



DISCLAIMER

Il materiale contenuto nel drive è stato raccolto e richiesto tramite autorizzazione ai ragazzi frequentanti il corso di studi di Informatica dell'Università degli Studi di Salerno. Gli appunti e gli esercizi nascono da un uso e consumo degli autori che li hanno creati e risistemati per tanto non ci assumiamo la responsabilità di eventuali mancanze o difetti all'interno del materiale pubblicato.

Il materiale sarà modificato aggiungendo il logo dell'associazione, in tal caso questo possa recare problemi ad alcuni autori di materiale pubblicato, tale persona può contattarci in privato ed elimineremo o modificheremo il materiale in base alle sue preferenze.

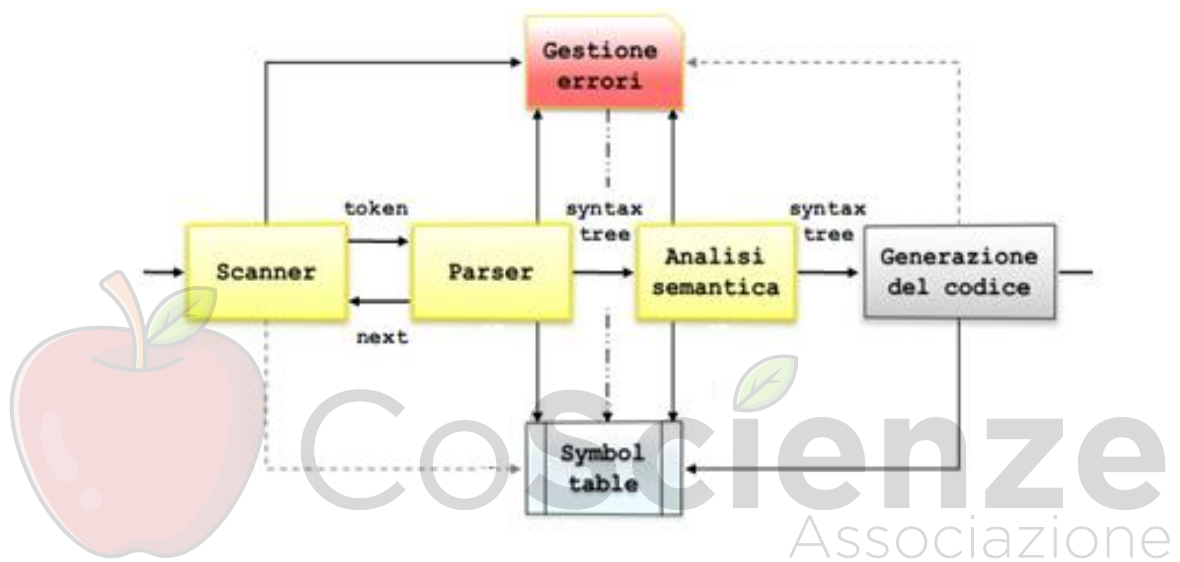
Ringraziamo eventuali segnalazioni di errori così da poter modificare e fornire il miglior materiale possibile a supporto degli studenti.



CoScienze
Associazione

2008/2009

Pierpaolo Corvese / Costante Luca



[COMPILATORI]

Sommario

LEZIONE 1 - INTRODUZIONE	4
PROCESSO DI TRADUZIONE	4
LEZIONE 2 – IL COMPILATORE	7
LEZIONE 3 – ANALISI LESSICALE (1 PARTE)	12
LEZIONE 4 – ANALISI LESSICALE (2 PARTE)	17
LEZIONE 5 – ANALISI LESSICALE (3 PARTE)	20
LEX.....	25
JFLEX.....	28
LEZIONE 6 – ANALISI SINTATTICA (1 PARTE).....	32
PARSING TOP-DOWN	37
LEZIONE 7 – ANALISI SINTATTICA (2 PARTE).....	39
LEZIONE 8 – ANALISI SINTATTICA (3 PARTE).....	46
FIRST.....	50
FOLLOW.....	51
LL(1).....	55
LEZIONE 9 – ANALISI SINTATTICA (4 PARTE).....	57
PARSING BOTTOM-UP	57
PREFISSO VITALE.....	60
LEZIONE 10 – ANALISI SINTATTICA (5 PARTE).....	63
PARSER LR	63
PARSING TABLE LR(0)	66
LEZIONE 11 - ESERCITAZIONE	69
LEZIONE 12 – ANALISI SINTATTICA (6 PARTE).....	72
PARSING TABLE SLR(1).....	72
PARSING TABLE LR(1)	74
PARSING TABLE LALR(1)	79
YACC	84
LEZIONE 13 - ESERCITAZIONE	86
JAVACUP.....	90
SINTASSI	91
LEZIONE 14 – ANALISI SEMANTICA (1 PARTE)	92
GRAMMATICHE C.F. AD ATTRIBUTI	92

LEZIONE 15 – ANALISI SEMANTICA (2 PARTE)	101
COSTRUZIONE DEGLI ALBERI DI SINTASSI: SYNTAX TREE.....	101
VALUTAZIONE BOTTOM-UP PER LE GRAMMATICHE S-ATTRIBUITE	104
LEZIONE 16 – ANALISI SEMANTICA (3 PARTE)	106
GRAMMATICA L-ATTRIBUITA.....	106
DEFINIZIONI GUIDATE DALLA SINTASSI	107
SCHEMA DI TRADUZIONE	108
LEZIONE 17 – ANALISI SEMANTICA (4 PARTE)	110
VALUTAZIONE BOTTOM-UP DEGLI ATTRIBUTI EREDITATI	110
LEZIONE 18 – ANALISI SEMANTICA (5 PARTE)	115
TYPE CHECKING	115
PARTE DICHIARATIVA	117
CORPO DEL PROGRAMMA.....	120
EQUIVALENZA DELLE ESPRESSIONI DI TIPO	122
CONVERSIONE DEI TIPI	123
LEZIONE 19 – GENERAZIONE DEL CODICE INTERMEDIO (1 PARTE)	124
IMPLEMENTAZIONE DEL CODICE A TRE INDIRIZZI	128
LEZIONE 20 – GENERAZIONE DEL CODICE INTERMEDIO (2 PARTE)	133
Type Checking (controllo dei tipi).....	133
Regole per il controllo dei tipi	133
Controllo del flusso.....	133
Espressioni Booleane.....	134
Codice Short Circuit	134
Statement di flusso di controllo	134
Traduzione di controllo di flusso di espressioni booleane	137
Eliminare i goto ridondanti	138
Backpatching	139
Generazione del codice in un solo passo usando il backpatching.....	139
Statement del flusso di controllo.....	142
LEZIONE 21 – GENERAZIONE DEL CODICE INTERMEDIO.....	144
LEZIONE 22 - ESERCITAZIONE	149
ESERCIZI VARI	151

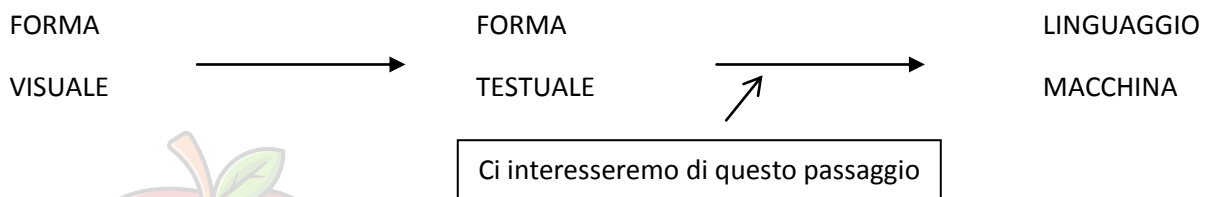
LEZIONE 1 - INTRODUZIONE

02/03/09

Applichiamo i risultati teorici dei linguaggi formali per definire un compilatore. **Un compilatore traduce il programma in linguaggio macchina il quale dipenderà dalla macchina:**

- Per i Macintosh si dovrà avere un linguaggio macchina per il processore Motorola;
- Per i PC si dovrà avere un linguaggio macchina per Intel.

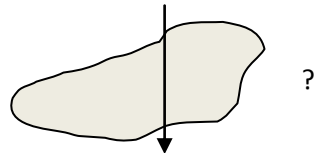
La traduzione di un linguaggio in linguaggio macchina, quindi, non è cosa standard da fare una volta per tutte, ma dipende dalla macchina. Anche le forme visuali dei linguaggi visuali saranno tradotte prima in forma testuale e poi in linguaggio macchina:



Ci riferiremo al modello di programmazione imperativo (Pascal, C etc...).

PROBLEMA: Tradurre un programma in un linguaggio ad alto livello in assembler in modo efficiente.

Es. `position := initial*rate+60`



`MOVF id3, R2`

`MOVF #60,0,R2`

La traduzione deve essere efficiente, cioè deve portare al minor numero di linee di codice assembler; tale problema è stato risolto con tecniche di ottimizzazione del codice (compilatori ottimizzati).

PROCESSO DI TRADUZIONE

I FASE: ANALISI LESSICALE (eseguita con un simulatore di automa finito)

Cioè leggere le istruzioni parola per parola e dire se ogni parola ha senso o meno, capire se i simboli sono parole del linguaggio usato, cioè se tutte le parole sono giuste.

Risultato: `id:= id*id+num`

Identificatore di numero ←

← Identificatore di variabile

Ad esempio in Pascal gli identificatori devono iniziare con una lettera.

L'analisi lessicale si può descrivere attraverso:

1. Espressioni regolari
2. Automi finiti

Vi è un software, LEX, che fa automaticamente l'analisi lessicale (è detto analizzatore lessicale).

II FASE: ANALISI SINTATTICA (eseguita con un simulatore di automa PUSH-DOWN): analizza come le parole sono correlate tra di loro.

Es. Pascal: position:= initial*rate AND 60 è sintatticamente errata.

Si fornisce un'interpretazione sintattica che costruisce una struttura gerarchica:

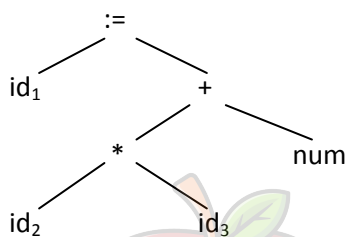


	Tabella dei simboli
1	position
2	initial
3	rate

L'analisi sintattica è descritta da:

1. Grammatiche context-free
2. Automi push-down

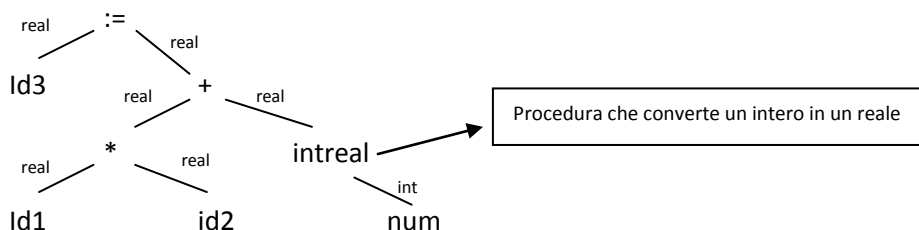
Vi è un software, YACC, che data la grammatica del linguaggio fa automaticamente l'analisi sintattica, cioè genera automaticamente la struttura sintattica.

III FASE: ANALISI SEMANTICA

Si cerca di capire se ciò che si è scritto ha un significato o meno. Ad esempio non ha senso assegnare un booleano ad una variabile intera, hanno assegnate quantità di memoria diversa. Quindi si controlla se tutti i tipi sono corretti, consistenti; si fa un controllo dei tipi, quando è possibile fa conversione di tipi. Ad esempio un intero può diventare un reale. Dobbiamo essere noi a dare una semantica alla macchina.

Es. real*real=real

Risultato:



A questo punto tutto è consistente: l'espressione appartiene al linguaggio; tutte le regole sono rispettate. Ora inizia una traduzione in espressioni più simili al codice macchina.

IV FASE: GENERAZIONE CODICE INTERMEDIO

Risultato: $\text{temp1} := \text{intreal}(60);$

 $\text{temp2} := \text{id}_1 * \text{id}_2$

 $\text{temp3} := \text{temp2} + \text{temp1}$

 $\text{id}_3 := \text{temp3}$

La caratteristica del codice intermedio è che tutte le istruzioni sono dello stesso tipo:

$\text{var} \leftarrow \text{argomento operatore argomento}$

sono analoghe a quelle del linguaggio macchina (operatore con due operandi). Solitamente il codice intermedio è molto lungo; per questo motivo si esegue la V fase.

V FASE: OTTIMIZZAZIONE: riduce il numero di istruzioni nel codice intermedio:

$\text{temp} := \text{id}_1 * \text{id}_2$

 $\text{id}_3 := \text{temp} + \text{intreal}(60)$

è un'ottimizzazione logica del codice intermedio che non dipende dalla macchina. Tale ottimizzazione renderà il linguaggio macchina finale abbastanza efficiente.

VI FASE: GENERAZIONE CODICE MACCHINA

Dai risultati precedenti (sappiamo i tipi di ogni variabile e che i tipi sono consistenti) si avrà che il codice macchina sarà consistente. Ad ogni variabile sarà associato il necessario spazio di memoria (possiamo fare questo assegnamento dato che ne conosciamo il tipo).

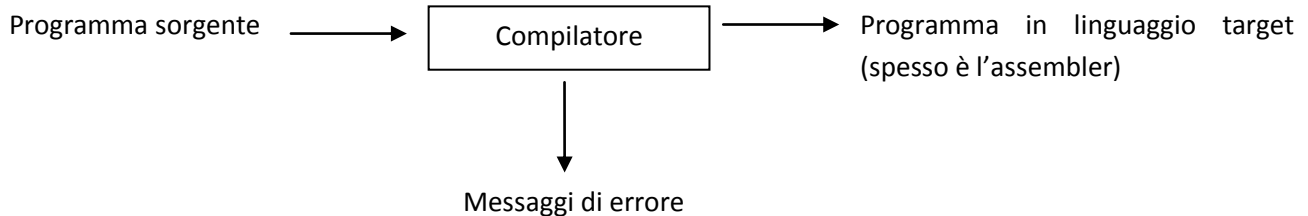
Es. $\text{MULF} \leftarrow \text{floating-point}$

Quindi le prime cinque fasi ci forniscono tutte le informazioni per poter costruire automaticamente il linguaggio macchina. A volte c'è un'ulteriore fase di ottimizzazione che dipende dalla macchina finale (si può fare solo se si conosce il linguaggio della macchina. Ad es. si minimizza il numero di cicli etc...).

LEZIONE 2 – IL COMPILATORE

04/03/09

Un compilatore è un programma che legge un programma scritto in un linguaggio e lo traduce in un programma equivalente scritto in un altro linguaggio. Durante questo processo di traduzione, il compilatore riporta all'utente la presenza di errori nel programma sorgente.



Il linguaggio target può essere sia un linguaggio ad alto livello che un linguaggio macchina.

Compilatore: da un programma se ne ricava un altro che va eseguito sulla macchina.

Interprete: da un programma si genera direttamente l'esecuzione, c'è un passaggio di meno.

Un programma compilato, che ha una fase di ottimizzazione, sarà più veloce di un linguaggio interpretato. Anche l'assembler è un linguaggio interpretato però l'interpretazione è molto rapida poiché avviene direttamente in hardware. Esistono "ambienti integrati" che hanno l'editore con cui scrivere il programma; ci sono "editori strutturati" che mettono in risalto le varie componenti del linguaggio che si scrivono. Ciò permette già una facile correzione di errori lessicali e sintattici nella fase di scrittura del programma.

Quindi l'editor strutturato aiuta a scrivere il programma con minori errori.

Es.

```
#include <stdio.h>

main(){

    printf("ciao");

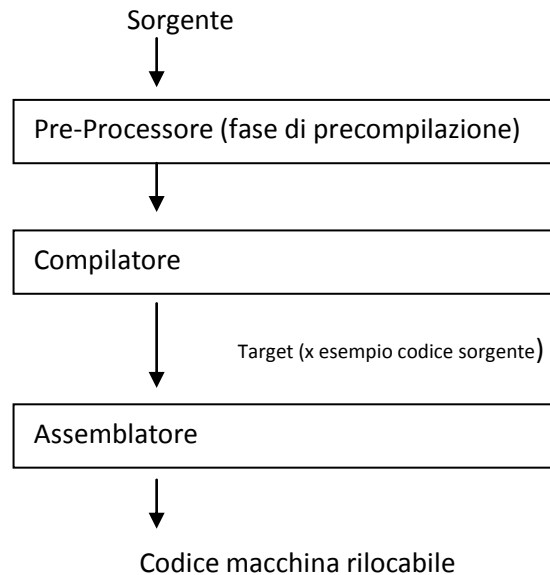
}
```

Programma scritto con l'editor
strutturato. Bisogna compilare
tale programma

Il file `stdio.h` contiene tutte le procedure delle funzioni di I/O. quindi per usare `printf` bisogna per forza includere il file `stdio.h` (non siamo noi a scrivere la procedura `printf` ma sarà presa dal file `stdio.h`). Il precompilatore considera tutte le direttive `#include` e va a prendere nei rispettivi file tutte le informazioni necessarie all'interno del programma (es. va a prendere il codice che implementa la `printf`). Quindi quello scritto non è ancora il codice sorgente da dare al compilatore ma deve prima passare per il precompilatore.

Es. `#define pigreco 3.14`

Il precompilatore che trova tale direttiva `"#define"` la sostituisce con `3.14`. Quindi la precompilazione analizza solo le macro e trasforma il testo sviluppando le macro e riportandole in puro linguaggio C. Si avrà così a disposizione il sorgente. Abbiamo questa visione:



Abbiamo a disposizione una serie di strumenti:

1. EDITORE STRUTTURALE
2. PRECOMPILATORE (costruisce il linguaggio sorgente in linguaggio puro);
3. COMPILATORE (dà una serie di errori o il compilato; individua dove è commesso l'errore e di che tipo di errore si tratta). Questa serie di strumenti facilita il compito dei programmatori rendendo maggiore il lavoro della macchina.

Nei compilatori c'è la possibilità di compilare con il DEBUGGER (correzione degli errori a tempo di esecuzione). Si applica quando il programma è stato tradotto correttamente nell'eseguibile ma non sappiamo ancora se l'esecuzione è corretta o meno (logicamente). Il debugger inserisce nel codice altro codice per il controllo delle variabili; si può poi eseguire il programma una linea alla volta e vedere il valore delle variabili in ogni linea; si può così controllare tutto il flusso di esecuzione ed i valori delle variabili. Se ci interessano i valori assunti dalle variabili in una particolare linea di codice è possibile mettere un break-point in modo da fermare l'esecuzione direttamente su quella linea ed avere i valori delle variabili (il codice con il debugger è più lungo). Concluso con il debugger, quindi assicuratisi della correttezza logica del programma si compila tutto con opzioni di ottimizzazioni. La compilazione ottimizzata la si fa solo alla fine, per la versione finale.

Il Target del compilatore può essere:

- Stringa di bit (codice assoluto);
- Linguaggio assembly (quindi si avrà bisogno di un assemblatore che traduca l'assembly in bit).

4. CODICE ASSEMBLER RILOCABILE: Rilocabile significa che nel programma assembler gli indirizzi sono relative alle proprie istruzioni:

es. 01 sto

02 JMP

10 LOAD

Il compilatore non sa a priori in quale zona di memoria sarà caricato tale codice, quindi si dovranno poi cambiare gli indirizzi per renderli da relativi ad assoluti (effettivi indirizzi di memoria). Per poter poi eseguire il codice ci vorrà il:

5. LOAD/LINK (carica/collega)

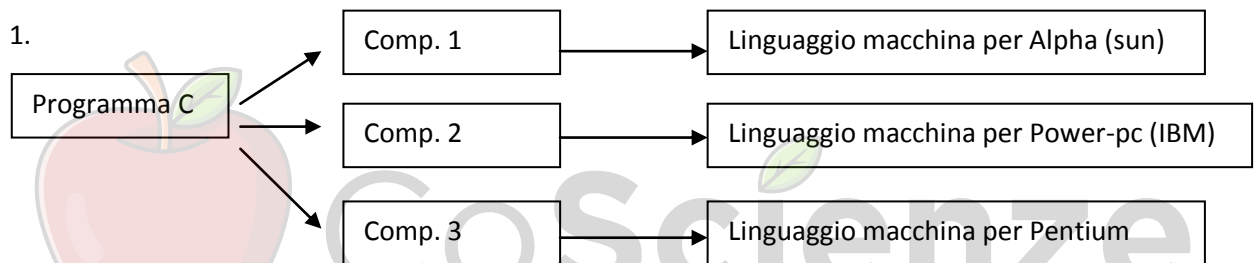
Che si occupa di caricare il programma in memoria centrale, assoluta (cambia gli indirizzi da relativi ad assoluti), e fa i collegamenti a procedure (es. printf) che non sono nel nostro codice ma in librerie esterne.

A questo punto si ha il codice assoluto da mandare in esecuzione sulla macchina. Ci sono compilatori che includono le fasi 3,4,5 (i compilatori sono collegati ai sistemi operativi e alle architetture). I linguaggi source e target possono essere qualsiasi, dipende da ciò che vogliamo fare).

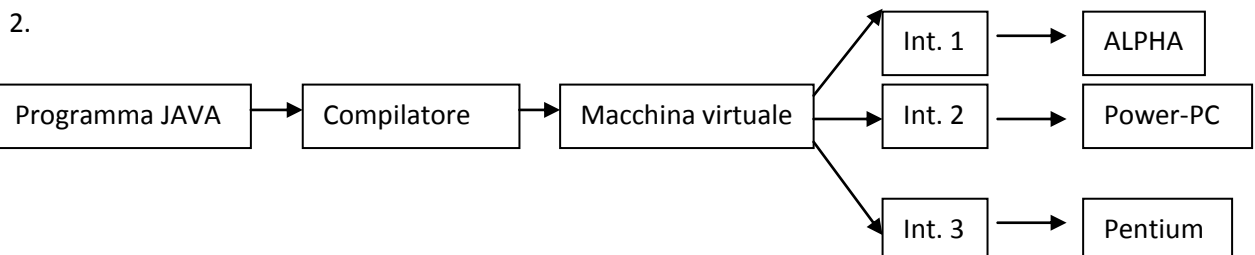
Generalmente il target è un codice eseguibile su macchina:

1. Direttamente eseguibile su macchine reali: assembler per Pentium o per powerPc;
2. Eseguibile su macchine virtuali (che non esistono): es. Java (il codice sarà poi interpretato);

In sostanza abbiamo questi due modelli:

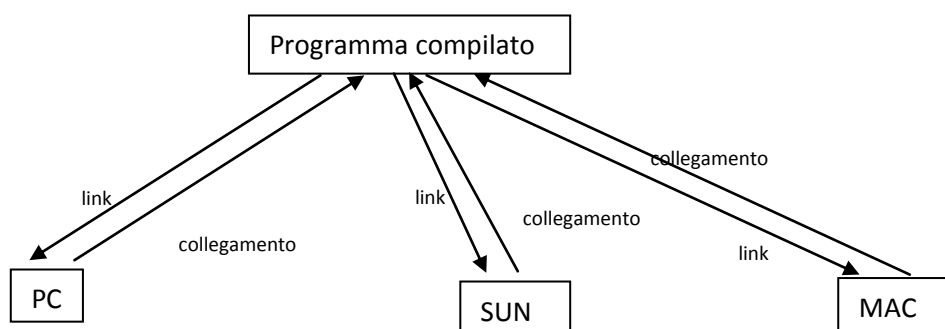


Tale modello è più efficiente ma meno portabile (quello che interessa ad oggi è però la portabilità);



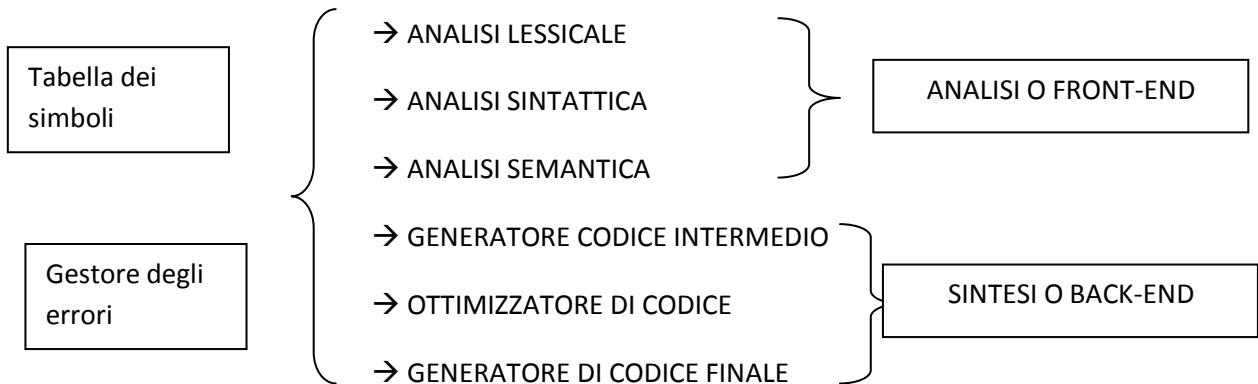
Tale modello è meno efficiente ma più portabile (compilatore multiplatforma)

L'obiettivo è creare un compilatore che genera un codice che sia eseguibile su tutte le macchine. Java è multiplatforma e nasce per la comunicazione su reti. Ci si può collegare con qualsiasi computer e tale computer può mandare codice da eseguire sulla nostra macchina:



Prima era necessario avere più versioni dello stesso programma compilato una per ogni possibile architettura collegabile. Ora invece il programma è compilato solo in Java, saranno poi le varie architetture ad avere i particolari interpreti. Il compilato Java è abbastanza a basso livello quindi di facile interpretazione. Il maggiore problema attuale è la comunicazione dei dati. Nella comunicazione tra computer vengono mandati file, chi li riceve li elabora, e poi li manda in esecuzione.

Abbiamo visto le varie fasi di compilazione:



La Tabella dei simboli è una memoria in cui memorizzare informazioni che saranno utili alle fasi successive; alla fine il generatore di codice ci troverà gli indirizzi di tutti le variabili (per ogni funzione di che tipo, quanti sono gli argomenti, etc...). Ogni fase avrà un gestore degli errori.

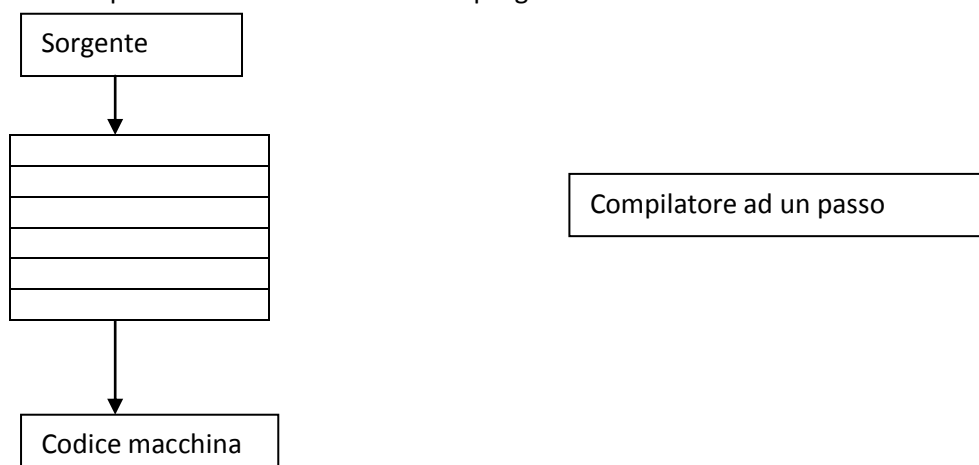
La fase di **ANALISI (front-end)** ed ha come input il codice sorgente) è legata solo al linguaggio di partenza; si cerca di capire la struttura interna del programma, tutte le sue componenti e lo si riporta in un formato più agevole: il codice intermedio.

La fase di **SINTESI (back-end)** ed ha come input la tavola dei simboli) è legata solo all'architettura del processore finale, al linguaggio d'arrivo; si mettono insieme tutte le informazioni ottenute dall'ANALISI per ottenere il codice macchina finale (fase di ottimizzazione).

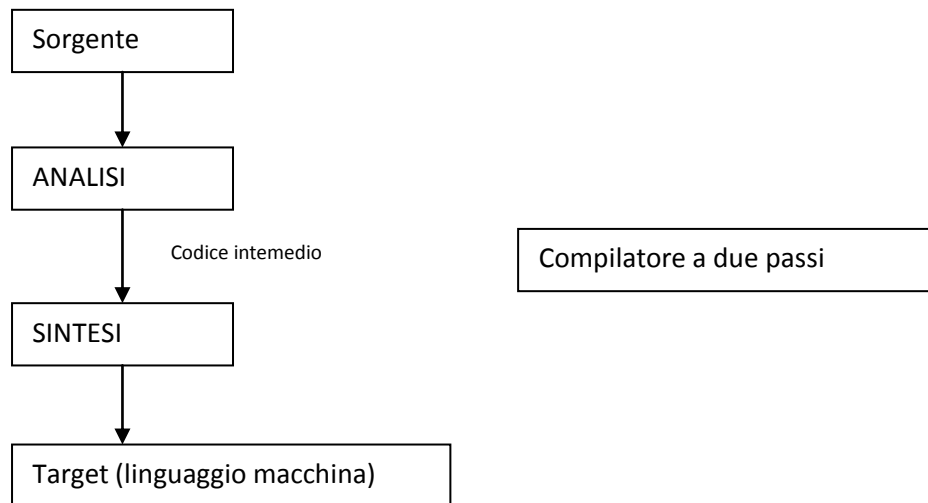
È possibile, per un linguaggio, avere un unico front-end e poi diversi back-end per le macchine diverse (si avrà sempre un unico codice intermedio); si minimizza così la costruzione del compilatore.

Abbiamo diviso la compilazione in più fasi, ma non è detto che ciascuna fase sia implementata da un diverso modulo software (cioè per es. avere sei moduli collegati tra di loro); la suddivisione vista è solo logica. Vi possono essere più soluzioni:

- Un compilatore scritto come un unico programma al cui interno vi sono tutte le fasi:



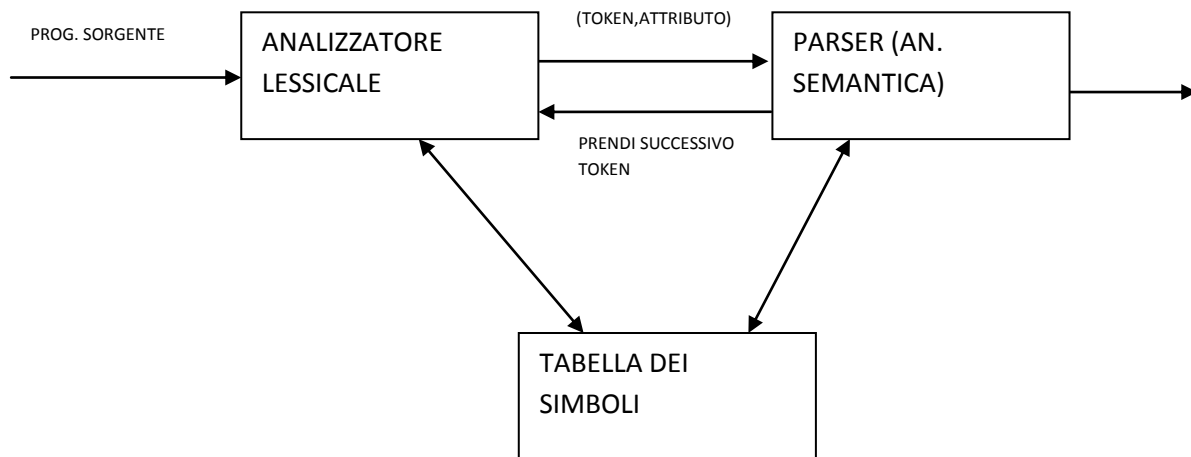
- Un compilatore scritto come due moduli software che interagiscono tra di loro:



LEZIONE 3 – ANALISI LESSICALE (1 PARTE)

09/03/2009

L'analizzatore lessicale è la prima fase logica di un compilatore, il suo principale compito è leggere i caratteri in input del codice sorgente, raggrupparli in lessemi e produrre come output una sequenza di token per ogni lessema nel codice sorgente e che successivamente il parser usa per l'analisi sintattica:



Poiché l'analizzatore lessicale è la parte del compilatore che legge il testo sorgente, potrebbe anche eseguire compiti secondari: estrapolare dal programma sorgente commenti e spazi nella forma di blank, tab e caratteri di newline; tenere traccia del numero di caratteri newline, cosicché un numero di linea può essere associato ad un messaggio di errore. In alcuni compilatori, l'analizzatore lessicale, fa una copia del programma sorgente con i messaggi di errore marcati. Lo stream di token è inviato al parser per l'analisi sintattica. Quando l'analizzatore lessicale incontra un lessema che costituisce un identificatore, esso ha bisogno di inserirlo in una tabella dei simboli. In alcuni casi l'informazione riguardante il tipo di identificatore può essere letto dalla tabella dei simboli attraverso l'analizzatore lessicale per aiutarla nella determinazione dell'opportuno token da passare al parser.

Di solito gli analizzatori lessicali sono divisi in due fasi logiche in cascata:

1. Lo scanning;
2. L'analisi lessicale;

È conveniente separare l'analisi lessicale dall'analisi sintattica in quanto:

- Si facilitano entrambe le fasi;
- Aumenta la portabilità del compilatore;
- Aumenta la leggibilità del codice;
- Uno stesso analizzatore lessicale può essere usato per più linguaggi.

Sono stati progettati molti strumenti per aiutare la costruzione di analizzatori lessicali e parser quando sono programmi separati; in genere potrebbero essere due funzioni dello stesso programma. Vi sono tre approcci generali per l'implementazione un analizzatore lessicale:

1. Usare un generatore di analizzatore lessicale, come il LEX, per produrre un analizzatore lessicale attraverso una descrizione basata sulle espressioni regolari.
2. Scrivere un analizzatore lessicale in un linguaggio di programmazione qualsiasi, usando le funzioni di I/O del linguaggio per leggere l'input (il programma sorgente).

3. Scrivere l'analizzatore lessicale in assembler e gestire esplicitamente la lettura dell'input (programma sorgente).

Le tre scelte sono elencate in ordine di difficoltà crescente per l'implementazione.

Poiché una grossa quantità di tempo può essere consumata per spostare i caratteri di input, molte tecniche di buffering sono state sviluppate per ridurre i tempi. Tra queste consideriamo quella che usa un buffer diviso in due unità contenenti N caratteri ciascuna (si riempie un'unità alla volta). In genere N è il numero di caratteri contenuti in un blocco del disco. Leggiamo N caratteri di input in ogni metà del buffer con un solo comando di lettura da sistema piuttosto che invocando un simile comando per ogni singolo carattere. Alla fine restano da leggere meno di N caratteri (dal disco); dopo tali caratteri nel buffer viene letto un carattere speciale "eof" che marca la fine del file; sarà un carattere diverso da un qualsiasi altro carattere in input. Vi saranno due puntatori al buffer e la stringa di caratteri tra tali puntatori sarà il lessema corrente. Inizialmente, entrambi i puntatori puntano al primo carattere del successivo lessema che è stato trovato. Uno, chiamato il **puntatore in avanti**, esamina fino a quando non trova un match per un pattern (finisce la parola, il lessema, che si sta leggendo). Una volta che il lessema è determinato (racchiuso tra due puntatori), il puntatore in avanti è collocato sul carattere alla sua destra. Dopo che il lessema letto è stato elaborato, entrambi i puntatori sono collocati sul carattere che segue il lessema. Con tale schema, commenti e spazi possono essere trattati come pattern che non producono token (quindi che non vengono presi in considerazione). Se il puntatore in avanti raggiunge il marcatore finale della prima metà del buffer, viene riempita la seconda parte con N nuovi caratteri input. Se il puntatore in avanti raggiunge il marcatore finale della seconda metà del buffer, viene riempito di N nuovi caratteri la prima metà e il puntatore in avanti è riportato all'inizio del buffer (viene gestito come un array circolare). Tale schema di buffering lavora abbastanza bene per la maggior parte del tempo, ma essendo la lunghezza del buffer limitata, ciò può rendere impossibile il riconoscimento dei token in situazioni in cui la distanza che il puntatore in avanti deve attraversare è maggiore della lunghezza del buffer.

L'analisi lessicale individua le unità lessicali di un linguaggio di programmazione (es. Pascal):

1. PAROLE CHIAVE: es. var, for, then, while, function, procedure, switch, true, false etc..
2. OPERATORI: es. +, -, *, **, >, <, AND, OR etc..
3. IDENTIFICATORI: sono tutti i nomi usati nel programma che non sono parole chiave e non iniziano con un numero: variabili, funzioni, procedure.
4. COSTANTI NUMERICHE: 5, 5.3, 5.3E-4;
5. SEPARATORI: sono simboli di punteggiatura: es. , ; : () spazi (da riconoscere ma non da usare) NL (a capo) etc...
6. STRINGHE COSTANTI: tutto ciò che è tra virgolette " " e ci saranno costanti numeriche o letterali. La stringa ha taglia variabile, in memoria sarà rappresentata da una sequenza di byte che finisce con un carattere particolare di fine stringa. Quindi se la stringa è di n caratteri, in memoria se ne devono prevedere n+1 (carattere di fine stringa).
7. CARATTERE COSTANTE: singoli caratteri tra apici ' A '. è un solo carattere e vi si assegna un solo byte di memoria. Quindi la rappresentazione di stringhe e caratteri in memoria è divisa.

Ogni linguaggio può avere diverse caratteristiche in base alle particolari esigenze:

- RISERVARE: alcune stringhe il cui significato è predefinito e non può essere cambiato dall'utente: le parole chiavi (funzioni, proprietà del linguaggio etc..). Ciò significa che uno stesso nome non può essere dichiarato come due tipi diversi (creerebbe problemi nell'assegnazione della memoria).

- Considerare gli spazi come separatori di parole: non è possibile mettere spazi nelle unità lessicali diventerebbero due unità lessicali.

L'implementazione del linguaggio deve avere specifiche formali di come sono fatte le unità lessicali, e indica come un programma deve essere riscritto per rispettare le specifiche date. Prima abbiamo diviso tutte le possibili sequenze di caratteri in un programma in classi di caratteri, l'analisi lessicale traduce il programma sorgente in categorie lessicali.

Agli elementi di ogni classe è associato un token:

CLASSI	TOKEN (nomi associati alle classi)
PAROLE CHIAVI	Ogni parola chiave la chiamiamo col suo stesso nome
OPERATORI <ul style="list-style-type: none"> - ASSEGNAZIONE - ARITMETICO - LOGICO 	<ul style="list-style-type: none"> - assop - arop - logop
COSTANTI NUMERICHE	Num
SEPARATORI	Ogni simbolo di punteggiatura avrà un suo nome, gli spazi li chiamiamo sep, ma non si considerano
STRINGHE COSTANTI	Iconst
CARATTERI COSTANTI	Cconst
IDENTIFICATORI	id

Es. Quindi l'analisi lessicale traduce:

somma:= 5+4+sommaz in

id assop num arop num arop id (sequenza di token).

Tale traduzione porta una perdita di informazioni poiché si tiene conto solo della classe a cui appartiene la sequenza letta ma si perde il lessema (sequenza di caratteri che formano fisicamente la parola) di ciò che si è letto. Le informazioni riportate sono sufficienti per l'analisi sintattica (soggetto-verbo) ma non per l'analisi semantica, per la quale serve sapere i tipi. Quindi l'analisi lessicale, oltre ad assegnare ogni parola ad una classe, deve tenere memoria nella **tabella dei simboli** del lessema della parola letta (l'unica informazione nota a questa fase, non sappiamo ancora il tipo). Quindi l'analisi lessicale restituirà al parser la coppia (token, attributo) dove l'attributo è un puntatore alla tabella dei simboli, punterà alla locazione dove è memorizzato il lessema del token. La tabella dei simboli sarà arricchita di informazioni (tipi, etc..) durante le varie fasi della compilazione, informazioni che saranno utilizzate per la generazione del codice finale.

Poiché per le costanti numeriche non si prevede l'aggiunta di altre informazioni, l'analisi lessicale può restituire direttamente la coppia (num, 5), (num, 4); lo stesso non si può fare per gli identificatori in quanto necessitano di ulteriori informazioni (es. l'analisi semantica aggiungerà il tipo nella tabella dei simboli, nella generazione del codice si metterà l'indirizzo di memoria in cui inizia e quante locazioni occuperà).

Di solito il controllo (il cervello) è l'analisi sintattica (che lavora solo sui token, la grammatica è scritta sui token). L'analizzatore sintattico richiede all'analisi lessicale il prossimo token il quale legge il sorgente e individua il token, inserisce il lessema nella tabella dei simboli, se non è già presente, ottiene il puntatore all'entrata nella tabella e restituisce la coppia (token, attributo). Il parser a seconda del token sa se l'attributo è un puntatore o una costante intera.

Abbiamo già detto che l'analisi si divide in due fasi logiche:

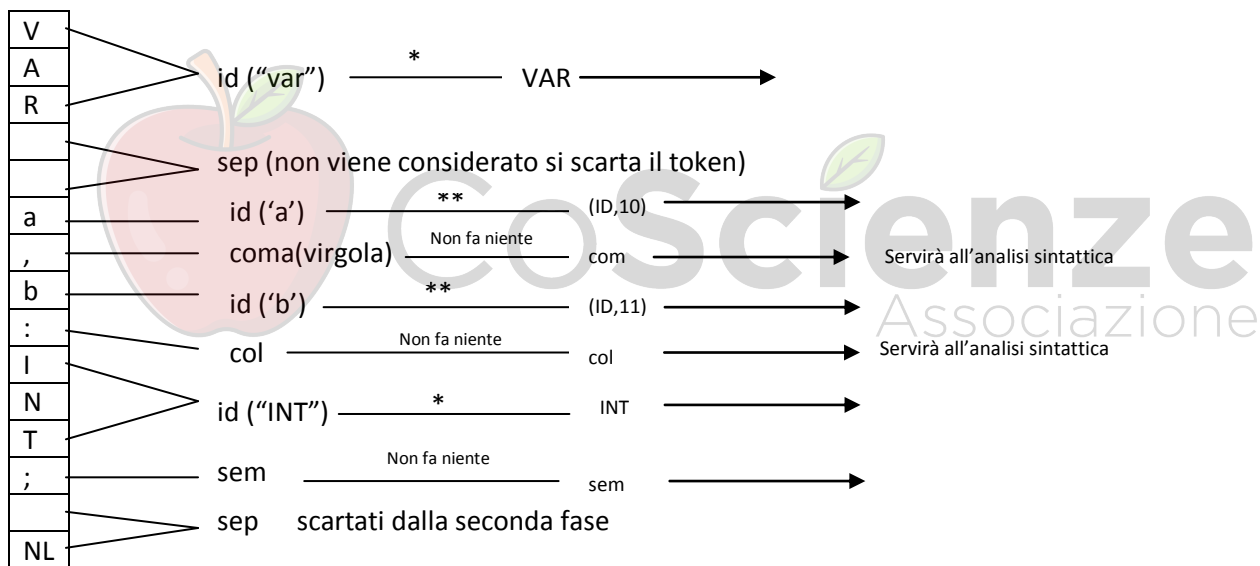
I FASE: riconoscere parole e associarle a token, riconosce fisicamente le parole. Tale fase può essere fatta automaticamente (lex) utilizzando automi finiti.

II FASE: creazione dell'input alla fase successiva, quindi si accede alla tabella dei simboli.

Per gestire la I FASE vi sono due tecniche:

I TECNICA: si usano gli automi finiti per riconoscere l'appartenenza della parola ad una delle classi; ci sarà un automa per ogni parola chiave, si capisce che si avrà un automa con troppi stati quindi si preferisce la 2°.

II TECNICA: tutte le parole chiave sono inserite nella tabella dei simboli, quindi visitando la tabella di simboli si vede se è una parola chiave.



* si vede se il lessema VAR fa parte delle parole chiavi, leggendo la tabella dei simboli, la trova quindi restituisce il token VAR.

TABELLA DEI SIMBOLI

Riempita inizialmente con tutte le parole chiave

10 11	VAR	Parola chiave
	begin	//
	INT	//
	Successivamente si inseriranno altre informazioni	
	a	
	b	

****** il lessema a non è trovato nella tabella dei simboli e viene inserito.

Così in ogni linea di programma è tradotta in una sequenza di informazioni logiche e l'esempio precedente diventa: VAR (ID,10) com (ID,11) col INT sem etc... Com, col, sem, ... sono nomi che dipendono dalla grammatica.



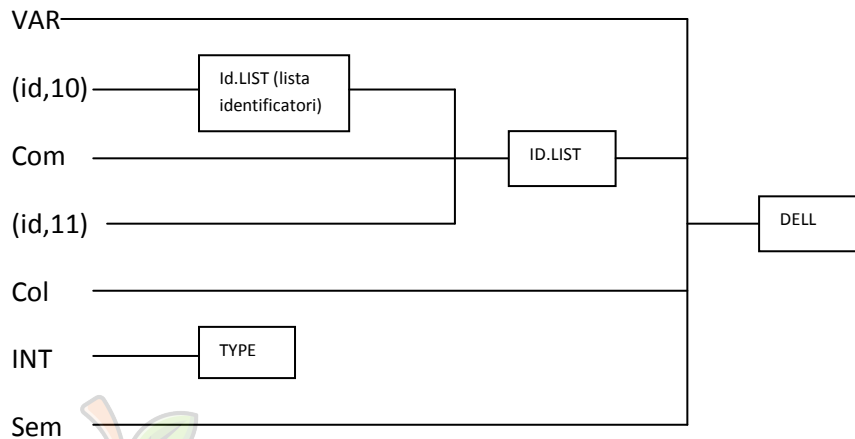
LEZIONE 4 – ANALISI LESSICALE (2 PARTE)

09/03/2009

Una grammatica determina delle regole. Ad esempio in italiano una regola sintattica è soggetto - verbo- compl. oggetto. Nel nostro caso una regola sintattica presente nell'analizzatore sintattico è:

- VAR seguito da una sequenza di identificatori divisi da virgole (coma) che termina con due punti (col). Si ha poi un tipo e un punto e virgola (semi-col).

Regola sintattica (dichiarazione):



Leggendola in questo senso si ha che una dichiarazione è formata da:

- Un VAR;
- Lista di identificatori;
- Col (:);
- Tipo;
- Sem (;);

A sua volta Id.LIST è formato da:

- ID.LIST
- Com(,)
- ID.

L'analisi semantica farà una visita dell'albero e scoprirà che gli elementi nelle locazioni 10 e 11 della tabella dei simboli sono INTEGER e potrà quindi aggiungere tale informazione nella tabella dei simboli.

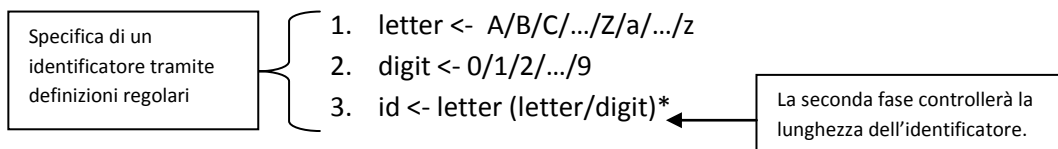
Convien usare la tecnica che inserisce le parole chiavi nella tabella dei simboli:

- È facile aggiungere una nuova parola chiave, senza toccare le specifiche del linguaggio.
- Altrimenti avremo molti stati.

COME DESCRIVERE I TOKEN DI UN LINGUAGGIO????

Usiamo le espressioni regolari. Se nella seconda fase utilizziamo la seconda tecnica, nella prima fase gli identificatori sono visti come le parole chiavi.

PAROLE CHIAVE e IDENTIFICATORI: è l'insieme delle stringhe alfanumeriche che iniziano con una lettera (per convenzione). Tale insieme è definito dalla seguenti definizioni regolari:



COSTANTI NUMERICHE: es. 5, 5.4, 5.4E-3 definiamo tale classe tramite definizioni regolari

1. digit <- 0/1/.../9
2. digits <- digit digit*
3. optional-fraction <- .digits/e
4. optional-exponent <- (E(+/-/e)digits)/e
5. num <- digits optional-fraction optional-exponent.

Alcuni costrutti si verificano frequentemente per cui, per essi, è conveniente introdurre delle abbreviazioni:

- l'operatore unario postfixo $^+$ significa una o più istanze di ...: $\Omega^* = \Omega^+ / \epsilon$, $\Omega^+ = \Omega M^*$. Ad esempio $1^* = \{\epsilon, 1, 11, 111 \dots\}$ $1^+ = \{1, 11, 111 \dots\}$;
- l'operatore unario postfixo $?$ significa (opzionale) nessuna o una istanza di ...: $\Omega? = \Omega / \epsilon$ (quindi in pratica significa 0 oppure 1). Se Ω è una espressione regolare allora $(\Omega)?$ è una espressione regolare che denota il linguaggio $L(\Omega) \cup \{\epsilon\}$;
- la notazione $[abc]$ dove $a, b, c \in \Sigma$, denota l'espressione regolare $a/b/c$. $[a-z]$ denota l'espressione regolare $a/b/.../z$
- le parentesi ed i simboli $- ? + / \dots$ sono chiamati 'metasimboli' e sono utilizzati per formare espressioni regolari; quando si vuole usare un meta simbolo come un simbolo usiamo $" "$. ad esempio $"("$;

Non tutti i linguaggi possono essere descritti con un'espressione regolare. Ad esempio, le espressioni regolari non possono essere usate per descrivere costrutti bilanciati o annidati, come l'insieme di tutte le stringhe di parentesi bilanciate che invece è descritto da una grammatica C.F. Le espressioni regolare possono essere usate solo per denotare un numero fissato di ripetizioni o un numero non specificato di ripetizioni di un dato costrutto. Ad esempio $a^n b^n$ non è regolare in quanto gli automi finiti non sanno ricordare il numero di elementi letti.

Quindi possiamo ridefinire:

digit $\rightarrow [0-9]$; letter $\rightarrow [a-z, A-Z]$;

Andiamo quindi a riprendere le grammatiche C.F. Una grammatica C.F. ha 4 componenti:

1. T \rightarrow insieme dei simboli terminali
2. N \rightarrow insieme dei simboli non terminali
3. P \rightarrow produzioni
4. S \rightarrow starting symbol

Ad esempio consideriamo questa definizione:

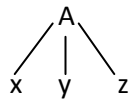
1. list \rightarrow list+digit
2. list \rightarrow list-digit
3. list \rightarrow digit
4. digit $\rightarrow 0/1/2/.../9$

L'espressione 9-5+2 è così nota: 9 è un list; 9-5 è un list; 9-5+2 è un list.

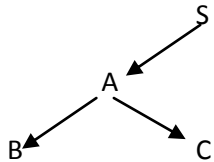
Questo è un esempio di derivazione; da uno starting symbol abbiamo generato un insieme di parole.

Metodo parse-tree:

$A \rightarrow xyz$ (produzione) oppure

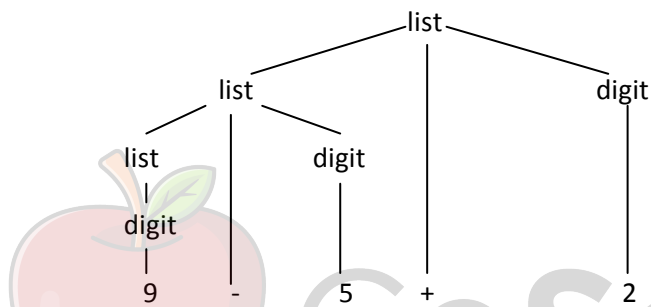


Quindi la rappresentazione della produzione $A \rightarrow BC$ diventa:

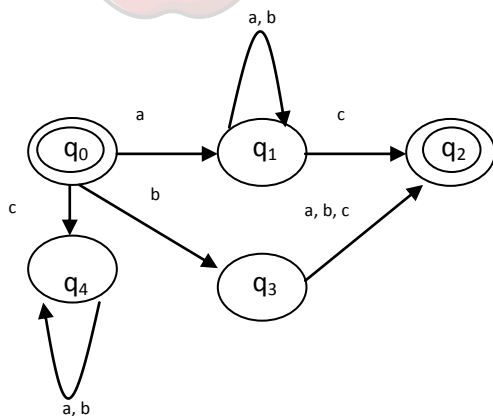


I nodi interni sono simboli non terminali mentre le foglie sono simboli terminali.

Ritornando all'esempio precedente (9-5+2) avremo questo albero:



L'automa $A = \langle Q, q_0, \delta, F, \Sigma \rangle$



Dato $\Sigma = \{a, b, c\}$ l'automa identifica: $\{\epsilon\}, \{a\} \cup \{a, b\}^* \cup \{c\} \cup \{a, b, c\}^* \dots \cup \{b\} \cup \Sigma^*$.

Abbiamo detto che un token può essere descritto tramite un insieme di definizioni regolari, vediamo ora come riconoscere i token, cioè, date le specifiche dei token tramite le espressioni regolari, come scrivere un programma che prende in input un programma sorgente e restituisce i token.

LEZIONE 5 – ANALISI LESSICALE (3 PARTE)

11/03/2009

Ci sono due metodi per passare dalla fase logica all'analizzatore lessicale:

1. METODO MANUALE: saremo noi a scrivere un programma che prende in input le definizioni regolari e restituisce un programma che è un analizzatore lessicale.
DEFINIZIONI REGOLARI \longrightarrow ANALIZZATORE LESSICALE
2. METODO AUTOMATICO: DEFINIZIONI REGOLARI \longrightarrow ANALIZZATORE LESSICALE

Date le definizioni regolari dei token le dobbiamo tradurre in un programma ottenendo così un analizzatore lessicale; nel secondo metodo tale traduzione sarà fatta automaticamente da LEX.

Consideriamo il **primo metodo** quello dove noi ci occupiamo di scrivere tutto. Consideriamo un esempio. Partendo da un testo (sorgente):

```
if a>1 then statment  
  
    else stament
```

A questo punto dobbiamo implementare il linguaggio, quindi abbiamo le specifiche del linguaggio in linguaggio naturale e dobbiamo fornire sia l'analisi lessicale che sintattica per le istruzioni descritte. Come prima cosa si scrive una grammatica:

if statment \rightarrow IF *espressione* THEN *statment*

IF *espressione* THEN *statment* ELSE *statment*

espressione \rightarrow *termine* REL.OP. *termine*

termine \rightarrow ID|NUMBER

IF, THEN, ELSE sono tutti terminali;
REL.OP = operatore relazionale;
Statment = non terminale;

Iniziamo così a distinguere i token, le unità logiche del linguaggio (le unità lessicali), e la struttura logica, sintattica, del costrutto letto. Quindi l'identificazione dei token e la struttura sintattica viene fatta insieme (tutto grazie alla grammatica poiché dà anche una descrizione logica ad alto livello del programma).

IF, THEN, ELSE, REL.OP, ID, NUMBER sono i token, le unità logiche, gli elementi terminali della grammatica. Una grammatica alternativa avrebbe previsto queste definizioni:

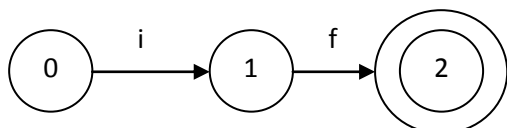
espressione \rightarrow termine \geq termine / termine \leq termine/ ...

in questo modo si sarebbe complicata la grammatica (presenti più token), ma si semplificherebbe l'analisi lessicale (che restituisce direttamente $<$, $>$, \leq , \geq , ...). Quindi scrivendo la grammatica si definiscono i token (le unità logiche). Bisogna ora descrivere i token: per ogni token bisogna descrivere qual è la classe dei lessemi che appartiene a quella unità logica. Le unità logiche (i token) sono descritte attraverso le espressioni regolari.

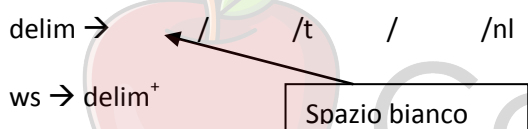
Consideriamo la I tecnica (quella manuale):

TOKEN	ESPRESSIONE REGOLARE
IF	if
THEN	then
ELSE	else
REL.OP.	</>/<=/>=</>/=
ID	letter(letter/digit)*
NUMBER	digit*(.digit+)?(E(+/-)?digit+)?

Per comprendere i lessemi usiamo gli automi che ci forniscono un modo per ottenere i vari lessemi. Ad esempio if è ottenuto grazie all'automa:



L'analisi lessicale deve restituire l'entità logiche all'analisi sintattica. Usando tali specifiche per i token, l'analizzatore lessicale, per ogni lessema che legge deve decidere a quale classe appartiene e quindi ricava l'unità logica da dare all'analisi sintattica. Per semplicità supponiamo che le parole chiave siano riservate. L'analizzatore lessicale riconoscerà le parole chiave if, then, else ed i lessemi che appartengono alle unità logiche relop, id, number. Assumiamo che i lessemi siano separati da spazi, costituiti da sequenze non nulle di blank (), tab (/t), newline (/nl). L'analizzatore lessicale dovrà essere in grado di riconoscere gli spazi, quindi si avranno anche le espressioni regolari per questi "caratteri" particolari:



Se viene trovato un match per ws, l'analizzatore lessicale non restituisce niente al parser (analizzatore sintattico) ma procede per trovare il token successivo allo spazio, che dovrà essere restituito al parser (per questo motivo non si specifica ws nella grammatica). Quindi è importante costruire un analizzatore lessicale che isoli un lessema dal successivo in modo da poter restituire la corretta copia (token, attributo).

Consideriamo l'operato dell'analizzatore. Cosa accade quando inizia il parsing??? Ad esempio il parsing inizia leggendo la "i" la quale soddisfa due regole (IF e ID) e poi legge la f soddisfacendo la prima espressione (IF). In caso di conflitto (ad es. IF e IFFO) cosa fare??? L'analizzatore lessicale deve rispettare le seguenti regole:

- ✓ **I REGOLA:** bisogna leggere il lessema più lungo fino al prossimo separatore (fine parola).
- ✓ **II REGOLA:** se il lessema soddisfa più regole si prende quella che viene prima.

Per tale motivo sono definite prima le parole chiave; tali regole evitano ambiguità (le regole le poniamo noi).

Quindi bisogna scrivere un programma che rispetta le due regole. La parte di riconoscimento riflette la prima fase logica dell'analisi lessicale; la seconda fase logica è la gestione della tabella dei simboli.

Schematizzando:

Espressioni regolari	Token	Valore attributo
WS	//	//
IF	IF	//(è il token che rappresenta l'informazione)
THEN	THEN	//
ELSE	ELSE	//
ID	ID	Puntatore alla tabella dei simboli oppure il numero (si tiene traccia del lessema per uso futuro)
NUM	NUM	Puntatore alla tabella dei simboli oppure il numero (varia a seconda di come lo realizziamo noi)
<	REL.OP.	LT (oppure puntatore)
>	REL.OP.	GT
<=	REL.OP.	LE
>=	REL.OP.	GT
<>	REL.OP.	NE
=	REL.OP.	EQ

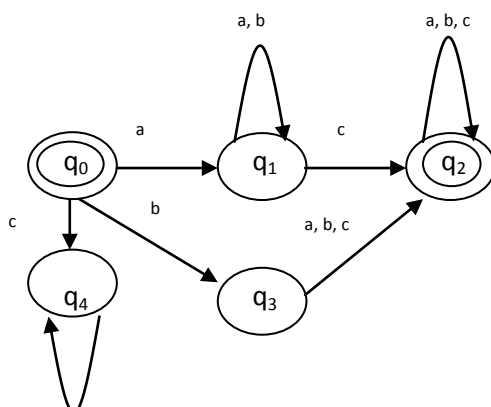
I fase: riconoscimento lessemi e ricerca di unità logiche (token) a cui appartengono. Fase di scanning, di riconoscimento.

II fase: gestione della tabella (riempimento se necessario della tabella). Fase di scrinning più operativa.

Bisogna fare un programma (analizzatore lessicale) che fa tale operazioni che prende in input un programma sorgente ed in base alle specifiche date (grammatica, espressioni regolari) deve fornire le specifiche output descritte (le coppie (token, attributo)). Per scrivere il programma usiamo i diagrammi di transizione (gli automi). Sono usati per la II tecnica: parole chiavi nella tabella dei simboli.

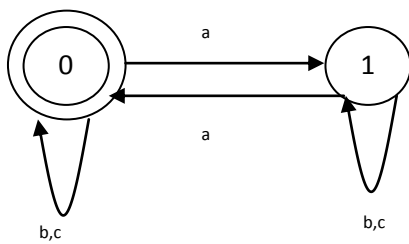
Per ogni espressione regolare si fa un diagramma di transizione.

Ad esempio: L'automa $A = \langle Q, q_0, \delta, F, \Sigma \rangle$

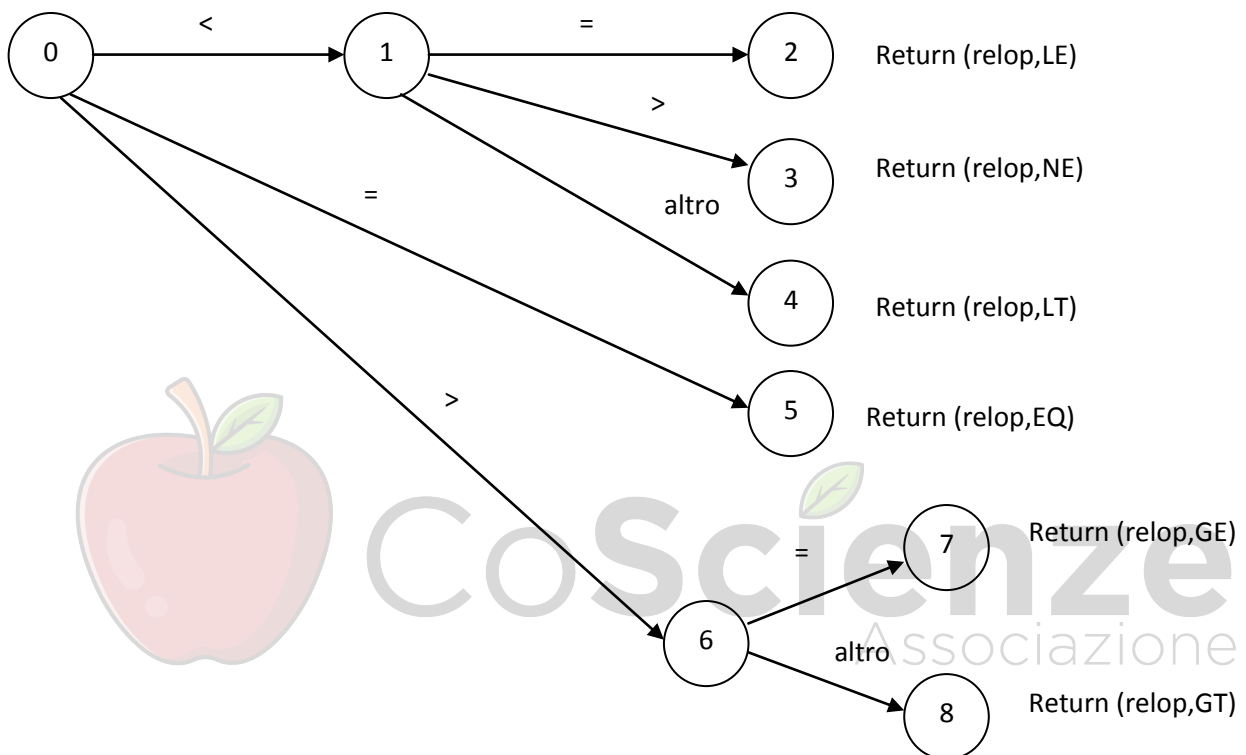


Dato $\Sigma = \{a, b, c\}$ l'automa identifica: $\{e\}, \{a\} \cup \{a, b\}^* \cup \{c\} \cup \{a, b, c\}^* \dots \cup \{b\} \cup \Sigma^*$.

Oppure il linguaggio $L = \{x \in \{a, b, c\}^* \text{ tale che } |x|_a \text{ è pari}\}$ è identificato dall'automa:

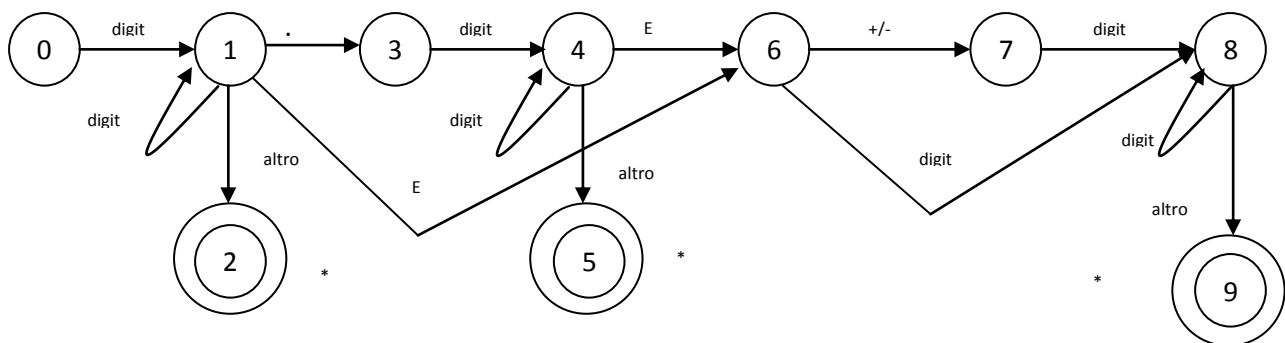


Questo è l'automa da usare per riconoscere i token REL.OP.:

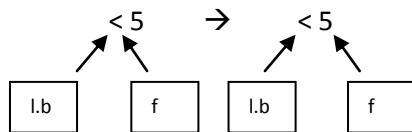


Usando questo metodo si nota come si rispetta la regola di leggere il lessema più lungo: per tale motivo leggendo < o > non si va in uno stato finale, ma si riconosceranno tali simboli se leggiamo altro. L'automa riconosce le espressioni del linguaggio. Infatti non consideriamo che dopo gli operatori ci sia lo spazio. Questa situazione deve essere poi implementata.

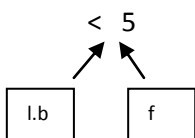
Vediamo un ultimo automa, quello per il riconoscimento delle parole che identificano i numeri:



Lo * sta ad indicare che dobbiamo restituire qualcosa (fare return()). Per riconoscere i vari lessemi usiamo (il programma usa) due puntatori, l.b ed f. Ciò che è racchiuso tra i due è il lessema. Per andare a determinare il prossimo lessema spostato in avanti il primo puntatore, l.b, dopo aver determinato il precedente lessema. Ad esempio: inizialmente l.b ed f puntano entrambi a <.

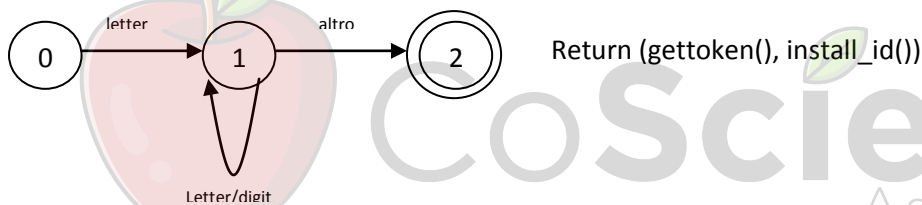


Quando f avanza e punta a 5 si va in uno stato finale, ma 5 non fa parte del lessema riconosciuto e quindi f dovrà tornare indietro. Il lessema riconosciuto, <, è compreso tra i due puntatori. Dopo che il lessema è stato riconosciuto entrambi i puntatori saranno spostati sul primo simbolo dopo quello puntato da f (entrambi puntano nello spazio bianco):



Utilizzando la II tecnica, abbiamo detto, che nella I fase le parole chiavi sono gestite come gli identificatori, poi in un secondo momento si interogherà la tabella dei simboli per sapere se il lessema letto è una parola chiave o effettivamente un identificatore.

Vediamo infine l'automa per il riconoscimento degli ID:



Quindi se si arriva nella stato di accettazione 2 vengono eseguiti i codici di gettoken() e install_id (li dobbiamo scrivere noi) per ottenere il token ed il valore dell'attributo che devono essere restituiti.

- gettoken(): può accedere al buffer per ottenere il lessema letto (è tra i due puntatori) cerca nella tabella dei simboli il lessema:
 - se è una parola chiave viene restituito il corrispondente token;
 - altrimenti è restituito il token id;
- install_id(): accede alla tabella dei simboli ed al buffer dove è stato identificato il lessema. Visita la tabella dei simboli:
 - se il lessema letto è marcato come una parola chiave non restituisce niente (restituisce il puntatore alla parola chiave);
 - se il lessema letto è una variabile del programma (identificatore) restituisce un puntatore (o riferimento) alla tabella dei simboli;
 - se il lessema letto non è nella tabella dei simboli, viene installato come una variabile e viene restituito un puntatore (o riferimento) ad esso.

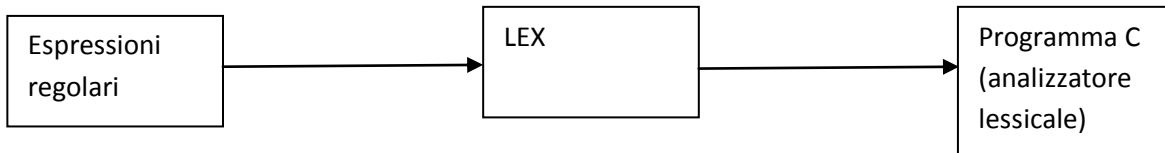
Ci sarà un programma principale che regola l'ispezione dei vari diagrammi. Si ispeziona prima il primo diagramma di transizione se non si arriva in uno stato finale si chiama una routine di fallimento fail() che riposiziona il puntatore in avanti all'inizio del lessema e si ricomincia il riconoscimento dal successivo diagramma (II regola). Se non ci sono altri diagrammi fail() chiama una routine di error-recovery (si è letto

qualcosa che non appartiene al linguaggio). I diagrammi di transizione sono messi nell'ordine giusto (l regola). Da tali diagrammi di transizione possiamo scrivere un programma.

Ora passiamo al **secondo metodo**.

LEX

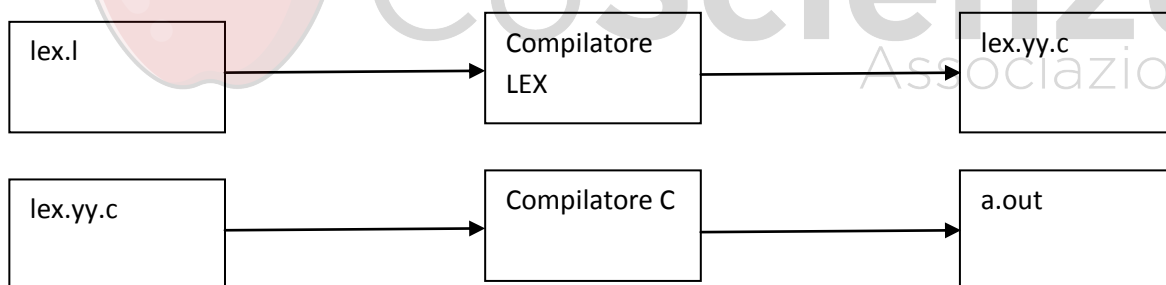
Si può evitare di fare tutto ciò utilizzando LEX. LEX è esso stesso un compilatore in quanto compila il linguaggio delle definizioni regolari:



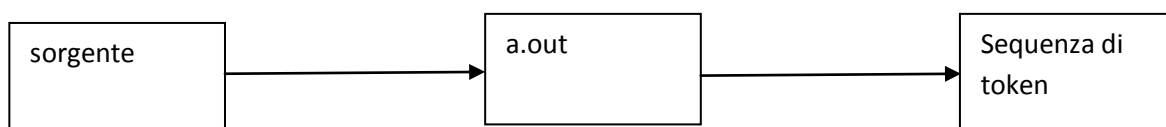
Sarà il LEX a fare tutto il lavoro che nella descrizione precedente dovevamo fare noi. Fa automaticamente la conversione delle espressioni regolari in C.

Il LEX è usato nel seguente modo:

- viene prima preparata una descrizione dell'analizzatore lessicale creando un file `lex.l`.
- poi il file `lex.l` è eseguito dal compilatore LEX per produrre un programma (codice) in C, il `lex.yy.c`; le azioni (operazioni da eseguire) associate alle espressioni regolari in `lex.l` sono pezzi di codice C e sono inviati direttamente al `lex.yy.c`
- il `lex.yy.c` è eseguito dal compilatore C che produce un programma oggetto `a.out` che è l'analizzatore lessicale il quale trasforma il sorgente in una sequenza di token.



L'a.out sarà il nostro analizzatore lessicale:



Il file `lex.l` contiene le definizioni regolari per i token, ed è scritto in un linguaggio ad altissimo livello (linguaggio LEX). L'a.out fa solo le operazioni di riconoscimento, le altre azioni (`gettoken()`, `install_id()`, la definizione e gestione della tabella dei simboli) dobbiamo scriverle noi direttamente in C nel file `lex.l`, dopo la rispettiva definizione regolare (ad ogni definizione bisogna aggiungere un insieme di azioni, cioè le operazioni da fare se si riconosce il token descritto dalla definizione).

Un file .l è diviso in tre parti:

1. dichiarazioni
%%
2. regole di traduzione
%%
3. procedure e funzioni ausiliarie (funzioni di gestione della tabella dei simboli)

1. La sezione dichiarativa include la dichiarazione di variabili, costanti e definizioni regolari; le definizioni regolari sono istruzioni e sono usate come componenti delle espressioni regolari che appaiono nelle regole di traduzione.
2. Le regole di traduzione sono istruzioni della forma:
 P1 {azione1}
 P2 {azione2}

 Pn {azione n}

dove ogni Pi è una espressione regolare e ogni azione è un frammento di programma che descrive il tipo di azione che l'analizzatore lessicale deve fare quando il pattern Pi match con un lessema; in LEX le azioni sono scritte in C.

3. Nella terza sezione saranno implementate (in C) le procedure ausiliarie utilizzate nelle azioni (es. `gettoken()`, `install_id()`). Tali procedure potrebbero essere implementate separatamente.

Vediamo come un analizzatore lessicale creato col LEX interagisce col parser:

attivato il parser, l'analizzatore lessicale inizia a leggere l'input, un carattere alla volta, finché trova il più lungo prefisso dell'input che matcha con una delle espressioni regolari Pi poi si esegue l'azione i. L'analizzatore lessicale restituisce una sola informazione, il token, al parser; per restituire anche un attributo bisogna settare una variabile globale detta `yylval`. Alla fine il compilato C sarà una funzione `yylex()` per tale motivo restituisce una sola informazione.

Vi saranno altre due variabili globali:

- `yytext`: puntatore all'inizio del lessema;
- `yyleng`: lunghezza del lessema;

Per concludere vediamo un esempio:

```
% { //definizioni di costanti: IF, THEN, ELSE ...
```

```
    #define IF      100;
```

```
    #define THEN    101;
```

```
    ....
```

```
% }
```

Ad ogni token è associato un intero. Diventano unità logiche.

```

/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}       {/* no action and no return */}
if         {return(IF);}
then       {return(THEN);}
else       {return(ELSE);}
{id}       {yyval = (int) installID(); return(ID);}
{number}   {yyval = (int) installNum(); return(NUMBER);}
"<"       {yyval = LT; return(RELOP);}
"<="      {yyval = LE; return(RELOP);}
"="        {yyval = EQ; return(RELOP);}
">"       {yyval = NE; return(RELOP);}
">"       {yyval = GT; return(RELOP);}
">="      {yyval = GE; return(RELOP);}

%%

int installID() {/* function to install the lexeme, whose
                  first character is pointed to by yytext,
                  and whose length is yyleng, into the
                  symbol table and return a pointer
                  thereto */
}

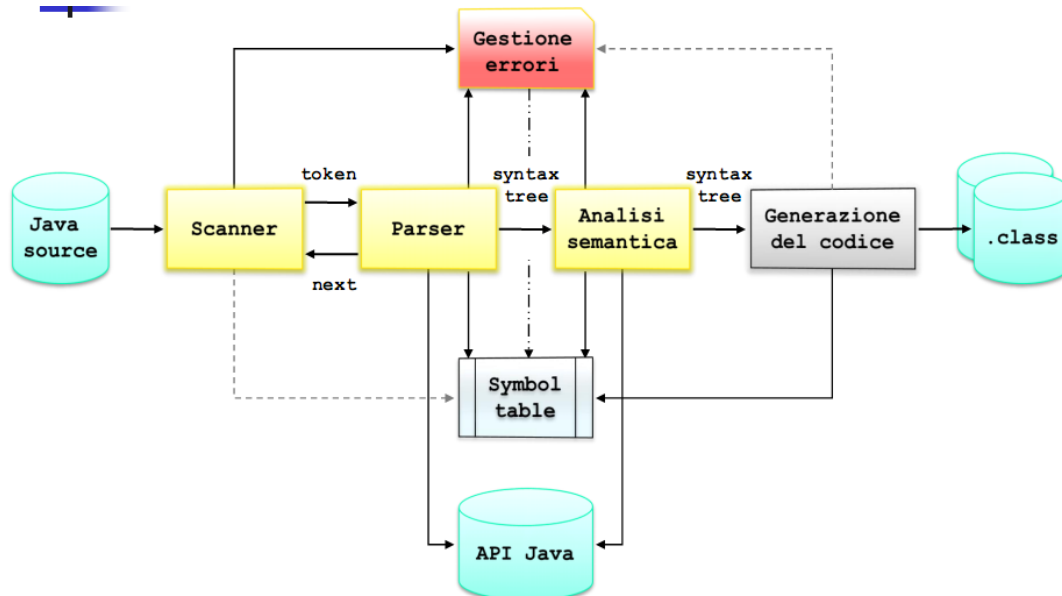
```

Tali implementazioni saranno direttamente riportate nel `lex.yy.c` (senza modifiche). Il programma ottenuto col LEX rispetta le due regole.

Tutto ciò che appare tra le parentesi `%{ ... }` è copiato direttamente, senza modifiche nell'analizzatore lessicale `lex.yy.c`; non è trattato come parte delle definizioni regolari o delle regole di traduzione. Lo stesso avviene per le procedure ausiliarie della terza sezione.

JFLEX

JFlex è un generatore di scanner per gli analizzatori lessicali. L'obiettivo del progetto di JFlex: Realizzazione di un compilatore Java (funzionalità minime), capace di tradurre un file di testo in linguaggio Java in formato bytecode (.class) interpretabile da una qualsiasi Java Virtual Machine.



I programmi sorgenti possono essere scritti utilizzando l'insieme dei caratteri Unicode. Gli Unicode escape vengono tradotti in uno stream di caratteri Unicode. Un Unicode escape è della forma `\uxxxx`, dove `xxxx` è un valore espresso in esadecimale.

Le sequenze di escape permettono di rappresentare i caratteri non-grafici presenti nelle stringhe e nei caratteri.

EscapeSequence:

```
\ b      /* \u0008: backspace BS */
\ t      /* \u0009: horizontal tab HT */
\ n      /* \u000a: linefeed LF */
\ f      /* \u000c: form feed FF */
\ r      /* \u000d: carriage return CR */
\ "      /* \u0022: double quote " */
\ '      /* \u0027: single quote ' */
\ \      /* \u005c: backslash \ */
OctalEscape /* \u0000 to \u00ff: from octal value */
```

OctalEscape:

```
\ OctalDigit
\ OctalDigit OctalDigit
\ ZeroToThree OctalDigit OctalDigit
```

OctalDigit: one of

```
0 1 2 3 4 5 6 7
```

ZeroToThree: one of

```
0 1 2 3
```

Jflex accetta espressioni regolari e permette di definire azioni associate ad ogni espressione regolare. È uno strumento per la generazione automatica di scanner a partire da un data specifica lex (file ".lex"). Lo scanner generato è usato decomporre uno stream dato in input in tante unità significative chiamate "token".

User code

%%

Options and Declarations

%%

Lexical rules

In qualsiasi parte della specifica sono accettati commenti della forma `/* comment text */` anche innestati e commenti che iniziano con `//`.

User Code

Il codice in questa sezione viene copiato senza modifiche all'inizio del file sorgente dello scanner generato, prima della dichiarazione della classe scanner. Questo è il punto per inserire la dichiarazione package e le import.

Options and declarations

In questa sezione è possibile inserire:

1. Le direttive per personalizzare lo scanner da generare.
2. Le dichiarazioni degli stati lessicali.
3. Le definizioni delle macro.

Ogni direttiva è posizionata all'inizio di ogni linea ed inizia con il carattere `%`: `%class "classname"`.

Scanning method

È possibile ridefinire il nome ed il tipo di ritorno e le eccezioni che possono essere lanciate dal metodo richiamato per trovare il token successivo.

- ✓ `%function "name"`: modifica il nome del metodo usato per la scansione, per default viene usato il nome `"yylex"`. Se si integra con Java Cup il metodo sarà `"next_token"`.
- ✓ `%int`: Il metodo usato per la scansione ritornerà valori di tipo `int`. Il valore di default ritornato in caso di end-of-file è la costante `YYEOF` (campo statico e pubblico nella classe generata).
- ✓ `%type "typename"`: Lo scanner ritorna un oggetto del tipo specificato. L'end-of-file comporta il ritorno del valore `null`. Utile quando ritorneremo un oggetto con tutte le informazioni lessicali al Parser: Lessema, linea, colonna, numero caratteri letti ...
- ✓ `%yylexthrow "exception 1" [, "exception 2", ...]`: Le eccezioni verranno dichiarate nella clausola `throws` del metodo usato per la scansione.

Macro definitions

La definizione di una macro ha la forma: `macroidentifier = regular expression`. Le macro sono utilizzate nella sezione delle regole lessicali: `{macroidentifier}`.

Pattern

- ✚ `|` : fa il match con la precedente o la successiva espressione: `r | s -> {r, s}`; `(ab | cd) -> {ab, cd}`
- ✚ `[]` : definisce un insieme (classe) di caratteri: `[0-9] -> {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}`; `[A-Z] -> {A,...,Z}`
- ✚ `[^]` : definisce una classe con elementi negati: `^[A-Z] -> {Tutti i caratteri tranne quelli maiuscoli}`; `^[A-Z\n] -> {Lo stesso di prima ma + il carattere \n}`
- ✚ `()` : raggruppa un'espressione regolare in una nuova espressione (serve a dare priorità)
- ✚ `rs` : Concatenazione di due espressioni: `[0-9][a-z] -> {0a, 0b, ... 8f, ...}`
- ✚ `r/s` : Prima espressione se e solo se seguita dalla seconda (s è un'espressione di controllo)
- ✚ `~r` : fa il match di tutto fino alla prima occorrenza di r
- ✚ `^r` : fa il match se e solo se r è all'inizio di una linea
- ✚ `r$` : fa il match se e solo se r è alla fine di una linea
- ✚ `<<EOF>>` : fa il match con il carattere di end-of-file

- ✚ "...": fa il match letterale del testo contenuto tra i doppi apici: "[abc]\\"foo" -> {[abc]"foo"} //
- Precisamente!!!
- ✚ {}: fa il match con un numero preciso di occorrenze dell'espressione: r{2,5} -> {rr, rrr, rrrr, rrrrr }
r{2,} -> { rr, rrr, ...}, r{4} -> { rrrr }
- ✚ '\x': Caratteri speciali di "escape"
 - \n (newline)
 - \t (tab)
 - \r (ritorno inizio linea)
 - \b (backspace)
 - \u (unicode)
 - \ (slash)

JFlex sempre fa il match con il token più lungo (numero di caratteri). Se due token hanno lo stesso numero di caratteri allora è scelto il token dell'espressione che viene definita per prima nel file lex.

Esempio

Dare una specifica in formato JFlex per il riconoscimento dei commenti in un file sorgente:

I commenti possono essere di 2 tipi, il primo inizia con `"/*` e termina con `*/`. Il secondo invece inizia con `/**` e termina con `*/`. Il secondo tipo di commento inoltre, può contenere parole chiavi identificate dal fatto che sono precedute dal carattere `@`. Dato in input un file sorgente, come output stampare tutti i commenti (integralmente) e le singole parole chiavi (senza il carattere `@`). Escludere tutto il resto.

%%

parola = [A-Z a-z]+

parola_chiave = "@"{parola}

parolas = {parola} | {parola_chiave}

commento1 = `"/*` {" "}* {parola})* `*/`

commento2 = `/**` {" "}* {parolas})* `*/`

%%

parola_chiave { syso(yytext().substring(1, yylength()-1)) }

commento1 { syso(yytext()) }

commento2 { syso(yytext()) }

Dare una specifica in formato JFlex per il riconoscimento di un sottoinsieme di URL definiti per diversi protocolli. Lo scanner deve essere in grado di riconoscere:

- più di un singolo schema, ed in particolare http:, ftp:, gopher:, e https:.
- i domini espressi come nome (es: www.unisa.it) e come indirizzo IP (es: 193.205.186.10).

- l'uso di una porta rispetto a quella di default (il dominio viene seguito da un ":" ed il numero della porta).
- la presenza delle ancore nell'URL (file .html seguito da un "#" ed il nome della ancora).
- le sequenze di escape in qualsiasi posizione nell'URL (il carattere "%" seguito da una coppia di cifre esadecimale). Si possono usare in qualsiasi punto tranne nello schema e per il carattere /.
- L'estensione può essere presente solo per i file.

Esempi

http://www.miosito.it:8080/file.html

ftp://10.9.9.71/prova.zip

http://altro.sito.com/%7Eutente/

http://altro.sito.com/%7Eutente/user.jsp

%%

protocollo = http | ftp | gopher | https | nntp | file

primolivello = it | com | gov | edu | net | uk | fr | de

porta = [1-9][0-9]{4}

escape = "%" [0-9A-F] [0-9A-F]

nome = [A-Z a-z]*

ipaddress = [0-9]{0,3} "." [0-9]{0,3} "." [0-9]{0,3} "." [0-9]{0,3}

dominio = "www." {nome} "." {primolivello}

ancora = "#" {nome}

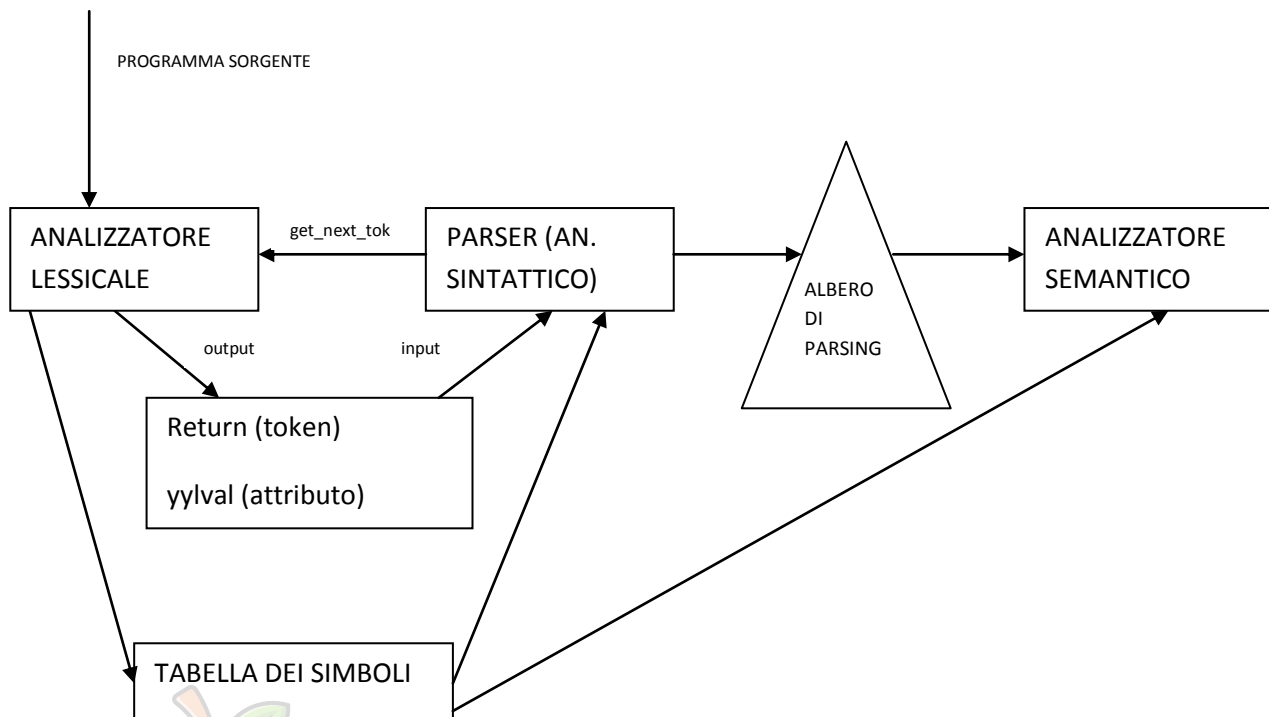
url = {protocollo} "://" ({dominio} | ipaddress) (":" {porta})? ("/" {nome})* ("/" | ("/" {nome} "." {nome} {ancora})?)? ({escape})?

%%

url printf("url valido" + yylval)

LEZIONE 6 – ANALISI SINTATTICA (1 PARTE)

16/03/2009



Il parser riceve una stringa di token dall'analizzatore lessicale e verifica che la stringa può essere generata dalla grammatica del linguaggio sorgente. Il parser restituirà l'**albero di parsing** che è la struttura sintattica del programma, è una struttura gerarchica che andrà in input nell'analizzatore semantico. La costruzione di un **analizzatore sintattico** è molto simile alla costruzione di un analizzatore lessicale. Infatti:

ANALIZZATORE LESSICALE: serviva un linguaggio di specifica dei token: le espressioni regolari (alle quali corrispondono gli automi finiti, le cui versioni deterministiche e non deterministiche sono equivalenti), e di un modo per tradurre tali specifiche in un software: l'analizzatore lessicale. Abbiamo visto due possibili modi per passare dalle specifiche all'analizzatore lessicale:

- Manuale: si passa per un diagramma di transizione poi lo si decodifica nel codice che implementa le espressioni regolari.
- Automatica: è il LEX che traduce le specifiche in un analizzatore lessicale.

ANALIZZATORE SINTATTICO: serve un linguaggio di specifiche, per specificare la sintassi di un linguaggio di programmazione: le grammatiche (alle grammatiche corrispondono gli automi "push-down" la cui versione non deterministica è più potente di quella deterministica). Ci servirà poi un modo per passare dalla grammatica, dalle specifiche all'analizzatore sintattico (software); ciò si può fare in due modi:

- Manuale: si passa per diagrammi di transizioni (diversi dai precedenti), poi li si decodifica in codice;
- Automatica: YACC prende in input la grammatica e restituisce direttamente l'analizzatore sintattico.

Ci sono diversi tipi di **parser**; essi si differenziano per il tipo di algoritmo che porta alla costruzione dell'albero di parsing:

- **PARSER UNIVERSALI** (non deterministici): sono costruiti a partire da grammatiche C.F. generali (individua tutte le possibili grammatiche che esistono): possono essere grammatiche complesse

(non si esclude il non determinismo), quindi il riconoscitore di tali grammatiche, deve essere non deterministico; il trattamento di tali grammatiche ha un'alta complessità di tempo. Ci sono due algoritmi di riferimento:

- Algoritmo di Cocke-Younger-Kosami;
- Algoritmo di Tamita;

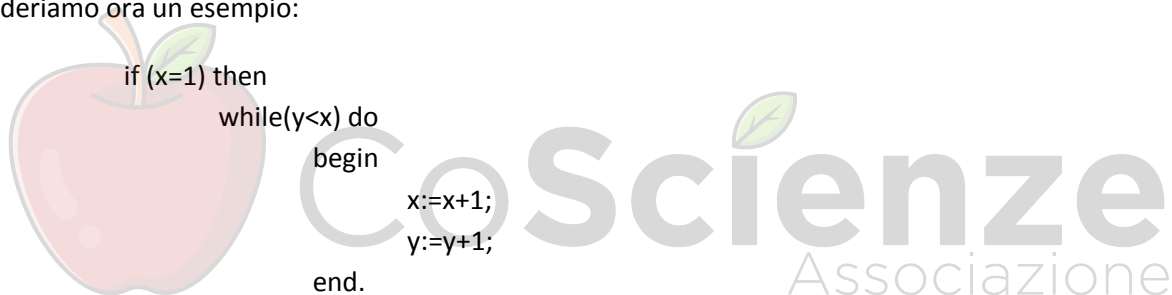
sono parser utilizzati nel linguaggio naturale (molto più complessi dei linguaggi di programmazione). Se n è la taglia dell'input tali parser hanno rispettivamente complessità $O(n^3)$ il primo e il secondo $O(n)$ nei casi normali ma può arrivare anche a complessità esponenziali.

Per i linguaggi di programmazione si usano i parser più efficienti:

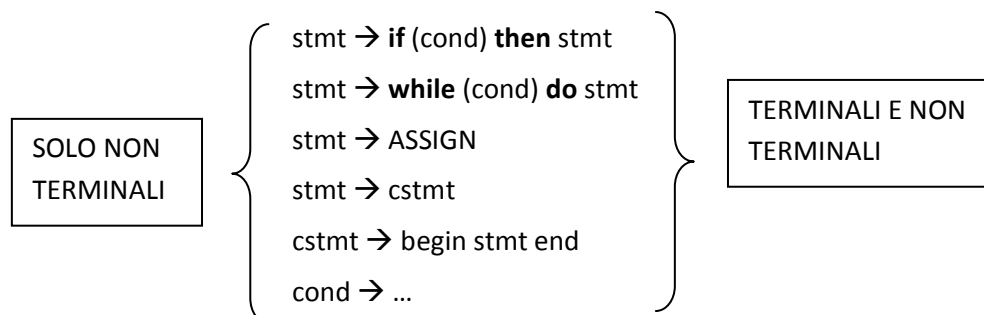
- **PARSER DETERMINISTICI:** che hanno complessità $O(n)$. Ci sono due tipi di parser principalmente usati:
 - Top - down: costruiscono gli alberi di parsing dalla radice alla foglia;
 - Bottom-up: costruiscono gli alberi di parsing dalle foglie alla radice;

Per l'analisi lessicale non c'era bisogno di distinguere il caso non deterministico da quello deterministico in quanto le due versioni di automi finiti sono equivalenti. Lo stesso non accade per gli automi push-down: è quindi necessaria la distinzione fatta. **Gli analizzatori lessicali sono basati su automi a stati finiti mentre gli analizzatori sintattici sono basati su automi push-down.**

Consideriamo ora un esempio:



Come prima cosa si costruisce una grammatica:



Dalle specifiche di un linguaggio di programmazione, formalizzandole, si ottiene una grammatica. Su tale grammatica decidiamo quali sono i token e quali i terminali. A questo punto si dà il programma sorgente all'analizzatore lessicale.

Un piccolo riepilogo sulle grammatiche

Un parse tree è una rappresentazione grafica di una derivazione che produce l'ordine nel quale le produzioni sono state applicate per sostituire i non-terminali. Ogni nodo interno di un parse tree rappresenta l'applicazione di una produzione e sono etichettati con non-terminali nella testa di una produzione; i figli di un nodo sono etichettati, da sinistra a destra, dai simboli del corpo della produzione. Le etichette delle foglie lette da sinistra a destra si dicono prodotto del parse tree.

Ad esempio:

$E \rightarrow E+E \mid E^*E \mid -(E) \mid id.$

1) Come derivo $-(id)$? $E \rightarrow -E$; $-E \rightarrow -(E)$; $-(E) \rightarrow id$

2) Come derivo $id+id*id$? $E \rightarrow E+E$; $E+E \rightarrow id+E$; $id+E \rightarrow id+E^*E$; $id+E^*E \rightarrow id+id^*E$; $id+id^*E \rightarrow id+id^*id$;

$A \rightarrow y$; $\alpha A \beta \rightarrow \alpha y \beta$

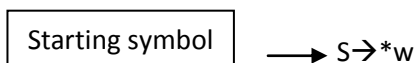
Come derivo $\alpha A \beta$?

Se ottengo αn da $\alpha 1$ con n passi allora ho derivato con n passi. $^* \rightarrow$ in 0 o più passi

$S \rightarrow^* \beta$ and $\beta \rightarrow y$ allora $\alpha \rightarrow^* y$.

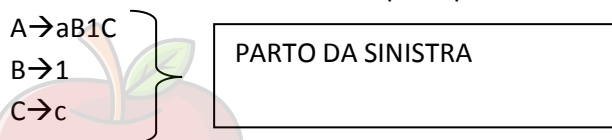
$S \rightarrow^* \alpha$ se α contiene solo simboli terminali allora è una sentenza della grammatica, altrimenti è detta forma sentenziale della grammatica.

Se $w \in L(G)$ allora:

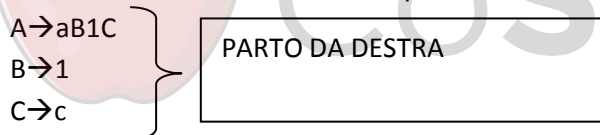


Abbiamo due tipi di derivazione:

- LEFT-MOST: viene derivato sempre il primo non-terminale più a sinistra.



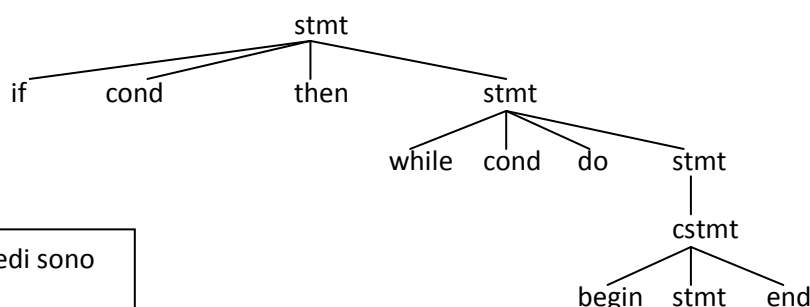
- RIGHT MOST: viene derivato sempre il non-terminale più a destra.



1° PROBLEMA: AMBIGUITÀ

La derivazione ci permette di costruire un albero, il cosiddetto parser tree. Costruendo l'albero però perdo l'ordine di derivazione. È da notare inoltre che nell'esempio n°2 fatto in precedenza è possibile ottenere anche un'altra derivazione sinistra: $E \rightarrow E^*E$; $E^*E \rightarrow E+E^*E$; $E+E^*E \rightarrow id+E^*E$; $id+E^*E \rightarrow id+id^*E$; $id+id^*E \rightarrow id+id^*id$.

In pratica è possibile costruire due alberi. Quale di questi due dovremo scegliere??? Ci troviamo di fronte ad un esempio di grammatica ambigua e vedremo tra poco come risolvere questo problema. Se l'analizzatore sintattico si basa su tale grammatica, dato in input il programma scritto, ci aspettiamo che costruisca l'albero di parsing:



I nodi intermedi sono non terminali

Quando non si riesce a costruire l'albero si ha un errore. È necessario individuare l'errore e segnalarlo insieme con la linea di codice dove è avvenuto.

Consideriamo un esempio classico di grammatica ambigua:

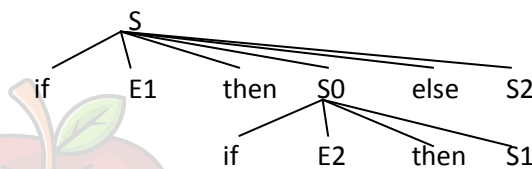
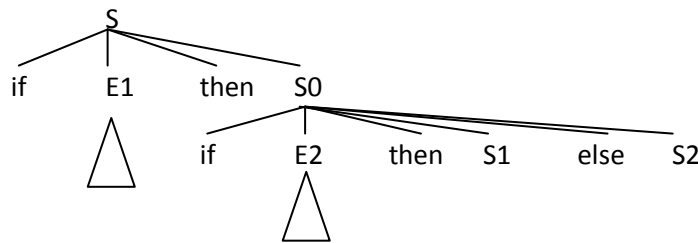
if E1 then((if E2 then S1)else S2)

Se avessimo la grammatica:

$S \rightarrow \text{if } E \text{ then } S$

$S \rightarrow \text{if } E \text{ then } S \text{ else } S'$ (sta ad indicare il riuso di S che non è terminale)

Questa grammatica non risulta essere una buona descrizione per la frase che abbiamo in quanto permette due possibili interpretazioni:



La grammatica, come detto già anticipato, è ambigua. Per la frase scritta si possono avere due alberi diversi di derivazione. Le grammatiche ambigue vanno riscritte per evitare il non determinismo; dovrà esistere un solo albero di derivazione. Nel nostro esempio si usa la regola che l'else si attacca all'ultimo then. La grammatica deve sempre rispettare la semantica di ciò che vogliamo; la grammatica deve riconoscere tutto e solo ciò che vogliamo. Una grammatica corretta per il nostro esempio è la seguente:

$\text{stmt} \rightarrow \text{stmt_else}$

$\text{stmt} \rightarrow \text{stmt_noelse}$

$\text{stmt_else} \rightarrow \text{if } E \text{ then stmt_else else stmt_else} | \dots$

$\text{stmt_noelse} \rightarrow \text{if } E \text{ then stmt_else else stmt_noelse} | \text{if } E \text{ then stmt}$

Si può vedere che da tale grammatica c'è un unico modo per derivare la frase precedente. Lo scopo principale è quello di evitare le grammatiche ambigue.

Confrontando ciò che abbiamo detto sull'analizzatore sintattico ci si rende conto che: l'albero di derivazione è generato dalla grammatica mentre l'albero di parsing è generato dall'analizzatore sintattico. Vista la relazione tra grammatica e analizzatore sintattico si ha che l'albero di derivazione e l'albero di parsing sono la stessa cosa.

Vediamo ora alcuni casi di possibili problemi.

2° PROBLEMA: GRAMMATICHE RICORSIVE

Grammatiche ricorsive a sinistra e le derivazioni left-most con:

$S \rightarrow SaBx$ un albero sbilanciato tutto a sinistra

Grammatiche ricorsive a destra e le derivazioni right-most con:

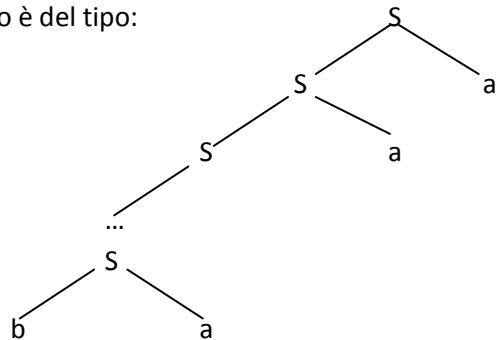
$S \rightarrow aBxS$ un albero sbilanciato tutto a destra

Vedremo che nei parser top-down la ricorsione a sinistra porta a dei loop infiniti; bisogna quindi eliminare le derivazioni sinistre. Vi è un algoritmo per eliminare le ricorsioni sinistre.

Consideriamo la grammatica:

$S \rightarrow Sa | b$ (genera il linguaggio ba^*)

L'albero è del tipo:



Come risolviamo questo problema???? Effettuiamo una trasformazione. Vediamo come trasformare la grammatica per eliminare la ricorsione sinistra:

$S \rightarrow bS'$

$S' \rightarrow aS'$

$S' \rightarrow \epsilon$

Eliminiamo la ricorsione a sinistra per ottenere quella a destra. Ciò non è un problema in quanto l'albero top-down ha problemi solo con la ricorsione a sinistra. Tale grammatica genera ancora lo stesso linguaggio ma non è più ricorsiva a sinistra ma bensì a destra.

Grammatica ricorsiva a sinistra più complessa:

$S \rightarrow Sa | Sc$

$S \rightarrow b | d$

Si trasforma nella grammatica ricorsiva destra:

$S \rightarrow bS' | dS'$

$S \rightarrow aS' | cS' | \epsilon$

Che riconosce lo stesso linguaggio. Quindi vediamo un algoritmo generico che ci permette di trasformare grammatiche ricorsive sinistre in grammatiche ricorsive destre. Data la grammatica ricorsiva sinistra:

$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_n$

$A \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$

Dove $\beta_1, \beta_2, \dots, \beta_n$ sono i simboli iniziali delle parole generate, si costruisce la grammatica ricorsiva destra equivalente:

$A \rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_n A'$

$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_n A' | \epsilon$

Tali tipi di ricorsione sono dette ricorsioni dirette. Vi sono anche grammatiche con ricorsione indiretta.

Esempio di ricorsione indiretta:

$S \rightarrow Ab$

$A \rightarrow Sc$

3° PROBLEMA: IL PROBLEMA DELLA FATTORIZZAZIONE A SINISTRA

Ad esempio data la grammatica:

$A \rightarrow abD$
 $A \rightarrow abCE$
 $D \rightarrow d$
 $C \rightarrow c$
 $E \rightarrow e$

Come determino quali tra le prime due derivazioni devo usare?? La soluzione si ha costruendo una grammatica alternativa:

$A \rightarrow abX$
 $X \rightarrow D \mid CE$
 $D \rightarrow d$
 $C \rightarrow c$
 $E \rightarrow e$

Si applica la cosiddetta **fattorizzazione a sinistra** a produzioni che hanno stesso non terminale a sinistra (nel nostro caso A) e nella parte destra hanno uno stesso prefisso (nel nostro caso ab). Generalizzando vi è un semplice algoritmo che esegue la fattorizzazione sinistra:

dato $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n$

con α prefisso comune a tutte le produzioni. Si introduce una nuova variabile A' e si sostituiscono le produzioni precedenti con le seguenti:

$A \rightarrow \alpha A'$
 $A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

Dopo la fattorizzazione a sinistra non ci saranno produzioni con stessa parte sinistra e stessi prefissi nella parte destra.

Abbiamo visto metodi per trasformare grammatiche in modo da riportarle in forme più utili ai nostri scopi:

- Non ambigue;
- Non ricorsive sinistre;
- Fattorizzate a sinistra.

PARSING TOP-DOWN

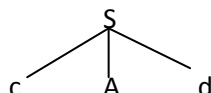
Il parsing top-down può essere visto come il problema di costruire un parse tree per una stringa data in input, iniziando dalla radice e creando i nodi del parse tree in preorder. Equivalentemente, top-down parsing può essere visto come la ricerca della derivazione leftmost per una stringa data in input.

Ad esempio consideriamo la grammatica:

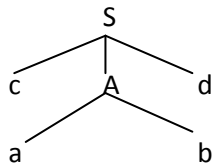
$S \rightarrow cAd$
 $A \rightarrow ab \mid a$
E la frase $w=cad$.

Le uniche due parole che sono riconosciute con questa grammatica sono cad e cabd

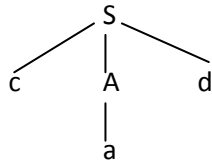
Simuliamo un parser top-down che data la grammatica o la frase crei l'albero di parsing della frase: si parte da S (starting symbol), e con un puntatore sul primo simbolo di w, si applica la prima regola che si incontra ottenendo l'albero:



La foglia più a sinistra coincide col primo simbolo di w , si è così generato il primo simbolo c . Si consuma, si avanza col puntatore su “ a ” (della frase w). Si espande A con la prima regola che si incontra:



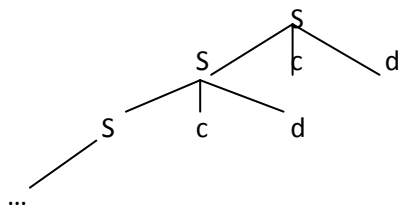
Si ha un match per la “ a ”. Si avanza il puntatore e confrontando si vede che b non fa match con d si ha quindi un insuccesso, si deve tornare indietro riportando il puntatore su “ a ” e vedere se è possibile applicare un’altra regola su A :



Si è così ottenuto un albero di derivazione per w . Nell’esempio si è usato l’algoritmo di discesa ricorsiva con back-tracking (si è tornati indietro nell’albero per applicare l’altra regola su A – le regole vengono applicate nell’ordine in cui si incontrano). Una grammatica ricorsiva sinistra può dare luogo ad un parser ricorsivo - discendente che, anche con back-tracking, porta ad un loop infinito; infatti quando proviamo ad espandere una variabile A , potremo ritrovarci nella situazione di poter espandere nuovamente A senza che nessun input sia stato consumato. In molti casi si può eliminare la ricorsione sinistra da una data grammatica (abbiamo visto come farlo) ottenendone una che può essere analizzata con un parser ricorsivo-discendente che non necessita di back-tracking, detto anche **parser predittivo**. Per poterlo costruire bisogna conoscere *dato l’input corrente “ a ” ed il simbolo non terminale A da espandere, quale delle produzioni $A \rightarrow \alpha_1 | \alpha_2 | \alpha_3 | \dots | \alpha_n$ è l’unica che deriva una stringa che inizi per “ a ” (con la fattorizzazione sinistra si ha tale informazione)*. La fattorizzazione a sinistra fa sì che ad ogni passo si possa espandere una sola regola, si elimina così il back-tracking. Ad esempio data la grammatica ricorsiva sinistra:

$S \rightarrow Scd | b$

E la parola da cui creare l’albero $w = bcd$.

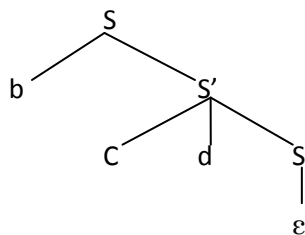


Loop infinito

Eliminando la ricorsione sinistra:

$S \rightarrow bS'$

$S' \rightarrow cdS' | \epsilon$



Eliminando la ricorsione sinistra si elimina il problema del loop infinito.

LEZIONE 7 – ANALISI SINTATTICA (2 PARTE)

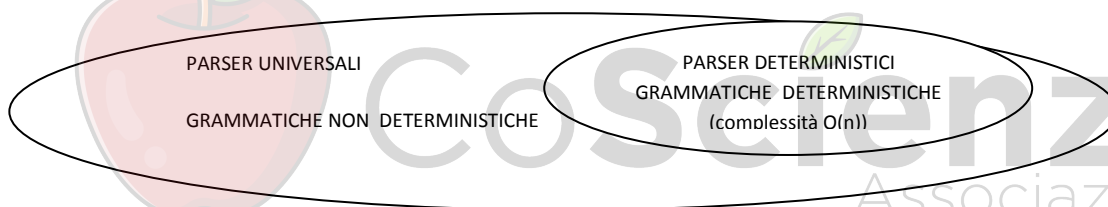
18/03/2009

Quindi, ci sono due algoritmi top-down:

- Recursive-descend: non fanno uso né di fattorizzazione sinistra, né di eliminazione di ricorsione sinistra. Usano il back-tracking e possono andare in loop.
- Predittivi: prima di applicarsi si fa la fattorizzazione sinistra e si elimina la ricorsione sinistra. Non usano back-tracking e non vanno in loop.

Si chiamano predittivi poiché ad ogni passo dell'algoritmo si può predire quale è la produzione corretta da applicare (letto l'input vi sarà una sola possibile produzione da applicare, quella il cui primo simbolo della parte destra fa match con l'input letto). Nell'analizzatore lessicale abbiamo usato le espressioni regolari per esprimere i token; poi abbiamo visto due modi per implementare l'analizzatore lessicale. Per descrivere la **sintassi** si usano le grammatiche: noi useremo le grammatiche context-free poiché sono semplici da usare (hanno un solo non terminale a sinistra); i linguaggi di programmazione non sono totalmente context-free: tramite le grammatiche context-free si riesce a descrivere il 90% di un linguaggio di programmazione, il resto verrà gestito diversamente. Useremo le grammatiche context-free anche per implementare il parser (che genererà l'albero di parsing – l'automa push-down) e gli automi push-down non deterministici, essendo più potenti, hanno una complessità di tempo maggiore di quello degli automi push-down deterministici.

GRAMMATICHE CONTEXT-FREE



Nella teoria dei linguaggi formali sono utilizzate solo le grammatiche context-free deterministiche. Per tali grammatiche abbiamo detto che possiamo implementare due tipi di parser: top-down e bottom-up. Questi due differiscono per come costruiscono l'albero di parsing.

Dalla grammatica che descrive il linguaggio si costruisce il parser, il quale dovrà costruire l'albero di parsing. Stiamo vedendo i parser **top-down** per i quali possono esistere due problemi:

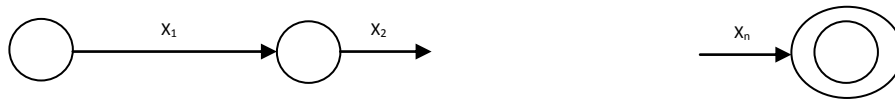
- Loop infinito: se ci sono produzioni ricorsive sinistre;
- Back-tracking: dato dalla fattorizzazione sinistra.

Abbiamo visto come trasformare la grammatica per eliminare tali problemi, cioè le produzioni ricorsive sinistre e la fattorizzazione sinistra, ottenendo così **parser predittivi**. Dalla grammatica così ottenuta si costruiscono i diagrammi di transizione (diversi da quelli incontrati nell'analisi lessicale): si costruisce un diagramma di transizione per ciascun simbolo non terminale. Le etichette sugli archi possono essere sia token (terminali) che non terminali. Si fa una transizione su un token se quel token è il successivo simbolo in input. Una transizione su un non terminale A corrisponde ad una chiamata alla funzione A. Ogni diagramma sarà implementato come una funzione.

Dunque per ogni non terminale A si costruisce un diagramma di transizione come segue:

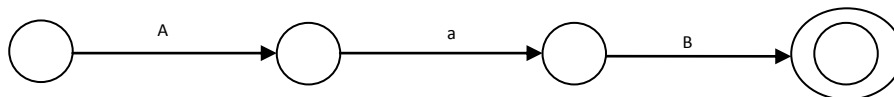
1. Si crea uno stato iniziale ed uno stato finale;

2. Per ogni produzione $A \rightarrow X_1 X_2 \dots X_n$ si crea un cammino dalla stato iniziale a quello finale con archi etichettati X_1, X_2, \dots, X_n .



Ad esempio con $S \rightarrow ABC$ avrò tre diagrammi: quello corrispondente ad A, quello a B e quello a C. Vediamo con più precisione un esempio:

$S \rightarrow AaB, A \rightarrow b, B \rightarrow c$. Il diagramma di transizione per S sarà così costruito:



Quando viene letta la "A" (lo stesso vale per la "B") si va ad esaminare il diagramma di A. Se arrivo in uno stato finale posso ritornare su e passare al prossimo simbolo (questo comportamento è molto simile a quello delle chiamate a funzioni ed è simulabile attraverso uno stack, che noi useremo in modo implicito). Quando esamino la "a" devo verificare solo se c'è match dato che a è un terminale. Se alla fine della parola raggiungo lo stato finale ho ottenuto una parola della grammatica. In pratica ho costruito un parser top-down.

Il parser predittivo lavora sui diagrammi di transizione comportandosi come segue:

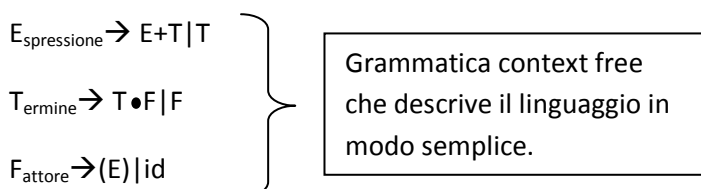
Parte dallo stato iniziale corrispondente all'assioma della grammatica; se dopo alcune azioni si trova nello stato s da cui parte un arco etichettato "a" che entra nello stato t e se il prossimo simbolo in input è "a", allora il parser sposta il cursore di una posizione a destra, sul prossimo simbolo in input, e va nello stato t. Se invece l'arco è etichettato con un non terminale A, il parser va nello stato iniziale del diagramma per A, senza spostare il cursore; se poi A arriva in uno stato finale, allora va nello stato t. Se c'è un arco etichettato e da s a t, allora il parser va da s nello stato t senza spostare il cursore sull'input.

Quindi un programma di parser predittivo, basato su diagrammi di transizione, tenta un match tra i simboli terminali sugli archi e quelli in input ed esegue chiamate a funzioni se trova un arco etichettato con un non terminale. Tale metodo funziona per diagrammi di transizione deterministici; se si hanno diagrammi di transizione non deterministici, cioè che prevedono più transizioni da uno stato per uno stesso input, o si eliminano le ambiguità introducendo opportune regole oppure non è possibile costruire un parser predittivo e si dovrà costruire un parser discendente ricorsivo che preveda dei ritorni indietro per tentare tutte le possibilità.

Prima di passare al passo successivo indichiamo una differenza:

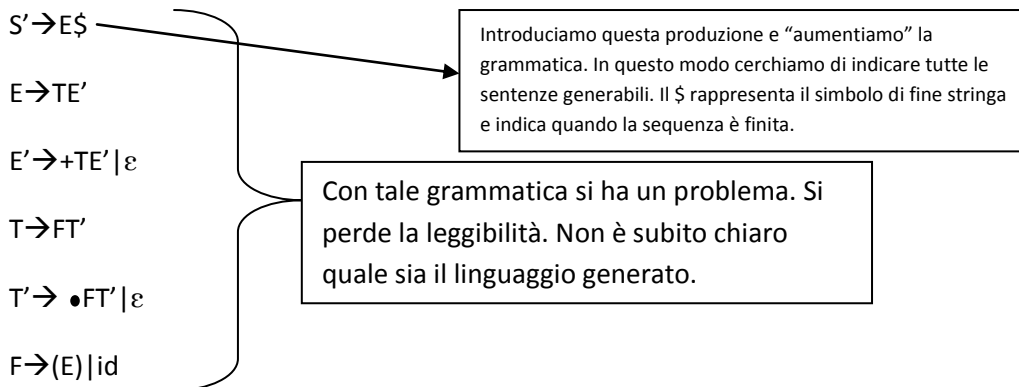
- Riconoscitore: è un analizzatore che restituisce SI oppure NO;
- Parser: produce in output un albero;

Consideriamo il seguente esempio:



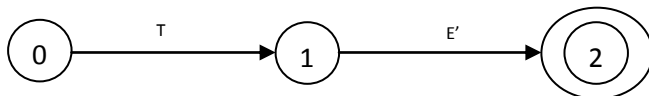
Frase generabili usando questa grammatica sono: id, id+id, id+(id+id) •d

Dalla grammatica si vuole costruire il parser top-down; ma tale grammatica non è buona (ricorsiva sinistra) e quindi bisogna prima trasformarla:

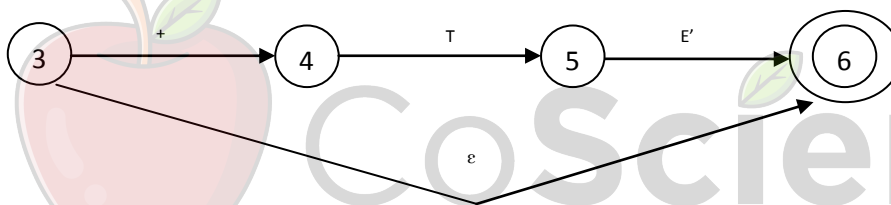


Vediamo ora, per tale grammatica, quali sono i diagrammi di transizione:

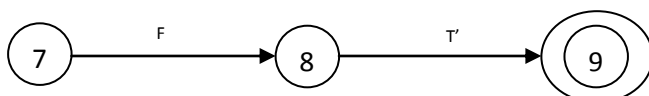
E:



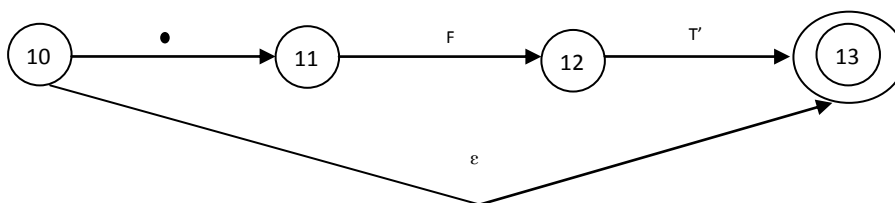
E':



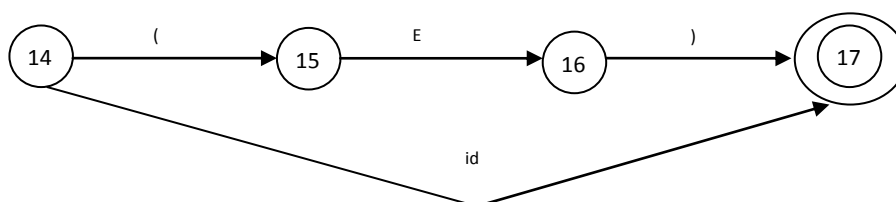
T:



T':



F:



I diagrammi per E' e T' sono non deterministici, poiché hanno la possibilità di una ε-mossa (la ε è sempre applicabile); si rompe tale non determinismo, ambiguità, introducendo la regola che se in input viene letto

“+” (oppure “•”) si fa la transizione il cui arco ha etichetta corrispondente, altrimenti si fa la transizione su ϵ . Eliminata l’ambiguità si può scrivere un programma di parsing predittivo per la grammatica data.

Bisogna ora convertire tali diagrammi in codice, ogni diagramma diventa una funzione:


```

▪ E()
  begin
    T(); E'();
  end.
▪ E'()
  begin
    if(ptr→"+") then
      begin
        match '+'; T(); E'();
      end.
    else
      match  $\epsilon$ ;
    end.
▪ T()
  begin
    F(); T'();
  end.
▪ T'()
  begin
    if(ptr→"•") then
      begin
        match '•'; F(); T'();
      end.
    else
      match  $\epsilon$ ;
    end.
▪ F()
  begin
    if(ptr→"(") then
      begin
        match '('; E(); match ')';
      end.
    if(ptr="id") then
      match 'id';
    (in tutti gli altri casi vi è un errore...
  end.

```

ptr è il puntatore sull'input.

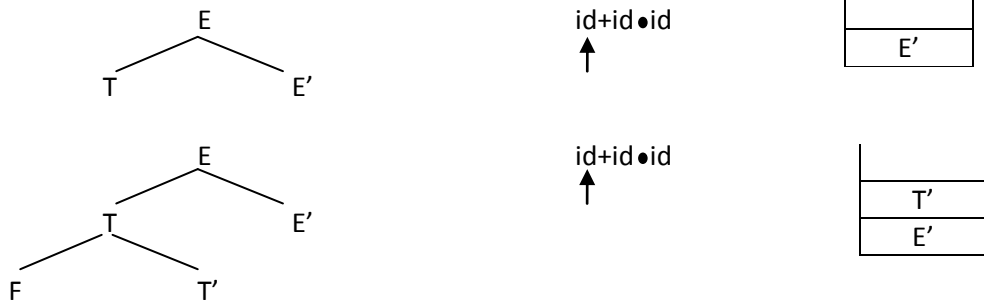
Match è una funzione che va a verificare la corrispondenza del simbolo a '+' e sposta ptr sul prossimo token con la funzione getnexttoken().



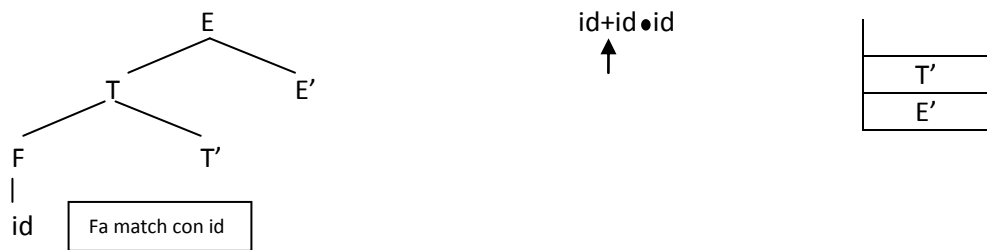
Dato il codice per match, quello scritto è un parser per le espressioni aritmetiche. Ora vediamo come costruire il parser in modo top-down per la frase `id+id•id`; vediamo i vari passi:

E (simbolo/stato iniziale) id+id•id;
 ↑
 Primo simbolo puntato

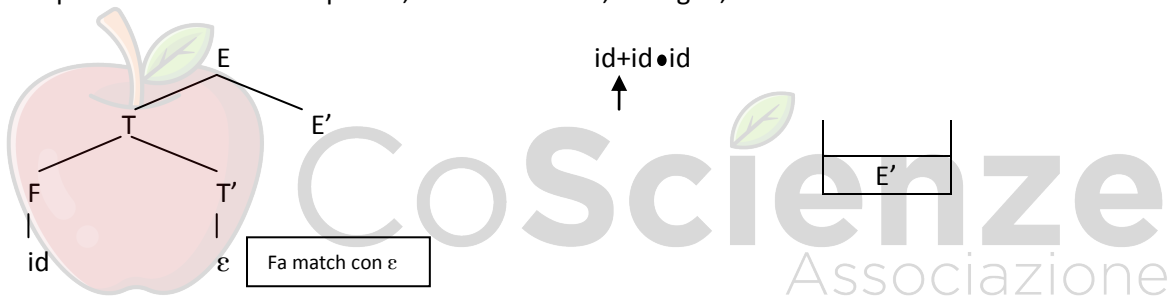
Come prima cosa da E si farà una chiamata alla funzione T (esprimiamo ciò con l’albero della derivazione della frase), ed esplicitiamo il contenuto dello stack ad ogni passo (conserva le informazioni utili alle funzioni che ancora devono terminare).



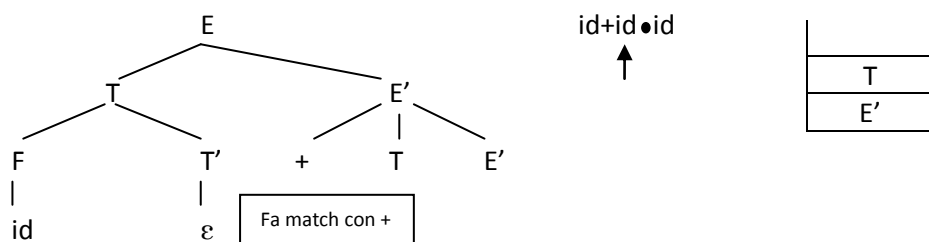
Ora che si va ad eseguire la funzione associata ad F si fa match 'id' e si sposta il cursore sul prossimo simbolo di input.



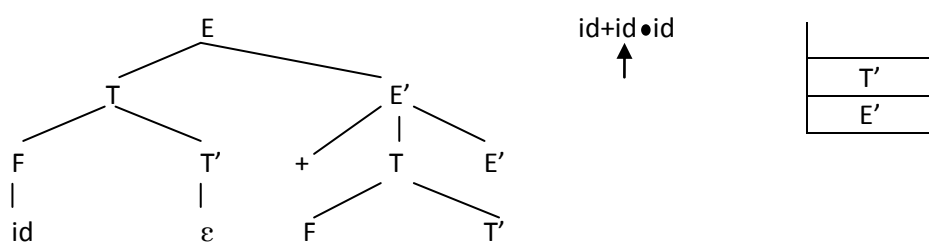
Quando la funzione F è stata eseguita il controllo ritorna a T che chiama T' ; nel frattempo si è spostato il cursore al prossimo simbolo di input '+'; T' non fa niente, o meglio, fa match ' ϵ '.



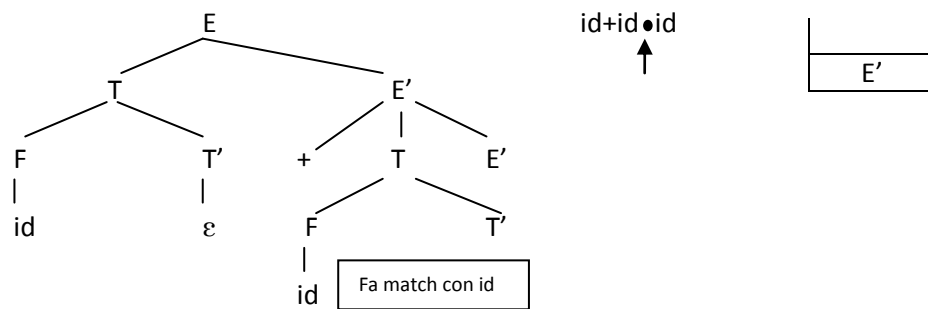
Concluso T' finisce anche T ; il controllo torna ad E che attiva E' , leggendo il suo codice si vede che la condizione dell'if è vera.



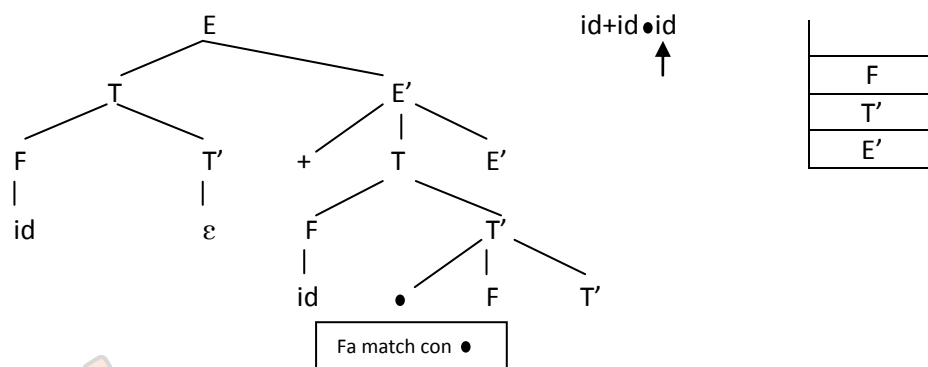
Eseguito match '+' va in esecuzione T che chiama F e mette T' in attesa.



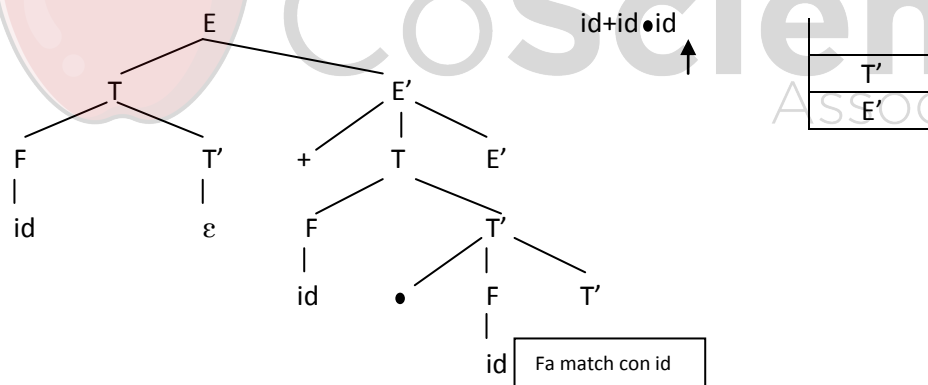
F esegue match 'id' sposta il cursore in avanti e passa il controllo a T' .



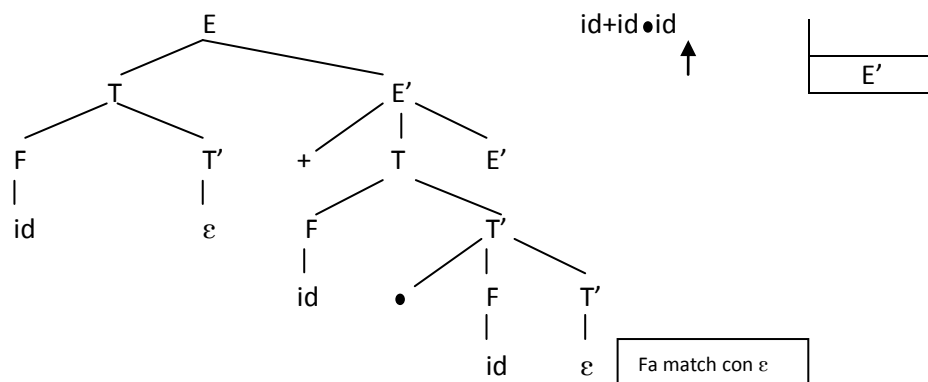
La condizione dell'if in T' è vera, quindi esegue $\text{match}' \bullet$ che sposta il cursore in avanti e mette in attesa F , T' .



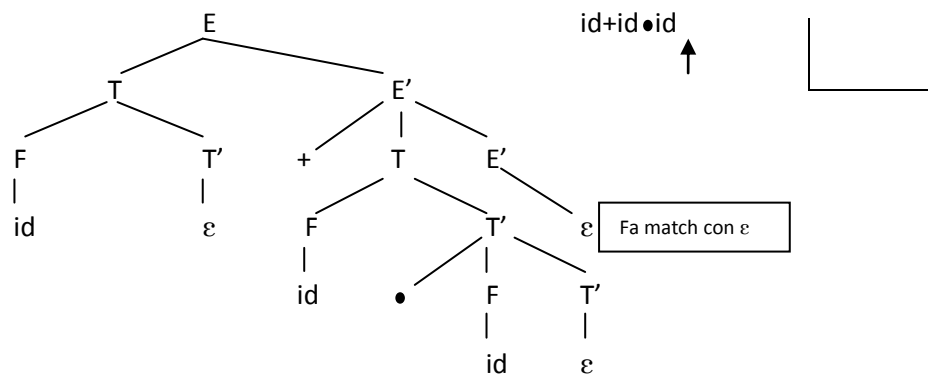
Eseguita anche $\text{match}' \bullet$ viene chiamata la F : la condizione sul secondo if è rispettata quindi esegue solo $\text{match}' \text{id}'$ e poi finisce.



T' esegue l'else quindi fa un $\text{match}' \epsilon'$ che praticamente non fa niente.



Finito T' rimane aperta solo la chiamata ad E' : E' fa l'else e quindi esegue solo $\text{match}' \epsilon'$.



Abbiamo così costruito un albero delle chiamate a funzioni. Sono arrivato in uno stato finale. Se si tiene traccia delle chiamate alle funzioni, quello ottenuto è l'albero di parsing. Se non si tiene un ricordo delle chiamate a funzioni fatte è solo un riconoscitore. Si è così costruito un parser predittivo top-down.

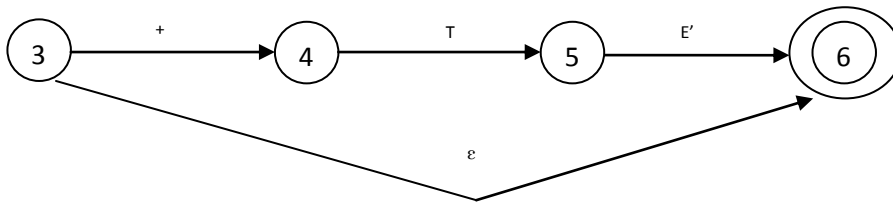
Ci sono tecniche euristiche per semplificare i diagrammi di transizioni. Quanto più semplici e pochi sono i diagrammi di transizione tanto più sarà più efficiente il programma che si ottiene.



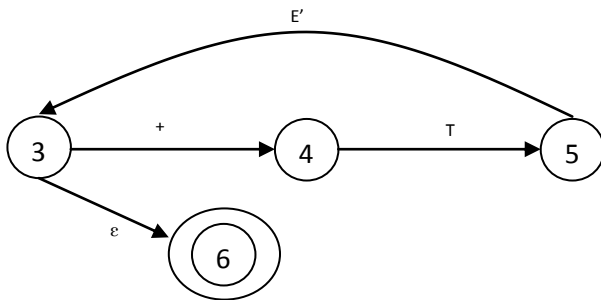
LEZIONE 8 – ANALISI SINTATTICA (3 PARTE)

30/03/2009

Vi sono dei modi per semplificare i diagrammi di transizione. Consideriamo il diagramma di transizione di E' già visto:



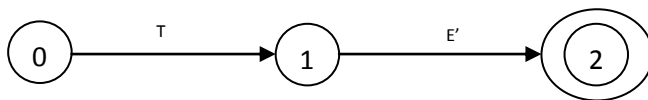
Questo diagramma può essere semplificato in questo:



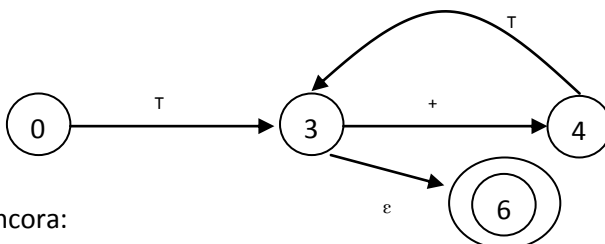
E ancora in:



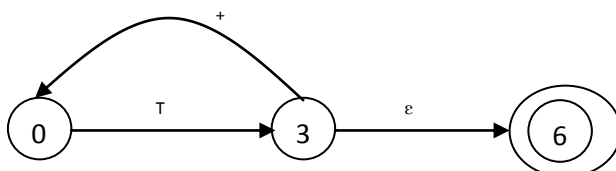
La semplificazione però determina, però, un bel problema: la perdita di leggibilità. Vediamo lo stesso procedimento con E :



Che semplificato diventa



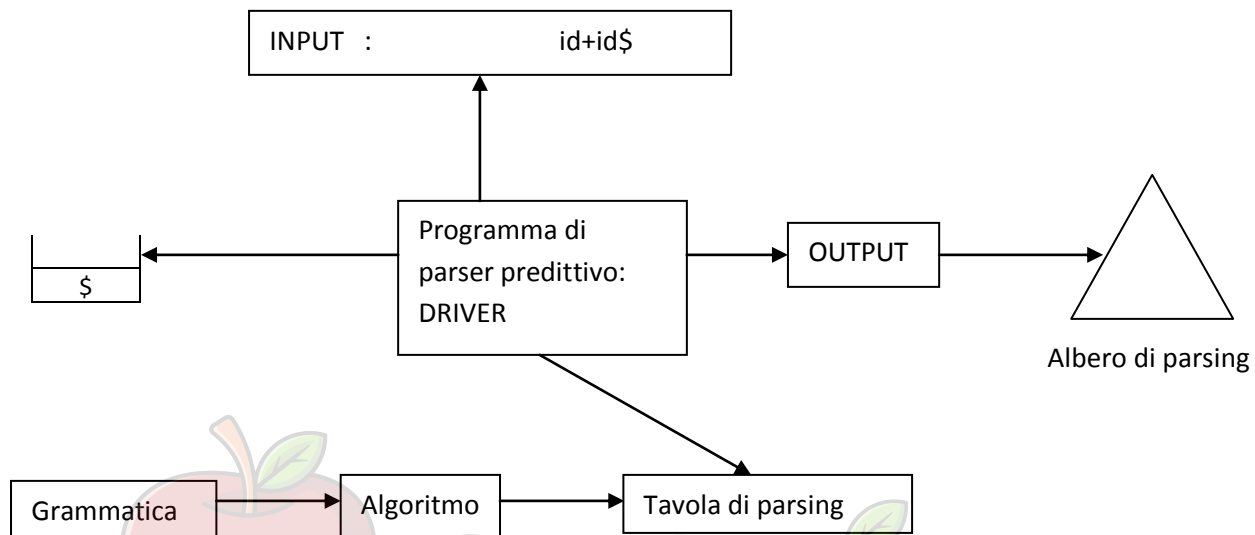
E ancora:



Come si può notare abbiamo sostituito la transizione E' con il rispettivo diagramma semplificato e poi si è effettuata un'ulteriore semplificazione. È possibile utilizzare le stesse tecniche per T e T' . L'utilizzo di queste tecniche permette di costruire dei parser più efficienti. **Essenzialmente ci sono due tecniche per costruire parser top-down:**

1. Si usano i diagrammi di transizione; vi è un uso dello stack implicito per la gestione dei record di attivazione delle chiamate a funzione.
2. Si usa uno stack "esplicitamente".

L'architettura ottenuta costruendo il parser con la seconda tecnica è la seguente:



- ✓ Lo stack serve per tenere traccia delle chiamate a funzioni e gestito a run-time. Il buffer di input contiene la stringa che deve essere analizzata seguita da \$ che è il simbolo usato per indicare la fine della stringa di input. Lo stack contiene una sequenza di simboli della grammatica, vi è un simbolo speciale il \$ per indicare il fondo dello stack. Inizialmente lo stack, oltre al \$ contiene l'assioma della grammatica (lo starting symbol).
- ✓ La tavola di parsing è un array bidimensionale $M[A,a]$ (A = simbolo grammatica, a = simbolo sentenza) dove " A " è un non terminale e " a " è un terminale oppure è il \$. Tale tavola dipende dalla grammatica, infatti è costruita a partire dalla grammatica; in corrispondenza di ogni grammatica si avrà una tavola di parsing diversa.
- ✓ Il driver è il programma che controlla il parser: considera X , il top dello stack, ed " a " il simbolo in input; tali simboli determinano l'azione del parser, ossia, una di queste tre:
 1. Se $X=a=\$ \rightarrow$ il parser si ferma e l'analisi finisce con successo (è una sentenza del linguaggio come l'"id" finale nell'esempio).
 2. Se $X=a!=\$ \rightarrow$ ciò significa che X è un terminale; il parser estrae X dallo stack e avanza il puntatore di input sul successivo simbolo (c'è stato un match come l'"id" iniziale dell'esempio).
 3. Se X è un non terminale (quindi $X!=a$ e $X!=\$$) il driver consulta l'elemento $M[X,a]$ della tavola di parsing. Tale elemento sarà una X -produzione della grammatica oppure un rilevatore di errori. Se è una X -produzione del tipo $X \rightarrow UVW$, il driver rimpiazza X (quindi si toglie dallo stack) con UVW dove U sarà il top dello stack (ricordarsi che abbiamo a che fare con una derivazione left-most).

Dalla descrizione si capisce che il driver è indipendente dalla grammatica; qualsiasi sia la grammatica il driver sarà sempre lo stesso. Quindi dato un parser (costruito con tale tecnica), se ne vogliamo un altro per un'altra grammatica bisogna solo cambiare la tavola di parsing. L'output del parser sarà l'insieme delle produzioni che il driver ha individuato nella tavola di parsing. Equivale all'albero di parsing costruito.

Ad esempio a partire dalla grammatica per le espressioni aritmetiche si ottiene la seguente tabella di parsing:

T e r m i n a l i	Non terminali					
	id	+	.	()	\$
	E	$E \rightarrow TE'$		$E \rightarrow TE'$		
	E'	$E' \rightarrow +TE'$	$E' \rightarrow \epsilon$		$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
	T	$T \rightarrow FT'$		$T \rightarrow FT'$		
	T'	$T' \rightarrow \epsilon$	$T' \rightarrow \bullet FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
	F	$F \rightarrow id$		$F \rightarrow (E)$		

Le produzioni nella tavola indicano come espandere il non terminale sul top dello stack. I vuoti indicano chiamate a procedure di gestione degli errori: syntax-error. Possono anche indicare dove si è verificato l'errore. Consideriamo E l'assioma della grammatica e id+id\$ l'input. Vediamo il funzionamento del parser.

		id+id\$	
	E		REGOLA 3
	\$		

Il driver vedendo E sul top dello stack ed id sull'input accede alla posizione (E,id) della tabella; trova $E \rightarrow TE'$, predice la produzione da usare e sostituisce E con E'T.

		id+id\$	
	T		REGOLA 3
	E'		
	\$		

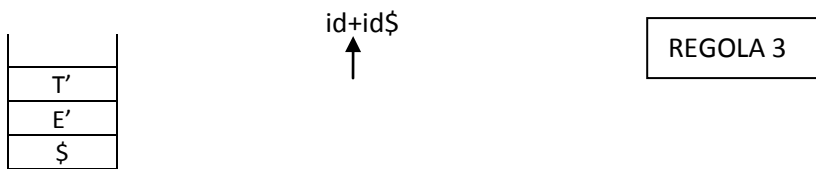
Ora si accede alla posizione (T,id).

		id+id\$	
	F		REGOLA 3
	T'		
	E'		
	\$		

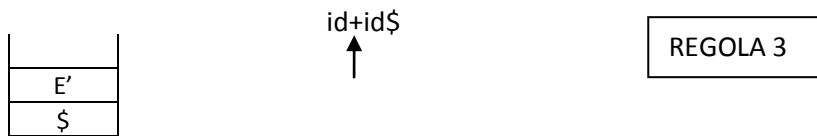
Si accede alla posizione (F,id).

		id+id\$	
	Id		REGOLA 2
	T'		
	E'		
	\$		

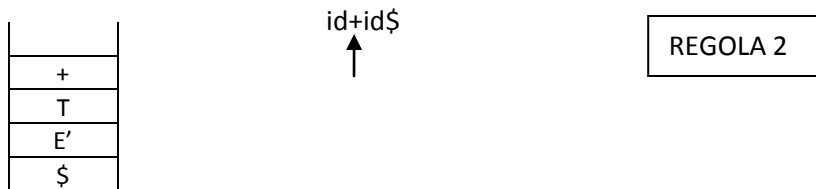
In questo caso si effettua il match, si elimina "id" dallo stack e si sposta a destra il puntatore sull'input.



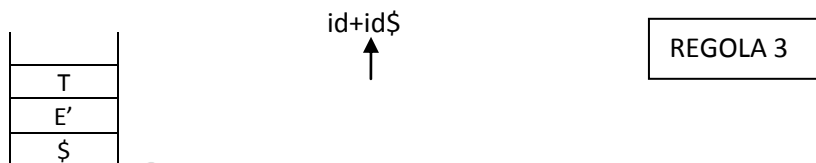
Ora si applica la $(T',+): T' \rightarrow \epsilon$ che rimpiazza T' dal top dello stack.



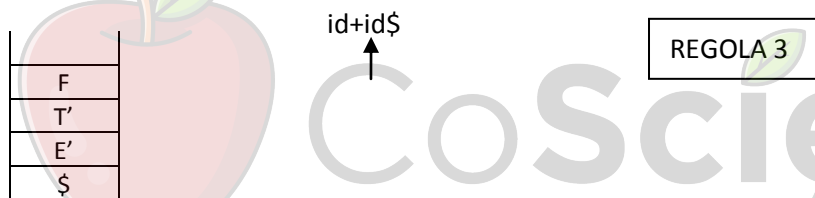
Ora si accede a $(E',+): E' \rightarrow +TE'$.



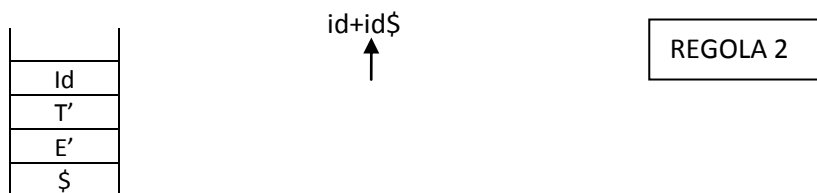
C'è il match di "+" che viene eliminato.



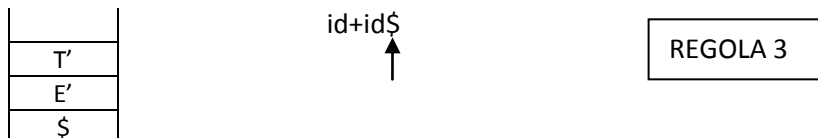
Ora si accede in $(T,id): T \rightarrow FT'$.



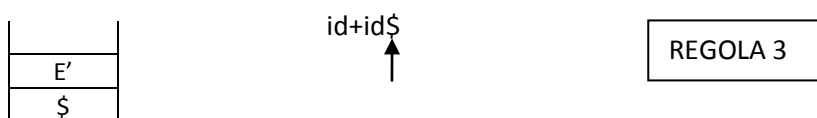
Ora si accede in $(F,id): F \rightarrow id$.



C'è un match che elimina "id" dallo stack.



Ora si accede in $(T',\$): T' \rightarrow \epsilon$ cancella T'.



Ora si accede in $(E',\$): E' \rightarrow \epsilon$ che cancella E'.



L'analisi si ferma con successo. Nelle posizioni vuote della tabella ci dovrebbero essere delle funzioni di errore. Si può vedere che se in input si dà una parola sbagliata, ossia non appartenente al linguaggio generato dalla grammatica, si finisce in una delle celle vuote. Per esempio se sul top dello stack vi è "E", in input si deve avere per forza "id" oppure "("; nel caso venisse letto un simbolo diverso si potrebbe attivare una funzione che segnala il carattere atteso. Si nota l'analogia di tale procedimento con quello che ha portato alla costruzione dell'albero di parsing, con la differenza che l'uso dello stack è esplicito.

La costruzione di parser top-down e bottom-up è supportata da due funzioni FIRST e FOLLOW associate ad una grammatica. Durante un parsing top-down queste funzioni aiutano a scegliere la produzione da applicare, basandosi sul prossimo simbolo di input. Vi sono due funzioni che meritano attenzione e rispetto e sono:

1. FIRST
2. FOLLOW

che ci permettono di riempire la tavola di parsing. Vediamole in dettaglio.

FIRST

Se α è una stringa di simboli della grammatica, $FIRST(\alpha)$ è l'insieme dei terminali che iniziano le stringhe derivate da α . Si prende solo il primo terminale di tutte le stringhe derivate da α . Se $\alpha \rightarrow^* \epsilon$ allora anche ϵ è in $FIRST(\alpha)$.

Vediamo una definizione di FIRST formale effettuata mediante induzione.

Per calcolare $FIRST(X)$ si applicano le seguenti regole:

1. Se x è un terminale allora $FIRST(x) = \{x\}$;
2. Se $X \rightarrow \epsilon$ è una produzione, allora ϵ è aggiunto a $FIRST(X)$;
3. Se X è un non terminale e $X \rightarrow Y_1 Y_2 \dots Y_k$ è una produzione allora:
 - a. Se ϵ è in $FIRST(Y_j)$ per ogni $j=1,2,\dots,k$ allora ϵ è aggiunto a $FIRST(X)$ (ϵ deve essere in ogni termine di X);
 - b. Se Y_1 non deriva ϵ allora $FIRST(X) = FIRST(Y_1)$;
 - c. Se $Y_1 \rightarrow^* \epsilon$ e Y_2 non deriva ϵ allora $FIRST(X)$ sarà dato da $FIRST(Y_1)$ unito $FIRST(Y_2)$ e così via (quindi ϵ non appartiene a $FIRST(X)$; in generale, se ϵ è in $FIRST(Y_1), FIRST(Y_2), \dots, FIRST(Y_{i-1})$, cioè se $Y_1 \dots Y_{i-1} \rightarrow^* \epsilon$ allora a $FIRST(X)$ aggiungiamo anche $FIRST(Y_i)$).

Vediamo ora come calcolare il FIRST di una stringa X_1, \dots, X_n :

- ✓ Se $\epsilon \in FIRST(X_i)$ per $i=1,\dots,n$ allora ϵ sarà in $FIRST(X_1 \dots X_n)$;
- ✓ Se ϵ non è in $FIRST(X_1)$ allora $FIRST(X_1 \dots X_n) = FIRST(X_1)$;
- ✓ Se ϵ è in $FIRST(X_1)$ allora $FIRST(X_1 \dots X_n) = FIRST(X_1) \cup \epsilon FIRST(X_2 \dots X_n)$ (la particolare unione sta ad indicare che l'unione viene fatta qualora ϵ appartiene a $FIRST(X_1)$, in generale a $FIRST(X_i)$); quindi in generale se ϵ è in $FIRST(X_i)$ ed ϵ è in $FIRST(X_2)$ allora a $FIRST(X_1 \dots X_n)$ si aggiunge anche $FIRST(X_3)$ e così via.

Quindi molte volte il calcolo del FIRST si riduce al calcolo di altri FIRST. Vediamo un esempio di calcolo del FIRST. Consideriamo la grammatica delle espressioni aritmetiche:

$S' \rightarrow E \$$

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow \bullet FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

Calcoliamo il $FIRST(E) = FIRST(TE') = FIRST(T) \cup \epsilon FIRST(E') \rightarrow FIRST(T) = FIRST(FT') = FIRST(F) \cup \epsilon FIRST(T') \rightarrow$
 $FIRST(F) = \{id, \{;\}$

Non si calcola né il $FIRST(E')$ né il $FIRST(T')$ poiché sia $FIRST(T)$ che $FIRST(F)$ non contengono ϵ .

Calcoliamo il $FIRST(E') = FIRST(+TE') \cup FIRST(\epsilon) = \{+, \epsilon\};$

Calcoliamo il $FIRST(T') = FIRST(\bullet FT') \cup FIRST(\epsilon) = \{\bullet, \epsilon\};$

FOLLOW

Si calcola sui simboli non terminali. Caso 1: sia A un non terminale, $FOLLOW(A)$ è l'insieme dei terminali "a" che possono apparire alla destra di A in qualche forma sentenziale, cioè, è l'insieme dei terminali "a" per i quali esiste una derivazione della forma $S \rightarrow^* \alpha A \beta$ qualsiasi siano α e β . Quindi per ogni forma sentenziale si prende solo il terminale che segue A. Caso 2: data $A \rightarrow X_1 X_2 \dots X_n$ si ha che: $FOLLOW(X_2) = FIRST(X_3 \dots X_n) \cup FOLLOW(A)$.

Se $X_3 \dots X_n$ tutti uguali a ϵ . Quindi il successivo di X_2 sarà il successivo di A.

Per calcolare il $FOLLOW()$ per tutti i non terminali si applicano le seguenti regole:

1. Porre $\$$ in $FOLLOW(S)$ (nella grammatica vi è sempre la regola $S' \rightarrow S\$$ anche se non è scritta esplicitamente. Quindi $\$$ appartiene a $FOLLOW(S)$ sempre).
2. Se vi è una produzione $A \rightarrow \alpha B \beta$, con α e β qualsiasi, ogni cosa in $FIRST(\beta)$, eccetto ϵ , è posta in $FOLLOW(B)$.
3. Se vi è una produzione $A \rightarrow \alpha B$ oppure $A \rightarrow \alpha B \beta$ dove ϵ è in $FIRST(\beta)$ cioè $\beta \rightarrow^* \epsilon$ allora ogni cosa in $FOLLOW(A)$ sarà posta in $FOLLOW(B)$.

Si nota che i $FOLLOW$ non contengono mai ϵ poiché la presenza di $S' \rightarrow S\$$ assicura che a fine stringa ci sarà almeno $\$$.

Vediamo un esempio di applicazione della terza regola:

$A \rightarrow \alpha B$

$A \rightarrow Aa$

Si ha la derivazione $A \rightarrow Aa \rightarrow \alpha Ba$, per cui ciò che segue A segue anche B.

La a segue anche B per cui dobbiamo includere il $FOLLOW(A)$;

Vediamo un altro esempio: si consideri la grammatica che genera le espressioni aritmetiche:

$S' \rightarrow E\$$

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow \bullet FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

Calcoliamo $FOLLOW(E)$: $\$ \in FOLLOW(E)$, $FIRST("(") \in FOLLOW(E)$. Quindi $FOLLOW(E) = \{, \$\}$.

Calcoliamo $FOLLOW(E')$: $FOLLOW(E) \subseteq FOLLOW(E')$, $FOLLOW(E') \subseteq FOLLOW(E')$; quindi $FOLLOW(E') = \{, \$\}$.

Calcoliamo $FOLLOW(T)$: $FIRST(E') \cup FOLLOW(E)$. Quindi $FOLLOW(T) = \{+, \$\}$

Calcoliamo $FOLLOW(T')$: $FOLLOW(T) \cup FOLLOW(T')$. Quindi $FOLLOW(T') = \{+, \$\}$

Calcoliamo $FOLLOW(F)$: $FIRST(T') \cup FOLLOW(T') (=FOLLOW(T))$. Quindi $FOLLOW(F) = \{\bullet, +, \$\}$.

Vediamo ancora un altro esempio:

$A \rightarrow Bx$

$A \rightarrow CD$

$B \rightarrow b$

$C \rightarrow c$

Per decidere quale produzione sulla A applicare si calcola: $FIRST(Bx) = FIRST(B) = \{b\}$ e $FIRST(CD) = FIRST(C) = \{c\}$ (poiché sia B che C non hanno ϵ , non si tiene in considerazione di x e D per rispettare la regola 3-b del FIRST). Quindi si userà la prima produzione, poiché può generare una parola che inizia con b, come voluto; mentre la seconda produzione genererà una parola che inizia per c e, in questo caso, non ci interessa. Nella tavola di parser avremo questa situazione:

	b	C	
A	$A \rightarrow Bx$	$A \rightarrow CD$	

Vediamo ancora un altro esempio:

$A \rightarrow BC$

$B \rightarrow a|\epsilon$

$C \rightarrow c|d|\epsilon$

Calcoliamo il $FIRST(BC) = FIRST(B) \cup FIRST(C)$. In questo caso, siccome nel $FIRST(B)$ c'è ϵ si ricorre al $FIRST(C)$. Siccome ϵ è anche nel $FIRST(C)$ sarà presente anche nel $FIRST(B)$. In conclusione il $FIRST(B) = \{a, c, d, \epsilon\}$.

Ancora un altro esempio. Nella grammatica delle espressioni aritmetiche se siamo in E e leggiamo +id si ha un errore poiché + non appartiene a $FIRST(E)$, cioè E' non genera nessuna parola che inizia con +.

Un ultimo esempio. Consideriamo la grammatica:

$S \rightarrow Bc$

$S \rightarrow BA$

$A \rightarrow x$

Avremo:

$FOLLOW(B) = \{c\} \cup FIRST(A) = \{c, x\}$;

$FOLLOW(S) \leq FOLLOW(A)$ e quindi il $FOLLOW(A) = FOLLOW(S)$.

Quindi usiamo tali funzioni, FIRST e FOLLOW, per costruire la tavola di parsing M per una grammatica G. Il problema principale è: dato un non terminale A ed un input abc..., quale produzione applicare per poter ottenere "a" a partire da A?

L'idea è la seguente:

se $A \rightarrow \alpha$ è una produzione e " a " $\in FIRST(\alpha)$, allora il parser espanderà A da α quando il simbolo in input è " a ". L'unica applicazione si verifica quando $\alpha = \epsilon$ oppure $\alpha \rightarrow * \epsilon$; in tal caso si espande di nuovo A da α se il simbolo in input è in $FOLLOW(A)$ oppure se \$ è in $FOLLOW(A)$.

Algoritmo: costruzione della tabella M per un parser predittivo.

Input: grammatica G.

Output: tabella M.

Metodo:

1. Per ogni produzione $A \rightarrow \alpha$ in G fare i passi 2 e 3.
2. Per ogni terminale " a " in $FIRST(\alpha)$ aggiungere $A \rightarrow \alpha$ in $M[A, a]$.
3. Se ϵ è in $FIRST(\alpha)$ aggiungere $A \rightarrow \alpha$ in $M[A, b]$ per ogni terminale b in $FOLLOW(A)$.
Se ϵ è in $FIRST(\alpha)$ e \$ è in $FOLLOW(A)$ aggiungere $A \rightarrow \alpha$ in $M[A, \$]$
4. Considerare ogni elemento non definito di M come un errore.

Tale algoritmo può essere applicato ad una qualsiasi grammatica G , per produrre la tabella M . Per applicare l'algoritmo prima si elimina la ricorsione sinistra alla grammatica e poi la si fattorizza a sinistra in modo da evitare i problemi determinati dalla presenza di questi due fattori. Vediamo un approfondimento sui passi 2 e 3.

PASSO 2: abbiamo un non terminale A (sullo stack), ed in input "a"; utilizzando le produzioni in $M[A,a]$ siamo sicuri di costruire una derivazione che genera una stringa che inizia per "a".

PASSO 3: ϵ in $FIRST(\alpha)$ significa che $\alpha \rightarrow^* \epsilon$ e quindi $A \rightarrow^* \epsilon$ e A è annullabile. In questo caso può capitare di avere A sullo stack ed un input b anche se $A \rightarrow^* b\alpha$. Significa (se l'input è corretto) che vi sarà una produzione la cui parte destra è del tipo $\alpha A b \beta$ o meglio che $b \in FOLLOW(A)$. Quindi per tutte le b appartenenti al $FOLLOW(A)$ bisognerà annullare la A (non genera niente di utile) e quindi si pone $A \rightarrow \alpha$ in $M[A,b]$. Se $\$$ è in $FOLLOW(A)$ si ha il caso in cui si ha $A\$$ e quindi bisogna solo annullare la A ; potrebbe essere finito l'input e quindi $A \rightarrow \alpha$ è posto in $M[A,\$]$.

Vediamo per chiarire un esempio

$S' \rightarrow Sx$

$S \rightarrow AB$

$A \rightarrow a$

$A \rightarrow \epsilon$

$B \rightarrow b$

$C \rightarrow c$

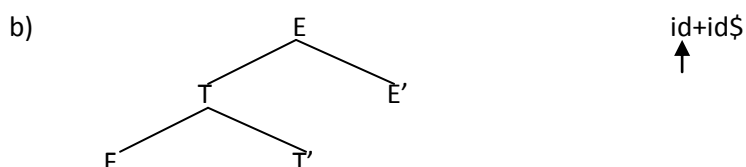
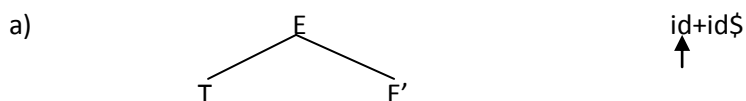
	S'	S	A	B
a		$S \rightarrow AB$	$A \rightarrow a$	
b		$S \rightarrow AB$	$A \rightarrow \epsilon$	$B \rightarrow b$
x	$S' \rightarrow Sx$	$S \rightarrow AB$	$A \rightarrow \epsilon$	$B \rightarrow \epsilon$

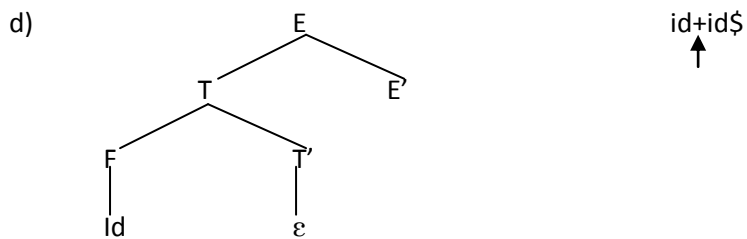
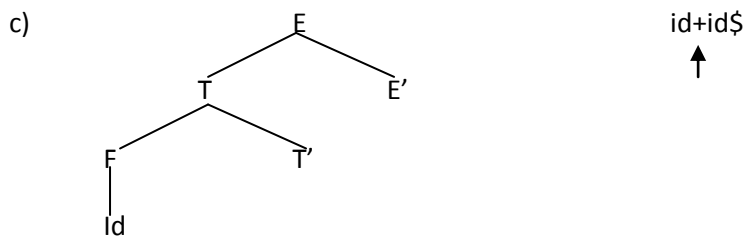
La regola $S \rightarrow AB$ la si applica per tutti i terminali in $FIRST(AB)$; $FIRST(AB) = \{a, b, \epsilon\}$. Ma in $FIRST(AB)$ vi è anche ϵ (cioè la S può scomparire), per cui la regola si applica anche i terminali in $FOLLOW(S)$ dove $FOLLOW(S) = \{x\}$. $FOLLOW(A) = FIRST(B) \cup \epsilon FOLLOW(S) = \{b, x\}$ e quindi $A \rightarrow \epsilon$ la si mette in $M[A,b]$ ed in $M[A,x]$. Stesso discorso per B : $FOLLOW(B) = FOLLOW(S) = \{x\}$.

Vediamo un esempio completo.

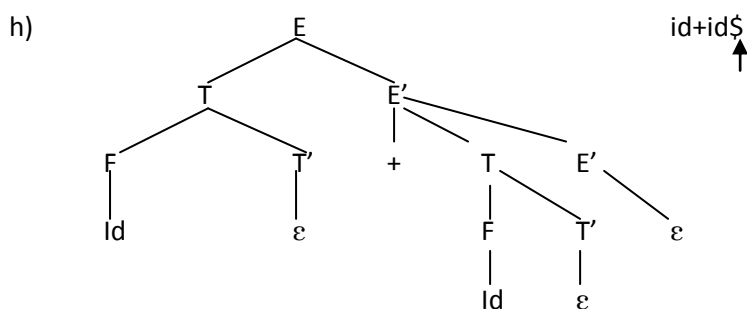
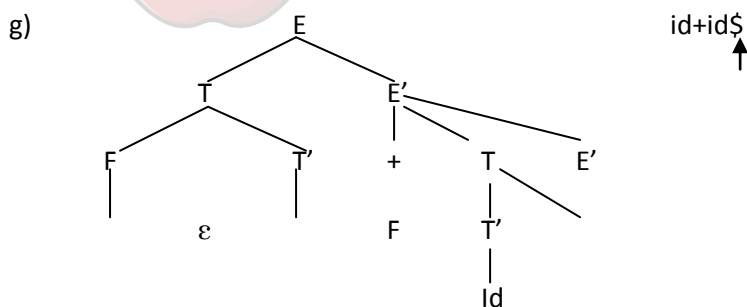
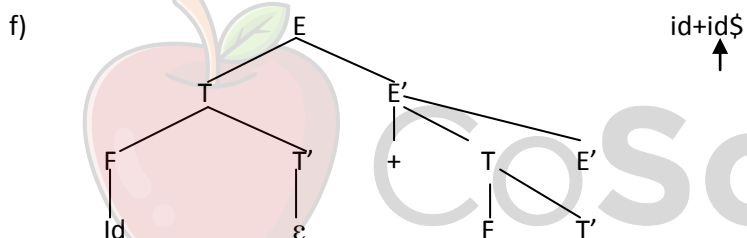
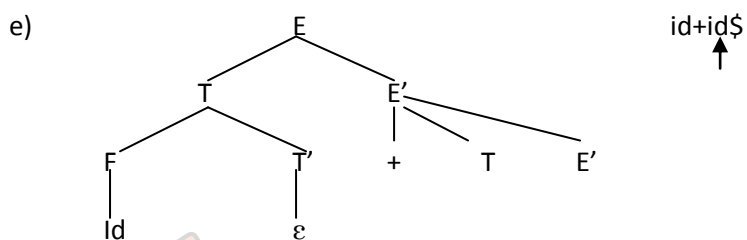
	id	+	\bullet	()	$\$$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$ (follow E')	$E' \rightarrow \epsilon$ (follow E')
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow \bullet FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Input: $id+id\$$ e si parte da E .





T non può generare +; lo si può eliminare se ciò che segue è un simbolo che può seguire T'. se faccio scomparire T' ciò che può seguire è: +, ●,), \$.



Con tale tecnica:

- L'input si scorre da sinistra a destra: L;
- Si ricostruisce una derivazione left-most: L;
- Ad ogni passo si è deciso quale produzione sviluppare considerando un solo simbolo: 1.

LL(1)

Quindi l'algoritmo di parsing si chiama LL(1). Dall'algoritmo vi sono tre possibili rappresentazioni:

- Con l'albero (come sopra);
- Con la derivazione;
- Con lo stack (l'altra volta).

Per alcune grammatiche, M può avere alcuni elementi che sono multidefiniti; tali grammatiche non sono LL e il parser non è deterministico, poiché per una stessa entrata vi sono due possibili produzioni. Se G è ricorsiva sinistra o ambigua, allora M avrà almeno un elemento multidefinito. Ad esempio:

$A \rightarrow aB$ avendo in input ac e si parte da A

$A \rightarrow aC$

$B \rightarrow b$

$C \rightarrow c$

	a	
A	$A \rightarrow aB$ $A \rightarrow aC$	

Non possiamo scegliere deterministicamente quale produzione scegliere; la grammatica non è LL. L'algoritmo LL-parsing è deterministico, non si può applicare la grammatiche che non sono LL. Le grammatiche LL non sono ambigue né ricorsive sinistre. Se una grammatica ha una tabella M con elementi multidefiniti si può tentare di renderli singoli effettuando delle trasformazioni come l'eliminazione della ricorsione sinistra e fattorizzando a sinistra quando è possibile. Vi sono anche grammatiche che dopo tali trasformazioni standard non sono LL. Ad esempio:

$A \rightarrow XB$

$A \rightarrow YC$

$X \rightarrow a|b$

$Y \rightarrow a|c$

	a	b	c
A	$A \rightarrow XB$ $A \rightarrow YC$	$A \rightarrow XB$	$A \rightarrow YC$

Questa grammatica nonostante sia fattorizzata a sinistra e non sia ricorsiva sinistra non è LL(1). Si potrebbe provare a trasformare in qualche modo la grammatica per renderla LL, ma non vi è nessuna tecnica standard. Come vedere se una grammatica G è LL(1)??

Una grammatica G è LL(1) solo se ogni qual volta $A \rightarrow \alpha/\beta$ sono due produzioni distinte di G, valgono le seguenti condizioni:

- ✓ 1 REGOLA: per nessun terminale "a", α e β derivano entrambe stringhe che iniziano per "a":
 $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$ (vuoto).

L'esempio precedente non rispettava tale condizione poiché $FIRST(XB) = \{a, b\}$ e $FIRST(YC) = \{a, c\}$: entrambi hanno la "a".

- ✓ 2 REGOLA: al più uno dei due, α o β , deriva la stringa vuota: $\alpha \rightarrow^* \epsilon$ e $\beta \rightarrow^* \epsilon$ e anche questo non deve capitare.

$S \rightarrow Ac$

$A \rightarrow X$

$A \rightarrow Y$

$X \rightarrow a|\epsilon$

$Y \rightarrow b|\epsilon$

	a	b	c
A	$A \rightarrow X$	$A \rightarrow Y$	$A \rightarrow X$ $A \rightarrow Y$ Follow(A)

$FIRST(X) = \{a, \epsilon\}$.

$FIRST(Y) = \{b, \epsilon\}$.

La prima regola è rispettata ma non la seconda. Poiché ϵ è sia in $FIRST(X)$ che in $FIRST(Y)$ bisogna considerare in entrambi i casi $FOLLOW(A)=\{c\}$.

- ✓ 3 REGOLA: se $\beta \rightarrow^* \epsilon$ allora α non deriva alcuna stringa che inizia con un terminale in $FOLLOW(A)$: se $\beta \rightarrow^* \epsilon$ allora $FIRST(\alpha) \cap FOLLOW(A) = \emptyset$. Lo stesso vale se $\alpha \rightarrow^* \epsilon$.

$S \rightarrow Ac$

$A \rightarrow X$

$A \rightarrow Yc$

$X \rightarrow a | \epsilon$

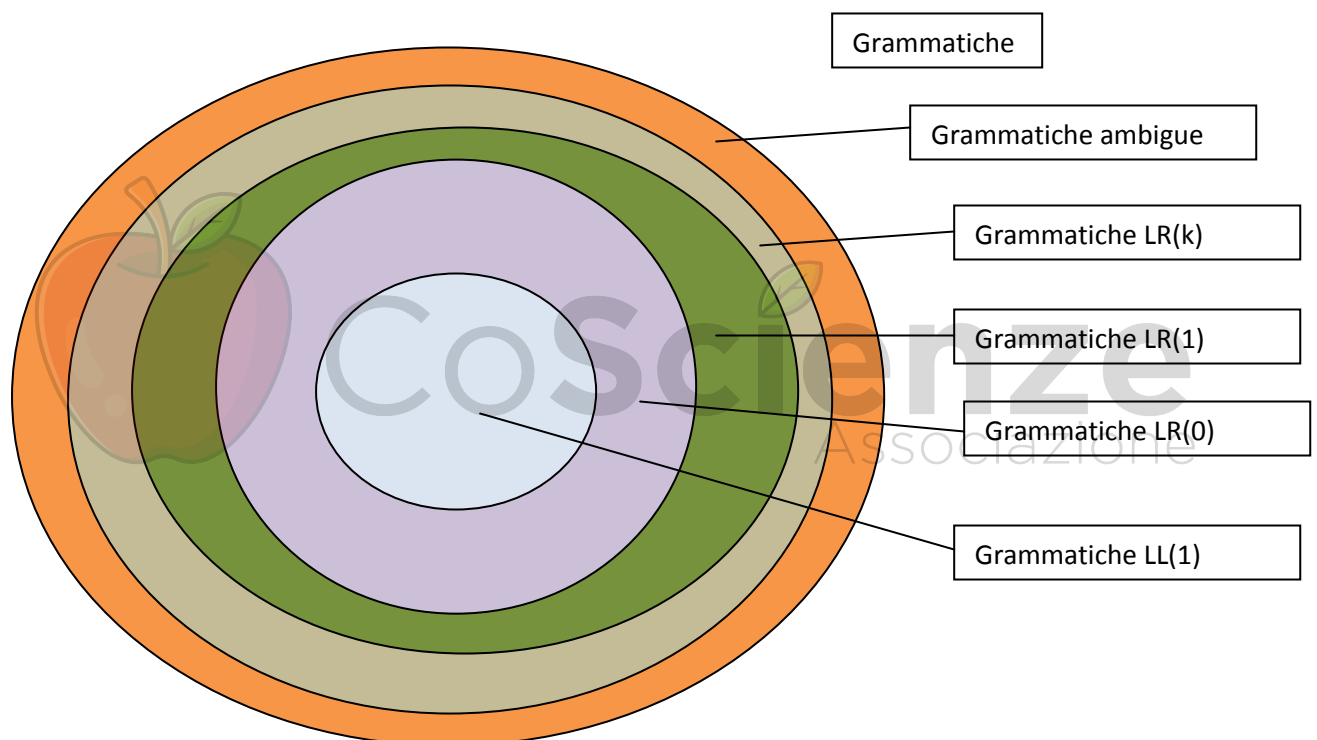
$Y \rightarrow b | \epsilon$

	a	b	c
A	$A \rightarrow X$	$A \rightarrow Yc$	$A \rightarrow X$ $A \rightarrow Yc$ $FOLLOW(A)$

$FIRST(X)=\{a, \epsilon\}$.

$FIRST(Yc)=\{b, c\}$

La 1 e la 2 sono rispettate ma la 3 no; infatti $FOLLOW(A)=\{c\}$. In conclusione vediamo una panoramica sulle grammatiche:



LR(0) significa che non ci sono simboli che ci aiutano a decidere quale produzione considerare.

LEZIONE 9 – ANALISI SINTATTICA (4 PARTE)

01/04/2009

PARSING BOTTOM-UP

Usiamo uno stile generale di analisi sintattica bottom-up nota come parsing shift-reduce. Una forma semplice di implementare un parsing shift-reduce è detta parsing d'operatore di precedenza. Un metodo più generale di parsing shift-reduce è detto parsing LR ed è usato per generatori automatici di parser (da YACC). Il parsing shift-reduce tenta di costruire un albero di derivazione per una stringa in input iniziando dalle foglie fino alla radice. Possiamo pensare a questo processo come ad un modo di ridurre una stringa w in input all'assioma di una grammatica. Ad ogni passo della riduzione una particolare sottostringa, che fa match con il lato destro di una produzione, è rimpiazzata dal simbolo alla sinistra della produzione, e se ad ogni passo la sottostringa è scelta correttamente, partendo dalla radice si ha una derivazione right-most.

In termini non formali un handle di una stringa è una sottostringa che fa match con il lato destro di una produzione e la cui riduzione al simbolo non terminale sul lato sinistro della produzione rappresenta un passo opposto rispetto alla derivazione right-most. Non sempre la sottostringa β che fa match con il lato destro di quale produzione $A \rightarrow \beta$ è un handle. Ciò è vero infatti solo nel caso in cui la riduzione con la produzione $A \rightarrow \beta$ produce una stringa che può essere derivata dall'assioma con una derivazione right-most. Per una grammatica ambigua, una forma sentenziale destra può avere più derivazioni right-most che la generano, per cui può avere più handle.

Vediamo come ottenere al contrario una derivazione right-most:

- Si parte da una stringa di terminali w ; se w è una forma sentenziale destra della grammatica allora $w = \gamma_n$ dove γ_n è l' n -esima forma sentenziale destra di qualche derivazione right-most non ancora nota: $S \rightarrow \gamma_0 \rightarrow \gamma_1 \rightarrow \dots \rightarrow \gamma_n = w$.

Per ricostruire tale derivazione dalla fine all'inizio, si trova l'handle β_n in γ_n e rimpiazziamo β_n con il lato sinistro di una produzione $A_n \rightarrow \beta_n$ per ottenere l' $(n-1)$ -esima forma sentenziale destra γ_{n-1} e si ripete lo stesso procedimento. Se si arriva ad una forma sentenziale destra costituita solo dall'assioma S ci si ferma e si annuncia con successo la fine del parsing. Le produzioni usate per le riduzioni costituiscono una derivazione right-most per la stringa di input w . Se la grammatica data non è ambigua si ha una sola derivazione right-most, quindi tale procedimento è deterministico; in caso contrario vi possono essere più possibilità di scelta. Vediamo un esempio. Consideriamo la grammatica ambigua:

$E \rightarrow E+E / E \bullet E / (E) / id$

Vi sono due derivazioni right-most per una stessa stringa $id_1 + id_2 \bullet id_3$ e sono:

1)

$E \rightarrow \underline{E+E} \rightarrow E+\underline{E \bullet E} \rightarrow E+E \bullet \underline{id_3} \rightarrow E+\underline{id_2 \bullet id_3} \rightarrow \underline{id_1 + id_2 \bullet id_3}$;

Gli handle sono

2)

$E \rightarrow \underline{E \bullet E} \rightarrow E \bullet \underline{id_3} \rightarrow E+\underline{E \bullet id_3} \rightarrow E+\underline{id_2 \bullet id_3} \rightarrow \underline{id_1 + id_2 \bullet id_3}$;

Gli handle sono

Si nota che le stringhe alla destra di un handle contengono solo terminali. Si nota una differenza tra le due derivazioni:

- La prima derivazione dà maggiore priorità a \bullet ;
- La seconda derivazione dà maggiore priorità a $+$.

Vediamo un altro esempio. Consideriamo la grammatica: $E \rightarrow E+E \mid E \bullet E \mid id$ e la stringa in input $id_1+id_2 \bullet id_3$. Le riduzioni fatte con un parser shift-reduce sono:

FORMA SENTENZIALE DESTRA	HANDLE	PRODUZIONE RIDUCENTE
$id_1+id_2 \bullet id_3$	id_1	$E \rightarrow id$
$E+id_2 \bullet id_3$	id_2	$E \rightarrow id$
$E+E \bullet id_3$	id_3	$E \rightarrow id$
$E+E \bullet E$	$E \bullet E$	$E \rightarrow E \bullet E$
$E+E$	$E+E$	$E \rightarrow E+E$
E	E	

Questo è lo starting symbol.

Gli handle vengono presi dalla produzione. In questo procedimento ci sono due problemi da risolvere:

- In una forma sentenziale destra, localizzare la sottostringa che deve essere ridotta, cioè localizzare l'handle;
- Determinare quale produzione scegliere per la riduzione nel caso in cui vi sono più produzioni con la sottostringa sul lato destro.

Vediamo che struttura dati usare per implementare un parser shift-reduce:

- Uno stack per mantenere i simboli della grammatica;
- Un buffer di input per mantenere la stringa w che deve essere analizzata;

Usiamo $\$$ per marcare il bottom dello stack e la fine della stringa di input. Inizialmente lo stack è vuoto e la stringa w è in input:

- STACK: $\$$
 - INPUT: $w\$$
- Stato iniziale

Il parser opera shiftando zero o più simboli input sullo stack finché un handle β è sul top dello stack. A questo punto il parser riduce β al lato sinistro dell'appropriata produzione. Il parser continua tale ciclo fino a che incontra un errore o lo stack contiene l'assioma e l'input è vuoto:

- STACK: $\$ \$$
 - INPUT: $\$$
- Stato finale

La computazione termina ed il parser si arresta. Vediamo le configurazioni successive di un tale parser shift-reduce:

STACK	INPUT	AZIONE
$\$$	$id_1+id_2 \bullet id_3 \$$	Shift
$\$id_1$	$+id_2 \bullet id_3 \$$	Reduce con $E \rightarrow id$
$\$E$ (non è handle faccio shift)	$+id_2 \bullet id_3 \$$	Shift
$\$E+$ (non è handle)	$id_2 \bullet id_3 \$$	Shift
$\$E+id_2$	$\bullet id_3 \$$	Reduce con $E \rightarrow id$
$\$E+E$ (non è handle)	$\bullet id_3 \$$	Shift
$\$E+E \bullet$ (non è handle)	$id_3 \$$	Shift

$\$E+E \bullet id_3(\text{è handle})$	$\$$	Reduce con $E \rightarrow id$
$\$E+E \bullet E$	$\$$	Reduce con $E \rightarrow E \bullet E$
$\$E+E$	$\$$	Reduce con $E \rightarrow E+E$
$\$E$	$\$$	accept

Un parser shift-reduce può fare 4 tipi di azioni:

1. SHIFT: il simbolo in input è shiftato sul top dello stack;
2. REDUCE: si rimpiazza l'handle sul top dello stack con un simbolo non terminale;
3. ACCEPT: il parser annuncia con successo il completamento dell'analisi; l'input è terminato e nello stack è rimasto solo l'assioma.
4. ERROR: il parser scopre un errore di sintassi e chiama una routine di copertura dell'errore. L'errore si verifica se non si riesce a trovare un handle e l'input non è stato completamente consumato.

L'uso dello stack per un parser shift-reduce è giustificato dal fatto che, con tale struttura un handle è sempre sul top dello stack e mai all'interno. Vediamo un esempio. Consideriamo la grammatica:

- 1) $S \rightarrow aABe$
- 2) $A \rightarrow Abc$
- 3) $A \rightarrow b$
- 4) $B \rightarrow d$

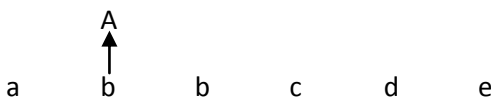
e la stringa in input $w=abbcde$. Vogliamo vedere come la stringa w è stata prodotta dall'assioma S . come per i parser top-down l'algoritmo può essere descritto (implementato) con tre rappresentazioni diverse:

1. Albero di derivazione;
2. Derivazione right-most;
3. Stack.

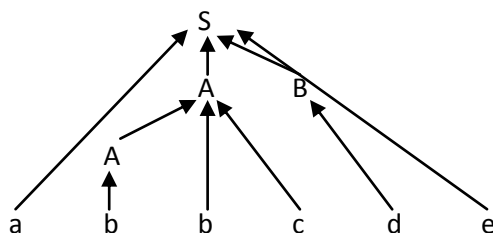
ALBERO DI DERIVAZIONE

L'input lo si legge da sinistra a destra. Partendo dalle foglie, ad ogni passo, bisogna trovare una produzione la cui parte destra coincide con una sottostringa dell'input (o della stringa ottenuta da quella in input per riduzioni) e quindi ridurre tale sottostringa col non terminale alla sinistra della produzione:

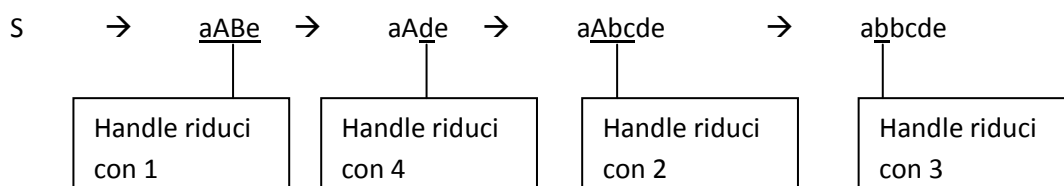
"ab" ha un handle, la "b":



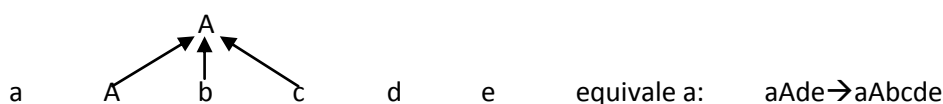
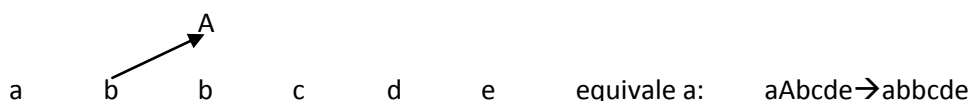
la seconda "b" e la "d" non sono handle poiché le loro riduzioni non portano alla costruzione di una derivazione right-most (quindi alla costruzione dell'albero). Per identificare gli handle si userà una tabella.



DERIVAZIONE RIGHT-MOST



La derivazione si costruisce dall'inizio alla fine (in questo senso \leftarrow). Si nota che le due rappresentazioni hanno lo stesso significato; ad ogni passo dell'una corrisponde un passo dell'altra:



Se ci fossero più handle la grammatica sarebbe ambigua, poiché entrambi garantirebbero la costruzione di una derivazione right-most per una stessa forma sentenziale. Dire che ci sono due diverse derivazioni right-most per una forma sentenziale significa che la grammatica è ambigua.

STACK

STACK	INPUT	AZIONE
\$	abbcde\$	Shift
\$a	bbcdde\$	Shift
\$ab	bcde\$	Reduce con $A \rightarrow b$
\$aA	bcde\$	Shift
\$aAb	cde\$	Shift
\$aAbc	de\$	Reduce con $A \rightarrow Abc$
\$aA	de\$	Shift
\$aAd	e\$	Reduce con $B \rightarrow d$
\$aAB	e\$	Shift
\$aABe	\$	Reduce con $S \rightarrow aABe$
\$S	\$	accept

PREFISSO VITALE

Un prefisso vitale è un prefisso di una forma sentenziale destra che porterà ad un handle. Lo stack conterrà tali prefissi ammissibili e la computazione continua solo se nello stack vi è un prefisso di una forma sentenziale che porta ad S. Se il contenuto dello stack non è prefisso di nessuna forma sentenziale che porta ad S, ci si può fermare; vi sarà un automa finito che ad ogni passo controlla se il contenuto dello stack è un prefisso vitale, cioè un prefisso che può portare al riconoscimento di un handle. Quindi i prefissi vitali sono il contenuto dello stack; se sullo stack ci sono simboli che non portano ad una handle, non porteranno ad S e quindi ci si ferma con un errore.

L'automa finito che sarà in grado di riconoscere i prefissi vitali è costruito a partire dalla grammatica con la tecnica degli item. Quindi l'algoritmo finale di parsing sarà formato da un automa finito ed uno stack che interagiscono fra loro.

Prefissi vitali di $a\underline{b}bcde = \{\epsilon, a, ab\}$; Prefissi vitali di $a\underline{A}bcde = \{\epsilon, a, aA, aAb, aAbc\}$; Prefissi vitali di $a\underline{A}de = \{\epsilon, a, aA, aAd\}$; Prefissi vitali di $\underline{aA}Be = \{\epsilon, a, aA, aAB, aABe\}$; Prefissi vitali di $S = \{\epsilon, S\}$;

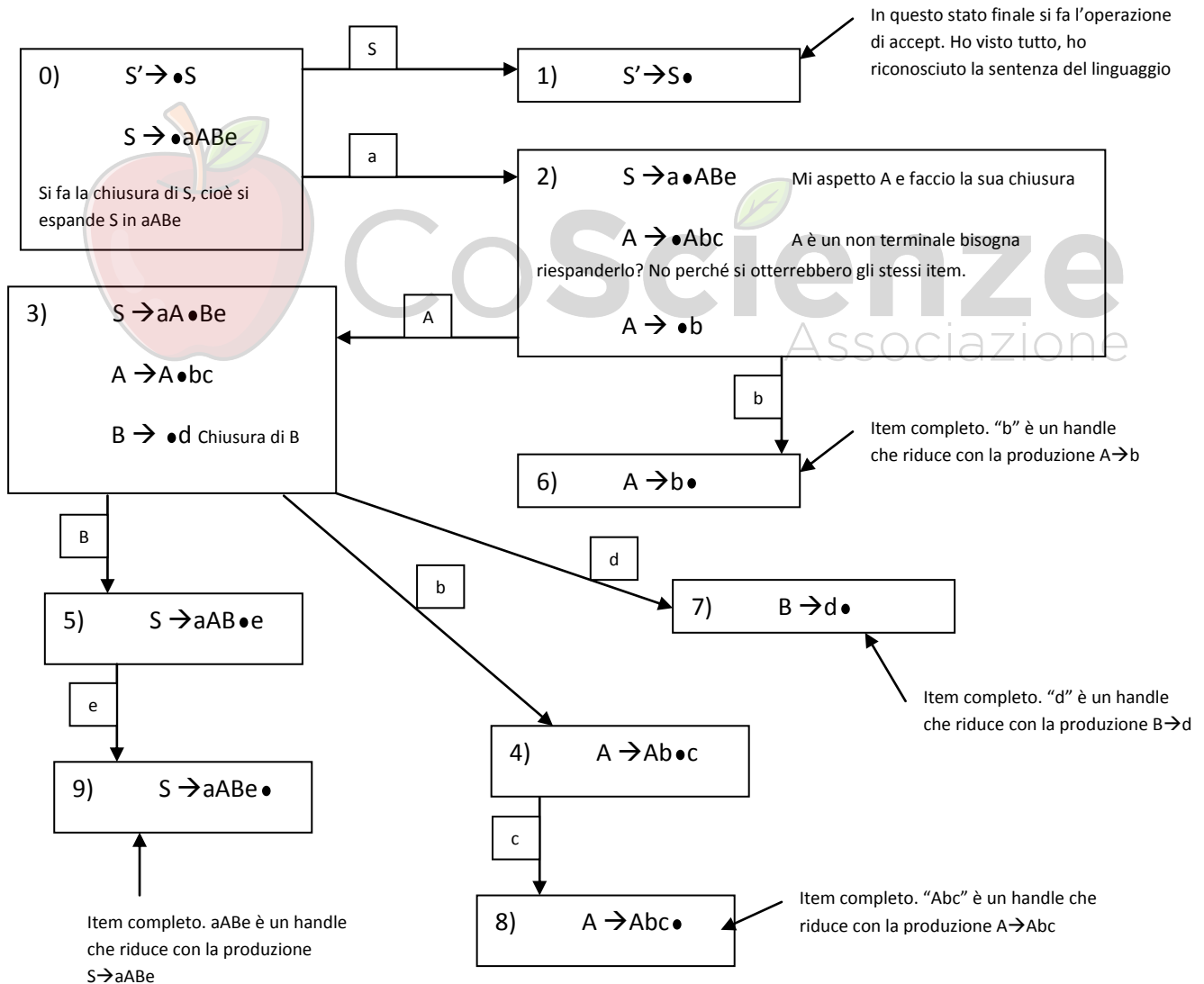
Si può notare che i contenuti dello stack: a, ab, aA, aAb, aAbc, aA, aAd, aAB, aABe, S sono sempre prefissi vitali. Costruiamo ora l'automa finito che riconosce i prefissi vitali. Un item è una produzione con un punto:

$A \rightarrow X \bullet Y$ il cui significato è: "è stato già visto X e si aspetta di vedere Y".

Prima di costruire l'automa si aumenta la grammatica aggiungendo la produzione: $S' \rightarrow S$. Ricordando la vecchia grammatica costruiamo l'automa:

- 1) $S \rightarrow aABe$
- 2) $A \rightarrow Abc$
- 3) $A \rightarrow b$
- 4) $B \rightarrow d$

L'automa:



Tutti gli stati dell'automa sono finali; ogni stato riconosce un prefisso vitale; quindi il linguaggio riconosciuto dall'automa è l'insieme di tutti i prefissi vitali di tutte le forme sentenziali destre. Vediamo i vari prefissi vitali:

- Prefisso vitale riconosciuto dallo stato 0: ϵ ;
- Prefisso vitale riconosciuto dallo stato 1: S;
- Prefisso vitale riconosciuto dallo stato 2: a;
- Prefisso vitale riconosciuto dallo stato 3: aA;
- Prefisso vitale riconosciuto dallo stato 4: aAb;
- Prefisso vitale riconosciuto dallo stato 5: aAB;
- Prefisso vitale riconosciuto dallo stato 6: ab;
- Prefisso vitale riconosciuto dallo stato 7: aAd;
- Prefisso vitale riconosciuto dallo stato 8: aABc;
- Prefisso vitale riconosciuto dallo stato 9: aABe;

Come si può notare ogni stato riconosce un prefisso vitale. Vediamo come avviene il riconoscimento di $w=abde$

STACK	INPUT	AZIONE
\$0a	abde\$	Shift "a"
\$0a2	bde\$	Shift "b"
\$0a2b6	de\$	Reduce con $A \rightarrow b$
\$0a2A3	de\$	Shift "d"
\$0a2A3d7	e\$	Reduce con $B \rightarrow d$
\$0a2A3B5	e\$	Shift "e"
\$0a2A3B5e9	\$	Reduce con $S \rightarrow aABe$
\$0S1	\$	Dallo stato 0 leggendo S si va nello stato di accept

Dopo la riduzione bisognerebbe rifare il processo per vedere se il contenuto dello stack è un prefisso vitale; bisognerebbe vedere se c'è un cammino che dallo stato 0 leggendo aA porta in uno stato valido; si evita di ricominciare ogni volta un cammino dallo stato 0 sfruttando le informazioni nello stack; quindi esaminando "b" si parte dallo stato 2 e, leggendo A, si va nello stato 3.

Quel riconoscimento equivale alla derivazione :

S \rightarrow aABe \rightarrow aAde \rightarrow abde (derivazioni right-most)

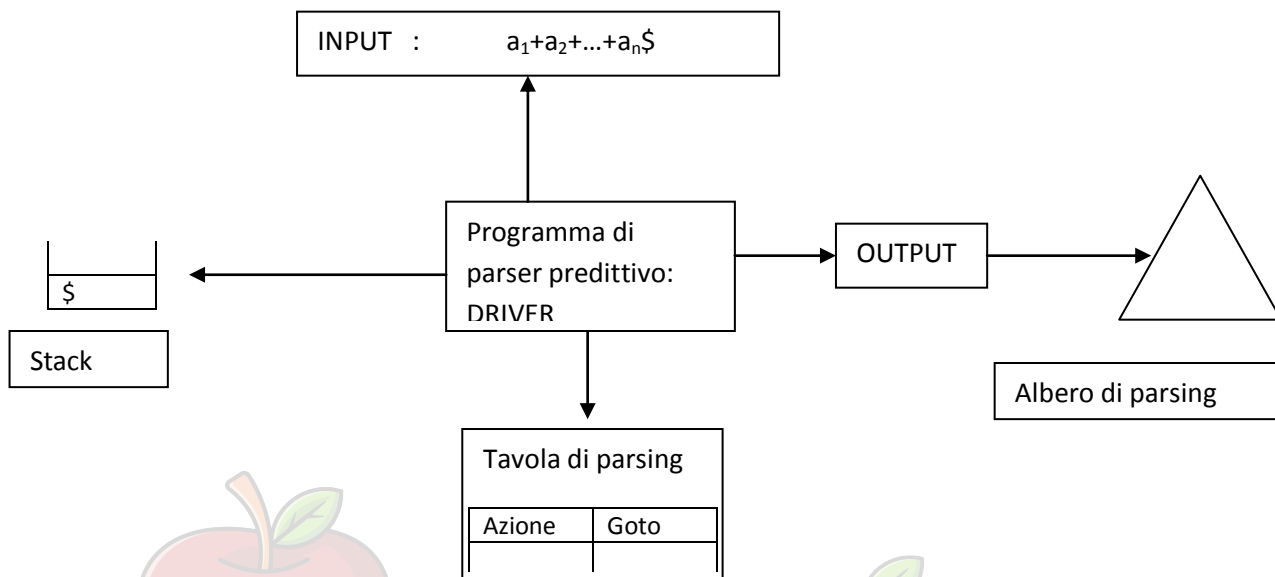
Ci sono grammatiche context free per le quali il parser shift-reduce non può essere usato poiché esso può trovarsi nella situazione in cui conoscendo il contenuto dello stack ed il simbolo in lettura, non può decidere se shiftare o ridurre (conflitto shift-reduce) oppure non può decidere quale delle diverse possibili riduzioni deve fare (conflitto reduce-reduce). Tali grammatiche non sono LR.

LEZIONE 10 – ANALISI SINTATTICA (5 PARTE)

06/04/2009

PARSER LR

Ricordiamo che il ruolo dell'analizzatore sintattico è prendere in input una grammatica ed una sentenza (frase) e dire, come output, se la sentenza appartiene o meno al linguaggio generato dalla grammatica. Lo schema di parser bottom-up è lo stesso di quello dei parser top-down:



Il programma driver è lo stesso per ogni parser LR ed esegue le operazioni: SHIFT, REDUCE, ACCEPT, ERROR. Il programma legge i caratteri da un buffer di input, uno alla volta, ed usa uno stack per immagazzinare una stringa della forma $S_0X_1S_1X_2S_2...X_nS_n$, dove S_n è il top, ogni X_i è un simbolo della grammatica di ogni S_i è uno "stato" (nelle implementazioni i simboli della grammatica non appaiono nello stack, ma per chiarezza li considereremo). La tabella di parsing si costruisce di volta in volta e dipende dalla grammatica che si ha. La tabella è costituita da due parti: "action" e "goto"; essa è indicizzata dagli stati (come righe) e dai simboli della grammatica:

	Sezione "action"	Sezione "goto"
Stati: S_i	a, b e tutti i terminali	A, B e tutti i non terminali
0		
1	"Questa sezione indica quali azioni fare"	"Se bisogna fare riduzione ci dice in quale stato bisogna andare dopo la riduzione"
2		
...		
n		

Tabella di parsing

La tabella di parsing si costruisce a partire dall'automa deterministico che riconosce l'insieme dei prefissi vitali. Essa contiene le quattro possibili azioni (shift, reduce, accept, error). Il programma driver che guida il parser LR si comporta come segue: esso determina S_m , lo stato attualmente sul top dello stack; ed a_i , il simbolo in lettura; consulta $action[S_m, a_i]$ che può avere uno dei quattro valori:

1. Shift S dove S è uno stato;
2. Reduce con la produzione $A \rightarrow \beta$;
3. Accetta;
4. Errore.

La funzione “goto” prende uno stato ed un simbolo della grammatica e produce uno stato. Una “configurazione” di un tale parser è una coppia:

1. La prima componente è il contenuto dello stack;
2. La seconda componente è l’input non ancora consumato ($S_0X_1S_1X_2S_2...X_nS_n,a_ia_{i+1}...a_m\$$).

Tale configurazione rappresenta la forma sentenziale destra: $X_1X_2...X_naia_{i+1}...a_m$. Vediamo come il parser cambia configurazione a seconda dell’azione che si esegue:

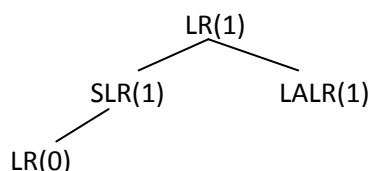
1. Se $action[S_m,a_i] = \text{shift } S$; il parser esegue uno shift entrando nella configurazione: $S_0X_1S_1X_2S_2...X_nS_na_iSa_{i+1}...a_n\$$; il parser ha shiftato sullo stack sia il simbolo in lettura che lo stato indicato in $action[S_m,a_i]$.
2. Se $action[S_m,a_i] = \text{riduci } A \rightarrow \beta$; il parser esegue una riduzione entrando nella configurazione $S_0X_1S_1X_2S_2...X_{m-r}S_{m-r}AS,a_{i+1}...a_n\$$ dove $S = \text{goto}[S_{m-r},A]$ ed “r” è la lunghezza di β . In questo caso il parser estrae dallo stack $2*r$ simboli (“r” stati ed “r” simboli della grammatica) lasciando lo stato S_{m-r} sul top; poi inserisce “A” ed “S” nello stack. Per i parser LR che costruiamo la sequenza $X_{m-r+1}...X_m$ estratta dallo stack fa match con β . L’output prodotto è rappresentato dalla produzione per la riduzione.
3. Se $action[S_m,a_i] = \text{accetta}$: l’analisi è completata con successo.
4. Se $action[S_m,a_i] = \text{errore}$: significa che il parser ha scoperto un errore e sarà chiamata una routine di ricopertura dell’errore;

L’analisi sintattica LR(k) è un’efficiente tecnica di analisi sintattica bottom-up che può essere usata per analizzare una grossa classe di grammatica context-free. L sta per scansione dell’input da sinistra a destra; R sta per costruzione di una derivazione right-most in senso inverso; K sta per il numero di simboli di lookahead in input, usati per prendere le decisioni durante l’analisi.

L’analisi LR è scelta per diverse ragioni:

- I parser LR possono essere costruiti per riconoscere tutti i costrutti dei linguaggi di programmazione per i quali è possibile scrivere grammatiche context free.
- Il metodo di analisi LR è il più generale metodo di analisi shift-reduce senza ritorno indietro.
- La classe delle grammatiche che possono essere analizzate usando i metodi LR è un grosso sottoinsieme proprio della classe delle grammatiche che possono essere analizzate con un parser predittivo.

Abbiamo detto che la parsing table si costruisce a partire dalla grammatica; vedremo quattro tipi di parsing table corrispondenti a quattro diverse classi di grammatiche: LR(0), SLR(1), LR(1), LALR(1).



L’efficienza è sempre $O(n)$ nel caso deterministico.

Tale rappresentazione ci dice che:

- La LR(0) è la parsing table più semplice ma è anche quella che presenta più facilmente conflitti;
- La SLR(1) elimina dei conflitti presenti nella LR(0), ma è più complessa; comunque può presentare dei conflitti;
- La LR(1) è la più potente nel senso che si eliminano la maggior parte dei conflitti; però la grammatica perde il suo potere descrittivo (motivo per il quale sono state introdotte le grammatiche); inoltre la parsing table può essere molto complessa (molti stati) e molto costosa.
- LALR(1) è la più utilizzata. Poiché l'LR(1) è troppo costosa in termini di stati si rinuncia ad identificare tutti i conflitti accorpendo alcuni stati del diagramma che hanno item in comune.

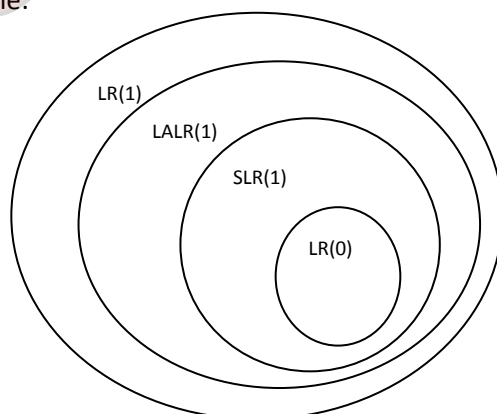
Vedremo le quattro tecnologie, si parte sempre dall'automa deterministico che riconosce i prefissi vitali. Nell'esempio precedente abbiamo considerato una grammatica LR(0). Per ciascuna sigla si possono definire:

- Gli item: gli item possono essere di tipo LR(0), SLR(1), LR(1), LALR(1).
- La grammatica: una grammatica è LR(1) se la parsing table LR(1) costruita a partire dalla grammatica non presenta conflitti. Analogamente per le altre sigle.
- I linguaggi: uno stesso linguaggio può essere generato da molte grammatiche. Un linguaggio è LR(0) (SLR(1), LR(1), LALR(1)) se esiste almeno una grammatica LR(0) (SLR(1), LR(1), LALR(1)) che lo genera.

Se abbiamo una certa grammatica G_1 che non è LR(1) non possiamo concludere niente sul linguaggio L_1 generato da G_1 ; non possiamo dire che L_1 non è LR(1) poiché potrebbe esistere un'altra grammatica G_2 che è LR(1) e che genera L_1 .

Una grammatica LR(0) è sicuramente anche SLR(1).

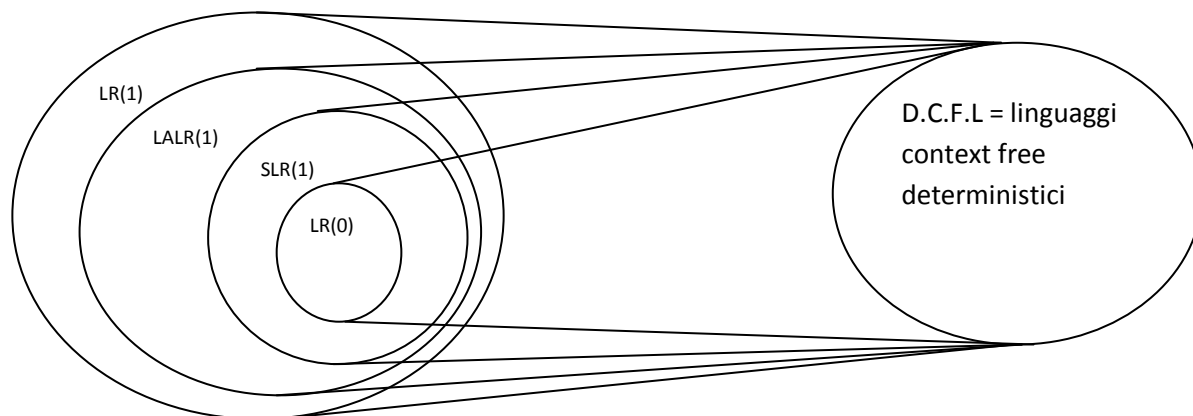
Classi di grammatiche:



Quindi se un linguaggio è LR(0) sarà anche SLR(1), LALR(1), LR(1). Ci possono essere più grammatiche, anche in una stessa classe, che generano uno stesso linguaggio. Un linguaggio è inerentemente ambiguo se ogni grammatica che lo genera è ambigua (porterà a parser non deterministici, non li vedremo).

- ✓ LR(0): l'azione si decide senza guardare il prossimo input.
- ✓ SLR(1): semplice LR(1).
- ✓ LR(1): è l'LR canonico e cerca di eliminare tutti i possibili conflitti.
- ✓ LALR(1): fonde alcuni stati dell'LR canonico. Look-ahead LR.

Tali grammatiche generano tutte la stessa classe di linguaggi:



Più aumento il lookahead più si eliminano gli eventuali conflitti e si complicano le grammatiche. Perché non limitarci alle più semplici LR(0) visto che hanno tutte lo stesso potere generativo?

La tecnica LR(0) è quella che più frequentemente porta ad un parser che presenta dei conflitti, non deterministico, ma vogliamo un parser deterministico, senza back-tracking e quindi bisogna eliminare i conflitti; per eliminare i conflitti si deve modificare la grammatica stravolgendone il significato, ma le grammatiche nascono per descrivere in modo chiaro, semplice, un linguaggio per non ha senso avere una grammatica poco comprensibile. Quindi si preferisce non cambiare la grammatica, ma la tecnica di costruzione della parsing table; anche se la si complica un poco, ma si è salvata la semplicità della grammatica. I linguaggi inerentemente ambigui corrispondono alla classe dei N.D.C.F.L.

Vediamo ora come costruire le varie parsing table: data una grammatica non sappiamo di che tipo è; se costruiamo una parsing table LR(0) e non presenta conflitti allora la grammatica è LR(0), se presenta conflitti non è LR(0); se costruiamo una parsing table SLR(1) e non presenta conflitti allora la grammatica è SLR(1) altrimenti, se presenta conflitti, non è LR(1). E così via.

Vediamo le varie tecniche per costruire le:

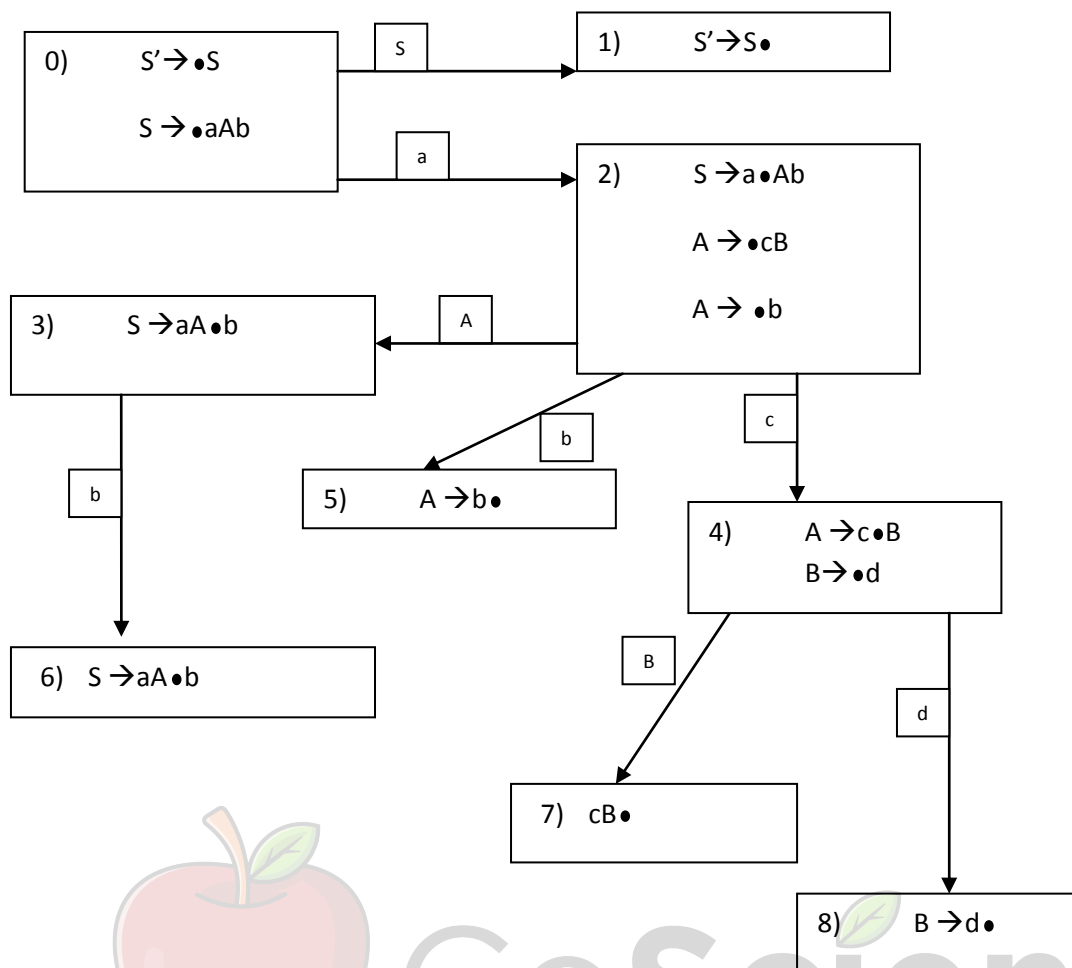
- Parsing table LR(0);
- Parsing table SLR(1);
- Parsing table LR(1);
- Parsing table LALR(1);

PARSING TABLE LR(0)

Consideriamo la grammatica:

1. $S \rightarrow aAb$
2. $A \rightarrow cB$
3. $A \rightarrow b$
4. $B \rightarrow d$

Come prima cosa si aggiunge la produzione $S' \rightarrow S$ e poi si costruisce l'automa che riconosce tutti i prefissi vitali.



Per verificare la correttezza dell'automa bisogna verificare se ogni item completo (stato 1,5,6,7,8) identifica una produzione della grammatica. Vediamo la tabella di parsing:

	ACTION (terminali)					GOTO (non terminali)			
	a	b	c	d	\$	S'	S	A	B
0	S2 (nello stato 0 vedo "a" e shift nello stato 2)						1 (si va in uno stato senza fare shift; ci sono solo transizioni)		
1					Accept				
2		S5	S4					3	
3		S6							
4				S8					7
5	R3	R3	R3	R3	R3				
6	R1	R1	R1	R1	R1				
7	R2	R2	R2	R2	R2				
8	R4	R4	R4	R4	R4				

Con Ri si indicano le riduzioni: "riduci usando la riduzione Ri". Negli altri casi (celle vuote) c'è errore.

Quando si parla di look-ahead 0,1,2... ci si riferisce solo alle azioni di reduce per gli item completi; per le operazioni di shift si ha sempre look-ahead 1 (per ogni tecnica) cioè in funzione del successivo simbolo in input si decide in quale stato transire.

L'LR(0) non calcola il look-ahead nel caso della reduce degli item completi. Vi è un look-ahead per gli shift:

- Quindi gli shift si fanno guardando il prossimo simbolo in input (look-ahead 1 per lo shift).
- Le reduce si fanno senza considerare il prossimo simbolo in input (look-ahead 0).

La differenza delle quattro tecniche di costruzione della parsing table è nel come calcolano il look-ahead sugli item completi, come decidere su quali simboli fare la riduzione; lo shift è fatto sempre allo stesso modo. La differenza tra i metodi non è nella chiusura e nel goto, ma nel calcolo del look-ahead per gli item completi. Ciò che distingue le varie tecniche è il look-ahead per le reduce.

Nell'LR(0) dopo il \bullet , degli item completi, non sappiamo quale deve essere il prossimo simbolo da leggere, quindi è prevista una reduce qualsiasi sia il prossimo simbolo in input (quindi look-ahead(0)). Le reduce si fanno solo sugli item completi. Per gli item non completi vi sono due possibilità:

- 1) Se il punto è seguito da un terminale si fa uno shift;
- 2) Se il punto è seguito da un non terminale si fa un goto.

Se il punto è seguito da un simbolo vi è sempre un look-ahead qualsiasi sia la tecnica usata. Quindi nella parsing table LR(0) si riesce ad essere dettagliati per gli shift ed i goto, mentre i reduce non sono dettagliati per i simboli: vengono fatti qualsiasi sia il simbolo in input (poiché non si calcola il look-ahead). Si nota che le reduce si fanno sempre anche su simboli errati, eventuali errori di reduce si evidenziano nei passi successivi.

Vediamo ora l'algoritmo che permette di costruire la tabella LR(0) formalmente:

Algoritmo: costruzione della tabella di analisi LR(0);

Input: la grammatica aumentata G' (con $S' \rightarrow S$);

Output: la tabella LR con le funzioni ACTION e GOTO per G' ;

Metodo:

- Costruire $C = \{I_0, I_1, \dots, I_n\}$ la collezione degli insiemi di item LR(0) per G' (in pratica gli stati dell'automa).
- Lo stato "i" è costruito da I_i . Le azioni dallo stato "i" sono determinate come segue:
 - Se $[A \rightarrow \alpha \bullet a \beta]$ è in I_i allora settiamo $action[i, a] = "S_j"$ (shift j) dove a è un terminale;
 - Se $[A \rightarrow \alpha \bullet]$ è in I_i , allora settiamo $action[i, a] = "reduce con A \rightarrow \alpha"$ per tutte le "a" che sono terminali con $A \neq S$;
 - Se $[S' \rightarrow S \bullet]$ è in I_i , allora settiamo $action[i, \$] = "accetta"$;se tali regole generano dei conflitti, allora diciamo che G non è LR(0) e l'algoritmo non produce un parser deterministico.
- Le transizioni goto per lo stato "i" sono costruite per tutti i non terminali "A" usando la regola: se $goto(I_i, A) = I_j$ allora settiamo $goto[i, A] = "j"$;
- Tutti gli elementi non definiti dalle regole 2 e 3 sono errori;
- Lo stato iniziale del parser è quello costruito dall'insieme degli item contenenti $[S' \rightarrow S]$;

La tabella così ottenuta è detta tabella LR(0) per G. Un parser che usa tale tabella è detto LR(0); una grammatica che ha una tale tabella è detta LR(0).

LEZIONE 11 - ESERCITAZIONE

08/04/2009

Costruire un parser top-down considerando la grammatica siffatta:

$N=\{S,L\}; \quad T=\{(\,,\,+\,,a\};$

$P=\{$
1. $S \rightarrow (L);$
2. $S \rightarrow (a);$
3. $L \rightarrow L+S;$
4. $L \rightarrow S;$
 $\};$

Nota bene: applicare trasformazioni se necessario.

SOLUZIONE:

Esaminando la grammatica si può notare che sono presenti due problemi: ricorsione a sinistra e fattorizzazione a sinistra. La ricorsione a sinistra è evidente nelle produzioni 3 e 4; la fattorizzazione è evidente nelle produzioni 1 e 2. Eliminiamo i problemi rendendo così la grammatica non ambigua.

Per eliminare la ricorsione sinistra trasformiamo la grammatica in questo modo:

$P=\{$
1. $S \rightarrow (L);$
2. $S \rightarrow (a);$
3. $L \rightarrow SL';$
4. $L' \rightarrow +SL' \mid \epsilon;$
 $\}$

Qui rimane ancora il problema della fattorizzazione a sinistra. Andiamo a rimuoverla. Trasformiamo la grammatica precedente in questa:

$P=\{$
1. $S \rightarrow (S';$
2. $S' \rightarrow L) \mid a);$
3. $L \rightarrow SL';$
4. $L' \rightarrow +SL' \mid \epsilon;$
 $\}$

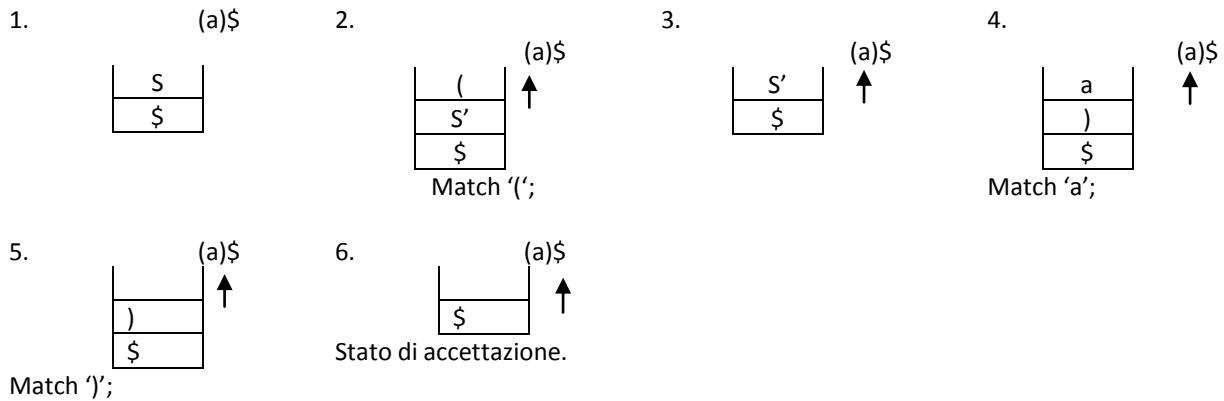
Ora per costruire l'albero top-down si deve costruire la tavola di parsing; per costruire e, in particolar modo, per riempire la tavola di parsing è necessario calcolare FIRST e FOLLOW:

$FIRST(S') = \{ (\}; \quad FIRST(L) = FIRST(L) = FIRST(S) = \{ (\}; \quad FIRST(a) = \{ a \}; \quad FIRST(SL') = FIRST(S) = \{ (\};$
 $FIRST(+SL') = FIRST(+)= \{ + \}; \quad FIRST(\epsilon) = FOLLOW(L') = FOLLOW(L) = \{) \};$

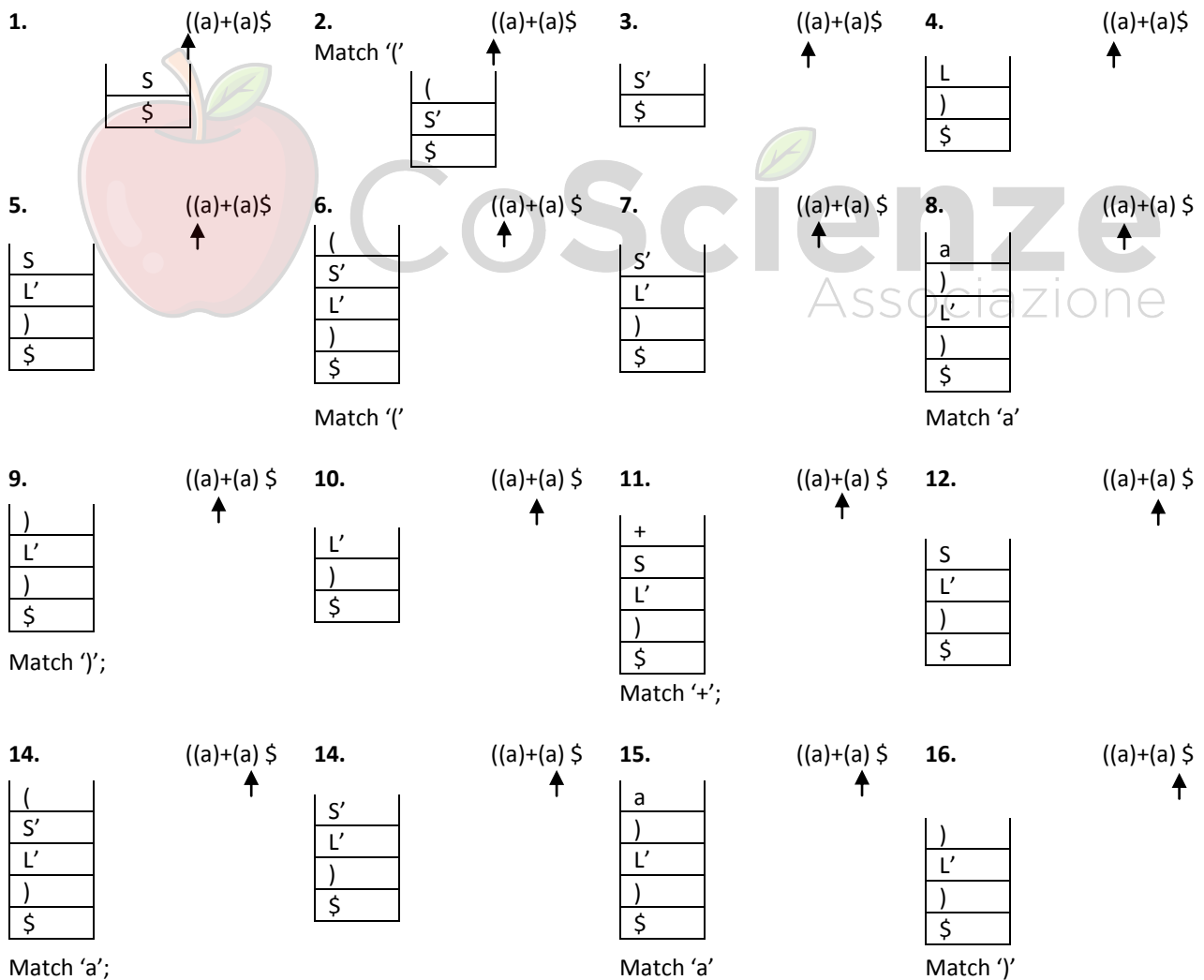
	()	+	a	\$
S	$S \rightarrow (S'$				
S'	$S' \rightarrow L)$			$S' \rightarrow a)$	
L	$L \rightarrow SL'$				
L'		$L' \rightarrow \epsilon$	$L' \rightarrow +SL'$		

Per verificare che la parsing table sia corretta è bene verificarla con due stringhe: una deve essere accettata dalla grammatica e quindi dalla parsing table; un'altra non deve essere accettata dalla grammatica e neanche dalla parsing table.

Consideriamo le due parole: (a) accettata dalla grammatica e ((a)+(a);



Vediamo l'altro caso: ((a)+(a)



17. ((a)+(a) \$

L'	↑
)	
\$	

Stato di errore. Non c'è match tra \$ e L'.

Questa grammatica è LL(1) e avremo un parser di tipo LL(1) perché abbiamo potuto costruire una parsing table per quella grammatica. Vi è anche un altro modo di costruire la parsing table. Si può verificare se una grammatica è LL(1) confrontandola con le tre condizioni già dette in una lezione. Se la grammatica non è LL(1) bisogna trasformarla ed eliminare i vari problemi presentatisi.

SECONDO ESERCIZIO

Considerando la grammatica:

P={

1. $S \rightarrow rA|StI$;
2. $S \rightarrow \varepsilon$;
3. $A \rightarrow aA$;
4. $A \rightarrow \varepsilon$;
5. $I \rightarrow iI$;
6. $I \rightarrow \varepsilon$;

}

Le tre regole che ho prima citato sono le seguenti:

1. Nella grammatica per ogni $A \rightarrow \beta$ e $A \rightarrow \alpha$ deve risultare:
 $FIRST(\alpha) \cap FIRST(\beta) = \text{VUOTO}$; Questa regola è rispettata nella nostra grammatica.
 $FIRST(S) \cap FIRST(S) = \text{VUOTO}$; $FIRST(A) \cap FIRST(A) = \text{VUOTO}$; $FIRST(I) \cap FIRST(I) = \text{VUOTO}$;
2. Avendo $A \rightarrow \alpha$ e $A \rightarrow \beta$ non deve essere possibile che $\alpha \rightarrow^* \varepsilon$ e $\alpha \rightarrow^* \beta$; quindi $FIRST(\alpha) \cap FIRST(\beta) = \text{VUOTO}$; Anche questa regola è rispettata dalla nostra grammatica.
 $FIRST(S) \cap FIRST(S) = \{\varepsilon\}$; $FIRST(A) \cap FIRST(A) = \{\varepsilon\}$; $FIRST(I) \cap FIRST(I) = \{\varepsilon\}$;
3. Avendo $A \rightarrow \alpha | \beta$ se $\beta \rightarrow^* \varepsilon$ allora α non deriva alcuna stringa che inizia con un terminale in $FOLLOW(A)$: se $\beta \rightarrow^* \varepsilon$ allora $FIRST(\alpha) \cap FOLLOW(A) = \emptyset$. Lo stesso vale se $\alpha \rightarrow^* \varepsilon$; Anche questa regola è rispettata dalla nostra grammatica.
 $FOLLOW(S) = \{t\}$; $FIRST(rA|StI) = \{r\}$; $FOLLOW(S) \cap FIRST(r...I) = \text{VUOTO}$;
 $FOLLOW(A) = FIRST(I) \cup \varepsilon$; $FIRST(S) \cup \varepsilon$; $FIRST(T) = \{i, r, t\}$; $FIRST(aA) = \{a\}$; $FOLLOW(A) \cap FIRST(aA) = \text{VUOTO}$;
 $FOLLOW(I) = FIRST(S) \cup \varepsilon$; $FIRST(T) = \{r, t\}$; $FIRST(iI) = \{i\}$; $FOLLOW(I) \cap FIRST(iI) = \text{VUOTO}$; Anche questa regola è rispettata dalla nostra grammatica.

Andiamo ora a costruire e riempire la parsing table:

	r	a	t	i	\$
S	$S \rightarrow rA StI$		$S \rightarrow \varepsilon$		$S \rightarrow \varepsilon$
A	$A \rightarrow \varepsilon$	$A \rightarrow aA$	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$	
I	$I \rightarrow \varepsilon$		$I \rightarrow \varepsilon$	$I \rightarrow iI$	$I \rightarrow \varepsilon$

$FIRST(S \rightarrow r...I) = \{r\}$;

$FIRST(S \rightarrow \varepsilon) = FOLLOW(S) \cup \varepsilon = \{t, \$\}$;

$FIRST(aA) = \{a\}$

$FIRST(\varepsilon) = FOLLOW(A) = FIRST(I) \cup \varepsilon$; $FIRST(S) \cup \varepsilon$; $FIRST(t) = \{i, r, t\}$;

$FIRST(iI) = \{i\}$;

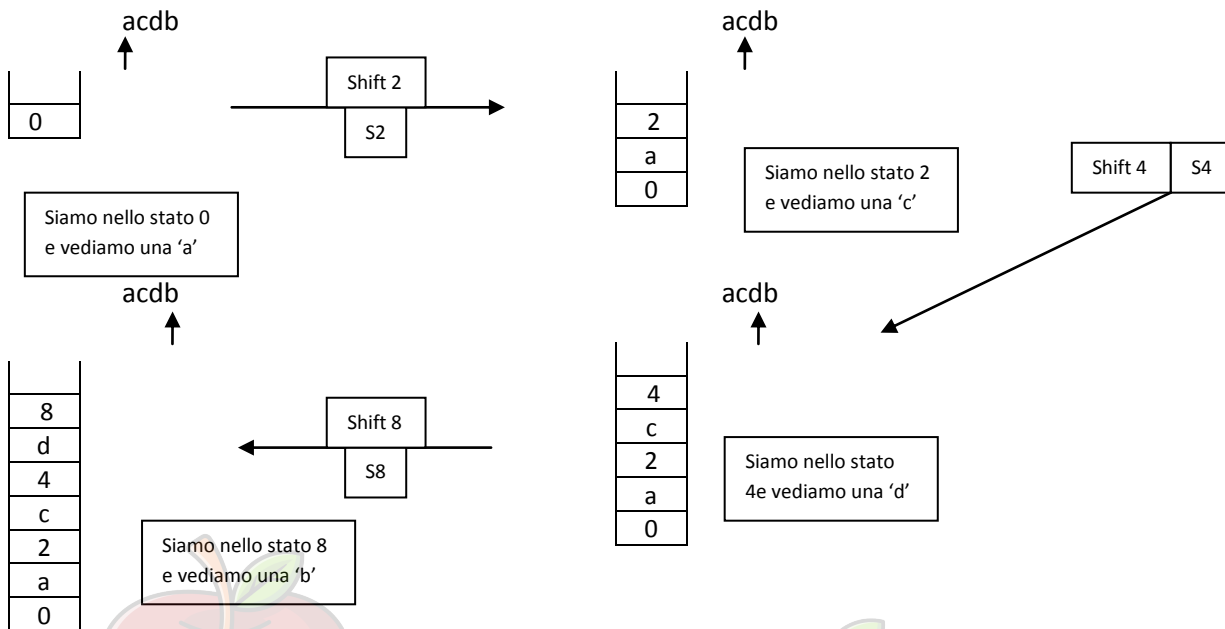
$FIRST(\varepsilon) = FOLLOW(S) = FIRST(t) \cup \varepsilon$; $FOLLOW(I) = \{r, t, \$\}$.

LEZIONE 12 – ANALISI SINTATTICA (6 PARTE)

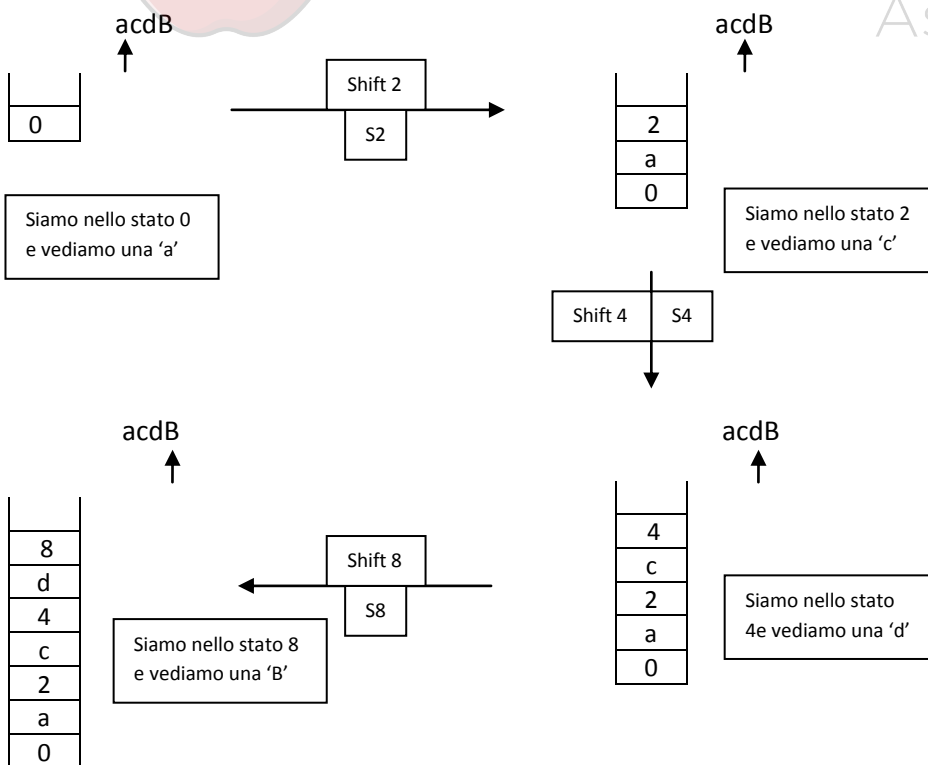
16/04/2009

PARSING TABLE SLR(1)

Vediamo come essere più precisi sulle reduce. Consideriamo un esempio: se in input abbiamo acdb e applichiamo l'algoritmo cosa succede????



La parsing table LR(0) per lo stato 8 prevede una riduzione qualsiasi sia il prossimo simbolo in input; se si commette un errore ce ne si accorge solo alla fine. Infatti vediamo un esempio con errore:



A questo punto nello stato 8 facciamo una riduzione. Appliciamo la riduzione $R_4: B \rightarrow D$

B
c
2
a
0

Facciamo un'ulteriore riduzione applicando $R_2: A \rightarrow cB$. Solo alla fine ci accorgiamo che c'è un errore. Vediamo ora come è l'algoritmo per la costruzione della tabella SLR(1).

Algoritmo: costruzione della tabella di analisi SLR.

Input: la grammatica aumentata G' (con $S' \rightarrow S$).

Output: la tabella SLR con le funzioni ACTION e GOTO per G' .

Metodo:

- Costruire $C = \{I_0, I_1, \dots, I_n\}$ la collezione degli insiemi di item LR(0) per G' (in pratica gli stati dell'automa).
- Lo stato "i" è costruito da I_i . Le azioni dallo stato "i" sono determinate come segue:
 - Se $[A \rightarrow \alpha \bullet a\beta]$ è in I_i e $\text{goto}(I_i, a) = I_j$ allora settiamo $\text{action}[i, a] = "S_j"$ (shift j) dove a è un terminale;
 - Se $[A \rightarrow \alpha \bullet]$ è in I_i , allora settiamo $\text{action}[i, a] = "reduce con A \rightarrow \alpha"$ per tutte le "a" $\in \text{FOLLOW}(A)$ dove $A \neq S$;
 - Se $[S' \rightarrow S]$ è in I_i , allora settiamo $\text{action}[i, \$] = "accetta"$;
 se tali regole generano dei conflitti, allora diciamo che G non è SLR(1) e l'algoritmo non produce un parser deterministico.
- Le transizioni goto per lo stato "i" sono costruite per tutti i non terminali "A" usando la regola: se $\text{goto}(I_i, A) = I_j$ allora settiamo $\text{goto}[i, A] = "j"$;
- Tutti gli elementi non definiti dalle regole 2 e 3 sono errori;
- Lo stato iniziale del parser è quello costruito dall'insieme degli item contenenti $[S' \rightarrow S]$;

La tabella così ottenuta è detta tabella SLR(1) per G. Un parser LR che usa tale tabella è detto parser SLR(1) per G; una grammatica che usa tale tabella è detta SLR(1). Ogni grammatica SLR(1) è non ambigua ma ci sono grammatiche non ambigue che non sono SLR(1).

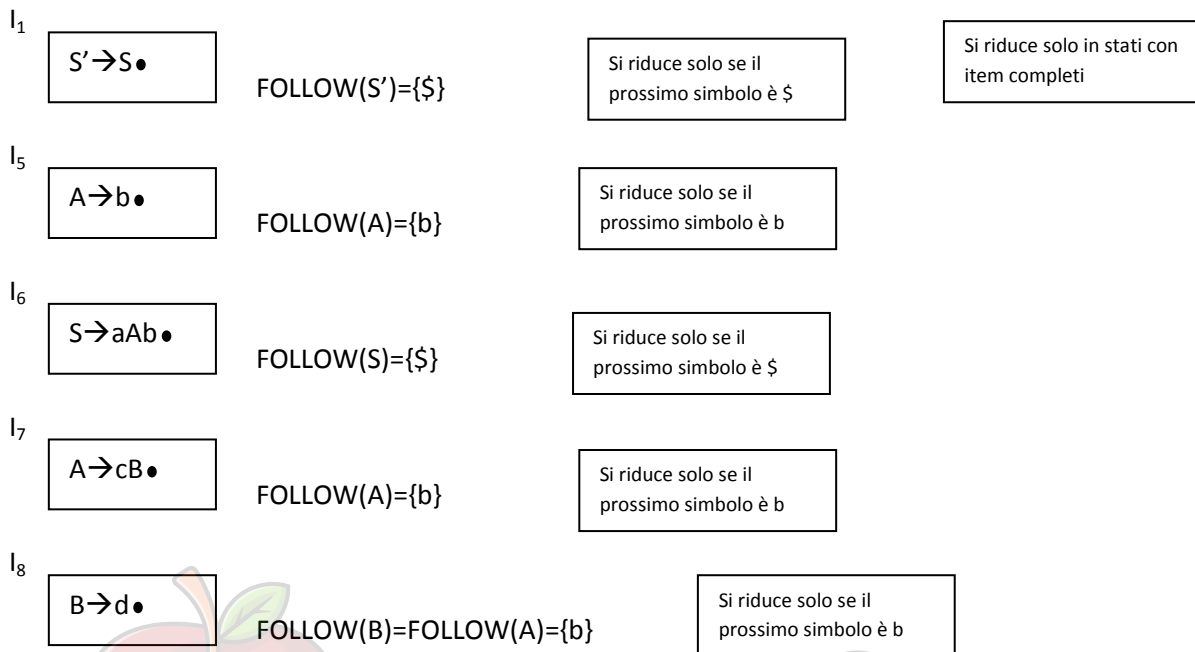
Se avessimo un look-ahead per i reduce potremmo dire subito se vi è un errore o meno, guardando (look-ahead) il prossimo simbolo in input. Vediamo come ottenere informazioni sui simboli ammissibili dopo il punto degli item completi: se dallo stato 8 si effettua una riduzione con la regola 4 si passa dalla forma sentenziale "acd" alla precedente "acB" (considerando sempre una derivazione right-most), cioè: $acB \rightarrow acd$??? Quindi affinché l'input che stiamo leggendo sia corretto è necessario che il simbolo che segue d deve appartenere al FOLLOW(B).

Si passa dall'LR(0) all'SLR(1) mettendo come look-ahead i FOLLOW dei non terminali che sono la parte sinistra degli item completi. L'item completo $S \rightarrow aAb \bullet$ avrà come look-ahead i simboli in FOLLOW(S). Si passa così da riduzioni incontrollate (LR(0)) a riduzioni solo sui simboli che possono seguire il non terminale (SLR(1)).

Passiamo dall'LR(0) all'SLR(1) calcolando come visto il look-ahead per gli item incompleti; queste informazioni vengono inserite nel dfa che riconosce i prefissi vitali.

Consideriamo la grammatica:

1. $S \rightarrow aAb$
2. $A \rightarrow cB$
3. $A \rightarrow b$
4. $B \rightarrow d$



A questo punto vediamo le modifiche che si hanno sulla parsing table:

	ACTION (terminali)					GOTO (non terminali)				
	a	b	c	d	\$	S'	S	A	B	
0	S2						1			
1					Accept					
2		S5	S4					3		
3		S6								
4				S8					7	
5		R3								
6					R1					
7		R2								
8		R4								

Si sono così dettagliate le Reduce sui singoli simboli. Si evitano in questo modo di fare riduzioni che portano ad errori; un importante miglioramento è che possono essere stati eliminati eventuali conflitti. Ad esempio se dallo stato 8 incontrando 'd' erano previste altre operazioni (shift) si è eliminato questo conflitto (nell'LR(0)) poiché ora la riduzione dallo stato 8 la si fa solo se il prossimo simbolo in input è 'b'. Quindi passando dall'LR(0) all'SLR(1) si sono eliminati possibili conflitti; quindi si ha un miglioramento. Si è avuto un miglioramento rispetto l'LR(0) ma è possibile dare informazioni ancora più precise per eliminare possibili conflitti ancora presenti.

PARSING TABLE LR(1)

Nell'SLR(1) si calcolano, per gli item completi, i FOLLOW generali, relativi alla grammatica. Considerando che gli stati costituiscono dei contesti particolari, dei punti particolari della fase di parsing, si osserva che per particolari stati può accadere che alcuni simboli nel FOLLOW non sono ammissibili; quindi si può

ottenere un ulteriore miglioramento (nell'LR(1)) calcolando per gli item completi un **FOLLOW CONTESTUALE**, cioè quel sottoinsieme del FOLLOW costituito dai simboli che in quel particolare stato possono seguire il non terminale che è la parte sinistra dell'item completo. Si effettua un ulteriore distinzione tra i simboli del FOLLOW. Questi miglioramenti sono necessari per risolvere alcuni conflitti. Effettuiamo quindi una breve panoramica tra i conflitti che si possono incontrare; in una parsing table si possono avere due tipi di conflitti:

- Shift-reduce;
- Reduce-reduce;

Ad esempio:

4) $A \rightarrow a \bullet bx$

$Z \rightarrow c \bullet$ riduci con la 3

b_5

	a	b	c	\$
...				
4	R_3	R_3/S_5	R_3	R_3

Con la rispettiva parsing table LR(0). In questa parsing table abbiamo un conflitto di tipo shift-reduce e possiamo affermare che la grammatica non è LR(0). Ora invece costruiamo la parsing table SLR(1) supponendo che $FOLLOW(Z) = \{a, c\}$ e quindi 'b' non appartiene al $FOLLOW(Z)$. In questo caso la parsing table SLR(1) diventa:

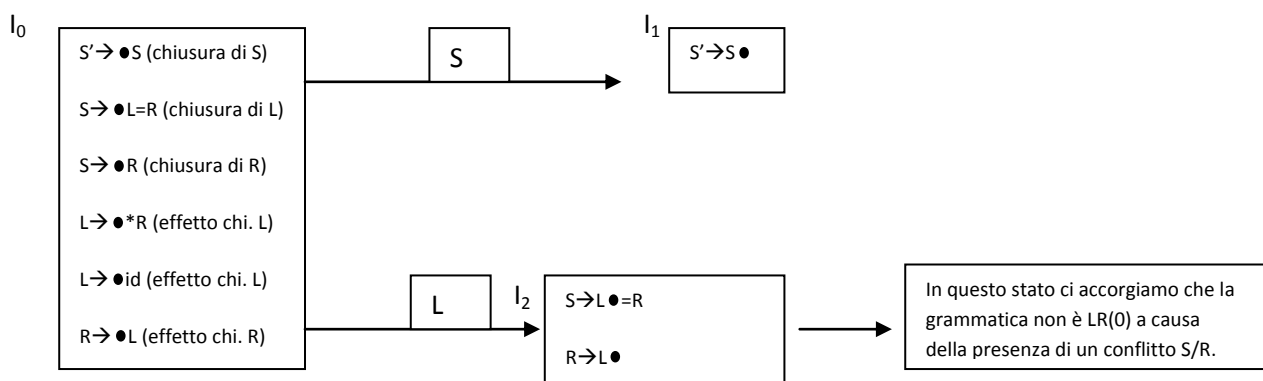
	a	b	c	\$
...				
4	R_3	S_5	R_3	

La riduzione con la produzione 3 si fa solo se il prossimo simbolo in input è un simbolo che può seguire Z in qualsiasi contesto, quindi solo per 'a' e 'c'. La parsing table SLR(1) non presenta conflitti quindi la grammatica è SLR(1). Se però b appartiene a $FOLLOW(Z)$ non si eliminerebbe il conflitto e la grammatica non sarebbe SLR(1). Data una grammatica abbiamo visto che ci sono tre regole per verificare se è LR(1) o meno, non c'è bisogno di costruire la parsing table LR(1). Per le grammatiche LR(0), SLR(1) non ci sono metodi diretti per definirle; l'unico modo per vedere se una grammatica è LR(0) (SLR(1)) è costruire la relativa parsing table LR(0) (SLR(1)) e vedere se presenta conflitti. Se non è presente alcun conflitto la grammatica data è LR(0) (SLR(1)) altrimenti non lo è. Vediamo un esempio.

Data la grammatica:

1. $S \rightarrow L=R$
2. $S \rightarrow R$
3. $L \rightarrow *R$
4. $R \rightarrow L$
5. $L \rightarrow id$

Innanzitutto aumentiamo la grammatica e poi costruiamo l'automa che riconosce i prefissi vitali:



Proviamo a vedere la parsing table LR(0):

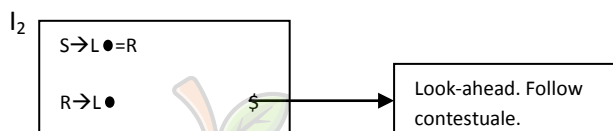
	=	Id	*	\$
...				
2	R_4/S_3	R_4	R_4	R_4

Vediamo invece la parsing table SLR(1). Per costruirla dobbiamo calcolare i follow.

$\text{Follow}(R) = \text{Follow}(S) \cup \text{Follow}(L) = \{\$ \} \cup \{=\} = \{ \$, = \}$. Poiché $"=" \in \text{Follow}(R)$ la grammatica non è SLR(1), in quanto non si risolve il conflitto nella parsing table SLR(1):

	=	Id	*	\$
...				
2	R_4/S_3			R_4

Vediamo come la parsing table LR(1) elimina tale conflitto. Si osserva che vi è un unico modo per arrivare all'item $R \rightarrow L \bullet$ dello stato 2: andando a ritroso $R \rightarrow L \bullet$, $R \rightarrow \bullet L$, $S \rightarrow \bullet R$, $S' \rightarrow \bullet S$ quindi l'unico percorso che ha portato all'item $R \rightarrow L \bullet$ dello stato 2 è: $S' \$ \rightarrow S \$ \rightarrow R \$ \rightarrow L \$$ che ci dice che L, in questo contesto, può essere seguito solo dal \$; quindi, in questo contesto, $"="$ non può mai seguire la L. Sfruttando tale conoscenza si costruisce:

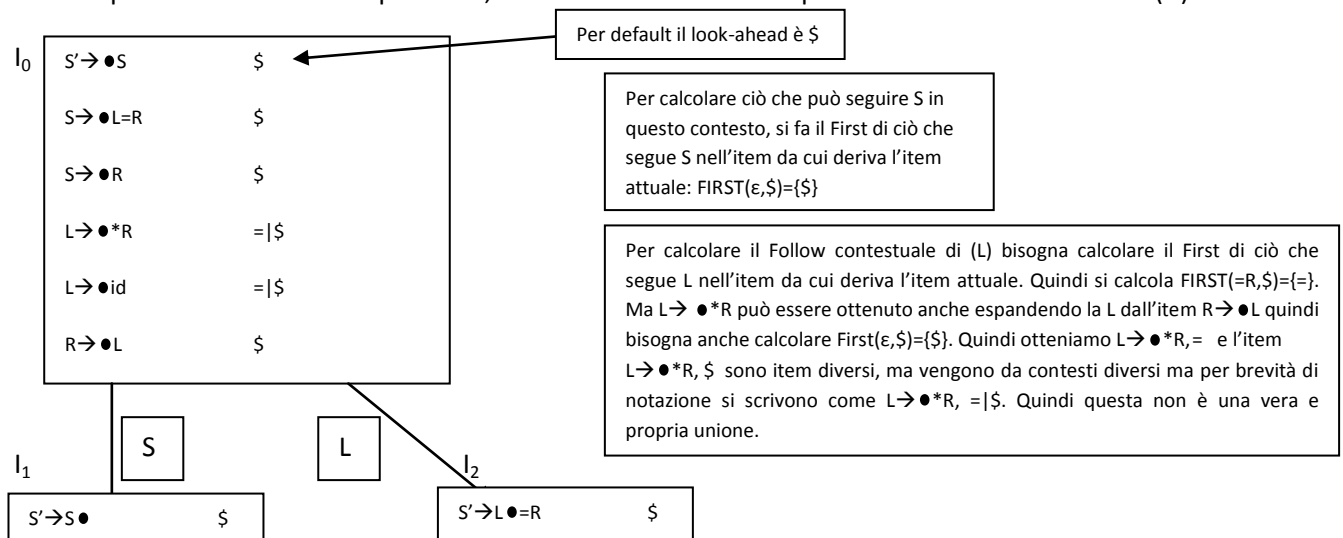


Si rompe così il conflitto ottenendo una parsing-table LR(1) senza conflitti:

	=	Id	*	\$
...				
2	S_3			R_4

Quindi se riusciamo a calcolare i FOLLOW CONTESTUALI si eliminano la maggior parte dei conflitti e si ottiene la parsing table LR(1):

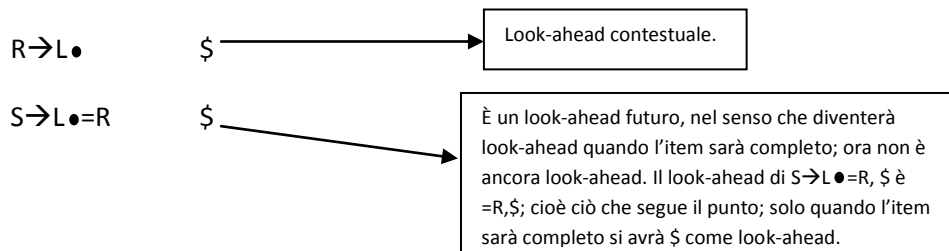
- Un item LR(1) è della forma $[A \rightarrow \alpha \bullet \beta, a]$ dove "a" è un non terminale oppure \$. 1 è la lunghezza della "a" detto look-ahead degli item. Il look-ahead non ha effetto su un item della forma $[A \rightarrow \alpha \bullet \beta, a]$ dove $\beta \neq \epsilon$; su item della forma $[A \rightarrow \alpha \bullet, a]$ produce una riduzione con la regola $A \rightarrow \alpha$ solo se il prossimo simbolo in input è "a", l'insieme di tali "a" è sempre un sottoinsieme di $\text{Follow}(A)$.



$L \rightarrow \bullet R$ può avere due diversi contesti (look-ahead):

- = se la L è stata ottenuta espandendo $S \rightarrow \bullet L = R$
- \$ se la L è stata ottenuta espandendo $R \rightarrow \bullet L$.

Quindi contesti diversi determinano look-ahead diversi. L'operazione di goto (shift) non agisce sul look-ahead come nei casi precedenti sposta solo il punto:



Con la parsing table LR(1) si va a calcolare in modo dettagliato il vero look-ahead che sarà un sottoinsieme del Follow. Formalmente un item LR(1) $[A \rightarrow \alpha \bullet \beta \quad a]$ è valido per un prefisso vitale γ se c'è una derivazione right-most $S \rightarrow \delta A W \rightarrow \delta \alpha \beta W$ dove:

1. $\gamma = \delta a$
2. "a" è il primo simbolo di W (first(W)) oppure $W = \epsilon$ e $a = \$$.

Ad esempio se consideriamo la grammatica $S \rightarrow BB$; $B \rightarrow aB/b$. nella derivazione right-most $S \rightarrow \bullet aaBab \rightarrow aaaBab$ si ha che l'item $[B \rightarrow a \bullet B \quad a]$ è valido per un prefisso vitale $\gamma = aaa$; infatti si è nella definizione ponendo: $\delta = aa$; $A = B$; $W = ab$; $\alpha = a$; $\beta = b$.

Nella derivazione $S \rightarrow \bullet BaB \rightarrow BaaB$ l'item $[B \rightarrow a \bullet B \quad \$]$ è valido per il prefisso vitale Baa.

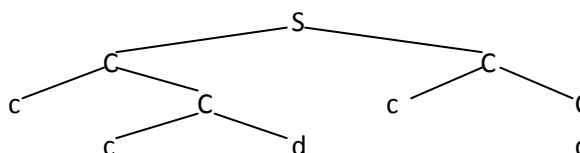
Ricapitolando:

- LR(0) \rightarrow non fa differenza di simboli; reduce su tutti i simboli;
- SLR(1) \rightarrow look-ahead sui simboli nel FOLLOW;
- LR(1) \rightarrow look-ahead sui simboli nel FOLLOW CONTESTUALE;
- LALR(1) \rightarrow look-ahead sui simboli nel FOLLOW in alcuni contesti;

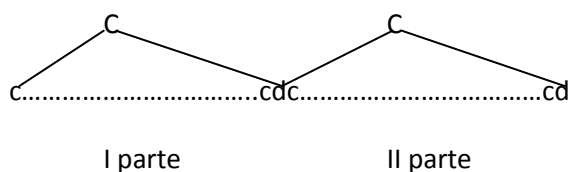
Nell'ultimo caso (LALR(1)) abbiamo una via di mezzo tra il follow generale SLR(1) e quello contestuale (LR(1)). Vedremo come unificare degli stati dell'LR(1) per ottenere gli stati per l'LALR(1).

Ora vediamo una grammatica:

1. $S' \rightarrow S$
2. $S \rightarrow CC$
3. $C \rightarrow cC$
4. $C \rightarrow d$



Tale grammatica genera frasi che possono essere divise in due parti:

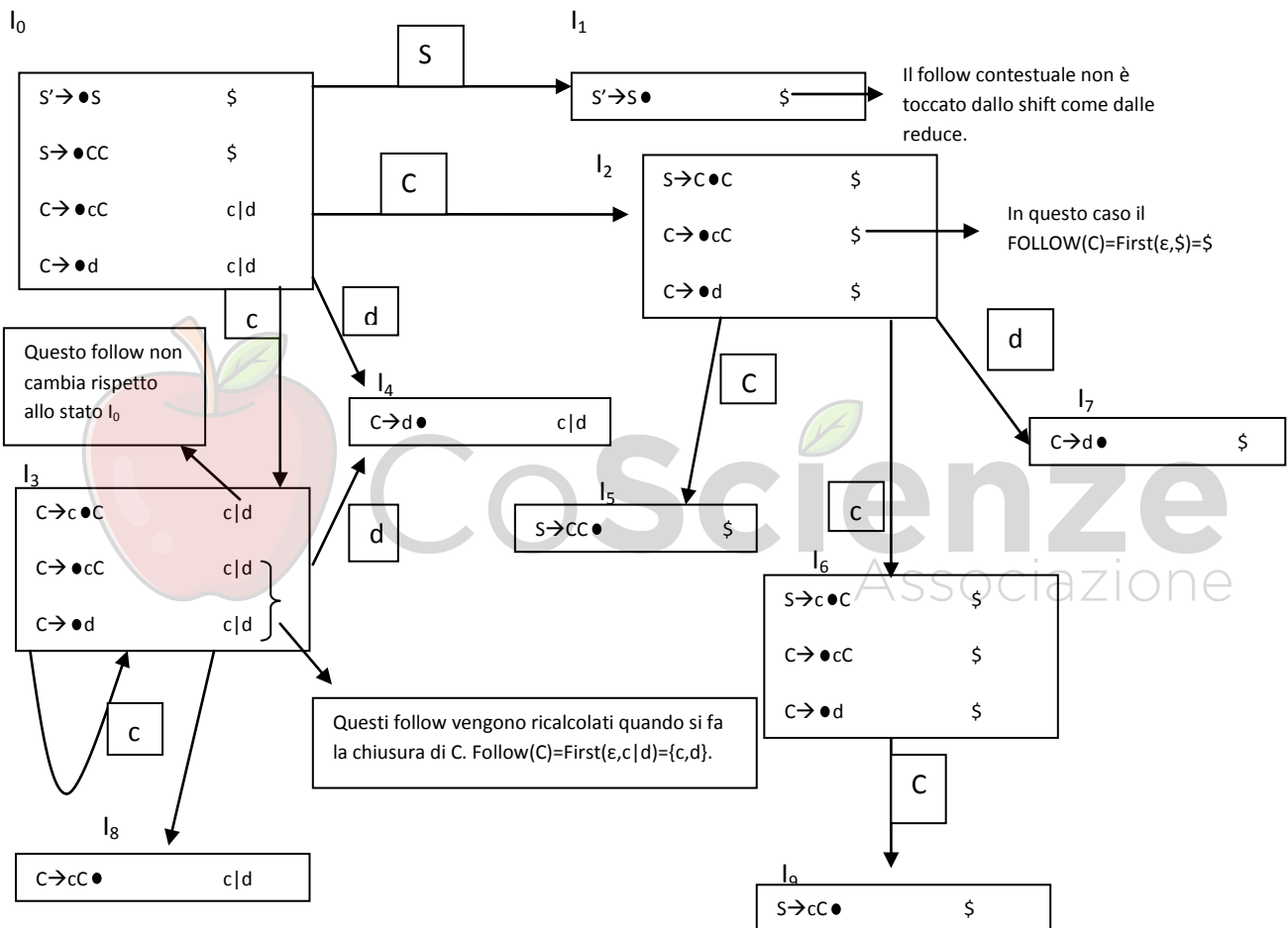


La prima C può essere seguita solo da "c" o "d" quindi avrà come look-ahead "c" e "d". La seconda C può essere seguita solo da \$; quindi avrà come look-ahead \$. $\text{Follow}(C)=\{c,d,\$ \}$

L'SLR(1) non fa distinzione fra le due C come look-ahead prende sempre tutto il $\text{Follow}(C)$. L'LR(1) distingue i contesti differenti delle due C, bisogna considerare i Follow Contestuali, ma per mantenere tali informazioni sono necessari troppi stati, per tale motivo si passerà all'LALR(1). L'LR(1) trova subito gli errori; non fa passi inutili come potrebbe capitare per l'LR(0) o l'SLR(1).

1. $S' \rightarrow S$
2. $S \rightarrow CC$
3. $C \rightarrow cC$
4. $C \rightarrow d$

Calcoliamo i Follow contestuali cioè gli item LR(1):



Con questo automa possiamo andare a costruire la tabella LR(1):

	c	d	\$	S	C
0	S3	S4		1	2
1			ACCEPT		
2	S6	S7			5
3	S3	S4			8
4	R4	R4			
5			R2		
6	S6	S7			9
7			R4		
8	R3	R3			
9			R3		

Visto il modo di procedere sull'esempio vediamo l'algoritmo in modo formale:

Algoritmo: Costruzione della tabella LR(1).

Input: una grammatica aumentata G' .

Output: la tabella con le funzioni Action e Goto per G' .

Metodo:

1. Costruire $C=\{I_0, I_1, \dots, I_n\}$ (insieme degli item) la collezione degli insiemi degli item LR(1) per G' .
2. Lo stato "i" del parser è costruito da I_i . Le azioni del parser per lo stato "i" sono determinate come segue:
 - ✓ Se $[A \rightarrow \alpha \bullet a \beta, b]$ è in I_i e $\text{goto}(I_i, a) = I_j$ allora settare $\text{Action}[i, a] = \text{"shift } j\text{"}$, dove "a" è un non terminale.
 - ✓ Se $[A \rightarrow \alpha \bullet, a]$ è in I_i dove $A \neq S'$ allora settare $\text{Action}[i, a] = \text{"Riduci con } A \rightarrow \alpha\text{"}$.
 - ✓ Se $[S' \rightarrow S \bullet, \$]$ è in I_i allora settare $\text{Action}[i, \$] = \text{"Accetta"}$Si è così riempita la parte Action della tabella (parte in cui possono verificarsi conflitti). Se vi sono dei conflitti causati da queste regole, allora la grammatica G' non è LR(1); la tabella non è deterministica. Riempiamo ora la sezione Goto:
3. Le transizioni goto per lo stato "i" sono determinate come segue: se $\text{goto}(I_i, A) = I_j$ allora $\text{Goto}[i, A] = j$.
4. Tutti gli elementi non definiti dalle regole 2 e 3 sono errori.
5. Lo stato iniziale del parser è quello costruito dell'insieme contenente l'item $[S' \rightarrow \bullet S, \$]$.

La tabella prodotta da questo algoritmo è detta Tabella canonica di analisi LR(1). Un parser che usa tale tabella è detto "parser canonico LR(1)". Se la tabella non ha elementi multidefiniti, allora la grammatica data è detta "grammatica LR(1)". Ogni grammatica SLR(1) è una grammatica LR(1); per una grammatica SLR(1) il parser LR canonico può avere più stati di un parser SLR.

PARSING TABLE LALR(1)

Vi sono due modi per costruire la parsing table LALR(1):

1. I TECNICA: non è efficiente. L'idea è di costruire prima gli stati LR(1) canonici e poi fondere gli stati che hanno il cuore (core) in comune (in pratica quelli che differiscono solo per i follow contestuali), facendo l'unione dei look-ahead; dall'automa ottenuto si costruisce la parsing table LALR(1).

Algoritmo: una facile costruzione della tabella LALR, ma costosa in termini di spazio.

Input: una grammatica aumentata G' .

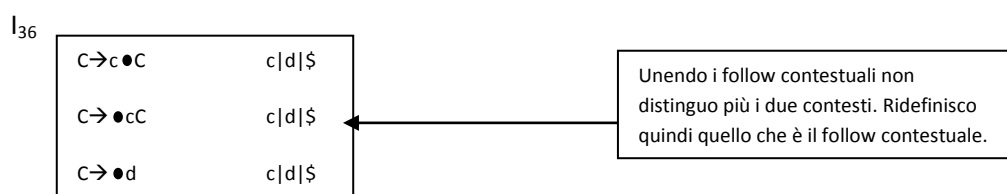
Output: la tabella LALR con le funzioni Action e Goto per G' .

Metodo:

- i. Costruire $C=\{I_0, I_1, \dots, I_n\}$ collezione degli insiemi di item LR(1).
- ii. Per ogni core (insieme degli item in uno stato) presente nell'insieme degli item LR(1), trovare tutti gli insiemi aventi quel core e rimpiazzarli con la loro unione (unendo anche i look-ahead).
- iii. Sia $C'=\{J_0, J_1, \dots, J_m\}$ il risultato ottenuto. Le azioni per lo stato "i" sono costruite da J_i nello stesso modo visto per l'algoritmo LR(1). Se vi sono dei conflitti allora la grammatica non è LALR(1); la tabella non è deterministica.
- iv. La sezione Goto è costruita come segue: se "J" è l'unione di uno o più insiemi di item LR(1), cioè, $J=I_1 \cup I_2 \cup \dots \cup I_k$ allora i core di $\text{goto}(I_1, X) \dots \text{goto}(I_k, X)$ sono gli stessi, poiché $I_1 \dots I_k$ hanno lo stesso core. Sia "k" l'unione di tutti gli insiemi di item aventi lo stesso core $\text{goto}(I_j, X)$ allora si pone $\text{goto}(J, X) = "k"$.
- v. Ugual a quello del precedente algoritmo.
- vi. Ugual a quello del precedente algoritmo.

La tabella prodotta dall'algoritmo è detta "Tabella di analisi LALR(1)" per G e se non ci sono conflitti G è detta "grammatica LALR(1)". Vediamo perché tale tecnica non è efficiente: la tabella LR(1) è la più potente, appena c'è un errore lo rileva senza fare operazioni aggiuntive, ma porta ad avere moltissimi stati, quindi è molto costosa in termini di memoria. Per tale motivo si passa all'LALR(1) la quale fonderà alcuni stati dell'LR(1) risparmiando così memoria. Tale guadagno lo si paga dovendo fare delle riduzioni in più prima di rilevare gli errori. Con l'algoritmo visto, prima di arrivare agli stati dell'LALR(1) bisogna comunque costruire gli stati dell'LR(1) e quindi avere a disposizione tutta la memoria necessaria; cioè con tale algoritmo non si evita lo svantaggio dell'LR(1). Vediamo un esempio: applicando l'algoritmo all'esempio precedente si trovano tre coppie di insiemi di item che possono essere unificati:

I_3 e I_6 sono uguali a meno dei look-ahead futuri poiché I_3 è sviluppato dalla prima C e I_6 è sviluppato dalla seconda C; I_3 e I_6 generano stati gemelli cioè creano due strade parallele che si differenziano solo per i look-ahead. Tali stati saranno rimpiazzati dalla loro unione:



I_4 e I_7 differiscono solo per i look-ahead e quindi saranno rimpiazzati dalla loro unione:



I_8 e I_9 differiscono solo per i look-ahead e quindi saranno rimpiazzati dalla loro unione:



La tabella LALR che si ottiene è:

	c	d	\$	S	C
0	S36	S47		1	2
1			ACCEPT		
2	S36	S47			5
36	S36	S47			89
47	R4	R4	R4		
5			R2		
89	R3	R3	R3		

ACTION GOTO

Su input errati il parser LALR potrebbe effettuare qualche riduzione in più, mentre il parser LR segnala subito l'errore (ciò rende il parser LALR(1) più inefficiente del parser LR(1) poiché deve fare più riduzioni per individuare gli errori). Infatti consideriamo l'input errato su ccd\$:

PARSER LR: produce sullo stack 0c3c3d4 e nello stato 4 segnala un errore (corrispondenza 4,\$).

PARSER LALR: produce sullo stack 0c36c36d47; nello stato 47 non rileva subito l'errore come LR(1), ma fa una riduzione con $C \rightarrow d$ (poiché lo stato 47 contiene la possibilità che il simbolo letto è prodotto dalla seconda C). Abbiamo 0c36c36c89; nello stato 89 con input \$ si fa ancora una riduzione con $C \rightarrow cC$ ottenendo 0c36c89; si è nella stessa condizione di prima e si riduce di nuovo con $C \rightarrow cC$: 0c2 nello stato 2 leggendo \$ si rileva l'errore.

Si nota che LR segnala subito l'errore senza fare riduzioni inutili, mentre LALR prima di segnalare l'errore fa tre riduzioni in più. Questa differenza è dovuta alla definizione del follow contestuale in LR e in LALR.

2. Il TECNICA: è efficiente (ma non la vedremo). Tale tecnica per arrivare all'LALR(1) invece di partire dall'LR(1) parte dall'SLR(1) evitando così elevati costi di memoria. Si calcolano gli SLR(1) item (cioè gli LR(0) item poiché vale che $SLR(1)_{item} = LR(0)_{item}$). Vi sarà poi una procedura che mette i look-ahead appropriati.

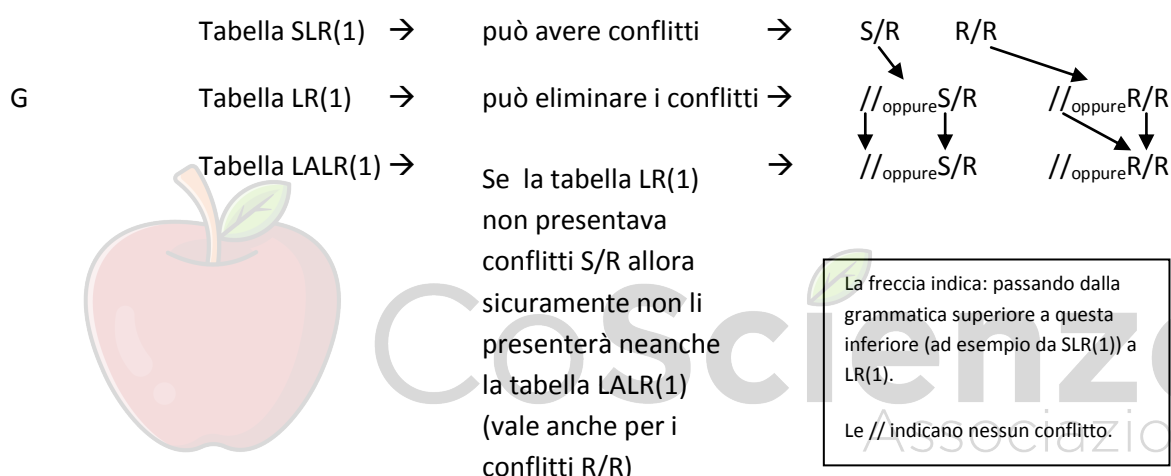
Look-ahead LALR(1) \leq look-ahead SLR(1)

inclusione

Con la prima tecnica può accadere, come avviene nell'esempio, che si ottiene una tabella LALR(1) che coincide con la tabella SLR(1).

CARATTERIZZAZIONE DEI CONFLITTI

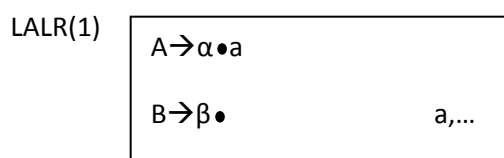
Data una grammatica G si possono applicare gli algoritmi visti per ottenere i diversi tipi di tabelle. Vediamo in che relazione sono i conflitti nelle varie tabelle:



Vediamo che per una grammatica G si ha che: se la tabella LR(1) non presenta conflitti S/R \rightarrow anche la tabella LALR(1) non presenta conflitti S/R.

Ciò indica che la fusione di stati non può introdurre conflitti S/R se non erano già presenti. Per dimostrare ciò si mostra che: se la tabella LALR(1) presenta conflitti S/R \rightarrow anche la tabella LR(1) presenta conflitti S/R.

Consideriamo uno stato LALR(1) con conflitto S/R:



Conflitto S/R sulla "a"

Questo conflitto può essere stato ottenuto dagli stati LR(1) in due modi diversi:

1. È uno stato presente anche tra gli LR(1), in questo caso si avrà lo stesso conflitto nell'LR(1).
2. È ottenuto come fusione di più stati LR(1), gli stati fusi dovranno avere tutti lo stesso "core" e possono differire solo per i look-ahead, lo stato LALR(1) è ottenuto prendendo il "core" comune a tutti gli stati LR(1) e l'unione di tutti i look-ahead. Quindi per avere ottenuto tale stato LALR(1) necessariamente deve esserci almeno uno stato LR(1) del tipo:

LR(1)	<div> $A \rightarrow \alpha \bullet a$ $B \rightarrow \beta \bullet$ </div> <div>a,...</div>	Conflitto S/R sulla "a"
-------	---	-------------------------

Che presenta un conflitto S/R. Gli altri stati LR(1) da fondere saranno del tipo:

LR(1)	<div> $A \rightarrow \alpha \bullet a$ $B \rightarrow \beta \bullet$ </div> <div>,...</div>	<div> $A \rightarrow \alpha \bullet a$ $B \rightarrow \beta \bullet$ </div> <div>a,...</div>
-------	--	---

Cioè almeno uno stato LR(1) da fondere presenta un conflitto S/R allora sicuramente anche lo stato LALR(1), ottenuto dalla fusione, presenta conflitti S/R. La fusione quindi non introduce conflitti S/R, può solo riportare conflitti già presenti dell'LR(1).

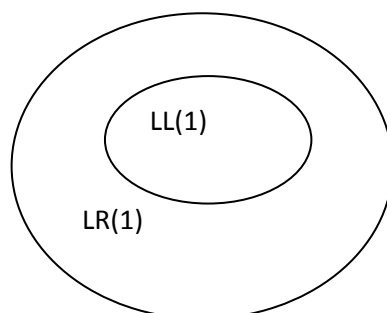
Lo stesso discorso però non può essere fatto per i conflitti R/R. Cosa può accadere??? È possibile che la tabella LR(1) non presenta conflitti R/R, mentre la tabella LALR(1) presenta conflitti R/R. Cioè la fusione può introdurre conflitti R/R non presenti nell'LR(1), per tale motivo la parsing-table LALR(1) è meno potente di quella LR(1).

LR(1)	<div> $A \rightarrow \alpha a \bullet$ $B \rightarrow \beta \bullet$ </div> <div>,c ,d</div>	<div> $A \rightarrow \alpha a \bullet$ $B \rightarrow \beta \bullet$ </div> <div>,d ,c</div>
-------	---	---

In questi due stati dell'automa non ci sono conflitti R/R. Se li uniamo però...

LALR(1)	<div> $A \rightarrow \alpha a \bullet$ $B \rightarrow \beta \bullet$ </div> <div>,c/d ,d/c</div>	CosScienze Associazione
---------	---	----------------------------

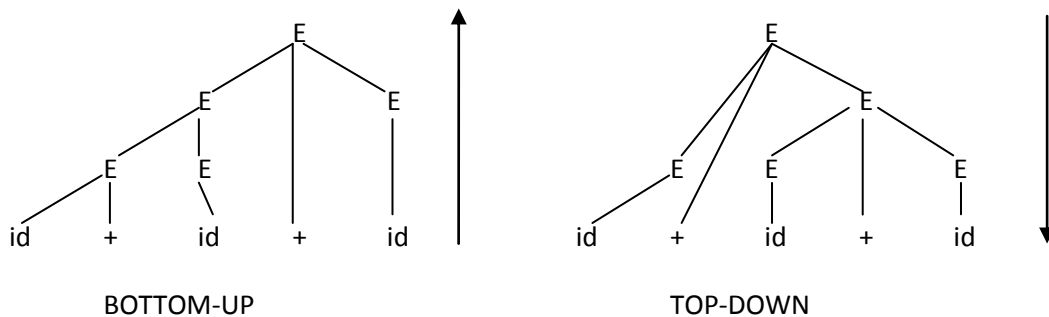
Questo stato presenta un conflitto R/R. Concludendo abbiamo che un linguaggio è LR(k) se esiste una grammatica LR(k) che lo genera. Poiché L è context free deterministic language può essere generato da una qualsiasi grammatica: LR(0), SLR(1), LALR(1), LR(1), ..., LR(k). La differenza è che la grammatica LR(0) sarà la più complessa con più produzioni per evitare conflitti, la grammatica SLR(1) un po' meno complessa etc. Fino ad arrivare alla LR(k) che sarà la più semplice. Le grammatiche di ogni linguaggio esistono, la complessità sta nel trovarle. Ci sono moltissime grammatiche che generano uno stesso linguaggio (basta aggiungere produzioni inutili). Le grammatiche però hanno la necessità di essere semplici.



Quindi possiamo scrivere una grammatica LR(1) che è più semplice di quella LL(1). Vediamo un esempio.

Sia G:
 $E \rightarrow E + E$
 $E \rightarrow E * E$
 $E \rightarrow (E)$
 $E \rightarrow id$

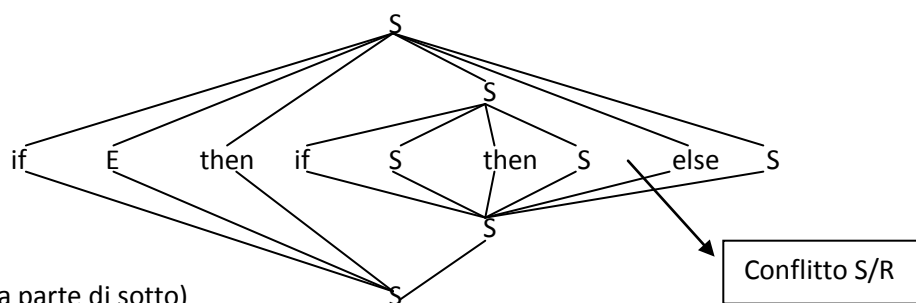
Tale grammatica è ambigua e le grammatiche ambigue non possono generare parser deterministici e quindi non è LR(1). Quindi, teoricamente, tale grammatica non si può usare per i parser visti. Possiamo considerare le regole dell'aritmetica per eliminare i conflitti ottenendo una parsing table deterministica. Se abbiamo solo G e la frase $id+id+id$ si possono ottenere due parsing table:



La possibilità di avere due parsing tree corrisponde all'avere un conflitto S/R nella parsing table quando si legge il secondo "+"; se si introduce l'informazione che la somma è associativa a sinistra si ha che solo il primo parsing-tree è corretto e si elimina il conflitto Shift; si ottiene così una grammatica non deterministica. La parsing table ottenuta non è LR(1) poiché non è ottenuta con l'algoritmo dell'LR(1) però funziona dato che è stata manipolata a mano con la regola che abbiamo deciso di usare. Altra possibile informazione che si può usare per rompere i conflitti è la "priorità tra le operazioni". Così facendo si eliminano i conflitti "manualmente"; la grammatica non è LR quindi porta ad un parser non deterministico, ma noi aggiustiamo la parsing table con le regole che abbiamo deciso, rendendo la tavola deterministica ottenendo un parser LR manipolato deterministico.

L'LR è molto più complesso poiché più potente dell'LL(1) può avere moltissimi stati; quindi può essere complesso eliminare a mano i conflitti che si presentano. Vi è un tool che "avuta la grammatica e le regole da rispettare restituisce la parsing table deterministica", cioè elimina automaticamente anche le ambiguità facendo riferimento alle regole introdotte; tale tool è YACC. Vediamo un esempio: consideriamo la grammatica ambigua che porta alla frase seguente. Possiamo avere due interpretazioni:

I parser tree



Il parser tree (la parte di sotto)

Il conflitto nasce quando si incontra l'else: a seconda della parsing table in corrispondenza dell'else si avrà un conflitto S/R. Se adottiamo la regola che l'else si attacca al then più vicino solo il secondo parser-tree è corretto e si elimina così l'ambiguità ottenendo una parsing table deterministica.

YACC

Data la grammatica YACC ci rileva se nella parsing-table ci sono conflitti, se li trova ci dice in che contesto si trovano. Noi introdurremo le regole per risolvere tali conflitti. YACC costruisce una parsing table LALR(1) utilizzando l'algoritmo efficiente. L'LALR(1) è la parsing-table più usata; ha meno stati dell'LR(1) ed eventuali conflitti saranno risolti con regole specifiche.

Un file YACC è diviso in più sezioni:

<nome>.y

%{

Dichiarazioni in C

%}

Dichiarazioni

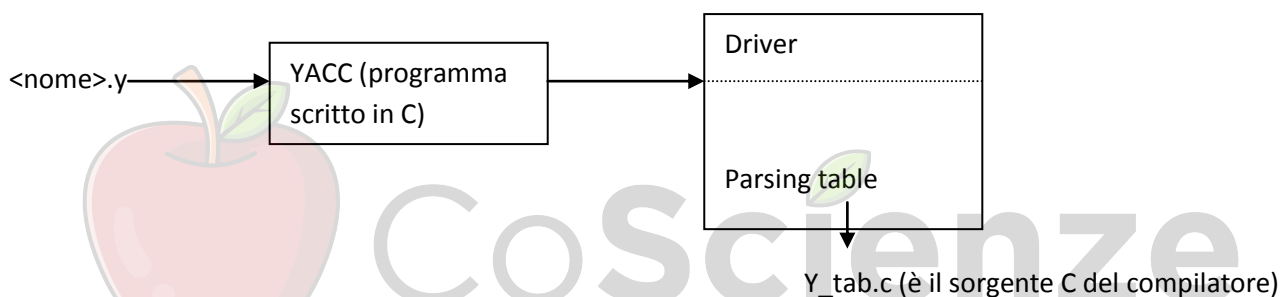
%%

GRAMMATICA

%%

Routines di servizio

Dovremo essere noi a scrivere un file YACC.



Il programma YACC è formato da due componenti:

YACC

DRIVER. Algoritmo di LR	Costruttore di parsing-table
-------------------------	------------------------------

Il Driver consulta la parsing table per decidere quale azione fare. YACC legge la grammatica da .y e consultando il driver costruisce la parsing-table.

Vediamo un esempio di un file YACC; quello per la grammatica delle espressioni aritmetiche con costanti:

%{ *#include <stdio.h>*

...

%}

DICHIARAZIONI

% token *NUM*

...

% left *'+' '-'*

% left *'*' '/'*

% right *uminus*

Sono dichiarazioni in C che saranno copiate senza manipolazioni nel file y_tab.c.

Lista dei token della grammatica restituiti dall'analisi lessicale.

La somma e la differenza sono associative a sinistra

Il meno unario è associativo a destra

L'ordine con cui sono poste tali dichiarazioni definisce la priorità tra le operazioni: '+' e '-' hanno stessa priorità che è minore della priorità di '*' e '/' e così via. Tali regole possono rompere eventuali conflitti.

%% *Routines di servizio*

Nelle routines di servizio va l'analizzatore lessicale `yylex()` (il codice che restituisce il token e l'attributo in `yylval`). Il programma C generato quando cerca il prossimo simbolo fa una chiamata a `yylex()`. `Yylex()` lo si può scrivere a mano oppure generare con LEX. `Yyerror` (stringa da stampare) che deve essere chiamata automaticamente quando vi è un errore di sintassi. Abbiamo visto come YACC risolve un conflitto S/R. Se vi è un conflitto R/R si sceglie di fare la R con la produzione che contiene l'operatore con maggiore priorità (considera l'operatore più a destra). Se non vi sono regole che risolvono conflitti YACC ci segnala i conflitti e li risolve per default:

S/R → fa lo shift

R/R → risolve il conflitto facendo la reduce con la produzione che viene prima.

Quindi nel caso di :

if C1 then if C2 then S1 else S2

Conflitto S/R su S1

non c'è bisogno di introdurre regole, poiché YACC per default fa lo Shift correttamente. L'LR-parsing non si può fare a mano, è troppo complesso.



LEZIONE 13 - ESERCITAZIONE

22/04/2009

Data la grammatica:

$X \rightarrow Ya$

$X \rightarrow bYc$

$X \rightarrow da$

$Y \rightarrow d$

Indicare se la grammatica è SLR(1)?? È LALR(1)???

$X' \rightarrow X$

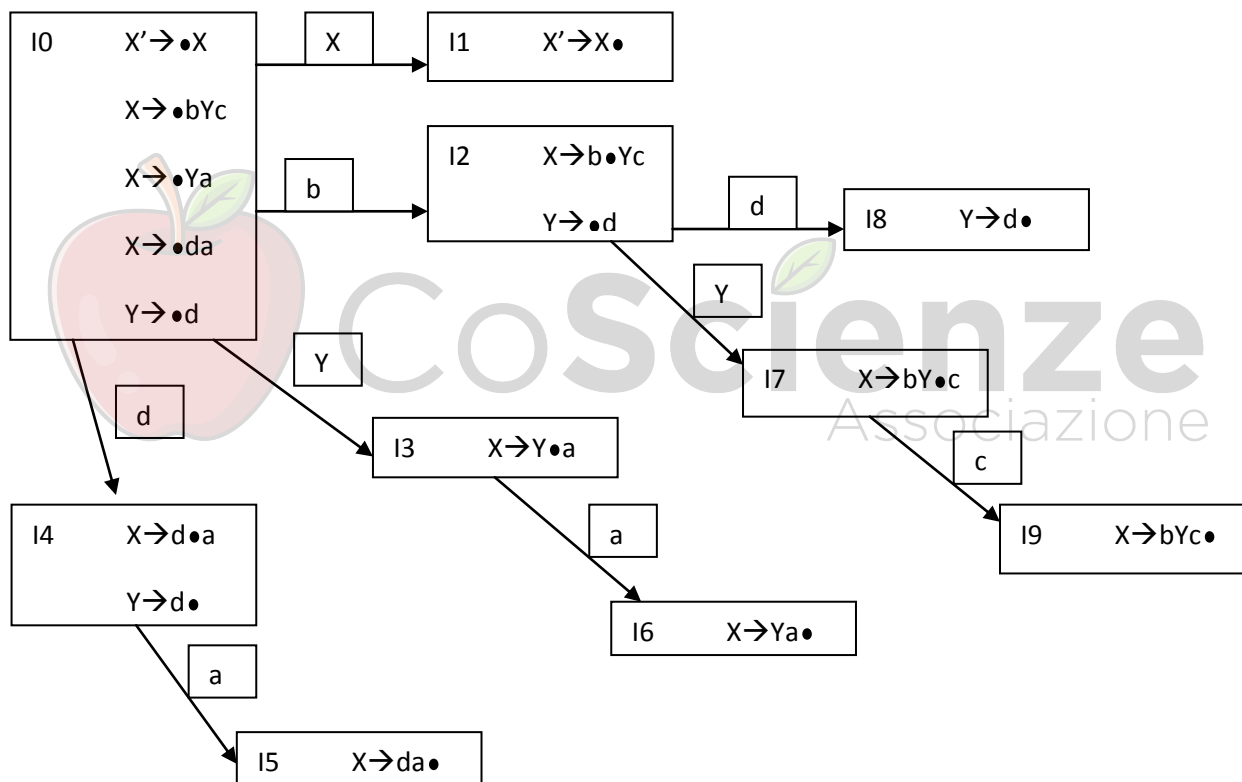
$X \rightarrow Ya$

$X \rightarrow bYc$

$X \rightarrow da$

$Y \rightarrow d$

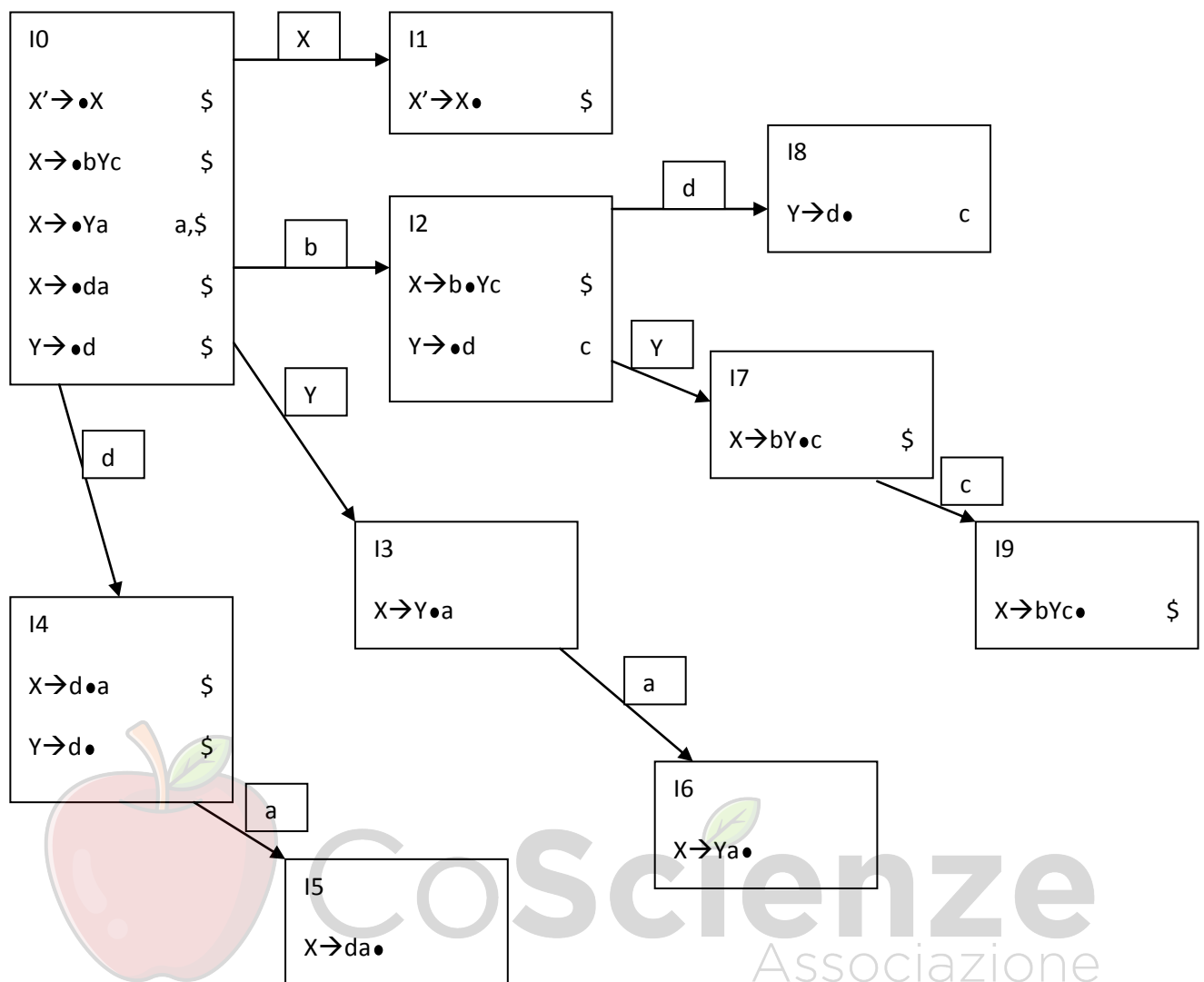
Costruiamo il diagramma degli stati e vediamo se ci sono conflitti:



Di sicuro la grammatica non è LR(0). C'è un conflitto nello stato 4 S/R. Trasformiamo la grammatica e vediamo i follow. Calcoliamo il Follow di Y.

$\text{Follow}(Y) = \{a, c\}$

Già qui possiamo notare che la grammatica non è SLR(1). Il conflitto in I4 rimane di tipo Shift/Reduce. A causa del Follow(Y) dovrei mettere reduce sia sotto "a" che sotto "c". Ma sotto "a" in I4 ci sarà anche la shift. Ora per vedere se è LALR(1) costruiamo la LR(1) e poi facciamo la fusione.



Il follow contestuale cambia nel momento in cui si effettua la chiusura su un non terminale. Dato che in I4 permane il conflitto S/R. Dato che la tabella non è LR(1), a causa della presenza del conflitto, allora la grammatica non è neanche LALR(1) come da teorema.

Vediamo un altro esempio, data la grammatica:

$S \rightarrow rAStI$

$A \rightarrow aA$

$A \rightarrow \epsilon$

$I \rightarrow iI$

$I \rightarrow \epsilon$

È SLR(1) o LALR(1)?

Aumentiamo la grammatica:

$S' \rightarrow S$

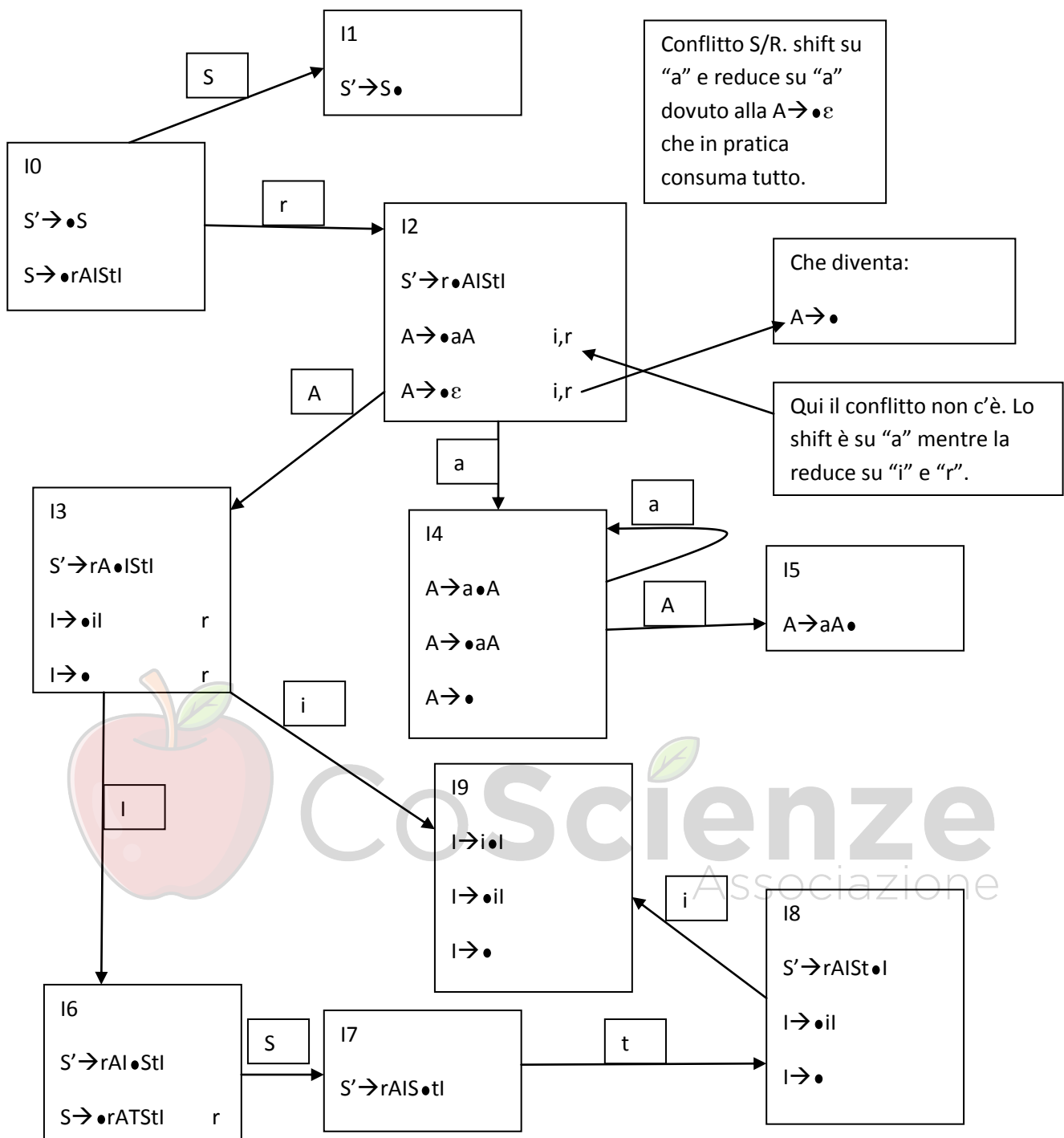
$S \rightarrow rAStI$

$A \rightarrow aA$

$A \rightarrow \epsilon$

$I \rightarrow iI$

$I \rightarrow \epsilon$



Dato che in SLR(1) non ci sono conflitti S/R allora non ci saranno neanche in LALR(1). Quindi la grammatica è sia SLR(1) che LALR(1).

ESERCIZIO:

Data la grammatica:

$S \rightarrow Aa$

$S \rightarrow bAc$

$S \rightarrow Bc$

$S \rightarrow bBa$

$A \rightarrow d$

$B \rightarrow d$

È LR(1)? È LALR(1)?

Per prima cosa aumentiamo la grammatica:

$S' \rightarrow S$

$S \rightarrow Aa$

$S \rightarrow bAc$

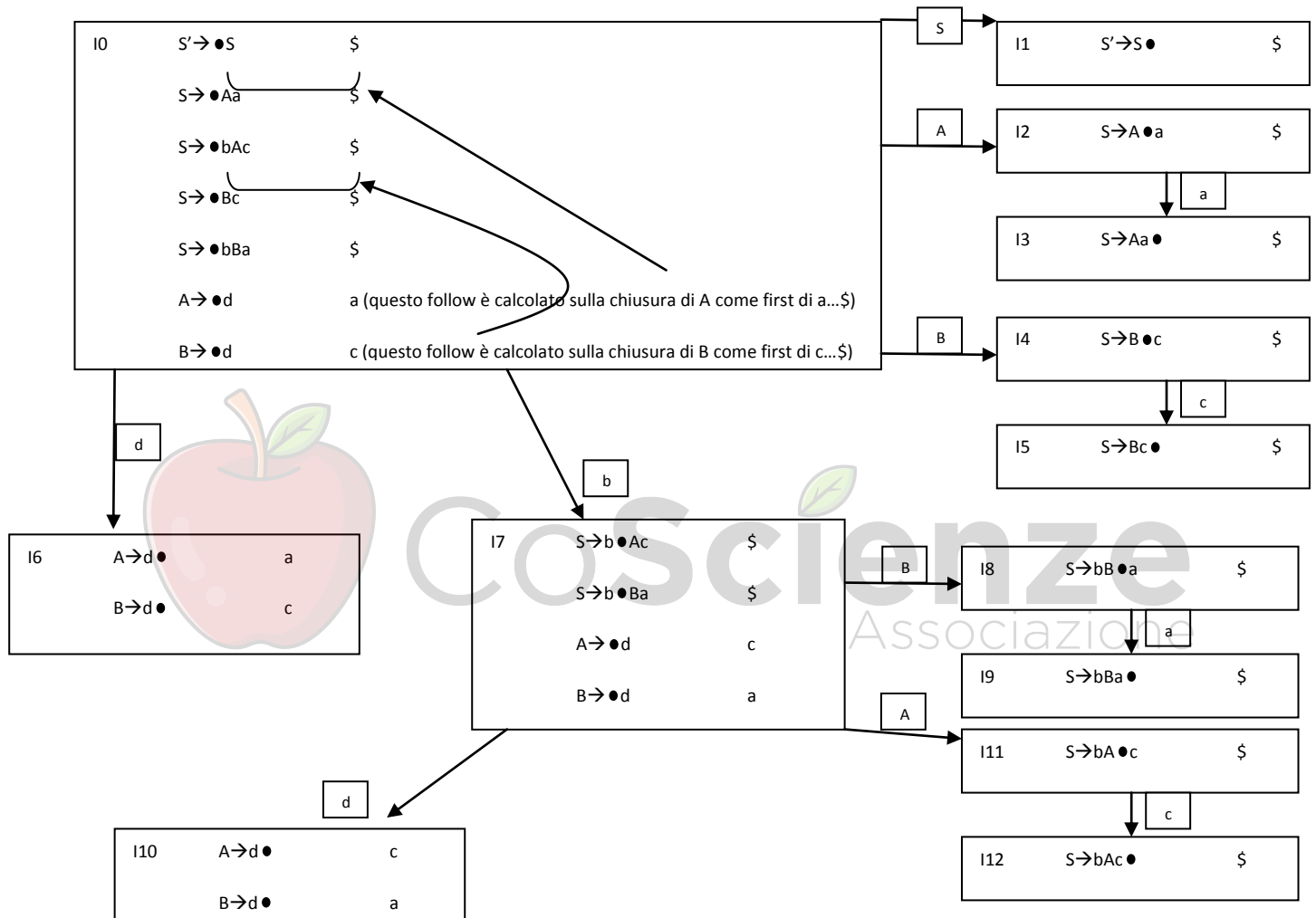
$S \rightarrow Bc$

$S \rightarrow bBa$

$A \rightarrow d$

$B \rightarrow d$

E andiamo a costruire l'automa con i follow contestuali.



Possiamo notare come gli unici stati che potrebbero generare conflitti sono l'I6 e l'I10 dove però i follow contestuali permettono di evitare il rischio di conflitto R/R. Non sono presenti dei conflitti S/R e quindi la grammatica è LR(1).

Passando all'LALR(1) invece si deve effettuare la fusione degli stati con core uguale. Ciò ci porta a fondere gli stati I10 e I6 in unico stato ottenendo:

I6-10	$A \rightarrow d \bullet$	a, c
	$B \rightarrow d \bullet$	a, c

che determina la presenza di un conflitto R/R. La grammatica quindi non è LALR(1).

JAVACUP

JavaCup è un generatore di parser LALR bottom up. Produce un parser scritto in java ed esso stesso è scritto in java. La direttiva **%cup** in JFlex abilita la compatibilità con JavaCup.

ESEMPIO

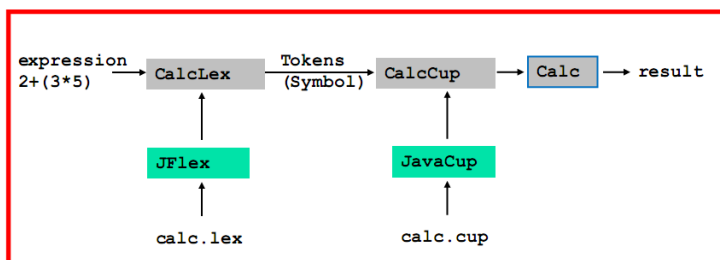
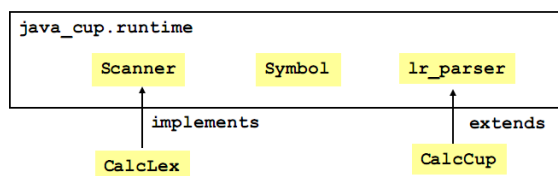
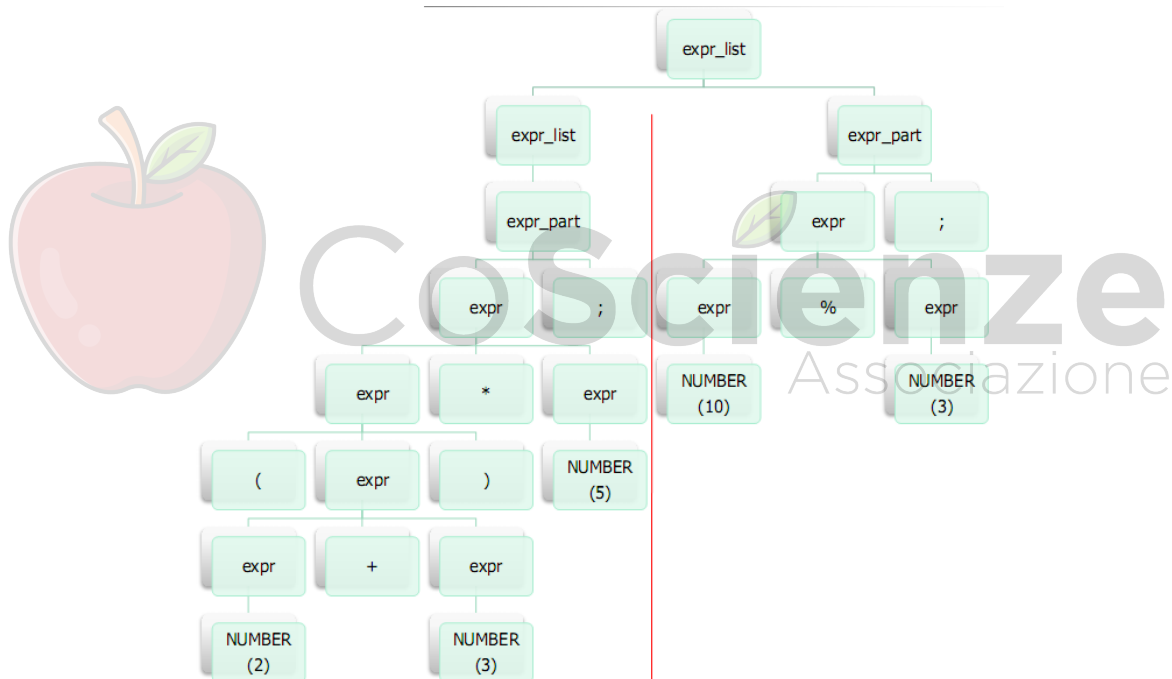
Questo esempio permette di valutare delle semplici espressioni aritmetiche tra interi.

```
expr_list : expr_list expr_part
          | expr_part

expr_part : expr ';'

expr : expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | expr '%' expr
      | '(' expr ')'
      | '-' expr
      | number
```

Albero di parsing per $(2.3)*5; 10 \% 3;$



SINTASSI

1. Specificare i package e le import
2. Codice scritto dall'utente che gli permette di includere una parte del generatore di parser
 - a. parser code { ... };
 - b. scan with { ... };
 - c. init with { };
3. lista dei simboli terminali e non terminali
4. dichiarazioni di precedenza (destra, sinistra e non associazione). Permettono di risolvere i conflitti S/R ma solo in casi di precedenza uguale.
5. la grammatica inizia con un non terminale. Ogni produzione alla sinistra deve avere un simbolo non terminale

Es. di grammatica

```
terminal PLUS, MINUS;
non terminal Integer expr;

expr ::= expr:e1 PLUS expr:e2
      { RESULT = new Integer(e1.intValue() + e2.intValue()); }
      |
      expr:e1 MINUS expr:e2
      { RESULT = new Integer(e1.intValue() - e2.intValue()); }
      ...
```

I metodi da implementare sono:

- ✓ `public void report_error (String message, Object info)`: questo metodo deve essere chiamato quando si deve mostrare un messaggio di errore. Tipicamente questo metodo provvede a generare un meccanismo sofisticato di reporting di errori.
- ✓ `public void report_fatal_error (String message, Object info)`: questo metodo deve essere chiamato quando si genera un errore irreversibile. Effettua la chiamata `report_error()` e successivamente interrompe la fase di parsing chiamando il metodo `done_parsing()` ed infine mostra l'eccezione.
- ✓ `public void syntax_error (Symbol cur_token)`: questo metodo è chiamato dal parser non appena si verifica un syntax error.
- ✓ `public void unrecovered_syntax_error (Symbol cur_token)`: questo metodo è chiamato da parser se non riesce ad effettuare il recovery dal syntax error.

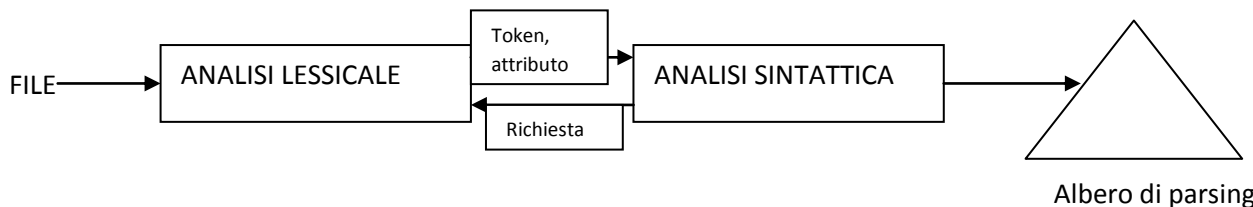
LEZIONE 14 – ANALISI SEMANTICA (1 PARTE)

GRAMMATICHE C.F. AD ATTRIBUTI

27/04/2009

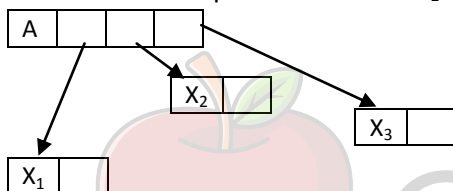
Abbiamo visto l'analisi lessicale e l'analisi sintattica, passi necessari per tradurre un linguaggio ad alto livello in un linguaggio macchina.

- L'ANALISI LESSICALE è una fase di riconoscimento, divide i lessemi in input, unità logiche.
- L'ANALISI SINTATTICA è la fase nella quale al parser arrivano solo i token e costruisce l'albero di parsing.



L'algoritmo di parsing dà un albero astratto (logico), ci dice come è fatto logicamente. Ora vogliamo vedere un "traduttore" che ci permetta di costruire una struttura dati ad albero, una rappresentazione fisica (reale). Vogliamo vedere come fare a passare da:

"ho utilizzato la produzione $A \rightarrow X_1X_2X_3$ " alla struttura a puntatori:



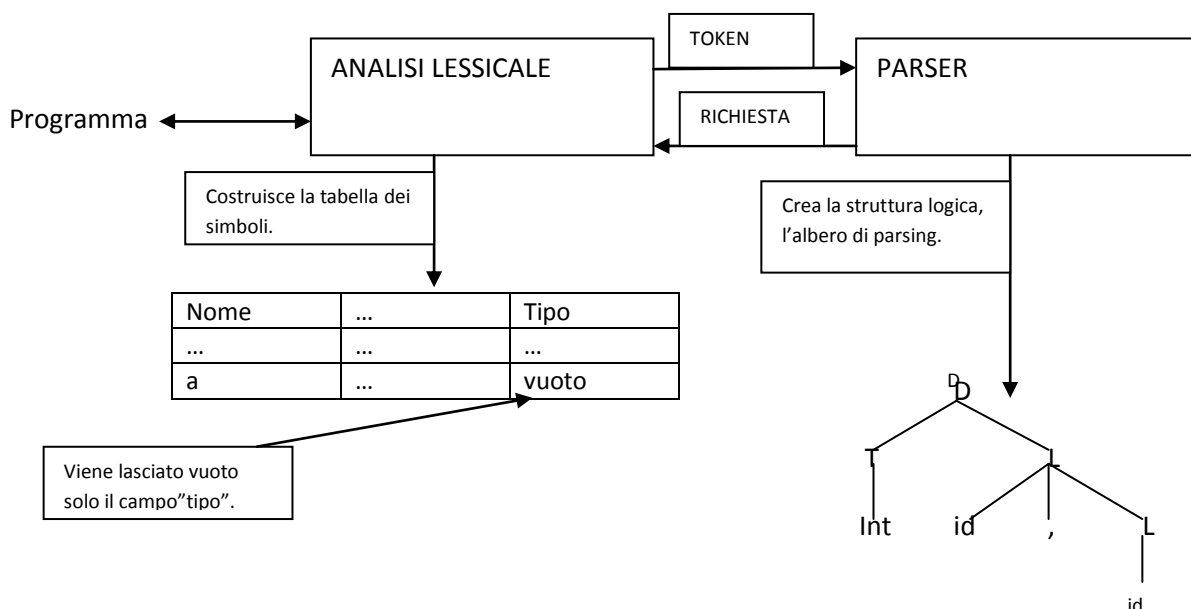
Per costruire tale albero useremo le regole semantiche inserite tra le produzioni in YACC. Nel nodo foglia ci saranno i terminali e le informazioni sui terminali saranno nella symbol table. Costruito l'albero fisico si passa all'ANALISI SEMANTICA che avrà il compito di fare il controllo dei tipi. Vediamo un esempio: se nel programma abbiamo:

int a,b;

a:=5.4;

vediamo come rilevare tale errore.

L'analisi semantica deve rilevare questo errore.



Se la grammatica è:

$P \rightarrow LDS$ $L_D \rightarrow D L_D$

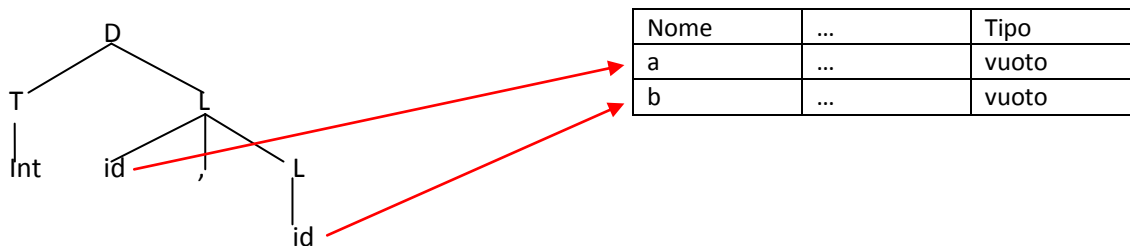
$D \rightarrow TL$ $L_D \rightarrow D$

$T \rightarrow \text{int} | \text{real}$

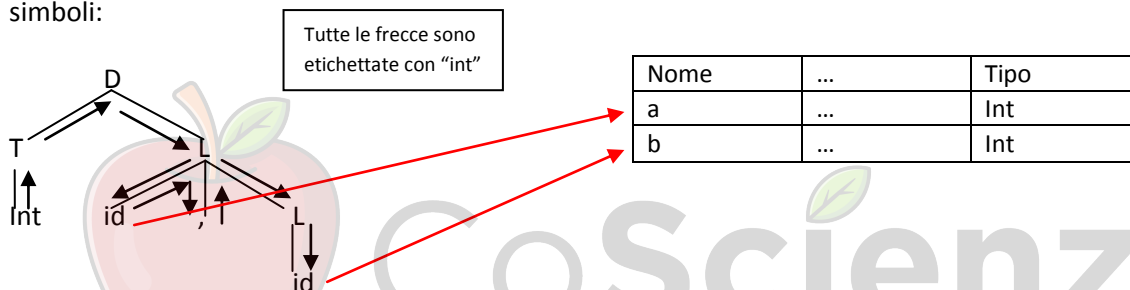
$L \rightarrow \text{id}, L$

$L \rightarrow \text{id}$

Dall'analisi lessicale conosciamo i puntatori alle variabili nella tabella dei simboli; possiamo aggiungere a queste, altre informazioni:



Ciò è ottenuto dall'analisi lessicale e dall'analisi sintattica (non abbiamo ancora visto come costruire l'albero fisico). Per aggiungere l'informazione devo visitare l'albero e aggiungere int nella tabella dei simboli:



"int" sarà una informazione che facciamo migrare in ogni nodo e quando arriva ad un nodo id, usiamo una funzione `add_type` per inserire int nel campo type puntato dall'id incontrato (passo l'informazione di type a tutti gli identificatori nel sottoalbero).

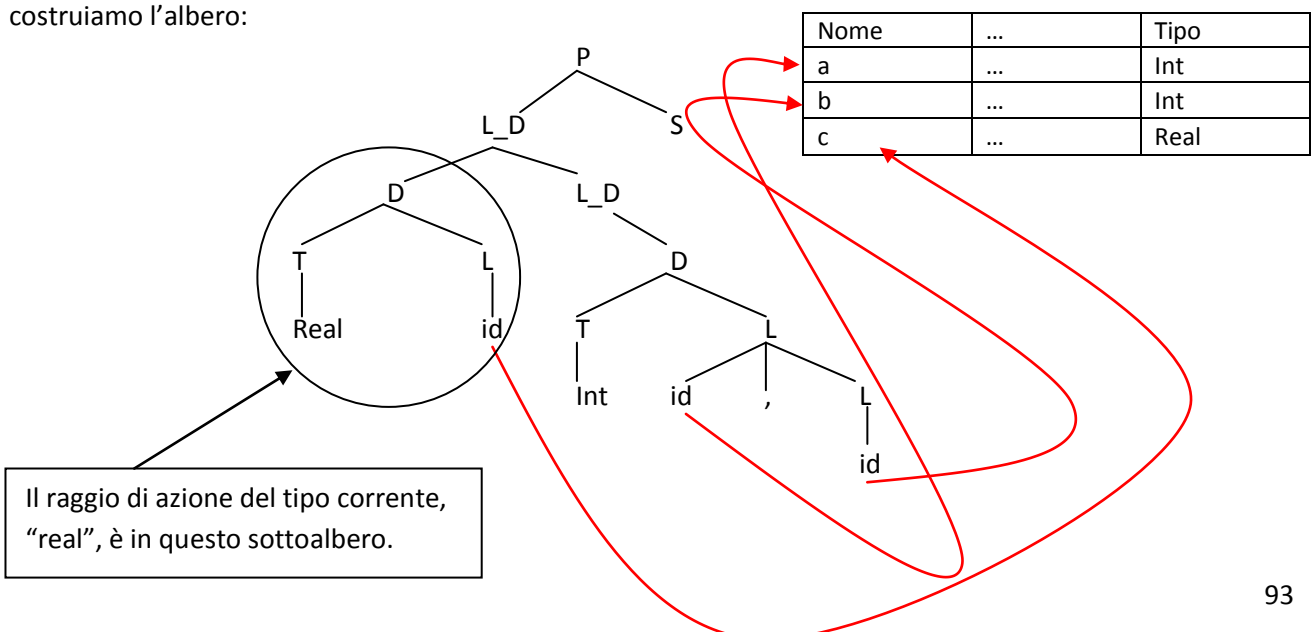
Vediamo un altro esempio:

real c;

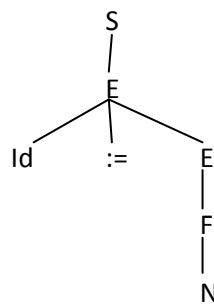
int a,b;

a:=5.4;

costruiamo l'albero:



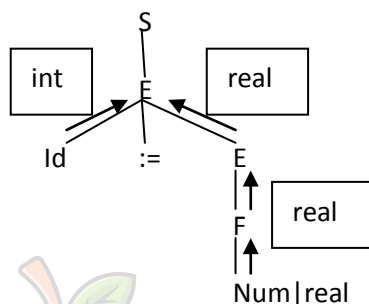
Quello che accade generalmente è che nella fase di dichiarazione delle variabili riempiamo la tabella dei simboli e nella fase d'uso delle variabili controlliamo i tipi attraverso la tabella. Con tale tecnica si riempiono i campi tipo della tabella dei simboli; useremo tali informazioni quando si andrà ad analizzare S, per il quale si avrà il seguente albero:



Nome	...	Tipo
a	...	Int

Possiamo conservare direttamente tale informazione oppure accedere alla tabella dei simboli...

Si visiterà l'albero facendo salire le informazioni sui tipi:



Vedremo come fare tali passaggi:

- Come costruire l'albero fisico?
- Come visitare l'albero?

A tale scopo ci servirà uno strumento teorico (implementato da YACC). Con YACC si realizza un compilatore ad un passo.

Ricordiamo infine che:

ANALISI LESSICALE	\leftrightarrow	AUTOMI FINITI
ANALISI SINTATTICA	\leftrightarrow	GRAMMATICHE CONTEXT FREE

E vediamo quale strumento teorico ci sarà utile per l'analisi semantica.

GRAMMATICHE CONTEXT FREE AD ATTRIBUTI

Tali grammatiche si dividono in:

- ✓ Definizioni guidate dalla sintassi (non è definito l'ordine con cui eseguire le regole SDD)
- ✓ Schemi di traduzioni (è specificato l'ordine con cui eseguire le regole).

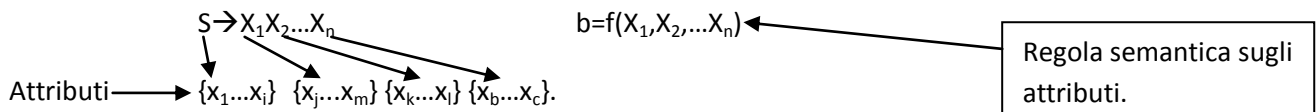
Si differenziano solo per dove sono poste le regole associate alle produzioni:

- ✓ Nelle definizioni guidate dalla sintassi le regole sono poste sempre alla fine delle produzioni; significa che non è specificato quando la regola deve essere eseguita.
- ✓ Negli schemi di traduzioni le regole semantiche sono poste all'interno delle produzioni, nel punto esatto in cui devono essere eseguite e saranno eseguite durante il riconoscimento.

Il nome traduzione di sintassi sta ad indicare le due definizioni precedenti: definizione guidata dalla sintassi e schema di traduzione.

Definizioni guidate dalla sintassi: sono una generalizzazione delle grammatiche C.F., in cui ad ogni simbolo della grammatica (terminale o non terminale) è associato un insieme di attributi, e ad ogni produzione si associa una regola semantica che agisce sugli attributi dei simboli coinvolti nella produzione.

Vediamo un esempio:



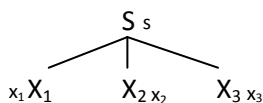
Si nota come la regola semantica è posta alla fine della produzione.

Gli ATTRIBUTI si dividono in:

- ✓ Sintetizzati
- ✓ Ereditati

Sia $S \rightarrow X_1 X_2 X_3$
 $\{s\} \{x_1\} \{x_2\} \{x_3\} \leftarrow$ attributi

Cioè si ha questo albero:



Un attributo è **sintetizzato** se dipende solo dagli attributi dei figli: se " $s = f(x_1, x_2, x_3)$ " allora s è sintetizzato" (sintesi dei nodi figli). Esempio: $s = x_1 + x_2 + x_3$ (questa è la regola).

Un attributo è **ereditato** se dipende solo dagli attributi dei fratelli e/o del padre: se " $x_1 = f(s, x_3)$ " allora x_1 è ereditato" (eredita informazioni dal nodo padre). Esempio $x_1 = s + x_3$ (questa è la regola).

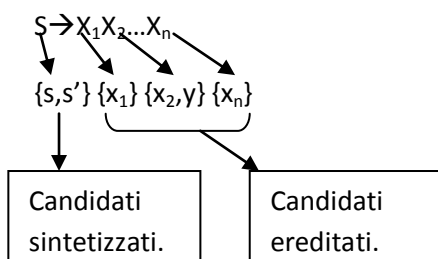
Di solito gli attributi sintetizzati sono solo quelli dei simboli che compaiono alla parte sinistra di una produzione. Relativamente alla produzione:

$S \rightarrow X_1 X_2 \dots X_n$ si ha:

- ✓ Gli attributi di S possono essere sintetizzati;
- ✓ Gli attributi di $X_1 \dots X_n$ possono essere ereditati;

Bisogna stare attenti a non definire dipendenze cicliche tra gli attributi; grammatiche con tali dipendenze non sono buone, non sono risolvibili. È possibile che siano presenti attributi né ereditati né sintetizzati.

Ad esempio:



Se $s = s' + 1$ allora " s " non è sintetizzato (si dice che è quasi sintetizzato).

Se $x_2 = x_1$ allora " x_2 " è ereditato.

Se $y = x_2 + 1$ allora " y " non è ereditato (si dice che è quasi ereditato).

" s " nella lista viene prima di " s " e quindi s è quasi sintetizzato. Anche " x_2 " e " y " rispondono a questa pseudo-regola. Le regole semantiche costruiscono le dipendenze tra gli attributi, i quali saranno rappresentati in un grafo. Una regola semantica può avere anche altri effetti: stampare un valore, modificare una variabile globale etc... un albero di derivazione che mostra i valori degli attributi per ogni

nodo è detto “albero di derivazione annotato”. A volte vi sono regole semantiche che definiscono attributi sintetizzati **fittizi**.

Vediamo un esempio: data la grammatica

$L \rightarrow En$
 $E \rightarrow E1+T$
 $E \rightarrow T$
 $T \rightarrow T1 * F$
 $T \rightarrow F$
 $F \rightarrow (E)$
 $F \rightarrow \text{digit}$

Costruiamo la definizione guidata dalla sintassi (la grammatica ad attributi) per valutare le espressioni generate da tale grammatica. Ad esempio: $3*5+4$;

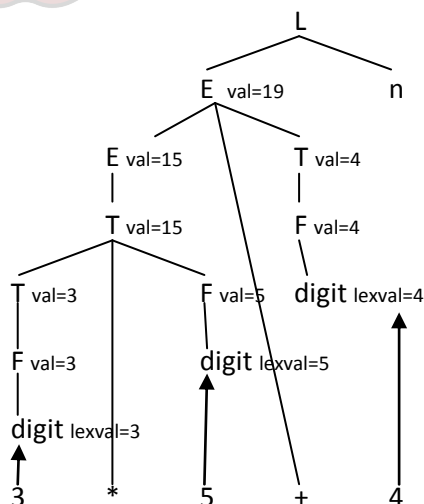
La sola grammatica scritta ci dice solo che la frase “ $3*5+4$ ” è corretta, ma non sa valutarla; introduciamo ora delle regole che ci permettono di tradurre la frase in input “ $3*5+4$ ” nel suo valore finale 19. Ad ogni simbolo non terminale (L,E,T,F) si associa l’attributo val(valore). A digit si associa l’attributo lexval (yyval in YACC; il valore del digit è restituito dall’analisi lessicale). Vogliamo introdurre regole in modo tale che dopo aver fatto l’ultima riduzione in L.val rimanga 19, così da poter stampare il risultato corretto. Si è già evidenziata la prima regola semantica:

1. $L \rightarrow En$ L.val = print(E.val)

Attributo fittizio sintetizzato.

L’azione da effettuare in corrispondenza della riduzione $L \rightarrow En$ è stampare il valore di E.val.

Il valore di E lo si deve calcolare durante la fase di parsing. Costruiamo prima l’albero, dal quale ricaveremo le altre regole semantiche:



Dall’analisi lessicale sappiamo i valori degli attributi lexval del token digit.

Per valutare ho bisogno di un attributo che contenga il valore dell’espressione. Osservando l’albero si costruiscono le regole da associare alle produzioni; si fanno migrare in alto i valori degli attributi.

$E \rightarrow E1+T$ $E.val = E1.val + T.val$
 $E \rightarrow T$ $E.val = T.val$
 $T \rightarrow T1 * F$ $T.val = T1.val * F.val$

$T \rightarrow F$

$T.val = F.val$

$F \rightarrow (E)$

$F.val = E.val$

$F \rightarrow \text{digit}$

$F.val = \text{digit.lexval}$

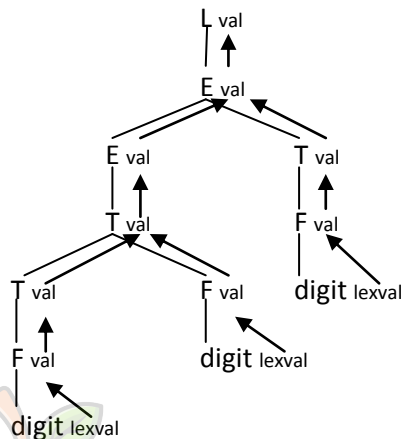
Le parentesi non
influenzano l'ordine di
valutazione.

Abbiamo praticamente ottenuto la definizione guidata dalla sintassi.

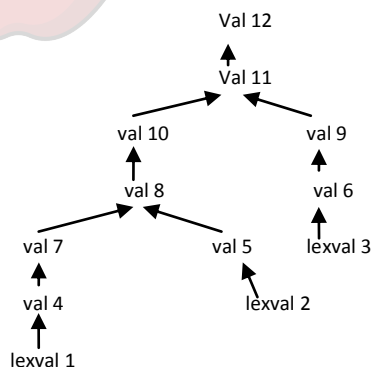
Per poter stampare correttamente il valore dell'espressione bisogna fare una visita bottom-up dell'albero.

Si può fare contemporaneamente la costruzione dell'albero e dei valori degli attributi in modo bottom-up.

Una visita top-down dell'albero è errata, poiché come prima cosa porterebbe a stampare il valore di $E.val$ che non è stato ancora calcolato. Per le grammatiche ad attributi è essenziale fare una corretta visita degli alberi a seconda della dipendenza degli attributi. Vediamo ora l'albero annotato (albero con nodi più attributi):



L'albero evidenzia le dipendenze tra gli attributi. La freccia verso l'alto indica la dipendenza tra i val (tra gli attributi). Gli attributi dei nodi foglia (i lexval) non hanno archi in entrata e quindi non dipendono da nessun attributo. Abbiamo costruito un grafo sulla frase applicando la grammatica. Su tale grafo si può effettuare l'ordine topologico dei nodi:



I numeri vanno ad indicare l'ordine. Fatto l'ordine topologico si possono calcolare correttamente gli attributi, seguendo l'ordinamento fatto. **Riassumendo i passi da fare in generale sono:**

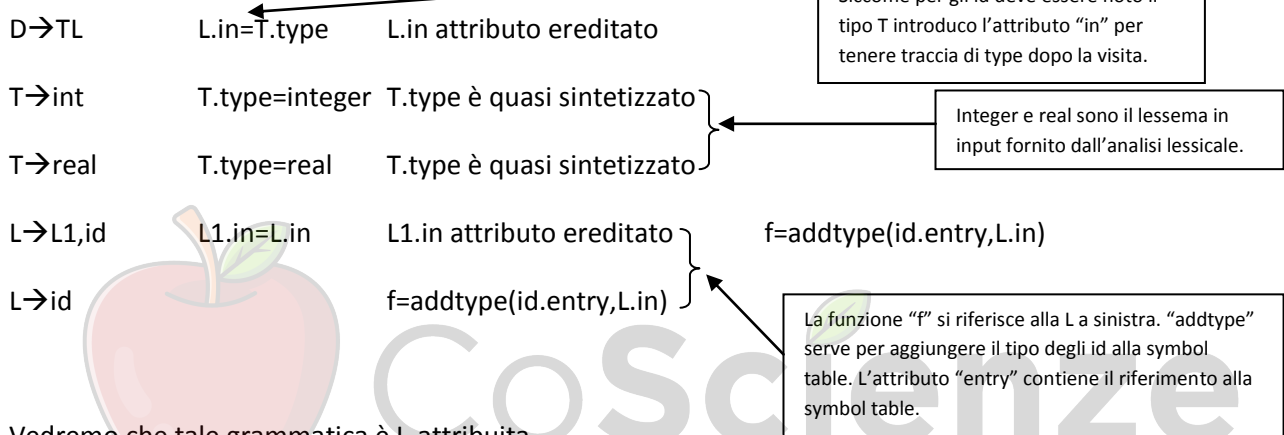
1. **Costruzione dell'albero di parsing;**
2. **Annotazione dell'albero (aggiunta di attributi);**
3. **Grafo delle dipendenze;**
4. **Ordine topologico;**
5. **Calcolo degli attributi.**

L'ordine topologico è un algoritmo del tipo: annota i nodi che non hanno archi entranti e cancella gli archi uscenti da tali nodi; ripeti tale procedimento. Vi sono due classi di grammatiche ad attributi per le quali si può evitare di fare l'ordinamento topologico.

Grammatiche S-attribuite: tutti gli attributi sono sintetizzati; si capisce che per tali grammatiche una visita bottom-up è sempre corretta. ($L.val = E.val + T.val$ indica che l'attributo di sinistra dipende dagli attributi di destra).

Grammatiche L-attribuite: vedremo in seguito tali grammatiche; sarà corretta, in ogni caso, una visita depth-first top-down; non sempre la visita bottom-up è corretta. In questo tipo di grammatiche sono presenti attributi ereditati.

Se la grammatica non è né S-attribuita, né L-attribuita allora bisogna fare l'ordine topologico per decidere come visitare l'albero. La grammatica dell'esempio è S-attribuita, quindi non era necessario fare l'ordinamento topologico, poiché una qualsiasi visita bottom-up è corretta per la valutazione dell'espressione. Fare l'ordinamento topologico è un'operazione costosa. Cosa fa la grammatica ad attributi dell'esempio precedente? Riconosce le espressioni aritmetiche e dà il valore finale dell'espressione come risultato della traduzione. Una grammatica ad attributi oltre al riconoscimento fa anche altro; riconosce un'espressione aritmetica e ci dice quanto vale. Vediamo un esempio: consideriamo la grammatica ad attributi:



Vedremo che tale grammatica è L-attribuita.

Una dichiarazione generata da D in questa definizione consiste della parola "int" o "real" seguita da una lista di identificatori. Le regole associate alle produzioni per L chiamano la funzione addtype per aggiungere il tipo di ogni identificatore nella tabella dei simboli. Le "f" sono attributi fittizi, si introducono ogni qual volta ci sono chiamate a funzioni che non restituiscono niente; indicano solo che si esegue la funzione. Consideriamo che la dichiarazione da riconoscere è

real id1, id2, id3;

Vediamo quali sono i passi da seguire:

1. COSTRUZIONE DELL'ALBERO DI DERIVAZIONE DELLA FRASE:

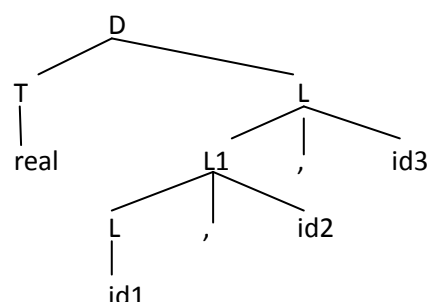


Tabella dei simboli:

Nome	...	Tipo
id1	...	?
id2	...	?
id3	...	?

Dobbiamo riempire la parte tipo!

2. ANNOTIAMO L'ALBERO CON GLI ATTRIBUTI:

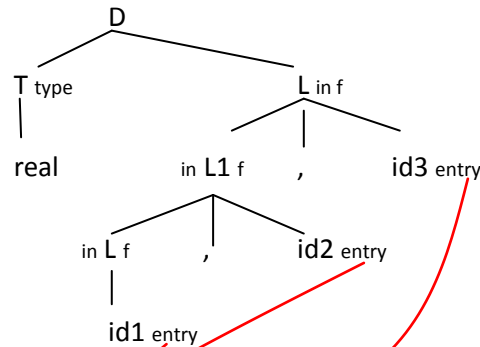


Tabella dei simboli:

Nome	...	Tipo
id1	...	?
id2	...	?
id3	...	?

Gli attributi entry sono dati dall'analisi lessicale, sono i puntatori alla tabella dei simboli, i lessemi. La funzione addtype ha come argomenti: un puntatore ed un tipo; ciò che fa è mettere il tipo nel campo tipo puntato dal puntatore nella tabella dei simboli.

3. COSTRUZIONE GRAFO DELLE DIPENDENZE:

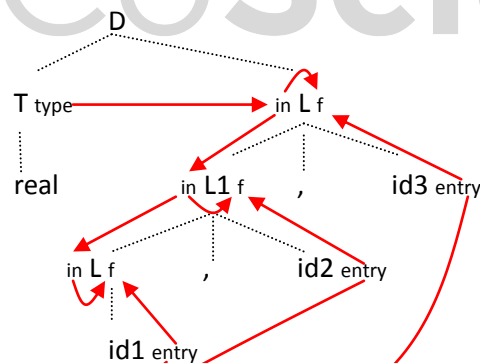


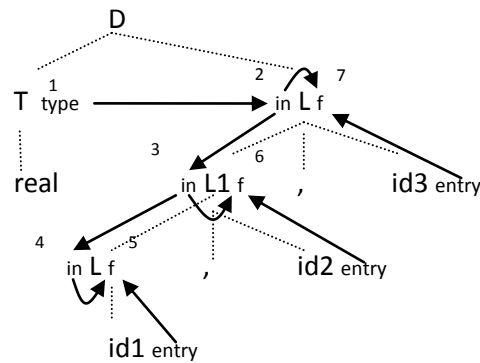
Tabella dei simboli:

Nome	...	Tipo
id1	...	?
id2	...	?
id3	...	?

Gli attributi fittizi si introducono per poter costruire il grafo delle dipendenze.

"In" e "type": sono variabili temporanee locali il cui unico scopo è far passare il tipo (real) nell'albero per farlo arrivare agli identificatori (id1, id2, id3). Cosa fa la grammatica attribuita?? Assegna ad ogni identificatore il proprio tipo nella tabella dei simboli (fase semantica) oltre a riconoscere dichiarazioni di tipo.

4. SI EFFETTUA L'ORDINE TOPOLOGICO



Secondo tale ordinamento, si visiterà l'albero da sinistra a destra. In tale grammatica tutti gli attributi ereditati dipendono da fratelli sinistri; quando ciò accade la visita da sinistra a destra è sempre corretta. Se un attributo dipende da un fratello destro la visita da sinistra a destra non è più corretta.

5. CALCOLO DEGLI ATTRIBUTI:

Valutiamo il valore assunto dagli attributi, seguendo l'ordine topologico; in tale modo si ottiene il seguente programma:

```
type1=real
in2=type1
in3=in2
in4=in3
f=addtype(id1.entry,in4)
f=addtype(id2.entry,in3)
f=addtype(id3.entry,in2)
```

Il cui effetto è riempire, nella tabella dei simboli il campo tipo degli identificatori con il valore "real". La tabella diventa:

Nome	...	Tipo
id1	...	real
id2	...	real
id3	...	real

$S.val$
 $\swarrow \quad \downarrow \quad \searrow$
 $X1 \quad X2 \quad X3$

$S \rightarrow X1 X2 X3$

$S.val = X1.val + X2.val$

$X1.val = S.val - X2.val$

Se la grammatica ha cicli non si può fare l'ordine topologico e si va in dead-lock.

L'esempio visto ha permesso di mettere, nella tabella dei simboli il tipo delle variabili dichiarate. Abbiamo finito la prima parte del controllo di tipo.

LEZIONE 15 – ANALISI SEMANTICA (2 PARTE)

29/04/2009

COSTRUZIONE DEGLI ALBERI DI SINTASSI: SYNTAX TREE

Vediamo la differenza tra PARSER TREE e SYNTAX TREE. Vediamo un esempio e consideriamo la grammatica:

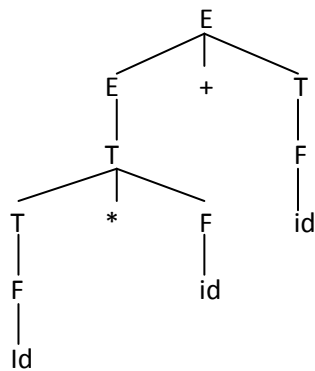
$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

Se in input si ha la frase "id*id+id" si ha:

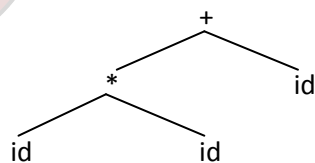
PARSER TREE:



Il parser tree tiene conto di tutti i non terminali e l'abbiamo visto come una struttura logica. Lo possiamo considerare un albero di derivazione: tiene conto di tutte le produzioni necessarie per riconoscere la frase.

SYNTAX TREE:

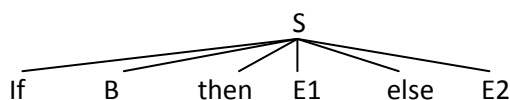
Come formato intermedio del programma non ci serve sapere quali sono le produzioni utilizzate, ma vogliamo esprimere una struttura sintattica minimale. Un syntax-tree riassume in modo sintetico le informazioni contenute nel parser-tree che servono:



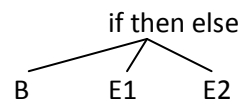
Ad esempio se si usa la produzione:

$S \rightarrow \text{if } B \text{ then } E1 \text{ else } E2$

Si ha questa differenza nei due alberi:



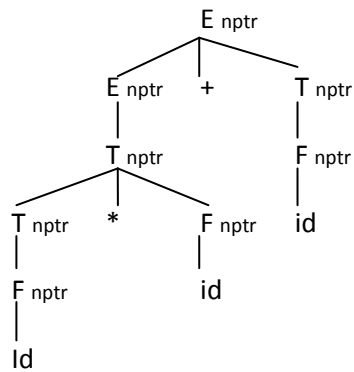
PARSER TREE



SYNTAX TREE

Il syntax tree si svincola dalla grammatica nel senso che non tiene conto dei non terminali; infatti nei syntax tree i non terminali (che servono per specificare le regole da applicare alle sentenze) non compaiono. In un syntax tree gli operatori e le parole chiavi sono associati a nodi interni e mai a foglie.

Vediamo come costruire il syntax tree attraverso la traduzione guidata dalla sintassi:



Con le seguenti regole:

$E \rightarrow E+T$ $E.nptr = \text{newNode}(+, E1.nptr, T.nptr)$

$T \rightarrow T * F$ $T.nptr = \text{newNode}(*, T.nptr, F.nptr)$

$F \rightarrow id$ $F.nptr = \text{newLeaf}(id.entry)$

$E \rightarrow T$ $E.nptr = T.nptr$

$T \rightarrow F$ $T.nptr = F.nptr$

Vediamo un altro esempio su un'altra grammatica:

$E \rightarrow E1+T$

$E \rightarrow E1-T$

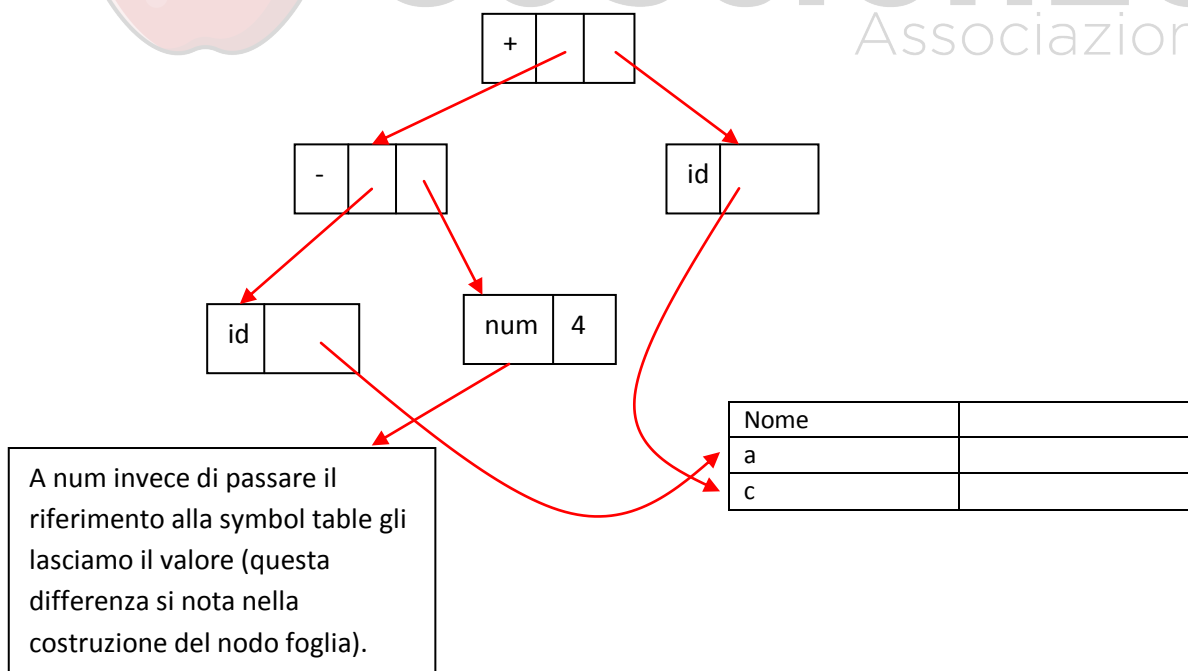
$E \rightarrow T$

$T \rightarrow (E)$

$T \rightarrow id$

$T \rightarrow num$

E la frase in input "a-4+c" vogliamo costruire fisicamente l'albero a puntatori:



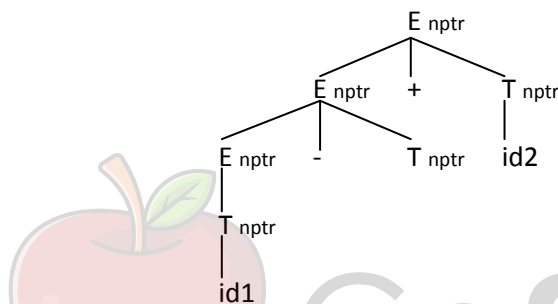
Questo è l'albero a puntatori che rappresenta l'albero sintattico della frase in input. Ogni nodo dell'albero è implementato con un record con più campi:

- In un nodo operatore, un campo identifica l'operatore ed i restanti campi sono puntatori ai nodi operandi;
- In un nodo operando, un campo ha il token dell'operando; vi sarà un altro campo che contiene i valori o i puntatori ai valori degli attributi dell'operando nel nodo.

Al posto dell'albero di derivazione si usa il syntax tree poiché è un albero minimale e risparmia memoria. Tale albero sarà costruito dall'analisi sintattica aumentata con regole semantiche. Le funzioni per creare i nodi dell'albero visto restituiscono un puntatore al nodo creato e sono:

- Mknod(op, left, right): crea un nodo operatore con label "op" e due campi contenenti i puntatori al figlio sinistro ed al figlio destro del nodo creato.
- Mkleaf(id,entry): crea un nodo con label "id" ed un campo contenente un puntatore entry all'elemento nella tabella dei simboli corrispondente ad id.
- Mkleaf(num,val): crea un nodo con label num ed un campo contenente il valore del numero.

Tali funzioni dipendono dalla grammatica e dall'albero che si vuole costruire. Il syntax tree si costruisce a partire dal parser-tree annotato: aggiungendo alla grammatica le corrette regole semantiche che costruiscono il syntax-tree. Parser tree annotato:



A partire da tale albero dobbiamo aggiungere delle regole semantiche alla grammatica in modo tale che, eseguendo tali regole secondo un ordine topologico del parser-tree, si ottiene la costruzione del syntax tree. Aumentiamo la grammatica con le regole semantiche:

$E \rightarrow E_1 + T$ $E.nptr = \text{mknod}(+, E_1.nptr, T.nptr)$
 $E \rightarrow E_1 - T$ $E.nptr = \text{mknod}(-, E_1.nptr, T.nptr)$
 $T \rightarrow (E)$ $T.nptr = E.nptr$
 $T \rightarrow \text{id}$ $T.nptr = \text{mkleaf}(\text{id}, \text{id.entry})$
 $T \rightarrow \text{num}$ $T.nptr = \text{mkleaf}(\text{num}, \text{num.val})$

Puntatore alla tabella dei simboli.

Per vedere in che ordine eseguire le regole per costruire un syntax tree non è necessario fare un ordine topologico del parser-tree poiché la grammatica è s-attribuita, quindi una qualsiasi visita bottom-up è corretta. Una possibile visita bottom up porta all'esecuzione, nell'ordine, delle seguenti azioni:

$T.nptr = \text{mkleaf}(\text{id1}, \text{id1.entry})$

$E.nptr = T.nptr$

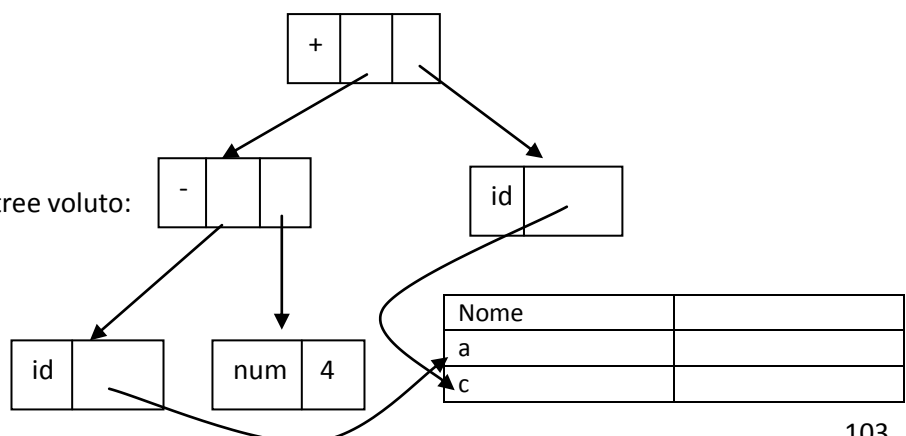
$T.nptr = \text{mkleaf}(\text{num}, \text{num.val})$

$E.nptr = \text{mknod}(-, E.nptr, E.nptr)$

$T.nptr = \text{mkleaf}(\text{id2}, \text{id2.entry})$

$E.nptr = \text{mknod}(+, E.nptr, T.nptr)$

Il cui effetto è costruire il syntax tree voluto:



Cosa fa tale grammatica?? Costruisce il syntax tree.

VALUTAZIONE BOTTOM-UP PER LE GRAMMATICHE S-ATTRIBUITE

Per le grammatiche s-attribuite la visita bottom-up per la valutazione dell'input la si può fare durante la costruzione dell'albero di parsing; quindi si può fare simultaneamente PARSE LR (bottom-up) e valutazione degli attributi. Quindi la valutazione degli attributi può essere implementata con l'aiuto di un generatore di parser LR (yacc). La valutazione è fatta usando lo stack del parser LR.

Nel parser LR quando si raggiungeva un item completo $S \rightarrow XYZ \bullet$ significava che nello stack vi era:

Z
Y
X
...

L'azione da eseguire era una riduzione con la produzione $S \rightarrow XYZ$ ed il contenuto dello stack diventa:

S
...

vediamo come sfruttare tale procedimento per i nostri scopi.

Per una grammatica s-attribuita, ad ogni produzione è associata una regola del tipo:

$S \rightarrow XYZ$ $S.s = X.x + Y.y + Z.z$
 $S.s = F(X.x, Y.y, Z.z)$

S.s attributo sintetizzato di S.

Ciò che si fa è di considerare insieme allo stack di parsing uno "stack semantico" che contiene i valori degli attributi sintetizzati; quindi nello stack, ad ogni simbolo è associato il valore dell'attributo corrispondente:

- Se il simbolo è terminale, l'attributo è dato dall'analisi lessicale;
- Se il simbolo è un non terminale si possono verificare due casi:
 - Non ha attributo; in questo caso il campo valore è indefinito;
 - Ha attributo; l'attributo sarà sintetizzato e vi sarà il suo valore calcolato con la corretta regola.

Con l'aggiunta del nuovo campo nello stack si avrà la seguente situazione:

		top
Z	Z.z	
Y	Y.y	top-1
X	X.x	
...	...	top-2

VAL=valore attributo Stack

Prima di applicare la riduzione con la produzione $S \rightarrow XYZ$, si eseguirà la regola che ci permette di calcolare l'attributo sintetizzato di S:

$val[ntop] = val[top-2] + val[top-1] + val[top]$

Nuovo top

Tale regola calcola S.s; facendo poi la riduzione il contenuto dello stack sarà:

		nTop
S	S.s	

VAL Stack semantico

La valutazione degli attributi sarà sempre corretta poiché l'albero di parsing lo si sta costruendo in modo bottom-up. Il modo di operare descritto è proprio il funzionamento di YACC (costruisce un parser bottom-

up usando grammatiche ad attributi). Quando si effettua una riduzione con una produzione che ha r simboli sul lato destro, il valore di $ntop$ sarà $top-r+1$. Vediamo che ciò che abbiamo descritto è il meccanismo con cui funziona YACC. In YACC si avrebbe:

$\$ \$ \quad \$1 \ \$2 \ \$3 \ \$4 \ \5

S: $X + Y + Z \quad \{\$ \$ = \$1 + \$3 + \$5\}$

Yacc usa la notazione $\$ \$$, $\$1$, $\$3$, $\$5$ poiché non ci dice esplicitamente come si chiama lo stack che usa.

- ✓ $\$ \$$ è il valore semantico di S.s, cioè $val[ntop]$;
- ✓ $\$1$ è il valore dell'attributo X.x, cioè $val[top-2]$;
- ✓ $\$3$ è il valore dell'attributo Y.y, cioè $val[top-1]$;
- ✓ $\$5$ è il valore dell'attributo Z.z, cioè $val[top]$.

Quindi YACC implementa la valutazione degli attributi durante la costruzione del Parser-tree (solo per Grammatiche S-attribuite). Se si vuole usare YACC per una traduzione guidata della sintassi (valutazione dell'input), la grammatica deve essere S-attribuita; se non lo è si deve cercare di renderla tale. (In YACC vi è una variabile "yystype" per definire lo stack semantico; per default yystype è uno stack di interi, ma a seconda dei casi lo si può definire correttamente). Se un simbolo ha più attributi si può definire uno stack di record per contenere i valori degli attributi. Quindi $\$1$, $\$...$ è una notazione che usiamo per semplicità, la notazione reale che fa riferimento allo stack di YACC userà yystype.

ES.

Produzioni	Frammento di codice
$L \rightarrow En$	Print ($val[top]$)
$E \rightarrow E1 + T$	$Val[ntop]=val [top-2] + val [top]$
$E \rightarrow T$	
$T \rightarrow T1 + F$	$Val[ntop]=val [top-2] + val [top]$
$T \rightarrow F$	
$F \rightarrow (E)$	$Val[ntop]=val [top-1]$
$F \rightarrow digit$	

Se l'input è $3*5+4n$, le sequenze di mosse vengono fatte dal parser sull'input, considerando anche la valutazione degli attributi sono:

INPUT	STACK		PRODUZIONE USATA
		VAL	
$3*5+4n$			
$*5+4n$	3	3	
$*5+4n$	F	3	$F \rightarrow digit$
$*5+4n$	T	3	$T \rightarrow F$
$5+4n$	T^*	3-	
$+4n$	T^*5	3-5	
$+4n$	T^*F	3-5	$F \rightarrow digit$
$+4n$	T	15	$T \rightarrow T1^*F$
$+4n$	E	15	$E \rightarrow T$
$4n$	E^+	15-	
n	E^+4	15-4	
n	E^+F	15-4	$F \rightarrow digit$
n	E^+T	15-4	$T \rightarrow F$
n	E	19	$E \rightarrow E1+T$
	En	19-	
	L	19	$L \rightarrow En$

Si osserva che dopo ciascuna riduzione, il top di val contiene il valore dell'attributo associato al lato sinistro della produzione con cui abbiamo ridotto.

LEZIONE 16 – ANALISI SEMANTICA (3 PARTE)

04/05/2009

GRAMMATICA L-ATTRIBUITA

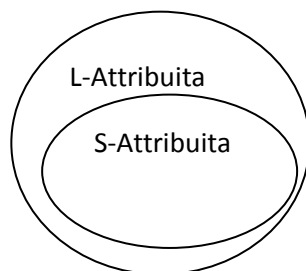
Una grammatica ad attributi è L-attribuita se gli attributi rispettano le seguenti proprietà:

- Sintetizzati: nessun vincolo;
- Ereditati: possono dipendere da:
 1. Attributi ereditati del padre;
 2. Attributi dei fratelli sinistri (o ereditati o sintetizzati);

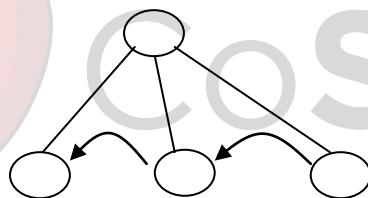
Se abbiamo la produzione $A \rightarrow X_1 X_2 \dots X_n$ gli attributi ereditati di X_j , con $1 \leq j \leq n$, possono dipendere solo da:

- Gli attributi ereditati di A;
- Gli attributi di $X_1 \dots X_{j-1}$;

Si osserva che ogni grammatica S-attribuita è anche L-attribuita.



Anche per le grammatiche L-attribuite si può evitare l'ordine topologico, una visita depth-first è sempre corretta (segue tutti i cammini da sinistra a destra).

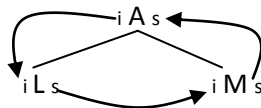


Non possiamo avere tali dipendenze tra gli attributi per una gramm. L-attrib

Ad esempio:

$A \rightarrow LM$

1. $L.i = l(A.i)$ "L.i è ereditato"
2. $M.i = m(L.s)$ "M.i è ereditato non puro – quasi ereditato"
3. $A.s = f(M.s)$ "A.s è sintetizzato".

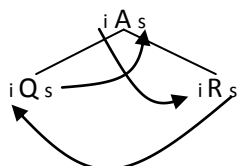


Una visita depth-first è corretta (da sinistra a destra) e possiamo sempre calcolare gli attributi voluti. La grammatica è infatti L-attribuita.

Altro esempio:

$A \rightarrow QR$

1. $R.i = r(A.i)$ "R.i è ereditato"
2. $Q.i = q(R.s)$ "Q.i è quasi ereditato e dipendente dal fratello destro"
3. $A.s = f(Q.s)$ "A.s è sintetizzato"

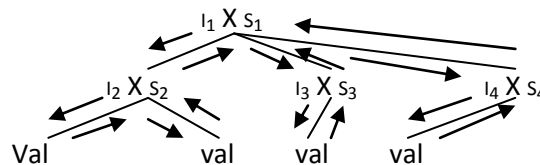


Una visita depth-first non è corretta, poiché per calcolare $Q.i$ serve conoscere $R.s$ ma con una visita depth-first si incontra prima $Q.i$ e poi $R.s$ per cui non è possibile calcolare $Q.i$. Infatti G non è L-attribuita poiché si ha la regola $Q.i = q(R.s)$ cioè un candidato attributo ereditato dipende da un attributo del fratello destro. In una visita depth-first dell'albero l'algoritmo che valuta gli attributi opera come segue:

La prima volta che incontra un nodo calcola tutti gli attributi "ereditati" di quel nodo. L'ultima volta che incontra un nodo calcola tutti gli attributi "sintetizzati" del nodo.

Ad esempio:

indichiamo con "i" gli attributi ereditati e con "s" gli attributi sintetizzati, una visita depth-first dell'albero annotato:



Valuterà gli attributi nel seguente ordine:

$I_1, I_2, S_2, I_3, S_3, I_4, S_4, S_1$

Abbiamo esplorato l'albero senza necessità di ricorrere all'ordine topologico.

- Gli S_j sono sintetizzati quindi possono dipendere da un qualsiasi figlio, per cui lo si calcola dopo che sono stati visitati tutti i figli.
- Gli I_j sono ereditati quindi non possono dipendere dai sintetizzati dal padre poiché un sintetizzato del padre può dipendere da altri nodi, anche da un fratello destro del nodo (che non abbiamo ancora visitato). Possono dipendere da un ereditato del padre o da un attributo dei fratelli sinistri, attributi che sono già stati tutti calcolati.

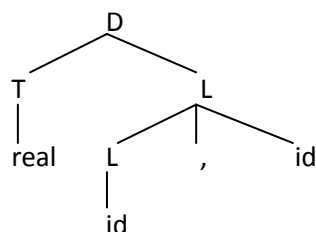
DEFINIZIONI GUIDATE DALLA SINTASSI

Sono grammatiche attribuite in cui le regole sono scritte sempre alla fine delle produzioni. Significa che non è specificato quando la regola va eseguita. Per le grammatiche S-attribuite tale posizione delle regole è sempre corretta poiché per poter applicare la regola bisogna prima avere calcolato gli attributi dei simboli alla destra della produzione. Può capitare di dovere applicare la regola prima di avere valutato tutti gli attributi della parte destra della produzione; ciò può accadere per gli attributi ereditati.

Ad esempio

$D \rightarrow TL$

" $L.in = T.type$ "



In questo caso si deve eseguire la regola prima di vedere L, poiché vogliamo mandare il type ai figli di L. Se in una grammatica attribuita è specificato dove va eseguita la regola è chiamata SCHEMA DI TRADUZIONE.

Ad esempio: $D \rightarrow T\{L.in = T.type\}L$ è uno schema di traduzione. Viene detto schema di traduzione perché aggiungiamo delle regole per tradurre un linguaggio di alto livello in linguaggio macchina.

La regola è inserita subito dopo la T.

Nel caso di grammatiche S-attribuite la “definizione guidata dalla sintassi” e lo “schema di traduzione” coincidono poiché l’azione va sempre alla fine delle produzioni. Vedremo come tradurre una definizione guidata dalla sintassi, L-attribuita, in uno schema di traduzione. Le grammatiche ad attributo sono lo strumento usato per fare un qualsiasi tipo di traduzione. Le grammatiche ad attributi generali le abbiamo date in forma di definizioni guidate dalla sintassi, cioè le regole sugli attributi sono poste alla fine delle produzioni. L’ordine di valutazione delle regole dipende dall’ordine topologico sul grafo delle dipendenze. Ad esempio riprendendo lo schema precedente:

SCHEMA DI TRADUZIONE

$T \rightarrow \text{real} \{T.\text{type} = \text{real}\}$ “tale regola la si calcola dopo aver visto real. T.type è sintetizzato.”

$D \rightarrow T\{L.\text{in} = T.\text{type}\}L$ “L.in è un attributo ereditato, quindi lo si calcola non appena si incontra L. T.type deve essere già stato valutato”. Se la regola fosse posta alla fine della produzione non potremo propagare real ai figli di L.

Quindi:

- Se in una produzione, un simbolo ha un attributo ereditato, calcolato dalla regola associata alla produzione, allora la regola è posta subito prima del simbolo.
- Le regole che calcolano attributi sintetizzati vanno alla fine delle produzioni, oppure possono essere inserite in qualsiasi posto nel quale sono già noti gli attributi da cui dipende l’attributo sintetizzato da calcolare cioè:

$S \rightarrow X_1 X_2 X_3 X_4 X_5$ $S.s = f(X_2.x, X_3.x)$

Poiché dopo aver visto X_3 possiamo calcolare S.s.

Regola principale delle grammatiche ad attributi:

Calcolare gli attributi quando conosciamo tutti gli argomenti. Così facendo si ottiene uno schema di traduzione.

Ad esempio:

costruiamo uno “schema di traduzione” che traduce una espressione infissa in una espressione postfissa:

espressione infissa	\rightarrow	espressione postfissa
$5+4-2$	\rightarrow	$5\ 4+2-$

Lo schema di traduzione deve riconoscere le espressioni infisse e deve stampare le corrispondenti espressioni postfisse. Vediamo una grammatica che genera le espressioni aritmetiche (in forma infissa):

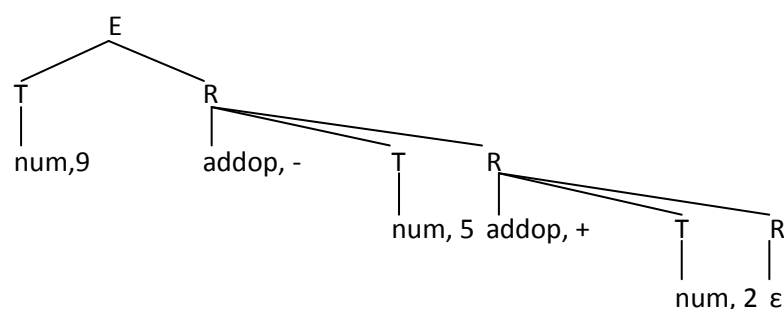
$E \rightarrow TR$ $R \rightarrow \text{addop } TR/\epsilon$ $T \rightarrow \text{num}$

Che è una grammatica L-attribuita.

“Addop” e “num” sono token:

- ✓ addop è il token che individua l’operatore “+” oppure “-” (lessema).
- ✓ num è il token che individua i numeri.

Vediamo come costruire le regole e dove inserirle per ottenere lo schema di traduzione voluto: consideriamo l’espressione $9-5+2$, costruiamo l’albero di derivazione per tale espressione:



Lo schema di traduzione dovrà stampare l'espressione postfissa 9 5-2+. Facciamo una visita depth-first dell'albero e vediamo quando effettuare le stampe per ottenere l'output voluto:

$E \rightarrow TR$

"quando si fa tale riduzione non c'è niente da stampare"

$R \rightarrow \text{addop } T\{\text{print}(\text{addop}, \text{lexeme})\}R \mid \epsilon$

Farla qui produrrebbe un errore (non abbiamo ancora il lessema di addop)

Se l'inserimento avvenisse in questo punto, avremmo la stampa dell'espressione in forma infissa.

$T \rightarrow \text{num}\{\text{print}(\text{num.val})\};$

Bisogna controllare che le regole aggiunte siano corrette in ogni contesto della grammatica e non in particolari situazioni.

Ad esempio: Costruire uno schema di traduzione tale che:

INPUT: stringa di caratteri;

OUTPUT: quante occorrenze di ogni carattere ci sono

abcbcc \rightarrow 1,2,3

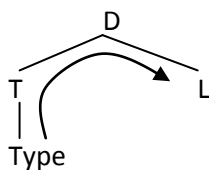
Costruire uno schema di traduzione che costruisce l'inversa della frase data in input:

abbcba \rightarrow bcbba

per tali esempi si costruisce l'albero di derivazione della frase in input, e sull'albero si vede dove si devono fare le print durante una visita. Se si usa Yacc, che usa solo grammatiche S-attribuite, bisognerà fare una visita bottom-up dell'albero (costruire la derivazione right-most inverso dell'input). Con Yacc si possono usare solo regole sintetizzate. Se la grammatica è L-attribuita e si vuole usare Yacc bisogna cercare di trasformarla in una S-attribuita equivalente oppure usare degli artifici. Ad esempio:

$D \rightarrow TL$

.... Che è una grammatica L-attribuita.



Non si può fare una visita bottom-up, poiché l'attributo "type" per arrivare ai discendenti di L deve salire per T e poi passare ad L. Così riusciamo ad avere, nelle foglie di L, "real" ed "il puntatore alla tabella dei simboli" così da poter fare l'"addtype". Se si vuole usare Yacc, cioè fare una visita bottom-up, una possibilità è conservare i puntatori, alla tabella dei simboli, in una struttura globale, quando si ottiene il tipo, si può scrivere una regola sintetizzata che scorre la struttura con i puntatori e riempie la tabella dei simboli con il tipo corretto. È un possibile trucco. Non sappiamo quanto può essere grande la struttura ausiliaria.

LEZIONE 17 – ANALISI SEMANTICA (4 PARTE)

06/05/2009

VALUTAZIONE BOTTOM-UP DEGLI ATTRIBUTI EREDITATI

Vediamo come trasformare uno schema di traduzione in modo che tutte le azioni si trovino alla fine delle produzioni. Si inseriscono dei nuovi simboli non terminali, dei marcatori, che generano ϵ nella grammatica di base. Ogni azione è rimpiazzata con un distinto marcatore M, l'azione sarà posta alla fine della produzione $M \rightarrow \epsilon$.

Ad esempio lo schema di traduzione:

$E \rightarrow TR$

$R \rightarrow +T\{\text{print}('+\')\}R \mid -T\{\text{print}('-',)\}R \mid \epsilon$

$T \rightarrow \text{num}\{\text{print}(\text{num.val})\}$

Usando i marcatori M ed N diventa:

$E \rightarrow TR$

$R \rightarrow +TMR \mid -TNR \mid \epsilon$

$T \rightarrow \text{num}\{\text{print}(\text{num.val})\}$

$M \rightarrow \epsilon\{\text{print}('+\')\}$

$N \rightarrow \epsilon\{\text{print}('-',)\}$

Così trasformata, la grammatica ad attributi, può essere implementata in Yacc (le regole non compaiono più all'interno delle produzioni). "Yacc esegue le azioni solo prima delle riduzioni, mai durante gli shift".

- ✓ Se la grammatica di partenza era LR, dopo tale trasformazione, può non essere più LR (risolve conflitti).
- ✓ Se la grammatica di partenza era LL dopo tali modifiche continuerà a godere di tale proprietà (rimane LL poiché per i marcatori introdotti vi è una sola produzione).

Vediamo un esempio: data la "definizione guidata dalla sintassi"

$D \rightarrow TL$ $L.in = T.type$

$T \rightarrow \text{int}$ $T.type = \text{int}$

$T \rightarrow \text{real}$ $T.type = \text{real}$

$L \rightarrow L1, id$ $L1.in = L.in; \text{addtype}(id.entry, L.in)$

$L \rightarrow id$ $\text{addtype}(id.entry, L.in)$

Si nota che è L-attribuita; passiamo allo "schema di traduzione":

$D \rightarrow T\{L.in = T.type\}L$

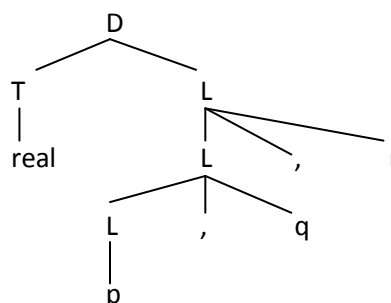
$T \rightarrow \text{int}\{T.type = \text{int}\}$

$T \rightarrow \text{real}\{T.type = \text{real}\}$

$L \rightarrow \{L1.in = L.in\} L1, id \{\text{addtype}(id.entry, L.in)\}$

$L \rightarrow id\{\text{addtype}(id.entry, L.in)\}$

Vediamo come implementarlo in Yacc (visita bottom-up): bisognerà apportare delle modifiche poiché la grammatica è L-attribuita (una possibile soluzione è conservare i puntatori alla tabella dei simboli, come descritto in precedenza). Esaminiamo le mosse fatte da Yacc (parser bottom-up) sull'input "real p,q,r"



Per vedere il funzionamento di Yacc usiamo lo stack con l'aggiunta del campo valore (valore dell'attributo sintetizzato corrispondente). Ignorando le azioni otteniamo:

INPUT	STACK		PRODUZIONE USATA
		VAL	
real p,q,r			
p,q,r	real		
p,q,r	T	real	$T \rightarrow \text{real}$
,q,r	Tp	real-p.entry	
,q,r	TL	real-p.entry	$L \rightarrow \text{id}$
q,r	TL,	real-p.entry-p.entry	
,r	TL,q	real-p.entry-p.entry-q.entry	
,r	TL	real-p.entry	$L \rightarrow L, \text{id}$
r	TL,	real-p.entry-p.entry	
4n	TL,r	real-p.entry-p.entry-q.entry-r.entry	
	TL	Real-p.entry	$L \rightarrow L, \text{id}$
	D		$D \rightarrow TL$

Si osserva che tale grammatica gode della particolare proprietà: ogni volta che si opera una riduzione ad L (o equivalentemente si esegue un addtype), T è nello stack sotto il lato destro della produzione per L; cioè T si trova sempre in un punto noto dello stack. Ciò ci consente di accedere all'attributo T.type per eseguire l'addtype. Sfruttando tale proprietà particolare possiamo implementare la grammatica attribuita in Yacc:

```

D:    TL
T:    int    {val[ntop]=int}
T:    real   {val[ntop]=real}
L:    L1, id  {addtype(val[top], val[top-3])} (in quanto val[top]= puntatore, T.s. =id.entry,
            val[top-3]=real=T.type)
L:    id      {addtype(val[top],val[top-1])}(in quanto val[top]=id.entry, val[top-1]=T.type)

```

Si nota che nell'addtype è usato direttamente T.type, e non compare L.in; per tale motivo possiamo eliminare la regola $\{L1.in=L.in\}$ che faceva scendere "real" verso il basso (nell'albero). Si è implementata la grammatica L-attribuita in Yacc sfruttando le particolari proprietà di cui gode. La ricerca di un valore di un attributo nello stack viene fatta solo se la grammatica permette di predire la posizione di tale valore.

Data una grammatica S-attribuita o L-attribuita, le regole semantiche (in uno schema di traduzione) sugli attributi, riferite ad uno stack, in una implementazione con un LR-parser (Yacc) cosa diventerebbero??

1. Se la grammatica è S-attribuita, la traduzione è semplice. Vediamo un esempio:

$S \rightarrow ABC$ $S.s = A.x + B.y$ che diventa $val[ntop] = val[top-2] + val[top-1]$ (e in Yacc diventa $$$ = $2 + 1). Lo stack è questo:

C	
B	By
A	Ax

2. Caso in cui la grammatica è L-attribuita. Consideriamo la seguente definizione guidata dalla sintassi:

```

S → aAC      C.i = A.s
S → bABC     C.i = A.s
C → c        C.s = g(C.i)

```

La grammatica non è S-attribuita poiché C.i è un attributo ereditato. È L-attribuita (**i è ereditato ed s è sintetizzato**). Per trasformarla in uno schema di traduzione potremo inserire le regole all'interno delle produzioni come segue:

$S \rightarrow aA\{C.i=A.s\}C$

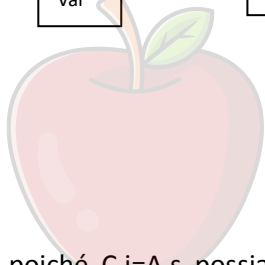
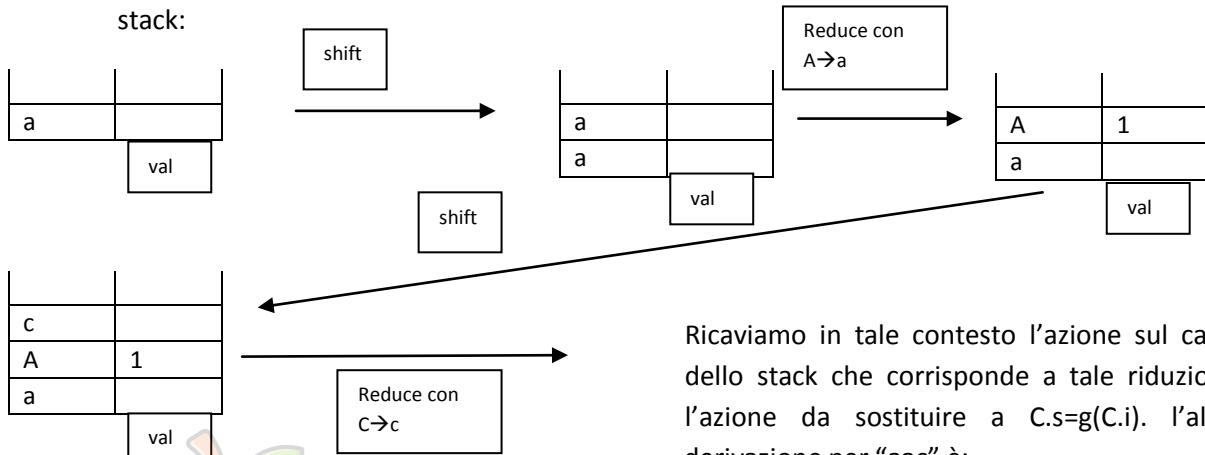
$S \rightarrow bA\{C.i=A.s\}BC$ oppure $S \rightarrow bAB\{C.i=A.s\}C$

$A \rightarrow a \quad A.s=1$

$B \rightarrow b \quad B.s=2$

Si osserva che, nello stack, tra A e C può esserci o meno B, per cui quando si effettua una riduzione con $C \rightarrow c$, il valore di C.i può trovarsi sia in $\text{val}[\text{top}-1]$ (nel caso in cui nello stack vi era aAC) che in $\text{val}[\text{top}-2]$ (nel caso in cui nello stack vi era aABC); quindi non si può fare riferimento ad un punto fisso dello stack, come nei casi precedenti.

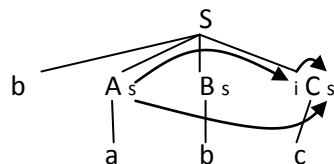
Analizziamo meglio tali situazioni facendo il parsing delle frasi "aac" e "babc" con riferimento allo stack:



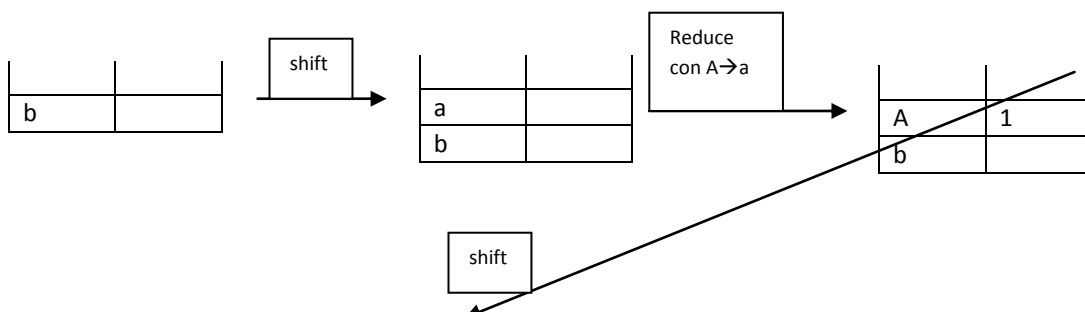
poiché $C.i=A.s$ possiamo saltare tale regola e prendere direttamente $A.s: C.s= g(A.s)$. La regola diventa: $\text{val}[\text{ntop}]=g[\text{top}-1]$:

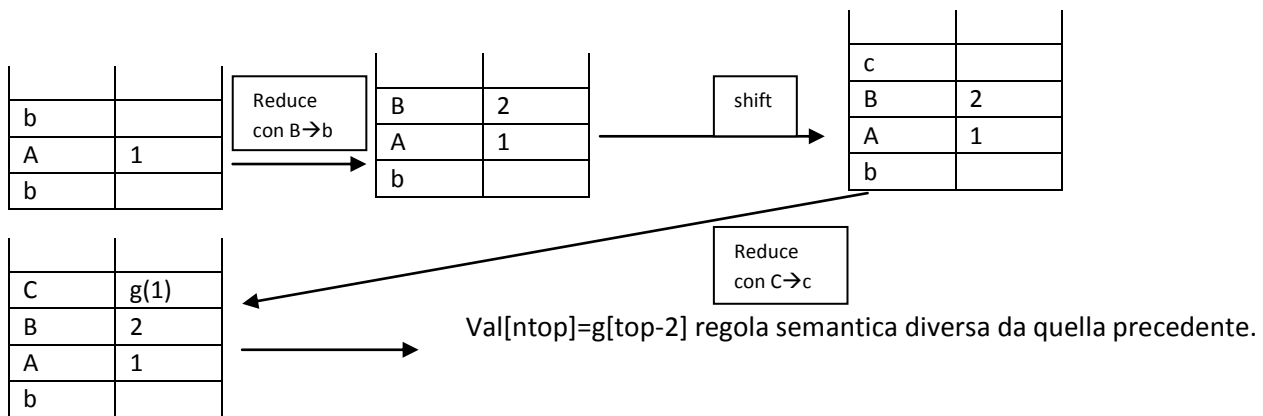
c	$g(1)$
A	1
a	

Vediamo di che regole necessita l'input "babc". Iniziamo a costruire l'albero di derivazione:



$A.s=1; C.i=A.s=1; C.s=g(C.i)=g(1);$



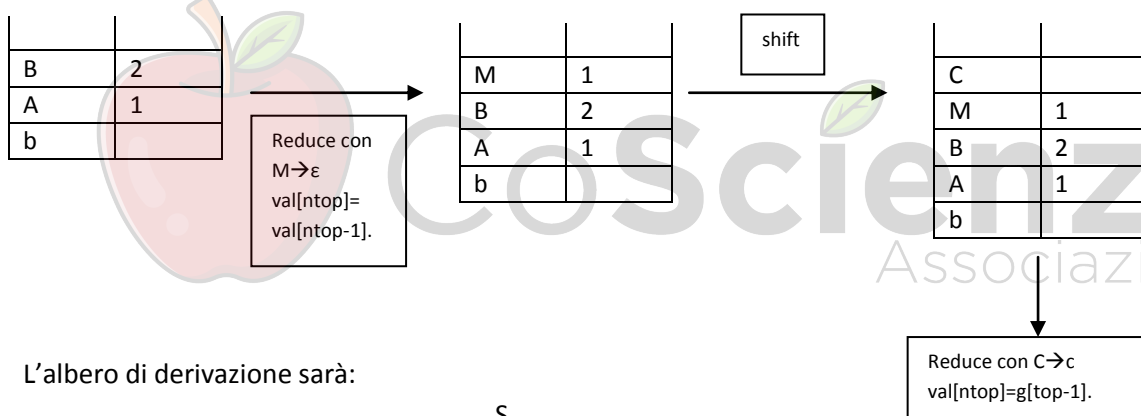


Quindi, per frasi diverse, una stessa riduzione ha bisogno di regole diverse; ma ad ogni produzione bisogna associare regole semantiche che vadano bene in ogni contesto possibile. Per tale motivo si modifica la grammatica con un artificio che elimina l'ambiguità evidenziata; si fa in modo che il valore di A.s si trovi sempre sotto la C in modo tale che la regola $val[ntop]=g[top-1]$ sia sempre corretta, in qualunque contesto; per tale motivo si introduce un marcatore M prima della C, nella seconda produzione, il cui scopo è fare arrivare il valore si A.s sotto la C (M ruba A.s per darlo a C.s):

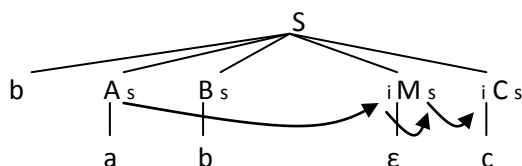
$S \rightarrow bABMC$ $M.i=A.s, C.i = M.s$

$M \rightarrow \epsilon$ $M.s=M.i$

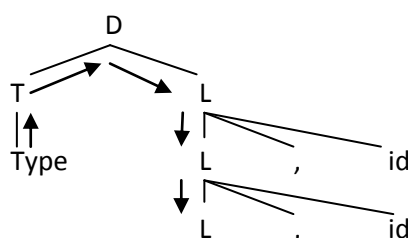
La situazione nello stack sarà la seguente:



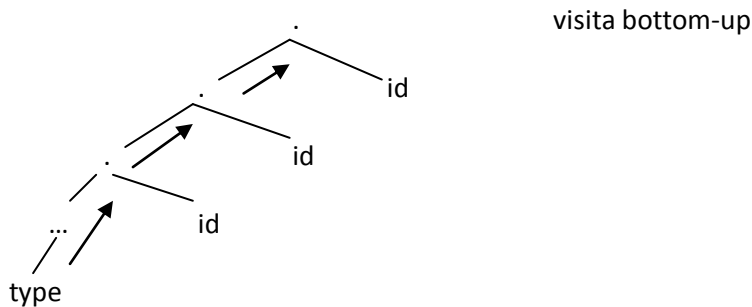
L'albero di derivazione sarà:



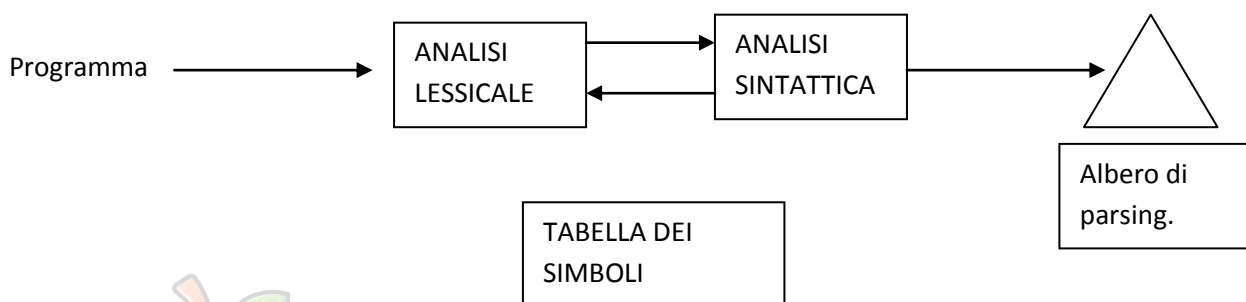
Si è così ottenuta una grammatica, con regole semantiche alla fine delle produzioni, che può essere implementata in Yacc. L'introduzione dei marcatori può rendere possibile valutare gli attributi durante l'analisi LR, anche se la grammatica è L-attribuita. Un altro modo per passare da una grammatica L-attribuita ad una grammatica con solo regole sintetizzate (S-attribuita) è di cambiare la grammatica. Ad esempio:



La visita dell'albero deve fare scendere il tipo nell'albero (visita depth-first). Si può riscrivere la grammatica in modo tale che il "tipo" sia in fondo all'albero, il problema sarà quindi di fare salire il tipo nell'albero, cioè si avrà bisogno di una visita bottom-up:



Riassumendo:



ANALISI LESSICALE:

- ✓ Diagramma di transizione
- ✓ Lex - automatico

ANALISI SINTATTICA:

- ✓ Top-down: diagramma di transizione
- ✓ Bottom-up: Yacc – automatico.

Le grammatiche ad attributi permettono di fare traduzioni guidate dalla sintassi, cioè man mano che si riconosce qualcosa possiamo costruire ciò che vogliamo. Ad esempio: albero a puntatori; immettere i tipi nella tabella dei simboli. Le grammatiche ad attributi sono un mezzo (come lo sono gli automi finiti e gli automi push-down) per aiutare la costruzione di un compilatore.

Yacc supporta l'implementatore di grammatiche attribuite (per la generazione di codice C).

LEZIONE 18 – ANALISI SEMANTICA (5 PARTE)

13/05/2009

L'analisi semantica fa tutto ciò che non è stato fatto dall'analisi sintattica; a differenza dell'analisi sintattica, ha memoria di ciò che ha visto. Dato un programma possiamo effettuare:

1. **Static checking;**
2. **Dynamic checking;**

Lo static checking viene effettuato in fase di compilazione. Controlla che il programma sorgente segua le convenzioni sintattiche e semantiche del linguaggio di programmazione. Tale controllo è fatto sul testo del programma. Il dynamic checking è effettuato in fase di esecuzione. Controlla l'esecuzione del programma; si manda il programma in esecuzione. Ad esempio con l'istruzione 5/i:

- Controllo statico: non rilevo errori;
- Controllo dinamico: si controlla se $i=0$; in questo caso si rileva un errore.

Noi ci occuperemo solo di static checking poiché quello dinamico non riguarda la fase di compilazione. I possibili controlli statici sono:

- **Controlli di tipo (type checks);**
- **Controllo di flusso;**
- **Controllo di unicità;**
- **Controllo sui nomi.**

Anche l'analizzatore lessicale e l'analizzatore sintattico sono controlli statici (fatti sul testo). Quelli elencati sopra sono quelli non fatti nelle fasi precedenti poiché non sono controllabili con costrutti context-free; ve ne sono anche altri. Vediamo degli esempi:

- **controllo di unicità di una variabile:** quando si inserisce il tipo nella tabella dei simboli si controlla prima se tale campo è stato riempito (se è stata dichiarata due volte la stessa variabile);
- **controllo di flusso:** in C l'istruzione break può essere usata solo all'interno di cicli o di un case, cioè solo in particolari contesti; se è utilizzato al di fuori di tali contesti si ha un errore (tale errore è rilevato dall'analisi sintattica);
- **controllo sui nomi:** in ADA se si usa il costrutto package bisogna assegnarli un nome; ad esempio package "stack" end "stack". Per tale controllo si può usare la tabella dei simboli.

Noi ci occuperemo del type checking.

TYPE CHECKING

Utilizzeremo grammatiche S-attribuite per costruire un type checker (controllore di tipi). Il type checker si posiziona logicamente tra la fase di analisi sintattica e quella di generazione del codice intermedio. Prima di arrivare al generatore di codice intermedio tutti i controlli statici sul testo del programma devono essere stati fatti: il programma deve essere corretto staticamente. Vediamo alcuni esempi di errori di tipi:

- in pascal la funzione mod è così definita: $(\text{int}, \text{int}) \rightarrow \text{int}$. L'istruzione $5.4 \bmod 3$ produrrebbe un mismatch di tipo.
- Se p è un puntatore, l'istruzione $p=5$ produce un errore di tipo.
- Se f è una funzione, l'uso di $f[i]$ crea un errore. Ad esempio se dichiariamo $f(a1,a2)$ e poi usiamo $f(a1,a2,a3,a4)$ vi sarà un errore.

Questo tipo di errori non è rilevato dall'analisi sintattica poiché sono costrutti corretti del linguaggio. Per poter rilevare l'errore dovrebbe avere memoria del corpo delle dichiarazioni, ma, in tale caso, non sarebbe un'analisi context-free. **L'analisi semantica fa ciò che l'analisi sintattica non riesce a fare.**

Int f(x1,x2) ←dichiarazione di funzione

...

$F(a_1, a_2, a_3)$ ← chiamata della funzione.

Le due istruzioni sono entrambe sintatticamente corrette (fanno parte dei costrutti del linguaggio). L'analisi sintattica non è in grado di confrontare le funzioni; è l'analisi semantica che quando incontra la dichiarazione arricchisce la tabella dei simboli col numero di argomenti e il tipo degli argomenti della funzione (scorrendo l'albero sintattico), cosicché quando si incontra una chiamata alla funzione si controlla (scorrendo l'albero di f) se è corretta.

L'analisi semantica ripercorre tutto il programma e aggiunge informazioni di tipo nella tabella dei simboli; ha la memoria che non aveva l'analisi sintattica. Sfruttando tali informazioni controlla che vi siano errori di tipo. Per arricchire la tabella dei simboli si usano le grammatiche ad attributi (S-attribuite).

Tale arricchimento può essere implementato in Yacc (facendo contemporaneamente più fasi). Se abbiamo $A[i]$, la verifica statica controlla che A sia un array e i un intero; la verifica dinamica controllerà che i valori assunti da i siano corretti. È fondamentale che una variabile sia utilizzata coerentemente a come è stata dichiarata, poiché la rappresentazione in memoria dei vari tipi è diversa; quindi l'interpretazione della memoria dipende dal tipo di dato che si sta leggendo.

Ad esempio considerando:

short i;	}	Occupano un byte di memoria, però l'interpretazione sarà diversa a seconda del significato da attribuirgli; short o char.
char a;		

Quindi il controllo dei tipi è fondamentale per l'interpretazione della memoria. Ad esempio per la maggior parte dei linguaggi si rileva un errore quando si vuole assegnare un real ad un intero, poiché porta una perdita di informazione: $4 \rightarrow 4.40$. Si è costruita un'algebra dei tipi: il tipo di un costrutto è detto "espressione di tipo". In termini informali un'espressione di tipo è un tipo base o è ottenuta applicando un operatore chiamato "costruttore di tipo" ad altre espressioni di tipo. L'insieme dei tipi base e dei tipi costruiti dipende dal linguaggio che deve essere controllato. Vediamo la definizione ricorsiva formale:

1. Un **"tipo base"** è un'espressione di tipo. Alcuni tipi base sono: integer, real, char ma fondamentalmente dipendono dal linguaggio che si vuole controllare.
"void" indica nessun tipo ma serve per controllare le istruzioni. Ad esempio:
 $a=1$; restituisce un tipo void (indicando che tutto è andato bene);
 $5+4$; restituisce un tipo integer che deriva dalla somma di due tipi integer.
"type error" è un tipo fittizio che ci servirà per segnalare errori.
2. Un **"nome di tipo"** è un'espressione di tipo. Ad esempio in Pascal si può fare:
 $\text{type } x = \text{int}$ e x sarà il nome di int. In C si può usare typedef allo stesso scopo.
3. Applicando costruttori di tipo ad espressioni di tipo, si hanno espressioni di tipo.
4. Le espressioni possono contenere variabili i cui valori sono espressioni di tipo.

Vediamo quali sono i costruttori di tipo:

- **Array:** se T è un'espressione di tipo, allora $\text{array}(I, T)$ è un'espressione di tipo che denota il tipo di un array con elementi di tipo T ed insieme indice I . spesso I è un range di interi (un numero). Con un'array abbiamo una serie di informazioni importanti: dimensione, tipo, indici. Ad esempio data la dichiarazione $\text{var } A: \text{array}[1 \dots 10] \text{ of integer}$; si ha che il tipo della variabile A è l'espressione di tipo $\text{array}(1 \dots 10, \text{integer})$.

Ad ogni variabile del programma assoceremo un'espressione tipo, la quale andrà a riempire il campo type corrispondente nella tabella dei simboli.

- **Prodotto cartesiano:** se T_1 e T_2 sono espressioni di tipo allora il loro prodotto cartesiano $T_1 \times T_2$ è un'espressione di tipo.

- **Record:** la differenza tra un record ed un prodotto cartesiano è che i campi di un record hanno un nome. Ogni campo di un record è il prodotto tra il nome del campo ed il tipo. Il record è un prodotto di campi. Ad esempio: *type row=record*

Addr:integer

Lex:array[1..15] of char

End;

Var table: array[1..10] of row.

Nella tabella dei simboli riempiamo il campo type della variabile table con l'espressione di tipo: *array(1...10, record(addr x integer)x(lex x array(1...15,char)))* oppure *array(1...10, row)*.

1 campo
2 campo
nome di tipo

- **Puntatore:** se T è un'espressione di tipo, allora *pointer(T)* è un'espressione di tipo che denota il tipo puntatore ad un oggetto di tipo T. Ad esempio: *var p: ↑row* nella tabella dei simboli si avrà:

Nome		Tipo
p		Pointer (row)

- **Funzione:** una funzione mappa gli elementi di un insieme (dominio) in un altro insieme (range). Ad esempio: l'espressione di tipo da associare alla funzione mod è *int x int → int*. Ancora un altro esempio: se abbiamo la dichiarazione *var function f(a, b:char): ↑integer*; l'espressione di tipo corrispondente ad f, da mettere nella tabella dei simboli è: *char x char → pointer(integer)*. Costruiremo le espressioni di tipo con una traduzione guidata dalla sintassi, poi lo si memorizzerà nella tabella dei simboli. Se poi nel corpo del programma troveremo l'istruzione: *f(1)+1* il type checker costruisce l'espressione di tipo per *f(1)*: *integer → integer* la confronta con l'espressione di tipo per f memorizzata nella tabella dei simboli; le espressioni sono diverse quindi dà un segnale di errore: type mismatch.

Con tale algebra dei tipi, una qualsiasi dichiarazione può essere codificata in un'espressione di tipo. Quindi il compito del type checker sarà:

1. Scorrere la parte dichiarativa, per ogni simbolo costruire l'espressione di tipo e metterlo nella tabella dei simboli.
2. Scorrere il corpo del programma, per ogni simbolo costruire l'espressione di tipo e confrontarla con quella memorizzata nella tabella dei simboli:
 - a. Se sono uguali, lo si è utilizzato correttamente;
 - b. Se sono diverse, vi è un errore nell'utilizzo; type mismatch.

Vediamo ad esempio la descrizione di un type checker:

Consideriamo la grammatica che genera programmi rappresentati dal simbolo non terminale P, consistenti di una sequenza di dichiarazioni D seguite da una singola istruzione S.

PARTE DICHIARATIVA: dichiarazioni ad ogni produzione associamo l'azione da svolgere:

P→D, S

D→D,D

D→id:T

T→char

T→integer

T→array[num]of T1

T→↑T1

T→T1→T2

addtype(id.entry, T.type)

T.type=char

T.type=integer

T.type=array(1...num.val, T1.type)

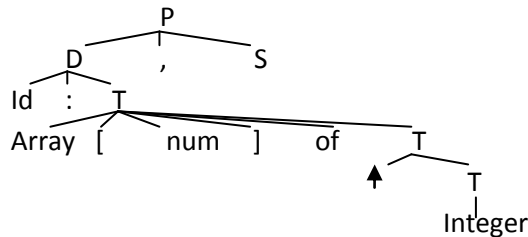
T.type=pointer(T1.type)

T.type=T1.type→T2.type

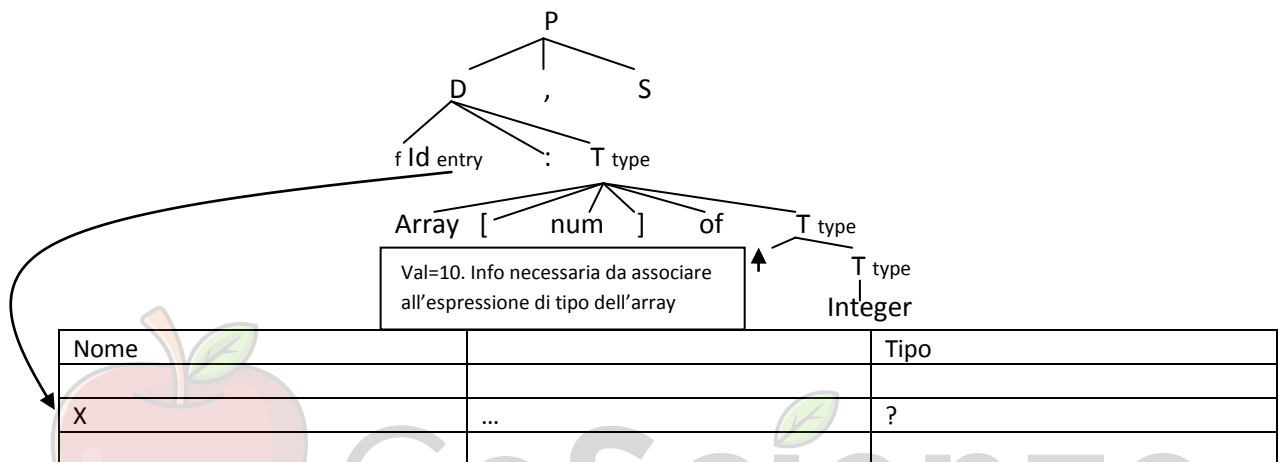
Il valore del token num sarà nell'attributo num.val. Con l'aggiunta delle azioni alla grammatica si può scorrere tutta la parte dichiarativa del programma ed inserire, per ogni variabile incontrata, l'espressione di tipo (che ne definisce il tipo) nella tabella dei simboli. Ad esempio se incontriamo la dichiarazione:

var x: array[1...10] of integer il type checker si comporterà come segue:

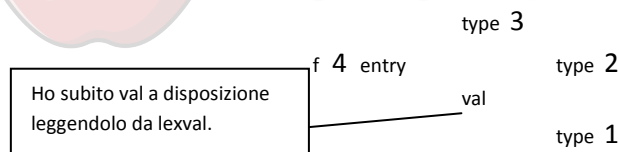
1. Costruisce l'albero di derivazione:



2. Annotiamo l'albero con gli attributi:



3. La grammatica è s-attribuita, quindi senza fare un ordine topologico, sappiamo che possiamo valutare gli attributi bottom-up:



type=integer

type=pointer(integer)

type=array(1...10, pointer(integer))

f=addtype(id.entry, array(1...10, pointer(integer)))

L'esecuzione di addtype riempie il campo tipo della tabella dei simboli:

Nome	...	Tipo
X	...	array(1...10, pointer(integer))

Così per ogni variabile si ha l'espressione di tipo nella tabella dei simboli. Se ad esempio ho $y=x+3$ ed x è un puntatore (come nell'esempio) questa istruzione è valida o meno? Se è valida è permesso lo spostamento del puntatore e il tipo di y è lo stesso di quello di x , o meglio, del puntatore di x .

Data la grammatica

$D \rightarrow T \text{ id } ; D \mid \epsilon$
 $T \rightarrow B C \mid \text{record } \{ 't' D '}'$
 $B \rightarrow \text{int} \mid \text{float}$
 $C \rightarrow \epsilon \mid [\text{num}] C$

Il frammento della grammatica di cui sopra, che tratta i tipi base e gli array, è stata usata per illustrare gli attributi ereditati. La differenza in questa sezione è che adesso consideriamo lo stile di memorizzazione

nonché i tipi. Il non-terminale D genera una sequenza di dichiarazioni. Il non-terminale T genera i tipi base, array o record di tipi. Il non-terminale B genera uno dei tipi base `int` e `float`. Il non-terminale C, per “componente”, genera stringhe di zero o più interi, ogni intero è racchiuso tra parentesi quadre. Un tipo array consiste di un tipo base specificato da B, seguito da componenti di array specificati dal non-terminale C. Un tipo record (la seconda produzione di T) è una sequenza di dichiarazioni per i campi di un record, tutti racchiusi tra parentesi graffe.

Per un nome di tipo, possiamo determinare la quantità di memoria che sarà necessaria per lo stesso a tempo di esecuzione. A tempo di compilazione, possiamo usare queste quantità per assegnare ogni nome ad un indirizzo relativo. Il tipo e l'indirizzo relativo sono salvati in una entry della symbol table per quel nome. Dati di varia lunghezza, come le stringhe, o dati le cui dimensioni non sono determinabili fino al tempo di esecuzione, come array dinamici, sono gestiti riservando una nota quantità di memoria fissata per un puntatore al dato. Supponiamo che la memoria è divisa in blocchi di byte contigui, dove un byte è la più piccola unità di indirizzamento della memoria. Tipicamente, un byte è otto bits e una certa quantità di numeri di byte formano una parola macchina. Oggetti multibyte sono memorizzati in byte consecutivi e viene dato l'indirizzo del primo byte.

La taglia di un tipo è il numero di unità di memoria necessaria per gli oggetti di quel tipo. Un tipo base, come un carattere, intero o float richiede un intero numero di byte. Per un accesso facile, la memoria per gli aggregati come gli array e le classi è allocata in un blocco contiguo di bytes. Lo schema di traduzione (SDT) nella figura che segue calcola i tipi e le loro taglie per tipi base e per array; i tipi record saranno discussi più avanti. La SDT usa attributi sintetizzati `type` e `width` per ogni non terminale e due variabili `t` e `w` per passare l'informazione sul tipo o sulla taglia da un nodo B in un parse tree al nodo per la produzione $C \rightarrow \epsilon$. In una definizione guidata da sintassi, `t` e `w` dovrebbero essere attributi ereditati di C.

Il corpo delle produzioni di T consistono di un non-terminale B, un azione e un non terminale C. L'azione tra B e C setta `t` al valore `B.type` e `w` al valore `B.width`. Se $B \rightarrow \text{int}$ allora `B.type` è settato a `integer` e `B.width` è settato a 4, la taglia di un intero. In maniera simile, se $B \rightarrow \text{float}$ allora `B.type` è `float` e `B.width` è 8, la taglia di un float.

Le produzioni per C determinano se T genera un tipo base o un tipo array. Se $C \rightarrow \epsilon$, allora `t` diventa `C.type` e `w` diventa `C.width`. Altrimenti, C specifica un componente array. L'azione per $C \rightarrow [\text{num}] C_1$ forma `C.type` applicando il costruttore di tipo array all'operando `num.value` e `C1.type`.

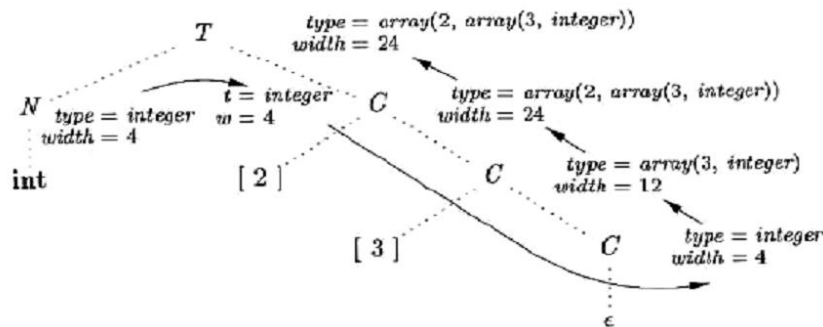
$$\begin{array}{ll} T \rightarrow B & \{ t = B.type; w = B.width; \} \\ C & \\ B \rightarrow \text{int} & \{ B.type = \text{integer}; B.width = 4; \} \\ B \rightarrow \text{float} & \{ B.type = \text{float}; B.width = 8; \} \\ C \rightarrow \epsilon & \{ C.type = t; C.width = w; \} \\ C \rightarrow [\text{num}] C_1 & \{ \text{array}(\text{num.value}, C_1.type); \\ & C.width = \text{num.value} \times C_1.width; \} \end{array}$$

La taglia di un array è ottenuta moltiplicando la taglia di un elemento per il numero di elementi di un array. Se indirizzi di interi consecutivi differiscono di 4, allora il calcolo degli indirizzi per un array di interi includerà le moltiplicazioni per 4. Tali moltiplicazioni forniscono opportune ottimizzazioni, pertanto è utili per il front-end esplicitarle.

Esempio

Il parse tree per il tipo `int [2][3]` è mostrato da linee punteggiate nella figura che segue. Le linee piene mostrano come i tipi e le taglie sono passate da B, sotto la catena di C attraverso le variabili `t` e `w`, e poi tornare indietro nella catena come attributi sintetizzati `type` e `width`. Le variabili `t` e `w` sono assegnate con i valori di `B.type` e `B.width` rispettivamente, prima che il sottoalbero con i nodi C sia esaminato. Il valore di `t` e

w sono usati al nodo per $C \rightarrow \epsilon$ per iniziare la valutazione degli attributi sintetizzati sopra la catena di nodi C.



CORPO DEL PROGRAMMA

$S \rightarrow id = E$

$S \rightarrow \text{if } BE \text{ then } S1$

$S \rightarrow \text{while } BE \text{ do } S1$

$S \rightarrow S1, S2$

Sviluppiamo la E:

$E \rightarrow \text{literal}$

$E \rightarrow \text{num}$

$E \rightarrow id$

$E \rightarrow E1 \text{ mod } E2$

$E \rightarrow E1[E2]$

$E \rightarrow E1 \uparrow$

$E \rightarrow E1(E2)$

E1=id
E2=num

Puntatore (la freccia non fa niente che la freccia sta dopo, dipende dal linguaggio).

Poi si può sviluppare anche BE. A tali produzioni bisogna associare azioni che oltre a costruire le espressioni di tipo per ogni simbolo, bisogna verificare che le espressioni ottenute coincidono con quelle memorizzate nella tabella dei simboli:

$E \rightarrow \text{literal}$

E.type=char

$E \rightarrow \text{num}$

E.type=integer

$E \rightarrow id$

E.type=lookup(id.entry) (la lookup è una funzione che preleva il tipo, già presente nella tabella, puntato da "a";

$E \rightarrow E1 \text{ mod } E2$

E.type=(if E1.type=integer and E2.type=integer then integer else type_error)

Viene effettuato un controllo sui tipi degli argomenti.

$E \rightarrow E1[E2]$

E.type=(if E2.type=integer and E1.type=array(S,t) then t else type_error)

$E \rightarrow E1 \uparrow$

E.type=(if E1.type=pointer(t) then t else type_error)

$E \rightarrow E1(E2)$

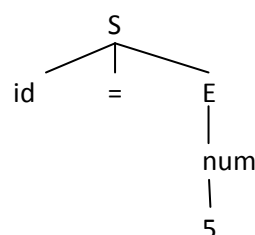
E.type=(if E2.type=S and E1.type=S \rightarrow t then t else type_error)

$S \rightarrow id=E$

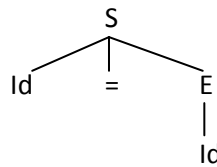
S.type=(if lookup(id.entry)=E.type then void else type_error)

Le istruzioni hanno come tipo void. Vediamo un paio di esempi:

x=5



X=y



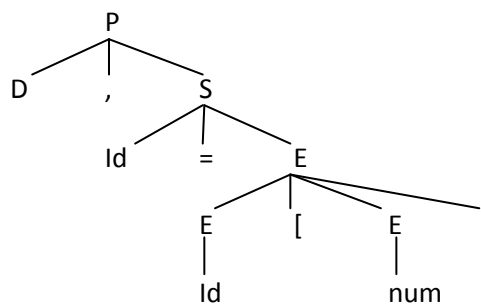
Supponiamo ora che oltre alla dichiarazione vista in precedenza, `x=array[1...10] of integer`, vi sia anche: `y=▲int`; dopo aver considerato la parte dichiarativa della tabella dei simboli si avrà:

TABELLA DEI SIMBOLI

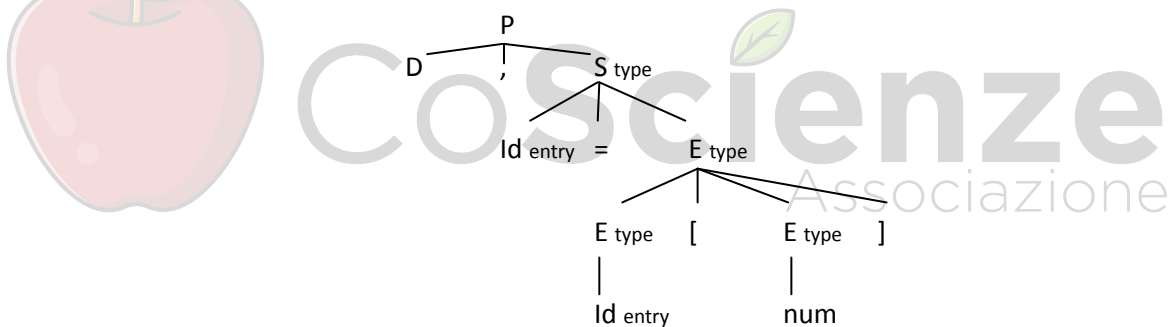
Nomi		Tipi
X	...	Array(1...10,pointer(integer))
Y	...	Pointer(integer)

Supponiamo che nel corpo del programma si incontri l'istruzione `y=x[z]`, vediamo come si comporta il type checker:

1. Costruisce l'albero:

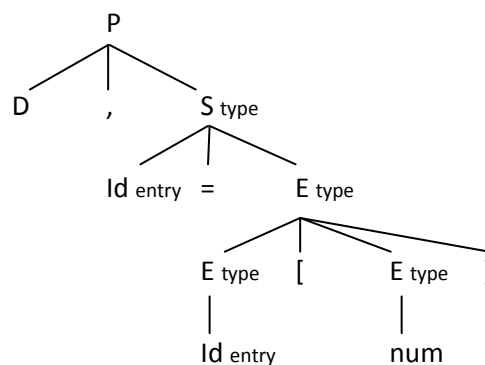


2. Annotiamo l'albero con gli attributi:



Nomi		Tipi
X	...	Array(1...10,pointer(integer))
Y	...	Pointer(integer)

3. La grammatica è s-attribuita, quindi per valutare gli attributi basta fare una visita bottom-up:



Type=array(1...10, pointer(integer))

Type=integer

Type=pointer(integer)

Type=void

E→num

E→E1[E2]

S→id=E

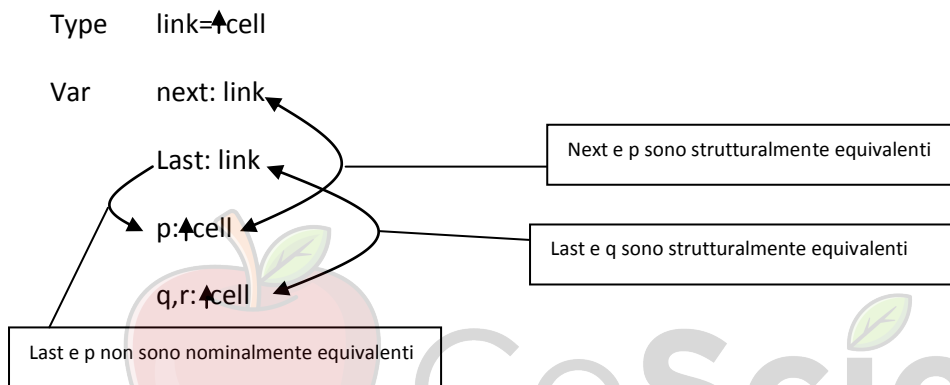
Non si è mai incontrato un type-error, quindi l'istruzione è corretta. Se avessimo incontrato un type-error, significava la presenza di un errore quindi bisognava dare un messaggio di mismatch di tipo (eventualmente indicando riga e colonna dell'errore).

EQUIVALENZA DELLE ESPRESSIONI DI TIPO

Vi sono due nozioni di equivalenza tra espressioni di tipo, in quanto nelle espressioni di tipo possono comparire anche nomi di tipi;

- ✓ Due espressioni di tipo sono equivalenti dal punto di vista **strutturale** se sostituendo a tutti i nomi i relativi tipi si ottengono due espressioni uguali (C, pascal)
- ✓ Due espressioni di tipo sono equivalenti dal punto di vista **nominale** se e solo se sono identiche. In questo caso si considerano anche espressioni di tipo contenenti nomi di tipo, e nomi diversi sono considerati come tipi diversi (caratteristica dei linguaggi fortemente tipati). L'equivalenza nominale è più facile da implementare.

Vediamo un esempio



link è il nome del tipo ↑cell.

- ✓ Secondo l'equivalenza strutturale tutte e 5 le variabili sono equivalenti, poiché hanno tutte la stessa espressione di tipo: pointer(cell)
- ✓ Secondo l'equivalenza nominale si ha che:
 - Next e lost sono equivalenti a link
 - P, q ed r sono equivalenti a pointer(cell)
 - Next e lost non sono equivalenti a p, q ed r.

Alcuni linguaggi controllano l'equivalenza strutturale mentre altri quella nominale.

Vi è un algoritmo ricorsivo che controlla l'equivalenza strutturale:

funzione: Sequiv(S,t) prende in input due tipi S e t e decide se tali tipi sono strutturalmente equivalenti

...

If S=array(S1,t1) and t=array(S2,t2) then Sequiv(S1,S2) and Sequiv(t1,t2).

S e t devono essere espressioni di tipo senza nomi, altrimenti non si controlla l'equivalenza strutturale. Si può creare il problema di chiamate ricorsive come ad esempio:

```
type    link=↑cell
       cell=record
           info=integer
           next=link
       end.
```

Si è utilizzato cell prima di dichiararlo. L'algoritmo precedente non va bene andrebbe in un ciclo infinito; per evitare tale problema (in C e in pascal) si usa un altro algoritmo per i campi del record.

CONVERSIONE DEI TIPI

L'analisi semantica può fare conversioni di tipo implicite. Consideriamo l'espressione $x+1$ con x reale ed i intero. Poiché la rappresentazione in memoria di questi due tipi è diversa, e soprattutto poiché le istruzioni macchina sono diverse per gli interi e per i reali (ciò avviene per più operatori, vi è un $+$ per i reali, ed un $+$ per gli interi), il compilatore dovrebbe prima convertire l'intero in reale e poi fare l'addizione.

La conversione è detta **implicita** se è fatta automaticamente dal compilatore:

$5+4.3$

$5.0+4.3$. non vi è perdita di informazioni.

La conversione è detta **esplicita** se è il programmatore che deve scrivere qualcosa per dare luogo alla conversione. Ad esempio:

$\text{real}(5) + 4.3$

Vediamo infine come l'analisi semantica fa la conversione di tipo implicita usando le grammatiche attribuite.

Produzioni	Regole semantiche
$E \rightarrow \text{num}$	$E.\text{type} = \text{integer}$
$E \rightarrow \text{num}.\text{num}$	$E.\text{type} = \text{real}$
$E \rightarrow \text{id}$	$E.\text{type} = \text{lookup}(\text{id.entry})$
$E \rightarrow E1 \text{ op } E2$	$E.\text{type} =$ if $E1.\text{type} = \text{integer}$ and $E2.\text{type} = \text{integer}$ then integer else if $E1.\text{type} = \text{integer}$ and $E2.\text{type} = \text{real}$ then real else if $E1.\text{type} = \text{real}$ and $E2.\text{type} = \text{integer}$ then real else if $E1.\text{type} = \text{real}$ and $E2.\text{type} = \text{real}$ then real else type_error)

Si capisce che per fare una conversione di tipo vi è una complessità di tempo, quindi è più efficiente scrivere direttamente il reale 5.0 . La conversione fatta in esecuzione prende più tempo ed è meno efficiente per i sistemi critici come quelli real-time.

LEZIONE 19 – GENERAZIONE DEL CODICE INTERMEDIO (1 PARTE)

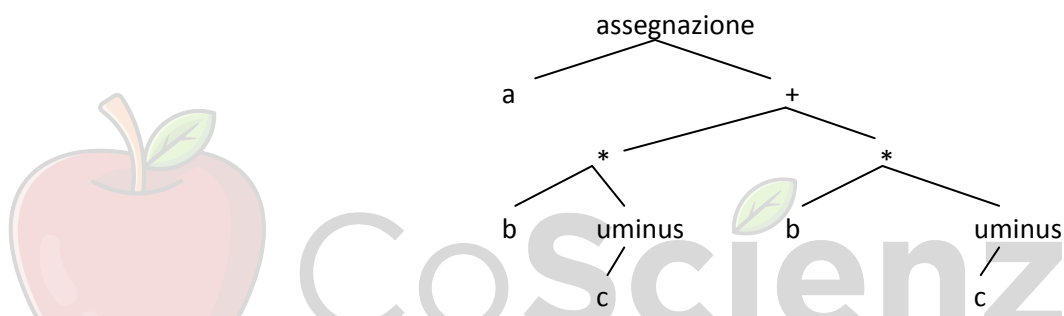
20/05/2009



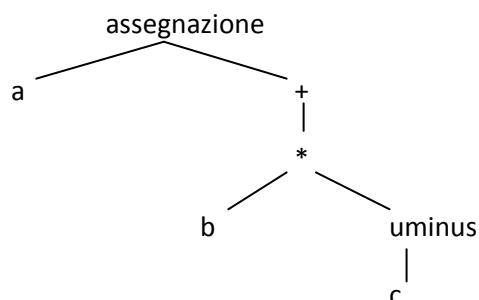
Vedremo che tipi di rappresentazione intermedia vi sono e come ottenerli. **Vi sono compilatori che saltano la rappresentazione intermedia e generano direttamente il codice macchina. È preferibile avere una rappresentazione intermedia poiché aumenta la portabilità del compilatore, in quanto la rappresentazione intermedia può essere indipendente dalla macchina finale, e permette l'ottimizzazione del codice.**

Ecco alcune possibili rappresentazioni intermedie: consideriamo l'espressione $a := b * -c + c * -c$

1. **ALBERO SINTATTICO:** non considera i non terminali, quindi è indipendente dalla grammatica usata (mentre il parser-tree dipende dalla grammatica usata). L'albero sintattico della frase data è:



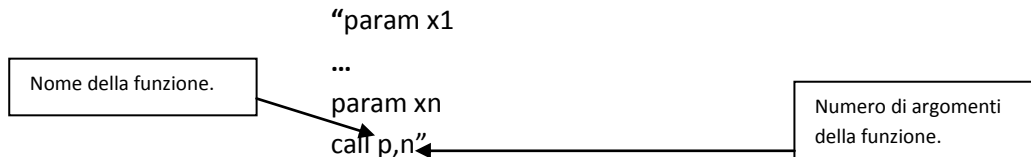
Tale albero può essere ottimizzato costruendo un dag (grafo diretto aciclico), usando sempre grammatiche ad attributi: un dag sintattico dà le stesse informazioni di un albero sintattico, ma in modo più compatto:



2. **NOTAZIONE POST-FISSA:** è una linearizzazione di un albero sintattico. Si ottiene facendo una visita post-order dell'albero sintattico (un nodo appare subito dopo i suoi figli). Per la nostra frase avremo questa notazione post-fissa: $abcuminus*bcuminus*+assegnazione$.
Note le priorità degli operatori, tale rappresentazione è totalmente equivalente all'albero sintattico; si può passare facilmente dall'una all'altra. Abbiamo visto che per ottenere traduzioni in tali forme intermedie si usano grammatiche ad attributi. Abbiamo già visto come ottenere un albero sintattico ed una notazione post-fissa, facendo uso di grammatiche ad attributi.
3. **CODICE A TRE INDIRIZZI:** è una sequenza di istruzioni della forma generale $x := y \text{ op } z$, è chiamato a tre indirizzi poiché può avere al più tre nomi di variabili (costanti, nomi, puntatori) ed un operatore (binario o unario). Tale forma intermedia è più simile al codice macchina che vogliamo generare;

quindi usando tale notazione, per il codice intermedio, la traduzione del codice intermedio in codice macchina sarà abbastanza rapida. Le istruzioni presenti in un codice a tre indirizzi possono avere etichette e possono gestire il controllo del flusso; quelle più usate sono:

- Assegnamenti** " $x:=y \text{ op } z$ " dove op è un operatore binario.
- Assegnamenti** " $x:= \text{op } z$ " dove op è un operatore unario.
- Copie** " $x:=y$ ".
- Salto incondizionato** "goto L" dove L sarà l'etichetta (serve ad identificare l'istruzione) dell'istruzione a tre indirizzi che sarà eseguita successivamente.
- Salto condizionato** (istruzioni di controllo) "if (x rel op y) goto L"
- Chiamata a funzione** $p(x_1, x_2, \dots, x_n)$ in codice a tre indirizzi diventa:

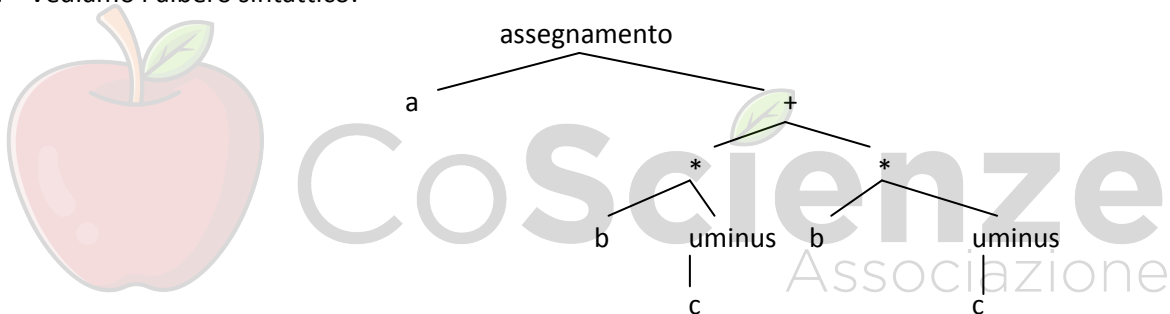


- Istruzioni indicizzate** (array) rappresentate come: " $x:=y[i]$ " e " $x[i]:=y$ "
- Istruzioni di indirizzamento** (puntatori) rappresentati come: coppia indirizzi, valore.

Il codice a tre indirizzi è un vero e proprio linguaggio.

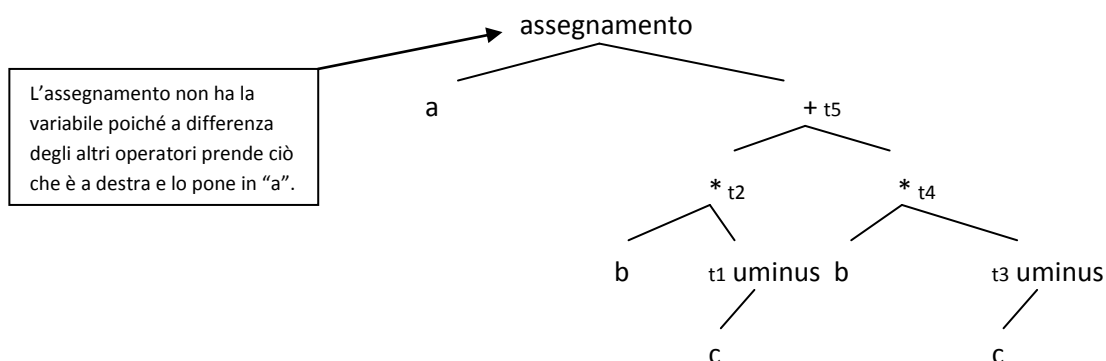
Vediamo quale sarà il codice a tre indirizzi generato per la nostra espressione $a:=b*-c+b*-c$.

- Vediamo l'albero sintattico:



ad ogni nodo interno si assegna una variabile temporanea:

- Albero annotato, dobbiamo memorizzare l'ordine di applicazione degli operatori:



La grammatica è S-attribuita (gli attributi sono tutti sintetizzati). Si evita di costruire il grafo delle dipendenze. Si scrivono direttamente le regole facendo la visita bottom-up.

- Facendo la visita bottom up si ottiene una prima sequenza di istruzioni a tre indirizzi:

$t1:=-c$ (foglia)

$t3:=-c$ (foglia)

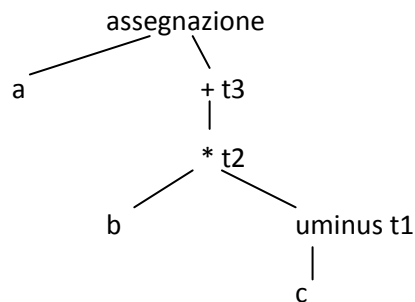
$t2:=b*t1$ → calcolabile perché già si è in possesso di $t1$.

$t4:=b*t3$

t5:=t2+t4

a:=t5

Naturalmente introducendo le variabili temporanee aumenta la memoria necessaria. Lo stesso procedimento lo si può fare sul dag sintattico:



ottenendo:

t1:=-c (foglia)

t2:=b*t1 → che è un'ottimizzazione del codice precedente

t3:=t2+t2

a:=t3

Le tre rappresentazioni viste (l'albero sintattico, la notazione post-fissa, il codice a tre indirizzi) sono logicamente equivalenti ma possono avere diversa rappresentazione in memoria. Vediamo una grammatica S-attribuita per generare tale codice a tre indirizzi (lo si potrà costruire durante l'analisi sintattica). La grammatica che genera le istruzioni di assegnazione è:

$S \rightarrow id := E$

$E \rightarrow E1 + E2$

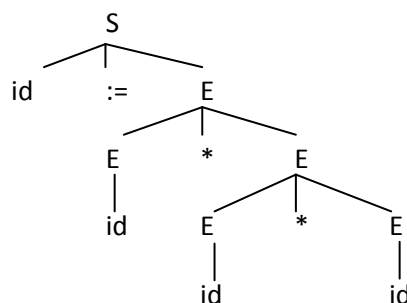
$E \rightarrow E1 * E2$

$E \rightarrow -E1$

$E \rightarrow id$

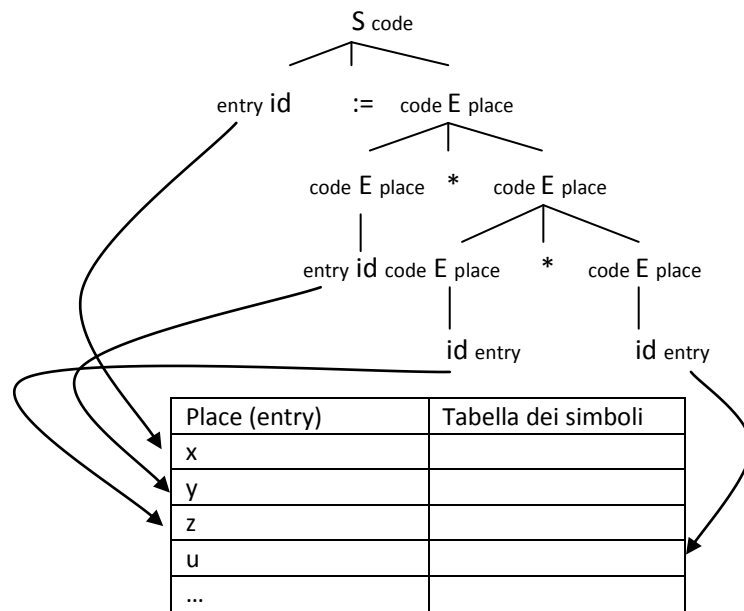
Vediamo quali regole associare a tale grammatica per generare il codice a tre indirizzi corrispondente ad un'istruzione di assegnamento; per esempio $x:=y*z*u$:

- a. Come prima cosa costruiamo l'albero di derivazione per la frase da tradurre:



Vogliamo costruire le regole in modo tale che alla fine nell'attributo "code" di S vi sia la sequenza di istruzioni a tre indirizzi che traducono l'assegnamento dato.

- b. Annotiamo l'albero con gli attributi. **Ogni non terminale avrà un attributo "code" che dovrà contenere il codice a tre indirizzi che traduce tutto il sottoalbero radicato in quel nodo; inoltre ad ogni nodo interno si associa una variabile temporanea il cui nome è posto nell'attributo "place" (indicante il valore dell'espressione in quel nodo):**



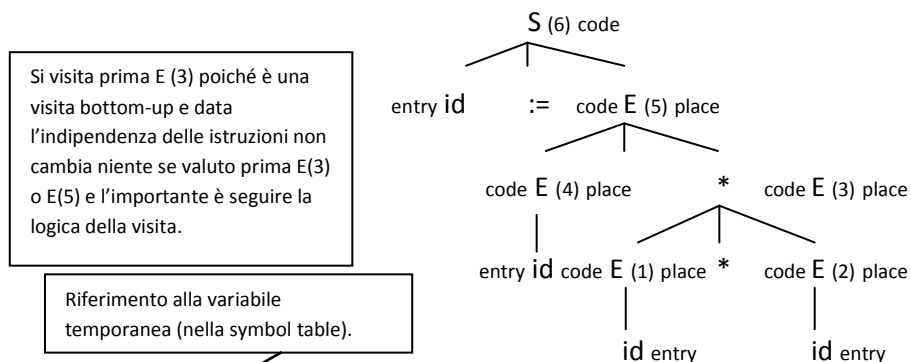
c. Visti gli attributi ed il loro significato, costruiamo le regole da associare alle produzioni:

- 1) $S \rightarrow id := E$ $S.code := E.code \parallel gen(id.place := E.place)$
- 2) $E \rightarrow E_1 * E_2$ $E.place := newtemp$
 $E.code := E_1.code \parallel E_2.code \parallel$
 $gen(E.place := E_1.place * E_2.place)$
- 3) $E \rightarrow -E_1$ $E.place := newtemp;$
 $E.code := E_1.code \parallel gen(E.place := '-' E_1.place)$
- 4) $E \rightarrow E_1 + E_2$ $E.place := newtemp$
 $E.code := E_1.code \parallel E_2.code \parallel$
 $gen(E.place := E_1.place + E_2.place)$
- 5) $E \rightarrow id$ $E.place := id.place; E.code := '';$

L'operatore \parallel sta per la concatenazione.

La funzione $gen()$ genera l'istruzione racchiusa nelle parentesi ($id.place \dots$).

d. Valutiamo gli attributi facendo una visita bottom-up dell'albero:



- 1) Place1 = z
 - 2) Place2 = u
 - 3) Place3 = t1
 - 4) Place4 = y
 - 5) Place5 = t2
 - 6)
- code1 = ''
code2 = ''
code3 = t1 := z * u
code4 = ''
code5 = t1 := z * u
t2 := y * t1
code6 = t1 := z * u
t2 := y * t1
x := t2

ES. Vediamo quali regole associare alla produzione $S \rightarrow \text{while } E \text{ do } S1$ per tradurre la frase in codice a tre indirizzi. Verrà generato il seguente codice:

```

S.begin      :      E.code
                If      E.place=0      goto  S.after
                S1.code
                goto S.begin

```

S è lo statement e tiene traccia dell'inizio e della fine dello statement.
E.place=0 individua "se E.place è falso"
S.after è una label.

```

S.after      :

```

Quindi le regole da associare alla produzione sono:

```

S.begin:= newlabel;
S.after:= newlabel;
S.code:= gen(S.begin ':') || E.code || gen('if' E.place '=' '0' 'goto' S.after) || S1.code || gen('goto'
S.begin) || gen(S.after ':');

```

Sapendo cosa vogliamo genere non è difficile scrivere le regole semantiche corrette.

IMPLEMENTAZIONE DEL CODICE A TRE INDIRIZZI

Un'istruzione a tre indirizzi è una forma astratta di codice intermedio. Vi sono tre possibili tecniche per implementarla. Vediamo le tre tecniche applicate su questo esempio (sequenza di istruzioni a tre indirizzi):

t1:=uminus c

t2:=b*t1

t3:=t2*b

a:=t3;

1. **QUADRUPLE:** sono array di record con quattro campi:

- a. Op: per l'operatore
- b. Arg1: per il primo argomento
- c. Arg2: per il secondo argomento
- d. Risultato: per il risultato dell'operazione;

Nei campi arg1, arg2 e risultato vi saranno dei puntatori alla tabella dei simboli. L'operatore param non ha né arg2, né risultato. I salti hanno la label nel risultato.

	Op	Arg1	Arg2	Risultato
0	Uminus	c		t1
1	*	b	t1	t2
2	*	t2	b	t3
3	:=	t3		a

2. **TRIPLE:** sono array di record con tre campi. Quindi rispetto alla rappresentazione precedente vi è un risparmio di memoria.

- a. Campo op: contiene l'operatore
- b. Campi arg1, arg2: possono avere puntatori sia alla tabella dei simboli che a terne dello stesso array.

	Op	Arg1	Arg2
0	Uminus	c	
1	*	b	0
2	*	1	b
3	:=	a	2

Puntatore alla tripla che genera il valore di tale argomento. Con tale rappresentazione non compaiono le variabili temporanee.

Il problema con tale rappresentazione è che non possiamo spostare l'ordine dei record nell'array altrimenti salterebbero i puntatori alle triple stesse. Tale problema non si avrà nelle quadruple. Nella fase di ottimizzazione può capitare di spostare l'ordine dei record, cioè l'ordine di esecuzione delle istruzioni (per esempio per fare ottimizzazione). Nell'implementazione bisogna considerare tale eventualità, per cui le triple non sono una buona rappresentazione. Vogliamo arrivare ad una rappresentazione che non abbia di tali problemi ed allo stesso tempo porta un risparmio di memoria. Si passa alle triple indirette.

3. **TRIPLE INDIRETTE:** accanto alle triple usiamo una tabella che stabilisce l'ordine di esecuzione.

N. istruzione	Op	Arg1	Arg2
14	Uminus	c	
15	*	V	14
16	*	15	b
17	:=	a	16

L'ordine in questo modo non verrà mai modificato, cosicché si ha sempre un corretto indirizzamento.

Questa è la tabella che determina l'ordine di esecuzione:

ordine di esecuzione	Istruzione
0	14
1	15
2	16
3	17

In questo modo se si vuole cambiare l'ordine di esecuzione delle istruzioni si modifica tale tabella, ma non si toccano mai le triple. Con le triple indirette si risparmia memoria rispetto alle quadruple e si ha la possibilità di cambiare l'ordine di esecuzione delle istruzioni.

Vediamo un esempio su come generare codice a partire da un intero programma:

program

var x:integer;

y: array[1...10] of real;

z:real;

procedure quicksort:

var k,v:integer;

procedure partition:

var i,j:integer;

begin

i:=k+x;

end

begin

...

end

begin

...

End

Livello 2

Livello 1

Livello 0

"K" è a un livello più alto. Come si fa? Ci si dovrà occupare di cercare k nello scope prima di Partition, poi al livello superiore e, eventualmente, ancora ai livelli superiori. Per fare i salti tra i vari livelli si usa uno stack. Ogni funzione nello stack avrà un determinato spazio allocato come vediamo nella pagina successiva.

Partendo dall'analisi lessicale fino ad arrivare alla generazione del codice intermedio avremo questa tabella dei simboli:

TABELLA DEI SIMBOLI (program) T0:

nome	Ampiezza	Offset	Tipo
X	4	0	Integer
Y	80	4	Array(1...10, real)
Z	8	84	Real
Quicksort			Puntatore alla tabella T1

Spazio occupato dalla variabile

Sono indirizzi relativi rilocabili, non assoluti; indirizzi di dove iniziano le variabili in memoria.

Le chiamate a funzioni sono gestite con record di attivazione in uno stack dove gli indirizzi relativi ripartono da 0 (azzeramento dell'offset). Per ogni funzione chiamata creiamo una tabella dei simboli.

TABELLA DEI SIMBOLI (quicksort) T1:

nome	Ampiezza	Offset	Tipo
K	4	0	Integer
V	4	4	Integer
Partition			Puntatore alla tabella T2

TABELLA DEI SIMBOLI (partition) T2:

nome	Ampiezza	Offset	Tipo
I	4	0	Integer
J	4	4	integer

Ogni funzione ha un suo record di attivazione. Lo stack iniziale possiamo vederlo così:

Partition
Quicksort
program

Generiamo ora il codice a tre indirizzi per l'istruzione $i := k + x$ all'interno della funzione partition: il codice a tre indirizzi non si crea per le dichiarazioni, ma solo per il corpo delle funzioni; le dichiarazioni servono per riempire la tabella dei simboli.

Consideriamo l'implementazione del codice a tre indirizzi con quadruple:

op	Arg1	Arg2	Risultato
+	K	x	i

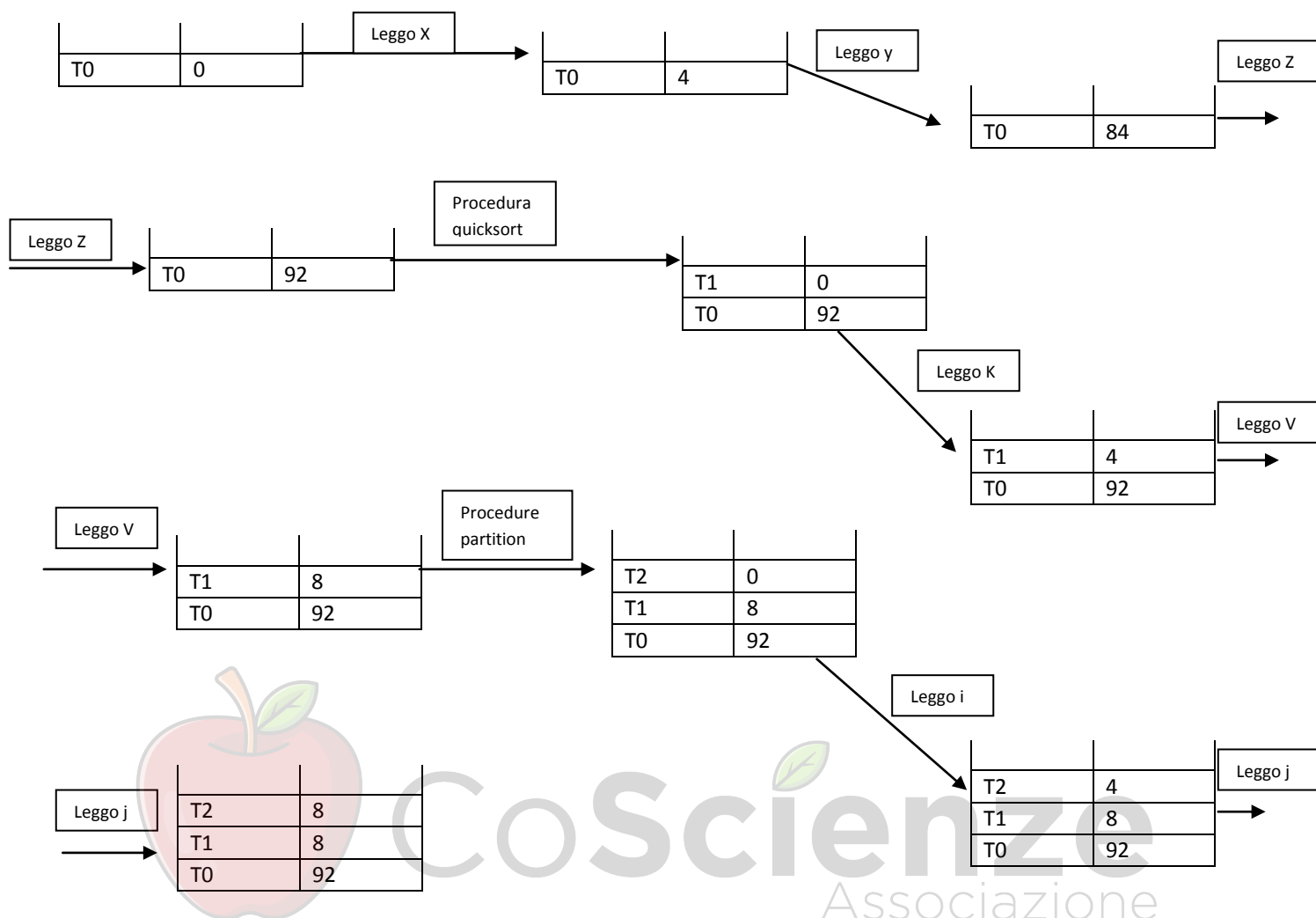
Si è nel record di attivazione di partition, quindi si cerca prima k nella tabella dei simboli di partition T2; non lo si trova quindi si segue il puntatore alla tabella dei simboli precedenti T1 (ogni tabella oltre ad avere un puntatore alla tabella successiva ha anche un puntatore alla tabella precedente) dove si trova il primo operando k; lo stesso avviene per x dove si arriva fino a T0. Per mantenere informazioni circa la tabella e l'offset che si ha in ogni punto del programma si usa uno stack. Quindi la condizione iniziale dello stack sarà:

T0	0

Offset attuale

Puntatore alla tabella dei simboli in esame, su cui lavorare.

Poi si incomincia a scorrere il programma in input; si scorrono tutte le dichiarazioni:



Ogni tabella dei simboli ha anche le seguenti informazioni:

T0:	head	0	92
T1:		1	8
T2:		2	8

Queste info servono per indicare la memoria necessaria per le variabili della funzione relativa alla tabella; è la somma delle ampiezze delle variabili.

Il valore dell'ampiezza totale necessaria ad una funzione è inserita nella tabella dei simboli non appena si fa un pop dallo stack in quanto i valori corretti sono nel campo offset dello stack (sul top) prima di un pop. Per fare tutte queste operazioni si useranno le grammatiche ad attributi. Le regole per le produzioni usano le funzioni:

- **Mktable (previous):** tale funzione è lanciata quando nel programma in input si incontra una funzione. Costruisce una nuova tabella dei simboli che avrà un puntatore alla tabella precedente (previous).
- **Enter (table, name, type, offset):** aggiunge il type e l'offset (prelevato dal top dello stack) alla variabile name nella tabella table.
- **Addwidth (table, width):** verrà eseguita alla fine di ogni funzione, preleva l'offset totale della table dal top dello stack e lo inserisce nella table.

- **Enterproc (table, name, newtable):** tale funzione è lanciata dopo un pop dallo stack, cioè quando finisce una funzione; aggiunge alla nuova tabella in esame il puntatore alla tabella che si sta abbandonando.

Come si inseriscono queste funzioni nella definizione guidata dalla sintassi????

Dichiarazioni:

$P \rightarrow MD$ {addwidth(top(tb|ptr), top(offset)); pop(tb|ptr); pop(offset);}
 $M \rightarrow \epsilon$ {t:= mktable(nil); push(t,tb|ptr); push(0,offset);}
 $D \rightarrow D1; D2$
 $D \rightarrow \text{proc id; ND1; S}$ {t:= top(tb|ptr); addwidth(t,top(offset)); pop(tb|ptr); pop(offset);
 enterproc(top(tb|ptr), id.name,t)}
 $D \rightarrow \text{id:T}$ { enter (top(tb|ptr), id.name, T.type, top(offset)); top(offset):=top(offset)+T.width;}
 $N \rightarrow \epsilon$ {t:=mktable(top(tb|ptr)); push(t,tb|ptr); push(0,offset); }

La forma statica a singolo assegnamento (SSA) è una rappresentazione intermedia che facilita alcune ottimizzazioni del codice. Due aspetti distinti fanno differire SSA dal codice a tre indirizzi. Il primo è che tutti gli assegnamenti in SSA sono variabili con nomi distinti; pertanto il termine singolo assegnamento statico. La figura che segue mostra lo stesso programma intermedio in un codice a tre indirizzi e in una forma a singolo assegnamento statica. Notare che i pedici distinguono ogni variabile nella forma SSA.

$p = a + b$
 $q = p - c$
 $p = q * d$
 $p = e - p$
 $q = p + q$

(a) Codice a tre indirizzi

$p_1 = a + b$
 $q_1 = p_1 - c$
 $p_2 = q_1 * d$
 $p_3 = e - p_2$
 $q_2 = p_3 + q_1$

(b) forma a singolo assegnamento statico

La stessa variabile può essere definita in due differenti control-flow path in un programma. Per esempio, il programma sorgente

```

if (flag)
    x = -1;
else
    x = 1;
y = x * a;
  
```

ha due percorsi di controllo del flusso in cui la variabile x è definita. Se usiamo due nomi differenti per x nella parte vera e nella parte falsa dell'if quale nome dovrebbe poi essere usato nell'assegnazione successiva? Quindi è qui che il secondo aspetto di SSA entra in gioco. SSA usa una notazione convenzionale chiama funzione- ϕ per combinare le due differenti definizioni di x:

```

if (flag)
    x1 = -1;
else
    x2 = 1;
x3 =  $\phi(x1, x2)$ ;
  
```

dove ϕ ha il valore corretto di x.

LEZIONE 20 – GENERAZIONE DEL CODICE INTERMEDIO (2 PARTE)

27/05/2009

Type Checking (controllo dei tipi)

Per eseguire un controllo dei tipi un compilatore deve assegnare un espressione di tipo ad ogni componente di un programma sorgente. Il compilatore deve poi determinare se queste espressioni di tipo sono conformi ad una collezione logica di regole che è detta sistema dei tipi per un linguaggio sorgente.

Il controllo dei tipi ha il potenziale per la cattura degli errori nei programmi. In principio, qualsiasi controllo può essere fatto in maniera dinamica, se il codice target porta il tipo di un elemento insieme con il valore di quell'elemento. Un sistema si dice *sound* se elimina la necessità di controlli dinamici per gli errori di tipo, perché esso consente di determinare staticamente che questi errori non possono verificarsi quando il programma sorgente è in esecuzione. Un implementazione di un linguaggio si dice *strettamente tipata* se un compilatore garantisce che i programmi che accetta eseguiranno senza errori di tipo. I programmi Java sono compilati in un bytecode indipendente dalla macchina che include informazioni di tipo dettagliate riguardo alle operazioni nel bytecode. Il codice importato è sempre controllato prima di essere eseguito per salvarsi da errori dolosi o colposi.

Regole per il controllo dei tipi

Il controllo dei tipi può essere considerato sotto due forme: sintesi o inferenza. La sintesi dei tipi costruisce il tipo di una espressione dal tipo delle sue sottoespressioni. Essa richiede che i nomi siano dichiarati prima che sia usati. Il tipo di $E1 + E2$ è definito in termini di tipi di $E1$ e $E2$. Una regola tipica per la sintesi dei tipi ha la seguente forma:

*if f ha tipo $s \rightarrow t$ e x ha tipo s ,
then l'espressione $f(x)$ ha tipo t*

Qui, f e x denota espressioni, e $s \rightarrow t$ denota una funzione da s a t . Questa regola per le funzioni con un argomento porta alle funzioni con più argomenti. La regola appena mostrata può essere adattata per $E1 + E2$ vedendo essa come funzione $\text{add}(E1, E2)$. L'inferenza di tipi determina il tipo di un costrutto di un linguaggio dal modo in cui esso è usato. Sia null una funzione che testa se una lista è vuota. Allora, dall'uso di $\text{null}(x)$ possiamo dire che x deve essere una lista. Il tipo degli elementi della lista non è noto; tutto quello che noi sappiamo è che x deve essere una lista di elementi di qualche tipo che è attualmente sconosciuto.

Le variabili che rappresentano le espressioni di tipo permettono di dire qualcosa riguardo i tipi non noti. Useremo le lettere greche per qualsiasi variabile di tipo in una espressione di tipo. Una regola tipica per l'inferenza di tipi ha questa forma:

*if $f(x)$ is an expression,
then for some α and β , f has type $\alpha \rightarrow \beta$ and x has type α*

Controllo del flusso

La traduzione di statement come un if-then-else o un ciclo while è legata alla traduzione di espressioni booleane. Nei linguaggi di programmazione, le espressioni booleane sono usate spesso per:

1. alterare il flusso di controllo. Le espressioni booleane sono usate come espressioni condizionali negli statement che alterano il flusso di controllo. Il valore di tali espressioni booleane è implicito in una posizione raggiunta in un programma. Per esempio, in `if (E) S` l'espressione E deve essere vera affinché lo statement S possa essere raggiunto.

2. calcolare valori logici. Una espressione booleana può rappresentare true o false come valore. Una tale espressione può essere valutata in analogia alle espressioni aritmetiche usando istruzioni a tre indirizzi con operatori logici.

L'intento di usare espressioni booleane è determinato dal suo contesto sintattico. Per esempio, una espressione di seguito alla parola chiave `if` è usata per alterare il flusso di controllo, mentre un'espressione sulla parte destra di una istruzione di assegnazione è usata per denotare un valore logico. Tali contesti sintattici possono essere specificati in vari modi: possiamo usare due differenti non terminali, usare attributi ereditati o settare un flag durante il parsing. Alternativamente, possiamo costruire un albero sintattico e invocare differenti procedure per i due differenti usi di espressioni booleane.

Espressioni Booleane

Le espressioni booleane sono composte da un operatore booleano (che denotiamo con `&&`, `||`, e `!` rispettivamente per AND, OR e NOT) applicato ad elementi che sono variabili booleane o espressioni relazionali. Le espressioni relazionali sono della forma `E1 rel E2` dove `E1` e `E2` sono espressioni aritmetiche. In questa sezione, consideriamo le espressioni booleane generate dalla seguente grammatica

$$B \rightarrow B \mid B \mid B \&\& B \mid !B \mid (B) \mid E \text{ rel } E \mid \text{true} \mid \text{false}$$

Usiamo l'attributo `rel.op` per indicare il tipo di operatore relazionale rappresentato da `rel`. Assumiamo che `||` e `&&` sono associative a sinistra e che `||` ha minore precedenza rispetto a `&&` e `!`. Data l'espressione `B1 || B2`, se determiniamo che `B1` è true allora possiamo concludere che tutta l'espressione è true senza che sia necessario valutare `B2`. In maniera simile, se abbiamo l'espressione `B1 && B2` allora se `B1` è false allora tutta l'espressione è false.

La definizione semantica del linguaggio di programmazione determina se tutte le parti di una espressione booleana devono essere valutate o no. Se la definizione del linguaggio permette (o richiede) che porzioni di una espressione booleana possano non essere valutate, allora il compilatore può ottimizzare la valutazione di una espressione booleana calcolando solo ciò che è strettamente necessario per determinare il suo valore. Così, in una espressione come `B1 || B2` né per `B1` né per `B2` è necessario valutarle completamente. Se sia `B1` che `B2` sono espressioni con effetti collaterali (ad esempio contengono una funzione che altera il valore di una variabile globale) allora si potrebbe ricevere una risposta non attesa.

Codice Short Circuit

In un codice short circuit (oppure jumping) gli operatori booleani `&&`, `||`, e `!` si traducono in salti. Gli operatori stessi non appaiono nel codice; invece, il valore di una espressione booleana è rappresentato da una posizione nella sequenza di codice.

Esempio Lo statement:

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

deve essere tradotto nel codice che segue. In questa traduzione, l'espressione booleana è true se il controllo raggiunge l'etichetta `L2`. Se l'espressione è false, il controllo va immediatamente all'istruzione con etichetta `L1` saltando `L2` e l'assegnamento `x = 0`.

```
if x < 100 goto L2
if False x > 200 goto L1
if False x != y goto L1
L2 : x=0
L1:
```

Statement di flusso di controllo

Consideriamo adesso la traduzione di espressioni booleane in un codice a tre indirizzi nel contesto di statement come quelli generati dalla grammatica che segue:

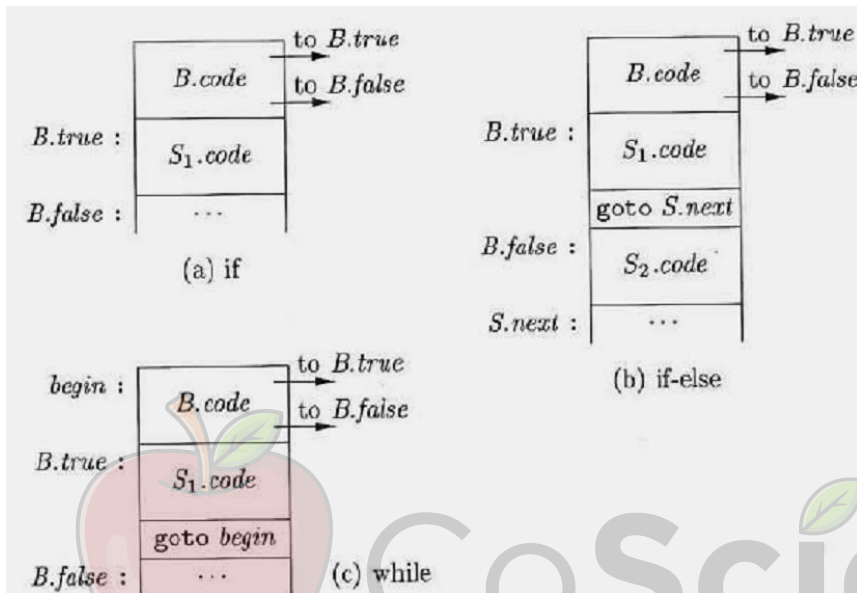
$$S \rightarrow \text{if}(B) S$$

$$S \rightarrow \text{if}(B) S1 \text{ else } S2$$

$$S \rightarrow \text{while}(B) S1$$

In queste produzioni, il non terminale B rappresenta una espressione booleana e il non terminale S rappresenta uno statement.

Sia S che B hanno un attributo sintetizzato `code` che dà la traduzione in codice a tre indirizzi. Per semplicità, costruiamo la traduzione $B.\text{code}$ e $S.\text{code}$ come stringhe, usando le definizioni guidate da sintassi. La traduzione di $\text{if}(B) S1$ consiste di $B.\text{code}$ seguito da $S1.\text{code}$, come illustrato nella figura che segue(a). All'interno di $B.\text{code}$ ci sono i salti basati sul valore di B . Se B è true, il controllo passa alla prima istruzione $S1.\text{code}$ e se B è false, il controllo passa all'istruzione immediatamente successiva a $S1.\text{code}$.



Le etichette per i salti $B.\text{code}$ e $S.\text{code}$ sono gestite usando attributi ereditati. Con una espressione booleana B , associamo due etichette: $B.\text{true}$ è l'etichetta con la quale il controllo passa all'istruzione se B è true, e $B.\text{false}$ è l'etichetta dell'istruzione alla quale il controllo passa se B è false. Ad uno statement S associamo un attributo ereditato $S.\text{next}$ che denota una etichetta per l'istruzione immediatamente successiva al codice di S . Un salto ad una etichetta L dall'interno di $S.\text{code}$ è evitata usando $S.\text{next}$.

La definizione guidata da sintassi nella figura che segue produce il codice a tre indirizzi per le espressioni booleane nel contesto di un **if**, **if-else** e un ciclo **while**. Assumiamo che `newlabel()` crea una nuova etichetta ogni volta che viene chiamata, e che `label(L)` attacca l'etichetta L alla prossima istruzione a tre indirizzi da generare.

Un programma consiste di uno statement generato $P \rightarrow S$. Le regole semantiche associate a queste produzioni inizializzano $S.\text{next}$ ad una nuova etichetta. $P.\text{code}$ consiste di $S.\text{code}$ seguito dalla nuova etichetta $S.\text{next}$. Il token `assign` nella produzione $S \rightarrow \text{assign simbolo usato per lo statement di assegnazione}$. Nella traduzione $S \rightarrow \text{if}(B) S1$ le regole semantiche nella figura che segue creano una nuova etichetta $B.\text{true}$ e agganciano essa alla prima istruzione a tre indirizzi generata per lo statement $S1$ come illustrato nella figura precedente (a). Così, i salti a $B.\text{true}$ all'interno del codice per B andrà al codice per $S1$. In più, settando $B.\text{false}$ a $S.\text{next}$, ci assicuriamo che il controllo salterà il codice di $S1$ se B è false.

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow \text{assign}$	$S.code = \text{assign}.code$
$S \rightarrow \text{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$
$S \rightarrow \text{if} (B) S_1 \text{ else } S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\quad \parallel label(B.true) \parallel S_1.code$ $\quad \parallel gen('goto' S.next)$ $\quad \parallel label(B.false) \parallel S_2.code$
$S \rightarrow \text{while} (B) S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \parallel B.code$ $\quad \parallel label(B.true) \parallel S_1.code$ $\quad \parallel gen('goto' begin)$
$S \rightarrow S_1 S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \parallel label(S_1.next) \parallel S_2.code$

- ✓ $P \rightarrow S$: le regole semantiche associate a questa produzione inizializzano la label $S.next$.
- ✓ $S \rightarrow \text{if} (B) S_1$: le regole semantiche creano una nuova label $B.true$ che verrà attaccata alla prima istruzione a 3 indirizzi di S .
- ✓ $S \rightarrow \text{if} (B) S_1 \text{ else } S_2$: il codice per l'espressione booleana B salta alla prima istruzione di S_1 se B è true mentre salta alla prima istruzione di S_2 se B è false. In più, il controllo passa sia per S_1 che per S_2 all'istruzione a tre indirizzi immediatamente successiva al codice di S – la sua etichetta è data dall'attributo ereditato $S.next$. Un goto esplicito $S.next$ appare dopo il codice di S_1 per saltare oltre il codice di S_2 . Nessuno goto è necessario alla fine del codice per S_2 visto che $S_2.next$ è lo stesso di $S.next$.
- ✓ $S \rightarrow \text{while} (B) S_1$: usiamo una variabile locale $begin$ per mantenere una nuova etichetta agganciata alla prima istruzione per lo statement di while, che è anche la prima istruzione di B . Usiamo una variabile piuttosto che un attributo perché $begin$ è locale alla regole semantiche per questa produzione. L'etichetta ereditata $S.next$ marca l'istruzione alla quale il controllo deve passare se B è false: quindi, $B.false$ è settata a $S.next$. Una nuova etichetta $B.true$ è agganciata alla prima istruzione di S_1 ; il codice per B genera il salto a questa etichetta se B è vera. Dopo il codice per S_1 piazziamo l'istruzione `goto begin` che causa un salto all'indietro all'inizio del codice per l'espressione booleana. Notare che $S_1.next$ è settato a questa etichetta $begin$, così i salti dall'interno di $S_1.code$ può andare direttamente a $begin$.
- ✓ Il codice per $S \rightarrow S_1 S_2$ consiste del codice per S_1 seguito dal codice per S_2 . Le regole semantiche gestiscono le etichette; la prima istruzione dopo il codice per S_1 è l'inizio del codice per S_2 ; e l'istruzione dopo il codice per S_2 è anche l'istruzione dopo il codice per S .

Traduzione di controllo di flusso di espressioni booleane

Le regole semantiche per le espressioni booleane in figura che segue, completano le regole semantiche per gli statement della figura precedente. Una espressione booleana è tradotta in istruzioni a tre indirizzi che valutano B usando salti condizionati e incondizionati ad una delle due etichette: B.true se B è true o B.false se B è false.

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \parallel B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \parallel label(B_1.false) \parallel B_2.code$
$B \rightarrow B_1 \&\& B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \parallel label(B_1.true) \parallel B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \text{ rel } E_2$	$B.code = E_1.code \parallel E_2.code$ $\parallel gen('if' E_1.addr \text{ rel } op E_2.addr 'goto' B.true)$ $\parallel gen('goto' B.false)$
$B \rightarrow \text{true}$	$B.code = gen('goto' B.true)$
$B \rightarrow \text{false}$	$B.code = gen('goto' B.false)$

La quarta produzione in figura $B \rightarrow E_1 \text{ rel } E_2$ è tradotta direttamente in una istruzione a tre indirizzi di confronto che salta all'appropriata locazione nel codice generato. Per esempio, B della forma $a < b$ viene tradotto in:

```
if a < b goto B.true
goto B.false
```

Le restanti produzioni di B sono tradotte come segue:

1. supponiamo che B sia della forma $B_1 \parallel B_2$. Se B_1 è true allora immediatamente sappiamo che tutta B è true, così $B_1.true$ è lo stesso di B.true. Se B_1 è false allora B_2 deve essere valutata, così rendiamo $B_1.false$ alla etichetta della prima istruzione del codice di B_2 . Le uscite su true e su false per B_2 sono le stesse uscite per true e per false di B, rispettivamente.
2. la traduzione di $B_1 \&\& B_2$ è simile.
3. nessun codice è necessario per una istruzione B della forma $!B_1$; basta solo scambiare le uscite per vero e per falso di B per ottenere le uscite per vero e per falso di B_1 .
4. le costanti true e false si traducono in B.true e B.false, rispettivamente.

Esempio Consideriamo di nuovo il seguente statement:

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

Usando la definizione guidata dalla sintassi fino ad ora descritta vorremmo ottenere il codice seguente:

```
if x < 100 goto L2
goto L3
L3: if x > 200 goto L4
goto L1
L4: if x != y goto L2
goto L1
L2: x = 0
L1:
```

Lo statement definito sopra costituisce un frammento di programma generato da $P \rightarrow S$. Le regole semantiche per la produzione generano una nuova etichetta L1 per l'istruzione che viene dopo il codice per S. Lo statement S ha la forma $\text{if } (B) S1$ dove S1 è $x = 0$; così le regole semantiche generano una nuova etichetta L2 e attaccano questa alla prima (soltanto, in questo caso) istruzione in S1.code che $p \ x = 0$.

Visto che $||$ ha minore precedenza rispetto a $\&\&$, l'espressione booleana ha la forma $B1 || B2$ dove B1 è $x < 100$. Seguendo la regola, B1.true è L2, l'etichetta dell'assegnazione $x = 0$; B1.false è una nuova etichetta L3, attaccata alla prima istruzione del codice di B2. Notare che il codice generato non è ottimale, vale a dire che la traduzione ha più istruzioni goto del codice mostrato qualche pagina prima in questo documento. L'istruzione goto L3 è ridondante, visto che L3 è l'etichetta alla istruzione immediatamente prossima. Le due istruzioni goto L1 possono essere eliminate usando l'iffalse invece del semplice if.

Eliminare i goto ridondanti

Nell'esempio appena mostrato. Il confronto $x > 200$ si traduce in un frammento di codice:

```
if x > 200 goto L4
goto L1
L4: ...
```

Invece, consideriamo l'istruzione

```
iffalse x > 200 goto L1
L4 ...
```

Questa istruzione di iffalse trae vantaggio sul naturale flusso da una istruzione alla successiva nella sequenza, così il controllo semplicemente "cade fino" all'etichetta L4 se $x > 200$ è false, evitando, quindi, un salto.

Nello stile di codice per if e while, il codice per lo statement S1 segue immediatamente il codice per l'espressione booleana B. Usando una speciale etichetta fall (ad esempio "non generare nessun salto") possiamo adattare le regole semantiche per permettere al controllo di cadere dal codice di B al codice di S1. Le nuove regole per $S \rightarrow \text{if } (B) S1$ settano B.true a fall:

```
B.true = fall
B.false = S1.next = S.next
S.code = B.code || S1.code
```

In maniera simile anche le regole per l'if-else e per il while settano B.true a fall. Adesso adattiamo le regole semantiche per le espressioni booleane per permettere al controllo di cadere il più possibile. Le nuove regole $B \rightarrow E1 \text{ rel } E2$, in figura che segue, genera due istruzioni se entrambe B.true e B.false sono etichette esplicite; cioè nessuna di esse è uguale a fall. Altrimenti, se B.true è una etichetta esplicita allora B.false deve essere uguale a fall così esse generano un'istruzione if che passa il controllo a fall se la condizione è false. Viceversa, se B.false è una etichetta esplicita, allora B.true deve essere fall generando così un iffalse. In tutti gli altri casi, ossia quando B.false e B.true sono entrambe fall, allora nessun salto viene generato.

```
test = E1.addr rel.op E2.addr
s = if B.true ≠ fall and B.false ≠ fall then
    gen('if' test 'goto' B.true) || gen('goto' B.false)
    else if B.true ≠ fall then gen('if' test 'goto' B.true)
    else if B.false ≠ fall then gen('iffalse' test 'goto' B.false)
    else ''
B.code = E1.code || E2.code || s
```

Nelle nuove regole per $B \rightarrow B1 || B2$ nella figura che segue, va notato che il significato dell'etichetta fall per B è differente dal significato che la stessa ha per B1. Supponiamo che B.true è fall; il controllo cade su B, se

B è vera. Sebbene B1 rende la valutazione di B vera, B1.true deve assicurare che il controllo salti il codice di B2 così da arrivare alla prossima istruzione che viene dopo B. In altri termini, se B1 è valutata false, il valore di verità di B è determinato dal valore di B2 così le regole nella figura sottostante assicurano che B1.false corrisponda al passaggio del controllo dal codice di B1 al codice di B2.

Le regole semantiche sono per $B \rightarrow B1 \ \&\& \ B2$ praticamente simili.

Backpatching

Un problema chiave che si presenta nella generazione del codice per le espressioni booleane e degli statement di flusso di controllo è il matching tra una istruzione di salto e la destinazione del salto stessa.

Per esempio, la traduzione di una espressione booleana B in `if (B) S` contiene un salto, quando B è falsa, all'istruzione che segue il codice di S. In un solo passo di traduzione, B deve essere tradotta prima che S sia esaminata. Quale sarà quindi il target dell'istruzione goto che salta oltre il codice per S?

Come abbiamo mostrato sino ad ora, negli esempi precedenti abbiamo usato delle etichette come attributi ereditati dove i salti dovevano essere generati. Ma un passo separato è quindi necessario per abbinare le etichette agli indirizzi.

Questa sezione prende in esame un metodo complementare, ossia il backpatching, nel quale la lista di salti sono passati come attributi sintetizzati. Più nel dettaglio, quando un salto viene generato, la destinazione del salto è temporaneamente non specificata. Ognuno di tali salti è posto in una lista di salti le cui etichette devono essere riempite con la corretta label quando è possibile determinarle. Tutti i salti della lista hanno la stessa etichetta target.

Generazione del codice in un solo passo usando il backpatching

Il backpatching può essere usato per generare codice per le espressioni booleane e gli statement del flusso di controllo in un solo passo. Le traduzioni che noi generiamo saranno della stessa forma di quelle viste sino ad ora, eccezion fatta per la gestione delle etichette. In questa sezione, gli attributi sintetizzati **truelist** e **falselist** del non terminale B sono usate per gestire le etichette nei salti sul codice per le espressioni booleane. In particolare, B.truelist sarà la lista di salti condizionati o non nelle quale inseriremo le etichette alle quale il controllo passa se B è true. Allo stesso modo, B.falselist è la lista delle istruzioni alle quali eventualmente il controllo passa se B è false. Per come viene generato il codice per B, i salti alle uscite per vero e per falso sono incomplete, con l'etichetta non riempita. Queste salti incompleti sono piazzati in liste puntate da B.truelist e B.falselist. In maniera simile, uno statement S ha un attributo sintetizzato S.nextlist che denota una lista di salti alle istruzioni immediatamente successive al codice di S. Per specificare, noi generiamo le istruzioni un array di istruzioni, e le etichette saranno indicizzate in questo array. Per manipolare le liste di salti, usiamo tre funzioni:

1. **makelist(i)** crea una nuova lista che contiene solo i, un indice nell'array delle istruzioni; makelist restituisce un puntatore alla lista creata.
2. **merge(p1, p2)** che concatena le liste puntate da p1 e p2 e restituisce un puntatore ad una lista concatenata
3. **backpatching(p, i)** che inserisce i come etichetta di destinazione per ogni istruzione sulla lista puntata da p.

Costruiamo adesso uno schema di traduzione adatto per la generazione del codice per espressioni booleane durante un parsing bottom-up. Un non terminale marker M nella grammatica causa una azione semantica che raccoglie, al momento opportuno, l'indice della istruzione successiva da generare. La grammatica è come la seguente:

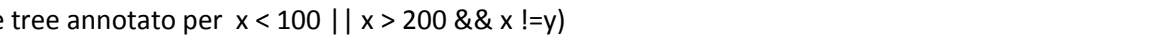
```
B → B1 || M B2 | B1 && M B2 | ! B1 | ( B1 ) | E1 rel E2 | true | false
M → ε
```

deniamo l'azione semantica (1) per la produzione $B \rightarrow B1 \mid B2$: Se $B1$ è vera, allora B è vera e i salti $B1.truelist$ diventano parte di $B.truelist$. Se $B1$ è falsa, allora, dobbiamo controllare $B2$, così il target per i salti $B1.falselist$ puntino all'inizio del codice per $B2$. Questo target è ottenuto usando il marker M . Il non terminale in questione produce come attributo sintetizzato $M.instr$, l'indirizzo della prossima istruzione, appena prima che inizi il codice di $B2$. Per ottenere questo indice dell'istruzione usiamo con la produzione $M \rightarrow \epsilon$ l'azione semantica: $\{M.instrs = nextinstr\}$. Il non terminale $nextinstr$ mantiene l'indice della istruzione successiva ad M . Questo valore sarà "backpatch" alla lista $B1.falselist$ (vale a dire, ad ogni istruzione della lista sarà associata una etichetta $M.instr$) quando

- è simile alla (1). L'azione (3) per $B \rightarrow !B$ scambia il valore da vero a falso. L'azione (4) ignora

semplicità, l'azione semantica (5) genera due istruzioni, un goto condizionato e uno non condizionato.

ta da B.truelist e B.falselist, rispettivamente.


$$X = 100 \text{ } // \text{ } X = 100 \text{ } \&\& X = 1 \text{ } ,$$

Il parse tree annotato è mostrato sopra. Per semplicità truelist, falselist e instr sono indicate con la loro iniziale. Le azioni sono eseguite durante una visita in profondità dell'albero. Visto che tutte le azioni appaiono alla fine della parti destre, esse possono essere eseguite in congiunzione con le riduzioni durante un parse bottom-up. In risposta alla riduzione di $x < 100$ a B attraverso la produzione (5) vengono generate le due istruzioni seguenti:

```
100: if x < 100 goto _
101: goto _
```

La numerazione parte arbitrariamente da 100. Il non terminale marker M nella produzione:

$B \rightarrow B1 \parallel M B2$

Registra il valore di nextinstr, il quale a questo momento è 102. La riduzione di $x > 200$ a B tramite la produzione (5) genera le istruzioni:

```
102: if x > 200 goto _
103: goto _
```

La sottoespressione $x > 200$ corrisponde a B1 nella produzione: $B \rightarrow B1 \&\& M B2$. Il non terminale marker M registra il valore corrente di nextinstr, il quale è adesso 104. Riducendo $x \neq y$ in B attraverso la produzione (5) viene generato:

```
104: if x != y goto _
105: goto _
```

Adesso riduciamo $B \rightarrow B1 \&\& M B2$. La corrispondente azione semantica chiama `backpatch(B1.truelist, M.instr)` per accoppiare l'uscita per vero di B1 alla prima istruzione di B2. Visto che B1.truelist è {102}, e M.instr è 104, questa chiamata a `backpatch` riempie con 104 l'istruzione in 102. Le sei istruzioni generate prima sono mostrate nella figura che segue (a). L'azione semantica associata alla riduzione finale

$B \rightarrow B1 \parallel M B2$

chiamata `backpatch({101}, 102)` che genera le istruzioni in figura che segue (b). L'intera espressione è vera se e solo se i goto delle istruzioni 100 e 104 sono raggiunti, ed è falsa se e solo se i goto alle istruzioni 103 e 105 sono raggiunti. Queste istruzioni avranno le loro destinazioni completamente riempite successivamente nella compilazione, non appena è stato visto cosa deve essere fatto sulla veridicità o falsità della espressione.

```
100: if x < 100 goto _
101: goto _
102: if x > 200 goto 104
103: goto _
104: if x != y goto _
105: goto _
```

(a) After backpatching 104 into instruction 102.

```
100: if x < 100 goto _
101: goto 102
102: if y > 200 goto 104
103: goto _
104: if x != y goto _
105: goto _
```

(b) After backpatching 102 into instruction 101.

Statement del flusso di controllo

Adesso usiamo il backpatching per tradurre gli statement del flusso di controllo in un solo passo. Consideriamo gli statement generati dalla seguente grammatica:

$$S \rightarrow \text{if}(B) S \mid \text{if}(B) S \text{ else } S \mid \text{while}(B) S \mid \{ L \} \mid A ;$$
$$L \rightarrow L S \mid S$$

Qui S denota uno statement, L una lista di statement, A uno statement di assegnazione e B è una espressione booleana. Notare che dovrebbero esserci altre produzioni, come quelle per gli statement di assegnazione. Comunque, le produzioni date sono sufficienti per illustrare le tecniche usate per tradurre gli statement del flusso di controllo. Lo stile di codice per l'if, if-else e il ciclo while è lo stesso dato nella sezione precedente. Facciamo l'assunzione che la sequenza di codice nell'array delle istruzioni riflette il naturale flusso di controllo da una istruzione alla successiva. Lo schema di traduzione nella figura che segue mantiene le liste dei salti che sono riempite quando le loro destinazioni vengono trovate. Il non terminale B per le espressioni booleane ha due liste `truelist` e `falselist` corrispondenti alle uscite per vero e per falso da B . Gli statement generati dal non terminale S e L hanno una lista di salti non riempiti, dati dall'attributo `nextlist`, che deve eventualmente essere completata attraverso il backpatching.

$S.\text{nextlist}$ è una lista di tutti i salti condizionati e non condizionati alla istruzione che segue al codice per lo statement S nell'ordine di esecuzione. $L.\text{nextlist}$ è definita in maniera simile. Consideriamo l'azione semantica (3). Lo stile di codice per la produzione $S \rightarrow \text{while}(B) S_1$ è stato mostrato nella sezione 4.6. Le due occorrenze del non terminale marker M nella produzione:

$$S \rightarrow \text{while } M_1(B) M_2 S_1$$

registrano i numeri delle istruzioni dell'inizio del codice per B e l'inizio del codice per S_1 . Le etichette corrispondenti sono `begin` e `B.true`, rispettivamente.

- 1) $S \rightarrow \text{if}(B) M S_1$ { `backpatch(B.truelist, M.instr);`
 `S.nextlist = merge(B.falselist, S1.nextlist);` }
- 2) $S \rightarrow \text{if}(B) M_1 S_1 N \text{ else } M_2 S_2$
 { `backpatch(B.truelist, M1.instr);`
 `backpatch(B.falselist, M2.instr);`
 `temp = merge(S1.nextlist, N.nextlist);`
 `S.nextlist = merge(temp, S2.nextlist);` }
- 3) $S \rightarrow \text{while } M_1(B) M_2 S_1$
 { `backpatch(S1.nextlist, M1.instr);`
 `backpatch(B.truelist, M2.instr);`
 `S.nextlist = B.falselist;`
 `emit('goto' M1.instr);` }
- 4) $S \rightarrow \{ L \}$ { `S.nextlist = L.nextlist;` }
- 5) $S \rightarrow A ;$ { `S.nextlist = null;` }
- 6) $M \rightarrow \epsilon$ { `M.instr = nextinstr;` }
- 7) $N \rightarrow \epsilon$ { `N.nextlist = makelist(nextinstr);`
 `emit('goto -');` }
- 8) $L \rightarrow L_1 M S$ { `backpatch(L1.nextlist, M.instr);`
 `L.nextlist = S.nextlist;` }
- 9) $L \rightarrow S$ { `L.nextlist = S.nextlist;` }

Di nuovo, la sola produzione per M è $M \rightarrow \epsilon$. L'azione (6) setta gli attributi `M.instr` al numero della prossima istruzione. Dopo che il corpo di S_1 del ciclo di `while` è stato eseguito, il controllo passa all'inizio. Perciò, quando riduciamo `while M1(B) M2 S1` a S , noi facciamo un `backpatch` di `S1.nextlist` per settare tutte le destinazioni della lista a `M1.instr`. Un salto esplicito all'inizio del codice per B è agganciato alla fine del codice per S_1 il controllo può anche "cadere oltre il fondo".

Su *B.truelist* è fatto un backpatch per andare all'inizio di *S1* settando tutte le destinazioni della lista *B.truelist* a *M2.instr*. L'uso di *S.nextlist* e *L.nextlist* diventa più convincente quando il codice generato riguarda uno statement condizionale *if (B) S1 else S2*. Se il controllo "cade oltre il fondo" di *S1*, dobbiamo includere alla fine del codice di *S1* un salto che va oltre il codice per *S2*. Usiamo a tal punto un nuovo non terminale marker per generare questi salti dopo *S1*. Sia *N* il non terminale marker in questione, esso avrà l'attributo *N.nextlist*, che sarà una lista che consiste di un numero di istruzioni di salto *goto_* che sono generate dall'azione semantica (7) per *N*. L'azione semantica (2) tratta proprio il caso di statement condizionale *if-else*:

$$S \rightarrow \text{if } (B) \text{ } M1 \text{ } S1 \text{ } N \text{ else } M2 \text{ } S2$$

Facciamo un backpatch dei salti, quando *B* è vera, all'istruzione *M1.instr*; la seconda è l'inizio del codice per *S1*. In maniera simile facciamo il backpatch dei salti quando *B* è falsa andando all'inizio del codice di *S2*. La lista *S.nextlist* include tutti i salti oltre *S1* e *S2* come pure il salto generato da *N*. (la variabile *temp* è una temporanea che viene usata per il merge delle liste) L'azione semantica (8) e (9) gestiscono le sequenze di statement. In $L \rightarrow L1 \text{ } M \text{ } S$.

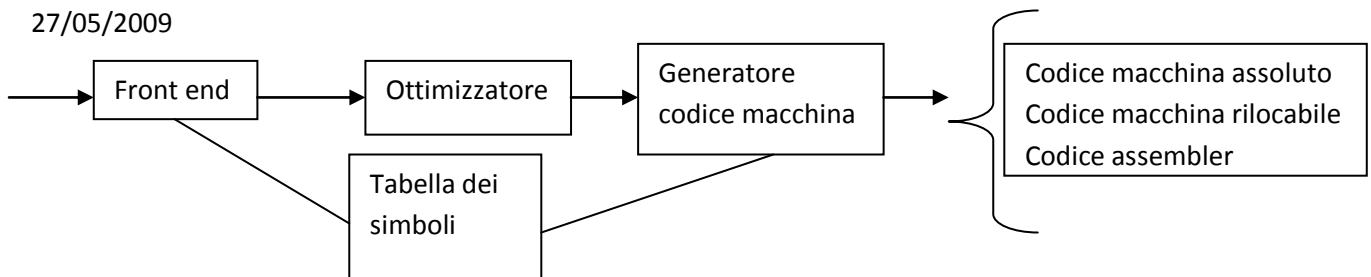
L'istruzione che segue il codice di *L1* nell'ordine di esecuzione è l'inizio di *S*. Così sulla lista *L1.nextlist* si fa un backpatch all'inizio del codice per *S*, che è dato da *M.instr*. In $L \rightarrow S$, *L.nextlist* è la stessa di *S.nextlist*. Notare che nessuna nuova istruzione è stata generata in queste regole semantiche, ad eccezione per le regole (3) e (7). Tutto il rimanente codice è generato da azioni semantiche associate alle espressioni e agli statement di assegnazione. Il flusso di controllo comporta il corretto backpatching così che le valutazioni delle assegnazioni e delle espressioni booleane saranno collegate opportunamente.



CoScienze
Associazione

LEZIONE 21 – GENERAZIONE DEL CODICE MACCHINA

27/05/2009



La fase di ottimizzazione è equivalente a risolvere un problema NP-completo. Usiamo algoritmi efficienti, che non arrivano all'ottimo, usando euristiche. È comunque la parte che ha reso i linguaggi ad alto livello efficienti quanto i linguaggi a basso livello. Il generatore di codice: traduce codice a tre indirizzi in codice macchina. **Deve essere un codice macchina ottimo, cioè deve sfruttare al massimo le risorse della macchina;** ma tale problema, così posto, è indecidibile, quindi non esistono algoritmi che garantiscono di arrivare al codice macchina ottimo.

Vediamo quali sono le caratteristiche dei possibili formati dell'output del generatore di codice:

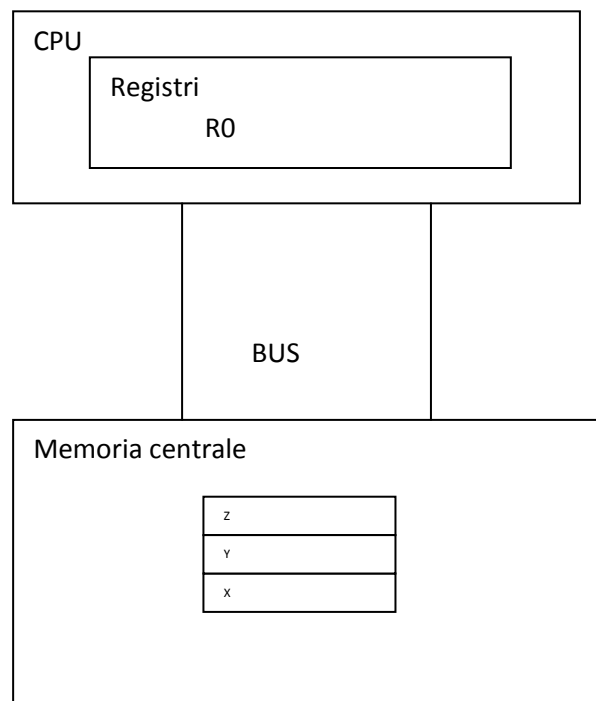
- **Codice macchina assoluto:** è posto in una locazione fissa della memoria, sono note le locazioni usate; il codice ottenuto può essere eseguito senza ulteriori fasi.
- **Codice macchina rilocabile:** si possono compilare separatamente più sottoprogrammi, poi si devono linkare per l'esecuzione dell'intero programma. Vi è una fase in più di linkaggio però vi è la possibilità di compilare separatamente i sotto-moduli.
- **Codice assembler:** è il formato più semplice; tale codice dovrà essere reinterpreto per essere eseguito dalla macchina. È quello cui ci riferiremo.

Vediamo un esempio:

supponiamo di avere l'istruzione a tre indirizzi: $x := y + z$ vediamo come viene tradotta in assembler:

- Se si ha a disposizione una macchina CISC (che permette operazioni complesse come quella di somma avendo un operando nella memoria centrale e uno nel registro) avremo:

1. MOV y,R0
2. ADD z, R0
3. MOV R0,x



I registri costituiscono una memoria velocissima poiché sono direttamente collegati alla CPU, mentre l'accesso alla memoria centrale è più lento. L'ideale sarebbe avere tutti i dati nei registri così da fare le operazioni il più velocemente possibile (ma i registri costano). Se non si hanno abbastanza registri bisogna fare riferimento alla memoria centrale.

- Per le macchine RISC l'istruzione di addizione si può fare solo tra il contenuto di due registri, quindi `ADD ,R0` è errata per tali macchine. Il codice diventa:
 1. `MOV y,R0`
 2. `MOV z,R1`
 3. `ADD R0,R1`
 4. `MOV R1,x`

È essenziale quindi conoscere due fattori importanti:

- 1. Conoscere il set di istruzioni dell'architettura a disposizione;**
- 2. Sapere il numero di registri a disposizione cercando di inserirvi le variabili di uso più frequente.**

Problema principale di generatore di codice: generare codice che sfrutta al massimo i registri, poiché l'accesso ai registri è molto veloce. Tutto ciò che vedremo sono tecniche per risolvere tale problema. Se l'istruzione a tre indirizzi successiva è `t:=x+k` un banale generatore di codice, non tenendo conto di ciò che è già contenuto nei registri, può produrre, con una grammatica ad attributi, le istruzioni:

```
MOV x,R0
ADD k,R0
MOV R0,t
```

Se invece sfruttiamo il fatto che `x` è già nel registro `R0`, la prima `MOV` può essere eliminata. Quindi per fare una buona traduzione bisogna avere una visione globale della situazione dei registri, tradurre meccanicamente un'istruzione a tre indirizzi alla volta non è una buona soluzione. Se `x` fosse una variabile temporanea, fittizia, cioè se non è necessario conservare il suo valore in memoria, si potrebbe anche eliminare l'ultima `MOV` del primo esempio. Se invece `x` è una variabile del programma non la si può eliminare.

Ad esempio

Sorgente: $x:=y+z*n$ \rightarrow Codice a tre indirizzi: $t:=z*n; x:=y+t$

`t` è una variabile fittizia, temporanea, alla quale il programma non farà mai riferimento. È introdotta dal compilatore. `x` è una variabile introdotta dal programmatore, le è assegnata una memoria, un offset, etc...; il programma potrà riferirsi più volte a tale variabile. Nei registri bisogna avere le istruzioni e i dati maggiormente usati (per questo si usano le tecniche euristiche). Ad esempio in un `for(int i=0; i<40; i++)` la variabile `i` verrà acceduta frequentemente e bisogna fare in modo che sia sempre in un registro. Se si riempiono tutti i registri, si userà solo quello che contiene una variabile che si usa più raramente; la difficoltà dei generatori di codice sta proprio nello stabilire quali variabili avere nei registri, cioè quali sono le variabili più usate e quali quelle meno usate. In C si possono dichiarare variabili forzandole a restare nei registri: `register int i`; si ritorna sempre al problema di sfruttare i registri e il set di istruzioni a disposizione.

Formalmente i due problemi principali del generatore di codice sono quelli del register allocation e del code selection. Bisogna anche conoscere il costo di ogni singola istruzione. Ad esempio le istruzioni che coinvolgono solo i registri hanno un costo inferiore a quelle che coinvolgono registri e memoria. Si cerca di produrre codice macchina breve, ma può capitare che codice troppo breve sia troppo costoso; in tali casi è preferibile avere qualche istruzione macchina in più, con un costo totale inferiore.

Data l'istruzione a tre indirizzi $a:=b+c$ la traduzione ottima in codice assembler è ADD Ri,Rj unica istruzione di costo 1; ma affinché ciò sia possibile deve accadere che b si trovi in Rj, c in Ri ponendo il risultato in Rj; cioè si mantengono il più possibili le variabili di *prossimo uso* nei registri.

Una variabile ha un uso se si trova a destra di un'assegnazione. Nell'istruzione precedente "b" e "c" hanno un uso. Se si ha: $b:=x+a$ $a:=b+c$ allora si ha un prossimo uso della b ma non un prossimo uso della a che viene ridi chiarata e non conviene mantenerla nel registro. Vi è un algoritmo che per ogni blocco di base ci dice quali sono le variabili che hanno un prossimo uso e ci permetterà di mantenere nei registri solo queste variabili. Considerando il caso precedente abbiamo questa doppia situazione:

- Se b non ha un prossimo uso, Rj può perdere b, quindi l'istruzione ADD Ri,Rj va bene
- Se b ha un prossimo uso l'istruzione ADD Ri,Rj non è ottima poiché fa perdere il valore di b nel registro Rj.

Per tenere traccia dei valori dei registri e delle locazioni delle variabili vi sono due tabelle:

1. DESCRITTORE DEI REGISTRI

REGISTRO	CONTENUTO DI REGISTRO
Ri	C
Rj	B
...	...

2. DESCRITTORE DI INDIRIZZI DI MEMORIA

Variabile	Dove si trova
C	Ri
B	Rj
...	...

Tali strutture dati ci permettono di sapere, in ogni momento, cosa c'è nei registri e dove sono poste le variabili. Supponiamo che con tale configurazione delle tabelle si incontra l'istruzione $a:=b+c$ e che sia "a" che "b" che "c" abbiano un prossimo uso. L'istruzione ADD Ri,Rj è da scartare poiché fa perdere b da Rj, ma b ha un prossimo uso quindi è preferibile che rimanga in un registro. Vogliamo creare istruzioni assembler che producono la seguente situazione:

1. Descrittore dei registri

Ri	c
Rj	b
Rk	a

2. Descrittore di indirizzi di memoria

c	Ri
b	Rj
a	Rk

Una buona soluzione è:

MOV Rj, Rk costo 1

ADD Ri, Rk costo 1

Se una delle tre variabili non ha un prossimo uso si potrebbe trovare una soluzione migliore. **Il concetto di prossimo uso è locale ai singoli blocchi di base; se lo si estende a tutto il codice si ha il concetto di *in vita*: una variabile è in vita se ha un prossimo uso ma questa volta si fa riferimento a tutto il programma e non ad un solo blocco base.**

ES.

Blocco di base

```
.  
.   
x:=n (x ha un prossimo uso)  
a:=x+y (a è in vita)
```

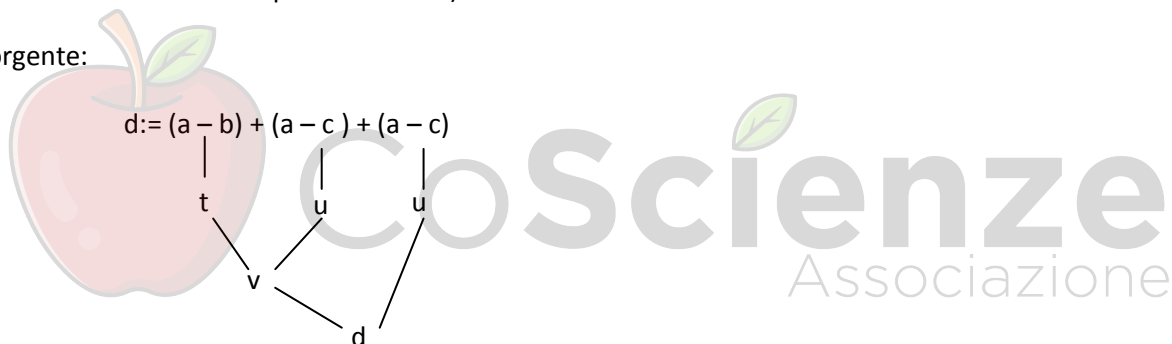
Blocco di base

```
.  
.   
z:a+b
```

Tali informazioni, prossimo uso e in vita, dovrebbero essere attribuiti delle variabili, i cui valori devono essere inseriti nella tabella dei simboli. **Le variabili da mantenere nei registri sono quelle che hanno un prossimo uso, se restano registri liberi si riempiono con le variabili in vita.**

Se non si fa un'analisi globale del programma, cioè se non si ha informazioni sulle variabili in vita, una possibilità è assumere che tutte le variabili del programma sono in vita, e le variabili temporanee non in vita. Le variabili temporanee non sono mai in vita, poiché sono variabili locali ai singoli blocchi di base, in blocchi diversi e non si troveranno mai temporanee uguali. **Il prossimo uso di una variabile non deve essere troppo lontano altrimenti è inutile bloccare un registro con un dato che sarà utilizzato dopo molte istruzioni.** Non vi è un criterio assoluto fisso per decidere quali variabili mantenere nei registri, dipende dai contesti. Nella generazione di codice intermedio si possono creare molte variabili temporanee che non saranno mai in vita contemporaneamente, alle quali si può assegnare le stesse locazioni di memoria (poiché non sono mai in vita contemporaneamente).

Es. sorgente:



Codice a 3 indirizzi:

t=a-b

u= a-c

v=t+u

d=v+u

Potremmo tradurla anche in notazione postfissa, sono tutte forme intermedie equivalenti. t,u,v sono temporaneamente introdotte dal generatore di codice intermedio. Vediamo come tradurre tali istruzioni in assembler: supponiamo di avere a disposizione due registri R0, R1 inizialmente vuoti. Guardando la 1 istruzione si nota che t e a hanno un prossimo uso, b non ha un prossimo uso, quindi bisognerebbe fare in modo che dopo la traduzione di tale istruzione nei due registri vi sia t ed a.

```
MOV  a, R0 } R0 | t
SUB   b, R0 }
MOV  a, R1 } R0 | t
SUB   c, R1 } R1 | u
```

```

ADD   R1, R0 } R0 | v
          } R1 | u
ADD   R1, R0 } R0 | d
          } R1 | u
MOV   R0, d

```

Tale codice non è ottimo, si potrebbe migliorare se dopo la traduzione di ogni istruzione si lasciano nei registri le variabili che hanno un prossimo uso. La soluzione migliore è inserire dopo la prima istruzione, la seguente MOV R0,R1 costo 1 il cui effetto è porre a in R1 in modo tale da cancellare successivamente la 3 MOV a, R1 che ha costo 2, maggiore. Tale trasformazione porta un miglioramento ma potrebbe essere troppo costosa in termini di tempo; vogliamo generatori di codici che siano efficienti, di solito la compilazione con l'ottimizzazione la si fa solo sulla versione finale del programma.

In conclusione vogliamo generatori di codice che siano efficienti; di solito la compilazione con l'ottimizzazione la si fa solo sulla fase finale del programma. Come già detto determinare un codice ottimo corrisponde ad un problema indecidibile. C'è infine da aggiungere che per determinare se una variabile ha un prossimo uso o è in vita comporta un costo di cui bisogna tenere conto quando si sviluppa il compilatore.



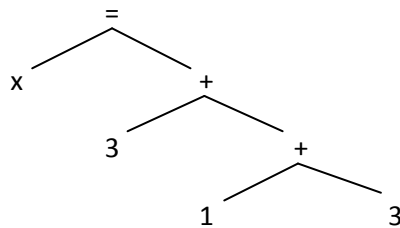
LEZIONE 22 - ESERCITAZIONE

03/06/2009

Trasformare il seguente codice in codice a tre indirizzi:

```
x=3+1+3;  
while(n>0) {  
    n=n-2*x;  
    if(n>3)  
        A:="mi sto impegnando";  
}  
n=2+m*n;
```

Vediamo un attimo l'albero sintattico per capire come fare la trasformazione:



Il codice $x=3+1+3$ diventerà:

t0=1+3

t1=3+t0

x=t1 dove t0 e t1 sono variabili temporanee.

Ricordando la lezione si nota che è necessaria la label "begin" per indicare l'inizio del while e poi i due salti: uno alla prima istruzione all'interno del while e un altro per la prima istruzione al di fuori del while.

Il codice a tre indirizzi sarà questo:

L0: if(n>0)
 goto L1

goto L2

L1: t1=2*x
 n=n-t1
 if(n>3)
 goto L3

goto L4

L3: A:"mi sto impegnando"

L4: goto L0

L2: t1=m*n
 n=2+t1.

Se usiamo la tecnica dell'if false otteniamo:

t0=1+3

x=3+t0

L0: if False (n>0)
 goto L1

t1=2*x

n=n-t1

if False(n>3)
 goto L0

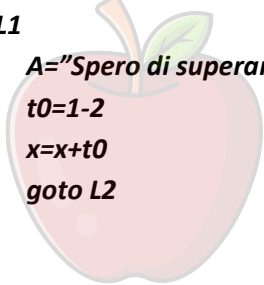
```
L1:    t1=m*n  
n=2+t1
```

Trasformare il seguente codice in codice a tre indirizzi:

```
if(n>3){  
    x=3;  
    while(x>0){  
        A="Spero di superare l'esame";  
        x=x+1-2;  
    }  
}
```

Che diventa:

```
if(n>3)  
    goto L0  
goto L1  
L0:    x=3  
L2:    if(x>0)  
        goto L3  
goto L1  
L3:    A="Spero di superare l'esame":  
        t0=1-2  
        x=x+t0  
        goto L2  
L1:
```



CoScienze
Associazione

ESERCIZI VARI

TRACCIA

Dato il brano di codice

*if (a > 1) {b = a + 1 * c} else b = max(1, 2)*

utilizzando le usuali convenzioni lessicali e grammaticali dei linguaggi di programmazione, e rifacendosi agli insegnamenti del libro di testo, mostrare il flusso di token in input alla fase di analisi sintattica ed il risultato finale di tale fase.

Svolgimento:

<id, 'if'>	<LPAR, '('>	<id, 'a'>	<RLOP, 'gt'>	<num, '1'>	<RPAR, ')'>	<LGRAFF, '{'>
<id, 'b'>	<ASSIGNOF, '='>	<id, 'a'>	<ADDOP, '+'>	<num, '1'>	<MULTOP, '*'>	<id, 'c'>
<RGRAFF, '}'>	<else, 'else'>	<id, 'b'>	<ASSIGNOF, '='>	<max, 'max'>	<LPAR, '('>	<num, '1'>
<sem, ';'>	<num, '2'>	<RPAR, ')'>				

TRACCIA

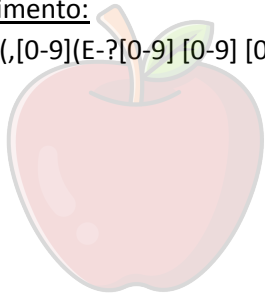
Dare la definizione regolare che descriva tutte le seguenti stringhe:

n n,d n,dE-m n,dEm

dove n è rappresentativo di un qualsiasi numero decimale a uno o più cifre, d di una sola cifra decimale, e m un numero decimale a compreso tra due e tre cifre .

Svolgimento:

$[0-9]^+(,[0-9](E-?[0-9] [0-9] [0-9]?)?)?$



CoScienze
Associazione