Riassunto Traduttori 2

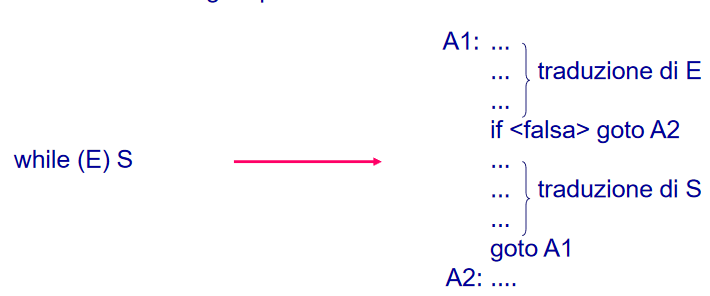
# Traduzione diretta

La traduzione diretta è una tecnica che permette di arricchire le produzioni grammaticali con regole di traduzione, generalmente istruzioni che effettuano una traduzione, l’idea è associare a ogni regola grammaticale una regola di traduzione, ad esempio:

S→ while(E) S

A questa produzione si associa una regola per la traduzione dei cicli while, ovvero:

Se la condizione calcolata si rivela falsa, allora viene eseguito il salto A2, altrimenti si va in S e con il salto A1 si torna all’inizio.

Tutti i dettagli della traduzione dipenderanno dalla struttura di E e di S (il corpo del ciclo), essi verranno ottenuti attraverso ulteriori regole associate alle produzioni per l’espansione di queste ultime. Le strutture sintattiche utilizzate da E e S si trovano anche in altri linguaggi come quelli di comandi, di markup oppure il linguaggio naturale. Nel caso dei linguaggi di markup, vi sono delle “parentesi” che permettono di indicare quali parti sono interessate, body ad esempio contiene tutto ciò che riguarda la parte visibile della pagina web, p indica i paragrafi, eccetera, per visualizzare la pagina si dovrà quindi leggere e ricostruire con apposite librerie. I linguaggi naturali sono difficili da comprendere per gli essere umani anche solo sintatticamente, in generale lo sono anche per i sistemi automatici in quanto l’elaborazione presente problemi che si cercano di evitare nei linguaggi formali, come l’ambiguità, per disambiguare una frase bisognerebbe capire se sono presenti riferimenti al soggetto nelle frasi successive. Alcuni linguaggi naturali sono controllati, una via di mezzo che permette di facilitarne la traduzione automatica, alcuni sono veri e propri linguaggi formali comprensibili all’uomo.

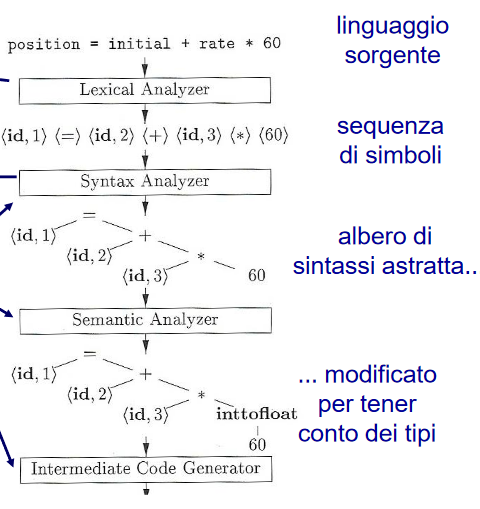
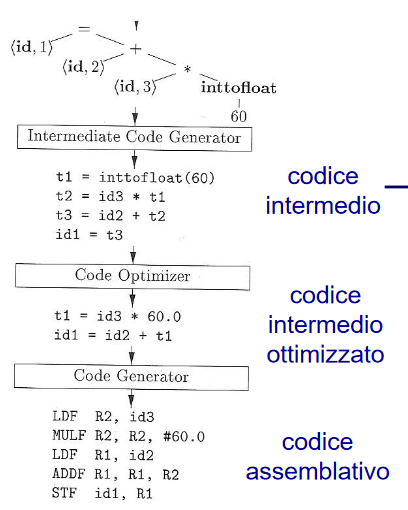
## Passi della traduzione:

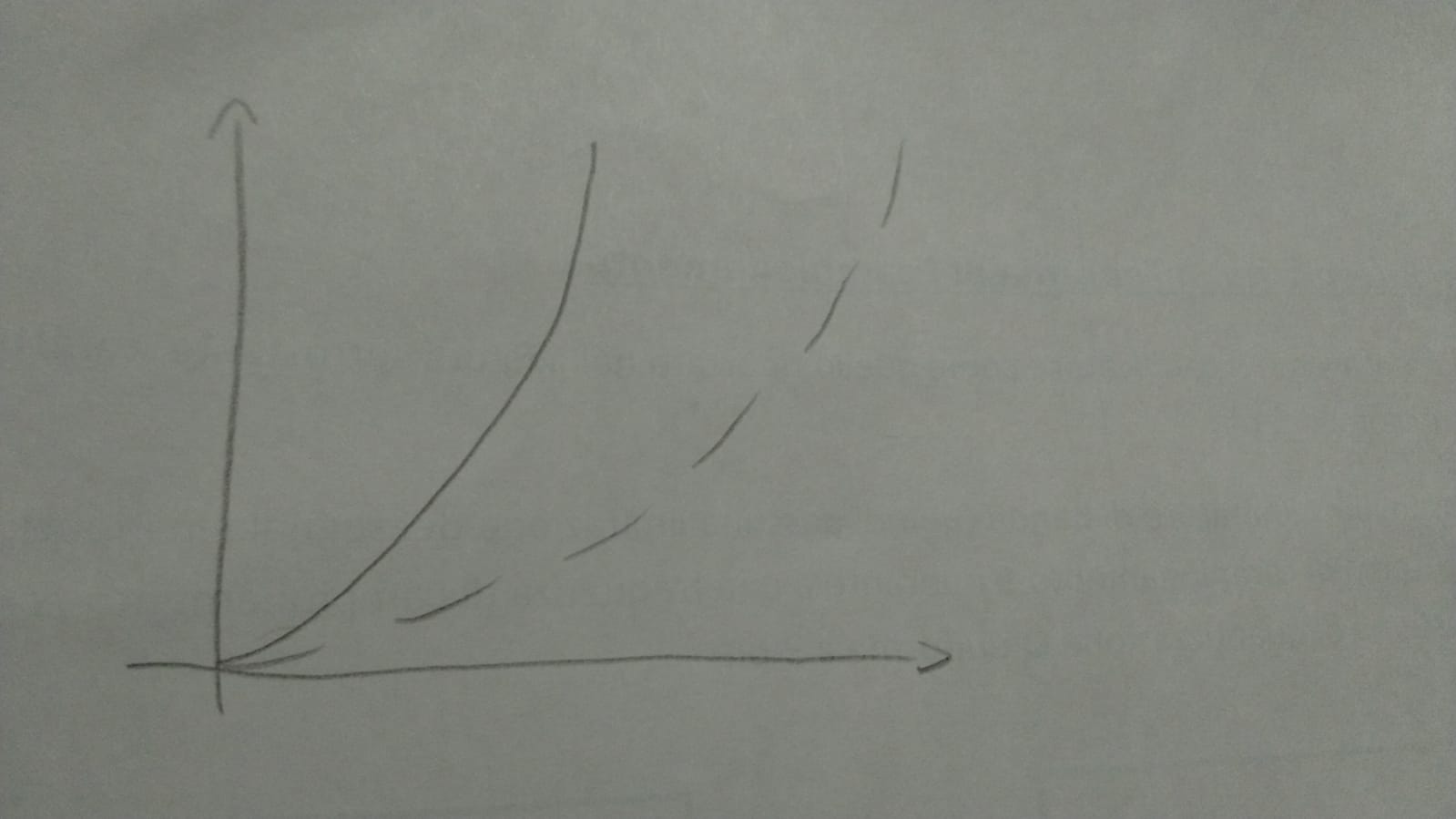
* Analisi lessicale: permette di riconoscere che le sequenze di caratteri di ogni variabile siano identificatori, dandone una descrizione attraverso l’espressione regolare lettera(lettera|cifra)\*, questa fase restituisce una sequenza di identificatori, simboli e cifre:

Input: position = initial + rate \* 60

Output: <id,1> <=> <id,2> <+> <id,3> <\*> <60>

I numeri vicino all’id indicano l’indice nella tabella dei simboli, che nel frattempo è stata compilata.

* Analisi sintattica: separata dalla prima per evitare dipendenze dal contesto, questa fase prende la sequenza di identificatori e simboli e crea l’albero di sintassi astratta, un albero di derivazione in cui i simboli delle operazioni hanno preso il posto dei non terminali che li hanno prodotti, esso segue le priorità degli operatori.
* Analisi semantica: in questa fase vengono fatte tutte le operazioni riguardanti i tipi, quindi si modifica l’albero per tenere conto di questi ultimi. Se ad esempio la variabile rate (corrispondente a id3) fosse float, c’è bisogno di una conversione in float del numero 60 per permettere la moltiplicazione. Inoltre, in questa fase è possibile scoprire errori riguardanti il tipo.
* Generazione del codice intermedio: Si genera del codice intermedia a partire dall’albero di sintassi, assomiglia a quello a basso livello senza però specificare i registri, permettendo quindi di riutilizzarlo meglio adattandolo alle architetture specifiche;
* Ottimizzatore: il codice intermedio viene ottimizzato in modo indipendente dall’architettura (è comunque possibile sfruttarla), il guadagno è dato dal fatto che è possibile mettere molteplici operazioni insieme, l’andamento della funzione rimane invariato: se ad esempio la funzione è di complessità O(n^2), si ha un caso di questo tipo:

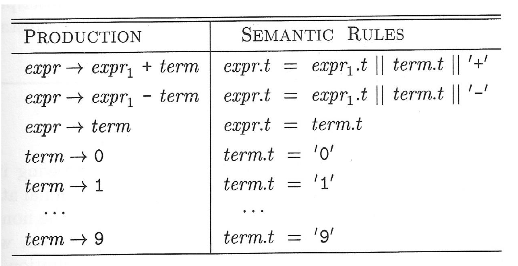
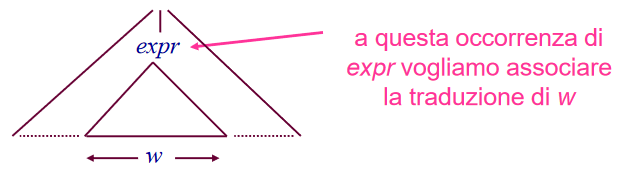


La linea indica l’algoritmo nel funzionamento normale mentre quella tratteggiata indica lo stesso algoritmo ma con le migliorie, quindi se k sono le migliorie applicate, si può dire che:

t(n)<=k\*n^2

t(n)<=k’\*n^2

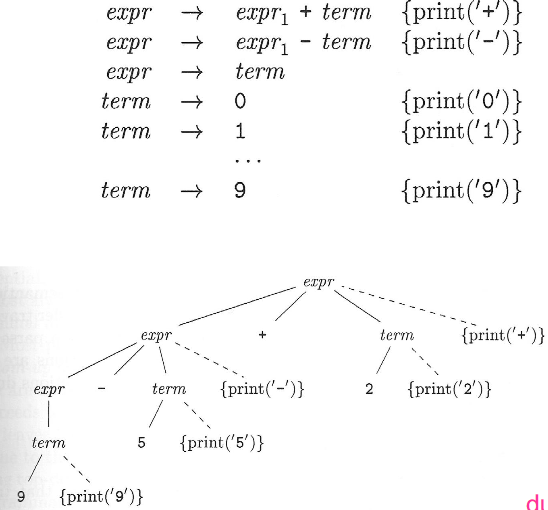
k’=0,7\*k

* Generazione del codice: generazione del codice macchina per la specifica architettura, rispetto al codice intermedio vengono utilizzati dei codici operativi specifici e i registri, questi ultimi contengono i risultati parziali delle operazioni, id1 infatti può essere riutilizzato di nuovo e quindi conviene registrare il suo valore in un registro per poi utilizzarlo in un secondo momento.

Esempio: Conversione da forma infissa a forma post-fissa.

Data la grammatica e l’operazione 9-5+2, come si può convertire in 95-2+?

Un primo metodo è calcolare il risultato concatenando sottoespressioni, per farlo si utilizzano i cosiddetti attributi, delle informazioni aggiuntive associate a ogni occorrenza del simbolo nell’albero di derivazione, essi indicano la traduzione dell’espressione generata dal corrispondente sottoalbero.

Come si può vedere dalla tabella, vi sono regole associate alle produzione, esse vengono applicate quando l’espressione genera una certa stringa, quella più a sinistra ad esempio genera 9-5, quindi verrà applicata la regola expr.t=expr1.t || term.t || “-”. In questo modo si hanno dati associati a ogni occorrenza, da essi verranno calcolati degli attributi dal basso verso l’alto, essi potrebbero essere necessari per la traduzione.

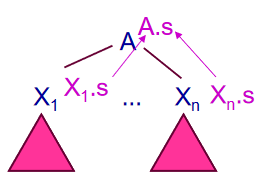
Un secondo metodo è utilizzare delle azioni da eseguire in un dato ordine rispetto alla struttura sintattica, solitamente si trovano alla destra della produzione. Considerando l’esempio precedente, prima viene stampato il carattere 9, poi il 5 e infine il -, questo perchè alla traduzione della produzione associata, ovvero expr→ expr-term, si traducono prima i due terminali.

## Azioni e attributi

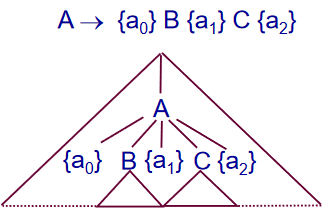
Le azioni sono infatti considerabili come nodi aggiuntivi dell’albero di derivazione, in base a dove sono piazzati, vengono eseguiti quando vengono incontrati durante la derivazione left-most.

L’ordine delle azioni è un aspetto delicato in quanto bisogna assicurare un corretto flusso di dati, con schemi di traduzione è possibile ricavarne uno, esso è relativo dato che si ha bisogno di un flusso di dati più sensato, inoltre bisogna considerare le eventuali dipendenze tra attributi. In ogni caso, le azioni possono essere eseguite:

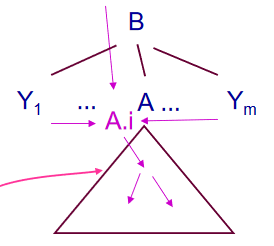
* dopo l’analisi sintattica, costruendo prima l’albero di derivazione, in tal caso l’ordine delle azione si ricava da un’elaborazione dell’albero stesso;
* durante l’analisi sintattica, qui l’ordine è dato dal punto in cui le azioni vengono eseguite e, se possono essere fatte, lo si fa a priori dato che non si può aspettare la visione di tutto l’input, rispetto a prima si ha il vantaggio di non memorizzare l’albero di derivazione.

Come già detto in precedenza, bisogna rispettare le possibili dipendenze tra gli attributi dei simboli, alcune di esse possono essere “tra fratelli” oppure “padre-figlio”, questi tipi di attributi vengono detti ereditati, tutti gli altri (ovvero quelli con dipendenze “figlio-padre”) sono detti sintetizzati. Per una distinzione più accurata, un attributo è sintetizzato quando il simbolo associato è a sinistra della produzione, se non è così allora è ereditato.

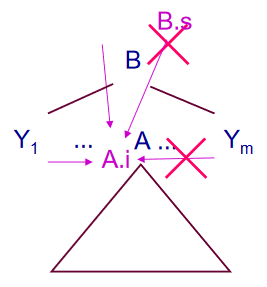
Esempio: data la grammatica e le regole semantiche, si considera l’input “int a,b,c;”, ricavando l’albero di derivazione come in figura.

Prima di tutto si visita il sottoalbero di T per prendere il tipo, in questo caso int, esso verrà passato a L con l’apposita regola, La lista di variabili ha tra le regole semantiche la funzione enter, permettendo così l’inserimento dell’id nella tabella dei simboli insieme al tipo. La definizione diretta della sintassi estende la grammatica context-free associando insiemi di attributi a ogni simbolo terminale e non, con essi si rappresenta ogni occorrenza dell’albero di derivazione, inoltre a ogni produzione può essere associato un insieme di azioni che prende come parametri gli attributi.

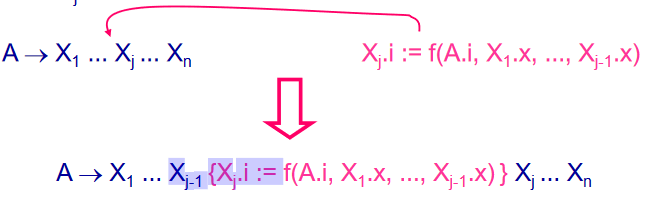
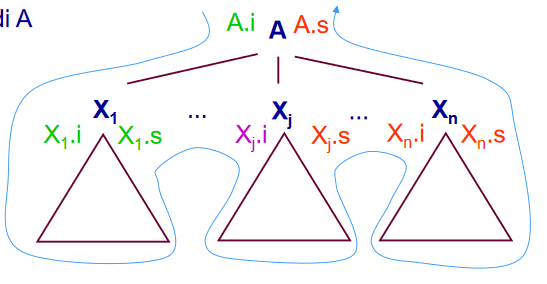
Ritornando a parlare di attributi, quelli sintetizzati rappresentano l’informazione che dipende da ciò che viene generato dal simbolo associato, quelli ereditati invece dipendono dal contesto in cui il simbolo occorre.

In generale gli attributi sintetizzati possono dipendere direttamente o meno da quelli ereditati e quindi anche dal contesto.

Esempio: data la grammatica e l’input “3\*5+4\n” si vuole ottenere il risultato, ovvero 19.

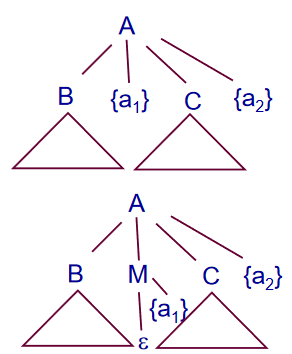
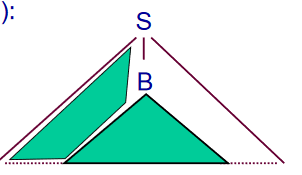
Gli attributi di T ed E saranno sintetizzati dato che prendono i risultato dal basso, cioè da quello che generano i due simboli, le azioni associate alla produzione riguardano la “conversione” del numero letto, il passaggio di quest’ultimo ai genitori e le operazioni. L’istruzione di stampa serve a stampare il numero ottenuto, per questo motivo viene fatta solamente quando si ritorna a L, il risultato è il seguente:

## Definizione S-attribuita

Una definizione è S-attribuita quando è composta da soli attributi sintetizzati, ogni ordinamento bottom-up va bene (l’albero ad esempio ne definisce uno parziale), ovvero che valuta gli attributi di un nodo dopo la valutazione di quelli dei figli. Per quanto riguarda la valutazione degli attributi, vi è un metodo generale: essa viene eseguita dopo l’analisi sintattica utilizzando il grafo delle dipendenze, un grafo in cui i nodi sono quelli dell’albero di derivazione mentre gli archi (orientati) rappresentano le dipendenze degli attributi rispetto alle azioni oppure dei padri rispetto ai figli.

Esempio: data la grammatica e le regole semantiche, si considera l’albero di derivazione.

il grafo si ricava guardando gli attributi, D non ne ha, T ne ha 1 così come id, entrambi non dipendono da nessuno dato che hanno assegnati dei valori costanti o provenienti dal parse, L ne ha due, L.in dipende dal T.type e da se stesso dato che bisogna passare ai figli il valore del padre, L.f dipende dall’input dell’azione, cioè id.name e L.in, l’albero decorato pu essere visto come un grafo delle dipendenze dato che alcuni valori dipendono da altri. Come per l’albero, il grafo delle dipendenze ha un ordinamento parziale, corrispondente a quello topologico, ogni ordine che rispetta il primo va bene, un modo per trovarlo è iniziare dai nodi senza archi entranti e continuare finchè i nodi visitati non hanno più archi uscenti.

Esempi: Per valutare gli attributi durante l’analisi sintattica, si suppone di avere già l’albero, è possibile rivedere le operazioni di shift e reduce semplicemente attraversando i sottoalberi nell’ordine leftmost e infine visitando il nodo interno, per farlo si effettua ricorsivamente l’attraversamento di un nodo finché non ha figli, in tal caso si visita. La sequenza di operazioni shift-reduce fatta in questo modo equivale a quella effettuata in bottom-up, ad esempio: 3\*5+4

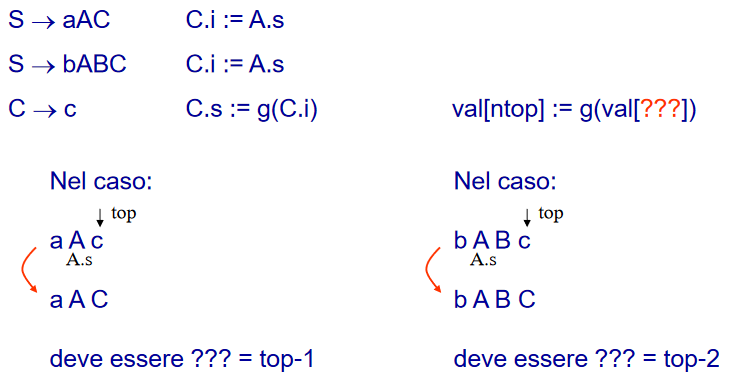
Data la grammatica e l’albero, il primo nodo da visitare è 3, la foglia più a sinistra, si procede poi verso destra fino alla fine, l’ordine è quindi il seguente:

S(3), R(F← 3), R(T← F), S(\*), S(5),...,R(E← E+T);

Questo ordine,è quello delle operazioni di analisi sintattica in base all’albero di derivazione, si può notare che esso corrisponde a quello della visita leftmost dell’albero.

Si suppone ora di avere una derivazione S-attribuita del tipo A→ X1,...,Xn con l’azione A.s:=f(X1.s,...,Xn.s) associata, l’idea è quella di effettuare l’azione quando la produzione viene ridotta, questo funziona perchè in questo modo tutti gli attributi s dei figli sono già disponibili, quindi la riduzione sul loro simbolo è già stata effettuata. Una cosa che è possibile notare è che l’ordine delle riduzioni equivale a quello di visita dell’albero e a quello di calcolo degli attributi, questo rende inutile il grafo delle dipendenze e lo stesso albero perchè già specificati in tale ordinamento.

## Memorizzare gli attributi sintetizzati

Gli attributi sintetizzati si memorizzano in una seconda pila che lavora parallelamente alla prima, a ogni elemento di uno stack corrisponde un elemento dell’altro nella stessa posizione, ad esempio: alla riduzione di A→ X1,...,Xn, si ha una situazione come la seguente:

I valori di X1,...,Xn stanno nell’intervallo di posizioni top, top-n+1, questa informazioni servono per calcolare la nuova posizione della cima della pila dopo la riduzione, è quindi possibile dire che l’azione A.s:=f(X1.s,...,Xn.s) si può scrivere come val[ntop]:=f(val[top-n+1],...,val[top]) con ntop:=top-n+1

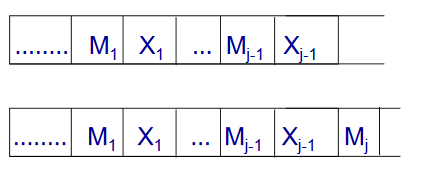
La gestione di top e ntop è tuttavia sempre uguale, di conseguenza è possibile non scriverlo in modo esplicito.

## 

## Ordine con gli schemi di traduzione

Un’ulteriore modo di trovare un ordinamento delle azioni è con gli schemi di traduzione, essi specificano l’ordine nel seguente modo (quando fatte dopo l’analisi sintattica): all’albero vengono aggiunti i nodi azioni, l’ordine sarà quindi dato dalla normale visita dell’albero. Dato che vi è la corrispondenza analisi sintattica-visita dell’albero trattata in precedenza, lo schema di traduzione è interpretabile come una specifica non solo nell’ordine della azioni ma anche del momento in cui eseguirle rispetto all’analisi sintattica.

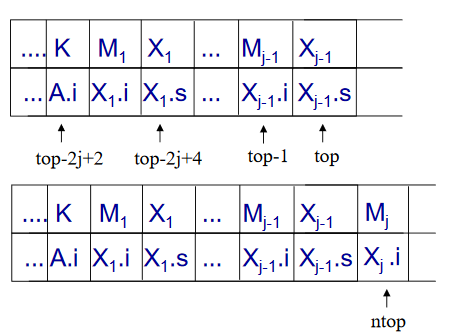
Esempio: si considera la produzione A→ {a0}B{a1}C{a2}



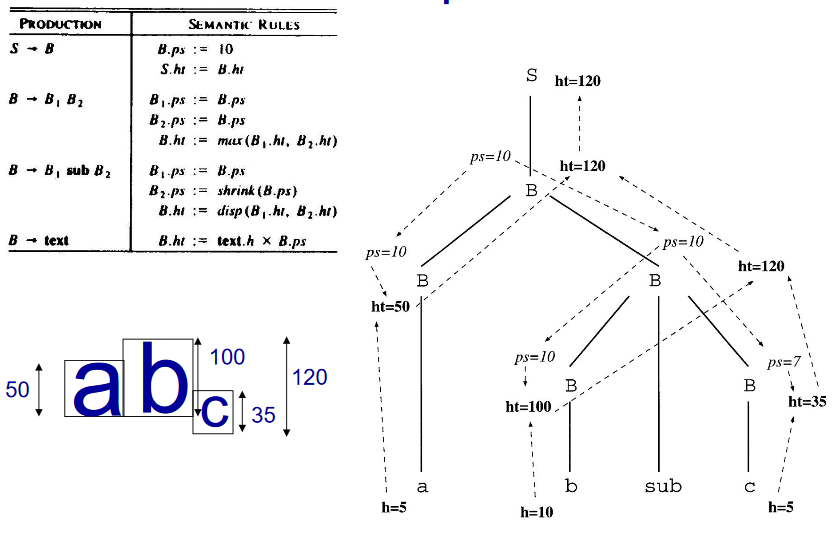
La prima azione che verrà eseguita sarà a0 e verrà effettuata prima di analizzare il sottoalbero di B, tra quest’ultimo e C si effettua a1 mentre l’ultima è a2, quest’ultima sarà al termine dell’analisi di A, ovvero quando BC verrà ridotta.

Da ciò si ottiene un metodo generale per convertir definizioni S-attribuite in schemi di traduzione: dal momento che si hanno solamente azioni che calcolano attributi sintetizzati, esse possono essere messe al fondo della parte destra della produzione, ad esempio: data la produzione A→ X1,...,Xn con A.s:=f(X1,...,Xn.s) come azione associata, il risultato che si ottiene è A→ X1,...,Xn{A.s:=f(X1,...,Xn.s)}, questo ordinamento va bene sia dopo l’analisi sintattica, sia durante.

## Definizione L-attribuita

Per quanto riguarda gli attributi ereditati di un simbolo A, esso dipende da tutti quelli dei simboli in cui A occorre alla destra della produzione e verranno utilizzati per il calcolo degli ereditati dei figli di A. Per effettuare queste operazioni durante l’analisi sintattica, si vorrebbero che questi calcoli vengano effettuati all’interno del sottoalbero di A, di conseguenza gli ereditati devono essere calcolati prima. Da ciò si può quindi dire che, per ogni figlio di un simbolo X, vengono calcolati gli attributi ereditati del figlio Xi e si controlla il suo sottoalbero, alla fine si calcolano i sintetizzati.

Dal momento che si calcolano gli ereditati prima dell’analisi sintattica, l’ereditato calcolato non deve dipendere né da ereditati dei fratelli a sinistra, né da sintetizzati del padre, il motivo è dato dal fatto che questi attributi vengono calcolati solo in seguito. per risolvere questo bisogna limitare l’utilizzo degli ereditati a quelli già calcolati, ovvero quelli dei fratelli a sinistra, per rappresentare solo l’informazione che dipende dal contesto che precede il simbolo interessato.

Un definizione diretta della sintassi che rispetta i precedenti limiti è detta L-attribuita, precisamente se ogni attributo è sintetizzato se dipende da un’azione associata a una produzione A→ X1,...,Xn oppure è ereditato se dipende dagli ereditati del padre oppure dei fratelli precedenti. Un modo generare per convertire la dds in uno schema di traduzione è il seguente: un’azione che calcola un ereditato di Xj verrà piazzata prima del simbolo in questione e non dovrà riferirsi ad altri ereditati che occorrono a sinistra, se invece calcola un sintetizzato, l’azione può essere piazzata in fondo:

Data la grammatica e le azioni associate, si utilizza sub per mettere una lettere come pedice di quella alla sua sinistra, un pedice è caratterizzato dall’essere più piccolo e più in basso rispetto alla baseline.

L’attributo h indica l’altezza del carattere senza considerare il font, ht è l’altezza totale dal carattere più in basso a quello più alto mentre ps (da cui dipende) è l’altezza del font, shrink e disp sono due azioni che permettono rispettivamente di ridurre il font e di abbassarlo rispetto alla baseline. Come si vede dall’albero, è possibile dire che ps è ereditato (dato che dipende dal genitore) e ha una valore costante che cambia con le azioni, ht è sintetizzato e dipende dal primo.

## Valutazione degli ereditati

Per sincronizzare le azioni che calcolano gli ereditati con l’analisi sintattica, se si ha una produzione A→ B{a1}C{a2}, è possibile ricondurre le azioni modificando la grammatica inserendo i cosiddetti marker, dei simboli non terminali che generano la stringa vuota a cui si assegna l’azione, in questo modo il linguaggio generato non cambia ma le azioni sono diventate foglie, quindi vengono effettuate alla riduzione, ad esempio:

A→ BMC{a2}

M→ “stringa vuota”{a1}

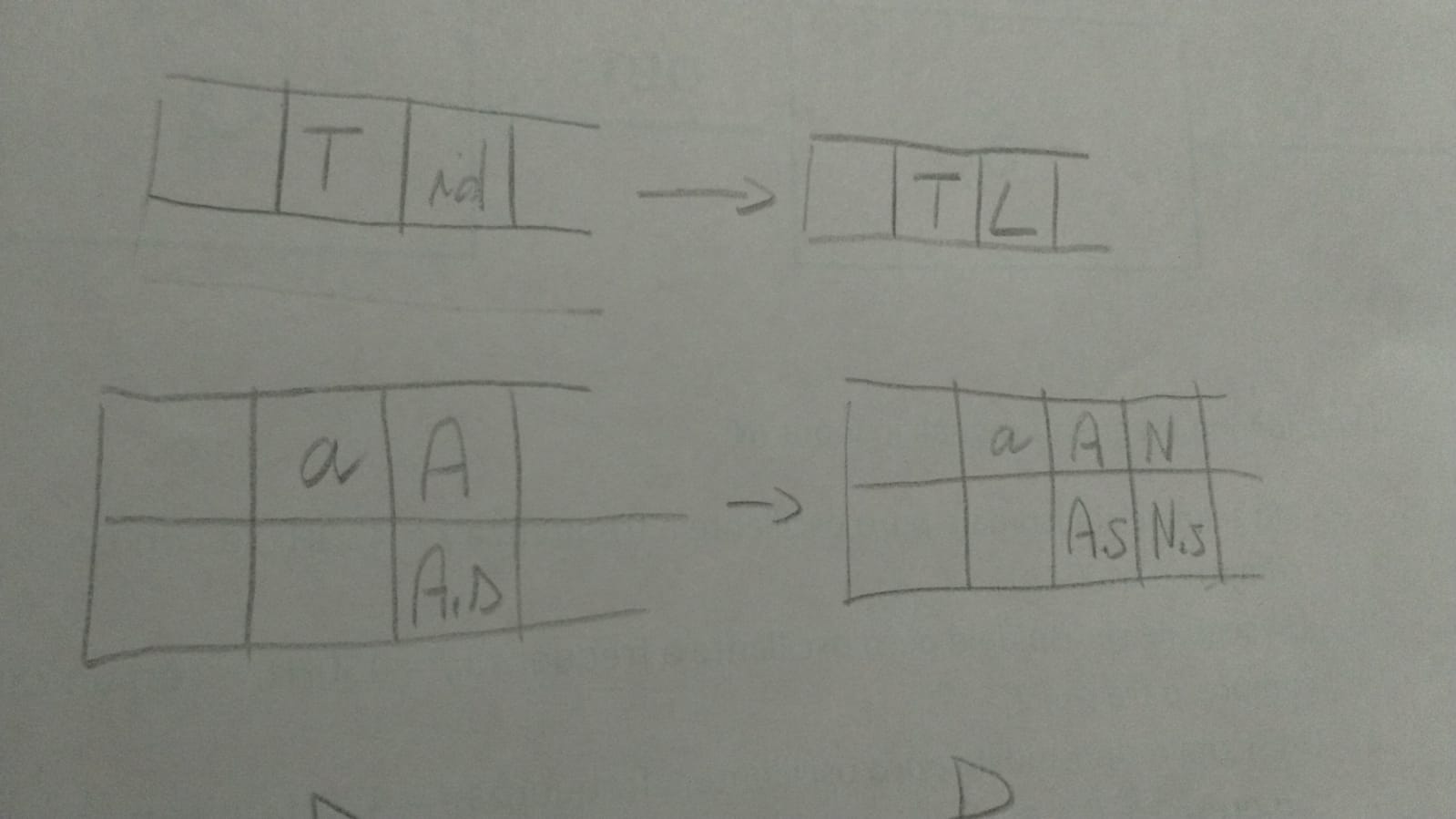
In questo esempio prima si svolgono le azioni di B e del suo sottoalbero, dopo si fa la riduzione di M (e quindi la sua azione )e infine si eseguono le azioni di C, del suo sottoalbero e a2 alla riduzione di A. L’analizzatore sintattico viene quindi esteso con le azioni, esse avvengono in contemporanea alla riduzione del simbolo associato.

Per quanto riguarda la memorizzazione degli ereditati, essi non vengono mantenuti in modo esplicito quanto sono una copia dei sintetizzati, quando serve il loro valore infatti si sa dove andarli a prendere.

Esempio: Data la grammatica, le regole e lo schema di traduzione, si ha che L eredita da T il tipo, contrassegnato dall’attributo T.type, esso verrà ereditato anche da tutti i figli del primo simbolo. Considerando int p,q,r:

* alla riduzione di int in T, il tipo viene passato al simbolo L;
* l’attributo L.in viene copiato e inserito nella tabella dei simboli quando le tre variabili vengono ridotte.

L’ulteriore azione che viene effettuata è la riduzione di L,id in L e di id in L, entrambe utilizzano l’atributo L.in rispettivamente passando T.type nel primo e L.in del padre nel secondo. Quando si porta T sulla pila, T.type viene inserito in quella “parallela” e, come si può notare dall’esempio, vi sono meno azioni, l’idea infatti è quella di pescare l’attributo dietamente dalla pila, precisamente dalla sua cima, considerando le produzioni L→ L,id e L→ id, si hanno delle pile come le seguenti:



Nel primo caso si pesca il tipo dalla posizione top-1 dato che T.type si trova una posizione prima della cima, la cosa è analoga anche nel secondo caso dove si pesca l’attributo nella posizione top-3, questo però è un caso particolare in cui T e L sono adiacenti, se così non fosse, questo metodo non funzionerebbe.

Per cercare un altro metodo di memorizzazione degli attributi, si considera il seguente esempio:

S→ aAC C.i=A.s

S-->bABC C.i=A.s

C→ c C.s=g(C.i) val[ntop] := g(val[????])

dove si va a prendere l’ereditato C.i?

Considerando gli alberi delle due produzioni, si ha che nel primo caso A è alla sinistra di C dopo la riduzione di c, la posizione è quindi top-1, nel secondo caso invece è top-2 dato che vi è B in mezzo. Per evitare di avere più posizioni, si generalizza il tutto utilizzando i marker, la prima produzione rimarrà invariata mentre la seconda avrà un marker M tra B e C che permette il recupero dell’ereditato alla posizione top-1, questo perchè l’attributo viene copiato prima dell’inserimento di M, la grammatica diventa quindi la seguente:

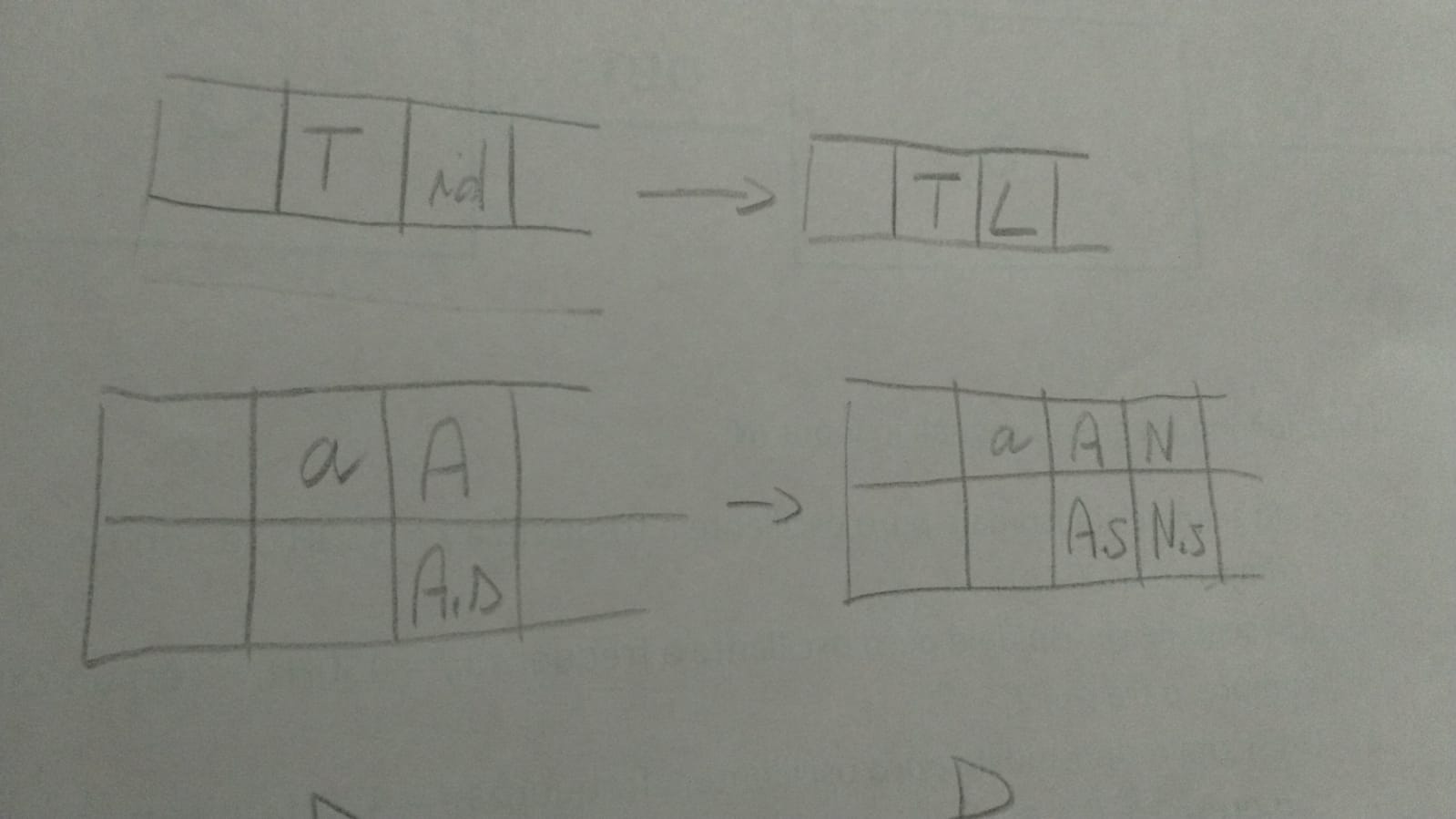
S→ aAC

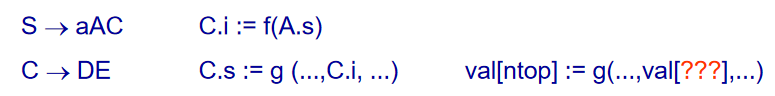
S→ bABMC

C→ c val[ntop] := g(val[top-1])

M→ “stringa vuota” val[ntop] := val[top-1]

Vi sono però casi in cui gli ereditati non sono copie di sintetizzati, in tal caso si utilizzano sempre i marker per tenere gli attributi ereditati di un altro simbolo, ad esempio:

Data la grammatica e le azioni associate, l’attributo C.i non risulta presente sulla pila, per risolvere ciò si inserisce un marker N prima del simbolo C, in questo modo C.i verrà realizzato attraverso N.s. Con S→ aANC si ha un punto della pila in cui sono presenti a e A, quando verrà ridotto N, il valore di A.s verrà passato a N.s, quest’ultimo verrà preso da C quando verrà ridotta la sua produzione.



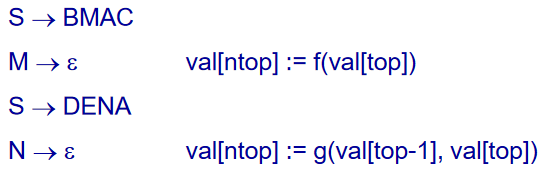
Dopo questi esempi, è ora possibile dare una descrizione generale per l’implementazione di uno a più marker per un simbolo A:

* Si introducono i marker subito prima di A nelle produzioni in cui occorre a destra;
* Nelle produzioni associate ai marker (i quali generano la stringa vuota) si fa in modo che vi sia l’ereditato di A in loro corrispondenza sulla pila.

Generalmente c’è bisogno di un marker per ogni produzione in cui A è a destra dato che l’azione da associare può essere diversa, l’ereditato di A sarà quindi rappresentato dal sintetizzato del marker, ovvero che sulla pila degli attributi il valore del marker sarà appunto quello che deve avere A.i.

Esempio: data la grammatica <slide 9> e le azione associate, vi sono due regole in cui A occorre a destra, quindi si aggiungono due marker M e N, dalle azioni associate si può capire che M prende in input il sintetizzato di B (che in quel momento è in cima alla pila)

mentre N prende quelli di D ed E (corrispondenti alle posizioni top-1 e top). La grammatica modificata sarà quindi la seguente:

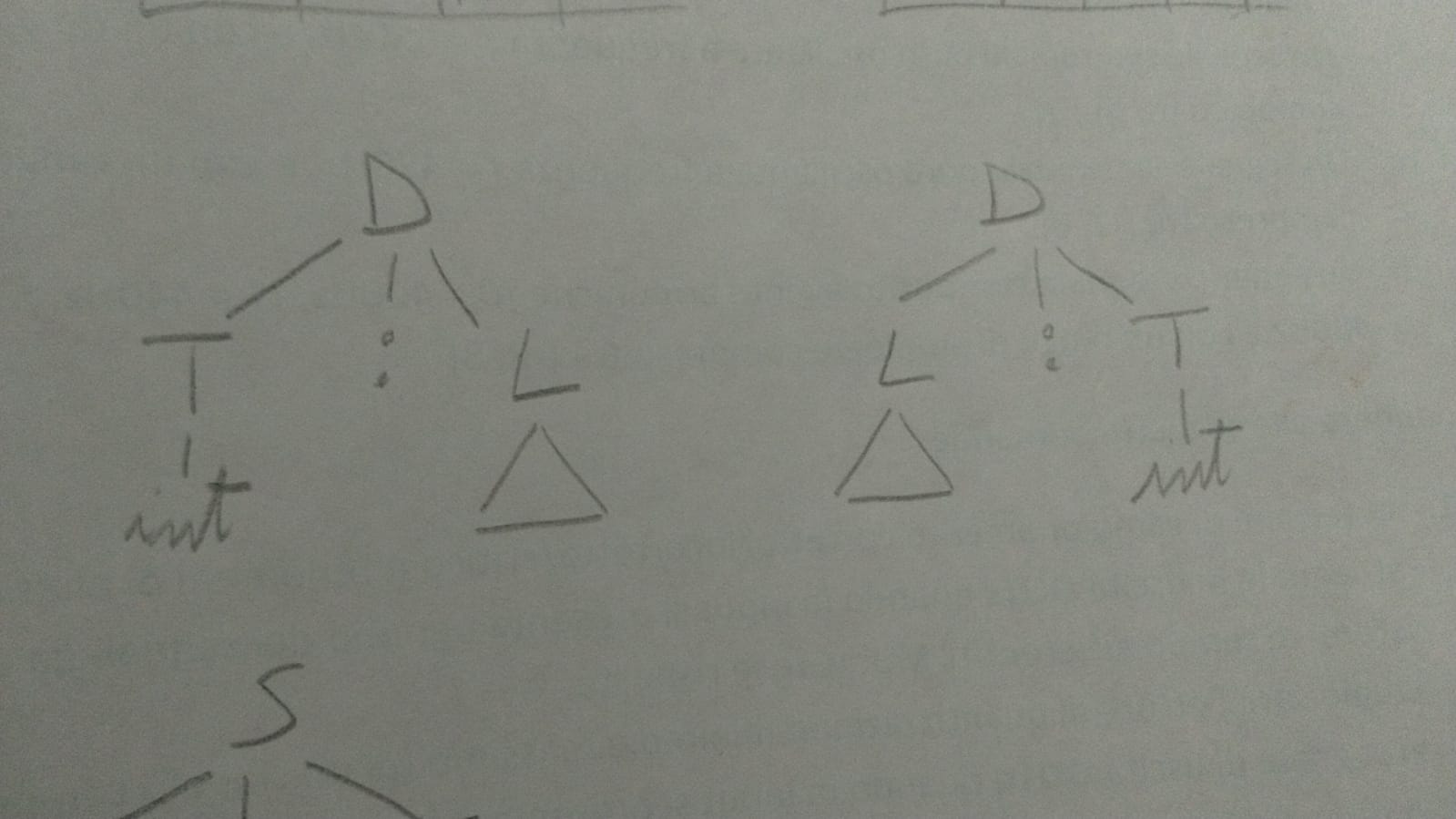
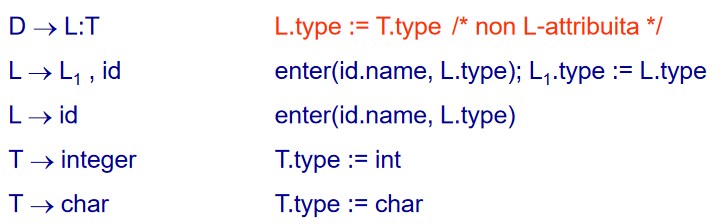


Non è sempre necessario aggiungere marker, infatti non lo si fa dove non ci sono ereditati in un simbolo oppure se l’ereditato di un simbolo è una copia di quello del fratello analizzato in precedenza, queste semplificazioni permettono di evitare conflitti di analisi sintattica, essi sono presenti quando un marker precede una produzione ricorsiva a sinistra. In parole semplici, i marker si utilizzano per:

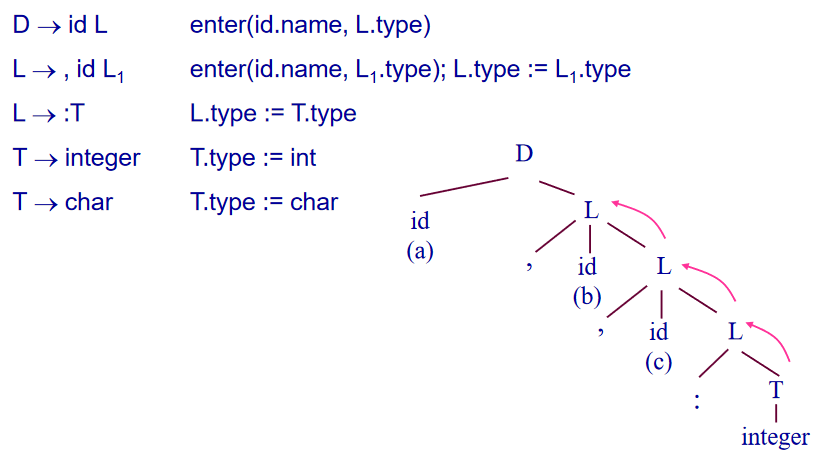
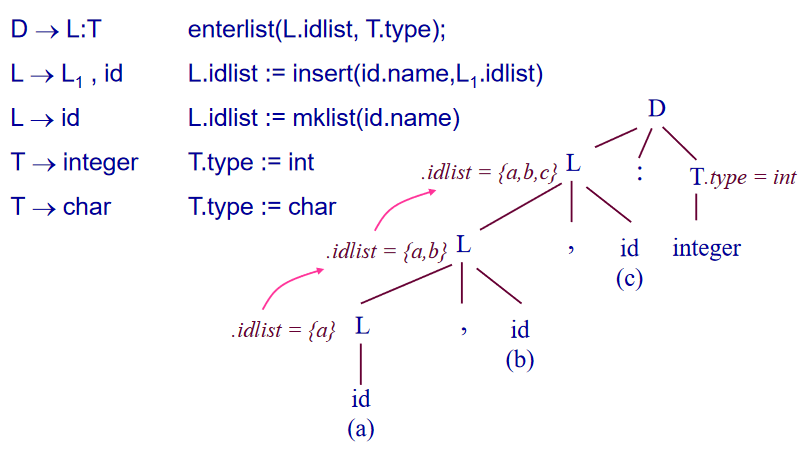
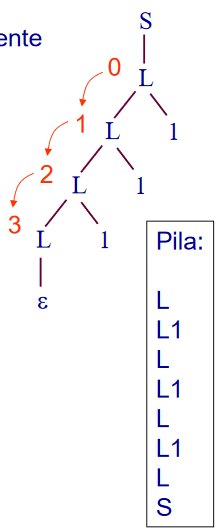
* eseguire azioni dell’analisi sintattica al momento opportuno ma sempre in corrispondenza di una riduzione;
* tenere un posto sulla pila degli attributi.

## Esempi problematici con dds non L-attribuite

In alcuni casi la dds è una definizione naturale non L-attribuita, ad esempio: data la grammatica e le azioni associate, si considerano i seguenti alberi di derivazione:



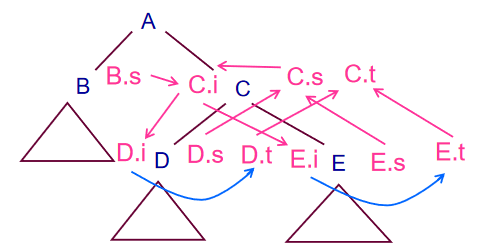
Da questi alberi è possibile notare che la prima azione (il passaggio dell’ereditato) funziona solamente nel primo albero dato che, per inserire un elemento nella tabella dei simboli, è necessario conoscere il suo tipo, di conseguenza non si può imporre ciò prima della lettura del tipo. Per risolvere questo tipo di problemi, un primo approccio è quello di cambiare la grammatica, c’è tuttavia il rischio di rendere la grammatica poco intuitiva:

Come si può vedere dall’esempio, la grammatica è stata resta associativa a destra, vengono infatti generata una lista di id che verrà appunto tappata dal tipo. Alla riduzione di T si inizia a “risalire” la lista inserendo ogni elemento nella tabella dei simbolo. Un secondo approccio consiste nel mantenere la stessa grammatica e utilizza un sintetizzato per memorizzare la lista di id, in tal caso alla riduzione di T si utilizza un’apposita azione (nell’esempio è insert()): Per rendere tutto questo esempio possibile, la dss è stata trasformata in S-attribuita, quando è L-attribuita è possibile realizzarla nel modo previsto? Dal momento che si vogliono escludere le dipendenze da destra verso sinistra (per definizione) e quella dall’alto al basso servono molto, si considera il seguente esempio:

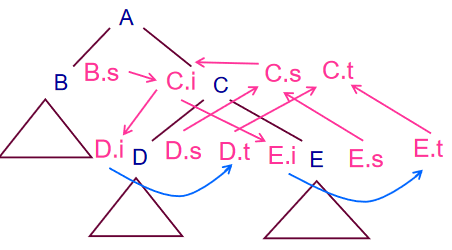
Data la grammatica, le azioni associate, l’albero e la pila, si ha L→ “stringa vuota” come prima riduzione, di conseguenza la prima azione eseguita sarà la stampa di L.c, tuttavia non sarà giusto dal momento la sua produzione non è ridotta e quindi il suo valore non è aggiornato. Inserendo dei marker prima del simboli e sostituendo le azioni, si ottiene l’effetto desiderato, tuttavia ciò non risolve il problema dato che vi sono conflitti di analisi sintattica quando vengono introdotti marker prima di simboli con produzioni ricorsive, in questo caso il prossimo simbolo potrebbe essere L (denotando la fine dell’input) oppure tante N quant’è lungo l’input. Per chiudere correttamente l’analisi sintattica correttamente, c’è bisogno di tante N quanti sono gli 1 in input, cosa che non si può sapere subito a meno di non leggerlo tutto, si può quindi concludere che questa è una di un sottoinsieme di dds L-attribuite in cui non è possibile effettuare le azioni durante il parsing.

Per effettuare la traduzione si può quindi costruire l’albero e il grafo delle dipendenze, calcolare l’ordine delle azioni rispetto all’input per poi eseguirle oppure (solo per dds S e L attribuite) eseguire queste ultime durante l’analisi sintattica, quest’ultimo modo non è sempre possibile da effettuare, in tal caso si utilizza la prima tecnica anche se serve un ordine per ogni input. Spesso è possibile prefissare un ordine in base alle produzioni applicate senza visitare tutto l’albero nè costruire il grafo delle dipendenze, come nell’esempio: data la grammatica e le azioni associate, si considerano le due produzione e si suppone che dall’ereditato di un simbolo si riesca a ricavare il sintetizzato si un altro, dopo si attraversa l’albero e in base alla produzione si sceglie l’azione:

* se la produzione è A→ LM allora si passa l’ereditato al figlio sinistro e si visita per calcolarne il sintetizzato, quest’ultimo verrà poi passato ai fratelli per calcolare i rispettivi sintetizzati;
* se invece la produzione è A→ QR, si fa la stessa cosa ma al contrario.

Non è tuttavia possibile definire una funzione di attraversamento univoca, infatti ne serve una per coppia simbolo-sintetizzato, chiamandole in un ordine opportuno. Questo metodo funziona se esiste un ordine parziale per gli attributi di un simbolo in modo che: per ogni produzione A→ X1,..,Xn:

* il grafo locale arricchito con gli archi dell’ordine sia aciclico;
* se esiste un cammino da b a c di uno stesso simbolo, allora il primo precede il secondo nell’ordine.

L’aciclicità del grafo viene verificato solo localmente alla produzione in quanto l’ordinamento tra attributi deve riflettere le dipendenze “nascoste” tra due simboli, ovvero quelle che corrispondono all frecce azzurre come nell’esempio. Se si avesse una dipendenza diretta D→ … con associata l’azione D.t=f(...,D.i,...), allora D.i precede D.t come dice la seconda regola, per C→ DE si ha quindi un cammino C.i-C.t nel grafo, quindi C.i precede C.t per lo stesso motivo di prima.

Se la precedente condizione vale, allora la dds è detta fortemente non circolare, quindi è possible definire funzioni per calcolare i sintetizzati di ogni simbolo utilizzando gli ereditati come parametri e chiamandoli rispettando l’ordine. Per esempio: dato il grafo locale di C→ DE, esso è aciclico, inoltre vi sono cammini da C.i a C.t, questo primo attributo viene calcolato col grafo locale di A→ BC, anch’esso acicliclo:



Da ciò è possibile dichiarare le seguenti funzioni:

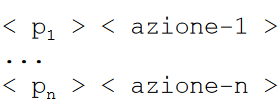
* Adummy(nodo): calcola l’attributo fittizio di A, in base alla produzione calcola i sintetizzati con le funzioni di attraversamento e gli ereditati;
* Cs(nodo): in base alla produzione, esegue l’attraversamento della parte destra della produzione, calcolando i sintetizzati, e restituendo quello della radice;
* Ct(nodo,in): fa la stessa cosa di Cs con la differenza che calcola gli ereditati.

Per trovare un ordine parziale che rispetta le regole, bisogna associare a ogni produzione un grafo locale (non arricchito) e un ordinamento senza precedenze, dopo si iterano le produzioni in cui si arricchisce il grafo con un arco per ogni coppia di attributi in cui è presente un cammino, se questo genera un ciclo, si esce con un risultato negativo. In caso di aciclicità, si può scrivere una funzione di Aa per calcolare il suo sintetizzato in cui, in base alla produzione, vengono calcolati gli attributi non presenti tra i parametri che precedono A.a nel grafo arricchito, se è ereditato, viene calcolato con l’azione della dds, altrimenti si usa l’apposita funzione.

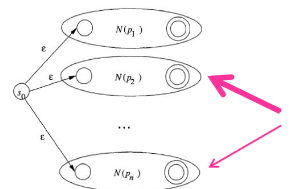
Un caso in cui non è possibile utilizzare il grafo delle dipendenze è il seguente: data la grammatica e le azioni associate, per la produzione A→ a, si ha un grafo aciclico come il primo, per A→ b invece si ha come risultato il secondo: qui non è possibile stabilire un ordine tra A.s e A.t, di conseguenza non è possibile indicare chi ha la precedenza. Per le dipendenze per le produzioni di A, A.j va inserito prima di A.s nell’ordine, la stessa cosa vale per A.i con A.t, questo forma un ciclo anche se in quelli singoli non è presente: il ciclo contiene quindi due dipendenze nascoste che non si verificano per uno stesso input.

## Lex & Yacc

Lex è uno strumento che genera un programma che simula automi a stati finiti eseguendo le azioni corrispondenti, il tutto viene specificato in un apposito file sorgente indicando il una riga la produzione e la rispettiva azione:



Dopo aver compilato il file.c generato, è possibile riconoscere ripetutamente e occorrenze di un pattern eseguendo le azioni corrispondenti. Il riconoscimento va avanti finché è possibile, questa cosa serve spesso dato che, come nel caso degli id e dei numeri, i prefissi sono occorrenza dello stesso pattern:

Nel caso in cui una stringa appartenga a più pattern, viene eseguita l’azione corrispondente al primo, questo serve per le parole riservate come while e if che rispettano la definizione di identificatore, per questo motivo vengono messe prima di questo tipo di pattern.

Yacc genera un programma da uno schema di traduzione, esso riconosce il linguaggio generato dalla grammatica data utilizzando un’analisi sintattica bottom-up LALR(1), eseguendo le azioni, questo strumento si può utilizzare insieme a lex come generatore di analizzatori lessicali, in tal caso lex esegue delle return per restituire il simbolo riconosciuto, questi verranno inseriti in una funzione nel file lex.yy.c per il riconoscimento, essa verrà chiamata in y.tab.c quando serve conoscere il prossimo. Con yacc è possibile utilizzare anche grammatiche ambigue, per risolvere i conflitti si dà priorità alla riduzione mentre, nel caso di conflitti reduce-reduce, si riduce la prima produzione in elenco, un’altro metodo per risolverle è dichiarare delle precedenze tra operatori:

%left '+' '-' e %left '\*' '/' dichiarano che gli + e - sono associativi a sinistra, anche \* e / lo sono e inoltre hanno un precedenza rispetto a chi si trova nelle righe precedenti, in questi casi il conflitto avviene tra due operatori della stessa riga che viene risolto a favore della riduzione, con %right si ottiene l’associatività a destra e il conflitto analogo che si genera viene risolto favorendo lo spostamento. Per quanto riguarda i conflitti tra operatori con priorità differenti, come \* e / con + e -; la riduzione dei primi è favorita allo spostamento dei secondi, al contrario si esegue uno spostamento quando bisogna decidere se spostare \* o / o ridurre + o -. Nelle azioni in yacc si utilizza $1, S2, eccetera, essi rappresentano gli attributi dei simboli nella parte destra della produzione, $$ invece rappresenta il simbolo nella parte sinistra ed è utilizzato nella azioni in fondo alle produzioni, questi simboli verranno convertiti da yacc in riferimenti rispetti alla cima della pila.Le azioni all’inizio o in mezzo alla produzione vengono trattate utilizzando dei marker, se in una di esse calcola un $$, esso si riferirà all’attributo del marker.

Esempio:

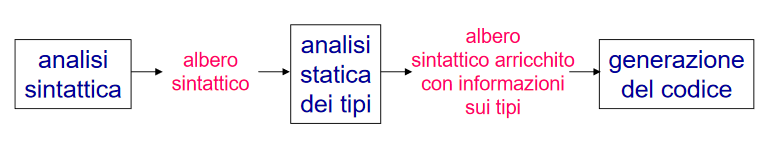
A→ B {$$ = …} C

C→ X1...Xn {...}

in tal caso, si utilizza $$ per il calcolo del marker nella produzione di A, inoltre ci si riferisce a esso utilizzando $0 e utilizzando nuovamente $$ per calcolare l’attributo di C, questo approccio è utilizzato per continuità, ovvero per trasformare allo stesso modo i vari simboli in riferimenti dalla cima della pila.

## Analisi dei tipi

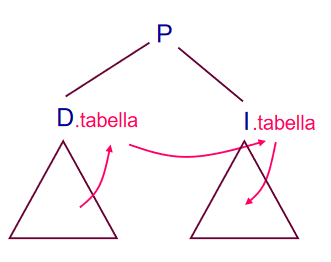
Con analisi dei tipi si intende se questi vengono utilizzati nel modo consono a quello dettato, essa viene utilizzata per verificare alcuni errori di programmazioni (ad esempio le variabili utilizzate in modo differente da quello dichiarato) e in alcuni casi per generare la traduzione corretta dell’istruzione, queste due mansioni fanno parte della cosiddetta analisi statica, ovvero quella fatta al momento della traduzione, quella dinamica è invece fatta durante l’esecuzione del programma dato che alcune verifiche non possono essere effettuate a priori a tempo di compilazione. Come si può vedere dall’immagine, l’analisi dei tipi avviene dopo l’analisi sintattica e arricchisce l’albero di derivazione con informazioni riguardanti essi, è anche possibile eseguire questa fase anche durante quella precedente.



Le azioni di analisi dei tipi vengono aggiunte alle produzioni e integrate alla traduzione, le verifiche che interessano sono:

* La verifica degli operatori, specie quelli compatibili con più tipi;
* la verifica del deferenziamento, esso deve essere applicato a un puntatore;
* la verifica dell’indice, applicabile solamente agli array, essa non può essere eseguita staticamente dato che non si ha la certezza che questo sia nel range richiesto, per verificarlo bisogna generare codice a tempo di esecuzione;
* la verifica delle funzioni, esse devono essere dichiarate in modo corretto, che venga restituito il tipo giusto e che il numero di argomenti sia corretto e coerente i tipi dati.

L’analisi dei tipi è fatta con azioni durante l’analisi sintattica, tuttavia i possibili errori presenti non sono di sintassi,ovvero che il parser si trova in uno stato in cui non vi sono operazioni da fare col l’input dato, quindi conclude che la stringa data non appartiene al linguaggio. Questi programmi con questi errori sono comunque generati dalla grammatica, per evitarli si dovrebbero utilizzare grammatiche context-free e quindi non è possibile farlo dato che servirebbero analizzatori sintattici più complessi. Con le grammatiche context-free è possibile utilizzare ogni tipo di identificatore, anche quelli non compatibili con l’operatore dato, che sia dichiarato o meno, tutto questo si fa per semplificare il tutto, l’idea è quella di riempire la tabella dei simboli durante l’analisi sintattica e, una volta che si hanno le istruzioni, si analizzano per verificarne il corretto funzionamento.

La tabella dei simboli è una struttura dati globale, in realtà può essere un attributo sintetizzato di un simbolo e passato come ereditato al fratello, tradizionalmente però la prima soluzione è quella più utilizzata.

Dentro la tabella dei simboli si memorizzano nome e tipo della variabile, vi sono inoltre ulteriori informazioni riguardanti la traduzione: il nome è una stringa, tipo conterrà la struttura dati che lo descrive, essa può essere un tipo base, un tipo enumerato oppure un tipo strutturato, in quest’ultimo bisogna indicare ulteriormente il tipo dei campi, per questo motivo i tipi strutturati sono ricorsivi. Per descrivere al meglio gli schemi di traduzione, si utilizzano le espressioni per denotarli in modo astratto e conciso: per i tipi base si utilizza semplicemente il nome, per quanto riguarda quelli strutturati, invece, l’espressione sarà data dal costruttore applicato ai tipi componenti, ad esempio:

Array di array di interi → array(10,array(10,integer));

Si può quindi dire che l’espressione di tipo è un tipo a sua volta.

Il costruttore pointer indica il puntatore ed è anch’esso combinabile con altri tipi:

puntatore a intero → pointer(integer);

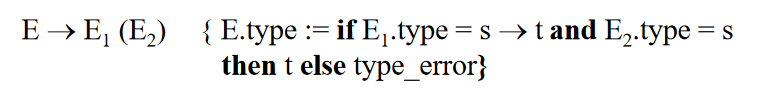
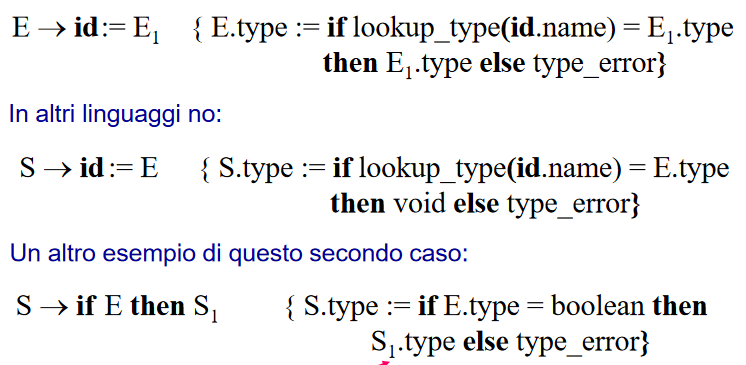
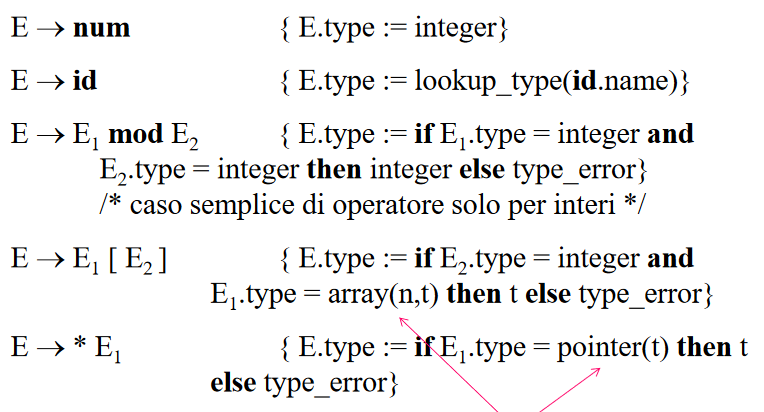
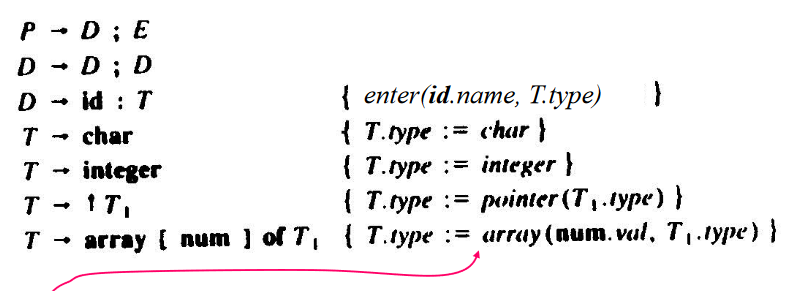
Per quanto riguarda le funzioni, si procede nel seguente modo: si utilizza il prodotto cartesiano per indicare il tipo dell’ennupla di argomenti, la freccia invece indica la funzione vera e propria.

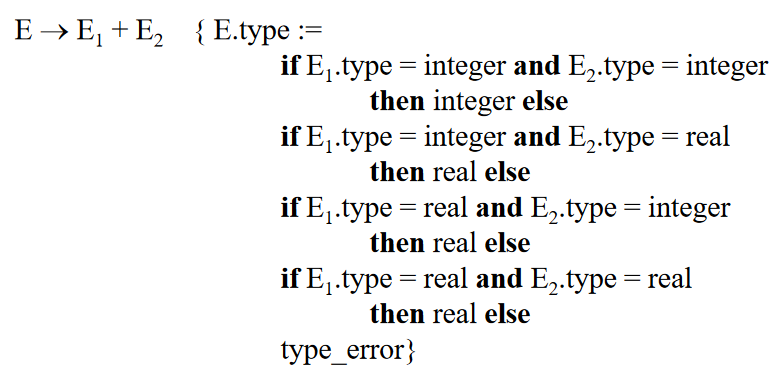
int → int: dato un intero, viene restituito un intero;

int x int → int: dati due interi, viene restituito un intero;

Esempio: data la grammatica e le azioni di tipo associate, ogni D genera una produzione id:T dove T indica il tipo della variabile e com può essere indicato, se è base si mette solo il nome, altrimenti si assegnano le espressioni trattate in precedenza. Le espressioni E possono essere numeri (l tipo sarà quello considerato per esso), un id (il suo tipo sarà preso dalla tabella dei simboli cercando in base al nome) oppure un’operazione, nel caso del resto, supponendo di avere solo tipi interi, il tipo sarà di E sarà intero, altrimenti ci sarà un errore.

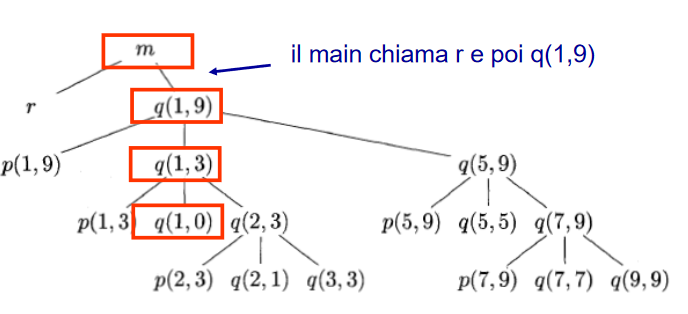
E può inoltre essere un array o un puntatore, nel primo caso si andrà a controllare se l’indice (ovvero E2) sia intero e che E1 sia di tipo array, nel secondo invece si va a controllare se E1 sia un puntatore, in entrambi i casi l’espressione t darà il tipo della struttura, altrimenti ci sarà un errore di tipo. Per quanto riguarda le istruzioni, esse sono a loro volta espressioni, tutti i controlli e le assegnazioni vengono effettuati qui, questo tuttavia dipende dal linguaggio utilizzato, in tal caso vengono considerate come statement, corrispondente al simbolo S nella grammatica. Le chiamate di funzione sono anch’esse espressioni, quindi data una funzione s1 x...x sn→ t, il tipo di E1 deve corrispondere a S così come quello di E2, se è così allora il tipo di E equivale a t, altrimenti ci sarà un errore. Nell’esempio si suppone che E2 possa generare una lista di espressioni separate da virgola e nelle produzioni corrispondenti venga sintetizzato il prodotto cartesiano di tutti come tipo di E2. Negli operatori con overloading ci vuole più attenzione dato che, anche se matematicamente è la stessa cosa, per il calcolatore non è così, inoltre ci possono essere conversioni automatiche senza o con perdita di informazione, in tal caso si suppone che sia ammesso mescolare variabili di tipo diverso e che la conversione sia implicita, altrimenti non si ammettono queste cose e quindi la conversione deve essere esplicita.



Limitandoci all’analisi dei tipi, l’azione associata deve considerare tutti i casi possibili, nell’esempio si può dire che il tipo risultante è intero quando i “sottotipi” iniziali lo sono, in tutti gli altri casi è reale.A meno di conversioni esplicite, il tipo preso in considerazione tra i sottotipo è quello “più grande”.

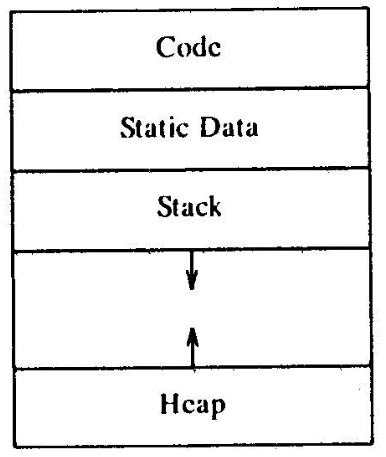
Esempio: utilizzando la grammatica trattata in precedenza, con la stringa a=b+c se il tipo di a equivale a quello di b+c, allora va bene, si ottiene lo stesso risultato anche con b+c intera dato che si può accettare questa assegnazione, nella generazione del codice si andrà quindi ad aggiungere un’istruzione per la conversione implicita.

## Ambiente runtime

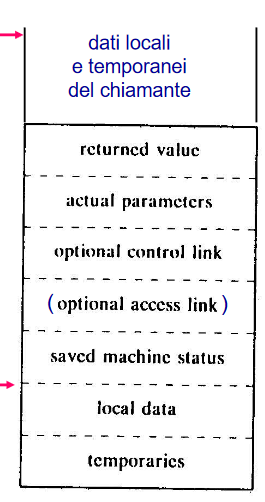
Dal momento che durante l’esecuzione vi sono dati globali e locali, per capire dove sono si fa il seguente esempio: l’ordinamento con il quicksort utilizza una funzione partition che ordina secondo un dato schema più due chiamate ricorsive per dividere a metà l’array e ordinare le due parti, questo dà luogo a un albero che descrive l’esecuzione mettendo in evidenza le chiamate come nell’immagine.

Se p(1,9) restituisce 4, le chiamate ricorsive si adeguano richiamando q(1,3) e q(5,9), prima o poi arriverà il caso base in cui queste chiamate restituiscono una parte vuota dell’array e, di conseguenza, finisce e diventa una foglia dell’albero. Una volta trovate tutte le foglie, di un nodo, si “risale” andando a concluderlo, quella parte di array è quindi ordinata.

In ogni momento dell’esecuzione si ha quindi bisogno di una pila in cui vi sono i parametri di quel dato record e di tutte le variabili locali, a queste ultime sono assegnati degli indirizzi i quali verranno utilizzati per la traduzione. Nel caso delle due chiamate ricorsive, se si stampano gli indirizzi di queste variabili si scopre che vengono utilizzati sempre gli stessi, ciò vuol dire che hanno la stessa dimensione. Dato che lo spazio occupato nella pila è riutilizzato, la distanza tra due variabili uguali tra due record equivale a quella degli stessi.

Gli indirizzi delle variabili locali sono relativi al record in cui si trovano, precisamente fanno riferimento al registro BP che punto alla base del record. L’allocazione avviene generalmente come in figura, il codice e i dati globali hanno una dimensione predeterminata a tempo di compilazione, cosa che non si può dire per heap e stack la cui dimensione dipende dall’esecuzione.

La pila segue l’ordine dato dall’immagine, da essa si può dire che:

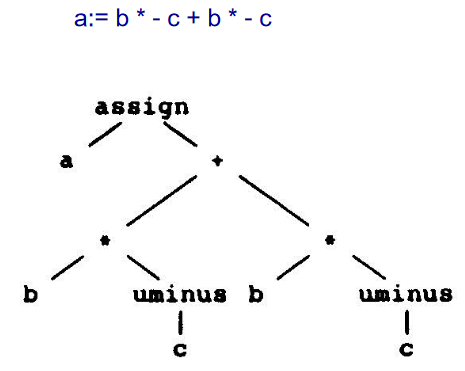
* le frecce indicano i limiti del record;
* il valore di ritorno indica cosa deve essere restituito dalla funzione ed è per questo che si trova prima di tutti, in questo modo è possibile accedere al suo indirizzo attraverso BS, senza conoscere la dimensione dei dati locali;
* i parametri sono ciò che viene passato alla funzione e, essendo di numero limitato, si trovano prima di BS;
* il link di controllo contiene informazioni che servono per il ritorno al record precedente come il valore di BS;
* lo stato della macchina contiene i valori dei registri da ripristinare alla fine;
* I dati locali e temporanei hanno dimensione variabile e quindi si trovano dopo BS, essi indicano rispettivamente le variabili locali e quelle temporanee, queste ultime vengono utilizzate per contenere i risultati intermedi delle operazioni.

Se si ha una funzione f(e1,...,e2), si valuta prima l’espressione ei, si fa una push di essa e dopo si passa alla successiva, questo permette di trattare le funzione variadiche:nel caso della printf(), per sapere quanti parametri ci sono e quali sono, si suppone che in pila ci sia un puntatore alla stringa e quello dei parametri subito dopo. Con questo ragionamento si ha la stringa in pila seguita dei parametri e dai registri salvati, questo però è un problema perchè, dato che il numero di parametri è variabile, gli indirizzi nella pila non risultano sempre uguali per due record con più o meno parametri, soprattutto per quanto riguarda la stringa (il parametro sempre presente). Per risolvere questo problema, si passano i parametri al contrario, in questo modo la stringa avrà un indirizzo fisso (e quindi noto) rispetto a BS, gli altri parametri invece vengono fatti scorrere finchè vi sono dei % nella stringa, interpretando i byte in base al formato richiesto. Per ammettere questo caso, il codice generato per una printf deve contenere:

* istruzione per il calcolo del primo parametro (in questo caso la stringa);
* istruzioni per calcolare il parametro i-esimo;
* push del parametro i-esimo;
* push del primo parametro;

Le prime due istruzioni sono necessarie in quanto il parametro può essere un’espressione e non una semplice variabile.

## Generazione del codice

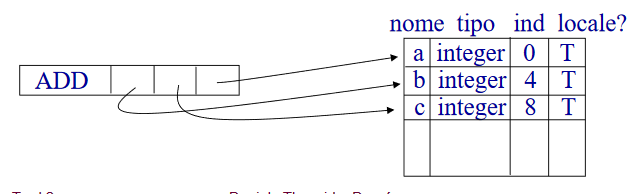
La generazione del codice intermedio serve a ottenere un codice univoco non adattato alla piattaforma in cui gira, gli alberi di sintassi possono essere visti come tale in quanto vi sono dipendenze tra operatori.

Il formato delle istruzioni che utilizzeremo sarà composto da tre indirizzi, i primi due sono indirizzi sorgenti mentre l’ultimo è la destinazione, all’inizio vi è un opcode indicante l’operazione:

OPCODE OP1 OP2 OP3

Per rendere più leggibile il codice, si utilizzano le seguenti notazioni:

* OP3=OP1 op OP2 per l’applicazione di un operatore;
* if OP1 relop OP2 goto OP3 per i salti condizionati.

Gli operandi possono essere costanti etichette oppure variabili, queste ultime possono corrispondere a indirizzi oppure a un nome simbolico nel caso in cui il codice non sia eseguito direttamente: con a=b+c, al posto dei nomi delle variabili vi sono i relativi puntatori alla tabella dei simboli.

Il campo locale della tabella indica che la variabile è locale con indirizzo locale. Se si vuole far eseguire il codice intermedio su una macchina virtuale,, è possibile eliminare la tabella dei simboli, in tal caso l’indirizzo dovrebbe avere un flag che indica se:

* si tratta di una variabile o costante;
* si tratta di un dato statico, quindi con indirizzo assoluto;
* si tratta di un dato locale con indirizzo relativo.

Quest’ultimo caso è una modalità disponibile nei linguaggio per architetture reali. Nel codice macchina gli operandi possono anche essere registri, questo caso però non lo prevediamo anche se è importante per motivi di efficienza, al loro posto utilizziamo le variabili temporanee. Per quanto riguarda le istruzioni, esse dovrebbero essere una per ogni tipo base, qui però ne utilizzeremo solo una per indicarle tutte: OP1=OP2. Oltre a queste vi sono operazioni aritmetiche, booleane, logiche e di conversione, inoltre vi sono anche assegnazioni con indicizzazione:OP1[OP2]=OP3.

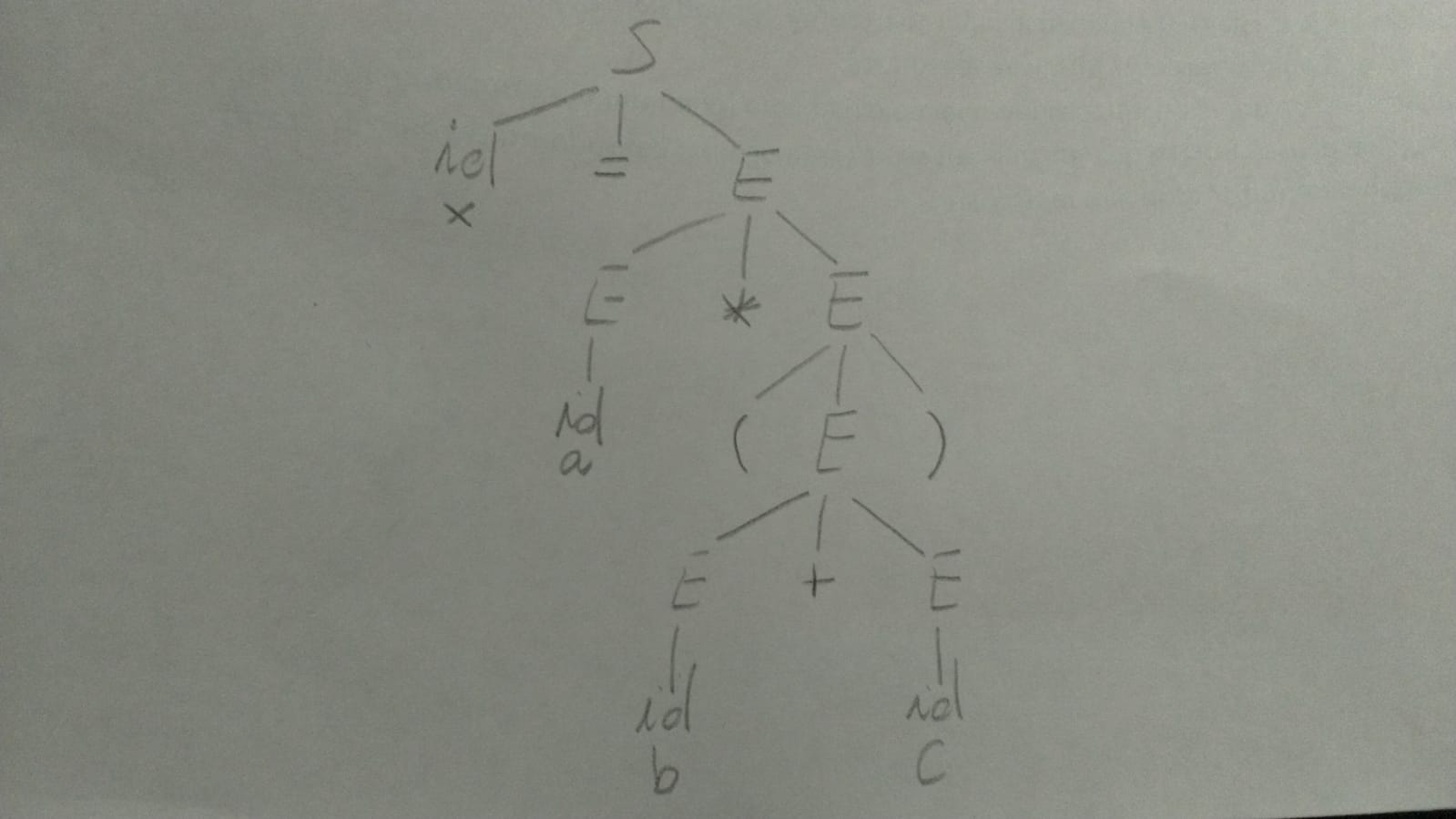
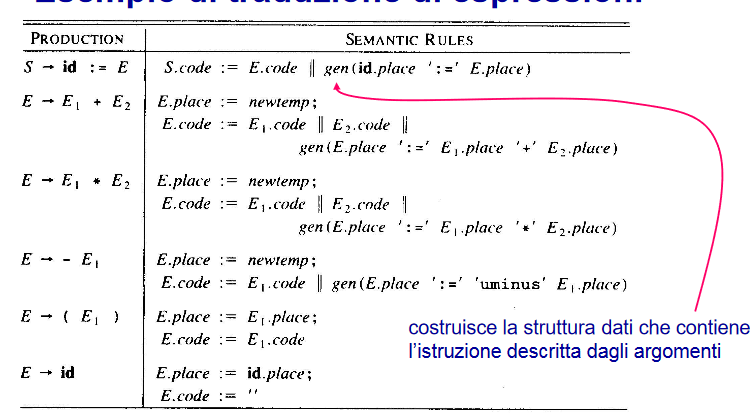
Nell’esempio sopra OP1 si riferisce a una variabile array e il dato si ottiene sommando OP2 all’indirizzo base, in questo caso però non interessa il suo valore ma il suo indirizzo dato che si trova nella parte sinistra dell’assegnazione, di OP3 invece interessa in valore. Le istruzioni di salto si indicano con goto per quello incondizionato e con if(cond) goto ind per la variante condizionata, ind indica l’indirizzo a cui saltare e può essere il numero dell’istruzione o un’etichetta: goto OP1; if(OP1) goto OP2 else goto OP3.

Ultime ma non meno importanti sono le istruzioni che utilizzano l’indirizzo di un operando o dereferenziano un puntatore:

OP1= &OP2; OP1=\*OP2

La chiamata a sottoprogrammi si esegue con call specificando l’indirizzo, essa salta all’indirizzo dato dopo aver salvato i registri nel record del chiamato, per passare per valore eventuali parametri si utilizza param seguito dall’operando. Alloc è l’istruzione che alloca spazio per variabili e temporanei andando a incrementare di N il valore di un registro puntatore alla cima della pila, l’uscita da un sottoprogramma avviene con record ripristinando i registri salvati e deallocando lo spazio mentre il termine di un programma avviene con halt. La traudizone si può eseguire seguendo uno dei seguenti stili:

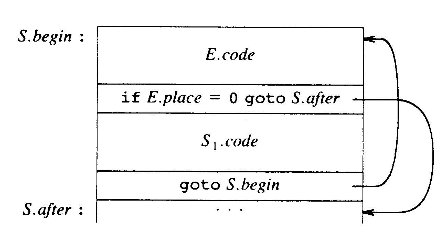
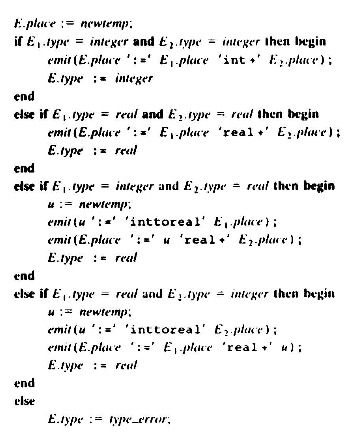
* dds con codice generato tramite un sintetizzato, esso viene ottenuto concatenando le sottoespressioni, in tal caso è meglio utilizzare le etichette come salti;
* schema di traduzione in cui il codice viene emesso attraverso delle azioni che aggiungo le istruzioni in appendice a un file, qui sono preferibili i salti con gli indirizzi, tuttavia il risultato ottenuto non è detto che sia sequenziale dal momento che non si sa precisamente la destinazione.

Esempio: Se S genera un’istruzione, l’attributo S.code genera la traduzione, l’azione gen invece genera la struttura dati che la contiene. E indica le operazioni e le variabile ed è munito di un’attributo place indicante il valore calcolato in real-time, precisamente è il puntatore alla variabile o al temporaneo. id invece non ha bisogno di codice associato e il suo attributo place indica la sua posizione nella tabella dei simboli. Considerando la seconda produzione, si ha che il codice di E è formato da quello di E1 e di E2, inoltre viene generata l’istruzione in cui viene calcolato E.place. Con x=a\*(b+c), si ha l’albero mostrato in figura, da ciò si può dire che la seconda produzione genera un temporaneo t1 contenente b+c, la terza genera t2 il cui valore è t2\*a e infine avviene l’assegnazione a x, il codice è quindi il seguente:

t1=a+b;

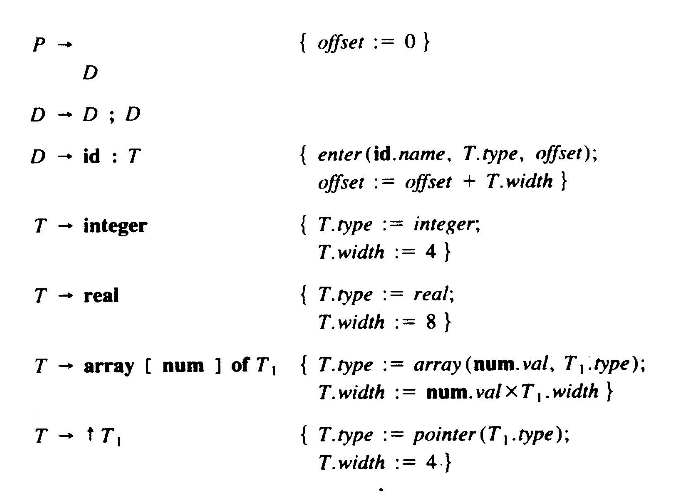
t2=a\*t1;

x=t2;

Nel caso delle istruzioni di controllo, precisamente nel ciclo while, se l’istruzione é vera si salta all’etichetta after che si trova in fondo alla fila, altrimenti si salta in cima dov’è presente begin per rivalutare l’espressione. Il secondo approccio presenta azioni che inseriscono istruzioni in appendice (emit) o prendo il puntatore a una variabile dalla tabella dei simboli (lookup), considerando la seconda produzione, prima vengono emesse le istruzioni per E1 e E2 e dopo quella del calcolo di E.place, inoltre verrà generato un nuovo temporaneo che sarà dato a quest’ultimo. Con la prima produzione si va a prendere il puntatore alla variabile id nella tabella dei simboli e, se viene trovato, gli assegna il valore del temporaneo, altrimenti riporta un errore. In entrambi i casi i due metodi devono tenere conto dei tipo e quindi bisogna fare l’analisi degli stessi e, se necessario, inserire istruzioni di conversione.

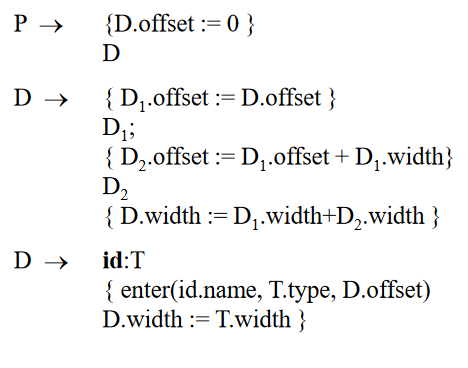
Esempio: vi sono quattro possibili combinazioni, nei casi estremi, ovvero quando gli operandi sono entrambi interi o entrambi reali, non vengono effettuate conversioni, altrimenti c’è bisogno della conversione da intero a reale per evitare di perdere informazioni, in ogni caso si emette l’istruzione e si assegna il tipo.

### Analisi delle dichiarazioni

Data la grammatica e le azioni associate, l’idea è quella di riempire la tabella dei simboli con dell’informazione, ovvero l’indirizzo relativo al record di attivazione, se si hanno tre variabili intere a,b e c, ognuna occupante 4 byte, l’indirizzo dipende da questa dimensione. Dalla grammatica si può ricavare l’albero in figura, osservandolo si può dire che, quando si incontra una generazione di tipo, l’idea è che nel suo sottoalbero venga occupato lo spazio occupato che, alla riduzione del tipo in T, venga assegnato appropriamente. L’idea alla base di D invece è quella di svolgere l’azione in prossimità della riduzione, se essa è la prima occorrenza incontrata allora l’offset è 0 (non vi è spazio occupato nel record), altrimenti si deve tenere conto anche delle D precedenti, ad esempio: 

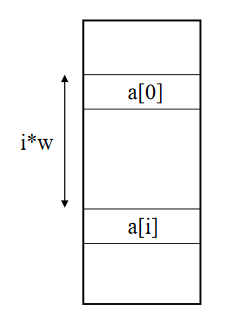
a, c interi e b reale

a è la prima variabile e avrà offset 0, b è la seconda e deve tenere conto della prima, l’offset è quindi 4 (dato che a è intero) e la sua dimensione è 8, per c intero si fa lo stesso ragionamento, per quest’ultima si ha un 12 di offset.

In questo modo è possibile riempire la tabella dei simboli di indirizzi relativi da associare alle variabili, si può ottenere un lavoro simile utilizzando degli attributi che emulano la tabella dei simboli, come nell’esempio:

con P si inizializza l’offset a 0 come in precedenza, esso è un ereditato di D che verrà passato ai figli e modificata alla riduzione tenendo conto del tipo della variabile e della dimensione occupata da quella precedente. Alla D padre viene quindi data come dimensione la somma di quelle dei figli, questo attributo sintetizzato serve per capire quanto si occupa.

## Accesso agli array

Per accedere alle celle degli array si suppone che nelle variabili globali cominci da un certo punto, infatti rispetto a BP si può indicare l’indirizzo relativo della cella 0 e calcolare tutti gli altri attraverso il prodotto tra l’indice e la dimensione del tipo.

l’indice dell’array è tuttavia noto solamente a tempo di esecuzione, di conseguenza tutte le azioni verranno fatte in quel momento. Per tradurre un’istruzione a[E1]=E2, bisogna prima calcolare E1,inserire il risultato in un temporaneo e moltiplicarlo con la dimensione del tipo, in seguito si effettua il calcolo di E2, si inserisce il risultato in un’ulteriore temporaneo e infine si va a effettuare l’assegnazione, il codice sarà quindi il seguente:

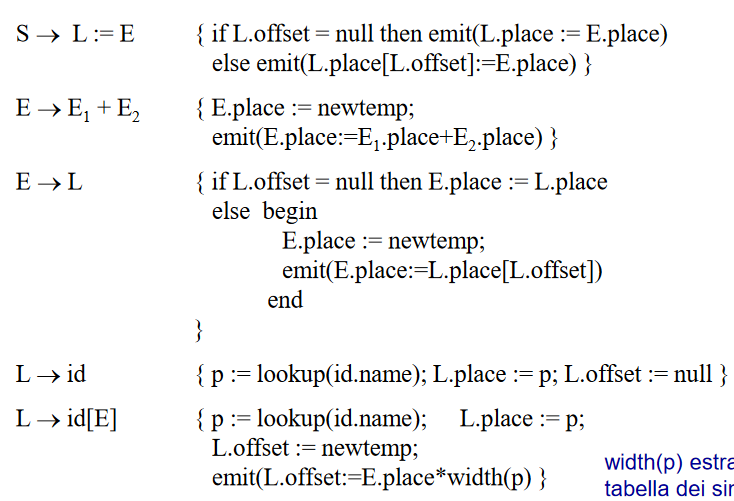
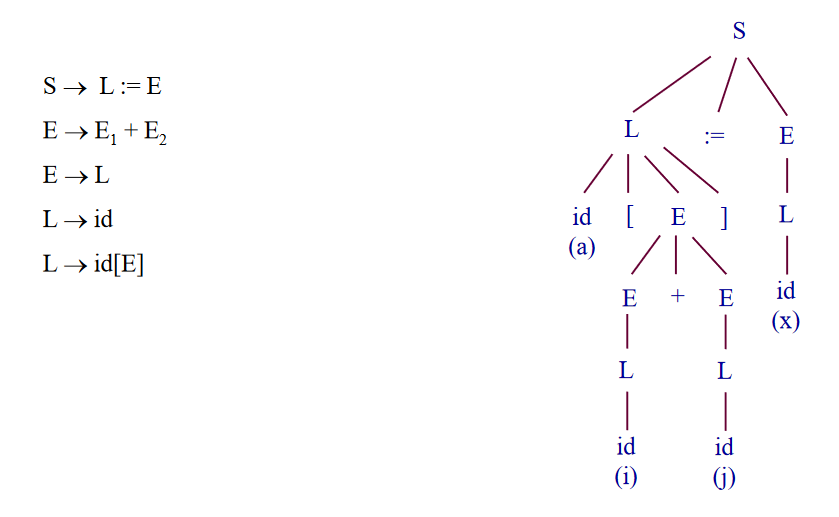
t1=risultato(E1)\*dimTipo;

t2=risultato(E2);

a[t1]=t2;

Nel caso in cui l’array a[E1] si trova nella parte destra di un’operazione, bisogna trovare il suo indirizzo per ottenere il valore della cella, per farlo si calcola t1=ris(E1)\*dimTipo e si assegna a t2.

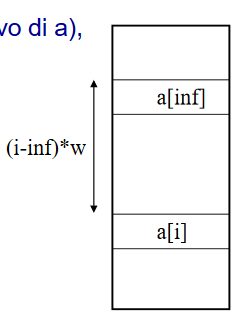
Esempio: a[i+j]=x



Per distinguere le normali variabili dagli array, si mette offset uguale a null nelle prime, vi saranno poi dei controlli per verificare in quale dei due casi ci si trova. Col lo schema di traduzione dato, in entrambi i casi L va a prendere l’indirizzo nella tabella dei simboli e lo assegna a place, inoltre se è un array si ricava l’offset utilizzando un temporaneo emesso da emit. La prima produzione indica l’istruzione di assegnazione, di conseguenza E.place viene copiato in L.place della variabile o dell’array (tenendo conto dell’offset in quest’ultimo caso), l’informazione L.place-L.offset combinata dà la locazione dell’array in cui copiare E.place. La terza produzione viene eseguita quando l’espressione genera una variabile, anche qui il passaggio degli attributi avviene in base a L, se è variabile il passaggio avviene normalmente, altrimenti si trova l’indirizzo, lo si mette in un temporaneo e si emette l’istruzione.

### Array che non iniziano da 0

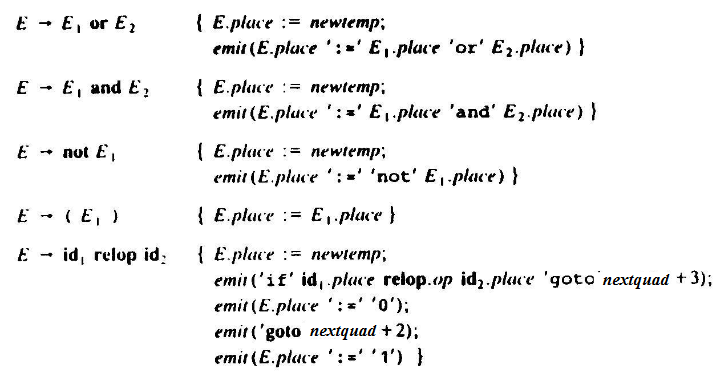
Alcuni linguaggi ammettono array la cui prima cella non è 0, in tal caso bisogna memorizzare la posizione iniziale dato che la distanza in byte non vale più come prima: dati inf e sup rispettivamente come estremo inferiore e superiore, per trovare l’elemento i-esimo si trova la distanza rispetto a inf e si moltiplica per la dimensione del tipo, il valore ottenuto dovrà poi essere sommato all’indirizzo dell’array rispetto a BS, ovvero a[inf], quindi:

indirizzo(array)+(i-inf)\*w

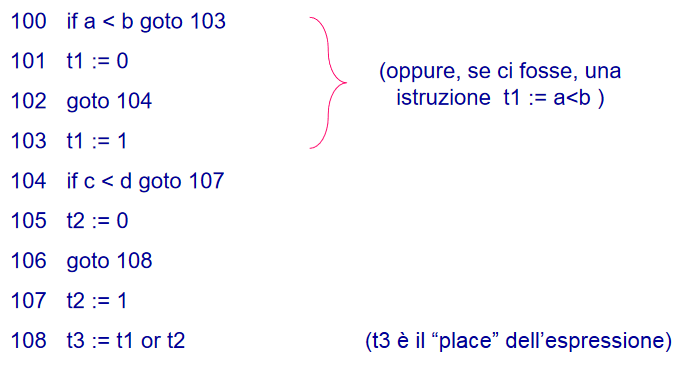
Tuttavia è possibile fare delle ottimizzazioni: riscrivendo la formula sopra come indirizzo(array)-inf\*w +i\*w, è possibile risparmiare un temporaneo andando a memorizzare la prima parte dell’addizione nella tabella dei simboli (come indirizzo di a) dato che nota a tempo, di compilazione, in questo modo si calcola solo i\*w e si va a prendere il valore pre-calcolato dalla tabella.

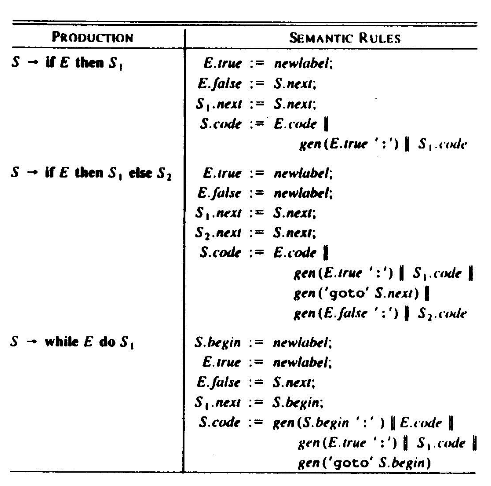
## 

## Traduzione di espressioni booleane

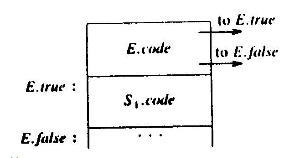
Per tradurre le espressioni booleane, la soluzione più intuitiva consiste nel valutarle come espressioni numeriche, come nell’esempio:

restringendo il linguaggio ai soli operatori booleani, la cosa complicata è che non vi sono istruzioni intermedie che utilizzano temporanei. La emit per un costrutto if...goto farà saltare all’indirizzo dato, nextquad è l’indirizzo della prossima istruzione da emettere ed è calcolato rispetto a quelli attuali, se ad esempio si parte da 100, tutte le istruzioni i-esime successive saranno all’indirizzo 100+i. Con la stringa a<b or c<d si ottiene il seguente codice:

Un altro sistema per la traduzione è quello basato sui salti in cui, a seconda della condizione, si hanno differenti destinazioni in cui vi è il codice per gestire quel caso,arrivare a un certo punto corrisponde quindi alla veridicità o meno di una condizione. Anche i i temporanei non vengono utilizzati e quindi si utilizzano altri metodi per le istruzioni composte, inoltre è possibile evitare di valutare tutta l’espressione in base ai valori di quelle più interne, ad esempio: con E1 or E2, se E1 è vera, tutta l’espressione è vera per via delle proprietà dell’or, quindi non è necessario valutare E2, altrimenti si ha il bisogno di farlo, un discorso simile vale anche per E1 and E2, infatti se E1 è falsa, allora lo è anche tutta l’espressione. Le destinazioni di un salto possono essere indirizzi o etichette simboliche, considerando quest’ultimo caso, oltre ai sintetizzati E.code e S.code vengono anche utilizzati due ereditati, ovvero E.true e E.false, essi conterranno le etichette a cui saltare quando E è rispettivamente vera o falsa. Ultimo ma non meno importante è l’ereditato S.next, utilizzato per saltare all’etichetta indicata al termine dello statement, esso serve per evitare i “salti a salti”. 

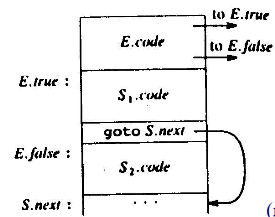
Esempio: S→ if(E) S1

E. code contiene una sequenza di istruzioni intermedie, esso contiene le due etichette in E.true e E.false indicanti i salti alle etichette nel codice. In questo caso se S genera if(E) S1 allora bisogna mettere delle azioni per valorizzarlo dato che a sinistra della produzione, lo stesso discorso si fa anche con E ma con gli ereditati (visto che è a destra) generando una nuova etichetta E.true da mettere all’inizio del codice di S1. E.false andare in una parte del codice che si trova intorno a S, ovvero quando quest’ultimo è finito, si può notare che S.next e E.false portano alla stessa destinazione, quindi avranno la stessa etichetta assegnata, lo stesso discorso vale anche per S1.next dato che, quando quest’ultimo finisce, finisce anche S.



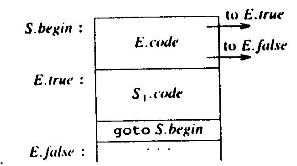
Esempio: S→ if(E) S1 else S2

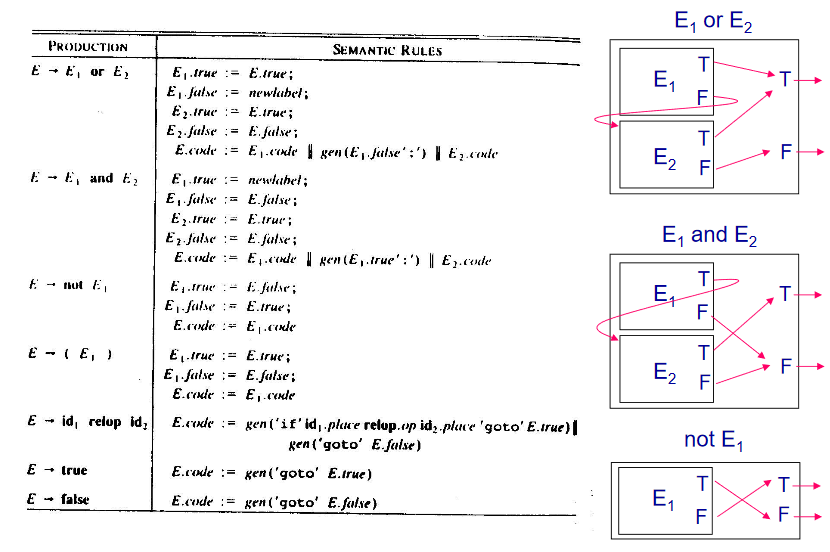
Questo esempio è molto simile al precedente, la differenza è che qui E.false non corrisponde più a S.next, infatti salterà all’inizio del codice di S2, inoltre S1.next salterà all’istruzione successiva a S2 (corrispondente a S.next in questo caso) per evitare di “invaderlo”.



Esempio: S→while(E) S1

In questo caso l’etichetta in E.true andrà messa subito dopo il codice di E, ovvero nel corpo del ciclo, E.false invece salterà fuori in una posizione data dall’intorno. Per quanto riguarda il corpo del ciclo, alla fine di S1 vi è un salto incondizionato (S.begin) che porta all’inizio del codice di E per poter rivalutare l’espressione, si può quindi dire che l’etichetta di S.begin e S.next è la stessa nel caso in cui non vi sia altro codice.



Si considerano ora le espressioni booleane, utilizzando la grammatica a destra, quando vi è un confronto, si genera l’istruzione e si assegnano le etichette a E.true e E.false in base al caso. Nelle espressioni tra parentesi vi è il solo passaggio di attributi tra padre e figlio, esso avviene anche con la negazione con la differenza che le due etichette vengono scambiate. Per quanto riguarda gli operatori logici or e and, si segue il ragionamento detto in precedenza: per l’or se la prima espressione è vera, allora si esce senza valutare la seconda, altrimenti si va a farlo, con l’and il ragionamento è simile con la differenza che si esce dal controllo con un’espressione falsa, i due schemi a destra mostrano in breve il loro modo di operare.

Esempio: a<b or c<d and e<f

Utilizzando l’input sopra e la grammatica di prima, l’albero di derivazione ottenuto è quello a destra. Le etichette Ltrue e Lfalse vengono generate alla E radice e assegnate ai rispettivi ereditati, essi sono locali al contesti di E e indicano dove andare alla fine di tutta l’espressione. Il figlio sinistro di E è l’operando sinistro di un or, seguendo il ragionamento di prima, l’ereditato E.true avrà Ltrue associata (dato che tutta l’espressione è di conseguenza vera), viene poi generata un’etichetta L1 per saltare al fratello quando l’espressione è falsa in modo da poterlo valutare, alla riduzione viene generata l’istruzione di confronto utilizzando le etichette memorizzate. Il figlio destro della radice eredita gli attributi dal padre, questo perchè, essendo in questo caso il secondo operando, deve tenere conto di tutte le possibili combinazioni, qui si ha una situazione inversa rispetto all’or, l’operatore and infatti non valuta tutta l’espressione se la prima è falsa, quindi si salta a Lfalse quando E è falsa e si genera una nuova etichetta da assegnare a E.true per andare a valutare il fratello (solo se il figlio sinistro è vero). Un secondo metodo di traduzione è utilizzare gli indirizzi di destinazione come salti, ciò però presenta problemi per quanti riguarda i salti in avanti, quello che si vorrebbe fare è saltare all’indirizzo dato, tuttavia non si sa quanto occupa ogni blocco di istruzioni non ancora generate e quindi non si sa con certezza dove si va.

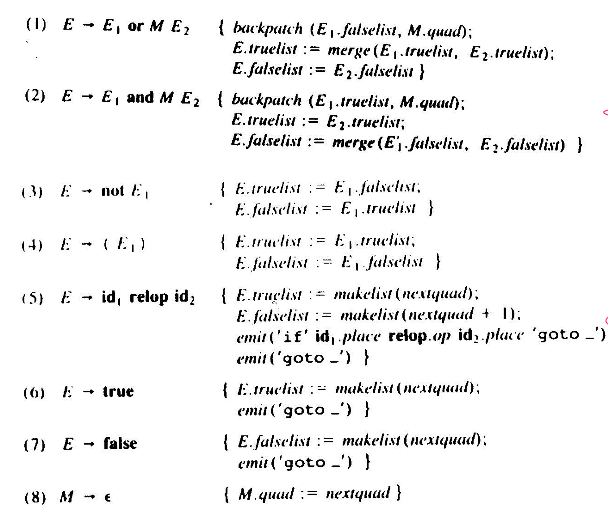
Esempio S→ if(E) S1 else S2

Se E1 genera a<b che risulta vera, essa dovrà portare all’inizio di S1, tuttavia non si sa quanto occuperà E2 quando si farà la emit di E1 dato che si trova in mezzo, non permettendo così l’esecuzione delle azione durante il parsing di E2. Questo problema potrebbe essere risolto valutando gli attributi dopo la costruzione dell’albero generando del codice intermedio, tuttavia in questo modo si hanno dipendenze da destra a sinistra tra ereditati. La soluzione è un arrangiamento: si emette del codice incompleto e dopo si ritorna in modo non sequenziale attraverso delle funzioni che riempiono i buchi lasciati, tutte queste istruzioni rimaste incomplete vengono inserite delle liste specifiche:

* E.truelist contiene tutte le istruzioni che saltano all’indirizzo in cui E è vera;
* E.falselist fa la stessa cosa ma con E falsa;
* S.nextlist contiene le istruzioni in cui saltare quando lo statement finisce.

La destinazione sarà nota quando l’istruzione verrà emessa, per farlo si utilizza un’apposita variabile, anche nel caso di istruzioni in sequenza, la funzione che tappa i buchi è backpatch e prende come input la lista di istruzioni da rattoppare e l’indirizzo di destinazione.

Esempio:

Data la grammatica e le azioni associate, nei casi base, ovvero quando l’espressione E genera true, false oppure un confronto, vengono create le due liste di istruzioni, infatti se E genera a<b, si guarda intorno per vedere se vi sono casi in cui si salta all’indietro (quindi si conosce la destinazione) oppure no, in quest’ultimo caso si lascia incompleto il codice. peremettere il codice si utilizza nextquad (una variabile che indica l’indirizzo della prossima istruzione da emettere) il cui valore verrà inserito in una delle liste in base al valore di E. Nelle operazioni logiche, l’idea è avere il codice di E1 seguito da quello di E2, quindi bisogna fare in modo che se è E1 è vera, si salti direttamente alla fine dell’espressione, per farlo si utilizza un marker M contenente il valore di nextquad, esso non è tuttavia necessario dato che è possibile effettuare la backpatch prima di andare a valutare E2.

E→ E1 or {backpatch(E.falselist,nextquad)} E2

E→ E1 and {backpatch(E.truelist,nextquad)} E2

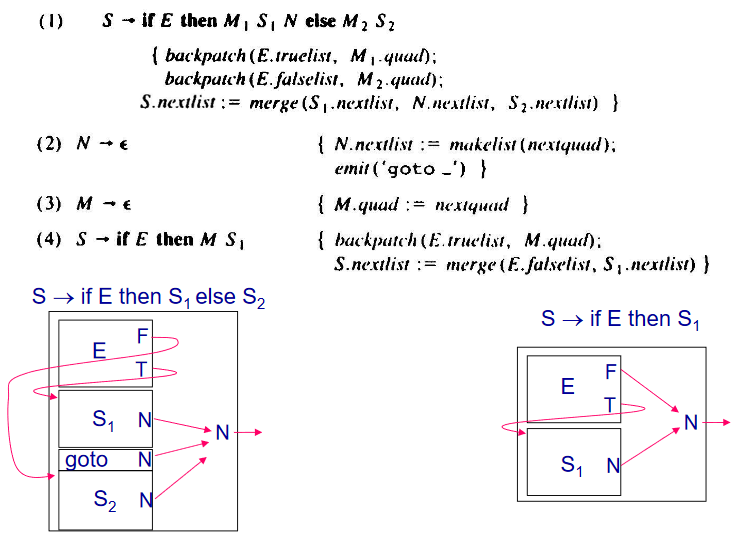
Dopo aver rattoppato i buchi con la backpatch (rispettivamente quelli “falsi” per l’or e quelli “veri” per l’and), si procede come nelle etichette simboliche con qualche differenza: nel caso dell’or, tutte le istruzioni “vere” delle due espressioni vengono unite in un’unica lista (basta che una E sia vera per portare alla stessa destinazione), per quanti riguarda invece le istruzioni “false”, si tengono in considerazione solamente quelle del secondo dato che quest’ultima sarà l’espressione da valutare se la prima è falsa, per l’and si fa la stessa cosa ma “al contrario”. Come per le etichette simboliche, nelle parentesi e nella negazione si è solo un passaggio di valore, rispettivamente normale per il primo e scambiato per il secondo.

Esempio: a<b or c<d and e<f

Data la grammatica e le azioni associate, con l’input sopra si ottiene l’albero a destra.

<slide 15>

Supponendo di partire dall’istruzione 100, la prima produzione che emette codice è il figlio sinistro di E radice, qui verranno create le due liste rispettivamente col valore di nextquad (100 in questo caso) e il suo successivo, quest’ultima istruzione verrà poi rattoppata dal figlio destro utilizzando l’indirizzo memorizzato dal marker come destinazione, in questo caso 102. Partendo da quest’ultimo, il figlio sinistro genera a sua volta le istruzioni e le inserisce nelle rispettive liste, qui però è presente un’and e quindi si ragiona diversamente: E padre si prende le istruzioni “vere” dal secondo e unisce le due liste false, la stessa cosa si fa per E radice ma al contrario. Vi sono ancora istruzioni che sono rimaste incomplete, questo però non è un problema perchè esse verranno riempite dall’intorno dell’espressione.

Esempio: if E then M1 S1 else M2 S2

In questo statement le prime istruzioni a essere emesse sono quelle di E, poi quelle di S1 e infine quelle di S2, quindi è possibile sapere dove arrivano i salti dell’espressione, quando essa è vera si va all’inizio di S1, altrimenti in S2. S1 e S2 sono separate da un salto che porta alla fine dello statement, questo serve per evitare di invadere S2 quando si finisce di valutare S1, al loro interno possono esservi salti che portano anch’essi alla fine, questo vuol dire che le nextlist di S, S1 e S2 è contengono le stesse istruzioni incomplete.

## Chiamate a procedure

Supponendo di generare una chiamata p con delle E come parametri, si calcola ognuna di esse, se l’espressione i-esima è un’operazione, allora si calcola e si mette in E.place il risultato, tutti questi verranno messi in pila utilizzando param, essi devono essere messi al contrario per risultare in ordine. Una procedura è formata da un id e da una lista di espressioni, nella produzione si suppone che venga generato il codice dell’istruzione e che l’azione associata genera il parametro, quella di S ad esempio serve per effettuare la call (la chiamata effettiva delle procedura che salva lo stato dei registri e carica il codice) effettuando un lookup dell’id nella tabella dei simboli, in tale occasione si controlla anche se il tipo della funzione. Per quanto riguarda tipo e numero di parametri, vi sono le funzioni checktype e checknum. Per la dichiarazione di una procedura con produzione L→ L1,S si ha che i salti di S portano a quella successiva, essi però rimangono incompleti e possono essere riempiti con uscite da cicli o quant’altro ma al di fuori di essi. Alla dichiarazione di una procedura, si alloca spazio per parametri e variabili locali, lo spazio delle temporanee però è ancora incerto, quindi si esegue un’apposita azione dopo aver visitato S utilizzando l’attributo width, infine si rattoppano gli eventuali salti mandando verso la return emessa. Nel caso in cui bisogna restituire un valore, si emette un param 0 prima dei parametri, lì si andrà a mettere il valore da restituire, esso potrà essere preso utilizzando l’istruzione getret il quale lo prende dalla cima della pila e lo inserirà in un temporaneo, la return infatti ripristina lo stato precedente riportando lo stack pointer dov’era prima e mettendo il valore in cima alla pila.

