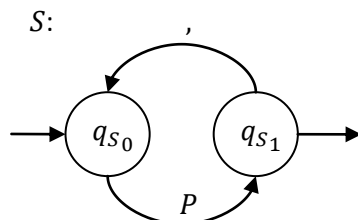
Il procedimento per la costruzione della rete di macchine è molto intuitivo. Ad esempio, data la grammatica:

$$S \rightarrow P(, P)^*$$

$$P \rightarrow aPa|bPb|a|b$$

La rete di macchine che si ottiene è la seguente:

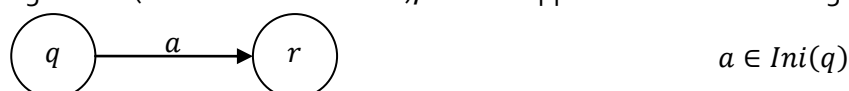
N.B. Questa macchina non è "normalized"



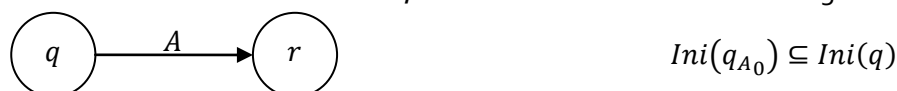
### Calcolo degli inizi di uno stato di una macchina della rete

Introduciamo adesso un procedimento che ci tornerà molto utile a breve: quello per il calcolo degli inizi di un certo stato  $q$  di una delle macchine della rete. Tale insieme viene indicato con  $Ini(q)$  e non è altro che l'insieme degli inizi del linguaggio  $L(q)$ ; esso viene calcolato sulla base delle seguenti regole:

1. Se nella macchina è definita una transizione dallo stato  $q$  ad un altro stato  $r$ , in corrispondenza del simbolo d'ingresso  $a$  (dove  $a$  è un terminale), allora  $a$  appartiene all'insieme degli inizi di  $q$ :



2. Se nella macchina è definita una transizione dallo stato  $q$  ad un altro stato  $r$ , in corrispondenza del simbolo  $A$  (dove  $A$  è un non terminale di  $G$ ), l'insieme degli inizi di  $q_{A_0}$ , ovvero dello stato iniziale della macchina associata al non terminale  $A$ , è un sottoinsieme dell'insieme degli inizi di  $q$ .



3. Se si è nella situazione descritta al punto 2 e inoltre  $L(q_{A_0})$  è un linguaggio annullabile, allora:

$$Ini(r) \subseteq Ini(q)$$

### Calcolo dei seguiti di simbolo non terminale della grammatica

Un altro procedimento simile al precedente e molto utile è quello per il calcolo dei seguiti di un certo stato simbolo non terminale  $A \in V$  della grammatica,  $Seg(A)$ , che rappresenta l'insieme dei termini che possono seguire  $A$  in una certa derivazione. Le regole da seguire sono le seguenti:

1. L'insieme dei seguiti dell'assioma  $S$  contiene sempre un simbolo di fine della stringa:  
 $\# \in Seg(S)$
2. Se in una macchina è definita una transizione dallo stato  $q$  ad un altro stato  $r$ , in corrispondenza del simbolo  $A$ , l'insieme dei seguiti di  $A$  ha come sottoinsieme l'insieme degli inizi di  $r$ .



3. Se nella macchina  $M_B$  (con  $B \neq A$ ) è definita una transizione del tipo descritto nel punto 2, e inoltre  $L(r)$  è annullabile, allora l'insieme dei seguiti di  $A$  contiene anche tutti i seguiti di  $B$ :

$$Seg(B) \subseteq Seg(A)$$

4. Se nella macchina  $M_B$  (con  $B \neq A$ ) è definita una transizione del tipo descritto nel punto 2, e inoltre  $r$  è uno stato finale di  $M_B$ , allora l'insieme dei seguiti di  $A$  contiene anche tutti i seguiti di  $B$ :

$$Seg(B) \subseteq Seg(A)$$

### Calcolo dell'insieme guida di ogni ramo di un automa della rete di macchine

È possibile calcolare per ogni ramo di ogni singolo AF della rete di macchine un insieme, detto *insieme guida*, che indica l'insieme dei caratteri che è possibile incontrare seguendo quel particolare ramo. In particolare, si indica con  $Gui_k(q \rightarrow_a r)$  l'insieme delle possibili sequenze di  $k$  caratteri dell'alfabeto di ingresso che possono essere seguite letti eseguendo la mossa  $q \rightarrow_a r$ .

Si noti che  $a$  può anche essere un simbolo non terminale, e che la mossa relativamente alla quale si calcola l'insieme guida può in realtà anche essere l'uscita da una delle macchine dell'automa: in altre parole, si ha un insieme guida associato anche alla freccia:

$$q \rightarrow$$

L'insieme guida con  $k = 1$  si calcola seguendo le regole seguenti:

1. Se  $b$  è un terminale:

$$Gui_1(q \rightarrow_b r) = \{b\}$$

2. Se  $A$  è un non terminale e il linguaggio  $L(q_{A_0})L(r)$  non è annullabile, allora:

$$Gui_1(q \rightarrow_A r) = Ini(L(q_{A_0})L(r))$$

3. Se  $A$  è un non terminale e il linguaggio  $L(q_{A_0})L(r)$  non è annullabile, allora, detta  $M_B$  la macchina in cui è definita la transizione  $q \rightarrow_A r$ , si ha:

$$Gui_1(q \rightarrow_A r) = Ini(L(q_{A_0})L(r)) \cup Seg(B)$$

4. Se  $q$  è lo stato finale di  $M_B$ , allora:

$$Gui_1(q \rightarrow) = Seg(B)$$

## Analisi sintattica discendente deterministica LL(1)

### I parsificatori LL

Un parsificatore LL è un parsificatore top-down, che analizza la stringa in input da sinistra a destra (da qui deriva la prima L, che sta per *Left to right*) e che costruisce una derivazione sinistra della stringa stessa (*Leftmost derivation*, da qui deriva la seconda L), operando sempre in modo deterministico.

I parsificatori LL sono in grado di eseguire l'analisi sintattica solamente di un sottoinsieme delle grammatiche libere; le grammatiche di tale sottoinsieme vengono appunto chiamate *grammatiche LL*.

Più nel dettaglio, si parla di *parsificatore LL(k)* quando il parsificatore, per eseguire l'analisi sintattica di una frase della grammatica, necessita di leggere  $k$  simboli della stringa in ingresso successivi rispetto a quello attualmente in analisi, prima di prendere la decisione relativa all'operazione da compiere. Nel dettaglio, e  $k = 0$  (cosa in realtà impossibile nell'analisi LL), la decisione viene presa solo sulla base dello stato attuale; se  $k = 1$  si considera anche il simbolo sotto la testina del nastro in ingresso, e così via.

Un linguaggio la cui analisi sintattica possa essere eseguita da un parsificatore  $LL(k)$  è detto semplicemente *linguaggio LL(k)*. Inoltre,  $k$  è detta anche *lunghezza della parsificazione*.

### Condizione sufficiente perché un linguaggio sia LL(1)

Una grammatica libera è del tipo  $LL(1)$  se, per ogni coppia di frecce uscenti da uno stesso stato di ciascuna delle macchine che costituiscono la rete ad essa associata, gli insiemi guida sono disgiunti.

In particolare, se in uno stato tutte le frecce uscenti hanno insiemi guida disgiunti, si dice che quello stato soddisfa la *condizione LL(1)*; se tutti gli stati di una rete di macchine soddisfano la condizione  $LL(1)$ , allora la grammatica alla quale è associata tale rete di macchine è una grammatica  $LL(1)$ .

### Proprietà

1. Una grammatica che contiene almeno una regola ricorsiva a sinistra non sarà mai di tipo  $LL(1)$ .
2. Tutti i linguaggi regolari ammettono una grammatica  $LL(1)$ .

### Descrizione dell'automa a pila in grado di eseguire l'analisi sintattica LL(1)

Se una grammatica  $G$  è LL(1), allora si può costruire un automa a pila deterministico che ne esegua l'analisi sintattica. Per introdurre tale automa a pila, chiamiamo  $x$  la stringa sorgente (quella fornita in input) e indichiamo con  $cc$  il carattere corrente (cioè quello posto sotto la testina del nastro in ingresso).

1. L'automa a pila costruito ha come alfabeto della pila l'alfabeto definito come l'unione degli stati di tutte le macchine che costituiscono la rete associata alla grammatica  $G$  data:

$$\Gamma = Q_A \cup Q_B \cup \dots$$

Il simbolo in cima alla pila rappresenta in sostanza lo stato nel quale ci si trova in quell'istante.

2. Inizialmente la pila contiene lo stato iniziale della macchina associata all'assioma, ovvero  $q_{S_0}$ .
3. L'automa a pila ha un solo stato interno.
4. Detta  $M_A$  la macchina attiva nell'istante considerato e detto  $s$  il simbolo in cima alla pila (cioè lo stato nel quale ci si trova nella rete di macchine), con  $s \in Q_A$ , sono definite le mosse seguenti:

a) *Mossa di scansione*

Se nella macchina  $M_A$  è definita la mossa  $\delta(s, cc) = s'$ , allora l'automa a pila consuma il carattere corrente  $cc$ , elimina dalla pila il simbolo  $s$  e al suo posto inserisce il simbolo  $s'$ .

b) *Mossa di chiamate*

Se nella macchina  $M_A$  è definita la mossa  $\delta(s, B) = s'$ , e  $cc \in Gui(s \rightarrow_B s')$ , allora l'automa a pila effettua una mossa spontanea, nella quale si sostituisce il simbolo in cima alla pila, inserendo  $q_{B_0}$  al posto di  $s$ , e non si procede nella lettura sul nastro di ingresso. In questo modo, la macchina attiva diventa  $M_B$ .

c) *Mossa di ritorno*

Se  $s$  è uno stato finale della macchina  $M_A$  in cui ci si trova e  $cc \in Gui(s \rightarrow)$ , allora l'automa a pila compie una mossa spontanea che cancella  $s$  dalla pila; detto  $r$  il nuovo simbolo in cima alla pila, si esegue la mossa spontanea verso lo stato  $s'$  tale che  $\delta(r, B) = s'$  e si sostituisce il simbolo in cima alla pila, cancellando  $s'$  e cambiandolo con  $r$ .

d) *Mossa di riconoscimento*

Se  $s$  è uno stato finale della macchina  $M_S$  associata all'assioma  $S$  e inoltre il carattere corrente è il terminatore della stringa:

$$c = \text{--}$$

Allora l'automa a pila accetta la stringa d'ingresso e termina l'analisi sintattica.

5. Nel caso in cui non possa essere eseguita nessuna delle mosse appena elencate, la stringa viene rifiutata ed il procedimento termina; altrimenti, si ripetono le mosse fino ad arrivare al riconoscimento.

### Allungamento della prospezione

È possibile aumentare la prospezione; occorre semplicemente sostituire all'insieme  $Gui_1$  l'insieme  $Gui_k$  ed estendere in modo ovvio (anche se non banale da un punto di vista implementativo) la procedura appena descritta, in modo che si proceda nella lettura dei caratteri successivi qualora non si possa decidere quale mossa compiere tra un insieme di mosse tutte possibili.

## Analisi sintattica ascendente LR(k)

### Analisi sintattica ascendente deterministica

#### I parsificatori LR

Un parsificatore LR è un parsificatore che analizza la stringa in input da sinistra a destra (da qui deriva la L, che sta per *Left to right*) e che costruisce una derivazione destra della stringa stessa (*Rightmost derivation*, da qui deriva la lettera R), operando sempre in modo deterministico.

I parsificatori LR sono in grado di eseguire l'analisi sintattica solamente di un sottoinsieme delle grammatiche libere; le grammatiche di tale sottoinsieme vengono appunto chiamate *grammatiche LR*.

Più nel dettaglio, si parla di *parsificatore LR(k)* quando il parsificatore, per eseguire l'analisi sintattica di una frase della grammatica, necessita di leggere  $k$  simboli della stringa in ingresso successivi rispetto a quello attualmente in analisi, prima di prendere la decisione relativa all'operazione da compiere.

Un linguaggio la cui analisi sintattica possa essere eseguita da un parsificatore LR(k) è detto semplicemente *linguaggio LR(k)*. Inoltre,  $k$  è detta anche *lunghezza della parsificazione*.

#### Analisi a spostamento e riduzione

Gli automi che analizzeremo per l'esecuzione dell'analisi sintattica LR sono dei particolari automi a pila deterministici, detti *a spostamento e riduzione*. Il principio di base prevede che l'automa esegua due tipi di operazioni:

1. Spostamento

L'automa legge la stringa di caratteri in ingresso da sinistra a destra, impilando i caratteri in cima alla pila, uno alla volta.

2. Riduzione

Quando l'automa riconosce in cima alla pila la parte destra di una regola (che viene detta *parte riducibile*), esegue l'operazione di *riduzione*, ovvero sostituisce la parte riducibile con la parte sinistra della corrispondente regola.

Il processo termina quando sulla pila è rimasto solamente il simbolo associato all'assioma della grammatica.

## Concetti preliminari per l'analisi LL(o): la macchina pilota

### Macchina pilota

Per eseguire l'analisi sintattica LR(o) è necessario innanzitutto costruire la rete di macchine associata alla grammatica, che può opzionalmente essere modificata aggiungendo un nuovo assioma  $S_0$ , nel modo seguente:

$$S_0 \rightarrow S \dashv$$

Questa modifica può essere eseguita anche nel caso dell'analisi LL(k). Il procedimento da seguire per costruire la rete di macchine è lo stesso illustrato per l'analisi LL; dopodiché, si deve costruire la macchina pilota, ovvero un automa a stati finiti, costruito sulla base della rete di macchine associata alla grammatica estesa di partenza, il quale sarà in grado di pilotare il parsificatore, consentendo il riconoscimento dei prefissi ascendenti validi.

In particolare, la macchina pilota è caratterizzata dal fatto che un suo stato è un insieme di stati degli automi finiti che costituiscono la rete di macchine di partenza (e perciò ogni stato è in realtà un macrostato).

### Termini relativi alla macchina pilota

- Stato di riduzione  
Uno stato di riduzione è uno stato finale di uno degli automi finiti della rete di macchine di partenza, appartenente ad un macrostato della macchina pilota.
- Stato di spostamento  
Uno stato di spostamento è uno stato non finale di uno degli automi finiti della rete di macchine di partenza, appartenente ad un macrostato della macchina pilota.
- Riduzione  
Una riduzione è uno stato della macchina pilota che contiene solo stati di riduzione.
- Spostamento  
Uno spostamento è uno stato della macchina pilota che contiene solo stati di spostamento.
- Macrostato misto  
Un macrostato misto è uno stato della macchina pilota che contiene sia stati di spostamento, sia stati di riduzione.

### Condizione LR(o)

Una grammatica è LR(o) se la macchina pilota ad essa associata non contiene stati misti, ed ogni suo stato non contiene più di uno stato di riduzione.

Se una grammatica è LR(o), allora dalla macchina pilota si ottiene l'automa a pila deterministico che riconosce le frasi di  $L(G)$  e ne costruisce gli alberi sintattici.

### Chiusura LR(o) di uno stato

Per poter costruire la macchina pilota è necessario eseguire un'operazione di chiusura, che andiamo ora ad illustrare.

La chiusura LR(o) di uno stato di uno degli automi a stati finiti della rete di macchine di partenza è definita mediante il seguente procedimento, nel quale lo stato del quale si calcola la chiusura è indicato con  $q$  e la chiusura LR(o) stessa viene indicata con  $C$ :

1. Si inizializza l'insieme  $C$  in modo che contenga solamente  $q$ , ovvero:  $C = \{q\}$ .
2. Per ognuno degli stati appartenenti a  $C$  si verifica se ci sono delle frecce uscenti etichettate con dei simboli non terminali: detto  $A$  il non terminale che etichetta tale freccia, si aggiunge a  $C$  lo stato iniziale della macchina  $M_A$  associata al non terminale  $A$ .
3. Si ripete il punto 2 fino al raggiungimento di un punto fisso.



### Chiusura LR(o) di un insieme di stati

La chiusura LR(o) di un insieme di stati è semplicemente l'unione delle chiusure LR(o) degli stati che costituiscono l'insieme di partenza:

$$\text{chius}(\{q_1, q_2, \dots, q_k\}) = \bigcup_{i=1}^k \text{chius}(q_i)$$

### Costruzione dell'automa pilota a partire dalla rete di macchine

L'automa pilota è definito dalla tupla:

$$N = (R, \Sigma \cup V, \Theta, I_0, R)$$

Dove:

1. Lo stato iniziale  $I_0$  dell'automa pilota è la chiusura dello stato iniziale  $q_{S_0}$  della macchina  $M_{S_0}$  associata all'assioma  $S_0$  della grammatica estesa  $G$  di partenza.
2. L'insieme dei simboli in ingresso è  $\Sigma_p = \Sigma \cup V$ , dove si intende compreso anche il carattere terminazione della stringa,  $\dashv$ .
3. L'insieme degli stati  $R$  e la funzione di transizione vengono calcolati iterativamente e in maniera congiunta, mediante il procedimento seguente:

- 3.1. Si inizializza l'insieme degli stati  $R$  in modo che contenga solo lo stato iniziale  $I_0$ :

$$R = \{I_0\}$$

- 3.2. Per ogni macrostato  $I$  contenuto in  $R$ , si considerano tutti i simboli  $X$  dell'alfabeto d'ingresso  $\Sigma_p$  e per ciascuno di essi:

- 3.2.1. Si considerano tutti gli stati  $q$  appartenenti ad  $I$  e per ciascuno di essi si guarda qual è lo stato  $q'$  al quale si arriva eseguendo la mossa contrassegnata dall'etichetta  $X$ , dopodiché si calcola la chiusura di  $q'$  e si esegue l'unione di tutte le chiusure ottenute; il risultato sarà il macrostato nel quale l'automa pilota si porta partendo da  $I$  con ingresso  $X$ :

$$\Theta(I, X) = \bigcup_{q \in I} \text{chius}(\delta(q, X))$$

- 3.2.2. Si aggiunge lo stato così ottenuto all'insieme degli stati dell'automa:

$$R = R \cup \{\Theta(I, X)\}$$

- 3.3. Si ripete il punto 3.2 fino al raggiungimento di un punto fisso.

4. Tutti gli stati di  $R$  sono anche finali:

$$R = F$$

### Rappresentazione alternativa dell'automa pilota

L'automa pilota, oltre che con la consueta notazione, può essere rappresentato associando agli stati delle etichette diverse, che mettono in evidenza le regole alle quali si fa riferimento. In sostanza quindi per ogni macrostato si vanno a vedere quali sono gli stati degli automi finiti della rete di macchina che costituiscono il macrostato stesso e, per ciascuno di essi, si individuano i vari cammini che portano allo stato finale della macchina in cui si trovano: si scrive poi la regola corrispondente a tale cammino, marcandola con un simbolo  $\bullet$  che precede il prossimo simbolo da leggere.

### Proprietà della macchina pilota

La macchina pilota deve avere sempre le seguenti proprietà:

1. Tutte le frecce entrati in un suo macrostato portano la stessa etichetta;
2. Ogni macrostato di riduzione non ha successori;
3. Ogni macrostato di spostamento ha almeno un successore.

### **Dalla macchina pilota all'automa a pila**

Il comportamento dell'automa a pila viene direttamente ricavato dalla macchina pilota così costruita, seguendo le seguenti regole:

1. I simboli che il nuovo automa può memorizzare sulla pila sono i macrostati della macchina pilota. Ad essi possono essere eventualmente aggiunti, con il solo scopo di migliorare la leggibilità, anche i simboli della grammatica scanditi per raggiungere gli stati stessi della macchina pilota.
2. Inizialmente, la pila contiene solamente il simbolo corrispondente al macrostato iniziale dall'automa pilota.
3. L'automa esamina un carattere alla volta della stringa in ingresso ed esegue la mossa che verrebbe compiuta nell'automa pilota ricevendo il carattere letto a partire dallo stato presente in cima alla pila. Tale operazione è detta di *spostamento*.
4. Dopo uno spostamento, se si giunge ad uno stato di riduzione (e, in base alla nostra ipotesi iniziale, uno stato di riduzione non può contenere più di una riduzione), allora si esegue una serie di operazioni dette di riduzione. Tali operazioni consistono nel togliere dalla pila tutti i simboli (e i macrostati corrispondenti) che costituiscono la parte destra della regola alla quale è relativa la riduzione; al termine, si inseriscono in cima alla pila il simbolo corrispondente alla parte sinistra della regola di riduzione e lo stato al quale si arriva nell'automa pilota se si parte dallo stato che è rimasto in cima alla pila, ricevendo come simbolo in ingresso il simbolo presente nella parte sinistra della regola analizzata.
5. Al termine della scansione, l'automa accetta la stringa sorgente se la pila è vuota o contiene il solo macrostato iniziale (operazione di *accettazione/rifiuto*).

### **Limiti dell'analisi LR(o)**

1. Un'importante limitazione dell'analisi LR(o) è che i linguaggi analizzati devono necessariamente essere privi di prefissi, cioè se una stringa appartiene al linguaggio, tutti i suoi prefissi non devono appartenere al linguaggio stesso.
2. Inoltre, se la grammatica contiene una regola vuota con anche un'alternativa non vuota, la grammatica viola la condizione LR(o).

## **Analisi LR(k)**

### **Introduzione della prospezione**

Per potenziare il metodo LR(o) trasformandolo nel metodo LR(1), si aggiunge ad ogni stato l'informazione sulla prospezione. In particolare, siccome ogni macrostato contiene in generale più stati, bisognerà calcolare l'insieme dei caratteri terminali di prospezione per ognuno degli stati appartenenti al macrostato stesso.

Intuitivamente, un macrostato è adeguato per l'analisi LR(1) se la prossima mossa dell'automa a pila è determinata in modo univoco dal macrostato posto in cima alla pila e dal carattere corrente letto dal nastro in ingresso, insieme agli insiemi di prospezione.

Un carattere di prospezione indica il simbolo che ci sarà in ingresso quando verrà eseguita la riduzione della parte destra della regola nella non terminale parte sinistra, cioè quando il simbolo • che marca la parte destra della regola sarà arrivato alla fine.

## Chiusura delle candidate

Nel caso LR(1) l'automa pilota è definito dalla tupla:

$$N = (R, \Sigma \cup V, \theta, I_0, R) \quad (\text{stessa dell'automa a pila per caso LR(0)})$$

Dove però, a differenza di quanto accadeva nel caso LR(0), ogni macrostato è un insieme di candidate, ovvero un insieme di coppie costituite da uno stato e da un simbolo terminale (o il terminatore della stringa):

$$(q, a) \in Q \times (\Sigma \cup \{-\})$$

A questo scopo, dobbiamo innanzitutto definire la chiusura di una candidata: la chiusura LR(1) di una candidata  $(q, a)$ , dove  $q$  è uno degli stati di uno degli automi a stati finiti della rete di macchine di partenza, mentre  $a$  è un simbolo terminale (o il simbolo  $-$ ), indicata con  $chius_1(< q, a >)$ , è definita nel modo seguente:

1. Si inizializza la chiusura  $C$  con la candidata  $< q, a >$  stessa:

$$C = \{< q, a >\}$$

2. Si considerano tutte le candidate  $< p, a >$  di  $C$  e si prendono in analisi tutti gli archi uscenti dagli stati  $p$  di tali candidate ed etichettati con un qualsiasi simbolo non terminale  $A$ ; detto  $p'$  lo stato di arrivo di tale mossa:

- a) Se  $L(p')$  non è annullabile, si aggiunge a  $C$  la candidata:

$$< q_{A_0}, Ini(L(p')) >$$

Se la parola  $L(p)L(p)$  (cioè la stringa che può essere derivata dallo stato  $pp$ ) non è annullabile (ovvero non può produrre la stringa vuota)

- b) Se  $L(p')$  è annullabile, si aggiunge a  $C$  la candidata:

$$< q_{A_0}, Ini(L(p')) \cup \{a\} >$$

3. Si ripete il passaggio descritto al punto 2 fino al raggiungimento del punto fisso.

Se si deve calcolare la chiusura di un insieme di candidate, si esegue semplicemente l'unione delle chiusure delle singole candidate che costituiscono l'insieme di partenza.

Si noti che l'insieme degli inizi di uno stato finale (di qualsiasi automa della rete) è sempre dato dall'insieme vuoto.

## Come costruire la l'automa pilota a partire dalla rete di macchine nel caso LR(1)

Nell'automa pilota:

1. Lo stato iniziale  $I_0$  dell'automa pilota è la chiusura della candidata avente come stato, lo stato iniziale  $\alpha$  della macchina  $M_{S_0}$  associata all'assioma  $S_0$  della grammatica estesa  $G$  di partenza, e come prospezione il simbolo terminatore  $-$ :

$$I_0 = chius_1(< q_0, - >)$$

2. L'insieme dei simboli in ingresso è  $\Sigma \cup V$ , dove si intende compreso anche il carattere terminazione della stringa,  $-$ .
3. L'insieme degli stati  $R$  e la funzione di transizione vengono calcolati iterativamente e in maniera congiunta, mediante il procedimento seguente:

- 3.1. Si inizializza l'insieme  $R$  in modo che contenga solo lo stato iniziale  $I_0$ .

- 3.2. Per ogni macrostato  $I$  già appartenente all'insieme  $R$ , e per ogni simbolo  $X$  appartenente all'alfabeto di ingresso dell'automa pilota, si considerano tutte le candidate appartenenti ad  $I$  e, detta  $< q, b >$  una generica candidata in  $I$ , si calcola la chiusura di  $< \delta(q, X), b >$ : l'unione di tutte le chiusure così ottenute costituirà un nuovo stato da inserire in  $R$ , e sarà lo stato di destinazione di una nuova mossa da  $I$  con ingresso  $X$ :

$$\theta(I, X) = \bigcup_{< q, b > \in I} chius(< \delta(q, X), b >)$$

$$R := R \cup \{\theta(I, X)\}$$

- 3.3. Si itera il punto 2 fino al raggiungimento di un punto fisso.

4. Tutti gli stati di  $R$  sono anche stati finali, perciò  $R = F$ .

### **Rappresentazione dell'automa pilota**

La macchina pilota viene indicata rappresentando gli stati mediante dei rettangoli; per ogni macrostato si indicano poi tutti gli stati che vi appartengono tra quelli dei vari automi della rete di macchine, indicando sia il numero dello stato, sia la relativa regola contrassegnata dal  $\bullet$ . I sottostati vengono separati da una linea orizzontale, che separa gli stati in base alla loro provenienza nel calcolo della funzione di chiusura. Inoltre, sulla destra, separati dallo stato per mezzo di una linea verticale, vengono indicati tutti i simboli che costituiscono l'insieme di prospezione associato a quel particolare stato (cioè anziché indicare le candidate  $\langle q, s_1 \rangle$  e  $\langle q, s_2 \rangle$  si indica l'insieme di prospezione  $\{s_1, s_2\}$  associato allo stato  $q$ ).

### **Terminologia**

- **Candidata di riduzione**

Candidata  $\langle q, s \rangle$  nella quale lo stato  $q$  (che è uno degli stati degli automi che costituiscono la rete di macchine) è uno stato finale;

- **Candidata di spostamento**

Candidata  $\langle q, s \rangle$  nella quale lo stato  $q$  non è uno stato finale;

### **Condizione LR(1)**

Una grammatica è LR(1) se, per ogni macrostato della macchina pilota, sono vere le due condizioni seguenti:

a) **Assenza di conflitti riduzione-spostamento**

Ogni candidata di riduzione ha un insieme di prospezione disgiunto dall'insieme dei terminali che etichettano gli archi uscenti dal macrostato stesso.

b) **Assenza di conflitti riduzione-riduzione** --> questo conflitto era invece possibile nell'automa pilota LR(0)

Se ci sono due candidate di riduzione, i loro insiemi di prospezione sono disgiunti.

### **Proprietà**

Ogni grammatica LL(1) è anche LR(1).

## **Analisi LALR(1)**

### **Procedimento costruttivo**

Una condizione intermedia tra la condizione LR(0) e la condizione LR(1) è la condizione LALR(1): in sostanza, si costruisce in questo caso una macchina pilota uguale a quella del caso LR(1), ma poi si fondono in uno stato unico tutti i macrostati della macchina pilota così ottenuta che differiscono tra loro solamente per gli insiemi di prospezione (cioè che nel caso LR(0) sarebbero indistinguibili). Gli insiemi di prospezione nella macchina pilota LALR(1) saranno quindi dati dall'unione degli insiemi di prospezione che si hanno nel caso LR(1).

### **Condizione LALR(1)**

La condizione da imporre è del tutto analoga a quella imposta nel caso LR(1), cioè devono valere le due seguenti condizioni:

- Per ogni stato, ogni candidata di riduzione ha un insieme di prospezione disgiunto dai simboli terminali che etichettano gli archi uscenti dallo stato stesso;
- Per ogni stato, se si hanno due candidate di riduzione, i loro insiemi di prospezione devono essere disgiunti.

## Esempio

### Testo

Costruire il riconoscitore deterministico dei prefissi ascendenti e indicare in quali stati sono violate le condizioni LR(1), LALR(1) e LR(0).

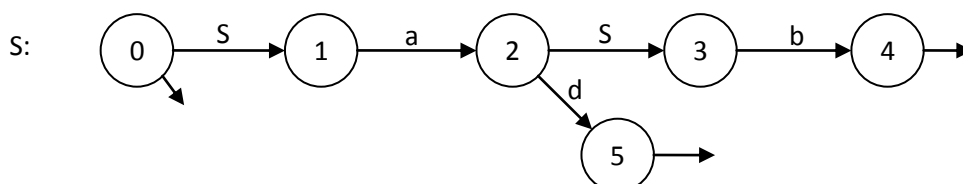
$$S \rightarrow SaSb$$

$$S \rightarrow Sad$$

$$S \rightarrow \varepsilon$$

### Analisi LR(0)

Proviamo prima di tutto a fare l'analisi LR(0). Costruiamo a tale scopo la rete di macchine, che in questo caso sarà costituita da una sola macchina, perché abbiamo solamente il non terminale S:



Lo stato iniziale della macchina pilota sarà:

$$I_0 = \text{chius}(0) = \{0\}$$

Eseguiamo qui "per esteso" tutti i calcoli per determinare le varie chiusure: nella realtà, la chiusura può anche essere calcolata direttamente partendo dai diagrammi, in maniera grafica più intuitiva e soprattutto molto più rapida.

$$\Theta(I_0, S) = \text{chius}(\delta(0, S)) = \text{chius}(1) = \{1\} = I_1$$

$$\Theta(I_1, a) = \text{chius}(\delta(1, a)) = \text{chius}(2) = \{2, 0\} = I_2$$

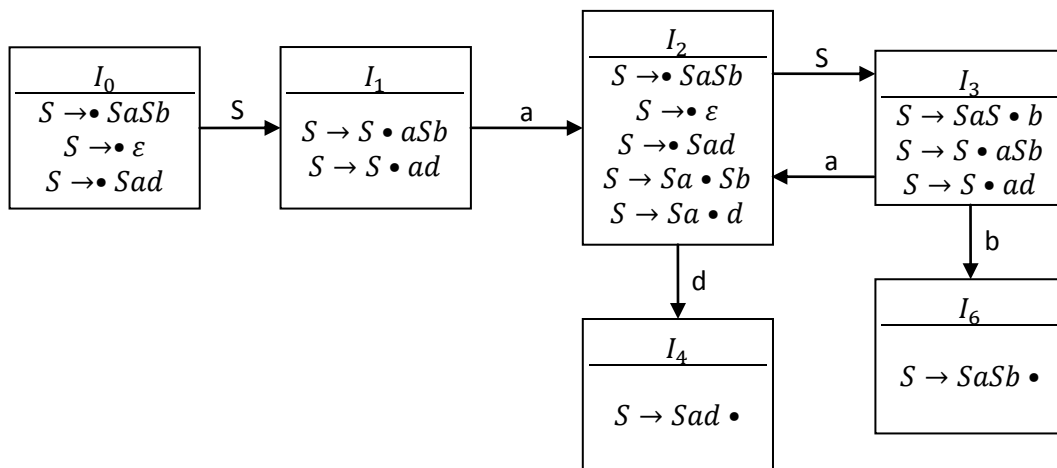
$$\Theta(I_2, S) = \text{chius}(\delta(2, S)) \cup \text{chius}(\delta(0, S)) = \text{chius}(3) \cup \text{chius}(1) = \{3, 1\} = I_3$$

$$\Theta(I_2, d) = \text{chius}(\delta(2, d)) = \text{chius}(5) = \{5\} = I_4$$

$$\Theta(I_3, a) = \text{chius}(\delta(1, a)) = \text{chius}(2) = \{2, 0\} = I_2$$

$$\Theta(I_3, b) = \text{chius}(\delta(3, b)) = \text{chius}(4) = \{4\} = I_6$$

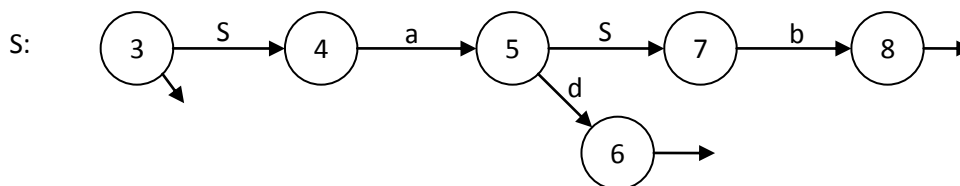
Otteniamo così:



Osserviamo che alcuni stati sono misti, perché contengono sia la riduzione  $S \rightarrow \varepsilon$  che delle regole di spostamento, perciò la grammatica non è LR(0).

# Analisi LR(1)

Riportiamo qui la rete di macchine associata alla grammatica data:



Si noti che sono stati cambiati i nomi degli stati, ma questo ovviamente non ha alcuna importanza (avremmo potuto lasciarli uguali).

Lo stato iniziale è:

$$I_0 = \text{chius}(< 3, \neg>) = \{< 3, \neg>, < 3, a >\} = I_1$$

Inoltre avremo (eseguendo ancora tutti i calcoli, nonostante si possa direttamente operare sul diagramma, risparmiando così molto tempo):

$$\begin{aligned} \Theta(I_0, S) &= \text{chius}(< \delta(3, S), \neg>) \cup \text{chius}(< \delta(3, S), a >) = \\ &= \text{chius}(< 4, \neg>) \cup \text{chius}(< 4, a >) = \{< 4, \neg>\} \cup \{< 4, a >\} = \{< 4, \neg>, < 4, a >\} = I_1 \end{aligned}$$

$$\begin{aligned} \Theta(I_1, a) &= \text{chius}(< \delta(4, a), a >) \cup \text{chius}(< \delta(4, a), \neg>) = \text{chius}(< 5, a >) \cup \text{chius}(< 5, \neg>) = \\ &= \{< 5, a >, < 3, b >, < 3, a >\} \cup \{< 5, \neg>, < 3, b >, < 3, a >\} = \\ &= \{< 5, a >, < 3, b >, < 3, a >, < 5, \neg>\} = I_2 \end{aligned}$$

$$\begin{aligned} \Theta(I_2, d) &= \text{chius}(< \delta(5, d), a >) \cup \text{chius}(< \delta(5, d), \neg>) = \text{chius}(< 6, a >) \cup \text{chius}(< 6, \neg>) = \\ &= \{< 6, a >\} \cup \{< 6, \neg>\} = \{< 6, a >, < 6, \neg>\} = I_3 \end{aligned}$$

$$\begin{aligned} \Theta(I_2, S) &= \text{chius}(< \delta(5, S), a >) \cup \text{chius}(< \delta(3, S), b >) \cup \text{chius}(< \delta(3, S), a >) \cup \text{chius}(< \delta(5, S), \neg>) = \\ &= \text{chius}(< 7, a >) \cup \text{chius}(< 4, b >) \cup \text{chius}(< 4, a >) \cup \text{chius}(< 7, \neg>) = \\ &= \{< 7, a >\} \cup \{< 4, b >\} \cup \{< 4, a >\} \cup \{< 7, \neg>\} = \{< 7, a >, < 4, b >, < 4, a >, < 7, \neg>\} = I_4 \end{aligned}$$

$$\begin{aligned} \Theta(I_4, a) &= \text{chius}(< \delta(4, a), b >) \cup \text{chius}(< \delta(4, a), a >) = \text{chius}(< 5, a >) \cup \text{chius}(< 5, b >) = \\ &= \{< 5, a >, < 3, b >, < 3, a >\} \cup \{< 5, b >, < 3, b >, < 3, a >\} = \{< 5, a >, < 3, b >, < 3, a >, < 5, b >\} = I_5 \end{aligned}$$

$$\begin{aligned} \Theta(I_4, b) &= \text{chius}(< \delta(7, b), a >) \cup \text{chius}(< \delta(7, b), \neg>) = \text{chius}(< 8, a >) \cup \text{chius}(< 8, \neg>) = \\ &= \{< 8, a >\} \cup \{< 8, \neg>\} = \{< 8, a >, < 8, \neg>\} = I_6 \end{aligned}$$

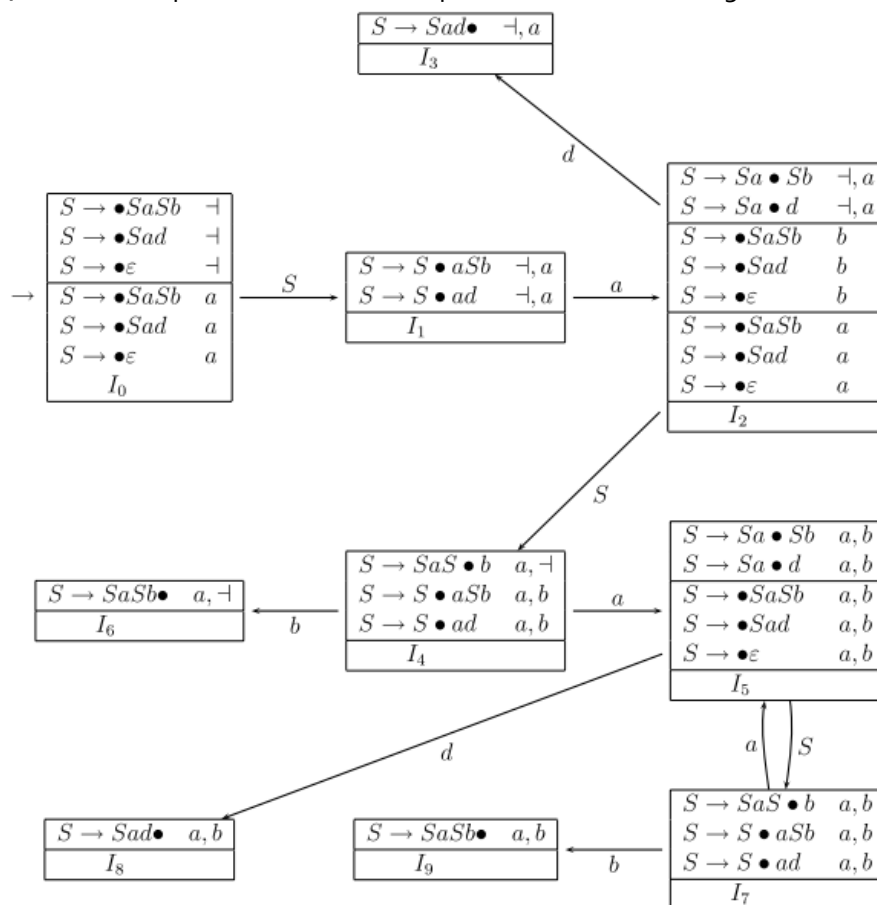
$$\begin{aligned} \Theta(I_5, S) &= \text{chius}(< \delta(5, S), a >) \cup \text{chius}(< \delta(3, S), b >) \cup \text{chius}(< \delta(3, S), a >) \cup \text{chius}(< \delta(5, S), b >) = \\ &= \text{chius}(< 7, a >) \cup \text{chius}(< 4, b >) \cup \text{chius}(< 4, a >) \cup \text{chius}(< 7, b >) = \\ &= \{< 7, a >\} \cup \{< 4, b >\} \cup \{< 4, a >\} \cup \{< 7, b >\} = \{< 7, a >, < 4, b >, < 4, a >, < 7, b >\} = I_7 \end{aligned}$$

$$\begin{aligned} \Theta(I_5, d) &= \text{chius}(< \delta(5, d), a >) \cup \text{chius}(< \delta(5, d), b >) = \text{chius}(< 6, a >) \cup \text{chius}(< 6, b >) = \\ &= \{< 6, a >\} \cup \{< 6, b >\} = \{< 6, a >, < 6, b >\} = I_8 \end{aligned}$$

$$\begin{aligned} \Theta(I_7, a) &= \text{chius}(< \delta(4, a), b >) \cup \text{chius}(< \delta(4, a), a >) = \text{chius}(< 5, a >) \cup \text{chius}(< 5, b >) = \\ &= \{< 5, a >, < 3, b >, < 3, a >\} \cup \{< 5, b >, < 3, b >, < 3, a >\} = \{< 5, a >, < 3, b >, < 3, a >, < 5, b >\} = I_5 \end{aligned}$$

$$\begin{aligned} \Theta(I_7, b) &= \text{chius}(< \delta(7, b), a >) \cup \text{chius}(< \delta(7, b), b >) = \text{chius}(< 8, a >) \cup \text{chius}(< 8, b >) = \\ &= \{< 8, a >\} \cup \{< 8, b >\} = \{< 8, a >, < 8, b >\} = I_9 \end{aligned}$$

Di conseguenza, la macchina pilota che costruiamo per l'analisi LR(1) è la seguente:



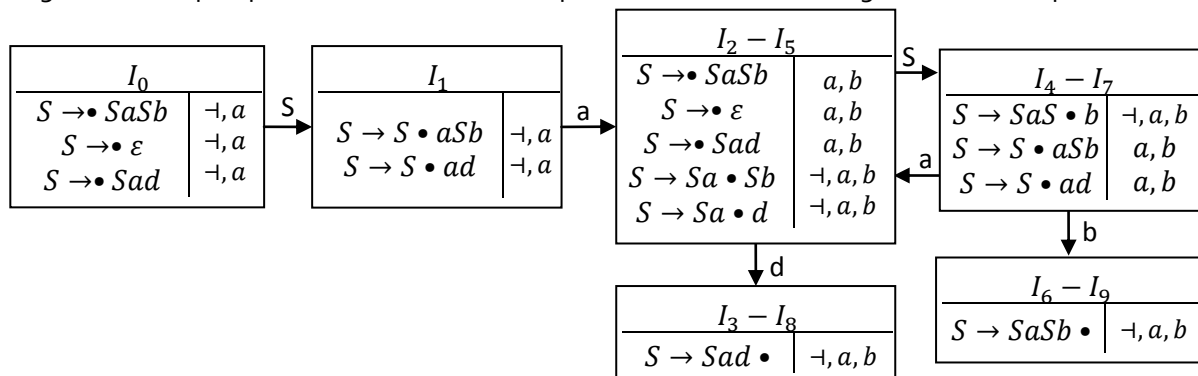
A questo punto, possiamo osservare che entrambe le condizioni LR(1), ovvero:

1. Ogni candidata di riduzione ha un insieme di prospezione disgiunto dall'insieme dei terminali che etichettano gli archi uscenti dal macrostato stesso (assenza conflitti riduzione-spostamento).
2. Se ci sono due candidate di riduzione, i loro insiemi di prospezione sono disgiunti (assenza conflitti riduzione-riduzione).

Sono verificate, perciò la grammatica è LR(1).

### Analisi LALR(1)

Se fondessimo tra loro gli stati con le stesse regole marcate nel presenti nel precedente automa pilota (ignorando quindi gli insiemi di prospezione e, anzi, considerando come insiemi di prospezione le unioni degli insiemi di prospezione che si avevano in partenza) otteniamo il seguente automa pilota:



Si può allora osservare che le ipotesi LALR(1) sono entrambe valide: per ogni stato, ogni candidata di riduzione ha un insieme di prospezione disgiunto dai simboli terminali che etichettano gli archi uscenti dallo stato stesso, e per ogni stato, se si hanno due candidate di riduzione, i loro insiemi di prospezione devono essere disgiunti. Perciò la grammatica è LALR(1).

# Analisi sintattica con il metodo di Early

## Il metodo di Early

### Metodo di Early

Un metodo che può essere applicato per l'analisi sintattica di tutte le grammatiche libere, anche ambigue, è il metodo di Early, che si basa su una simulazione di un automa a pila non deterministico.

L'algoritmo è un analizzatore discendente che sviluppa simultaneamente tutte le possibili derivazioni sinistre di una stringa data.

Per utilizzare il metodo, è necessario ancora una volta costruire per prima cosa la rete di macchine associata alla grammatica. La stringa viene letta in ordine, da sinistra a destra, e per ogni carattere letto si produce una struttura che contiene una serie di coppie del tipo:

$$(s, p)$$

Dove  $s$  è uno stato della rete di macchine e  $p$  è un **puntatore all'indietro**, che consente di stabilire qual è lo stato al quale ritornare quando è stata terminata la costruzione del sottoalbero in analisi (cioè qual è lo stato nel quale si trovava originariamente il non terminale che costituisce la parte sinistra della regola corrispondente allo stato  $s$ ). Le coppie sono dette anche **obiettivi**.

Al termine della procedura, si avrà un numero di stati pari al numero di caratteri della stringa in ingresso, più uno stato iniziale.

## Il procedimento

### Fase di inizializzazione

Per prima cosa occorre eseguire l'inizializzazione, che consiste nell'eseguire le seguenti operazioni:

1. Si inizializza lo stato  $E[0]$  in modo che contenga solamente la coppia costituita dallo stato  $q_{S_0}$ , che è lo stato iniziale della rete di macchine (cioè quello dell'automa associato all'assioma  $S$ ), con puntatore 0:

$$E[0] = \{(q_{S_0}, 0)\}$$

2. Tutti gli altri stati  $E[i]$ , con  $i \in [1, n]$ , vengono inizializzati come insiemi vuoti:

$$E[i] = \emptyset \quad \forall i \in [1, n]$$

3. Si inizializza l'indice che punta allo stato in analisi al valore zero:

$$i = 0$$

### Le fasi successive

Dopo la fase di inizializzazione, si applicano le operazioni di predizione, completamento e scansione. Si noti che al passo  $i$  l'algoritmo può aggiungere elementi solamente all'insieme  $E[i]$  e al successivo, ovvero  $E[i + 1]$ .

Se nessuna delle tre tipologie ha aggiunto nuove coppie ad  $E[i]$ , si passa all'analisi di  $E[i + 1]$ , cioè si passa alla fase successiva dell'algoritmo. Altrimenti, le regole vengono nuovamente applicate.

L'algoritmo può terminare in 2 modi:

1. Prematuramente, quando si deve passare alla fase  $i + 1$ , ma  $E[i + 1] = \emptyset$  (con  $i < n$ ).
2. Con successo, quando  $E[n]$  è stato completato e contiene almeno una coppia completata dell'assioma, cioè una coppia del tipo:

$$(S \rightarrow \alpha \bullet, 0)$$



### Operazione di predizione

L'operazione di predizione corrisponde intuitivamente a quella di chiusura. In sostanza, per ogni stato in  $E[i]$  nel quale il simbolo  $\bullet$  precede immediatamente un non terminale, si aggiunge ad  $E[i]$  stesso un nuovo obiettivo, avente come stato lo stato iniziale del non terminale che segue il simbolo  $\bullet$ , e come puntatore all'indietro l'indice  $i$  stesso; in simboli:

$$\forall (q \equiv A \rightarrow \alpha \bullet B \gamma, j) \in E[i], \text{ aggiungo } (q_{B_0} \equiv B \rightarrow \bullet \delta, i) \text{ a } E[i]$$

Nel caso particolare in cui  $B$  sia annullabile, si aggiungono all'insieme  $E[i]$  anche le coppie ottenute semplicemente spostando in avanti il puntatore, ovvero portandolo oltre  $B$ :

$$\forall (q \equiv A \rightarrow \alpha B \bullet \gamma, j)$$

### Operazione di scansione

L'operazione di predizione corrisponde intuitivamente a quella di spostamento. In sostanza, per ogni stato in  $E[i]$  nel quale il simbolo  $\bullet$  precede immediatamente il terminale che occupa la posizione  $i + 1$  nella stringa in ingresso, si aggiunge ad  $E[i + 1]$  un nuovo obiettivo, analogo al precedente, ma con il simbolo  $\bullet$  spostato dopo il terminale (ovviamente quindi lo stato sarà diverso, il puntatore no):

$$\forall (q \equiv A \rightarrow \alpha \bullet a \gamma, j) \in E[i] \text{ con } a = x_{i+1}, \text{ aggiungo } (r \equiv A \rightarrow \alpha a \bullet \gamma, j) \text{ a } E[i + 1]$$

### Operazione di completamento

L'operazione di completamento corrisponde intuitivamente a quella di riduzione. In sostanza, per ogni coppia completata presente in  $E[i]$ , si vanno a riprendere tutte le coppie nello stato indicato dal puntatore all'indietro della coppia completata stessa, nelle quali il simbolo  $\bullet$  preceda immediatamente il non terminale che costituisce la parte sinistra della regola corrispondente allo stato dell'obiettivo in analisi, e si aggiunge ad  $E[i]$  la coppia ottenuta da quest'ultimo, spostando il simbolo  $\bullet$  in avanti di una posizione e lasciando invariato il puntatore all'indietro. In simboli:

$$\forall (q \equiv A \rightarrow \alpha \bullet, j) \in E[i], \forall (r \equiv B \rightarrow \beta \bullet A \gamma, k) \in E[j] \text{ aggiungo } (s \equiv B \rightarrow \beta A \bullet \gamma, k) \text{ a } E[i]$$

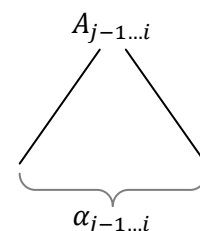
## Altre osservazioni

### Rappresentazione grafica degli stati

Gli stati vengono rappresentati graficamente mediante dei rettangoli, nei quali vengono elencati tutti gli obiettivi. Gli obiettivi ottenuti da diverse iterazioni del metodo sono separati tra loro per mezzo di una linea orizzontale. Dallo stato  $E[1]$  in poi si indica accanto allo stato anche il simbolo in ingresso associato.

### Costruzione dell'albero

Per costruire l'albero, basta ricordare una semplice regola: se nello stato  $E[i]$  si ha l'obiettivo  $(A \rightarrow \alpha, j)$ , si sta costruendo un albero del tipo mostrato nella figura a lato.



## Esempio

Se consideriamo come esempio la grammatica:

$$S \rightarrow E$$

$$E \rightarrow E + E | a$$

Otteniamo con l'analisi di Early il seguente risultato:

E[0]	a E[1]	+ E[2]	a E[3]	+ E[4]	a E[5]
$S \rightarrow \bullet E, 0$ $E \rightarrow \bullet E + E   \bullet a, 0$	$E \rightarrow a \bullet, 0$ $S \rightarrow E \bullet, 0$ $E \rightarrow E \bullet + E, 0$	$E \rightarrow E + \bullet E, 0$ $E \rightarrow \bullet E + E   \bullet a, 2$	$E \rightarrow a \bullet, 2$ $E \rightarrow E + E \bullet, 0$ $E \rightarrow E \bullet + E, 2$ $S \rightarrow E \bullet, 0$ $E \rightarrow E \bullet + E, 0$	$E \rightarrow E + \bullet E, 0$ $E \rightarrow E + \bullet E, 2$ $E \rightarrow \bullet E + E   \bullet a, 4$	$E \rightarrow a \bullet, 4$ $E \rightarrow E + E \bullet, 2$ $E \rightarrow E \bullet + E, 4$ $E \rightarrow E + E \bullet, 0$ $E \rightarrow E \bullet + E, 2$ $S \rightarrow E \bullet, 0$ $E \rightarrow E \bullet + E, 0$

# Analisi statica dei programmi

## L'analisi statica dei programmi

### ***Cos'è l'analisi statica***

L'analisi statica dei programmi è un insieme di operazioni eseguite a priori, ovvero non durante l'esecuzione del programma stesso, che ha lo scopo di estrarre delle informazioni dal codice sorgente o da una forma intermedia di codice, che consentano poi di eseguire alcune operazioni vantaggiose.

Solitamente l'analisi statica viene eseguita dopo un primo stadio di compilazione, che traduce il sorgente in una rappresentazione intermedia, e consiste in 3 operazioni fondamentali:

1. Verifica

È un esame della correttezza del programma

2. Ottimizzazione

È la trasformazione del programma finalizzata a migliorarne l'efficienza di esecuzione (ad esempio, permette di ottenere un'allocazione ottimale dei registri alle variabili, ...).

3. Schedulazione

Consiste nella modifica dell'ordine delle istruzioni, finalizzata a sfruttare meglio la pipeline e le diverse unità funzionali del processore.

Noi considereremo solamente l'analisi intraprocedurale, e non quella interprocedurale (cioè ci occuperemo solo dell'analisi del codice di una singola procedura).

## Grafo di controllo del flusso

### ***Rappresentazione del programma come automa: il grafo di controllo del flusso***

Per eseguire l'analisi statica di un programma, è conveniente rappresentare il programma stesso mediante un grafo, detto *grafo* (o *automa*) *di controllo del flusso* (*control-flow graph*). Tale automa ha come nodi le istruzioni dello specifico programma in analisi, e come archi le relazioni di dipendenza. Si noti quindi che l'automa è associato ad un singolo programma, e non all'intera famiglia dei programmi.

Le stringhe riconosciute dall'automa sono le sequenze temporali di esecuzione delle istruzioni del programma stesso.

Si noti però che in realtà il grafo di controllo non rappresenta fedelmente il programma sorgente, perché ad esempio nelle istruzioni condizionali non si ha alcuna distinzione eseguita in base all'esito del controllo della condizione stessa. Di conseguenza non è sempre vero che un cammino del grafo sia realmente eseguibile; tuttavia, in generale è indecidibile se un cammino del grafo di controllo possa o meno essere eseguito. L'analisi che vedremo è quindi di tipo cautelativo: in alcuni casi potrebbe segnalare errori che in realtà sono inesistenti.

### ***Definizione ed uso delle variabili***

Ad ogni stato del control-flow graph possiamo associare due insiemi:

1. L'insieme definisce (def)

È l'insieme che contiene tutte le variabili che vengono scritte dall'istruzione in analisi.

2. L'insieme usa

È l'insieme che contiene tutte le variabili che vengono lette dall'istruzione in analisi.

## Intervallo di vita delle variabili e definizioni inutili

### **Variabile viva**

Una variabile è viva in un certo punto  $p$  del programma se esiste qualche istruzione che potrebbe essere eseguita successivamente e che fa uso del valore che quella stessa variabile ha nel punto  $p$ .

Più formalmente, una variabile  $a$  è viva in un punto  $p$  del programma (che può essere l'ingresso o l'uscita di un'istruzione) se nel grafo esiste un cammino da  $p$  ad un altro nodo  $q$  tale che:

1. Il cammino non passa per un'istruzione  $r$  diversa da  $q$  per la quale si ha  $a \in \text{def}(r)$ .
2. L'istruzione  $q$  usa  $a$ , cioè  $a \in \text{usa}(q)$ .

### **Variabile viva all'uscita di un nodo**

Una variabile  $a$  è viva all'uscita di un nodo se è viva su almeno uno degli archi uscenti dal nodo.

### **Variabile viva all'entrata di un nodo**

Una variabile  $a$  è viva all'entrata di un nodo se è viva su almeno uno degli archi entranti nel nodo.

### **Calcolo degli intervalli di vita delle variabili mediante equazioni di flusso**

Per calcolare gli intervalli di vita delle variabili, si possono usare le equazioni di flusso, che permettono di determinare contemporaneamente tutte le variabili vive in ogni punto del grafo. Le regole da applicare sono le seguenti:

Per ogni nodo  $p$ , si impone:

$$\begin{aligned} \text{vive}_{in}(p) &= \text{usa}(p) \cup (\text{vive}_{out}(p) \setminus \text{def}(p)) \\ \text{vive}_{out}(p) &= \bigcup_{q \in \text{succ}(p)} \text{vive}_{in}(q) \end{aligned}$$

In particolare, si ottiene ovviamente che se  $p$  è finale, si ha:

$$\text{vive}_{out}(p) = \emptyset$$

Infatti, un nodo finale è un nodo che non ha successori.

Detto  $n$  il numero di istruzioni, otteniamo così un sistema di  $2n$  equazioni, che possiamo risolvere iterativamente, iniziando ciascuno dei  $2n$  insiemi incogniti all'insieme vuoto, e aggiungendo di volta in volta elementi ai vari insiemi, riapplicando le regole precedenti, fino al raggiungimento di un punto fisso.

### **Applicazioni del risultato**

L'individuazione degli intervalli di vita delle variabili serve per:

1. L'assegnazione della memoria

Se due variabili non sono mai vive contemporaneamente possono essere allocate nella stessa locazione di memoria o nello stesso registro.

2. L'eliminazione di definizioni inutili

Se una variabile non è viva all'uscita dell'istruzione che la definisce, vuol dire che la definizione della variabile è del tutto inutile e può essere eliminata.

For the exam: they usually ask liveness analysis but can also sometimes ask useless definition

## Definizioni raggiungenti

### Definizione raggiungente

Si dice che la definizione di una variabile  $a$  all'interno dell'istruzione  $q$  raggiunge l'ingresso di un'istruzione  $p$  (che potrebbe anche coincidere con  $q$ ) se esiste un cammino da  $q$  a  $p$  che non passa per alcun nodo diverso da  $q$  che ridefinisca la variabile  $a$ .

In generale, la definizione della variabile  $a$  nell'istruzione  $q$  verrà indicata con  $a_q$ .

### Insiemi $in$ , $out$ e $sop$

Definiamo preliminarmente i seguenti insiemi:

- L'insieme  $in(p)$  di tutte le definizioni che raggiungono l'ingresso dell'istruzione  $p$ .
- L'insieme  $out(p)$  di tutte le definizioni che raggiungono l'uscita dell'istruzione  $p$ .
- L'insieme  $sop(p)$  di tutte le definizioni che raggiungono l'ingresso ma non l'uscita di  $p$ , perché vengono soppresse da  $p$ .

### Individuazione delle definizioni raggiungenti mediante l'uso di equazioni di flusso

Anche in questo caso possiamo utilizzare le equazioni di flusso per individuare le definizioni raggiungenti. Le regole sono le seguenti:

$$out(p) = def(p) \cup (in(p) \setminus sop(p))$$

$$in(p) = \bigcup_{q \in pred(p)} out(q)$$

Si ricava così che, se  $p$  è il nodo iniziale, cioè il nodo senza predecessori, allora:

$$in(p) = \emptyset$$

Anche in questo caso, possiamo utilizzare queste equazioni per ottenere un sistema di  $2n$  equazioni in  $2n$  insiemi incogniti, risolubile iterativamente iniziando come vuoti tutti gli insiemi in gioco.

### Applicazione: sostituzione di una variabile con una costante

Possiamo sostituire nell'istruzione  $p$  una costante opportuna  $k$  alla variabile  $a$  se valgono le seguenti condizioni:

1. Esiste un'istruzione  $q$  che definisca  $a := k$  (con  $k$  costante) tale che  $a_q$  raggiunge l'ingresso di  $p$ .
2. Non ci sono altre definizioni  $a_r$  (con  $r$  diverso da  $q$ ) che raggiungano l'ingresso di  $p$ .

## Disponibilità di una variabile

### Disponibilità

Diciamo che una variabile  $a$  è disponibile all'ingresso dell'istruzione  $p$  (cioè subito prima della sua esecuzione) se nel grafo di controllo ogni cammino che parta dal nodo iniziale e che arrivi all'ingresso di  $p$  contiene almeno una definizione di  $a$ . In caso contrario, si dice che la variabile è indisponibile.

### Istruzione non ben inizializzata

Un'istruzione  $p$  non è ben inizializzata se usa qualche variabile indisponibile al suo ingresso, ovvero se esiste un predecessore  $q$  di  $p$  le cui definizioni raggiungenti non comprendono tutte le variabili usate in  $p$ . Più formalmente:

$$\exists p \in pred(p) \text{ t.c. } \sim (usa(p) \subseteq out'(q))$$

# Traduzione e analisi semantica

## La traduzione sintattica

### Traduzione sintattica (o compilazione)

L'operazione di traduzione sintattica (detta anche di compilazione) consiste nell'ottenere, a partire da una certa stringa detta *sorgente* (*source*), una nuova stringa, detta *immagine* (*target*).

Le due stringhe possono appartenere a linguaggi diversi, detti *linguaggio sorgente*  $L_S$  e *linguaggio immagine*  $L_I$ . La macchina che esegue questa operazione viene semplicemente chiamata *traduttore sintattico* oppure *compilatore*.

Formalmente possiamo quindi considerare la traduzione come una relazione:

$$\tau \subseteq L_S \times L_I$$

Dove si intende che  $\tau$  sarà un insieme di coppie di stringhe, che stanno a significare che la prima stringa (appartenente a  $L_S$ ) si traduce nella seconda stringa della coppia (appartenente a  $L_I$ ).

### Traduzione ambigua

Sulla base della definizione fornita, è possibile che una stessa stringa  $x_1$  di  $L_S$  abbia più immagini in  $L_I$ . Ad esempio, possiamo avere una traduzione come la seguente:

$$\tau = \{(x_1, y_1), (x_1, y_2), (x_2, y_2)\}$$

Nella quale  $x_1$  ha come immagine sia  $y_1$  che  $y_2$ . In questo caso, si parla di traduzione ambigua o a più valori, oppure ancora di *traduzione non univoca*.

### Traduzione inversa

La traduzione inversa di una traduzione  $\tau$  è semplicemente la relazione inversa di  $\tau$ , ovvero  $\tau^{-1}$ .

### Proprietà delle traduzioni

Le principali proprietà che una traduzione può avere sono le seguenti:

1. Iniettività

Una traduzione è iniettiva se è univoca, ovvero ad ogni stringa del linguaggio sorgente corrisponde solamente una stringa del linguaggio immagine.

2. Suriettività

Una traduzione è suriettiva se ogni stringa del linguaggio immagine è immagine di almeno una stringa del linguaggio sorgente.

3. Biunivocità

Una traduzione è biunivoca se è sia suriettiva che iniettiva.

### Tipologie di traduzione

1. Traduzione puramente sintattica

La traduzione puramente sintattica si basa esclusivamente sulla sintassi del linguaggio sorgente. Essa viene realizzata per mezzo di automi a stati finiti, espressioni regolari e grammatiche.

2. Traduzione guidata dalla sintassi

La traduzione guidata dalla sintassi, oltre ad utilizzare la sintassi del linguaggio sorgente, sfrutta anche alcune variabili che consentono di eseguire la traduzione stessa. Si utilizza in questo caso il modello delle *grammatiche con attributi*, che studieremo in seguito.

## Grammatiche di traduzione

### Schemi sintattici per la traduzione puramente sintattica

Una traduzione puramente sintattica può essere formalizzata in vari modi equivalenti:

1. Mediante una *grammatica di traduzione*
2. Mediante un *2I-automa* (detto anche *macchina di Rabin & Scott*).
3. Mediante un *IO-automa*.
4. Mediante una espressione regolare di traduzione.

Iniziamo qui vedendo la prima modalità.

### Grammatiche di traduzione

Dati il linguaggio sorgente  $L_S$  e dato il linguaggio pozzo  $L_P$ , e date due grammatiche  $G_S$  e  $G_I$  che generino tali linguaggi, se valgono le seguenti condizioni:

1. Esiste una corrispondenza uno ad uno tra le regole di  $G_S$  e di  $G_I$ ;
2. Le regole corrispondenti delle 2 grammatiche differiscono solo per i terminali che vi compaiono;
3. In regole corrispondenti di  $G_I$  e  $G_S$  compaiono gli stessi non terminali, nello stesso ordine.

Allora si può costruire una grammatica di traduzione ( $G_{trad}$ ), che è in realtà solo una rappresentazione più sintetica della coppia di grammatiche date.

La grammatica di traduzione prevede semplicemente che si riscrivano le stesse regole di  $G_S$  e di  $G_I$ , sostituendo ai terminali dei simboli del tipo:

$$\frac{a_1}{a_2}$$

Dove  $a_1 \in \Sigma_S^*$  e  $a_2 \in \Sigma_I^*$ , e dove  $a_1$  è la sequenza di terminali che compare in un certo punto della grammatica  $G_S$  e  $a_2$  è la sequenza di terminali che compare nello stesso punto della corrispondente regola di  $G_I$ . Ad esempio, se si ha la seguente coppia di regole corrispondenti:

$$S \rightarrow abSc$$

$$S \rightarrow Sc$$

Si otterrà nella grammatica di traduzione una regola del tipo:

$$S \rightarrow \frac{ab}{\varepsilon} S \frac{c}{c}$$

La grammatica di traduzione così ottenuta può essere ambigua solo nel caso in cui la grammatica  $G_I$  sorgente sia ambigua. La grammatica sorgente deve essere considerata ambigua anche nel caso in cui abbia delle regole duplicate (inserite ad esempio perché la stessa regola deve corrispondere a più regole della grammatica immagine  $G_I$ ).

### Esempio

Ad esempio, se vogliamo costruire la grammatica che permetta di tradurre le espressioni aritmetiche contenenti le operazioni di somma e sottrazione nella tradizionale forma infissa, in modo da ottenere le corrispondenti espressioni in forma polacca postfissa, possiamo utilizzare la seguente grammatica:

$$E \rightarrow E \frac{+}{\varepsilon} E \frac{\varepsilon}{add}$$

$$E \rightarrow E \frac{-}{\varepsilon} E \frac{\varepsilon}{sub}$$

$$E \rightarrow \left( \frac{E}{\varepsilon} \right) \frac{\varepsilon}{\varepsilon}$$

$$E \rightarrow \frac{i}{i}$$

### Costruzione dell'IO-automa a partire dalla grammatica

A partire dalla grammatica, si può pensare di costruire un automa a pila che, oltre a riconoscere le stringhe di  $G_I$ , emetta la corrispondente traduzione. Tuttavia, l'automa così ottenuto è non deterministico, perciò il procedimento non ha un'applicazione pratica.

### Costruzione del parsificatore deterministico LL(1) con azioni di stampa

Se la grammatica sorgente  $G_S$  è una grammatica LL(1) (lo stesso varrebbe per il caso LL(k), ma per semplicità analizziamo il caso più semplice), possiamo sempre arricchire il parsificatore LL(1) costruito mediante le tecniche già viste, in modo che sia in grado di eseguire anche le operazioni di stampa per ottenere la traduzione delle strutture stringhe riconosciute.

### Costruzione del parsificatore deterministico LR(1) con azioni di stampa

Per quanto riguarda l'analisi LR(1), non vale la stessa regola vista per la LL(1): anche se  $G_S$  è LR(1), talvolta non si può arricchire il parsificatore LR(1) per  $G_S$  con azioni di stampa per eseguire la traduzione. Perché ciò accada, è necessario che la grammatica  $G_I$  sia in forma LR(1) e che contemporaneamente la grammatica di traduzione sia in forma postfissa, ovvero che tutte le regole di  $G_I$  siano del tipo:

$$A \rightarrow \gamma w$$

Dove:

$$\gamma \in V^*$$

$$w \in \Sigma_I^*$$

In termini più semplici, non c'è alcun non terminale che preceda un simbolo terminale, in nessuna delle regole di  $G_I$ . In questo modo, il nuovo automa pilota emetterà le traduzioni solamente durante le riduzioni, evitando scritture premature ed erranee.

Si noti però che portare la grammatica in forma postfissa potrebbe far cadere la condizione LR(1) di  $G_I$ .

## Traduzioni regolari

### Il 2l-automa e l'IO-automa

Nel caso particolare in cui la traduzione sia regolare, ovvero nel caso in cui la grammatica di traduzione sia lineare a destra, possiamo costruire un automa finito che riconosca  $L(G_t)$ . Tale automa può essere pensato in due modi, formalmente molto diversi, ma che vengono rappresentati mediante lo stesso diagramma degli stati e delle transizioni:

#### 1. 2l-automa (o di Rabin & Scott)

L'automa viene pensato come un automa a stati finiti con due diversi nastri d'ingresso: uno che riceve la stringa appartenente a  $L_S$ , uno usato per ricevere la stringa di  $L_I$ . L'automa verifica che effettivamente le stringhe ricevute appartengano al linguaggio sorgente e al linguaggio immagine e che siano l'una la traduzione dell'altra; in caso contrario, emette errore.

L'automa così pensato è un automa deterministico.

#### 2. IO-automa

L'automa è visto come un automa che ha un solo nastro in ingresso, sul quale riceve la stringa appartenente al linguaggio sorgente, e che emette su un nastro d'uscita la corrispondente traduzione. Nel caso generale, l'automa così ottenuto non è deterministico.

Entrambi gli automi vengono banalmente ottenuti dalla grammatica di traduzione lineare a destra, costruendo l'automa che la riconosca.

### Espressione regolare di traduzione

Naturalmente, dal 2l-automa possiamo anche determinare un'espressione regolare che generi il linguaggio riconosciuto dal 2l-automa stesso. Tale espressione regolare sarà un'espressione regolare di traduzione, il cui significato è ovvio. Ad esempio:

$$\left(\frac{a^2}{b^2}\right)^* \cup \frac{a}{c} \left(\frac{a^2}{c^2}\right)^*$$

### ~~Teorema di Nivat~~

Una relazione di traduzione regolare  $\tau$  può sempre essere espressa equivalentemente mediante una grammatica di traduzione lineare a destra (o lineare a sinistra), mediante un 2l-automa finito o mediante un'espressione regolare di traduzione.

### Traduzioni non regolari di linguaggi regolari

Si noti che, anche nel caso in cui  $L_S$  e  $L_I$  siano entrambi linguaggi regolari, è possibile che la corrispondente traduzione non sia regolare.

## Grammatiche ad attributi

### Traduzione guidata dalla sintassi

I traduttori guidati dalla sintassi sono dei traduttori che usano delle funzioni operanti sull'albero sintattico, calcolando delle variabili, dette anche *attributi semantici*. Il valore degli attributi esprime il significato o la semantica della frase sorgente.

### Grammatica con attributi

Si costruisce allora un modello, noto come *grammatica con attributi*, che è una grammatica nella quale ad ogni regola si può associare una o più procedure che consentano di calcolare gli attributi associati ai non terminali che compaiono nella regola stessa. Tali procedure sono dei programmi non formalizzati, perciò la grammatica con attributi non è un modello formale, ma è un metodo di ingegneria del software per progettare i compilatori in modo ordinato e coerente.

Gli attributi sono associati ai vari simboli non terminali o terminali che compaiono nella grammatica. Al termine, l'obiettivo è quello di associare i valori di tutti gli attributi previsti per ogni non terminale della grammatica che compaia nell'albero sintattico della stringa ricevuta in ingresso. È possibile che non terminali diversi prevedano il calcolo di attributi semantici diversi. L'albero con l'aggiunta di tali valori è detto *albero decorato*.

### Attributi destri e sinistri

- Attributi sinistri o sintetizzati

Un attributo è detto sinistro o sintetizzato se consente di calcolare un attributo associato al non terminale presente come parte sinistra della regola; sarà perciò del tipo:

$$a_0 := f(\dots)$$

- Attributi destri o ereditati

Un attributo è detto destro o ereditato se consente di calcolare un attributo associato a uno dei non terminali presente nella parte destra della regola; sarà perciò del tipo:

$$a_k := f(\dots), k \geq 1$$

Un attributo non può essere contemporaneamente destro e sinistro.

### Compilazione a due passate

La prima soluzione implementativa per l'uso di grammatiche ad attributi è quella che prevede l'uso di una tecnica di compilazione a due passate, nella quale cioè prima si esegue la parsificazione (o analisi sintattica) della stringa in ingresso, ottenendo così l'albero sintattico associato alla stringa stessa, e poi si esegue la valutazione (o analisi semantica), che consente di ottenere l'albero sintattico *decorato*, ovvero con l'aggiunta degli attributi.

In questo modo chiaramente si hanno prestazioni inferiori rispetto all'analisi ad un'unica passata, ma si evita ogni problema di ambiguità in fase di calcolo degli attributi, perché in tale fase si dispone già dell'albero sintattico.

### Come specificare le funzioni semantiche

Le funzioni semantiche vengono specificate accanto alle regole stesse. Gli attributi che compaiono nelle procedure vengono associati ai corrispondenti non terminali utilizzando dei pedici, con significato ovvio, come mostrato nell'esempio seguente:

$N_0 \rightarrow D_1 \cdot D_2$	$v_0 := v_1 + v_2 \cdot 2^{-l_2}$	
$D_0 \rightarrow D_1 B_2$	$v_0 := 2 \cdot v_1 + v_2$	$l_0 := l_1 + 1$
$D_0 \rightarrow B_1$	$v_0 := v_1$	$l_0 := 1$
$B_0 \rightarrow 0$	$v_0 := 0$	
$B_0 \rightarrow 1$	$v_0 := 1$	

L'esempio appena riportato permette di tradurre un numero espresso in binario, ottenendo così la corrispondente rappresentazione in decimale.



### Calcolo degli attributi

Se si sta eseguendo la compilazione a 2 passate, dato un albero sintattico, è sufficiente applicare ad ogni nodo una funzione semantica, cominciando dai nodi le cui procedure hanno argomenti noti (che sono in genere le foglie dell'albero). Il risultato finale della traduzione sarà il valore che, al termine di tutto il calcolo, è associato alla radice dell'albero (cioè all'assioma). I valori di tutti gli altri attributi servono solamente come dati intermedi, perciò al termine del calcolo non servono più.

### Definizione più rigorosa di grammatiche ad attributi

Una definizione un po' più rigorosa di grammatica con attributi è data dai seguenti elementi:

1. Sia data dalla tupla:

$$G = \langle V, \Sigma, P, S \rangle$$

Dove  $V$  è l'insieme dei non terminali,  $\Sigma$  è l'alfabeto d'ingresso,  $P$  è l'insieme di produzioni e  $S$  l'assioma.

2. Per ciascuno dei simboli terminali o non terminali è definito un certo insieme di attributi associati:

$$\forall D \in (V \cup \Sigma), \text{attr}(D) = \{\alpha, \beta, \dots\}$$

Gli attributi possono essere o destri o sinistri. Un attributo può essere associato a diversi simboli terminali e/o non terminali della grammatica. Non è però possibile che un attributo associato ad un simbolo terminale sia di tipo sinistro, e spesso gli attributi associati ai terminali assumono valori costanti.

3. Per ogni attributo è specificato un dominio, cioè l'insieme dei valori che può assumere.
4. Si definisce un insieme di funzioni (o regole, o procedure) semantiche, ciascuna delle quali è associata ad una produzione appartenente a  $P$ , detta *supporto* della funzione. Le funzioni semantiche sono espresse in un metalinguaggio (tipicamente un linguaggio programmatico). Più funzioni possono avere lo stesso supporto. Le funzioni semantiche associate ad una regola del tipo:

$$D_0 \rightarrow D_1 D_2 \dots D_n$$

sono del tipo:

$$\alpha_k := f((\text{attr}(D_0) \cup \text{attr}(D_1) \cup \dots \cup \text{attr}(D_n)) / \text{attr}(D_k)), \quad 0 \leq k \leq n$$

Non è possibile che funzioni diverse che calcolino lo stesso attributo dello stesso non terminale abbiano lo stesso supporto. L'insieme delle funzioni semantiche associate ad una produzione  $p$  è detto  $\text{fun}(p)$ .

5. Per ogni produzione  $p$ , devono valere le seguenti regole:
  - a) Per ogni attributo sinistro  $\sigma_0$  associato al non terminale che compare come parte sinistra di  $p$ , deve esistere una e una sola funzione che lo definisca.
  - b) Per ogni attributo sinistro  $\sigma_i$  (con  $1 \leq i \leq n$ ) associato ad un simbolo che compare nella parte destra di  $p$ , non deve esistere alcuna funzione in  $\text{fun}(p)$  che lo definisca (l'attributo è sinistro!).
  - c) Per ogni attributo destro  $\sigma_0$  associato al non terminale che compare come parte sinistra di  $p$ , non deve esistere alcuna funzione in  $\text{fun}(p)$  che lo definisca (l'attributo è destro!).
  - d) Per ogni attributo destro  $\sigma_i$  (con  $1 \leq i \leq n$ ) associato ad un simbolo che compare nella parte destra di  $p$ , deve esistere una e una sola funzione in  $\text{fun}(p)$  che lo definisca.

### Attributi interni ed esterni

- Attributi interni

Gli attributi interni di una produzione  $p$  sono gli attributi associati a simboli che compaiono in  $p$ , il cui valore viene stabilito per mezzo di funzioni di  $fun(p)$ . Sono perciò gli attributi sinistri  $\sigma_0$  e gli attributi destri  $\sigma_i$  (con  $i \geq 1$ ).

- Attributi esterni

Gli attributi esterni di una produzione  $p$  sono gli attributi associati a simboli che compaiono in  $p$ , il cui valore non viene stabilito per mezzo di funzioni di  $fun(p)$ . Sono perciò gli attributi destri  $\sigma_0$  e gli attributi sinistri  $\sigma_i$  (con  $i \geq 1$ ).

### Grafo delle dipendenze di una produzione

Per ogni produzione  $p \in P$ , possiamo costruire un grafo delle dipendenze, che è un grafo orientato costruito nel modo seguente:

1. Si costruisce il semplicissimo albero della regola, che ha come radice la sua parte sinistra e come foglie, tutte allo stesso livello, tutti i simboli che compaiono nella parte destra della regola.
2. Si indicano accanto ad ogni terminale e non terminale tutti i relativi attributi (non è necessario indicare il pedice). In particolare:
  - a) Gli attributi sintetizzati vengono indicati a sinistra del nodo.
  - b) Gli attributi ereditati vengono indicati a destra del nodo.
3. Si inserisce un arco orientato da un attributo  $a_1$  ad un altro attributo  $a_2$  se e solo se esiste almeno una produzione in  $fun(p)$  che permette di calcolare  $a_2$  sulla base del valore di  $a_1$ .

### Grafo delle dipendenze di un albero

In maniera ovvia, possiamo "incollare tra loro" tutti i grafi delle dipendenze delle varie produzioni che compaiono in un albero sintattico, ottenendo così il grafo delle dipendenze dell'albero sintattico.

### Grammatiche acicliche

Una grammatica ad attributi è detta aciclica (o *loop-free*) se ogni albero ha un grafo delle dipendenze aciclico.

### Ordine di valutazione degli attributi dato un albero sintattico

Dato un albero sintattico, per stabilire l'ordine nel quale calcolare gli attributi che vi compaiono è sufficiente costruire un ordinamento topologico di tutti gli attributi che vi compaiono, sulla base della relazione espressa dal grafo delle dipendenze dell'albero stesso.

In sostanza, per trovare l'ordinamento topologico si procede semplicemente andando ad individuare ogni volta un nodo che non abbia alcun arco entrante: tale nodo (attributo) può essere valutato, perché non dipende da alcun attributo incognito. Il nodo viene poi cancellato dal grafo delle dipendenze, così come tutti gli archi da esso uscenti; si procede così fino ad aver eliminato tutti gli archi. Se ad un certo punto non è più possibile proseguire, significa che il grafo di partenza non era aciclico.

Si noti che è indispensabile che il primo assegnamento avvenga sulla base di un valore costante.

### Valutazione ad una sola scansione

Per migliorare le prestazioni, si può costruire un analizzatore che preveda che si passi attraverso ogni nodo dell'albero una e una sola volta, calcolandone tutti gli attributi "in una sola volta". Si ottiene così un valutatore ad una singola scansione, che esegue una visita in profondità dell'albero. In sostanza, si eseguono per ogni nodo tutte le seguenti operazioni, partendo dall'assioma (radice dell'albero):

1. Visita il sottoalbero  $t_N$  avente radice nel nodo  $N$  e:
  - 1.1. Si calcolano tutti gli attributi destri del nodo  $N$
  - 1.2. Per ogni figlio  $N_i$  di  $N$ , si esegue la visita in profondità del sottoalbero  $t_i$  (in un ordine qualsiasi).
  - 1.3. Si calcolano tutti gli attributi sinistri del nodo  $N$ .

### **Grafo dei fratelli di una produzione**

Data la produzione  $p$ :

$$D_0 \rightarrow D_1 D_2 \dots D_n, \quad n \geq 1$$

Possiamo definire un grafo, detto grafo dei fratelli, il cui significato è simile ma non uguale a quello del grafo delle dipendenze della produzione. I nodi in questo caso non sono infatti gli attributi, bensì i simboli che compaiono nella parte destra di  $p$ , cioè i terminali e/o non terminali:

$$D_1, D_2, \dots, D_n$$

Nel grafo dei fratelli si ha un arco orientato da  $D_i$  a  $D_j$  se e solo se esiste un attributo associato a  $D_j$  che si calcola sulla base di un attributo qualsiasi associato a  $D_i$ .

Costruire il grafo dei fratelli a partire dal grafo delle dipendenze significa quindi collassare tra loro tutti i nodi che corrispondono ad attributi associati allo stesso simbolo, eliminando poi il nodo associato all'assioma.

### **Condizione per la visita ad una sola scansione**

Non tutte le grammatiche ad attributi consentono la valutazione ad una sola scansione. È possibile eseguire la valutazione ad una sola scansione a patto che, per ogni produzione  $p$ :

$$D_0 \rightarrow D_1 D_2 \dots D_n, \quad n \geq 1$$

Valgano le seguenti 4 condizioni:

1. Il grafo delle dipendenze di  $p$  è aciclico.
2. Il grafo delle dipendenze di  $p$  non contiene alcun cammino che va da un attributo sinistro ad un attributo destro associati allo stesso simbolo della parte destra di  $p$ .
3. Nel grafo delle dipendenze di  $p$  non esiste alcun arco da un attributo sinistro del padre ad un attributo destro di uno qualsiasi dei nodi figli.
4. Il grafo dei fratelli di  $p$  è aciclico.

Si noti che se si utilizzano solo attributi sintetizzati, allora la grammatica è sempre ad una sola scansione.

### **Costruzione del valutatore ad una sola scansione**

Per costruire il valutatore ad una sola scansione, per ogni produzione:

$$p: D_0 \rightarrow D_1 D_2 \dots D_n, \quad n \geq 1$$

Si eseguono i seguenti passi:

1. Si costruisce un ordine topologico dei non terminali  $D_1 D_2 \dots D_n$  rispetto al grafo dei fratelli.
2. Per ogni simbolo della parte destra, si costruisce un ordine topologico degli attributi destri (cioè degli attributi ereditati).
3. Si costruisce un ordine topologico degli attributi sinistri (sintetizzati) del non terminale  $D_0$ .

Dopodiché, si procede nel modo seguente:

1. Si parte dalla radice dell'albero, e si considera la produzione corrispondente.
2. Per ogni produzione:
  - 2.1. Si calcolano gli attributi ereditati di ogni simbolo della parte destra, eseguendo le operazioni nell'ordine topologico precedentemente individuato.
  - 2.2. Si passa ad analizzare singolarmente tutti i sottoalberi associati ai non terminali della parte destra di  $p$ , nell'ordine topologico precedentemente individuato.
  - 2.3. Si calcolano gli attributi sinistri di  $D_0$ , nell'ordine topologico precedentemente individuato.

## Analisi sintattica e semantica integrate

### *Eeguire contemporaneamente l'analisi sintattica e l'analisi semantica*

Fino ad ora abbiamo supposto di partire dagli alberi sintattici già individuati da un parsificatore. Proviamo adesso a pensare a come eseguire contemporaneamente le due operazioni di costruzione dell'albero sintattico e di decorazione dell'albero stesso con gli attributi, in modo da ottenere prestazioni superiori.

### *Parsificatore a discesa ricorsiva con attributi*

Una delle possibilità che si hanno è quella di costruire un parsificatore a discesa ricorsiva con attributi.

Il modo di procedere è simile rispetto a quello imposto dall'algoritmo ad una sola scansione, ma in questo caso i sottoalberi di un certo nodo devono essere necessariamente analizzati nell'ordine naturale (da sinistra verso destra), e non secondo un ordine topologico qualsiasi.

Per questo motivo è necessario aggiungere ulteriori condizioni.

### *La condizione L*

Sia data una grammatica adatta all'analisi discendente deterministica, cioè una grammatica  $LL(k)$ . È possibile costruire un parsificatore a discesa ricorsiva con attributi se vale la condizione  $L$ , che è data dalle due seguenti condizioni:

1. La grammatica è una grammatica ad una sola scansione.
2. Per ogni produzione  $p$ , il grafo dei fratelli di  $p$  non possiede alcun arco diretto da un nodo più a destra del nodo al quale l'arco stesso arriva, cioè non esistono archi del tipo:

$$D_i \rightarrow D_j, i > j$$

# Laboratorio

## Compilatore

### Concetto di compilatore

Un compilatore è uno strumento software, il cui scopo è quello di tradurre un programma espresso in un linguaggio sorgente  $L_0$  in un programma, semanticamente equivalente al precedente, espresso in un altro linguaggio  $L_1$ .

### Stadi del compilatore

Il compilatore è organizzato come una pipeline: ogni stadio applica delle trasformazioni al programma in input, producendo così un programma in output da fornire allo stadio successivo della pipeline. Un semplice compilatore contiene almeno 3 stadi:

1. Front-end

Lo stadio di front-end è lo stadio che consente di astrarre dall'hardware.

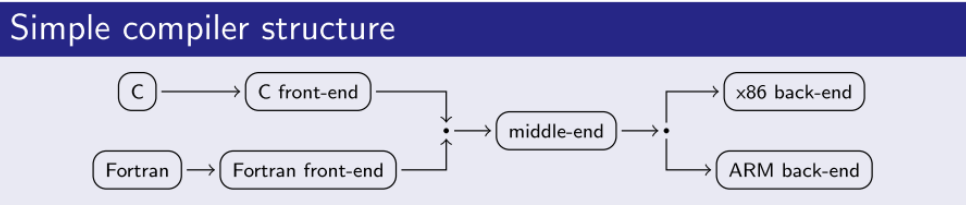
2. Middle-end

Lo stadio di middle-end è lo stadio che astrarre sia dal linguaggio ad alto livello, sia dall'hardware.

3. Back-end

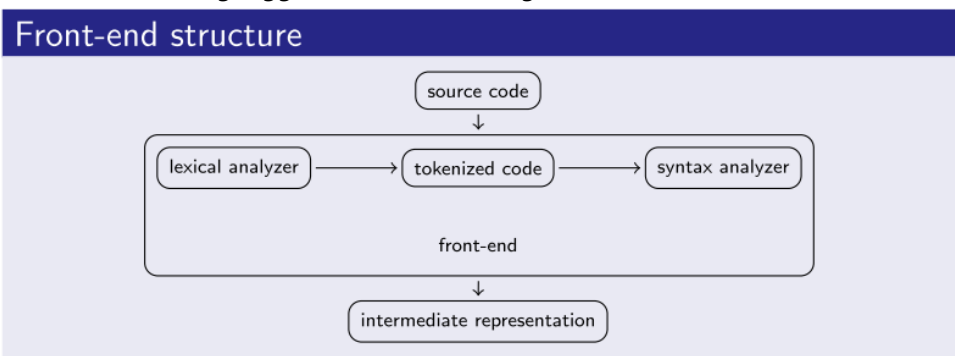
Lo stadio di back-end è lo stadio che astrae dal linguaggio ad alto livello.

In sostanza, quando un programma viene fornito in input ad un compilatore, il programma attraversa questi tre stadi, nell'ordine indicato: dapprima si ottiene un programma intermedio, indipendente dall'hardware, mediante l'uso di uno stadio di *front-end*, che dipende dal linguaggio di programmazione del programma sorgente; dopodiché, il nuovo programma attraversa lo stadio di *middle-end*, indipendente sia dal linguaggio di programmazione, sia dall'hardware, ed infine attraversa lo stadio di *back-end* che, indipendentemente dal linguaggio di alto livello sorgente, trasforma il programma intermedio che riceve in linguaggio macchina (e perciò tale stadio sarà diverso a seconda dell'hardware utilizzato).



### Struttura dello stadio di front-end

Lo stadio di front-end ha lo scopo di trasformare il programma sorgente in una forma intermedia, riconoscendo i costrutti del linguaggio ed individuando gli eventuali errori di sintassi.



Noi ci concentreremo solamente sull'analisi dello stadio di font-end, analizzando le varie fasi che lo costituiscono, e che sono indicate nella precedente figura.

## L'analisi lessicale

### Introduzione e terminologia

Vediamo prima di tutto il significato di alcuni termini fondamentali:

- Lessicale  
Il termine lessicale significa "relativo alle parole o al vocabolario di un linguaggio inteso come distinto dalla sua grammatica e dalla sua costruzione".
- Parola  
Una parola è un semplice costrutto. In senso tecnico, le parole sono più semplici di quelle in linguaggio naturale, ma sono molto più numerose, perciò non possono essere semplicemente enumerate. La loro struttura può però essere descritta mediante espressioni regolari.
- Analisi lessicale  
L'analisi lessicale ha lo scopo di:
  - a) Individuare i token (cioè gli identificatori delle variabili, i vari simboli usati come operatori, ...).
  - b) Decorare i token con delle informazioni aggiuntive (ad esempio, nel caso degli identificatori, il relativo nome, ...).

L'analisi lessicale viene eseguita per mezzo di uno scanner, solitamente generato in maniera automatica sulla base di una descrizione che utilizza espressioni regolari (questo perché la codifica a mano risulterebbe complessa e facilmente soggetta ad errori).

In sostanza, lo scanner è solamente un grande automa a stati finiti.

### Flex

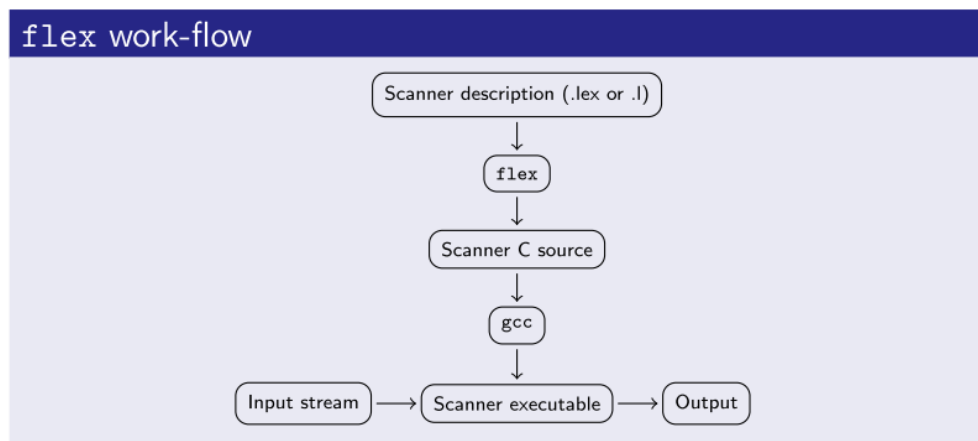
Per generare uno scanner che sia in grado di riconoscere le parole, è possibile utilizzare lo strumento *flex*, che è disponibile nei sistemi operativi UNIX all'interno del repository della distribuzione:

```
Debian:    aptitude install flex
Fedora:    yum install flex
```

In alcuni casi lo scanner è sufficiente e può essere utilizzato sia per individuare le parole, sia per eseguire delle azioni semantiche. Tuttavia, la traduzione non è un'operazione semplice, perciò nella maggior parte dei casi lo scanner si limita a preparare l'input per l'analisi semantica, eseguendo 3 operazioni:

1. Individuazione delle parole (come gli identificatori);
2. Pulizia (ad esempio, eliminando i commenti);
3. Aggiunta di informazioni alle parole (come ad esempio i nomi degli identificatori).

Lo strumento flex è inserito in una catena di strumenti, come riportato in figura:



## Formato di un file flex

Un file flex ha il formato seguente:

### 1. Definizione dei token

Contiene la definizione di alcuni token, che permettono semplicemente di associare un certo nome ad un insieme di caratteri. Per definire tali insiemi di caratteri, si utilizzano delle espressioni regolari. In genere, si usano con lo scopo di definire dei concetti molto semplici, che verranno poi "riutilizzati" nelle regole.

Di seguito sono riportati due esempi di definizione di token, che corrispondono ad una generica cifra decimale e ad un generico identificatore, che inizi con una lettera maiuscola o minuscola e possa contenere, oltre a tali lettere, anche il simbolo underscore e le cifre decimali:

```
DIGIT      [0-9]
ID         [A-Za-z][a-zA-Z0-9_]*
```

### 2. Regole

Ogni regola definisce un'espressione regolare che permetta di riconoscere un certo costrutto; inoltre, ogni regola definisce una certa porzione di codice C, che indica quali sono le azioni che devono essere eseguite quando viene riconosciuto il costrutto indicato dall'espressione regolare alla quale il codice stesso è associato. Le azioni possono anche accedere ai dati riconosciuti (*matched data*).

Se lo scanner è molto semplice, l'azione definisce direttamente le azioni che devono essere eseguite; in caso contrario, le azioni restituiscono semplicemente il tipo di token riconosciuto, che verrà poi elaborato in una fase successiva.

Di seguito è riportato un semplicissimo esempio di azione:

```
{DIGIT}+ { return NUMBER; }
```

### 3. Codice utente

Segue infine il codice utente, che è del semplice codice C, copiato così com'è. Contiene la funzione main, ma anche tutta una serie di funzioni che possono essere invocate dalle azioni, ...

La struttura del file è la seguente:

```
//Definizioni
%%
//Regole
%%
//Codice utente
```

Si noti poi che del codice arbitrario può essere inserito in ognuna delle altre sezioni, utilizzando la seguente notazione:

```
%{
    //Codice C
}%
```

Ad esempio, si usa il codice scritto in questo modo all'interno della sezione delle definizioni, per inserire tutte le primitive di inclusione di librerie:

```
%{
    #include<limits.h>
    #include<string.h>
}%
```

### Specificare le espressioni regolari

Le espressioni regolari in flex vengono specificate nel modo seguente:

Sintassi	Stringe riconosciute
$x$	Il carattere $x$ .
$.$	Tutti i caratteri ad eccezione del terminatore di riga.
$[xyz]$	Il carattere $x$ , oppure il carattere $y$ , oppure il carattere $z$ .
$[a-z]$	Ogni lettera tra $a$ e $z$ , cioè tutte le lettere minuscole.
$[\^a-z]$	Ogni carattere che non sia una lettera tra $a$ e $z$ .
$\{X\}$	Un'espansione della definizione di $X$ , dove $X$ è un token definito nell'apposita sezione.
"hello"	La stringa hello.

Esistono poi gli operatori di composizione tra diverse espressioni regolari:

Sintassi	Stringe riconosciute
$R$	L'espressione regolare $R$ .
$RS$	Concatenazione tra le due espressioni regolari $R$ ed $S$ .
$R S$	$R$ oppure $S$ .
$R^*$	Operatore stella.
$R^+$	Operatore croce.
$R?$	Opzionalità (zero o una occorrenza di $R$ ).
$R\{m,n\}$	Un numero di occorrenze di $R$ minimo pari a $m$ , massimo pari a $n$ (estremi inclusi).
$R\{n, \}$	Un numero di occorrenze di $R$ almeno pari ad $n$ .
$R\{n\}$	Un numero di occorrenze di $R$ esattamente pari ad $n$ .
$(R)$	Parentesi utilizzate per specificare a piacimento le precedenze tra gli operatori.
$^R$	Specifica che $R$ deve trovarsi all'inizio della linea.
$R\$$	Specifica che $R$ deve trovarsi alla fine della linea.

### Regole utilizzate dallo scanner

Lo scanner flex utilizza le seguenti regole:

1. La "longest matching rule"  
Se più di una espressione regolare corrisponde ad una certa stringa, tra quelle individuate si applica la regola che genera la stringa più lunga.
2. La "first rule"  
Se si hanno più regole che generano stringhe della stessa lunghezza e tutte corrispondono ad una certa stringa, allora quella che viene applicata è la prima che è scritta nel codice.
3. La "default action rule"  
Se nessuna regola riconosce una certa stringa, allora si passa a considerare il carattere successivo in input, e si copia in output il carattere che non è stato riconosciuto da alcuna espressione regolare. Lo scanner va poi avanti secondo le regole già descritte.



## L'analisi semantica

### **Sintassi**

La sintassi è lo studio delle regole mediante le quali le parole o altri elementi che costituiscono la struttura di una frase vengono combinati tra loro per ottenere delle frasi grammaticali.

### **Analisi sintattica**

L'analisi sintattica, come abbiamo già visto nella parte di teoria, è quell'insieme di operazioni che, data in input una stringa di testo, consentono di determinarne la struttura, ovvero:

1. Si determina come sono combinati tra loro gli elementi che costituiscono la stringa in input;
2. Si determinano le precedenze tra gli operatori.

La struttura è definita per mezzo di una grammatica. L'analisi sintattica viene eseguita a partire da un flusso in input *tokenizzato*, cioè che sia già stato analizzato da un analizzatore lessicale.

Il risultato dell'analisi sintattica è l'albero sintattico associato alla stringa ricevuta in input.

L'analisi sintattica viene eseguita da un parser che, come sappiamo, può essere di tipo LL oppure LR.

### **Analisi semantica**

L'analisi semantica consiste invece nel decorare l'albero sintattico ottenuto mediante l'analisi sintattica, aggiungendo ad esso i vari attributi appositamente calcolati.

### **Bison**

Bison è un parser LR; più precisamente, si tratta di un parser LALR(1). Può essere installato mediante i seguenti comandi:

```
Debian:    aptitude install bison
Fedora:    yum install bison
```

Come ogni altro parser, bison consuma dei token precedentemente prodotti da un analizzatore lessicale, che nel caso tipico è flex.

### **Struttura di un file bison**

Un file bison ha la seguente struttura:

1. Definizioni C  
Contiene le definizioni delle variabili, l'inclusione degli header, ...
2. Definizioni  
Definisce tutti i token che potranno essere consumati e consente di indicare le precedenze tra i token stessi.
3. Regole grammaticali  
Contiene tutte le regole grammaticali e le operazioni semantiche.
4. Codice utente  
Contiene del codice C, ovvero la funzione main e tutta una serie di funzioni che possono essere invocate all'interno delle azioni definite dalle regole grammaticali.

In particolare, la struttura è la seguente:

```
%{
    //Definizioni C
}%
//Definizioni
%%
//Regole grammaticali
%%
//Codice utente
```

### Definizione delle regole grammaticali

Le regole grammaticali vengono definite nel modo seguente (molto simile ad una tradizionale grammatica non contestuale):

```
nome_non_terminale :    possibilità1
                     |    possibilità2
                     |    ...
                     ;
```

Dove le varie possibilità (cioè le diverse alternative) saranno ad esempio del tipo:

```
non_terminale1 non_terminale2 TOKEN
```

I token, che vengono riconosciuti perché convenzionalmente indicati in maiuscolo, a differenza dei non terminali che vengono indicati sempre con lettere minuscole, compaiono così, senza particolari simboli aggiuntivi. Il significato ad esempio della regola:

```
nt1      :
          |      nt2 TOKENA
```

È uguale a quello della regola della grammatica:

$$nt_1 \rightarrow \varepsilon | nt_2 TOKEN_A$$

Dove però  $TOKEN_A$  corrisponde di fatto ad una sequenza di terminali già riconosciuta dall'analizzatore lessicale (possiamo quindi considerarlo come un singolo terminale), mentre  $nt_2$  e  $nt_1$  sono dei non terminali definiti all'interno del file bison stesso, mediante le regole della grammatica.

### Definizione dei token e gestione delle precedenze

Nel caso in cui la grammatica sia ambigua, si sa dalla teoria che si otterranno diversi alberi sintattici associati ad una certa stringa, perciò la compilazione non può essere eseguita, in quanto si avrebbero più possibili programmi in output, con comportamenti diversi tra loro, e non si sarebbe in grado di individuarne uno "corretto" tra tutti quelli possibili.

Un modo per gestire l'ambiguità è quello di definire delle priorità tra i token. I token vengono definiti nel modo seguente:

```
%left TOKEN_1 TOKEN_2
%right TOKEN_3
```

In questo modo, indichiamo che:

1. I primi due token sono associativi a sinistra.
2. Il terzo token è associativo a destra.
3. I primi due token hanno la stessa precedenza.
4. Il terzo token ha precedenza più alta rispetto ai primi due.

I token non ambigui, per i quali non è necessario specificare l'associatività, vengono dichiarati anche mediante la sintassi:

```
%token TOKEN_4
```

### Azioni semantiche

È possibile inoltre aggiungere delle azioni semantiche, in codice C, accanto ad ognuna delle alternative, nel modo seguente:

```
nome_non_terminale :    possibilità1 { /* Codice C */ }
                     |    possibilità2 { /* Codice C */ }
                     |    ...
                     ;
```

È anche possibile inserire delle azioni semantiche "in mezzo" alla parte destra della regola. Le azioni semantiche vengono eseguite immediatamente dopo il riconoscimento della regola semantica precedente, ogni volta che tale riconoscimento avviene.

Se l'azione non è in coda alla parte destra della regola, ovvero:

```
lh2      :      rhs1 { /*Azione1*/ } rhs2 { /*Azione2*/ }
```

La sequenza di operazioni eseguite è la seguente (almeno dal punto di vista concettuale):

1. Si riconosce rhs1.
2. Si esegue la prima azione semantica.
3. Si riconosce rhs2.
4. Si esegue la seconda azione semantica.
5. Si riconosce lhs.

Ogni simbolo ha ad esso associato un certo valore, che di default è una variabile intera (come vedremo in seguito, il tipo può anche essere personalizzato). Tale valore viene associato sia ai terminali che ai non terminali. Il valore associato ai vari simboli può essere utilizzato all'interno delle azioni. In particolare, ogni azione può utilizzare la variabile associata a tutti i simboli che compaiono alla sua sinistra. Fa eccezione la variabile associata alla parte sinistra della regola stessa, che può essere sempre acceduta, ma solo in modalità di lettura, e mai in modalità di scrittura, in quanto è sempre un attributo sintetizzato, e mai ereditato.

L'accesso avviene nel modo descritto mediante il seguente esempio; data la regola:

```
lhs : rhs1 { /*Azione1*/ } rhs2 { /*Azione2*/ }
```

- La variabile associata a lhs viene indicata con \$\$.
- La variabile associata a rhs1 viene indicata con \$1.
- La variabile associata alla prima azione viene indicata con \$2.
- La variabile associata a rhs2 viene indicata con \$3.

### Compilare i sorgenti flex/bison

Per generare il parser e lo scanner occorre eseguire i seguenti comandi:

```
$ bison -d nome_bison.y
$ flex nome_flex.le
```

Per ottenere l'eseguibile finale, si deve poi eseguire il comando:

```
$ gcc nome_bison.tab.c lex.yy.c
```

## ACSE

### Cos'è ACSE

ACSE, acronimo di *Advanced Compiler System for Education*, è un semplice stadio front-end di un compilatore, che accetta un linguaggio con sintassi simile a quella del C, generando un codice intermedio del tipo RISC. La parte di laboratorio della prova d'esame richiede di modificare i sorgenti di ACSE per aggiungere il supporto a tipologie di istruzioni non previste.

In particolare, modificheremo solo il tool *acse*, che consente di ottenere, partendo dal sorgente iniziale, il codice assembly corrispondente.

### Cos'è LANCE

LANCE, acronimo di *LANguage for Compiler Education*, è il linguaggio simil-C che viene riconosciuto dal compilatore ACSE. Supporta le seguenti operazioni basilari:

1. Un insieme standard di operazioni aritmetiche, logiche e relazionali.
2. Un insieme ridotto di operazioni di controllo del flusso, che comprende solamente l'istruzione condizionale if ed i cicli while e do while.
3. Un'istruzione di input, del tipo:

```
read(var)
```

che consente di memorizzare in var il numero intero letto da tastiera.

4. Un'istruzione di output, del tipo:

```
write(var)
```

che consente di stampare a video la variabile intera var.

Per quanto riguarda i tipi di dati, sono supportati solamente:

1. Un tipo scalare, ovvero i dati interi.
2. Gli array unidimensionali di interi.

### Istruzioni assembler

Le istruzioni assembler nelle quali verranno tradotte le istruzioni LANCE sono del tipo:

Istruzione	Significato
ADD R3 R1 R2	$R_3 = R_1 + R_2$
SUB R3 R1 R2	$R_3 = R_1 - R_2$
ADDI R3 R1 #4	$R_3 = R_1 + 4$
SUBI R3 R1 #5	$R_3 = R_1 - 5$
LOAD R1 Lo	Carica in R1 il valore nella locazione di memoria Lo
BEQ Lo	Salta all'istruzione Lo se l'ultima operazione aritmetica ha dato risultato nullo.
BNE Lo	Salta all'istruzione Lo se l'ultima operazione aritmetica non ha dato zero.

È possibile anche utilizzare l'indirizzamento indiretto da registro, usando la notazione (R1).

Si hanno poi a disposizione due registri speciali:

1. Il registro Ro, che contiene sempre il valore 0 e non può essere sovrascritto.
2. Il registro di stato, che viene letto e scritto implicitamente dalle altre operazioni, e che contiene i flag utilizzati anche per eseguire i salti condizionali.

In ogni caso, per generare le istruzioni assembly sono definite, con ovvio significato, funzioni come ad esempio:

- gen\_add\_instruction
- gen\_addi\_instruction
- gen\_read\_instruction
- gen\_beq\_instruction
- ...

## Sorgenti ACSE

### L'analizzatore lessicale (flex), ovvero lo scanner

```
%option noyywrap
%{
    #include <string.h>
    #include "axe_struct.h"
    #include "collections.h"
    #include "Acse.tab.h"
    #include "axe_constants.h"
    extern int line_num;
    extern int num_error;
    extern int yyerror(const char* errmsg);
}%
/*=====TOKEN DEFINITIONS=====*/
DIGIT      [0-9]
ID          [a-zA-Z_][a-zA-Z0-9_]*
/*=====TOKENS=====*/
%option noyywrap
%x comment
%%
"\r\n"      { ++line_num; }
"\n"        { ++line_num; }
[ \t\f\v]+  { /* Ignore whitespace. */ }
"//[^\n]*"   { ++line_num; /* ignore comment lines */ }
"/*"        BEGIN(comment);
<comment>[^*\n]*
<comment>[^*\n]*\n      { ++line_num; }
<comment>"*" + [^*/\n]*
<comment>"*" + [^*/\n]*\n { ++line_num; }
<comment>"*" + "/"      BEGIN(INITIAL);
"{"          { return LBRACE; }
"}"          { return RBRACE; }
"["          { return LSQUARE; }
"]"          { return RSQUARE; }
"("          { return LPAR; }
")"          { return RPAR; }
";"          { return SEMI; }
":"          { return COLON; }
"+"          { return PLUS; }
"-"          { return MINUS; }
"*"          { return MUL_OP; }
"/"          { return DIV_OP; }
%"           { return MOD_OP; }
"&"          { return AND_OP; }
"|"          { return OR_OP; }
"!"          { return NOT_OP; }
"="          { return ASSIGN; }
"<"          { return LT; }
">"          { return GT; }
"<<"         { return SHL_OP; }
">>"         { return SHR_OP; }
"=="         { return EQ; }
"!="         { return NOTEQ; }
"<="         { return LTEQ; }
">="         { return GTEQ; }
"&&"         { return ANDAND; }
"||"         { return OROR; }
","          { return COMMA; }
"do"         { return DO; }
"else"       { return ELSE; }
"for"        { return FOR; }
"if"         { return IF; }
"int"        { yyval.intval = INTEGER_TYPE; return TYPE; }
"while"      { return WHILE; }
"return"     { return RETURN; }
"read"       { return READ; }
"write"      { return WRITE; }
{ID}         { yyval.svalue=strdup(yytext); return IDENTIFIER; }
{DIGIT}+     { yyval.intval = atoi( yytext );
               return(NUMBER); }
.            { yyerror("Error: unexpected token");
               num_error++;
               return (-1); /* invalid token */
}
```

## ***L'analizzatore sintattico (bison), ovvero il parser***

```
%{
#include "axe_struct.h"
#include "axe_engine.h"
#include "symbol_table.h"
#include "axe_errors.h"
#include "collections.h"
#include "axe_expressions.h"
#include "axe_gencode.h"
#include "axe_utils.h"
#include "axe_array.h"
#include "axe_io_manager.h"
int line_num;
int num_error;           /* number of errors */
int num_warning;         /* number of warnings */
t_program_infos *program; /* ALL PROGRAM INFORMATION */
}%
/*=====
                                SEMANTIC RECORDS
=====*/
%union {
    int intval;
    char *svalue;
    t_axe_expression expr;
    t_axe_declaration *decl;
    t_list *list;
    t_axe_label *label;
    t_while_statement while_stmt;
}
/*=====
                                TOKENS
=====*/
%start program
%token LBRACE RBRACE LPAR RPAR LSQUARE RSQUARE
%token SEMI COLON PLUS MINUS MUL_OP DIV_OP MOD_OP
%token AND_OP OR_OP NOT_OP
%token ASSIGN LT GT SHL_OP SHR_OP EQ NOTEQ LTEQ GTEQ
%token ANDAND OROR
%token COMMA
%token FOR
%token RETURN
%token READ
%token WRITE
%token <label> DO
%token <while_stmt> WHILE
%token <label> IF
%token <label> ELSE
%token <intval> TYPE
%token <svalue> IDENTIFIER
%token <intval> NUMBER
%type <expr> exp
%type <decl> declaration
%type <list> declaration_list
%type <label> if_stmt
/*=====
                                OPERATOR PRECEDENCES
=====*/
%left COMMA
%left ASSIGN
%left OROR
%left ANDAND
%left OR_OP
%left AND_OP
%left EQ NOTEQ
%left LT GT LTEQ GTEQ
%left SHL_OP SHR_OP
%left MINUS PLUS
%left MUL_OP DIV_OP
%right NOT
```

```

/*=====
BISON GRAMMAR
=====*/
%%
program : var_declarations statements
        {
            /* Notify the end of program and generates HALT */
            set_end_Program(program);
            YYACCEPT;
        }
;
var_declarations : var_declarations var_declaration
                 | /* empty */
;
var_declaration : TYPE declaration_list SEMI
                { set_new_variables(program, $1, $2); }
;
declaration_list : declaration_list COMMA declaration
                 { $$ = addElement($1, $3, -1); }
                 | declaration
                 { $$ = addElement(NULL, $1, -1); }
;
declaration : IDENTIFIER ASSIGN NUMBER
            { $$ = alloc_declaration($1, 0, 0, $3);
              if ($$ == NULL)
                  notifyError(AXE_OUT_OF_MEMORY);
            }
            | IDENTIFIER LSQUARE NUMBER RSQUARE
            { $$ = alloc_declaration($1, 1, $3, 0);
              if ($$ == NULL)
                  notifyError(AXE_OUT_OF_MEMORY);
            }
            | IDENTIFIER
            {
                $$ = alloc_declaration($1, 0, 0, 0);
                if ($$ == NULL)
                    notifyError(AXE_OUT_OF_MEMORY);
            }
;
code_block : statement
            | LBRACE statements RBRACE
;
statements : statements statement
            | statement
;
statement : assign_statement SEMI
            | control_statement
            | read_write_statement SEMI
            | SEMI
            { gen_nop_instruction(program); }
;
control_statement : if_statement
                  | while_statement
                  | do_while_statement SEMI
                  | return_statement SEMI
;
read_write_statement : read_statement
                     | write_statement
;
assign_statement : IDENTIFIER LSQUARE exp RSQUARE ASSIGN exp
                 {
                     storeArrayElement(program, $1, $3, $6);
                     free($1);
                 }
                 | IDENTIFIER ASSIGN exp
                 {
                     int location;
                     t_axe_instruction *instr;
                     location = get_symbol_location(program, $1, 0);
                     if ($3.expression_type == IMMEDIATE)
                         instr = gen_addi_instruction
                             (program, location, REG_0, $3.value);
                     else
                         instr = gen_add_instruction
                             (program, location, REG_0, $3.value, CG_DIRECT_ALL);
                     free($1);
                 }
;

```

```

if_statement      : if_stmt
                    {
                        assignLabel(program, $1);
                    }
                    | if_stmt ELSE
                    {
                        $2 = newLabel(program);
                        gen_bt_instruction (program, $2, 0);
                        assignLabel(program, $1);
                    }
                    code_block
                    {
                        assignLabel(program, $2);
                    }
;
if_stmt      : IF
              {
                  $1 = newLabel(program);
              }
              LPAR exp RPAR
              {
                  if ($4.expression_type == IMMEDIATE)
                      gen_load_immediate(program, $4.value);
                  else
                      gen_andb_instruction(program, $4.value,
                                             $4.value, $4.value, CG_DIRECT_ALL);
                      gen_beq_instruction (program, $1, 0);
              }
              code_block { $$ = $1; }
;
while_statement  : WHILE
                  {
                      $1 = create_while_statement();
                      $1.label_condition = assignNewLabel(program);
                  }
                  LPAR exp RPAR
                  {
                      if ($4.expression_type == IMMEDIATE)
                          gen_load_immediate(program, $4.value);
                      else
                          gen_andb_instruction(program, $4.value,
                                                 $4.value, $4.value, CG_DIRECT_ALL);
                      $1.label_end = newLabel(program);
                      gen_beq_instruction (program, $1.label_end, 0);
                  }
                  code_block
                  {
                      /* jump to the beginning of the loop */
                      gen_bt_instruction
                          (program, $1.label_condition, 0);
                      /* fix the label 'label_end' */
                      assignLabel(program, $1.label_end);
                  }
;
do_while_statement : DO
                    {
                        $1 = newLabel(program);
                        assignLabel(program, $1);
                    }
                    code_block WHILE LPAR exp RPAR
                    {
                        if ($6.expression_type == IMMEDIATE)
                            gen_load_immediate(program, $6.value);
                        else
                            gen_andb_instruction(program, $6.value,
                                                   $6.value, $6.value, CG_DIRECT_ALL);
                            gen_bne_instruction (program, $1, 0);
                    }
;
return_statement : RETURN
;
read_statement  : READ LPAR IDENTIFIER RPAR
                  {
                      int location;
                      location = get_symbol_location(program, $3, 0);
                      gen_read_instruction (program, location);
                      free($3);
                  }
;

```



```

write_statement : WRITE LPAR exp RPAR
{
    int location;
    if ($3.expression_type == IMMEDIATE)
    {
        location = gen_load_immediate(program, $3.value);
    }
    else
        location = $3.value;
    gen_write_instruction (program, location);
}

;
exp: NUMBER      { $$ = create_expression ($1, IMMEDIATE); }
| IDENTIFIER     {
    int location;
    location = get_symbol_location(program, $1, 0);
    $$ = create_expression (location, REGISTER);
    free($1);
}
| IDENTIFIER LSQUARE exp RSQUARE {
    int reg;
    reg = loadArrayElement(program, $1, $3);
    $$ = create_expression (reg, REGISTER);
    free($1);
}
| NOT_OP NUMBER  { if ($2 == 0)
    $$ = create_expression (1, IMMEDIATE);
    else
        $$ = create_expression (0, IMMEDIATE);
}
| NOT_OP IDENTIFIER {
    int identifier_location;
    int output_register;
    identifier_location =
        get_symbol_location(program, $2, 0);
    output_register =
        getNewRegister(program);
    gen_notl_instruction (program, output_register
        , identifier_location);
    $$ = create_expression (output_register, REGISTER);
    free($2);
}
| exp AND_OP exp {
    $$ = handle_bin_numeric_op(program, $1, $3, ANDB); }
| exp OR_OP exp {
    $$ = handle_bin_numeric_op(program, $1, $3, ORB); }
| exp PLUS exp {
    $$ = handle_bin_numeric_op(program, $1, $3, ADD); }
| exp MINUS exp {
    $$ = handle_bin_numeric_op(program, $1, $3, SUB); }
| exp MUL_OP exp {
    $$ = handle_bin_numeric_op(program, $1, $3, MUL); }
| exp DIV_OP exp {
    $$ = handle_bin_numeric_op(program, $1, $3, DIV); }
| exp LT exp {
    $$ = handle_binary_comparison (program, $1, $3, _LT_); }
| exp GT exp {
    $$ = handle_binary_comparison (program, $1, $3, _GT_); }
| exp EQ exp {
    $$ = handle_binary_comparison (program, $1, $3, _EQ_); }
| exp NOTEQ exp {
    $$ = handle_binary_comparison (program, $1, $3, _NOTEQ_); }
| exp LTEQ exp {
    $$ = handle_binary_comparison (program, $1, $3, _LTEQ_); }
| exp GTEQ exp {
    $$ = handle_binary_comparison (program, $1, $3, _GTEQ_); }
| exp SHL_OP exp {
    $$ = handle_bin_numeric_op(program, $1, $3, SHL); }
| exp SHR_OP exp {
    $$ = handle_bin_numeric_op(program, $1, $3, SHR); }
| exp ANDAND exp {
    $$ = handle_bin_numeric_op(program, $1, $3, ANDL); }
| exp OROR exp {
    $$ = handle_bin_numeric_op(program, $1, $3, ORL); }
| LPAR exp RPAR { $$ = $2; }

```

```

| MINUS exp      { if ($2.expression_type == IMMEDIATE)
                  {
                      $$ = $2;
                      $$>value = - ($$.value);
                  }
                  else
                  {
                      t_>axe_expression exp_r0;
                      exp_r0.value = REG_0;
                      exp_r0.expression_type = REGISTER;
                      $$ = handle_bin_numeric_op
                          (program, exp_r0, $2, SUB);
                  }
                  }

;
%%
/*=====
                                MAIN
=====*/
int main (int argc, char **argv)
{
    /* initialize all the compiler data structures and global variables */
    init_compiler(argc, argv);

    yyparse();

    return 0;
}
/*=====
                                YYERROR
=====*/
int yyerror(const char* errmsg)
{
    errorcode = AXE_SYNTAX_ERROR;

    return 0;
}

```

### Altri file

- [Jumnote.txt](#)

```

*****
HOW TO GENERATE JUMPS
*****
This is an example:
gen_beq_instruction( ... label ... )
Generates a jump-if-equal instruction (i.e., jump if flag zero is set)
to 'label'.
That means a jump to 'label' if the preceding expression is FALSE.

```

- [axe\\_array.h](#)

```

/* *****
 * axe_array.h
 * CODE GENERATION FOR ARRAY MANAGEMENT (LOAD/STORE)
 * *****/
/* Generates the instructions for loading the content of an
 * array element into a register.
 * ID: array name
 * index: index of the array cell
 * Returns the register number
 * */
extern int loadArrayElement(t_program_infos *program
                          , char *ID, t_>axe_expression index);

/* Generetas the instructions for loading an array cell
 * address into a register.*/
extern int loadArrayAddress(t_program_infos *program
                          , char *ID, t_>axe_expression index);

/* Generates the instructions for storing data into the array
 * cell indexed by ID and index*/
extern void storeArrayElement(t_program_infos *program, char *ID
                          , t_>axe_expression index, t_>axe_expression data);

```

- axe\_engine.h

```

/*****
 * axe_engine.h
 *
 * Contains t_program_infos and some functions for LABEL MANAGEMENT
 * (reserve, fix, assign)
 * *****/
typedef struct t_program_infos
{
    t_list *variables;
    t_list *instructions;
    t_list *data;
    t_axe_label_manager *lmanager;
    t_symbol_table *sy_table;
    int current_register;
} t_program_infos;

/* initialize the informations associated with the program. */
extern t_program_infos * allocProgramInfos();

/* add a new instruction to the current program. This function is directly
 * called by all the functions defined in 'axe_gencode.h' */
extern void addInstruction(t_program_infos *program, t_axe_instruction *instr);

/* reserve a new label identifier and return the identifier to the caller */
extern t_axe_label * newLabel(t_program_infos *program);

/* assign the given label identifier to the next instruction. Returns
 * the label assigned; otherwise (an error occurred) LABEL_UNSPECIFIED */
extern t_axe_label * assignLabel(t_program_infos *program, t_axe_label *label);

/* reserve and fix a new label. It returns either the label assigned or the
 * value LABEL_UNSPECIFIED if an error occurred */
extern t_axe_label * assignNewLabel(t_program_infos *program);

/* add a variable to the program */
extern void createVariable(t_program_infos *program
    , char *ID, int type, int isArray, int arraySize, int init_val);

/* get a previously allocated variable */
extern t_axe_variable * getVariable
    (t_program_infos *program, char *ID);

/* get the label that marks the starting address of the variable
 * with name "ID" */
extern t_axe_label * getLabelFromVariableID
    (t_program_infos *program, char *ID);

/* get a register still not used. This function returns
 * the ID of the register found*/
extern int getNewRegister(t_program_infos *program);

```

- axe\_expression.h

```

/*****
 * axe_expressions.h
 * *****/
/* This function generates instructions for binary numeric
 * operations. It takes as input two expressions and a binary
 * operation identifier, and it returns a new expression that
 * represents the result of the specified binary operation
 * applied to 'exp1' and 'exp2'.
 *
 * Valid values for 'binop' are:
 * ADD
 * ANDB
 * ORB
 * SUB
 * MUL
 * DIV */
extern t_axe_expression handle_bin_numeric_op (t_program_infos *program
        , t_axe_expression exp1, t_axe_expression exp2, int binop);

/* This function generates instructions that perform a
 * comparison between two values. It takes as input two
 * expressions and a binary comparison identifier, and it
 * returns a new expression that represents the result of the
 * specified binary comparison between 'exp1' and 'exp2'.
 *
 * Valid values for 'condition' are:
 * _LT_      (used when is needed to test if the value of 'exp1' is less than
 *            the value of 'exp2')
 * _GT_      (used when is needed to test if the value of 'exp1' is greater than
 *            the value of 'exp2')
 * _EQ_      (used when is needed to test if the value of 'exp1' is equal to
 *            the value of 'exp2')
 * _NOTEQ_   (used when is needed to test if the value of 'exp1' is not equal to
 *            the value of 'exp2')
 * _LTEQ_    (used when is needed to test if the value of 'exp1' is less than
 *            or equal to the value of 'exp2')
 * _GTEQ_    (used when is needed to test if the value of 'exp1' is greater than
 *            the value of 'exp2') */
extern t_axe_expression handle_binary_comparison (t_program_infos *program
        , t_axe_expression exp1, t_axe_expression exp2, int condition);

```

- axe\_gencode.h

```

/* *****/
 * axe_gencode.h
 * CODE GENERATION
 * *****/
/*-----
 *
 *                      NOP & HALT
 *-----*/
extern t_axe_instruction * gen_nop_instruction
        (t_program_infos *program);

extern t_axe_instruction * gen_halt_instruction
        (t_program_infos *program);

```

```

/*-----
 *
 *                      UNARY OPERATIONS
 *-----*/

/* A LOAD instruction requires the following parameters:
 * 1. A destination register (where will be loaded the requested value)
 * 2. A label information (can be a NULL pointer. If so, the address
 *    value will be taken into consideration)
 * 3. A direct address (if label is different from NULL) */
extern t_axe_instruction * gen_load_instruction
    (t_program_infos *program, int r_dest, t_axe_label *label, int address);

extern t_axe_instruction * gen_read_instruction
    (t_program_infos *program, int r_dest);

extern t_axe_instruction * gen_write_instruction
    (t_program_infos *program, int r_dest);

/* A STORE instruction copies a value from a register to a
 * specific memory location. The memory location can be
 * either a label identifier or a address reference.
 * In order to create a STORE instruction the caller must
 * provide a valid register location ('r_dest') and an
 * instance of 't_axe_label' or a numeric address */
extern t_axe_instruction * gen_store_instruction
    (t_program_infos *program, int r_dest, t_axe_label *label, int address);

/* A MOVA instruction copies an address value into a register.
 * An address can be either an instance of 't_axe_label'
 * or a number (numeric address) */
extern t_axe_instruction * gen_mova_instruction
    (t_program_infos *program, int r_dest, t_axe_label *label, int address);

/*-----
 *
 *                      STATUS REGISTER TEST INSTRUCTIONS
 *-----*/

/* rdest = 1 if the last numeric operation is gte 0
 * rdest = 0 otherwise */
extern t_axe_instruction * gen_sge_instruction
    (t_program_infos *program, int r_dest);

/* EQ */
extern t_axe_instruction * gen_seq_instruction
    (t_program_infos *program, int r_dest);

/* GT */
extern t_axe_instruction * gen_sgt_instruction
    (t_program_infos *program, int r_dest);

/* LE */
extern t_axe_instruction * gen_sle_instruction
    (t_program_infos *program, int r_dest);

/* LT */
extern t_axe_instruction * gen_slt_instruction
    (t_program_infos *program, int r_dest);

/* NE */
extern t_axe_instruction * gen_sne_instruction
    (t_program_infos *program, int r_dest);

```

```

/*-----
*
*          BINARY OPERATIONS
*-----*/
/*-----
* REGISTER = REGISTER OP IMMEDIATE
*
* Suffix "li" means logical
* Suffix "bi" means bitwise
* Prefix "e" means exclusive
*-----*/
extern t_axe_instruction * gen_addi_instruction
    (t_program_infos *program, int r_dest, int r_source1, int immediate);

extern t_axe_instruction * gen_subi_instruction
    (t_program_infos *program, int r_dest, int r_source1, int immediate);

extern t_axe_instruction * gen_andli_instruction
    (t_program_infos *program, int r_dest, int r_source1, int immediate);

extern t_axe_instruction * gen_orli_instruction
    (t_program_infos *program, int r_dest, int r_source1, int immediate);

extern t_axe_instruction * gen_eorli_instruction
    (t_program_infos *program, int r_dest, int r_source1, int immediate);

extern t_axe_instruction * gen_andbi_instruction
    (t_program_infos *program, int r_dest, int r_source1, int immediate);

extern t_axe_instruction * gen_muli_instruction
    (t_program_infos *program, int r_dest, int r_source1, int immediate);

extern t_axe_instruction * gen_orbi_instruction
    (t_program_infos *program, int r_dest, int r_source1, int immediate);

extern t_axe_instruction * gen_eorbi_instruction
    (t_program_infos *program, int r_dest, int r_source1, int immediate);

extern t_axe_instruction * gen_divi_instruction
    (t_program_infos *program, int r_dest, int r_source1, int immediate);

// SHL = shift left
extern t_axe_instruction * gen_shli_instruction
    (t_program_infos *program, int r_dest, int r_source1, int immediate);

extern t_axe_instruction * gen_shri_instruction
    (t_program_infos *program, int r_dest, int r_source1, int immediate);

extern t_axe_instruction * gen_notl_instruction
    (t_program_infos *program, int r_dest, int r_source1);

extern t_axe_instruction * gen_notb_instruction
    (t_program_infos *program, int r_dest, int r_source1);

/*-----
*
*          TERNARY OPERATIONS
*
*          REGISTER = REGISTER OP REGISTER
*
* Suffix "l" means logical
* Suffix "b" means bitwise
*-----*/
extern t_axe_instruction * gen_add_instruction (t_program_infos *program
    , int r_dest, int r_source1, int r_source2, int flags);

extern t_axe_instruction * gen_sub_instruction (t_program_infos *program
    , int r_dest, int r_source1, int r_source2, int flags);

extern t_axe_instruction * gen_andl_instruction (t_program_infos *program
    , int r_dest, int r_source1, int r_source2, int flags);

extern t_axe_instruction * gen_orl_instruction (t_program_infos *program
    , int r_dest, int r_source1, int r_source2, int flags);

```

```

extern t_axe_instruction * gen_eorl_instruction (t_program_infos *program
    , int r_dest, int r_source1, int r_source2, int flags);

extern t_axe_instruction * gen_andb_instruction (t_program_infos *program
    , int r_dest, int r_source1, int r_source2, int flags);

extern t_axe_instruction * gen_orb_instruction (t_program_infos *program
    , int r_dest, int r_source1, int r_source2, int flags);

extern t_axe_instruction * gen_eorb_instruction (t_program_infos *program
    , int r_dest, int r_source1, int r_source2, int flags);

extern t_axe_instruction * gen_mul_instruction (t_program_infos *program
    , int r_dest, int r_source1, int r_source2, int flags);

extern t_axe_instruction * gen_div_instruction (t_program_infos *program
    , int r_dest, int r_source1, int r_source2, int flags);

extern t_axe_instruction * gen_shl_instruction (t_program_infos *program
    , int r_dest, int r_source1, int r_source2, int flags);

extern t_axe_instruction * gen_shr_instruction (t_program_infos *program
    , int r_dest, int r_source1, int r_source2, int flags);

extern t_axe_instruction * gen_neg_instruction (t_program_infos *program
    , int r_dest, int r_source, int flags);

extern t_axe_instruction * gen_spcl_instruction (t_program_infos *program
    , int r_dest, int r_source1, int r_source2, int flags);

/*-----
 *                JUMP INSTRUCTIONS
 *-----*/
extern t_axe_instruction * gen_bt_instruction
    (t_program_infos *program, t_axe_label *label, int addr);

extern t_axe_instruction * gen_bf_instruction
    (t_program_infos *program, t_axe_label *label, int addr);

extern t_axe_instruction * gen_bhi_instruction
    (t_program_infos *program, t_axe_label *label, int addr);

extern t_axe_instruction * gen_bls_instruction
    (t_program_infos *program, t_axe_label *label, int addr);

extern t_axe_instruction * gen_bcc_instruction
    (t_program_infos *program, t_axe_label *label, int addr);

extern t_axe_instruction * gen_bcs_instruction
    (t_program_infos *program, t_axe_label *label, int addr);

extern t_axe_instruction * gen_bne_instruction
    (t_program_infos *program, t_axe_label *label, int addr);

extern t_axe_instruction * gen_beq_instruction
    (t_program_infos *program, t_axe_label *label, int addr);

extern t_axe_instruction * gen_bvc_instruction
    (t_program_infos *program, t_axe_label *label, int addr);

extern t_axe_instruction * gen_bvs_instruction
    (t_program_infos *program, t_axe_label *label, int addr);

extern t_axe_instruction * gen_bpl_instruction
    (t_program_infos *program, t_axe_label *label, int addr);

extern t_axe_instruction * gen_bmi_instruction
    (t_program_infos *program, t_axe_label *label, int addr);

```

```
extern t_axe_instruction * gen_bge_instruction
    (t_program_infos *program, t_axe_label *label, int addr);

extern t_axe_instruction * gen_blt_instruction
    (t_program_infos *program, t_axe_label *label, int addr);

extern t_axe_instruction * gen_bgt_instruction
    (t_program_infos *program, t_axe_label *label, int addr);

extern t_axe_instruction * gen_ble_instruction
    (t_program_infos *program, t_axe_label *label, int addr);
/*
    See also:
    axe_utils.h for gen_load_immediate()
*/
```

- axe\_labels.h

```
/* *****
 * axe_labels.h
 * AUXILIARY FUNCTIONS FOR LABEL MANAGEMENT
 * *****/
/* get the number of labels inside the list of labels */
extern int get_number_of_labels(t_axe_label_manager *lmanager);

/* return TRUE if the two labels hold the same identifier */
extern int compareLabels(t_axe_label *labelA, t_axe_label *labelB);

/* test if a label will be assigned to the next instruction */
extern int isAssignedLabel(t_axe_label_manager *lmanager);
/*
    See also:
    axe_engine.h for the main label-related functions
*/
```

- axe\_struct.h

```
/* *****
 * axe_struct.h
 * FUNDAMENTAL DATA STRUCTURES
 * *****/
typedef struct t_axe_label
{
    int labelID; /* label identifier */
} t_axe_label;

typedef struct t_axe_register
{
    int ID; /* an identifier of the register */
    int indirect; /* a boolean value: 1 if the register value is a pointer */
} t_axe_register;

typedef struct t_axe_address
{
    int addr; /* a Program Counter */
    t_axe_label *labelID; /* a label identifier */
    int type; /* one of ADDRESS_TYPE or LABEL_TYPE */
} t_axe_address;
```



```

/* A structure that defines the internal data of a 'Acse variable' */
typedef struct t_axe_variable
{
    int type;          /* a valid data type @see 'axe_constants.h' */
    int isArray;       /* must be TRUE if the current variable is an array */
    int arraySize;     /* the size of the array. This information is useful only
                       * if the field 'isArray' is TRUE */
    int init_val;      /* initial value of the current variable. Actually it is
                       * implemented as a integer value. 'int' is
                       * the only supported type at the moment,
                       * future developments could consist of a modification of
                       * the supported type system. Thus, maybe init_val will be
                       * modified in future. */
    char *ID;          /* variable identifier (should never be a NULL
                       * pointer or an empty string "") */
    t_axe_label *labelID; /* a label that refers to the location
                       * of the variable inside the data segment */
} t_axe_variable;

/* a symbolic assembly instruction */
typedef struct t_axe_instruction
{
    int opcode;        /* instruction opcode (for example: AXE_ADD) */
    t_axe_register *reg_1; /* destination register */
    t_axe_register *reg_2; /* first source register */
    t_axe_register *reg_3; /* second source register */
    int immediate;      /* immediate value */
    t_axe_address *address; /* an address operand */
    char *user_comment;  /* if defined it is set to the source code
                       * instruction that generated the current
                       * assembly. This string will be written
                       * into the output code as a comment */
    t_axe_label *labelID; /* a label associated with the current
                       * instruction */
} t_axe_instruction;

/* this structure is used in order to define assembler directives.
 * Directives are used in many cases such the definition of variables
 * inside the data segment. Every instance 't_axe_data' contains
 * all the informations about a single directive.
 * An example is the directive .word that is required when the assembler
 * must reserve a word of data inside the data segment. */
typedef struct t_axe_data
{
    int directiveType; /* the type of the current directive
                       * (for example: DIR_WORD) */
    int value;          /* the value associated with the directive */
    t_axe_label *labelID; /* label associated with the current data */
} t_axe_data;

typedef struct t_axe_expression
{
    int value;          /* an immediate value or a register identifier */
    int expression_type; /* actually only integer values are supported */
} t_axe_expression;

typedef struct t_axe_declaration
{
    int isArray;        /* must be TRUE if the current variable is an array */
    int arraySize;      /* the size of the array. This information is useful only
                       * if the field 'isArray' is TRUE */
    int init_val;       /* initial value of the current variable. */
    char *ID;           /* variable identifier (should never be a NULL pointer
                       * or an empty string "") */
} t_axe_declaration;

```

```

typedef struct t_while_statement
{
    t_axe_label *label_condition; /* this label points to the expression
                                   * that is used as loop condition */
    t_axe_label *label_end;       /* this label points to the instruction
                                   * that follows the while construct */
} t_while_statement;

/* create a label */
extern t_axe_label * alloc_label(int value);

/* create an expression */
extern t_axe_expression create_expression (int value, int type);

/* create an instance that will maintain infos about a while statement */
extern t_while_statement create_while_statement();

/* create an instance of 't_axe_register' */
extern t_axe_register * alloc_register(int ID, int indirect);

/* create an instance of 't_axe_instruction' */
extern t_axe_instruction * alloc_instruction(int opcode);

/* create an instance of 't_axe_address' */
extern t_axe_address * alloc_address(int type, int address, t_axe_label *label);

/* create an instance of 't_axe_data' */
extern t_axe_data * alloc_data(int directiveType, int value, t_axe_label *label)
;

/* create an instance of 't_axe_variable' */
extern t_axe_variable * alloc_variable
    (char *ID, int type, int isArray, int arraySize, int init_val);

/* finalize an instance of 't_axe_variable' */
extern void free_variable (t_axe_variable *variable);

/* create an instance of 't_axe_variable' */
extern t_axe_declaration * alloc_declaration
    (char *ID, int isArray, int arraySize, int init_val);

/* finalize an instruction info. */
extern void free_Instruction(t_axe_instruction *inst);

/* finalize a data info. */
extern void free_Data(t_axe_data *data);

```

- **collections.h**

```

/* *****
 * collections.h - A double linked list
 * ***** */
/* a list element */
typedef struct t_list
{
    void *data;
    struct t_list *next;
    struct t_list *prev;
} t_list;

/* add an element 'data' to the list 'list' at position 'pos'.*/
extern t_list * addElement(t_list *list, void * data, int pos);

/* add sorted */
extern t_list * addSorted(t_list *list, void * data
    , int (*compareFunc)(void *a, void *b));

/* add an element to the end of the list */
extern t_list * addLast(t_list *list, void * data);

/* add an element at the beginning of the list */
extern t_list * addFirst(t_list *list, void * data);

/* remove an element at the beginning of the list */
extern t_list * removeFirst(t_list *list);

/* remove an element from the list */
extern t_list * removeElement(t_list *list, void * data);

/* remove a link from the list 'list' */
extern t_list * removeElementLink(t_list *list, t_list *element);

/* find an element inside the list 'list'. The current implementation calls the
 * CustomfindElement' passing a NULL reference as 'func' */
extern t_list * findElement(t_list *list, void *data);

/* find an element inside the list 'list'. */
extern t_list * CustomfindElement(t_list *list, void *data
    , int (*compareFunc)(void *a, void *b));

/* find the position of an 'element' inside the 'list'. -1 if not found */
extern int getPosition(t_list *list, t_list *element);

/* find the length of 'list' */
extern int getLength(t_list *list);

/* remove all the elements of a list */
extern void freeList(t_list *list);

/* get the last element of the list. Returns NULL if the list is empty
 * or list is a NULL pointer */
extern t_list * getLastElement(t_list *list);

/* retrieve the list element at position 'position' inside the 'list'.
 * Returns NULL if: the list is empty, the list is a NULL pointer or
 * the list holds less than 'position' elements. */
extern t_list * getElementAt(t_list *list, unsigned int position);

/* create a new list with the same elements */
extern t_list * cloneList(t_list *list);

/* add a list of elements to another list */
extern t_list * addList(t_list *list, t_list *elements);

/* add a list of elements to a set */
extern t_list * addListToSet(t_list *list, t_list *elements
    , int (*compareFunc)(void *a, void *b), int *modified);

```

- axe\_utils.h

```
/* *****
 * axe_utils.h - Some important functions to access the list of symbols
 * ***** */
/* create a variable for each 't_axe_declaration' inside
 * the list 'variables'. Each new variable will be of type
 * 'varType'. */
extern void set_new_variables(t_program_infos *program
    , int varType, t_list *variables);

/* Given a variable/symbol identifier (ID) this function
 * returns a register location where the value is stored
 * (the value of the variable identified by 'ID').
 * If the variable/symbol has never been loaded from memory
 * to a register, first this function searches
 * for a free register, then it assign the variable with the given
 * ID to the register just found.
 * Once computed, the location (a register identifier) is returned
 * as output to the caller.
 * This function generates a LOAD instruction
 * only if the flag 'genLoad' is set to 1; otherwise it simply reserve
 * a register location for a new variable in the symbol table.
 * If an error occurs, get_symbol_location returns a REG_INVALID errorcode */
extern int get_symbol_location(t_program_infos *program
    , char *ID, int genLoad);

/* Generate the instruction to load an 'immediate' value into a new register.
 * It returns the new register identifier or REG_INVALID if an error occurs */
extern int gen_load_immediate(t_program_infos *program, int immediate);

/* Notify the end of the program. This function is directly called
 * from the parser when the parsing process is ended */
extern void set_end_Program(t_program_infos *program);
```

- symbol\_table.h

```
/* *****
 * symbol_table.h
 * Some important functions to manage the symbol table
 * ***** */
/* a symbol inside the sy_table. An element of the symbol table is composed by
 * three fields: <ID>, <type> and <Location>.
 * 'ID' is a not-NULL string that is used as key identifier for a symbol
 * inside the table.
 * 'type' is an integer value that is used to determine the correct type
 * of a symbol. Valid values for 'type' are defined into "sy_table_constants.h".
 * 'reg_location' refers to a register location (i.e. which register contains
 * the value of 'ID'). */
typedef struct
{
    char *ID;           /* symbol identifier */
    int type;           /* type associated with the symbol */
    int reg_location;   /* a register location */
}t_symbol;

/* put a symbol into the symbol table */
extern int putSym(t_symbol_table *table, char *ID, int type);

/* set the location of the symbol with ID as identifier */
extern int setLocation(t_symbol_table *table, char *ID, int reg);

/* get the location of the symbol with the given ID */
extern int getLocation(t_symbol_table *table, char *ID, int *errorcode);

/* get the type associated with the symbol with ID as identifier */
extern int getTypeFromID(t_symbol_table *table, char *ID, int type);

/* given a register identifier (location), it returns the ID of the variable
 * stored inside the register 'location'. This function returns NULL
 * if the location is an invalid location. */
extern char * getIDfromLocation(t_symbol_table *table
    , int location, int *errorcode);

/* See also:   axe_utils.h  for wrapper functions related to variables
              axe_array.h  for wrapper functions related to array variables */
```