

Introduzione

Tuesday, September 27, 2022 12:16 PM

LEZIONE 1 : https://virtuale.unibo.it/pluginfile.php/1354697/mod_resource/content/0/Lez01-AM22-23.pdf

1. Macchine astratte

lunedì 24 ottobre 2022 14:33

1.1 - Linguaggi e macchine

Un linguaggio e una macchina vengono insieme:

Ogni macchina fisica ha il suo linguaggio

(La macchina fisica della CPU ha come linguaggio gli 1 e 0 binari che gli arrivano, i segnali fisici che modificano lo stato dell'ALU, dei registri e della memoria in un modo o nell'altro).

Un linguaggio può essere eseguito su più macchine

A livello più base possibile, il ciclo "*fetch – decode – execute*" è un **interprete** che legge le istruzioni scritte in linguaggio macchina e le esegue.

1.2 - MACCHINA ASTRATTA

Una **macchina astratta** è l'insieme di strutture dati ed algoritmi in grado di memorizzare ed eseguire programmi.

Componente **essenziale** di una macchina astratta è l'**interprete**, che è costituito da:

- Operazioni per l'**elaborazione dei dati primitivi**
 - o MF: controllo e sfruttamento dell'ALU
- OP e Strutture dati per il **controllo di sequenza**:
 - o MF: incremento del PC, salti
- OP e SD per il **controllo del trasferimento di dati**:
 - o MF: gestione dei metodi di indirizzamento
- OP e SD per la **gestione della memoria**
 - o MF: indirizzamento e trasferimento di blocchi, etc.

1.3 - LINGUAGGI MACCHINA

- $M :=$ macchina astratta
- $L_M :=$ linguaggio macchina di M
- L_M è il linguaggio compreso dall'interprete di M
 - o I programmi sono particolari dati primitivi su cui opera l'interprete
- L_M può essere rappresentato in diversi modi:
 - o Internamente come struttura dati in memoria
 - o Esternamente come delle stringhe (il codice che noi scriviamo di solito, per esempio)

Poi seguono informazioni basilari sui seguenti argomenti:

- CISC e RICS
- Istruzioni a due operandi
- Macchine HW come macchine A
- Processori come macchine astratte (molto concrete)
- Micropogrammazione

Vediamo poi come la parte dell'interpretazione **viene fatta nel linguaggio della macchina ospite**. Quindi le strutture dati e gli algoritmi, sono effettivamente scritti nel linguaggio della macchina ospite.

È meno veloce ma è molto più flessibile.

Possiamo ricorsivamente definire macchine astratte via software in modo da creare linguaggi sempre più sofisticati, anche se più lenti.

In generale, viene definita una gerarchia all'interno di tutti i calcolatori per questi passaggi, ecco un esempio:



2. Implementazione di un linguaggio

lunedì 24 ottobre 2022 14:54

2.1 - Informazioni preliminari

\mathcal{P}_r^L è un programma \mathcal{P}_r scritto in un linguaggio L .

A questo è associato una funzione parziale \mathcal{P}^L .

Questa funzione associa vari input degli output.

Se non è definito per determinati input, allora quel valore non è associato ad alcun valore dell'output.

$$\mathcal{P}^L(\text{input}) = \text{output}$$

\mathcal{P}^L mantiene la semantica del programma \mathcal{P}_r .

2.2 - OBIETTIVO

Abbiamo L da implementare

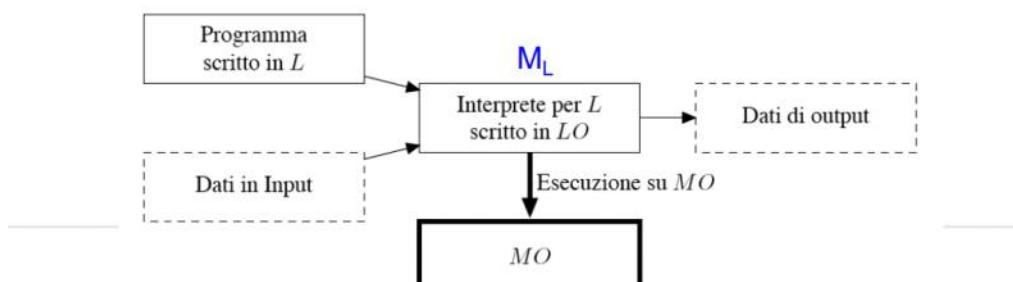
(Ovvero dobbiamo realizzare una MA chiamata M_L)

Abbiamo Mo_{Lo} , ovvero la macchina ospite che interpreta il linguaggio Lo .

Vogliamo implementare L su Mo_{Lo}

2.3 - SOLUZIONE 1 - Interpretazione

- M_L è realizzata scrivendo un *interprete* per L su Mo_{Lo} :



- Più formalmente

Definizione 1.3 (Interprete) Un interprete per il linguaggio L , scritto nel linguaggio Lo , è un programma che realizza una funzione parziale

$$\mathcal{I}_{\mathcal{L}}^{Lo} : (\text{Prog}^{\mathcal{L}} \times \mathcal{D}) \rightarrow \mathcal{D} \quad \text{tale che} \quad \mathcal{I}_{\mathcal{L}}^{Lo}(\mathcal{P}_r^{\mathcal{L}}, \text{Input}) = \mathcal{P}^{\mathcal{L}}(\text{Input}). \quad (1.1)$$

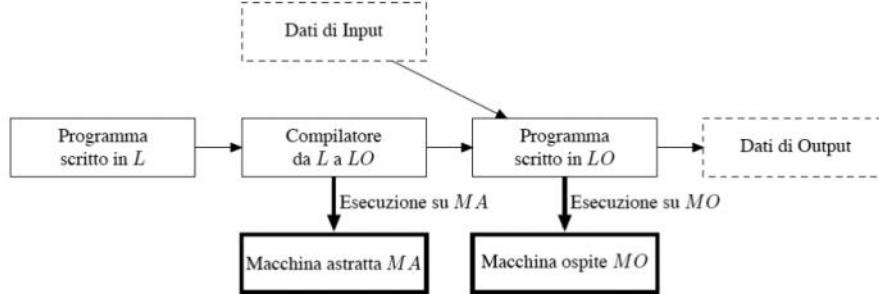
Ovvero l'interprete “calcola la corretta semantica” del programma!

2.4 - SOLUZIONE 2 - Compilatore

- I programmi in \mathcal{L} sono *tradotti* in programmi *equivalenti* in $\mathcal{L}o$
- Traduzione effettuata da un (altro) programma

$$C^{\mathcal{L}a}_{\mathcal{L}, \mathcal{L}o}$$

il **compilatore** da \mathcal{L} a $\mathcal{L}o$ scritto in $\mathcal{L}a$



- Più formalmente:

Definizione 1.4 (Compilatore) Un compilatore da \mathcal{L} a $\mathcal{L}o$ è un programma che realizza una funzione

$$\mathcal{C}_{\mathcal{L}, \mathcal{L}o} : \mathcal{P}rog^{\mathcal{L}} \rightarrow \mathcal{P}rog^{\mathcal{L}o}$$

tale che, dato un programma $\mathcal{P}_r^{\mathcal{L}}$, se

$$\mathcal{C}_{\mathcal{L}, \mathcal{L}o}(\mathcal{P}_r^{\mathcal{L}}) = \mathcal{P}_c^{\mathcal{L}o} \quad (1.2)$$

allora, per ogni Input $\in \mathcal{D}^5$,

$$\mathcal{P}^{\mathcal{L}}(\text{Input}) = \mathcal{P}_c^{\mathcal{L}o}(\text{Input}). \quad (1.3)$$

Ovvero il compilatore **“preserva la semantica”** del programma: il programma originale $\mathcal{P}_r^{\mathcal{L}}$ e quello tradotto $\mathcal{P}_c^{\mathcal{L}o}$ calcolano la stessa funzione: $\mathcal{P}^{\mathcal{L}} = \mathcal{P}_c^{\mathcal{L}o}$!

3. Esempi di esercizi e interpretazioni

lunedì 24 ottobre 2022 15:06

- $I^{\mathcal{L}_0}_{\mathcal{L}_1}$ = un interprete scritto in \mathcal{L}_0 che esegue programmi scritti in \mathcal{L}_1
- La macchina ospite è $M_{\mathcal{L}_0}$, mentre la macchina astratta realizzata su di essa dall'interprete è $M_{\mathcal{L}_1}$
- $I^{\mathcal{L}_0}_{\mathcal{L}_1}(P^{\mathcal{L}_1}, x)$ = risultato del calcolo del programma P (scritto in \mathcal{L}_1) con x come dato in input
- $I^{\mathcal{L}_0}_{\mathcal{L}_1}(I^{\mathcal{L}_1}_{\mathcal{L}_2}(P^{\mathcal{L}_2}, x))$ = risultato del calcolo di P (scritto in \mathcal{L}_2) con x come input
- La macchina ospite è $M_{\mathcal{L}_0}$, sulla quale è realizzata una macchina intermedia $M_{\mathcal{L}_1}$, grazie al primo interprete, mentre la macchina astratta $M_{\mathcal{L}_2}$, grazie al secondo interprete, è in grado di eseguire il programma P .

- $C^{\mathcal{L}_0}_{\mathcal{L}_1, \mathcal{L}_2}$ = un compilatore scritto in \mathcal{L}_0 che traduce programmi scritti in \mathcal{L}_1 in equivalenti programmi scritti in \mathcal{L}_2
- $I^{\mathcal{L}_0}_{\mathcal{L}_1}(C^{\mathcal{L}_1}_{\mathcal{L}_2, \mathcal{L}_3}, P^{\mathcal{L}_2}) = P_1^{\mathcal{L}_3}$ cioè l'interprete, eseguito sulla macchina ospite $M_{\mathcal{L}_0}$, realizza la macchina astratta $M_{\mathcal{L}_1}$ ed esegue un compilatore (scritto in \mathcal{L}_1) che traduce il programma P (scritto in \mathcal{L}_2) in un equivalente programma P_1 (scritto in \mathcal{L}_3)

$$\bullet I^{\mathcal{L}o}_{\mathcal{L}1}(C^{\mathcal{L}1}_{\mathcal{L}2,\mathcal{L}3}, C^{\mathcal{L}2}_{\mathcal{L}3,\mathcal{L}0}) = C^{\mathcal{L}3}_{\mathcal{L}3,\mathcal{L}0}$$

$$\bullet I^{\mathcal{L}o}_{\mathcal{L}1}(C^{\mathcal{L}1}_{\mathcal{L}0,\mathcal{L}2}, I^{\mathcal{L}o}_{\mathcal{L}1}) = I^{\mathcal{L}2}_{\mathcal{L}1}$$

$$\bullet I^{\mathcal{L}o}_{\mathcal{L}1}(C^{\mathcal{L}1}_{\mathcal{L}2,\mathcal{L}0}, I^{\mathcal{L}2}_{\mathcal{L}1}) = I^{\mathcal{L}o}_{\mathcal{L}1}$$

Attenzione! L'interprete ottenuto non è lo stesso interprete da cui siamo partiti, ma solo equivalente.

$$\bullet I^{\mathcal{L}o}_{\mathcal{L}1}(C^{\mathcal{L}2}_{\mathcal{L}0,\mathcal{L}1}, C^{\mathcal{L}o}_{\mathcal{L}2,\mathcal{L}1}) = \text{errore}$$

non si può perché l'interprete non può eseguire un compilatore scritto in L2.

mercoledì 31 maggio 2023 14:34

Domande orale di questo capitolo

mercoledì 31 maggio 2023 14:34

1. Che cosa è una macchina astratta? E in cosa si differenzia da una macchina fisica?

Dato un linguaggio di programmazione L, una macchina astratta M_L è un insieme di strutture dati e algoritmi per memorizzare ed eseguire programmi scritti in L.

Una macchina fisica è una particolare macchina astratta le cui strutture dati e algoritmi sono circuiti, porte logiche e dispositivi elettronici.

2. Che cos'è un interprete? In cosa consiste il ciclo fetch-decode-execute?

Un interprete scritto in un linguaggio L_0 e che interpreta un linguaggio L_1 è un programma che realizza una funzione parziale del tipo:

$$I_{L_1}^{L_0}(Programma^{L_1} \times D) \rightarrow D \quad \text{tale che} \quad I_{L_1}^{L_0}(P^{L_1}, D) = P^{L_1}(D)$$

L'interprete è il componente di una macchina astratta che serve per eseguire istruzioni. In particolare, contiene sistemi di gestione della memoria, controllo dei dati e controllo di flusso.

Il ciclo FDE è il ciclo eseguito da un interprete che consiste nel prelevare, decodificare ed eseguire un'istruzione.

3. Cos'è il linguaggio macchina?

Data una macchina astratta A, il linguaggio compreso dal suo interprete (L) è il suo linguaggio macchina.

4. Possono esistere macchine diverse con lo stesso linguaggio macchina?

Sì, in quanto dipende dal modo in cui l'interprete è implementato e le strutture dati che utilizza. Nel caso di una macchina hardware (come macchina astratta) si possono avere processori diversi che implementano lo stesso set di istruzioni.

5. In quali modi è possibile implementare una macchina astratta?

Hardware, software e firmware:

Realizzazione con **hardware**:

- È molto complicato implementare i costrutti di alto livello ma rende l'esecuzione di un programma scritto nel linguaggio della macchina astratta estremamente veloce, a scapito della flessibilità nel caso si dovesse modificare il linguaggio, perché significherebbe modificare l'intera architettura della macchina fisica.
- Lo si usa per linguaggi **di basso livello**.

Realizzazione con **software**:

- Ottima flessibilità ma bassa velocità. Comunque si basa su altre macchine astratte che dovranno essere implementate per poterlo effettivamente eseguire.

Realizzazione con **firmware**:

- Si tratta di una via di mezzo. Si ha una velocità maggiore del software ed una flessibilità maggiore dell'hardware, ma non una velocità maggiore dell'hardware e una flessibilità maggiore del software.
- Possibile grazie a microprogrammi scritti in linguaggi basso

6. Che cos'è un compilatore?

Un compilatore da L_0 a L_1 è un programma che realizza la funzione:

$$C_{L_0, L_1} : \text{Prog}_{L_0} \rightarrow \text{Prog}_{L_1}$$

Tale che, dato un programma P^{L_0} , se

$$C_{L_0, L_1}(P^{L_0}) = P^{L_1}$$

Allora, per ogni input, si ha:

$$P^{L_0}(\text{Input}) = P^{L_1}(\text{Input})$$

7. Relativamente alla tecnica d'implementazione software, descrivere la tecnica d'implementazione interpretativa pura e quella compilativa pura.

Dato un linguaggio L_1 e una macchina astratta ospite per L_o , un'impelmentazione può essere:

- **Compilativa pura** se si fornisce un compilatore C_{L_1, L_o} (Che viene eseguito su una qualsiasi macchina astratta in nostro possesso, non necessariamente M_o). Ogni istruzione di L_1 viene tradotta in una o più istruzioni di L_o . In questo caso, quindi, la traduzione avviene interamente **prima** della fase di esecuzione.
- Interpretativa pura se si fornisce un interprete $I_{L_1}^{L_o}$, quindi scritto in linguaggio L_o e che interpreta L_1 . Non vi è traduzione ma solo decodifica, che avviene durante l'esecuzione. Ad ogni istruzione L_1 viene assegnata una insieme di istruzioni L_o che vengono eseguite **direttamente**.

8. Quando un interprete si può dire corretto? Quando un compilatore si può dire corretto?

Interprete o compilatori si possono dire corretti se preservano la semantica del linguaggio che hanno tradotto nel linguaggio nel quale traducono. Questo vuol dire che devono interpretare, o compilare, in un linguaggio "risultante" con uguale o maggiore espressività, in generale ci basta considerare la Turing Completezza (considerata la memoria infinita).

9. Confrontare l'implementazione di una macchina astratta su una macchina ospite per mezzo di un interprete o di un compilatore.

Se si sceglie l'**implementazione interpretativa** si ha una velocità minore, in quanto ai tempi di esecuzione del programma stesso si aggiungono i tempi di traduzione del codice sorgente. Non viene generato codice, in quanto il codice tradotto viene eseguito direttamente. Si può debuggare più facilmente in quanto si può capire qual è l'istruzione che è stata decodificata per ultima e cosa in questo ha dato un problema a livello di runtime.
Si ha più flessibilità, per esempio per modificare a runtime il funzionamento del programma.

Se si sceglie invece l'**implementazione compilativa** si ha una maggiore velocità di esecuzione e ogni istruzione viene tradotta una sola volta, in fase appunto di compilazione. Si vanno a perdere dei vantaggi, come la possibilità di fare debugging in modo facile in quanto si vanno a perdere informazioni sul codice sorgente.

10. Come vengono implementate nella realtà le macchine astratte? Che cos'è la macchina intermedia?

Nella realtà, non si usa un approccio interpretativo o compilativo puro, bensì si va a fare un'operazione di questo tipo:

Sia L il linguaggio del programma che dobbiamo eseguire e sia L_F il linguaggio della macchina fisica che abbiamo, noi: compiliamo L in L_i , dove L_i è un linguaggio intermedio, di una macchina astratta che si interpone tra quella fisica e quella astratta del linguaggio L , e poi usiamo un interprete da L_i a L_F .

11. Quando si dice che una implementazione è di tipo interpretativo e quando ti tipo compilativo? Fare esempi di linguaggi la cui implementazione è di un tipo o dell'altro.

Si dice che sia di tipo interpretativo quando il linguaggio della macchina intermedia è molto lontano da quello della macchina fisica, ovvero c'è ancora lavoro di decoding da fare e da gestire e pertanto molto del carico ricade sull'interprete.

Un linguaggio che ha questo tipo di approccio è Java e la sua JVM, la quale è molto diversa dal linguaggio della macchina fisica sulla quale si appoggia e pertanto deve fare ancora un grande lavoro di traduzione in fase di esecuzione.

Si dice che sia invece di tipo compilativo quando il linguaggio della macchina intermedia è molto vicino, se non identico, a quello della macchina fisica. Di solito il linguaggio della macchina intermedia va ad aggiungere funzioni come la gestione dell'I/O che sono al di fuori di quello che la macchina fisica ha nel suo di linguaggio.

Un esempio è il C, il quale va ad eseguire una compilazione e durante l'esecuzione il binario generato è essenzialmente identico a quello della macchina fisica, se solo non fosse per queste gestioni che si devono aggiungere durante runtime.

12. L'interprete e il compilatore si possono sempre realizzare?

Sì, a patto che di avere a disposizione un linguaggio "espressivo" almeno tanto quanto quello da implementare. Questo si collega alla Turing-Completezza.

13. Che cos'è l'implementazione via kernel?

L'implementazione via kernel centra con un modo di generare un compilatore. Se si deve implementare il linguaggio L in un linguaggio della macchina ospite L_0 :

- Si implementa un opportuno sottoinsieme H di L
- Si usa H come linguaggio di implementazione

Si implementa poi un compilatore oppure un interpretatore del tipo:

- C_{L,L_0}^H
- I_L^H

A questo punto, si deve implementare il linguaggio H piuttosto che l'intero linguaggio L, che, in quanto H è più piccolo, risulta molto più semplice.

Questo approccio è comune nei sistemi operativi, che creano inizialmente un nucleo e poi l'intero sistema usando le primitive offerte dal nucleo.

Ci semplifica l'implementazione in quanto H è più vicino di L al linguaggio della macchina ospite e ci facilità la portabilità, in quanto basta re implementare H in un diverso linguaggio macchina.

14. Quando si parla di bootstrapping?

Si tratta di usare compilatori per creare nuovi compilatori, i quali compilano diversamente da quelli che li hanno generati.

Essenzialmente ci si riduce a creare (e necessitare) solo determinati strumenti (ovvero interpreti e compilatori) e il resto li andiamo a generare in modo "automatico" attraverso la compilazione dei compilatori.

Questo ci permette anche di creare nuovi compilatori in modo "automatico" in modo più facile, in quanto ci basta cambiare anche solo uno dei compilatori che noi abbiamo a disposizione (e facendolo in modo semplice, ovvero non andando magari a scrivere codice in binario ma comunque usando un linguaggio di alto livello).

"Il compilatore è scritto nel linguaggio che si intende compilare, poi viene tradotto usando interpreti o altri compilatori."

Introduzione

Tuesday, September 27, 2022 12:16 PM

SLIDES (Lez 1):

https://virtuale.unibo.it/pluginfile.php/1295149/mod_resource/content/0/Lez01-Gorrieri.pdf

SLIDES (Lez 2):

https://virtuale.unibo.it/pluginfile.php/1295150/mod_resource/content/0/Lez02-Gorrieri.pdf

SLIDES (Lez 3):

https://virtuale.unibo.it/pluginfile.php/1295151/mod_resource/content/0/Lez03-Gorrieri.pdf

SLIDES (Lez 4):

https://virtuale.unibo.it/pluginfile.php/1295152/mod_resource/content/0/Lez04-Gorrieri.pdf

SLIDES (Lez 5):

https://virtuale.unibo.it/pluginfile.php/1295153/mod_resource/content/0/Lez05-Gorrieri.pdf

1. Linguaggi (Naturali o Artificiali)

Tuesday, September 27, 2022 12:17 PM

1.1 Linguaggi (Naturali o Artificiali)

La descrizione di un linguaggio avviene su 3 dimensioni:

- **Sintassi:**
 - Regole di formazione
 - Quando una frase è corretta?
 - "Relazione tra segni"
- **Semantica**
 - Attribuzione di significati
 - Cosa significa una frase corretta?
 - "Relazione tra segni e significati"
- **Pragmatica**
 - In quale modo frasi corrette e sensate sono usate
 - "Relazione tra segni, significati e utente"

Per un linguaggio eseguibile è importante anche:

- **Implementazione**
 - Come eseguire una frase corretta, rispettando la semantica

1.2 - LA SINTASSI

Ha diversi aspetti, in particolare:

- Aspetto lessicale
 - Ovvero le parole che si possono usare
 - Descrizione del lessico:
 - Dizionari (per le lingue naturali)
 - Strutture più complesse per i linguaggi artificiali
 - Errore? Vocabolo inesistente
- Aspetto grammaticale
 - Frasi corrette si possono costruire con il lessico
 - Descrizione attraverso regole grammaticali (in numero finito)
 - Le frasi che possiamo costruire usando queste regole sono infinite
 - Errore? Frase scorretta
 - Anche se il lessico è corretto, la grammatica può essere sbagliata

1.3 - LA SEMANTICA

Se pensiamo ai linguaggi naturali, il primo mattoncino che ci possiamo immaginare esista è un dizionario. Quindi:

- Per il lessico
 - Dizionario

Successivamente, ci sono altre cose da considerare per la corretta comprensione del linguaggio:

- Per le frasi, devo sapere:
 - A quale linguaggio appartiene quello che sto interpretando?
 - Su quale linguaggio basarmi per dare significato?
 - Bisogna basarsi su qualcosa di noto, cioè che non abbia a sua volta bisogno di essere spiegato

- Per esempio, se noi dobbiamo comprendere il cinese, lo potremmo tradurre in italiano e questo è un linguaggio "noto", di cui noi riusciamo a comprendere il significato.
- Essenzialmente, non possiamo andare a ricorsione infinita per la comprensione di un linguaggio, ma dobbiamo avere una base dalla quale comprendere.
- (Essenzialmente, si intende il linguaggio macchina nel calcolatore)
 - All'interno di un linguaggio di programmazione sequenziale si dà significato associando un input-output ad una frase
 - Quindi l'insieme delle frasi corrette in L vengono associate ad un insieme di funzioni matematiche, per esempio

1.4 - PRAGMATICA

- Insieme di regole che guidano l'uso
 - Ovvero, noi utilizziamo norme di programmazione
 - Per esempio, noi di solito non usiamo il goto, anche se è disponibile in molti linguaggi, oppure non modifichiamo l'indice del for dentro la funzione, per evitare confusione.
 - Per esempio usare il "lei" piuttosto che il "tu" quando ci si riferisce a qualcuno che non si conosce

1.5 - IMPLEMENTAZIONE

Eseguire una frase sintatticamente corretta rispettando la semantica.

Per esempio, un compilatore deve preservare la semantica del linguaggio che sta compilando.

1.6 - LESSICO E FRASI DI UN LINGUAGGIO

Alfabeto:

- Un insieme finito di simboli
 - (a, b, c, ...)

Con l'alfabeto si compone un **lessico**:

- Insieme di sequenze finite (parole) costituite con caratteri (simboli) dell'alfabeto
- (OSSERVAZIONE: Lessico è un alfabeto per frasi)

Con il lessico, io posso costruire delle **frasi**:

- Un insieme di sequenze finite costituite con parole del lessico
 - Tipicamente infinito contabile

Definiamo quindi cos'è un **linguaggio formale**:

- Un alfabeto A è un insieme finito i cui elementi sono detti simboli
- Una parola su alfabeto A è una sequenza finita di simboli di A
- Un linguaggio formale L su A è un insieme di parole su A

Definizione di linguaggio formale L (matematica):

- Un linguaggio formale L su alfabeto A è un sottoinsieme A^* ($L \subseteq A^*$) dove

$$A^* = \left(\bigcup_{n \geq 0} A^n \right), \quad \text{dove } A^0 = \left\{ \begin{array}{l} \xi \\ \text{(sequenza senza} \\ \text{alcun simbolo)} \end{array} \right\}$$

E poi abbiamo $A^{n+1} = A \cdot A^n, \quad n \geq 0$

$$A \quad \stackrel{\cdot}{\underset{\text{concatenamento}}{\cdot}} \quad A^n = \{aw \mid a \in A \wedge w \in A^n\}$$

Osservazione:

A^* è un insieme infinito contabile.

Dato un ordinamento $<$ sui simboli di A (ad esempio $a < b < c < \dots$), possiamo elencare tutte le parole di A^* come segue:

- Prima elenco tutte le parole vuote ε (A^0)
- Poi elenco le parole di lunghezza 1 (A^1) secondo l'ordinamento $<$
 - a, b, c, \dots
- Poi elenco le parole in A^2 secondo $<$
 - $aa, ab, ac, \dots, ba, bb, bc, \dots$
- Poi elenco le parole in A^3 secondo $<$
 - ...
- E così via

Se ammettessimo alfabeti con un numero infinito di simboli?

$$A = \{a_0, a_1, a_2, \dots\}$$

A^* sarebbe ancora contabile? Cioè è possibile elencare tutte le possibili parole di A^* ?

La risposta è affermativa.

A è in corrispondenza biunivoca con \mathbb{N} .

- $\mathbb{N} \times \mathbb{N}$ è numerabile (dovetailing)
 - $f^2: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, f^2 è biunivoca
- \mathbb{N}^k è numerabile
 - Si può vedere sulle slide
- $\mathbb{N}^* = \bigcup_{n \geq 0} \mathbb{N}^k$ è numerabile
- Per le definizioni rigorose, vedere le slides

In quanto **L è infinito, come possiamo rappresentarlo?**

(E come possiamo capire se una parola appartiene a L?)

Utilizziamo due tecniche:

- **Generativo** / Sintetico:
 - Linguaggio := insieme delle stringhe generate da una struttura finita detta **GRAMMATICA** (o struttura di fase)
- **Riconoscitivo** / Analitico:
 - Linguaggio := Insieme delle stringhe riconosciute da una struttura finita della **AUTOMA**

Da notare che A^* è infinito contabile.

$L \subseteq A^*$ è anch'esso infinito contabile.

Ma $P(A^*)$, ovvero l'insieme di tutti i linguaggi su alfabeto A , è equipotente a \mathbb{R} .

Un "formalismo" è un linguaggio su un alfabeto \Rightarrow i suoi "programmi" (grammatiche/automati) sono numerabili.

1.7 - NOTAZIONI E DEFINIZIONI AUSILIARIE (PAROLE)

- **Lunghezza** di una parola (o stringa)
 - $|\varepsilon| = 0$, $|aw| = 1 + |w|$, es: $|abc| = 3$
- **Concatenazione**

- x concatenato a y , ovvero xy , è la parola ottenuta giustapponendo x e y .
- $w = xy \Leftrightarrow \begin{cases} |w| = |x| + |y| \\ w(j) = x(j) & \text{per } 1 \leq j \leq |x| \\ w(|x| + j) = y(j) & \text{per } 1 \leq j \leq |y| \end{cases}$
($w(j)$ indica il j -esimo simbolo di w)

- **Sottostringa**

- v sottostringa di $w \Leftrightarrow \exists x, y \in A^*$ tali che $w = xvy$
 - (x o y possono essere ϵ)
 - Ogni stringa è sottostringa di se stessa
 - ϵ è sottostringa di tutte le stringhe

- **Suffisso e prefisso**

- v è suffisso di $w \Leftrightarrow \exists x \in A^*. w = xv$
- v è prefisso di $w \Leftrightarrow \exists x \in A^*. w = vx$

- **Potenza n -esima:**

- $w^0 = \epsilon, w^1 = w, w^2 = ww, \dots$
- $(ab)^3 = ababab, \dots$

- **Linguaggio L su alfabeto A :**

- $L \subseteq A^*$
Se $A = \{a\}$, allora $\emptyset, \{\epsilon\}, \{a, aaa\}$ sono linguaggi finiti.
 $L_1 = \{a^n \mid n \geq 0\} = \{\epsilon, a, aa, aaa, \dots\}$ è un linguaggio infinito

1.8 - NOTAZIONI E DEFINIZIONI AUSILIARIE (LINGUAGGI)

- **Complemento**

- $\bar{L} = \{w \in A^* \mid w \notin L\} = A^* \setminus L$

- **Unione e intersezione**

- $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$
- $L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$

- **Concatenazione**

- $L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$
- Osservazione: concatenazione \neq prodotto cartesiano
 - Il prodotto cartesiano tra $L_1 \times L_2 = \{(w_1, w_2) \mid w_1 \in L_1 \wedge w_2 \in L_2\}$
 - La concatenazione $L_1 \cdot L_2 = w_1$

- **Potenza di un linguaggio**

- $L^0 = \{\epsilon\}, L^{n+1} = L \cdot L^n, \forall n \geq 0$

- **Chiusura / Stella di Kleene / Ripetizione**

- $L^* = \bigcup_{n \geq 0} L^n$
- $L^+ = \bigcup_{n \geq 1} L^n$, (chiusa positiva)

2. Definizione finita di un linguaggio

Friday, October 14, 2022 12:26 PM

2.1 - LINGUAGGIO PALINDROMO

Supponiamo di voler creare un linguaggio composto da tutte le stringhe definibili palindrome, ovvero che lette da destra a sinistra o da sinistra a destra sono uguali. Supponiamo quindi:

$$A = \{a, b\}, \quad L = \{\varepsilon, a, b, aa, bb, aba, bab, aaa, bbb, \dots\}$$

Abbiamo diversi modi attraverso i quali definire questo linguaggio:

- Backus-Naum Form (BNF):

$$\langle P \rangle ::= \varepsilon \mid a \mid b \mid a\langle P \rangle a \mid b\langle P \rangle b$$

- Come grammatica:

- $P \rightarrow \varepsilon$
 - $P \rightarrow a$
 - ...
 - $P \rightarrow bPb$

Oppure più brevemente:

- $P \rightarrow \varepsilon \mid a \mid b \mid aPa \mid bPb$
 - È una definizione ricorsiva in cui:
 - P è un **simbolo non terminale**
 - ε, a, b sono **simboli terminali**

- Insieme di assiomi e regole di inferenza:

$$\overline{\varepsilon \in L(P)}, \quad \overline{a \in L(P)}, \quad \overline{b \in L(P)}$$

$$\frac{W \in L(P)}{aWa \in L(P)}, \quad \frac{W \in L(P)}{bWb \in L(P)}$$

$L(P)$ = "Linguaggio generato a partire da nonterminale P "

È il più piccolo linguaggio generato da questi assiomi e da queste regole di inferenza.

2.2 - GRAMMATICHE

Definizione di una **grammatica libera da contesto**:

$$(NT, T, R, S) \text{ dove}$$

- $NT :=$ Non terminale
 - $T :=$ Terminale
 - $R :=$ Regole
 - È un insieme finito di produzioni (o regole) della forma $V \rightarrow w : V \in NT \wedge w \in (T \cup NT)^*$

- $S :=$ Starting point

Esempio:

$$G = (\{S, A, B\}, \{a, b\}, R, S), \quad \text{Con } R = \left\{ \begin{array}{l} S \xrightarrow{1} AB \\ A \xrightarrow{2} aA \\ A \xrightarrow{3} a \\ B \xrightarrow{4} bB \\ B \xrightarrow{5} b \end{array} \right\}$$

Quindi ha delle regole che sono (meno formalmente):

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid b \end{aligned}$$

2.3 - DERIVAZIONI

Data $G = (NT, T, R, S)$ libera da contesto diciamo che **da v si deriva immediatamente w** (o anche "v si riscrive in un passo w") e lo denotiamo con $v \Rightarrow w$, se:

$$\frac{v = xAy \quad (A \rightarrow z) \in R \quad w = xzy}{v \Rightarrow w} \quad \text{tale che } x, y, z \in (T \cup NT)^*$$

Diciamo che **da v si deriva w** (o anche "v si riscrive in w"), e lo indichiamo con $v \Rightarrow^* w$, se esiste una sequenza finita (eventualmente vuota) di derivazione immediata $v \Rightarrow w_0 \Rightarrow \dots \Rightarrow w$, ovvero:

$$\frac{}{v \Rightarrow^* v} \qquad \frac{v \Rightarrow^* w \quad w \Rightarrow z}{v \Rightarrow^* z}$$

\Rightarrow^* è la chiamata riflessiva e transitiva della relazione \Rightarrow

Da notare che può succedere che più derivazioni generino la stessa stringa. Questo rende la grammatica non univoca.

2.4 - LINGUAGGIO GENERATO

Il **linguaggio generato** da una grammatica $G(NT, T, R, S)$ è l'insieme:

$$L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$$

Come facciamo a determinare $L(G)$? E come facciamo a determinare se una stringa appartiene a $L(G)$?

La tecnica "naif" è quella di iniziare da S e provare tutte le stringhe fino a quando troviamo w . (non deterministico)

ESERCIZIO

Calcolare la grammatica di:

$$L = \{a^n b^m b^n a^m \mid n, m \geq 0\}$$

Dobbiamo comporre la stringa

Consideriamo, per comodità, che:

$$L = \{a^n b^{n+m} a^m \mid n, m \geq 0\}$$

Quando noi aggiungiamo una b al centro, dobbiamo o aggiungere una a prima o una a dopo.
Possiamo decidere di aggiungere prima tutte le a prima, oppure tutte le a dopo

$$S = \varepsilon \mid aA \mid Ba \mid aABa$$

$$A = b \mid aAb$$

$$B = b \mid bBa$$

3. Alberi di derivazione e sintattici

Friday, October 14, 2022 3:11 PM

3.1 - ALBERI DI DERIVAZIONE

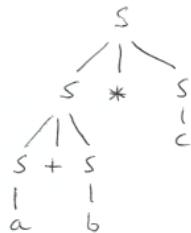
Consideriamo la grammatica

$$S \rightarrow a \mid b \mid c \mid S + S \mid S * S$$

Consideriamo la derivazione

$$S \Rightarrow \underline{S} * S \Rightarrow \underline{S} + S * S \Rightarrow a + \underline{S} * S \Rightarrow a + b * \underline{S} \Rightarrow a + b * c$$

- Derivazione **leftmost**
 - Ad ogni passaggio ricaviamo il non terminale più a sinistra
- È possibile associare un albero di **derivazione ad esso**:



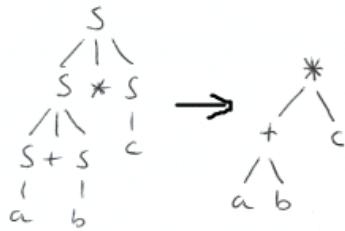
Consideriamo invece un'altra derivazione:

$$S \Rightarrow S * \underline{S} \Rightarrow \underline{S} * c \Rightarrow S + \underline{S} * c \Rightarrow \underline{S} + b * c \Rightarrow a + b * c$$

- Derivazione **rightmost**
- Genera lo stesso albero di derivazione

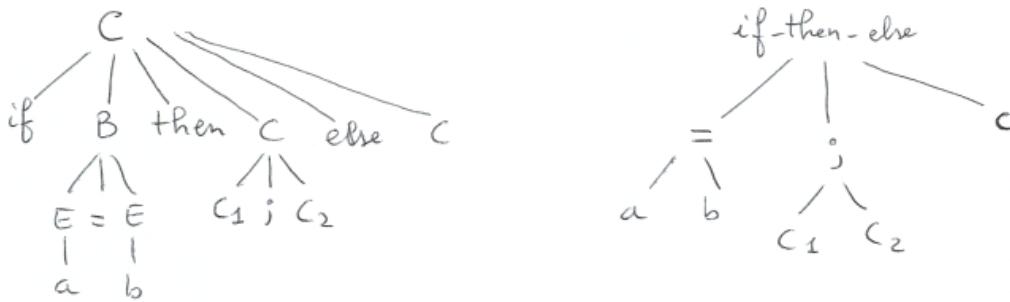
OSSERVAZIONI (IMPORTANTE):

- 1) Un albero di derivazione riassume tante derivazioni diverse che sono equivalenti tra loro (nel senso che generano lo stesso albero)
- 2) Esiste una corrispondenza biunivoca tra derivazioni canoniche sinistre (o destre) e alberi di derivazioni:
 - i. Dato un albero di derivazione, esiste una sola derivazione **leftmost** che lo genera. (vale anche per **rightmost**) (non è ovvio!)
 - ii. Data una derivazione **leftmost**, ad essa viene associata univocamente un albero di derivazione. (vale anche per **rightmost**)
- 3) L'albero di derivazione fornisce informazione semantiche, ovvero "quali operandi per quali operatori"
Si può quindi scrivere anche un albero sintattico, ricavato dall'albero di derivazione:



3.2 - ALBERO SINTATTICO

L'albero sintattico è un albero dove i nodi interni sono etichettati con gli operatori e le foglie sono gli operandi. Questa è una tipica rappresentazione interna dei compilatori per generare *codice intermedio*.



In figura: Il primo albero di derivazione, si può riscrivere come l'albero sintattico a destra.

3.3 - ALBERO DI DERIVAZIONE (DEFINIZIONE FORMALE)

Data una grammatica libera $G = (NT, T, R, S)$ un **albero di derivazione** (o di parsing) è un albero ordinato in cui:

- Ogni nodo è etichettato con un simbolo in $NT \cup \{\epsilon\} \cup T$
- La radice è etichettata con S
- Ogni nodo interno è etichettato con un simbolo in NT
- Se il nodo n
 - o Ha etichetta $A \in NT$
 - o I suoi figli sono nell'ordine m_1, \dots, m_k con etichette $x_1, \dots, x_k (\in NT \cup T)$
Allora $A \rightarrow x_1 \dots x_k$ è una produzione in R
- Se il nodo n ha etichetta ϵ , allora n è una foglia, è foglia unica e, detto A suo padre, $A \rightarrow \epsilon$ è una produzione di R
- Se inoltre ogni nodo foglia è etichettato su $T \cup \{\epsilon\}$, allora l'albero corrisponde ad una **derivazione completa**.

Teorema:

Una stringa $w \in T^*$ appartiene a $L(G) \Leftrightarrow$ ammette un albero di derivazione completo (le cui foglie, lette da sx a dx danno una stringa w)

Quindi visita in ordine anticipato tralasciando i non terminali

3.4 - GRAMMATICA AMBIGUA

Definizione:

Una grammatica libera G è ambigua se $\exists w \in L(G)$ che ammette più alberi di derivazione.

Definizione:

Un linguaggio L è ambiguo se tutte le grammatiche G, tali che $L(G)=L$, sono ambigue

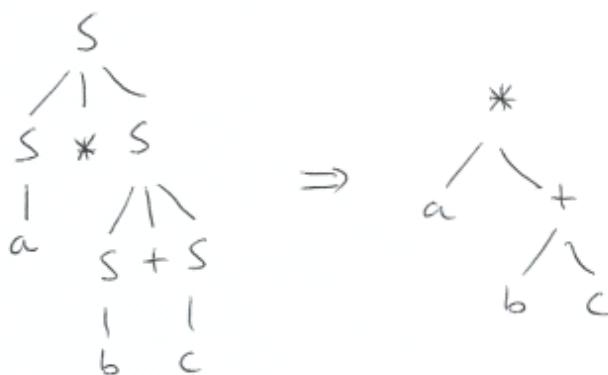
Alcune grammatiche possono essere modificate per non essere più ambigue e generare lo stesso linguaggio. Ma questo non può essere fatto per tutte.

Le grammatiche (patologiche) per le quali non è possibile rimuovere l'ambiguità. Il linguaggio di queste grammatiche è ambiguo.

Esempio di lingua ambigua a slide 8. Ovviamente, dimostrare che un linguaggio è ambiguo risulta difficile, soprattutto dal punto di vista formale.

Analizziamo una grammatica ambigua:

$$S \rightarrow a \mid b \mid c \mid S + S \mid S * S$$



Notiamo che $a * b + c$ è una stringa che può essere espressa con due alberi di derivazione diversi. Pertanto la semantica non è univoca: la stessa semantica, data dall'albero sopra e da quello sotto, sono "uguali" (anche se non dovrebbero, in quanto la sintassi è diversa).

Pertanto, la grammatica, così come definita sopra, è inutilizzabile per dare la semantica di $a * b + c$.

Consideriamo i problemi che si possono risolvere per renderla non ambigua:

- 1) Precedenza del $*$ sul $+$, in modo che $a * b + c$ sia interpretato come $\sim (a * b) + c$
- 2) Associatività del $+$ e del $*$, che possiamo scegliere sia a destra o a sinistra.

Per esempio:

$E \rightarrow E + T \mid T$ vuol dire che E è associativa a sinistra

$T \rightarrow A * T \mid A$ vuol dire che T è associativa a destra

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

Inoltre, in questo esempio e con questo ordine, abbiamo che il * ha precedenza sul + perché è più interno nell'albero generato.

C'è un problema però!

Usando solo questa grammatica, un albero semantico che ha la semantica del secondo albero nell'esempio prima non è possibile generarlo.

Pertanto, con questa grammatica non esprimiamo interamente ciò che la nostra semantica dovrebbe poter esprimere: concettualmente, dovremmo poter dare precedenza ad una moltiplicazione oppure ad una somma (modificare la semantica) in base a come costruiamo la nostra stringa (grammatica).

Il modo che la matematica usa sono delle parentesi:

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

$$T \rightarrow A * T \mid A$$

$$A \rightarrow a \mid b \mid c \mid \begin{array}{c} (E) \\ \text{le parentesi} \\ \text{sono zucchero} \\ \text{sintattico} \end{array}$$

Le parentesi sono dei termini ausiliari che generano un albero di sintassi coerente.

Ovvero, noi vogliamo generare $a * (b + c)$, per esempio, che ha un albero di sintassi corretto con la grammatica ambigua (che è astratta, pertanto) il quale avrà un diverso significato rispetto a quello generato con la non ambigua scritta sopra.

Noi stiamo completando, senza renderla non ambigua, la grammatica concreta che otteniamo a partire dalla grammatica astratta sopra. Se noi ignoriamo lo zucchero sintattico, la semantica è coerente a quella del linguaggio che vogliamo esprimere, è non ambigua ed è più completa.

DEFINIZIONE - Sintassi concreta

Grammatica non ambigua che fa uso di zucchero sintattico (come nell'ultimo esempio)

DEFINIZIONE - Sintassi astratta

Grammatica semplice ed intuitiva, ma ambigua.

Dall'albero di derivazione da sintassi concreta, estraiamo un albero sintattico di sintassi astratta, detto albero sintattico astratto.

Seguendo il nostro esempio di prima, quindi: creiamo un albero di derivazione usando la grammatica non ambigua, usando questa creiamo l'albero sintattico e a questo rimuoviamo lo zucchero sintattico: adesso abbiamo la sintassi astratta della stringa.

Vedere gli ultimi esempi di Lez-03 per avere più dettagli di quest'ultimo passaggio.

4. Vincoli sintattici

Saturday, October 15, 2022 7:59 PM

4.1 - VINCOLI SINTATTICI CONTESTUALI (Semantica statica)

Esempi dove ci servono vincoli non inclusi nelle grammatiche libere da contesto:

- Una variabile in uso deve prima essere dichiarata
- Compatibilità di tipo di un argomento
 - o Ovvero $x := e \Rightarrow "x"$ e l'espressione " e " devono essere dello stesso tipo
- Il numero e il tipo di parametri attuali di una chiamata di procedura deve essere uguale al numero e al tipo di parametri formali della dichiarazione

Sono vincoli sintattici, ma le grammatiche libere non riescono a usare il contesto (con una BNF, per esempio, non possiamo controllare che una variabile deve prima essere dichiarata e poi usata).

Ci sono due soluzioni:

1. Grammatiche dipendenti dal contesto
 - a. La complessità diventa esponenziale, pertanto non è molto pratico
2. Usare controlli "ad hoc"
 - a. Nella costruzione di un compilatore, abbiamo la parte chiamata "analisi semantica" che si occupa di questi controlli.

4.2 - NEI LINGUAGGI DI PROGRAMMAZIONE

Vediamo la differenza tra sintassi e semantica in programmazione

I vincoli sintattici contestuali "appartengono" alla sintassi.

Tuttavia, tradizionalmente, nel gergo dei linguaggi di programmazione, si intende:

- **Sintassi**
 - o Quello che si descrive in BNF
- **Semantica**
 - o Tutto il resto
 - o Anche i vincoli contestuali, i quali quindi sono "**vincoli semantici**", detti di **semantica statica**, cioè vincoli che possono essere verificati ispezionando il codice del programma, **senza mandarlo in esecuzione**.

4.3 - SEMANTICA DINAMICA vs SEMANTICA STATICÀ

- Per semantica statica si intendono l'insieme dei controlli che vengono fatto sul testo del programma senza eseguirlo. Questa si chiama anche sintassi contestuale.
- Per semantica dinamica si intende una rappresentazione formale dell'esecuzione del programma, la quale può mostrare errori dinamici, cioè che avvengono durante l'esecuzione.
 - o Un esempio è fare la divisione tra due numeri con un denominatore scritto dall'utente, e avere che l'input è 0.
Quindi fare $n/0$ ci dà un errore durante l'esecuzione.
 - o In questo caso, non possiamo statisticamente sapere se l'errore si verificherà perché l'occorrere dell'errore dipende dall'input che l'utente fornirà quando il programma viene eseguito.

4.4 - SEMANTICA DINAMICA

Serve a fornire un modello matematico che descriva, indipendentemente dall'architettura su cui il programma viene eseguito, il comportamento del programma. La sua implementazione deve rispettare questa specifica del linguaggio.

Esempio:

$$P = \quad x := x + 1, \quad \text{Store } \sigma = \text{insieme di associazioni tra nomi e valori}$$

$$\underbrace{\langle x := x + 1, \sigma \rangle}_{\begin{array}{l} \text{valuto il comando} \\ \text{utilizzando uno store } \sigma \end{array}} \longrightarrow \underbrace{\sigma[x \mapsto \sigma(x) + 1]}_{\begin{array}{l} \text{il risultato è uno store} \\ \text{"aggiornato" in cui ad } x \text{ è} \\ \text{associato il valore } \sigma(x) + 1 \end{array}}$$

A chi serve la semantica dinamica?

- Al programmatore
 - Analisi del programma
 - Deve sapere esattamente cosa debba fare il suo programma
 - Deve poter dimostrare proprietà del suo programma ("termina sempre per ogni input",...)
- Al progettista del linguaggio
 - Strumento di specifica del linguaggio
 - Deve poter dimostrare proprietà del linguaggio ("è Turing completo?",...)
- All'implementatore del linguaggio
 - Riferimento per dimostrare la correttezza dell'implementazione

Da ricordare anche che un compilatore è corretto quando preserva la semantica dinamica (e non quella statica), quindi per dimostrare che un compilatore è corretto serve avere una semantica per il linguaggio sorgente e per il linguaggio oggetto.

4.5 - DEFINIZIONE SEMANTICA

Vi sono due tecniche, che si associano ad ogni programma sequenziale:

- **Operazionale:** (Macchina astratta a stati e transizioni)
 - Si costruisce una specie di automa che, passo a passo, mostra l'effetto della esecuzione delle varie istruzioni
 - Enfasi su **come** si calcola
 - Vedremo l'uso di questa semantica per un semplice linguaggio imperativo
- **Denotazionale:**
 - Si associa ad ogni programma sequenziale una funzione da input e un output (inclusa strutture ausiliarie dette ambiente e memoria)
 - Enfasi su **cosa** si calcola (vengono nascosti i passi intermedi del calcolo)

4.6 - PRAGMATICA

Insieme di "regole" e consigli su come usare un linguaggio:

- Usare "while" per cicli indeterminati, "for" per cicli predeterminati.
- Non usare il "go-to" per saltare da una parte all'altra
- ...

4.7 - IMPLEMENTAZIONE

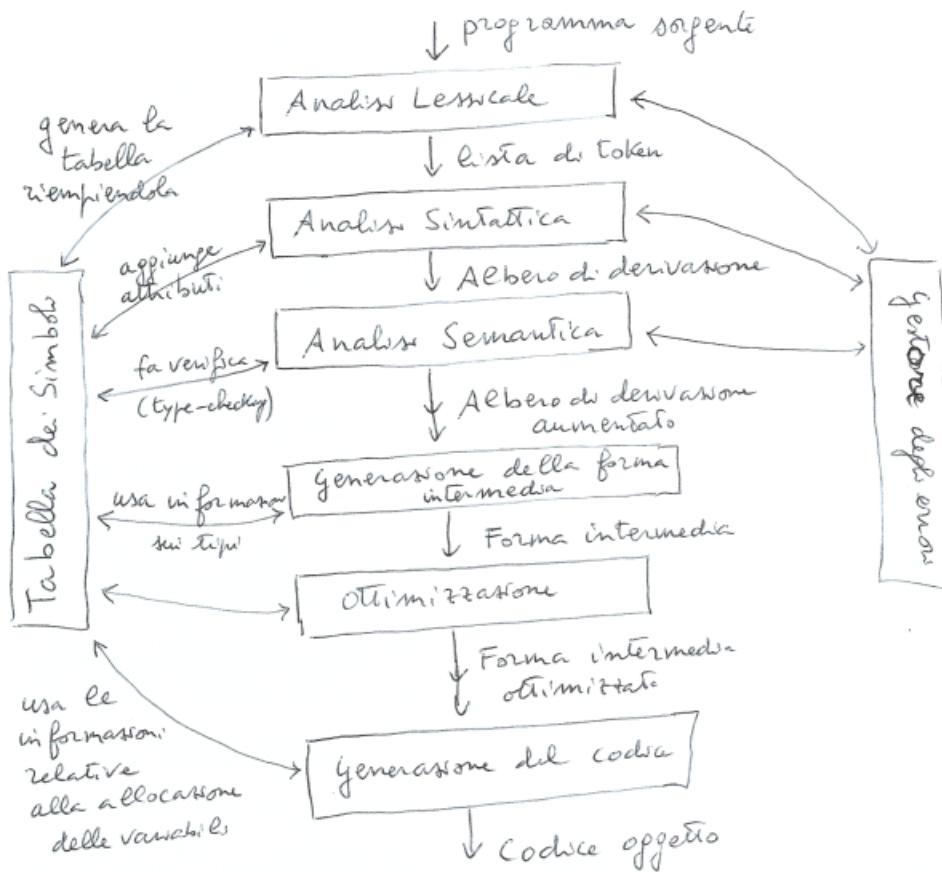
Ovvero la scrittura di un compilatore (o un interprete) per una macchina ospite già realizzata, costruendo così una macchina astratta per il linguaggio.

- Correttezza dell'implementazione?
 - Bisogna dimostrare che preserva la semantica del linguaggio, ovvero che il programma sorgente e quello oggetto calcolano la stessa funzione (per i linguaggi sequenziali).
- Costo dell'implementazione
 - È il compilatore in grado di produrre codice efficiente?
 - Certi costrutti linguistici, scelti dal progettista del linguaggio, sono troppo onerosi in pratica?

5. Struttura di un compilatore

lunedì 17 ottobre 2022 17:28

5.1 - COME E' FORMATO



Ogni errore rilevato (nelle prime 3 fasi) non blocca il compilatore, ma genera un opportuno messaggio d'errore

5.2 - ANALISI LESSICALE (SCANNER)

Spezza il programma nei **componenti sintattici primitivi**, chiamati **tokens** (identificatori, numeri, operatori, parentesi, parole riservate, ...).

- Controlla solo che il lessico sia ammissibile
 - o Operatori inesistenti, identificatori non ammessi, simboli non previsti, etc...
- Riempie parzialmente la tabella dei simboli
 - o Con gli identificatori di variabili, procedure, funzioni, etc...

Per realizzare uno scanner dobbiamo studiare:

- **Grammatiche regolari**
 - o Ovvvero grammatiche che hanno un'unico non-terminale per produzione e il cui non

terminale si trova sempre sulla destra (o sulla sinistra) di tutte le produzioni

- **Espressioni regolari**
 - Un formalismo usato per descrivere i linguaggi generati da grammatiche regolari
- **Automi a stati finiti** (NFA - DFA)
 - Uno strumento che permette di riconoscere i linguaggi "regolari"

5.3 - ANALISI SINTATTICA (PARSER)

Usando la lista di token fatta dallo scanner, il **parser produce l'albero di derivazione** del programma, riconoscendo se le frasi sono sintatticamente corrette.

- Controlla che (ad esempio)
 - Le parentesi usate in un'espressione aritmetica sono bilanciate
 - Che i comandi siano composti secondo le regole grammatiche
 - *if x = 5 then then x := 3*

Per realizzare un parser dobbiamo studiare:

- **Grammatiche libere da contesto**
- **Automi a pila** (pushdown automata) soprattutto nella versione **determinata** (DPDA)

5.4 - ANALISI SEMANTICA

Esegue dei controlli di semantica statica (ovvero sintassi contestuale) per rilevare eventuali errori "semantici".

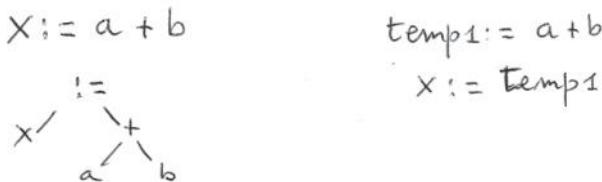
- Arricchire l'albero di derivazione generato dal Parser con informazioni di **tipo**
- Verificare i tipi negli assegnamenti, parametri attuali e parametri formali, dichiarazioni e uso di variabili, ...
- Genera opportuni messaggi di errore

Ultima fase che fa controlli di errore che non interrompono

5.5 - GENERAZIONE DELLA FORMA INTERMEDIA

Genera codice scritto in un linguaggio intermedio, facilmente traducibile nel linguaggio macchina di varie macchine diverse.

- Easy to produce - easy to translate
 - Utilizza operazioni molto semplici, tipicamente "three-address code"
 - *Tipo: x := y + z*
 - Nel generare codice intermedio, si segue la struttura dell'albero sintattico, ricavato dall'albero di derivazione



5.6 - OTTIMIZZAZIONE

Si effettuano ottimizzazioni sul codice intermedio in modo da renderlo più efficiente:

- Rimozione del codice inutile (dead code)
- Espansione in linea di chiamate di funzione
- Fattorizzazione di sotto-espressioni
- Mettere fuori dai cicli sotto-espressioni che non variano

5.7 - GENERAZIONE DEL CODICE

Viene generato codice per una specifica architettura (include anche l'assegnazione dei registri e ottimizzazioni specifiche per quell'architettura).

- Quali variabili conviene tenere nei registri del processore?

5.8 - TABELLA DEI SIMBOLI

Memorizza le informazioni sui nomi presenti nel programma (identificatori delle variabili, funzioni, procedure)

- Ad esempio, per le matrici, mette come attributo la dimensione e il tipo dell'elemento

6. Semantica operazionale strutturale

mercoledì 19 ottobre 2022 00:12

6.1 - LINGUAGGIO

Premessa: perché definiamo un linguaggio?

- Per poter parlare della semantica di un linguaggio, dobbiamo prima definire il linguaggio stesso. In questo caso, definiamo un linguaggio molto semplice, che comprendere booleani, espressioni matematiche, numeri naturali e alcuni comandi basilari.
- Lo definiamo attraverso una sintassi astratta (in modo che sia semplice), anche se questo lo rende ambigua. In realtà, il fatto che sia ambigua a noi non fa molta differenza: a livello semantico, usiamo solo gli alberi sintattici, quindi togliamo l'ambiguità dell'interpretazione di una stringa. Questa ambiguità dovrebbe essere risolta dalla grammatica concreta, che a noi non concerne definire.

INSIEMI DI BASE

- Booleani:
 - o $\{tt, ff\} = \Pi$
 - o Metavariabili
 - $t, t_1, t' \in \Pi$
- Numeri naturali:
 - o $\{0, 1, \dots\} = \mathbb{N}$
 - o Metavariabili
 - $n, m, p \in \mathbb{N}$
- Variabili
 - o $\{a, b, \dots, z\} = Var$
 - o Metavariabili
 - $v \in Var$

NOTA: Cosa sono le metavariabili?

Le metavariabili vengono usate per riferirsi ad un qualsiasi elemento dell'insieme, senza riferirsi in particolare ad uno. Sono dei **"placeholder"** per un effettivo numero, valore di verità, o nome di variabile che appartiene al suo rispettivo insieme.

Se noi scriviamo $v + m$ negli assiomi, regole di inferenza, o nella grammatica, intendiamo dire che c'è la somma tra una qualsiasi variabile (a, b, \dots) e un numero naturale, quindi in un mondo $v + m = a + 5$ oppure in un altro $v + m = b + 2$, etc. Ci servono per capire che siamo arrivati ad un elemento "indivisibile" della grammatica che vedremo sotto.

INSIEMI DERIVATI (BNF)

- **Espressioni aritmetiche:**
 - o $e \in Exp$
 - o $e ::= m \mid v \mid e + e \mid e - e \mid e * e$
- **Espressioni booleane:**
 - o $b \in Bexp$
 - o $b ::= t \mid e = e \mid b \text{ or } b \mid \sim b$
- **Comandi:**
 - o $c \in Com$
 - o $c ::= skip \mid v := e \mid c; c \mid \text{while } b \text{ do } c \mid \text{if } b \text{ then } c \text{ else } c$

NOTA: Grammatica ambigua!

Come specificato nella premessa, questa grammatica è ambigua, ma questo non è un problema: quando noi andiamo ad analizzare la semantica, analizziamo gli alberi sintattici, quindi non dobbiamo interpretare la stringa "noi stessi" nel livello della semantica, ma ci penserà il parser usando una grammatica concreta (e non ambigua).

6.2 - DEFINIRE LA SEMANTICA

Definiamo, per ogni categoria sintattica (*Exp, Bexp, Com*) un modello detto "**sistema di transizione**".

DEFINIZIONE

Un sistema di transizione è una tripla (tupla ordinata di tre elementi) formata da:

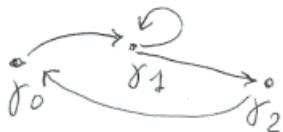
$\langle \Gamma, T, \rightarrow \rangle$ dove:

- Γ
 - Insieme di stati** (o configurazioni)
 - Può essere **infinito**
 - Ogni stato si indica con $\gamma_n (\gamma_0, \gamma_1, \dots)$
- T
 - $T \subseteq \Gamma$
 - Insieme di stati terminali**
- \rightarrow
 - $\rightarrow \subseteq \Gamma \times \Gamma$
 - Relazioni di transizione** (da uno stato all'altro), per esempio possiamo avere $(\gamma_0, \gamma_1) \in \rightarrow$, che per abuso di notazione si può anche scrivere come $\gamma_0 \rightarrow \gamma_1$.

- La computazione a partire dallo stato γ_0 è una sequenza $\gamma_0 \rightarrow \gamma_1 \rightarrow \dots$ che può essere finita o infinita.
- Con \rightarrow^* indichiamo la chiusura riflessiva e transitiva di \rightarrow , ovvero:

$$\circ \quad \frac{}{\gamma \rightarrow^* \gamma} \qquad \frac{\gamma \rightarrow^* \gamma' \quad \gamma' \rightarrow \gamma''}{\gamma \rightarrow^* \gamma''}$$

- Le relazioni \rightarrow si possono rappresentare come grafici



PROBLEMI DI QUESTA TRIPLOA

- 1) Γ è spesso un insieme infinito contabile \Rightarrow **deve avere una rappresentazione finita** implicita **attraverso una grammatica** / BNF. Quindi, essenzialmente, Γ coincide con uno dei linguaggi delle 3 categorie che abbiamo creato in precedenza:
 - a. $\Gamma_e = \{ \langle e, \sigma \rangle \mid e \in \underbrace{\text{Exp}}_{\text{definito in BNF}}, \sigma \in \text{Store} \}$
- 2) $\rightarrow \subseteq \Gamma \times \Gamma$ è una relazione costituita, tipicamente, da infinite coppie $\gamma \rightarrow \gamma' \Rightarrow$ **abbiamo bisogno di una rappresentazione finita** implicita come minima relazione che soddisfa un

certo insieme finito di assiomi e regole di inferenza

- 3) Per dare significato alle variabili introduciamo lo store σ .

$$\sigma := \text{Var} \mapsto \mathbb{N}$$

Associa ad ogni variabile un valore. È una funzione, un esempio (di come si può scrivere) è:

$$\sigma = \left\{ \frac{x_1}{n_1}, \frac{x_2}{n_2}, \dots, \frac{x_k}{n_k} \right\}$$

$$\text{Var} = \{x_1, \dots, x_k\}$$

$$\text{Store} = \{\sigma \mid \sigma: \text{Var} \mapsto \mathbb{N}\}$$

DEFINIZIONE DI IS - ID - ES - ED - IP - EP

Quando si scrive "I" si intende "Interno":

Si valutano tutti gli argomenti prima di "passare" allo stato successivo.

Per esempio, per la semantica delle espressioni sotto, esse sono tutte I.

Quando si scrive "E" si intende "Esterno":

Si valuta solo parzialmente le espressioni prima di "passare" allo stato successivo.

Per esempio, per valutare un OR non ci è necessario valutare la parte destra per capire il significato dell'espressione: possiamo passare al passaggio successivo dove considerando la parte non valutata dell'espressione.

S e D indicano da dove partiamo:

S indica sinistra, D indica destra, ed indica da dove si iniziano a valutare gli argomenti.

P significa "Parallelismo":

Vuol dire che le operazioni sono eseguite in parallelo: per esempio, nella somma, piuttosto che fare la valutazione sinistra e poi destra (in quanto interna le fa entrambe) ma facciamo in un passaggio sia la destra che la sinistra e poi l'ultimo passaggio dove facciamo la somma di $m + n$.

IP risulta non deterministica ma confluente, cioè termina in modo univoco.

NOTA

$$IS = ID$$

$$ES \neq ID \neq ED \neq ES \neq IS \neq ED$$

6.3 - SEMANTICA DELLE ESPRESSIONI ARITMETICHE

$\langle \Gamma_e, T_e, \rightarrow_e \rangle$ dove

- $\Gamma_e = \{ \langle e, \sigma \rangle \mid e \in \text{Exp}, \sigma \in \text{Store} \}$
- $T_e = \{ \langle n, \sigma \rangle \mid n \in \mathbb{N}, \sigma \in \text{Store} \}$
- la relazione \rightarrow_e è definita come la minima relazione che soddisfa gli assiomi e le regole di inferenza qui sotto:

$$\begin{array}{l}
 (\text{Var}) \quad \frac{}{\langle v, \sigma \rangle \rightarrow_e \langle \sigma(v), \sigma \rangle} \xrightarrow{\text{stato}} \text{terminale} \\
 (\text{Sum}_1) \quad \frac{\langle e_0, \sigma \rangle \rightarrow_e \langle e'_0, \sigma' \rangle}{\langle e_0 + e_1, \sigma \rangle \rightarrow_e \langle e'_0 + e_1, \sigma' \rangle} \\
 (\text{Sum}_2) \quad \frac{\langle e_1, \sigma \rangle \rightarrow_e \langle e'_1, \sigma' \rangle}{\langle m + e_1, \sigma \rangle \rightarrow_e \langle m + e'_1, \sigma' \rangle} \\
 (\text{Sum}_3) \quad \frac{-}{\langle m + m', \sigma \rangle \rightarrow_e \langle p, \sigma \rangle} \quad \text{se } p = m + m' \\
 (\text{Sub}_1) \quad \frac{\langle e_0, \sigma \rangle \rightarrow_e \langle e'_0, \sigma' \rangle}{\langle e_0 - e_1, \sigma \rangle \rightarrow_e \langle e'_0 - e_1, \sigma' \rangle} \\
 (\text{Sub}_2) \quad \frac{\langle e_1, \sigma \rangle \rightarrow_e \langle e'_1, \sigma' \rangle}{\langle m - e_1, \sigma \rangle \rightarrow_e \langle m - e'_1, \sigma' \rangle} \\
 (\text{Sub}_3) \quad \frac{-}{\langle m - m', \sigma \rangle \rightarrow_e \langle p, \sigma \rangle} \quad \begin{array}{l} \text{se } m \geq m' \text{ e} \\ p = m - m' \end{array}
 \end{array}$$

OSSERVAZIONI:

- 1) Gli assiomi e le regole di inferenza sono scrivibili attraverso Prolog (anche se inefficace).
- 2) Lo store σ non viene mai modificato da questi assiomi e regole di inferenza.
 - a. Questo è anche importante, in quanto implica che se $\sigma = \sigma'$ nelle premesse, allora lo è ugualmente nelle conclusioni.
- 3) L'assioma (Sub_3) Permette di derivare la transizione $\langle m - m', \sigma \rangle \rightarrow_e \langle p, \sigma \rangle$ solo se $m \geq m'$ (in quanto abbiamo solo numeri naturali, e $p \notin \mathbb{N}$ se $m < m'$). Altrimenti, lo stato $\langle m - m', \sigma \rangle$ è bloccato, pur non essendo un terminale. (È un caso di "errore", come vedremo in seguito)
- 4) Le regole (Sum_1) e (Sum_2) riflettono una disciplina / regola di valutazione detta IS (spiegata prima)

Potete vedere esempi "pratici" sulle slide.

6.4 - DIMOSTRAZIONE CHE \rightarrow_e È DETERMINISTICA

Obiettivo da dimostrare:

$$\forall \gamma, \gamma', \gamma'' \quad \gamma \rightarrow_e \gamma' \wedge \gamma \rightarrow_e \gamma'' \Rightarrow \gamma' = \gamma''$$

Usiamo l'**induzione strutturale**.

Supponiamo di voler dimostrare una certa $P(e)$ che vale per ogni $e \in \text{Exp}$

Se dimostriamo che:

$$1) \quad \forall m \in \mathbb{N}, \quad P(m) \text{ è vera}$$

2) $\forall v \in Var, P(v)$ è vera

$$3) \frac{P(e_0), P(e_1) \text{ vere}}{P(e_0 + e_1) \text{ vera}}$$

$$4) \frac{P(e_0), P(e_1) \text{ vere}}{P(e_0 - e_1) \text{ vera}}$$

$$5) \frac{P(e_0), P(e_1) \text{ vere}}{P(e_0 * e_1) \text{ vera}}$$

Allora concludiamo che $\forall e \in Exp \ P(e)$ è vera.

Dimostriamo ora che $P(e)$ è vera $\forall e \in Exp$ usando induzione strutturale.

Nel nostro caso

$$P(e) = \forall \sigma, \gamma', \gamma'' \quad (\langle e, \sigma \rangle \rightarrow_e \gamma' \wedge \langle e, \sigma \rangle \rightarrow_e \gamma'') \Rightarrow \gamma' = \gamma''$$

Dimostriamo che $P(e)$ è vera $\forall e \in Exp$ usando l'induzione strutturale:

1) $e = m \in \mathbb{N}$

- $\langle m, \sigma \rangle \not\rightarrow$ e allora $P(m)$ è vera perchè la premessa dell'implicazione è falsa.

2) $e = v \in Var$

- Per la regola (Var), l'unica transizione derivabile per $\langle v, \sigma \rangle$ è $\langle v, \sigma \rangle \rightarrow_e \langle \sigma(v), \sigma \rangle$
- Poiché σ è una funzione (quindi $\sigma(v)$ è univoca) e c'è solo la regola (Var) che è applicabile, la tesi segue: $\langle v, \sigma \rangle \rightarrow_e \gamma' \wedge \langle v, \sigma \rangle \rightarrow_e \gamma'' \Rightarrow \gamma' = \gamma''$

3) $e = e_0 + e_1$

- Supponiamo che

- $\langle e_0 + e_1, \sigma \rangle \rightarrow_e \gamma'$
- $\langle e_0 + e_1, \sigma \rangle \rightarrow_e \gamma''$

- Dobbiamo dimostrare che $\gamma' = \gamma''$

- Ci sono tre sotto-casi da analizzare, ovvero Sum1, Sum2 e Sum3.

- $\langle e_0, \sigma \rangle \rightarrow \langle e'_0, \sigma' \rangle \quad e \quad \gamma' = \langle e'_0 + e_1, \sigma \rangle$
- In questo caso, $e_0 \notin \mathbb{N}$ e l'unica regola che ho applicato è (Sum1). Allora se $\langle e_0 + e_1, \sigma \rangle \rightarrow_e \gamma''$ è necessario che $\langle e_0, \sigma \rangle \rightarrow \langle e''_0, \sigma'' \rangle \quad e \quad \gamma' = \langle e''_0 + e_1, \sigma'' \rangle$
- Ma, per ipotesi induttiva, $P(e_0)$ vale, per cui deve essere che $e'_0 = e''_0$ e $\sigma' = \sigma''$, da cui discende $\gamma' = \gamma''$
- QED.

- ... Alte dimostrazioni da vedere sulla slide 11 lezione 5.

CONCLUSIONE

In quanto abbiamo determinato che \rightarrow_e è deterministica, a partire da $\langle e, \sigma \rangle$, arriveremmo su una sola configurazione terminale $\langle n, \sigma \rangle$: "n è il valore di e in σ ".

È possibile quindi definire una funzione "eval":

$eval: Exp \times Store \dashrightarrow \mathbb{N}$

(Funzione parziale, perché non sempre raggiunge stati terminali, tipo nelle sottrazioni)

$$eval(e, \sigma) = \begin{cases} mse \langle e, \sigma \rangle \rightarrow^* \langle m, \sigma \rangle \\ \text{indefinita altrimenti} \end{cases}$$

ESEMPIO

$$\begin{aligned} \text{eval}\left(\langle(x+2)-y, \{x/5, y/3\}\rangle\right) &= 4 \\ \text{eval}\left(\langle(x+2)-y, \{x/2, y/5\}\rangle\right) &= \text{indefinito} \end{aligned}$$

EQUIVALENZA TRA ESPRESSIONI

$$e \equiv e' \Leftrightarrow \forall \sigma \in Store, \quad \text{eval}(e, \sigma) = \text{eval}(e', \sigma)$$

$$\text{ESEMPIO: } (v_1 + v_2) + v_3 \equiv v_1 + (v_2 + v_3)$$

6.5 - SEMANTICA ESPRESSIONI BOOLEANE

$$b ::= t \mid e = e \mid b \text{ or } b \mid \neg b$$

$$\langle I_b, T_b, \rightarrow_b \rangle \quad \text{dove } I_b = \{\langle b, \sigma \rangle \mid b \in B_{\text{exp}}, \sigma \in Store\} \\ T_b = \{\langle tt, \sigma \rangle, \langle ff, \sigma \rangle \mid \sigma \in Store\}$$

$e \rightarrow_b$ è la minima relazione generata dai seguenti assiomi e regole d'inferenza

$$(Eq_1) \frac{\langle e_0, \sigma \rangle \rightarrow_e \langle e'_0, \sigma' \rangle}{\langle e_0 = e_1, \sigma \rangle \rightarrow_b \langle e'_0 = e_1, \sigma' \rangle} \quad \left. \begin{array}{l} \\ \\ \end{array} \right\} IS$$

$$(Eq_2) \frac{\langle e_1, \sigma \rangle \rightarrow_e \langle e'_1, \sigma' \rangle}{\langle m = e_1, \sigma \rangle \rightarrow_b \langle m = e'_1, \sigma' \rangle} \quad \left. \begin{array}{l} \\ \\ \end{array} \right\} ES$$

$$(Eq_3) \frac{\langle m = n, \sigma \rangle \rightarrow_b \langle t, \sigma \rangle}{\langle m = n, \sigma \rangle \rightarrow_b \langle t, \sigma \rangle} \quad \text{dove } t = \begin{cases} tt & se m = n \\ ff & se m \neq n \end{cases}$$

$$(Or_1) \frac{\langle b_0, \sigma \rangle \rightarrow_b \langle b'_0, \sigma' \rangle}{\langle b_0 \text{ or } b_1, \sigma \rangle \rightarrow_b \langle b'_0 \text{ or } b_1, \sigma' \rangle} \quad \left. \begin{array}{l} \\ \\ \end{array} \right\} ES \leftarrow \begin{array}{l} \text{comincia} \\ \text{con elag.} \\ \text{asx} \end{array}$$

$$(Or_2) \frac{}{\langle tt \text{ or } b_1, \sigma \rangle \rightarrow_b \langle tt, \sigma \rangle} \quad \left. \begin{array}{l} \\ \\ \end{array} \right\} \begin{array}{l} \text{non valuto} \\ \text{sempre tutte} \\ \text{due gli argomenti!} \end{array}$$

$$(Or_3) \frac{}{\langle ff \text{ or } b_1, \sigma \rangle \rightarrow_b \langle b_1, \sigma \rangle}$$

$$(Neg_1) \frac{\langle b, \sigma \rangle \rightarrow_b \langle b'_1, \sigma' \rangle}{\langle \neg b, \sigma \rangle \rightarrow_b \langle \neg b'_1, \sigma' \rangle}$$

$$(Neg_2) \frac{}{\langle \neg t, \sigma \rangle \rightarrow_b \langle t', \sigma \rangle} \quad \text{dove } t' = \begin{cases} tt & se t = ff \\ ff & se t = tt \end{cases}$$

OSSERVAZIONI

- 1) Lo store non viene mai modificato
- 2) \rightarrow_b è deterministica
- 3) Si può definire $\text{eval}_b(b, \sigma) = \begin{cases} t & se \langle b, \sigma \rangle \rightarrow^* \langle t, \sigma \rangle \\ \text{indefinita} & \text{altrimenti} \end{cases}$
- 4) $b \equiv b' \Leftrightarrow \forall \sigma \in Store, \quad \text{eval}_b(b, \sigma) = \text{eval}_b(b', \sigma)$

6.6 - SEMANTICA DEI COMANDI

$C ::= \text{skip} \mid V := e \mid c;c \mid \text{if } b \text{ then } c \text{ else } c \mid \text{while } b \text{ do } c$

$$\langle \Gamma_c, T_c, \rightarrow_c \rangle \text{ dove } \begin{aligned} \Gamma_c &= \{ \langle c, \sigma \rangle \mid c \in \text{Com}, \sigma \in \text{store} \} \\ &\cup \{ \sigma \mid \sigma \in \text{store} \} \end{aligned}$$

$$T_c = \{ \sigma \mid \sigma \in \text{store} \} = \text{store}$$

$\rightarrow_c \subseteq \Gamma_c \times \Gamma_c$ è la più piccola relazione generata dai seguenti assiomi e regole di inferenza

$$(\text{skip}) \quad \frac{}{\langle \text{skip}, \sigma \rangle \rightarrow_c \sigma}$$

$$(\text{Ass}) \quad \frac{\langle e, \sigma \rangle \xrightarrow{*} \langle m, \sigma \rangle}{\langle V := e, \sigma \rangle \rightarrow_c \sigma[m/V]}$$

dove $\sigma[m/V] \models V' = \begin{cases} m & \text{se } V = V \\ \sigma(V') & \text{altrimenti} \end{cases}$

"aggiornamento dello store"

Ad es: se $\sigma = \{X_1, Y_1\}$, allora $\sigma[3/X] = \{X_3, Y_1\}$

$$(\text{Seq}_1) \quad \frac{\langle c_0, \sigma \rangle \rightarrow_c \langle c'_0, \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \rightarrow_c \langle c'_0; c_1, \sigma' \rangle}$$

$$(\text{Seq}_2) \quad \frac{\langle c_0, \sigma \rangle \rightarrow_c \sigma'}{\langle c_0; c_1, \sigma \rangle \rightarrow_c \langle c_1, \sigma' \rangle}$$

$$(\text{If}_1) \quad \frac{\langle b, \sigma \rangle \xrightarrow{*} \langle \text{tt}, \sigma \rangle}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow_c \langle c_0, \sigma \rangle} \quad (19)$$

$$(\text{If}_2) \quad \frac{\langle b, \sigma \rangle \xrightarrow{*} \langle \text{ff}, \sigma \rangle}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow_c \langle c_1, \sigma \rangle}$$

$$(\text{Wh}_1) \quad \frac{\langle b, \sigma \rangle \xrightarrow{*} \langle \text{tt}, \sigma \rangle}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow_c \langle c; \text{while } b \text{ do } c, \sigma \rangle}$$

$$(\text{Wh}_2) \quad \frac{\langle b, \sigma \rangle \xrightarrow{*} \langle \text{ff}, \sigma \rangle}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow_c \sigma}$$

$$(\text{Alt-Wh}) \quad \frac{-}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow_c \langle \text{if } b \text{ then } c; \text{while } b \text{ do } c \text{ else skip, } \sigma \rangle}$$

OSSERVAZIONI

1) \rightarrow_c è deterministica

2) $\text{exec}: \text{Com} \times \text{Store} \dashrightarrow \text{Store}$

$$\text{exec}(c, \sigma) = \begin{cases} \sigma' \text{ se } \langle c, \sigma \rangle \xrightarrow{*} \sigma' \\ \text{indefinita altrimenti} \end{cases}$$

- Notiamo che divergenza e deadlock sono equiparati:

- Se noi entriamo in un loop infinito, il programma potrebbe essere "corretto", deadlock invece implica un errore, solitamente. Quindi, descriveremo tra poco come completare la semantica per poter riferire i casi di errore.

6.7 - SEMANTICA DEGLI ERRORI DINAMICI (A RUNTIME)

$$\Gamma' = \Gamma \cup \{ \text{err} \} \quad T' = T \cup \{ \text{err} \}$$

$$\text{eval} : \text{Exp} \times \text{Store} \xrightarrow[\text{(totale)}]{} \mathbb{N} \cup \{ \text{err} \}$$

$$\text{eval}_b : \text{Bexp} \times \text{Store} \longrightarrow \mathbb{N} \cup \{ \text{err} \}$$

$$\text{exec} : \text{Com} \times \text{Store} \longrightarrow \text{Store} \cup \{ \text{err} \}$$

(rimane parziale
per divergenza)

Regole per generare "errore"

$$(\text{Sub}_4) \quad \frac{\langle m - m', \sigma \rangle \rightarrow \text{err}}{m' > m} \quad \begin{array}{l} \text{unico} \\ \text{errore} \\ \text{possibile} \\ \text{nel motivo} \\ \text{semplice bing.} \end{array}$$

Regole per propagare l'errore (in Exp)

$$\begin{aligned} (\text{Sum}_4) \quad & \frac{\langle e_0, \sigma \rangle \xrightarrow{e} \text{err}}{\langle e_0 + e_1, \sigma \rangle \xrightarrow{e} \text{err}} \\ (\text{Sum}_5) \quad & \frac{\langle e_1, \sigma \rangle \xrightarrow{e} \text{err}}{\langle m + e_1, \sigma \rangle \xrightarrow{e} \text{err}} \end{aligned} \quad \left. \begin{array}{l} \text{analoga} \\ \text{regola anche} \\ \text{per } - \text{ e } * \end{array} \right\}$$

Regole per propagare l'errore (in Bexp)

• analoga a per Eq (Eq_4 e Eq_5)

$$(\text{Or}_4) \quad \frac{\langle b_0, \sigma \rangle \xrightarrow{b} \text{err}}{\langle b_0 \text{ or } b_1, \sigma \rangle \xrightarrow{b} \text{err}}$$

$$(\text{Neg}_3) \quad \frac{\langle b, \sigma \rangle \xrightarrow{b} \text{err}}{\langle \neg b, \sigma \rangle \xrightarrow{b} \text{err}}$$

Propagazione di errore (in Com)

$$(Ass_2) \quad \frac{\langle e, G \rangle \xrightarrow{*} \text{err}}{\langle v := e, G \rangle \xrightarrow{c} \text{err}}$$

$$(Seq_3) \quad \frac{\langle c_0, G \rangle \xrightarrow{c} \text{err}}{\langle c_0; c_1, G \rangle \xrightarrow{c} \text{err}}$$

$$(If_3) \quad \frac{\langle b, G \rangle \xrightarrow{*} \text{err}}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, G \rangle \xrightarrow{c} \text{err}}$$

$$(Wh_3) \quad \frac{\langle b, G \rangle \xrightarrow{*} \text{err}}{\langle \text{while } b \text{ do } c, G \rangle \xrightarrow{c} \text{err}}$$

OSSERVAZIONE:

- 1) Tutte queste regole di errore sono necessarie per un unico comando (Sub4).
In un normale linguaggio, ci sono tantissimi errori diversi da gestire, quindi le regole diventano molte di più.

6.8 - NONDETERMINISMO E PARALLELISMO

$c ::= \text{skip} \mid v := e \mid \dots \mid \underline{c_0 \circ c_1} \mid \underline{c_0 \text{ par } c_1}$

$$(Nd_1) \quad \frac{-}{\langle c_0 \circ c_1, G \rangle \xrightarrow{c} \langle c_0, G \rangle}$$

$$(Nd_2) \quad \frac{-}{\langle c_0 \circ c_1, G \rangle \xrightarrow{c} \langle c_1, G \rangle}$$

$$(Par_1) \quad \frac{\langle c_0, G \rangle \xrightarrow{c} \langle c_0', G' \rangle}{\langle c_0 \text{ par } c_1, G \rangle \xrightarrow{c} \langle c_0', \text{par } c_1, G' \rangle}$$

$$(Par_2) \quad \frac{\langle c_0, G \rangle \xrightarrow{c} G'}{\langle c_0 \text{ par } c_1, G \rangle \xrightarrow{c} \langle c_1, G' \rangle}$$

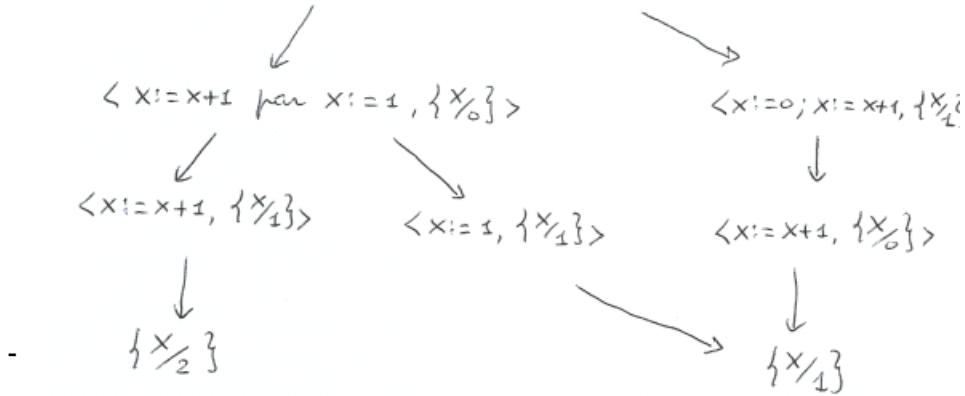
$$(Par_3) \quad \frac{\langle c_1, G \rangle \xrightarrow{c} \langle c_1', G' \rangle}{\langle c_0 \text{ par } c_1, G \rangle \xrightarrow{c} \langle c_0 \text{ par } c_1', G' \rangle}$$

$$(Par_4) \quad \frac{\langle c_1, G \rangle \xrightarrow{c} G'}{\langle c_0 \text{ par } c_1, G \rangle \xrightarrow{c} \langle c_0, G' \rangle}$$

OSSERVAZIONI

- Questo **non è più deterministico**
- Lo dimostriamo facendo un esempio:

$\langle (x := 0; x := x+1) \text{ par } x := 1, \{x_0\} \rangle$ (25)



- Due possibili risultati finali: $\{x_1\}$ o $\{x_2\}$
- Per modellare questa situazione,

$\text{exec}; \text{Com} \times \text{Store} \rightarrow \wp(\text{Store})$

$$\text{exec}\left((x := 0; x := x+1) \text{ par } x := 1, \{x_0\}\right) = \{\{x_1\}, \{x_2\}\}$$

- Da notare che abbiamo anche definito un nuovo exec
- **Per la gestione dell'errore:**

$$\circ \quad \frac{\underline{\langle c_0, \sigma \rangle \rightarrow_c \text{err}}}{\langle c_0 \text{ par } c_1, \sigma \rangle \rightarrow_c \text{err}} \quad \frac{\underline{\langle c_1, \sigma \rangle \rightarrow_c \text{err}}}{\langle c_0 \text{ par } c_1, \sigma \rangle \rightarrow_c \text{err}}$$

Domande orale di questo capitolo

04 June 2023 18:22

15. Quali sono i livelli di descrizione di un linguaggio?

Un linguaggio viene descritto su 4 livelli:

- Grammatica

Ovvero capire quali frasi sono corrette.

Si divide in due aspetti:

- **Analisi lessicale**

Usando un alfabeto sono riconosciute le corrette sequenze alfanumeriche che costituiscono i token (ovvero le parole) del linguaggio.

In un linguaggio comune, questo viene fatto con un dizionario.

Nei nostri casi, lo facciamo con le grammatiche regolari.

- **Analisi sintattica**

Ovvero capire quando una frase è corretta, ovvero **quali sequenze di token costituiscono frasi legali**.

La sintassi è quindi una relazione tra segni: tra tutte le sequenze di parole ne troviamo un sottoinsieme che costituiscono delle frasi del linguaggio.

- Semantica

Ovvero capire il significato delle frasi del linguaggio e quali di queste hanno "senso". Bisogna avere una base dalla quale si riesce a capire, non si può andare a tradurre il significato in modo ricorsivo per sempre: nel caso dei computer questa cosa è il linguaggio macchina, per le persone potrebbe essere per esempio la madrelingua.

È una relazione tra segni (ovvero le frasi corrette) e significati (entità autonome che esistono a prescindere da cosa usiamo per descriverle)

- Pragmatica

Altre regole che determinano come usare un linguaggio dal punto di vista pratico. Usare il lei al posto del tu in un contesto formale oppure evitare di usare il goto, anche se il ciò è permesso da tutte le altre regole. Essenzialmente come usare una frase dentro un contesto.

- Implementazione

Ovvero, di un linguaggio di programmazione, come esso è effettivamente implementato, ovvero come viene eseguito da un punto di vista pratico, come le frasi "operative" del linguaggio realizzano lo stato di cui stanno parlando.

16. Sintassi: qual è l'aspetto lessicale e quale quello grammaticale di un linguaggio?

Vedere risposta scritta sopra.

17. Cos'è un alfabeto? Cos'è una parola o stringa? Cos'è A*? Tale insieme è enumerabile?

Un alfabeto è una collezione di simboli.

Una parola o stringa è un insieme di simboli che vengono concatenati.

A* è la chiusura di Kleene dell'alfabeto A, che viene descritto nel seguente modo:

$$A^* = \bigcup_{n=0}^{\infty} A^n$$

Dove:

$$\begin{aligned} A^n &= A \cdot A^{n-1} = \{aw \mid a \in A \wedge w \in A^{n-1}\} \\ A^0 &= \{\varepsilon\} \end{aligned}$$

**18. Definizione di potenza di una stringa. Definizione di potenza di un linguaggio.
Definizione di chiusura / iterazione (stella di Kleene) di un linguaggio**

Potenza di una stringa:

$$w^n = \underbrace{w \cdot \dots \cdot w}_{n \text{ volte}}, \quad \text{per esempio } (ab)^3 = ababab$$

Potenza di un linguaggio:

$$L^n = L \cdot L^{n-1}, \quad L^0 = \{\varepsilon\}$$

Dove la concatenazione funziona nel seguente modo: $L_1 \cdot L_2 = \{wv \mid w \in L_1 \wedge v \in L_2\}$

Stella di Kleene

Scritta sopra

19. Definizione di grammatica libera da contesto. Come si deriva una stringa? Qual è il linguaggio generato da una grammatica libera?

Una grammatica libera da contesto è una quadrupla formata da:

(NT, T, R, S) , ovvero:

NT - Insieme dei simboli non terminali

T - Insieme dei simboli terminali

R - Insieme delle produzioni

Ovvero espressioni che sono nella forma $V \rightarrow w$, $V \in NT \wedge w \in \{T \cup NT\}^*$

$S \in NT$ - Simbolo iniziale

Fissata una grammatica $G = (NT, T, R, S)$ e due stringhe che bisogna verificare sia una la derivazione dell'altra $v \Rightarrow w$ ($v, w \in \{T \cup NT\}^*$), allora diciamo che $v \Rightarrow w$, ovvero v deriva immediatamente w, se:

$$\frac{v = xAy \quad (A \Rightarrow z) \in R \quad w = xzy}{v \Rightarrow w}$$

Vi è poi la definizione di "deriva", che è la chiusura transitiva e riflessiva della derivazione immediata, ovvero:

$$\frac{}{v \Rightarrow^* v}, \quad \frac{v \Rightarrow^* w' \quad w' \Rightarrow w}{v \Rightarrow^* w}$$

Il linguaggio generato da una grammatica libera $G = (NT, T, R, S)$ è l'insieme:

$L[G] = \{w \in T^* \mid S \Rightarrow^* w\}$, ovvero tutte le stringhe formate unicamente da simboli terminali che si possono produrre partendo dal simbolo iniziale.

20. Cos'è un albero di derivazione? Cos'è una derivazione canonica sinistra / destra? Esiste una corrispondenza biunivoca tra alberi di derivazioni e derivazioni canoniche?

Data una grammatica libera $G = (NT, T, R, S)$, un **albero di derivazione** è un albero ordinato in cui:

1. Ogni nodo è etichettato con un simbolo $T \cup NT \cup \epsilon$
2. La radice è etichettata con S
3. Ogni nodo interno è etichettato con un simbolo in NT
4. Se il nodo n
 - a. Ha etichetta $A \in NT$ e l'insieme delle etichette dei suoi figli è $\{X_i \in NT \cup T \mid \forall i \in [1, k]\}$, allora esiste una produzione $A \rightarrow X_1 \dots X_k \in R$
 - b. Se il nodo ha etichetta ϵ , allora n è una foglia, è l'unico figlio, e sia detto A suo padre, allora $A \rightarrow \epsilon$ è una produzione in R
5. Se inoltre ogni nodo foglia è etichettato su $T \cup \epsilon$, allora l'albero di derivazione si dice sia completo.

Una derivazione sinistra vuol dire che viene espanso sempre il non terminale più a sinistra usando una produzione. Quella destra è uguale ma espande quello più a destra. Derivazioni destre e sinistre possono generare gli stessi alberi.

Questa corrispondenza esiste:

- Ogni derivazione è associata ad un albero di derivazione ed ogni albero di derivazione è associato ad una derivazione.

21. Quando una grammatica è ambigua? (fare un esempio) Quando linguaggio è ambiguo? (fare un esempio).

Una grammatica è ambigua quando vi sono due produzioni che danno la stessa stringa ma due alberi di derivazione distinti.

Un esempio di grammatica ambigua è:

$$(A, a, \{A \rightarrow A + A | A - A | a\}, A)$$

Un linguaggio è ambiguo quando ogni sua grammatica risulta ambigua. Questi linguaggi si dicono inerentemente ambigui.

Non vi sono linguaggi regolari che sono inerentemente ambigui.

Un esempio di linguaggio ambiguo è:

$$\{a^n b^m c^m \mid m, n \geq 1\} \cup \{a^m b^m c^n \mid m, n \geq 1\}$$

22. È possibile rimuovere l'ambiguità dalla grammatica delle espressioni aritmetiche? Come?

Vengono fatte due cose:

- Dare una certa precedenza agli operandi (Per esempio indicare che il * ha una precedenza maggiore del +). Questo genera dell'associatività intrinseca a come una stringa esce
- La prima operazione non basta per mantenere tutta l'espressività delle espressioni aritmetiche: si aggiungono le parentesi, che fungono da zucchero sintattico, ovvero servono per dare tutto il potere espressivo al linguaggio. Poi queste verranno tolte dall'effettivo albero sintattico che verrà generato.

23. **Cos'è l'albero di sintassi astratta? Che differenza c'è tra sintassi concreta e sintassi astratta? Cos'è lo zucchero sintattico?**

L'albero di sintassi astratta è un albero di derivazione che soddisfa i vincoli contestuali di un linguaggio in cui compaiono solo i terminali.

La sintassi concreta è una grammatica non ambigua che fa uso di zucchero sintattico. La sintassi astratta è una grammatica semplice ed intuitiva, ma ambigua. Dall'albero di derivazione da sintassi concreta ne estraiamo un albero sintattico da sintassi astratta, detto albero sintattico astratto.

Per zucchero sintattico si intendono simboli atti solo a chiarire la derivazione di una stringa ed è essenzialmente ciò che differenza la sintassi astratta e la sintassi concreta di un linguaggio.

24. **Fare esempi di vincoli sintattici contestuali. Possono essere catturati attraverso grammatiche libere?**

Un esempio di vincolo sintattico contestuale è per esempio capire se il numero di parametri formali coincide con il numero di parametri attuali di una funzione e della sua chiamata. Dobbiamo sapere, appunto, quanti questi sono ed il ciò dipende dal contesto nel quale siamo.

Un'altro esempio è che un identificatore deve essere dichiarato prima dell'uso.

Un'altro ancora è che prima di usare una variabile ci deve essere un assegnamento su di essa.

Questo non può essere catturato attraverso le grammatiche libere, ma solo attraverso:

- Grammatiche dipendenti da contesto, molto complesse da scrivere
- Metodi ad hoc, che sistemanano il dubbio attraverso controlli specifici.

25. **Cosa s'intende per semantica statica? E per semantica dinamica?**

Praticamente, nei linguaggi di programmazione, si va a fare una distinzione all'interno della grammatica: anche controlli che dovrebbero essere eseguiti nella grammatica, ovvero la correttezza delle frasi, vengono "posticipati" per essere fatti dall'analizzatore semantico.

All'interno dei linguaggi di programmazione pertanto:

- Analisi sintattica:
 - Tutto ciò che fa una BNF
- Analisi semantica:
 - Analisi sintattica contestuale
 - E anche altro

Quindi, all'interno di questa seconda fase, si differenziano:

- Semantica statica
 - Ovvero tutti quei controlli che possono essere eseguiti prima dell'esecuzione,

analizzando il codice del programma

- Semantica dinamica
 - o Ovvero vengono eseguiti controlli su una rappresentazione formale dell'esecuzione del programma, che possono mostrare problemi durante l'esecuzione effettiva di esso.
 - o Per esempio, dividere per l'input dell'utente, quando questo input potrebbe essere 0, è un errore di semantica dinamica.

26. Elencare le varie fasi in cui si articola un compilatore. Descrivere in dettaglio ogni singola fase.

Vi sono 6 fasi:

- Analisi lessicale
 - o In questa fase si controlla che tutte le singole parole siano corrette e si vanno a generare dei token che vengono aggiunti alla tabella dei simboli
- Analisi sintattica
 - o Vengono controllate le "frasi" e si generano gli alberi di derivazione
 - Ogni foglia di tale albero è costituita da un token contenuto nella lista e leggendo da sinistra a destra le foglie dell'albero si deve ottenere una frase legale per il linguaggio
 - o Si eseguono controlli sul numero di parentesi, sulla struttura del codice (if,else,loop etc.) e se rispettano come dovrebbe venire scritto
- Analisi semantica
 - o Vengono eseguiti controlli di semantica statica, ovvero si verificano se i parametri attuali corrispondono con quelli formali, che le variabili hanno un tipo, etc.
 - o Viene arricchito l'albero di derivazione con informazioni sui tipi dei token e altre informazioni.
- Codice intermedio
 - o Viene generato un codice intermedio che è sia indipende dall'architettura che dal sorgente.
 - o È formato genericamente da operazioni con 3 operandi ($x = y + z$ per esempio)
 - o È facile da produrre, in quanto segue al struttura dell'albero sintattico, ed è facile da tradurre, in quanto è molto più semplice
- Ottimizzazione forma intermedia
 - o Rimozione di codice inutile
 - o Espansioni in-line di chiamate di funzione
 - o Fattorizzazione di sottoespressioni per non calcolare più volte lo stesso valore
 - o Mette fuori dai cicli le sottoespressioni che non variano
 - o Altro, dipendentemente dal compilatore
- Generazione del codice
 - o Viene generato il codice oggetto per una specifica architettura (include assegnazione dei registri e ottimizzazioni specifiche per l'architettura e il codice oggetto.)

27. A chi serve definire la semantica di un linguaggio e perché?

Definire la semantica di un linguaggio di programmazione è molto più difficile della sintassi. La complessità nasce dall'esigenza di mediare tra due istanze contrapposte: la ricerca dell'esattezza, da una parte, e quella della flessibilità dall'altra in modo da rimuovere l'ambiguità per un utente, ma anche da lasciare spazio all'implementazione.

A chi serve la semantica dinamica?

- o Al programmatore
 - Analisi del programma
 - Deve sapere esattamente cosa debba fare il suo programma

- Deve poter dimostrare proprietà del suo programma ("termina sempre per ogni input",...)
- Al progettista del linguaggio
 - Strumento di specifica del linguaggio
 - Deve poter dimostrare proprietà del linguaggio ("è Turing completo?",...)
- All'implementatore del linguaggio
 - Riferimento per dimostrare la correttezza dell'implementazione

28. Quali tecniche si usano per dare semantica ad un linguaggio di programmazione?

Vi sono 2 tecniche:

- **Operazionale:** (Macchina astratta a stati e transizioni)
 - Si costruisce una specie di automa che, passo a passo, mostra l'effetto della esecuzione delle varie istruzioni
 - Enfasi su **come** si calcola
 - Vedremo l'uso di questa semantica per un semplice linguaggio imperativo
- **Denotazionale:**
 - Si associa ad ogni programma sequenziale una funzione da input e un output (incluse strutture ausiliarie dette ambiente e memoria)
 - Enfasi su **cosa** si calcola (vengono nascosti i passi intermedi del calcolo)

29. Imparare le regole di semantica operazionale SOS per il semplice linguaggio presentato a lezione. Regole di valutazione interna-sinistra ed esterna-sinistra.

Tante cose da scrivere, non ho voglia

30. Cosa s'intende per pragmatica? Cosa si intende per implementazione di un linguaggio?

Pragmatica:

Insieme di "regole" e consigli su come usare un linguaggio:

- Usare "while" per cicli indeterminati, "for" per cicli predeterminati.
- Non usare il "go-to" per saltare da una parte all'altra
- ...

Implementazione:

Ovvero la scrittura di un compilatore (o un interprete) per una macchina ospite già realizzata, costruendo così una macchina astratta per il linguaggio.

- Correttezza dell'implementazione?
 - Bisogna dimostrare che preserva la semantica del linguaggio, ovvero che il programma sorgente e quello oggetto calcolano la stessa funzione (per i linguaggi sequenziali).
- Costo dell'implementazione
 - È il compilatore in grado di produrre codice efficiente?
 - Certi costrutti linguistici, scelti dal progettista del linguaggio, sono troppo onerosi in pratica?

Introduzione

venerdì 21 ottobre 2022 12:27

SLIDES (Lez 6): https://virtuale.unibo.it/pluginfile.php/1295154/mod_resource/content/0/Lez06-Gorrieri.pdf
Per più dettagli sugli automi non deterministici: https://en.wikipedia.org/wiki/Nondeterministic_finite_automaton
SLIDES (Lez 7): https://virtuale.unibo.it/pluginfile.php/1295155/mod_resource/content/0/Lez07-Gorrieri.pdf

1. Espressioni Regolari

venerdì 21 ottobre 2022 12:28

1.1 - Analisi lessicale attraverso linguaggi regolari.

Si tratta di riconoscere nella stringa in ingresso gruppi/sequenze di simboli che corrispondono a specifiche categorie sintattiche (ad esempio identificatori, parole riservate, operatori aritmetici,...)

La stringa in input è trasformata in una sequenza di simboli astratti, detti token:

programma input → *Analizzatore lessicale* → *Lista token*
Scanner

1.2 - Cosa è un token?

Token = coppia (nome, valore)

- **Nome**
 - Simbolo astratto che rappresenta una categoria sintattica
- **Valore**
 - Una sequenza di simboli del testo in ingresso

ESEMPIO:

$\langle \text{Ide}, x_1 \rangle$

Questo token è formato da:

- **Nome (Ide)**
 - È l'informazione che identifica una classe di token (**identificatori**)
- **Valore (x1)**
 - È l'informazione che identifica uno specifico token (l'identificatore x1)
- **Pattern**
 - È la descrizione generale della forma dei valori di una classe di token, ad esempio
$$(x \mid y)(x \mid y \mid 0 \mid 1)^*$$
Espressione regolare
- **Lessema**
 - È una stringa istanza di un pattern.
 - Nel nostro esempio x1 è un lessema istanza del pattern descritto sopra.

IMPORTANTE

Vedremo che ad **ogni nome di categoria sintattica** si associa un **pattern**, e ogni valore è un lessema.

OSSERVAZIONE:

Per completezza, specifichiamo che lo scanner non associa effettivamente i valori trovati come valore, ma **associa dei puntatori ai valori che si trovano nella tabella dei simboli**.

$\langle \text{Ide}, x \rangle$ è in realtà $\langle \text{Ide}, \text{puntatore a } x \rightarrow \text{nella tabella dei simboli} \rangle$

1.3 - ESPRESSIONI REGOLARI

Fissato un alfabeto $A = \{a_1, a_2, \dots, a_n\}$, definiamo le espressioni regolari su A con la seguente BNF

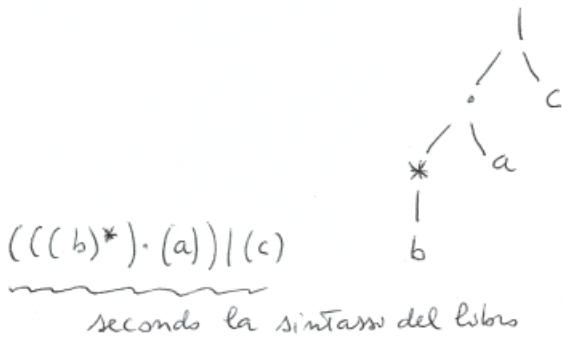
$$r ::= \emptyset \mid \varepsilon \mid a_{\forall a \in A} \mid r \cdot r \mid r \mid r \mid r^*$$

(È una grammatica astratta ambigua, per disambiguarla si possono usare delle parentesi)

Per semplicità, si assume che:

- La concatenazione, la disgiunzione (e anche la ripetizione) associano a **sinistra**.
- La precedenza tra operatori sia: $* > \cdot > |$
- La concatenazione \cdot è di solito omessa

Per cui, per esempio $b^*a|c$ corrisponde all'albero sintattico:



1.4 - LINGUAGGIO DENOTATO DA UN'ESPRESSIONE REGOLARE

Dato l'alfabeto A , definiamo la funzione

$$\mathcal{L} : \text{Exp. reg} \rightarrow \mathcal{P}(A^*)$$

Nota: $\mathcal{P}(A^)$*

Con \mathcal{P} si intende l'insieme delle parti dell'insieme A^* , quindi l'insieme di tutti i possibili sottoinsiemi di tutte le combinazioni dell'alfabeto A . Peranto, un elemento di $\mathcal{P}(A^*)$ al quale si può associare un'espressione regolare, è un sottoinsieme dell' A^* che è formato delle cose che l'espressione regolare esprime. $\mathcal{L}[(a|b)c] = \{bc, ac\} \subseteq A^* = \bigcup_{n \geq 0}^{\infty} A^n = \bigcup_{n \geq 0}^{\infty} \{aw \mid a \in A \wedge w \in A^n\}$

Definita nel seguente modo:

- $\mathcal{L} \left[\begin{array}{l} \emptyset \\ \text{simbolo di} \\ \text{exp. reg} \end{array} \right] = \begin{array}{l} \emptyset \\ \text{linguaggio} \\ \text{vuoto} \end{array}$
- $\mathcal{L}[\varepsilon] = \begin{array}{l} \{\varepsilon\} \\ \text{linguaggio} \\ \text{con solo} \\ \text{la stringa} \\ \text{vuota} \end{array}$
- $\mathcal{L}[a] = \{a\}$
- $\mathcal{L}[r_1 \cdot r_2] = \mathcal{L}[r_1] \cdot \mathcal{L}[r_2]$
- $\mathcal{L}[r_1|r_2] = \mathcal{L}[r_1] \cup \mathcal{L}[r_2]$
- $\mathcal{L}[r^*] = (\mathcal{L}[r])^*$

- $\mathcal{L}[(r)] = \mathcal{L}[r]$

NOTA:

$$\begin{aligned} L_1 \cdot L_2 &= \{xy \mid x \in L_1 \wedge y \in L_2\} \\ L_1 \cup L_2 &= \{x \mid x \in L_1 \vee x \in L_2\} \\ L^0 &= \{\varepsilon\}, \quad L^{n+1} = L \cdot L^n \\ L^* &= \bigcup_{n \geq 0}^{\infty} L^n, \quad L^+ = \bigcup_{n \geq 1}^{\infty} L^n \end{aligned}$$

1.5 - LINGUAGGIO REGOLARE

- **DEFINIZIONE**

Un linguaggio $L \subseteq A^*$ è detto regolare $\Leftrightarrow \exists \text{exp. reg}$ tale che $L = \mathcal{L}[r]$

- **PROPOSIZIONE**

Ogni linguaggio finito è regolare

(Dimostrazione: puoi mettere | tra tutte le stringhe del linguaggio e avrai un elenco di stringhe, che sono un'espressione regolare, separate da |).

- **OSSERVAZIONE**

Esistono linguaggi regolari infiniti:

$$\begin{aligned} \mathcal{L}[a^*b] &= \mathcal{L}[a^*] \cdot \mathcal{L}[b] = \{a\}^* \cdot \{b\} = \\ &= \bigcup_{n \geq 0} \{a\}^n \cdot \{b\} = \{\varepsilon, a, aa, \dots\} \cdot \{b\} = \\ &= \{a^n b \mid n \geq 0\} \end{aligned}$$

1.6 - ALTRI OPERATORI AUSILIARI

- **RIPETIZIONE POSITIVA**

$r^+ = r^*r$ oppure rr^*

- **POSSIBILITA'**

$r? = (r|\varepsilon)$

- **ELENCO**

$[a_1, \dots, a_n] = (a_1 | \dots | a_n)$

$[a_1 - a_n] = (a_1 | \dots | a_n)$

$[a - zA - Z]$ per indicare tutte le lettere dell'alfabeto

$[0 - 9]$ per indicare tutte le cifre decimali

- **ESEMPIO**

Vedere slides a pagina 9

1.7 - DEFINIZIONI REGOLARI

Una definizione regolare su un alfabeto A è costituita da **una lista di definizioni**:

$$\begin{aligned} d_1 &:= r_1 \\ d_2 &:= r_2 \\ &\vdots \\ d_k &:= r_k \end{aligned}$$

Dove i valori d_i sono simboli "nuovi" e ogni r_i è una espressione regolare sull'alfabeto esteso $A \cup$

ESEMPIO:

numconsegno := *segno* *cifre* (\cdot *cifre*)?
segno := $-$ | $+$
cifre := *cifra*⁺
cifra := [0 – 9]

1.8 - EQUIVALENZE TRA ESPRESSIONI REGOLARI

DEFINIZIONE

Due espressioni regolari r ed s si dicono equivalenti $\Leftrightarrow \mathcal{L}[r] = \mathcal{L}[s]$ (ovvero, denotano lo stesso linguaggio).

Denotiamo con questa relazione con $r \simeq s$

Esistono diverse leggi per \simeq , alcune sono le seguenti:

- $r|s \simeq s|r$, | commutativa
- $r|(s|t) \simeq (r|s)|t$, | associativa
- $r|r \simeq r$, | idempotente
- $r \cdot (s \cdot t) \simeq (r \cdot s) \cdot t$ · associativa
- $\varepsilon \cdot r \simeq r \simeq r \cdot \varepsilon$ ε elemento neutro
- $(r^*)^* = r^*$ * è idempotenza
- $r(s|t) = rs|rt$ · distributiva a sinistra su |
- $(r|s)t = rt|st$ · distributiva a destra su |
- *Tante altre cose per il vuoto e ε che si possono leggere a slide 10*

2. Automi a stati finiti

venerdì 21 ottobre 2022 12:32

2.1 - DESCRIZIONE INIZIALE

2.1.1 - Quale problema dobbiamo risolvere?

Abbiamo una **memoria finita**. (Quindi eseguiamo un insieme finito di controlli, che sono gli "stati" q_0, \dots, q_n)
Dobbiamo **leggere una stringa in input** e riconoscerla.

Dobbiamo **dare un output** (se l'abbiamo riconosciuta o no)

2.1.2 - Come viene fatto?

Descrizione delle caratteristiche:

Abbiamo una **testina di lettura**, la quale parte **dal primo elemento** della stringa in input e che legge un carattere alla volta.

Il **controllo parte dallo stato** iniziale dei controlli finiti chiamato q_0 .

Funzionamento dei controlli:

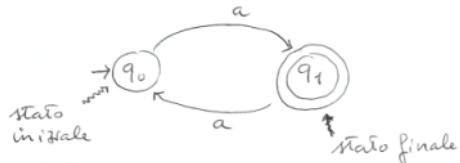
Ripetere:

- i. Leggere il carattere in input, usando la testina di lettura
 - ii. In base allo stato attuale decide:
 - Di cambiare stato (in base alla condizione)
 - Sposta la testina di lettura sull'input successivo (Se c'è ancora input)
- }
- } Fino a che:
- Ha finito di leggere l'input (e riconosce la stringa se ha raggiunto uno **stato finale**)
 - Si è bloccato in quanto per la coppia (stato corrente, input attuale) non era specificato lo stato successivo.

Questa funzione può essere rappresentata usando un **diagramma di transizione**.

2.2 - DIAGRAMMA DI TRANSIZIONE

Il funzionamento di un automa finito è ben rappresentato attraverso un diagramma di transizione:



(Nota: **stato iniziale mostrato con una freccetta**, la quale non parte da nulla e non vuol dire che c'è uno stato omesso prima, ma è un modo per mostrare che q_0 è lo stato dal quale inizia il diagramma di transizione, mentre abbiamo che q_1 è uno stato finale in quanto ha un **doppio bordo**)

Il diagramma di transizione sopra rappresenta: $L = \{a^{2n+1} \mid n \geq 0\} = \{a^n \mid n \text{ è dispari}\} = L[a(aa)^*]$

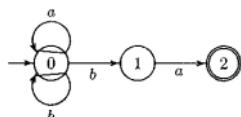
Riconoscere una stringa w vuol dire **trovare un cammino etichettato w sul grafo a partire dallo stato iniziale e che finisce su uno stato finale**.

(Nota: se in questo automa l'input $w = ab$ allora l'automa si blocca in q_1 , senza aver completato la lettura dell'input w)

2.3 - AUTOMI FINITI NON DETERMINISTICI

ESEMPIO

Se noi dobbiamo tradurre $(a|b)^*ba$ potremmo, inizialmente pensarlo di definire come segue:



Tuttavia notiamo che uno stesso input non necessariamente implica uno stesso output: si tratta quindi di un automa non deterministico.

Se fosse deterministico, ogni input offrirebbe una e una sola mossa di cambio di stato.

NOTE PER IL LINGUAGGIO E ACRONIMI USATI:

Automa finito = Automa a stati finiti

Automa finito non deterministico = NFA

Automa finito deterministico = DFA

NFA sta per "Non deterministic finite state automaton"

DFA sta per "Deterministic finite state automaton"

DEFINIZIONE - Automa finito non deterministico (NFA, "Non deterministic finite state automaton")

Un automa finito non deterministico (NFA) è una quintupla $(\Sigma, Q, \delta, q_0, F)$ dove:

1) Σ è un alfabeto finito

i. Corrisponde con i simboli in input

2) Q è un insieme finito

i. I suoi elementi sono $q_0, \dots q_k$ e sono gli stati

3) δ è la funzione di transizione con tipo

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

Che dunque associa un insieme di stati ad ogni coppia (q, a) formata da uno stato e un simbolo di input.

Nota: $\mathcal{P}(Q) = \text{Insieme delle parti di } Q$, ovvero se $Q = \{q_0, q_1\}$, allora $\mathcal{P}(Q) = \{\emptyset, \{q_0\}, \{q_1\}, \{q_0, q_1\}\}$. Quindi, quando parto da un certo stato q , con un input a , potrei avere associato uno degli insiemi che sono elementi dell'insieme delle parti.

Inoltre, avere $\delta(q, a) = \emptyset$ significa che non vi sono archi etichettati a uscenti da q .

(Il significato è (quasi) equivalente a: $\delta(q, \sigma) = Q' \subseteq Q$)

4) $q_0 \in Q$ (Stato iniziale)**5) $F \subseteq Q$ (Stati finali)****NOTA SUGLI ALBERI NFA**

Gli alberi NFA sono:

- o Comodi, in quanto sono facili e intuitivi da costruire
- o Inefficienti, in quanto per capire se una stringa è accettabile (vedere definizione sotto) ci possono essere tante strade alternative, che implicano una possibilità di fallire e di necessitare backtracking

Possiamo sempre aggiungere degli archi vuoti ad un NFA, pertanto per un linguaggio L , se esiste un NFA N tale che $L = \mathcal{L}[N]$, allora esistono infiniti $N1, N2, \dots, Nk, \dots$ tali che $L = \mathcal{L}[N_i]$ per $i = 1, 2, \dots$

NOTA IMPORTANTE (Archi etichettati ϵ)

La definizione tiene conto del fatto che alcuni archi possono essere etichettati con la stringa vuota ϵ . Tali archi possono essere attraversati "silenziosamente", senza consumare simboli in input.

NOTA (CURIOSITA')

Questo tipo di NFA è anche definito come NFA- ϵ in quanto include anche le ϵ "moves". È una generalizzazione degli NFA, in quanto quelli che non contengono ϵ moves sono un sottoinsieme proprio degli NFA- ϵ . Un NFA senza ϵ moves ha la funzione di transizione definita come: $\delta: Q \times \Sigma \rightarrow \mathcal{P}(Q)$

DEFINIZIONE - Stringa accettata

Un NFA $N = (\Sigma, Q, \delta, q_0, F)$ accetta la stringa $x = a_1 \dots a_n \Leftrightarrow$ nel diagramma di transizione esiste un cammino che inizia in q_0 e termina in uno stato di F nel quale la stringa che si ottiene concatenando le etichette degli archi percorsi è esattamente x .

DESCRIZIONE FORMALE

- o Descrizione istantanea:

$$(\begin{array}{c} q \\ \text{stato corrente} \end{array} , \begin{array}{c} w \\ \text{input da leggere} \end{array})$$

- o Mossa:

$$\frac{q' \in \delta(q, \sigma)}{(q, \sigma w) \vdash_N (q', w)}, \quad \sigma \in \Sigma \cup \{\epsilon\}, \quad w \in \Sigma^*$$

- o Cammino: (Chiusura riflessiva e transitiva di \vdash_N)

$$\frac{(q, w) \vdash_N^* (q', w') \quad (q', w') \vdash_N (q'', w'')}{(q, w) \vdash_N^* (q'', w'')}$$

- o Accettazione / Riconoscimento:

$$w \in \mathcal{L}[N] \Leftrightarrow \exists q \in F. (q_0, w) \vdash_N^* (q, \epsilon)$$

DEFINIZIONE - Linguaggio accettato

Il linguaggio accettato da un NFA N è l'insieme $\mathcal{L}[N] = \{x \in \Sigma^* \mid N \text{ accetta } x\}$

DEFINIZIONE - NFA equivalenti

Due NFA $N1$ e $N2$ si dicono equivalenti \Leftrightarrow accettano lo stesso linguaggio, cioè se $\mathcal{L}[N1] = \mathcal{L}[N2]$

2.4 - AUTOMI FINITI DETERMINISTICI**DEFINIZIONE - DFA**

Un automa finito deterministico (DFA) è una quintupla $(\Sigma, Q, \delta, q_0, F)$ dove Σ, Q, q_0 e F sono **definito come**
per un NFA, mentre la funzione di transizione δ ha tipo:

$$\begin{aligned}\delta: Q \times \Sigma &\rightarrow Q \\ (\delta(q, \sigma)) &= q'\end{aligned}$$

OSSERVAZIONI:

Un DFA è un particolare tipo di NFA tale che:

- $\forall q \in Q, \quad \delta(q, \varepsilon) = \emptyset$
(Non ci sono transizioni ε)
- $\forall \sigma \in \Sigma, \forall q \in Q \quad \exists q' \in Q. \quad \delta(q, \sigma) = \{q'\}$
(L'insieme delle mosse possibili è sempre un singoletto)

PROPOSIZIONE

Per ogni NFA, è possibile costruire un DFA ad esso equivalente

Come fare?

Idea: seguire contemporaneamente tutti i possibili cammini alternativi dell'NFA \Rightarrow gli stati del DFA che andiamo a costruire sono costituiti da insiemi di stati dell'NFA.

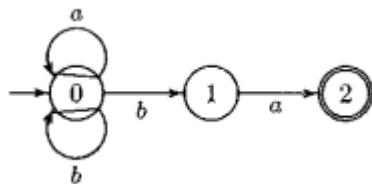
3. Costruire automi deterministici

sabato 22 ottobre 2022 22:59

3.0 - Esempio di traduzione di NFA in DFA

Prendiamo l'espressione regolare $(a|b)^*ba$.

Essa può essere rappresentata usando le transizioni dell'automa non deterministico che seguono:



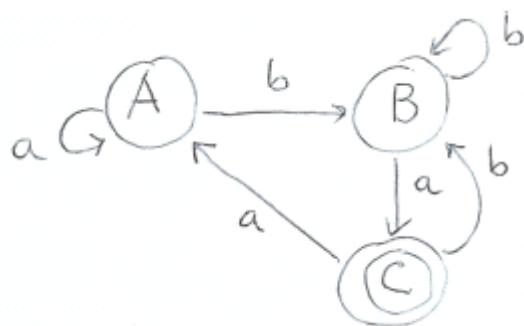
$$\begin{aligned}\delta(q_0, a) &= \{q_0\} \\ \delta(q_0, b) &= \{q_0, q_1\} \\ \delta(q_1, a) &= \{q_2\}\end{aligned}$$

Con anche:

$$\begin{aligned}q_0 &= \text{start} \\ F &= \{q_2\}\end{aligned}$$

Notiamo che qua, abbiamo che per b abbiamo 2 possibili mosse in q_0 , mentre poi abbiamo che q_1, q_2 non hanno una mossa per ogni lettera $\in \Sigma$.

Possiamo scrivere un automa deterministico che si occupa dello stesso scopo nel modo seguente:



$$\begin{aligned}\delta(A, a) &= A \\ \delta(A, b) &= B \\ \delta(B, a) &= C \\ \delta(B, b) &= B \\ \delta(C, a) &= A \\ \delta(C, b) &= B\end{aligned}$$

Con stato iniziale A e stati terminali {C}

Notiamo che stiamo considerando tutti i possibili cammini che si possono scegliere, a partire da ogni nodo, per ogni lettera dell'alfabeto.

Da un punto di vista intuitivo, è come se noi lavorassimo per capire la nostra soluzione ad

ogni passaggio: se iniziamo a riconoscere il pattern finale ("ba" terminali dell'*exp. reg*), allora ci avviciniamo a C.

Se noi per esempio arrivassimo a leggere *aab* possiamo immaginare di leggere una *a* subito dopo, e terminare la lettura in quanto *aaba* è valido. Se invece dopo l'ultima *a* ce ne fosse un'altra, allora non siamo nel "caso" del pattern delle ultime due lettere *ba* e dobbiamo "riiniziare la lettura per matchare questo pattern", ripartendo dal nodo iniziale dove quindi continuiamo a considerare *a|b*.

3.1 - ε – closure e mosse

Sia q uno stato di un NFA. **La ε – closure di q è l'insieme degli stati raggiungibili da q solo con mosse ε .**

$$\overline{\{q\} \subseteq \varepsilon - \text{closure}(q)}, \quad \frac{p \in \varepsilon - \text{closure}(q)}{\delta(p, \varepsilon) \subseteq \varepsilon - \text{closure}(q)}, \quad (\text{Come prima } p \text{ si prenderà } q)$$

Sia P un insieme di stati di un NFA

$$\varepsilon - \text{closure}(P) = \bigcup_{p \in P} \varepsilon - \text{closure}(p)$$

$$\text{mossa} : \mathcal{P}(Q) \times \Sigma \rightarrow \mathcal{P}(Q)$$

$$\text{mossa}(P, a) = \bigcup_{p \in P} \delta(p, a)$$

$$\Delta(A, b) = \varepsilon - \text{closure}(\text{mossa}(A, b))$$

Quest'ultima rappresenta la funzione di transizione del DFA

3.1.2 - DEFINIZIONE DELL'ALGORITMO PER IL CALCOLO DELL' ε – closure

```

Inizializzazione T = P;
Inizializzazione ε – closure(P) = P;
while T ≠ Ø do {
    scegli un r ∈ T e rimuovilo da T
    for each s ∈ δ(r, ε) do {
        if s ∉ ε – closure(P){
            add s to ε – closure(P);
            add s to T
        }
    }
}
}
```

Spiegazione dell'algoritmo:

Calcolo la ε – closure per un insieme di stati. Prendo ognuno di questi stati e ci faccio un'operazione sopra: per ogni relazione del tipo $\delta(r, \varepsilon)$ (Quindi che partono dallo stato che sto guardando e hanno un etichetta ε), se ancora non ho aggiunto questa transizione all'elenco della ε -closure, allora lo aggiungo e aggiungo anche lo stato di "arrivo" all'interno dell'elenco degli stati che devo analizzare.

È un po' come fare un BFS dei grafi, solo che non aggiungi tutti i nodi adiacenti diversi da quelli già

marcati, ma aggiungi solo quelli i cui archi hanno una certa etichetta se non sono già stati visitati.

3.1.3 - DEFINIZIONE MATEMATICA DI LINGUAGGIO RICONOSCIUTO USANDO ϵ – closure:

Definizione del prof:

$$\begin{aligned}\hat{\delta} : Q \times \Sigma^* &\rightarrow \mathcal{P}(Q) \\ \hat{\delta}(q, \epsilon) &= \text{closure}(q) \\ \hat{\delta}(q, xa) &= \epsilon - \text{closure}(P) \\ \text{dove } P &= \{p \in Q \mid \exists r \in \hat{\delta}(q, x) \wedge p \in \delta(r, a)\}\end{aligned}$$

$$w \in L[N] \Leftrightarrow \exists p \in F \text{ tale che } p \in \hat{\delta}(q_0, w)$$

Da Wikipedia:

In words, $\hat{\delta}(r, x)$ is the set of all states reachable from state r by consuming the string x . The string w is accepted if some accepting state in F can be reached from the start state q_0 by consum-

Sempre da Wikipedia, ma meglio (E denota ϵ – closure):

Prese dalla pagina Wikipedia inglese di "Nondeterministic finite automaton", link nell'"Introduzione".

Insieme di spiegazioni migliori che ho trovato a riguardo, leggendo la parte informale si capisce molto l'idea che c'è dietro queste definizioni. Inoltre, la parte formale è equivalente a quella definita dal professore.

$\delta^*(q, w)$ denota il set di tutti gli stati raggiungibili dall'automa dopo aver iniziato con lo stato $q \in Q$ ed aver consumato la stringa $w \in \Sigma^*$.

La funzione $\delta^* : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$ può essere definito in modo ricorsivo come segue:

$$\delta^*(q, \epsilon) = E(q), \forall q \in Q$$

Informalmente:

Leggere la stringa vuota può portare l'automa da uno stato q ad un qualsiasi altro stato che si trova nell' ϵ -closure di q .

$$\delta^*(q, wa) = \bigcup_{r \in \delta^*(q, w)} E(\delta(r, a)), \quad \forall q \in Q, \forall w \in \Sigma^*, \forall a \in \Sigma$$

Informalmente:

Leggere una stringa w può portare l'automa da uno stato q ad un qualsiasi stato r attraverso una computazione ricorsiva di $\delta^*(q, w)$;

Dopo di questo, leggere il simbolo a lo porta da uno stato r ad un qualsiasi altro stato dell' ϵ -closure di $\delta(r, a)$

Si dice che un automa accetta una stringa w se:

$$\delta^*(q_0, w) \cap F \neq \emptyset$$

Informalmente:

Se consumando w , l'automa viene portato da uno stato q_0 ad un qualche stato terminale di F .

3.1.4 - DEFINIZIONE DELL'ALGORITMO PER IL CALCOLO DEI SOTTOINSIEMI

Ovvero, algoritmo usato per calcolare A, B, C, \dots che denotano gli stati di un automa deterministico.

```
Dato un NFA  $N = (\Sigma, Q, \delta, q_0, F)$ 
Inizializzazione  $S = \epsilon - closure(q_0);$  S è lo stato iniziale del DFA
Inizializzazione  $T = \{S\};$  T è l'insieme degli stati del DFA
while  $P \in T$  non marcato do {
    marca  $P$ 
    for each  $a \in \Sigma$  do {
         $R = \epsilon - closure(mossa(P, a))$ 
        if  $R \notin T$  {
            add  $R$  to  $T$ 
        }
        definisci  $\Delta(P, a) = R$ 
    }
}
```

Spiegazione dell'algoritmo:

Prima di tutto calcoliamo lo stato iniziale del DFA, dato da tutti gli stati raggiungibili attraverso mosse ϵ a partire dallo stato iniziale q_0 .

A partire da questo insieme di stati definiti come S :

Guardo, per ogni lettera dell'alfabeto, l'insieme $\epsilon - closure$ che posso raggiungere dopo aver fatto la mossa della lettera dell'alfabeto che sto guardando da ogni stato appartenente ad S . Quindi: per ogni stato raggiungibili da q_0 con mosse ϵ , faccio la mossa a se è possibile: se lo è, allora lo stato che raggiungo e ogni stato raggiungibile da mosse ϵ fanno parte di quell'insieme.

Se il nuovo insieme di stati trovato ancora non è stato mai trovato, allora viene aggiunto come nuovo stato del DFA e definisco il movimento dallo stato prima per la lettera che guardavo come questo nuovo stato che ho creato. Se già ci fosse, allora lo ridefinisco come uno stato che già appartiene all'insieme di stati del DFA.

3.1.5 - DEFINIZIONE - DFA M_N

Definiamo il DFA $M_N = (\Sigma, T, \Delta, \epsilon - closure(q_0), F)$ dove $R \in F \Leftrightarrow \exists q \in R . q \in F$

Questa è la definizione del nostro DFA a partire da un NFA.

- Nel caso pessimo, $T = \mathcal{P}(Q)$, cioè il DFA M_N costruito a partire dall'NFA N ha un numero di stati pari a 2^n , dove $n = |Q|$. Notiamo che quindi è esponenziale.
- Di solito però, T è molto più piccolo di $\mathcal{P}(Q)$

Si possono andare a vedere degli esempi di casi pessimi nelle slide 27-28 della lezione 6.

3.2 - EQUIVALENZA NFA - DFA

3.2.1 - TEOREMA

Sia $N = (\Sigma, Q, \delta, q_0, F)$ un NFA e sia M_N l'automa ottenuto con la costruzione per sottoinsiemi. Allora M_N è un DFA e si ha che:

$$L[N] = L[M_N]$$

3.2.2 - COROLLARIO

La classe di linguaggi riconosciuti dagli NFA coincide con la classe di linguaggi riconosciuti dai DFA.

3.2.3 - DEMOSTRAZIONE

Sia $N = (\Sigma, Q, \delta, q_0, F)$ un NFA e sia $M_N = (\Sigma, T, \Delta, \varepsilon - closure(q_0), F)$ l'automa ottenuto con l'algoritmo.

Dobbiamo dimostrare che M_N :

- È deterministico
 - $\Delta(A, a)$ è definita $\forall A \in T, \forall a \in \Sigma$
 - $\Delta(A, a)$ è univoca, in quanto il risultato di $\Delta(A, a)$ è un altro (unico) elemento di T.
- Quindi M_N è deterministico

Ci rimane da dimostrare che $L[N] = L[M_N]$

Definizioni e osservazioni prima di procedere alla dimostrazione:

- Per un DFA, $\forall R \in T, \varepsilon - closure(R) = R$ perché non ci sono mosse ε .
- Definiamo $i_M = \varepsilon - closure(q_0)$ lo stato iniziale di M_N

Dobbiamo dimostrare che $L[N] = L[M_N]$

Ovvero che $\{x \in \Sigma^* | N \text{ accetta } x\} = \{x \in \Sigma^* | M_N \text{ accetta } x\}$

Che un automa accetti la stringa x , l'abbiamo definito in modo formale come:

$$x \text{ accettato da } N \Leftrightarrow \exists p \in F \text{ tale che } p \in \hat{\delta}(q_0, w)$$

Partiamo con il dimostrare che:

$$\forall w \in \Sigma^*, \quad \hat{\delta}(q_0, w) = \hat{\Delta}(i_m, w)$$

Ovvero cerchiamo l'equivalenza dell'insieme di stati raggiungibili a partire da q_0 e consumando la stringa w con l'insieme di stati raggiungibili da i_m consumando la stringa w .

Lo dimostriamo per induzione sulla lunghezza di w :

Caso base: $|w| = 0$, ovvero $w = \varepsilon$

$$\hat{\delta}(q_0, \varepsilon) = \varepsilon - closure(q_0)$$

$$\Delta(i_M, \varepsilon) = \varepsilon - closure(i_M) = i_M = \varepsilon - closure(q_0)$$

Passo induttivo: $w = xa$, con $a \in \Sigma$ e $x \in \Sigma^*$

Per ipotesi induttiva, sappiamo che:

$$\hat{\delta}(q_0, x) = \hat{\Delta}(i_m, x) = \{P_1, \dots, P_K\}$$

Per definizione di $\hat{\delta}$

$$\hat{\delta}(q_0, xa) = \varepsilon - closure\left(\bigcup_{i=1}^k \delta(P_i, a)\right)$$

Similmente

$$\widehat{\Delta}(i_M, xa) = \Delta(\{P_1, \dots, P_k\}, a)$$

In base all'algoritmo, la definizione di Δ ci dice che:

$$\begin{aligned}\Delta(\{P_1, \dots, P_k\}, a) &= \varepsilon - \text{closure}(\text{mossa}(\{P_1, \dots, P_k\}, a)) \\ &= \varepsilon - \text{closure}\left(\bigcup_{i=1}^k \delta(P_i, a)\right) = \widehat{\delta}(q_0, xa)\end{aligned}$$

Quindi abbiamo dimostrato che $\forall w \in \Sigma^*, \widehat{\delta}(q_0, w) = \widehat{\Delta}(i_m, w)$

Dobbiamo dimostrare che $w \in L[N] \Leftrightarrow w \in L[M_N]$

Ci possiamo ridurre a dimostrare che:

$$\exists p \in F \text{ tale che } p \in \widehat{\delta}(q_0, w) = \widehat{\Delta}(i_m, w)$$

In quanto $p \in F$, allora anche $p \in \mathcal{F}$

Quindi $w \in L[M_N]$

Quindi $L[N] = L[M_N]$

■

4. Grammatiche regolari riassunto

domenica 23 ottobre 2022 17:45

4.1 - Da ESPRESSIONI REGOLARI a NFA equivalenti

4.1.2 - TEOREMA

Data una espressione regolare s , possiamo costruire un NFA $N[s]$ tale che:

$$\mathcal{L}[s] = L[N[s]]$$

(Ovvero gli NFA riconoscono "tutti" i linguaggi regolari!
Vedremo inoltre che gli NFA riconoscono "solo" linguaggi regolari.)

DIMOSTRAZIONE SU PAGINA 4.

Contiene come trasformare da espressione regolare in NFA

4.2 - GRAMMATICHE REGOLARI

4.2.1 - DEFINIZIONE

Una grammatica libera è regolare \Leftrightarrow ogni produzione è della forma $V \rightarrow aW$ oppure $V \rightarrow a$ (Dove $V, W \in NT$ e $a \in T$). Per il simbolo iniziale S è ammessa anche la produzione $S \rightarrow \epsilon$.

4.2.2 - DEFINIZIONE LASCA

A volte useremo la definizione più lasca che ci permette produzioni $V \rightarrow \epsilon$ anche per non terminali diversi da S , ovvero il simbolo iniziale

4.2.3 - TEOREMA

Data una grammatica regolare G si può costruire un NFA N_G equivalente.

DIMOSTRAZIONE SU PAGINA 4.

Contiene come trasformare da una grammatica in un NFA

4.3 - DA DFA A GRAMMATICHE REGOLARI

4.3.1 - TEOREMA

Da un DFA M possiamo definire una grammatica G_M tale che $L[M] = L(G_M)$

DIMOSTRAZIONE SU PAGINA 4

Contiene come trasformare da DFA a una grammatica regolare

4.4 - GRAMMATICHE REGOLARI ED ESPRESSIONI REGOLARI

4.4.1 - TEOREMA

Il linguaggio definito da una grammatica regolare G è un linguaggio regolare, cioè è possibile costruire un'espressione regolare s_g tale che:

$$L(G) = \mathcal{L}[s_g]$$

DIMOSTRAZIONE SU PAGINA 4.

Contiene come passare da grammatiche regolari ad espressioni regolari e viceversa

4.5 - RIASSUMENDO

$$\begin{array}{ccc} \text{Espressioni regolari} & \rightarrow & \text{NFA} \\ \uparrow & & \downarrow \\ \text{Grammatiche regolari} & \leftarrow & \text{DFA} \end{array}$$

Conseguenza:

Tutti questi formalismi sono equivalenti.

Tutti generano / riconoscono / descrivono la stessa classe di linguaggi, ovvero i
linguaggi regolari

4.1 Grammatiche regolari con esempi, spiegazioni e dimostrazioni

domenica 23 ottobre 2022 16:45

4.1 - Da ESPRESSIONI REGOLARI a NFA equivalenti

4.1.2 - TEOREMA

Data una espressione regolare s , possiamo costruire un NFA $N[s]$ tale che:

$$\mathcal{L}[s] = L[N[s]]$$

(Ovvero gli NFA riconoscono "tutti" i linguaggi regolari!
Vedremo inoltre che gli NFA riconoscono "solo" linguaggi regolari.)

DIMOSTRAZIONE

Procediamo per induzione sulla sintassi (astratta) della espressione regolare s .

Costruiremo $N[s]$, ovvero un possibile NFA associato alla espressione regolare s , in modo da mantenere i seguenti due invarianti (per semplificare la costruzione):

- 1) Lo stato iniziale non ha archi entranti
- 2) $N[s]$ ha un solo stato finale senza archi uscenti

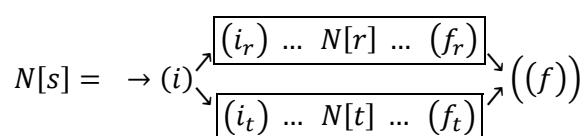
Nota sulla sintassi usata per la scrittura di questi diagrammi di transizione:

- $()$ → indica uno stato
 $\rightarrow ()$ → indica lo stato iniziale
 $(())$ → indica uno stato finale

Dentro ci possono essere dei nomi, ovvero i nomi degli stati

Esaminiamo i vari casi:

- $s = \emptyset, \quad N[s] = \rightarrow () \xrightarrow{} (())$
 - Abbiamo due stati non connessi
 - Osserviamo che $\mathcal{L}[\emptyset] = \emptyset = L[N[s]]$
- $s = \varepsilon, \quad N[s] = \rightarrow () \xrightarrow{\varepsilon} (())$
 - Passaggio etichettato ε
 - Osserviamo che $\mathcal{L}[\varepsilon] = \{\varepsilon\} = L[N[s]]$
- $s = a, \quad N[s] = \rightarrow () \xrightarrow{a} (())$
 - Passaggio etichettato a
 - Osserviamo che $\mathcal{L}[a] = \{a\} = L[N[s]]$
- $s = r|t$



- Osserviamo che

$$\mathcal{L}[r|t] = \mathcal{L}[r] \cup \mathcal{L}[t] \underset{\substack{\equiv \\ \text{per ipotesi} \\ \text{induttiva}}}{=} L[N[r]] \cup L[N[t]] = L[N[r|t]]$$

- $s = r \cdot t$

$$N[s] = \rightarrow (i_r) \dots N[r] \dots (f_r = i_t) \dots N[t] \dots ((f_t))$$

□ *Osserviamo che*

$$\mathcal{L}[r \cdot t] = \mathcal{L}[r] \cdot \mathcal{L}[t] \underset{\substack{\equiv \\ \text{per ipotesi} \\ \text{induttiva}}}{=} L[N[r]] \cdot L[N[t]] = L[N[r \cdot t]]$$

- $s = r^*$

$$N[s] = \rightarrow (i) \xrightarrow{\quad} \begin{matrix} (i_r) \dots & N[r] \dots (f_r) \\ \nwarrow & \xleftarrow{\varepsilon} & \swarrow \\ & \xrightarrow{\quad} & \end{matrix} \xrightarrow{\quad} ((f))$$

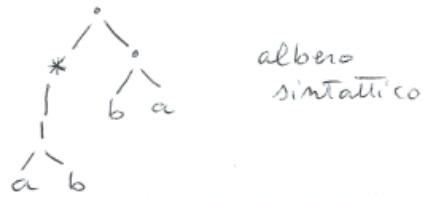
□ *Osserviamo che*

$$\mathcal{L}[r^*] = (\mathcal{L}[r])^* \underset{\substack{\equiv \\ \text{per ipotesi} \\ \text{induttiva}}}{=} (L[N[r]])^* = L[N[r^*]]$$

ESEMPIO

$$S = (a|b)^* ba$$

Partiamo dalle foglie e risaliamo alla radice (bottom-up)



albero sintattico

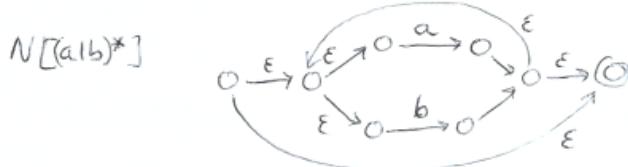
$$N[a] \quad O \xrightarrow{a} \circ$$

$$N[b] \quad O \xrightarrow{b} \circ$$

$$N[a|b] \quad O \xrightarrow{\epsilon} O \xrightarrow{a} O \xrightarrow{\epsilon} \circ$$

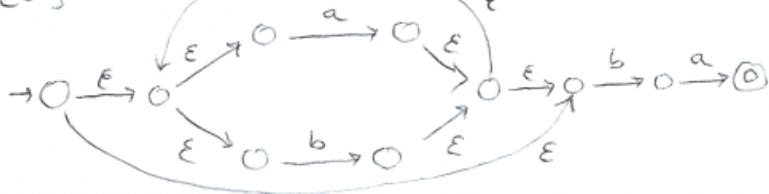
$$\quad \quad \quad \downarrow \quad \quad \quad \downarrow$$

$$\quad \quad \quad b \quad \quad \quad \epsilon$$



$$N[ba] \quad O \xrightarrow{b} O \xrightarrow{a} \circ$$

$$N[S]$$



4.2 - GRAMMATICHE REGOLARI

4.2.1 - DEFINIZIONE

Una grammatica libera è regolare \Leftrightarrow ogni produzione è della forma $V \rightarrow aW$ oppure $V \rightarrow a$ (Dove $V, W \in NT$ e $a \in T$). Per il simbolo iniziale S è ammessa anche la produzione $S \rightarrow \epsilon$.

4.2.2 - DEFINIZIONE LASCA

A volte useremo la definizione più lasca che ci permette produzioni $V \rightarrow \epsilon$ anche per non terminali diversi da S , ovvero il simbolo iniziale

4.2.3 - TEOREMA

Data una grammatica regolare G si può costruire un NFA N_G equivalente.

DIMOSTRAZIONE

Sia $G = (NT, T, R, S)$ allora $N_G = (T, Q, \delta, S, \{\epsilon\})$ è definito come segue:

- $Q = NT \cup \{\epsilon\}$
- $F = \{\epsilon\}$, $q_0 = S$
- δ è definita come:

- $Z \in \delta(V, a)$ se $V \rightarrow aZ \in R$
- $\varepsilon \in \delta(V, a)$ se $V \rightarrow a \in R$
- $\varepsilon \in \delta(S, \varepsilon)$ se $S \rightarrow \varepsilon \in R$

Si può dimostrare che $s \Rightarrow_G^* w \Leftrightarrow (s, w) \vdash_{N_G}^* (\varepsilon, \varepsilon)$
 (Non lo dimostriamo)

4.3 - DA DFA A GRAMMATICHE REGOLARI

4.3.1 - TEOREMA

Da un DFA M possiamo definire una grammatica G_M tale che $L[M] = L(G_M)$

DIMOSTRAZIONE

Sia $M = (\Sigma, Q, \delta, q_0, F)$ il DFA.

La grammatica $G_M = (Q, \Sigma, R, q_0)$ ha:

- Per non terminali gli stati di M
- Per terminali l'alfabeto di M
- Per simbolo iniziale, lo stato iniziale di M
- Per produzioni R :
 - Per ogni $\delta(q_i, a) = q_j$ la produzione $q_i \rightarrow aq_j \in R$
 Inoltre, se $q_j \in F$ anche $q_i \rightarrow a \in R$
 - Se $q_0 \in F$ allora $q_0 \rightarrow \varepsilon \in R$

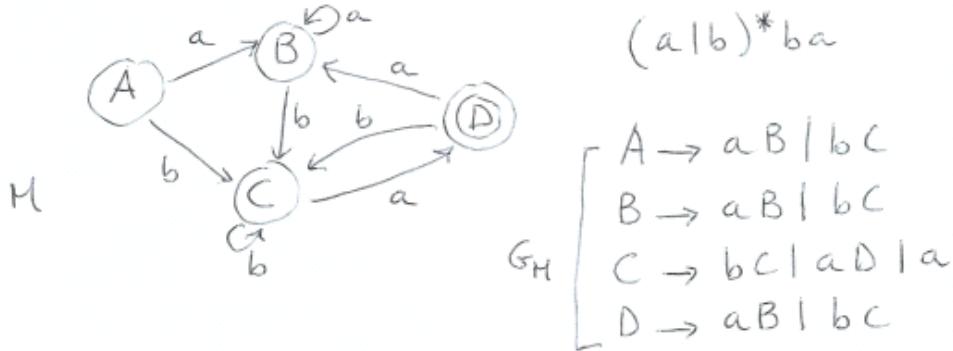
Versione alternativa che usa la definizione di grammatica regolare più lasca:

- Per produzioni R :
 - Per ogni $\delta(q_i, a) = q_j$ la produzione $q_i \rightarrow aq_j \in R$
 - Se $q \in F$ allora $q \rightarrow \varepsilon \in R$

Si può inoltre dimostrare che:

$$w \in L[M] \Leftrightarrow w \in L(G_M)$$

ESEMPIO:



Alternativamente

$$G_M = \begin{cases} A \rightarrow aB \mid bC \\ B \rightarrow aB \mid bC \\ C \rightarrow bC \mid aD \\ D \rightarrow aB \mid bC \mid \epsilon \end{cases}$$

La variante alternativa viene realizzata usando la definizione lasca.

4.4 - GRAMMATICHE REGOLARI ED ESPRESSIONI REGOLARI

4.4.1 - TEOREMA

Il linguaggio definito da una grammatica regolare G è un linguaggio regolare, cioè è possibile costruire un'espressione regolare s_g tale che:

$$L(G) = \mathcal{L}[s_g]$$

DIMOSTRAZIONE (Sketch)

Idea della prova:

Caso semplice: un solo non terminale

$$A \rightarrow aA \mid b \mid \epsilon$$

È intuitivo vedere che $a^*(b|\epsilon)$ è l'espressione regolare associata.

Caso medio: due non terminali

$$\begin{aligned} A &\rightarrow aA \mid bB \mid c \\ B &\rightarrow cA \mid aB \mid d \end{aligned}$$

Ricaviamo B dalla seconda "equazione":

$$B \simeq a^*(cA|d)$$

Ora sostituiamo B nella prima espressione regolare

$$A \simeq aA \mid ba^*(cA|d) \mid c$$

Con opportune manipolazioni su espressioni regolari possiamo riscrivere:

$$A \simeq aA \mid ba^*cA \mid ba^*d \mid c$$

E quindi

$$A \simeq (a|ba^*c)A \mid ba^*d \mid c$$

Ora che è una forma "semplice" possiamo riscriverla come:

$$A = (a|ba^*c)^*(ba^*d|c)$$

In generale

$$A_1 \approx a_{11} A_1 \mid \dots \mid a_{1m} A_m \mid b_{11} \mid \dots \mid b_{1p_1}$$

$$A_2 \approx a_{21} A_1 \mid \dots \mid a_{2n} A_n \mid b_{21} \mid \dots \mid b_{2p_2}$$

⋮

$$A_m \approx a_{m1} A_1 \mid \dots \mid a_{mn} A_n \mid b_{m1} \mid \dots \mid b_{mp_m}$$

Si parte con

$$A_m \approx s_m [A_1, \dots, A_{m-1}] \quad \text{cioè si costruisce una esp. regolare per } A_m \text{ che usa } A_1, \dots, A_{m-1}$$

Poi si procede sostituendo A_m (o meglio $s_m [A_1, \dots, A_{m-1}]$ al posto di A_m) nell'equazione per A_{m-1} , cioè

$$A_{m-1} \approx s_{m-1} [A_1, \dots, A_{m-2}]$$

e così via fino ad arrivare ad A_1 (che è il simbolo iniziale)

4.5 - RIASSUMENDO

$$\begin{array}{ccc} \text{Espressioni regolari} & \rightarrow & \text{NFA} \\ \uparrow & & \downarrow \\ \text{Grammatiche regolari} & \leftarrow & \text{DFA} \end{array}$$

Conseguenza:

Tutti questi formalismi sono equivalenti.

Tutti generano / riconoscono / descrivono la stessa classe di linguaggi, ovvero i linguaggi regolari

5. Minimizzazione

domenica 23 ottobre 2022 19:15

5.1 - Obiettivo

La scorsa volta abbiamo visto l'equivalenza tra tutti i formalismi visti fin ora (Grammatiche regolari, NFA, DFA, Espressioni regolari).

Vogliamo quindi realizzare uno scanner, il quale lo:

- Scriviamo in Exp.reg (pattern)
- Traduciamo in NFA
- Traduciamo in DFA (in modo che sia eseguibile)
 - o Tuttavia, nel caso peggiore, il DFA è esponenzialmente più grande del DFA:
 - Dobbiamo trovare un modo per minimizzare la dimensione del DFA.

Il nuovo DFA che cerchiamo:

- Occupa meno memoria (quindi ha meno stati)
- Il DFA è unico (a meno di isomorfismi)
- È equivalente al DFA da cui siamo partiti

5.2 - Introduzione di una nozione

Abbiamo precedentemente definito $\hat{\delta}$ per NFA, ora la definiamo per DFA:

Per un DFA $N = (\Sigma, Q, \delta, q_0, F)$

$\hat{\delta} : Q \times \Sigma^* \rightarrow Q$ viene definita come:

$$\begin{aligned}\hat{\delta}(q, \varepsilon) &= q \\ \hat{\delta}(q, xa) &= \delta(\hat{\delta}(q, x), a)\end{aligned}$$

E quindi

$$w \in L[N] \Leftrightarrow \hat{\delta}(q_0, w) \in F$$

5.3 - Equivalenza / Indistinguibilità

5.3.1 - DEFINIZIONE

Due stati q_1 e q_2 di un DFA N sono **equivalenti** (o indistinguibili) se:

$$\forall x \in \Sigma^* \quad \hat{\delta}(q_1, x) \in F \Leftrightarrow \hat{\delta}(q_2, x) \in F$$

Cioè se $L[N, q_1] = L[N, q_2]$

Simmetricamente, due stati q_1 e q_2 **non sono equivalenti** se:

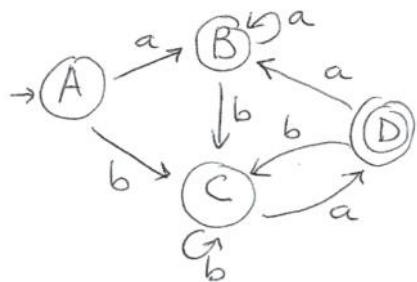
$$\begin{aligned}\exists x \in \Sigma^* \text{ tale che} \\ \hat{\delta}(q_1, x) \in F \quad \text{ma} \quad \hat{\delta}(q_2, x) \notin F \\ \text{oppure} \\ \hat{\delta}(q_1, x) \notin F \quad \text{ma} \quad \hat{\delta}(q_2, x) \in F\end{aligned}$$

q_1 e q_2 sono "distinguibili"

Strategia: cercare di distinguere due stati considerando le $x \in \Sigma^*$ a partire dalla più corta (ε)

ESEMPIO: (Importante per capire)

Sia dato il DFA:



Cerco di vedere quali coppie di stati non sono equivalenti e comincio con la stringa vuota.

Le coppie sono: (A, B) , (B, C) , (A, C) , (A, D) , (B, D) , (C, D)

a. Analizziamo il caso ε .

Notiamo che gli unici stati che con una stringa vuota diventano finali, sono gli stati finali. Quindi analizzare il caso ε coincide con il **distinguere ogni stato in F da ogni stato in $Q \setminus F$**

(A, D) (B, D) e (C, D) si possono quindi escludere

b. Consideriamo ora le stringhe di lunghezza 1, ovvero "a" e "b":

i. "a" distingue B e C perché:

$$\delta(B, a) = B \quad e \quad \delta(C, a) = D$$

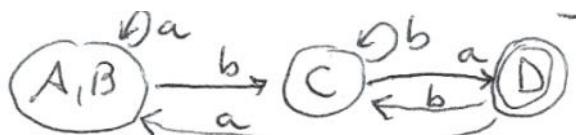
ii. "a" distingue A e C perché:

$$\delta(A, a) = B \quad e \quad \delta(C, a) = D$$

iii. "b" non mi permette di fare distinzioni e non ci sono altre distinzioni per "a".

c. Procedo ora con le stringhe lunghe 2, ma non riesco a fare nessuna ulteriore distinzione, quindi il mio "algoritmo" è finito.

Dopo questa analisi, capiamo che **A e B sono equivalenti**, in quanto la coppia A,B non è mai stata esclusa da uno dei passaggi fatti. Quindi possiamo ricreare uno stato unico chiamato A,B che ha gli stessi passaggi che avevano A,B (in quanto erano indistinguibili i passaggi coincidono) e a cui arrivano gli i passaggi sia di B che di A.



5.3.2 - DEFINIZIONE - Relazione "distinguibili"

Dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$ definiamo una famiglia di relazioni $\sim_i \subseteq Q \times Q$ nel seguente modo:

$$\sim_0 = \underbrace{F \times F}_{\begin{array}{c} \text{stati che non possono essere distinti} \\ \text{da } \varepsilon, \text{ l'unica parola di lunghezza 0} \end{array}} \cup (Q \setminus F) \times (Q \setminus F)$$

$$q_1 \sim_{i+1} q_2 \Leftrightarrow \forall a \in \Sigma \quad \delta(q_1, a) \sim_i \delta(q_2, a)$$

Che significa: q_1 e q_2 sono in relazione \sim_{i+1} se $\forall x \in \Sigma^*$ con $|x| \leq i + 1$
 $\hat{\delta}(q_1, x) \in F \Leftrightarrow \hat{\delta}(q_2, x) \in F$

Ovvero: sono in relazione i -esima se fino alle stringhe di lunghezza i -esima scritta con l'alfabeto sono equivalenti, ovvero appartengono entrambe a F se una appartiene

OSSERVAZIONI (IMPORTANTI):

- 1) La relazione $Id = \{(q, q) | q \in Q\}$ è tale che $Id \subseteq \sim_i, \forall i$
Ovvero, un nodo è sempre equivalente a se stesso (E lo denotiamo con la relazione "Id" = identità)
- 2) \sim_i è una relazione di equivalenza per ogni i
 - i. \sim_0 ha due classi di equivalenza (F e $Q \setminus F$)
 - ii. Per tutte vale riflessività, simmetria e transitività
- 3) $\sim_{i+1} \subseteq \sim_i$
Vuol dire che ad ogni passo, rimuovo qualche coppia (quindi il sottoinsieme risultante è proprio)
- 4) Se esiste k tale che $\sim_k = \sim_{k+1}$, allora $\forall j > k$ vale $\sim_j = \sim_k$
"Non appena la relazione non viene modificata in un passo, ho trovato la soluzione"
Dimostrazione con un esempio si trova a slide 6 della lezione 8.
- 5) Un tale k esiste ed è sicuramente minore di $|\sim_0| = |F|^2 + |Q \setminus F|^2$
Perché ad ogni passo rimuovo solo 1 coppia

Si può anche dimostrare che $k \leq |Q| - 1$

Ovvero che è minore della lunghezza del maggiore cammino aciclico.

5.3.3 - ALGORITMO ITERATIVO - Tabella a scala

Costruire una tabella a scala

Marcare con x_0 ogni coppia (q_1, q_2) tale che $q_1 \in F$ e $q_2 \in Q \setminus F$ (o viceversa)

$b := true; i := 1$

$while b do$ {

$b := false$

Per ogni coppia (q_1, q_2) non marcata do {

$if \exists a \in \Sigma \text{ con } (\delta(q_1, a), \delta(q_2, a)) \text{ già marcata}$

$marca (q_1, q_2) \text{ con } x_i$

$b := true$

$i++;$

}

}

Alla fine, sia J l'insieme di coppia non marcate

La relazione di equivalenza \sim è la chiusura riflessiva e simmetrica di J , cioè:

$$\sim = J \cup \{(q_1, q_2) | (q_1, q_2) \in J\} \cup \{(q, q) | q \in Q\}$$

NOTA:

Al round i , dopo averlo finito, le coppie non marcate sono \sim_i .

ESEMPI:

Vi sono degli esempi sulle slide, slide 13-15.

5.3.4 - TEOREMA

Dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$ l'algoritmo di riempimento della tabella a scala termina.
Due stati (q_1, q_2) sono distinguibili \Leftrightarrow nella tabella (q_1, q_2) e/o (q_2, q_1) sono marcate.

DIMOSTRAZIONE

Semplice sulle slide, slide 12 lezione 8

5.4 - AUTOMA MINIMO

5.4.1 DEFINIZIONE

Dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$, l'automa minimo equivalente $M_{MIN} = (\Sigma, Q_{MIN}, \delta_{MIN}, [q_0], F_{MIN})$ è dato da:

- $Q_{MIN} = \{[q] \mid q \in Q\}$ con $[q] = \{q' \in Q \mid q \sim q'\}$
(Gli stati di M_{MIN} sono le classi di equivalenza di M)
- $\delta_{MIN}([q], a) = [\delta(q, a)]$
- $F_{MIN} = \{[q] \mid q \in F\}$

OSSERVAZIONI:

Non mi sono stati distinti che sono tra loro equivalenti in M_{MIN}

5.4.2 - NOTAZIONE

Stessa definizione che abbiamo fatto per NFA e DFA ma adesso anche per DFA minimizzati.
DFA - Min:

$$\begin{aligned}\hat{\delta}_{MIN} : Q_{MIN} \times \Sigma^* &\rightarrow Q_{MIN} \\ \hat{\delta}_{MIN}([q], \varepsilon) &= [q] \\ \hat{\delta}_{MIN}([q], xa) &= \delta_{MIN}(\hat{\delta}_{MIN}([q], x), a)\end{aligned}$$

$$w \in L[M_{MIN}] \Leftrightarrow \hat{\delta}_{MIN}([q_0], w) \in F_{MIN}$$

5.4.3 - TEOREMA - Equivalenza linguaggio tra M e M_{MIN}

Dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$, l'automa $M_{MIN} = (\Sigma, Q_{MIN}, \delta_{MIN}, [q_0], F_{MIN})$ riconosce lo stesso linguaggio di M , ed ha il minor numero di stati tra tutti gli automi deterministici per questo linguaggio.

DIMOSTRAZIONE

Slides 17-19 di lezione 8.

LEX - Generatore di Scanner

lunedì 7 novembre 2022 14:21

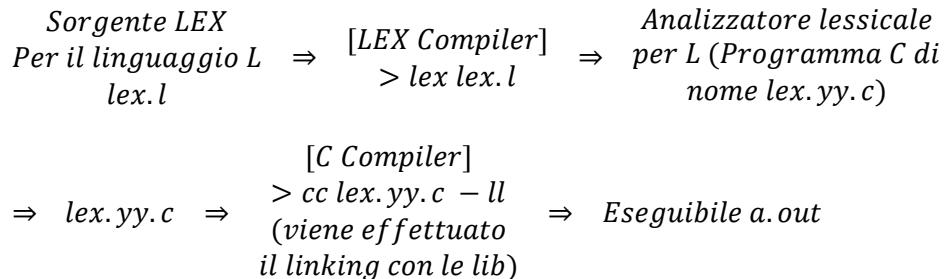
1. Cosa è LEX?

LEX è un software del 1975, FLEX è la versione più recente (del 1987). Website: flex.sourceforge.net

LEX/FLEX è un generatore di analizzatori lessicali:

- Input
 - o Un file di tipo ".l" che contiene un insieme di definizioni regolari ed una serie di assiomi corrispondenti.
 - Output
 - o Un programma C che realizza l'automa riconoscitore e che associa ad ogni istanza di una definizione la relativa azione

Procedimento:



2. Come è fatto un file di input di LEX?

Diviso in 3 parti:

- ## 1) Dichiarazioni (definizioni regolari)

Ad esempio:

- cifra $[0 - 9]$
 - cifre $[0 - 9]^+$
 - Ide $[c_1 \dots c_n]^{+}$

0/0

- 2) Regole $\underbrace{\{espressioni\ regolari\}}_{pattern}$ $\underbrace{\{azione\}}_{frammento\ di\ codice\ C}$

A esempio:

- {cifre} {printf("<NUM,%S>", yy text)}
 - {ide} {printf("<IDE,%S>", yy text)}
 - yytext : vengono sostituiti con il testo matched

%%

- ### 3) Funzioni ausiliarie

Se si usano funzioni complesse nella parte "azione" si possono definire qui.

3. Come funziona il programma *lex. yy. c*?

Il programma C ottenuto, e compilato per renderlo eseguibile in *a.out*, implementa esenzialmente il DFA riconosciutore dell'insieme delle espressioni regolari contenuto nelle "Regole".

- **Scandidce il testo sorgente alla ricerca** di una stringa che corrisponda a (cioè sia un lessema per) una delle espressioni regolari (pattern di categoria sintattica)
- **Quando riconosce un lessema, esegue l'azione** specificata, e passa in output il risultato dell'azione al posto del lessema
- **Quando l'input non corrisponde a nessuna pattern**, lo lascia inalterato e **"segnala"** la cosa al **"gestore degli errori"**

4. ESEMPI

Si possono vedere sulle slide, 3-7.

5. Utilizzo con YACC

Il programma generato da LEX non è usato da solo, ma viene chiamato come subroutine da YACC (un generatore di analizzatori sintattici).

Alcune variabili comuni permettono di scambiare dati.

lex.yy.c viene usato "on-demand" da YACC per richiedere il token successivo.

yylex() restituisce il nome del token, mentre il valore del token è condiviso nella variabile *yylval*.

ESEMPIO SLIDE 9

6. Proprietà algoritmiche dei Ling. Reg.

lunedì 7 novembre 2022 15:30

6.1 - Linguaggi liberi non esprimibili con espressioni regolari

Sappiamo che i linguaggi regolari sono un sottoinsieme dei linguaggi liberi.

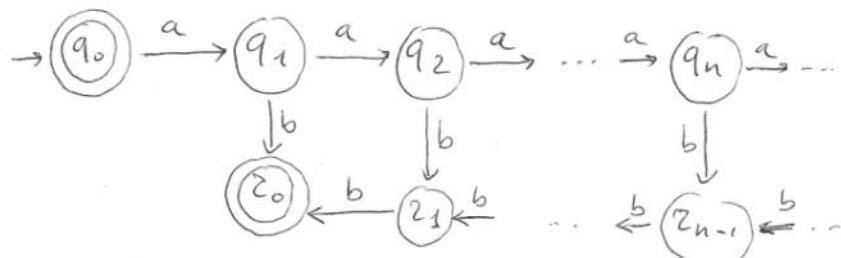
Ora mostriamo alcuni **linguaggi che non sono esprimibili attraverso una grammatica regolare**:

$$L = \{a^n b^n \mid n \geq 0\}$$

È sicuramente **libero** perché viene generato dalla grammatica libera $S \rightarrow \varepsilon \mid aSb$
Tuttavia, **possiamo dimostrare che non è regolare**.

Perché?

Intuitivamente, necessita di contare illimitatamente e per farlo mi servirebbero **infiniti** stati



Da notare che per esempio:

$$L = \{a^n b^m \mid n \geq 0 \wedge m \geq 0\}$$

È un linguaggio regolare, in quanto equivale a $a^* b^*$

Vediamo ora le proprietà algoritmiche che hanno le espressioni regolari.

6.2 - Pumping Lemma - DA SAPERE DIMOSTRARE

L è un linguaggio regolare $\Rightarrow \exists N > 0$ tale che $\forall z \in L$ con $|z| \geq N$, $\exists u, v, w$ tali che:

- $z = uvw$
- $|uv| \leq N$
- $|v| \geq 1$
- $\forall k \geq 0, \quad uv^k w \in L$

Inoltre, N è minore o uguale del numero di stati del DFA minimo che accetta L .

Informalmente,

Ci dice che tutte le stringhe sufficientemente lunghe possono essere "pompatte", ovvero possono avere una sezione in mezzo che viene ripetuta un numero arbitrario di volte, producendo una nuova stringa che anch'essa appartiene al linguaggio.

È valido anche per linguaggi finiti in quanto se noi selezioniamo N maggiore della stringa maggiore + 1, allora abbiamo un insieme vuoto per cui dobbiamo dimostrare delle proprietà, quindi $\forall z \in \{\emptyset\} \Rightarrow P(z)$, ma le premesse sono false, quindi è vero.

Quindi, linguaggi finiti sono tutti regolari. Per i linguaggi infiniti, solo se almeno una parte è esprimibile con una $*$. In particolare, se esiste una z in L tale che la cardinalità di z è maggiore di N , allora l'automa riconosce un linguaggio infinito.

DIMOSTRAZIONE

Sia $N = |Q_M|$, dove M è il DFA minimo che accetta L .

Sia $z = a_1 a_2 \dots a_m \in L$ con $m \geq N$

Quindi

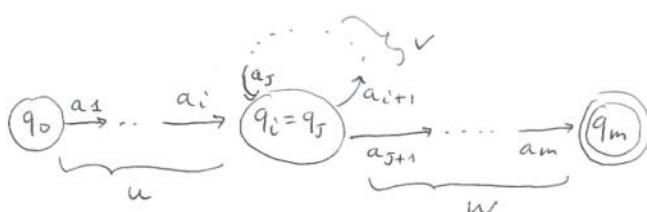
$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \dots \xrightarrow{a_m} q_m \in F$$

Ora 0-m è dato da $m + 1$ stati con $m + 1 > N$

$\Rightarrow \exists i, j : (i \neq j)$ tali che $q_i = q_j$

Essenzialmente: ci sono solo N nodi, ma io visito $N+1$ nodi, quindi ho sicuramente ripetuto un nodo. Inoltre, ho ripetuto questa visita del nodo in un momento diverso, ovvero l'ho visitato come i -esimo e j -esimo.

Dal punto di vista grafico possiamo esprimere come:



Poiché $i \neq j \Rightarrow v = a_{i+1} \dots a_j$ è tale che $|v| \geq 1$

La condizione $|uv| \leq N$ mi dice che (se m è molto grande potrebbero esserci più cicli) che prendo il primo ciclo!

Immaginiamo $a^* b^*$

Perché $\forall z \in L$, vale che noi selezioniamo $aaaaaa$, e N possiamo selezionarlo come $= 1$ e aver selezionato b come ciclo e non riusciamo a confermare la regola. Però è rispettata.

Vediamo che:

$$\left. \begin{array}{l} uv^0w = uw \in L \\ uv^1w = uvw = z \in L \\ uv^2w = uvwv \in L \\ \dots \\ uv^k w \in L \end{array} \right\} \forall k \geq 0, \quad uv^k w \in L, \quad \begin{array}{l} \text{Perchè il ciclo } v \\ \text{può essere percorso} \\ \text{un numero arbitrario} \\ \text{di volte (anche zero)} \end{array}$$

6.2.1 - Come usarlo?

Possiamo usare il pumping-lemma per dimostrare che un linguaggio non è regolare:

$\neg P \Rightarrow L$ non è regolare

Quindi assumiamo che non vale il pumping lemma dobbiamo dimostrare che non è regolare.

Ovvero:

SE

$$\forall N > 0 \exists z \in L \text{ con } |z| \geq N$$

$\forall uvw$ se

- $uvw = z$
- $|uv| \leq N$
- $|v| \geq 1$

allora $\exists k \geq 0, uv^k w \notin L$

ALLORA

L non è regolare

6.3 - ESEMPI

6.3.1 - ESEMPIO 1

Dimostriamo che $L = \{a^n b^n \mid n \geq 1\}$ non è regolare:

- Fissiamo un N generico ($\forall N > 0$)
- Scegliamo $z = a^N b^N$ ($\exists z \in L$ con $|z| \geq N$)
- Guardiamo tutte le possibili scomposizioni di z in tre sottostringhe ($\forall uvw$) tali che:
 - $z = uvw$
 - $|uv| \leq N$
 - $|v| \geq 1$

La condizione $|uv| \leq N$ impone che u e v siano fatte di sole "a", Quindi $v = a^j$ con $j \geq 1$.

- $\exists k = 2$ tale che $uv^2w = uvvw = a^{n+j}b^n \notin L$

$\Rightarrow L$ non è regolare

6.3.2 - ESEMPIO 2

Questo e altri 4 esempi sono presenti sulle slide, fino a slide 18.

6.4 - ALTRE PROPRIETA' DEI LINGUAGGI REGOLARI

La classe dei linguaggi regolari è chiusa per:

- 1) Unione
- 2) Concatenamento
- 3) Stella di Kleene
- 4) Complementazione
- 5) Intersezione

DIMOSTRAZIONE:

(1), (2) e (3) sono ovvio. Ad esempio, se L_1 e L_2 sono regolari, allora esistono S_1 e S_2 espressioni regolari tali che $L_1 = \mathcal{L}[S_1]$ e $L_2 = \mathcal{L}[S_2]$. Ma allora $L_1 \cup L_2 = \mathcal{L}[S_1 \mid S_2]$, cioè $L_1 \cup L_2$ è regolare.

(4) è vero perché, dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$ tale che $L = L[M]$ possiamo costruire il DFA $\bar{M} = (\Sigma, Q, \delta, q_0, Q \setminus F)$ tale che $\bar{L} = L[\bar{M}]$.

Infatti $w \in L[M] \Leftrightarrow w \notin L[\bar{M}]$ e quindi $L[\bar{M}] = \Sigma^* \setminus L[M]$

(5) discende dalla legge di De Morgan

$$L(M_1) \cap L(M_2) = \overline{\overline{L(M_1)}} \cup \overline{\overline{L(M_2)}}$$

La chiusura per intersezione può essere utile per dimostrare che un linguaggio non è regolare. Infatti, se $L \cap L_{reg} = L_{non-reg} \Rightarrow L$ non è regolare

Esempio:

$$L = \{w \in \{a, b\}^* \mid \text{in } w \text{ occorrono tanti "a" quanti "b"}\}$$

L è regolare? Se lo fosse, allora $L \cap a^*b^*$ dovrebbe essere regolare, ma:

$$L \cap a^*b^* = \{a^n b^n \mid n \geq 0\}, \text{ che abbiamo dimostrato non essere regolare}$$

Quindi L non è regolare.

6.5 - Modo alternativo per dimostrare la chiusura per \cap

Dati:

$$M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$$

$$M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$$

$$M_1 \cap M_2 = (Q_2 \times Q_1, \Sigma, \delta, (q_{01}, q_{02}), F_2 \times F_1)$$

$$\text{Con } \delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$$

È tale che $L[M] = L[M_1] \cap L[M_2]$

(M_1, M_2 e M sono DFA)

Ci sono poi delle proprietà sulle slide (Slide 21)

Per i linguaggi regolari, si possono decidere (cioè verificare algoritmamente) le seguenti proprietà:
 $(M, M_1, M_2 \text{ sono DFA})$

- $w \in L[M] ?$ Basta leggere w e vedere se si è finiti su uno stato finale

- $L[M] = \emptyset ?$ "Esiste un cammino ^{acciaio} dallo stato iniziale ad uno finale?"

- $L[M] = A^* ? \quad (\Leftrightarrow L[\bar{M}] = \emptyset)$

- $L[M_1] \subseteq L[M_2] ? \quad (\Leftrightarrow L[\bar{M}_2] \cap L[M_1] = \emptyset)$

- $L[M_1] = L[M_2] ? \quad (\Leftrightarrow L[M_1] \subseteq L[M_2] \wedge L[M_2] \subseteq L[M_1])$

o anche: costruisco il DFA minimo per M_1 e M_2 e
verifco se sono isomorfi

Venifco se sono isomorfi

- $L[M]$ è infinito? "Esiste una parola $z \in L[M]$
tale che $n \leq |z| \leq 2n$ dà
 $n = |Q|?$ "

Domande orale di questo capitolo

lunedì 5 giugno 2023 11:34

31. Cosa fa l'analizzatore lessicale?

L'analizzatore lessicale, anche chiamato scanner, prende in input il codice e ne restituisce l'elenco dei token. In particolare, associa ogni token ad una "categoria", e questo viene dedotto usando il pattern associato ad essa.

Altra risposta: scopo dell'analizzatore lessicale è quello di riconoscere nella stringa in ingresso alcuni gruppi di caratteri che corrispondono a certe categorie sintattiche. In tal modo la stringa in ingresso è trasformata in una sequenza di simboli astratti, detti token, che sono poi passati all'analizzatore sintattico.

32. Cos'è un token?

È un'informazione astratta che rappresenta una stringa del testo in ingresso. È una coppia (nome, valore): il nome del token è un simbolo astratto che rappresenta una categoria sintattica, il valore invece è costituito da una sequenza di simboli del testo in ingresso.

33. Cos'è un pattern? Come lo si rappresenta? Cos'è un lessema?

Un pattern è una descrizione del linguaggio al quale un token appartiene. Questa descrizione viene fatta usando le espressioni regolari. Un lessema è una particolare istanza di un pattern, ovvero una stringa appartenente al linguaggio.

34. Definizione di espressioni regolari (sintassi) e di linguaggio associato (semantica)

Le espressioni regolari si possono definire attraverso la seguente BNF:

$$r ::= \emptyset \mid \varepsilon \mid r \cdot r \mid r|r \mid r^* \mid \underset{\forall a \in A}{a}$$

Abbiamo inoltre che:

- Sono tutte associative a sinistra
- La precedenza è $*$ > \cdot > $|$

Il linguaggio associato invece ad un'espressione regolare è una funzione del tipo:

$$\mathcal{L} : \text{Exp. reg} \mapsto \mathcal{P}(A^*)$$

E funziona nel seguente modo:

$$\begin{aligned}\mathcal{L}[\emptyset] &= \emptyset \\ \mathcal{L}[\varepsilon] &= \{\varepsilon\} \\ \mathcal{L}[a] &= \{a\}, \quad \forall a \in A \\ \mathcal{L}[r \cdot r] &= \mathcal{L}[r] \cdot \mathcal{L}[r] \\ \mathcal{L}[r|r] &= \mathcal{L}[r] \cup \mathcal{L}[r] \\ \mathcal{L}[r^*] &= \bigcup_{n=0}^{\infty} \mathcal{L}[r]^n\end{aligned}$$

$$\mathcal{L}[(r)] = \mathcal{L}[r]$$

35. Quali sono i linguaggi regolari? I linguaggi finiti sono tutti regolari? Esistono linguaggi infiniti regolari?

I linguaggi regolari sono tutti quei linguaggi per cui esiste un'espressione regolare associata a tale linguaggio.

I linguaggi finiti sono regolari in quanto ci basterebbe mettere un "or" tra tutte le singole stringhe.

Esistono linguaggi infiniti regolari, questi vengono generati quando si usa la *.

36. Definizione di equivalenza tra espressioni regolari. Elencare alcune leggi di equivalenza.

Due espressioni regolari si dicono equivalenti se vengono associate allo stesso linguaggio regolare.

Alcune leggi di equivalenza potrebbero essere:

$$\begin{aligned} r|r &\cong r \\ a^*|\varepsilon &\cong a^* \\ r^{**} &\cong r^* \\ r \cdot \varepsilon &\cong \varepsilon \cdot r \cong r \end{aligned}$$

37. Cos'è una definizione regolare e a cosa serve?

È una lista di definizioni di simboli in cui, ad ogni simbolo, è associato un'espressione regolare. Praticamente noi andiamo a definire:

$$d_1, \dots, d_k \quad \text{dove} \quad d_1 = r_1, \quad \dots, \quad d_k = r_k$$

Dove r_1, \dots, r_k sono espressioni regolari e d_1, \dots, d_k sono definizioni.

All'interno delle espressioni regolari che usano queste definizioni (tra cui anche r_1, \dots, r_k) l'alfabeto diventa $A \cup \{d_1, \dots, d_k\}$ (Ovvero possiamo usare i nuovi nomi dati alle definizioni all'interno delle altre espressioni regolari come simboli).

È utile per esprimere espressioni regolari complesse, in quanto le si divide in sotto-espressioni in cui ognuna è associata ad un simbolo.

38. Definizione di NFA (automa finito non deterministico). Discutere cosa si intenda per non deterministico. Mettere in relazione la definizione formale con la rappresentazione grafica come diagramma di transizioni.

Un NFA è una quintupla:

$$(\Sigma, Q, \delta, q_0, F)$$

Dove:

- Σ è l'insieme di simboli dell'alfabeto

- Q è l'insieme di stati
- δ è una funzione di transizione, del tipo:
 - $\delta : Q \times (\Sigma \times \{\varepsilon\}) \mapsto \mathcal{P}(Q)$
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme di stati finali.

Per non deterministico si intende che con gli stessi input, l'automa può dare risultati anche diversi. Questo succedere in due casi: o ci sono più nodi per lo stesso simbolo, oppure si usa il simbolo ε per la transizione, che non consuma input. Questo vuole anche dire che non si può con certezza in modo immediato se una parola appartiene o meno in quanto si potrebbero incorrere in degli errori a volte e potrebbe andare bene in altre, quindi avviene spesso il backtracking per andare a fare tutti i tentativi possibili.

L'ultima parte della domanda è facilissima e lunghissima quindi skip.

39. Come si definisce il linguaggio accettato da un NFA? Definizione di equivalenza tra NFA.

Il linguaggio accettato dall'NFA N è:

$$L[N] = \{w \in \Sigma^* \mid (q_o, w) \vdash_N^* (q, \varepsilon) \wedge q \in F\}$$

Dove $(q_0, w) \vdash_N^* (q, \varepsilon)$ vuol dire che c'è un insieme di mosse che si possono arrivare per passare da uno stato dove si deve consumare la stringa w ad uno stato dove si è consumata tutta la stringa.

Questo viene definito come:

$$\frac{}{(q_0, w) \vdash_N^* (q_0, w)}, \quad \frac{(q, w) \vdash_N^* (q', w') \quad (q', w') \vdash_N (q'', w'')}{(q, w) \vdash_N^* (q'', w'')}$$

Dove la \vdash_N senza asterisco è definita come la mossa ed è:

$$\frac{q' \in \delta(q, a)}{(q, aw) \vdash_N (q', w)}$$

Due NFA sono equivalenti se entrambi accettano lo stesso linguaggio, ovvero:

Siano N, N' due NFA, allora N equivalente $N' \Leftrightarrow L[N] = L[N']$

40. Definizione di DFA (automa finito deterministico). Discutere cosa si intende per deterministico. Mostrare che un DFA è un caso speciale di NFA, ovvero la classe dei DFA è un sottoinsieme della classe degli NFA.

Un DFA è una quintupla definita come segue:

$$(\Sigma, Q, \Delta, q_0, \mathfrak{F})$$

Dove:

- Σ è l'insieme dei simboli dell'alfabeto
- Q è l'insieme degli stati
- Δ è una funzione del tipo:

- $\Delta : T \times \Sigma \mapsto T$
- q_0 è lo stato iniziale
- F è l'insieme degli stati finali

Quindi è identico ad un automa non deterministico se non per la sua funzione di transizione Δ .

È deterministico in quanto per ogni singolo simbolo in Σ esiste una sola mossa possibile. Questo vuol dire che, dato lo stesso input, si avrà sempre lo stesso risultato, ovvero riconosciuto oppure non riconosciuto, per ogni automa DFA e per ogni input.

Un DFA è un caso speciale degli NFA in quanto non vi sono archi etichettati con ϵ e inoltre nelle funzioni di transizione si ha che $\Delta(q, a)$ è sempre esattamente costituita da un solo stato. Gli NFA hanno quindi più libertà nella loro scrittura dei DFA, quindi i DFA sono un caso speciale.

- 41. Dato un NFA, come si ricava un equivalente DFA? Ovvero descrivere come è definita la costruzione per sottoinsiemi. Definizione di epsilon-closure e algoritmo associato. Qual è la complessità della costruzione per sottoinsiemi nel caso pessimo? Ovvero se NFA ha n stati, quanti stati può avere il DFA equivalente?**

A partire da un NFA, per trovare un DFA si usa la **costruzione per sottoinsiemi**, la quale internamente fa uso della **epsilon-closure**. Quindi definiamo questi due algoritmi:

- Epsilon-closure di q :

```

Impostare  $\epsilon - closure(q) = \{q\}$ 
Impostare  $T = \{q\}$ 
while  $T \neq \emptyset$  do:
    get  $x = pop(T)$ 
     $\forall z \in \delta(x, \epsilon) \notin \epsilon - closure(q)$ 
        add  $z$  in  $\epsilon - closure(q)$ 
        add  $z$  in  $T$ 

```

- Costruzione per sottoinsiemi: sia N un NFA definito da $(\Sigma, Q, \delta, q_0, F)$

```

Impostare  $A = \epsilon - closure(q_0)$ 
Impostare  $T = \{A\}$ 
while  $X \in T$  non marcato
    marca  $X$ 
     $\forall \sigma \in \Sigma$  do
        if  $\delta(X, \sigma) \notin T$ 
            sia  $Y = \epsilon - closure(\text{mossa}(X, \sigma))$ 
            add  $Y$  in  $T$ 
        Definisci  $\Delta(X, \sigma) = Y$ 

```

Definiamo matematicamente l'epsilon closure:

$$\frac{}{q \in \epsilon - closure(q)} \quad \frac{z \in \epsilon - closure(q)}{\delta(z, \epsilon) \subseteq \epsilon - closure(q)}$$

La mossa invece è:

$$mossa : \mathcal{P}(Q) \times \Sigma \mapsto \mathcal{P}(Q)$$

$$mossa(P, a) = \bigcup_{\forall p \in P} \delta(p, a)$$

Nel caso pessimo, se un NFA ha n stati, allora il DFA creato usando la costruzione per sottoinsiemi ha 2^n stati, quindi è esponenziale.

- 42. Dato i due punti precedenti, enunciare il teorema che dice che la classe dei linguaggi riconosciuti da NFA coincide con la classe dei linguaggi riconosciuti da DFA.**

Sia N un NFA e sia M un DFA costruito partendo da N usando la costruzione per sottoinsiemi. Allora:

$$L[N] = L[M]$$

Per il corollario si ha che la classe dei linguaggi riconosciuti dagli NFA è uguale a quella riconosciuta dai DFA.

- 43. Come si costruisce un NFA a partire da una espressione regolare, in modo che il linguaggio riconosciuto dall'NFA sia lo stesso del linguaggio associato all'espressione regolare?**

Si costruisce nel seguente modo:

(Si vanno a definire dei modi per costruire un NFA per ogni singola operazione della grammatica regolare. In seguito, si va a dimostrare che il linguaggio è equivalente per questi, allora induttivamente è equivalente per qualsiasi trasformazione che li usa).

Questi mattoncini sono:

$$\bullet \quad S = \emptyset \quad N[S] = \rightarrow O \quad \textcircled{O}$$

Osserva che $L[\emptyset] = \emptyset = L[N[\emptyset]]$

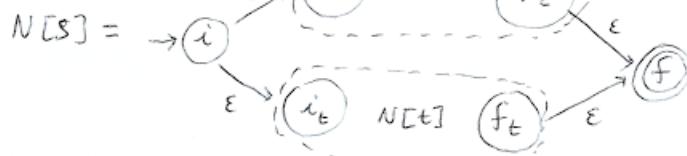
$$\bullet \quad S = \epsilon \quad N[S] = \rightarrow O \xrightarrow{\epsilon} \textcircled{O}$$

Osserva che $L[\epsilon] = \{\epsilon\} = L[N[\epsilon]]$

$$\cdot S = a \quad N[S] = \rightarrow O \xrightarrow{a} O \quad (32)$$

Osserva che $L[a] = \{a\} = L[N[S]]$

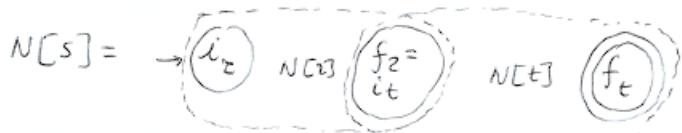
$$\cdot S = r | t$$



Osserva che

$$L[r|t] = L[r] \cup L[t] \stackrel{\text{per ipotesi induttiva}}{=} L[N[r]] \cup L[N[t]] = L[N[r|t]]$$

$$\cdot S = r \cdot t$$



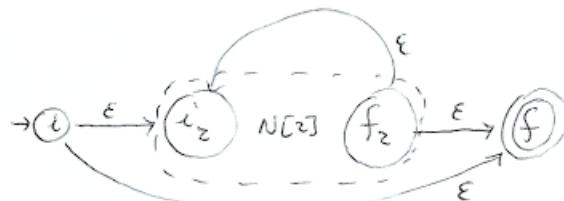
Abbiamo fatto insieme il finale di $N[r]$ con l'iniziale di $N[t]$

Osserva che

$$L[r \cdot t] = L[r] \cdot L[t] \stackrel{\text{per ipotesi induttiva}}{=} L[N[r]] \cdot L[N[t]] = L[N[r \cdot t]]$$

$$\cdot S = r^*$$

$$N[S] =$$



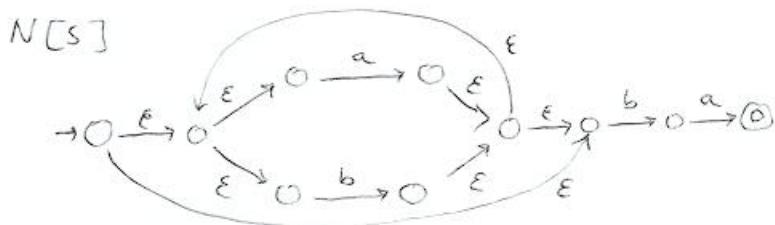
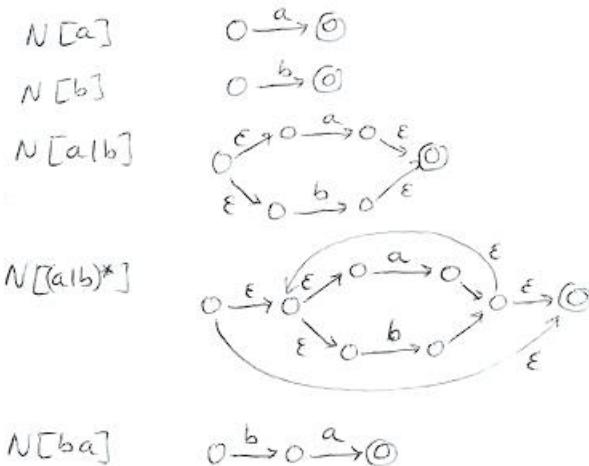
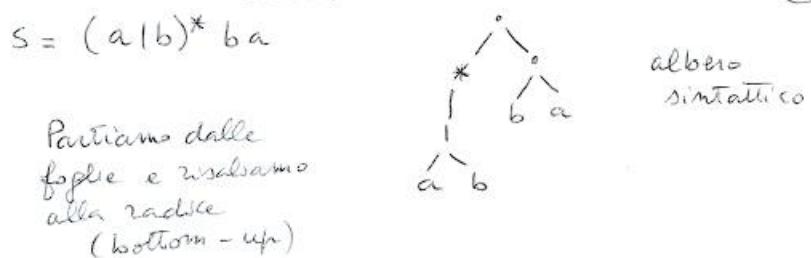
Osserva che

$$L[r^*] = (L[r])^* = (L[N[r]])^* = L[N[r^*]]$$

per ipotesi induttiva

c.v.d

Una volta che si hanno questi, di un'espressione regolare si crea l'albero di derivazione e si parte dalle foglie, come nell'esempio successivo:



44. Definizione di grammatica regolare:

Una grammatica regolare è una grammatica libera da contesto, quindi una quadrupla del tipo (NT, T, R, S) dove ogni produzione in R è del tipo:

$$A \rightarrow aX, \quad \forall a \in T, \forall A, X \in NT$$

oppure

$$A \rightarrow a, \quad \forall a \in T, \forall A \in NT$$

Inoltre, può esistere la produzione ϵ solo da S , quindi:

$$S \rightarrow \epsilon \quad \text{è ammessa.}$$

Per la definizione più lasca, la ϵ – *production* può essere presente in qualsiasi non terminale.

45. Come si associa ad una grammatica regolare un equivalente NFA?

Si deve prima costruire l'espressione regolare e poi si va a costruire un NFA usando ciò che vi è scritto sopra.

Si possono anche associare ai non terminali gli stati e far sì che le transizioni siano la lettera nella produzione.

Per le produzioni del tipo $A \rightarrow a$, la transizione $\delta(A, a) = X \in F$
Se vi è la produzione ε allora S è uno stato finale.

46. Dato un DFA, come si costruire una grammatica regolare equivalente?

Dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$, si costruire la grammatica $G_M = (Q, \Sigma, R, q_0)$ nel seguente modo:

- Per nonterminali gli stati di M
- Per terminali, l'alfabeto di M
- Per simbolo iniziale, lo stato iniziale di M
- Per le produzioni R :
 - $\forall \delta(q_i, a) = q_j$, la produzione $q_i \rightarrow aq_j \in R$
 - se $\delta(q_i, a) = q_j \in F$, allora $q_i \rightarrow a \in R$
 - se $q_0 \in F$, allora $q_0 \rightarrow \varepsilon \in R$

Si può anche costruire in modo più lasco dicendo anche che:

- $\forall \delta(q_i, a) = q_j$ la produzione $q_i \rightarrow aq_j \in R$
- se $q \in F$, allora $q \rightarrow \varepsilon \in R$

47. Data una grammatica regolare, come si costruire un'espressione regolare equivalente?

In generale, sia data la grammatica regolare del tipo:

$$\begin{aligned} A_1 &= a_{1,1}A_1 \mid \dots \mid a_{1,n}A_n \mid b_{1,1} \mid \dots \mid b_{1,p_1} \\ A_2 &= a_{2,1}A_1 \mid \dots \mid a_{2,n}A_n \mid b_{2,1} \mid \dots \mid b_{2,p_2} \\ &\dots \\ A_n &= a_{n,1}A_1 \mid \dots \mid a_{n,n}A_n \mid b_{n,1} \mid \dots \mid b_{n,p_n} \end{aligned}$$

Si parte con creare:

$$A_n = S_n[A_1, \dots, A_{n-1}]$$

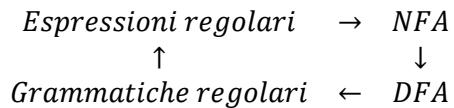
Ovvero si genera l'espressione regolare contenente i nonterminali fino ad A_{n-1} e la si sostituisce alla definizione di A_n .

Poi si fa la stessa cosa per A_{n-1}

$$A_{n-1} = S_n[A_1, \dots, A_{n-2}]$$

E così via, fino ad arrivare ad A_1 . A questo punto, usando equivalenze, ci si può ridurre ad una espressione regolare.

48. Descrivere, con un diagramma riassuntivo, tutte le relazioni fra i formalismi introdotti: NFA, DFA, grammatiche regolari, espressioni regolari. Questo diagramma dimostra che tutti questi formalismi sono equivalenti e descrivono la classe dei linguaggi regolari.



Conseguenza:

Tutti questi formalismi sono equivalenti.
 Tutti generano / riconoscono / descrivono la stessa classe di linguaggi, ovvero i **linguaggi regolari**
 $\xrightarrow{\text{gram.reg}}$ $\xrightarrow{\text{NFA/DFA}}$ $\xrightarrow{\text{exp.reg}}$

Queste cose vengono dimostrate perché:

- Da 1 a 2 abbiamo una costruzione (mattoncini che poi si uniscono a partire dalle foglie dell'albero di derivazione)
- Da 2 a 3 abbiamo una costruzione (costruzione per sottoinsiemi)
- Da 3 a 4 abbiamo una costruzione (una per definizione rigorosa e una per definizione lasca)
- Da 4 a 1 abbiamo una costruzione (sostituzione e incrociare le dita)

49. Definizione di stati equivalenti in un DFA. Quando due stati di un DFA sono distinguibili?

Due stati $q_1, q_2 \in Q$ di un DFA sono equivalenti se e solo, $\forall x \in \Sigma^*, \hat{\delta}(q_1, x) \in F \Leftrightarrow \hat{\delta}(q_2, x) \in F$. Si può anche scrivere come: $L[N, q_1] = L[N, q_2]$, dove N è il DFA al quale q_1 e q_2 appartengono.

Due stati $q_1, q_2 \in Q$ sono distinguibili se $\exists x \in \Sigma^*$ tale che:

$$\hat{\delta}(q_1, x) \in F \wedge \hat{\delta}(q_2, x) \notin F$$

Oppure

$$\hat{\delta}(q_1, x) \notin F \wedge \hat{\delta}(q_2, x) \in F$$

50. Come funziona l'algoritmo iterativo con tabella a scala per produrre le classi di equivalenza di stati di un DFA?

Definiamo la relazione di equivalenza fino ad un stringa di lunghezza i come:

$$\sim_0 \subseteq F \times F \cup \{Q \setminus F \times Q \setminus F\}$$

$$q_1 \sim_i q_2 \Leftrightarrow \delta(q_1, a) \sim_{i-1} \delta(q_2, a), \quad \forall a \in \Sigma$$

L'algoritmo iterativo con tabella a scala funziona nel seguente modo:

```

Costruire una tabella a scala
Marcare con  $x_0$  tutte le coppie che appartengono a  $\sim_0$ 
 $b = true$ 
for  $i > 0$  e  $b \neq false$ :
   $b = false$ 
   $\forall a \in \Sigma$  do
     $\forall (q, q') \text{ non marcate, if } (\delta(q, a), \delta(q', a)) \text{ sono segnati con } x_i$ 
  
```

```

    segna  $(\delta(q, a), \delta(q', a))$  con  $x_{i+1}$ 
    b = true
i ++

```

51. Una volta determinate le classi di equivalenza degli stati di un DFA, come si costruisce l'automa minimo associato?

Dopo che abbiamo determinato le classi minime di un DFA $M = (\Sigma, Q, \delta, q_0, F)$, il DFA minimo associato, sia esso $M_m = (\Sigma, Q_m, \delta_m, [q_0], F_m)$ dove:

$$\begin{aligned}Q_m &= \{[q] \mid q \in Q\} \text{ con } [q] = \{q' \in Q \mid q \sim q'\} \\ \delta_m([q], a) &= [\delta(q, a)] \\ F_m &= \{[q] \mid q \in F\}\end{aligned}$$

52. Cos'è Lex? Qual è il suo input e il suo output?

Lex è un generatore di scanner, ovvero di analizzatori lessicali.

Richiede in input un file ".l" che è formato da un insieme di definizioni regolari e da un insieme di azioni da eseguire.

L'output è un file ".c" che si può compilare e, una volta compilato, si può usare per scannare un file. Esso è composto da automi riconoscitori delle definizioni regolari inserite e che esegue le azioni elencate

53. Qual è la struttura di un file ".l"? Come funziona l'analizzatore lessicale prodotto da Lex?

È formato da 3 parti:

(1) Dichiarazioni (ovvero definizioni regolari) della forma:

cirfra [0 – 9]
cifre *cifra* +
ide [a – zA – Z]([a – zA – Z]|[0 – 9])*

%% (*separatore tra dichiarazioni e azioni*)

(2) Azioni, del tipo: {exp. regolare} {azione}

{*cifre*} {printf("<Num, %s>", *yytext*)}
Dove yytext viene sostituito con il valore

%%

(3) Funzioni ausiliare

Nel caso quelle inserite nelle azioni fossero molto complicate e si volessero definire in modo più chiaro.

Il programma C ottenuto, e compilato per renderlo eseguibile, implementa essenzialmente il DFA riconoscitore dell'insieme delle espressioni regolari contenute nell'oggetto:

- Scandidisce il testo sorgente alla ricerca di una stringa che corrisponda a (cioè sia un lessema

- per) una delle espressioni regolari (pattern di categoria sintattica)
- Quando riconosce un lessema, esegue l'azione specificata, e passa in output il risultato dell'azione al posto del lessema
 - Quando l'input non corrisponde a nessun pattern, lo lascia inalterato e "segnala" la cosa al "gestore degli errori".

54. Come si interfaccia Lex con Yacc?

Il programma generato da LEX non è usato da solo, ma viene chiamato come subroutine da YACC (un generatore di analizzatori sintattici).

Alcune variabili comuni permettono di scambiare dati.

lex.yy.c viene usato "on-demand" da YACC per richiedere il token successivo.

yylex() restituisce il nome del token, mentre il valore del token è condiviso nella variabile *yylval*.

55. Intestazione e dimostrazione del pumping lemma.

L regolare $\Rightarrow \exists N \in \mathbb{N} : \forall w \in L : |w| \geq N \Rightarrow$

$\exists u, v, z$ tali che $w = uvz$

$|v| \geq 1$

$|uv| \leq N$

$\forall k \in \mathbb{N}, \quad uv^k z \in L$

Dimostrazione

Se si considera una grammatica regolare L , allora si può considerare un DFA N minimizzato associato ad essa. Siano il numero di stati di questo DFA $= N$.

Allora prendiamo una stringa appartenente a L lunga N , essa percorre $N + 1$ stati.

Sicuramente, pertanto, ha ripercorso uno degli stati in un ciclo. Questo ciclo, pertanto, può essere sia saltato ($k=0$), sia ripetuto n volte ($k \in \mathbb{N}$) e la parola verrebbe ugualmente riconosciuta.

Notiamo che questa parte ripetuta è almeno lunga 1, in quanto si trova effettivamente nella stringa.

Inoltre notiamo che si troverà sicuramente prima di N o ad N , non si potrà trovare dopo N , in quanto l'automa ha N stati e si fanno $N+1$ transizioni e si ha un loop prima della fine (o alla fine, ma non al di fuori). Quindi come regola può essere considerata valida.

Abbiamo dimostrato che quindi possiamo cambiare il numero di k volte in cui si ripete il ciclo a nostro piacimento

Quindi, in quanto L è regolare, abbiamo determinato N , scelto una stringa w che sia maggiore o uguale in lunghezza di N , e abbiamo dimostrato che tutte le proprietà valgono.

Scritta in modo migliore:

Ipotesi:

$M = \text{DFA minimo}$ che accetta L

$N = \text{Numero di stati di } M = |Q_M|$

$z = \text{Stringa appartenente al linguaggio } L$

$m = \text{Lunghezza della stringa, } m \geq N$

Dimostrazione

Prendiamo l'elenco dei $m+1$ stati del percorso che riconoscono la stringa z (stato iniziale + ogni stato per ogni carattere della stringa)

Prendiamo gli $N+1$ stati di questo elenco.

Visto che $m + 1 \geq N + 1$, allora si passerà sicuramente più volte nello stesso stato del percorso.

Quindi ad un certo punto c'è un loop, dove $q_i = q_j$. Quindi diciamo che gli u sono gli stati fino al loop, poi ci sono gli stati di v nel loop, ed infine c'è w .

Di sicuro uv è minore di N perché i e j sono stati scelti tra i primi $N+1$ stati (con cui attenzione si consumano N archi/simboli). Visto che c'è un loop allora $i \neq j$, allora $v = a_{i+1} \dots a_j$ e quindi $|v| \geq 1$, in quanto nel caso pessimo $i + 1 = j$, quindi ha almeno lunghezza 1 il loop, e negli altri casi ha una lunghezza maggiore

La stringa può essere pompata perché possiamo eseguire il ciclo quante volte ci pare tanto l'automa è deterministico e finirà sempre in uno stato finale quindi riconoscerà la stringa.

56. Come si può utilizzare il pumping lemma (a rovescio) per dimostrare che un linguaggio non è regolare?

Si può usare il pumping lemma a rovescio nel seguente modo:

$$\neg P \Rightarrow \neg L \text{ regolare}$$

Quindi, si fa la negazione del pumping lemma per dimostrare che il risultante non è regolare.

$$\forall N > 0, \exists z \in L, |z| \geq N \Rightarrow$$

$$\begin{aligned} \forall uvw \text{ se } z &= uvw \\ |uv| &\leq N \\ |v| &\geq 1 \\ \text{allora} \\ \exists k \geq 0, \quad &uv^k w \notin L \end{aligned}$$

$$\Rightarrow L \text{ non è regolare}$$

Esempio di utilizzo:

$$\text{Sia } L = \{a^n b^n \mid n \geq 0\}$$

Fissiamo un N generico ($\forall N > 0$)

$$\text{Scegliamo } z = a^N b^N (\exists z \in L \text{ con } |z| \geq N)$$

Vediamo tutte le sottostringhe possibili tali che valga:

- $|v| \geq 1$
- $|uv| \leq N$
- $z = uvw$

Notiamo che possiamo scegliere uv solo come combinazione di a, per rispettare la seconda premessa. Quindi, abbiamo che v in particolare sarà a^j con $j \geq 1$.

A questo punto noi possiamo pompare una volta questa v, per ottenere:

$$uvvw = a^{N+j} b^N \notin L$$

Quindi abbiamo dimostrato che L non è un linguaggio regolare.

57. Quali sono le proprietà di chiusura dei linguaggi regolari? Quali proprietà si possono decidere (ovvero verificare algoritmicamente)?

La classe dei linguaggi regolari è chiusa per:

1. Unione
2. Concatenamento
3. Stella di Kleene
4. Complementazione
5. Intersezione

DIMOSTRAZIONE:

(1), (2) e (3) sono ovvio. Ad esempio, se L_1 e L_2 sono regolari, allora esistono S_1 e S_2 espressioni regolari tali che $L_1 = \mathcal{L}[S_1]$ e $L_2 = \mathcal{L}[S_2]$. Ma allora $L_1 \cup L_2 = \mathcal{L}[S_1 | S_2]$, cioè $L_1 \cup L_2$ è regolare.

(4) è vero perché, dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$ tale che $L = L[M]$ possiamo costruire il DFA $\bar{M} = (\Sigma, Q, \delta, q_0, Q \setminus F)$ tale che $\bar{L} = L[\bar{M}]$.

Infatti $w \in L[M] \Leftrightarrow w \notin L[\bar{M}]$ e quindi $L[\bar{M}] = \Sigma^* \setminus L[M]$

(5) discende dalla legge di De Morgan

$$L(M_1) \cap L(M_2) = \overline{L(M_1)} \cup \overline{L(M_2)}$$

Le proprietà che si possono verificare algoritmicamente sono:

- Per i linguaggi regolari, si possono decidere (cioè verificare algoritmicamente) le seguenti proprietà:
(M, M_1, M_2 sono DFA)
- $w \in L[M] ?$ Basta leggere w e vedere se si è finiti su uno stato finale
 - $L[M] = \emptyset ?$ "Esiste un cammino ^{acciaio} dallo stato iniziale ad uno finale?"
 - $L[M] = A^* ?$ ($\Leftrightarrow L[\bar{M}] = \emptyset$)
 - $L[M_1] \subseteq L[M_2] ?$ ($\Leftrightarrow L[\bar{M}_2] \cap L[M_1] = \emptyset$)
 - $L[M_1] = L[M_2] ?$ ($\Leftrightarrow L[M_1] \subseteq L[M_2] \wedge L[M_2] \subseteq L[M_1]$)
o anche: costruisco il DFA minimo per M_1 e M_2 e
verifico se sono isomorfi
 - $L[M]$ è infinito? "Esiste una parola $z \in L[M]$
tale che $n \leq z \leq 2n$ dare
 $m = |Q| ?$ "

Introduzione

lunedì 7 novembre 2022 19:10

1. Analisi sintattica: Linguaggi liberi

lunedì 7 novembre 2022 19:11

1.1 - Introduzione

Per l'analisi sintattica, fino adesso, abbiamo osservato solo **linguaggi regolari** (NFA/DFA/Reg.Exp). Ora, invece, iniziamo a vedere una classe di linguaggi più generale, ovvero i **linguaggi libri da contesto**:

$$L_{\text{Liberi}} \supset L_{\text{Reg}}$$

La differenza è:

<i>Regolari</i>	<i>Libere</i>
$V \rightarrow aV$	
$V \rightarrow aW$	
$W \rightarrow \epsilon$	
$V, W \in NT$	$V \rightarrow \alpha$
$a \in T$	$\alpha \in (T \cup NT)^*$

Per poter interpretare le **grammatiche libere** abbiamo bisogno di **automi a pila (PDA)**, invece di automi finiti (NFA/DFA):

- PDA
 - o Non-deterministico \approx Linguaggi libri
 - o Deterministico
 - Utile per costruire compilatori
 - Possono essere di diversi tipi, lo vedremo più avanti

Utile il ripasso di:

- Grammatica libera $G = (NT, T, R, S)$
- Se G libera, allora $L(G)$ libera
- La relazione \Rightarrow (di derivazione in un passo)
- La relazione \Rightarrow^* (di derivazione in uno o più passi)

1.1.2 - DEFINIZIONE

Derivazione canonica sinistra (leftmost):

Derivazione in cui, a partire da S , viene **sempre** riscritto il nonterminale più a **sinistra**.

Derivazione canonica destra (rightmost):

Derivazione in cui, a partire da S , viene **sempre** riscritto il nonterminale più a **destra**.

Osservazione:

Esiste una corrispondenza biunivoca tra derivazione leftmost(/rightmost) e alberi di derivazione.

2. PDA

giovedì 10 novembre 2022 18:51

2.1 - Il "senso" dei PDA

Immaginiamo di dover far sì che un NFA / DFA riesca a **riconoscere un linguaggio libero**. Esempio di linguaggio libero:

$$L = \{ww^R \mid w \in \{a, b\}^*\}$$

Dove w^R indica il "reverse" di w , ovvero se $w = ab$ allora $w^R = ba$

L è libero perché $L = L(G)$ con $G = \{ S \rightarrow \varepsilon \mid aSa \mid bSb \}$

Come facciamo a far sì che un automa riesca a riconoscere **se una stringa appartiene a questo linguaggio?**

Deve avere una certa memoria, ovvero deve ricordare quello che è successo in precedenza ed usarlo per capire la correttezza di quello che legge dopo.

Immaginiamo che a metà stringa decida di cambiare e iniziare a fare controlli usando la sua memoria: se trova la stessa stringa che ha letto fino a metà, ma al contrario, allora è una parola che appartiene al linguaggio.

Ma come capisce quando è arrivata a metà? Per un automa, questa cosa è non deterministica: succede se deve succedere, essenzialmente.

Meglio: se **può** succedere, allora è riconosciuta. Se in nessun modo può succedere che si arrivi ad uno stato terminale (/ automa sia vuoto) allora non è riconosciuto.

Quindi: notiamo che noi dobbiamo accumulare (parte dell')output.

Questo lo faremo con una memoria ausiliaria che è **una pila**.

- La pila può crescere **senza limite**. (Memoria ausiliaria illimitata, perché può essere arbitrariamente lunga una stringa da riconoscere)
- Può leggere solo l'elemento top della pila
- Può rimuovere solo l'elemento top della pila
- Può inserire un nuovo elemento solo in testa

2.2 - DEFINIZIONE PDA

Un **automa a pila nondeterministico (PDA)** è una 7-pla $(\Sigma, Q, \Gamma, \delta, q_0, \perp, F)$ dove:

○ Σ

Alfabeto finito (simboli in input)

○ Q

Insieme finito di stati

○ Γ

Insieme finito di simboli della pila

○ δ

Funzione di transizione con tipo:

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

- q_0
Stato iniziale

- $\perp \in \Gamma$
Il simbolo iniziale sulla pila

- $F \subseteq Q$
Insieme degli stati finali

2.2.1 - Funzione di transizione

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \mathcal{P}_{fin} \left(Q \times \Gamma^* \text{ sulla pila scrive una stringa di lunghezza qualsiasi (anche } \varepsilon \text{) di simboli in } \Gamma \right)$$

consuma un simbolo dell'input (Σ) oppure no (ε) consuma il simbolo top della pila

Dato uno stato q , un simbolo in input (oppure no) e un simbolo top della pila

Ti porta a (un insieme di possibili)

Stati e scrivere qualcosa in cima alla pila.

È non deterministica, perché uno stato può portare a più di uno stato/cose scritte in cima alla pila.
In matematiche:

$$|\delta(q, \sigma, A)| \text{ può essere } > 1 \\ (\forall q \in Q, \forall \sigma \in \Sigma \cup \{\varepsilon\}, A \in \Gamma)$$

Ma anche se richiedessimo che sia ≤ 1 , comunque c'è nondeterminismo con ε .

2.3 - Transizioni di un PDA

- Descrizione istantanea (o configurazione):

(q, w, β)

- $q \in Q$ (stato corrente)
 - $w \in \Sigma^*$ (input ancora non letto)
 - $\beta \in \Gamma^*$ (stringa sulla pila)
- (Per convenzione il top è il "simbolo più a sinistra")

- Mossa

$$1) \quad \frac{(q', \alpha) \in \delta(q, a, x) \quad a \in \Sigma}{(q, aw, x\beta) \vdash_N (q', w, \alpha\beta)}$$

Per portare un cambiamento da uno stato istantaneo ad un altro, dobbiamo poter consumare l'input e trasformare la pila in modo che sia "compatibile"

$$1) \quad \frac{(q', \alpha) \in \delta(q, \varepsilon, x)}{(q, w, x\beta) \vdash_N (q', w, \alpha\beta)}$$

Per portare un cambiamento da uno stato istantaneo ad un altro, senza consumare input, dobbiamo trovare che questo cambiamento appartiene alle ε – move dello stato corrente.

- Computazione / cammino

(Chiusura riflessiva e transitiva di \vdash_N)

$$\frac{}{(q, w, \beta) \vdash_N^* (q, w, \beta)}, \quad \frac{(q, w, b) \vdash_N^* (q', w', \beta') \quad (q', w', \beta') \vdash_N^* (q'', w'', \beta'')}{(q, w, \beta) \vdash_N^* (q'', w'', \beta'')}$$

2.4 - Linguaggi accettati

Ci sono due modalità di riconoscimento:

Consideriamo $N = (\Sigma, Q, \Gamma, \delta, q_0, \perp, F)$

1) Per stato finale

$$L[N] = \{w \in \Sigma^* \mid (q_0, w, \perp) \vdash_N^* (q, \varepsilon, \alpha) \text{ con } q \in F\}$$

1) Per pila vuota

$$P[N] = \{w \in \Sigma^* \mid (q_0, w, \perp) \vdash_N^* (q, \varepsilon, \varepsilon)\}$$

OSSERVAZIONE:

Per un dato PDA N, spesso

$$L[N] \neq P[N]$$

2.5 - Classe dei linguaggi riconosciuti

La classe dei linguaggi riconosciuti per pila vuota o per stato finale **non cambia**.

TEOREMA

- i. Se $L = P[N]$, possiamo costruire N' tale che $L = L[N']$
- ii. Se $L = L[N]$, possiamo costruire N' tale che $L = P[N']$

DIMOSTRAZIONE

Slide 13 Lezione 10.

Praticamente, per (i) puoi portare tutti gli stati che hanno una pila vuota ad uno stato terminale con una mossa ε .

Per (ii) puoi portare tutti gli stati terminali ad uno stato che consuma tutta la pila.

2.6 - TEOREMA

Un linguaggio L è libero da contesto \Leftrightarrow è accettato da un PDA

DIMOSTRAZIONE

$\Rightarrow)$ Se L è libero, allora $\exists PDA\ N$ tale che $L = P[N]$

Se L è libero, allora $\exists G = (NT, T, R, S)$ libera tale che $L = L(G)$.

Costruiamo quindi PDA N:

$$(T, \{q\}, T \cup NT, \delta, q, S, \emptyset)$$

Dove la funzione di transizione δ simula la costruzione di una derivazione canonica sinistra (riscriviamo sempre il nonterminale sul top, che è quello più a sinistra nella derivazione leftmost):

$$\begin{aligned}\delta(q, \varepsilon, A) &= \{(q, \beta) \mid A \rightarrow \beta \in\}, & \forall A \in NT \\ \delta(q, a, a) &= \{(q, \varepsilon)\}, & \forall A \in T\end{aligned}$$

A parole:

Ogni volta che N ha un nonterminale A in cima alla pila, sceglie non deterministicamente una produzione per A, senza consumare l'input. Se invece sulla pila c'è un terminale a e se l'input presenta proprio a , questi vengono consumati. Se invece l'input è $b \neq a$, ci si blocca, e si fa backtracking provando con un'altra produzione.

Si può dimostrare, per induzione sulla lunghezza di w, che:

$$S \Rightarrow^* w \Leftrightarrow (q, w, s) \vdash_N^* (q, \varepsilon, \varepsilon)$$

$\Leftarrow)$ Molto più difficile, viene solo schematizzata sulle slide (slide 18 lez 10).

3. Proprietà e pumping theorem

giovedì 10 novembre 2022 20:13

3.1 - Teorema - Proprietà 1

I linguaggi liberi sono chiusi per:

- 1) Unione
- 2) Concatenazione
- 3) Ripetizione (stella di Kleene)

DIMOSTRAZIONE

Slide 1 lezione 11

3.2 - Teorema - Proprietà 2

Intersezione $L_1 \cap L_2$ con

- L_1 libero
- L_2 regolare
- (o viceversa)

È un linguaggio libero.

DIMOSTRAZIONE

Slide 2 lezione 11

Concettualmente, tu crei una fusione dei due automi, dove facendo un passaggio con un input per un automa, lo fa anche per l'altro. In quanto il PDA del linguaggio libero ha sicuramente più "step" dell'automa DFA, in quanto ha passaggi in più riguardanti la pila. Quindi puoi essenzialmente ricreare il DFA dentro il PDA.

APPLICAZIONI

Per esempio, per dimostrare che $L = \{ww \mid w \in \{a, b\}^*\}$ è libero o non possiamo fare:

Allora $L \cap a^*b^*a^*b^*$ dovrebbe essere libero.

Però possiamo dimostrare (facilmente) che la loro intersezione, ovvero $\{a^n b^m a^n b^m \mid n, m \geq 0\}$, non è libera.

Quindi neanche L è libero, in quanto abbiamo trovato che intersecato che con un linguaggio regolare non ci dà un linguaggio libero.

OSSERVAZIONI

- 1) $L_1 \cap L_2$ non regolare \wedge L_2 regolare \Rightarrow L_1 non regolare

(Perché i linguaggi regolari sono chiusi per intersezione)

- 2) $L_1 \cap L_2$ non libero \wedge L_2 regolare \Rightarrow L_1 non libero

(Perché l'intersezione tra libero e regolare dà un libero)

3.3 - Teorema - Proprietà 3

I linguaggi liberi non sono chiusi per intersezione.

I linguaggi liberi non sono chiusi per complementazione. (Se sono chiusi per complementazione dovrebbero anche essere chiusi per intersezioni).

3.4 - PUMPING THEOREM

Se L è libero, allora $\exists N > 0$ tale che, $\forall z \in L$ tali che $|z| \geq N$, $\exists u, v, w, x, y$ tali che:

- 1) $z = uvwxy$
- 2) $|vwx| \leq N$
- 3) $|vx| \geq 1$
- 4) $\forall k \geq 0 \quad uv^kwx^ky \in L$

DIMOSTRAZIONE (Non completa)

Sulle slide si trova 5-6-7 lezione 11

Copio una dimostrazione che mi sembra più chiara da internet:

Sia G una grammatica libera.

In particolare, per questa dimostrazione si consideri G una grammatica del tipo "*Chomsky normal form*". Queste sono grammatiche che possono avere solo 3 tipi di produzioni, ovvero $A \rightarrow BC$ oppure $A \rightarrow a$ oppure $S \rightarrow \epsilon$. Tutte le grammatiche libere da contesto possono essere portate a questa forma e viceversa.

G ha k variabili.

$$L(G) = G$$

Consideriamo $N = 2^k$

Da notare che N viene anche spesso scritta come p ("pumping length")

Consideriamo $z \in L$ tale che $|z| \geq N = 2^k$

Consideriamo quindi un albero di derivazione per z : l'altezza di questo albero è almeno $k+1$.

Per dimostrare questo, dobbiamo considerare la sua forma. Quindi possiamo immaginare che:

In quanto ha k simboli, allora per arrivare ad esprimere 2^k lettere diverse, ogni lettera è sibling di 1 al massimo. E noi miriamo al massimo numero di sibling, così da avere la minor altezza possibile.

Può avere al massimo 1 sibling in quanto usiamo le *Chomsky normal form*.

Se non usassimo questa forma, allora dovremmo considerare $N = b^{|NT|+1}$, dove b è il massimo numero di simboli che compaiono nella parte destra di una produzione in R .

Ok basta mi arrendo con questa dimostrazione, ci son troppe cose a caso. Però mi ha aiutato, se qualcuno leggesse sti appunti, questa è la dimostrazione (slide 5-6):

<https://courses.enrgr.illinois.edu/cs373/sp2013/Lectures/lec17.pdf>

DIMOSTRAZIONE DEL PROF:

Sia $G = (NT, T, R, S)$ una grammatica tale che $L = L(G)$

Sia b il massimo fattore di ramificazione di un albero di derivazione (ovvero il massimo numero di simboli che compaiono nella parte destra di una produzione in R)

$$b = \max\{ |a| \text{ tale che } A \rightarrow a \in R \}$$

(OSS: $b \geq 2$, altrimenti la grammatica sarebbe banale (ovvero ogni non terminale ci porta ad un terminale oppure ad un unico non terminale. AKA stiamo definendo un

linguaggio composto da una sola parola))

Un albero di altezza h (con 0 la radice) e fattore di ramificazione b , ha al più b^h foglie.

Fissiamo $N = b^{|NT|+1}$ (Quindi $N > b^{|NT|}$ dato che $b \geq 2$)

Allora, ogni albero di derivazione per z , con $|z| \geq N$, deve avere altezza almeno $|NT| + 1$. Questo è valido per quello che abbiamo detto prima.

QUINDI, prendiamo una qualunque $z \in L : |z| \geq N$.

Consideriamo il suo albero di derivazione (se ne possiede più di uno, perché G è ambigua, prendiamo quello col minor numero di nodi).

DUNQUE,

$$|z| \geq N$$

- \Rightarrow albero con altezza $\geq |NT| + 1$
- $\Rightarrow \exists$ un cammino da radice S ad una foglia con almeno $|NT| + 2$ nodi
- \Rightarrow Quel cammino attraversa $|NT| + 1$ nodi interni etichettati con un non-terminale (la foglia è etichettata con un terminale)
- \Rightarrow Almeno un nonterminale si ripete in quel cammino

ALLORA

$$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvwx$$

E il secondo passaggio lo possiamo ripetere quante volte vogliamo.

$$S \Rightarrow^* uAy \Rightarrow^* uw$$

$$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvwx$$

$$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow uv^2Ax^2y \Rightarrow^* uv^2wx^2y$$

...

Quindi è valido $\forall k$, ora dobbiamo solo verificare i vincoli:

- $|vx| \geq 1$
Ovvio: se entrambe ε , allora l'albero per $k = 0$ genererebbe ancora z , ed avrebbe meno nodi, contraddicendo l'ipotesi di aver scelto il più piccolo albero
- $|vwx| \leq N$
Ovvio: il cammino da A alla foglia è di lunghezza $\leq |NT| + 1$ (cioè usa $|NT| + 2$ nodi al massimo, di cui uno è il terminale foglia) \Rightarrow la A in alto non può generare parole più lunghe di $b^{|NT|+1} = N$

3.5 - USO del PUMPING THEOREM

$\neg P(L) \Rightarrow L$ non è libero

Ovvero, se riusciamo a dimostrare che non vale il *pumping theorem* per un linguaggio L , allora esso non è libero. Non è sufficiente affinché un linguaggio sia libero: se il pumping theorem vale per un linguaggio, non significa che esso è libero, ma bisogna dimostrare anche altre proprietà del linguaggio.

Per fare questo dobbiamo quindi dimostrare che:

Se $\forall N > 0 \exists z \in L$ con $|z| \geq N$ tale che

$\forall u, v, w, x, y$. (se (1) $z = uvwx^ky$

(2) $|vwx| \leq N$

(3) $|v^k| \geq 1$

allora $\exists k \geq 0$. $uv^kw^kx^ky \notin L$)

allora L non è libero

SEGUONO ESEMPI SLIDE 9-11

4. Oltre i linguaggi liberi

venerdì 11 novembre 2022 16:49

Curiosità e cose che impareremo nelle slide 11-14 di lezione 11.

Comunque esistono gerarchie di:

- Grammatiche
- Linguaggi
- Automi

(Non sembrano avere una corrispondenza 1 a 1)

5. Linguaggi deterministici

venerdì 11 novembre 2022 16:55

5.1 - DEFINIZIONE - PDA Deterministico

Un PDA $N = (\Sigma, Q, \Gamma, \delta, q_0, \perp, F)$ è **deterministico** (DPDA) \Leftrightarrow

- 1) $\forall q \in Q, \forall z \in \Gamma, \delta(q, \varepsilon, z) \neq \emptyset \Rightarrow \delta(q, a, z) = \emptyset \quad \forall a \in \Sigma$
- 2) $\forall q \in Q, \forall z \in \Gamma, \forall a \in \Sigma \cup \{\varepsilon\}, |\delta(q, a, z)| \leq 1$

5.2 - DEFINIZIONE - Linguaggio libero deterministico

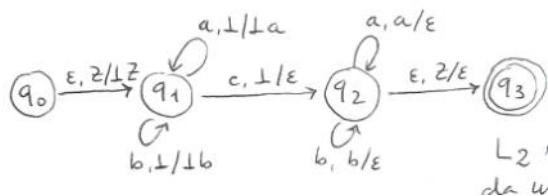
Un linguaggio è libero deterministico se è accettato **per stato finale** da un DPDA

5.2.1 - TEOREMA

La classe dei linguaggi liberi deterministici è inclusa **propriamente** nella classe dei linguaggi libri.

ESEMPIO:

$L_1 = \{ww^R \mid w \in \{a, b\}^*\}$ è libero, ma si può dimostrare che non esiste un DPDA che lo riconosce.
 $L_2 = \{wcw^R \mid w \in \{a, b\}^*\}$ è libero deterministico. Questo è il DPDA associato:



5.3 - PROPOSIZIONE

Se L è regolare, allora \exists DPDA N tale che $L = L[N]$ (NB. Per stato finale).

DIMOSTRAZIONE

Se L è regolare, allora \exists DFA M tale che $L = L[M]$.

A partire da M , posso costruire un DPDA N che si comporta come M senza mai manipolare lo stack.
 $\Rightarrow L = L[N]$

5.4 - FATTO

Un linguaggio libero deterministico L è riconosciuto da un DPDA per pila vuota $\Leftrightarrow L$ gode della "prefix property":

$$\neg \exists x, y \in L \text{ tali che } x \text{ è prefisso di } y$$

Osserva che $L = \{wcw^R \mid w \in \{a, b\}^*\}$ gode della prefix property, quindi si può anche creare un DPDA per pila vuota.

QUINDI

- a. Se L è libero deterministico non gode della *prefix property*, non può essere riconosciuto da un DPDA per pila vuota.
- b. Se L è libero deterministico e gode della *prefix property*, allora può essere riconosciuto da un DPDA per pila vuota.
- c. Se L è libero deterministico, allora $L\$ = \{w\$ \mid w \in L\}$ gode della *prefix property*.
 $\Rightarrow L\$$ può essere riconosciuto da un DPDA per pila vuota

SEGUONO ESEMPI SLIDE 3-5 LEZIONE 12

5.5 - PROPOSIZIONE

Se L è libero deterministico (cioè riconosciuto da un DPDA per stato finale), allora L è generabile da una grammatica libera **non ambigua**.

\Rightarrow I linguaggi liberi deterministici non sono ambigu.

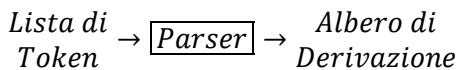
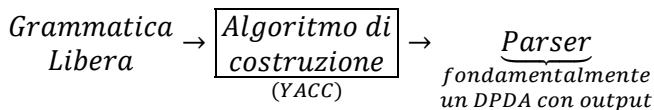
5.6 - PROPRIETA' DEI LINGUAGGI LIBERI DETERMINISTICI

- **Sono chiusi per complementazione**
 - o Cioè se \exists DPDA N tale che $L = L[N]$, allora esiste un DPDA N' tale che $\bar{L} = L[N']$, dove $\bar{L} = \Sigma^* \setminus L$.
 - o Idea della prova: bisogna rendere totale la δ di N , eventualmente aggiungendo degli stati non finale, e poi N' si ottiene da questo N "aumentato", semplicemente scambiando finale non finali (come fatto con i DFA per dimostrare che i regolari sono chiusi per complementazione)
- **Non sono chiusi per intersezione**
 - o $L_1 = \{a^n b^n c^m \mid n, m \geq 0\}$ è libero deterministico
 - o $L_2 = \{a^m b^n c^n \mid n, m \geq 0\}$ è libero deterministico
 - o MA
 - o $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$ non è libero
- **Non sono chiusi per unione**
 - o Se, per assurdo, lo fossero, allora
 - $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$
 - o E quindi risulterebbero chiusi per intersezione \Rightarrow impossibile!

6. Analizzatori sintattici (Parser)

lunedì 14 novembre 2022 17:26

6.1 - Introduzione



I Parser possono essere:

- NONDETERMINISTICI

- Se, durante la ricerca di una derivazione, **si scopre che una scelta è improduttiva** e non porta a riconoscere l'input, **il parser torna indietro** (backtracking), disfa parte della derivazione appena costruita, e sceglie un'altra produzione, tornando a leggere (parte de)l'input

- DETERMINISTICI

- Leggono l'input una sola volta, ogni loro decisione è **definitiva**

Entrambi cercano di sfruttare informazioni dell'input per guidare la ricerca della derivazione.

I Parser possono anche essere:

- TOP-DOWN

- Ricostruiscono una derivazione **leftmost** a partire dal simbolo iniziale S (**all'inizio sulla pila**)

- BOTTOM-UP

- Ricostruiscono una derivazione **rightmost** (a rovescio) a partire dalla stringa w, cercando di ridurla al simbolo iniziale S (**alla fine sulla pila**)

Entrambi cercano di sfruttare quello che vedono dall'input per guidare la della derivazione.

Noi, in particolare, vedremo:

- Parse top-down deterministici

- ottenuti a partire da grammatiche LL(K), in particolare LL(1)

- Parser bottom-up deterministici

- ottenuti a partire da grammatiche LR(K), in particolare LR(0), SLR(1), LR(1) e LALR(1).

7. Introduzione al top-down parsing

lunedì 14 novembre 2022 17:45

7.1 - DEFINIZIONE

Data $G = (NT, T, R, S)$ libera, costruiamo il PDA $M = (T, \{q\}, T \cup NT, \delta, q, S, \emptyset)$ che riconosce per pila vuota (da notare che non ci sono stati terminali), dove δ è definita come:

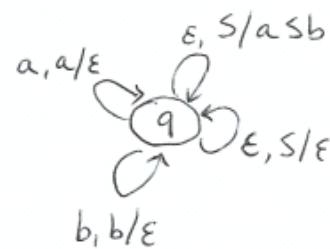
- $(q, \beta) \in \delta(q, \varepsilon, A) \quad \text{se } A \rightarrow \beta \in R \quad (\text{espandi})$
- $(q, \varepsilon) \in \delta(q, a, a) \quad \forall a \in T \quad (\text{consuma})$

Tale che $L(G) = P[M]$

7.2 - ESEMPI

$S \rightarrow aSb \mid \varepsilon$, immaginiamo di avere la stringa formata nel modo $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$.

Possiamo fare il PDA:



E usandolo possiamo portarci in stati istantanei del tipo:

$(q, aabb, S) \vdash (q, aabb, aSb) \vdash (q, abb, Sb) \vdash (q, abb, aSbb) \vdash (q, bb, Sbb) \vdash (q, bb, bb) \vdash (q, b, b)$
 $\vdash (q, \varepsilon, \varepsilon)$

Ogni passaggio di **espansione** lo possiamo immaginare come formare l'albero di derivazione a partire da S .

Se abbiamo come risultato una pila vuota, allora vuol dire che i passaggi di espansione che abbiamo fatto sono proprio l'albero di derivazione che cercavamo.

- Derivazione canonica sinistra (leftmost)
- Costruiamo l'albero dall'alto al basso

Notiamo che questo PDA è non deterministico, in quanto abbiamo due passaggi diversi per la transizione $\delta(q, \varepsilon, S)$.

Possiamo renderlo deterministico **scegliendo la produzione in base al simbolo in lettura (look-ahead)**:

- Se leggo "a" \Rightarrow espando $S \rightarrow aSb$
- Se leggo "b" \Rightarrow espando $S \rightarrow \varepsilon$

$G [S \rightarrow aSb \mid \varepsilon]$ vedremo essere una grammatica di classe LL(1), nel senso che consente di costruire un Parser/DPDA. In particolare:

$$\begin{array}{ccc}
 \text{left} - \text{to} - \text{right} / \text{left} - \text{derivation} & & (1) \\
 \overbrace{\quad \quad \quad \quad \quad}^L \quad \overbrace{\quad \quad \quad \quad \quad}^L & & \overbrace{\quad \quad \quad \quad \quad}^L \\
 \text{modo in cui viene letto l'input} & \text{genera una derivazione leftmost} & \text{guarda un solo simbolo di look-ahead}
 \end{array}$$

Un altro esempio (per comprendere meglio il look-ahead) è:

$$G \left[S \rightarrow aSb \mid ab \right]$$

In questo caso devo guardare 2 simboli di look-ahead (in avanti):

- Se leggo "aa" \Rightarrow espando $S \rightarrow aSb$
- Se leggo "ab" \Rightarrow espando $S \rightarrow ab$

Quindi G è di classe LL(2).

7.3 - LINGUAGGI NON ADATTI

ESEMPIO 1

$$G \left[S \rightarrow Sb \mid a, \quad L(G) = ab^* \right]$$

G è left-recursive!

La mossa $\delta(q, \varepsilon, S) \Rightarrow (q, Sb)$ diverge.

Inoltre, per colpa di questa mossa:

- (Se voglio renderlo deterministico) Non consumo mai l'input
 - o Perché non so quante b ho in input visto che leggo da sinistra a destra
 - o Quindi continuo a fare questa mossa
- Non deterministicamente può funzionare, ma non ci basta usare il look-ahead per risolvere questo problema.

\Rightarrow bisogna manipolare la grammatica G per ottenere una equivalente G' senza ricorsione sinistra.

$$G' \left[\begin{array}{l} S \rightarrow aA \\ A \rightarrow bA \mid \varepsilon \end{array} \right] \quad L(G) = L(G')$$

G' è adatto al top-down parsing:

- Se leggo "a" e sulla pila c'è S
 \Rightarrow espando con $S \rightarrow aA$
- Se leggo "b" e sulla pila c'è A
 \Rightarrow espando con $A \rightarrow bA$
- Se leggo "\$" e sulla pila c'è A
 \Rightarrow espando con $A \rightarrow \varepsilon$

G' è LL(1)

ESEMPIO 2

$$G \left[\begin{array}{l} S \rightarrow abB \mid accC \\ B \rightarrow \dots \\ C \rightarrow \dots \end{array} \right]$$

- Se leggo "a" e sulla pila c'è S, allora come espando S?

- O leggo due simboli, quindi diventa LL(2)
 - $ab \Rightarrow S \rightarrow abB$, $ac \Rightarrow S \rightarrow acC$
- Oppure "fattorizzo"
 - $$G \left[\begin{array}{c} S \rightarrow aE \\ E \rightarrow bB \mid cC \\ \dots \end{array} \right]$$
 - Ora non c'è non determinismo: basta 1 solo simbolo di look-around

8. Introduzione al bottom-up parsing

martedì 15 novembre 2022 16:39

8.1 - DEFINIZIONE

Data una grammatica libera $G = (NT, T, R, S)$ costruiamo un PDA M che riconosce $L(G) \cdot \$$

$$M = (T, \{q\}, T \cup NT \cup \{Z\}, \delta, q, Z, \emptyset)$$

Dove definiamo δ nel seguente modo:

1. $(q, aX) \in \delta(q, a, X), \quad \forall a \in T, \quad \forall x \in T \cup NT \cup \{Z\}, \quad (\text{SHIFT})$
2. $(q, A) \in \delta(q, \varepsilon, a^R), \quad \text{se } A \rightarrow a \in R, \quad (\text{REDUCE})$
 - a. Generalizzazione dei PDA in cui si consuma una stringa sulla pila anziché solo sul top.
 - b. Questo perché noi stiamo consumando la stringa a^R
3. $(q, \$) \in \delta(q, \$, SZ) \quad (\text{ACCEPT})$
 - a. In particolare, S deve essere la fine sulla pila
 - b. $\$$ è il simbolo che indica la fine dell'output

Questo ci costruisce una LR(K)

In quanto siamo leggendo da sinistra a destra la stringa in input.

Stiamo "calcolando" la derivazione rightmost.

Inoltre utilizziamo K simboli di look-ahead.

Gli esempi sono molto lunghi da riscrivere, ma si possono leggere da slide 13 a 19.
Aiutano molto con la comprensione.

8.2 - NOTE

C'è parecchio non-determinismo:

- Conflitti **shift-reduce**
 - o Se facessi più shift avrei un reduce diverso, che mi porta ad un percorso infruttuoso
- Conflitti **reduce-reduce**
 - o Quando si possono fare più reduce diversi per quello che abbiamo nella pila.

⇒ Per ottenere un DPDA, serve introdurre informazioni aggiuntive per risolvere i conflitti:

- Più stati
 - o O strutture particolari di supporto alla decisione: DFA dei prefissi variabili
- Look-ahead
 - o Guardare l'input in avanti

È buona norma inoltre fare shift solo se si generano prefissi di produzioni sulla stringa: se fare un shift ci porta in una situazione che non è un prefisso di alcuna produzione, allora evito di farlo.

8.3 - PROBLEMA IMPORTANTE

$S \rightarrow \varepsilon$ è sempre applicabile come riduzione. ($(q, S) \in \delta(q, \varepsilon, \varepsilon)$ oppure $(q, SX) \in \delta(q, \varepsilon, X)$)

È bene quindi evitare di usare grammatiche libere con produzioni ε quando si vuole usare la tecnica di bottom-up parsing (shift-reduce).

9. Semplificare le grammatiche

martedì 15 novembre 2022 17:29

9.1 - Cosa dobbiamo fare

Per avere PDA più efficienti / più piccoli e con **minore nondeterminismo** è necessario semplificare le grammatiche attraverso:

- 1) Eliminare le **produzioni ϵ** (del tipo $A \rightarrow \epsilon$) che sono inadatti al **bottom-up** parsing.
- 2) Eliminare le **produzioni unitarie** (del tipo $A \rightarrow B$) che possono creare dei cicli $A \Rightarrow^+ A$.
- 3) Eliminare **simboli inutili**, cioè quei terminali e nonterminali che non sono raggiungibili / generabili a partire dal simbolo iniziale S.
- 4) Eliminare la **ricorsione sinistra** (del tipo $A \rightarrow A\alpha$), perché inadatta al **top-down** parsing.
- 5) **Fattorizzare** la grammatica, per ottenere grammatiche con meno nondeterminismo nel **top-down** parsing.

9.1 Eliminare le produzioni ε

mercoledì 16 novembre 2022 12:12

9.1.1 - Obiettivo

- **Input**
 - o G libera con produzioni ε (del tipo $A \rightarrow \varepsilon$)
- **Output**
 - o G' libera, senza produzioni ε , tale che $L(G') = L(G) \setminus \{\varepsilon\}$

9.1.2 - Osservazione

Se $\varepsilon \in L(G)$ e vogliamo una G'' tale che $L(G) = L(G'')$, basta considerare $G' = (NT, T, S, R')$ (dove G' è l'output dell'algoritmo) e definire $G'' = G' \cup \{S' \rightarrow \varepsilon | S\}$

$$G'' = (NT \cup \{S'\}, T, S', R' \cup \{S' \rightarrow \varepsilon | S\})$$

A parole: se vogliamo trasformare G' , che è una grammatica senza produzioni nulle, e vogliamo ritrasformarla in G'' che ha anche la produzione nulla, allora ci basta aggiungere un nuovo simbolo iniziale che può generare o una produzione nulla o "riparte" da S come la G normale.

9.1.3 - DEFINIZIONE - Simbolo annullabile

$A \in NT$ tale che $A \Rightarrow^+ \varepsilon$

$N(G) = \{A \in NT \mid A \Rightarrow^+ \varepsilon\}$ viene calcolato induttivamente:

0. $N_0(G) = \{A \in NT \mid A \rightarrow \varepsilon \in R\}$
- i. $N_{i+1}(G) = N_i(G) \cup \{B \in NT \mid B \rightarrow C_1 \dots C_k \in R \text{ e } C_1, \dots C_k \in N_i(G)\}$

OSSERVAZIONE

$N_i(G) \subseteq N_{i+1}(G)$ perché aggiungo qualcosa ad ogni passo

$\exists i_c$ tale che $N_{i_c}(G) = N_{i_c+1}(G)$ perché NT è finito

FATTO

L'insieme $N(G) = N_{i_c}(G)$ è esattamente l'insieme di tutti i simboli annullabili.

9.1.4 - ALGORITMO PER FARE G'

Una volta calcolato $N(G)$ per $G = (NT, T, S, R)$, costruiamo la grammatica $G' = (NT, T, S, R')$ dove modifichiamo le produzioni.

$\forall R$ tale che $A \rightarrow \alpha$ con $\alpha \neq \varepsilon$

In cui occorrono simboli annullabili z_1, \dots, z_k

Mettiamo in R' tutte le produzioni del tipo:

$A \rightarrow \alpha'$, dove α' si ottiene da α cancellando tutti i possibili sottoinsiemi di z_1, \dots, z_k (incluso \emptyset) **ad eccezione** del caso in cui α' risulta in ε .

- o In G' non mettiamo produzioni $A \rightarrow \varepsilon \in R$
- o In G' non introduciamo mai produzioni del tipo $A \rightarrow \varepsilon$

PROPOSIZIONE SU QUESTO ALGORITMO

Data una grammatica libera G , la grammatica G' determinata dall'algoritmo sopra non ha ϵ -produzioni e $L(G') = L(G) \setminus \{\epsilon\}$

ESEMPIO

Esempio:

$$G = \begin{cases} S \rightarrow AB \\ A \rightarrow aAA \mid \epsilon \\ B \rightarrow bBB \mid \epsilon \end{cases} \quad \begin{aligned} N_0(G) &= \{A, B\} \\ N_1(G) &= \{A, B, S\} \\ &= N(G) \end{aligned}$$

(23)

Consideriamo $S \rightarrow AB \in R$. Dobbiamo considerare
4 casi:

$$\begin{aligned} \emptyset &\Rightarrow S \rightarrow AB & S \rightarrow AB \mid A \mid B \\ \{B\} &\Rightarrow S \rightarrow A \\ \{A\} &\Rightarrow S \rightarrow B \\ \{A, B\} &\Rightarrow S \not\rightarrow \epsilon \text{ non la mettiamo in } R' \end{aligned}$$

Ora consideriamo $A \rightarrow aAA \in R$. Dobbiamo considerare

$$\begin{aligned} 4 \text{ casi:} \\ \emptyset &\Rightarrow A \rightarrow aAA \\ \{A\} &\Rightarrow A \rightarrow aA \} \text{ duplicata} \\ \{A\} &\Rightarrow A \rightarrow aa \\ \{A, A\} &\Rightarrow A \rightarrow a \end{aligned} \Rightarrow A \rightarrow aAA \mid aA \mid a$$

Ora consideriamo $B \rightarrow bBB$, e nello stesso modo
che prima, otteniamo $B \rightarrow bBB \mid bB \mid b$

Le due ϵ -produzioni $A \rightarrow \epsilon$ e $B \rightarrow \epsilon$ sono ignorate.

$$\Rightarrow G' = \begin{cases} S \rightarrow AB \mid A \mid B \\ A \rightarrow aAA \mid aA \mid a \\ B \rightarrow bBB \mid bB \mid b \end{cases}$$

N.B. $\epsilon \in L(G)$ $S \Rightarrow AB \Rightarrow B \Rightarrow \epsilon$
 $\epsilon \notin L(G')$

$$G'' = \begin{cases} S' \rightarrow \epsilon \mid S \\ S \text{ come per } G' \end{cases} \quad \text{e tale che } L(G) = L(G'')$$

9.2 Eliminare le produzioni unitarie

mercoledì 16 novembre 2022 15:22

9.2.1 - DEFINIZIONI - Produzioni unitarie

- Produzioni unitarie:
 - o $A \rightarrow B$, con $A, B \in NT$
- Copie unitarie:
 - o (A, B) tale che $A \Rightarrow^* B$
Usando solo produzioni unitarie

9.2.2 - CALCOLARE coppie unitarie induttivamente

- i. $U_0(G) = \{(A, A) \mid A \in NT\}$
- i. $U_{i+1}(G) = U_i(G) \cup \{(A, C) \mid (A, B) \in U_i(G) \text{ e } B \rightarrow C \in R\}$

OSSERVAZIONI

$U_i(G) \subseteq U_{i+1}(G)$
 $\exists i_c$ tale che $U_{i_c}(G) = U_{i_c+1}(G)$ perché NT è finito.

Per definizione, $U(G) = U_{i_c}(G)$, detto insieme di tutte le coppie unitarie.

9.2.3 - ALGORITMO

Data $G=(NT, T, R, S)$ libera, si definisce $G'=(NT, T, R', S)$ dove, per ogni $(A, B) \in U(G)$, R' contiene tutte le produzioni $A \rightarrow \alpha$, dove $B \rightarrow \alpha \in R$ e non è unitaria.

OSSERVAZIONE

Poiché, per ogni $A \in NT$, la coppia $(A, A) \in U(G)$, R' contiene tutte le produzioni non-unitarie di R , e in aggiunta un po' di altro.

9.2.4 - TEOREMA SULL'ALGORITMO

Sia $G=(NT, T, R, S)$ libera e sia $U(G)$ l'insieme delle sue coppie unitarie.

Sia $G'=(NT, T, R', S)$ la grammatica ottenuta dall'algoritmo sopra.

Allora G' non ha produzioni unitarie e $L(G)=L(G')$.

9.2.5 - ESEMPIO

Esempio $\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * A \mid A \\ A \rightarrow a \mid b \mid (E) \end{array}$

gram. non ambigua
 G per espr. aritmetiche
 • però ha prod. unitarie
 $E \rightarrow T$ e $T \rightarrow A$

$$U_0(G) = \{(E, E), (T, T), (A, A)\}$$

$$U_1(G) = U_0(G) \cup \{(E, T), (T, A)\}$$

$$U_2(G) = U_1(G) \cup \{(E, A)\} = U_3(G) = U(G)$$

$$G' \left\{ \begin{array}{ll} E \rightarrow E + T & \text{produzioni presenti} \\ T \rightarrow T * A & \\ A \rightarrow a \mid b \mid (E) & \text{grazie a } U_0(G) \end{array} \right.$$

in aggiunta

$$\begin{array}{ll} E \rightarrow T * A & \text{perché } (E, T) \in U_1(G) \\ T \rightarrow a \mid b \mid (E) & \text{perché } (T, A) \in U_1(G) \\ E \rightarrow a \mid b \mid (E) & \text{perché } (E, A) \in U_2(G) \end{array}$$

Quindi, riassumendo, G' è

$$\begin{array}{l} E \rightarrow E + T \mid T * A \mid a \mid b \mid (E) \\ T \rightarrow T * A \mid a \mid b \mid (E) \\ A \rightarrow a \mid b \mid (E) \end{array}$$

non contiene produzioni unitarie ed è equivalente
a G

9.3 Eliminare i simboli inutili

mercoledì 16 novembre 2022 15:41

9.3.1 - DEFINIZIONI

Un simbolo $X \in T \cup NT$ è:

- **Generatore** $\Leftrightarrow \exists w \in T^* \text{ con } X \Rightarrow^* w$
- **Raggiungibile** $\Leftrightarrow S \Rightarrow^* \alpha X \beta \text{ per qualche } \alpha, \beta \in (T \cup NT)^*$
- **Utile** \Leftrightarrow è sia generatore, sia raggiungibile.

Ovvero:

$$S \Rightarrow^* \alpha X \beta \Rightarrow^* z \in L(G)$$

Cioè X compare in almeno una derivazione di una stringa $z \in L(G)$

ESEMPIO:

$$G = \left[\begin{array}{l} S \rightarrow AB \\ B \rightarrow b \end{array} \right] a$$

Prima capiamo quali sono i generatori, ovvero l'insieme di simboli tali che in un numero finito di passaggi si forma una stringa appartenente ad un * dei terminali.

In questo caso, abbiamo:

$$\left. \begin{array}{l} a \Rightarrow^* a \\ b \Rightarrow^* b \\ S \Rightarrow^* a \\ B \Rightarrow^* b \end{array} \right\} \Rightarrow \text{generatori} = \{S, B, a, b\}$$

Notiamo che manca la A: essa, quindi, non è un generatore. Possiamo eliminare tutte le produzioni che includono la A.

$$G' = \left[\begin{array}{l} S \rightarrow a \\ B \rightarrow b \end{array} \right]$$

Ora B non è più raggiungibile da S, allora eliminiamo anche B, e otteniamo la grammatica G'' (equivalente a G ma che non contiene simboli inutili):

$$G'' = [S \rightarrow a]$$

9.3.2 - Come calcolare i generatori?

Ricorda: X è un generatore $\Leftrightarrow \exists w \in T^*. x \Rightarrow^* w$

- 1) $G_0(G) = T$, se $a \in T, a \Rightarrow^* a$ quindi tutti i terminali sono generatori
- 2) $G_{i+1}(G) = G_i(G) \cup \{B \in NT \mid B \rightarrow c_1 \dots c_k \in R \wedge c_1, \dots, c_k \in G_i(G)\}$

OSSERVAZIONI

Nel passo 2 è contemplato anche il caso $k=0$, cioè $B \rightarrow \epsilon \in R$
Ovvero: se $B \rightarrow \epsilon \in R$, B è un generatore

- $G_i(G) \subseteq G_{i+1}(G)$

- $\exists i_c$ tale che $G_{i_c}(G) = G_{i_c+1}(G)$ perché $T \cup NT$ è finito.

FATTO

L'insieme $G(G) = G_{i_c}(G)$ è esattamente l'insieme di tutti i simboli generatori di G .

9.3.3 - Come calcolare i raggiungibili?

Ricorda: X è raggiungibile $\Leftrightarrow S \Rightarrow^* aX\beta$ per qualche $a, \beta \in (NT \cup T)^*$

- 1) $R_0(G) = \{S\}$
- 2) $R_{i+1}(G) = R_i(G) \cup \bigcup_{\substack{B \in R_i(G) \\ B \rightarrow x_1 \dots x_k \in R}} \{x_1, \dots, x_k\}$

(L'unione di tutti i simboli che vengono generati da S , poi da quelli generati da S , etc..)
 $(Ex: S \rightarrow AB \mid A \rightarrow a \mid C \rightarrow d \text{ qua vengon raggiunti solo "S", "A", "B" e "a" da questo})$

OSSERVAZIONI

- $R_i(G) \subseteq R_{i+1}(G)$
- $\exists i_c$ tale che $R_{i_c}(G) = R_{i_c+1}(G)$ perché $T \cup NT$ è finito.

FATTO

L'insieme $R(G) = R_{i_c}(G)$ è esattamente l'insieme di tutti i simboli generatori di G .

9.3.4 - Eliminare i simboli inutili

TEOREMA

Sia $G = (NT, T, R, S)$ una grammatica libera tale che $L(G) \neq \emptyset$

- 1) Sia G_1 la grammatica che si ottiene da G eliminando tutti i simboli che non appartengono a $G(G)$, e tutte le produzioni che fanno uso di almeno uno di tali simboli
 - a. Poiché $L(G) \neq \emptyset$ allora S è un generatore e così rimarrà in G_1 e poi in G_2
- 2) Sia G_2 la grammatica che ottiene da G_1 eliminando tutti i simboli che non appartengono a $R(G_1)$, e tutte le produzioni che fanno uso di almeno uno di tali simboli.

Allora G_2 non ha simboli inutili e $L(G_2) = L(G)$

DIMOSTRAZIONE

$L(G_2) \subseteq L(G)$ è ovvio, perché G_2 contiene meno produzioni di G (e tutte quelle che contiene ci sono anche in G)

$L(G) \subseteq L(G_2)$: dobbiamo dimostrare che se $S \Rightarrow_G^* w$ allora $S \Rightarrow_{G_2}^* w$.

Ma ogni simbolo usato nella derivazione $S \Rightarrow_G^* w$ è ovviamente sia raggiungibile, sia generatore!!
 Allora quella derivazione è anche una derivazione per G_2

■

OSSERVAZIONE

L'ordine delle operazioni è importante: se prima elimino i non raggiungibili e poi i non generatori, posso aver lasciato dei simboli che diventano non raggiungibili.

Vi è un esempio per la comprensione a slide 11 Lez 13.

9.4 Forme normali

giovedì 24 novembre 2022 22:19

9.4.1 - Un po' di teoria

Esistono due forme, chiamate "forme normali", che garantiscono certe proprietà particolari della grammatica:

- Forma normale di Chomsky
- Forma normale di Greibach

FORMA NORMALE DI CHOMSKY

Tutte le sue produzioni sono della forma:

$$\begin{aligned} A &\rightarrow BC \\ A &\rightarrow a \end{aligned}$$

Con ε solo presenti in $S \rightarrow \varepsilon$

OSSERVAZIONI

Se G libera è in forma normale di Chomsky, allora

- Non ha ε – produzioni
- Non ha produzioni unitarie

Inoltre:

- Ogni grammatica libera G può essere trasformata in una equivalente G' in forma normale di Chomsky.

FORMA NORMALE DI GREIBACH

Tutte le sue produzioni sono della forma:

$$\begin{aligned} A &\rightarrow aBC \\ A &\rightarrow aB \\ A &\rightarrow a \end{aligned}$$

Con ε solo presenti in $S \rightarrow \varepsilon$

OSSERVAZIONI

Se G libera è in forma normale di Greibach, allora

- Non ha ε – produzioni
- Non è mai ricorsiva a sx!
- Ogni produzione applicata in una derivazione, allunga il prefisso di terminali \Rightarrow parser costruito da forme normali di Greibach sono meno non deterministici.

Inoltre:

- Ogni grammatica libera G può essere trasformata in una equivalente G' in forma normale di Greibach.

9.5 Eliminare la ricorsione sinistra

giovedì 24 novembre 2022 22:25

9.5.1 - Obiettivo

Dobbiamo eliminare la ricorsione sinistra in quanto è un problema per il parser top-down.

Definiamo produzione ricorsiva la produzione del tipo:

- produzione ricorsiva sx : $A \rightarrow A\alpha \in R$
- G è ricorsiva sx : $A \Rightarrow^+ A\alpha$ per qualche $A \in NT$ e $\alpha \in (T \cup NT)^*$

Quindi l'obiettivo del nostro algoritmo è:

- Input:
 - o G libero senza ϵ – produzioni, senza produzioni unitarie, ma con ricorsione sx non-immediata.
- Output:
 - o G' libera, senza ricorsione, ma può avere ϵ – produzioni

9.5.2 - Rimuovere ricorsione sinistra immediata

Per rimuovere la ricorsione sinistra istantanea possiamo usare questo algoritmo:

$$A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

(Le stringhe β_i non cominciano per A)

Queste produzioni posso essere rimpiazzate da:

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \dots \mid \beta_m A' \\ A' &\rightarrow a_1 A' \mid \dots \mid a_n A' \mid \epsilon \end{aligned}$$

9.5.3 - Algoritmo

Algoritmo per eliminare la ricorsione sinistra immediata e non-immediata.

Sia $NT = \{A_1, A_2, \dots, A_n\}$ in un ordine fisso

For i from 1 to n {

- 1) For j from 1 to i-1 {
 - Sostituisci ogni produzione della forma $A_i \rightarrow A_j \alpha$ con le produzioni $A_i \rightarrow \beta_1 \alpha \mid \dots \mid \beta_k \alpha$ dove $A_j \rightarrow \beta_1 \mid \dots \mid \beta_k$
 - Ovvero le produzioni correnti per A_j
- 2) Elimina la ricorsione immediata su A_i

ESEMPIO DELL'ALGORITMO

$$S \rightarrow Ba \mid b$$

$$B \rightarrow Bc \mid Sc \mid d$$

(38)

$$NT = \begin{cases} S, B \\ 1, 2 \end{cases}$$

- Per $i=1$ (cioè per S), il ciclo interno ($\gamma_{da\ 1 a\ 1}$) non viene eseguito; siccome non c'è ricorsione immediata per S , non faccio niente!
- Per $i=2$ (cioè $A_1=B$), il ciclo interno ($\gamma_{da\ 1 a\ 1}$) si esegue solo per $A_2=A_1=S$. Allora la produzione $B \rightarrow Sc$ viene rimpiazzata con

$$B \rightarrow Bac \mid bc$$

Ora le produzioni complesse per B sono

$$B \rightarrow Bc \mid Bac \mid bc \mid d$$

dalle quali dobbiamo eliminare la ricorsione immediata!

Il risultato è

$$B \rightarrow bcB' \mid dB'$$

$$B' \rightarrow cB' \mid acB' \mid \epsilon$$

ovvero la grammatica risultante è

$$S \rightarrow Ba \mid b$$

$$B \rightarrow bcB' \mid dB'$$

$$B' \rightarrow cB' \mid acB' \mid \epsilon$$

9.6 Fattorizzazione a sinistra

giovedì 24 novembre 2022 22:40

9.6.1 - Obiettivo

È importante nel top-down parsing.

Esempio (per la comprensione):

$$A \rightarrow aBbC \mid aBd$$

Se, in top-down parsing, sulla pila ha A e leggo in input a, non sono in grado di determinare quale produzione scegliere (nondeterminismo).

⇒ Raccolgo la parte comune (aB) alle 2 produzioni e introduco un nuovo nonterminale per rappresentare il "resto". Quindi abbiamo una nuova grammatica del tipo:

$$A \rightarrow aBA'$$

$$A' \rightarrow bC \mid d$$

9.6.2 - Algoritmo generale

- Inizializza $N = NT$
- Ripeti: il ciclo presente finché nessuna modifica è più possibile a N o all'insieme delle produzioni {
 - for each $A \in N$ {
 - Sia α il prefisso più lungo comune alle parti destre di alcune produzioni di A;
 - if $\alpha \neq \epsilon$ {
 - sia A' un nuovo nonterminale; $N := N \cup \{A'\}$;
 - riempiezza tutte le produzioni per A
 - $A \rightarrow \alpha \beta_1 \dots \mid \alpha \beta_k \} \gamma_1 \mid \dots \mid \gamma_n$ con le produzioni:
 $A \rightarrow \alpha A' \mid \gamma_1 \mid \dots \mid \gamma_n$
 $A' \rightarrow \beta_1 \mid \dots \mid \beta_k$
 }

9.6.3 - ESEMPIO

$$\left\{ \begin{array}{l} E \rightarrow T \mid T + E \mid T - E \\ \quad T \rightarrow A \mid A * T \quad \Rightarrow \text{si può fattorizzare} \\ \quad A \rightarrow a \mid b \mid (E) \end{array} \right.$$

$$\left\{ \begin{array}{l} E \rightarrow TE' \\ E' \rightarrow \varepsilon \mid + E \mid - E \\ \quad T \rightarrow AT' \quad \Rightarrow \text{versione equivalente fattorizzata} \\ \quad T' \rightarrow \varepsilon \mid * T \\ \quad A \rightarrow a \mid b \mid (E) \end{array} \right.$$

10. Top-down Parser (First e Follow)

venerdì 2 dicembre 2022 22:41

10.1 - Parser a discesa ricorsiva

Data una grammatica libera $G = (NT, T, S, R)$, per ogni nonterminale A con produzioni:

$$A \rightarrow x_1^1 \dots x_{n1}^1 \mid \dots \mid x_1^k \dots x_{nk}^k$$

Definiamo la funzione:

```
function A() {
    - Scegli nondeterministicamente h tra 1 e k
        Ovvero una produzione  $A \rightarrow x_1^h \dots x_{nh}^h$ 
    - For i from 1 to nh {
            if  $x_i^h \in NT$  then  $x_i^h()$ ;
            else if  $x_i^h =$  simbolo corrente in input then
                avanza di un simbolo sull'input
            else Fail();
                backtracking
                Torniamo alla
                scelta delle
                produzioni e
                ne scegliamo
                un'altra
        }
    return;
}
```

Si comincia invocando la funzione per S.

SEGUONO ESEMPI SULL'USO DI QUESTO ALGORITMO,

A slide 2

NOTE:

Il parser a discesa ricorsiva è molto inefficiente:

- NONDETERMINISMO:

Necessita di esplorare, nel peggior caso, tutte le alternative.

Quindi è esponenziale nella lunghezza della stringa w , dove la base b è data dal massimo numero di produzioni per uno stesso nonterminale. Costo computazionale:

$$O(b^{|w|})$$

Vista la sua inefficienza cerchiamo di guidare leggermente la scelta delle produzioni.
Come?

Guardiamo il prossimo carattere (o i prossimi caratteri) dell'input da leggere.

10.2 - Nuove funzioni ausiliarie

Introduciamo **First** e **Follow**.

ATTENZIONE:

Da qui in poi **assumiamo di avere un simbolo speciale \$**, che non faccia parte dei simboli di nessuna grammatica, che useremo come **segnalatore della fine dell'input**.

Perché? Un parser top-down è essenzialmente un DPDA che riconosce per pila vuota. Allora è necessario che L gode della *prefix property*, e $L \cdot \$$ sicuramente gode di questa proprietà.

FIRST

Data una grammatica libera G e $a \in (T \cup NT)^*$ diciamo che $\text{First}(a)$ è l'insieme dei **terminali** che possono stare in prima posizione in una stringa che si deriva da a .

- Per $x \in T, x \in \text{First}(a) \Leftrightarrow a \Rightarrow^* x\beta$ per $\beta \in (T \cup NT)^*$
- Se $a \Rightarrow^* \varepsilon$, allora $\varepsilon \in \text{First}(a)$

IDEA: so sempre quale produzione scegliere in base all'input che leggo se non ho first in comune. Per esempio, se due first sono $\{a\}$ e $\{c\}$ e leggo "a", allora so che dovrò scegliere la produzione con first $\{a\}$.

Non è però abbastanza: un esempio di nondeterminismo (anche se non ci sono first in comune, ovvero se l'intersezione tra tutte le first è \emptyset) si trova a slide 5.

ESEMPI SULLE SLIDE 4-5

FOLLOW

Data una grammatica libera G e $A \in NT$ diciamo che $\text{Follow}(A)$ è l'insieme dei terminali che possono comparire immediatamente a destra di A in una forma sentenziale.

- Per $x \in T, x \in \text{Follow}(A) \Leftrightarrow S \Rightarrow^* \alpha A x \beta$
Per qualche $\alpha, \beta \in (T \cup NT)^*$
- $\$ \in \text{Follow}(A)$ se $S \Rightarrow^* \alpha A$
(Poiché $S \Rightarrow^* S$, allora $\$ \in \text{Follow}(S)!$)

10.3 - ALGORITMI FIRST E FOLLOW

10.3.1 - Calcolo dei FIRST

Sia $N(G) \subseteq NT$ l'insieme dei simboli annullabili ($A \in N(G) \Leftrightarrow A \Rightarrow^* \varepsilon$)

- $\forall x \in T, \quad \text{First}(x) = \{x\}$
- $\forall X \in NT, \quad \text{inizializza } \text{First}(X) = \emptyset$
- Ripeti il seguente ciclo finché nessun $\text{First}(X)$ viene più modificato in una iterazione:
 - Per ogni produzione $X \rightarrow y_1 \dots y_k$

- Per ogni i da 1 a k
 $\text{Se } (y_1, \dots, y_{i-1} \in N(G)) \quad // \text{True se } i = 1$
 Allora $\text{First}(X) := \text{First}(X) \cup (\text{First}(y_i) \setminus \{\varepsilon\})$

- Per ogni $X \in N(G)$, $\text{First}(X) := \text{First}(X) \cup \{\varepsilon\}$

Concettualmente, vado a vedere per ogni produzione se posso annullare tutto quello che c'è prima di i , e se sì, allora lo aggiungi ai First, se no allora lo ignoro.

Il motivo per cui sottraggo ε è perché se dovesse essere un simbolo annullabile, allora in ogni caso lo aggiungo alla fine, ma se non lo fosse, allora non lo è perché un suo figlio lo è, in quanto può generare altre cose.

Un altro modo più semplice per fare questi passaggi è:

- $\text{First}(\varepsilon) = \{\varepsilon\}$
- $\text{First}(X\beta) = \text{First}(X) \text{ se } x \notin N(G)$
- $\text{First}(X\beta) = (\text{First}(X) \setminus \{\varepsilon\}) \cup \text{First}(\beta) \text{ se } X \in NT(G)$

In pratica, se $A \rightarrow a_1 | \dots | a_k$, allora:

$$\text{First}(A) = \text{First}(a_1) \cup \dots \cup \text{First}(a_k)$$

ESEMPIO SLIDE 8

10.3.2 - Calcolo del Follow(X) (Con $X \in NT$)

- Per ogni $X \in NT$, inizializza $\text{Follow}(X) := \emptyset$
- $\text{Follow}(S) := \{\$\}$
- Ripeti il seguente ciclo finché nessun $\text{Follow}(X)$ viene più modificato in una iterazione:
 - 1) Per ogni produzione $X \rightarrow \alpha y \beta$
 $\text{Follow}(y) := \text{Follow}(y) \cup (\text{First}(\beta) \setminus \{\varepsilon\})$
 - 2) Per ogni produzione $X \rightarrow \alpha y$ e per ogni produzione $X \rightarrow \alpha y \beta$ con $\varepsilon \in \text{First}(\beta)$
 $\text{Follow}(y) := \text{Follow}(y) \cup \text{Follow}(X)$

In pratica, bisogna cercare tutte le produzioni in cui $y \in NT$ appare e, per ognuna di esse, applicare la 1 o la 2 sopra.

Concettualmente il primo lo applico sempre, il secondo solo se il caso è giusto. Non è un se mutualmente esclusivo, è "sincrono".

Per esempio nell'esempio che c'è a slide 10:

$E \rightarrow TE'$, dovrei considerare sia $y = T$ che $y = E'$ con il passaggio 1, e poi dovrei $y = T$ e $y = E'$ per il passaggio 2. Nel secondo passaggio, dovrei considerare il primo caso e poi il secondo, rispettivamente, per T e E'.

10.3.3 - NOTA

Il **first** viene calcolato sia dei terminali che dei non terminali.

Il **follow** viene calcolato solo dei nonterminali.

10.1 Tabella di parsing LL(1)

sabato 3 dicembre 2022 17:22

10.1.1 - Tabella di parsing LL(1)

Strumento per risolvere il nondeterminismo.

$L \rightarrow$ input left-to-right

$L \rightarrow$ derivazione leftmost

(1) \rightarrow simbolo di look-ahead

Matrice bidimensionale M

- Righe: nonterminali
- Colonne: terminali (più \$)
- Casella (A, a) : $M[A, a]$ contiene le produzioni che possono essere scelte dal parser mentre tenta di espandere A e l'input corrente è a.

Se ogni casella contiene al più una produzione, allora il parser è **deterministico!**

Come si riempie la tabella?

Per ogni produzione $A \rightarrow a$

- 1) Per ogni $x \in T$ e $x \in First(a)$, inserisco $A \rightarrow a$ nella casella $M[A, x]$
- 2) Se $\varepsilon \in First(a)$, inserisco $A \rightarrow a$ in tutte le caselle $M[A, x]$ per $x \in Follow(A)$
(x può essere \$)

Ogni casella vuota, dopo aver elaborato tutte le produzioni, è un errore (cioè la funzione ricorsiva chiama "fail").

10.1.2 - DEFINIZIONE di grammatica LL(1)

Una grammatica è LL(1) \Leftrightarrow ogni casella della tabella di parsing LL(1) contiene al più una produzione.

10.1.3 - DEFINIZIONE di parser "predittivo" deterministico

Se G è LL(1), allora il parser ricostruisce l'albero di derivazione, per l'input w, in modo top-down, predicendo quale produzione usare (tra le molte possibili) guardando il prossimo carattere dell'input.

10.1.4 - TEOREMA

G è LL(1) \Leftrightarrow per ogni coppia di produzioni distinte con la stessa testa $A \rightarrow \alpha \mid \beta$ si ha che:

- 1) $First(\alpha) \cap First(\beta) = \emptyset$
- 2) Se $\varepsilon \in First(\alpha)$, allora $First(\beta) \cap Follow(A) = \emptyset$
Se $\varepsilon \in First(\beta)$, allora $First(\alpha) \cap Follow(A) = \emptyset$

DIMOSTRAZIONE

Se sono soddisfatte le condizioni 1) e 2) per ogni coppia di produzioni distinte con medesima testa, allora la tabella di parsing LL(1) contiene al più una produzione in ogni

casella.

Ma vale anche il viceversa!

SEGUONO DIVERSI ESEMPI, FINO A SLIDE 19

SLIDE 15: Parser LL(1) non ricorsivo usando esplicitamente una pila

10.1.5 - TEOREMA

Ogni linguaggio regolare è generabile da una grammatica G di classe LL(1).

DIMOSTRAZIONE

Se L è regolare, allora $\exists \text{DFA } M = (Q, \Sigma, \delta, q_0, F)$ tale che $L = L[M]$

A partire da M, costruiamo la grammatica regolare $G = (NT, T, S, R)$ con $NT = \{[q] \mid q \in Q\}$, $T = \Sigma$, $S = [q_0]$ e R definita da:

- Se $\delta(q, a) = q'$, allora $[q] \rightarrow a[q'] \in R$
- Se $q \in F$, allora $[q] \rightarrow \epsilon \in R$

(Seconda tecnica per trasformare un DFA in grammatica regolare)

$\Rightarrow G$ è LL(1)

Infatti, poiché M è deterministico, da ogni $q \in Q$ per ogni $a \in \Sigma$ $\exists! q' . q \xrightarrow{a} q'$, cioè $[q]$ avrà una sola produzione $[q] \rightarrow a[q']$ che inizia per "a".

Inoltre, se q è finale, allora $[q] \rightarrow \epsilon$ è applicabile solo per i $\text{Follow}([q]) = \{\$\}$ \Rightarrow nessun conflitto nel riempimento della tabella di parsing, perché nessuna produzione genera \$.

10.2 Grammatiche LL(K)

sabato 3 dicembre 2022 18:19

10.2.1 - First e Follow k

$First_k(a)$

$$w \in First_k(a) \Leftrightarrow \begin{array}{c} a \Rightarrow^* w\beta \text{ con } |w| = k, w \in T^*, \beta \in (T \cup NT)^* \\ \text{oppure} \\ a \Rightarrow^* w \text{ con } |w| \leq k, w \in T^* \end{array}$$

$Follow_k(A)$

$$w \in Follow_k(A) \Leftrightarrow \begin{array}{c} S \Rightarrow^* aAw\beta \text{ con } |w| = k, w \in T^*, a, \beta \in (T \cup NT)^* \\ \text{oppure} \\ a \Rightarrow^* aAw \text{ con } |w| \leq k, w \in T^* \end{array}$$

Tabella di parsing LL(K):

- Righe: nonterminali
- Colonne: $\{w \in T^* \mid |w| \leq k\}$ - (Solo quelli necessari)

- Per ogni produzione $A \rightarrow a, M[A, w]$ contiene $A \rightarrow a$, per ogni $w \in First_k(a)$ ($w \neq \varepsilon$) e per ogni $w \in Follow_k(A)$ se $\varepsilon \in First_k(a)$
- Se ogni entrata/casella contiene al più una produzione, allora G è LL(K)

N.B.

Le colonne sono tante quanti i w che appartengono a $First_k(a)$ per $A \rightarrow a$ oppure a $Follow_k(A)$ per $A \rightarrow a$. (se $\varepsilon \in First_k(a)$) E questo va fatto per tutte le produzioni.

10.2.2 - TEOREMA

- 1) Una grammatica ricorsiva sinistra non è LL(K) per nessuno K.
- 2) Una grammatica ambigua non è LL(K) per nessun K
- 3) Se G è LL(K) per qualche K, allora G non è ambigua
- 4) Se G è LL(K), allora L(G) è libero deterministico
- 5) Esiste L libero deterministico tale che non esiste G di classe LL(K) - per nessun k - tale che $L=L(G)$

10.2.3 - DEFINIZIONE

Un linguaggio L è di classe LL(K) se esiste G di classe LL(K) tale che $L=L(G)$

10.2.4 - PROPOSIZIONE

Per ogni $k \geq 0$, la classe dei linguaggi $LL(K+1)$ contiene strettamente la classe dei linguaggi $LL(K)$.

$$LL(0) \subset LL(1) \subset LL(2) \subset \dots \subset LL(K)$$

G è $LL(0)$ se ogni $A \in NT$ ha una sola produzione $\Rightarrow L(G)=\{w\}$ (una sola parola al massimo)

In pratica si usa solo $LL(1)$!

Se G non è $LL(1)$, spesso la si può manipolare (fattorizzare, eliminare ricorsione sx, disambiguare, etc...) trasformandola in una equivalente $LL(1)$.

SEGUONO ALTRI ESEMPI

sabato 3 dicembre 2022 18:58

11.1 Parser bottom-up

sabato 3 dicembre 2022 19:00

11.1.1 - OPERAZIONI BASE

Vi sono due operazioni base che dobbiamo definire, in modo da comprenderle meglio dopo. Esse sono il **shift** e il **reduce**.

- **SHIFT:**

Un simbolo terminale viene spostato dall'input sulla pila

- **REDUCE:**

Una serie di simboli (terminali e nonterminali) sulla cima della pila corrisponde al "reverse" della parte destra di una produzione, ovvero $A \rightarrow \alpha \in R \quad - \quad \alpha^R$ sulla pila

La stringa α^R viene rimossa dalla pila e sostituita con A. (" α viene ridotta ad A")

Noi andiamo a creare un parser shift-reduce nondeterministico, che è essenzialmente un **PDA con un solo stato che riconosce per pila vuota** il linguaggio $L \cdot \$$.

Parser LR:

- L: leggo da sinistra a destra
- R: derivazione rightmost

11.1.2 - Parser SHIFT-REDUCE NONDETERMINISTICO

- **INPUT:**

- o Una grammatica G con simbolo iniziale S
- o Una stringa $w \in T^*$

- **OUTPUT:**

- o Se $w \in L(G)$, allora resistuisco la sua derivazione rightmost a rovescio

ALGORITMO DI CREAZIONE

- Inizializziamo la pila a \$
- Inizializziamo l'input con w\$
- Usiamo il PDA seguente per trovare la derivazione per w\$

$$M = (T \cup \{\$\}, \{q\}, T \cup NT \cup \{\$\}, \delta, \$, \emptyset)$$

Quindi:

- o Simboli letti dall'input sono i terminali (l'unione con il dollaro l'ho aggiunta io perché ha senso, mi sa che sulle slide sia sbagliato)
- o Gli stati sono solo 1
- o I simboli della pila sono i terminali, i nonterminali e \$
- o Abbiamo delle transizioni δ che sono definite nel modo seguente:
 - $(q, aX) \in \delta(q, a, X), \quad \forall a \in T, \quad \forall X \in \Gamma, \quad (\text{SHIFT})$
 - $(q, A) \in \delta(q, \varepsilon, \alpha^R), \quad \forall A \rightarrow \alpha \in R, \quad (\text{REDUCE})$

- $(q, \varepsilon) \in \delta(q, \$, \$\$)$, (ACCEPT)
 - La pila inizia con \$
 - Non ci sono stati terminali (in quanto è un PDA che riconosce per pila vuota)
- Ogni volta che faccio "Reduce" forniamo in output la produzione usata
- Alla fine, \$S\$ sulla pila, con \$ in input \Rightarrow OK, accettiamo la stringa

OSSERVAZIONE

È una **generalizzazione del PDA** "tradizionale" in quanto "reduce" e "accept" **consumano più di 1 solo simbolo della pila**, bensì consuma una serie di caratteri contigui a cominciare dalla cima.

ESEMPIO A SLIDE 3 E 4.

Sia sul funzionamento, con simulazione della pila, sia con gli output che genera.

MA CI SONO CONFLITTI

- Shift-reduce
 - Scegliere di fare uno shift piuttosto che un reduce
- Reduce-Reduce
 - Scegliere di fare un reduce piuttosto che un altro

11.2 Parser LR

martedì 6 dicembre 2022 22:58

11.2.1 - Come è fatto?

Una configurazione ("foto" durante il funzionamento) di un parser LR è:

$$(\underbrace{s_0 \dots s_n}_{\substack{\text{stack degli} \\ \text{stati}}}, \underbrace{x_1 \dots x_m}_{\substack{\text{stack dei} \\ \text{simboli}}}, \underbrace{a_i \dots a_k \$}_{\substack{\text{resto} \\ \text{dell'input}}})$$

Essenzialmente, il parser legge l'input uno ad uno. Mentre lo fa, consulta una *tabella di parsing* che viene generata a partire da un DFA (automa canonico) dei prefissi viabili.

Man mano che legge l'input, si salva i simboli che ha letto + quelli che ha "compreso" (che ha de-trasformato usando le produzioni, ovvero se ho tipo $A \rightarrow a$, allora "de-trasformo" a in A).

In concomitanza, usando lo stato attuale (ogni volta che cambia stato lo mette all'interno della pila degli stati), e l'input che sta per leggere, capisce cosa fare (confrontando la tabella di parsing).

Consumato tutto l'input, se è giusto, lo "accetta".

NOTA

$x_1 \dots x_m a_i \dots a_k$ è una stringa intermedia della derivazione canonica destra.

(Ha già sviluppato tutti gli a , sta sviluppando quelli che si trovano a sinistra, ovvero le X , a partire dalla più destra (che è quella che stiamo "comprendendo" in quel momento))

11.2.2 - MOSSE DEL PARSER LR

- 1) Prima legge: $\underbrace{[stato top]}_{s_n}$ e $\underbrace{[simbolo corrente dell'input]}_{a_i}$
- 2) Consulta la tabella di parsing LR $M[s_n, a_i]$ (y,x come in programmazione)
 - Se $M[s_n, a_i] = shift \underline{s}$
Allora la nuova configurazione è:
 $(s_0 \dots \underline{s_n}, x_1 \dots x_m \underline{a_i}, a_{i+1} \dots a_k)$

SPIEGATO:

Quando leggo un input a_i e sono nella configurazione attuale s_n vado a controllare la tabella nelle coordinate determinate da questa coppia.

Se all'interno di questa cella della tabella di parsing ho SN ($S1, S2, \dots$) allora devo fare un SHIFT con un numero "s".

Questo numero lo spingo in cima alla pila degli stati.

Inoltre, aggiungo l'input letto in cima alla stack dei simboli.

E infine, lo tolgo dalla coda "resto dell'input".

SPIEGAZIONE INTUIZIONE:

Ho, concettualmente, cambiato la mia "conoscenza" del sistema (adesso so che ho letto a, oppure che ho letto b, oppure) oppure (, etc.) e questa nuova conoscenza mi fa lavorare in modo diverso per approcciare le successive lettere (quindi cambio stato).

Questo cambio di conoscenza si basa però sia su quello che ho appena letto, che su *tutto* quello che ho letto in precedenza.

Solo da questa definizione non c'è nulla che ci specifica un funzionamento in base al look-ahead.

- Se $M[s_n, a_i] = \text{reduce } A \rightarrow \beta$

Allora la nuova configurazione è:

$(s_0 \dots s_{n-r} \underline{s}, x_1 \dots x_{m-r} A, a_i \dots a_k \$)$

Dove $r = |\beta|$ e $M[s_{n-r}, A] = \text{goto } \underline{s}$

Cioè fa tre passi:

1. Faccio "pop" di r elementi dei 2 stack
2. Metto A in cima alla pila dei simboli
3. Calcolo il nuovo stato top, guardando $M[s_{n-r}, A] = \text{goto } \underline{s}$
E metto \underline{s} in cima alla pila degli stati.

DA NOTARE

Non sto consumando l'input.

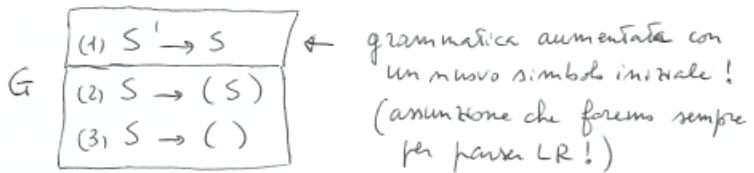
Sto "lavorando" sulla stack dei simboli che abbiamo.

- Se $M[s_n, a_i] = \text{accept} \Rightarrow \text{FINE!}$
- Se $M[s_n, a_i] = \text{"bianco"} \Rightarrow \text{errore!}$

ESEMPIO DELLE SLIDES:

Un esempio

(8)

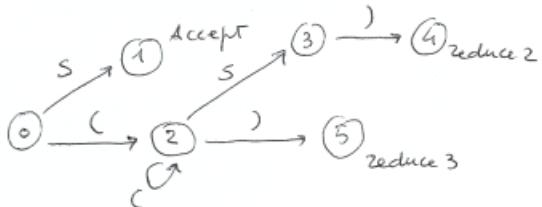


generata a partire
dal DFA → Tabella di Parsing LR(0)
dei prefissi
viablei ← non usa look-ahead!

State	Azione			Goto
	()	\$	
0	s_2			g_1
1			Accept	
2	s_2	s_5		g_3
3		s_4		
4	r_2	r_2	r_2	
5	r_3	r_3	r_3	

 $s_2 = \text{shift}$
 state_2 $r_3 = \text{reduce}$
 $S \rightarrow ()$ $g_1 = \text{goto } 1$

stack stat:	input	Azione	Output
$(0, \epsilon, (()) \$)$ stack simboli		s_2	-
$(0_2, (, () \$)$	s_2	-	
$(0_2_2, (,) \$)$	s_5	-	
$(0_2_2_5, ((),) \$)$ ↓ goto 3	r_3	$S \rightarrow ()$	$(())$
$(0_2_3, (S,) \$)$	s_4	-	(S)
$(0_2_3_4, (S, \$))$ ↓ goto 1	r_2	$S \rightarrow (S)$	S
$(0_1, S, \$)$	Accept		

DFA dei prefissi viablei / Automa canonico LR(0) (9)

11.3 Prefissi viabile

mercoledì 7 dicembre 2022 00:53

11.3.1 - DEFINIZIONE (1)

Un prefisso viabile è una stringa $\in (T \cup NT)^*$ che può apparire sulla pila di un parser bottom-up per una computazione che accetta un input.

In pratica c'è bisogno che la parte che sta in cima alla pila sia un prefisso di una parte destra di una produzione, in quanto se non lo fosse noi non potremmo arrivare a fare la "reduce".

11.3.2 - DEFINIZIONE (2) (Su grammatica G libera)

Una stringa $\gamma \in (T \cup NT)^*$ è un prefisso viabile per G

\Leftrightarrow

esiste una derivazione rightmost

$$S \Rightarrow^* \delta A y \Rightarrow \delta \alpha \beta y = \gamma \beta y$$

Per qualche $y \in T^*$, $\delta \in (T \cup NT)^*$ e per una produzione $A \rightarrow \alpha \beta$.

Inoltre S è un prefisso viabile per definizione.

Un prefisso viabile è **completo** se $\beta = \varepsilon$.

In tal caso α è detto **maniglia (handle)** per γy , ovvero in cima alla pila trovo α^R e posso fare una reduce.

SPIEGAZIONE (mia)

Data una grammatica libera G noi definiamo le stringhe γ (Da notare che possono contenere sia terminali che non terminali) come prefissi viabili

SSE

Possiamo derivare in modo rightmost, a partire da S una stringa del tipo:

δ che può essere terminali e non terminali (quindi prima della nostra produzione A, noi potremmo aver generato cose e ricordo che per la derivazione rightmost, la parte che espandiamo è sempre la destra, quindi tendenzialmente δ ce la possiamo immaginare come un insieme di non terminali (ma anche terminali, ci possono essere tante produzioni diverse))
Poi, dopo quello che abbiamo generato sulla destra, con una produzione a partire da A generiamo due parti. La parte di sinistra di questa produzione sarà α .

Da notare che in una produzione per esempio $A \rightarrow ab$

Abbiamo sia a che è un prefisso viabile

Che ab , il quale è anche una maniglia (in quanto è completo)

Da notare che dovremmo anche aggiungere il δ .

Per esempio, se fosse $S \rightarrow aA \mid A$

Allora il avremmo 4 prefissi viabili

Ovvero aa, aab (handle), a, ab (handle)

Data una grammatica, quindi, i prefissi viabili sono tutte le cose che puoi trovare

COME FACCIAMO A:

- 1) Riconoscere le maniglie sulla cima della pila e ridurle?
 - a. Concettualmente, notare che solo le maniglie dobbiamo / possiamo ridurre
 - b. Se non sono maniglie, quello che ci aiutano a fare è riconoscere a cosa ci stiamo avvicinando.
- 2) Scegliere, tra più riduzioni, solo quelle che producono sulla pila un nuovo prefisso viabile?
- 3) Scegliere spostamenti che completino i prefissi viabili sulla pila e facciano comparire una nuova maniglia?
 - a. Come detto negli altri due punti sopra, noi lavoriamo per modificare i prefissi viabili in modo da ottenere maniglie, ovvero le uniche che ci interessa ridurre.

11.3.3 - TEOREMA

Data G libera, i prefissi viabili di G costituiscono un linguaggio regolare e può essere descritto con un DFA.

DIMOSTRAZIONE

\Rightarrow Il parser (cioè un PDA) può consultare il DFA dei prefissi viabili (ovvero la tabella di parsing) per decidere cosa fare

- Se la pila contiene un prefisso viabile completo, allora il parser riduce
- Se la pila contiene un prefisso viabile incompleto, allora il parser shifta
- Se la pila non contiene un prefisso viabile, allora errore

A seconda di come è fatto il DFA, il parser può risultare deterministico o meno, eventualmente sfruttando informazioni di look-ahead e sui follow dei nonterminali per ottenere determinismo.

11.3.4 - Operazioni sulla pila (conoscendo i prefissi viabili)

Anche se, concettualmente, ad ogni modifica della pila il DFA la riscandidisce da capo per determinare come sia fatto il prefisso viabile corrente, questo non è necessario perché *ogni prefisso di un prefisso viabile è un prefisso viabile!*

SPIEGAZIONE (mia)

Cioè, noi dobbiamo capire cosa stiamo leggendo ad ogni passaggio. Per leggerlo, dovremmo ripercorrere ogni volta da capo il DFA della tabella di parsing (così capiamo con quale stringa attuale stiamo ragionando).

Ma farlo ripartire da capo è inutile, in quanto i prefissi dei prefissi viabili sono anch'essi prefissi viabili: ripercorrere l'albero non serve in quanto ci basta modificare gli stati "dinamicamente" in base allo stato nel quale siamo.

Esempio: se ci troviamo nello stato 3, non dobbiamo ripercorrere l'albero quando leggiamo "a", ci basta semplicemente fare la mossa "a" a partire dallo stato 3, in quanto questi due passaggi sarebbero equivalenti, in quanto se dovesse essere un prefisso prima, lo sarebbe anche adesso.

Quindi noi modifichiamo la pila degli stati del DFA nei quali ci troviamo in 2 modi:

1) SHIFT

La pila passa da $\$y$ a $\$yx$. In tal caso il DFA si trova nello stato S dopo aver elaborato $\$y \Rightarrow$ basta far ripartire il DFA da S con input X.

2) REDUCE $A \rightarrow \alpha$

La pila passa da $\$ya$ con $\$yA$. In tal caso il DFA si trova nello stato S dopo aver elaborato $\$y\alpha$, non c'è bisogno di far ripartire il DFA dalla base della pila, basta ripristinarne lo stato in cui si trovava subito prima di elaborare il primo simbolo di α e fornirgli il simbolo A in input.

⇒ Mi serve lo stack degli stati del DFA!

(Abbiamo finalmente trovato l'utilità degli stati del DFA che abbiamo mostrato nell'esempio iniziale)

(Per completezza, il professore ci specifica che lo stack degli stati è l'unica cosa che ci serve, lo stack dei simboli è superfluo ma lo utilizziamo solo per ragioni didattiche. E giustamente, se ci si pensa, lo usiamo solo per trascrivere man mano cosa abbiamo appena letto e mostrarcene graficamente che ha senso quello che stiamo facendo).

11.4 Item LR(0) e NFA dei prefissi viabili

giovedì 8 dicembre 2022 12:37

11.4.1 - ITEM LR(0) DEFINIZIONE

Uno stato del DFA dei prefissi viabili (chiamato **automa canonico LR(0)**) è costituito da un insieme di **item LR(0)**.

ITEM LR(0)

È una produzione con indicata, con un punto, una posizione della sua parte destra.

ESEMPIO

$A \rightarrow xyz$ genera 4 item:

$$\begin{aligned} A &\rightarrow .xyz \\ A &\rightarrow x.yz \\ A &\rightarrow xy.z \\ A &\rightarrow xyz. \end{aligned}$$

Il punto " ." indica quanta parte della produzione è già stata analizzata

COSA VUOL DIRE IL PUNTO NEL DFA?

- Se $A \rightarrow \alpha. \beta$
è nello stato del DFA in cima alla pila, allora vuol dire che α è sulla pila dei simboli che ci si aspetta che l'input da leggere contenga (o possa venir ridotto a) β
- Se $A \rightarrow \alpha$.
è nello stato del DFA in cima alla pila, allora sulla pila dei simboli c'è la maniglia α e possiamo fare la reduce.

11.4.2 - COME costruire l'NFA dei prefissi viabili?

Data $G = (NT, T, S, R)$ libera, prendiamo la grammatica aumentata con un nuovo simbolo iniziale S' ed una produzione $S' \rightarrow S$.

L'NFA non è l'ultimo dei passaggi, noi vogliamo ottenere un DFA. Tuttavia è comunque il primo passaggio da fare. Da notare che il DFA che otteniamo non è davvero un DFA, ma semplicemente non presenta le ε -transition.

L'NFA dei prefissi viabili di G si ottiene nel seguente modo:

- $[S' \rightarrow .S]$ è lo stato iniziale
- Dallo stato $[A \rightarrow a.X\beta]$ c'è una transizione allo stato $[A \rightarrow aX.\beta]$ etichettata X , per $X \in T \cup NT$
- Dallo stato $[A \rightarrow \alpha.X\beta]$, per $X \in NT$ e per ogni produzione $X \rightarrow \gamma$, c'è una ε – transition verso lo stato $[X \rightarrow .\gamma]$

OSSERVAZIONE:

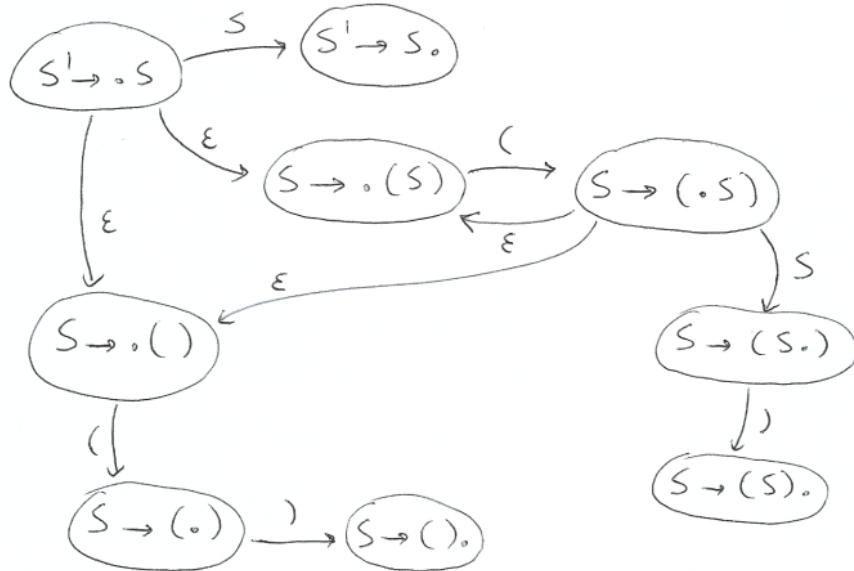
Non serve definire degli stati finali, perché l'NFA serve solo come ausilio al parser.

Non viene mai usato con l'intendo di "ho riconosciuto o no questa stringa", come i normali NFA che usavamo per le grammatiche regolari, ma serve letteralmente come strumento.

Esempio

- (1) $S' \rightarrow S$
- (2) $S \rightarrow (S)$
- (3) $S \rightarrow ()$

possibili item LR(0) per G (15)

$$\left[\begin{array}{l} S' \rightarrow .S | S_0 \\ S \rightarrow .(S) | (.S) | (S.) | (S). \\ S \rightarrow .() | (.) | () \end{array} \right]$$


11.5 Automa canonico LR(0)

giovedì 8 dicembre 2022 12:57

11.5.1 - NOTE

- È il DFA che si ottiene dall'NFA dei prefissi viabili mediante uno di due metodi:
La costruzione per sottoinsiemi (come per i normali NFA)
Oppure
In modo diretto usando le funzioni ausiliarie $Clos(I)$ e $Goto(I, X)$
Dove I è un insieme di item è $X \in T \cup NT$

11.5.2 - COSTRUZIONE

Costruzione diretta dell'Automa Canonico LR(0)

$Clos(I)$ {
 ripeti finché I è modificato {
 per ogni item $A \rightarrow \alpha \cdot X \beta \in I$
 per ogni produzione $X \rightarrow \gamma$
 aggiungi $X \rightarrow \cdot \gamma$ ad I;
 }
 return I;
}

$Goto(I, X)$ {
 inizializza $J = \emptyset$;
 per ogni item $A \rightarrow \alpha \cdot X \beta \in I$
 aggiungi $A \rightarrow \alpha X \cdot \beta$ a J;
 return $Clos(J)$;

Inizializza $S^0 = \{ Clos(\{ S^0 \rightarrow \cdot S^0 \}) \}$;

Inizializza $\delta = \emptyset$;

Ripeti finché $S^0 \cup \delta$ vengono modificati:

per ogni $I \in S^0$
per ogni item $A \rightarrow \alpha \cdot X \beta \in I$ {
 sia $J = Goto(I, X)$;
 aggiungi J a S^0 ;
 aggiungi $\delta(I, X) = J$ a δ ;

aggiungi $\mathcal{I} \cup S$;
 aggiungi $\delta(\mathcal{I}, X) = \mathcal{I} \cup \delta$;
 }
 return S' e δ .

SPIEGAZIONE ALGORITMO

(Approssimativa ma essenzialmente tutto quello che serve per fare l'esercizio al compito)

Si prende la produzione S' e si fa la closure di S' ovvero:

Per ogni nonterminale di S' che si trova dopo il punto (cioè solo S) prendo l'item 0 (con il "." all'inizio) e lo aggiungo alla closure.

Per esempio: $S' \rightarrow S, \quad S \rightarrow (S), \quad S \rightarrow ()$

In questo caso, la closure di S' è:

$S' \rightarrow .S, \quad S \rightarrow .(S), \quad S \rightarrow .()$

Poi, per ognuno dei caratteri che vengono letti spostando il punto a partire da quest'insieme di item, creo un nuovo insieme all'interno del quale faccio la closure:

$S' \rightarrow S.$ viene generato dalla lettura di S , e dopo il puntino non c'è nessun nonterminale.
Inoltre, nell'insieme delle produzioni, questa è l'unica raggiunta leggendo S . Le altre possono solo leggere "(".

Quindi questo è un altro stato del DFA (in particolare **si tratta dello stato accept**).

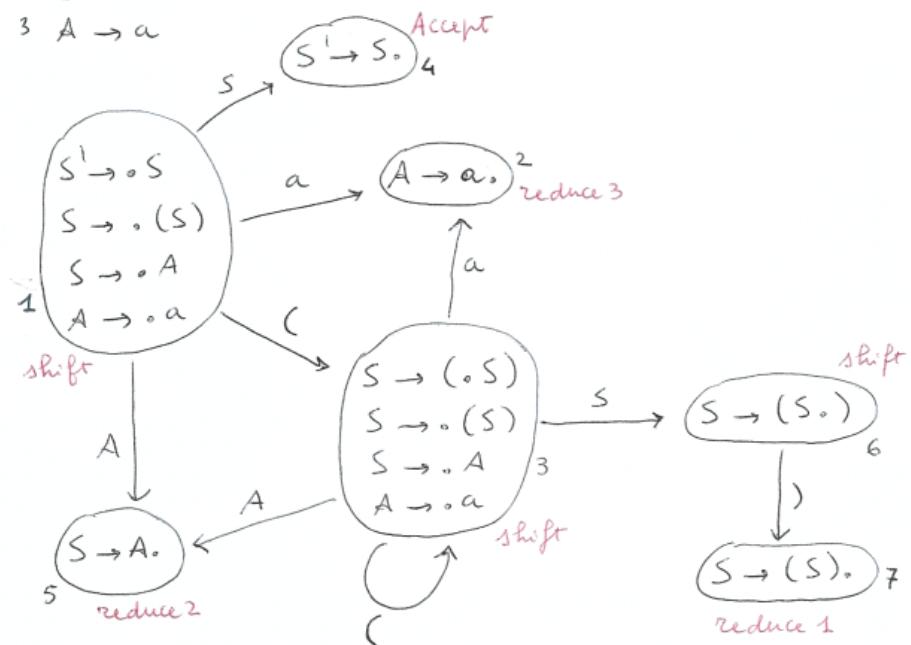
Invece se leggo "(", modifco sia il secondo che il terzo elemento dell'insieme dello stato 0.
Quindi adesso devo fare la closure di $\{S \rightarrow (.S), S \rightarrow (.)\}$. Notò che dopo il punto ho solo la S come nonterminale, quindi devo fare la closure di S . Quindi l'insieme totale è $\{S \rightarrow (.S), S \rightarrow (.), S \rightarrow (S), S \rightarrow ()\}$

Se ricapito in un insieme che ho già fatto, allora da uno stato passa allo stesso. Tipo qua, se leggo la parentesi tonda aperta, allora genero lo stesso insieme, quindi torno sullo stesso. $\xrightarrow{()}$
punta su se stesso.

ESEMPIO PER LUNGO DI QUESTA GRAMMATICA:

- 0 $S^1 \rightarrow S$
- 1 $S \rightarrow (S)$
- 2 $S \rightarrow A$
- 3 $A \rightarrow a$

$$L(G) = \{ ({}^n a)^n \mid n \geq 0 \}$$



In questo caso (ed è così sempre per grammatiche LR(0)), possiamo associare una azione ad ogni stato

- stato di shift, come 1, 3 e 6
- stato di reduce, come 2, 5, 7
- stato di accept, come 4

Ma non sarà sempre così ---

11.6 Tabella di parsing LR(0)

giovedì 8 dicembre 2022 13:16

11.6.1 - DEFINIZIONE

- Matrice bidimensionale M

- o Righe
Stati dell'automa canonico LR(0)/LR(1)

- o Colonne

$$\underbrace{T \cup \{\$\}}_{azioni} \cup \underbrace{NT}_{goto}$$

(Le colonne si mettono in questo ordine, quindi prima i terminali, poi \$ e infine i non terminali)

- $M[s, X]$ contiene le azioni che può compiere un parser LR con "s" in cima alla pila degli stati e "X" simbolo in input (o nonterminale)

- o Se $M[s, X]$ è "bianca"/vuota, allora indica **ERRORE** (ovvero è uno stato irragigungibile per una stringa in input che appartiene al linguaggio)
- o Se $M[s, X]$ contiene più azioni, allora **CONFLITTO** (Il parser non è deterministico)

11.6.2 - COME CREARLA

CASO LR(0)

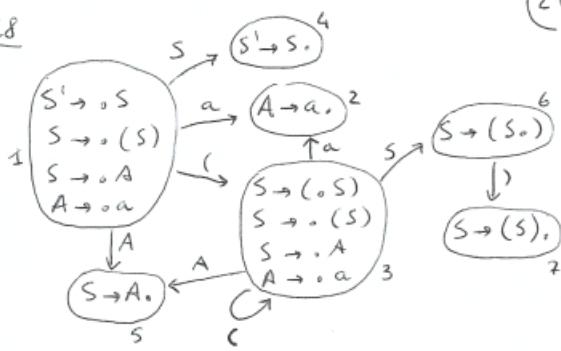
Per ogni stato S dell'automa canonico LR(0)

- Se $x \in T$ e $s \xrightarrow{x} t$ nell'automa LR(0), inserisci **shift t** in $M[s,x]$
- Se $A \rightarrow \alpha \in S$ e $A \neq S'$, inserisci **reduce A → α** in $M[s,x]$ per tutti gli $x \in T \cup \{\$\}$
- Se $S' \rightarrow S \in S$, inserisci **accept** in $M[s,\$]$
- Se $A \in NT$ e $S \xrightarrow{A} t$ nell'automa LR(0), inserisci **goto t** in $M[s,A]$

ESEMPIO:

Esempio di fg 18

- o $S^1 \rightarrow S$
- 1 $S \rightarrow (S)$
- 2 $S \rightarrow A$
- 3 $A \rightarrow a$



(21)

	a	()	\$	S	A
1	S_2	S_3			g_4	g_5
2	r_3	r_3	r_3	r_3		
3	S_2	S_3			g_6	g_5
4				acc		
5	r_2	r_2	r_2	r_2		
6			S_7			
7	r_1	r_1	r_1	r_1		

Non ci sono
completivi
 $\Rightarrow G$ e'
di classe
LR(0)

- $(1, \epsilon, ((a))\$)$ $(14, S, \$)$
Accept
 $(13, (, (a))\$)$
 $(133, ((, a))\$)$
 $(1332, ((a,))\$)$ reduce $A \rightarrow a$
 $\xrightarrow{\text{goto 5}}$
 $(1335, ((A,))\$)$ reduce $S \rightarrow A$
 $\xrightarrow{\text{goto 5}}$
 $(1336, ((S,))\$)$
 $(13367, ((S),))\$)$ reduce $S \rightarrow (S)$
 $\xrightarrow{\text{goto 6}}$
 $(136, (S,))\$)$
 $(1367, (S), \$)$ reduce $S \rightarrow (S)$

ALGORITMO (In pseudo linguaggio):

Il generico Parser LR (con solo lo stack degli stati) (2.2)

- Inizializza la pila con $\$ s_0$; % cima della pila è s_0
 s_0 è stato l'intervallo dell'automa
- Inizializza i_c con il primo carattere in input;
- while (true) {
 - $s = \text{top}(\text{pila});$ % top non rimuove la Testa
 - case $M[s, i_c]$ of
 - shift t : { push t sulla pila;
avanza i_c sull'input; }
 - accept : { output ('accept'); break; }
 - reduce A \rightarrow α : { pop $|\alpha|$ stati dalla pila;
 $s_1 = \text{Top}(\text{pila});$ % s_1 contiene $B \rightarrow \gamma, A\delta$
sia $s_2 = M[s_1, A]$ % $M[s_1, A]$ è goto s_2
push s_2 sulla pila;
output la produzione $A \rightarrow \alpha$;
}
 - else error(); % carelle bianca

12.1 Quando LR(0) - SLR(0)

giovedì 8 dicembre 2022 13:34

12.1.1 - QUANDO QUALCOSA NON È LR(0)

Non sempre siamo fortunati abbastanza da avere una grammatica LR(0), a volte possiamo incorrere in conflitti all'interno della tabella di parsing che la rende non deterministica.

Ci sono dei modi per determinare quando una grammatica non è LR(0):

1) Se G ha produzioni ϵ , allora G è difficilmente LR(0)

Banalmente, è LR(0) solo quando lo stato che contiene l'item $A \rightarrow \cdot$ non contiene un item del tipo $B \rightarrow \alpha \cdot a\beta$. Ma notiamo che in tutti gli altri casi, abbiamo un dubbio sull'operazione da fare: stiamo facendo uno shift oppure c'è una produzione di cui possiamo fare il reduce?

2) Se L è libero deterministico e gode della *prefix property* allora L è LR(0)

("L gode della prefix property se $\exists x, y \in L$ tali che x è prefisso di y ")

Se L è libero deterministico ma non è LR(0), allora L non gode della prefix property.

3) Se L è LR(0) ed è finita, allora gode della prefix property.

Se L è finito e non gode della prefix property, allora L non è LR(0)
(per esempio $L = \{a, ab\}$)

4) Se L è LR(0) ma è infinito, può non godere della prefix property.

ESEMPIO: $S \rightarrow Sa \mid a$ è LR(0) ma non gode della prefix property

12.1.2 - TABELLA DI PARSING SLR(1)

SLR(1) si intende dire che si usa un simbolo di look ahead (da qui l'1 nelle parentesi) e che è semplificata (da questo viene la S in SLR, la LR(1) è più lunga ed usa **esplicitamente** il look-ahead già nella definizione di ITEM LR(1))

- Colonne (Come LR(0))
 $T \cup \{\$\} \cup NT$
- Righe
Stati dell'automa canonico LR(0)

COME SI RIEMPE LA TABELLA?

Per ogni stato dell'automa LR(0)

- Se $x \in T$ e $s \xrightarrow{x} t$ nell'automa LR(0), inserisci **shift t** in $M[s,x]$
- Se $A \rightarrow \alpha \cdot \in S$ e $A \neq S'$, inserisci **reduce A $\rightarrow \alpha$** in $M[s,x]$ per tutti gli $x \in Follow(A)$
 - Per LR(0) era "per tutti gli $x \in T \cup \{\$\}$ "
 - Questo è quello che cambia da LR(0)
- Se $S' \rightarrow S \in S$, inserisci **accept** in $M[s,\$]$
- Se $A \in NT$ e $S \xrightarrow{A} t$ nell'automa LR(0), inserisci **goto t** in $M[s,A]$

EFFETTO: Limitare l'uso della reduce solo a casi plausibili!

12.2 LR(1)

giovedì 8 dicembre 2022 16:27

12.2.1 - CASO DI PROBLEMA CON SLR(1)

A volte semplicemente rimuovere una scelta usando il Follow() di una lettera non è abbastanza, bisogna effettivamente controllare i caratteri successivi.

Immaginiamo di avere uno stato dove:

$\begin{pmatrix} S \rightarrow V. = E \\ E \rightarrow V. \end{pmatrix}$, possiamo notare che il primo è uno shift, il secondo è un reduce, quindi c'è un conflitto: non sappiamo quale delle due operazioni scegliere.

L'esempio è preso dalle slide, dove viene presentata la grammatica:

$$G \left[\begin{array}{l} S' \rightarrow S \\ S \rightarrow V = E \mid E \\ E \rightarrow V \\ V \rightarrow x \mid *E \end{array} \right]$$

Possiamo notare che per quanto $= \in Follow(E)$, perché possiamo arrivare ad avere un stringa del tipo $*E = E$, non c'è nessuna produzione tale che $S \Rightarrow^* E = \dots$

Quindi possiamo metterci ad analizzare cosa c'è dopo la V che stiamo leggendo: se c'è $=$, allora sicuramente non si tratterà della produzione $E \rightarrow V$, mentre se non c'è, allora si tratterà di essa. Questo perché non è possibile derivare $S \Rightarrow^* E = \dots$, quindi sicuramente non è quello che voglio trovare.

Quindi ci serve indicare espressamente il look-ahead, non possiamo più farlo in modo "semplificato" per questa grammatica.

L'intuizione che c'è dietro e segnare il simbolo che si aspetta di trovare dopo la produzione che si sta cercando. Ovvero:

$[A \rightarrow \alpha. \beta, x]$:

- Sta cercando di riconoscere la maniglia $\alpha\beta$
- Di essa, α è già sulla pila dei simboli letti
- Sull'input si aspetta una stringa derivabile da βx
 - Cioè x può essere fatta in **questa** derivazione
 - Subito dopo β , inizierò a trovare la x

$[A \rightarrow \alpha., x]$:

- Fai la reduce $A \rightarrow \alpha$ se il prossimo input è proprio x

Per esempio, prima, se l'input dopo era $$$, quindi la stringa terminava, allora doveva fare la reduce. In caso contrario, doveva fare lo shift (dopo l'uguale, dopo i caratteri generati da E, si trova il terminale della stringa).

Negli item LR(1) si ha una coppia formata da:

- Un item LR(0) (detto nucleo o core)
- Un simbolo di look-ahead in $T \cup \{\$\}$

12.2.2 - Costruire LR(1)

NFA

- Stati
Item LR(1) della grammatica aumentata
- $[S' \rightarrow .S, \$]$
È lo stato iniziale
- Dallo stato $[A \rightarrow \alpha.X\beta, a]$ c'è una transizione allo stato $[A \rightarrow \alpha X. \beta, a]$ etichettata X, per $X \in T \cup NT$
- Dallo stato $[A \rightarrow \alpha.X\beta, a]$, per $X \in NT$ e per ogni produzione $X \rightarrow \gamma$, c'è una ε -transition verso lo stato $[X \rightarrow .\gamma, b]$ per ogni $b \in First(\beta a)$ (Ricordiamo che i first sono singoli terminali o $\$$)

DFA - AUTOMA CANONICO LR(1)

Si può ottenere usando la costruzione per sottoinsiemi dell'NFA.

OPPURE

In modo diretto usando $Clos(I)$ e $Goto(I, X)$ partendo dallo stato iniziale $clos([S' \rightarrow .S, \$])$ -
Questo è quello voluto dal prof durante l'esame

La costruzione diretta funziona nel seguente modo:

Parti con il closure della parte iniziale.
Usi l'algoritmo della closure per crearti l'insieme che rappresenta uno stato.
Poi, usi la goto per ogni simbolo dell'alfabeto per ottenere le funzioni di transizione.

Definizione delle funzioni:

```

Clos(I) {
    Ripeti finché I è modificato {
        Per ogni item  $[A \rightarrow \alpha.X\beta, y] \in I$ 
            Per ogni produzione  $X \rightarrow \gamma$ 
                Per ogni  $b \in First(\beta y)$ 
                    Aggiungi  $[X \rightarrow .\gamma, b]$  a I
    }
    Ritorna I;
}

```

```

Goto(I, X) {
    Inizializza  $J = \emptyset$ 
    Per ogni item  $[A \rightarrow \alpha.X\beta, y] \in I$ 
        Aggiungi  $[A \rightarrow \alpha X. \beta, y]$  a J
    Return  $Clos(J)$ 
}

```

DA NOTARE che creiamo le goto usando i simboli che possono essere possibilmente letti sopra. Dobbiamo farlo per ogni I e con ogni X possibile, ovvero ogni $T \cup NT$, e vedere cosa succede. Se il caso è vuoto, allora questo stato non si aggiunge.
 Concettualmente, se si aggiungesse sarebbe lo stato prima indicato come "bianco"/vuoto che in generale indica un caso d'errore.

12.2.3 - Come riempire la tabella di parsing LR(1)

Per ogni *stato* s dell'automa LR(1):

- 1) Se $x \in T$ e $s \xrightarrow{x} \underline{t}$ nell'automa LR(1), inserisci *shift* \underline{t} in $M[s,x]$
- 2) Se $[A \rightarrow \alpha., x] \in S$ e $A \neq S'$, inserisci *reduce* $A \rightarrow \alpha$ in $M[s,x]$ (*solo per x del look-ahead*)
- 3) Se $[S' \rightarrow S., \$] \in S$, inserisci *Accept* in $M[s,\$]$
- 4) Se $A \in NT$ e $s \xrightarrow{A} \underline{t}$ nell'automa LR(1), inserisci *goto* \underline{t} in $M[s,A]$

- Ogni casella rimasta vuota è un errore

- **DEFINIZIONE GRAMMATICA LR(1)**

Una grammatica libera G è di classe LR(1) se ogni casella della sua tabella di parsing LR(1) ha al più un elemento. (questo vuol dire che la tabella di parsing LR(1) non presenta conflitti, ovvero è deterministica)

13.1 LALR(1)

venerdì 9 dicembre 2022 19:58

13.1.1 - Come viene fatto

La tabella di parsing LALR(1) si ottiene da quella LR(1) fondendo insieme gli stati con lo stesso nucleo

- Tante righe quanti gli stati dell'automa LR(0)
- Meno "reduce" della tabella SLR(1)

Ci sono alcuni esempi, tuttavia non viene ben esplicitato in quanto non è necessario farlo.

14. Conclusioni

venerdì 9 dicembre 2022 20:08

14.1 - Ricapitolando

	LR(0)	SLR(1)	LR(1)
Shift t	$M[s, x] \text{ se } s \xrightarrow{x} t, \text{ per } x \in T$	Uguale	Uguale
Accept	$M[s, \$] \text{ se } S' \rightarrow S. \in s$	Uguale	Uguale
Reduce $A \rightarrow \alpha$	$M[s, x] \text{ se } A \rightarrow \alpha. \in S$ $\forall x \in T \cup \{\$\}$	$M[s, x] \text{ se } A \rightarrow \alpha. \in S$ $\forall x \in Follow(A)$	$M[s, x] \text{ se } [A \rightarrow \alpha., x] \in S$
Goto t	$M[s, A] \text{ se } s \xrightarrow{A} t \text{ per } A \in NT$	Uguale	Uguale

15. Altre cose sui linguaggi

domenica 18 dicembre 2022 21:53

15.1 - Definizione di un linguaggio di una certa classe

Un linguaggio L è di classe X se $\exists G$ di classe X tale che $L = L(G)$
(Dove X sta per $LR(0), SLR(1), LL(1) \dots$)

15.2 - Teoremi su cose

- Un linguaggio è libero deterministico se è accettato da un DPDA per stato finale
- Ogni linguaggio regolare è generato da una grammatica di classe $LL(1)$
- Esistono linguaggi regolari che non sono $LR(0)$ (Per esempio $L = \{a, ab\}$)

Domande 1

09 June 2023 17:39

58. Cosa si intende per analisi sintattica? Cos'è un parser?

Usando la lista di token fatta

llo scanner, il **parser produce l'albero di derivazione** del programma, riconoscendo se le frasi sono sintatticamente corrette.

- Controlla che (ad esempio)
 - o Le parentesi usate in un'espressione aritmetica sono bilanciate
 - o Che i comandi siano composti secondo le regole grammatiche
 - *if x = 5 then then x := 3*

Per realizzare un parser dobbiamo studiare:

- **Grammatiche libere da contesto**
- **Automi a pila** (pushdown automata) soprattutto nella versione **determinata** (DPDA)

Il parser è quindi una componente di un compilatore che prende in input una lista di token, prodotti dall'analizzatore lessicale, e genera un albero di derivazione relativo alla grammatica libera che viene usata.

Il parser viene generato a partire da una grammatica libera.

59. Definizione di automa a pila nondeterministico (PDA). Definizione di configurazione (o descrizione istantanea), mossa in un passo e mossa in più passi.

Un automa a pila nondeterministico (PDA) è una sestupla definita nel modo seguente:

$(\Sigma, \Gamma, Q, \delta, q_0, F)$ dove:

- $\Sigma :=$ Alfabeto dei simboli in input (che devono essere consumati, la stringa da analizzare usando il PDA)
- $\Gamma :=$ Alfabeto dei simboli della pila
- $Q :=$ Insieme degli stati
- $\delta : Q \times \Sigma \cup \{\epsilon\} \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma^*)$
 - *Quindi, la delta è una funzione di transizione che:*
 - A partire da uno stato, consuma uno o nessun simbolo in input e un simbolo sulla pila
 - E passa ad un qualsiasi numero di stati con un qualsiasi numero di nuovi caratteri sulla pila
- $q_0 \in Q :=$ Lo stato iniziale
- $F \subseteq Q :=$ Stati finali

Una descrizione istantanea di un PDA è una tripla formata da:

$q :=$ lo stato attuale

$\sigma \in \Sigma^* :=$ L'input ancora da consumare

$\gamma \in \Gamma^* :=$ L'insieme di caratteri sulla pila (Quelli più a sinistra sono in cima)

Una mossa in un passo è definita come:

$$\frac{(q', b') \in \delta(q, a, b) \quad a \in \Sigma}{(q, ax, bw) \vdash_{PDA} (q', x, b'w)} \quad oppure \quad \frac{(q', b') \in \delta(q, \varepsilon, b)}{(q, x, bw) \vdash_{PDA} (q', x, b'w)}$$

La mossa in più passi è la chiusura transitiva e riflessiva:

$$\frac{}{(q, w, \beta) \vdash_N^* (q, w, \beta)}, \quad \frac{(q, w, b) \vdash_N^* (q', w', \beta') \quad (q', w', \beta') \vdash_N^* (q'', w'', \beta'')}{(q, w, \beta) \vdash_N^* (q'', w'', \beta'')}$$

60. Definizione di linguaggio accettato da un PDA per stato finale o per pila vuota. Sono equivalenti queste due modalità di riconoscimento?

Sia N un PDA $= (\Sigma, \Gamma, Q, \delta, q_0, F)$, riconoscere per stato finale o per pila vuota vuole dire rispettivamente:

$$L[N] = \{w \in \Sigma^* : (q_0, w, \perp) \vdash_N (q, \varepsilon, x) \wedge q \in F \wedge x \in \Gamma^*\}$$

$$P[N] = \{w \in \Sigma^* : (q_0, w, \perp) \vdash_N (q, \varepsilon, \varepsilon)\}$$

Sono equivalenti in quanto è possibile costruire un PDA che riconosce lo stesso linguaggio per pila vuota a partire da uno che lo riconosce per stato finale e viceversa.

Spesso però lo stesso PDA non riconosce lo stesso linguaggio per pila vuota e per stato finale.

61. Mostrare come data una grammatica libera G , sia possibile costruire un PDA P equivalente. È possibile costruire, dato un PDA P , una grammatica equivalente G ? Concludere che la classe dei linguaggi liberi coincide con la classe dei linguaggi riconosciuti da PDA.

È possibile, ma il risultato è molto nondeterministico.

Si può costruire un top down parser del tipo:

Sia la grammatica libera $G = (NT, T, R, S)$

Allora abbiamo un parser del tipo $N = (T, \{q\}, NT \cup T, \delta, q, S, \emptyset)$

Dove vi saranno le funzioni di transizioni del seguente tipo:

Per ogni nonterminali, vi solo le produzioni con una δ del tipo:

$$\delta(q_0, \varepsilon, X) = \{(q_0, Y) : X \in NT, Y \in (NT \cup T)^*, X \rightarrow Y \in R\}$$

Per ogni terminale invece si ha:

$$\delta(q_0, a, a) = \{(q_0, \varepsilon)\} \quad \forall a \in T$$

In questo modo si crea un PDA N tal che $L = L(G) = P[N]$.

Si può anche fare il contrario, oppure partendo da PDA P si può costruire una grammatica G . Vengono forniti due lemmi per la dimostrazione, che non abbiamo approfondito, ovvero che ogni

PDA N può essere simulato in un PDA N' con un solo stato, aumentando opportunamente i simboli sulla pila e che ogni PDA con un solo stato ha un equivalente grammatica libera.

Pertanto questo ci dice che la classe dei linguaggi liberi coincide con quella dei PDA in quanto da un formalismo possiamo ridurci a trovare l'altro e viceversa.

62. Quali sono le proprietà di chiusura dei linguaggi libri? L'intersezione di un linguaggio libero con un linguaggio regolare è un linguaggio libero?

Le proprietà di chiusura dei linguaggi libri sono:

- Unione: aggiungendo $S \rightarrow S_1 | S_2$
- Concatenazione: aggiungendo $S \rightarrow S_1 S_2$
- Ripetizione: aggiungendo $S \rightarrow S S_1 | \varepsilon$

Non sono chiusi per intersezione e pertanto neanche per complementazione in quanto si potrebbe sennò usare De Morgan sull'unione.

L'intersezione con un linguaggio libero e uno regolare da sempre un linguaggio libero.

63. Intestazione e dimostrazione del "Pumping Theorem".

L'intestazione del pumping theorem è:

G è una grammatica libera \Rightarrow
 $\exists N > 0, \quad \forall z \in L(G), \quad |z| \geq N, \quad \exists u, v, w, x, y \text{ tale che:}$

- $z = uvwxy$
- $|vwx| \leq N$
- $|vx| \geq 1$
- $\forall k \in \mathbb{N}, \quad uv^k wx^k y \in L(G)$

Dimostriamo ora il pumping theorem.

Sia $G = (NT, T, R, S)$ una grammatica libera tale che $L = L(G)$.

Sia b il massimo fattore di ramificazione di un albero di derivazione (ovvero il massimo numero di simboli che compaiono nella parte destra di una produzione in R .)

$$b = \max\{|a| \mid A \rightarrow a \in R\}$$

($b \geq 2$ altrimenti la grammatica sarebbe banale)

Un albero di derivazione di altezza h (con 0 la radice) è fattore di ramificazione b , ha al più b^h foglie.

(Immaginiamo di avere per ogni produzione il fattore di ramificazione massimo.

Immaginiamo questo fattore di ramificazione massimo sia due. Allora ogni livello di altezza genera due rami. Ad altezza 0 si ha 1 nodo, altezza 1 si hanno 2 nodi, altezza 2 si hanno 4 nodi, altezza 3 si hanno 8 nodi, etc.)

Fissiamo $N = b^{|NT|+1}$ (Quindi $N > b^{|NT|}$ dato che $b \geq 2$)

Allora ogni albero di derivazione per z , con $|z| \geq N$, deve avere altezza almeno $|NT| + 1$.

Prendiamo una qualunque $z \in L$, con $|z| \geq N$.

Consideriamo il suo albero di derivazione (se ne possiede più di uno, perché G è ambigua,

prendiamo quello con minor numero di nodi).

Dunque:

- $|z| \geq N \Rightarrow$ Albero con altezza $\geq |NT| + 1$
- $\Rightarrow \exists$ cammino da radice S ad una foglia con almeno $|NT|+2$ nodi
- \Rightarrow Quel cammino attraverso $|NT|+1$ nodi intermedi etichettati con nonterminali
(La foglia è etichettata da un terminale)
- \Rightarrow Almeno un nonterminale si ripete in quel cammino

Allora si può divedere $S \Rightarrow^* z$ in:

$$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvwxy$$

*La parte dove si passa da uAy ad $uvAxy$ si può ripetere più volte.
Per esempio:*

$$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uv^2Ax^2y \Rightarrow^* uv^2wx^2y$$

Possiamo anche non ripeterla, in quanto vediamo che per la produzione di A si può passare sia, in un numero finito di passaggi, a vAx che a w .

Quindi bisogna verificare se i vincoli sono veri:

- $|vx| \geq 1$: Vero, in quanto se non lo fosse, pertanto sia v che x fossero ε allora l'albero per $k = 0$ genererebbe ancora z ma con meno nodi, in quanto lo genera per $k=1$ (e per $k \in \mathbb{N}$). Quindi questo contraddice l'ipotesi di aver scelto il più piccolo albero.
- $|vwx| \leq N$: Vero, perché per passare da A a vwx usiamo sicuramente un numero minore o uguale a $|NT|+1$ nodi, in quanto la profondità massima è $|NT|+2$ e c'è un terminale. Quindi, in quanto il massimo numero di produzioni di ogni NT è b, allora sicuramente la stringa non potrà essere più lunga di $b^{|NT|+1}$ che è proprio la nostra N. Quindi anche se ci fossero $|NT|+1$ nodi, caso peggiore, allora la stringa non potrebbe essere più lunga di N.

64. Come si può utilizzare tale teorema (a rovescio) per dimostrare che un linguaggio non è libero?

Per dimostrare $\neg L \text{ libero}$, ci possiamo ridurre a dimostrare che $\neg \text{Pumping theorem}$, ovvero:

$$\begin{aligned} L \text{ libero} &\Rightarrow \text{Proprietà} \\ \neg \text{Proprietà} &\Rightarrow \neg L \text{ libero} \end{aligned}$$

Scriviamo quindi cosa è la proprietà da dimostrare negata:

$$\begin{aligned} \forall N > 0, \quad \exists z \in L(G), \quad |z| \geq N, \quad \forall u, v, w, x, y \quad \text{se:} \\ - z &= uvwxy \\ - |vwx| &\leq N \\ - |vx| &\geq 1 \end{aligned}$$

$$\text{Allora } \exists k \in \mathbb{N}, \quad uv^kwx^ky \notin L(G)$$

Facciamo un esempio, dimostriamo che $L = \{a^n b^n c^n : n \geq 0\}$ non è un linguaggio libero.

Fissiamo una N generica. Scegliamo una stringa z del tipo:

$$z = a^N b^N c^N. \text{ Abbiamo che } |z| \geq N \text{ e che } z \in L(G)$$

Dimostriamo che, indipendentemente da come spezzettiamo la stringa, c'è sempre una k per cui questo spezzettamento non è corretto.

Dobbiamo sceglierli in tal modo che $|vwx| \leq N$ e $|vx| \geq 1$

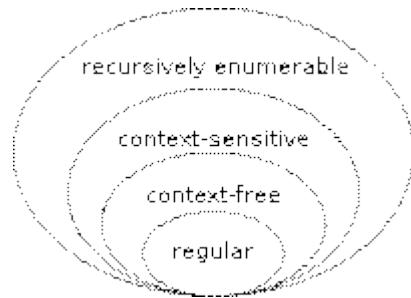
Notiamo che quello che dovremmo fare sarebbe aumentare di uno stesso numero il numero di a, b e c. Notiamo che però:

- Se la v contenesse solo a e la x contenesse solo b, si sbilancia
- Se la v contenesse solo b e la x contenesse solo c, si sbilancia
- Se sia v che c contenessero la stessa lettera, si sbilancia
- In altre combinazioni di v e x che sono formati da porzioni di lettere adiacenti nella stringa, si sbilancia
- E per qualsiasi altra combinazione, $|uwx|$ non sarebbe minore o uguale a N.

Quindi $a^N b^N c^N$ non è libero da contesto

65. Classificazione di Chomsky delle grammatiche e dei linguaggi. Definizione di grammatica dipendente da contesto e di grammatica monotona. Quale tipo di automi corrisponde ad ogni classe?

La classificazione di Chomsky permette di mostrare l'appartenenza delle varie grammatiche e di quale sia un sottoinsieme di un'altra.



Vengono divise in tipologie, ecco come vengono trascritte da Wikipedia:

Gram.	Linguaggio	Automa	Regole di produzione	Esempi
Tipo 0	Ricorsivamente numerabile	Macchina di Turing	$\gamma \rightarrow \alpha$ (γ non - empty)	$L = \{w \mid w \text{ descrive una Turing Machine che termina}\}$
Tipo 1	Dipendente da contesto	Macchina di Turing linearmente limitata	$\alpha A \beta \rightarrow \alpha \gamma \beta$	$L = \{a^n b^n c^n \mid n > 0\}$
Tipo 2	Libero da contesto	PDA	$A \rightarrow \alpha$	$L = \{a^n b^n \mid n > 0\}$
Tipo 3	Regolare	NFA/DFA	$A \rightarrow a$	$L = \{a^n \mid n \geq 0\}$

Una grammatica dipendente da contesto è una grammatica in cui le regole di produzione consentono di sostituire una variabile non terminale con una stringa di simboli che può includere sia simboli terminali che simboli non terminali. Tuttavia, ciò non impone restrizioni sulla lunghezza della stringa sostitutiva rispetto alla stringa originale.

In altre parole, la lunghezza della stringa può aumentare, diminuire, o rimanere la stessa durante la sostituzione delle regole di produzione.

Una grammatica monotona è una grammatica dipendente da contesto in cui tutte le regole di produzione sostituiscono una variabile non terminale con una stringa di simboli che ha la stessa o maggiore lunghezza della stringa originale. In altre parole, una grammatica monotona non diminuisce la lunghezza delle stringhe durante la sostituzione delle regole di produzione. Pertanto, ogni sostituzione della variabile non terminale non può abbreviare la lunghezza della stringa.

Tutte le grammatiche monotone sono grammatiche dipendenti da contesto, ma non tutte le grammatiche dipendenti da contesto sono monotone. La monotonía è una restrizione aggiuntiva imposta alle regole di produzione che garantisce che la lunghezza delle stringhe non diminuisca durante le sostituzioni.

I linguaggi monotoní e dipendenti sono equivalenti, le grammatiche dipendenti sono invece contenuto dalle monotone.

Leggendo su internet, le grammatiche dipendenti sono definite in quel modo, ma altri autori si riferiscono alle grammatiche monotone quando scrivono grammatiche dipendenti da contesto. In quanto le prime sono uguali alle seconde, allora si possono scrivere grammatiche di uno o di un altro tipo e sono equivalenti, pertanto non è davvero un problema. (Questo se sono "non-contracting", ovvero se non hanno produzioni di dimensione minore, a parte per S che può avere una produzione vuota)

SCRITTE DALLE SLIDE:

- Grammatiche regolari

$$A \rightarrow aB, \quad A \rightarrow a, \quad S \rightarrow \varepsilon$$
- Grammatiche libere da contesto

$$A \rightarrow \gamma, \quad \gamma \in (NT \cup T \setminus \{S\})^+, \quad S \rightarrow \varepsilon$$
- Grammatiche dipendenti da contesto

$$\gamma A \delta \rightarrow \gamma w \delta, \quad \gamma, \delta \in (NT \cup T \setminus \{S\})^*, \quad w \in (NT \cup T \setminus \{S\})^+, \quad S \rightarrow \varepsilon$$
- Grammatiche monotone

$$\gamma \rightarrow \delta, \quad |\gamma| \leq |\delta|$$
- Grammatiche generali

$$\gamma \rightarrow \delta, \quad \text{senza vincoli}$$

66. Definizione di DPDA (automa a pila deterministico) e di linguaggio libero deterministico.

Un DPDA è N è una 7-upla rappresentata da:

$(\Sigma, Q, \Gamma, \delta, q_0, \perp, F)$ dove ognuno è definito nel seguente modo:

- $\Sigma :=$ Insieme dei simboli in input
- $Q :=$ Insieme degli stati dell'automa
- $\Gamma :=$ Insieme dei simboli sulla pila
- $\delta :=$ funzione di transizione che fa:
$$\delta : Q \times (\Sigma \times \{\varepsilon\}) \times \Gamma \mapsto Q \times (\Gamma \cup \varepsilon)$$
Tale che, $\forall q \in Q, \forall A \in \Gamma, \forall \sigma \in \Sigma \cup \{\varepsilon\}$:
$$|\delta(q, \varepsilon, A)| \neq 0 \Rightarrow |\delta(q, \sigma, A)| = 0, \quad \forall \sigma \in \Sigma$$
$$|\delta(q, a, A)| \leq 1, \quad \forall A \in \Gamma, \quad \forall a \in \Sigma \cup \{\varepsilon\}$$

(Per ricordarmelo a parole: per ogni cosa c'è al massimo una mossa. Inoltre, se c'è una mossa epsilon per un elemento della pila, allora non vi sono mosse che consumano lo stesso elemento della pila e non sono epsilon).

- $q_0 \in Q :=$ stato iniziale
- $\perp \in \Gamma :=$ elemento iniziale della pila
- $F \subseteq Q :=$ Stati finali

Un linguaggio libero deterministico è un linguaggio riconosciuto da un DPDA per stato finale.

67. La classe dei linguaggi liberi deterministici è strettamente inclusa in quella dei linguaggi libri? Contiene strettamente la classe dei linguaggi regolari?

Sì e sì. Per la seconda, si può costruire un DPDA a partire dal DFA senza mai andare a toccare la pila.

68. Che cosa dice la prefix property e perché è interessante per i DPDA?

La prefix property ci dice che nel linguaggio L non esistono $x, y \in L$ tale che x è prefisso di y , ovvero $y = xz$.

È importante in quanto:

L gode della prefix property $\Leftrightarrow L$ riconosciuto da un DPDA per pila vuota

69. Usando un endmarker \$, si può riconoscere un linguaggio libero deterministico che non gode della prefix property anche per pila vuota? Come?

Sì, aggiungendo \$ alla fine di ogni stringa del linguaggio libero deterministico non ci saranno mai due stringhe che siano una il prefisso dell'altra, in quanto ogni stringa sarà "univoca" per l'avere \$ solo come finale, e mai in mezzo.

Pertanto potrà essere riconosciuto da un DPDA per pila vuota in quanto godrà della prefix property.

70. Un linguaggio libero deterministico è ambiguo?

No, non sono mai ambigui perché sappiamo che se L è libero deterministico, allora L è generabile da una G libera non ambigua.

71. Proprietà di chiusura dei linguaggi libri deterministici: chiusi per complementazione,

ma non per unione o intersezione.

I linguaggi sono chiusi per complementazione. In particolare, per la dimostrazione, si deve anche completare δ , aumentando N con più stati, eventualmente finali o non finali. A questo punto si scambiano finali e non finali.

Non sono chiusi per intersezione e neanche per unione in quanto si potrebbe usare De Morgan.

Esempio di intersezione per cui non sono chiusi:

$$L_1 = \{a^n b^n c^m | n, m \geq 0\} \text{ è lib det}$$

$$L_2 = \{a^n b^m c^m | n, m \geq 0\} \text{ è lib det}$$

Ma

$$L_1 \cap L_2 = \{a^n b^n c^n | n \geq 0\} \text{ non è lib}$$

72. Da cosa si parte per costruire un analizzatore sintattico (ovvero parser)? Da un'espressione regolare? Da una grammatica libera? Da un PDA?

A partire da una grammatica libera si andrà a costruire un PDA che riconosce le sue produzioni. Esistono PDA diversi, ma sono sempre PDA, per il top-down e il bottom-up parsing.

73. Cosa prende in input e cosa produce in output un parser?

Un parser prende in input una lista di token, provenienti dallo scanner, e in output ritorna un albero di derivazione.

74. Che differenza c'è tra un parser nondeterministico ed uno deterministico?

Un parser nondeterministico compie scelte che, per quanto probabili, non è completamente sicuro della loro correttezza, e pertanto fa dei "tentativi": nel caso sbagliasse, dovrebbe tornare indietro e provare in un altro modo, fino a quando prova tutti i casi oppure ne trova uno che funziona.

Un parser deterministico è completamente sicuro delle sue scelte, ognuna di esse è finale. Nel caso di errore, c'è sicuramente stato e non ha bisogno di tornare indietro per provare altre "strade".

75. Quali sono le due tecniche essenziali per costruire parser?

Sono le tecniche bottom-up e top-down.

La tecnica top-down parte dall'avere sulla pila il simbolo iniziale "S" e poi espanderlo usando le produzioni. Quando sulla pila vi sono dei terminali che coincidono con la stringa che si legge, essi vengono tolti dalla pila. Quando si incontrano di nuovo nonterminali, essi si espandono.

Vi è quindi la parte di *expand* e la parte di *consume*.

Si parte da S sulla pila e si espande man mano, ricostruendo la stringa.

Definizione : Data una grammatica libera $G = (NT, T, S, R)$, costruiamo il PDA $M = (T, \{q\}, T \cup NT, \delta, q, S, \emptyset)$ che riconosce per pila vuota (infatti non ha stati finali) e dove δ è definita così:

$$\begin{array}{ll} (q, \beta) \in \delta(q, \epsilon, A) & \text{se } A \rightarrow \beta \in R \\ (q, \epsilon) \in \delta(q, a, A) & \forall a \in T \end{array} \quad \begin{array}{l} (\text{espandi}) \\ (\text{consuma}) \end{array}$$

tale che $L(g) = P[M]$.

La tecnica bottom-up parte dall'avere sulla pila un simbolo iniziale "Z", che non fa parte dei terminali o dei nonterminali. Man mano che legge la stringa, esegue delle shift, inserendola dentro la stack. Quando vi è una stringa che fa parte di una produzione nella stack, allora viene fatto un reduce e si trova il nonterminale della produzione. Si continua questo procedimento fino al trovare il simbolo finale del dollaro, con il quale si fa un accept nel caso sia tutto corretto.

Vi è quindi *shift*, *reduce* ed *accept*.

Si parte da una pila vuota e si inseriscono i caratteri letti, ricostruendo la S

Definizione : Data una grammatica libera $G = (NT, T, R, S)$ costruiamo un PDA M che riconosce $L(G) \$$ con $M = (T, \{q\}, T \cup NT \cup \{z\}, \delta, q, z, \emptyset)$, dove la funzione δ è così definita:

$$\begin{array}{ll} (q, aX) \in \delta(q, a, X) & \forall a \in T, \forall x \in T \cup NT \cup \{z\} \\ (q, A) \in \delta(q, \epsilon, \alpha^R) & \text{se } A \rightarrow \alpha \in R \\ (q, \epsilon) \in \delta(q, \$, SZ) & \end{array} \quad \begin{array}{l} (\text{shift}) \\ (\text{reduce}) \\ (\text{accept}) \end{array}$$

76. Le tecniche top-down e bottom-up in che cosa differiscono?

Innanzitutto, l'albero di derivazione che generano:

- Top down genera una derivazione leftmost (albero di derivazione generato dall'espansione del nonterminale più a sinistra)
- Bottom up genera una derivazione rightmost (albero di derivazione generato dall'espansione del nonterminale più a destra)

Cambiano anche il tipo di grammatiche che accettano e quali determinano un più forte nondeterminismo e quali meno.

Inoltre, bottom up parte dalla stringa e genera la S, mentre top down parte da S e genera la stringa.

77. Quali tipi di grammatiche non sono adatte al top-down parsing?

Le grammatiche non adatte al top-down parsing sono quelle:

- Left recursive
- Ovvero il terminale si espande in lui stesso seguito da un altro

Le grammatiche non adatte al bottom-up parser sono quelle:

- Con produzioni epsilon
- In quanto sono sempre applicabili e rendono il parser fortemente nondeterministico

78. Cosa sono le produzioni epsilon e cosa sono i simboli non-terminali annullabili?

Le produzioni epsilon sono qualsiasi tipo di produzione:

$$A \rightarrow \epsilon \in R$$

I simboli annullabili sono non terminali tali che $A \Rightarrow^+ \epsilon$. Li indichiamo come $N(G)$, data una

grammatica G. In particolare, definiamo N(G) in modo ricorsivo:

$$N_0(G) = \{A \in NT \mid A \rightarrow \varepsilon\}$$

$$N_i(G) = N_{i-1}(G) \cup \{A \in NT \mid A \rightarrow C_1 \dots C_n \in R \wedge C_1, \dots, C_n \in N_{i-1}(G)\}$$

$\exists i_C$ tale che $N_{i_C}(G) = N_{i_C+1}(G)$ in quanto NT è finito

$$N_{i_C}(G) = N(G)$$

79. Come si può trasformare una grammatica G che contiene produzioni epsilon in una grammatica G' che non ne contiene, preservando il linguaggio a meno di epsilon?

Una volta calcolato N(G) per $G=(NT, T, S, R)$, costruiamo la grammatica $G'=(NT, T, S, R')$ dove modifichiamo le produzioni.

$\forall R$ tale che $A \rightarrow \alpha$ con $\alpha \neq \varepsilon$

In cui occorrono simboli annullabili z_1, \dots, z_k

Mettiamo in R' tutte le produzioni del tipo:

$A \rightarrow \alpha'$, dove α' si ottiene da α cancellando tutti i possibili sottoinsiemi di z_1, \dots, z_k (incluso \emptyset) **ad eccezione** del caso in cui α' risulta in ε .

- In G' non mettiamo produzioni $A \rightarrow \varepsilon \in R$
- In G' non introduciamo mai produzioni del tipo $A \rightarrow \varepsilon$

Quindi:

- Ci troviamo $N(G)$
- Modifichiamo ogni produzione dove appare un elemento di $N(G)$:
 - Consideriamo gli elementi dell'insieme potenza generato dal sottoinsieme di $N(G)$ di non terminali che compaiono nella produzione che si sta correggendo.
 - Per ognuno di questi elementi, rimuoviamo il non terminale che appare nella produzione.
 - Consideriamo le produzioni così ottenute al posto della originale, eliminando quelle che sarebbero produzioni ε

Esempio:

Esempio:

(23)

$$G = \begin{cases} S \rightarrow AB \\ A \rightarrow aAA | \epsilon \\ B \rightarrow bBB | \epsilon \end{cases}$$

$$\begin{aligned} N_0(G) &= \{A, B\} \\ N_1(G) &= \{A, B, S\} \\ &= N(G) \end{aligned}$$

Consideriamo $S \rightarrow AB \in R$. Dobbiamo considerare i casi:

$$\begin{array}{ll} \emptyset \Rightarrow S \rightarrow AB & S \rightarrow AB | A | B \\ \{B\} \Rightarrow S \rightarrow A & \\ \{A\} \Rightarrow S \rightarrow B & \nearrow \\ \{A, B\} \Rightarrow S \not\rightarrow \epsilon & \text{non la mettiamo in } R' \end{array}$$

Ora consideriamo $A \rightarrow aAA \in R$. Dobbiamo considerare i casi:

$$\begin{array}{ll} \emptyset \Rightarrow A \rightarrow aAA & \\ \{A\} \Rightarrow A \rightarrow aA & \} \text{ duplicata} \\ \{A\} \Rightarrow A \rightarrow aa & \\ \{A, A\} \Rightarrow A \rightarrow a & \Rightarrow A \rightarrow aAA | aA | a \end{array}$$

Ora consideriamo $B \rightarrow bBB$, e nello stesso modo da prima, otteniamo $B \rightarrow bBB | bB | b$

Le due ϵ -produzioni $A \rightarrow \epsilon$ e $B \rightarrow \epsilon$ sono ignorate.

$$\Rightarrow G' = \begin{cases} S \rightarrow AB | A | B \\ A \rightarrow aAA | aA | a \\ B \rightarrow bBB | bB | b \end{cases}$$

$$\underline{\text{N.B.}} \quad \epsilon \in L(G) \quad S \Rightarrow AB \Rightarrow B \Rightarrow \epsilon \\ \epsilon \notin L(G')$$

$$G'' = \left[\begin{array}{l} S' \rightarrow \epsilon | S \\ S \text{ come per } G' \end{array} \right] \quad \text{e tale che } L(G) = L(G'')$$

80. Cosa sono le produzioni unitarie e cosa sono le coppie unitarie?

Una produzione unitaria è una produzione del tipo:

$$A \rightarrow B \in R, \quad \text{tale che } A \in NT \text{ e } B \in NT$$

Un coppia unitaria è una coppia del tipo (A, B) , con $A \in NT$ e $B \in NT$ tale che:

$$A \Rightarrow^* B, \quad \text{con solo produzioni unitarie}$$

Definiamo quindi l'insieme delle produzioni unitarie $U(G)$ della grammatica G nel seguente modo:

$$U_0(G) = \{(A, A) \mid A \in NT\}$$

$$U_i(G) = U_{i-1}(G) \cup \{(A, B) \mid A, B, C \in NT \wedge (A, C) \in U_{i-1}(G) \wedge C \rightarrow B \in R\}$$

Esiste una i_C tale che:

$$U_{i_c}(G) = U_{i_c+1}(G)$$

E quindi:

$$U_{i_c}(G) = U(G)$$

- 81. Come si può trasformare una grammatica G che contiene produzioni unitarie in una equivalente G' che non ne contiene?**

Creiamo una G' tale che ogni sua produzione non presenta le produzioni unitarie. Inoltre, per ogni coppia unitaria, andiamo ad assegnare le produzioni del secondo nonterminale a quelle del primo.

- 82. Data una grammatica G , quali sono i suoi simboli utili? Quali sono i suoi simboli generatori? Quali i suoi simboli raggiungibili?**

I suoi simboli utili sono quelli che sono sia generatori che raggiungibili.
I suoi simboli generatori sono quei simboli tali che:

$$a \Rightarrow^* b \quad \text{tale che } b \in T^*, \quad a \in NT \cup T$$

I simboli raggiungibili sono invece quei simboli tali che:

$$S \Rightarrow^* aXy \quad \text{tale che } S \text{ è simbolo iniziale}, \quad a, y \in (T \cup NT)^*, \quad X \in (NT \cup T)$$

- 83. Come si calcolano i generatori? Come si calcolano i raggiungibili?**

Definiamo, come per le altre, i generatori e i raggiungibili in modo induttivo:

Generatori:

1. $G_0(G) = T$, $\quad \text{se } a \in T, a \Rightarrow^* a \quad \text{quindi tutti i terminali sono generatori}$
2. $G_{i+1}(G) = G_i(G) \cup \{B \in NT \mid B \rightarrow c_1 \dots c_k \in R \wedge c_1, \dots, c_k \in G_i(G)\}$

Raggiungibili:

1. $R_0(G) = \{S\}$
2. $R_i(G) = R_{i-1}(G) \cup \bigcup_{\substack{B \in R_{i-1}(G) \\ B \rightarrow c_1 \dots c_n \in R}} \{c_1, \dots, c_n\}$

- 84. In che modo si eliminano i simboli inutili di una grammatica? È importante l'ordine delle operazioni da svolgere?**

Vanno prima rimossi i simboli generatori e poi quelli raggiungibili.

È importante l'ordine da svolgere in quanto se si rimuovessero prima i raggiungibili e poi i generatori, rimuovendo questi ultimi potremmo avere dei nuovi simboli non raggiungibili, quindi non avremmo concluso la nostra eliminazione.

Per eliminare questi simboli:

- 1- Si eliminano tutte le produzioni che contengono simboli non generatori
- 2- Si eliminano tutte le produzioni che contengono simboli non raggiungibili

85. Quando si dice che una grammatica è ricorsiva sinistra? Come si elimina la ricorsione immediata? E quella non immediata? Perché serve eliminare la ricorsione sinistra?

Si definisce ricorsione sinistra:

$$A \Rightarrow^+ Aa, \quad A \in NT, \quad a \in (T \cup NT)^*$$

Per eliminare la ricorsione sinistra immediata bisogna fare:

Sia una ricorsione sinistra immediata definita come:

$$A \rightarrow Aa_1 \mid \dots \mid Aa_n \mid b_1 \mid \dots \mid b_k$$

Allora possiamo creare un equivalente senza ricorsione sinistra immediata usando un nuovo simbolo A' in questo modo:

$$\begin{aligned} A &\rightarrow b_1 A' \mid \dots \mid b_k A' \\ A' &\rightarrow a_1 A' \mid \dots \mid a_n A' \mid \varepsilon \end{aligned}$$

Per eliminare la ricorsione sinistra non immediata:

impostiamo un ordine nei non terminali: $\{A_1, \dots, A_n\}$

per i da 1 a n {

per j da 1 a $i - 1$ {

Modificare tutte le produzioni del tipo $A_i \rightarrow A_j a$ in:

$A_i \rightarrow ba$, dove b sono tutte le produzioni $A_j \rightarrow b$

}

Rimuovi ricorsione immediata per A_i

}

Serve eliminare la ricorsione sinistra in quanto non è possibile fare parsing top-down usando la ricorsione sinistra. Espandendo A con A di nuovo, non legge mai il simbolo che dovrebbe leggere in input come primo elemento, pertanto non consuma mai ma fa solo espansione.

86. Cosa vuol dire fattorizzare a sinistra una grammatica? Perché serve fattorizzare?

Significare raccogliere i terminali comuni che si trovano all'inizio delle produzioni, sostituendo quello che segue con un nuovo nonterminale che ripete la produzione rimanente.

Serve fattorizzare in quanto elimina molto nondeterminismo per i parser top down. Riduce di molto il look ahead.

87. Cos'è un parser a discesa ricorsiva?

È un parser molto nondeterministico che è definito dalla seguente funzione:

```
function A () {
  Scegli una delle k produzioni di A: h tra 1 e k
  A → x1h ... xnhh
```

```

 $\forall x_i^h \{$ 
    if  $x_i^h \in NT$  allora  $x_i^h()$ 
    else if  $x_i^h$  è simbolo in input allora rimuovilo dagli input
    else  $Fail()$ 
        backtracking
        fino a quando
        va bene
     $\}$ 
 $\}$ 

```

Si inizia chiamando S().

Domande 2

mercoledì 14 giugno 2023 10:47

88. Definizione di $First(\alpha)$ e di $Follow(A)$.

$First(\alpha)$ sono i $T \cup NT \cup \{\varepsilon\}$ che vengono definiti nel seguente modo:

$$\begin{aligned}\alpha \Rightarrow^* a\beta &\Rightarrow a \in First(\alpha), \quad \text{per qualche } \beta \in (NT \cup T)^* \\ \alpha \Rightarrow^* \varepsilon &\Rightarrow \varepsilon \in First(\alpha)\end{aligned}$$

I $Follow(A)$ sono definiti nel seguente modo:

$$\begin{aligned}b \in T \wedge S \Rightarrow^* xAb\gamma &\Rightarrow b \in Follow(A), \quad \text{per qualche } x, \gamma \in (T \cup NT)^* \\ S \Rightarrow^* xA &\Rightarrow \$ \in Follow(A)\end{aligned}$$

89. Algoritmi per calcolare $First(\alpha)$ e $Follow(A)$

Per calcolare i $First(\alpha)$:

$\forall t \in T, \quad First(t) = \{t\}$
 $\forall nt \in NT, \quad First(nt) = \emptyset$
determinare $N(G)$ (simboli annullabili di G)
while avvengono modifiche, do:
per ogni produzione $nt \rightarrow y_1 \dots y_k$
da $i = 1$ a k :
se $y_1, \dots, y_{i-1} \in N(G)$
allora $First(nt) = First(nt) \cup First(y_i) \setminus \{\varepsilon\}$
se $nt \rightarrow \varepsilon \in R$, allora $First(nt) = First(nt) \cup \{\varepsilon\}$

Per calcolare i $Follow(A)$

$\forall nt \in NT, \quad Follow(nt) = \emptyset$
 $Follow(S) = \{\$\}$
 $\forall A \rightarrow xB\gamma \in R \text{ add } First(\gamma) \setminus \{\varepsilon\} \text{ to } Follow(B)$
 $\forall A \rightarrow xB \in R \text{ o } \forall A \rightarrow xB\gamma \in R \wedge \varepsilon \in First(\gamma) \text{ add } Follow(A) \text{ to } Follow(B)$

90. Come è fatta e come si riempie una tabella di parsing LL(1)?

Una tabella di parsing LL(1) è una tabella che ha:

Come colonne, i simboli terminali e il dollaro
Come righe, i simboli non terminali

Come elementi della tabella le produzioni.

In particolare, nella posizione $M[X, i_c]$ ha la produzione:

$X \rightarrow Y \in R \text{ tale che } i_c \in First(Y)$

Inoltre, se $\varepsilon \in First(Y)$, allora:

$X \rightarrow Y \in R$ viene inserito in $M[X, t]$, $\forall t \in Follow(X)$

91. Quando una grammatica G si dice di classe LL(1)? Quali sono le condizioni necessarie e sufficienti per G affinché sia di classe LL(1)?

Una grammatica si dice di classe LL(1) quando si può creare una tabella di parsing LL(1) che non ha più di una produzione in ogni sua cella.

Le condizioni sufficienti e necessarie affinché G sia di classe LL(1) sono:

Si considerino coppie di produzioni a partire dallo stesso non terminale, ovvero: $A \rightarrow a|b$. G è di classe LL(1) \Leftrightarrow valgono entrambe le seguenti affermazioni:

- $First(a) \cap First(b) = \emptyset$
- $\varepsilon \in First(a) \Rightarrow Follow(a) \cap First(b) = \emptyset$
- $\varepsilon \in First(b) \Rightarrow Follow(b) \cap First(a) = \emptyset$

92. Perché un parser di questo tipo è chiamato LL?

Un parser di questo tipo è chiamato LL è in quanto:

- $L :=$ Legge la stringa in input left to right
- $L :=$ Ritorna l'albero di derivazione della derivazione sinistra della stringa in input

93. Come funziona il parser LL(1) con una pila?

Inizializza una pila con S\$.

Inizializziamo il simbolo iniziale della pila con X

Inizializziamo il simbolo iniziale in lettura con i_C

Fino a quando ($X \neq \$$)

Se $X \in NT$ allora expand:

pop di X dalla pila

push della produzione di $M[X, i_C]$ sulla pila

Se cella vuota, allora errore

set X to top della pila

Se $X = i_C$ allora consume

rimuovi X dalla pila

consuma il simbolo in input i_C

imposta X to top della pila e i_C to top della stringa in lettura

Altrimenti

Errore

Se $i_C \neq \$$, errore

94. È vero che ogni linguaggio regolare è pure LL(1)?

Sì, ogni linguaggio regolare è LL(1) in quanto si può costruire un DFA da esso, dal quale si può costruire una grammatica tale che i NT siano gli stati del DFA. Il DFA è deterministico quindi anche le transizioni lo saranno, pertanto non ci possono essere più di 1 transizione (1 produzione) con lo stesso nonterminale, pertanto l'intersezione tra i first è vuota, per ogni coppia di produzioni.

95. Come sono definiti $First_k(\alpha)$ e di $Follow_k(A)$ per $k \geq 2$?

$First_k(\alpha) = \{x \in T^* \mid |x| \leq k \wedge \alpha \Rightarrow^* x\beta, \text{ per qualche } \beta \in (T \cup NT)^*\} \cup \{\varepsilon \text{ se } \alpha \Rightarrow \varepsilon\}$

$Follow_k(A)$

$= \{x \in T^* \mid |x| \leq k \wedge S \Rightarrow^* dAx\gamma, \text{ per quale } d, \gamma \in (T \cup NT)^*\} \cup \{\$ \text{ se } S \Rightarrow^* dA \text{ per qualche } d \in (T \cup NT)^*\}$

96. Come si definisce una grammatica LL(k)? Ed un linguaggio LL(k)? Come si relazionano tra di loro?

Una grammatica LL(k) è una grammatica per la quale si può creare una tabella di parsing LL(k) tale che non vi siano più di una produzione per ogni cella.

Un linguaggio LL(k) è un linguaggio per cui è possibile avere una grammatica LL(k).

Ci possono essere grammatiche LL(k) per cui il linguaggio in realtà è LL(x) con $x < k$.

97. Che relazione esiste tra grammatiche LL(k) e grammatiche ambigue? E con le grammatiche ricorsive sinistre?

Una grammatica ambigua o una grammatica ricorsiva sinistra non è LL(k) per nessun k.

98. Esistono linguaggi liberi che non sono LL(k) per nessun k? Ed esistono linguaggi liberi deterministici che non sono LL(k) per nessun k?

Sì, possono esserci sia linguaggi liberi che linguaggi liberi deterministici che non sono LL(k) per nessun k.

Però se G è LL(k) per qualche k, allora L(G) è libero deterministico e non è ambiguo.

99. Cos'è un parser bottom-up (o shift-reduce)? Qual è il suo input e il suo output? Perché sono chiamati parser LR?

Un parser bottom-up è un parser che ricostruisce la S a partire dalla stringa, usando in modo inverso le produzioni. Compie tre operazioni

shift := legge un simbolo della stringa in input e lo mette nella pila dei simboli letti
reduce := se ha una produzione $A \rightarrow a$ e sulla pila dei simboli letti ha a^R , allora applica una reduce e sostituisce a con A sulla pila.
accept := se legge $\$$ ed ha il simbolo iniziale, sia esso S, allora accetta la stringa.

Si può andare anche a definire un parser bottom-up usando un PDA non deterministico nel seguente modo:

$$N = (\Sigma, Q, \Gamma, \delta, q_0, \perp, F) = (T \cup \{\$\}, \{q\}, T \cup NT \cup \{\$\}, \delta, q, \$, \emptyset)$$

Ed esso riconosce, per pila vuota, attraverso una funzione di transizione definita nel seguente

modo:

$$\begin{aligned}\delta(q, a, A) &= (q, aA), & \forall a \in T - \text{operazione di shift} \\ \delta(q, \varepsilon, a^R) &= (q, Y), & \forall Y \rightarrow a - \text{operazione di reduce} \\ \delta(q, \$, \$) &= (q, \varepsilon), & \text{operazione di accept}\end{aligned}$$

Notiamo che esso ha un solo stato e che effettua esattamente le operazioni descritte, anche se in questa definizione lo fa in modo nondeterministico.

Il suo input è una lista di token, mentre il suo output è un albero di derivazione rightmost.

Sono chiamati LR in quanto leggono da sinistra a destra l'input (L) e creano l'albero di derivazione rightmost.

100. Che tipo di conflitti si possono presentare in un parser del genere? Quando si presenta un conflitto (shift/reduce o reduce/reduce), quale azione bisogna scegliere?

Si possono presentare due tipi di conflitti, ovvero:

- Shift/reduce
 - o Se per esempio uno stato di un LR(0) presenta due item del tipo:
 $S \rightarrow a.$; $S \rightarrow a.x$
 - o In questo caso, vi è un conflitto nella tabella di parsing in quanto la stessa cella presenta due opzioni.
- Reduce/reduce
 - o Se non si sa quale di due reduce fare, per esempio se ci sono due item del tipo:
 $S \rightarrow a.$; $B \rightarrow a.$
 - o In questo caso, nella tabella di parsing le celle presentano due reduce possibili.

Scegliere una delle due azioni non è deterministico, in quanto ciascuna, per come è costruito una tabella di parsing, è potenzialmente corretta. Quindi bisogna cambiare la grammatica o bisogna cambiare il parser bottom up, aumentando la sua complessità a SLR(k), LR(k+1) o LALR(k).

Si può andare ad aiutare la scelta di uno dei due algoritmi analizzando i prefissi viabili e capendo quale scelta ci porta ad un prefisso viabile.

101. Cos'è un prefisso viabile? Come lo si definisce in termini di una grammatica?

Sia la grammatica $G = (NT, T, R, S)$. Un prefisso viabile per la grammatica G è:

$$\gamma \in (T \cup NT)^* : S \Rightarrow^* \delta A y \Rightarrow \delta \alpha \beta y = \gamma \beta y, \quad \text{per qualche } y, \beta \in (T \cup NT)^* \text{ e } A \rightarrow \alpha \beta \in R$$

γ è un prefisso viabile per definizione.

Se $\beta = \varepsilon$ allora γ si dice prefisso viabile completo e α è una maniglia per γy , quindi ci posso eseguire una reduce.

Essenzialmente è una stringa composta da terminali e nonterminali che si trova a destra di una possibile produzione, anche incompleta, che è prefisso di βy .

Si dice che un prefisso viabile è una stringa che può apparire sulla pila di un parser bottom up per una computazione che accetta un input, ovvero "qualcosa di corretto".

102. Cos'è un item LR(0)? Come si generano tutti gli item di una grammatica (aumentata con un simbolo iniziale nuovo S')?

Un item in LR(0) è una produzione con un delimitatore, spesso un punto, che si può interporre tra qualsiasi lettera della parte destra della produzione.

Il punto ci va ad indicare quale parte della produzione è già stata letta, alla sua sinistra, e quale deve ancora essere letta per fare la reduce, a destra.

Si creano facendo un nuovo item per ogni posizione dove si può mettere un punto nelle produzioni della grammatica.

103. Com'è fatto il NFA dei prefissi viabili? Come si ricava il DFA dei prefissi viabili, detto anche automa canonico LR(0)?

L'NFA è fatto usando il seguente algoritmo:

$$q_0 = \{S' \rightarrow .S\}$$

$$Q = \{q_0\}$$

Fino a quando non ci sono più modifiche:

Sia X uno stato con item $Y \rightarrow a.Xb$:

Se $X \in NT, \forall X \rightarrow \gamma$ si aggiunge un nuovo stato con dentro l'item X

$\rightarrow \gamma$ e $\delta(X, \epsilon) = \text{nuovo stato}$

Se $X \in T \cup NT$ si aggiunge un arco $\delta(R, X) = \text{stato con } [Y \rightarrow aX.b]$

Il DFA dei prefissi viabili può essere ottenuto a partire dall'NFA usando due modi:

- La costruzione per sottoinsiemi vista quando abbiamo visto i linguaggi regolari
- Attraverso un metodo diretto, per il quale serve definire $Clos(X)$ e $Goto(X, y)$

Definiamo quindi $Clos(X)$:

Ripeti fino a quando ci sono cambiamenti in X :

Per ogni produzione $A \rightarrow a.Yb$

Per ogni produzione $Y \rightarrow \gamma$

aggiungi $Y \rightarrow .\gamma$ come item di X

Ritorna X

Definiamo ora il $Goto(X, y)$

$$J = \emptyset$$

Per ogni item $A \rightarrow a.yb$ in X :

aggiungi $A \rightarrow ay.b$ in J

Ritorna $Clos(J)$

Per la costruzione immediata diciamo:

Definiamo lo stato iniziale $I_0 = Clos(S' \rightarrow .S)$

Definiamo l'insieme degli stati $Z = \{I_0\}$

Ripeti fino a quando cambia Z o δ

Per ogni item $A \rightarrow a.Xb$ dello stato Y

$$J = Goto(Y, X)$$

$$\delta(Y, X) = J$$

Aggiungi J a Z

104. Come è fatta una tabella di parsing LR(0)? Come la si riempie a partire dall'automa canonico LR(0)? Quando una grammatica è di classe LR(0)?

Una tabella di parsing LR(0) è formata da:

- Le colonne sono indicizzate da $T \cup NT \cup \{\$\}$
- Le righe sono indicizzate dagli stati del DFA
- Le celle vengono riempite nel seguente modo, usando il DFA:
 - o Con l'arco $s \xrightarrow{n} t$ con $n \in T$ si fa *shift* t in $M[s, n]$
 - o Con l'arco $s \xrightarrow{n} t$ con $n \in NT$ si fa *goto* t in $M[s, n]$
 - o Se in uno stato s c'è un una reduce, ovvero un item del tipo $A \rightarrow a$. Allora $\forall x \in T \cup \{\$\}$ in $M[s, x]$ si scrive *reduce* s .
 - o Se $S' \rightarrow S$ in s allora *accept* in $M[s, \$]$

Una grammatica è di classe LR(0) quando si riesce a creare una tabella di parsing con al più una operazione per ogni cella.

105. Come è fatto il parser LR(0) che utilizza la tabella di parsing LR(0)? Quanti stack servono?

Un parser LR(0) può venire descritto, in un suo stato istantaneo, da una 3-pla:

$$(s_1 \dots s_n, x_1 \dots x_k, a_1 \dots a_j)$$

Dove $s_1 \dots s_n$ sono gli stati percorsi del DFA

$x_1 \dots x_k$ sono gli elementi attualmente sulla pila

$a_1 \dots a_j$ è l'input ancora da leggere.

Esso legge dalla pila in input il prossimo input, valutando cosa fare usando lo stato attuale sulla pila degli stati e confrontandosi quindi con il DFA.

Nel caso decidesse di fare uno shift, allora sposterebbe l'input da una pila all'altra e cambierebbe lo stato.

Nel caso decidesse di fare un reduce, rimuoverebbe tanti oggetti dalla pila dei simboli attuali quanto la lunghezza della produzione e altrettanti stati dallo stack degli stati del DFA.

Nel caso facesse un accept, allora dovrebbe avere esattamente lo stato con l'accept sulla pila degli stati e $\$\$$ sulla pila dei simboli e $\$$ sulla pila in input.

All'interno di questa descrizione didattica vengono usate due pile.

Nella pratica, viene usata solo la pila contenente gli stati percorsi dal DFA, in quanto non gli serve una "visualizzazione", come a noi che la eseguiamo a mano, dello stato degli elementi letti, in quanto questo viene pressoché memorizzato da come ci stiamo muovendo nel DFA.

106. Esistono grammatiche non LR(0)? Fare un esempio semplice.

Un esempio di grammatica non LR(0) è la grammatica:

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow a \mid ab \end{aligned}$$

Vi è un conflitto shift-reduce, tra fare lo shift o ridurre quando si legge a.

107. Come è fatta e come si riempie una tabella di parsing SLR(1)? Cosa vuol dire l'acronimo SLR? Perché si mette 1 come parametro?

Si parte innanzitutto dal DFA canonico per LR(0). Si riempiono le cose come per LR(0), tutto tranne per le reduce, che vengono inserite nel seguente modo:

$$[A \rightarrow a.] \in I \Rightarrow M[I, x] \quad \forall x \in \text{Follow}(A) \text{ e } A \neq S$$

La S sta per *simple*, in quanto è un modo semplificato di aggiungere un lookahead.
Si mette 1 come parametro perché si considera un lookahead per fare le reduce.

108. Quando una grammatica è di classe SLR(1)? Esistono grammatiche non di classe SLR(1)?

Una grammatica è di classe SLR(1) quando la tabella di parsing non presenta due scelte all'interno di un'unica cella. Esistono grammatiche che non sono di classe SLR(1).

109. Cos'è un item LR(1)? Come si costruire il NFA LR(1)? E come l'automa canonico LR(1)?

Un item LR(1) è composto da due parti:

- Il nucleo, che è definito esattamente come un item di LR(0), ovvero è una produzione con un delimitatore, spesso un punto, che separa la parte letta e la parte non letta della produzione e si può trovare in qualsiasi suo punto.
- Il simbolo di lookahead, il quale ci permette di stabilire cosa si dovrebbe trovare dopo la produzione.

Per costruire un NFA LR(1) si usa il seguente algoritmo:

Si definisce lo stato iniziale come $[S' \rightarrow .S, \$]$

Si ripete la seguente operazione fino a quando non ci sono più cambiamenti:

$$\begin{aligned} & \forall X \in T \cup NT \quad \text{tali che lo stato } S \text{ sia l'item } [A \rightarrow a.Xb, y] \\ & \delta(S, X) = J \text{ dove } J \text{ ha un item } [A \rightarrow aX.b, y] \\ & \forall X \in NT \quad \text{tali che lo stato } S \text{ sia l'item } [A \rightarrow a.Xb, y] \text{ e } \forall X \rightarrow \gamma \\ & \delta(S, \varepsilon) = J \text{ dove } J \text{ ha un item } [X \rightarrow \gamma, \text{First}(b \cdot y)] \end{aligned}$$

Per costruire un DFA LR(1) si devono usare le due funzioni ausiliare $Clos(I)$ e $Goto(I, X)$

$Clos(I)$:

$$J = \emptyset$$

Esegui fino a quando cambiamenti in J

Sia $Z = [A \rightarrow a.Xb, y]$ un item di I:

Per ogni $X \rightarrow \gamma \in R$

Per ogni $\beta \in \text{First}(by)$

Add $[X \rightarrow .\gamma, \beta]$ a J

Ritorna J

$Goto(I, X)$:

$$J = \emptyset$$

$\forall Z \text{ item } = [A \rightarrow a.Xb, y] \text{ di I:}$

Aggiungi $Clos(A \rightarrow aX.b, y)$ a J

Ritorna J

Algoritmo di costruzione

$$I_0 = \text{Clos}([S' \rightarrow .S, \$])$$

$$\delta = \emptyset, \quad S = \{I_0\}$$

Finche δ e S vengono modificati:

Sia K un elemento di S

Sia $[A \rightarrow a. Yb, y]$ un item di K

Sia $L = \text{Goto}(Y, X)$

Aggiungi L a S

$$\delta(K, Y) = L$$

110. Come è fatta e come si riempie una tabella di parsing LR(1)?

Una tabella di parsing LR(1) ha:

- Le colonne indicizzate con $T \cup NT \cup \$$
- Le righe indicizzate con gli stati del DFA dei prefissi viabili (automa canonico)
- Le celle vengono riempite nel seguente modo:

- o Se $s \xrightarrow{x} t$ e $x \in NT$, allora $M[s, x] = \text{goto } t$
- o Se $s \xrightarrow{x} t$ e $x \in T$, allora $M[s, x] = \text{shift } t$
- o Se lo stato s ha l'item $[S' \rightarrow S. ., \$]$, $M[s, \$] = \text{accept}$
- o Se lo stato s , e sia n la produzione $A \rightarrow a$, ha l'item $[A \rightarrow a. , y]$,
 $M[s, y] = \text{reduce } n$

111. Come è fatto una tabella di parsing LALR(1)? Quando una grammatica è di classe LALR(1)?

Una tabella di parsing LALR(1) viene fatta nello stesso modo delle altre tabelle di parsing. LALR(1) deriva dal fondere insieme gli stati che hanno gli stessi nuclei. In questo modo, si ottengono un numero di stati uguali all'LR(0), ma si hanno molte meno cose non deterministiche.

Una grammatica è di classe LALR(1) quando non vi sono più di due produzioni per la stessa cella della tabella di parsing.

112. Esistono grammatiche LALR(1) che non sono SLR(1)? Esistono grammatiche LR(1) che non sono LALR(1)? Mostrare una grammatica che è LR(1) ma non LALR(1).

Sì, sì.

$$G = \left[\begin{array}{l} S \rightarrow aAa \\ S \rightarrow bAb \\ S \rightarrow aBb \\ S \rightarrow bBa \\ A \rightarrow c \\ B \rightarrow c \end{array} \right], \quad \text{questa qua esce LR(1) ma non LALR(1)}$$

113. Come si può generalizzare l'idea per ogni $k \geq 2$? Ovvero, quando una grammatica è di classe LR(k), SLR(k) o LALR(k)?

Per generare LR(k):

- Item $LR(k) = [item\ LR(0), \beta]$, con $|\beta| \leq k$
- Item iniziale = $[S' \rightarrow .S, \$]$
- Quando $[A \rightarrow a.X\gamma, \beta] \in S$ (Stato dell'automa canonico $LR(k)$), allora pure $[X \rightarrow .\delta, w] \in S$, se $X \rightarrow \delta$ è una produzione e $w \in First_k(\gamma\beta)$
- Colonne su T^k

Grammatiche SLR(k):

- Si parte dall'automa canonico $LR(0)$ e si riempie la tabella di parsing $SLR(k)$ che ha colonne su T^k secondo la legge:
 - o $[A \rightarrow a.] \in S$ e $A \neq S'$, inserisco "reduce $A \rightarrow a$ " in $M[S, w]$, $\forall w \in Follow_k(A)$

Grammatiche LALR(k)

- Si parte dall'automa canonico $LR(k)$ e si fondono insieme gli stati con lo stesso nucleo.

- 114. Come si relazionano le grammatiche della famiglia LR (LR,SRL,LALR) al variare di k?**
Come si relazionano le grammatiche LR(k) e LL(k) al variare di k? Le grammatiche LR(k) e LL(k) sono sempre non ambigue? Esistono grammatiche ambigue che sono LL(k) o LR(k) per qualche k? Esistono grammatiche che non sono LR(k) per nessun k?

$SLR(k) \subset LALR(k) \subset LR(k)$

$LL(k) \subset LR(k)$, $LL(k)$ non sono $\subset LR(k)$

LL e LR sono sempre non ambigue.

No, non sono mai ambigue.

Sì, quelle ambigue non sono mai LR(k) per nessun k.

- 115. Come si relazionano i linguaggi della famiglia LR(k) rispetto a quelli LL(k)? Esistono linguaggi liberi deterministici che non sono LR(k) per qualche k? Esistono linguaggi liberi deterministici che non sono LL(k) per qualche k?**

Scritto sopra.

No, tutti i linguaggi libri deterministici posso avere grammatiche $SLR(1)$.

Sì, esistono linguaggi libri deterministici che non sono $LL(k)$ per nessun k.

- 116. Esiste un linguaggio regolare che non è LR(0)? Come si relazionano i linguaggi LR(0) rispetto a quelli LL(1)? La prefix property è una condizione necessaria e sufficiente affinché un linguaggio sia di classe LR(0)?**

Sì, per esempio il linguaggio $\{a, ac\}$ non è $LR(0)$ ma è un linguaggio regolare in quanto finito.

I linguaggi $LR(0)$ hanno elementi in comune e non in comune con quelli $LL(1)$, nessuno dei due è strettamente sottoinsieme dell'altro.

$L \text{ libdet} \wedge \text{PrefixP}(L) \Rightarrow L \text{ è LR}(0)$
 $L \text{ è LR}(0) \text{ e finito} \Rightarrow L \text{ gode della prefix property}$
 $L \text{ non gode della prefix property ed è finito} \Rightarrow L \text{ non è LR}(0)$
 $L \text{ è LR}(0) \text{ e infinito} \Rightarrow L \text{ può non godere della prefix property.}$

117. La classe dei linguaggi SLR(1) coincide con la classe dei linguaggi liberi deterministici?

Sì.

118. Cos'è YACC? Quale è il suo input e qual è il suo output? Come si ottiene un parser eseguibile a partire da un file .y? Come agisce il parser generato da YACC in sintonia con lo scanner generato da Lex?

YACC è un generatore di analizzatori sintattici. Prende in input un file .y che contiene la descrizione della grammatica libera e dà in output un file C che realizza il parser LALR(1) con una costruzione diretta non vista a lezione.

Si ottiene un eseguibile dando i file .c che risultato da YACC ad un compilatore di c, indicando le varie librerie che deve usare.

L'analizzatore sintattico generato da YACC usa l'analizzatore lessicale come subroutine, chiedendoli per esempio il token successivo e gestendo alcune variabili in comune, come *yylval*, permettendo quindi scambiarsi informazioni tra di loro.

119. Come è la struttura di un file .y di YACC? Cosa sono le regole? Cos'è l'azione semantica?

Un file YACC è formato nel seguente modo:

%{Prologo%}

- Parte opzionale che contiene definizioni di macro e altre dichiarazioni di variabili o funzioni che saranno usate nelle sezioni seguenti.
- Viene copiato da YACC nell'output (*y.tab.c*) in modo da precedere la definizione della funzione "yparse" che effettivamente farà l'analizzatore sintattico

Definizioni

- Contiene dichiarazioni di simboli usati nella descrizione della grammatica (nomi di token, condivisioni con lex)
- In questa sezione è possibile dichiarare la precedenza e l'associatività di alcuni termini / operatori

%

Regole

- Una produzione della forma
 - o $\text{nonterm} \rightarrow \text{corpo}_1 \mid \dots \mid \text{corpo}_k$
- È espressa in YACC con le regole:
 - o $\text{nonterm}: \text{corpo}_1 \{ \text{azione semantica}_1 \}$
 - | ...

| $corpo_k \{azione\ semantica_k\}$

;

- Il nonterminale è una stringa alfanumerica non dichiara in precedenza come token
 - o Un carattere tra singoli apici, 'a', è un terminale
 - o Il simbolo iniziale della grammatica è il nonterminale usato nella prima regola
- Le azioni semantiche sono codice C eseguito al momento in cui il parser riduce mediante quella produzione.
 - o Calcola il "valore semantico" della testa della "produzione", in funzione dei valori semanticici dei simboli che compongono il corpo. Il "valore semantico" può essere:
 - L'albero di derivazione, nel caso in cui stiamo producendo un compilatore che genera alberi esplicitamente
 - Il codice intermedio, connesso alla produzione, se andiamo direttamente a produrre codice intermedio
 - La vera e propria valutazione dell'espressione, se stiamo producendo un interprete.
 - o Si può usare \$i per usare l'i-esimo valore trovato nella produzione.

%

Funzioni ausiliarie

- Contiene le funzioni di supporto per la generazione del parser, tra questi `yylex()`, funzione che invoca l'analizzatore lessicale e che restituisce il nome del token (che deve essere stato definito della sezione "definizioni" di YACC) e il suo valore (nella variabile `yyval`)
- Deve o contenere `yylex()` con la sua definizione
- Oppure, se viene generato con Lex, allora deve fare `#include "lex.yy.c"`

120. È possibile gestire grammatiche ambigue con YACC, specificando le associatività e le priorità fra gli operatori per risolvere l'ambiguità? Come si comporta YACC in presenza di conflitti?

Sì, queste si possono specificare all'interno delle definizioni, ovvero la parte che sussegue il prologo e precede le regole. Possiamo indicare con la parola chiave `left` o `right` l'associatività rispettivamente destra o sinistra dei nonterminali che desideriamo.

In generale, in caso manchino le indicazioni su come risolvere conflitti:

- Risolve shift/reduce a favore di shift
- Risolve reduce/reduce a favore della produzione elencata prima

Domande 3

mercoledì 14 giugno 2023 23:07

- 121. È possibile costruire un programma Check che, preso in input un qualunque programma P, restituisce 1 se P è corretto e 0 se P è scorretto? Ovvero esiste un qualche compilatore che può scovare tutti i possibili errori di un programma?**

Non si può fare.

Assumiamo per assurdo che si possa costruire un programma che controlla 1 dei parametri della correttezza di un programma, ovvero il suo terminare. Questo programma ritorna 1 se termina per ogni output e ritorna 0 altrimenti:

$$H(P, x) = \begin{cases} 1 & \text{se } P(x) \downarrow \\ 0 & \text{se } P(x) \uparrow \end{cases}$$

Supponiamo ora di creare un secondo programma che mi ritorna 0 se converge e per una x e 1 se diverge per una x:

$$K(P) = \begin{cases} 1 & \text{se } P(P) \downarrow \\ 0 & \text{se } P(P) \uparrow \end{cases}$$

Ora, diamo in pasto a F H e P:

$$G(P) = \begin{cases} 1 & \text{se } K(P) = 0 \\ \uparrow & \text{se } K(P) = 1 \end{cases}$$

Ma se noi a G diamo in input G, allora essa ritorna uno se essa diverge e diverge se essa converge. Quindi è assurdo. Quindi nulla di questo può esistere.

- 122. Cosa dice il problema della fermata (Halting problem)? (L'errore in esame è la possibilità di non terminare il calcolo). Come si dimostra che il problema non può essere risolto?**

Il Halting problem è un problema relativo al capire se è possibile capire a priori quando un programma terminerà e quando invece diverge.

Supponiamo di riuscire a creare un programma che ritorna 1 se il programma converge e ritorna 0 se il programma diverge, che è un programma che fa la funzione che noi vogliamo.

$$H(P, x) = \begin{cases} 1 & \text{se } P(x) \downarrow \\ 0 & \text{se } P(x) \uparrow \end{cases}$$

Ora, con la conoscenza di scrivere questo programma, noi possiamo andare a creare un altro programma ausiliare, ovvero:

$$F(P) = \begin{cases} 1 & \text{se } P(P) \downarrow \\ 0 & \text{se } P(P) \uparrow \end{cases}$$

Infine, possiamo creare un ultimo programma, visto che siamo riusciti a fare questi due, del tipo:

$$G(P) = \begin{cases} 1 & \text{se } F(P) = 0 \\ \uparrow & \text{se } F(P) = 1 \end{cases} \Rightarrow G(G) \uparrow \Leftrightarrow F(P) = 1 \Leftrightarrow G(G) \downarrow$$

123. Quando un problema è decidibile?

Un problema è decidibile se si può rispondere sia di vero, che di falso, in tempo finito. Ovvero la soluzione vera o falsa converge in tempo finito.

Un problema semi decidibile se si può rispondere vero in tempo finito, ma non si può rispondere a falso in tempo finito.

124. Quali sono tipici esempi di proprietà indecidibili per i linguaggi di programmazione?

Esempi di proprietà indecidibili sono:

Terminazione, ovvero ritornare vero se converge e falso se diverge

Divergenza, ovvero vero se diverge e falso se converge

Equivalenza

Calcolo di una costante, ovvero se P calcola un valore costante per ogni input

Generazione di errori *run-time*

125. Cos'è una macchina di Turing (MdT)? Che cosa calcola?

Un macchina di Turing è una 6-upla formata da:

$$(Q, A, B, q_0, q_f, \delta)$$

Q sono stati finiti

A è l'alfabeto finito in input

B è l'alfabeto finito del nastro, super insieme di quello in input, ed ha la casella vuota

q_0 è lo stato iniziale

q_f è lo stato finale

$$\delta: Q \times B \rightarrow Q \times B \times \{sx, dx\}$$

Una macchina di Turing può essere vista come una macchina che calcola funzioni parziali binarie, ovvero data una stringa, ritorna 1 se si arriva in uno stato finale, 0 se non si arriva in uno stato finale ma non si può più continuare, oppure diverge.

126. Quando un linguaggio è detto Turing-completo? Cosa afferma la tesi di Church-Turing e perché non si può dimostrare?

Un linguaggio è detto Turing completo se può calcolare tutte le funzioni che calcolabili da una macchina di Turing.

La tesi di Church-Turing afferma che:

- Se una funzione può essere calcolata algoritmicamente in un qualche formalismo, allora è calcolabile con MdT.
- Non è possibile verificarla in quanto non c'è una definizione formale di cosa voglia dire "algoritmicamente", pertanto bisognerebbe farlo per ogni formalismo, ma ci sarebbero infinite prove e chiunque potrebbe proporre nuovi formalismi da dimostrare.

127. I normali linguaggi di programmazione sono Turing completi? Cosa afferma il teorema di Jacopini-Bohm?

I normali linguaggi sono Turing completi se si assume che la memoria che usano è sufficiente per eseguire il programma / infinita in generale.

Se un linguaggio prevede almeno le 4 operazioni:

- Iterazione indeterminata o ricorsione
- *If – then – else*
- Concatenazione dei comandi
- Istruzioni di assegnamento

Allora suddetto linguaggio è Turing completo.

128. Quale relazione esiste tra espressività di un formalismo e decidibilità di proprietà dello stesso? Fare una panoramica prendendo in esame i 3 formalismi MdT, PDA, DFA e le due proprietà $w \in L(m)$ (Ovvero è w riconosciuta dalla macchina M?) e $L(M_1) = L(M_2)$ (Le due macchine sono equivalenti, ovvero riconoscono lo stesso linguaggio?)

Più si vanno a diminuire le possibilità expressive di una grammatica e più proprietà vanno ad assumere da questo punto di vista.

Per le macchine di Turing, entrambe le proprietà sono indecidibili.

Per i PDA è decidibile l'appartenenza ma non l'uguaglianza.

Per i DFA è riconoscibile sia l'appartenenza che l'uguaglianza.

PRIMO

lunedì 5 dicembre 2022 16:44

Costruire una grammatica G che generi il linguaggio $L = \{a^{n+1}b^n a^m b^{m+k} \mid n, k \geq 0, m \geq 1\}$

SOLUZIONE

$$\begin{aligned} S &:= aAB \\ A &:= aAb \mid \varepsilon \\ B &:= aB'bC \\ B' &:= aB'b \mid \varepsilon \\ C &:= bC \mid \varepsilon \end{aligned}$$

ALTRA

$$\begin{aligned} S &:= aAaBb \\ A &:= aAb \mid \varepsilon \\ B &:= aBb \mid Bb \mid \varepsilon \end{aligned}$$

Esercizio - LR(0)

lunedì 5 dicembre 2022 17:21

ESEMPIO

$$\begin{cases} S' \rightarrow S \\ S \rightarrow (S) \\ S \rightarrow A \\ A \rightarrow a \end{cases}$$

$$\begin{pmatrix} S' \rightarrow S \\ S \rightarrow (S) \\ S \rightarrow A \\ A \rightarrow a \end{pmatrix}$$

$\xrightarrow{s} (S' \rightarrow S.)$ è accept

$$\xrightarrow{\cdot} \begin{pmatrix} S \rightarrow (.S) \\ S \rightarrow (S) \\ S \rightarrow A \\ A \rightarrow a \end{pmatrix}$$

$$\xrightarrow{s} (S \rightarrow (S.))$$

$$\xrightarrow{\cdot} (S \rightarrow (S.))$$

$$\xrightarrow{A} (S \rightarrow A.)$$

$$\xrightarrow{a} (A \rightarrow a.)$$

$\xrightarrow{\cdot} \text{torna indietro}$

$$\xrightarrow{A} (S \rightarrow A.)$$

$$\xrightarrow{a} (A \rightarrow a.)$$



lunedì 12 dicembre 2022 12:10

Terzo parziale (19 maggio)

lunedì 12 dicembre 2022 12:10

1. Descrivere le regole di semantica operazionale strutturata per l'espressione booleana b_0 and b_1 , secondo la disciplina di valutazione esterna-sinistra (ES). Mostrare un esempio di una espressione di quel tipo tale che la valutazione ES e quella ED (esterna-destra) non sono uguali.
2. Costruire una grammatica G che generi il linguaggio $L = \{a^n b^{2k} c^k d^{n+1} \mid n, k \geq 0\}$.
3. Classificare il linguaggio L del punto precedente, ovvero dire se L è regolare, oppure libero ma non regolare, oppure non libero, giustificando adeguatamente la risposta.
4. Si consideri l'espressione regolare $a\epsilon(b|\emptyset)^*$. Si costruisca l'automa NFA M associato, secondo la costruzione vista a lezione. Si trasformi l'NFA M nell'equivalente DFA M' , secondo la costruzione per sottoinsiemi vista a lezione.
5. Preso il DFA M' calcolato al punto precedente, si verifichi se è minimo; se non lo fosse, lo si minimizzi per ottenere un DFA M'' ; quindi si ricavi da M'' la grammatica regolare associata, seguendo la costruzione vista a lezione; quindi si semplifichi la grammatica ottenuta, eliminando i simboli inutili; infine, si ricavi dalla grammatica semplificata l'espressione regolare associata.
6. Sia $L = \{a^n b^n \mid n \geq 1\}$ e $R = \{a^{2n} b^m \mid n, m \geq 0\}$. Sfruttando le proprietà di chiusura, si può concludere se il linguaggio $L \cap R$ è regolare, oppure libero, oppure non libero? Giustificare la risposta.
7. Mostrare che $L = \{a^n b^{n+1} c^m \mid n, m \geq 0\}$ è libero deterministico, costruendo un opportuno DPDA che riconosca L per stato finale.
8. Si consideri la seguente grammatica G con simbolo iniziale S :

$$\begin{array}{lcl} S & \rightarrow & ACB \\ A & \rightarrow & \epsilon \mid aA \\ B & \rightarrow & b \mid bB \\ C & \rightarrow & \epsilon \mid cSC \end{array}$$

- (i) Si calcolino i First e i Follow per tutti i nonterminali. (ii) La grammatica G è di classe LL(1)?
(iii) Si rimuovano le produzioni epsilon per ottenere una grammatica G' senza produzioni epsilon, che sia equivalente a G . (iv) Se la grammatica risultante presenta produzioni unitarie, si rimuovano, ottenendo una grammatica equivalente G'' .
9. Si consideri la grammatica G con simbolo iniziale S :
$$\begin{array}{lcl} S & \rightarrow & Sb \mid Sa \mid A \\ A & \rightarrow & c \mid cA \end{array}$$

(i) Determinare il linguaggio generato $L(G)$. (ii) Verificare che G non è di classe LL(1). (iii) Manipolare la grammatica per ottenerne una equivalente G' di classe LL(1). (iv) Costruire il parser LL(1) per G' . (v) Mostrare il funzionamento del parser LL(1) su input cba .

 10. Si consideri la grammatica G del punto precedente. (i) Costruire l'automa canonico LR(0). (ii) Costruire la tabella di parsing SLR(1) e verificare se ci sono conflitti. (iii) Mostrare il funzionamento del parser SLR(1) per l'input cba .

Esercizio 1.

lunedì 12 dicembre 2022 12:11

1. Descrivere le regole di semantica operazionale strutturata per l'espressione booleana $b_0 \text{ and } b_1$, secondo la disciplina di valutazione esterna-sinistra (ES). Mostrare un esempio di una espressione di quel tipo tale che la valutazione ES e quella ED (esterna-destra) non sono uguali.

SOLUZIONE

In quanto è esterna sinistra:

- o Non dobbiamo valutare tutti i membri dell'espressione, ma solo quelli utili al fine di capirla
- o Iniziamo a valutarli a partire dalla sinistra

$$\frac{\langle b_0, \sigma \rangle \vdash \langle b'_0, \sigma' \rangle}{\langle b_0 \text{ and } b_1, \sigma \rangle \vdash \langle b'_0 \text{ and } b_1, \sigma' \rangle} (\text{AND}_1)$$

$$\frac{}{\overline{\langle ff \text{ and } b_0, \sigma \rangle \vdash \langle ff, \sigma \rangle}}$$

$$\frac{}{\overline{\langle tt \text{ and } b_0, \sigma \rangle \vdash \langle b_0, \sigma \rangle}}$$

$$\frac{}{\overline{\langle tt \text{ and } \frac{10}{0} = 2, \sigma \rangle \vdash \langle tt, \sigma \rangle}}, \quad \text{Se ES, non da errore in quanto è true}$$

$$\frac{\frac{\frac{\langle \frac{10}{0}, \sigma \rangle \vdash \text{ERRORE}}{\langle \frac{10}{0} = 2, \sigma \rangle \vdash \text{ERRORE}}}{\langle tt \text{ and } \frac{10}{0} = 2, \sigma \rangle \vdash \text{ERRORE}}}{\langle tt \text{ and } \frac{10}{0} = 2, \sigma \rangle \vdash \text{ERRORE}}, \quad \begin{array}{l} \text{se ED, mi da errore in quanto trova } \frac{10}{0} \\ \text{se ES, non da errore in quanto è true} \end{array}$$

Esercizio 2.

lunedì 12 dicembre 2022 12:33

2. Costruire una grammatica G che generi il linguaggio $L = \{a^n b^{2k} c^k d^{n+1} \mid n, k \geq 0\}$.

SOLUZIONE

$$\begin{aligned} S &\rightarrow aSd \mid Bd \\ B &\rightarrow bbBc \mid \varepsilon \end{aligned}$$

Esercizio 3.

lunedì 12 dicembre 2022 12:36

3. Classificare il linguaggio L del punto precedente, ovvero dire se L è regolare, oppure libero ma non regolare, oppure non libero, giustificando adeguatamente la risposta.

SOLUZIONE

Linguaggio del punto precedente:

$$\begin{aligned} S &\rightarrow aSd \mid Bd \\ B &\rightarrow bbBc \mid \epsilon \end{aligned}$$

Innanzitutto, è libero in quanto è generato da una grammatica libera.

Per verificare che non sia regolare, uso il pumping theorem:

Fissiamo una N generica, sceglioamo una stringa $z = a^N b^2 c^1 d^{N+1}$, in quanto $|z| \geq N$
Adesso la suddividiamo in 3 sottostringhe, uvw , tali che $|uv| \leq N$, $|v| \geq 1$.

In quanto $|uv| \leq N$, allora v deve per forza essere a^j con j maggiore di 1.

Adesso, dobbiamo dimostrare che esiste un k maggiore di 0 tale che $uv^k w$ non appartiene al linguaggio, allora non è regolare.

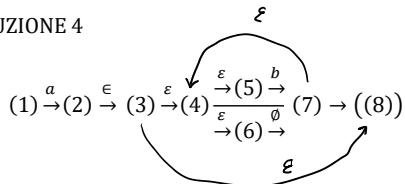
Scegliamo una k uguale a 2 e abbiamo $a^{N+j} b^2 c^1 d^{N+1} \notin L$

Esercizio 4. e 5.

lunedì 12 dicembre 2022 12:57

4. Si consideri l'espressione regolare $a\epsilon(b|\emptyset)^*$. Si costruisca l'automa NFA M associato, secondo la costruzione vista a lezione. Si trasformi l'NFA M nell'equivalente DFA M' , secondo la costruzione per sottoinsiemi vista a lezione.
5. Preso il DFA M' calcolato al punto precedente, si verifichi se è minimo; se non lo fosse, lo si minimizzi per ottenere un DFA M'' ; quindi si ricavi da M'' la grammatica regolare associata, seguendo la costruzione vista a lezione; quindi si semplifichi la grammatica ottenuta, eliminando i simboli inutili; infine, si ricavi dalla grammatica semplificata l'espressione regolare associata.

SOLUZIONE 4



Adesso che abbiamo il NFA, facciamo l'algoritmo per sottoinsiemi per determinare il DFA

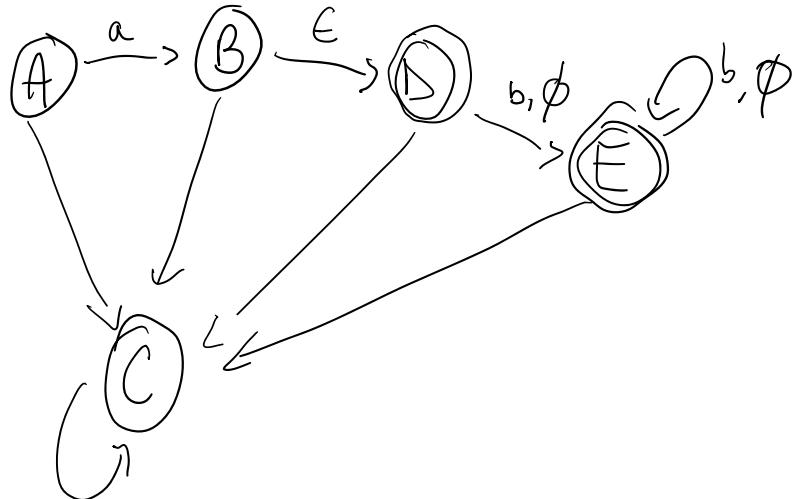
$$\begin{aligned} A &= \{1\} \\ \Delta(A, a) &= \{2\} = B \\ \Delta(A, b) &= \emptyset = C \\ \Delta(A, \epsilon) &= \emptyset = C \end{aligned}$$

$$\begin{aligned} \Delta(B, a) &= C \\ \Delta(B, b) &= C \\ \Delta(B, \epsilon) &= \{3, 4, 5, 6, 8\} = D \\ \Delta(B, \emptyset) &= C \end{aligned}$$

$$\begin{aligned} \Delta(C, a) &= C \\ \Delta(C, b) &= C \\ \Delta(C, \epsilon) &= C \\ \Delta(C, \emptyset) &= C \end{aligned}$$

$$\begin{aligned} \Delta(D, a) &= C \\ \Delta(D, b) &= \{7, 8, 4, 5, 6\} = E \\ \Delta(D, \epsilon) &= C \\ \Delta(D, \emptyset) &= \{7, 8, 4, 5, 6\} = E \end{aligned}$$

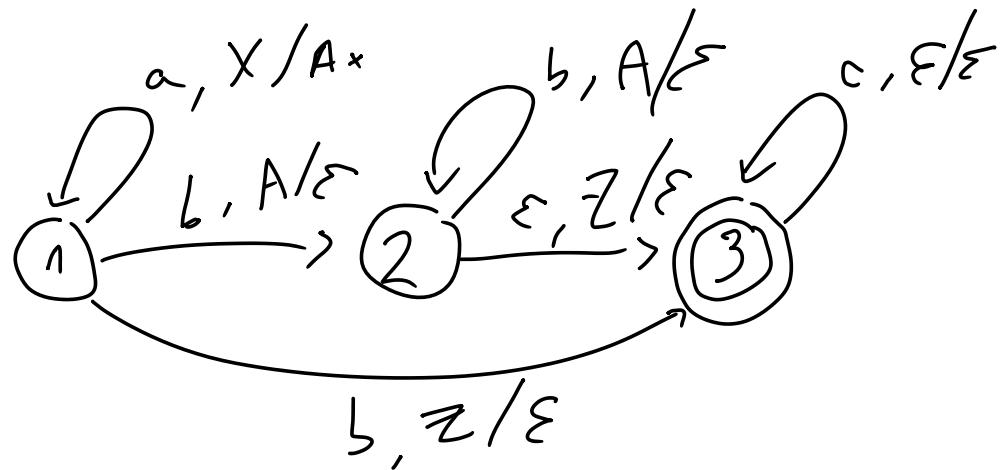
$$\begin{aligned} \Delta(E, a) &= C \\ \Delta(E, b) &= \{7, 8, 4, 5, 6\} = E \\ \Delta(E, \epsilon) &= C \\ \Delta(E, \emptyset) &= \{7, 8, 4, 5, 6\} = E \end{aligned}$$



Esercizio 7

lunedì 12 dicembre 2022 13:39

7. Mostrare che $L = \{a^n b^{n+1} c^m \mid n, m \geq 0\}$ è libero deterministico, costruendo un opportuno DPDA che riconosca L per stato finale.



Esercizio 8

lunedì 12 dicembre 2022 13:45

-
8. Si consideri la seguente grammatica G con simbolo iniziale S :

$$\begin{array}{lcl} S & \rightarrow & ACB \\ A & \rightarrow & \epsilon \mid aA \\ B & \rightarrow & b \mid bB \\ C & \rightarrow & \epsilon \mid cSC \end{array}$$

- (i) Si calcolino i First e i Follow per tutti i nonterminali. (ii) La grammatica G è di classe LL(1)?
(iii) Si rimuovano le produzione epsilon per ottenere una grammatica G' senza produzioni epsilon, che sia equivalente a G . (iv) Se la grammatica risultante presenta produzioni unitarie, si rimuovano, ottenendo una grammatica equivalente G'' .

SOLUZIONE

i. $\text{First}(S) = \{a, c, b\}$
 $\text{Follow}(S) = \{\$, c, b\}$

$$\begin{array}{l} \text{First}(A) = \{a, \epsilon\} \\ \text{Follow}(A) = \{c, b\} \end{array}$$

$$\begin{array}{l} \text{First}(C) = \{c, \epsilon\} \\ \text{Follow}(C) = \{b\} \end{array}$$

$$\begin{array}{l} \text{First}(B) = \{b\} \\ \text{Follow}(B) = \{\$, c, b\} \end{array}$$

- ii. Abbiamo che nella produzione $B \rightarrow b \mid bB$ il $\text{First}(b) \cap \text{First}(bB) \neq \emptyset$
Quindi non è di classe LL(1)

- iii. Troviamo $N_0 = \{A, C\}$, poi $N_1 = \{A, C\}$ quindi $N(G) = \{A, C\}$

$$\begin{array}{l} \emptyset \Rightarrow S \rightarrow ACB \\ \{A\} \Rightarrow S \rightarrow CB \\ \{C\} \Rightarrow S \rightarrow AB \\ \{A, C\} \Rightarrow S \rightarrow B \end{array}$$

$$\begin{array}{l} \emptyset \Rightarrow A \rightarrow aA \\ \{A\} \Rightarrow A \rightarrow a \end{array}$$

$$\begin{array}{l} \emptyset \Rightarrow C \rightarrow cSC \\ \{C\} \Rightarrow C \rightarrow cS \end{array}$$

$$\begin{array}{l} S \rightarrow ABC \mid CD \mid AB \mid B \\ A \rightarrow aA \mid a \\ B \rightarrow bB \mid b \\ C \rightarrow cSC \mid cS \end{array}$$

4. Rimuoviamo le produzioni unitarie da qua:

$$S \rightarrow ABC \mid CB \mid AB \mid B$$

$$A \rightarrow aA \mid a$$

$$B \rightarrow bB \mid b$$

$$C \rightarrow cSC \mid cS$$

Produzioni unitarie sono solo $S \rightarrow B$

Quindi

$$S \rightarrow ABC \mid CD \mid AB \mid bB \mid b$$

$$A \rightarrow aA \mid a$$

$$B \rightarrow bB \mid b$$

$$C \rightarrow cSC \mid cS$$

Esercizio 9 e 10

lunedì 12 dicembre 2022 14:29

9. Si consideri la grammatica G con simbolo iniziale S :

$$\begin{array}{l} S \rightarrow Sb \mid Sa \mid A \\ A \rightarrow c \mid ca \end{array}$$

- (i) Determinare il linguaggio generato $L(G)$. (ii) Verificare che G non è di classe LL(1). (iii) Manipolare la grammatica per ottenerne una equivalente G' di classe LL(1). (iv) Costruire il parser LL(1) per G' . (v) Mostrare il funzionamento del parser LL(1) su input cba .

10. Si consideri la grammatica G del punto precedente. (i) Costruire l'automa canonico LR(0). (ii) Costruire la tabella di parsing SLR(1) e verificare se ci sono conflitti. (iii) Mostrare il funzionamento del parser SLR(1) per l'input cba .

SOLUZIONE

9 - 1

Il linguaggio generato da $L(G)$ è $L(G) = \{c^n \cdot \{a, b\}^*, n\}$

$$S \rightarrow Sb \mid Sa \mid A$$

$$A \rightarrow c \mid ca$$

$$S \rightarrow Sb \rightarrow Ab$$

First di S :

$$c$$

First di A :

$$c$$

Follow di S :

$$b, a, \$$$

Follow di A :

$$b, a, \$$$

- Per ogni $X \in NT$, inizializza $Follow(X) := \emptyset$

- $Follow(S) := \{\$\}$

- Ripeti il seguente ciclo finché nessun $Follow(X)$ viene più modificato in una iterazione:

- 1) Per ogni produzione $X \rightarrow ay\beta$

$$Follow(y) := Follow(y) \cup (First(\beta) \setminus \{\epsilon\})$$

- 2) Per ogni produzione $X \rightarrow ay$ e per ogni produzione $X \rightarrow ay\beta$ con $\epsilon \in First(\beta)$

$$Follow(y) := Follow(y) \cup Follow(X)$$

$$Follow(S) = \{\$\}$$

$$Follow(A) = \{\Box\}$$

$$S \rightarrow Sb$$

$$X = S$$

- 1) $a = \epsilon$

$$y = S$$

$$\beta = b$$

$$Follow(S) := \{\$\} \cup (First(b) \setminus \{\epsilon\})$$

Linguaggio del punto precedente:

$$\begin{aligned} S &\rightarrow aSd \mid Bd \\ B &\rightarrow bbBc \mid \varepsilon \end{aligned}$$

$$L = \{a^n b^{2m} c^m d^{n+1} \mid n, m \geq 0\}$$

Adesso usiamo il pumping lemma per dimostrare che non è regolare:

SE
 $(\forall N > 0, \exists z \in L \text{ tale che } |z| \geq N)$

Fissiamo un N generico (tale che $N > 0$)
Scelgo $z = a^N b^{2N} c^N d^{N+1}$
Vale che $|z| \geq N$

Adesso scelgo una sottostringa uvw tale che:

$$\begin{aligned} z &= uvw \\ |uv| &\leq N \\ |v| &\geq 1 \end{aligned}$$

Per come ho costruito z , $uv = a^M$ tale che $M \leq N$
Per come ho costruito uv e uvw ho che $v = a^X$ con $X \geq 1$

Dobbiamo dimostrare che $\exists k \geq 0$ tale che $uv^k w \notin L$
Se noi scegliamo $k = 2$, allora abbiamo $v^2 = a^{2X}$, quindi:
 $a^{N+X} b^{2N} c^N d^{N+1} \notin L$

$$\begin{aligned} a^M &= uv \\ v &= a^X \\ a^M &= a^{M-X} a^X \end{aligned}$$

$$\begin{aligned} a^{M+X} &= a^{M-X} a^X a^X \\ v^2 &= a^X a^X \end{aligned}$$

Tutto vero

ALLORA possiamo dire che:

L non è regolare

$$L = \{a^n b^n c^n \mid n \geq 0\}$$

Fissiamo una N generica ($N > 0$)

Scegliamo $z = a^N b^N c^N$ (quindi vale che $|z| \geq N$)

Adesso dobbiamo scegliere $uvwxy = z$ tali che:

$$|vwx| \leq N$$

$$|vx| \geq 1$$

PER ESEMPIO:

$aabbcc$

$$|vwx| \leq 2$$

$= aa, ab, bb, bc, cc, a, b, c$

Per come abbiamo scelto z, ci sono 5 casi:

$$a^M$$

$$a^x b^{M-x}$$

$$b^M$$

$$b^x c^{M-x}$$

$$c^M$$

Dobbiamo dimostrare che esiste un k, per ognuno di questi casi, tale che $uv^k wx^k y$ non appartiene al linguaggio

- 1) $a^{N+qualcosa} b^N c^N \notin L$
- 2) $a^{N+qualcosa} b^{N+qualcosa} c^N \notin L$
- 3) $a^N b^{N+qualcosa} c^N \notin L$
- 4) $a^N b^{N+qualcosa} c^{N+qualcosa} \notin L$
- 5) $a^N b^N c^{N+qualcosa} \notin L$

repeat c until b

Da fare c una volta e poi da rifare mentre b è falsa:

$$\frac{-}{\langle \text{repeat } c \text{ until } b; \sigma \rangle \vdash \langle c; \text{while} \neg b \text{ do } c; \sigma \rangle}$$

$$\frac{-}{\langle \text{while } b \text{ do } c; \sigma \rangle \vdash \langle \text{if } b \text{ then } c; \text{while } b \text{ do } c; \sigma' \rangle}^{(\text{while}_1)}$$

$$\frac{\langle b, \sigma \rangle \vdash \langle b', \sigma' \rangle}{\langle \text{if } b \text{ then } c; \sigma \rangle \vdash \langle \text{if } b' \text{ then } c; \sigma' \rangle}^{(if_1)}$$

$$\frac{-}{\langle \text{if } tt \text{ then } c; \sigma \rangle \vdash \langle c; \sigma \rangle}$$

$$\frac{-}{\langle \text{if } ff \text{ then } c; \sigma \rangle \vdash \sigma}$$

mercoledì 14 dicembre 2022 13:46

Parziale

mercoledì 14 dicembre 2022 13:46

Esercizio su grammatica regolare

$S \rightarrow$

PROVA

domenica 18 dicembre 2022 20:54

7. Dimostrare che il linguaggio $L = \{a^{n^3} \mid n \geq 0\}$ non è regolare. A quale classe appartiene il linguaggio L^* ?

Soluzione:

Dimostriamo innanzitutto che non è regolare. Per dimostrare che non è regolare, dimostriamo il contrario del pumping lemma, ovvero dimostriamo che la proprietà che implica in pumping lemma non vale:

Fissiamo un N generica ($N \geq 1$)

Scegliamo una stringa $z = a^{N^3} \in L$ tale che $|z| \geq N$

Suddividiamo la stringa z in 3 sottostringhe u, v, w tali che:

$$z = uvw$$

$$|uv| \leq N$$

$$|v| \geq 1$$

Per come è costruito il linguaggio, abbiamo che $v = a^i, i \geq 1$.

Dobbiamo quindi dimostrare che $\exists k \geq 0$ tale che $uv^k w \notin L$

Quindi se scegliamo $k = 2$, abbiamo:

$$uv^2w = a^{N^3+i} \notin L$$

Quindi abbiamo dimostrato che questa proprietà non vale, quindi non è regolare.

Per capire a quale classe appartiene il linguaggio, suddividiamo usando le proprietà delle potenze il linguaggio in:

$$L = \left\{ \underbrace{(a^N a^N \dots)}_{N^3} \right\}$$

$$a^{n^3} \Rightarrow a^{n \cdot n^2} \Rightarrow a^{\overbrace{n+n+\dots+n}^{n^2}} \Rightarrow \underbrace{a^n}_{n^2}$$

$$L = \left\{ a^{n^3} \mid n \geq 0 \right\} = \left\{ \underbrace{a^n}_{n^2} \mid n \geq 0 \right\}$$

Se la classe dei linguaggi liberi si può idealmente vedere come $a^n b^n$

$$\frac{\langle b_0; \sigma \rangle \rightarrow \langle b'_0; \sigma' \rangle}{\langle b_0 \text{ or } b_1; \sigma \rangle \rightarrow \langle b'_0 \text{ or } b_1; \sigma' \rangle} (or_1)$$

$$\frac{\langle b_1; \sigma \rangle \rightarrow \langle b'_1; \sigma' \rangle}{\langle b_0 \text{ or } b_1; \sigma \rangle \rightarrow \langle b_0 \text{ or } b'_1; \sigma' \rangle} (or_2)$$

$$\frac{\langle b_0; \sigma \rangle \rightarrow \langle b'_0; \sigma' \rangle}{\langle b_0 \text{ or } t; \sigma \rangle \rightarrow \langle b'_0 \text{ or } t; \sigma' \rangle} (or_3)$$

(Dove t può essere tt o ff)

Un esempio di esecuzione diversa è per esempio:

$$\langle tt \text{ or } (3 - 5) = 2; \sigma \rangle$$

$$\rightarrow_{ES} \langle tt; \sigma \rangle$$

$$\frac{\langle b_1; \sigma \rangle \rightarrow \langle b'_1; \sigma' \rangle}{\langle t \text{ or } b_1; \sigma \rangle \rightarrow \langle t \text{ or } b'_1; \sigma' \rangle} (or_4)$$

$$\rightarrow_{IP} err$$

(Dove t può essere tt o ff)

$$\frac{-}{\langle t_0 \text{ or } t_1; \sigma \rangle \rightarrow \langle t'; \sigma \rangle} (or_5)$$

Dove si usa al tabella di verità:

t_0	t_1	t'
Ff	Ff	Ff
Tt	Ff	Tt
Ff	Tt	Tt
Tt	Tt	tt

1. Nomi

martedì 21 febbraio 2023 09:27