UNIVERSITÀ DEGLI STUDI DI FERRARA

- EX LABORE FRUCTUS -

Capitolo 5
Riconoscitori per Grammatiche Context-Free. PDA

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi

QUESTO MATERIALE DIDATTICO È PER USO PERSONALE DELLO STUDENTE ED È COPERTO DA COPYRIGHT. NE È SEVERAMENTE VIETATA LA RIPRODUZIONE O IL RIUTILIZZO ANCHE PARZIALE, AI SENSI E PER GLI EFFETTI DELLA LEGGE SUL DIRITTO D'AUTORE.

QUALI MACCHINE per QUALI LINGUAGGI ?

Chi riconosce i diversi tipi di linguaggi?

GRAMMATICHE	AUTOMI RICONOSCITORI
• Tipo 0	• ***Se L(G) è riconoscibile***, occorre una Macchina di Turing
• Tipo 1	• Macchina di Turing (con nastro limitato)
• Tipo 2 *(context-free)*	• Push-down automaton (ASF + stack)
• Tipo 3 *(regolari)*	• Automa a Stati Finiti (ASF)

PUSH - DOWN AUTOMATA (PDA)

Un linguaggio di Tipo 2 *non può essere riconosciuto* da un RSF.

ESEMPIO

Sia A = { **0**, **1**, **c** } un alfabeto, con:

$$S \rightarrow 0\,S\,0 \mid 1\,S\,1 \mid c$$

Il linguaggio generato è

$$L = \{ word\ c\ word^R \}$$

dove ***word***^{R} indica il ribaltamento della stringa ***word***, che a sua volta indica tutte le possibili sequenze di 0 e 1, inclusa la stringa vuota ε .

Questo linguaggio non è riconoscibile da un RSF perché occorre memorizzare la stringa ***word, la cui lunghezza non è limitata a priori.***

Per riconoscere un linguaggio di Tipo 2 occorre un *Push-Down Automaton: ASF + STACK*

- Come vedremo, **formalmente, lo stack è definito come *sequenza di simboli:*** per convenzione, quello più a destra si considererà in cima alla pila.
- Come un ASF, un PDA legge un simbolo d'ingresso e transita in un nuovo stato, ma in più *produce una nuova configurazione dello stack,* in funzione del simbolo d'ingresso e di quello in cima allo stack.
- Un PDA può o meno prevedere ***ε -mosse***, sorta di transizioni "spontanee" che manipolano lo stack senza consumare simboli di ingresso.

PUSH - DOWN AUTOMATA (PDA)

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

- ***Criterio dello stato finale:*** come in un RSF, il linguaggio accettato è l'insieme delle stringhe di ingresso che portano il PDA *in uno degli stati finali*.
- ***Criterio dello stack vuoto:*** appoggiandosi al nuovo concetto di *stack*, il linguaggio accettato è definito come l'insieme delle stringhe di ingresso che portano il PDA nella configurazione di *stack vuoto*.

Il primo criterio implica la definizione dell'insieme degli *stati finali*, ergo la definizione a lato diventa una *settupla.*

DEFINIZIONE DI PDA

Un PDA è una *sestupla:*

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

dove

- **A = alfabeto**
- **S = insieme degli stati**
- S_0 **= stato iniziale** $\in S$
- **sfn:** $(A \cup \{\varepsilon\}) \times S \times Z \rightarrow Q$ con Q sottoinsieme finito di $S \times Z^*$
- **Z = alfabeto dei *simboli interni***
- $Z_0 \in Z$ = *simbolo iniziale* sullo stack

Questa definizione *include* il caso delle ε *-mosse:* per *escluderle* basta definire sfn sul dominio $A \times S \times Z$.

PUSH - DOWN AUTOMATA (PDA)

La funzione `sfn`, dati:

- un *simbolo di ingresso* — a∈ A
- lo *stato attuale* — s∈ S
- il *simbolo interno attualmente al top dello stack* — z∈ Z

opera come segue:

- ***consuma*** il simbolo d'ingresso a
- ***effettua una POP dallo stack,*** prelevando così il simbolo interno attualmente al top, z
- ***fornisce due risultati***
  (*s'*,*z'*) = *sfn*(*a*,*s*,*z*)
- ***porta l'automa nello stato futuro*** s'
- ***effettua una PUSH sullo stack*** di *zero o più simboli interni* z'∈ Z*

DEFINIZIONE DI PDA

Un PDA è una *sestupla:*

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

dove

- ♦ **A = alfabeto**
- ♦ **S = insieme degli stati**
- ♦ **S_0= stato iniziale ∈ S**
- ♦ **sfn: (A∪ {ε }) × S × Z → Q**
  con Q sottoinsieme finito di S × **Z***
- ♦ **Z = alfabeto dei *simboli interni***
- ♦ **Z_0 ∈ Z** = *simbolo iniziale* sullo stack

Questa definizione *include* il caso delle ε -*mosse:* per *escluderle* basta definire sfn sul dominio A × S × Z.

PDA: ESEMPIO

Si consideri il linguaggio generato da:

Alfabeto: A = { **0, 1, c** }
Produzioni: P = { S → 0 S 0 | 1 S 1 | c }
Linguaggio: L = { ***word* c *word^R*** }
dove *word^R* indica il ribaltamento di *word*
e *word* indica tutte le possibili sequenze di 0 e 1, inclusa la stringa vuota ε .

Questo automa accetta il linguaggio L con il *criterio dello stack vuoto.*

INTUITIVAMENTE:

- Q1 è lo stato di *accumulo* in cui si memorizzano i simboli *prima della* '**c**' *centrale*
- Q2 è lo stato di *svuotamento* in cui si recuperano i simboli dallo stack *e si verifica che corrispondano a quelli in input*
- quando si trova l'elemento centrale '**c**', si commuta da Q1 a Q2.

Definiamo il PDA come segue:

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

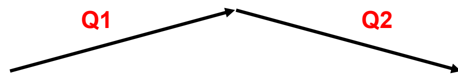
A = { **0, 1, c** }
S = { **Q1=S_0, Q2** }
Z = { Zero, Uno, Centro }, **Z_0** = Centro

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

PDA: ESEMPIO

FUNZIONAMENTO

- all'inizio lo stack contiene per ipotesi il simbolo interno *Centro*
- si consumano i simboli di ingresso operando come da tabella
- la stringa è riconosciuta se alla fine lo stack contiene solo il simbolo interno *Centro* iniziale, che viene infine consumato con una ε -mossa.



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

Stringa di ingresso: 01c10

- Input: 0 — Stack: Centro — Stato: Q1 → Stato: Q1 — Pop: Centro Push: CentroZero
- Input: 1 — Stack: CentroZero — Stato: Q1 → Stato: Q1 — Pop: Zero Push: ZeroUno
- Input: c — Stack: CentroZeroUno — Stato: Q1 → Stato: Q2 — Pop: Uno Push: Uno
- Input: 1 — Stack: CentroZeroUno — Stato: Q2 → Stato: Q2 — Pop: Uno Push: ε
- Input: 0 — Stack: CentroZero — Stato: Q2 → Stato: Q2 — Pop: Zero Push: ε
- Input: ε — Stack: Centro — Stato: Q2 → Stato: Q2 — Pop: Centro Push: ε

stack vuoto → stringa riconosciuta

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 5.1
PDA Non deterministici

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi
QUESTO MATERIALE DIDATTICO È PER USO PERSONALE DELLO STUDENTE ED È COPERTO DA COPYRIGHT. NE È SEVERAMENTE VIETATA LA RIPRODUZIONE O IL RIUTILIZZO ANCHE PARZIALE, AI SENSI E PER GLI EFFETTI DELLA LEGGE SUL DIRITTO D'AUTORE.

PDA NON DETERMINISTICI

Anche un PDA può essere *non deterministico:* in tal caso, la funzione sfn produce *insiemi di elementi di Q* (Q sottoinsieme finito di S × Z*)

Ad esempio, il PDA tale che

$sfn(Q_0, a, Z) = \{ (Q_1,Z_1), (Q_2,Z_2), \ldots (Q_k, Z_k) \}$

è *non deterministico* in quanto l'automa, nello stato Q_0, con simbolo interno in cima allo stack Z, e con ingresso a, può scegliere in quale stato futuro portarsi; in base alla scelta che fa, cambia anche il set di simboli da porre sullo stack.

DEFINIZIONE DI PDA *nondet*

Un PDA *nondet* è una sestupla:

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

dove

- A = alfabeto *(A* = chiusura)*
- S = insieme degli stati
- S_0 = stato iniziale $\in$ S
- sfn: $(A \cup \{\varepsilon\}) \times S \times Z \to Q^n$ con Q sottoinsieme finito di $S \times Z^*$
- Z = alfabeto dei *simboli interni*
- $Z_0 \in Z$ = *simb. iniz.* sullo stack

Questa definizione *include* il caso delle ε *-mosse:* per *escluderle* basta definire sfn sul dominio $A \times S \times Z$.

PDA NON DETERMINISTICI

Il *non-determinismo* dell'automa può emergere sotto due aspetti:

- **l'automa, in un certo stato Q_0, con simbolo interno in cima allo stack Z, e con ingresso x, può portarsi *in uno qualunque degli stati futuri previsti:***
  $sfn(Q_0, x, Z) = \{ (Q_1,Z_1), (Q_2,Z_2), \ldots (Q_k, Z_k) \}$
- **l'automa, in un certo stato Q_i, con simbolo interno in cima allo stack Z, e con ingresso x, può *leggere o non leggere il simbolo di ingresso x.***
  Ciò accade se sono definite *entrambe* le mosse:
  $sfn(Q_i, x, Z)$ e $sfn(Q_i, \varepsilon, Z)$
  di cui *la seconda è una* ε *-mossa.*

  In tal caso, infatti, l'automa può sia leggere x sia non farlo, scattando autonomamente senza leggere nulla.

PDA NON DETERMINISTICI: VANTAGGI E SVANTAGGI

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

Problema

- **i migliori algoritmi hanno complessità dell'ordine del *cubo della lunghezza* della stringa da riconoscere, che però si riduce al *quadrato se la grammatica non è ambigua.***

VERSO PDA DETERMINISTICI

Si può rinunciare ai PDA non deterministici?

In generale, no:

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

.. ma in molti casi di interesse pratico, sì:

Esistono linguaggi context-free *riconoscibili da PDA deterministici* (linguaggi context-free deterministici)

- **In tal caso, la complessità di calcolo del PDA deterministico è *lineare* rispetto alla lunghezza della stringa da riconoscere**

PDA DETERMINISTICI

Cosa serve per ottenerlo?

Viste le condizioni precedenti, *non deve succedere* che l'automa, in un dato stato `Q0`, con simbolo in cima allo stack `Z` e ingresso `x`, possa:

- **portarsi *in più stati futuri***
  $sfn(Q_0, x, Z) = \{ (Q_1,Z_1), (Q_2,Z_2), \ldots (Q_k, Z_k) \}$
- **optare se *leggere o non leggere il simbolo di ingresso* `x` a causa della presenza di *entrambe* le mosse**
  $sfn(Q_i, x, Z)$ e $sfn(Q_i, \varepsilon, Z)$
  di cui *la seconda è una* ε *-mossa.*

Dovremo capire *come tradurre questi vincoli sulla grammatica,* in modo da sapere *che regole scrivere* (e quali *non* scrivere) per *assicurarsi che il risultato sia un linguaggio deterministico.*

PDA DETERMINISTICI: PROPRIETÀ

MIX FRA LINGUAGGI DETERMINISTICI

- **L'unione, l'intersezione e il concatenamento di due linguaggi deterministici *non* danno necessariamente luogo a un linguaggio deterministico**
- **Il complemento di un linguaggio deterministico invece è deterministico** (ovvio…)

MIX FRA LINGUAGGI DETERMINISTICI E REGOLARI

- **Se L è un linguaggio deterministico e R un linguaggio *regolare,* il *linguaggio quoziente L/R (ossia l'insieme delle stringhe di L private di un suffisso regolare)* è deterministico**
- **Se L è un linguaggio deterministico e R un linguaggio *regolare,* il *concatenamento L.R (ossia l'insieme delle stringhe di L con un suffisso regolare)* è deterministico**

UNIVERSITÀ
DEGLI STUDI
DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 5.2
Analisi Ricorsiva Discendente

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi
QUESTO MATERIALE DIDATTICO È PER USO PERSONALE DELLO STUDENTE ED È COPERTO DA COPYRIGHT. NE È SEVERAMENTE VIETATA LA RIPRODUZIONE O IL RIUTILIZZO ANCHE PARZIALE, AI SENSI E PER GLI EFFETTI DELLA LEGGE SUL DIRITTO D'AUTORE.

REALIZZAZIONE DI PDA DETERMINISTICI

Come realizzare in pratica un PDA deterministico?

- si può seguire la definizione (auguri..)
- **oppure si può *adottare un approccio che manipoli uno stack con la stessa logica di un PDA***
  - **la presenza dello stack è la vera differenza rispetto al RSF**
  - **una macchina virtuale che abbia uno stack *può essere fatta funzionare come un PDA* pilotandola "opportunamente"**
  - **si potrebbe pilotare uno stack in modo esplicito, "a mano"..**
  - **.. ma è molto più comodo *sfruttare eventuali costrutti dei linguaggi di programmazione che lo facciano per noi***
    → chiamate ricorsive di funzioni e procedure !

REALIZZAZIONE DI PDA DETERMINISTICI

I linguaggi di programmazione che supportano *chiamate ricorsive di funzioni e procedure* *gestiscono già implicitamente uno stack* → POSSIAMO SFRUTTARLO!

- **ogni chiamata di funzione implica l'allocazione sullo stack di un *record di attivazione***
- **quando la funzione termina, il record di attivazione viene *automaticamente deallocato***
- **basta mettere i dati da manipolare nelle *variabili locali* e *negli argomenti* della funzione..**
- **.. e gestire il tutto con furbizia**

ANALISI RICORSIVA DISCENDENTE (Top-Down Recursive-Descent Parsing)

Il risultato è la tecnica nota come

ANALISI RICORSIVA DISCENDENTE

- **si introduce *una funzione per ogni metasimbolo* della grammatica e *la si chiama ogni volta che si incontra quel metasimbolo***
- **ogni funzione copre le regole di quel metasimbolo,** ossia *riconosce il sotto-linguaggio* corrispondente
  - termina normalmente, o restituisce un segno di successo, se incontra simboli *coerenti con le proprie regole*
  - abortisce, o restituisce un qualche segno di fallimento, se incontra simboli *che non corrispondono alle sue regole.*

ESEMPIO (1)

Il "solito" linguaggio L = { ***word*** `c` ***word***^{R} }
Alfabeto: A = { **0**, **1**, **c** }
Regole: `S → 0 S 0 | 1 S 1 | c`

1. **Introdurre tante funzioni quanti i metasimboli**
   Qui c'è solo S → una funzione sola, `S()`
2. **Chiamare una funzione ogni volta che si incontra il suo metasimbolo**
   Qui c'è solo S → ogni volta che lo troviamo, invochiamo `S()`
3. **Ogni funzione deve coprire le regole di quel metasimbolo**
   Qui le regole sono tre → ci saranno tre casi:
   - se il simbolo d'ingresso è `0` → seguire la *prima regola*
   - se il simbolo d'ingresso è `1` → seguire la *seconda regola*
   - se il simbolo d'ingresso è `c` → seguire la *terza regola*

ESEMPIO (2)

Prima regola: `S → 0 S 0`
Seconda regola: `S → 1 S 1`
Terza regola: `S → c`

Caso *prima regola* (il simbolo d'ingresso è `0`)

- consumiamo il carattere d'ingresso `0`
- invochiamo ricorsivamente la funzione `S()`
- consumiamo un nuovo carattere d'ingresso e verifichiamo che sia `0`
  - se la verifica ha esito positivo, significa che la funzione *ha incontrato simboli coerenti con le proprie regole* → termina normalmente, o restituisce un segno di successo
  - se invece tale verifica ha esito negativo, significa che la funzione *ha incontrato simboli che non corrispondono alle sue regole.* → abortisce, o restituisce un qualche segno di fallimento

ESEMPIO (3)

Prima regola: S → 0 S 0
Seconda regola: S → 1 S 1
Terza regola: S → c

Caso *seconda regola* (il simbolo d'ingresso è 1)

- consumiamo il carattere d'ingresso 1
- invochiamo ricorsivamente la funzione S()
- consumiamo un nuovo carattere d'ingresso e verifichiamo che sia 1
  - se la verifica ha esito positivo, significa che la funzione *ha incontrato simboli coerenti con le proprie regole* → termina normalmente, o restituisce un segno di successo
  - se invece tale verifica ha esito negativo, significa che la funzione *ha incontrato simboli che non corrispondono alle sue regole.* → abortisce, o restituisce un qualche segno di fallimento

ESEMPIO (4)

Prima regola: S → 0 S 0
Seconda regola: S → 1 S 1
Terza regola: S → c

Caso *prima regola* (il simbolo d'ingresso è c)

- consumiamo il carattere d'ingresso c
- NON invochiamo altre funzioni
- NON consumiamo nuovi caratteri d'ingresso e NON verifichiamo nulla
  - la funzione *ha incontrato simboli coerenti con le proprie regole* → termina normalmente, o restituisce un segno di successo

Osserva: i primi due casi sono identici a meno di un parametro (il carattere di ingresso da consumare e controllare), quindi possono utilmente essere compattati a livello di codice

ESEMPIO (5)

S → 0 S 0
S → 1 S 1
S → c

Una codifica di principio in linguaggio C / C++

```
char ch; // globale
bool S()
{
 char first;
 ch = nextchar();              /* recupera il prossimo carattere di ingresso */
 switch (ch)
 {case 'c':  ch = nextchar(); return true;
  case '0':
  case '1':  first = ch;                          /* push */
             if (S())
                if(ch==first)
                { ch = nextchar(); return true; /* pop */
                }
                else return false;
             else return false;
  default:   return false;
 }
}
```

Le istanze di `first` nei record di attivazione rappresentano *de facto* lo stack del PDA.

ESEMPIO (6)

Il codice di contorno

```
#include <stdio.h>
char ch; // globale
int i=0;
char input[20] = "...";
char nextchar()
{ char ctemp;
  ctemp=input[i];    i++;
  return ctemp;
}
int main()
{ if(S()) printf("success!");
  else printf("failure");
}
```

Input fisso simulato per test

Alcuni run

110c011	success!
110011	failure

Capitolo 5.3
Analisi Ricorsiva Discendente - esempio

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi
QUESTO MATERIALE DIDATTICO È PER USO PERSONALE DELLO STUDENTE ED È COPERTO DA COPYRIGHT. NE È SEVERAMENTE VIETATA LA RIPRODUZIONE O IL RIUTILIZZO ANCHE PARZIALE, AI SENSI E PER GLI EFFETTI DELLA LEGGE SUL DIRITTO D'AUTORE.

GRAMMATICHE LL(1) : ESEMPIO

Si consideri la grammatica:

VT = { p, q, a, b, d, x, y }
VN = { S *(scopo)*, X, Y }

Produzioni:

```
S →  p X | q Y
X → a X b | x
Y → a Y d | y
```

- **Le parti *destre* delle *produzioni di uno stesso meta-simbolo* iniziano *tutte con un simbolo terminale diverso.***
- **È quindi sufficiente *guardare avanti di un carattere* per scegliere con certezza la produzione con cui proseguire l'analisi**
- **Se non esistono produzioni *compatibili con quell'input*, ERRORE.**

ESEMPIO: IL PDA DETERMINISTICO

Una codifica di principio in linguaggio C / C++

- Serve un contratto chiaro su *chi legge l'input* e *quando lo fa*
- Ipotesi:
  - *ogni funzione trova nella variabile globale* `ch` *il prossimo carattere da analizzare,* già letto ma non ancora analizzato
  - ergo ogni funzione, prima della `return`, *effettua una lettura da input a beneficio di chi verrà dopo di lei*
  - il `main` effettua la prima lettura *prima* di invocare la funzione di top-level

Lettura da input di nuovi simboli
- **o sempre PRIMA di agire**
- **o sempre DOPO aver agito**

```
char ch;            /* variabile globale*/
main()
{ ch = nextchar();            /* recupera il prossimo carattere di ingresso */
  if (S()) printf("Frase accettata."); else printf("Errore");
}
```

ESEMPIO: IL PDA DETERMINISTICO

Una codifica di principio in linguaggio C / C++

```
S →  p X | q Y
X → a X b | x
Y → a Y d | y
```

```
bool S()
{switch (ch)
 { case 'p':   ch = nextchar(); return X();   /* produzione S ::= pX */
   case 'q':   ch = nextchar(); return Y();   /* produzione S ::= qY */
 }
 return false; /* nessuna produzione  corrispondente → ERRORE */
}
bool X()
{ switch (ch)
  {case 'a': ch = nextchar();                  /* produzione X ::= aXb */
             if (X())
              { if (ch=='b') { ch=nextchar(); return true; }
                else return false;       /* non corrisponde → ERRORE */
              } else return false;       /* non corrisponde → ERRORE */
   case 'x': ch = nextchar(); return true;     /* produzione X ::= x */
  }
 return false; /* nessuna produzione  corrispondente → ERRORE */
}
```

/ analogamente si scrive la funzione* `Y()` *per le altre produzioni */*

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 5.4
Analisi Ricorsiva Discendente - esempio Haskell

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi
QUESTO MATERIALE DIDATTICO È PER USO PERSONALE DELLO STUDENTE ED È COPERTO DA COPYRIGHT. NE È SEVERAMENTE VIETATA LA RIPRODUZIONE O IL RIUTILIZZO ANCHE PARZIALE, AI SENSI E PER GLI EFFETTI DELLA LEGGE SUL DIRITTO D'AUTORE.

Analisi Ricors. Disc. Haskell

- Una semplice implementazione Haskell dell'analisi ricorsiva discendente
- Ogni funzione (associata ad un nonterminale) deve fornire due risultati:
  - un Bool (per dire se la stringa è stata riconosciuta)
  - il resto della stringa, su cui deve essere eseguito il parsing

```
S → p X | q Y
X → a X b | x
Y → a Y d | y
```

```
s :: String -> (Bool,String)
s ('p':t) = x t
s ('q':t) = y t
s xs = (False,xs)

x ('a':t) =
    let (b,r) = x t
    in (b && head r == 'b', tail r)
x ('x':t) = (True,t)
x xs = (False,xs)
```

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 5.5
LL(k)

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi
QUESTO MATERIALE DIDATTICO È PER USO PERSONALE DELLO STUDENTE ED È COPERTO DA COPYRIGHT. NE È SEVERAMENTE VIETATA LA RIPRODUZIONE O IL RIUTILIZZO ANCHE PARZIALE, AI SENSI E PER GLI EFFETTI DELLA LEGGE SUL DIRITTO D'AUTORE.

ANALISI RICORSIVA DISCENDENTE: VANTAGGI e LIMITI

VANTAGGI

- **è immediato scrivere il riconoscitore a partire dalla grammatica**
- **il mapping fra struttura della grammatica e del riconoscitore *riduce la probabilità di errori* e *migliora la leggibilità e la modificabilità del codice***
- **è facilitata l'inserzione di azioni nella fase di analisi**
  (come la creazione di rappresentazioni interne del programma..)

LIMITI

- **l'analisi ricorsiva discendente *non è sempre applicabile***
- **l'approccio *funziona solo se non ci sono mai "dubbi" su quale regola applicare* in una qualsiasi situazione**

Ciò suggerisce di identificare una *classe ristretta di grammatiche context-free*, che *garantisca il determinismo* dell'analisi sintattica discendente.

ANALISI RICORSIVA DISCENDENTE DETERMINISTICA

Per rendere deterministica l'analisi top-down bisogna *mettersi nella condizioni di poter dedurre la mossa giusta in base alle informazioni "disponibili",* senza dover tirare a indovinare

Cosa si intende per "informazioni disponibili"?

- **sicuramente, *le regole che abbiamo usato fin lì* e soprattutto *i simboli di input che abbiamo letto e consumato fin lì* → *IL PASSATO***
- **spesso, però, la mera conoscenza del passato non basta: si ipotizza perciò di *poter "vedere avanti" di k simboli,* ossia di poter "sbirciare" l'input *ancora da leggere* → *UN OCCHIO SUL FUTURO PROSSIMO***

GRAMMATICHE LL(*k*)

Si definiscono *grammatiche LL(k)* quelle che sono analizzabili *in modo deterministico*

- **procedendo *Left to right***
- **applicando la *Left-most derivation*** (derivazione canonica sinistra)
- **guardando avanti di *al più k simboli***

In sostanza, se una grammatica è LL(*k*), è sempre possibile *scegliere con certezza* la produzione da usare per procedere, guardando avanti al più di *k* simboli sull'input.

Rivestono particolare interesse le *GRAMMATICHE LL*(1)
- **quelle in cui basta *guardare avanti di un solo simbolo* per poter operare in modo deterministico.**

GRAMMATICHE LL(1) : ESEMPIO

Si consideri la grammatica:

VT = { `p`, `q`, `a`, `b`, `d`, `x`, `y` }
VN = { `S` *(scopo)*, `X`, `Y` }

Produzioni:

```
S →   p X | q Y
X → a X b | x
Y → a Y d | y
```

- **Le parti *destre* delle *produzioni di uno stesso meta-simbolo* iniziano *tutte con un simbolo terminale diverso.***
- **È quindi sufficiente *guardare avanti di un carattere* per scegliere con certezza la produzione con cui proseguire l'analisi**
- **Se non esistono produzioni *compatibili con quell'input*, ERRORE.**

ESEMPIO: riconoscimento della frase `paaaxbbb`

Frase	Produzione	Derivazione
`paaaxbbb`	`S ::= pX`	`pX`
`aaaxbbb`	`X ::= aXb`	`paXb`
`aaxbb`	`X ::= aXb`	`paaXbb`
`axb`	`X ::= aXb`	`paaaXbbb`
`x`	`X ::= x`	`paaaxbbb`
	nessuna	`paaaxbbb`

SEPARARE MOTORE E GRAMMATICA

- **Applicare l'analisi ricorsiva discendente è un *processo meccanico* semplice..**
- **.. ma dà luogo a un insieme di funzioni che *cablano nel codice* il comportamento del PDA.**

Può essere invece opportuno *SEPARARE il motore* (invariante rispetto alle regole) *dalle regole della specifica grammatica.*

Si costruisce a questo scopo una *TABELLA DI PARSING*
- **simile alla tabella delle transizioni di un RSF**
- **ma indica *la prossima produzione da applicare***

Il motore del parser *(parsing engine)* svolgerà le singole azioni *consultando la tabella di parsing.*

PARSING TABLES
ESEMPI DETERMINISTICI

Il solito linguaggio: L = { ***word*** c ***word***^{R} }

	0	1	c
S	S → 0 S 0	S → 1 S 1	S → c

L = { if c then cmd (endif | else cmd) }
Produzioni:
S → if c then cmd X   X → endif | else cmd

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

Parser LL non ricorsivo

```
Pila := S$; /*cima della pila, inseriamo $ in fondo*/
X := S;     /* top della pila */
input := w$; ic := primo carattere di input ;
while (X ≠ $)        /* while ( pila non vuota ) */
{  if (X è un terminale)
      if (X = ic)
      {   pop X dalla pila ; avanza ic su input ;
      }
      else errore();
   else                /* X non terminale */
      if (M[X,ic] = X → Y1 ··· Yn )
      {   pop X dalla pila ;
          push Y1 ··· Yn sulla pila (Y1 in cima );
      }
      else errore();
   X=top(Pila);
}
if (ic ≠ $) errore();
```

Esempio

	0	1	c
S	S → 0 S 0	S → 1 S 1	S → c

```
Pila := S$;
X := S;         /* top della pila */
input := w$; ic := primo car. input ;
while (X ≠ $)
{  if (X è un terminale)
      if (X = ic)
      {   pop X dalla pila ;
          avanza ic su input ;
      }
      else errore();
   else         /* X non terminale */
      if (M[X,ic] = X → Y1 ··· Yn )
      {   pop X dalla pila ;
          push Y1 ··· Yn sulla pila
      }
      else errore();
   X=top(Pila);
}
if (ic ≠ $) errore();
```

Stack	Input
S$	01c10$
0S0$	01c10$
S0$	1c10$
1S10$	1c10$
S10$	c10$
c10$	c10$
10$	10$
0$	0$
$	$

ESEMPIO

Si consideri la grammatica:

VT = { p, q, a, b, d, x, y }
VN = { S *(scopo)*, X, Y }

Produzioni:

S → p X | q Y
X → a X b | x
Y → a Y d | y

- **Si disegni la parsing table**

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

L'ESEMPIO: PARSING TABLE

Si consideri la grammatica:

VT = { p, q, a, b, d, x, y }
VN = { S *(scopo)*, X, Y }

Produzioni:

```
S →  p X | q Y
X → a X b | x
Y → a Y d | y
```

- **Costruendo la tabella di parsing è facile constatare che il parser è deterministico**
  - ossia, che la grammatica è LL(1)
- **perché *ogni cella contiene una sola produzione*** e quindi *non c'è mai dubbio su quale sia la prossima mossa da fare.*

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

Capitolo 5.6
LL(1)
regole che iniziano con un nonterminale

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi

QUESTO MATERIALE DIDATTICO È PER USO PERSONALE DELLO STUDENTE ED È COPERTO DA COPYRIGHT. NE È SEVERAMENTE VIETATA LA RIPRODUZIONE O IL RIUTILIZZO ANCHE PARZIALE, AI SENSI E PER GLI EFFETTI DELLA LEGGE SUL DIRITTO D'AUTORE.

L'ESEMPIO: PARSING TABLE

Si consideri la grammatica:

VT = { p, q, a, b, d, x, y }
VN = { S *(scopo)*, X, Y }

Produzioni:

```
S →  p X | q Y
X → a X b | x
Y → a Y d | y
```

- **Costruendo la tabella di parsing è facile constatare che il parser è deterministico**
  - ossia, che la grammatica è LL(1)
- **perché *ogni cella contiene una sola produzione*** e quindi *non c'è mai dubbio su quale sia la prossima mossa da fare.*

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

GENERALIZZAZIONE

- **Spesso però le parti destre delle produzioni *di uno stesso meta-simbolo non* iniziano tutte con un simbolo *terminale***
- **Quindi, non è immediatamente chiaro *quali siano gli input "ammissibili"***
- **Occorre *ragionare considerando più produzioni* fino a "svelare" l'arcano.**

Si consideri l'esempio a lato:

- ***se la frase d'ingresso inizia con* p*, va scelta la produzione* S → R Y** poiché solo la sua (sotto)produzione R → pXb può produrre un p iniziale.
- **Analogamente, se la frase di input inizia con q *va scelta la produzione* S → T Z**, poiché solo la sua sottoproduzione T → qYd può produrre un q iniziale.

VT = { p, q, a, b, d, y }
VN = { S, X, Y, T, Z, R }

Produzioni:

```
S →   R Y | T Z
R →   p X b
T →   q Y d
Z →   y
X → a X b | x
Y → a Y d | y
```

Questo porta a generalizzare il concetto di "simbolo iniziale".

STARTER SYMBOL SET

Definiamo

- ***starter set* del non-terminale $A \in VN$ l'insieme**

  $FIRST(A) = \{ a \in VT \mid A \stackrel{+}{\Rightarrow} a\beta \}$, con $\beta \in V^*$

- ***starter set* della forma di frase α l'insieme**

  $FIRST(\alpha) = \{ a \in VT \mid \alpha \stackrel{*}{\Rightarrow} a\beta \}$, con $\alpha \in V^+$ e $\beta \in V^*$

In sostanza, gli starter set sono i simboli iniziali di un dato meta-simbolo (o di una specifica sua riscrittura) *ricavati anche da più produzioni* se non sono immediatamente evidenti al primo livello.

Ciò permette di generalizzare facilmente l'approccio precedente:

- PRIMA: le parti destre delle produzioni di uno stesso meta-simbolo *iniziano tutte con un simbolo terminale diverso.*
- **ORA: le produzioni di uno stesso meta-simbolo *sono caratterizzate da starter symbol set diversi.***

STARTER SYMBOLS

Definiamo

- ***starter set* del non-terminale $A \in VN$ l'insieme**

  $FIRST(A) = \{ a \in VT \mid A \stackrel{+}{\Rightarrow} a\beta \}$, con $\beta \in V^*$

- ***starter set* della forma di frase α l'insieme**

  $FIRST(\alpha) = \{ a \in VT \mid \alpha \stackrel{*}{\Rightarrow} a\beta \}$, con $\alpha \in V^+$ e $\beta \in V^*$

CONDIZIONE NECESSARIA* perché una grammatica sia LL(1) è che per ogni nonterminale X *gli starter set* dei metasimboli con cui iniziano le parti destre delle produzioni per *X* siano *disgiunti

Come vedremo, la condizione diventa anche *sufficiente se nessun metasimbolo genera la stringa vuota.*

Calcolare gli starter symbol set (versione senza ε)

Per calcolare **FIRST**(X) per tutti i non-terminali X, si applicano le seguenti regole finché non è più possibile aggiungere nulla

- Se X è un terminale, allora **FIRST**(X)={X}
- Se X è un non-terminale e c'è una regola

  $$X \rightarrow Y_1, Y_2, \ldots, Y_k$$

  allora aggiungi il nonterminale **a** a **FIRST**(X) se
  - **a** $\in$ **FIRST**(Y_i) e
  - per tutti gli Y_j con $j<i$, $Y_j \stackrel{+}{\Rightarrow} \varepsilon$

Quindi se Y_1 non può produrre ε, non controlliamo Y_2, etc.

ESEMPIO

Nel caso a lato, gli *starter symbol set* dei metasimboli con cui iniziano le *parti destre* di *produzioni alternative* sono:

```
FIRST(R) = {p}
FIRST(T) = {q}
```

Essi sono *disgiunti:* FIRST(R) ∩ FIRST(T) = ∅ quindi, la grammatica *può essere* LL(1).

Poiché nessuno genera la stringa vuota, la condizione è anche *sufficiente.*

VT = { p, q, a, b, d, y }

VN = { S, X, Y, T, Z, R }

Produzioni:

```
S → R Y | T Z
R → p X b
T → q Y d
Z → y
X → a X b | x
Y → a Y d | y
```

Perché la stringa vuota fa differenza?

- se una produzione genera la stringa vuota, quel meta-simbolo *può sparire* quando viene sostituito in un'altra regola
- ergo, regole che *sembrano* iniziare con un certo metasimbolo *in realtà iniziano col successivo* e questo *va messo in conto!*

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Capitolo 5.7
LL(1)
ϵ-rules

Corso di Laurea Magistrale in Ingegneria Informatica e dell'automazione

Anno accademico 2019/2020

Prof. MARCO GAVANELLI

Si ringrazia il Prof. Enrico Denti per aver fornito la prima versione di questi lucidi

QUESTO MATERIALE DIDATTICO È PER USO PERSONALE DELLO STUDENTE ED È COPERTO DA COPYRIGHT. NE È SEVERAMENTE VIETATA LA RIPRODUZIONE O IL RIUTILIZZO ANCHE PARZIALE, AI SENSI E PER GLI EFFETTI DELLA LEGGE SUL DIRITTO D'AUTORE.

IL PROBLEMA DELLA STRINGA VUOTA

Nell'esempio a lato:

- **gli starter set di A e B nelle due parti destre di `S → AB | B` sono disgiunti:**
  `FIRST(A) = {a}` `FIRST(B) = {b,c}`
- **Nessuno genera la stringa vuota → LL(1)**

ESEMPIO 1

Produzioni senza ε -rules:

```
S →   A B | B
A →   a A
B →   b B | c
```

Invece, nell'esempio a lato:

- **gli starter set di A e B sono gli stessi, ma..**
- **..stavolta *A può sparire*: ergo,**
  lo starter set di `A` non caratterizza più completamente la produzione `S→AB`
- ***Per avere la visione completa bisogna considerare anche `B`,* che determina l'iniziale quando `A` manca.**

ESEMPIO 2

Produzioni **con** ε -rules:

```
S →   A B | B
A →   a A | ε
B →   b B | c
```

```
FIRST(S→AB)= {a,b,c}
FIRST(S→B)  = {b,c}
```

Aggiungendo la stringa vuota

- Si consideri la seguente grammatica:

S → AB | C   START(S) = {   }
A → aA | ε   START(A) = {   }
B → bBb | c   START(B) = {   }
C → ε   START(C) = {   }

	a	b	c	
S				
A				
B				
C				

Aggiungendo la stringa vuota

- Si consideri la seguente grammatica:

S → AB | C
A → aA | ε
B → bBb | c
C → ε

Aggiungendo la stringa vuota

- Se la grammatica comprende la stringa vuota, significa che alcune produzioni possono annullarsi

$$A \rightarrow \varepsilon$$

- In tal caso, dovremo inserire la regola $A \rightarrow \varepsilon$ nella parsing table in corrispondenza del nonterminale A e di tutti i simboli che possono seguire A
- Visto che A può annullarsi, lo stesso vale per eventuali regole $X \rightarrow A$, $X \rightarrow AA$, $X \rightarrow AX$, ecc., e così via ricorsivamente
- Ha senso costruire l'insieme **FOLLOW**(A) dei simboli che possono seguire A
- Oltre ai simboli dell'alfabeto, va considerato anche il terminatore $

Creazione tabella di parsing

- Per scrivere un riconoscitore, in genere si inserisce un terminatore, spesso identificato col simbolo **$**, nella stringa da riconoscere.
- Si creano poi gli insiemi FIRST (o SS) e FOLLOW per ciascun non-terminale
- Infine, si costruisce la tabella di parsing

Calcolare l'insieme FOLLOW

Per calcolare **FOLLOW**(A) per ciascun non-terminale A, si applicano le seguenti regole finché non è più possibile aggiungere nulla:

- Si inserisce $ in **FOLLOW**(S) (se S è lo scopo)
- Se c'è una produzione $A \rightarrow \alpha B \beta$ allora tutto ciò che è in **FIRST**(β) (eccetto la stringa vuota ε) viene inserito in **FOLLOW**(B)
- Se c'è una produzione
  - $A \rightarrow \alpha B$ oppure
  - $A \rightarrow \alpha B \beta$, dove β si può riscrivere in ε,

  allora tutto ciò che è in **FOLLOW**(A) viene ricopiato in **FOLLOW**(B)

Costruire la Parsing Table

Per ogni regola di produzione $A \rightarrow \alpha$

- per ogni terminale **a** in FIRST(α), aggiungere $A \rightarrow \alpha$ ad M[A,**a**]
- se $\alpha \rightarrow^{+} \varepsilon$,
  - allora per ogni terminale **b** in FOLLOW(A), aggiungere $A \rightarrow \alpha$ ad M[A,**b**]
- se $\alpha \rightarrow^{+} \varepsilon$ e **$** è in FOLLOW(A)
  - allora aggiungere $A \rightarrow \alpha$ ad M[A,**$**]

Un metodo semplice per verificare se $\alpha \rightarrow^{+} \varepsilon$ è
•inserire ε fra i possibili simboli in FIRST(A)
•Qui, verificare solo se ε è in FIRST(α)

Calcolare gli starter symbol set (versione con ε)

Per calcolare **FIRST**(X) si applicano le seguenti regole finché non è più possibile aggiungere nulla:

- Se X è un terminale, allora **FIRST**(X)={X}
- Se X è un non-terminale e c'è una regola

$$X \to Y_1, Y_2, \ldots, Y_k$$

  allora aggiungi il non-terminale **a** a **FIRST**(X) se
  - **a** ∈ **FIRST**(Y_i) e
  - per tutti gli Y_j con $j<i$, ε ∈ **FIRST**(Y_j)

  Se per tutti gli Y_j, **FIRST**(Y_j) contiene ε, aggiungi ε a **FIRST**(X)
- Se X → ε allora aggiungi ε a **FIRST**(X)

Esercizio (29 giu 2016)

- Si consideri la seguente grammatica:

  S → Ax | yB

  B → ε | zB

  A → ε | BaS
- 1. Si classifichi la grammatica secondo Chomsky.
- 2. La grammatica è LL(1)? Se sì, si scriva la parsing table del PDA riconoscitore.
  Se no, si motivi il perché.
- Qualora nei punti precedenti si sia riusciti a ottenere un riconoscitore, si mostri come l'automa riconosce le stringhe **`ayzx`** e **`zaxx`**, mostrando l'evoluzione dello stack.

DIRECTOR SYMBOL SET

Definiamo *Director Symbol Set* della produzione A→α l'unione di due insiemi:

- **lo *starter symbol set***
- **il nuovo *following symbol set:***

$$DS(A\to\alpha) = FIRST(\alpha) \cup FOLLOW(A) \quad \text{se } \alpha \stackrel{*}{\Rightarrow} \varepsilon$$

dove FOLLOW(A) denota l'insieme dei simboli che, *nel caso A generi ε* , possono *seguire* la frase generata da A:

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

NB: durante la derivazione possono apparire fra **A** e **a** dei simboli non-terminali, che però scompaiono successivamente trasformandosi in ε .

DIRECTOR SYMBOL SET

In pratica,

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

Questo permette di riformulare la condizione LL(1) come segue:

CONDIZIONE NECESSARIA E SUFFICIENTE* perché una grammatica sia LL(1) è che i director symbol set relativi a produzioni alternative siano *disgiunti.

RIPRENDENDO L'ULTIMO ESEMPIO…

Riconsideriamo il caso a lato:

- Come già visto, gli *starter set* relativi alle parti destre delle due produzioni di A sono:
  `FIRST(PQ)={p,q}` `FOLLOW(BC)={b,e}`
- Anche se disgiunti, non si può concludere che la grammatica sia LL(1) perché vi sono ε -rules
- **Infatti, i *director symbol set* risultano:**
  `DS(A → PQ)={p,q,b,e}`
  `DS(A → BC)={b,e}`

che non sono disgiunti → *non è LL(1)*

IL SOLITO ESEMPIO

Produzioni:

```
S →   A B
A →   P Q | B C
P →   p P | ε
Q →   q Q | ε
B →   b B | e
C →   c C | f
```

Infatti, poiché A compare solo nella produzione **S → A B**, se A genera ε , i simboli che possono seguire la frase generata da A sono quelli relativi a B, ossia `{b,e}`.

LINGUAGGI LL(1)

- **Le grammatiche «più immediate» di un linguaggio *possono non essere LL(1)*** – ad esempio, se contengono ricorsioni sinistre.
- **Dunque, *come si fa a stabilire* se il linguaggio generato da una grammatica sia LL(1)?**

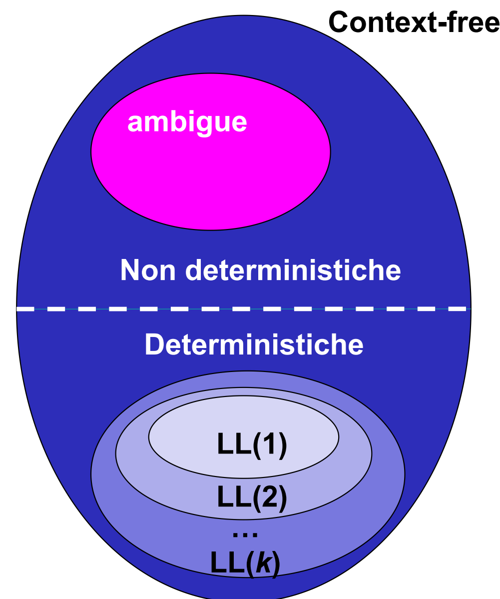
Valgono le seguenti proprietà:

- **Stabilire se una grammatica sia LL(1) è un problema *decidibile***
- **Stabilire se un linguaggio sia LL(1) è un problema *indecidibile***
- **Non tutti i linguaggi context-free possiedono una grammatica LL(1)**
- **Se L è un linguaggio context-free ma il suo complemento non lo è, allora L è nondeterministico.**

In pratica è tuttavia spesso possibile trovare una grammatica equivalente LL(1) applicando le due note tecniche di *eliminazione della ricorsione sinistra* e *raccoglimento a fattor comune.*

Grammatiche context-free

- Le grammatiche ambigue, chiaramente, sono nondeterministiche:
  - se ci sono due alberi di parsing diversi per la stessa frase, vuol dire che da qualche parte nella grammatica posso fare una scelta non deterministica
- Nelle grammatiche LL(k) riesco a scegliere la regola da utilizzare guardando i primi k simboli, quindi sono deterministiche



IL PROBLEMA DELLA RICORSIONE SINISTRA

PROBLEMA:
la ricorsione sinistra non va d'accordo con il requisito LL(1)

- Motivo: le produzioni ricorsive a sinistra, della forma **A ::= Aα | a**, *danno sempre luogo a starter symbol set identici* per le due alternative

Fortunatamente, la ricorsione sinistra si può sempre eliminare..

- **..MA così facendo la grammatica cambia** (diventa ricorsiva a destra) e con essa ***cambia l'albero di derivazione*** di una frase

Importa?

- *NO,* a livello di linguaggio riconosciuto (le frasi lecite restano le stesse)
- *SÌ, se si sfrutta l'albero per inserire azioni semantiche*
  albero diverso = semantica diversa!

  Potremmo non poterci permettere questa differenza!

RACCOGLIMENTO A FATTOR COMUNE

Questa tecnica *non sempre consente di rendere la grammatica LL(1)* poiché la sua applicabilità generale contrasta con l'indecidibilità dei linguaggi LL(1). È tuttavia applicabile in svariati casi pratici.

La tecnica consiste nell' isolare il prefisso più lungo comune a due produzioni: ciò può, in certi casi, rendere la grammatica LL(1).

La grammatica a lato chiaramente non è LL(1).
- **Però, è possibile isolare il prefisso** a S **comune alle prime due produzioni**

ESEMPIO
S → a S b | a S c
S → ε

- **Raccogliamo dunque il prefisso comune…**
- **e introduciamo un nuovo metasimbolo** X **per esprimere la parte che segue il prefisso comune.**

ESEMPIO
S → a S X | ε
X → b | c

IN QUESTO CASO, la grammatica ottenuta è LL(1).

RACCOGLIMENTO A FATTOR COMUNE

Questa tecnica non funziona nei casi in cui il raccoglimento porta a un ciclo di riscritture senza fine.

- **La grammatica a lato non è LL(1), poiché DS(A → Bb) = { d,f } e DS(A → Cc) = { d,h }**
- **Apparentemente non c'è prefisso comune, tuttavia si può provare a sostituire B e C**

CONTROESEMPIO
A → B b | C c
B → d B e | f
C → d C g | h

- **La prima regola** (che conteneva due alternative) **si suddivide perciò in quattro alternative**
- **Nelle prime si può tentare il raccoglimento:**

CONTROESEMPIO
A → d B e b | d C g c
A → f b | h c

- **Sfortunatamente, per E sorge lo stesso problema che c'era inizialmente per A**
- **Tentare di raccogliere ancora non risolve nulla: il problema si ripropone ulteriormente.**

CONTROESEMPIO
A → d E | f b | h c
E → B e b | C g c

Esercizio

- Si consideri la seguente grammatica:

```
S -> S B | y
B -> B x | A x
A -> z | z S y
```

- Si dica se la grammatica è LL(1), motivando opportunamente
- Qualora non lo sia, modificare la grammatica per renderla LL(1)
- Scrivere la parsing table
- Mostrare l'esecuzione dell'algoritmo di parsing con parsing table mostrando l'evoluzione dello stack e dell'input quando la stringa di ingresso è "**yzxzyyxx$**".
- Mostrare l'albero di derivazione (parse tree) per la stessa frase (eventualmente nella grammatica modificata. Facoltativo: anche nella grammatica originaria)

21 luglio 2016

- Si consideri la grammatica G = <{a, b, d, f }, {A, B, C}, P, A>
- A → AB | a
- B → bC | d
- C → dCa | f
- Si classifichi la grammatica secondo Chomsky.
- La grammatica è LL(1)? Se sì, si scriva la parsing table del PDA riconoscitore. Se no, si motivi il perché.
- Qualora la grammatica non sia LL(1), se ne scriva una equivalente G ′ che sia LL(1)
- Si scriva la parsing table associata alla grammatica G ′ . Qualora non si sia ottenuta una grammatica LL(1), si scriva comunque la parsing table, evidenziando i conflitti
- Qualora nei punti precedenti si sia riusciti a ottenere uno o più riconoscitori deterministici, si mostri come gli automi riconoscono le stringhe abdfad e add, mostrando l'evoluzione dello stack.

OLTRE L’ANALISI LL(k)

Riassumendo:

- Le grammatiche LL(k) consentono l’analisi deterministica delle frasi *Left to right,* con *Left-most derivation* e usando *k simboli di lookahead*
- Non tutti i linguaggi context-free possiedono una grammatica LL(k)
- **Esistono però tecniche più potenti dell'analisi LL:** le grammatiche LR(k) consentono l’analisi deterministica delle frasi *Left to right,* con *Right-most derivation* e usando *k simboli di lookahead*
- L’**analisi LR è *meno naturale* dell'analisi LL ma è *superiore dal punto di vista teorico:* «arriva dove l’LL non arriva»**
- Alcuni linguaggi context-free che *NON* sono analizzabili in modo deterministico con tecniche LL *sono invece riconoscibili in modo deterministico con tecniche LR*