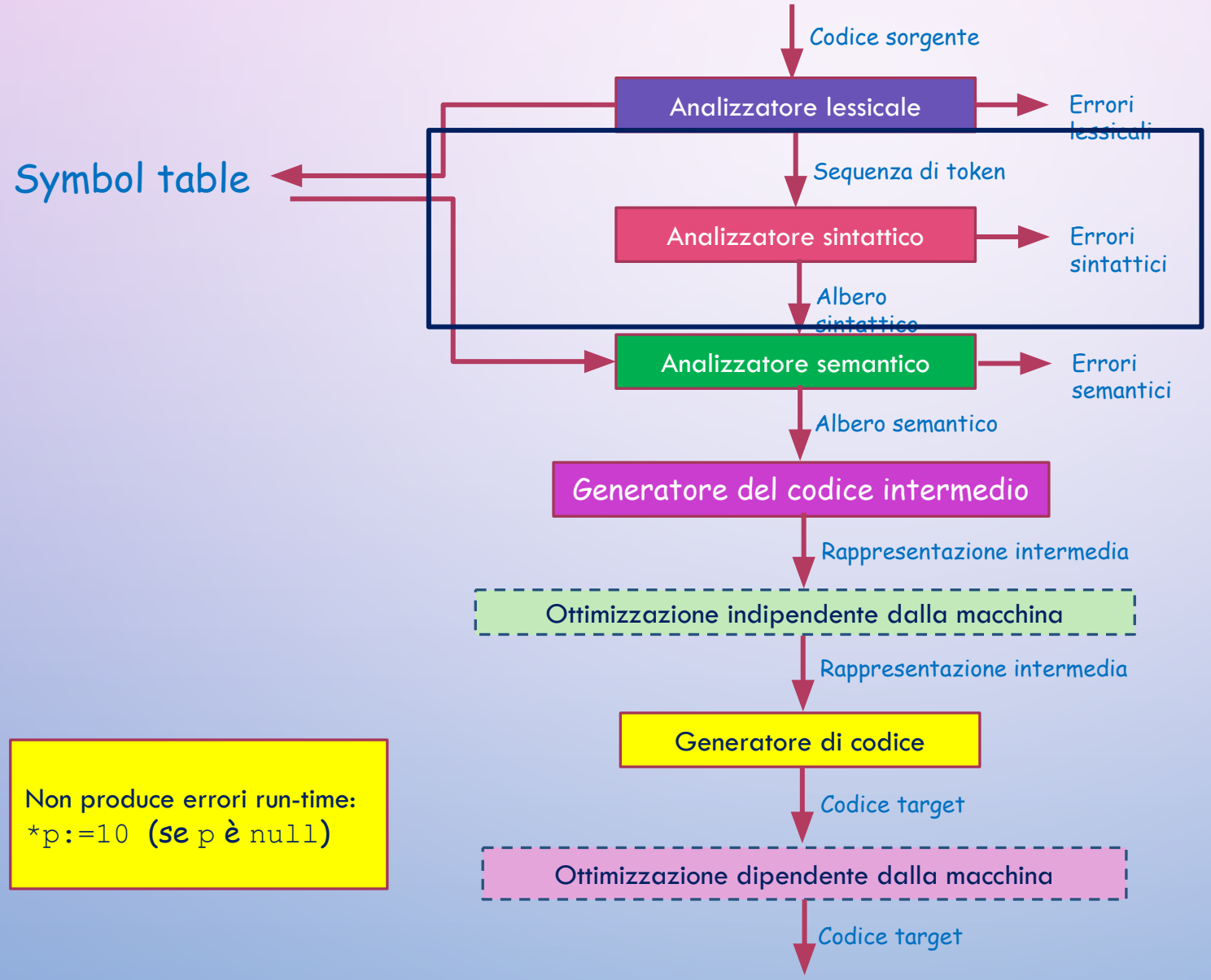
ANALISI SINTATTICA



Errore lessicale: simbolo non riconosciuto

Errore sintattico:
`If x:=3 then…`

Errore semantico:
`x:=y+s`
Con `x,y int` e
`s string`

Non produce errori run-time:
`*p:=10` (**se** `p` **è** `null`)

SINTASSI DI UN LINGUAGGIO DI PROGRAMMAZIONE

La sintassi dei costrutti di un comune linguaggio di Programmazione può essere descritta da una Grammatica context-free.

Un linguaggio si dice context-free se è generato da una Grammatica context-free.

GRAMMATICHE CONTEXT FREE

Una **grammatica context-free (CFG)** è una quadrupla G=(T,N,S,P) dove:

T è l'alfabeto dei simboli terminali (= token Lessicali);

N è l'alfabeto dei simboli intermedi o variabili o non terminali (= categorie grammaticali);

S ∈ N è l'assioma;

P è l'insieme delle produzioni o regole grammaticali della forma

$A \rightarrow \alpha$, dove $A \in N$ e $\alpha \in (N \cup T)^*$

Esempi :

- Instr → **if** expr **then** instr **else** instr
- Frase → soggetto verbo complemento

LINGUAGGIO GENERATO DA UNA GRAMMATICA

Il linguaggio generato da una grammatica *G* è l'insieme delle stringhe di simboli terminali ottenute a partire dall'assioma con una o più derivazioni.

Una derivazione consiste nell'applicazione di una sequenza di una o più produzioni:

$$\eta A \delta \Rightarrow^* \eta \alpha \delta$$

Dove η, α, δ sono stringhe in $T \cup N$ e $A \in N$ e $A \Rightarrow^* \alpha$ è una sequenza di regole di produzione della grammatica *G*, a partire dal simbolo non terminale A

ESEMPI

Grammatica che genera la struttura di un libro: G=(N,T,P,S) con N={S,A,B}, T={f,t,r}

- S->fA (f frontespizio, A serie di capitoli)
- A->AtB|tB (t titolo, B serie di righe)
- B->rB|r (r riga)

Il linguaggio generato è l'insieme di tutte le stringhe che rappresentano la struttura corretta di un libro, ftrrrrrtrrrtrr. Si noti che tale linguaggio è anche regolare, L=f(tr+)+

Tuttavia esistono linguaggi context-free non regolari, per esempio:

$L=\{a^n b^n, n>0\}$ è generato dalla grammatica S->aSb|ab

ESEMPI DI LINGUAGGI CF

Linguaggio finito:

- S->aBc B->ab|Ca C->c

L={aabc, acac}

Linguaggio infinito

G=({E,T,F}, {i,+,*,), (}, E, P)

1. E->E+T
2. E-> T
3. T-> T*F
4. T-> F
5. F-> (E)
6. F-> i

L={i, i+i, i+i+i, i*i, (i+i)*i,...}

Analizziamo tre diverse derivazioni:

$E \overset{1}{\Rightarrow} E+T \overset{2}{\Rightarrow} T+T \overset{4}{\Rightarrow} F+T \overset{6}{\Rightarrow} i+T \overset{3}{\Rightarrow} i+T*F \overset{4}{\Rightarrow} i+F*F \overset{6}{\Rightarrow} i+i*F \overset{6}{\Rightarrow} i+i*i$

$E \overset{1}{\Rightarrow} E+T \overset{3}{\Rightarrow} E+T*F \overset{6}{\Rightarrow} E+T*i \overset{4}{\Rightarrow} E+F*i \overset{6}{\Rightarrow} E+i*i \overset{2}{\Rightarrow} T+i*i \overset{4}{\Rightarrow} F+i*i \overset{6}{\Rightarrow} i+i*i$

$E \overset{1}{\Rightarrow} E+T \overset{3}{\Rightarrow} E+T*F \overset{2}{\Rightarrow} T+T*F \overset{4}{\Rightarrow} T+F*F \overset{6}{\Rightarrow} T+F*i \overset{4}{\Rightarrow} F+F*i \overset{6}{\Rightarrow} F+i*i \overset{6}{\Rightarrow} i+i*i$

Le tre derivazioni generano la stessa frase

ALBERI DI DERIVAZIONE

derivazione leftmost

$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow i+T \Rightarrow i+T*F \Rightarrow i+F*F \Rightarrow i+i*F \Rightarrow i+i*i$

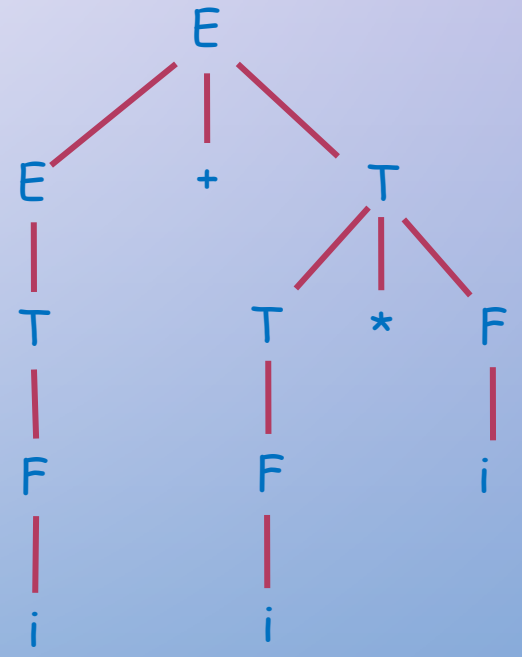
$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*i \Rightarrow E+F*i \Rightarrow E+i*i \Rightarrow T+i*i \Rightarrow F+i*i \Rightarrow i+i*i$

$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow T+T*F \Rightarrow T+F*F \Rightarrow T+F*i \Rightarrow F+F*i \Rightarrow F+i*i \Rightarrow i+i*i$

Un ***albero di derivazione*** è una rappresentazione ad albero di una derivazione, in cui

- La radice è etichettata con l'assioma
- Ogni simbolo non terminale è un nodo interno
- Ogni simbolo terminale è una foglia
- Ogni simbolo è connesso al simbolo che l'ha generato

Nel caso dell'esempio precedente lo stesso albero rappresenta le tre derivazioni precedenti dipendenti dall'ordine in cui effettuo le stesse derivazioni.



Ogni frase di una grammatica CF può essere generata da una derivazione sinistra (oppure destra), ossia con la sostituzione del simbolo non terminale più a sinistra (destra risp.).

Se esiste una frase per cui è possibile trovare due diverse derivazioni sinistre distinte (o equivalentemente due alberi sintattici diversi) la grammatica si dice **ambigua.**

GRAMMATICHE RIDOTTE

Una grammatica G si dice pulita o ridotta, se:

- 1. Ogni simbolo non terminale A è raggiungibile dall'assioma;
- 2. Ogni simbolo non terminale A genera un linguaggio non vuoto;
- 3. Non sono consentite derivazioni circolari: $A \Rightarrow^* A$

Data una grammatica CF, esiste un algoritmo per "ripulire" una grammatica.

TEOREMA: Data una grammatica G ridotta, condizione necessaria e sufficiente affinchè L(G) sia infinito è che G permetta derivazioni ricorsive, ovvero del tipo $A \Rightarrow^n xAy$

GRAMMATICHE AMBIGUE

Una grammatica si dice ambigua quando ci sono due alberi di derivazione o parsing differenti per la stessa frase (o, equivalentemente, due derivazioni leftmost o sinistre per la stessa frase)

Un linguaggio per cui esistono solo grammatiche ambigue si dice **inerentemente ambiguo**;

In generale stabilire se una data CFG sia ambigua o se un dato linguaggio sia inerentemente ambiguo sono problemi indecidibili.

Per alcune applicazioni si usano metodi che coinvolgono grammatiche ambigue unite alle regole che servono per eliminare le ambiguità.

I costrutti per cui il parsing è difficile coinvolgono per lo più regole di precedenza o associatività

GRAMMATICA AMBIGUA

$E \rightarrow n$

$E \rightarrow (E)$

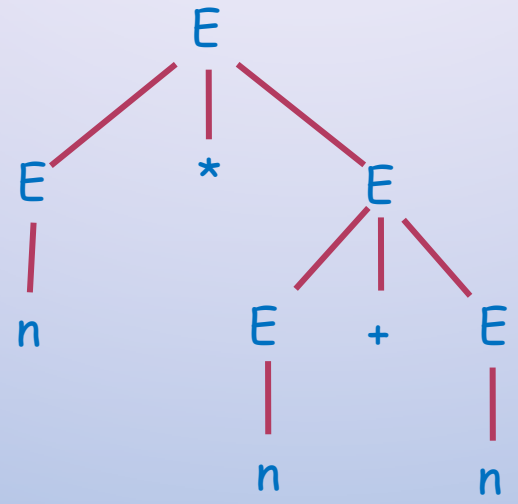
$E \rightarrow E+E$

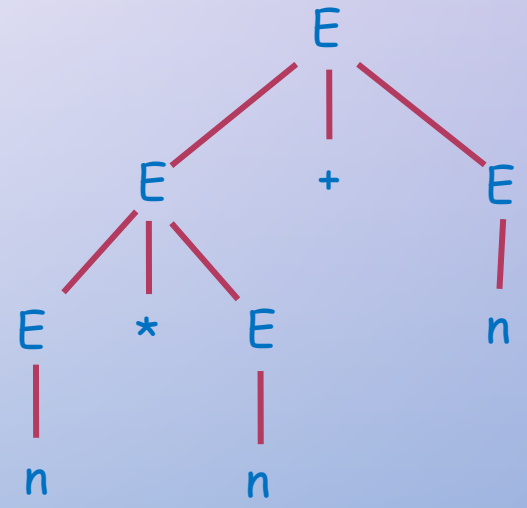
$E \rightarrow E-E$

$E \rightarrow E*E$

$E \rightarrow E/E$

Due diverse derivazioni per la frase n*n+n





ELIMINAZIONE DI AMBIGUITÀ

In alcuni casi particolari certe ambiguità possono essere eliminate.

1. Ci sono grammatiche che presentano una ricorsione sinistra e una ricorsione destra nella stessa regola.

   ESEMPIO:

   E->E+E|i

La frase i+i+i ha due derivazioni sinistre.

E' possibile eliminarla stabilendo un ordine di derivazione, per esempio:

E->i+E|i oppure E->E+i|i

ELIMINAZIONE DI AMBIGUITÀ

In altre grammatiche si possono avere una ricorsione sinistra e una destra in regole diverse.

ESEMPIO

A->aA|Ab|c L=a*cb*

In questo caso si può stabilire un ordine tra le derivazioni in maniera tale da far generare prima i rami di destra e poi quelli di sinistra:

S->aS|X

X->Xb|c

ELIMINARE L'AMBIGUITÀ

Grammatica non ambigua per espressioni aritmetiche (Operatori postfissi):

$E \rightarrow EE + \mid EE - \mid EE * \mid EE / \mid \mathbf{number}$

Grammatica non ambigua per espressioni aritmetiche (Operatori Infissi):

Scelta: associatività a sinistra, Precedenza di * e / su + e -

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid n$

ESEMPIO:
GRAMMATICA PER LINGUAGGIO JAVA

```
stmt -> id=espr ;|
      if (espr) stmt |
      if (espr) stmt else stmt |
      while (espr) stmt |
      do stmt while (espr); |
      {stmts}
stmts->stmts stmt |
     ε
```

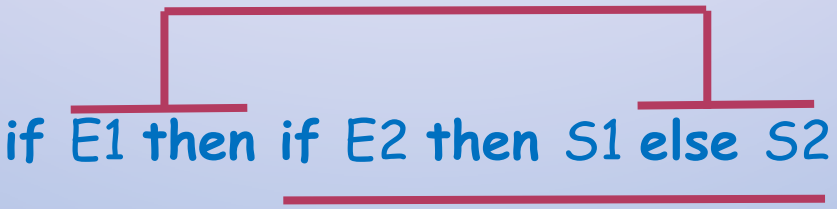
AMBIGUITÀ DELL'ELSE "PENDENTE"

Un comune tipo di ambiguità riguarda le frasi condizionali.

Si consideri la grammatica

stmt->**if** expr **then** stmt | **if** expr **then** stmt **else** stmt | **other**

Ad esempio:



ha due parse tree.

ELIMINAZIONE DELL'AMBIGUITÀ DELL'ELSE PENDENTE

Idea: basta distinguere gli statement in matched o completi (statement che contengono sia **then** che **else** e tale che sia dopo il then che dopo l'else ci siano statement matched) e statement open (statement con condizionali semplici o tali che il primo statement sia matched e il secondo open)

Solo gli statement matched possono precedere l'else. Quindi la grammatica diventa:

stmt-> matched_stmt | open_stmt

matched_stmt -> **if** expr **then** matched_stmt **else** matched_stmt | **Other**

open_stmt-> **if** expr **then** stmt | **If** expr **then** matched_stmt **else** open_stmt

LINGUAGGIO INERENTEMENTE AMBIGUO

Esistono tuttavia linguaggi che sono inerentemente ambigui, ossia tali che ogni grammatica che li genera è ambigua.

Esempio:

$L=\{a^i b^j c^k \ , i,j,k>=0$ con i=j oppure j=k$\}$

Le stringhe $a^i b^i c^i$ hanno necessariamente due alberi di derivazione distinti

Una grammatica:

S->XC|AY

X->aXb|ε

C->cC|ε

Y->bYc|ε

A->aA|ε

LINGUAGGI NON CONTEXT-FREE

Esistono anche linguaggi che non sono context free:

$L=\{wcw \mid w \in (a|b)^*\}$

Rappresenta il problema di controllare che gli identificatori siano dichiarati prima del loro uso.

$L=\{a^n b^m c^n d^m \mid n,m>0\}$

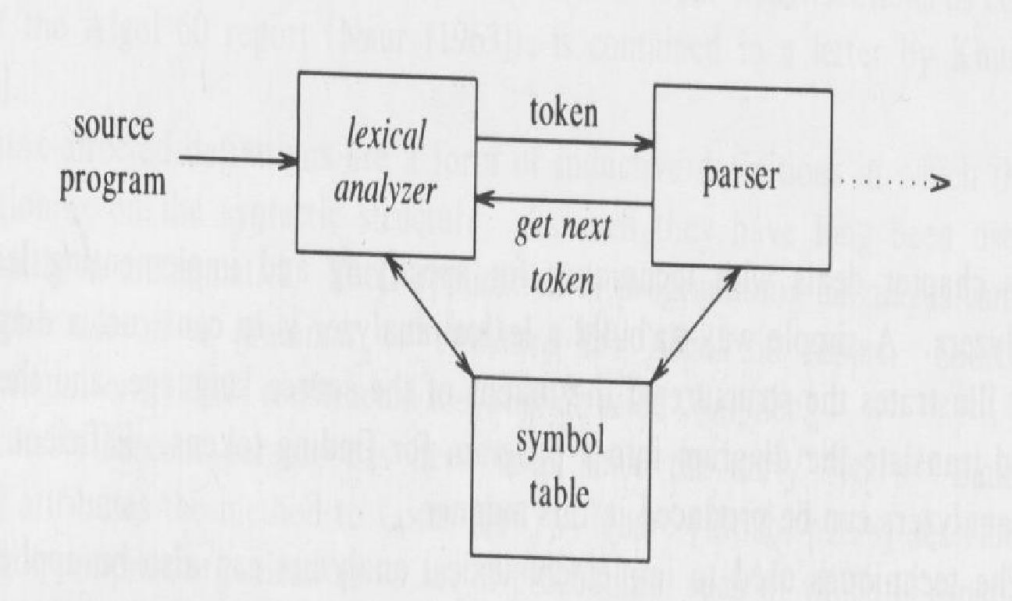
Rappresenta il problema di controllare il numero dei parametri formali di due funzioni coincida con quello dei parametri attuali delle rispettive funzioni.

COMPITO PRINCIPALE DEL PARSER

Data una grammatica CF che genera un linguaggio L e data una frase x, rispondere alla domanda: x appartiene a L?

- Se la risposta è sì, esibire un albero di derivazione di x
- Se la risposta è no, esibire eventuali errori sintattici

RUOLI DEL PARSER



- Svolge un ruolo centrale nel front-end:
- Attiva l'analizzatore lessicale con richieste
- Verifica la correttezza sintattica
- Costruisce l'**albero di parsing**
- Gestisce gli errori comuni di sintassi
- Raccoglie informazioni sui token nella symbol table
- Realizza alcuni tipi di analisi sematica
- ...

PERCHÉ USARE LE GRAMMATICHE

- Una grammatica fornisce una specifica della sintassi di un linguaggio di Programmazione in modo semplice e facile da capire
- A partire da certe classi di grammatiche è possibile costruire in modo automatico parser efficienti in grado di stabilire se un certo programma è ben formato.
- Un parser può rivelare alcune ambiguità sintattiche difficili da notare in fase di progettazione del linguaggio.
- Una grammatica progettata adeguatamente impartisce una struttura ad un linguaggio di programmazione che è utile per la traduzione in codice oggetto e per la ricerca degli errori.
- Aggiungere nuovi costrutti a linguaggi descritti mediante grammatiche è più facile.

RICONOSCERE UN LINGUAGGIO CONTEXT FREE

AUTOMI A PILA

GRAMMATICHE CF E AUTOMI A PILA

E' ben noto che la classe dei linguaggi CF coincide con quella dei linguaggi accettati da automi dotati di una memoria ausiliaria a pila.

Un primo algoritmo di riconoscimento potrebbe basarsi sulla simulazione di un automa a pila.

Tuttavia il modello da considerare è non deterministico poiché la classe dei linguaggi riconosciuti da automi a pila deterministici è strettamente inclusa in quella dei linguaggi CF.

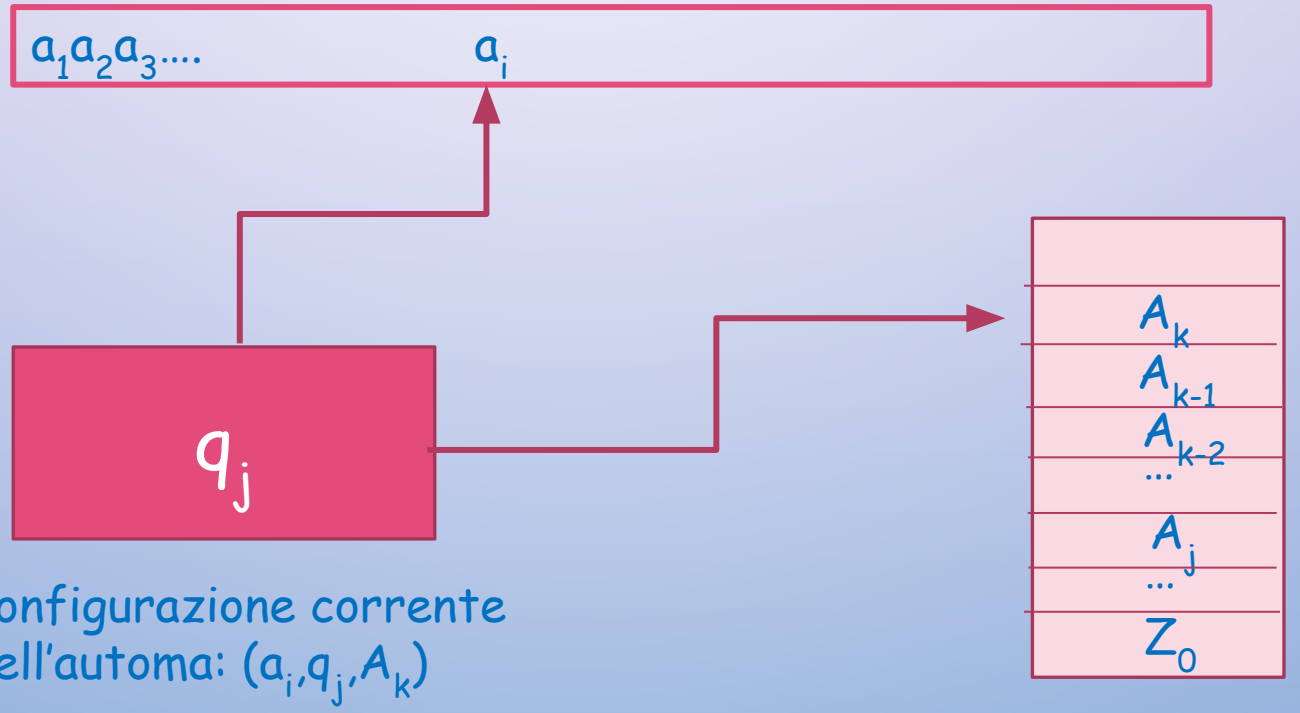
Ad ogni passo, l'automa sceglie in modo non deterministico una delle regole applicabili in funzione dello stato, del simbolo corrente e del simbolo in cima alla pila.

AUTOMI A PILA (PUSHDOWN AUTOMATA)

Sono automi finiti dotati di una memoria ausiliaria organizzata come una pila illimitata:

Operazioni:
Push(B) sulla pila;
test pila vuota;
Pop sulla pila;

Un insieme finito di stati Q



Configurazione corrente dell'automa: (a_i,q_j,A_k)

La pila è dotata di un simbolo iniziale Z_0, detto fondo. Conterrà all'inizio Z_0S, dove S è l'assioma.
Per ogni configurazione l'automa può cambiare stato e modificare la pila;

TRANSIZIONE IN UN PDA

Ad ogni passo il PDA:

- Legge il carattere corrente avanzando con la testina o compie una mossa spontanea senza muovere la testina
- Legge il simbolo in cima alla pila, lo estrae dalla pila (Pop) oppure legge il simbolo Z_0 se la pila è vuota
- In funzione dei valori dello stato corrente, del carattere corrente e del simbolo letto nella pila, calcola il nuovo stato
- Impila uno o più simboli (o nessuno)
- Può effettuare delle mosse anche con la sola lettura del top della pila, senza leggere la stringa di input (mosse spontanee)

DALLA GRAMMATICA CF AL PDA CON UN SOLO STATO

Regola	Mossa	Commento
$A \to B\alpha_1\alpha_2\ldots\alpha_n$	if testa=A then pop; (mossa spontanea) push($\alpha_n\ldots\alpha_1$B)	Per riconoscere A si devono riconoscere $B\alpha_1\alpha_2\ldots\alpha_n$

$A \to b\alpha_1\alpha_2\ldots\alpha_n$	if car=b and testa=A then
	pop;
	push($\alpha_n\ldots\alpha_1$);
	avanza testina lettura;
		Il primo carattere atteso era b ed è stato letto. Restano da riconoscere $\alpha_1\alpha_2\ldots\alpha_n$

$A \to \varepsilon$	if testa=A then pop;(mossa spontanea)	La stringa vuota che deriva da A è stata riconosciuta
Per ogni carattere b	if car=b and testa=b then pop; avanza testina lettura;	Il primo carattere atteso era b ed è stato letto
---	if car=\$ and testa=Z_0 then accetta; alt;	La stringa è stata scandita per intero e la pila è vuota
Tutti gli altri casi	reject	L'input non appartiene al linguaggio

ESEMPIO

$L=\{a^n b^m \mid n>=m>1\}$

	REGOLE	MOSSE
1.	S->aS	if car=a and testa=S then pop; push(S); avanza;
2.	S->A	if testa=S then pop; push(A);
3.	A->aAb	if car=a and testa=A then pop; push(bA); avanza;
4.	A->ab	if car=a and testa=A then pop; push(b);avanza;
5.	---	If car=b and testa=b then pop; avanza;
6.	---	If car=$ and testa=Z_0 then accetta; alt;

Osservazione:
L'automa ottenuto non è deterministico.
Infatti:

Esempio: aaabb
Al primo passo la scelta tra la prima e la seconda regola non è deterministica;

AUTOMI A PILA E GRAMMATICHE CF

- L'automa costruito riconosce una stringa se e solo se la grammatica la genera;
- L'automa simula le derivazioni sinistre della grammatica;
- Si dimostra che la famiglia dei linguaggi CF coincide con quella dei linguaggi riconosciuti da automi a pila non deterministici con un solo stato (ossia le transizioni dipendono solo dall'elemento in testa alla pila e dal carattere letto nel nastro)

COMPLESSITÀ DI CALCOLO DI UN PDA - LIMITE SUPERIORE

Se la grammatica è nella forma di normale di Greibach (ogni regola inizia con un terminale e non contiene altri terminali $A \to bY$ con $Y \in N^*$), allora non ci sono mosse spontanee e la pila non impila mai simboli terminali. Allora data una stringa x di lunghezza n, se la derivazione da S esiste, avrà lunghezza esattamente n.

Se K è il numero massimo di alternative per ogni non terminale A,

$A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_K$

allora ad ogni passo si hanno al più K scelte. Per esplorare tutte le scelte, si ha una complessità esponenziale $O(K^n)$

Si può fare molto meglio!

NON DETERMINISMO DI PDA

In un generico automa a pila sono presenti tre forme possibili di non-determinismo

- Incertezza tra più mosse di lettura: per stato q, carattere a e simbolo A della pila, $\delta(q,a,A)$ ha più di un valore;
- Incertezza tra una mossa spontanea (senza lettura) e una mossa di lettura;
- Incertezza tra più mosse spontanee;

Se nessuna delle tre forme è presente, l'automa a pila è deterministico, e il linguaggio riconosciuto è detto **context-free deterministico**

PARSER DETERMINISTICI E NON DETERMINISTICI

Un parser è l'implementazione di un automa a pila che riconosce il linguaggio generato da una certa grammatica.

Se un parser è deterministico allora ogni frase è riconosciuta o non riconosciuta con un solo calcolo (l'automa a pila è deterministico).

LINGUAGGI DETERMINISTICI E NON DETERMINISTICI

Un linguaggio riconosciuto da parser deterministico si dice **Linguaggio context-free deterministico** e può essere generato da una grammatica non ambigua.

La famiglia dei linguaggi context-free deterministici è strettamente contenuta in quella dei linguaggi context-free.

Se un linguaggio è inerentemente ambiguo allora nessun parser per il linguaggio può essere deterministico.

Esempio: $L=\{a^ib^jc^k$ con i=j oppure $j=k\}=\{a^ib^ic^* \mid i>=0\} \cup \{a^*b^ic^i \mid i>=0\}$.

Esistono linguaggi non ambigui ma il cui parsing è effettuato da Parser non deterministici (due calcoli diversi ma solo uno termina con successo):

Esempio: $L=\{a^nb^n \mid n>0\} \cup \{a^nb^{2n} \mid n>0\}$

DOMANDA

il linguaggio delle parole palindrome generato dalla grammatica?

S->aSa | bSb | a | b | ε

E' deterministico?

TIPI DI PARSER

Metodi universali

- (Cocke-Younger-Kasami 1967): usa la programmazione dinamica per stabilire se una data stringa appartiene ad un dato Linguaggio Context-Free. Richiede che la grammatica sia in Forma Normale di Chomsky (CNF). L'algoritmo richiede tempo $O(n^3)$.
- (Earley 1970): in grado di trattare qualsiasi grammatica CF. L'algoritmo richiede tempo $O(n^3)$.
- (Valiant 1975) Ha un interesse principalmente teorico in quanto dimostra che la complessità del problema del riconoscimento dei linguaggi CF non è $O(n^3)$, ma si riduce al calcolo del prodotto di matrici booleane, che può essere risolto con l'algoritmo di Strassen in $O(n^{2.81})$ o anche meno con algoritmi più recenti.

Il problema dell'esistenza di algoritmi quadratici per il riconoscimento dei linguaggi CF generici è ancora aperto!

TIPI DI PARSER

Metodi lineari

in O(n), su certe grammatiche riconosciute da Automi a pila deterministici:

- Analisi discendente (top-down), più intuitiva, adatta a Grammatiche semplici;
- Analisi ascendente (bottom-up), più sofisticata, più utilizzata dai Generatori automatici di analizzatori sintattici, poiché necessita di poche manipolazioni della grammatica.