

LFC (Linguaggi Formali e Compilatori) - Note del Corso

Edoardo Lenzi

October 4, 2017

Contents

1	Introduzione	2
1.1	Front-End of the Compiler	2
1.2	Back-End of the Compiler	2
1.3	Lexical analysis	2
1.4	Session syntax analyzer	3
1.5	Semantic analyzer	3
1.6	Intermediate code generation	3
1.7	Code generation	3
2	2	4
3	Lezione 1	5
3.1	Uso del Pumping Lemma per determinare linguaggi liberi	8

Chapter 1

Introduzione

Un compilatore é un programma che legge un **linguaggio source** e lo traduce in un **equivalente linguaggio di programmazione target**. Solitamente il compilatore compila in **assembly** e poi un **assembler** produce codice macchina. Se il target language é un programma eseguibile può processare input e produrre output.

Un **interprete** é un altro tipo di language processor, invece di tradurre il linguaggio lo esegue direttamente quindi piglia sia il source program che gli input e processa l'output

Infine il **preprocessore** risolve le macro nel sorgente codificandole in linguaggio nativo (espandendole) prima di compilare.

Solitamente il compilato va piú veloce mentre l'interprete ti dà diagnosi più accurate dato che esegue il codice. Nel caso di Java compilo il sorgente in linguaggio intermedio **bytecode** che poi interpreto sulla JVM.

Il **linker** "linka" assieme moduli e librerie dove ho riferimenti ad altri file (risolve gli indirizzi). Il **loader** invece fa il merge in memoria per l'esecuzione.

1.1 Front-End of the Compiler

La **parte analitica** del processo di compilazione spacca la sorgente in parti costituenti e impone su di esse una struttura grammaticale (stile dtd); sfrutta questa struttura per creare una rappresentazione intermedia. Se non passa la validazione grammaticale mi tira errori. Il sorgente viene storicizzato in una struttura dati chiamata **symbol table**.

1.2 Back-End of the Compiler

La **parte di sintesi** invece traduce il sorgente guardando la rappresentazione intermedia e la symbol table; le parti di analisi e sintesi sono chiamate anche **front-end of the compiler** mentre le restanti **back-end**.

1.3 Lexical analysis

Fa uno scan e raggruppa le parole in **lexems**, per ogni lexem genera un **token** della forma

(token name, attribute value)

Il **token name** é un simbolo astratto usato nella syntax analysis mentre il **value** é un puntatore alla symbol table entry.

ie)

`position = initial + rate * 60` diventa (id, 1) (=) (id, 2) (+) (id, 3) (*) (60)
gli operatori matematici sono simboli astratti che non hanno attribute value (?non sono referenziati nella symbol table?).

1.4 Session syntax analyzer

É un parsing, con i token crea una **rappresentazione ad albero (syntax tree)** nel quale il nodo é un operatore e i figli gli operandi.

gli operatori devono avere priorit  per costruire l'albero, la struttura grammaticale serve anche a definire le priorit  degli operatori.

1.5 Semantic analyzer

Piglia il **syntax tree** e guarda se   semanticamente consistente con la definizione del linguaggio. (ie **type checking**). Il linguaggio puo ammettere cast impliciti chiamati **coercizioni** o tirare cogne.

“into float”   una coercizione dell'intero 60 in float dato che gli altri operandi sono float.

1.6 Intermediate code generation

Nel processo di compilazione posso avere varie rappresentazioni intermedie come alberi etc.. Dopo semantic analysis solitamente creo una codice basso livello, machine-like, *“easy to produce and easy to translate into target machine code”*. Nella figura ho un tree address code ricavato dal syntax tree.

In un tree address code a destra ho al massimo un operatore (assembly like), e le operazioni sono in ordine.

Devo avere variabiline intermedie

1.7 Code generation

Segue la fase opzionale di **code optimization**, prende la rappresentazione inermedia e la mappa in un target language. Le istruzioni intermedie vengono tradote in istruzioni macchina (presumibilmente).

Devo capire come mappare variabili su registri

Nella symbol table devo storicizzare tutti gli attributi di un variable name.

Solitamente posso agglomerare le fasi di analisi in front end pass e le altre in back end pass.

Chapter 2

2

Syntax describe la forma appropriata di un programma Semantic describe il significato...
Per specificare la sintassi uso BNF (Backus Naur Form) context-free grammar

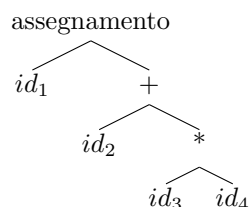
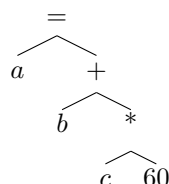
Chapter 3

Lezione 1

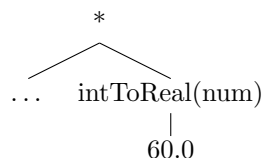
Vecchio sito

Riguardo linguaggi di analisi lessicale, si usano **simboli e caratteri**. Ho una tavola dei simboli creata in base al programma (e al compilatore). Restituisce un **token** (puntatore) ad un record nella tavola dei simboli. La maggior parte delle implementazioni usano un numero come **identificatore**. Analisi sintattica si basa sulla grammatica del linguaggio. Grammatica generativa vedere se é possibile generare una frase con la grammatica (syntax error). abstract syntax tree; nelle grammatiche formali metto in evidenza l'assegnamento.

$a = b + c \cdot 60$

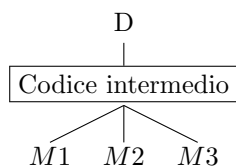


L'analisi semantica si occupa di vedere se c'è una corretta semantica (variabili dichiarate precedentemente). Se * necessita di un float allora 60 dev'essere convertito a float



Generazione di codice intermedio:

$temp_1 =$	$intToReal(60)$	VISITA DELL'ALBERO
$temp_2 =$	$id_3 * temp_1$	
$temp_3 =$	$id_2 + temp_2$	
$id_1 =$	$temp_3$	



Una **grammatica** é una tupla $G(V, T, S, P)$ con:

V	vocabolario
T	set simboli terminali
S	start symbol
P	set delle produzioni
$V \setminus T$	simboli
ϵ	parola vuota, non può essere un terminale!

Per convenzione i caratteri in maiuscolo denotano simboli non terminali mentre in minuscolo terminali. Quindi i simboli in T sono tutti lettere minuscole.

Considero X, Y variabili, generico simbolo in V e $\alpha \beta \delta$ stringe su V^* (ripetere 0+ volte i simboli) $S \rightarrow aSb$, $S \rightarrow \epsilon$, $S \rightarrow A$, T=a, b non terminali $(V \setminus T) = S$, A

Lo start symbol é usato nella prima produzione solo se produzioni libere dal contesto. Produzione legittima $aS \rightarrow b$ S a destra di una P non libera dal contesto.

Una grammatica generata é libera dal contesto (context free) se \forall sua produzione ha la forma: $A \rightarrow \beta$ con A simbolo non terminale. ($\alpha \rightarrow \beta$, α deve contenere simboli non terminali).

$S \rightarrow ASb| \epsilon$, $S \rightarrow \epsilon$ (privo di derivazione)

$S \rightarrow aSb \rightarrow ab$ $S \rightarrow aSb \rightarrow aaSbb \rightarrow aabb \implies \{a^n b^n / n \geq 0\}$ Struttura di bilanciamento per parentesi (blocchi begin end).

$\mu = \mu_1 \alpha \mu_2$, $\alpha \rightarrow \beta$ é produzione di grammatica G e γ é uguale a $\mu_1 \beta \mu_2$ derivazione in piú passi μ deriva in uno o piú passi data la grammatica G se ho una sequenza $L(G)$ linguaggio generato da grammatica = insieme di stringhe

$S \rightarrow aSb$, $S \rightarrow aAb$, $S \rightarrow ab$, $A \rightarrow aaAb$, $A \rightarrow \epsilon$ $\{w / w \in T^* \wedge s \implies +W\}$ $\{a^n b^n / n > 0, S = 0, S = 1\}$ $L = \{a^n b^n / n > 0\}$

Dato il linguaggio L possono esistere piú grammatiche diverse tra loro che generano L é indecidibile il linguaggio generato dalla grammatica G, \nexists *algadatoGedatoaLchediceche* $L = L(G)$.

$S \rightarrow aSBc|abc$ $cB \rightarrow Bc$ $bB \rightarrow bb$

$S \rightarrow AB$ $A \rightarrow A$ $A \rightarrow a$ $B \rightarrow Bb$ $B \rightarrow b$ tutto ciò che deriva da A é indipendente da ciò che deriva da B. aaa Ab b aaa $aaAb$ bb a^5 $aaAb$ b^3 $a^{2n+1}b^{n+1}$, $n \geq 0$

$S \rightarrow aB$, $S \rightarrow \epsilon$, $L(G) = \emptyset$, $L(G) = \{\epsilon\}$, $S \rightarrow 0B|1A$, $A \rightarrow 0|0S|1AA$, $B \rightarrow 1|1S|0BB$

Definisco G_1 / il linguaggio L generato da G_1 = insieme delle parole $L(G) = \{a^k b^N, k > 0\}$ $S \rightarrow ab|aS|Sb$ $S \rightarrow aS|aB$ $B \rightarrow bB|b$, $S \rightarrow^+ a^i S$

$S \rightarrow AB$, $A \rightarrow Aa|a$, $B \rightarrow bB|b$

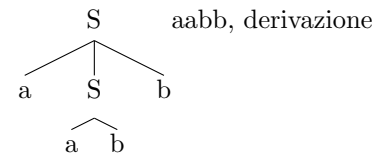
$L(G) = \{a^i b^m c^{2n}\}$

$S \rightarrow AB$, $A \rightarrow Aa|a$, $B \rightarrow bBcc|bcc$

$a^k b^n d^{2k}$, $S \rightarrow Sdd|aBdd$, $B \rightarrow bB|b$, $S \rightarrow aSdd|abdd$

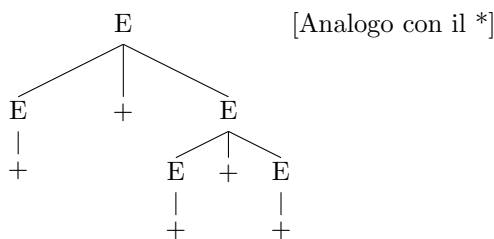
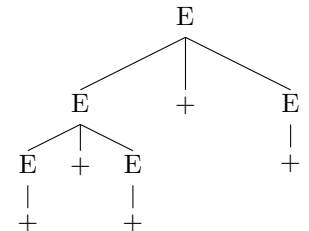
non terminale $\alpha \rightarrow \beta$ stringa grammatiche dipendenti dal contesto per vedere la differenza fra context dependent e quelle libere. Grammatiche libere si prestano in modo naturale a descrivere derivazioni in viste ad albero.

$S \rightarrow aSb|ab$ abstract syntax tree da albero di derivazione



canonica $\mu \rightarrow \gamma$ Ambiguità di G Nel caso di grammatiche libere si definiscono derivazioni canoniche destre e sinistre (rightmost/leftmost derivating) nel caso di rightmost si richiede che ad ogni passo di derivazione ($\mu \rightarrow \gamma$) venga rimpiazzato il terminale t a destra in μ (rispettivamente leftmost ρ a sinistra).

G é ambigua se $\exists L(G)$ / esistono per w due derivazioni canoniche distinte entrambe destre o entrambe sinistre. $E \rightarrow E + E|E * E|+$ (il + associa a sinistra)



$S \rightarrow \text{if } b \text{ then } S \mid \text{if } b \text{ then } S \text{ else } S \mid \text{altro if } b \text{ then if } b \text{ then altro else altro if } b \text{ then } S \text{ else } S$

$G(V, T, S, P)$ Un linguaggio L é libero (da contesto) se esiste una grammatica libera G tale che $L = L(G)$. In generale dato un linguaggio generale L ed una grammatica G \nexists un algoritmo per dimostrare che $L = L(G)$

Grammatiche libere chiusura (se faccio operazioni su stringhe restano grammatiche libere)

Lemma: La classe dei linguaggi liberi é chiusa rispetto all'unione. Il linguaggio che contiene tutte e sole le parole $W \in L_1 \cup L_2$ é esso stesso un linguaggio libero.

L_1 é libero $\implies \exists G_1 = (V_1, T_1, S_1, P_1)$ / $L_1 = L(G_1)$ [analogo per L_2]

avendo ridenominati i non-terminali di G_1 e G_2 in modo da non avere anonimia

Lemma: La classe dei linguaggi liberi é chiusa rispetto alla concatenazione (se L_1, L_2 sono liberi allora $\{v_1 v_2 / v_1 \in L_1 \wedge v_2 \in L_2\}$ é un linguaggio libero).

$G = (V, T, S, P)$ $\alpha \rightarrow \beta$, $\alpha, \beta \in B^+$ con al meno un terminale In una grammatica libera A (*nonterminale*) $\rightarrow \beta$ Le grammatiche libere sono quelle in cui tutte le produzioni in P hanno tutte forma $A \rightarrow \beta$.

Un linguaggio é libero se \exists una grammatica libera che lo genera

ie) linguaggio $\{a^n b^n / n > 0\}$ libero perché \exists una grammatica libera che lo genera (G_1). $G_1 \quad S \rightarrow aSb / ab$ G_1 libera $G_2 \quad s \rightarrow aAb, A \rightarrow aaAb, A \rightarrow \epsilon$ G_2 non libera \square

$L = \{vw / w \in \{a, b\}^*\}$ Con il **pumping lemma** posso dimostrare se un linguaggio é libero o meno.

Pumping lemma: Sia L un linguaggio libero allora $\exists p \in \mathbb{N}^+ / \forall z \in L : |z| > p \exists uvwxy / z = uvwxy \wedge |vwx| \leq p \wedge |vx| > 0 \wedge \forall i \in \mathbb{N} uv^iwx^iy \in L$.

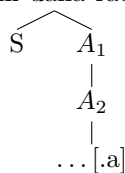
$\exists p \in \mathbb{N}^+$ esiste una costante $p > 0$
 $\forall z \in L : |z| > p$ ogni parola con più elementi di p
 $\exists uvwxy / z = uvwxy$ esistono 5 sottostringhe che compongono z
 $|vwx| \leq p$ la lunghezza delle 3 stringhe centrali é n
 $|vx| > 0$ la seconda e la quarta non sono mai ent
 $\forall i \in \mathbb{N} uv^iwx^iy \in L$ se ripeto i volte (i può essere 0) la 2 e la

$G \quad S \rightarrow aSb / ab$ $G \quad S \rightarrow A, A \rightarrow aAb / ab$

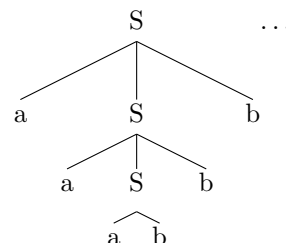
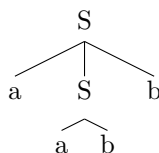
Chomsky normal form (no doppioni)

ie) $G_1 \quad S \rightarrow aSb / ab$ $G_2 \quad B \rightarrow aBb / ab$ \leftarrow doppioni $G \quad S \rightarrow A, A \rightarrow aAb / ab$

Dim L é linguaggio libero $\implies \exists$ una grammatica G in Chomsky Normal Form tale che $L = L(G)$. Definisco P come la lunghezza della parola più lunga che può essere derivata usando un albero di derivazione i cui cammini dalla radice sono lunghi al più come il numero di simboli non terminali della grammatica ($|V \setminus T|$). $S \rightarrow aSb / ab$ Albero di derivazione: piglio un non terminale e lo



espando con figli quanto vale β

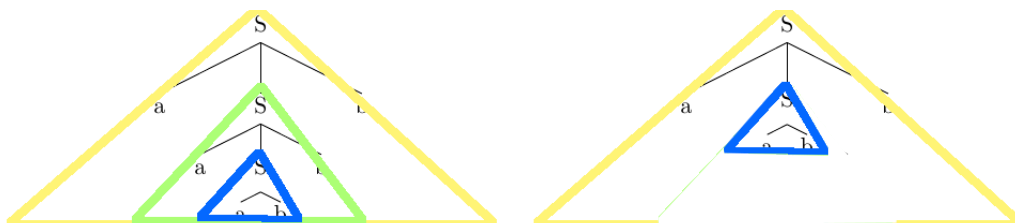
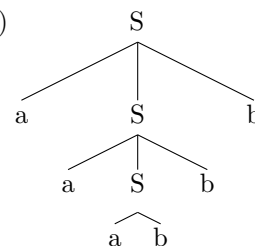


P é la lunghezza della parola più lunga che posso limitare. $S \rightarrow A_1 \rightarrow A_2 \rightarrow \dots \rightarrow A_k \rightarrow a$ [tutti simboli non terminali tranne a]

ie) $p = 2$, se prendo una qualunque parola più lunga di 3 generata da G , $aaaabbbb$ la posso dividere in 5 con due pumpable $u=aa, v=a, w=abb, x=b, y=b$

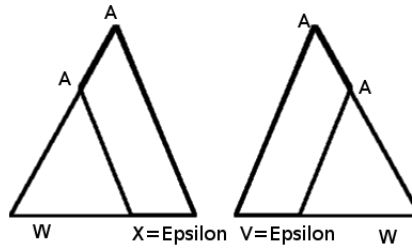
Se prendo $z \in L \wedge |z| > p \implies$ ho dovuto usare un albero di derivazione / \exists al meno un cammino più lungo di $|V \setminus T|$ per definizione di z . \implies ho un non terminale ripetuto al meno due volte $\implies \exists$ al meno un non terminale che occorre al meno 2 volte lungo quel cammino

con l'un-pumping la parola sta sempre nel linguaggio (taglio un pezzo di albero) ie)

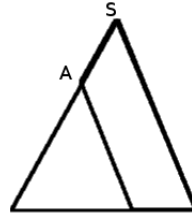


\square

Dato che w e x non possono essere entrambi nulli al massimo avrò $A \rightarrow aA$, o $A \rightarrow Aa$



ie) linguaggio libero $\{a, b\}$, $S \rightarrow ab$



3.1 Uso del Pumping Lemma per determinare linguaggi liberi

Tesi: Sia L un linguaggio libero allora $\exists p \in \mathbb{N}^+ / \forall z \in L : |z| > p \exists uvwxy / (z = uvwxy \wedge |vwx| \leq p \wedge |vx| > 0 \wedge \forall i \in \mathbb{N} uv^iwx^iy \in L)$

Considero $L_1 = ww / w \in \{a, b\}^*$ non é libero

Dim: Suppongo L_1 libero, sia p un numero naturale positivo qualunque (se scelgo un p arbitrario la dim non serve a nulla). Sia $z = a^p b^p a^p b^p$ allora $z \in L_1$, $|z| > p$. Siano $uvwxy / z = uvwxy \wedge |vwx| \leq p \wedge |vx| > 0$, distinguiamo varie possibilità:

- 1) vwx é composto da 'a' che occorrono a sinistra (w_1)
- 2) vwx é a cavallo e contiene sia 'a' che 'b' in w_1
- 3) vwx contiene solo 'b' in w_1
- 4) 'b' in w_1 e 'a' in w_2

5,6,7) ...speculare su w_2

Nei casi 1, 3, 5, 7 considero le parole $z^1 = uv^0wx^0y$ ($i=0$); nel caso 1 sono certo di togliere alcune occorrenze di a quindi avrò $z^1 = a^k b^p a^p b^p$, $k < p \implies z^1 \notin L$. Nel caso 3 $z^1 = a^p b^k a^p b^p$, $k < p \implies z^1 \notin L$. Nel caso 5 $z^1 = a^p b^p a^k b^p$, $k < p \implies z^1 \notin L$. Nel caso 7 $z^1 = a^p b^p a^p b^k$, $k < p \implies z^1 \notin L$.

Nei casi 2, 4, 6 invece avrò ancora $z^1 = uv^0wx^0y$ ($i=0$); Nel caso 2 $z^1 = a^k b^p a^p b^p$, o $a^p b^k a^p b^p$, o $a^j b^k a^p b^p$, $j, k < p \implies z^1 \notin L$