GOOD
OOD

Linguaggi di programmazione

Prof. Roberto Gorrieri (I modulo)

e

Prof. Maurizio Gabbrielli e

Dr. Saverio Giallorenzo (II modulo)

Corso annuale (12 crediti) diviso in due moduli

facciamo le presentazioni…

- Roberto Gorrieri
  - Professore di Informatica
  - Laurea in Scienze dell'Informazione, Pisa, 1986
  - Dottore di Ricerca in Informatica, Pisa, 1991
  - Insegnato sempre a Bologna e Cesena, prima come ricercatore (dal 1991), poi come Prof. Associato (dal 1992), quindi come Prof. Ordinario (dal 2000)

- Interessi di ricerca:
  - Metodi Formali per la specifica, l'analisi e la verifica di sistemi concorrenti e distribuiti (corso di Modelli e Sistemi Concorrenti alla Laurea Magistrale): automi, linguaggio CCS, logiche modali e temporali, reti di Petri --- *naturale prosecuzione di alcuni temi di questo modulo*
  - Metodi Formali per la Sicurezza: flusso d'informazioni, analisi di protocolli di sicurezza.
  - Teoria dei linguaggi di programmazione: semantica.

facciamo le presentazioni…

- Maurizio Gabbrielli
  - Professore di Informatica
  - Laurea in Scienze dell'Informazione, Pisa, 1988
  - Dottore di Ricerca in Informatica, Pisa, 1993
  - Ricercatore al CWI Amsterdam 1993-95
  - Ricercatore Pisa (1995-98), associato a Udine (1998-2001), ordinario a Bologna (2001-)
  - Attualmente direttore del Dipartimento di Informatica

- Interessi di ricerca:
  - Teoria dei linguaggi di programmazione
  - Linguaggi logici e con vincoli
  - Intelligenza Artificiale (in particolare, metodi di machine learning per i linguaggi di programmazione)

facciamo le presentazioni…

- Saverio Giallorenzo
  - Ricercatore a Tempo Determinato, Univ. di Bologna, 2020
  - Assegnista di Ricerca, Univ. of Southern Denmark, 2018-20
  - Assegnista di Ricerca, Univ. di Bologna, 2016-17,
  - Dottore di Ricerca in Informatica, Univ. di Bologna, 2016
  - Laurea Magistrale in Science di Internet, Univ. di Bologna, 2012

- Interessi di ricerca:
  - Linguaggi di Programmazione Orientati ai Servizi (microservizi, serverless)
  - Teoria di Linguaggi di Programmazione (specifiche formali, invarianti, analisi)
  - Design e Sviluppo di Linguaggi (esempi: Jolie, Choral, APP, AIOCJ)
  - Sicurezza (architetture software e contrasto malware

Info

- Email: roberto.gorrieri@unibo.it
  maurizio.gabbrielli@unibo.it
  saverio.giallorenzo2@unibo.it
- Ricevimento:
  - Dopo lezione (o, meglio, per appuntamento via email)
- Web: https://virtuale.unibo.it/course/view.php?id=35781
- Ma anche (per i compiti vecchi)
  - www.cs.unibo.it/~gorrieri/LP/lp.html

    login: esame

    password: linguaggi

  https://saveriogiallorenzo.com/teaching/#pl
- Altri Web

  www.cs.unibo.it/~gabbri/corsi/linguaggi.html

  www.cs.unibo.it/~martini/PP/LP-index.html

  www.unibo.it/sitoweb/gianluigi.zavattaro

Modalità d'esame

- Appelli regolari: (giugno (2), luglio, settembre, gennaio, febbraio)
  - scritto + orale (da svolgersi *nella stessa sessione*)
  - Allo scritto non è consentita la consultazione di materiale.
  - Ci si presenta ad entrambe le prove muniti del tesserino universitario (e possibilmente di un documento d'identità)
  - Sono ammessi all'orale i candidati con voto allo scritto >= 18.

- Iscrizione obbligatoria agli appelli attraverso almaesami.

- Per agevolare l'orale: lista di domande possibili, in un file pdf sul sito del corso (solo per la mia parte).

Modalità d’esame (2)

- Parziali
  - Compitini al termine di ciascuna delle due parti. Se in entrambi i parziali viene ottenuta una valutazione >= 16 e la somma delle due valutazioni è >= 36 non e' necessario fare lo scritto. Orale conseguente da fare nella sessione estiva.

- Parziale 1 di fine modulo 1:
- Martedì 20 dicembre 2022, ore 10:00-12:30 in Ercolani 1 (o online, se la situazione peggiorasse)
- Iscrizione obbligatoria ai parziali attraverso almaesami.
- Ricordo che sono ammessi al parziale 2 di fine maggio 2023 solo coloro i quali abbiano superato il parziale 1 (con voto >=16)

Obiettivi, 1

Computer Science is no more about computers than Astronomy is about telescopes. -- E.W. Dijkstra (1930 – 2002); Premio Turing 1972.

A Good Programming Language is a Conceptual Universe for thinking about Programming. -- Alan Perlis (1922 – 1990); Primo premio Turing 1966; primo presidente ACM; primo direttore CS Dpt a CMU

Obiettivi, 2

- Conoscere decine di linguaggi diversi?
  - Conoscere meccanismi comuni a centinaia di linguaggi!
- Per evitare l'invecchiamento della tecnologia...
  - Competenza trasversale sui LP
    - Imparare cosa è importante riguardo a vari linguaggi
    - Comprendere idee e metodologie di programmazione
    - Comprendere i linguaggi che usate (C, C++, Java) per confronto con altri linguaggi
    - Comprendere la storia dei concetti e loro evoluzione
    - Conoscere il costo dei diversi meccanismi
  - Pensiero critico
    - Proprietà dei linguaggi, non documentazione

Gli attrezzi dell'informatico

C

Altri linguaggi

Se hai solo un martello, tutto ti sembra un chiodo

Programma del *corso* -- *Gorrieri*

- Evoluzione dei linguaggi di programmazione: Dai linguaggi assembler ai linguaggi ad alto livello.
- Macchine astratte, interpreti, compilatori.
- Compilatori: struttura generale (fasi/moduli in cui è organizzato)
- Come descrivere un linguaggio: sintassi, semantica, pragmatica
- Sintassi (BNF) e semantica (tecnica SOS).
- Grammatiche regolari, automi a stati finiti, espressioni regolari: equivalenze e risultati principali (ad esempio, minimizzazione).
- Scanner: costruzione di analizzatori lessicali (cenni a lex).
- Grammatiche libere da contesto, automi a pila: equivalenze e risultati principali (ad esempio, pumping theorem).
- Grammatiche libere deterministiche: algoritmi di riconoscimento e costruzione dell'albero di derivazione; grammatiche LL(1), LR(0), SLR(1), LR(1), LALR(1).
- Parser: costruzione di analizzatori sintattici (cenni a yacc).
- Fondamenti: esistono vincoli che un compilatore non può verificare; proprietà indecidibili; Macchine di Turing (cenni).

Prosecuzione:

- Compilatori e Interpreti (LM: Laneve)
- Informatica Teorica (LT: Asperti)

Programma del *corso* -- *Gabbrielli e Giallorenzo*

- Nomi: blocchi e ambiente; regole di scope (statico vs dinamico)
- Gestione della memoria: statica, dinamica; pila e heap; regole di scoping; garbage collection.
- Strutturare il controllo: espressioni e comandi
- Astrarre sul controllo: funzioni e procedure.
- Parametri e modalità di passaggio: per valore, per riferimento, per risultato, per nome. Parametri funzionali. Chiusure. Eccezioni
- Strutturare i dati: tipi di dato, sistemi di tipo, controlli.
- Astrarre sui dati: tipi di dato astratti
- Paradigma orientato agli oggetti: classi e oggetti, ereditarietà, selezione dinamica dei metodi, astrazione.
- *Cenni al paradigma funzionale: funzioni di ordine superiore, strategie di valutazione, tipi, lambda-calcolo, cenni a Scala*
- *Cenni al paradigma logico: unificazione, pattern-matching, risoluzione, semantica operazionale, backtracking, cenni a Prolog*
- *Cenni alla programmazione concorrente e service oriented: Jolie.*

Prosecuzione:

- *Emerging Programming Paradigms (LM: Sacerdoti Coen) Si studiano linguaggi quali Go, Scala, Erlang, Rust*
- *Scalable and Cloud Programming (LM: Zavattaro) Scala ed altri linguaggi*

Libro di testo di riferimento (per i due moduli)

Approfondimenti sul libro di testo

Materiale didattico aggiuntivo (approfondimenti nel libro) non stampato nel libro:

sulla pagina web

http://www.ateneonline.it/gabbrielli/

Capitoli rilevanti per Modulo 1

- Capitolo 1: Macchine Astratte
- Capitolo 2: Descrivere un linguaggio di programmazione (parte non sul libro: semantica operazionale – sul sito ci sono i miei appunti a mano)
- Capitolo 3: Analisi Lessicale: Linguaggi regolari
- Capitolo 4: Analisi Sintattica: Linguaggi liberi
- Capitolo 5: Fondamenti (lezione conclusiva, come introduzione al corso di Informatica Teorica)

- Seguite le lezioni perché non tutto è sul libro, e in più a lezione faccio molti esempi ed esercizi in più di quanto c'è sul libro.

Libri di testo complementari per il Modulo 1

Altri libri di testo complementari per il Modulo 1

Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
Compilatori Principi, tecniche e strumenti - 2/Ed.
Pearson

Libro di testo (versione precedente)

Vecchio libro di testo del corso:

M. Gabbrielli, S. Martini, *Linguaggi di programmazione: principi e paradigmi*. McGraw-Hill Italia, 2005.

Refusi: www.cs.unibo.it/~martini/PP/errata.html

- Ne esiste anche una versione in Inglese, edita da Springer.
- Non contiene materiale per il modulo 1 su grammatiche, ma copre tutto il modulo 2

Orario delle lezioni

Da Martedì 20 Settembre 2022 a Venerdì 16 Dicembre 2022 (venerdì 23/09 in G1)

Ora inizio	lun	mar	mer	gio	ven
11:00-12:00					
12:00-13:00		M2			
13:00-14:00		M2			M2/G1
14:00-15:00					M2/G1
15:00-16:00					M2/G1
16:00-17:00					

Cosa dovete sapere

- aspetti propriamente linguistici (quale paradigma, quali costrutti e perchè);
- come i costrutti linguistici possono essere implementati ed il relativo costo;
- aspetti architetturali che influenzano l'implementazione;
- tecniche di traduzione (compilazione).

(compilatori)

Architetture

Perché studiare i linguaggi in generale?

- Progettare un nuovo linguaggio?
  - non così raro: linguaggi specifici per classi di applicazioni
  - una GUI è un linguaggio per la sua applicazione
- Imparare più facilmente un nuovo linguaggio
  - costrutti già noti e meccanismi semantici di base
- Riassumendo: non imparare un linguaggio specifico ma imparare i principi generali dei linguaggi

Competenze da acquisire

- Competenza tecnica (scritto)
  - Esercizi
- Competenza linguistica (*orale*!)
  - correttezza terminologica
  - completezza di esposizione
  - Per aiutarvi, metterò sul sito del corso un elenco di domande frequenti che vengono fatte all'orale (per la mia parte)
- Competenza implementativa
  - come è implementato il meccanismo X?
  - collegamenti con i corsi di Architetture, Programmazione e Sistemi Operativi.
- Competenza critica
  - A language that doesn't affect the way you think about programming, is not worth knowing. (A.Perlis)

Come studiare

- Seguire *attivamente* le lezioni (domande, commenti, critiche)
- Studiare (subito) sul manuale
  - competenza tecnica
- Esercizi scritti
  - per verificare la competenza tecnica
- Ripetere (da soli o con altri)
  - per memorizzare
  - per migliorare la competenza linguistica
- Ri-studiare il manuale da capo
  - chiedersi:
    - come implemento questo meccanismo?
    - di quali informazioni ho bisogno per tale implementazione?
    - quanto costa questo meccanismo?
- Andare avanti e indietro sul manuale
  - quali collegamenti?
  - quali problemi?
  - competenza critica... *A language that doesn't affect the way you think about programming, is not worth knowing. (A.Perlis)*

Dai linguaggi hw ai meccanismi di astrazione

- Anni 1950--60:
  - Compilazione dei programmi per ottenere efficienza
  - Connessione diretta fra hw e linguaggi: integers, reals, goto
  - *Programmers cheap, machines expensive –> Keep the machine busy !*

- Oggi:
  - Compilazione di programmi costruiti in modo efficiente
  - Connessione diretta fra sw design e linguaggio: encapsulation, records, ereditarietà, dichiaratività
  - *Programmers expensive, machine cheap –> Keep the programmer busy!*

Evoluzione dei linguaggi

- Anni 40-50 Preistoria: (linguaggio macchina e assembler)
- Anni 50-60: Primi linguaggi alto livello:
  - FORTRAN (Backus 1957) per calcolo numerico-scientifico (su IBM704): uso di notazione matematica in espressioni (es. i+2*j)
  - ALGOL (58, 60, W) (Naur 1958) come linguaggio universale, per esprimere algoritmi: Indipendenza dalla macchina, vicinanza con la notazione matematica, call by name, funzioni ricorsive, type systems, data structures (anche dinamiche)
  - LISP (McCarthy 59-60) per intelligenza artificiale (List Processor): funzioni su una certa classe di espressioni simboliche (S-Expressions), ordine superiore (antesignano del paradigma funzionale)
  - COBOL (59-60) per applicazioni commerciali (record, archivi)
  - ALGOL 68 linguaggio barocco, complessa progettazione, ma contiene anche idee importanti (struttura a blocchi)
  - SIMULA (Nygaard & Dahl): da ALGOL, primo linguaggio con classi e oggetti (antesignano del paradigma object-oriented)

Evoluzione dei linguaggi II

- Anni 70:
  - PL/I (IBM): Fortran + COBOL, primo esempio di linguaggio multi-uso (PL deriva da Multi-Purpose Prog. Lang.) Scarso successo
  - Pascal (Wirth): Evoluzione (e semplificazione) di Algol W. Essenzialmente linguaggio didattico fino agli anni 80 (poi rimpiazzato da C e Java). Portabilità grazie a codice intermedio (P-Code)
  - C (Ritchie): Programmazione di sistema (Unix). Ha avuto notevole successo fino ai nostri giorni.
  - Prolog: (Colmerauer, Kowalski): uso esplicito della logica come base di un linguaggio di programmazione (primo linguaggio logico)
  - SmallTalk: Incapsulamento, oggetti e classi. O-O.
  - ML (Milner): Funzionale. Nato come Meta Linguaggio per un sistema semi-automatico di prova di proprietà dei programmi. Sistema di tipi evoluto.

Evoluzione dei linguaggi III

- Anni 80:
  - ADA: Linguaggio real-time di US Dep. Of Defense. Linguaggio per sviluppo di programmi molto grandi da parte di team. Include quasi tutto quello che esisteva al tempo, inclusi costrutti per il controllo della concorrenza
  - Postscript: page description language nell'area di desktop publishing elettronico
  - C++ Estensione Object Oriented di C
  - Standard ML
  - CLP (linguaggi logici con vincoli). Linguaggi che permettono di manipolare relazioni su opportuni domini. Usati per problemi combinatori, ottimizzazione, intelligenza artificiale etc.

Evoluzione dei linguaggi IV

- Anni 90:
  - Java (Goslin): Altamente portabile; object-oriented; programmi spediti sulla rete (applet). Elevata portabilità (Write once, run anywhere!) --> Java Virtual Machine e ByteCode
  - PERL: Linguaggio per applicazioni di text-processing nato come linguaggio di shell-scripting in UNIX
  - HTML linguaggi di marcatura per ipertesti (usato per creare pagine web)
  - XML linguaggio per la rappresentazione di dati semi-strutturati (ad esempio, pagine web).

Evoluzione dei linguaggi V

- Anni dal 2000 in poi:
  - service-oriented computing: paradigma che usa i "servizi" (di solito web services), come unità di base di computazione, per progettare ed implementare applicazioni integrate di business.
  - WSDL: web services description lamguage
  - WS-BPEL: Web Services Business Process Execution Language
  - **Jolie *(sviluppato a Bologna!)***

  - **Agent Based Programming** (vedi corso di Sacerdoti Coen in LM) Go, Erlang, Rust, ecc..
  - **Scalable Cloud Programming** (vedi corso di Zavattaro) – Scala e Spark

Linguaggi imperativi e linguaggi dichiarativi

- Linguaggi imperativi
- Basati sulla nozione di stato (insieme di locazioni di memoria contenenti dei valori)
- Le istruzioni sono **comandi** che cambiano lo stato.
- Es. X:= X+1; X:= X+2 …
- C, Pascal, FORTRAN, COBOL …
- Analogia con le frasi *imperative* dove il soggetto è implicito (e.g. taglia quella mela)

- Linguaggi dichiarativi
- Basati sulla nozione di funzioni o relazione.
- Le istruzioni sono **dichiarazioni** di nuovi valori, in modo diretto oppure per composizione di funzioni o relazioni.
- Es. (fun(X). X+2) 3;
- Analogia con le frasi *dichiarative* (e.g., quella mela è tagliata)

Linguaggi dichiarativi

- Linguaggi funzionali: Lisp, Scheme, ML, Haskell
  - Basati sulla nozione di funzione: risultato del programma = valore esplicito di una espressione
  - applicazione e definizione di funzioni, ricorsione
  - Programmare = costruire la funzione che calcola il risultato

- Linguaggi logici (e con vincoli): Prolog, CLP
  - Basati su relazioni: risultato = valore di alcune variabili determinato da alcune relazioni
  - Istruzioni = implicazioni logiche fra opportune formule, che possono essere viste come regole di riscrittura
  - Programmare = specificare la relazione che definisce il valore delle variabili di interesse (sostanzialmente ``dire come è fatta la soluzione'')

Un semplice esempio: calcolo del Fattoriale

PASCAL *(imperativo)*

```
function Fatt (n: integer): integer;
    var
        i, acc: integer;
    begin
        acc:=1;
        for i:=1 to n do
                acc:=acc*i;
        Fatt:=acc;
    end;
```

Un semplice esempio: calcolo del Fattoriale (2)

SCHEME *(funzionale)*

```
(define (fatt n)
    (cond [(= n 0)  1]
          [else  (*  n  (fatt (- n 1)))]
    ))
```

Un semplice esempio: calcolo del Fattoriale (3)

PROLOG *(logico)*

```
Fatt(0, 1).
Fatt(N, Z) :- Fatt(N-1, Y), Z is Y*N.
                                Prod(Y, N, Z).

Prod(0, X, 0).
Prod(N+1, X, Z) :- Prod(N, X, Y), Sum(Y, X, Z).
Sum(0, X, X).
Sum(Y+1, X, Z+1) :- Sum(Y, X, Z).
```

Linguaggi orientati agli oggetti

- Linguaggi orientati agli oggetti: Java, C++, Smalltalk
  - linguaggi imperativi con alcune metodologie di progettazione dichiarative
  - Oggetti (istanze di opportune classi) che contengono i dati (concetti imperativi) e metodi come funzioni per operare su tali oggetti (concetti dichiarativi)
  - Incapsulamento
  - Ereditarietà
  - … molto altro più avanti

Storia dei linguaggi e dei paradigmi -- Timeline

Link utili – da Wikipedia English

- Per un elenco di (quasi) tutti i linguaggi di programmazione esistenti

  https://en.wikipedia.org/wiki/

  List_of_programming_languages

  (non è possibile conoscerli tutti!)

- Per una timeline piu' articolata e completa

  https://en.wikipedia.org/wiki/

  Timeline_of_programming_languages

- Per un confronto tra le dozzine di linguaggi di programmazione …..

  https://en.wikipedia.org/wiki/

  Comparison_of_programming_languages

Come confrontare linguaggi?

Caratteristiche intrinseche (dipendenti dalla sua sintassi e classe di primitive che mette a disposizione):

- Espressività: cosa/quanto può calcolare? Turing-completezza come criterio assoluto per i linguaggi sequenziali
- Didattica: quanto sono semplici le primitive e quanto velocemente lo si può apprendere
- Leggibilità: quanto è facile, leggendo un programma, capire cosa fa.
- Robustezza: la capacità del linguaggio di prevenire, nei limiti del possibile, gli errori di programmazione (ad esempio, perchè è tipato).

Come confrontare linguaggi? (2)

Altre caratteristiche intrinseche :

•Generalità: la facilità con cui il linguaggio si presta a codificare algoritmi e soluzioni di problemi in campi diversi.

•Efficienza: la velocità di esecuzione e l'uso oculato delle risorse del sistema su cui il programma gira. In genere i programmi scritti in linguaggi molto astratti tendono ad essere lenti e voraci di risorse; in compenso facilitano molto la vita del programmatore, accelerando lo sviluppo di nuovi programmi ed eliminando intere classi di errori di programmazione possibili. Viceversa un linguaggio meno astratto ma più vicino alla reale struttura di un computer genererà programmi molto piccoli e veloci ma a costo di uno sviluppo più lungo e difficoltoso.

Come confrontare linguaggi? (3)

Caratteristiche esterne (dipendenti dall'ambiente di lavoro)

•Diffusione: più è numerosa la comunità per un linguaggio, tanto più è facile trovare materiale, aiuto, librerie di funzioni, documentazione, consigli. Inoltre ci sono un maggior numero di software house che producono strumenti di sviluppo per quel linguaggio, e di qualità migliore.

•Standardizzazione: un produttore sente spesso la tentazione di introdurre delle variazioni sintattiche o delle migliorie ad un linguaggio, originando un *dialetto* del linguaggio e fidelizzando così i programmatori al suo prodotto: ma più dialetti esistono, più la comunità di programmatori si frammenta in sottocomunità più piccole e quindi meno utili. Per questo è importante l'esistenza di uno standard ufficiale, di solito definito dall'ANSI (American National Standards Institute) o dall'ISO (International Organization for Standardization).

Come confrontare linguaggi? (4)

Altre caratteristiche esterne

- Integrabilità: dovendo scrivere programmi grandi, è facile trovarsi a dover integrare parti di codice precedente scritte in altri linguaggi: se un dato linguaggio consente di farlo facilmente, questo è decisamente un punto a suo favore.
- Portabilità: la possibilità che portando il codice scritto per una certa **piattaforma** (CPU + architettura + sistema operativo) su un'altra piattaforma, questo funzioni subito, senza doverlo modificare. Questa è, ad esempio, una delle motivazioni principali per l'introduzione del linguaggio Java, che è altamente portabile grazie alla Java Virtual Machine (JVM).

Macchine astratte

Interpreti e compilatori

Linguaggi, macchine astratte, implementazioni

- **Argomenti**:
  - Macchine astratte e linguaggi
  - Gerarchie di macchine astratte e di linguaggi
  - Implementare un linguaggio: Interpreti e Compilatori

la macchina di von Neumann

Una struttura estremamente semplice

...ma anche estremamente flessibile e suscettibile di molte varianti

...e il suo “interprete”

- La macchina esegue un ciclo ripetitivo

programmi e dati sono indistinguibili
e risiedono nella memoria interna

La macchina esiste per eseguire il proprio linguaggio

il processore tipico

può essere complicato a piacere:

architetture parallele

con *piping*

con *prefetch*

Morale

- Una macchina fisica esiste per eseguire il suo linguaggio

- Linguaggio e macchina "vengono assieme", esistono in simbiosi. Ma:
  - una macchina corrisponde ad un (il suo) linguaggio
  - un linguaggio può essere eseguito da più macchine

- Cuore di una macchina fisica:
  - il ciclo fondamentale *fetch-decode-execute*: l'interprete

Una macchina fisica è la realizzazione "a fili" di un particolare algoritmo che, sfruttando alcune strutture dati, è capace di "eseguire" programmi scritti in un certo linguaggio, detto il linguaggio macchina.

Macchina astratta

- Una Macchina Astratta (MA) è un insieme di strutture dati ed algoritmi in grado di memorizzare ed eseguire programmi. Componente essenziale è l'interprete

Interprete

È il componente che *interpreta* le istruzioni ed è costituito da

- operazioni per l'elaborazione dei dati primitivi

  MF: controllo e sfruttamento della ALU

  MACCHINA FISICA

- operazioni e strutture dati per il controllo della sequenza di esecuzione delle operazioni

  MF: incremento del PC; salti

- operazioni e strutture dati per il controllo del trasferimento dei dati

  MF: gestione dei metodi di indirizzamento

- operazioni e strutture dati per la gestione della memoria

  MF: indirizzamento e trasferimento di blocchi ecc.

La *struttura* dell'interprete è la stessa per una qualunque MA; cambiano i diversi componenti di esso

Interprete

Start
Acquisisci la prossima istruzione
fetch
Decodifica
decode
Acquisisci gli operandi
execute
Seleziona
Esegui OP_1
Esegui OP_2
Esegui OP_n
Esegui *HALT*
Stop
Memorizza il risultato

Linguaggio Macchina

- **M** Macchina Astratta
- $\mathbf{L_M}$ Linguaggio Macchina di M
- $\mathbf{L_M}$ è il linguaggio che è “compreso” dall’interprete di **M**:
  - I programmi sono particolari dati primitivi su cui opera l’interprete
- Ai componenti di **M** corrispondono opportuni componenti di $\mathbf{L_M}$
- Diverse rappresentazioni dei programmi scritti in $\mathbf{L_M}$
  - Interna: strutture dati in memoria
  - Esterna: stringa di caratteri

Linguaggio della Macchina HW

- Set di istruzioni direttamente implementate nella ALU:
  - **CISC** (Complex Instruction Set Computers) – complesse (spesso molte)
  - **RISC** (Reduced Instruction Set Computers) – semplici (a volte poche)
- Tipica istruzione a due operandi (occupa una parola di memoria)

  Codice-operativo Operando1 Operando2
- Ad esempio (in qualche assembler – rappresentazione esterna)
  - ADD R5, R0

    somma i contenuti dei registri R5 e R0 e memorizza il risultato in R0
  - ADD (R5) (R0)

    somma i contenuti delle celle di memoria i cui indirizzi sono contenuti nei registri R5 e R0, e memorizza il risultato nella cella il cui indirizzo è in R0
- In realtà, rappresentazione interna:
  - Opportune sequenze di bit, dette parole (codice operativo, modo d'indirizzamento e indirizzi assoluti)

Macchina hw come macchina astratta

La Macchina HW è un tipo (molto concreto…) di MA, il cui interprete esegue:

•Operazioni primitive -> operazioni aritmetico logiche, di manipolazione di stringhe di bit, lettura e scrittura su registri e celle di memoria, input/output

•Controllo sequenza -> Salti, salti condizionali, chiamate di subroutine mediante registro contatore programma e strutture dati per punti di ritorno sottoprogrammi

•Controllo dati -> acquisizione operandi e memorizzazione risultato

- Architettura a registri -> registri indice e indirizzamento indiretto
- Architettura a pila -> gestione pila

•Gestione memoria ->

- Architettura a registri -> niente (memoria statica)
- Architettura a pila -> allocazione e recupero dati sulla pila.

la macchina hw

- Una processore convenzionale è una (forma molto concreta di...) macchina astratta

- Il suo linguaggio è il linguaggio macchina

cambia l'architettura,
cambia l'interprete,
ma non il linguaggio:
$L_M = L_{Mspider}$

Un'architettura più sofisticata può interpretare lo stesso linguaggio macchina

Scatole cinesi

- Alcune macchine hw (e.g. non RISC) sono *microprogrammate*:
  - ciclo fetch-execute di MF non è realizzato in hardware;
  - ogni istruzione di L_{MF} è realizzata mediante istruzioni di più basso livello, dette μistruzioni (micro-istruzioni)
  - interpretata da un μinterprete (micro-interprete)

Scatole cinesi

- Alcune macchine hw (eg non RISC) sono *microprogrammate*:
  - ciclo fetch-execute di MF non è realizzato in hardware;
  - ogni istruzione di L_{MF} è realizzata mediante istruzioni di più basso livello, dette μistruzioni (micro-istruzioni)
  - interpretata da un μinterprete (micro-interprete)

Un programma in linguaggio macchina (in L_{MF}):
- viene interpretato da un interprete (scritto in L_{MAPF})
- a sua volta eseguito dal (μ)interprete (hw) di MAPF

Realizzare una macchina astratta

Possiamo realizzare una MA (con varie combinazioni delle seguenti tecniche):

1) realizzazione in HARDWARE
2) *emulazione* o simulazione via FIRMWARE

1) Teoricamente sempre possibile ma
   - usata solo per macchine di basso livello o macchine dedicate
   - massima velocità
   - flessibilità nulla.

2) strutture dati e algoritmi MA realizzati mediante microprogrammi, che risiedono in una memoria di sola lettura (ROM)
   - macchina ospite (fisica) microprogrammabile
   - alta velocità
   - flessibilità maggiore che HW puro.

Realizzazione mediante sw: l'interprete

3) *interpretazione* o simulazione via SOFTWARE

3) strutture dati e algoritmi della macchina astratta MA realizzati mediante programmi scritti nel linguaggio della macchina ospite MO

- macchina ospite qualsiasi
- minore velocità
- massima flessibilità.

Realizzazione di una macchina astratta

- Nella realtà, la MA viene realizzata su di una MO fisica mediante una combinazione delle tre tecniche viste

- Tre livelli, *non necessariamente ogni livello maschera completamente i livelli sottostanti*

c’era un volta un re

C’era una volta un re, seduto sul sofà, che disse alla sua serva:
“Raccontami una storia!”
E la serva incominciò:

C’era una volta un re, seduto sul sofà, che disse alla sua serva:
“Raccontami una storia!”
E la serva incominciò:

C’era una volta un re, seduto sul sofà, che disse alla sua serva:
“Raccontami una storia!”
E la serva incominciò:

C’era una volta un re, seduto sul sofà, che disse alla sua serva:
“Raccontami una storia!”
E la serva incominciò:

livelli software

La possibilità di realizzare macchine astratte via software
rende possibili macchine il cui linguaggio è sofisticato

mediante il quale linguaggio realizzare macchine
il cui linguaggio sia ancora più sofisticato

mediante il quale linguaggio realizzare macchine
il cui linguaggio sia ancora più sofisticato

mediante il quale linguaggio realizzare macchine
il cui linguaggio sia ancora più sofisticato

...

gerarchia di macchine astratte

un'architettura informatica (*hw* o *sw*) si struttura in una serie di macchine astratte gerarchiche

M_i:
- usa i servizi forniti da M_{i-1} (cioè il linguaggio $L_{M_{i-i}}$)
- per fornire servizi a M_{i+1}(cioè realizzare un interprete per il linguaggio $L_{M_{i+i}}$)
- nasconde (entro certi limiti) la macchina M_{i-1}

Stando al livello i può non essere noto (e in genere non serve sapere...) quale sia il livello 0 (hw)

una gerarchia canonica

esplicitiamo la gerarchia

Usando un word processor, WP:

interazione *diretta solo* con WP

scrivamo
salviamo il file
cerchiamo altri files
ecc.

usando i menù (i *servizi*, o *primitive*) di WP

esplicitiamo la gerarchia, 2

Ma per soddisfare la nostra richiesta:

"salva questo file"

WP richiede al livello inferiore (l'interfaccia del SO)

l'apertura di una finestra per il nome del file

l'interfaccia del SO richiede al livello inferiore (il SO)

la disponibilità di un'area di memorizzazione

il SO richiede al *file system*

un indirizzo logico per il file

il *file system* richiede alla gestione fisica

l'indice di alcuni blocchi sul disco

.....

esempio: applicazione

Esempio: Web application

esempio: protocolli

un'analogia famosa

esempio: i protocolli di internet

Implementare un linguaggio

- Scartiamo la soluzione hw
- Assimiliamo sw e firmware

- Sono dati:
  - un linguaggio $\mathcal{L}$ da implementare
    - cioè di cui realizzare la macchina astratta $M_{\mathcal{L}}$
  - una macchina astratta $Mo_{\mathcal{L}o}$ (macchina *ospite*)
    - col suo linguaggio $\mathcal{L}o$

- Si vuole
  - implementare $\mathcal{L}$ su $Mo_{\mathcal{L}o}$

- Due modi radicalmente diversi…

Preliminari e notazione

- Funzione: $f : A \rightarrow B$

  corrispondenza tra elementi di A e di B: ad ogni elemento $a \in A$ la funzione f associa un solo elemento $b \in B$.

- Funzione *parziale*: $f : A \text{ -o}\rightarrow B$

  corrispondenza che *può* essere non definita su qualche $a \in A$

- $\mathcal{P}_r^{\mathcal{L}}$ indica un programma $\mathcal{P}_r$ scritto nel linguaggio $\mathcal{L}$
- A $\mathcal{P}_r^{\mathcal{L}}$ è associata una funzione parziale $\mathcal{P}^{\mathcal{L}}$ sull'insieme dei dati (non definizione per un certo input = non terminazione del programma per quell'input) che rappresenta la semantica del programma $\mathcal{P}_r^{\mathcal{L}}$ $\quad \mathcal{P}^{\mathcal{L}} : \mathcal{D} \rightarrow \mathcal{D}$

$$\mathcal{P}^{\mathcal{L}}(Input) = Output$$

Esempio: semantica di un prog. sequenziale

Frammento di programma sequenziale

$\mathcal{P}_r^{\mathcal{L}}$ =

```
read(x);
If (x == 1) then print(x)
            else  while true do skip;
```

E sua **semantica** come funzione da input ad output

$\mathcal{P}^{\mathcal{L}}$ (x) = 1 se x = 1, indefinita altrimenti

(N.B.: per programmi concorrenti la questione è più delicata)

Implementazione interpretativa pura

1° TIPO

- $M_{\mathcal{L}}$ è realizzata scrivendo un *interprete* per $\mathcal{L}$ su $Mo_{\mathcal{L}o}$:

- Più formalmente

Definizione 1.3 (Interprete) *Un interprete per il linguaggio $\mathcal{L}$, scritto nel linguaggio $\mathcal{L}o$, è un programma che realizza una funzione parziale*

$$\mathcal{I}_{\mathcal{L}}^{\mathcal{L}o} : (\mathcal{P}rog^{\mathcal{L}} \times \mathcal{D}) \rightarrow \mathcal{D} \quad \textit{tale che} \quad \mathcal{I}_{\mathcal{L}}^{\mathcal{L}o}(\mathcal{P}_{\mathsf{r}}^{\mathcal{L}}, Input) = \mathcal{P}^{\mathcal{L}}(Input). \qquad (1.1)$$

INTERPRETE PER IL LINGUAGGIO $\mathcal{L}$
SCRITTO CON IL LINGUAGGIO $\mathcal{L}_o$

Ovvero l'interprete "calcola la corretta semantica" del programma!

Implementazione compilativa pura

2' TIPO

- 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 $\mathcal{L}a$

Implementazione compilativa pura, 2

- 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

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

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

$$\mathcal{P}^{\mathcal{L}}(Input) = \mathcal{P}c^{\mathcal{L}o}(Input). \tag{1.3}$$

Ovvero il compilatore "preserva la semantica" del programma: il programma originale P_r^L e quello tradotto $P_r c^{L0}$ calcolano la stessa funzione: $P^L = Pc^{L0}$!

Intermezzo

- Cosa indicano le seguenti scritture?
  - $I^{\mathcal{L}0}_{\mathcal{L}1}$ = un interprete scritto in $\mathcal{L}0$ che esegue programmi scritti in $\mathcal{L}1$
  - La macchina ospite è $\mathcal{M}_{\mathcal{L}0}$, mentre la macchina astratta realizzata su di essa dall'interprete è $\mathcal{M}_{\mathcal{L}1}$
  - $I^{\mathcal{L}0}_{\mathcal{L}1}(\mathcal{P}^{\mathcal{L}1}, x)$ = risultato del calcolo del programma $\mathcal{P}$ (scritto in $\mathcal{L}1$) con x come dato in input
  - $I^{\mathcal{L}0}_{\mathcal{L}1}(I^{\mathcal{L}1}_{\mathcal{L}2}(\mathcal{P}^{\mathcal{L}2}, x))$ = risultato del calcolo di $\mathcal{P}$ (scritto in $\mathcal{L}2$) con x come input
  - La macchina ospite è $\mathcal{M}_{\mathcal{L}0}$, sulla quale è realizzata una macchina intermedia $\mathcal{M}_{\mathcal{L}1}$, grazie al primo interprete, mentre la macchina astratta $\mathcal{M}_{\mathcal{L}2}$, grazie al secondo interprete, è in grado di eseguire il programma $\mathcal{P}$.

Intermezzo (2)

- Cosa indicano le seguenti scritture?

- $C^{\mathcal{L}0}_{\mathcal{L}1,\mathcal{L}2}$ = un compilatore scritto in $\mathcal{L}0$ che traduce programmi scritti in $\mathcal{L}1$ in equivalenti programmi scritti in $\mathcal{L}2$
- $I^{\mathcal{L}0}_{\mathcal{L}1}(C^{\mathcal{L}1}_{\mathcal{L}2,\mathcal{L}3}, \mathcal{P}^{\mathcal{L}2}) = \mathcal{P}1^{\mathcal{L}3}$ cioè l'interprete, eseguito sulla macchina ospite $\mathcal{M}_{\mathcal{L}0}$, realizza la macchina astratta $\mathcal{M}_{\mathcal{L}1}$ ed esegue un compilatore (scritto in $\mathcal{L}1$) che traduce il programma $\mathcal{P}$ (scritto in $\mathcal{L}2$) in un equivalente programma $\mathcal{P}1$ (scritto in $\mathcal{L}3$)

Compilazione o interpretazione ?

- Implementazione interpretativa pura:
  - scarsa efficienza della macchina $\mathbf{M}_{\mathcal{L}}$

Ai tempi di esecuzione, vanno aggiunti i tempi necessari alla decodifica (mentre in impl. Compilativa, la traduzione viene fatta prima di eseguire)

Esempio:

Istruzione for (I = 1, I <= n, I = I+1) C;

Decodifica (al momento)

```
P2:
     R1 = 1
     R2 = n
L1:  if R1 > R2 then goto L2
     traduzione di C
     ...
     R1 = R1 + 1
     goto L1
L2:  ....
```

Se la stessa istruzione viene eseguita più volte, più volte l'interprete dovrà decodificarla (ad esempio C sopra)

Compilazione o interpretazione ? (2)

- Implementazione interpretativa pura:
  - buona flessibilità: permette di interagire con l'esecuzione del programma (facile generare strumenti di debugging "dinamici" step-by-step)
  - Più facile da realizzare: il compilatore è di solito molto più complesso
  - Occupa meno memoria, perché non viene effettivamente generato codice da memorizzare (questione poco rilevante oggigiorno, ma che aveva interesse quando le memorie erano molto piccole e costose).

Compilazione o interpretazione ? (3)

- Implementazione compilativa pura:
  - difficile, data la lontananza fra $\mathcal{L}$ e $\mathcal{L}o$
  - buona efficienza:
    - 1) costo decodifica a carico del compilatore
    - 2) ogni istruzione è tradotta una sola volta
  - scarsa flessibilità
  - perdita di info sulla struttura (astrazione) del programma sorgente (ad esempio: se si verifica un errore a run-time potrebbe essere molto difficile capire quale istruzione del programma sorgente lo ha determinato)
  - (occupazione di memoria del codice prodotto)

Nel caso reale

- **Entrambe le componenti coesistono**
  1) Alcune istruzione (es. ingresso/uscita) sono sempre simulate
  2) I programmi devono essere tradotti nella rappresentazione interna o in un codice intermedio

- La macchina *intermedia* ed il suo linguaggio $\mathcal{L}i$:

Implementazione di tipo interpretativo

- L'interprete della macchina intermedia $M_{\mathcal{L}i}$ è sostanzialmente diverso dall'interprete della macchina ospite $M_{\mathcal{L}o}$

Esempio: Java

L_{MA} = Java
L_{Mi} = Bytecode
Mi = JVM
M0 = opportuna macchina che esegue l'interprete della JVM

Implementazione di tipo compilativo

- Interprete della macchina intermedia $M_{\mathcal{L}i}$ = interprete della macchina ospite $M_{\mathcal{L}o}$ + opportuni meccanismi (p.e. I/O, gestione della memoria ecc.)

  =

  supporto a run time di $\mathcal{L}$

Esempio: C

L_{MA} = C
L_{Mi} = linguaggio generato da cc +
supporto per gestione memoria, I/O ecc.
Mi = M0 + interprete per le chiamate del supporto a run-time
M0 = macchina ospite

supporto a run-time

Cosa compilare e cosa interpretare

- In linea di principio:
  - traduzione per quei costrutti di $\mathcal{L}$ che corrispondono da vicino a costrutti di $\mathcal{L}o$;
  - simulazione (cioè: interpretazione) per gli altri.

- soluzione di tipo compilativo
  - privilegia **l'efficienza**
- soluzione di tipo interpretativo
  - privilegia **flessibilità e portabilità**

Linguaggi reali

- Linguaggi tipicamente implementati in modo compilativo
  - **C, C++, FORTRAN, Pascal, ADA**
- Linguaggi tipicamente implementati in modo interpretativo
  - **LISP, ML, Perl, Postscript, Pascal, Prolog, Smalltalk, Java**

Esempi

- **Programmi Macchina Rilocabili**:
  - Implementazione *puramente* compilativa
  - Il traduttore è il caricatore (o link editor):
    - traduce codice macchina rilocabile in codice macchina eseguibile
  - Interprete = interprete HW (o FW)
- Es. P: indirizzi 0-999 — caricatore → P: indirizzi 0-999
- Q: indirizzi 0-1999 → Q: indirizzi 1000-2999

- **Linguaggio ASSEMBLER**
  - Implementazione *puramente* compilativa
  - Il traduttore è l'assemblatore:
    - traduce codice ASSEMBLER (che usa nomi simbolici) in codice macchina (rilocabile)

Esempi

- **FORTRAN (77)**
  - Implementazione di tipo compilativo (non puro!)
  - traduzione fatta da compilatore che produce codice eseguibile della macchina ospite
  - Linker per unire i vari sottoprogrammi
  - Supporto a Run-Time = programmi (in linguaggio di $\mathbf{M_o}$) che simulano I/O e alcune operazioni aritmetiche non fornite da $\mathbf{M_o}$
  - Supporto a Run-Time minimo:
    - no strutture dati dinamiche
    - no ricorsione
    - no gestione dinamica della memoria (Eccetto FORTRAN 90)

- Dunque: FORTRAN è compilato in un linguaggio $\mathcal{L}_I$ la cui macchina astratta estende M_o con il supporto a run-time

Esempi

- **PASCAL e C**
  - Implementazione di tipo compilativo:
  - traduzione fatta da compilatore che produce codice eseguibile della macchina ospite;
  - Supporto a Run-Time per
    - operazioni di I/O
    - gestione (dinamica) memoria (puntatori, heap)
    - gestione controllo sequenza (ricorsione)
    - controllo dati (blocchi in Pascal), sottoprogrammi, ambiente locale dinamico
  - Necessaria estesa simulazione per realizzare queste componenti, usando opportune strutture dati (pile).

Esempi

- **PASCAL**
  - implementazione di tipo interpretativo
  - traduzione fatta da compilatore che produce codice di una macchina intermedia (P-code)
  - macchina intermedia a Pila, con interprete e strutture simili a quelle di M_{pascal}
  - interprete di P-code simulato su macchina ospite
  - più portabile dell'implementazione compilativa

L'interprete e il compilatore si possono sempre realizzare?

- Interprete esegue L_{sorg} ed è scritto in un linguaggio L_{osp}
- Compilatore traduce L_{sorg} in L_{dest} ed è scritto in un linguaggio L_{osp}
- Compilatore preserva la semantica del programma tradotto, quindi preserva l'insieme delle funzioni che il linguaggio L_{sorg} può calcolare!
- Allora è necessario che il linguaggio L_{dest} sia non meno espressivo di L_{sorg}, ovvero che calcoli "almeno quanto" L_{sorg}.
- Stessa cosa deve valere per L_{osp}
- Fondamenti: Macchine di Turing e Turing-completezza
- Tutti i linguaggi di programmazione "veri" sequenziali sono Turing-completi (se forniti di memoria "illimitata"), ovvero sono ugualmente espressivi.
- Vedremo formalismi/linguaggi molto meno espressivi ma che sono utilissimi perché gli algoritmi relativi sono efficienti:
  - Automi finiti (analizzatori lessicali)
  - Automi a pila (analizzatori sintattici)

Come viene generato un compilatore

- **Strumenti automatici**
  - Lex generatore di analizzatori lessicali
  - Yacc generatore di analizzatori sintattici

  data una descrizione formale della sintassi producono codice che riconosce le stringhe sintatticamente legali

- **Implementazione via kernel**

per implementare $\mathcal{L}$ si scrive un interprete o un compilatore in $\mathcal{H}$, sottoinsieme ristretto di $\mathcal{L}$

$C^{\mathcal{H}}{}_{\mathcal{L},\mathcal{L}o}$ compilatore scritto in $\mathcal{H}$ che traduce $\mathcal{L}$ in $\mathcal{L}o$

$I^{\mathcal{H}}{}_{\mathcal{L}}$ interprete per $\mathcal{L}$, scritto in $\mathcal{H}$

Si implementa poi a mano un compilatore (o un interprete) per $\mathcal{H}$

se $\mathcal{H}$ è piccolo e ben scelto, non richiede grande sforzo

Implementazione via kernel

- Per implementare $\mathcal{L}$
  - Si implementa un opportuno sottoinsieme $\mathcal{H}$ di $\mathcal{L}$
  - Si usa $\mathcal{H}$ come linguaggio di implementazione
- Usato per i sistemi operativi:
  - si realizza prima il nucleo e poi l'intero sistema usando le primitive offerte dal nucleo
- Semplifica l'implementazione: il linguaggio $\mathcal{H}$ è più vicino di $\mathcal{L}$ a quello della macchina ospite
- Facilita la portabilità: basta re-implementare $\mathcal{H}$ nel nuovo linguaggio macchina

Generazione di compilatori: bootstrapping

USI COMPILATORE PER GENERARE UN COMPILATORE

- I primi ambienti Pascal includevano
  - un compilatore in Pascal da Pascal a P-code: $\mathcal{C}^{\text{Pascal}}{}_{\text{Pascal,P-Code}}$
  - lo stesso compilatore, tradotto in P-code: $\mathcal{C}^{\text{P-Code}}{}_{\text{Pascal,P-Code}}$
  - un interprete per P-code, scritto in Pascal: $\mathcal{I}^{\text{Pascal}}{}_{\text{P-code}}$

- Per aver un'implementazione locale su una specifica Mo:
  - produci (a mano) una traduzione di $\mathcal{I}^{\text{Pascal}}{}_{\text{P-code}}$ nel linguaggio di Mo: $\mathcal{I}^{\mathcal{L}o}{}_{\text{P-code}}$

- Eseguire su Mo un programma P in Pascal su dati x:

$$\mathcal{I}^{\mathcal{L}o}{}_{\text{P-code}}(\mathcal{C}^{\text{P-Code}}{}_{\text{Pascal,P-Code}}, \text{P}) = \text{P'} \quad \text{(scritto in P-code)}$$
$$\mathcal{I}^{\mathcal{L}o}{}_{\text{P-code}}(\text{P'}, \text{x}) = \text{risultato voluto}$$

- Possiamo fare meglio? SI

Generazione di compilatori: bootstrapping, 2

- *Migliorare l'efficienza*, con un compilatore scritto in $\mathcal{L}o$
- A mano, facendo hacking di $C^{\text{Pascal}}{}_{\text{Pascal,P-Code}}$, produci $C^{\text{Pascal}}{}_{\text{Pascal},\mathcal{L}o}$
- Adesso abbiamo:
  - $C^{\text{Pascal}}{}_{\text{Pascal,P-Code}}$
  - $C^{\text{P-Code}}{}_{\text{Pascal,P-Code}}$
  - $\mathcal{I}^{\text{Pascal}}{}_{\text{P-code}}$
  - $\mathcal{I}^{\mathcal{L}o}{}_{\text{P-code}}$
  - $C^{\text{Pascal}}{}_{\text{Pascal},\mathcal{L}o}$
- Bootstrapping

$$\mathcal{I}^{\mathcal{L}o}{}_{\text{P-code}}(C^{\text{P-Code}}{}_{\text{Pascal,P-Code}}, C^{\text{Pascal}}{}_{\text{Pascal},\mathcal{L}o}) = C^{\text{P-Code}}{}_{\text{Pascal},\mathcal{L}o}$$

$$\mathcal{I}^{\mathcal{L}o}{}_{\text{P-code}}(C^{\text{P-Code}}{}_{\text{Pascal},\mathcal{L}o}, C^{\text{Pascal}}{}_{\text{Pascal},\mathcal{L}o}) = C^{\mathcal{L}o}{}_{\text{Pascal},\mathcal{L}o}$$

Esercizi

- Cosa viene calcolato nei seguenti casi?

- $I^{\mathcal{L}0}_{\mathcal{L}1}(C^{\mathcal{L}1}_{\boxed{\mathcal{L}2,\mathcal{L}3}}, C^{\cancel{\mathcal{L}2}\ \mathcal{L}3}_{\mathcal{L}3,\mathcal{L}0}) = ?$ $\quad C^{\mathcal{L}3}_{\mathcal{L}3,\mathcal{L}0}$

- $I^{\mathcal{L}0}_{\mathcal{L}1}(C^{\mathcal{L}1}_{\boxed{\mathcal{L}0,\mathcal{L}2}}, I^{\cancel{\mathcal{L}0}\ \mathcal{L}2}_{\mathcal{L}1}) = ?$ $\quad I^{\mathcal{L}2}_{\mathcal{L}1}$

- $I^{\mathcal{L}0}_{\mathcal{L}1}(C^{\mathcal{L}1}_{\boxed{\mathcal{L}2,\mathcal{L}0}}, I^{\cancel{\mathcal{L}2}\ \mathcal{L}0}_{\mathcal{L}1}) = ?$ $\quad I^{\mathcal{L}0}_{\mathcal{L}1}$ nota: elimino del precedente $I^{\mathcal{L}0}_{\mathcal{L}1}$

- $I^{\mathcal{L}0}_{\mathcal{L}1}(C^{\mathcal{L}2}_{\mathcal{L}0,\mathcal{L}1}, C^{\mathcal{L}0}_{\mathcal{L}2,\mathcal{L}1}) = ?$ $\quad$ ERRORE

Soluzioni

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

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

- $\mathcal{I}^{\mathcal{L}0}_{\mathcal{L}1}(\mathcal{C}^{\mathcal{L}2}_{\mathcal{L}0,\mathcal{L}1}, \mathcal{C}^{\mathcal{L}0}_{\mathcal{L}2,\mathcal{L}1}) = \mathit{errore}$

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

Capitolo 2:

Descrivere un linguaggio di programmazione

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 significato / cosa significa una frase corretta? / "relazione tra segni e significato"
- PRAGMATICA (in quale modo frasi corrette e sensate sono usate / "relazione tra segni, significati e utenti"

Per un linguaggio eseguibile:

- IMPLEMENTAZIONE (come eseguire una frase corretta, rispettandone la semantica)

Sintassi

- **Aspetto Lessicale** (parole che si possono usare)
  - descrizione del lessico
    - Dizionari (x ling. naturali)
    - Strutture più complesse per i ling. artificiali
  - errore? Vocabolo inesistente
    es: "canu", "thon"...
- **Aspetto Grammaticale** (frasi corrette che si possono costruire con il lessico)
  - descrizione attraverso regole grammaticali (in numero finito)
  - Le frasi generabili sono infinite
  - errore? frase scorretta
    es: "La cane abbaiano"
    (lessico corretto, ma frase grammaticamente scorretta)

SEMANTICA

- Per il lessico → Dizionari
- Per le frasi, devo sapere

1) a quale ling. la frase appartiene!!

es: I VITELLI DEI ROMANI SONO BELLI

(VA, O VITELLIO, AL SUONO DEL DIO ROMANO DELLA GUERRA)

2) su quale linguaggio basarmi per dare significato (bisogna basarsi su qualcosa di noto, cioè che non abbia a sua volta bisogno di essere spiegato!)

PRAGMATICA

- insieme di regole che guidano l'uso

Es: • usare il "lei" invece del "tu" quando ci si rivolge ad uno sconosciuto

• evitare di usare i "goto" in programmazione

IMPLEMENTAZIONE

: eseguire una frase sintatticamente corretta rispettandone la semantica

Programma sorgente	Tradotto in $\mathcal{L}_0$
P: I = I+1	Q: MOV I R_0
	ADD 1 R_0
	MOV R_0 I

La semantica di P è la funzione f, che è pure la semantica del programma compilato Q: l'implementazione Q di P preserva la semantica di P!

Lessico e Frasi di un Linguaggio

Alfabeto a, b, c, ... (tipicamente finito)

→ **Lessico**: insieme di sequenze finite (parole) costruite con caratteri (simboli) dell'alfabeto

→ **Frasi**: insieme di sequenze finite costruite con parole del lessico (tipicamente infinito contabile)

Oss: "Lessico" è un "Alfabeto" per "Frasi"!

⇒ Astraiamoci e definiamo 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 in A
- Un linguaggio formale L su A è un insieme di parole su A

Un ling. formale L su alfabeto A è un sottoinsieme di A^* ($L \subseteq A^*$), dove

$$A^* = \bigcup_{n \geq 0} A^n \qquad \text{dove} \quad A^0 = \{\varepsilon\}$$

parola vuota (sequenza senza alcun simbolo)

e $A^{n+1} = A \circ A^n \quad n \geq 0$

con $A \circ A^n = \{ aw \mid a \in A \text{ e } w \in A^n \}$

concatenato

note: $A^1 = A$

Osservazione: A^* è un insieme infinito contabile

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

- prima elenco la parola vuota ε (A^0)
- poi elenco le parole di lunghezza 1 (A^1) secondo l'ordinamento $<$

  $a, b, c, \ldots$

- poi elenco le parole in A^2 secondo $<$

  $aa, ab, ac, \ldots, ba, bb, bc, \ldots$

- poi elenco le parole in A^3 secondo $<$

  $\vdots$

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 parole in A^*?

La risposta è affermativa!

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

- $\mathbb{N} \times \mathbb{N}$ è numerabile (dove-tailing)

$$f^2 : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N} \quad \text{biunivoca}$$

- $\mathbb{N}^k$ è numerabile

$$f^k(x_1, x_2, \dots, x_k) = \text{if } k = 2 \text{ then } f^2(x_1, x_2) \text{ else } f^2(x_1, f^{k-1}(x_2, \dots x_k))$$

- $\mathbb{N}^* = \bigcup_{k \geq 0} \mathbb{N}^k$ è numerabile

$$f(x_1, \dots, x_k) = f^2(k, f^k(x_1, \dots, x_k))$$

Dove - Tailing

$x_2 \backslash x_1$	0	1	2	3	4	5	...
0	0	1	3	6	10	15	
1	2	4	7	11	16		
2	5	8	12	17			
3	9	13	18				
4	14	19					
5	20						
⋮							

si riempie la tabella "per diagonali"

Funzione di codifica: $f^2 : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

$$f^2(x_1, x_2) = \frac{(x_1+x_2)(x_1+x_2+1)}{2} + x_2$$

Es: $f^2(2,1) = \frac{3 \cdot 4}{2} + 1 = 7$

Oss: • $x_1 + x_2$ è il numero di diagonali completamente riempite al di sotto della entry (x_1, x_2)

• la somma di tutti gli elementi in queste $x_1 + x_2$ diagonali è:

$$1 + 2 + 3 + \dots + (x_1+x_2) = \frac{(x_1+x_2)\cdot(x_1+x_2+1)}{2}$$

(metodo di Gauss)

(6 quater

Funzione di decodifica

$f_1^2 : \mathbb{N} \to \mathbb{N}$ $\quad f_2^2 : \mathbb{N} \to \mathbb{N}$

tali che $f_1^2(f^2(x_1, x_2)) = x_1$

$f_2^2(f^2(x_1, x_2)) = x_2$

Bisogna trovare
- $K_x = \max\left\{ i \mid \frac{i \cdot (i+1)}{2} \leq x \right\}$
- allora $K_x = x_1 + x_2$
- e quindi

$$x_2 = x - \frac{K_x(K_x+1)}{2}$$

$$x_1 = K_x - x_2$$

Es: $f_1^2(7) = 2$ perché $K_7 = 3$ $\left(\frac{3 \cdot 4}{2} \leq 7 \right)$

e quindi

$$x_2 = 7 - \frac{3 \cdot 4}{2} = 1$$

$$x_1 = 3 - 1 = 2$$

Notazioni e Definizioni Ausiliarie

- **Lunghezza** di una parola (o stringa)

$|\varepsilon| = 0$ $\quad |aw| = 1 + |w|$ $\quad$ es: $|abc| = 3$

- **Concatenazione**: x concatenata a y, xy, è la parola ottenuta giustapponendo x e y.

$$\boxed{w = xy \iff \begin{cases} \cdot\ |w| = |x| + |y| \\ \cdot\ w(j) = x(j) & \text{per } 1 \le j \le |x| \\ \cdot\ w(|x|+j) = y(j) & \text{per } 1 \le j \le |y| \end{cases}}$$

($w(j)$ indica il j-esimo simbolo di w)

Leggi della concatenazione

$x(yz) = (xy)z$ $\quad$ associatività xyz

$x\varepsilon = x = \varepsilon x$ $\quad$ ε è elemento neutro

- **Sottostringa v di una stringa w**

$\iff \exists\, x, y \in A^*$ tali che $w = xvy$

(dove x e y possono essere ε)

Oss: 1) Ogni stringa è sottostringa di se stessa

2) ε è sottostringa di ogni stringa

- **Suffisso**

  v è suffisso di w sse $\exists x \in A^*.\ w = xv$

- **Prefisso**

  v è prefisso di w sse $\exists x \in A^*.\ w = vx$

- **Potenza n-esima** di una stringa w

  $$w^0 = \varepsilon \qquad w^{n+1} = w w^n \qquad \forall n \geq 0$$

  Es: $(ab)^1 = ab \quad (ab)^2 = abab$

  $(ab)^3 = ababab \quad (ab)^4 = ababab$

- **Linguaggio L su alfabeto A** : $L \subseteq A^*$

  Se $A = \{a\}$, allora $\emptyset$, $\{\varepsilon\}$, $\{a, aaa\}$ sono ling. finiti.

  $$L_1 = \{a, aa, aaa, \dots\} = \{a^n \mid n \geq 1\}$$
  $$= A^* \setminus \{\varepsilon\}$$
  $$\uparrow$$
  $$= \{a^n \mid n \geq 0\}$$

  $$L_2 = \{\varepsilon, aa, aaaa, \dots\} = \{a^{2n} \mid n \geq 0\}$$

  $$L_3 = \{a, aaa, aaaaa, \dots\} = \{a^{2n+1} \mid n \geq 0\}$$

Operazioni su Linguaggi

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

- **Unione** e **Intersezione** (ovvi)

$$L_1 \cup L_2 = \{w \mid w \in L_1 \text{ o } w \in L_2\}$$

$$L_1 \cap L_2 = \{w \mid w \in L_1 \text{ e } w \in L_2\}$$

- **Concatenazione**

$$L_1 \circ L_2 = \{w_1 w_2 \mid w_1 \in L_1 \text{ e } w_2 \in L_2\}$$

Oss: Concatenazione $\neq$ Prodotto Cartesiano

$$L_1 \times L_2 = \{(w_1, w_2) \mid w_1 \in L_1 \text{ e } w_2 \in L_2\}$$

I due insiemi non sono sempre equipotenti perché valgono le leggi della concatenazione

$$x(yz) = (xy)z \qquad x\varepsilon = x = \varepsilon x$$

Es: $L_1 = \{a, ab\}$ $\qquad L_2 = \{bc, c\}$

$$L_1 \circ L_2 = \{abc, ac, abbc\}$$

$$L_1 \times L_2 = \{(a, bc), (a, c), (ab, bc), (ab, c)\}$$

N.B: $a(bc) = (ab)c = abc$

Esempi

- $L_1 = \{ a^n \mid n \geq 0 \}$ $\quad L_2 = \{ b \}$

  $L_1 \circ L_2 = \{ a^n b \mid n \geq 0 \}$

- $L_1 = \{ a^{2n} \mid n \geq 0 \}$ $\quad L_2 = \{ b^m \mid m \geq 0 \}$

  $L_1 \circ L_2 = \{ a^{2n} b^m \mid n, m \geq 0 \}$

- $L_1 = \{ a^n b^n \mid n \geq 0 \}$ $\quad L_2 = \{ b^n \mid n \geq 0 \}$

  $L_1 \circ L_2 = \{ a^n b^{n+m} \mid n, m \geq 0 \}$

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

- $L_1 = \{ a^n \mid n \geq 1 \}$ $\quad L_2 = \{ a^n b^n \mid n \geq 0 \}$

  $L_1 \circ L_2 = \{ a^{n+m} b^m \mid n \geq 1, m \geq 0 \}$

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

- $L_1 = \{ a b^n \mid n \geq 1 \}$ $\quad L_2 = \{ a, c \} \cup \{ b^n \mid n \geq 1 \}$

  $L_1 \circ L_2 = \{ a b^n a \mid n \geq 1 \} \cup \{ a b^n c \mid n \geq 1 \}$

  $\cup \{ a b^n \mid n \geq 2 \}$

- $A = \{0, 1\}$ $\quad L_1 = \{ w \in A^* \mid w$ contiene un numero pari di "0" $\}$

  $L_2 = \{ w \in A^* \mid w = 0y$ e $y \in \{1\}^* \}$

  $L_1 \circ L_2 = \{ w \in A^* \mid w$ ha un numero dispari di "0" $\}$

POTENZA di un linguaggio

$$L^0 = \{\varepsilon\}$$
$$L^{n+1} = L \cdot L^n \qquad \forall n \geq 0$$

CHIUSURA / STELLA DI KLEENE / RIPETIZIONE

$$L^* = \bigcup_{n \geq 0} L^n$$

$$L^+ = \bigcup_{n \geq 1} L^n \qquad \text{(chiusura positiva)}$$

Esempi

$L = \{aa\}$ $\qquad L^* = \{\varepsilon, aa, aaaa, \dots\}$
$\qquad = \{(aa)^n \mid n \geq 0\}$

$L = \{a, aaa\}$ $\qquad L^* = \{a^n \mid n \geq 0\}$

$L = \{aa, aaa\}$ $\qquad L^* = \{a^n \mid n \neq 1\}$

Rappresentazione Finita dei Linguaggi

Se L è infinito, come possiamo rappresentarlo?
(e come possiamo determinare se una certa stringa $w \in L$?)

Esempio 1) $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ infinito contabile

- Non possiamo "memorizzare" $\mathbb{N}$: servirebbe una memoria infinita
- Possiamo trovare una rappresentazione finita implicita:

$$\frac{}{0 \in \mathbb{N}} \qquad \frac{x \in \mathbb{N}}{S(x) \in \mathbb{N}} \qquad \text{(Peano)}$$

successivo

GENERO $\left\{ \begin{array}{l} \mathbb{N} \text{ è il più piccolo insieme generato} \\ \text{dall'assioma e dalla regola!} \end{array} \right.$

$$\mathbb{N} = \{0, S(0), S(S(0)), S(S(S(0))), \dots\}$$

Esempio 2) $\mathbb{P} = \{0, 2, 4, 6, 8, \dots\}$

- Non possiamo "memorizzare" $\mathbb{P}$
- Possiamo trovare una rappresentazione finita implicita:

```
fun pari (x: integer): boolean
    pari := (x mod 2 = 0)
```

Riconosco se $x \in \mathbb{P}$ $\left\{ \right.$ Il programma è una rappresentazione finita implicita della funzione (infinita) che calcola

PER I LING. ESISTONO 2 TECNICHE

1) GENERATIVO / SINTETICO:

Linguaggio = insieme delle stringhe generate da una struttura finita detta GRAMMATICA (a struttura di frase)

2) RICONOSCITIVO / ANALITICO:

Linguaggio = insieme delle stringhe riconosciute da una struttura finita detta AUTOMA

OSS: NON TUTTI I LINGUAGGI POSSONO ESSERE GENERATI DA GRAMMATICHE O RICONOSCIUTI DA AUTOMI !!

- A^* è equipotente a $\mathbb{N}$
- $L \subseteq A^*$ è contabile (finito o infinito)
- $\mathcal{P}(A^*)$ è l'insieme di tutti i ling. su alfabeto A, ma $\mathcal{P}(A^*)$ è equipotente a $\mathbb{R}$, l'insieme dei numeri reali!
- Un "formalismo" è un linguaggio su un alfabeto
  $\Rightarrow$ i suoi "programmi" (grammatiche/automi) sono numerabili!

Se di grammatiche su alfabeto A. ce ne sono tante quante $\mathbb{N}$, e se i linguaggi su A ce ne sono tanti quanti $\mathbb{R}$, allora tantissimi linguaggi non possono essere rappresentati finitamente!

	A^*				
	w_1	w_2	w_3	w_4	...
$L(G_1)$	X		X		
$L(G_2)$			X	X	
$L(G_3)$			X		
$L(G_4)$	X			X	
⋮					

ARGOMENTO DIAGONALE DI CANTOR

$$L(G_1) = \{ w_1, w_3, \dots \}$$
$$L(G_2) = \{ w_3, w_4, \dots \}$$
$$L(G_3) = \{ w_3, \dots \}$$
$$\vdots$$

$$\bar{D} = \{ w_i \mid w_i \notin L(G_i) \} \qquad \bar{D} = \{ w_2, \dots \}$$

$\bar{D}$ è un linguaggio, ma non esiste alcuna G_k tale che $\bar{D} = L(G_k)$ perché:

- se $w_k \in L(G_k)$, allora $w_k \notin \bar{D}$
- se $w_k \notin L(G_k)$, allora $w_k \in \bar{D}$

$\Rightarrow$ $\bar{D}$ non compare nell'elenco dei ling. rappresentabili con una grammatica

Definire finitamente un linguaggio

Esempio (1): palindrome = "parola che letta da sx a dx è uguale a sé stessa letta da dx a sx"

es: abba amoroma

(Madam I'm Adam)

in senso esteso

(I Topi non avevano mipoti)

$A = \{a, b\}$

$L = \{\varepsilon, a, b, aa, bb, aba, bab, aaa, bbb, abba, baab, aaaa, bbbb, \dots\}$

Una palindrome può essere

- o la stringa ε
- oppure a
- oppure b
- oppure a "palindrome" a
- oppure b "palindrome" b

Backus-Naur Form (BNF)

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

può essere

oppure

Come grammatica:

$$P \to \varepsilon \mid a \mid b \mid aPa \mid bPb$$

Definizione ricorsiva in cui

- P è detto "simbolo nonterminale"
- a, b sono i "simboli terminali"

Alternativamente: (insieme di assiomi e regole d'inferenza)

$$\frac{}{\varepsilon \in L(P)} \qquad \frac{}{a \in L(P)} \qquad \frac{}{b \in L(P)}$$

$$\frac{w \in L(P)}{awa \in L(P)} \qquad \frac{w \in L(P)}{bwb \in L(P)}$$

$L(P)$ = "linguaggio generato a partire dal nonterminale P"

è il più piccolo insieme generato da questi assiomi e regole d'inferenza

Esempio (2): **Espressioni aritmetiche** formate a partire dalle variabili "a" e "b" con gli operatori *, + e le parentesi (,)

Una <u>expr</u> può essere

- la variabile a,
- oppure la variabile b,
- oppure <u>expr</u> * <u>expr</u>,
- oppure <u>expr</u> + <u>expr</u>,
- oppure (<u>expr</u>)

<u>BNF</u>:

$$\langle E \rangle ::= a \mid b \mid \langle E \rangle * \langle E \rangle \mid \langle E \rangle + \langle E \rangle \mid (\langle E \rangle)$$

<u>Grammatica</u>

$$E \rightarrow a$$
$$E \rightarrow b$$
$$E \rightarrow E * E$$
$$E \rightarrow E + E$$
$$E \rightarrow (E)$$

più brevemente

$$E \rightarrow a \mid b \mid E * E \mid E + E \mid (E)$$

Come derivare una stringa?

$$P \to \varepsilon \mid a \mid b \mid aPa \mid bPb$$

Si parte da P e si applicano le produzioni

Ad es: abba è una palindrome perché

$$\underbrace{P \Rightarrow aPa}_{P \to aPa} \underbrace{\Rightarrow abPba}_{P \to bPb} \underbrace{\Rightarrow abba}_{P \to \varepsilon} \quad \text{cioè } P \Rightarrow^* abba$$

$$\underbrace{P \Rightarrow bPb}_{P \to bPb} \underbrace{\Rightarrow bbPbb}_{P \to bPb} \underbrace{\Rightarrow bbabb}_{P \to a} \quad \text{cioè } P \Rightarrow^* bbabb$$

$$\frac{\varepsilon \in L(P)}{bb = b\varepsilon b \in L(P)}$$

$$\frac{\dfrac{a \in L(P)}{bab \in L(P)}}{bbabb \in L(P)}$$

$$\frac{}{abba = ab\varepsilon ba \in L(P)}$$

Altro esempio

$$E \to a \mid b \mid E * E \mid E + E \mid (E)$$

Come dimostrare che $a + (a * b) \in L(E)$?

$$E \Rightarrow \underline{E} + E \Rightarrow a + \underline{E} \Rightarrow a + (\underline{E}) \Rightarrow a + (\underline{E} * E)$$

$$\Rightarrow a + (a * \underline{E}) \Rightarrow a + (a * b)$$

$$\dfrac{a \in L(E) \qquad \dfrac{\dfrac{a \in L(E) \qquad b \in L(E)}{a * b \in L(E)}}{(a * b) \in L(E)}}{a + (a * b) \in L(E)}$$

GRAMMATICHE

- Tante classi di grammatiche
  - regolari, libere (da contesto), dipendenti dal contesto, monotone, generali (o "a struttura di frase")
  - tutte seguono lo stesso pattern, differenziandosi solo per come sono caratterizzate le produzioni (o regole)

  (Le vedremo in seguito: classificazione di Chomsky)
- Quelle più utili (rapporto tra espressività e facilità d'analisi) sono le "libere"

Def Una grammatica libera da contesto è una quadrupla (NT, T, R, S) dove

- NT è un insieme finito di simboli non-terminali (di solito lettere maiuscole $A, B, \ldots$)
- T è un insieme finito di simboli terminali (di solito lettere minuscole $a, b, c, \ldots$)
- $S \in NT$ è detto simbolo iniziale
- R è un insieme finito di produzioni (o regole) della forma

  $V \rightarrow w$ dove $V \in NT$ e $w \in (T \cup NT)^*$

Esempi

(1) $G = (\{S\}, \{a, b, +, *\}, S, R)$

con $R = \{ S \xrightarrow{1} a,$
$S \xrightarrow{2} b,$
$S \xrightarrow{3} S+S,$
$S \xrightarrow{4} S*S \}$

$$\boxed{S \to a \mid b \mid S+S \mid S*S}$$

N.B. R è un insieme: numeriamo le produzioni per dare loro un "nome", non per ordinarle.

(2) $G_1 = (\{S, A\}, \{a, b\}, S, R_1)$

con $R_1 = \left\{ \begin{array}{l} S \xrightarrow{1} aAb \\ A \xrightarrow{2} aAb \\ A \xrightarrow{3} \varepsilon \end{array} \right\}$

$$\boxed{\begin{array}{l} S \to aAb \\ A \to aAb \mid \varepsilon \end{array}}$$

$G_2 = (\{S, A, B\}, \{a, b\}, S, R_2)$

con $R_2 = \left\{ \begin{array}{l} S \xrightarrow{1} AB \\ A \xrightarrow{2} aA \\ A \xrightarrow{3} a \\ B \xrightarrow{4} bB \\ B \xrightarrow{5} b \end{array} \right\}$

$$\boxed{\begin{array}{l} S \to AB \\ A \to aA \mid a \\ B \to bB \mid b \end{array}}$$

$G_3 = (\{S\}, \{a, b\}, S, R_3)$

con $R_3 = \left\{ \begin{array}{l} S \xrightarrow{1} aSb \\ S \xrightarrow{2} ab \end{array} \right\}$

$$\boxed{S \to aSb \mid ab}$$

N.B. Di solito, usiamo solo la forma per rappresentare una grammatica.

Esempio (3)

$$E \rightarrow I \mid E+E \mid E*E \mid -E \mid (E)$$
$$I \rightarrow a \mid b \mid Ia \mid Ib$$

$L(I) = \{a, b\}^+$ qualunque sequenza non vuota di "a" e "b"

↳ identificatori (insieme infinito)

$L(E)$ = espressioni aritmetiche costruite sugli identificatori

(cioè I genera "il lessico" (identificatori) che E può usare per costruire le expr. aritmetiche)

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 in w"), e lo denotiamo con $v \Rightarrow w$, se

$$\boxed{\frac{v = xAy \quad (A \to z) \in R \quad w = xzy}{v \Rightarrow w} \qquad x, y, z \in (T \cup NT)^*}$$

Diciamo che da v si deriva w (o anche "v si riscrive in w"), e lo denotiamo con $v \Rightarrow^* w$, se esiste una sequenza finita (eventualmente vuota) di derivazioni immediate

$$v \Rightarrow w_0 \Rightarrow w_1 \Rightarrow \dots \Rightarrow w$$

Cioè

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

$\Rightarrow^*$ è la chiusura riflessiva e transitiva della relazione $\Rightarrow$

Esempi

(1) G $\quad S \to a \mid b \mid S+S \mid S*S$

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

cioè $S+S \Rightarrow^* S*S+a*S$

(2) G_1 $\quad S \to aAb$
$\quad\quad A \to aAb \mid \varepsilon$

$$\underline{A}A \Rightarrow a\underline{A}bA \Rightarrow ab\underline{A} \Rightarrow aba\underline{A}b \Rightarrow abab$$

cioè $AA \Rightarrow^* abab$

(3) G_2 $\quad S \to AB$
$\quad\quad A \to aA \mid a$
$\quad\quad B \to bB \mid b$

$$S \Rightarrow A\underline{B} \Rightarrow Ab\underline{B} \Rightarrow \underline{A}bb \Rightarrow abb$$

cioè $S \Rightarrow^* abb$

ma anche

$$S \Rightarrow \underline{A}B \Rightarrow a\underline{B} \Rightarrow ab\underline{B} \Rightarrow abb$$

cioè ci possono essere più derivazioni diverse che generano la stessa stringa

LINGUAGGIO GENERATO

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

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

solo terminali

si parte dal simbolo iniziale S

Data G, come faccio a determinare L(G)?
E a verificare se $w \in L(G)$?

- studieremo tecniche opportune che in alcuni casi saranno anche efficienti
- algoritmo "naif": partire da S e provare ad applicare in tutti i modi possibili le produzioni per trovare una derivazione che generi w
  (nondeterminismo ⇒ backtracking)

In alcuni casi, tale verifica è semplice

Esempi

G_3 $\quad S \to aSb \mid ab$

$$S \Rightarrow ab$$
$$S \Rightarrow aSb \Rightarrow aabb$$
$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$$
$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \ldots$$

È chiaro che

$$L(G_3) = \{a^n b^n \mid n \geq 1\}$$

G_1 $\quad S \to aAb$

$A \to aAb \mid \varepsilon$

$$S \Rightarrow aAb \Rightarrow ab$$
$$S \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aabb$$
$$S \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aaaAbbb \Rightarrow aaabbb$$
$$S \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aaaAbbb \Rightarrow a^4Ab^4 \ldots$$

È chiaro che

$$L(G_1) = \{a^n b^n \mid n \geq 1\}$$

N.B. • G_1 e G_3 sono due grammatiche equivalenti perché $L(G_1) = L(G_3)$

• In generale esistono infinite grammatiche diverse che generano lo stesso ling.

Esempi

G_2

$$S \to AB$$
$$A \to aA \mid a$$
$$B \to bB \mid b$$

$$L(G_2) = L(S) = L(A) \cdot L(B)$$

$$A \overset{a}{\Rightarrow} aA \overset{aa}{\Rightarrow} aaA \overset{aaa}{\Rightarrow} aaaA \ldots$$

$$L(A) = \{a^n \mid n \geq 1\}$$

idem per $L(B) = \{b^n \mid n \geq 1\}$

$$\Rightarrow L(A) \cdot L(B) = \{a^n b^m \mid n, m \geq 1\}$$

$$\left.\begin{array}{l} S \to AB \\ A \to aA \mid \varepsilon \\ B \to Bb \mid \varepsilon \end{array}\right] G$$

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

N.B. $\varepsilon \in L(G)$ perché $S \Rightarrow \underline{A}B \Rightarrow \underline{B} \Rightarrow \varepsilon$

Esempi

$S \to AB$

$A \to aAb \mid \varepsilon$

$B \to bB \mid \varepsilon$

$L(S) = L(A) \cdot L(B)$

dove $L(A) = \{a^n b^n \mid n \geq 0\}$

$L(B) = \{b^n \mid n \geq 0\}$

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

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

All'incontrario, dato un linguaggio, come determinare una grammatica che lo genera?

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

$S \to AB$

$A \to \varepsilon \mid aA$

$B \to aBb \mid \varepsilon$

$L = \{a^{2n+1} \mid n \geq 0\} = \{a, aaa, aaaaa, \ldots\}$

$S \to a \mid aaS$

$L = \{ a^{2n} \; b^{2m+1} \mid n, m \geq 0 \}$

$S \to aaS \mid bB$

$B \to \varepsilon \mid bbB$

$L = \{ab, ac, ad\}$

$S \to ab \mid ac \mid ad$

oppure

$S \to aA$

$A \to b \mid c \mid d$

$L = \{ a^{2n} b^{m} c^{2m} \mid n \geq 0, m \geq 1 \}$

$S \to aaS \mid bBcc$

$B \to \varepsilon \mid bBcc$

$L = \{ a^{n} \; b^{k} \; c^{n} \mid n \geq 0, k \geq 1 \}$

$S \to aSc \mid B$

$B \to b \mid bB$

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

$S \to AB$

$A \to aAb \mid \varepsilon$

$B \to cBd \mid \varepsilon$

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

$S \to aSd \mid bAc \mid \varepsilon$

$A \to bAc \mid \varepsilon$

$L = \{ w \in \{a,b\}^* \mid \exists x \in \{a,b\}^*.\ w = xaa \}$

STRINGHE CHE TERMINANO CON aa

$S \to aS \mid bS \mid aa$

$L = \{ w \in \{(,)\}^* \mid$ in w tutte le parentesi sono bilanciate $\}$

$S \to \varepsilon \mid (S) \mid SS$

$L = \{ w \in \{a,b\}^* \mid$ in w occorrono tante "a" quante "b" $\}$

$S \to \varepsilon \mid aSb \mid bSa \mid SS$

$L = \{ ww^R \mid w \in \{a,b\}^* \}$

palindromi di lunghezza pari

w^R è definita:

$\varepsilon^R = \varepsilon$

$(aw)^R = w^R a$

$S \to \varepsilon \mid aSa \mid bSb$

$$L = \{ a^{2n} b^n \mid n \geq 0 \}$$

$$S \to aaSb \mid \varepsilon$$

$L = \{ w \in \{a,b\}^* \mid$ in w le "a" siano dispari e le "b" pari $\}$

$$S \to aB \mid bA$$
$$B \to aS \mid bC \mid \varepsilon$$
$$A \to aC \mid bS$$
$$C \to aA \mid bB$$

S = a pari / b pari

B = a dispari / b pari

C = a dispari / b dispari

A = a pari / b dispari

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

$$S \to aSc \mid aBb \mid \varepsilon$$
$$B \to aBb \mid \varepsilon$$

Ultimo esempio (Hard!)

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

$S \to \varepsilon \mid aSb \mid aAa \mid bBb \mid bCa$

$A \to \varepsilon \mid aAa \mid bCa$

$B \to \varepsilon \mid bBb \mid bCa$

$C \to \varepsilon \mid bCa$

Esercizio

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

$G = ?$

$$\boxed{\begin{aligned} S &\to \varepsilon \mid AB \\ A &\to \varepsilon \mid aAb \\ B &\to \varepsilon \mid bBa \end{aligned}}$$

Derivazioni e Alberi

Consideriamo la grammatica

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

- Consideriamo la derivazione

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

  - leftmost: ad ogni passo riscriviamo il nonterminale più a sx
  - è possibile associarle un albero di derivazione

- Consideriamo un'altra derivazione per a+b*c

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

  - rightmost: ad ogni passo riscriviamo il nonterminale più a dx
  - genera lo stesso albero di derivazione

Osservazioni:

(1) Un albero di derivazione "riassume" tante derivazioni diverse, ma tutte "equivalenti" (nel senso che tutte generano lo stesso albero)

(2) Esiste una corrispondenza biunivoca tra derivazioni canoniche sinistre (o destre) e alberi di derivazione!

- dato un albero di derivazione, esiste una sola derivazione leftmost che lo genera (non ovvio!)
- dato un albero di derivazione, esiste una sola derivazione rightmost che lo genera
- data una derivazione leftmost (o rightmost), ad essa viene associato univocamente un albero di derivazione (ovvio)

(3) L'albero di derivazione fornisce informazioni semantiche: "quali operandi per quali operatori"

← albero sintattico ricavato dall'albero di derivazione

Alberi Sintattici da Alberi di Deriv.

- Albero di soli terminali dove i nodi interni sono etichettati da operatori e le foglie sono gli operandi
- Tipica rappresentazione interna usata dal compilatore per generare codice intermedio

Albero di Derivazione

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

- ogni nodo è etichettato con un simbolo in $NT \cup \{\varepsilon\} \cup T$
- la radice è etichettata con S
- ogni nodo interno è etichettato con un simbolo in NT
- se il nodo n
  - ha etichetta $A \in NT$ e
  - i suoi figli sono nell'ordine $m_1, \dots m_k$ con etichetta $X_1, \dots X_k$ (in $NT \cup T$), allora

    $A \to X_1 \dots X_k$ è una produzione in R
- se il nodo n ha etichetta ε, allora n è una foglia, è figlio unico e, detto A suo padre, $A \to \varepsilon$ è una produzione di R
- se inoltre ogni nodo foglia è etichettato su $T \cup \{\varepsilon\}$, allora l'albero corrisponde ad una derivazione completa.

Teorema: Una stringa $w \in T^*$ appartiene a $L(G)$ sse ammette un albero di derivazione completo (le cui foglie, lette da sx a dx diano la stringa w)

cioè visita in ordine anticipato tralasciando i nonterminali

Ambiguità

- Consideriamo la grammatica

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

- Due diverse derivazioni leftmost per $a*b+c$

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

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

Per questa grammatica, la stringa $a*b+c$ ha più di un albero di derivazione $\Rightarrow$ la grammatica è ambigua

(Inutilizzabile per dare semantica a $a*b+c$)

- Poiché tutti e due questi alberi sintattici sono derivabili per a*b+c, la semantica non è univoca! Ambiguità!
- Bisogna utilizzare grammatiche non ambigue, o manipolare grammatiche ambigue per disambiguarle
  (problema non facile e non sempre risolvibile)

Grammatica Ambigua

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

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

---

Alcune grammatiche possono essere manipolate in modo da

- rimuovere l'ambiguità, e
- generare lo stesso linguaggio

Esempio

$$\left.\begin{array}{l} S \to A \mid \varepsilon \\ A \to \varepsilon \end{array}\right]$$

G è ambigua perché

$$\begin{array}{c} S \\ | \\ \varepsilon \end{array} \qquad \begin{array}{c} S \\ | \\ A \\ | \\ \varepsilon \end{array}$$

Tuttavia $\exists\, G'$ non ambigua tale che $L(G) = L(G')$

$$G' \left[S \to \varepsilon \right.$$

Esistono grammatiche (patologiche) dalle quali non è possibile rimuovere l'ambiguità.

$\Rightarrow$ il linguaggio generato da queste grammatiche è ambiguo!

Esempio di un ling. ambiguo

Sappiamo che $L_1 = \{a^n b^n \mid n \geq 1\}$ è libero.
Non è difficile immaginare che

$L_2 = \{a^n b^n c^m d^m \mid n, m \geq 1\}$ sia libero non ambiguo

ed anche

$L_3 = \{a^n b^m c^m d^n \mid n, m \geq 1\}$ sia libero non ambiguo

È facile verificare che

$L = L_2 \cup L_3$ è libero (perché l'unione di 2 ling. liberi è un ling. libero)

Tuttavia si può dimostrare che L è ambiguo!
Infatti, tutte le stringhe del tipo $a^n b^n c^n d^n$ hanno doppia derivazione

- una attraverso le produzione che generano il ling. L_2
- e un'altra attraverso le produzioni che generano il ling. L_3

(dimostrazione tecnicamente difficile)

N.B. G_1 $\quad S_1 \to a S_1 b \mid ab$

G_2 $\quad S_2 \to S_1 S_1' \qquad S_1' \to c S_1' d \mid cd$

G_3 $\quad S_3 \to a S_3 d \mid a S_3' d \qquad S_3' \to b S_3' c \mid bc$

G $\quad S_4 \to S_2 \mid S_3$

Rimuovere l'ambiguità (1)

$S \to a \mid b \mid c \mid S+S \mid S*S$ è ambigua!

Problemi: (1) precedenza del * rispetto al + in modo che $a*b+c$ sia interpretato come

(2) associatività del + e del *

$a+b+c$

Bisogna scegliere se associare a dx o a sx

associatività a sx

$E \to E+T \mid T$

$T \to A*T \mid A$

$A \to a \mid b \mid c$

* ha la precedenza sul +, perché è più interno nell'albero generato

associatività a dx

$L(S) = L(E)$

$\Rightarrow$ la nuova gr. è non ambigua ed equivalente

ma l'albero

non è generabile!

Per ottenere (quasi) lo stesso linguaggio, ma sicuramente gli stessi alberi sintattici, modifichiamo la grammatica così:

$$E \to E + T \mid T$$
$$T \to A * T \mid A$$
$$A \to a \mid b \mid c \mid (E)$$

nuovi terminali ausiliari, usati per disambiguare

$a * (b + c)$

albero sintattico astratto da sintassi "concreta" che prevede anche terminali ausiliari "(" e ")" semanticamente irrilevanti.

zucchero sintattico!

Sintassi Concreta:

grammatica non ambigua che fa uso di zucchero sintattico (come per E sopra)

Sintassi Astratta:

grammatica semplice ed intuitiva, ma ambigua.

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

Rimuovere l'ambiguità (2)

Consideriamo questa porzione di gram. per comandi

<stm> ::= if <expr> then <stm> |
if <expr> then <stm> else <stm> | ...

È ambigua. Infatti if E_1 then if E_2 then S_1 else S_2 ammette due alberi di derivazione

(interpretazione usuale)

(albero "errato")

Grammatica Concreta

<stm> ::= <matched-stm> | <unmatched-stm>

<matched-stm> ::= if <expr> then <matched-stm>
else <matched-stm> | ... | begin <stm> end

<unmatched-stm> ::= if <expr> then <stm> |
if <expr> then <matched-stm>
else <unmatched-stm> | ...

Esercizio : Espressioni booleane

$S \to a \mid b \mid \sim S \mid S \text{ or } S$ (sintassi astratta)

1) È ambigua? Sì

2) Grammatica ~~non~~ ambigua equivalente?

$B \to B \text{ or } C \mid C$

$C \to \sim C \mid A$

$A \to a \mid b$

- ~ lega più fortemente dell'or
- or associativo a SX

3) La nuova grammatica genera l'albero

? NO

4) Modificare la grammatica per generare quell'albero

Sintassi Concreta

$B \to B \text{ or } C \mid C$

$C \to \sim C \mid A$

$A \to a \mid b \mid (B)$

Vincoli Sintattici Contestuali

(Semantica Statica)

Esempi:

- una variabile in uso deve prima essere dichiarata
- Compatibilità di tipo in un assegnamento

  x := e "x" e l'espressione "e" devono avere lo stesso tipo
- il numero (e il tipo) dei parametri attuali di una chiamata di procedura deve essere uguale al numero (e al tipo) dei parametri formali della dichiarazione

Sono vincoli sintattici, ma non esprimibili per mezzo di grammatiche libere (o BNF), perché queste non sono in grado di descrivere vincoli che dipendono dal contesto (ad es: dalle dichiarazioni)

Soluzioni? Ci sono due alternative:

(1) Usare strumenti più potenti, come le grammatiche dipendenti dal contesto

Poco pratico: complessità del problema "$w \in L(G)$?" (con G contestuale) esponenziale nella lunghezza di w e nella taglia di G

(2) Usare controlli "ad hoc"

Nella costruzione di un compilatore, la fase di "Analisi Semantica" fa questi controlli

Sintassi o Semantica?

I vincoli sintattici contestuali "appartengono" alla sintassi.

Ma tradizionalmente, nel gergo dei LP, si intende:

- Sintassi: quello che si descrive in BNF (grammatiche libere)
- Semantica: tutto il resto....

⇓

I vincoli contestuali sono dunque vincoli "semantici", detti di semantica statica, cioè vincoli che possono essere verificati ispezionando il codice del programma, senza mandarlo in esecuzione

Il compilatore delega questi controlli di semantica statica (o sintassi contestuale) alla 3ª fase, denominata

Analisi Semantica

Semantica Statica vs. Semantica Dinamica

Per semantica statica (o sintassi contestuale) si intende l'insieme di quei controlli che possono essere fatti sul testo del programma, senza eseguirlo. Ad esempio, Type checking

```
int A;
bool B;
A := B      ← errore di tipo
```

Per semantica dinamica si intende una rappresentazione formale dell'esecuzione del programma, la quale può mostrare errori "dinamici", cioè durante l'esecuzione

```
P=  int A, B;
    read (A);
    B := 10/A;
    :
```

Se l'utente fornisce in input il valore 0 per A, allora la divisione 10/A dà errore in esecuzione.

In questo caso, non possiamo staticamente sapere se l'errore si verificherà perché l'occorrere dell'errore dipende dall'input che l'utente fornirà quando il programma verrà eseguito.

Semantica Dinamica

- Fornire un modello matematico che descriva, indipendentemente dall'architettura su cui il programma viene eseguito, il "comportamento" del programma.
- L'implementazione del linguaggio (in cui è scritto il programma) deve rispettare questa "specifica" del linguaggio

Esempio

$P = \quad x := x + 1$

store
σ = insieme di associazioni tra nomi e valori

modello "grafo"

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

(Astratto e indipendente dall'architettura)

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 (ad es: "termina sempre per ogni possibile input?")

- **Al progettista del linguaggio**:
  - strumento di specifica del linguaggio
  - deve poter dimostrare proprietà del linguaggio (ad es: "è Turing-completo?")

- **All'implementatore del linguaggio**
  - riferimento per dimostrare la correttezza dell'implementazione

Ricorda: un compilatore è corretto quando preserva la semantica (dinamica!), quindi per dimostrare che un compilatore è corretto serve avere una semantica per il ling. sorgente e per il ling. oggetto.

Come si fa a definire la semantica?

Due tecniche principali:

- **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

- **Denotazionale**: si associa ad ogni programma sequenziale una funzione da input ad output (incluse strutture ausiliarie dette ambiente e memoria)

  Enfasi su cosa si calcola

  (vengono nascosti i passi intermedi del calcolo)

Vedremo l'uso della semantica operazionale per un semplice linguaggio imperativo

Altri aspetti della descrizione di un linguaggio

PRAGMATICA: insieme di "regole"/consigli sul modo in cui è meglio usare le istruzioni a disposizione
("buone maniere" per ling. naturali, "stile di programmazione" per ling. artificiali)

Esempi:

- evitare le istruzioni di salto (GOTO) quando possibile (altrimenti i programmi diventano presto incomprensibili)
- usare le variabili di controllo del for solo a quello scopo (per evitare side-effect pericolosi)
- scelta della modalità più appropriata di passaggio di parametri ad una funzione (ad es: mai usare il passaggio per valore per strutture dati grandissime – spreco di memoria)
- scelta tra iterazione determinata (FOR – da preferirsi, quando possibile) e indeterminata (while/repeat)

Infine ... IMPLEMENTAZIONE

ovvero scrivere 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, 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?

Struttura di un Compilatore

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

Fasi Principali della Compilazione

Analisi Lessicale (Scanner): spezza il programma sorgente nei componenti sintattici primitivi, chiamati tokens (identificatori, numeri, operatori, parentesi, parole riservate, ...)

- controlla solo che il "lessico" sia ammissibile (operatori inesistenti, identificatori non ammessi, simboli non previsti, ...)
- riempie parzialmente la tabella dei simboli (per gli identificatori di variabili, procedure, funzioni...)

Per realizzare uno Scanner, avremo bisogno di studiare:

- grammatiche regolari (un sotto-tipo di grammatiche libere le cui produzioni sono del tipo $A \to aB$ oppure $A \to a$)
- 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"

Analisi Sintattica (Parser)

A partire dalla lista di tokens, generata dallo scanner, il parser produce l'albero di derivazione del programma, riconoscendo se le frasi sono sintatticamente corrette.

Ad esempio, controlla che

- le parentesi usate in espressioni aritmetiche siano bilanciate

  (a (+ b))) 2 errori! Quali?

- che i comandi siano composti secondo le regole grammaticali

  if x = 5 then then x := 3 errore!

Per realizzare un Parser, avremo bisogno di:

- grammatiche libere da contesto
- automi a pila (pushdown automata - PDA) soprattutto nella loro versione deterministica (DPDA)

Analisi Semantica

Esegue dei controlli di semantica statica (ovvero sintattici contestuali) per rilevare eventuali errori "semantici"

- arricchisce l'albero di derivazione generato dal Parser con informazioni sui tipi
- verifica i tipi negli assegnamenti, parametri attuali vs. formali, dichiarazione e uso di variabili, ...
- genera eventualmente opportuni msg d'errore.

Generazione della Forma Intermedia

Genera codice scritto in un ling. intermedio, indipendente dall'architettura, facilmente traducibile nel linguaggio macchina di varie macchine diverse.

- easy to produce — easy to translate

- utilizza operazioni molto semplici, tipicamente "three-address code" del tipo $x := y + z$
- nel generare codice intermedio, si segue la struttura dell'albero sintattico, ricavato dall'albero di derivazione

$X := a + b$

temp1 := a + b
x := temp1

Ottimizzazione

Si effettuano ottimizzazione sul codice intermedio per renderlo più efficiente:

- rimozione di codice inutile (dead code)
- espansione in linea di chiamate di funzioni
- fattorizzazione di sottoespressioni
- mettere fuori dai cicli sottoespressioni che non variano

Alla fine si ottiene un codice intermedio ottimizzato.

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?

Tabella dei simboli

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

- ad es. per le matrici, mette come attributo la dimensione e il tipo dei suoi elementi

Un esempio istruttivo

Prendiamo un piccolo frammento di un programma PASCAL

```
var position, initial, rate : real
position := initial + rate * 60
```

e vediamo come viene trasformata l'istruzione

position := initial + rate * 60

nelle varie fasi del compilatore

position := initial + rate * 60

↓

Analizzatore Lessicale

↓

<id , 1> <:=> <id , 2> <op,+> <id , 3> <op,*> <const,60>

più brevemente

$id_1 := id_2 + id_3 * 60$

dove 1, 2, e 3 sono indirizzi nella Tabella dei Simboli

		Attributo
1	position	real
2	initial	real
3	rate	real
	⋮	

$id_1 := id_2 + id_3 * 60$

↓

Analizzatore Sintattico

↓

Albero Sintattico

↓

Analizzatore Semantico

↓

Poiché tutti gli identificatori sono di tipo "real", allora tutte le operazioni saranno su "real" ed anche 60 deve essere convertito in "real"

↓

↓

Generazione del codice intermedio

↓

```
temp1 := into-real(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

- usiamo "three-address code" come linguaggio intermedio
- la generazione del codice segue l'albero sintattico bottom-up
- un temporaneo viene generato per memorizzare il risultato di ogni operazione
- ci sono istruzioni senza operazioni "vere" (l'ultima)

⇓

Ottimizziamo questo codice!

↓

Ottimizzazione

↓

$temp_1 := id_3 * 60.0$

$id_1 := id_2 * Temp_1$

conversione fatta dal compilatore (non necessaria la funzione into-real)

↓

Generazione del Codice

↓

```
MOVF  id3, R2
MULF  #60.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```

Codice Assembler (o codice macchina rilocabile)

la "F" sta per "floating-point arithmetics"

deve essere trattata come costante (e non come indirizzo di memoria)

Semantica Operazionale Strutturata

Linguaggio

- definito tramite sintassi astratta (semplice, intuitiva ma ambigua)
- una stringa viene sempre accoppiata ad un albero sintattico (non ambiguo)

Insiemi di base

- booleani : $\{tt, ff\}$ metavariabili $t, t_1, t' \in \Pi$
- numeri naturali : $\{0, 1, 2, \dots\}$ " $n, m, p \in \mathbb{N}$
- variabili : $\{a, b, c, \dots z\}$ " $v \in Var$

  ↖ in numero finito

Insiemi derivati (in notazione BNF)

- Espressioni Aritmetiche : $e \in Exp$

  $e ::= n \mid v \mid e+e \mid e-e \mid e*e$

- Espressioni Booleane : $b \in Bexp$

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

- Comandi : $c \in Com$

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

Sintassi concreta non ambigua molto più complicata, ma non serve nel dare semantica

Infatti:

Parser: Programma scritto in sintassi concreta $\longrightarrow$ Albero sintattico di sintassi astratta

Quindi, nel dare semantica, possiamo partire dagli alberi di sintassi astratta

---

while b do c_1 ; c_2

è ambigua in sintassi astratta

questo è l'albero inteso in sintassi concreta

Se vogliamo ottenere questo albero, dobbiamo usare begin e end

while b do begin
c_1 ; c_2
end

zucchero sintattico

in sintassi concreta

Come dare semantica?

Definiamo, per ogni categoria sintattica (cioè per Exp, Bexp e Com) un modello detto "Sistema di Transizione"

<u>Def</u> Un sistema di transizione è una tripla $\langle \Gamma, T, \rightarrow \rangle$ dove

- Γ è l'insieme (possibilmente infinito) di stati (o configurazioni)
- $T \subseteq \Gamma$ è l'insieme degli stati terminali
- $\rightarrow \subseteq \Gamma \times \Gamma$ è la relazione di transizione

- Una computazione a partire dallo stato γ_0 è una sequenza $\gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \dots$ che può essere finita o infinita
- Con $\rightarrow^*$ indichiamo la chiusura riflessiva e transitiva di $\rightarrow$, ovvero

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

<u>Esempio</u> (con rappresentazione grafica)

grafo in cui i nodi sono gli stati e gli archi le transizioni

Problemi

(1) Γ è di solito un insieme infinito contabile $\Rightarrow$ necessità di trovare una rappresentazione finita implicita attraverso grammatiche/BNF

Cioè Γ essenzialmente coincide con uno dei linguaggi delle 3 categorie sintattiche

E1:
$$\Gamma_e = \{ \langle e, \sigma \rangle \mid e \in \underbrace{Expr}_{\text{questo è definito in BNF}}, \sigma \in Store \}$$

(2) $\rightarrow \subseteq \Gamma \times \Gamma$ è una relazione costituita, tipicamente, da infinite coppie $\gamma \rightarrow \gamma'$

$\Rightarrow$ necessità di trovare una rappresentazione finita implicita come minima relazione che soddisfa un certo insieme finito di assiomi e regole d'inferenza

(3) Per dare significato alle variabili (che possono solo assumere valore su $\mathbb{N}$) è necessario introdurre uno Store $\sigma : Var \rightarrow \mathbb{N}$, come funzione che associa ad ogni variabile un valore

$$\sigma = \{ x_1/n_1, \; x_2/n_2, \; \dots \; x_k/n_k \}$$

se supponiamo che $Var = \{x_1, x_2, \dots x_k\}$

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

Semantica delle Espressioni Aritmetiche

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

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

(Var)
$$\frac{-}{\langle V, \sigma \rangle \rightarrow_e \langle \sigma(V), \sigma \rangle}$$
stato terminale

(Sum1)
$$\frac{\langle e_0, \sigma \rangle \rightarrow_e \langle e_0', \sigma' \rangle}{\langle e_0 + e_1, \sigma \rangle \rightarrow_e \langle e_0' + e_1, \sigma' \rangle}$$

(Sum2)
$$\frac{\langle e_1, \sigma \rangle \rightarrow_e \langle e_1', \sigma' \rangle}{\langle m + e_1, \sigma \rangle \rightarrow_e \langle m + e_1', \sigma' \rangle}$$

(Sum3)
$$\frac{-}{\langle m + m', \sigma \rangle \rightarrow_e \langle p, \sigma \rangle}$$
se $p = m + m'$

(Sub1)
$$\frac{\langle e_0, \sigma \rangle \rightarrow_e \langle e_0', \sigma' \rangle}{\langle e_0 - e_1, \sigma \rangle \rightarrow_e \langle e_0' - e_1, \sigma' \rangle}$$

(Sub2)
$$\frac{\langle e_1, \sigma \rangle \rightarrow_e \langle e_1', \sigma' \rangle}{\langle m - e_1, \sigma \rangle \rightarrow_e \langle m - e_1', \sigma' \rangle}$$

(Sub3)
$$\frac{}{\langle m - m', \sigma \rangle \rightarrow_e \langle p, \sigma \rangle}$$
se $m \geq m'$ e $p = m - m'$

Le 3 regole per la moltiplicazione fatele per esercizio (sono del tutto analoghe a quelle della somma).

Osservazioni:

(1) Gli assiomi e le regole SOS sono facilmente implementabili in PROLOG, definendo così un semplice (ma inefficiente) interprete per le espressioni aritmetiche

(2) Lo store σ non viene mai modificato durante la valutazione di una espressione!

- nessuno dei 3 assiomi (Var, Sum_3, Sub_3) va a modificare lo store σ
- se supponiamo che, per ogni regola, $\sigma = \sigma'$ nella premessa della regola, allora $\sigma = \sigma'$ anche nella conclusione! Ovvero σ non è mai modificato

→ [esempio di dimostrazione per induzione sulla prova (di $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$)]

(3) L'assioma Sub_3 permette di derivare la transizione $\langle m - m', \sigma \rangle \rightarrow_e \langle p, \sigma \rangle$ solo se $m \geq m'$. Altrimenti lo stato $\langle m - m', \sigma \rangle$ è bloccato, pur non essendo uno stato terminale

(è un caso di "errore", come vedremo)

(4) Le regole Sum_1 e Sum_2 riflettono una disciplina/regola di valutazione detta IS (<u>Interna Sinistra</u>):

- prima valutiamo l'argomento più a SX, ovvero e_0;
- quando terminiamo la valutazione di e_0, si passa a valutare e_1;
- quando termina anche la valutazione di e_1, allora si fa la somma (Sum_3)

<u>Alternativamente</u> ID (<u>Interna Destra</u>)

$$(Sum_1') \quad \frac{\langle e_1, \sigma \rangle \rightarrow_e \langle e_1', \sigma' \rangle}{\langle e_0 + e_1, \sigma \rangle \rightarrow_e \langle e_0 + e_1', \sigma' \rangle}$$

$$(Sum_2') \quad \frac{\langle e_0, \sigma \rangle \rightarrow_e \langle e_0', \sigma' \rangle}{\langle e_0 + m, \sigma \rangle \rightarrow_e \langle e_0' + m, \sigma' \rangle}$$

+ (Sum_3) invariata

- qui si valuta prima e_1, poi e_0 ed infine si fa la somma

Esempi di derivazione di transizioni

$$(\text{Var})\ \frac{}{\langle x, \{x/5, y/3\}\rangle \longrightarrow \langle 5, \{x/5, y/3\}\rangle}$$

$$(\text{Sum}_1)\ \frac{}{\langle x+2, \{x/5, y/3\}\rangle \longrightarrow \langle 5+2, \{x/5, y/3\}\rangle}$$

$$(\text{Sub}_1)\ \frac{}{\underbrace{\langle (x+2)-y, \{x/5, y/3\}\rangle}_{\gamma_0} \longrightarrow \underbrace{\langle (5+2)-y, \{x/5, y/3\}\rangle}_{\gamma_1}}$$

$$(\text{Sum}_3)\ \frac{}{\langle 5+2, \{x/5, y/3\}\rangle \longrightarrow \langle 7, \{x/5, y/3\}\rangle}$$

$$(\text{Sub}_1)\ \frac{}{\underbrace{\langle (5+2)-y, \{x/5, y/3\}\rangle}_{\gamma_1} \longrightarrow \underbrace{\langle 7-y, \{x/5, y/3\}\rangle}_{\gamma_2}}$$

$$(\text{Var})\ \frac{}{\langle y, \{x/5, y/3\}\rangle \longrightarrow \langle 3, \{x/5, y/3\}\rangle}$$

$$(\text{Sub}_2)\ \frac{}{\underbrace{\langle 7-y, \{x/5, y/3\}\rangle}_{\gamma_2} \longrightarrow \underbrace{\langle 7-3, \{x/5, y/3\}\rangle}_{\gamma_3}}$$

$$(\text{Sub}_3)\ \frac{}{\underbrace{\langle 7-3, \{x/5, y/3\}\rangle}_{\gamma_3} \longrightarrow \underbrace{\langle 4, \{x/5, y/3\}\rangle}_{\gamma_4}}$$

$$\boxed{\gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_3 \rightarrow \gamma_4 \in T_e}$$

cioè $\langle (x+2)-y, \{x/5, y/3\}\rangle \rightarrow^* \langle 4, \{x/5, y/3\}\rangle$

ottenendo così il valore dell'espressione in $\{x/5, y/3\}$

Vogliamo ora dimostrare che $\rightarrow_e$ è deterministica, ovvero che

se $\gamma \rightarrow_e \gamma'$ e $\gamma \rightarrow_e \gamma''$, allora $\gamma' = \gamma''$ $\forall \gamma, \gamma', \gamma''$

Useremo una tecnica di prova che si chiama

INDUZIONE STRUTTURALE

Supponiamo di voler dimostrare una certa proprietà $P(e)$ vera per ogni $e \in Exp$

$(e ::= m \mid v \mid e+e \mid e-e \mid e*e)$

Se dimostriamo che:

1) $\forall m \in \mathbb{N}$ $P(m)$ 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 $\forall e \in Exp$ $P(e)$ vera.

Nel nostro caso $P(e) = \forall \sigma, \gamma', \gamma''$

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

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

1) $e = m \in \mathbb{N}$ Osserva che $\langle m, \sigma \rangle \not\to$ e allora $P(m)$ è vera perché la premessa dell'implicazione è falsa (caso "vacuo")

2) $e = v \in Var$ Per la regola (Var), l'unica transizione derivabile per $\langle v, \sigma \rangle$ è

$$\langle v, \sigma \rangle \to_e \langle \sigma(v), \sigma \rangle$$

Poiché σ è una funzione (cioè $\sigma(v)$ è univoco) e c'è solo la regola (Var) che è applicabile, la Tesi segue:

$$\langle v, \sigma \rangle \to_e \gamma' \wedge \langle v, \sigma \rangle \to_e \gamma'' \Rightarrow \gamma' = \gamma''$$

3) $e = e_0 + e_1$ Supponiamo che $\langle e_0 + e_1, \sigma \rangle \to \gamma'$, $\langle e_0 + e_1, \sigma \rangle \to \gamma''$

Vogliamo dimostrare che $\gamma' = \gamma''$.

Ci sono 3 sottocasi da esaminare in accordo al modo in cui derivo $\langle e_0 + e_1, \sigma \rangle \to \gamma'$

(a) $\langle e_0, \sigma \rangle \to \langle e_0', \sigma' \rangle$ e $\gamma' = \langle e_0' + e_1, \sigma' \rangle$

In questo caso, $e_0 \notin \mathbb{N}$ e l'unica regola che ho applicato è (Sum_1). Allora se $\langle e_0 + e_1, \sigma \rangle \to \gamma''$, è necessario che

$$\langle e_0, \sigma \rangle \to \langle e_0'', \sigma'' \rangle \quad e \quad \gamma'' = \langle e_0'' + e_1, \sigma'' \rangle$$

Ma, per ipotesi induttiva, $P(e_0)$ vale, per cui deve essere $e_0' = e_0''$ e $\sigma' = \sigma''$, da cui discende $\gamma' = \gamma''$ <u>c.v.d.</u>

(b) $e_0 = m \in \mathbb{N}$ ed $\langle e_1, \sigma \rangle \to \langle e_1', \sigma' \rangle$

In questo caso, $e_1 \notin \mathbb{N}$ e la regola che ho applicato è (Sum_2). Allora se

$\langle e_0 + e_1, \sigma \rangle \to \gamma''$, è necessario che

$\langle e_1, \sigma \rangle \to \langle e_1'', \sigma'' \rangle$ e $\gamma'' = \langle e_0 + e_1'', \sigma'' \rangle$.

Ma, per ipotesi induttiva, $P(e_1)$ vale, per cui deve essere $e_1' = e_1''$ e $\sigma' = \sigma''$, da cui discende $\gamma' = \gamma''$ <u>c.v.d</u>

(c) $e_0 \in \mathbb{N}$ ed $e_1 \in \mathbb{N}$.

In questo caso, solo (Sum_3) è applicabile, ottenendo <u>una sola possibile transizione</u>

$\langle e_0 + e_1, \sigma \rangle \to \langle p, \sigma \rangle$ dove $p = e_0 + e_1$

Quindi la Tesi segue:

$\langle e_0 + e_1, \sigma \rangle \to \gamma' \wedge \langle e_0 + e_1, \sigma \rangle \to \gamma'' \Rightarrow \gamma' = \gamma''$

<u>c.v.d</u>

4) $e = e_1 - e_2$

5) $e = e_1 * e_2$

Questi 2 casi sono del tutto analoghi al caso $e = e_0 + e_1$

Fateli per esercizio

<u>Fine della prova</u>

Poiché $\rightarrow_e$ è deterministica, a partire da $\langle e, \sigma \rangle$, arriveremo su una sola configurazione terminale $\langle n, \sigma \rangle$: "$n$ è il valore di e in σ"

È possibile perciò definire una funzione

$$eval : Expr \times Store \rightarrow \mathbb{N}$$

che dia semantica alle espressioni

funzione parziale! perché non sempre raggiunge uno stato terminale

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

<u>Ad esempio</u>:

- $eval((x+2)-y, \{x/5, y/3\}) = 4$ perché

$$\langle (x+2)-y, \{x/5, y/3\} \rangle \rightarrow^* \langle 4, \{x/5, y/3\} \rangle$$

- $eval((x+2)-y, \{x/2, y/7\}) =$ indefinito perché

$$\langle (x+2)-y, \{x/2, y/7\} \rangle \rightarrow^* \langle 4-7, \{x/2, y/7\} \rangle \not\rightarrow$$

configurazione bloccata, ma non terminale

Equivalenza tra espressioni

$e \equiv e'$ sse $\forall \sigma \in Store$ $eval(e,\sigma) = eval(e',\sigma)$

(è equivalente a)

Ad es: $V_1 + (V_2 + V_3) \equiv (V_1 + V_2) + V_3$

Oss: Eval è definita rispetto alla disciplina di valutazione IS. A rigore $eval_{IS}$.
Si può dimostrare che anche per ID, il risultato della valutazione è lo stesso:

$$eval_{IS} = eval_{ID}$$

dove $eval_{ID}(e,\sigma) = \begin{cases} m & \text{se } \langle e,\sigma\rangle \xrightarrow[ID]{*} \langle m,\sigma\rangle \\ \text{indef.} & \text{altrimenti} \end{cases}$

Come vedremo, è possibile definire anche altre discipline di valutazione, come ES, ED, EP

E = esterna S = Sinistra D = Destra P = Parallel

ma queste sono diverse da IS = ID.

La regola IP prevede che le regole SOS per il + siano $(Sum_1^{*}) + (Sum_1') + (Sum_3)$

vedi foglio 5 vedi foglio 7 vedi fogl. 5

La relazione $\rightarrow_{IP}$ risulta nondeterministica ma "confluente", cioè termina in modo univoco.

Semantica delle Espr. Booleane

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

$\langle \Gamma_b, T_b, \rightarrow_b \rangle$ dove $\Gamma_b = \{\langle b, \sigma\rangle \mid b \in Bexp,\ \sigma \in Store\}$

$T_b = \{\langle tt, \sigma\rangle, \langle ff, \sigma\rangle \mid \sigma \in Store\}$

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

(Eq1) $$\frac{\langle e_0, \sigma\rangle \rightarrow_e \langle e_0', \sigma'\rangle}{\langle e_0 = e_1, \sigma\rangle \rightarrow_b \langle e_0' = e_1, \sigma'\rangle}$$

(Eq2) $$\frac{\langle e_1, \sigma\rangle \rightarrow_e \langle e_1', \sigma'\rangle}{\langle m = e_1, \sigma\rangle \rightarrow_b \langle m = e_1', \sigma'\rangle}$$

(Eq3) $$\frac{}{\langle m = n, \sigma\rangle \rightarrow_b \langle t, \sigma\rangle} \quad \text{dove } t = \begin{cases} tt & \text{se } m = n \\ ff & \text{se } m \neq n \end{cases}$$

IS

(Or1) $$\frac{\langle b_0, \sigma\rangle \rightarrow_b \langle b_0', \sigma'\rangle}{\langle b_0 \text{ or } b_1, \sigma\rangle \rightarrow_b \langle b_0' \text{ or } b_1, \sigma'\rangle}$$

(Or2) $$\frac{}{\langle tt \text{ or } b_1, \sigma\rangle \rightarrow_b \langle tt, \sigma\rangle}$$

(Or3) $$\frac{}{\langle ff \text{ or } b_1, \sigma\rangle \rightarrow_b \langle b_1, \sigma\rangle}$$

ES ← cominciò con l'arg. a sx

non valuto sempre tutte e due gli argomenti!

(Neg1) $$\frac{\langle b, \sigma\rangle \rightarrow_b \langle b', \sigma'\rangle}{\langle \sim b, \sigma\rangle \rightarrow_b \langle \sim b', \sigma'\rangle}$$

(Neg2) $$\frac{}{\langle \sim t, \sigma\rangle \rightarrow_b \langle t', \sigma\rangle} \quad \text{dove } t' = \begin{cases} tt & \text{se } t = ff \\ ff & \text{se } t = tt \end{cases}$$

Osservazioni:

(1) Lo store σ non viene mai modificato durante la valutazione di una bexp, perché:

- nessuno dei 4 assiomi (Eq_3, Or_2, Or_3, Neg_2) va a modificare lo store
- per le regole Eq_1 e Eq_2, sappiamo già che le transizioni nella premessa hanno $\sigma = \sigma'$, quindi anche nella conclusione $\sigma = \sigma'$
- Per le regole Or_1 e Neg_1, se supponiamo che $\sigma = \sigma'$ nella premessa, allora $\sigma = \sigma'$ anche nella conclusione

(Esempio di dimostrazione per induzione sulla prova:

"se $\langle b, \sigma \rangle \rightarrow_b \langle b', \sigma' \rangle$, allora $\sigma = \sigma'$")

Lo dimostro per induzione sulla prova di)

(2) $\rightarrow_b$ è deterministica, ovvero

se $\gamma \rightarrow_b \gamma'$ e $\gamma \rightarrow_b \gamma''$, allora $\gamma' = \gamma''$

(Si può dimostrare, come fatto per $\rightarrow_e$)

Per cui si può facilmente definire

$$eval_b(b, \sigma) = \begin{cases} t & \text{se } \langle b, \sigma \rangle \rightarrow^* \langle t, \sigma \rangle \\ \text{indefinito} & \text{altrimenti} \end{cases}$$

$b \equiv b'$ sse $\forall \sigma \in Store.\ eval_b(b, \sigma) = eval_b(b', \sigma)$

ad es: $\sim((3 = V) \text{ or } (3 = 4)) \equiv \sim(V = 3)$

Si possono definire per b_0 or b_1 regole di valutazione diverse da ES. Ad esempio ED o IS, ma non sono tutte equivalenti.

<u>ED</u>

$$(Or_1') \quad \frac{\langle b_1, \sigma\rangle \rightarrow_b \langle b_1', \sigma'\rangle}{\langle b_0 \text{ or } b_1, \sigma\rangle \rightarrow_b \langle b_0 \text{ or } b_1', \sigma'\rangle}$$

$$(Or_2') \quad \frac{}{\langle b_0 \text{ or } tt, \sigma\rangle \rightarrow_b \langle tt, \sigma\rangle}$$

$$(Or_3') \quad \frac{}{\langle b_0 \text{ or } ff, \sigma\rangle \rightarrow_b \langle b_0, \sigma\rangle}$$

<u>Esempio</u> $\gamma = \langle (3-5) = 6 \text{ or } tt, \sigma\rangle$

$\gamma \not\rightarrow_{ES}$ perché $3-5$ è bloccato (Sub3 non è applicabile)

$\gamma \rightarrow_{ED} \langle tt, \sigma\rangle$

In generale,

$$eval_{bES} \neq eval_{bED} \neq eval_{bIS} = eval_{bID}$$

<u>Esercizio</u>: Provate a definire le regole per b_0 or b_1 secondo la disciplina IS.

Altri esercizi / esempi

1) Descrivere le regole ES per $\langle e_0 * e_1, \sigma \rangle$

$$\frac{}{\langle 0 * e_1, \sigma \rangle \longrightarrow \langle 0, \sigma \rangle}$$

← non valuto e_1!

$$\frac{}{\langle 1 * e_1, \sigma \rangle \longrightarrow \langle e_1, \sigma \rangle}$$

← ho già fatto il prodotto, ma ora devo valutare e_1 per avere il risultato

$$\frac{\langle e_0, \sigma \rangle \longrightarrow \langle e_0', \sigma' \rangle}{\langle e_0 * e_1, \sigma \rangle \longrightarrow \langle e_0' * e_1, \sigma' \rangle}$$

$$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e_1', \sigma' \rangle}{\langle m * e_1, \sigma \rangle \longrightarrow \langle m * e_1', \sigma' \rangle} \quad m \neq 0, 1$$

$$\frac{}{\langle m * n, \sigma \rangle \longrightarrow \langle p, \sigma \rangle} \quad \begin{array}{l} m \neq 0, 1 \\ p = m * n \end{array}$$

2) Dimostrare che $eval_{IS} \neq eval_{ES}$ per $e_0 * e_1$

$$\langle 0 * (3-5), \sigma \rangle \not\longrightarrow_{IS}$$

$$\langle 0 * (3-5), \sigma \rangle \longrightarrow_{ES} \langle 0, \sigma \rangle$$

3) Regole ED per $e_0 * e_1$

4) Regole ES per b_0 and b_1

Semantica dei Comandi

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

$\langle \Gamma_c, T_c, \rightarrow_c \rangle$ dove $\Gamma_c = \{\langle c, \sigma \rangle \mid c \in Com, \sigma \in Store\} \cup \{\sigma \mid \sigma \in Store\}$

$T_c = \{\sigma \mid \sigma \in Store\} = Store$

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

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

(Ass) $$\frac{\langle e, \sigma \rangle \rightarrow_e^* \langle m, \sigma \rangle}{\langle v := e, \sigma \rangle \rightarrow_c \sigma[m/v]}$$

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

"aggiornamento dello store"

Ad es: se $\sigma = \{x/1, y/1\}$, allora $\sigma[3/x] = \{x/3, y/1\}$

(Seq1) $$\frac{\langle c_0, \sigma \rangle \rightarrow_c \langle c_0', \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \rightarrow_c \langle c_0'; c_1, \sigma' \rangle}$$

(Seq2) $$\frac{\langle c_0, \sigma \rangle \rightarrow \sigma'}{\langle c_0; c_1, \sigma \rangle \rightarrow \langle c_1, \sigma' \rangle}$$

(If$_1$)
$$\frac{\langle b,\sigma\rangle \to_b^* \langle tt,\sigma\rangle}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1,\sigma\rangle \to_c \langle c_0,\sigma\rangle}$$

(If$_2$)
$$\frac{\langle b,\sigma\rangle \to_b^* \langle ff,\sigma\rangle}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1,\sigma\rangle \to_c \langle c_1,\sigma\rangle}$$

(Wh$_1$)
$$\frac{\langle b,\sigma\rangle \to_b^* \langle tt,\sigma\rangle}{\langle \text{while } b \text{ do } c,\sigma\rangle \to_c \langle c;\text{while } b \text{ do } c,\sigma\rangle}$$

(Wh$_2$)
$$\frac{\langle b,\sigma\rangle \to_b^* \langle ff,\sigma\rangle}{\langle \text{while } b \text{ do } c,\sigma\rangle \to_c \sigma}$$

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

Esempi

(Wh$_1$)
$$\frac{\langle tt,\sigma\rangle \to_b^* \langle tt,\sigma\rangle}{\underbrace{\langle \text{while } tt \text{ do skip},\sigma\rangle}_{\gamma_0} \to_c \underbrace{\langle \text{skip};\text{while } tt \text{ do skip},\sigma\rangle}_{\gamma_1}}$$

(Skip)
$$\frac{}{\langle \text{skip},\sigma\rangle \to \sigma}$$

(Seq$_2$)
$$\frac{}{\underbrace{\langle \text{skip};\text{while } tt \text{ do skip},\sigma\rangle}_{\gamma_1} \to \underbrace{\langle \text{while } tt \text{ do skip},\sigma\rangle}_{\gamma_0}}$$

Programma che non termina mai!

$$(Wh_1)\ \frac{\langle tt,\sigma\rangle \to_b^* \langle tt,\sigma\rangle \qquad \sigma=\{x/_0\}}{\underbrace{\langle \text{while } tt \text{ do } x:=x+1,\sigma\rangle}_{\gamma_0} \to \underbrace{\langle x:=x+1;\text{while}\ldots,\sigma\rangle}_{\gamma_0'}}$$

$$(Var)\ \frac{-}{\langle x,\sigma\rangle \to \langle 0,\sigma\rangle} \qquad \sigma=\{x/_0\}$$

$$(Sum_1)\ \frac{\langle x+1,\sigma\rangle \to \langle 0+1,\sigma\rangle \to^* \langle 1,\sigma\rangle}{} \quad \text{usando } (Sum_3)$$

$$(Ass)\ \frac{\langle x:=x+1,\sigma\rangle \to \sigma[1/x]=\{x/_1\}}{}$$

$$(Seq_2)\ \frac{}{\underbrace{\langle x:=x+1;\text{while } tt \text{ do } x:=x+1,\sigma\rangle}_{\gamma_0'} \to \underbrace{\langle \text{while}\ldots,\{x/_1\}\rangle}_{\gamma_1}}$$

$$\gamma_i = \langle \text{while } tt \text{ do } x:=x+1,\ \{x/_i\}\rangle$$

$$\gamma_i' = \langle x:=x+1;\ \text{while } tt \text{ do } x:=x+1,\ \{x/_i\}\rangle$$

$$\gamma_0 \to \gamma_0' \to \gamma_1 \to \gamma_1' \to \gamma_2 \to \gamma_2' \to \ldots$$

$$(Sub_3)\ \frac{}{\langle 3-5,\sigma\rangle \not\to}$$

$$(Ass)\ \frac{}{\underbrace{\langle x:=(3-5),\sigma\rangle}_{\gamma} \not\to}$$

γ

CODICE BLOCCATO

$3-5 \not\geq 0$

Prop: $\longrightarrow_c$ è deterministica, cioè

"se $\gamma \rightarrow_c \gamma'$ e $\gamma \rightarrow_c \gamma''$, allora $\gamma' = \gamma''$"

$$\text{exec} : \text{Com} \times \text{Store} \xrightarrow{\text{parziale}} \text{Store}$$

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

Oss: Divergenza e deadlock sono equiparati

$c_1 = \text{while tt do skip}$

$\langle c_1, \sigma \rangle \longrightarrow \langle \text{skip}; c_1, \sigma \rangle$

$c_2 = x := (3-5)$

$\langle c_2, \sigma \rangle \not\rightarrow$

$$\text{exec}(c_1, \sigma) = \boxed{\text{indefinito}} = \text{exec}(c_2, \sigma)$$

Allora, se vogliamo semanticamente distinguere questi aspetti computazionali, dobbiamo raffinare la semantica, introducendo più dettagli.

Errori Dinamici (a run time)

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

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

$eval_b : Bexp \times Store \longrightarrow \mathbb{T} \cup \{err\}$

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

(rimane parziale per divergenza)

Regola per generare "errore"

$$(Sub_4) \quad \frac{}{\langle m - m', \sigma \rangle \rightarrow err} \; m' > m$$

unico errore possibile nel nostro semplice ling.

Regole per propagare l'errore (in Exp)

$$(Sum_4) \quad \frac{\langle e_0, \sigma \rangle \rightarrow_e err}{\langle e_0 + e_1, \sigma \rangle \rightarrow_e err}$$

$$(Sum_5) \quad \frac{\langle e_1, \sigma \rangle \rightarrow_e err}{\langle m + e_1, \sigma \rangle \rightarrow_e err}$$

analoghe regole anche per $-$ e $*$

Regole per propagare l'errore (in Bexp)

- analoghe a per Eq (Eq_4 e Eq_5)

$$(Or_4) \quad \frac{\langle b_0, \sigma \rangle \rightarrow_b err}{\langle b_0 \text{ or } b_1, \sigma \rangle \rightarrow_b err}$$

$$(Neg_3) \quad \frac{\langle b, \sigma \rangle \rightarrow_b err}{\langle \sim b, \sigma \rangle \rightarrow_b err}$$

Propagazione di errore (in Com)

(Ass$_2$) $$\frac{\langle e,\sigma\rangle \rightarrow_e^* err}{\langle v := e,\sigma\rangle \rightarrow_c err}$$

(Seq$_3$) $$\frac{\langle c_0,\sigma\rangle \rightarrow_c err}{\langle c_0; c_1,\sigma\rangle \rightarrow_c err}$$

(If$_3$) $$\frac{\langle b,\sigma\rangle \rightarrow_b^* err}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1,\sigma\rangle \rightarrow_c err}$$

(Wh$_3$) $$\frac{\langle b,\sigma\rangle \rightarrow_b^* err}{\langle \text{while } b \text{ do } c,\sigma\rangle \rightarrow_c err}$$

- Tutte queste regole per un solo tipo di errore dinamico ((Sub$_4$)).
- Altri errori dinamici in un ling. più esteso:
  - se ammettiamo la divisione
    - divisione per zero
  - se ammettiamo un tetto massimo (maxinteger)
    - overflow
  - se includiamo la radice quadrata
    - $\sqrt{\text{negativi}}$
  - se aggiungiamo gli array
    - superare i limiti della dimensione dell'array

⋮

(servono moltissime regole!)

Nondeterminismo e Parallelismo

$$c ::= skip \mid v := e \mid \ldots \mid \underline{c \text{ or } c} \mid \underline{c \text{ par } c}$$

(Nd1)
$$\frac{-}{\langle c_0 \text{ or } c_1, \sigma \rangle \longrightarrow_c \langle c_0, \sigma \rangle}$$

(Nd2)
$$\frac{}{\langle c_0 \text{ or } c_1, \sigma \rangle \longrightarrow_c \langle c_1, \sigma \rangle}$$

(Par1)
$$\frac{\langle c_0, \sigma \rangle \longrightarrow_c \langle c_0', \sigma' \rangle}{\langle c_0 \text{ par } c_1, \sigma \rangle \longrightarrow_c \langle c_0' \text{ par } c_1, \sigma' \rangle}$$

(Par2)
$$\frac{\langle c_0, \sigma \rangle \longrightarrow_c \sigma'}{\langle c_0 \text{ par } c_1, \sigma \rangle \longrightarrow_c \langle c_1, \sigma' \rangle}$$

(Par3)
$$\frac{\langle c_1, \sigma \rangle \longrightarrow_c \langle c_1', \sigma' \rangle}{\langle c_0 \text{ par } c_1, \sigma \rangle \longrightarrow_c \langle c_0 \text{ par } c_1', \sigma' \rangle}$$

(Par4)
$$\frac{\langle c_1, \sigma \rangle \longrightarrow \sigma'}{\langle c_0 \text{ par } c_1, \sigma \rangle \longrightarrow \langle c_0, \sigma' \rangle}$$

- Verificare che $\longrightarrow_c$ non è più deterministica sul linguaggio esteso con "or" e "par"

Per farlo vedere, costruisco il sistema di transizione per $(x := 0; x := x + 1)$ par $x := 1$

$$\langle (x:=0;\ x:=x+1) \text{ par } x:=1,\ \{x/_0\}\rangle$$

$$\langle x:=x+1 \text{ par } x:=1,\ \{x/_0\}\rangle \qquad \langle x:=0;\ x:=x+1,\ \{x/_1\}\rangle$$

$$\langle x:=x+1,\ \{x/_1\}\rangle \qquad \langle x:=1,\ \{x/_1\}\rangle \qquad \langle x:=x+1,\ \{x/_0\}\rangle$$

$$\{x/_2\} \qquad \{x/_1\}$$

- Due possibili risultati finali: $\{x/_1\}$ o $\{x/_2\}$
- Per modellare questa situazione,

$$\text{exec}: \text{Com} \times \text{Store} \longrightarrow \mathcal{P}(\text{Store})$$

$$\text{exec}\left((x:=0;\ x:=x+1) \text{ par } x:=1,\ \{x/_0\}\right) = \{\{x/_1\},\ \{x/_2\}\}$$

Gestione dell'errore dinamico

$$\frac{\langle c_0, \sigma\rangle \rightarrow_c \text{err}}{\langle c_0 \text{ par } c_1, \sigma\rangle \rightarrow_c \text{err}} \qquad \frac{\langle c_1, \sigma\rangle \rightarrow_c \text{err}}{\langle c_0 \text{ par } c_1, \sigma\rangle \rightarrow_c \text{err}}$$

Capitolo 3

Analisi Lessicale – Linguaggi Regolari

Analisi Lessicale: riconoscere nella stringa in ingresso gruppi/sequenze di simboli che corrispondono a specifiche categorie sintattiche (ad es: identificatori, parole riservate, operatori aritmetici, ...)

La stringa in input è trasformata in una sequenza di simboli astratti, detti <u>token</u>

Cos'è un token?

Token = coppia (nome, valore)

- nome = simbolo astratto che rappresenta una categoria sintattica
- valore = una sequenza di simboli del testo in ingresso

ad es: <Ide, x_1>

Dato il token <Ide, x_1>, diciamo che

- Nome (Ide) è l'informazione che identifica una classe di token (identificatori)
- Valore (x_1) è l'informazione che identifica uno specifico token (l'identificatore x_1)
- Pattern è la descrizione generale della forma dei valori di una classe di token. Ad esempio, $\underbrace{(x|y)(x|y|0|1)^*}_{\text{espressione regolare (che vedremo)}}$
- Lessema è una singola istanza di un pattern.
  Nel nostro esempio x_1 è un lessema istanza del pattern $(x|y)(x|y|0|1)^*$

Vedremo che ad ogni "nome" di categoria sintattica è associato un "pattern" che specifica i possibili "valori" che possono essere presi per quel nome, come "lessemi".

Esempio

Dalla stringa C

```
if (x == 0) printf("zero")
```

un analizzatore lessicale potrebbe produrre la seguente sequenza di token

<IF> <(> <Ide, x> <OPREL, ==>

<CONST-NUM, 0> <)> <Ide, printf>

<(> <CONST-STRING, zero> <)>

In realtà, normalmente lo scanner associa agli identificatori un indirizzo nella tabella dei simboli

<Ide, x> è in realtà <Ide, puntatore alla tabella dei simboli>

Espressioni Regolari

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

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

CONCATENAZIONE DISGIUNZIONE

(Questa è una sintassi astratta ambigua; per disambiguare, possiamo usare le parentesi)

Per semplicità, si assume che:

- la concatenazione, la disgiunzione (e anche la ripetizione) associno a sx
- la precedenza tra gli operatori sia: $* > \cdot > |$
- la concatenazione $\cdot$ è di solito omessa

per cui, ad esempio, $b^*a \mid c$ corrisponde all'albero sintattico

$(((b)^*) \cdot (a)) \mid (c)$

secondo la sintassi del libro

Linguaggio denotato da una espressione regolare

Dato l'alfabeto A, definiamo la funzione

$$\mathcal{L} : \text{Exp-Reg} \to \mathcal{P}(A^*)$$

come segue:

INDUZIONE STRUTTURALE

simbolo di espressione regolare

$\mathcal{L}[\phi] = \phi$ ← linguaggio vuoto

$\mathcal{L}[\varepsilon] = \{\varepsilon\}$ ← ling. che contiene una sola stringa, quella vuota

$\mathcal{L}[a] = \{a\}$

$\mathcal{L}[r_1 \cdot r_2] = \mathcal{L}[r_1] \cdot \mathcal{L}[r_2]$

$\mathcal{L}[r_1 | r_2] = \mathcal{L}[r_1] \cup \mathcal{L}[r_2]$

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

Se usate le parentesi: $\mathcal{L}[(r)] = \mathcal{L}[r]$

Ricordo che:

$$L_1 \cdot L_2 = \{ xy \mid x \in L_1, y \in L_2 \}$$

$$L_1 \cup L_2 = \{ x \mid x \in L_1 \text{ o } x \in L_2 \}$$

$$L^0 = \{\varepsilon\} \qquad L^{n+1} = L \cdot L^n$$

$$L^* = \bigcup_{n \geq 0} L^n \qquad L^+ = \bigcup_{n \geq 1} L^n$$

Linguaggio Regolare

Def: Un linguaggio $L \subseteq A^*$ è detto regolare sse $\exists$ una espressione regolare r tale che

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

Prop: Ogni linguaggio finito è regolare

Ad es: $L = \{a, bc\}$ $\quad r = a \mid bc$

$$\begin{aligned}\mathcal{L}[a \mid bc] &= \mathcal{L}[a] \cup \mathcal{L}[bc] = \\ &= \{a\} \cup \mathcal{L}[b] \cdot \mathcal{L}[c] = \\ &= \{a\} \cup \{b\} \cdot \{c\} = \\ &= \{a, bc\} = L\end{aligned}$$

(Non serve mai l'operatore * di ripetizione)

Oss: Esistono linguaggi regolari infiniti

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

$$\mathcal{L}[a \mid a^* b] = \mathcal{L}[a] \cup \mathcal{L}[a^* b] = \{a\} \cup \{a^n b \mid n \geq 0\}$$

$$\begin{aligned}\mathcal{L}[(a \mid b) \cdot b^*] &= \mathcal{L}[a \mid b] \cdot \mathcal{L}[b^*] = (\mathcal{L}[a] \cup \mathcal{L}[b]) \cdot (\mathcal{L}[b])^* \\ &= \{a, b\} \cdot \{b\}^* = \{ab^n \mid n \geq 0\} \cup \{b^n \mid n \geq 1\}\end{aligned}$$

Esempi di Espr. Regolari

$A = \{0, 1\}$

- $0^* 1 0^*$

  $L_1 = \{w \in A^* \mid w \text{ contiene un solo } 1\}$

- $(0|1)^* 1 (0|1)^*$

  $L_2 = \{w \in A^* \mid w \text{ contiene almeno un } 1\}$

- $(0|1)^* 001 (0|1)^*$

  $L_3 = \{w \in A^* \mid w \text{ contiene } 001 \text{ come sottostringa}\}$

- $1^* (0 1 1^*)^*$ nota: 010110111 valido poiché con $(\ldots)^*$ i rantini possono essere uguali

  $L_4 = \{w \in A^* \mid \text{ogni occorrenza di } 0 \text{ è seguita immediatamente da almeno un } 1\}$

- $((0|1)(0|1))^*$

  $L_5 = \{w \in A^* \mid w \text{ è di lunghezza pari}\}$

- $(0|1)^* 1$

  $L_6 = \{w \in A^* \mid w \text{ termina con } 1\}$

Altri Operatori Ausiliari

Ripetizione positiva: r^+ $\quad (rr^* \text{ o } r^*r)$

Possibilità: $r?$ $\quad (r|\varepsilon)$

Elenco: $[a_1, \dots, a_n]$ per $a_1, \dots, a_n \in A$
(sta per $a_1|a_2|\dots|a_n$)

Se gli a_i sono ordinati in modo che $a_i < a_{i+1}$
allora
$[a_1 - a_n]$ (sta per $a_1|a_2|\dots|a_n$)

Esempio

- Numeri decimali senza segno

$$[0-9]^+ (\varepsilon \mid . [0-9]^+)$$

dove l'alfabeto è $\{0, 1, \dots, 9, .\}$

Alternativamente: $[0-9]^+ (. [0-9]^+)?$

Definizioni Regolari

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

$$d_1 := r_1$$
$$d_2 := r_2$$
$$\vdots$$
$$d_k := r_k$$

dove i vari d_i sono simboli "nuovi" e ogni r_i è una espressione regolare sull'alfabeto esteso $A \cup \{d_1, \dots, d_k\}$

Esempi

(1) Numeri decimali con segno

numconsegno := $\boxed{\text{segno cifre (. cifre)?}}$

segno := − | +

cifre := cifra$^+$

cifra := [0 − 9]

$(-|+)\ [0\text{-}9]^+\ (.\ [0\text{-}9]^+)?$

(2) Identificatori

identifier := letter (letter | digit)*

letter := [a − z A − Z]

digit := [0 − 9]

Esercizio: Provate a definire un tipo particolare di identificatore che deve soddisfare questi requisiti:

- iniziare con una lettera maiuscola
- contenere almeno una cifra
- terminare col simbolo "!"

Alfabeto = { a, ..., z, A, ..., Z, 0, ..., 9, ! }

$[A-Z]\left([a-9]\right)^*\ [0-9]\ \left([a-9]\right)^*\ !$

Equivalenza Tra Espr. Regolari

Def. Due espressioni regolari r e s sono equivalenti sse $\mathcal{L}[r] = \mathcal{L}[s]$ (cioè denotano lo stesso linguaggio) e lo denotiamo con $r \simeq s$

Esistono molte leggi per $\simeq$. Alcune sono le seguenti:

$r \mid s \simeq s \mid r$	$\mid$ è commutativa
$r \mid (s \mid t) \simeq (r \mid s) \mid t$	$\mid$ è associativa
$r \mid r \simeq r$	$\mid$ è idempotente
$r \cdot (s \cdot t) \simeq (r \cdot s) \cdot t$	$\cdot$ è associativa
$\varepsilon \cdot r \simeq r \simeq r \cdot \varepsilon$	ε è l'elemento neutro per $\cdot$
$(r^*)^* \simeq r^*$	$*$ è idempotente
$r(s \mid t) \simeq rs \mid rt$	$\cdot$ distribuisce a sx su $\mid$
$(r \mid s)t \simeq rt \mid st$	$\cdot$ distribuisce a dx su $\mid$

$\phi^* \simeq \varepsilon$	$(\varepsilon \mid r)^* \simeq r^*$
$r \mid \phi \simeq r$	$(r^* s^*)^* \simeq (r \mid s)^*$
$r \cdot \phi \simeq \phi \simeq \phi \cdot r$	$\vdots$

Dimostrare queste ~~eqq~~ leggi non è sempre facile, ma in alcuni casi lo è.

$$\mathcal{L}[r|s] = \mathcal{L}[r] \cup \mathcal{L}[s] = \mathcal{L}[s] \cup \mathcal{L}[r] = \mathcal{L}[s|r]$$

$$\mathcal{L}[r|r] = \mathcal{L}[r] \cup \mathcal{L}[r] = \mathcal{L}[r]$$

$$\mathcal{L}[r\varepsilon] = \mathcal{L}[r] \cdot \{\varepsilon\} = \mathcal{L}[r]$$

$$\mathcal{L}[\phi^*] = (\mathcal{L}[\phi])^* = \phi^* = \phi^0 \cup \phi^1 \cup \phi^2 \ldots$$
$$= \{\varepsilon\} \cup \phi \cup \phi \ldots = \{\varepsilon\} = \mathcal{L}[\varepsilon]$$

$$\mathcal{L}[r \cdot \phi] = \mathcal{L}[r] \cdot \mathcal{L}[\phi] = \mathcal{L}[r] \cdot \phi = \phi = \mathcal{L}[\phi]$$

Le espressioni regolari servono per specificare il pattern di una categoria sintattica, ovvero la forma dei possibili lessemi.

Ma come riconoscere se una certa sequenza in ingresso è un lessema per una certa categoria sintattica?

$\Rightarrow$ Automi (a stati finiti)

Automi (a stati) Finiti

controllo finito

Caratteristiche:

- memoria finita (data dal numero degli stati $q_0, \dots q_n$)
- input: stringa da riconoscere
- output: 1 bit (Sì/No)

Descrizione iniziale

- testina di lettura posizionata sul primo carattere dell'input
- controllo su stato iniziale q_0

Funzionamento: Ripeti

- leggi il carattere in input; in base allo stato in cui si trova decide:
  - di cambiare di stato
  - e di spostare la testina sull'input successivo

  fino a che
- ha finito di leggere l'input (e riconosce la stringa se ha raggiunto uno stato finale)
- oppure si è bloccato prima perché per la coppia (stato corrente, input attuale) non era specificato uno stato successivo

Diagrammi di Transizione

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

"aaaa" non finisce in q_1 poiché ha numero di a PARI

Riconoscere una stringa w significa trovare un cammino etichettato w sul grafo a partire dallo stato iniziale che finisce su uno stato finale

$$L = \{ a^{2n+1} \mid n \geq 0 \} = \{ a^n \mid n \text{ è dispari} \}$$
$$= \mathcal{L}[a(aa)^*]$$

N.B. Se in input ho w = ab, l'automa si blocca in q_1, senza essere stata completata la lettura dell'input w

sta a riassumere 10 diverse transizioni, una per ciascun simbolo 0,..,9

diagramma di transizione per $[0\text{-}9]^+(\varepsilon \mid . [0\text{-}9]^+)$

Esempi

$L = \{ w \in \{0,1\}^* \mid$ in w il numero di 0 e 1 è sempre pari $\}$

N.B. Deterministico: per ogni coppia (q, s), con $q \in \{q_0, q_1, q_2, q_3\}$ e $s \in \{0,1\}$, esiste una e una sola mossa possibile

$L = \mathcal{L}[(a|b)^* ba]$

N.B.: • $ba \in [M]$ perché esiste un cammino da q_0 a q_2 etichettato ba

• NONDETERMINISTICO
- (q_0, b) offre 2 mosse o su q_0 o su q_1
- (q_1, b) non offre mosse
- $(q_2, a/b)$ non offre mosse

$L[M] = \mathcal{L}[a^* b^*]$

- nondeterministico:

transizioni ε $\longrightarrow$ in q_0 può non spostare la testina di lettura (ovvero non leggere l'input) e spostarsi in q_1

Se vogliamo un automa deterministico per $\mathcal{L}[a^*b^*]$, possiamo definire

$\Sigma = \{0, 1\}$ $Q = \{q_0, q_1, q_2\}$
$F = \{q_0, q_1\}$

δ	a	b
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_2	q_2

stato pozzo (di errore)

- non ci sono transizioni etichettate ε
- da ogni stato per ognuno dei due simboli (a e b), esce una e una sola transizione

↳ questo automa è deterministico

Automi Finiti Nondeterministici (NFA)

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

- Σ è un alfabeto finito di simboli in input
- Q è un insieme finito di stati
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati finali
- δ è la funzione di transizione con tipo

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

↳ insieme delle parti di Q

$$\boxed{\delta(q, \sigma) = Q' \subseteq Q}$$

Esempio

$\Sigma = \{a, b\}$ $\quad Q = \{q_0, q_1, q_2\}$ $\quad q_0$ iniziale

$F = \{q_2\}$

δ	a	b	ε
q_0	$\{q_0\}$	$\{q_0, q_1\}$	$\emptyset$
q_1	$\{q_2\}$	$\emptyset$	$\emptyset$
q_2	$\emptyset$	$\emptyset$	$\emptyset$

Come rappresentare questo NFA con un diagramma di transizioni?

Esercizio • Disegnate il diagramma di transizione per $M = (\Sigma, Q, \delta, q_0, F)$ dove

$\Sigma = \{0, 1\}$ $\quad Q = \{q_0, q_1, q_2\}$ $\quad q_0$ iniziale $\quad F = \{q_1\}$

δ	0	1	ε
q_0	$\emptyset$	$\{q_0, q_2\}$	$\{q_1\}$
q_1	$\{q_1\}$	$\emptyset$	$\{q_0\}$
q_2	$\{q_2, q_0\}$	$\emptyset$	$\emptyset$

(1) • Ricavare dai diagrammi di transizione a pg 14 la definizione di NFA come quintupla $(\Sigma, Q, \delta, q_0, F)$

Linguaggio Riconosciuto/Accettato

Def (informale) Un NFA $N = (\Sigma, Q, \delta, q_0, F)$ accetta $w = a_1 \dots a_n$ sse nel diagramma di transizione esiste un cammino da q_0 ad uno stato in F nel quale la stringa che si ottiene concatenando le etichette degli archi percorsi è esattamente w.

Più formalmente:

- **Descrizione istantanea**: (q, w)
  - q: stato corrente
  - w: input da leggere
- **Mossa**

$$\frac{q' \in \delta(q, \sigma)}{(q, \sigma w) \vdash_N (q', w)} \qquad \sigma \in \Sigma \cup \{\varepsilon\}, \; w \in \Sigma^*$$

- **Cammino** (chiusura riflessiva e transitiva di $\vdash_N$)

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

- **Accettazione/Riconoscimento**

$$w \in L[N] \text{ sse } \exists q \in F.\ (q_0, w) \vdash_N^* (q, \varepsilon)$$

- Il linguaggio accettato da N, indicato con $L[N]$, è

$$L[N] = \{ w \in \Sigma^* \mid \exists q \in F.\ (q_0, w) \vdash_N^* (q, \varepsilon) \}$$

- Due NFA N_1 e N_2 si dicono equivalenti sse accettano lo stesso ling., cioè se $L[N_1] = L[N_2]$

Esempi / Esercizi

Costruire un NFA per i linguaggi

$L_1 = \{ a^n b^m \mid n, m \geq 1 \}$

$L_2 = \{ a^{3k+2} \mid k \geq 0 \}$

$L_3 = \{ w \in (a|b)^* \mid w$ contiene la sottostringa $aba \}$

$L_4 = \{ w \in (a|b)^* \mid w$ contiene esattamente 3 "a" $\}$

Soluzioni

$L[N_1] = L_1$

$L[N_2] = L_2$

$L[N_3] = L_3$

$L[N_4] = L_4$

<u>N.B.</u> Per un linguaggio L, se esiste un NFA N tale che $L = L[N]$, allora $\exists$ infiniti $N_1, N_2, \ldots N_k, \ldots$ tali che $L = L[N_i]$ per $i = 1, 2, \ldots$

$L[N_1] = \mathcal{L}[(a|b)^*ba]$

$L[N_2] = (a|b)^*ba$

e ne vedremo altri sempre per questo linguaggio!

NFA: – sono "comodi": facile costruirli, ma
– inefficienti: accettare w significa cercare un cammino su un grafo "nondeterministico" ⇒ tante potenziali strade alternative (possibili failure ⇒ backtracking)

DFA: Deterministic Finite Automata

– $\delta(q, \sigma)$ è sempre un singoletto (solo una mossa possibile)
– non ci sono le mosse ε

⇒ • scansione completa dell'input garantita (non si blocca mai)
• in un tempo $O(n)$ – dove n è la lunghezza di w – sappiamo se w è accettata o meno
• ma più difficili da definire

Automi Finiti Deterministici

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

$$(\delta(q, \sigma) = q')$$

Osservazione: Un DFA è un particolare tipo di NFA tale che:

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

NFA o DFA ?

$\Sigma = \{a\}$

NFA, DA 1 ESCONO 0 ARCHI

ORA DFA

NFA o DFA ?

$\Sigma = \{a, b\}$

NFA, non completo per tutti gli elementi di Σ

Vogliamo ora dimostrare che i DFA sono tanto espressivi quanto gli NFA, sebbene siano un sottoinsieme proprio degli NFA.

Prop: 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

Tecnica detta "Costruzione per sottoinsiemi" (Subset Construction)

Prima di formalizzarla, vediamo un paio d'esempi.

Primo esempio semplice

NFA che vogliamo trasformare in un equivalente DFA

Stato iniziale del DFA: $\{q_0\} = A$

stato finale perché contiene $q_2 \in F$

questo è un DFA del tutto equivalente

- Passando dallo stato $A = \{q_0\}$ allo stato $B = \{q_0, q_1\}$, ho seguito contemporaneamente tutte e due le transizioni etichettate b
- Passando dallo stato $B = \{q_0, q_1\}$ a $C = \{q_0, q_2\}$, ho seguito sia la transizione $q_0 \xrightarrow{a} q_0$, sia la transizione $q_1 \xrightarrow{a} q_2$

$$\text{mossa}(B, a) = \bigcup_{q \in B} \delta(q, a) = \delta(q_0, a) \cup \delta(q_1, a) = \{q_0, q_2\} = C$$

Secondo Esempio

NFA

Lo stato iniziale del DFA è dato da 0, più tutti gli stati che possono raggiungere da 0 con mosse ε

$$\varepsilon\text{-closure}(0) = \{0, 1, 2, 3, 7\} \quad (A)$$

Per calcolare lo stato che raggiungo da A leggendo "a", devo vedere quali stati in A possono fare "a" e poi farne la ε-closure

$$\Delta(A, a) = \varepsilon\text{-closure}(\text{mossa}(A, a)) =$$
$$= \varepsilon\text{-closure}\left(\bigcup_{q \in A} \delta(q, a)\right) =$$
$$= \varepsilon\text{-closure}(\{4\}) = \{\underline{4}, 6, 7, 1, 2, 3\} = B$$

$$\Delta(A, b) = \varepsilon\text{-closure}(\text{mossa}(A, b)) =$$
$$= \varepsilon\text{-closure}\left(\bigcup_{q \in A} \delta(q, b)\right) =$$
$$= \varepsilon\text{-closure}(\{5, 8\}) = \{1, 2, 3, \underline{5}, 6, 7, \underline{8}\} = C$$

$$\Delta(B, a) = \varepsilon\text{-closure}(\{4\}) = B$$
$$\Delta(B, b) = \varepsilon\text{-closure}(\{5, 8\}) = C$$
$$\Delta(C, a) = \varepsilon\text{-closure}(\{4, 9\}) = \{\underline{4}, \underline{9}, 1, 2, 3, 6, 7\} = D$$

finale

$$\Delta(C, b) = \varepsilon\text{-closure}(\{5, 8\}) = C$$

$\Delta(D, a) = \varepsilon\text{-closure}(\{4\}) = B$

$\Delta(D, b) = \varepsilon\text{-closure}(\{5, 8\}) = C$

DFA ottenuto per costruzione per sottoinsiemi

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

$$\frac{}{\{q\} \subseteq \varepsilon\text{-closure}(q)} \qquad \frac{p \in \varepsilon\text{-closure}(q)}{\delta(p, \varepsilon) \subseteq \varepsilon\text{-closure}(q)}$$

Sia P un insieme di stati di un NFA.

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

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

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

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

funzione di transizione del DFA

Algoritmo per calcolare la ε-closure(P)

NON LO CHIEDE ALL'ESAME

```
Inizializzare T = P;
Inizializzare ε-closure(P) = P;
while T ≠ ∅ 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;
                        }
        }
```

Oss: Usando ε-closure, si può definire il ling. riconosciuto da un NFA in modo elegante:

$$\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$$

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

$$\hat{\delta}(q, xa) = \varepsilon\text{-closure}(P)$$

dove $P = \{ p \in Q \mid \exists r \in \hat{\delta}(q, x) \text{ e } p \in \delta(r, a) \}$

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

Costruzione dei Sottoinsiemi

Dato un NFA $N = (\Sigma, Q, \delta, q_0, F)$:

- Inizializza $S = \varepsilon\text{-closure}(q_0)$; // S stato iniziale del DFA
- Inizializza $T = \{S\}$; // T è l'insieme degli stati del DFA
  // S non è marcato all'inizio
- Finché c'è un $P \in T$ non marcato {
  - marca P;
  - for each $a \in \Sigma$ {
    - $R = \varepsilon\text{-closure}(\text{mossa}(P, a))$;
    - if $R \notin T$ {
      add R to T; // R non ha marca
      }
    - definisci $\Delta(P, a) = R$;
    }
  }

Definiamo il DFA $M_N = (\Sigma, T, \Delta, \varepsilon\text{-closure}(q_0), \mathcal{F})$ dove $R \in \mathcal{F}$ sse $\exists q \in R$ con $q \in F$.

- 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 $n = |Q|$.
  ↑ esponenziale
- Di solito T è molto più piccolo di $\mathcal{P}(Q)$

Esempio di caso pessimo

NFA N con 2 stati

$A = \{q_0\}$

$B = \{q_0, q_1\}$

$C = \{q_1\}$

$D = \emptyset$

DFA M_N con $2^2 = 4$ stati

Il DFA M_N può essere "esponenzialmente" più grande dell'NFA N

Altro caso molto sfortunato

$L[N] = \{w \in (a|b)^* \mid$ il simbolo n-esimo da dx è $b\}$

Anche in questo caso, gli stati del DFA sono 2^n mentre gli stati dell'NFA sono $n+1$

Provate a disegnare il DFA per

vedrete che avrà 2^2 stati

Esempio Composto

NFA N

$A = \varepsilon\text{-closure}(q_0) = \{q_0, q_1, q_2, q_3\}$

$\Delta(A, a) = \varepsilon\text{-closure}(\text{mossa}(A, a)) = \varepsilon\text{-closure}(\{q_0, q_4\}) =$

$= \{q_0, q_1, q_2, q_3, q_4\} = B$

$\Delta(A, b) = \varepsilon\text{-closure}(\text{mossa}(A, b)) = \varepsilon\text{-closure}(\{q_2, q_4\}) =$

$= \{q_2, q_3, q_4\} = C$

$\Delta(B, a) = \varepsilon\text{-closure}(\text{mossa}(B, a)) = \varepsilon\text{-closure}(\{q_0, q_4\}) = B$

$\Delta(B, b) = \varepsilon\text{-closure}(\text{mossa}(B, b)) = \varepsilon\text{-closure}(\{q_2, q_4\}) = C$

$\Delta(C, a) = \varepsilon\text{-closure}(\text{mossa}(C, a)) = \varepsilon\text{-closure}(\{q_4\}) =$

$= \{q_3, q_4\} = D$

$\Delta(C, b) = \varepsilon\text{-closure}(\text{mossa}(C, b)) = \varepsilon\text{-closure}(\{q_4\}) = D$

$\Delta(D, a) = \varepsilon\text{-closure}(\text{mossa}(D, a)) = \varepsilon\text{-closure}(\{q_4\}) = D$

$\Delta(D, b) = \varepsilon\text{-closure}(\text{mossa}(D, b)) = \varepsilon\text{-closure}(\phi) = \phi = E$

$\Delta(E, a) = \varepsilon\text{-closure}(\text{mossa}(E, a)) = \varepsilon\text{-closure}(\phi) = \phi = E$

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

DFA M_N

Equivalenza NFA - DFA

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

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

Corollario La classe dei linguaggi riconosciuti dagli NFA coincide con la classe dei ling. riconosciuti dai DFA

Dimostrazione del Teorema:

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

(1) M_N è deterministico! Infatti $\Delta(A, a)$ è definita per ogni coppia (A, a) con $A \in T$ e $a \in \Sigma$ in modo univoco, e il risultato di $\Delta(A, a)$ è un elemento di T.

(2) Quindi dobbiamo "solo" dimostrare che $L[N] = L[M_N]$

Oss: Per un DFA, $\varepsilon\text{-closure}(R) = R$ con $R \in T$, perché non ci sono mosse ε

Notazione Chiamiamo $i_M = \varepsilon\text{-closure}(q_0)$ lo stato iniziale di M_N

Vogliamo dimostrare che $\forall w \in \Sigma^*$

$$\hat{\delta}(q_0, w) = \hat{\Delta}(i_M, w)$$

(Per def. $\hat{\delta}$ vedi pg. 24)

per induzione sulla lunghezza di w.

Caso base : $|w| = 0$ cioè $w = \varepsilon$

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

$$\hat{\Delta}(i_M, \varepsilon) = \varepsilon\text{-closure}(i_M) = i_M = \varepsilon\text{-closure}(q_0)$$

ok

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 $\hat{\delta}$,

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

Similmente

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

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

$$\Delta(\{P_1, \dots P_k\}, a) = \varepsilon\text{-closure}(\text{mossa}(\{P_1, \dots P_k\}, a)) =$$

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

$$= \hat{\delta}(q_0, xa)$$

ok

Infine, abbiamo che

$w \in L[N]$ sse $\exists p \in \hat{\delta}(q_0, w)$ con $p \in F$

sse $\exists p \in \hat{\Delta}(i_M, w)$ con $p \in F$

sse $\hat{\Delta}(i_M, w) \in \mathcal{F}$

sse $w \in L[M_N]$ $\quad \forall w \in \Sigma^*$

Quindi $L[N] = L[M_N]$ <u>c.v.d</u>.

Da Espressioni Regolari a NFA equivalenti

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 che gli NFA riconoscono "solo" linguaggi regolari)

Dimostrazione per induzione sulla sintassi (astratta) della espressione regolare s

Costruiremo $N[s]$, cioè 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

Esaminiamo i vari casi:

- $s = \phi$ $\quad N[s] = \rightarrow 0 \quad \textcircled{0}$ (due stati non connessi)

Osserva che $\mathcal{L}[\phi] = \phi = L[N[s]]$

- $s = \varepsilon$ $\quad N[s] = \rightarrow 0 \xrightarrow{\varepsilon} \textcircled{0}$

Osserva che $\mathcal{L}[\varepsilon] = \{\varepsilon\} = L[N[s]]$

- $s = a$

$N[s] = \rightarrow 0 \xrightarrow{a} (0)$

Osserva che $\mathcal{L}[a] = \{a\} = L[N[s]]$

- $s = r|t$

$N[s] =$

Osserva che

per ipotesi induttiva

$$\mathcal{L}[r|t] = \mathcal{L}[r] \cup \mathcal{L}[t] = L[N[r]] \cup L[N[t]] = L[N[r|t]]$$

- $s = r \cdot t$

$N[s] =$

Abbiamo fuso insieme il finale di $N[r]$ con l'iniziale di $N[t]$

Osserva che

per ipotesi induttiva

$$\mathcal{L}[r \cdot t] = \mathcal{L}[r] \cdot \mathcal{L}[t] = L[N[r]] \cdot L[N[t]] = L[N[r \cdot t]]$$

- $s = r^*$

$N[s] =$

Osserva che

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

per ipotesi induttiva

c.v.d

Esempio

$S = (a|b)^* \, b\, a$

albero sintattico

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

$N[a]$

$N[b]$

$N[a|b]$

$N[(a|b)^*]$

$N[ba]$

$N[S]$

Altro Esempio

$s = a\,b^*a$

$N[s]$

Costruiamo il DFA associato a $N[s]$

$A = \varepsilon\text{-closure}(0) = \{0\}$

$\Delta(A, a) = \varepsilon\text{-closure}(\{1\}) = \{1, 2, 4\} = B$

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

$\Delta(B, a) = \varepsilon\text{-closure}(\{5\}) = \{5\} = D$

$\Delta(B, b) = \varepsilon\text{-closure}(\{3\}) = \{3, 4, 2\} = E$

$\Delta(D, a) = \emptyset = C$

$\Delta(D, b) = \emptyset = C$

$\Delta(E, a) = \varepsilon\text{-closure}(\{5\}) = D$

$\Delta(E, b) = \varepsilon\text{-closure}(\{3\}) = E$

$\Delta(C, a) = \emptyset = C$

$\Delta(C, b) = \emptyset = C$

DFA associato a $N[a\,b^*a]$

Grammatiche Regolari

Def: Una grammatica libera è regolare sse ogni produzione è della forma $V \to aW$ oppure $V \to a$ dove $V, W \in NT$ e $a \in T$. Per il simbolo iniziale S è ammessa anche la produzione $S \to \varepsilon$.

(N.B. A volte useremo una definizione più lasca che permette produzioni $V \to \varepsilon$ anche per nonterminali diversi da S.)

Esempio (1)

$$\left.\begin{array}{l} A \to aA \mid bB \mid bA \\ B \to a \end{array}\right]$$

G è regolare

$L(G) = (a|b)^*ba$!!

Proviamo ad associare un NFA a G:

(solito automa NFA per $(a|b)^*ba$)

- Abbiamo associato ad ogni nonterminale uno stato dell'NFA
- abbiamo aggiunto uno stato finale "ε"

È possibile, in generale, associare ad ogni grammatica regolare un NFA equivalente? Sì

Esempio (2)

$$\left.\begin{array}{l} A \to aA \mid bB \\ B \to bB \mid aC \mid a \\ C \to aA \mid bB \end{array}\right] \quad G \text{ è regolare}$$

Non è immediato riconoscere che $L[N] = (a|b)^* ba$

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

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

- $Q = NT \cup \{\varepsilon\}$ ($\cdot\ F = \{\varepsilon\}$ $\cdot\ q_0 = S$)
- δ è definita come:
  - $Z \in \delta(V, a)$ se $V \to aZ \in R$
  - $\varepsilon \in \delta(V, a)$ se $V \to a \in R$
  - $\varepsilon \in \delta(S, \varepsilon)$ se $S \to \varepsilon \in R$

Si può dimostrare che

$$S \Rightarrow_G^* w \quad \text{(con la grammatica } G\text{)}$$

sse

$$(S, w) \vdash_{N_G}^* (\varepsilon, \varepsilon) \quad \text{(con l'automa } N_G\text{)}$$

(non lo dimostreremo)

$\boxed{}$

Da DFA a grammatiche regolari

Teorema

Da un DFA M, possiamo definire una grammatica regolare 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 nonterminali 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 $q_i \to a\, q_j \in R$

    inoltre se $q_j \in F$, anche $q_i \to a \in R$
  - se $q_0 \in F$, allora $q_0 \to \varepsilon \in R$

(Versione alternativa che usa la def. di grammatica regolare più lasca:

- per ogni $\delta(q_i, a) = q_j$ la produzione $q_i \to a\, q_j \in R$
- se $q \in F$, allora $q \to \varepsilon \in R$

Si può dimostrare che

$$w \in L[M] \text{ sse } w \in L(G_M)$$

Esempi

M

aa^* è l'espressione regolare che descrive $L[M]$

Secondo la costruzione appena descritta

$$G_M \begin{cases} A \to aB \mid a \\ B \to aB \mid a \end{cases}$$

Secondo la variante, invece

$$G_M \begin{cases} A \to aB \\ B \to aB \mid \varepsilon \end{cases}$$

a rigore questa non è regolare perché ammette $B \to \varepsilon$

M

$(a|b)^*ba$

$$G_M \begin{cases} A \to aB \mid bC \\ B \to aB \mid bC \\ C \to bC \mid aD \mid a \\ D \to aB \mid bC \end{cases}$$

Alternativamente

$$G_M \begin{cases} A \to aB \mid bC \\ B \to aB \mid bC \\ C \to bC \mid aD \\ D \to aB \mid bC \mid \varepsilon \end{cases}$$

Domanda: È possibile costruire una G_M regolare partendo da un NFA (anziché da un DFA)?

Suggerimento: pensate alle transizioni ε.

Grammatiche Regolari ed Espr. Regolari

Teorema Il linguaggio definito da una grammatica regolare G è un linguaggio regolare, cioè è possibile costruire una espressione regolare s_G tale che

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

Sketch della dimostrazione

Idea della prova:

Caso semplice: un solo nonterminale

$$A \to aA \mid b \mid \varepsilon$$

È intuitivo vedere che $a^*(b \mid \varepsilon)$ è la espressione regolare associata.

Caso medio: due nonterminali

$$\left.\begin{array}{l} A \to aA \mid bB \mid c \\ B \to cA \mid aB \mid d \end{array}\right\}$$ lo si può vedere come un sistema da risolvere

Ricaviamo B dalla seconda "equazione"

$$B \approx a^*(cA \mid d)$$

Ora sostituiamo B nella prima "equazione" dove A compare nella espr. regolare

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

Con opportune manipolazioni su espr. regolari, usando leggi che abbiamo visto, possiamo scrivere

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

e quindi

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

Ora siamo nella forma "<u>semplice</u>" e sappiamo come fare

A ha associata la espr. regolare

$$(a \mid ba^*c)^* (ba^*d \mid c)$$

<u>In generale</u>

$$A_1 \approx a_{11} A_1 \mid \dots \mid a_{1n} A_n \mid b_{11} \mid \dots \mid b_{1p_1}$$

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

$$\vdots$$

$$A_n \approx a_{n1} A_1 \mid \dots \mid a_{nn} A_n \mid b_{n1} \mid \dots \mid b_{np_n}$$

Si parte con

$$A_n \approx S_n [A_1, \dots, A_{n-1}]$$

cioè si costruisce una espr. regolare per A_n che usa $A_1, \dots, A_{n-1}$

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

$$A_{n-1} \approx S_{n-1} [A_1, \dots, A_{n-2}]$$

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

$\boxed{///}$

Esempi

(1)

$$\left.\begin{array}{l} A \to aB \mid \varepsilon \\ B \to bA \mid \varepsilon \end{array}\right] G$$

- Per B la espr. regolare associata è $(bA \mid \varepsilon)$
- Sostituisco questa al posto di B nella prima "equazione"

$$A \simeq a(bA \mid \varepsilon) \mid \varepsilon$$

- manipolo la espr. regolare

$$A \simeq abA \mid a \mid \varepsilon$$

- siamo ora nel caso semplice

$$A \simeq (ab)^*(a \mid \varepsilon)$$

S_G

---

(2)

$$\left.\begin{array}{l} A \to aB \mid bA \mid d \\ B \to aB \mid c \end{array}\right] G$$

- Per B ottengo subito l'espr. regolare a^*c
- Sostituisco nella prima equazione

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

- ora siamo nel caso semplice

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

S_G

OSS: La grammatica regolare con unica produzione $A \to aA \mid G$ definisce il linguaggio vuoto, non a^*

Quindi $S_G = \emptyset$.

NON PUÒ TERMINARE

Riassumendo

Conseguenza: Tutti questi formalismi sono equivalenti.

Tutti generano / riconoscono / descrivono

gram. reg. — NFA/DFA — espr. reg.

la stessa classe di linguaggi, ovvero i linguaggi REGOLARI

Per costruire uno scanner (analizzatore lessicale) (42bis), si parte dalla specifica dei pattern associati alle categorie sintattiche del linguaggio, mediante espressioni regolari

Expr. Reg (pattern) $\longrightarrow$ NFA $\longrightarrow$ DFA

ma se l'NFA ha n stati, il DFA potrebbe avere fino a 2^n stati!

$\Rightarrow$ serve trovare un DFA equivalente più piccolo possibile

min-DFA (minimizzato)

- per occupare meno memoria (minor numero possibile di stati)
- tale DFA minimo è unico (a meno di isomorfismo)
- ed è equivalente al DFA da cui siamo partiti

Minimizzazione

$$L[N] = L[N, q_0] = (a|b)\, a^* b\, (a|b)^*$$

$$L[N, q_1] = a^* b\, (a|b)^* = L[N, q_2]$$

$\Rightarrow$ q_1 e q_2 sono due stati equivalenti (o indistinguibili)

Se fondiamo insieme q_1 e q_2, otteniamo un DFA più piccolo

Questo DFA è minimo perché non ci sono due stati tra loro equivalenti. Infatti

q_3 è finale e quindi $L[N, q_3] \ni \varepsilon$

q_1, q_2 è non finale e quindi $\varepsilon \notin L[N, q_1 q_2]$
ma $b \in L[N, q_1 q_2]$

q_0 è non finale ma $b \notin L[N, q_0]$

Prima delle definizioni formali, ci serve introdurre una notazione:

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

$\hat{\delta} : Q \times \Sigma^* \to Q$ è definita come

$$\hat{\delta}(q, \varepsilon) = q$$
$$\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$$

e quindi

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

Equivalenza / Indistinguibilità

Def Due stati q_1 e q_2 di un DFA N sono equivalenti (o indistinguibili) se $\forall x \in \Sigma^*$

$$\hat{\delta}(q_1, x) \in F \text{ sse } \hat{\delta}(q_2, x) \in F$$

cioè se $L[N, q_1] = L[N, q_2]$

Simmetricamente, due stati q_1 e q_2 non sono equivalenti se $\exists x \in \Sigma^*$ tale che

$$\hat{\delta}(q_1, x) \in F \text{ ma } \hat{\delta}(q_2, x) \notin F$$

o

$$\hat{\delta}(q_1, x) \notin F \text{ ma } \hat{\delta}(q_2, x) \in F$$

q_1 e q_2 sono "distinguibili"

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

Cerco di vedere quali coppie di stati NON sono equivalenti, a cominciare dalla stringa ε

1) ε distingue ogni stato in F (FINALE) da ogni stato in Q\F (NON FINALE)

~~(A, D)~~ ~~(B, D)~~ ~~(C, D)~~

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

2.1) • "a" distingue B e C perché

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

ma (B, D) l'avevo già cancellata al passo precedente ($B \in Q \setminus F$, mentre $D \in F$)

• "a" distingue A e C perché

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

ma $B \in Q \setminus F$ mentre $D \in F$

(2.2) "b" non mi permette di fare ulteriori distinzioni

3) Procedo ora con le stringhe lunghe 2, ma non riesco a fare nessuna ulteriore distinzione $\Rightarrow$ FINE A e B sono equivalenti

$\Rightarrow$

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

- $\sim_0 = \underbrace{F \times F \cup (Q \setminus F) \times (Q \setminus F)}$

  stati che non possono essere distinti da ε, l'unica parola di lunghezza 0.

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

  che significa: q_1 e q_2 sono in relazione $\sim_{i+1}$ se $\forall x \in \Sigma^*$ con $|x| \leq i+1$

  $$\hat{\delta}(q_1, x) \in F \text{ sse } \hat{\delta}(q_2, x) \in F$$

<u>Oss:</u> (1) La relazione $Id = \{(q,q) \mid q \in Q\}$ è tale che $Id \subseteq \sim_i$ per ogni i

"Uno stato è sempre equivalente a se stesso"

(2) $\sim_i$ è una relazione d'equivalenza per ogni i.

- $\sim_0$ ha 2 sole classi d'equivalenza, ovvero F e $Q \setminus F$
- per $\sim_i$, riflessività e simmetria sono ovvie, mentre la transitività è meno banale:

  $$q_1 \sim_i q_2 \text{ e } q_2 \sim_i q_3 \Rightarrow q_1 \sim_i q_3$$

(3) $\sim_{i+1} \subseteq \sim_i$ Ad ogni passo, rimuovo qualche coppia!

$$\sim_0 \supseteq \sim_1 \supseteq \sim_2 \supseteq \sim_3 \dots$$

catena decrescente di relazioni di equivalenza
(non crescente)

(4) Se esiste K tale che $\sim_K = \sim_{K+1}$, allora $\forall J > K$ vale che $\sim_J = \sim_K$ (*vedi prossima pag.)

"cioè, non appena la relazione non viene modificata in un passo, ho trovato "la soluzione"")

(5) Un tale K esiste sicuramente ed è minore di $|\sim_0| = |F \times F| + |(Q \setminus F) \times (Q \setminus F)| = |F|^2 + |Q \setminus F|^2$

perché, nella peggiore delle ipotesi, ad ogni passo iterativo rimuovo solo una (meglio 2) coppia.

(In realtà si può dimostrare che $K \leq |Q| - 1$, perché $|Q|-1$ è la lunghezza del massimo cammino aciclico: infatti, negli esempi che abbiamo visto, ho considerato solo stringhe "corte" che non portavano a cicli)

Vogliamo dimostrare, con un esempio, che se $\sim_2 = \sim_1$, allora anche $\sim_3 = \sim_2$.

Supponiamo, per assurdo, che $\sim_2 = \sim_1$ ma che $\sim_3 \neq \sim_2$ cioè $\exists A, B$ tali che $A \sim_2 B$ ma $A \not\sim_3 B$, cioè

$$\sim_3 \subset \sim_2 = \sim_1$$

Allora, poiché $A \not\sim_3 B$, deve essere

$$A \xrightarrow{a_1} A_1 \xrightarrow{a_2} A_2 \xrightarrow{a_3} A_3 \qquad A_3 \in F$$
$$B \xrightarrow{a_1} B_1 \xrightarrow{a_2} B_2 \xrightarrow{a_3} B_3 \qquad B_3 \notin F$$

(o viceversa)

Ma allora $A_1 \not\sim_2 B_1$ e poiché $\sim_2 = \sim_1$, deve essere $A_1 \not\sim_1 B_1$. Ma allora

$$A_1 \xrightarrow{b} A_4 \qquad A_4 \in F, B_4 \notin F \text{ (o viceversa)}$$
$$B_1 \xrightarrow{b} B_4$$

Ma allora $A \not\sim_2 B$ perché

$$A \xrightarrow{a_1} A_1 \xrightarrow{b} A_4$$
$$B \xrightarrow{a_1} B_1 \xrightarrow{b} B_4$$

contraddicendo l'ipotesi iniziale.

$\Rightarrow$ non è possibile che $\sim_1 = \sim_2$ e $\sim_3 \subset \sim_2$!

$$\boxed{\text{Cioè se } \sim_{K+1} = \sim_K, \text{ allora } \forall J > K \ \sim_J = \sim_K}$$

$\sim_0 = \{ (A,A), (A,B), (A,F),$
NF/NF $(B,B), (B,A), (B,F),$
$(F,F), (F,A), (F,B) \}$

$\cup \{ (C,C), (C,D), (C,E),$
F/F $(D,D), (D,C), (D,E),$
$(E,E), (E,C), (E,D) \}$

$\sim_1 = \{ (A,A), (A,B),$
$(B,B), (B,A),$
$(F,F),$
$(C,C), (C,D), (C,E),$
$(D,D), (D,C), (D,E),$
$(E,E), (E,C), (E,D) \}$

$\sim_2 = \sim_1$ OK - FINITO!

Quali sono gli stati equivalenti? Classi d'eq.?

- A e B
- F
- C, D, E

Automa minimo risultante dalla fusione degli stati equivalenti

$L = 0^* 1 0^*$

$\sim_0 = \{ \{q_0, q_1, \dots q_{n-1}\}, \{q_n\} \}$

$\sim_1 = \{ \{q_0, q_1, \dots, q_{n-2}\}, \{q_{n-1}\}, \{q_n\} \}$

$\vdots$

$\sim_{n-1} = \{ \{q_0\}, \{q_1\}, \{q_2\}, \dots, \{q_{n-1}\}, \{q_n\} \}$

= Identità (ogni stato è equivalente solo a se stesso!)

<u>Esempio di caso pessimo</u>: la relazione finale $\sim$ è ottenuta dopo un numero di iterazioni pari al numero n, con $|Q| = n+1$

<u>(da 0 a $n-1$)</u>

Possiamo ricavare un algoritmo pratico da questa idea?

- Tabella con solo coppie "vere"
- al round 0 dell'algoritmo iterativo, metto una marca X_0 per segnalare che la coppia è distinta (finale, nonfinale) (nonfinale, finale)
- al round 1, metto la marca X_1 per distinguere le coppie (q_1, q_2) non ancora marcate che per qualche $a \in \Sigma$ ha $(\delta(q_1, a), \delta(q_2, a))$ già marcata
- al round 2, metto la marca X_2 per ---

se non riesco a mettere nessuna nuova marca in un round $\Rightarrow$ STOP!

B					
C	X_0	X_0			
D	X_0	X_0			
E	X_0	X_0			
F	X_1	X_1	X_0	X_0	X_0
	A	B	C	D	E

Le entrate della tabella che rimangono non marcate mi dicono quali stati sono equivalenti

	A	B	C	D	E	F	G
B	X_1						
C	X_0	X_0					
D	X_1	X_1	X_0				
E		X_1	X_0	X_1			
F	X_1	X_1	X_0		X_1		
G	X_2	X_1	X_0	X_1	X_2	X_1	
H	X_1		X_0	X_1	X_1	X_1	X_1

← dall'ultimo al 2° →

← dal primo al penultimo →

- X_0 finale/non finale
- X_1 $\delta(A,1) = F$
  $\delta(B,1) = C$
  $(C,F)\ X_0$
  ⋮
- X_2 $\delta(A,1) = F$
  $\delta(G,1) = E$
  $(E,F)\ X_1$
  $\delta(E,1) = F$
  $\delta(G,1) = E$
  $(E,F)\ X_1$

$A \sim E$ $B \sim H$ $D \sim F$

Automa Minimo Risultante

Algoritmo Iterativo (Tabella a Scala)

0) Costruire la tabella a scala

1) Marca X_0 ogni coppia (q_1, q_2) tale che $q_1 \in F$ e $q_2 \in Q \setminus F$ (o viceversa)

2) $b := true$; $i := 1$;

3) <u>while</u> b <u>do</u> {

   3.1) – $b := false$;

   3.2) – per ogni coppia (q_1, q_2) non marcata do { <u>if</u> $\exists a \in \Sigma$ con $(\delta(q_1, a), \delta(q_2, a))$ già marcata <u>then</u> { – marca (q_1, q_2) con X_i; – $b := true$ }

   }

   3.3) – $i := i + 1$

   }

4) Al termine sia J l'insieme delle coppie non marcate

5) La relazione di equivalenza $\sim$ è la chiusura riflessiva e simmetrica di J, cioè

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

<u>N.B.</u> al round i, la parte di tabella non marcata definisce $\sim_i$ (una volta chiusa riflessivamente e simmetricamente)

Teorema Dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$, l'algoritmo di riempimento della Tabella a scala termina. Due stati p e q sono distinguibili (ovvero non equivalenti) sse la casella (p,q) (o (q,p)) è marcata (e quindi sono equivalenti sse la casella non è marcata.)

Dim: • Termina sempre: poiché $\exists k.\ \sim_k = \sim_{k+1}$ e quindi l'algoritmo iterativo termina entro k round.

• $\Rightarrow$) Se p e q sono distinguibili, allora $\exists x \in \Sigma^*$. $\hat{\delta}(p,x) \in F$ e $\hat{\delta}(q,x) \notin F$ (o viceversa). Se prendo $k = |x|$, allora di sicuro $(p,q) \notin \sim_k$, cioè (p,q) viene marcata entro il round k

• $\Leftarrow$) Se (p,q) sono marcati, allora sicuramente (p,q) sono distinguibili:

basta prendere la "catena" di coppie/simboli che portano ad una coppia non presente in $\sim_0$. Ad esempio

$$\underset{x_2}{(p,q)} \xrightarrow{a} \underset{x_1}{(p',q')} \xrightarrow{b} \underset{x_0}{(p'',q'')}$$

$\Rightarrow$ ab è la stringa che distingue p e q.

$\boxed{\text{IIII}}$

Esempio

1) $a^* \varepsilon b^*$ Costruire l'NFA associato

2) Ricavare il DFA equivalente

3) Minimizzare il DFA

B			
C	$\times_1$	$\times_1$	
D	$\times_0$	$\times_0$	$\times_0$
	A	B	C

4) Ricavare la grammatica regolare dal DFA minimo

perché lo stato è sia iniziale che finale

$A \to aA \mid a \mid bC \mid b \mid \varepsilon$

$C \to bC \mid b \mid aD$

$D \to aD \mid bD$

Alternativamente

$A \to aA \mid \varepsilon \mid bC$

$C \to bC \mid \varepsilon \mid aD$

$D \to aD \mid bD$

5) Semplificare la grammatica, rimuovendo i simboli inutili

$A \to aA \mid a \mid bC \mid b \mid \varepsilon$
$C \to bC \mid b$

$A \to aA \mid \varepsilon \mid bC$
$C \to bC \mid \varepsilon$

6) Ricavare dalla grammatica semplificata, l'espressione regolare associata

$C \simeq b^*b$, e sostituendo
$A \simeq aA \mid a \mid bb^*b \mid b \mid \varepsilon$
cioè
$A \simeq a^*(a \mid bb^*b \mid b \mid \varepsilon)$

$C \simeq b^*$, e sostituendo
$A \simeq aA \mid \varepsilon \mid bb^*$
cioè
$A \simeq a^*(\varepsilon \mid bb^*)$

Nota che l'espressione regolare finale è diversa da quella di partenza, ma è ad essa equivalente!

Si può dimostrare, ad esempio, che

$$a^* \varepsilon b^* \simeq a^*(\varepsilon \mid bb^*)$$

Questo esempio dimostra che questo diagramma

commuta, ovvero partendo da una espr. regolare S, ritorno su una espr. regolare S' tale che

$$S \simeq S'$$

Altro Esempio

1) $a\,b^*a$ Costruire NFA associato

2) Costruire DFA associato

$A \xrightarrow{a} B$
$B \xrightarrow{a} D$ $\Rightarrow$ (B,D) è x_0
$\Downarrow$
(A,B) è x_1

B	x_1			
C	x_1			
D	x_0	x_0	x_0	
E	x_2	x_1	x_1	x_0
	A	B	C	D

$B \sim C$

3) Minimizzare

4) Gr. regolare

$A \to aB \mid \cancel{bE}$

$B \to bB \mid \cancel{aD} \mid a$

$\cancel{D} \to \cancel{aE} \mid \cancel{bE}$

$\cancel{E} \to \cancel{aE} \mid \cancel{bE}$

5) Semplificare la gram.

$A \to aB$

$B \to bB \mid a$

6) Espressione regolare

$B \simeq b^*a$

$A \simeq a\,b^*a$

(per caso uguale a quella di partenza!)

Esercizio: ricavare la gram. regolare con la regola alternativa, poi semplificarla, poi ricavare l'espr. regolare

Automa Minimo

Dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$, l'automa minimo equivalente $M_{min} = (\Sigma, Q_{min}, \delta_{min}, \underbrace{[q_0]}_{\text{CLASSE EQUIVALENZA DI } q_0}, F_{min})$ 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 classi di equivalenza di stati di M)

- $\delta_{min}([q], a) = [\delta(q, a)]$
- $F_{min} = \{ [q] \mid q \in F \}$

Oss: Non esistono 2 stati distinti in M_{min} che siano tra loro equivalenti: $[q] \neq [q'] \Rightarrow q \not\sim q'$

Notazione (già vista per M, ed estesa a M_{min})

DFA M	DFA M_{min}
$\hat{\delta} : Q \times \Sigma^* \to Q$	$\hat{\delta}_{min} : Q_{min} \times \Sigma^* \to Q_{min}$
$\hat{\delta}(q, \varepsilon) = q$	$\hat{\delta}_{min}([q], \varepsilon) = [q]$
$\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$	$\hat{\delta}_{min}([q], xa) = \delta_{min}(\hat{\delta}_{min}([q], x), a)$
$w \in L[M]$ sse $\hat{\delta}(q_0, w) \in F$	$w \in L[M_{min}]$ sse $\hat{\delta}_{min}([q_0], w) \in F_{min}$

Teorema

NON LO CHIEDE

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 minimo numero di stati tra tutti gli automi deterministici per questo linguaggio.

Dimostrazione:

(1) M_{min} è ben definito, cioè la definizione di δ_{min} non dipende dallo specifico stato scelto per rappresentare la classe di equivalenza.

Infatti se $q \sim q'$, allora

1) $[q] = [q']$ e

2) $\delta_{min}([q], a) = [\delta(q, a)]$
$= [\delta(q', a)]$
$= \delta_{min}([q'], a)$

se non fosse vero, allora $\delta(q,a)$ e $\delta(q',a)$ sarebbero distinguibili, e quindi pure q e q'

(2) Per dimostrare che $L[M] = L[M_{min}]$, dimostriamo che

$$\hat{\delta}(q_0, w) = r \quad \text{sse} \quad \hat{\delta}_{min}([q_0], w) = [r]$$

ovvero che

$$\hat{\delta}_{min}([q_0], w) = [\hat{\delta}(q_0, w)]$$

Si fa la dimostrazione per induzione su $|w|$

<u>Base</u>: $|w|=0$ cioè $w=\varepsilon$

$$\hat{\delta}(q_0,\varepsilon)=q_0 \qquad \hat{\delta}_{min}([q_0],\varepsilon)=[q_0]=[\hat{\delta}(q_0,\varepsilon)]$$

<u>ok</u>

<u>Passo Induttivo</u> $w=xa$

$$\hat{\delta}(q_0,xa)=\delta(\hat{\delta}(q_0,x),a)$$

e

$$\hat{\delta}_{min}([q_0],xa) \underset{\substack{\uparrow \\ \text{per def.} \\ \text{di } \hat{\delta}_{min}}}{=} \delta_{min}(\underbrace{\hat{\delta}_{min}([q_0],x)}_{\text{per ipotesi induttiva}},a)$$

$$= \delta_{min}(\overbrace{[\hat{\delta}(q_0,x)]}^{\shortparallel},a)$$

per def. di δ_{min} $\longrightarrow$ $= [\delta(\hat{\delta}(q_0,x),a)]$

per def. di $\hat{\delta}$ $\longrightarrow$ $= [\hat{\delta}(q_0,xa)]$

ok

<u>Quindi</u>

$w\in L[M]$ sse $\hat{\delta}(q_0,w)=r\in F$

$$\Updownarrow$$

$\hat{\delta}_{min}([q_0],w)=[r]\in F_{min}$ sse $w\in L[M_{min}]$

(3) Rimane da dimostrare che è minimo, ovvero che un qualunque altro automa deterministico N non può avere meno stati

Supponiamo esista un DFA N tale che $L[N] = L[M_{min}]$, ma con un numero di stati inferiore a quello di M_{min}.

<u>Oss 1</u>: gli stati iniziali di N e M_{min} devono essere equivalenti (nell'automa $N \cup M_{min}$) perché $L[N] = L[M_{min}]$

si suppone che N e M_{min} siano disgiunti!

<u>Oss 2</u>: Se p in Q_{min} e q in N sono equivalenti, nell'automa $(N \cup M_{min})$, allora sono equivalenti anche i loro successori $\forall a \in \Sigma$.

<u>Oss 3</u>: - N non ha stati irraggiungibili dal suo iniziale (altrimenti potrei costruire N' con ancora meno stati)

- M_{min} non ha stati irraggiungibili per costruzione

$\Rightarrow$ ogni stato di M_{min} è equivalente ad almeno uno stato di N (per <u>oss</u> (2))

<u>Oss 4</u>: Poiché N ha meno stati di M_{min}, due stati p e p' di M_{min} devono essere equivalenti ad uno stesso stato q di N. Ma la relazione di eq. $\sim$ è transitiva $\Rightarrow$ p e p' devono essere equivalenti! Ma questo è impossibile!

Per costruzione di M_{min}, non ci sono 2 stati diversi in M_{min} equivalenti tra loro!

LEX : Generatore di Scanner

- **Lex** è un software, che gira sotto Unix, originariamente scritto da Mike Lesk & Eric Schmidt nel 1975

- **FLEX** (versione più recente) è un free software scritto da Vern Paxson nel 1987 (va con Bison)

  flex.sourceforge.net

Lex/Flex è un generatore di analizzatori lessicali

- input: un file di tipo .l che contiene un insieme di definizioni regolari ed una serie di azioni corrispondenti

- output: un programma C che realizza l'automa riconoscitore e che associa ad ogni istanza di una definizione la relativa azione

lex.yy.c → [Compilatore C] → a.out (eseguibile)

> cc lex.yy.c -ll

↖ per linkare le librerie di subroutine di Lex

programma sorgente scritto in $\mathcal{L}$ prog → [Analizzatore Lessicale per $\mathcal{L}$] → lista di token

> a.out prog

Come è fatto un file di input lex.l ?

Diviso in 3 parti

(1) Dichiarazioni (definizioni regolari)

Ad Es:

cifra [0-9]
cifre [0-9]+
ide [a-zA-Z]([a-zA-Z]|[0-9])*

%%

(2) Regole {espr. regolare} {azione}

pattern — frammento di codice C

Ad es:

{cifre} {printf("<NUM,%s>", yytext);}
{ide} {printf("<IDE, %s>", yytext);}

variabile che contiene il testo matched!

%%

(3) Funzioni Ausiliarie

Se si usano funzioni complesse nella parte "azione", queste potrebbero essere definite qui

Come funziona il programma lex.yy.c ?

Il programma C ottenuto, e compilato per renderlo eseguibile in a.out, implementa essenzialmente il DFA riconoscitore dell'insieme delle espressioni regolari contenute nelle "Regole".

- scandisce il testo sorgente alla ricerca di una stringa che corrisponda a (cioè sia un lessema per) una delle espr. 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".

Esempio banale

(No dichiarazioni)

```
%% Regole
P1    {a}    { printf("Z"); }
P2    {abb}  { printf("K"); }
P3    {a*b+} { printf("Y"); }
```

pattern — azione corrispondente

- il pattern P_1 è un prefisso di P_2 ⇒ cerca il pattern più lungo
- il pattern P_2 è un caso speciale di P_3 ⇒ riconosci il primo in elenco

Se l'input è b a b b a, lex.yy.c riconosce i seguenti lessemi

$$\underbrace{b}_{P_3}\ \underbrace{a\,b\,b}_{P_2}\ \underbrace{a}_{P_1}$$

che "traduce" in Y K Z

Oss: non spezzo a b b in $\underbrace{a}_{P_1}\ \underbrace{bb}_{P_3}$

Ma come fa a riconoscere questi lessemi (pattern)?

NFA per ciascuno dei 3 pattern

P_1

P_2

P_3

automa NFA per P_3 non canonico

NFA per i 3 pattern insieme

E ora fa una "simulazione" DFA dell'NFA risultante

- stato iniziale = ε-closure(0) = {0,1,3,7}

Stato	Simbolo input a	Simbolo input b	Pattern riconosciuto	Azione corrispondente
0137	247	8	nessuno	—
247	7	58	$P_1 = a$	printf("z")
8	—	8	$P_3 = a^* b^+$	printf("y")
7	7	8	nessuno	—
58	—	68	$P_3 = a^* b^+$	printf("y")
68	—	8	$P_2 = abb$	printf("k")

a.out procede in questo modo:

- va avanti a leggere l'input finché non si blocca
- se lo stato in cui si trova è di riconoscimento ⇒ ok. Altrimenti, torna indietro fino a trovare uno stato di riconoscimento
  (questo garantisce che se più pattern possono essere riconosciuti sullo stesso prefisso, il più lungo viene scelto)
- se arrivato al "match", vediamo che potenzialmente più di un pattern andrebbe bene, allora prendiamo quello elencato prima
  (nell'esempio lo stato 68 è ok sia per P_2 che per P_3, ma prendiamo P_2 perché primo in elenco)

Vediamo il funzionamento sull'input

b a b b a

Inizio in $0137 \xrightarrow{b} 8 \xrightarrow{a}$

Mi sono bloccato in 8 ⇒ b è un lessema del pattern $P_3 = a^* b^+$

Ora l'input residuo è abba

Ricominciamo da 0137

$$0137 \xrightarrow{a} 247 \xrightarrow{b} 58 \xrightarrow{b} 68 \not\xrightarrow{a}$$

Mi sono bloccato in 68 $\Rightarrow$ abb è un lessema del pattern $p_2 = abb$

(• Nota che sono passato attraverso stati di riconoscimento, ma ho proseguito fino al blocco $\Rightarrow$ ho preso il lessema più lungo!

• Nota che 68 è OK sia per p_2 che per p_3, ma ho scelto il primo in elenco!)

Ricominciamo da 0137 col residuo a

$$0137 \xrightarrow{a} 247 \quad \text{FINE}$$

$\Rightarrow$ a è un lessema del pattern $p_1 = a$

Nei file . l di un ling. di progr., le parole riservate vanno messe prima degli identificatori

%

WHILE	Az_1	- se trovo "while", allora riconosco la parola riservata perché è listata prima
IF	Az_2	
⋮		
IDE	Az_{ide}	- se trovo "whiles", $\Rightarrow$ IDE perché prendo il match più lungo!

Un esempio più interessante

```
cifra           [0-9]
cifre           [0-9]+
ide             [a-zA-Z][a-zA-Z0-9]*
separatore      [ \t\n]*
razionale       {cifre}"."{cifre}
bin             [+*]

%%

{cifre}                          {printf("<NUM,%s>", yytext);}
{razionale}                      {printf("<FRACT,%s>", yytext);}
":="|";"|while|"("|")"|"{"|"}"    {printf("<%s>",yytext);}
{ide}                            {printf("<IDE,%s>", yytext);}
{separatore}                     {printf("␣");}
{bin}|"-"                        {printf("<OP2,%s>", yytext);}
```

blank, tab, newline

variabile con testo matched

Figura 3.10 Un semplice analizzatore lessicale in Lex.

Applicando lo scanner prodotto da Lex su un progr.

pigreco := 3.14;
temp1 := pigreco * (temp$_1$ + 2);

si genera la sequenza di token

<IDE, pigreco> <:=> <FRACT, 3.14> <;>
<IDE, temp1> <:=> <IDE, pigreco> <OP2,*>
<(> <IDE, Temp1> <OP2, +> <NUM, 2> <)> <;>

Utilizzo con YACC

- il programma generato da LEX non è usato da solo
- è una subroutine per YACC (generatore di analizzatori sintattici)
- alcune variabili comuni ai due programmi (come ad esempio yylval) permettono di scambiare informazioni
- lex.yy.c è usato da YACC "on demand", richiedendo il token successivo (alla bisogna)
- yylex() restituisce il nome del token, mentre il valore del token è condiviso nella variabile yylval

Esempio d'uso con YACC

```
%{
#define NUM 1
#define FRACT 2
#define IDE 3
#define OP2 4

#define PIU 41
#define PER 42
#define MEN 43
%}

cifra           [0-9]
cifre           [0-9]+
ide             [a-zA-Z][a-zA-Z0-9]*
separatore      [ \t\n]*
razionale       {cifre}"."{cifre}
bin             [+*]
%%
{separatore}        { }
{cifre}             {yylval = inConstTable(); return(NUM);}
{razionale}         {yylval = inConstTable(); return(FRACT);}
{ide}               {yylval = inSymbolTable(); return(IDE);}
+                   {yylval = PIU; return(OP2);}
*                   {yylval = PER; return(OP2);}
-                   {yylval = MEN; return(OP2);}
%%
int inConstTable(){/*Converte la stringa puntata da yytext in
                     uno (NUM) o due (FRACT) interi, li inserisce
                     nella tabella delle costanti e restituisce
                     (come int) un puntatore all'elemento
                     inserito*/
}
int inSymbTable(){/*Inserisce nella tabella dei simboli la
                    stringa puntata da yytext e restituisce (come
                    int) un puntatore all'elemento inserito*/
}
```

dichiarazioni ausiliarie

dichiarazioni vere

variabile condivisa tra Lex e YACC
lex mette in yylval il puntatore nella tabella al lessema appena individuato

REGOLE

DEF. AUSILIARIE

Figura 3.12 Un programma Lex che interagisce con Yacc.

PROPRIETÀ ALGORITMICHE DEI LING. REGOLARI

Finora non abbiamo dimostrato che esistono ling. non regolari.

Sappiamo che una gr. regolare è un caso speciale di gr. libera. Quindi sicuramente

Ora vedremo una proprietà algoritmica dei ling. regolari che sarà utile per dimostrare che certi linguaggi, non possedendo quella proprietà, non sono regolari!

Ad es: $L = \{a^n b^n \mid n \geq 0\}$ è sicuramente libero perché generato da una gr. libera $S \to \varepsilon \mid aSb$ e dimostreremo che non è regolare!

Intuitivamente, perché $\{a^n b^n \mid n \geq 0\}$ non può essere regolare?

Necessità di contare illimitatamente e per farlo mi servirebbero infiniti stati

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

Se devo farlo per ogni $n \geq 0$, mi servono infiniti stati! Un NFA/DFA ha solo un numero finito di stati, per cui può solo "contare" tanto quanto gli stati che ha.

<u>Attenzione</u>

$\{a^n b^m \mid n, m \geq 0\}$ è regolare e corrisponde all'espr. regolare $a^* b^*$

Vediamo ora la proprietà algoritmica che hanno tutti i ling. regolari.

Pumping Lemma

Se L è un ling. regolare, allora $\exists N > 0$ 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.

Dimostrazione

DA FARE :)

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 $q_i = q_j$

- 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 essere più cicli) prendo il primo ciclo!

$$\left.\begin{array}{l} uv^0w = uw \in L \\ uv^1w = uvw = z \in L \\ uv^2w = uvvw \in L \\ \vdots \\ uv^kw \in L \end{array}\right\} \quad \forall k \geq 0 \;\; uv^kw \in L$$

perché il ciclo v può essere percorso un numero arbitrario di volte (anche zero)

$\boxed{///}$

Oss: Se L è finito, allora non esiste nessuna $z \in L$ con $|z| \geq N$ e quindi "l'implicazione" è vera perché è falsa la premessa

Oss: Se $\exists z \in L$ con $|z| \geq N$, allora M riconosce un ling. infinito

---

Come usare il pumping lemma per dimostrare che un linguaggio non è regolare?

Se L è regolare $\Rightarrow$ P (dice il pumping lemma)

Se $\sim P \Rightarrow$ L non è regolare

$\uparrow$

questa è l'implicazione che useremo!

Negazione del Pumping Lemma

Se $[\forall N > 0 \ \exists z \in L$ con $|z| \geq N$

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

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

allora L non è regolare

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

$L_1 = \{ a^n \mid n$ è un quadrato perfetto$\}$

$= \{ a^{n^2} \mid n \geq 0 \}$ è regolare? <u>No</u>

- Fissiamo $N > 0$ generico $(\forall N > 0)$
- Scegliamo $z = a^{N^2}$ $(\exists z \in L$ con $|z| \geq N)$
- Per ogni possibile scomposizione di $z = uvw$ tale che $(\forall uvw)$

  (i) $|uv| \leq N$

  (ii) $|v| \geq 1$

  deve essere che $1 \leq |v| \leq N$

- Prendiamo $K = 2$ cioè consideriamo uv^2w

  Vale che

$$|z| = N^2 < |uv^2w| = |z| + |v| \leq N^2 + N < N^2 + 2N + 1 = (N+1)^2$$

cioè uv^2w ha una lunghezza strettamente compresa tra N^2 e $(N+1)^2$

$$\Rightarrow uv^2w \notin L_1$$

$\Rightarrow L_1$ non è regolare

$L_2 = \{ a^n \mid n$ è una potenza di $2 \}$

$= \{ a^{2^n} \mid n \geq 0 \}$ è regolare? NO

- Fissiamo $N > 0$ $\quad (\forall N > 0)$
- Scegliamo $z = a^{2^N}$ $\quad (\exists z \in L$ con $|z| \geq N)$
- Per ogni u, v, w tali che $z = uvw$ e

(i) $|uv| \leq N$

(ii) $|v| \geq 1$

deve essere $1 \leq |v| \leq N$

- Prendiamo $k = 2$, cioè consideriamo uv^2w

$$|z| = 2^N < |uv^2w| = |z| + |v| \leq 2^N + N < 2^N + 2^N = 2^{N+1}$$

cioè uv^2w ha una lunghezza strettamente compresa tra 2^N e 2^{N+1}

$\Rightarrow uv^2w \notin L_2$

$\Rightarrow L_2$ non è regolare

$L_3 = \{ ww \mid w \in \{a,b\}^* \}$ è regolare? NO

- Fissiamo $N > 0$ $\quad (\forall N > 0)$
- Scegliamo $z = a^N b^N a^N b^N$ $\quad (\exists z \in L$ con $|z| \geq N)$
- Per ogni u, v, w tali che $z = uvw$ e $|uv| \leq N$ e $|v| \geq 1$ deve essere che v è fatta solo di "a" con $1 \leq |v| \leq N$
  Sia $v = a^J$ con $J \geq 1$
- Prendiamo $K = 2$.

$$uv^2w = a^{N+J} b^N a^N b^N \notin L_3$$

$\Rightarrow L_3$ non è regolare

$L_4 = \{ ww^R \mid w \in \{a,b\}^* \}$

- Fissiamo $N > 0$ $\quad (\forall N > 0)$
- Scegliamo $z = a^N b^N b^N a^N$ $\quad (\exists z \in L$ con $|z| \geq N)$
- Per ogni u, v, w tali che $z = uvw$, $|uv| \leq N$ e $|v| \geq 1$ deve essere che v è fatta solo di "a" $\Rightarrow v = a^J$ con $J \geq 1$
- Prendiamo $K = 2$

$$uv^2w = a^{N+J} b^N b^N a^N \notin L_4$$

$\Rightarrow L_4$ non è regolare

$L_5 = \{a^n \mid n$ è un numero primo$\}$

- Fissiamo $N > 0$ $\quad (\forall N > 0)$
- Scegliamo $z = a^p$ con p primo e $p \geq N+2$
- Per ogni u, v, w tali che $z = uvw$ e

$$(i)\ |uv| \leq N$$

$$(ii)\ |v| \geq 1$$

deve essere $1 \leq |v| \leq N$. Sia $m = |v|$.

Allora $|uv^0w| = |uw| = p - m$ (che deve essere un primo se $uv^0w \in L_5$)

- Ora consideriamo $K = p - m$

$$|uv^{p-m}w| = |uw| + (p-m) \cdot |v| =$$

$$= (p-m) + (p-m) \cdot m$$

$$= (p-m) \cdot (1+m)$$ che non è un numero primo!!

$$\Rightarrow uv^{p-m}w \notin L_5$$

$\Rightarrow$ L_5 non è regolare

$(p-m) \cdot (1+m)$ non è primo se $(p-m) \neq 1$ e $(1+m) \neq 1$. Ma
- $m \geq 1 \Rightarrow (1+m) > 1$
- $p \geq N+2$ e $m \leq N$ da cui $(p-m) \geq 2$

Altre Proprietà dei Ling. Regolari

La classe dei ling. regolari è chiusa per

(1) unione

(2) concatenazione

(3) stella di Kleene

(4) complementazione

(5) intersezione

Dim:

(1), (2) e (3) sono ovvie. Ad esempio, se L_1 e L_2 sono regolari, allora esistono s_1 e s_2 espr. 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 vera 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 $w \in L[M]$ sse $w \notin L[\bar{M}]$ e quindi

$$L[\bar{M}] = \Sigma^* \setminus L[M]$$

(5) discende dalla legge di De Morgan

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

M

$L[M] = a^+$

$\bar{M}$

$L[\bar{M}] = \{\varepsilon\}$

La chiusura per intersezione può essere utile per dimostrare che un ling. non è regolare.

Infatti, se $L \cap L_{reg} = L_{non\text{-}reg} \Rightarrow L$ non è reg.

$L = \{w \in \{a,b\}^* \mid$ in w occorrono tante "a" quante "b" $\}$

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

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

$\Rightarrow L$ non è regolare

Modo alternativo per dimostrare la chiusura per $\cap$

Dati $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ e $M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$,

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

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 acicliclico 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? "$\exists$ una parola $z \in L[M]$ tale che $n \leq z \leq 2n$ dove $n = |Q|$?"

Capitolo 4

Analisi sintattica: Linguaggi Liberi

- classe di linguaggi più generale di quella dei ling. regolari

- grammatiche libere da contesto più generali di quelle regolari

Regolari		Libere
$V \to aW$	$V, W \in NT$	$V \to \alpha \quad \alpha \in (T \cup NT)^*$
$V \to a$	$a \in T$	
$S \to \varepsilon$		

- automi a pila (pushdown automata - PDA) invece di automi finiti (NFA - DFA)

PDA
- nondeterministici $\approx$ Ling. liberi
- deterministici : utili per costruire compilatori
  - top-down - grammatiche LL(k)
  - bottom-up - grammatiche LR(k)

Analisi Sintattica

lista di token (prodotta dall'analizzatore lessicale)

↓

prodotto a partire da una grammatica libera ⇝

Analizzatore Sintattico
PARSER

↓

albero di derivazione
(o anche albero sintattico)
relativo alla grammatica usata per costruire il parser

Ripasso di definizioni già date qualche lezione fa

Una grammatica libera $G = (NT, T, R, S)$ è tale che

- NT è un insieme finito di simboli nonterminali
- T è un insieme finito di simboli terminali
- $S \in NT$ è il simbolo iniziale
- R è un insieme finito di produzioni (o regole) del tipo

$$V \rightarrow \alpha \quad \text{con } \alpha \in (T \cup NT)^* \text{ e } V \in NT$$

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

N.B. Se G è libera, allora $L(G)$ è libero!

La relazione $\Rightarrow$ di derivazione in un passo

$$\frac{V = xAy \qquad (A \to z) \in R \qquad w = xzy}{V \Rightarrow w} \qquad x, y \in (T \cup NT)^*$$

La relazione $\Rightarrow^*$ di derivazione in 0 o più passi

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

(chiusura riflessiva e transitiva di $\Rightarrow$)

Def: Derivazione canonica sinistra (leftmost): derivazione in cui, a partire da S, viene sempre riscritto il nonterminale più a SX

Def: Derivazione canonica destra (rightmost): derivazione in cui, a partire da S, viene sempre riscritto il nonterminale più a dx

OSS: Esiste una corrispondenza biunivoca tra derivazioni leftmost (/ rightmost) e alberi di derivazione

Come bisogna potenziare gli NFA/DFA per far sì che riconoscano ling. liberi?

Esempio: $L = \{ ww^R \mid w \in \{a,b\}^* \}$

- L è libero perché $L = L(G)$ con $G \left[S \to \varepsilon \mid aSa \mid bSb \right]$ libera.
- L non è regolare (esempio discusso in precedenza)
- $L = \{ \varepsilon, aa, bb, aaaa, abba, baab, bbbb, \dots \}$

- L'automa deve "ricordare" la prima metà dell'input (cioè w) in modo da confrontarla con la seconda metà (cioè w^R) in ordine inverso

$\Downarrow$

necessità di accumulare (parte dell') l'input!

$\Downarrow$

fornire all'automa una memoria ausiliaria: una pila che può crescere illimitatamente

- Quando si accorge che w è finita?
  In genere, non c'è nessun criterio: nondeterministicamente l'automa assume di essere arrivato al centro/metà ed inizia da lì in poi a confrontare all'indietro!

- la pila può crescere senza limiti (memoria ausiliaria illimitata, perché w può essere arbitrariamente lunga)

- Ma può leggere solo l'elemento top della pila
- Può rimuovere solo l'elemento top della pila
- Può inserire un nuovo elemento solo in testa

Def: Un automa a pila nondeterministico (PDA) è una 7-pla $(\Sigma, Q, \Gamma, \delta, q_0, \perp, F)$ dove

- Σ è un alfabeto finito (simboli in input)
- Q è un insieme finito di stati
- Γ è un insieme finito di simboli della pila
- δ è la funzione di transizione con tipo

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow \mathcal{P}_{fin}(Q \times \Gamma^*)$$

- q_0 è lo stato iniziale
- $\perp \in \Gamma$ è il simbolo iniziale sulla pila
- $F \subseteq Q$ è l'insieme degli stati finali

$$\delta : Q \times \underbrace{(\Sigma \cup \{\varepsilon\})}_{} \times \Gamma \longrightarrow \mathcal{P}_{fin}(Q \times \Gamma^*)$$

consuma un simbolo dell'input (Σ) oppure no (ε)

consuma il simbolo top della pila

sulla pila scrive una stringa di lunghezza qualsiasi (anche ε) di simboli in Γ (detto "gamma")

- è <u>nondeterministico</u> perché

(1) $|\delta(q, \sigma, A)|$ può essere > 1

(per $q \in Q$, $\sigma \in \Sigma \cup \{\varepsilon\}$, $A \in \Gamma$)

(2) ma se anche richiedessimo che

$|\delta(q, \sigma, A)| \leq 1 \quad \forall q \in Q, \forall \sigma \in \Sigma \cup \{\varepsilon\}, \forall A \in \Gamma$

potrebbe esistere $q' \in Q$, $b \in \Sigma$, $B \in \Gamma$ tali che

$|\delta(q', \varepsilon, B)| = 1$ e $|\delta(q', b, B)| = 1$

$\Rightarrow$ nondeterminismo tra mossa "ε" e mossa "b"

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) $$\frac{(q', \alpha) \in \delta(q, a, X) \quad a \in \Sigma}{(q, aw, X\beta) \vdash_N (q', w, \alpha\beta)}$$

(2) $$\frac{(q', \alpha) \in \delta(q, \varepsilon, X)}{(q, w, X\beta) \vdash_N (q', w, \alpha\beta)}$$

- **computazione / cammino**

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

(chiusura riflessiva e transitiva di $\vdash_N$)

Linguaggio Accettato

Due modalità di riconoscimento:

(1) Per stato finale

$$L[N] = \{ w \in \Sigma^* \mid (q_0, w, \perp) \vdash_N^* (q, \varepsilon, \alpha) \text{ con } q \in F \}$$

(2) Per pila vuota

$$P[N] = \{ w \in \Sigma^* \mid (q_0, w, \perp) \vdash_N^* (q, \varepsilon, \varepsilon) \}$$

$$N = (\Sigma, Q, \Gamma, \delta, q_0, \perp, F)$$

Oss: Per un certo PDA N, spesso

$$L[N] \neq P[N]$$

Oss: Dimostreremo però che se $L = L[N]$, allora esiste N' tale che $L = P[N']$, e viceversa, se $L = P[N]$, allora esisterà N'' tale che

$$L = L[N'']$$

cioè non cambia la classe dei linguaggi riconosciuti da PDA per stato finale o per pila vuota

Esempio: PDA per $\{ww^R \mid w \in \{a,b\}^*\}$

$N = (\Sigma, Q, \Gamma, \delta, q_0, \perp, F)$ dove

- $\Sigma = \{a, b\}$
- $Q = \{s_0, s_1, s_2\}$
- $\Gamma = \{a, b, \perp\}$
- $q_0 = s_0$
- $F = \{s_2\}$

STATO ATTUALE, INPUT, STACK

$$\delta(s_0, a, X) = \{(s_0, aX)\} \quad \forall X \in \Gamma$$

$$\delta(s_0, b, X) = \{(s_0, bX)\} \quad \forall X \in \Gamma$$

$$\delta(s_0, \varepsilon, X) = \{(s_1, X)\} \quad \forall X \in \Gamma$$

$$\delta(s_1, a, a) = \{(s_1, \varepsilon)\}$$

$$\delta(s_1, b, b) = \{(s_1, \varepsilon)\}$$

$$\delta(s_1, \varepsilon, \perp) = \{(s_2, \varepsilon)\}$$

$$\delta(s_2, \alpha, X) = \emptyset \quad \forall \alpha \in \Sigma \cup \{\varepsilon\}, \ \forall X \in \Gamma$$

Diagramma di Transizione

- esiste una "computazione" per abba che porta l'automa N allo stato finale s_2 (così come a svuotare la pila)

  $\Rightarrow$ abba è riconosciuta (sia per stato finale, sia per pila vuota)

- si può vedere che "svuoto la pila sse vado sullo stato s_2"

$$\Rightarrow L[N] = P[N]$$

per stato finale

per pila vuota

Esempio: PDA per $L = \{wcw^R \mid w \in \{a,b\}^*\}$

- N è deterministico:
  - $|\delta(q,a,Z)| \leq 1 \quad \forall q \in Q, \forall a \in \Sigma \cup \{\varepsilon\}, \forall Z \in \Gamma$
  - se $\delta(q,\varepsilon,Z) \neq \emptyset$, allora $\delta(q,a,Z) = \emptyset$ $\forall a \in \Sigma$

Oss: diversamente dai DFA, un PDA deterministico non garantisce di leggere tutto l'input. Ad es: N si blocca con input acc

- $L = \mathbf{P}[N]$ cioè il linguaggio L è riconosciuto per pila vuota

- $L[N] = \{wcy \mid w \in \{a,b\}^*$ e y è prefisso di $w^R\}$

Ad es: $ac \in L[N]$ $\quad w = a$, c, $y = \varepsilon$

- Se l'input è acab, questo non viene letto tutto: N si blocca dopo aver letto aca sullo stato finale cioè riconosce aca ma non acab

$$L = \{ wcw^R \mid w \in \{a,b\}^* \}$$

$$L = L[N] = P[N]$$

N è deterministico e riconosce L sia per stato finale, sia per pila vuota

---

La classe dei linguaggi riconosciuti per pila vuota o per stato finale non cambia

Teorema

(i) Se $L = P[N]$ (per pila vuota), possiamo costruire N' tale che $L = L[N']$

(ii) Se $L = L[N]$ (per stato finale), possiamo costruire N' tale che $L = P[N']$.

Dimostrazione

(i) N riconosce per pila vuota (con $\perp$ come iniziale) $\Rightarrow$ costruiamo N' che riconosce per stato finale (con Z come iniziale)

(ii) N riconosce per stato finale (con $\perp$ come iniziale) $\Rightarrow$ costruiamo N' che riconosce per pila vuota (con Z come iniziale)

serve per evitare che uno stato di N possa vuotare lo stack ($Z \notin \Gamma$)

Allora l'unico stato che può vuotare lo stack è f

Esercizi (da ling. a PDA)

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

Z simbolo iniziale $\quad \Gamma = \{A, Z\} \quad X \in \Gamma$

$L = L[N] = P[N]$

---

2) $L = \{ a^n b^{n+1} c^m \mid n, m \geq 0 \}$

$L = L[N]$ ← per stato finale

$P[N] = \emptyset$ ← PER PILA VUOTA

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

$\Gamma = \{ A_1, A_2, Z \}$

$L = L[N]$

per stato finale

All'incontrario, come ottenere un PDA a partire da una grammatica libera?

Un possibile metodo (top-down) è il seguente

$S \to aSb \mid \varepsilon$

- S simbolo iniziale sulla pila
- riconosce per pila vuota

$A \to aAa \mid bAb \mid \varepsilon$

PDA con un solo stato, con A sulla pila come simbolo iniziale, che riconosce per pila vuota

abba input

Costruisco una derivazione leftmost a partire da A

$A \Rightarrow aAa \Rightarrow abAba \Rightarrow abba$

<u>Teorema</u> Un linguaggio L è libero da contesto sse è accettato da un PDA

<u>Dim</u>)

$\Rightarrow$) Cioè 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 il PDA $N = (\underset{\Sigma}{T}, \{q\}, \underset{\Gamma}{T \cup NT}, \delta, q, \underset{\perp}{S}, \emptyset)$, dove la funzione di transizione δ simula la costruzione di una derivazione canonica sinistra (riscrivo sempre il nonterminale sul top, che è quello più a sx nella derivazione leftmost)

$$\delta(q, \varepsilon, A) = \{(q, \beta) \mid A \to \beta \in R\} \quad \forall A \in NT$$

$$\delta(q, a, a) = \{(q, \varepsilon)\} \quad \forall a \in T$$

Ogni volta che N ha un nonterminale A in cima alla pila, sceglie nondeterministicamente 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

$$S \Rightarrow^* w \quad \text{sse} \quad (q, w, S) \vdash_N^* (q, \varepsilon, \varepsilon)$$

con derivazione leftmost

$\Leftarrow$) Cioè se $L = P[N]$, allora esiste G libera tale che $L = L(G)$.

Molto più difficile: prova solo schematizzata

<u>Lemma 1</u>: Ogni PDA N può essere simulato da un PDA N' con un solo stato (aumentando opportunamente i simboli della pile)

<u>Lemma 2</u>: Ogni PDA con un solo stato, ha una equivalente grammatica libera.

<u>Idea della prova</u>:

Se $(q, B_1 B_2 \ldots B_k) \in \delta(q, a, A)$ con $a \in \Sigma \cup \{\varepsilon\}$ nel PDA con un solo stato, allora la grammatica G contiene la produzione

$$A \to a B_1 \ldots B_k$$

Si può dimostrare che $S \Rightarrow_G^* w$ sse $(q, w, S) \vdash_N^* (q, \varepsilon, \varepsilon)$

<u>Riassumendo</u>:

Un linguaggio L è libero

sse

è accettato da un PDA (nondeterministico)