Universtità degli Studi “G. D’Annunzio”

Chieti – Pescara

Facoltà di Economia

Corso di Laurea in Economia Informatica

Pescara

TESINA DI LAUREA

STRUMENTI FORMALI PER L’ANALISI LESSICALE – SINTATTICA DEI COMPILATORI

LAUREANDA

CATERINA MANDOLINI

RELATORE

STEFANO BISTARELLI

Anno Accademico 2003-2004

INDICE

1. Introduzione

Il titolo di questo lavoro è *strumenti formali per l'analisi lessicale-sintattica dei compilatori*. Soffermiamoci sugli elementi che compongono il titolo per capire di che cosa parleremo.

1) COMPILATORI

2) ANALISI LESSICALE – SINTATTICA

3) STRUMENTI FORMALI

1) Compilatori.

Che cos'è un compilatore?

Dal punto di vista funzionale (cioè se teniamo in considerazione la funzione che esso realizza) un compilatore è un traduttore così come lo concepiamo nella accezione comune.

Un traduttore ITA-ENG (italiano-inglese) compie un lavoro (funzione) che consiste nel tradurre testi scritti in italiano in testi scritti in inglese. Un compilatore, analogamente, realizza la funzione di traduzione da un linguaggio ad un altro.

2) Analisi Lessicale-Sintattica

IL primo passo nella funzione di traduzione è la verifica che il testo da tradurre sia scritto nel linguaggio "giusto".

IL traduttore ITA-ENG non è in grado di tradurre un testo scritto in russo.

Allo stesso modo il compilatore deve effettuare un controllo per accertasi che il testo (programma) che ha ricevuto in input sia scritto nel linguaggio "giusto".

Questo avviene tramite l'analisi lessicale-sintattica - più brevemente diremo analisi sintattica.

3) Linguaggi Formali

Alcuni elementi della teoria dei linguaggi formali sono la base per la progettazione, lo sviluppo e la realizzazione dei compilatori.

In questo lavoro, dopo aver illustrato a grandi linee il funzionamento di un compilatore, esamineremo alcuni elementi della teoria dei linguaggi formali che ne costituiscono il fondamento teorico.

Infine cercheremo di capire come si articolano l'analisi lessicale e sintattica.

2. Il compilatore come traduttore da linguaggi di alto livello a linguaggi di basso livello

P1 equivalente a P2

Un compilatore è un programma che riceve in input un programma, detto sorgente, scritto in un determinato linguaggio (anch'esso detto sorgente) e restituisce in output un programma, detto oggetto, scritto in un altro linguaggio (detto linguaggio oggetto).

La cosa importante è che P1 sia equivalente a P2, vale a dire che P2 esegua le stesse istruzioni specificate in P1.

Anche per il traduttore ITA-ENG vale la stessa regola. Il testo tradotto non è un qualunque testo in inglese, ma è un testo in inglese che ha lo stesso significato del testo in italiano.

Si dice che il processo di traduzione deve preservare la semantica.

LINGUAGGI DI PROGRAMMAZIONE

- Alto livello: Pascal, C, Java

X=Y+5

- Basso livello: Linguaggio macchina

001 0110

010 0001

I linguaggi di programmazione si distinguono in linguaggi di alto livello e di basso livello.

I linguaggi di alto livello sono "vicini" all'uomo, nel senso che è più facile scrivere e comprendere un programma scritto in un linguaggio di alto livello (ed è quindi minore la probabilità di commettere errori) rispetto ad un linguaggio di basso livello.

Nell'esempio s'intuisce il significato dell'istruzione scritta nel linguaggio di alto livello, mentre è più difficile interpretare la sequenza di bit delle istruzioni in linguaggio macchina.

Tipicamente il linguaggio sorgente è un linguaggio di alto livello e il linguaggio oggetto è il linguaggio macchina.

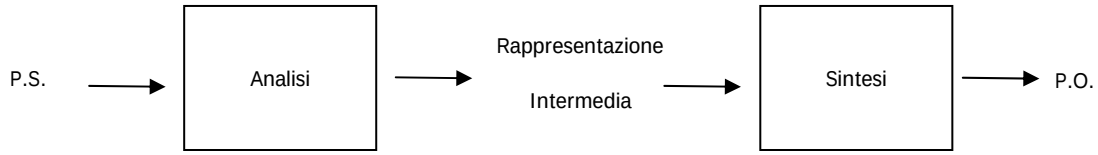
IL COMPILATORE CI CONSENTE

DI SFRUTTARE LE POTENZIALITÀ DELLA MACCHINA

(VELOCITÀ E PRECISIONE NELL'ESECUZIONE DELLE ISTRUZIONI)

EVITANDOCI L'ONERE DI IMPARARE IL LINGUAGGIO MACCHINA

3. Modello analisi - sintesi



Nel modello analisi-sintesi il compilatore viene visto come la composizione di due moduli (funzioni): il modulo dell'analisi e il modulo della sintesi.

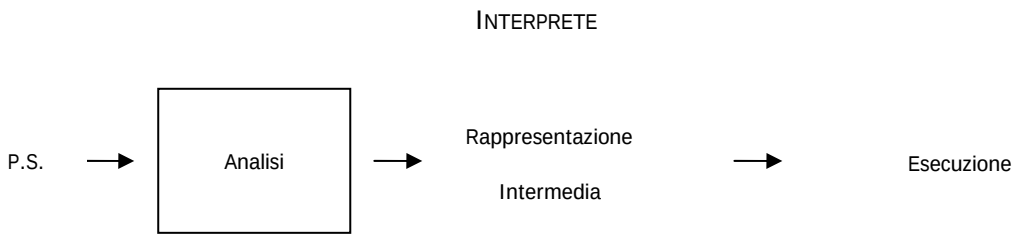
Il modulo dell'analisi è deputato all'analisi della struttura del programma (in particolare alla verifica che il programma sia stato scritto nel linguaggio "giusto") e alla costruzione della rappresentazione intermedia.

Il modulo della sintesi costruisce il programma oggetto a partire dalla rappresentazione intermedia.

Il modulo dell'analisi si articola in tre sotto-moduli: analisi lessicale, analisi sintattica e analisi semantica. I primi due verificano la correttezza sintattica del programma ed è nostro interesse capire come ciò avvenga; il terzo effettua una serie di controlli di tipo semantico: in un esempio ne descriveremo brevemente uno, il controllo statico dei tipi.

Per inciso, evidenziamo la differenza che intercorre tra la classe dei compilatori e un'altra classe di programmi con la quale egualmente possiamo sfruttare le potenzialità di calcolo della macchina utilizzando un linguaggio di alto livello.

Questa è la classe degli interpreti.



Il compilatore produce in output il programma oggetto il quale viene trasformato in un programma eseguibile (il cosiddetto .EXE). Il .EXE (insieme di istruzioni in linguaggio macchina) può essere mandato in esecuzione in qualsiasi momento successivo e ogni qualvolta l'utente lo desideri.

L'interprete, al contrario, non restituisce in output il programma oggetto;costruisce la rappresentazione intermedia e a partire da questa fa eseguire alla macchina le istruzioni contenute nel programma sorgente.

In questo caso l'esecuzione avviene passo-passo: le istruzioni vengono interpretate ed eseguite una alla volta dalla macchina virtuale del linguaggio oggetto.

4. Il modulo dell'analisi

- Analizzatore Lessicale C=2.795 .....Errore!!!
- Analizzatore sintattico 2.795:=C .....Errore!!!
- Analizzatore semantico C:=2.795 .....Errore!!!

L'analisi viene effettuata attraverso l'analizzatore lessicale, l'analizzatore sintattico (detto anche parser) e l'analizzatore semantico.

Vediamo cosa accade negli esempi.

L'intenzione è quella di assegnare alla variabile C il numero 2.795. Supponiamo che l'istruzione di assegnamento corretta sia :=.

1) Nel primo caso l'analizzatore lessicale dà errore perché non riconosce il simbolo dell'assegnamento. Infatti abbiamo supposto che il simbolo corretto sia: = e non =.
2) Nel secondo caso l'analizzatore lessicale non ha nulla da eccepire. L'analizzatore sintattico invece dà errore perché non riconosce la struttura della frase. L'ordine corretto è C:=2.795 e non 2.795:=C.
3) Nel terzo caso l'analizzatore lessicale e quello sintattico non rilevano errori ma, se per ipotesi abbiamo dichiarato C come una variabile di tipo carattere e tentiamo di assegnare il valore 2.795, l'analizzatore semantico solleva un'obiezione perché ci avverte che il tentativo di assegnare un valore

rappresentato con un numero di bit più grande dei bit necessari per contenere un carattere incorre in un troncamento del valore.

Come abbiamo anticipato tralasceremo l'analisi semantica (nell'esempio abbiamo avuto una idea del controllo statico dei tipi – statico perché avviene in sede di compilazione del programma e non a tempo di esecuzione) e concentriamo la nostra attenzione sull'analisi lessicale e sul parsing.

ANALIZZATORE LESSICALE E PARSER

Verifica della correttezza sintattica del programma

L'analizzatore lessicale e il parser verificano la correttezza sintattica del programma; ciò vuol dire che verificano che il programma sia stato scritto con i simboli corretti (analisi lessicale) e che tali simboli compaiano nell'ordine corretto (parsing).

Esaminiamo gli elementi della teoria dei linguaggi formali che ci consentono di capire il funzionamento dell'analizzatore lessicale e del parser.

5. Elementi della teoria dei linguaggi formali

- $\forall x \exists y\ (f(x)<y)$

  Il linguaggio matematico è un linguaggio formale o informale?

- Conta il numero dei bit

  La lingua italiana è un linguaggio formale o informale?

La risposta a tali domande dipende dal significato che diamo al termine formale. Nell'accezione comune il termine formale vuol dire qualcosa di rigoroso, non ambiguo. E diciamo che il linguaggio matematico è un linguaggio formale perché l'interpretazione di una formula matematica non lascia spazio ad ambiguità. Per contro, l'italiano è un linguaggio informale. La frase nell'esempio ha infatti due significati.

Nella teoria dei linguaggi formali i linguaggi vengono considerati solo dal punto di vista della forma e non del significato. Più precisamente, ma in maniera altrettanto semplice, ciò che conta sono i simboli e le relazioni tra essi.

Pertanto, da questo punto di vista, sia il linguaggio matematico che l'italiano sono linguaggi formali, cioè sono insiemi costituiti da frasi le quali non sono altro che sequenze finite di simboli.

LINGUAGGIO FORMALE

E' un insieme costituito da un numero (finito o infinito) di frasi.

Ogni frase è una sequenza finita di simboli.

In base alla definizione testè data siamo in grado di costruire un semplice linguaggio formale.

UN SEMPLICE LINGUAGGIO FORMALE

$H = \{f_1, f_2, ..., f_n, ...\}$

$f_1 = ♥$, $f_2 = ♥♥$, ... , $f_n = ♥^n$,...

Dal punto di vista formale, H e l'italiano sono oggetti della stessa natura.

Procediamo in maniera più formale (intendendo più rigorosa) nella trattazione degli elementi della teoria dei linguaggi formali.

Questa teoria individua le caratteristiche e le proprietà dei linguaggi in base alla loro struttura, cioè in base ai simboli e all'ordine in cui possono comparire nelle frasi. Un problema tipico è quello della decidibilità di un linguaggio: un linguaggio è decidibile se è possibile scrivere un algoritmo che presa in input una frase, vale a dire una sequenza di simboli (abbiamo detto che dal punto di vista formale un linguaggio è un insieme di frasi ciascuna delle quali è un insieme di simboli), è in grado di affermare "SI la sequenza appartiene al linguaggio" oppure "NO". I linguaggi più "semplici" (semplici dal punto di vista della struttura) hanno questa proprietà. Quelli più "articolati" non ce l'hanno. Pertanto la verifica che il programma sorgente ricevuto in input dal compilatore sia scritto nel linguaggio "giusto"consiste nel *decidere* se la stringa di simboli ricevuta in input (il programma) appartiene o meno al linguaggio sorgente.

Esaminiamo le definizioni di base.

ALFABETO

Insieme finito di simboli

$A = \{a, b, ..., z\}$ $B = \{0,1\}$ $C = \{♥\}$

L'alfabeto è un insieme finito di simboli.

STRINGA

Sequenza finita di simboli

A = {a, b, ..., z}

α_1 = economia α_2 = nomiaeco

β_1 = corso di laurea in economia informatica

β_2 = economia informatica corso laurea in di

Dato un alfabeto A una stringa (o frase) è una sequenza finita di simboli appartenenti ad A.

STELLA DI KLEENE

A* è l'insieme delle stringhe su A

A = {a, b, ..., z}

A* = {a, b, ..., z, aa, ab, ..., az, ba, bb, ...,

corso di laurea in economia informatica, ...}

A* è l'insieme delle stringhe che posso comporre utilizzando i simboli dell'alfabeto A.

Arriviamo alla definizione di linguaggio formale. Un linguaggio formale su un alfabeto A è un sottoinsieme di A* (cioè è un insieme di stringhe costruite con i simboli di A).

LINGUAGGIO FORMALE

è un insieme di stringhe su un alfabeto A

$L \subseteq A^*$

A = {a, b, ..., z}

L_1 = {a, aa, b, economia informatica, zaz}

L_2 = linguaggio della lingua italiana

Come facciamo a rappresentare L?

Se |L| < + inf non c'è nessun problema: basta elencare le stringhe che compongono il linguaggio.

Se |L| = + inf invece si pone il problema della rappresentazione di un insieme infinito mediante una struttura finita.

Un modo è quello delle grammatiche generative.

La specificazione di una grammatica generativa avviene mediante la definizione di un certo numero di categorie sintattiche e delle relazioni esistenti fra queste.

Perché ci poniamo il problema della rappresentazione di L?

Il motivo è il seguente: vogliamo che il computer risolva il problema della decidibilità di un linguaggio (vale a dire che il computer riconosca che il programma sorgente sia scritto o meno nel linguaggio sorgente); per fare ciò dobbiamo fornire al computer la stringa e il linguaggio (in realtà, una rappresentazione della stringa e del linguaggio). La stringa è una sequenza *finita* di simboli, per cui non si hanno particolari problemi per la sua rappresentazione. Il linguaggio, tipicamente è un insieme *infinito*, di qui il problema di individuarne una rappresentazione finita.

Torniamo alle grammatiche generative.

GRAMMATICHE GENERATIVE: esempio

<FRASE> ha la seguente struttura: <SOGGETTO> <PREDICATO> <COMPLEMENTO>

<COMPLEMENTO>ha la seguente struttura: <ARTICOLO> <NOME>

Nell'esempio diciamo che una frase è costituita da un soggetto seguito da un predicato a sua volta seguito da un complemento e che complemento è la concatenazione di un articolo e di un nome.

Scegliamo un numero limitato di istanze di soggetti e di nomi e specifichiamo le relazioni tra le categorie sintattiche e le istanze.

<FRASE> $\rightarrow$ <SOGGETTO> <PREDICATO> <COMPLEMENTO>

<SOGGETTO> $\rightarrow$ Luca | Maria

<PREDICATO> $\rightarrow$ guarda | chiama

<COMPLEMENTO> $\rightarrow$ <ARTICOLO> <NOME>

<ARTICOLO> $\rightarrow$ il

<NOME> $\rightarrow$ cane

I simboli tra parentesi angolate vengono detti non terminali, gli altri terminali. La relazione freccia è una operazione di riscrittura. <FRASE> è il simbolo iniziale.

Il linguaggio generato dalla grammatica è l'insieme delle

stringhe su A = {a, ... z} derivabili dalla categoria sintattica <FRASE>

mediante un numero finito di passi di riscrittura.

Vediamo un esempio.

Luca chiama il cane

1) <FRASE>

2) <SOGGETTO> <PREDICATO> <COMPLEMENTO>

3) Luca <PREDICATO> <COMPLEMENTO>

4) Luca chiama <COMPLEMENTO>

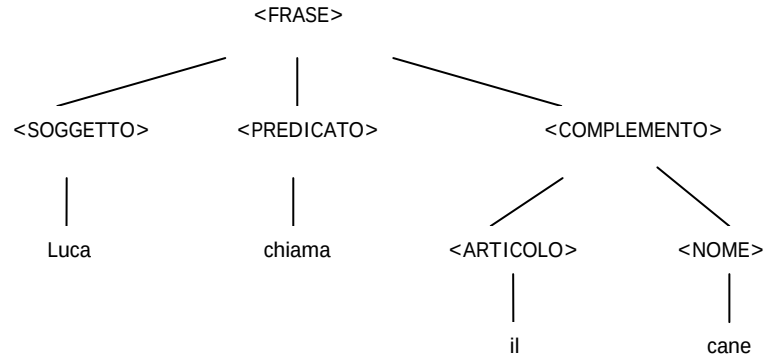
5) Luca chiama <ARTICOLO> <NOME>

6) Luca chiama il <NOME>

7) Luca chiama il cane

La frase “Luca chiama il cane” appartiene al linguaggio considerato perché è ottenuta a partire da <FRASE> applicando un numero finito di regole di riscrittura.

ALBERO DI DERIVAZIONE



Il processo di riscrittura può essere rappresentato da un albero in cui:

- la radice è il simbolo iniziale;
- le foglie sono i simboli terminali;
- i nodi intermedi sono i simboli non terminali;
- gli archi rappresentano le relazioni di riscrittura.

GRAMMATICHE REGOLARI

$A \rightarrow a\ B$

$A \rightarrow a$

GRAMMATICHE LIBERE DA CONTESTO

$A \rightarrow \beta$

Le grammatiche vengono classificate in base alle regole di riscrittura. Le classi di grammatiche cui siamo interessati sono le grammatiche regolari e le grammatiche libere da contesto.

Nelle grammatiche regolari ogni simbolo non terminale può essere riscritto con un terminale oppure con una sequenza terminale-non terminale. Nel processo di riscrittura di queste grammatiche il simbolo non terminale si trova sempre a destra.

Nelle grammatiche libere da contesto ogni simbolo non-terminale può essere riscritto con una stringa qualsiasi di terminali o non terminali.

Chiameremo le grammatiche libere da contesto più semplicemente grammatiche libere.

L'intera dicitura *libere da contesto* serve per differenziare tali grammatiche da quelle *dipendenti dal contesto*. Le regole di riscrittura in una grammatica dipendente dal contesto sono del tipo: $\alpha\ A\ \beta \rightarrow \alpha\ \gamma\ \beta$, cioè il non terminale A si riscrive in γ se è compreso tra α e β.

Invece in una grammatica libera da contesto il non terminale A si riscrive in β indipendentemente dalla stringa che precede e da quella che segue.

I linguaggi regolari e i linguaggi liberi sono classi di linguaggi piuttosto "semplici" ma che godono di importanti proprietà. Per entrambi esiste un'altra rappresentazione, oltre alle grammatiche a struttura di frase: l'automa riconoscitore.

L'automa riconoscitore ci consente di risolvere il problema della decidibilità dei linguaggi regolari e dei linguaggi liberi a un costo basso: rispettivamente O(n) e $O(n^3)$.

Inoltre per alcuni tipi di grammatiche libere (per le quali vengono introdotte alcune restrizioni sulle regole di riscrittura) la complessità del riconoscimento del linguaggio si riduce a O(n) (costo minimo).

Esaminiamo le grammatiche regolari.

GRAMMATICHE REGOLARI

$L=\{a^n b^m, n \geq 1, m \geq 1\}$

L è un linguaggio regolare

Grammatica:

$A \rightarrow a\ A$

$A \rightarrow b\ B$

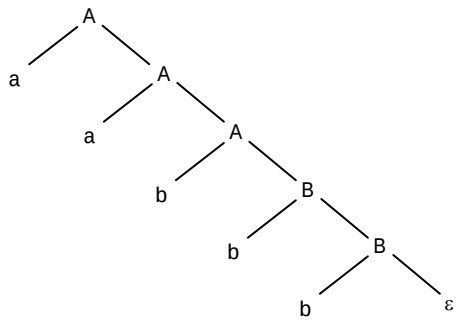
$B \rightarrow b\ B$

$B \rightarrow \varepsilon$

Le stringhe del linguaggio $a^n b^m$ sono formate da un certo numero di a cui segue un certo numero di b.

E' un linguaggio regolare: infatti esiste la grammatica regolare che lo genera.

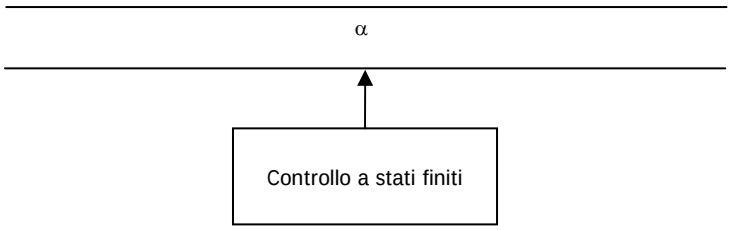
ALBERO DI DERIVAZIONE DELLA STRINGA aaabbb



L'albero di derivazione della stringa aabbb è una catena. Per questo motivo si dice che la struttura dei linguaggi regolari è di tipo lineare. Ciò deriva dal fatto che è il non terminale riscritto (l'unico) si trova sempre a destra. Proprio questa caratteristica fa sì che per questi linguaggi esista l'automa riconoscitore a stati finiti.

AUTOMA A STATI FINITI

Nastro di input

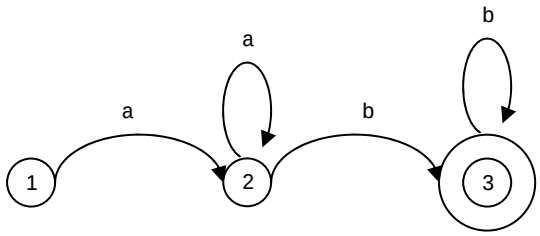


L'automa si trova in uno stato iniziale e legge la stringa in ingresso un simbolo alla volta.

A seconda del simbolo letto transita o meno in un altro stato.

Se al termine della lettura l'automa si trova nello stato finale (stato n. 3, indicato con il doppio cerchio) la stringa viene accettata altrimenti rifiutata.

$$L=\{a^n\, b^m,\ n \geq 1,\ m \geq 1\}$$



Esaminiamo le grammatiche libere.

GRAMMATICHE LIBERE

$L=\{a^n\, b^n,\ n \geq 1\}$

L è un linguaggio libero

Grammatica:

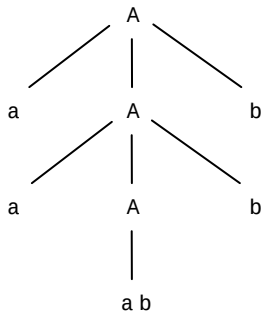
$A \rightarrow a\ A\ b$

$A \rightarrow a\ b$

$A \rightarrow \varepsilon$

Le stringhe di L sono formate da un certo numero di a seguito dallo stesso numero di b.

ALBERO DI DERIVAZIONE DELLA STRINGA aaabbb



I linguaggi liberi sono a struttura non lineare. Si dice che sono a struttura gerarchica. L'albero di derivazione non è una catena, ma è un albero.

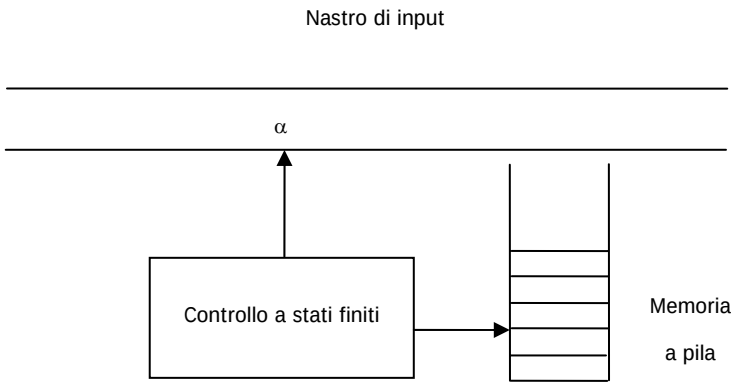
Si può dimostrare che non esiste l'automa a stati finiti per il linguaggio $L=\{a^n b^n , n\geq 1\}$. D'altra parte gli automi a stati finiti sono meccanismi che non hanno memoria o al più hanno memoria limitata – legata al numero degli stati.

Infatti, possiamo costruire l'automa per il linguaggio $L= \{a^{100} b^{100}\}$ che è in grado di controllare che il numero di a e di b nella stringa siano esattamente 100.

Ma per linguaggio $L=\{a^n b^n , n\geq 1\}$ abbiamo bisogno di un meccanismo che conti il numero di a di b, qualsiasi esso sia.

Vediamo gli automi a pila.

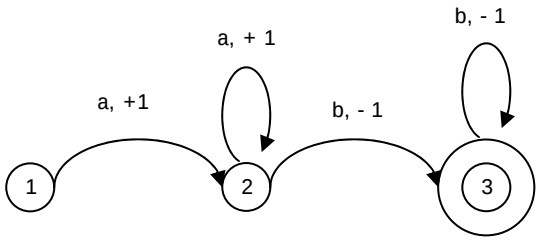
AUTOMA A PILA



La transizione da uno stato all'altro dipende oltre che dal simbolo letto nella stringa in input anche dal simbolo letto in cima alla pila.

Nella memoria a pila i simboli vengono scritti e letti solo dalla cima. I simboli +1 e -1 indicano rispettivamente l'operazione di push e di pop.

$L=\{a^n\, b^n,\ n \geq 1\}$



Se la lettura della stringa in input termina nello stesso momento in cui viene cancellato l'ultimo simbolo (la pila diventa vuota) allora l'automa accetta la stringa altrimenti la rifiuta.

Se sostituiamo al simbolo a la parentesi graffa aperta { e al simbolo b la parentesi graffa chiusa } il linguaggio L diventa l'insieme delle stringhe che hanno tante parentesi graffe aperte quante chiuse.

Ciò rappresenta la regola base di molti linguaggi di programmazione.

Ad esempio nel linguaggio C abbiamo le parentesi {}; nel linguaggio Pascal invece c'è Begin e End.

Naturalmente all'interno delle parentesi ci saranno i comandi, le dichiarazioni e i commenti.

In maniera estremamente semplificata rappresentiamo la sintassi di un linguaggio di programmazione con la seguente grammatica

$$S \rightarrow \{D;\ C\}$$

$$D \rightarrow \{D_1\}$$

$$D_1 \rightarrow \ldots\ldots$$

$$C \rightarrow \{C_1\}$$

$$C_1 \rightarrow \ldots\ldots$$

6. L' analizzatore lessicale e parser

Esaminiamo i concetti e la terminologia di base che riguardano l'analizzatore lessicale e il parser.

L'ANALIZZATORE LESSICALE

Tre concetti: token, lessema e pattern.

Associato a ogni TOKEN del linguaggio sorgente c'è una regola chiamata PATTERN, che descrive l'insieme delle LESSEMI (parole) che rappresentano quel TOKEN (categoria sintattica).

Ad esempio, se il token è NUM, i lessemi sono i numeri 1, 2 , 1288, 756, ecc., il pattern è "qualsiasi sequenza di simboli appartenenti all'alfabeto {0,..,9} che non cominciano per 0". Se il token è ID, i lessemi sono x, y9,

pippo, reddito2004, ecc., il pattern è “qualsiasi sequenza di simboli appartenenti all’alfabeto A={a,...,z, A,...,Z} o all’alfabeto B={0,...,9} tale che il primo simbolo appartenga ad A (cioè che non cominci per numero)”.

L’analizzatore lessicale fa lo scanning dell’input da sinistra a destra e produce una sequenza di token

IL PARSER

Prende in input una stringa di token A1,...,An e determina se è sintatticamente corretta rispetto a una grammatica libera G.

Esistono tre tipi di parser:

- Parser universali (in grado di fare il parsing rispetto a qualsiasi grammatica libera; sono tuttavia inefficienti: la complessità varia da $O(n^2)$ a $O(n^3)$);
- Parser top-down (il parse-tree viene costruito dalla radice);
- Parser bottom-up (il parse-tree viene costruito dalle foglie).

I parser top-down e bottom-up più efficienti funzionano su sottoclassi di grammatiche libere; alcune di queste, come le grammatiche LL e LR, sono sufficientemente espressive da descrivere la maggior parte dei costrutti del linguaggi di programmazione.

TRADE-OFF tra efficienza ed potere espressivo.

C’è una stretta collaborazione tra l’analizzatore lessicale e il parser. L’analizzatore lessicale fa lo scanning della stringa in input e procede al riconoscimento delle parole (lessemi). Trovata una parola l’analizzatore lessicale comunica al parser la corrispondente categoria sintattica (token). Il parser costruisce (meglio, tenta di costruire) il parse-tree.

7. Interazione LEX e YACC

STRUMENTI AUTOMATICI

Analizzatore lessicale: LEX

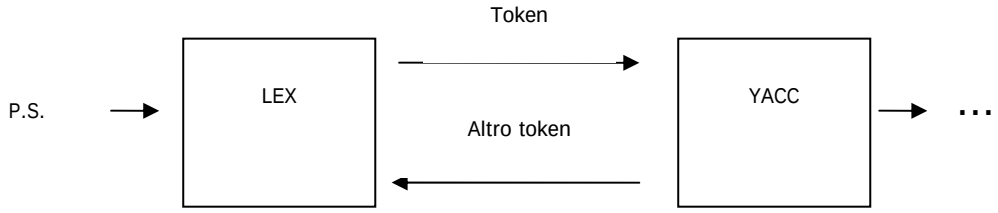
Analizzatore sintattico: YACC

(YET ANOTHER COMPILER COMPILER)

Lex e Yacc sono due tool che consentono di costruire in maniera molto semplice e veloce l’analizzatore lessicale e il parser per i linguaggi generati da particolari grammatiche libere: le grammatiche LR. I parser LR (che riconoscono linguaggi generati da grammatiche LR) sono efficienti; la complessità dell’algoritmo di riconoscimento è O(n).

In assenza di questi e di altri strumenti la progettazione e l'implementazione di un compilatore costituiscono un corposo e complesso lavoro. La realizzazione del primo compilatore Fortran ha richiesto un lavoro di équipe della durata di 18 anni.

INTERAZIONE LEX – YACC



Esaminiamo la struttura di un programma Lex.

Sintassi LEX

$P_1, P_2, ..., P_n$

%%

P_1 {Azione 1}

P_2 {Azione 2}

...

P_n {Azione n}

La sezione dichiarazioni comprende le definizioni regolari P_1, P_2,... P_n.

Le definizioni regolari specificano i pattern.

Nella sezione comandi viene indicata per ogni pattern la relativa azione, che Lex deve compiere nel momento in cui trova un lessema che corrisponde al pattern.

Lex viene attivato da Yacc. Trovato un lessema Lex esegue l'azione che normalmente consiste in un ritorno di controllo al parser unitamente al token. Se nell'azione non viene previsto il ritorno di controllo, lex continua ad effettuare lo scanning per individuare il lessema successivo.

In questo modo l'analizzatore lessicale pulisce il testo dagli spazi bianchi, i ritorni a capo, i commenti, ecc.

Lex può passare anche altre informazioni a Yacc oltre ai token. Queste informazioni (dette attributi) vengono memorizzate in una variabile (yylval per la precisione) alla quale accede anche Yacc.

Programma LEX: esempio

```
LETTERA [A-Z a-z]
CIFRA [0-9]
ID   {LETTERA}({ LETTERA } | { LETTERA })*
%%

{ID} {yylval=0; return (ID);}
```

L'esempio fornisce la descrizione di un programma Lex che riconosce gli identificatori: nella sezione dichiarazioni abbiamo definito il pattern del token ID (sequenza di lettere e cifre che comincia per lettera). Nella sezione comandi abbiamo definito l'azione semantica che Lex deve compiere quando individua un lessema appartenente a ID: nella fattispecie yylval viene settata a 0 e viene restituito il controllo a Yacc unitamente al token ID.

Vediamo adesso qual è la struttura di un programma Yacc.

Sintassi YACC

```
% token   T1, T2,..., Tn,
%%

A1: β1   {Azione semantica 1}
A2: β2   {Azione semantica 2}
...
An: βn   {Azione semantica n}
```

La sezione dichiarazioni comprende la dichiarazione dei token.

Nella sezione comandi ci sono le regole di traduzione.

Ogni istruzione è composta da una produzione della grammatica e della relativa azione semantica.

L'azione semantica è un insieme di istruzioni C.

La variabile $$ contiene il valore dell'attributo associato al non terminale a sinistra. La variabile $i contiene il valore dell'attributo relativo all'i-esimo simbolo (terminale o non terminale) della stringa a destra.

L'azione semantica è compiuta nel momento in cui il parser, costruendo l'albero, individua il non terminale (nodo padre) a sinistra perché ha riconosciuto la relativa stringa a destra (nodi figli). Quindi il valore dell'attributo $$ è funzione degli attributi $i.

Il seguente programma effettua la traduzione dal linguaggio delle espressioni al linguaggio dei numeri. Ad es. l'espressione (5*3+1) viene tradotta in 16.

Programma YACC: esempio

```
% token   NUMERO
%%
espr: espr '+' espr   {$$= $1 + $3;}
    | espr '-' espr   {$$= $1 - $3;}
    | espr '*' espr   {$$= $1 * $3;}
    | espr '/' espr   {$$= $1 / $3;}
    | espr '-' espr   {$$= $1 - $3;}
        | '(' espr ')' {$$= $2}
            | NUMERO
```

Programma LEX

```
NUMERO [0-9]
%%
{NUMERO} {yylval=valore();
return (NUMERO);}
```

Osservazione.

Sembrerebbe che Yacc sia più amichevole di Lex, in quanto per il primo è sufficiente specificare le regole di riscrittura della grammatica mentre per il secondo è necessario scrivere le definizioni regolari.

In realtà anche la sintassi di Lex è strutturata in maniera tale da rendere immediata la specificazione dei token (sebbene senza passare per regole della grammatica regolare). Infatti le definizioni regolari si basano su un altro formalismo, quello delle espressioni regolari, mediante il quale è possibile specificare i linguaggi regolari.

D'altra parte quando pensiamo alla definizione di un token, cioè ad un pattern, pensiamo a "un insieme di simboli che cominciano per …tranne che per.. che possono appartenere a questo insieme oppure a quello" ed è più naturale dare queste specifiche in termini di espressioni regolari piuttosto che di linguaggi derivabili da regole di riscrittura.

8. Altri usi dei compilatori

Abbiamo detto che il compilatore tipicamente è un traduttore da linguaggi di alto livello a linguaggi di basso livello. Tuttavia abbiamo visto nell'esempio precedente che il compilatore può essere impiegato nella traduzione di stringhe appartenenti a due linguaggi qualsiasi. Infatti il linguaggio delle espressioni contiene, ad esempio, stringhe del tipo: 5*8 + (7-6*2) e il linguaggio dei numeri contiene stringhe costituite da sequenze di cifre da 0 a 9, ad esempio: 1, 725, 6669,1001.

Esiste una classe di linguaggi, i linguaggi visuali, le cui frasi non sono costituite da stringhe (cioè collezioni di simboli disposte in concatenazione lineare) bensì da sentenze visuali (vale a dire collezioni di simboli disposti nello spazio).

LINGUAGGIO VISUALE

E' un insieme di sentenze visuali

SENTENZA VISUALE

E' una collezione di oggetti grafici disposti attraverso relazioni spaziali

OGGETTO GRAFICO

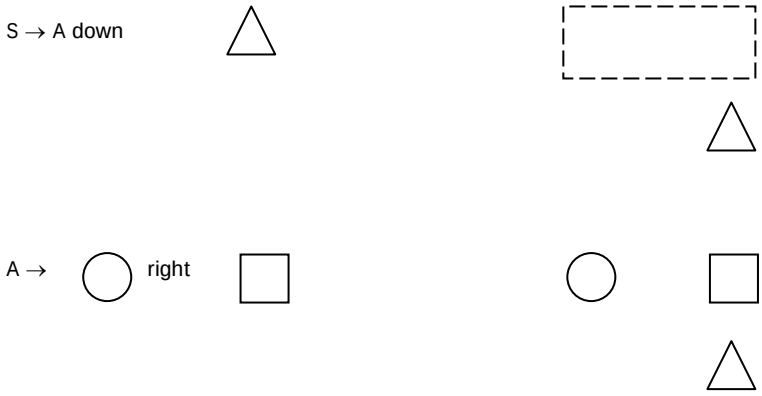
E' una coppia: (Immagine, attributi sintattici)

La grammatica generativa viene arricchita con altre informazioni.

Le produzioni sono del tipo A→X1 R1 X2 R2 …Rm-1Xm dove:

- A è un oggetto grafico non terminale
- Xi sono oggetti grafici terminali e non terminali
- Ri è una collezione di relazioni spaziali

A → X1 R1 X2 R2 ...Rm-1 Xm



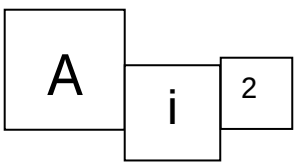
Una grammatica a struttura di frase è un istanza particolare di un linguaggio visuale in cui non è necessario specificare le relazioni Ri perché i simboli si riscrivono mediante concatenazione lineare.

Alcuni formattatori di testo sono compilatori. Essi traducono dal linguaggio delle istruzioni (che specificano il grassetto, il titolo, gli elenchi puntati e numerati, ecc.) al linguaggio visuale che dispone nello spazio, e con la formattazione specificata, i caratteri.

STRINGA NEL LINGUAGGIO SORGENTE

A sub {i sup 2}

STRINGA NEL LINGUAGGIO OGGETTO: A_i^2



APPENDICE

1. Grammatiche a struttura di frase

Una grammatica a struttura di frase è una quadrupla G=(V,T,S,P) dove:

- V è un insieme finito non vuoto;
- $T \subset V$ è l'insieme dei simboli terminali;
- (V-T) è l'insieme dei simboli non terminali;
- $S \in (V-T)$ è il simbolo iniziale;
- P è un insieme finito di produzioni $\alpha \rightarrow \beta$ dove $\alpha \in V^+$ e $\beta \in V^*$

2. Classificazione delle grammatiche alla Chomsky

Nella definizione di grammatica a struttura di frase non si impone nessun vincolo sulle produzioni. Le grammatiche di questo tipo sono chiamate grammatiche di ***tipo 0***.

Imponendo delle restrizioni sulla natura delle produzioni si ottengono altri tipi di grammatiche.

Una grammatica è detta ***monotona*** se per ogni produzione $\alpha \rightarrow \beta$, si ha che $|\alpha| \leq |\beta|$.

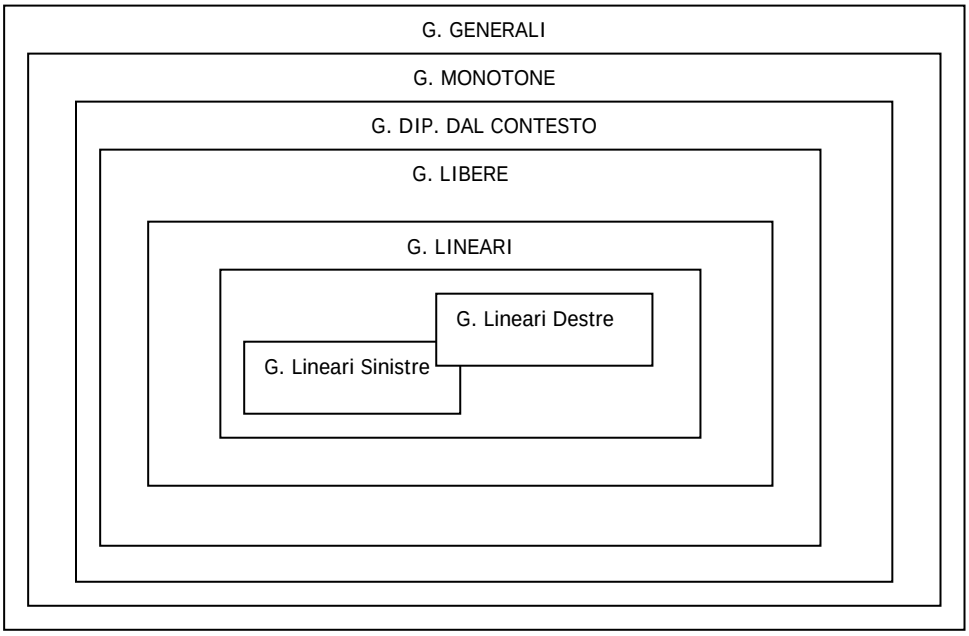
Ciò significa che ogniqualvolta si applica una produzione, la stringa risultante non è mai più corta di quella che l'ha generata.

Una grammatica è detta ***dipendente dal contesto*** se ogni produzione è del tipo $\gamma A \delta \rightarrow \gamma \beta \delta$, con $\gamma, \delta \in V^*$, $\beta \in V^+$, $A \in (V-T)$: cioè A può essere sostituito da β solo se è presente nel contesto $\gamma A \delta$.

Una grammatica è detta ***libera da contesto*** o semplicemente ***libera***, se ogni produzione è del tipo $A \rightarrow \beta$, con $\beta \in V^+$: cioè A può essere sostituita da β indipendente dal contesto in cui si trova.

Una grammatica è detta ***lineare*** se ogni produzione è del tipo $A \rightarrow \beta$, con $\beta \in V^+$ e β contenente al più un simbolo non terminale.

Una grammatica è detta ***lineare destra*** o ***regolare*** se ogni produzione è del tipo $A \rightarrow aB$ oppure $A \rightarrow a$. Se invece ogni produzione è del tipo $A \rightarrow Ba$ oppure $A \rightarrow a$, la grammatica è detta ***lineare sinistra***.



3. Grammatiche Regolari

3.1. Le espressioni regolari

Le espressioni regolari su un alfabeto $A=\{a_1, a_2, ..., a_n\}$ costituiscono un formalismo per la rappresentazione di linguaggi.

Per descrivere tale formalismo possiamo servirci di una grammatica:

$$G = (\{E\} \cup T, T, E, P)$$

dove:

- $T = \{\lambda, \phi, +, *, \cdot,), (\} \cup A$
- e P è costituito dalle produzioni:

$$E \rightarrow (E+E) \mid (E \cdot E) \mid (E)^*$$

$$E \rightarrow \lambda \mid \phi \mid a_1 \mid a_2 \mid ... \mid a_n$$

Il linguaggio generato da questa grammatica costituisce l'insieme delle espressioni regolari sull'alfabeto A.

3.2. Automi a stati finiti deterministici

Si chiama *automa a stati finiti deterministici* (per brevità lo indicheremo con ASFD) un sistema M definito come segue:

$$M=(Q, A, t, q_o, F)$$

dove:

- Q è un insieme finito di stati;
- A è un insieme finito di caratteri che costituiscono l'alfabeto;
- t: $Q \times A \rightarrow Q$ è una funzione che associa ad ogni coppia (stato, carattere) uno stato;
- q_o è lo *stato iniziale* e $q_o \in Q$;
- F è l'insieme degli *stati finali*, $F \subseteq Q$.

Per completare la definizione di automa come riconoscitore di parole sull'alfabeto A, consideriamo un'altra funzione t′: $Q \times A^* \rightarrow Q$, definita, sull'insieme degli stati Q e sull'insieme delle parole A^*, e data ricorsivamente da:

$$t'(q,\lambda) = q$$
$$t'(q, xa) = t(t'(q, x), a) \qquad \text{con } x \in A^* \text{ ed } a \in A.$$

Mentre con t si indica una singola transizione dell'automa, con t′ si indicano tutte le transizioni effettuate dall'automa per effetto della parola in ingresso. Quando questo non darà luogo ad ambiguità useremo il simbolo t per indicare entrambe le funzioni.

Diamo la definizione di linguaggio riconosciuto (o accettato) da un ASFD. Sia $L \subseteq A^*$ un linguaggio sull'alfabeto A, si dice che L è *riconoscibile mediante un automa finito* quando esiste un automa M = (Q, A, t, q_o, F) per cui $L = L(M) = \{x \mid x \in A^*, t(q_o, x) \in F\}$.

3.3. Automi a stati finiti non deterministici

Un *automa a stati finiti non deterministico* (per brevità ASFND) è un sistema M definito come segue:

$$M=(Q, A, t, , F)$$

dove:

- Q, A, q_o, F sono definiti come per gli automi finiti;
- t: $Q \times A \rightarrow 2^Q$ con 2^Q insieme potenza di Q o insieme di tutti i sottoinsiemi di Q.

In questo modo ad ogni transizione non è associato un unico stato ma un insieme finito di stati, quindi se $t(q_o, a) = \{q_1, q_2, q_3\}$ questo significa che la macchina nello stato q_o se legge **a** può transitare in uno di tre possibili stati.

La funzione di transizione si estende al dominio $Q \times A^*$ definendo:

$$t'(q,\lambda) = \{q\} \qquad \text{e} \qquad t'(q, xa) = \bigcup_{p \in t'(q,xa)} t(p,a)$$

e al dominio $2^Q \times A^*$ con

$$t''(\{p_1, p_2, \ldots, p_k\}, x) = \bigcup_{i=1}^{k} t'(p_i, x)$$

Nel seguito indicheremo una qualsiasi di queste funzioni di transizioni con t se dal contesto sarà chiaro a quale delle tre si fa riferimento.

La definizione di parola riconosciuta da questo tipo di automa è: M riconosce una parola x se esiste almeno una sequenza di stati che inizia con q_o e termina con uno stato finale e tale che le singole transizioni siano consistenti con la parola letta e la descrizione dell'automa.

Il linguaggio riconosciuto da M è l'insieme:

$$L(M) = \{x \mid p \in Q, \text{ tale che } p \in t(q_o, x) \wedge p \in F\}$$

3.4. Equivalenza tra grammatiche regolari, espressioni regolari, automi a stati finiti deterministici ed automi a stati finiti non deterministici

I linguaggi generati da grammatiche regolari sono rappresentabili mediante espressioni regolari.

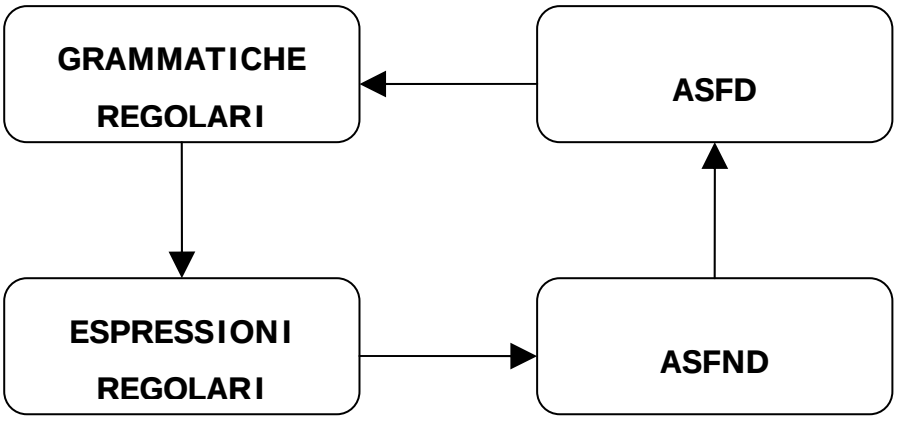
Tra gli insiemi $L \subseteq A^*$ riconosciuti da un ASFD ed i linguaggi generati dalle grammatiche regolari esiste una corrispondenza.

Infatti dato un automa a stati finiti M, esiste una grammatica regolare tale che L(G)=L(M).

Inoltre per ogni ASFND, N, ne esiste uno equivalente deterministico, D, tale che L(N)=L(D).

Infine se α è un'espressione regolare allora esiste un ASFND, N, tale che L(N) è l'insieme di parole rappresentate da α.

L'equivalenza tra i quattro formalismi è rappresentata dallo schema seguente.



4. Grammatiche Libere

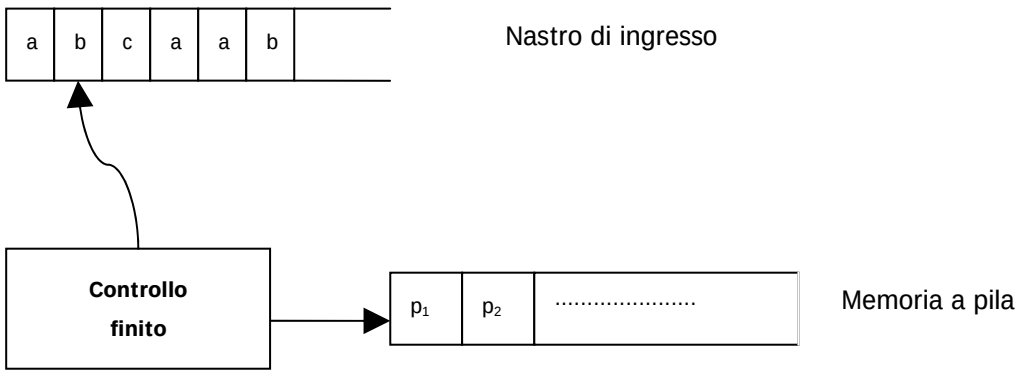
4.1. Automi a pila

La limitazione nella capacità di riconoscimento degli ASFD dipende dal fatto che questi posseggono una memoria, costituita dagli stati, finita.

Per questo motivo essi non sono in grado di riconoscere linguaggi che, per la loro struttura, richiedono di ricordare una quantità di "informazioni" non limitate. Come nel caso del linguaggio $a^n b^n$ che richiede di ricordare il numero di "a" letti per poter controllare che li segua uno stesso numero di "b".

Per aumentare la capacità di riconoscimento dell'automa a stati finiti occorre avere la possibilità di ricordare una quantità illimitata di informazione e far dipendere l'azione da compiere non solo dal carattere letto e dallo stato ma anche da un altro parametro che dipenda in qualche modo dall'informazione memorizzata. Precisiamo meglio questi concetti descrivendo un modello che chiameremo ***automa a pila*** (AP).

Esso è costituito da un controllo a stati finiti, da un nastro di ingresso e da una memoria ausiliaria a pila di lunghezza infinita alla quale si può quindi accedere *solo* dalla testa.



In ogni situazione l'AP può compiere due tipi di mosse:

1) Leggere il contenuto di una cella del nastro ed il simbolo in cima alla pila e, in base a ciò ed allo stato in cui si trova, passare in un nuovo stato e sostituire il simbolo letto dalla pila con una stringa (eventualmente λ), mentre la testina sul nastro di ingresso avanza di una posizione a destra. In cima alla pila apparirà ora il primo simbolo della stringa che vi è appena stata scritta.

2) Come nella mossa del primo tipo, ma senza leggere alcuna simbolo dal nastro (viene letto λ) e senza avanzamento della testina.

La mossa di tipo 2 permette all'AP di manipolare il contenuto della pila senza leggere ulteriori simboli dal nastro; tuttavia, qualora l'automa intendesse leggere qualche simbolo in posizione interna nella pila, esso perderebbe completamente traccia di quanto lo precedeva.

Per definire il linguaggio riconosciuto da un AP vi sono due possibilità distinte. Una parola è accettata da un AP se, dopo averla letta:

1) l'automa si trova in uno stato finale; oppure
2) la pila risulta vuota.

In entrambi i casi comunque viene riconosciuta la stessa classe di linguaggi.

Per questo tipo di automi, invece, non vale una proprietà molto importante che gli ASF posseggono e cioè l'equivalenza tra automi deterministici e non deterministici.

Un automa a pila non deterministico M è un sistema $(Q, A, R, t, q_o, Z_o, F)$ dove:

- Q è un insieme finito di stati;
- A è un alfabeto finito, detto alfabeto del nastro;
- R è un alfabeto finito, detto alfabeto della pila;
- $q_o \in Q$ è lo stato iniziale;
- $Z_o \in R$ è il simbolo iniziale, cioè l'unico simbolo che appare all'inizio della pila
- $F \subseteq Q$ è l'insieme degli stati finali;
- t è una funzione da $Q \times (A \cup \{\lambda\}) \times R$ nei sottoinsiemi finiti di $Q \times R^*$.

La t definisce in quali stati può passare l'automa e cosa scrivere nella pila dato lo stato in cui si trova ed i simboli letti sul nastro e sulla pila.

Un automa a pila è deterministico quando la funzione di transizione t specifica non più di una mossa per ogni configurazione ed in particolare le mosse di tipo 2 sono possibili solo quando quelle di tipo 1 non sono permesse. La classe dei linguaggi riconosciuti da APD è più ristretta di quella dei linguaggi riconosciuti da APND. Quest'ultima infatti coincide con la classe dei linguaggi liberi, mentre la prima coincide con la classe dei linguaggi generati da un particolare tipo di grammatiche libere: le grammatiche LR(k).

BIBLIOGRAFIA

- AIELLO, ALBANO, ATTARDI, MONTANARI

  Teoria della commutabilità, logica, teoria dei linguaggi formali

  Materiali didattici ETS

- AHO, SETHI, ULLMAN

  Compilers. Principles, techniques and tools.