Corso di Laurea Magistrale in Ingegneria Informatica
A.A. 2011-2012

Linguaggi Formali e Compilatori

Analisi sintattica (parser)

Giacomo PISCITELLI

Analisi sintattica: il parser

Problema di base dell'analisi sintattica: data una sequenza di token e una specifica della sintassi, **stabilire se la sequenza è una frase ammessa dalla sintassi**.

A questo scopo principale, se ne aggiungono altri due non meno importanti:

- verificare se la sequenza di token rispetta le regole semantiche del linguaggio;
- costruire una rappresentazione intermedia (IR) del codice sorgente

La sintassi è in genere specificata in termini di un linguaggio non contestuale (tipo 2), in cui ogni token costituisce un simbolo atomico del linguaggio.

Pertanto, il problema equivale al problema del **riconoscimento** nei linguaggi non contestuali: data una stringa (sequenza di simboli) e un linguaggio non contestuale, stabilire se la stringa appartiene al linguaggio.

Analisi sintattica: il parser

Data una sequenza ***s*** di token, l'analizzatore sintattico verifica se la sequenza appartiene al linguaggio generato dalla grammatica ***G***.

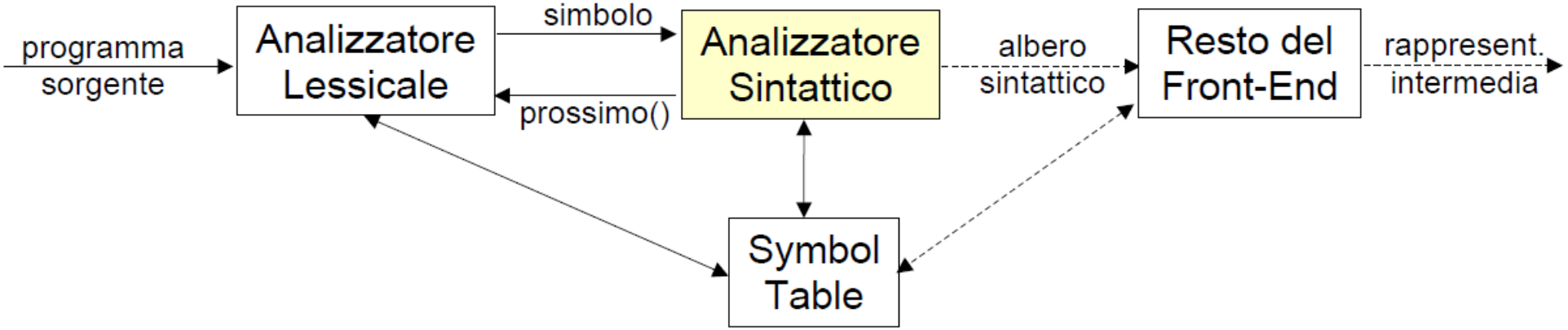
A questo scopo, l'analizzatore sintattico cerca di costruire l'albero sintattico di *s*:

- in caso positivo, restituisce in uscita l'albero sintattico per la sequenza di input nella grammatica *G*;
- in caso negativo, restituisce un errore (errore sintattico).

Analisi sintattica: il parser

Un parser deve riconoscere la struttura di una stringa di ingresso, che è fornita in termini di regole di produzione di una Context Free Grammar (CFG), di una BNF, o di diagrammi sintattici.

Un parser è una macchina "astratta" che raggruppa input in accordo con regole grammaticali.



Costrutti che iniziano con parole chiave (es. while o float) sono relativamente facili da analizzare, perché la parola chiave guida la scelta della regola (produzione) grammaticale che deve essere applicata in corrispondenza dell'input. Più impegnativo si presenta il lavoro nel caso di espressioni, a causa delle regole di associatività e di precedenza tra operatori.

Analisi sintattica: la sintassi

La **sintassi** è costituita da un insieme di regole che definiscono le frasi formalmente corrette e allo stesso tempo permettono di assegnare ad esse una struttura (albero sintattico) che ne indica la decomposizione nei costituenti immediati.

Ad esempio, la struttura di una frase (ovvero di un programma) di un linguaggio programmativo ha come costituenti le parti dichiarative e quelle esecutive.

Le parti dichiarative definiscono i dati usati dal programma.

Le parti esecutive si articolano nelle istruzioni, che possono essere di vari tipi: assegnamenti, istruzioni condizionali, frasi di lettura, ecc.

I costituenti del livello più basso sono gli elementi lessicali già considerati, che dalla sintassi sono visti come atomi indecomponibili. Infatti la loro definizione spetta al livello lessicale.

Analisi sintattica: sintassi context free

La teoria formale dei linguaggi offre diversi modelli, ma nella quasi totalità dei casi il tipo di sintassi adottata è quello noto come sintassi libera o non-contestuale (context-free), che corrisponde al tipo 2 della gerarchia di Chomsky.

Perchè

- ✓ i metodi sintattici per il trattamento del linguaggio sono semplici ed efficienti;
- ✓ la definizione del linguaggio attraverso le regole delle sintassi libere è diretta ed intuitiva;
- ✓ gli algoritmi deterministici di riconoscimento delle frasi sono veloci (hanno complessità lineare) e facili da realizzare partendo dalla sintassi.

Tutti questi vantaggi hanno imposto le sintassi libere come l'unico metodo pratico per definire formalmente la struttura di un linguaggio.

Limiti di una sintassi context free

Ma quello che si può ottenere con una sintassi libera è limitato e spesso insufficiente: la sintassi non basta a definire le frasi corrette del linguaggio di programmazione, perché una stringa sintatticamente corretta non è detto che lo sia in assoluto.

Infatti essa potrebbe violare altre condizioni non esprimibili nel modello noncontestuale.

Ad esempio è noto che la sintassi non può esprimere le seguenti condizioni:

- in un programma ogni identificatore di variabile deve comparire in una dichiarazione;
- il tipo del parametro attuale di un sottoprogramma deve essere compatibile con quello del parametro formale corrispondente.

Condizioni, quelle precedenti, che sono giustificate dalla necessità di **attribuire un significato** agli elementi sintattici e di **stabilire delle regole semantiche** alla composizione di elementi sintattici.

Da queste critiche sarebbe sbagliato concludere che i metodi sintattici sono inutili: al contrario essi sono indispensabili come supporto (concettuale e progettuale) su cui poggiare i più completi strumenti della semantica.

Grammatiche di tipo context free

Le strutture ricorsive presenti nei linguaggi di programmazione sono definite da una grammatica context-free.

Derivazioni (o produzioni)

Le produzioni devono avere le seguenti proprietà:

- ✓ nessuna produzione deve essere inutile o ridondante (i.e., **A** → **A**),
- ✓ nessun simbolo non-terminale senza produzione corrispondente (e.g., **A** → **Ba** con **B** non definito),
- ✓ nessun ciclo infinito (e.g., **A** → **Aa** senza altre produzioni per **A**),
- ✓ nessuna ambiguità: assenza di più alberi sintattici per la stessa espressione,
- ✓ descrivere correttamente il linguaggio.

Analisi sintattica e albero sintattico

L'analisi sintattica può essere vista come un processo per costruire gli alberi sintattici (**parse tree**).

Il parser verifica se un programma sorgente soddisfa le regole implicate da una grammatica context-free o no.

Se le soddisfa, il parser crea l'albero sintattico del programma, altrimenti genera un messaggio di errore.

Una grammatica non contestuale fornisce una specifica rigorosa della sintassi dei linguaggi di programmazione

Il progetto della grammatica costituisce la fase iniziale del progetto del compilatore.

Esistono strumenti automatici per costruire automaticamente il compilatore dalla grammatica.

Metodi dell'Analisi sintattica

I metodi di analisi sintattica sono di 3 tipi:

- ✓ **universali**: sono metodi di parsing *general-purpose* – e quindi applicabili non unicamente con le grammatiche libere da contesto – che hanno, però, l'inconveniente di essere troppo inefficienti (con complessità $O(n^3)$ o, nel migliore dei casi $O(n^2)$) per essere impiegati in compilatori produttivi
- ✓ **top-down**: l'albero sintattico della stringa di input viene generato a partire dalla radice (assioma), scendendo fino alle foglie (simboli terminali della stringa di input)
- ✓ **bottom-up**: l'albero sintattico della stringa di input viene generato a partire dalle foglie (simboli terminali della stringa di input), risalendo fino alla radice (assioma)

Il parser può lavorare in una varietà di modi **ma esso, sia nel metodo top-down che in quello bottom-up, processa l'input da sinistra a destra, un simbolo alla volta.**

Metodi dell'Analisi sintattica

Parser sia top-down che bottom-up possono essere realizzati in modo efficiente solo per alcune sottoclassi di grammatiche context-free. Diverse di tali sottoclassi, quali in particolare le grammatiche LL e LR, sono però sufficientemente espressive per descrivere la quasi totalità dei costrutti sintattici presenti nei moderni linguaggi di programmazione.

I parser LL sono per lo più impiegati nei casi di analizzatori sviluppati ad hoc per grammatiche che usano generare l'albero sintattico con metodi top-down, mentre i parser per la più ampia classe delle grammatiche LR sono normalmente alla base dei tool automatici di analisi sintattica.

LL per top-down — Una "left-most derivation" è una derivazione nella quale durante ogni passo solo il non-terminale più a sinistra è sostituito.

LR per bottom-up — Una "right-most derivation" è una derivazione nella quale durante ogni passo solo il non-terminale più a destra è sostituito.

Left-Most e Right-Most Derivation

Considerata la grammatica

$$E \Rightarrow E + E \mid E * E \mid -E \mid (E) \mid \mathbf{id}$$

e presa la stringa

$$-(\mathbf{id} + \mathbf{id})$$

tale stringa può essere derivata da E attraverso:

- una *Left-Most Derivation*

  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathit{E} + E) \Rightarrow -(\mathbf{id} + \mathit{E}) \Rightarrow -(\mathbf{id} + \mathbf{id})$

  nella quale si sostituisce il non terminale più a sinistra

- oppure una *Right-Most Derivation*

  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + \mathit{E}) \Rightarrow -(\mathit{E} + \mathbf{id}) \Rightarrow -(\mathbf{id} + \mathbf{id})$

  nella quale si sostituisce il non terminale più a destra

Un ***top-down parser*** cerca una left-most derivation dell'istruzione del *source*

Un ***bottom-up parser*** cerca una right-most derivation dell'istruzione del *source*

PARSER TOP-DOWN

Il "parse tree" è creato dalla radice alle foglie.

Un parser top-down può essere realizzato come:

- **Recursive-Descent Parsing**
- **Predictive Parsing**

Analisi Sintattica a Discesa Ricorsiva

Il **Recursive-Descent Parsing** consiste in un insieme di procedure, una per ciascun non-terminale: l'esecuzione inizia con la procedura corrispondente all'assioma e termina con successo se è in grado di scandire l'intera stringa dell'istruzione correntemente in input.

Il codice di una tipica procedura non è deterministico, perché esso inizia con la scelta della procedura associata a un non-terminale, là dove non è specificato come effettuare tale scelta.

Il metodo viene reso deterministico prevedendo un ciclo che prova (backtracking) successivamente tutte le procedure alternative associate ad un non-terminale.

- Il backtracking è richiesto (se la scelta di una regola non risulta quella corretta), anche se spesso non necessario. Chiaramente, la chiave per un efficace processo di parsing è nella scelta della produzione da effettuare al passo 1.
- È una tecnica di parsing generale ma di norma non viene utilizzata, perché non è efficiente.

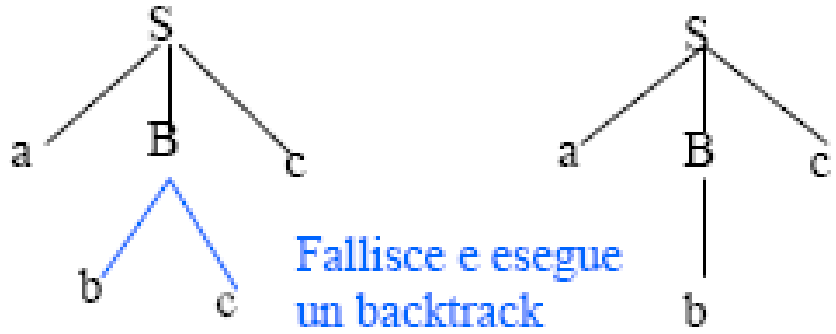
Analisi Sintattica a Discesa Ricorsiva

Recursive-Descent Parsing

È necessario il Backtracking.
Cerca di trovare una left-most derivation.

S → aBc
B → bc | b

input: **abc**



Una grammatica è "**left recursive**" se ha un non-terminale A tale che:

A → Aα per qualche stringa **α**

Una grammatica left-recursive può portare a un loop infinito un parser recursive descent (anche con backtracking): la grammatica, infatti, riscrive sempre a sinistra.
Per realizzare un parser recursive descent è quindi necessario convertire una grammatica left-recursive in una non left-recursive.

Analisi Sintattica predittiva

- il **Predictive Parsing** consiste in un recursive-descent parser che non ha bisogno di backtracking, che può essere costruito per una classe di grammatiche dette LL(1)
  - Efficiente
  - Recursive Predictive Parsing è una forma speciale di parser discendente ricorsivo senza backtracking.
  - Sono necessarie forme particolari di grammatiche

Per poter applicare parsing predittivo (senza backtracking) in molti casi basta scrivere una grammatica con attenzione, evitando ambiguità, eliminando la left-recursion e applicando il left-factoring (cfr. nel seguito).

PARSER TOP-DOWN: predictive parsing

In un parser top-down, la scelta sbagliata di una produzione implica che un parser debba riconsiderare scelte fatte in precedenza (backtracking).

È possibile utilizzare il contesto per migliorare l'efficienza?
Quanto contesto è necessario acquisire (*lookahead*)?

È possibile evitare il backtracking

- Utilizzando il lookahead per decidere quale regola applicare
- In generale, può essere necessario un lookahead arbitrariamente grande
- Importanti sottoclassi di CFG richiedono solo un lookahead limitato
- La maggior parte dei costrutti dei linguaggi di programmazione ricade in queste sottoclassi

PARSER TOP-DOWN: predictive parsing

Esempio

```
istr →    if ..... |
          while ..... |
          switch ..... |
          for .....
```

Quando troviamo il non-terminale *istr*

- se il token è uguale a **`if`** scegliamo la prima regola
- se il token è uguale a **`while`** scegliamo la seconda regola
- etc

Fattorizzazione sinistra di una grammatica

Due funzioni ci permettono di capire quale produzione scegliere, leggendo il prossimo simbolo in input: **FIRST** e **FOLLOW**.
La fattorizzazione sinistra implica che la grammatica venga trasformata in modo che ogni non-terminale con due o più alternative (produzioni) consenta al parser di scegliere facilmente una delle alternative.

Data una produzione del tipo **A ::= α | β** , il parser deve essere in grado di scegliere tra α e β.

Definiamo **FIRST(α)** come l'insieme di terminali che appaiono come primo simbolo in qualche stringa derivata da α.

Se la regola **A ::= α | β** appare nella grammatica e si verifica che:

FIRST(α) ∩ FIRST(β) = ∅

allora al parser è sufficiente il lookahead di un simbolo per espandere in modo corretto la produzione, perché tale simbolo può appartenere solo ad uno dei 2 insiemi **FIRST(α)** e **FIRST(β)**, ma non a entrambi.

PARSER TOP-DOWN: predictive parsing

Idea: perché non usare una grammatica *left-factored* per scegliere subito la regola da applicare, riconducendosi ad un parsing predittivo ed evitando così di dover effettuare numerosi backtracking?

- Analisi dell'input mediante procedure ricorsive:
  - $\forall$ nonterminale $\rightarrow$ associazione di una procedura ricorsiva
  - Discriminazione delle alternative (produzioni) in base al simbolo corrente
  - $\forall$ alternativa $\rightarrow$ spazzolamento della parte destra (lista di simboli grammaticali):
    1. Terminale $\rightarrow$ match con il simbolo corrente
    2. Nonterminale $\rightarrow$ chiamata della procedura corrispondente
- Schema della procedura associata ad $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$:

```
procedure A()
begin
  if corrente ∈ FIRST(α1) then
    spazzola α1
  else if corrente ∈ FIRST(α2) then
    spazzola α2
  ...
  else if corrente ∈ FIRST(αn) then
    spazzola αn
  else
    errore()
end;
```

Insieme dei simboli lessicali che iniziano le istanze di α_1

Omesso quando $A \rightarrow \varepsilon$ è una alternativa (in generale, quando: $\varepsilon \in \text{FIRST}(\alpha_i)$)

PARSER TOP-DOWN: predictive parsing

Problema: le produzioni con **ε** sul lato destro non sono gestite correttamente da

FIRST(α) ∩ FIRST(β) = ∅

Se **A ::= α | β** e **ε ∈ FIRST(α)**

allora ci dobbiamo assicurare che **FIRST(β)** sia disgiunto anche dai simboli terminali che possono immediatamente seguire "**A**".

Definiamo **FOLLOW(A)** l'insieme dei simboli terminali che possono apparire immediatamente dopo "**A**" in qualche stringa derivata da "**A**".

Definiamo **FIRST$^+$(A ::= α)** come

- **FIRST(α) ∪ FOLLOW(A)** se **ε ∈ FIRST(α)**
- **FIRST(α)** altrimenti

Una grammatica è LL(1) se e solo se **A ::= α | β** implica

FIRST$^+$(A ::= α) ∩ FIRST$^+$(A ::= β) = ∅

GRAMMATICHE LL(1)

Un grammatica si dice **LL(1)** se:

- fa scanning da sinistra (Left) a destra
- produce derivazioni Leftmost
- usa un solo simbolo di lookahead ad ogni passo del parsing

Tutti i parser LL(1) operano in un tempo lineare utilizzando uno spazio lineare.

LL(k): LL(1) possono essere estese con l'utilizzo di k simboli di look-ahead, in modo che attraverso l'utilizzo di più simboli le grammatiche restino predittive.

PARSER LL(1)

LL(1) Parser

- **input buffer**
  Stringa di cui effettuare l'analisi sintattica. Assumeremo che essa termini con un simbolo di fine stringa $.
- **output**
  Una produzione rappresenta un passo della sequenza di derivazioni (left-most derivation) della stringa in un input buffer.
- **stack**
  Contiene i simboli terminali del linguaggio
  Alla fine della pila c'è il simbolo speciale di fine stringa $.
  Inizialmente lo stack contiene il simbolo speciale $ e l'assioma S.
  $S ← initial stack
  Quando stack è vuoto il parsing è completato.
- **parsing table**
  E' un array bidimensionale M[A,a]
  Ogni riga contiene un simbolo non-terminale
  Ogni colonna un simbolo terminale o **$**
  Ogni entry contiene una produzione.

PARSER LL(1): parser actions

Detti
X il simbolo in cima alla pila (si noti che **X ∈ {T ∪ V ∪ $}**)
a il simbolo di ingresso

Sono possibili quattro diverse azioni:

1. **X = $** e **a = $** → il parser termina (successful completion)
2. **X = a** e **X ≠ $** → pops() e **a** = nextToken()
3. **X ∈ V** → il parser esamina M[X,a].
   a) If M[X,a] contiene una produzione $X \rightarrow Y_1 Y_2 \ldots Y_k$,
   b) {
   c) pushes $Y_k, Y_{k-1}, \ldots, Y_1$ into the stack.
   d) outputs la produzione $X \rightarrow Y_1 Y_2 \ldots Y_k$ per rappresentare un passo di derivazione.
4. nessuno dei casi precedenti → errore

Tutte le caselle vuote della parser table corrispondono ad errori
X ∈ T e **X ≠ a**, → errore

PARSER LL(1): parser actions (1/2)

Esempio

S → aBa
B → bB | ε

Si rammenti che viene applicata una left-most derivation, cioè una derivazione nella quale durante ogni passo solo il non-terminale più a sinistra è sostituito.

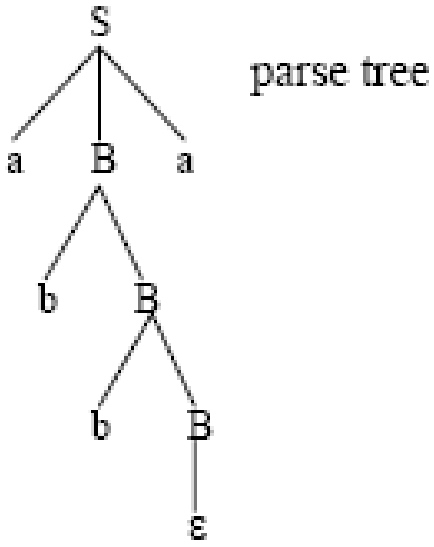
stack	input	output
$S	abba$	S → aBa
$aBa	abba$	
$aB	bba$	B → bB
$aBb	bba$	
$aB	ba$	B → bB
$aBb	ba$	
$aB	a$	B → ε
$a	a$	
$	$	

	a	b	$
S	S → aBa		
B	B → ε	B → bB	

PARSER LL(1): parser actions (2/2)

Outputs: **S → aBa** **B → bB** **B → bB** **B → ε**

Derivation(left-most): $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



parse tree

PARSER LL(1): algoritmo per la costruzione della parse table

Avendo definito:

FIRST(α) l'insieme dei simboli terminali che possono occupare la prima posizione nella stringa che si ottiene dalla derivazione di **α**, dove **α** è una stringa di simboli di T e V.
Se **α** deriva **ε**, allora **ε** ∈ FIRST(**α**).

FOLLOW(A) l'insieme dei simboli terminali che si trovano immediatamente dopo il *non-terminale A* nelle stringhe derivate dall'assioma

ne deriva che, se $ è il carattere di fine stringa,:

a ∈ FOLLOW(A) iff $S \Rightarrow^* \alpha A a \beta$
$ ∈ FOLLOW(A) iff $S \Rightarrow^* \alpha A$

e perciò l'algoritmo per la costruzione della parse table nel caso di parser LL(1) è:

∀ **A** → **α** della grammatica G

∀ a ∈ FIRST(**α**) ➔ aggiungere **A** → **α** in M[A,a]

Se **ε** ∈ FIRST(**α**) ➔ ∀ a ∈ FOLLOW(A) aggiungere **A** → **α** in M[A,a]

Se **ε** ∈FIRST(**α**) e $ ∈ FOLLOW(A) ➔ aggiungere **A** → **α** in M[A,$]

PARSER BOTTOM-UP

Anche se un front-end può effettuare la traduzione direttamente, senza costruire esplicitamente l'albero sintattico, è sicuramente conveniente descrivere l'analisi come il processo di costruzione degli alberi sintattici.

Shift-reduce (o impila-riduci) parsing

Left scan, **R**ight-most derivation: sviluppata da Knuth (1965) genera una right most derivation. Molto laborioso da realizzare ex-novo, è però adottato da molti generatori automatici di parser, tra cui YACC (BISON).

Ad ogni step di "riduzione", a partire dal simbolo più a destra, una specifica sottostringa, conforme al corpo di una derivazione, viene sostituita dal simbolo terminale alla sinistra della derivazione considerata.

In altri termini una riduzione è l'esatto opposto di un passo di derivazione.

PARSER BOTTOM-UP: Shift-reduce parsing

Esempio

La grammatica:

V = {S, A, B, C}
T = {a, b, d, e}

P = { S → aA	R1
A → BaA	R2
A → e	R3
B → aC	R4
B → AdC	R5
C → b	R6

Come costruire il parse tree di **aedbae** in modo **bottom-up**?

PARSER BOTTOM-UP

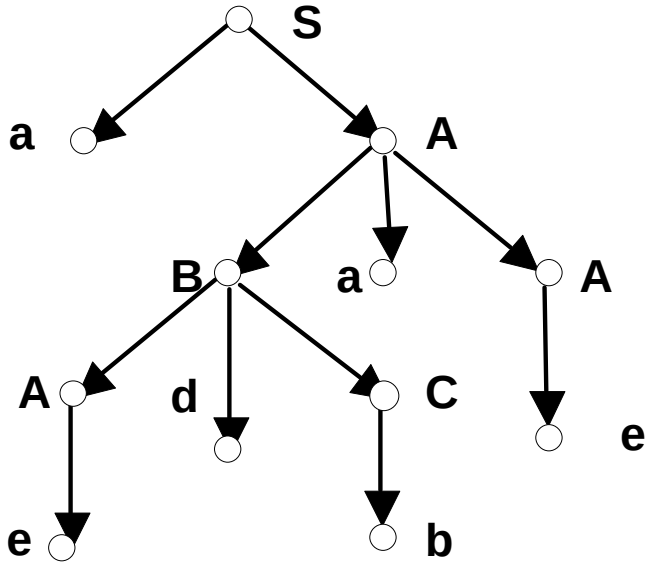
Attraverso un automa a pila che applichi in maniera inversa le regole di derivazione alla base della sintassi, operando quella che viene denominata una "**riduzione**".

stack		input	azione
$		**aedbae**$	shift **a**
$**a**		**edbae**$	shift **e**
$**ae**		**dbae**$	reduce **A** → **e**
$**aA**	!!!	**dbae**$	shift **d**
$**aAd**		**bae**$	shift **b**
$**aAdb**		**ae**$	reduce **C** → **b**
$**aAdC**		**ae**$	reduce **B** → **AdC**
$**aB**		**ae**$	shift **a**
$**aBa**		**e**$	shift **e**
$**aBae**		$	reduce **A** → **e**
$**aBaA**		$	reduce **A** → **BaA**
$**aA**		$	reduce **S** → **aA**
$**S**		$	accept

Si noti come la prima volta che sullo stack appare la stringa **aA** non viene effettuata la riduzione **S** → **aA** perché non potrebbe essere accettata l'intera stringa.

PARSER BOTTOM-UP

Il parse tree ricostruito



Nel caso specifico la successione di riduzioni è la seguente:

aedbae, **aAdbae**, **aAdbaA**, **aAdCaA**, **aBaA**, **aA**, **S**

Le stringhe in tale sequenza sono formate dalle radici di tutti i sotto-alberi presenti nell'albero della grammatica.

Durante tale analisi bottom-up, le decisioni chiave riguardano **quando** effettuare la riduzione e **quale** derivazione applicare via via che l'analisi procede.

L'assunzione di tali decisioni è guidata dall'introduzione del concetto di "**maniglia**" (handle) riportata nel seguito.

PARSER BOTTOM-UP: Shift-reduce parsing

Shift-reduce parsing: LR(k)

Un metodo di parsing bottom-up efficiente che può essere usato per un'ampia classe di grammatiche libere.

LR(k) parsing:

- **L** indica che l'input viene letto da sinistra a destra (**L**eft to right)
- **R** indica che viene ricostruita una derivazione **R**ightmost rovesciata (il non terminale più a destra viene sostituito per primo)
- **k** indica che vengono usati k simboli di lookahead (se k=1 spesso "(1)" viene omesso e quindi si parla semplicemente di LR parsing)

L'uso di uno stack nel parsing shift-reduce è giustificato dal fatto che l'handle apparirà sempre al top dello stack e mai all'interno di esso.

Vi sono, però, CFG alle quali tale parsing non può essere applicato: si pensi alla situazione in cui, noto il contenuto dello stack e il prossimo simbolo in input, non si sa decidere se effettuare un'operazione push-down o pop-up (conflitto **shift/reduce** delle grammatiche ambigue) o quale di più possibili riduzioni applicare (conflitto **reduce/reduce**).

PARSER BOTTOM-UP: Shift-reduce parsing

Vantaggi

- Può essere costruito un parser LR per tutti i costrutti dei linguaggi di programmazione per i quali può essere scritta una grammatica di tipo 2
- LR è il metodo di parsing shift-reduce senza backtracking più generale che si conosca e, nonostante ciò, può essere implementato in maniera efficiente tanto quanto altri metodi di parsing shift-reduce meno generali
- La classe di grammatiche che possono essere analizzate LR è più ampia di quella delle grammatiche che possono essere analizzate LL con un parser predittivo
- Tutti i parser LR rilevano un errore di sintassi prima possibile rispetto ad una scansione dell'input da sinistra a destra

Svantaggio

- La costruzione di un parser LR per un linguaggio di programmazione tipico è troppo complicata per essere fatta a mano. C'è bisogno di un tool apposito, un generatore di parser LR, che applichi gli algoritmi opportuni e definisca la tabella del parser. Esempi di generatori di questo tipo sono Yacc o Bison.

Questi tool sono molto utili anche perché danno informazione diagnostica: se c'è qualche problema (ad esempio, in caso di grammatica ambigua, il tool restituisce abbastanza informazione per determinare dove si crea l'ambiguità).

Il parsing bottom-up shift-reduce

Il parsing bottom-up costruisce l'albero sintattico iniziando dalle foglie e salendo verso la radice.

Si rammenti che è una "right-most derivation", nella quale durante ogni passo solo il non-terminale più a destra è sostituito.

Esempio:

Se le produzioni della sintassi sono le seguenti

$$E \rightarrow T + E \mid T$$
$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

e consideriamo la stringa: int * int + int

Parsing bottom-up

Il parsing bottom-up riduce una stringa al simbolo iniziale invertendo le produzioni:

int * int + int	$T \rightarrow$ int
int * T + int	$T \rightarrow$ int * T
T + int	$T \rightarrow$ int
T + T	$E \rightarrow T$
T + E	$E \rightarrow T + E$
E	

Il parse tree

int * int + int

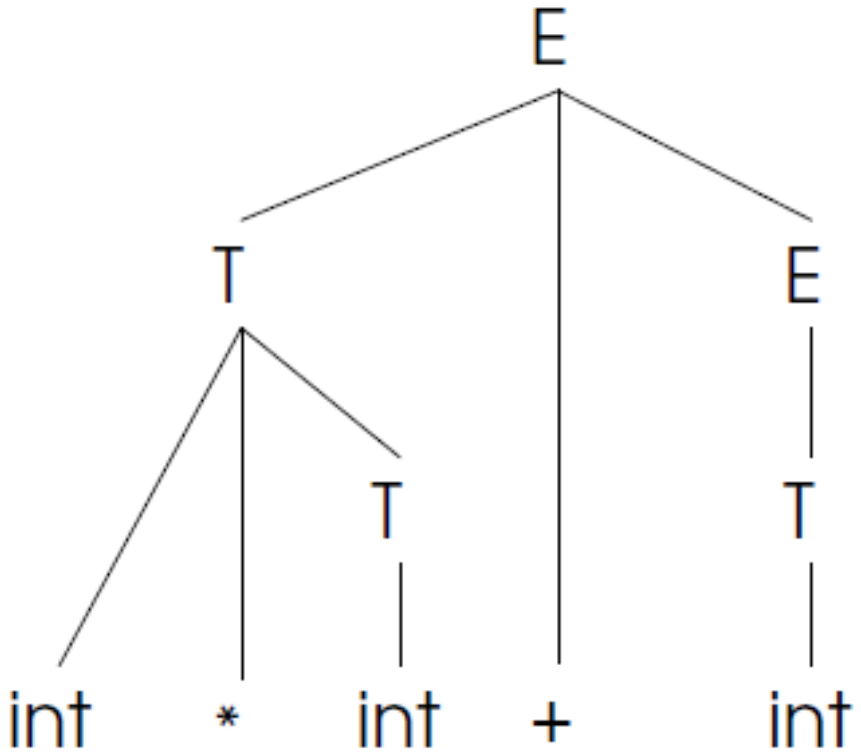
int * T + int

T + int

T + T

T + E

E



Un algoritmo per il parsing bottom-up

Sia I la stringa in input

repeat

prendi una sottostringa non vuota β di I

dove $X \rightarrow \beta$ e' una produzione

se non c'e' nessun β, torna indietro

rimpiazza β con X in I

until I = S (simbolo iniziale) o tutte le possibilita' sono state provate

Operazioni di Shift (impila) e Reduce (riduci)

Il parsing bottom-up usa solo due tipi di azioni:

Shift (scorri)

Reduce (riduci)

Handle (maniglie)

- Decisioni da prendere durante il parsing bottom-up:
  - Quale sottostringa ridurre
  - Che produzione usare

- Maniglia (handle): sottostringa β tale che $X \rightarrow \beta$ e' una produzione che corrisponde ad un passo in una derivazione da destra della stringa in input

- Nota: la sottostringa piu' a sinistra che coincide con la parte destra di una produzione non e' sempre una maniglia

Come decidere quando fare shift o reduce?

- Esempio:

  $E \rightarrow T + E \mid T$

  $T \rightarrow int * T \mid int \mid (E)$

- Consideriamo int | * int + int
  - Potremmo ridurre tramite $T \rightarrow int$ arrivando a T | * int + int
  - Errore: non potremmo più arrivare al simbolo iniziale E

Maniglie

- Intuizione: Vogliamo ridurre solo se il risultato può ancora essere ridotto fino al simbolo iniziale

- Prendiamo una derivazione da destra:

$S \rightarrow^* \alpha X \omega \rightarrow \alpha\beta\omega$

  - Allora $\alpha\beta$ e' una maniglia di $\alpha\beta\omega$

In un parsing shift-reduce, le maniglie appaiono solo in cima alla pila, mai nel mezzo

Perche'?

- Induzione informale sul numero di passi di riduzione:
  - All'inizio e' vero, e la pila e' vuota
  - Dopo aver ridotto una maniglia
    - Il nonterminale piu' a destra e' in cima alla pila
    - La prossima maniglia deve essere alla destra del nonterminale piu' a destra, perche' e' una derivazione da destra
    - La sequenza di mosse shift raggiunge la prossima maniglia

Sommario sulle maniglie

- In un parsing shift-reduce, le maniglie appaiono sempre in cima alla pila
- Le maniglie non sono mai alla sinistra del nonterminale piu' a destra
- Quindi non dobbiamo mai muovere verso sinistra il simbolo |
- Gli algoritmi di parsing bottom-up si basano sul riconoscimento delle maniglie

Riconoscere le maniglie

- Non ci sono algoritmi efficienti per riconoscere le maniglie
- Soluzione: usare delle euristiche che cercano di indovinare quali pile sono delle maniglie
- Per alcune grammatiche (grammatiche SRL = LR(1)), le euristiche indovinano sempre correttamente

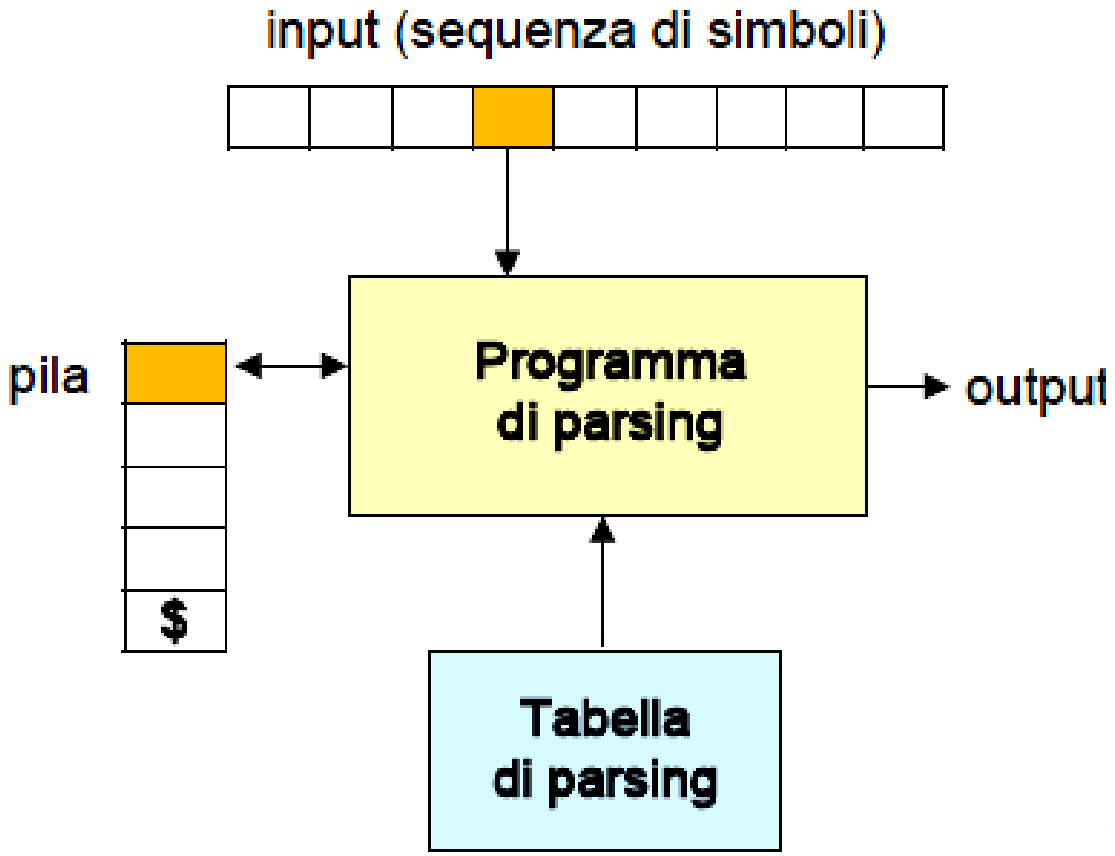
Shift-Reduce Parsing

Il programma di parsing shift-reduce è una forma di analisi bottom-up in cui uno stack contiene i simboli della grammatica e un buffer di input contiene il resto della stringa da analizzare. In altri termini altro non è che un **automa a pila** e la tabella di parsing in base a cui esso opera non fa che riportare la funzione di transizione di stato che guida il funzionamento dell'automa.

Le possibili operazioni che l'automa può compiere sono:

- **shift** per fare lo shift sul top dello stack del prossimo simbolo in input
- **reduce** per operare la riduzione della parte destra della stringa che si trova al top dello stack; ciò implica di decidere con quale non terminale ridurre e di localizzare la parte sinistra
- **accept** per operare una accettazione, cioè la sostituzione di tutta la stringa con l'assioma
- **error** per indicare la rilevazione di un errore

Shift-Reduce Parsing: Il programma o automa



Costruzione dell'albero

L'esempio con le due operazioni Shift-Reduce

|int * int + int	shift
int | * int + int	shift
int * | int + int	shift
int * int | + int	reduce T → int
int * T | + int	reduce T → int * T
T | + int	shift
T + | int	shift
T + int |	reduce T → int
T + T |	reduce E → T
T + E |	reduce E → T + E
E |	

Costruzione dell'albero

In dettaglio (1)

|int * int + int

int * int + int

↑

Costruzione dell'albero

In dettaglio (2)

|int * int + int

int | * int + int

int * int + int

↑

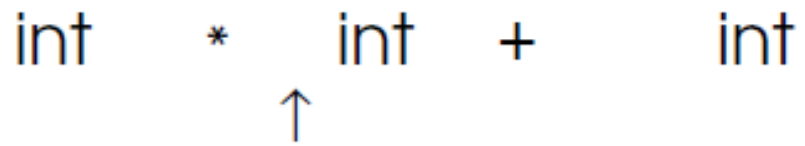
Costruzione dell'albero

In dettaglio (3)

|int * int + int

int | * int + int

int * | int + int



Costruzione dell'albero

In dettaglio (4)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * int + int

↑

Costruzione dell'albero

In dettaglio (5)

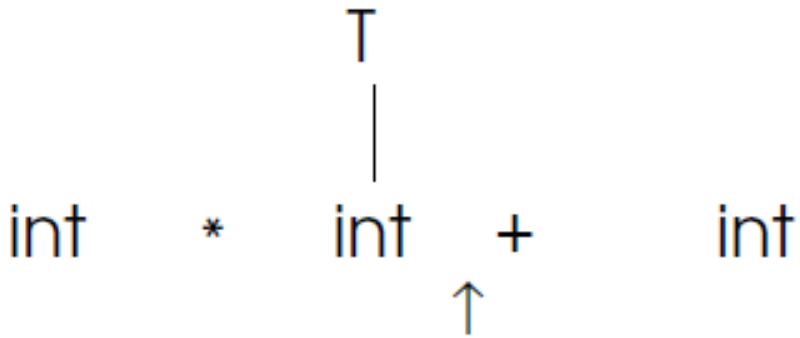
|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int



Costruzione dell'albero

In dettaglio (6)

|int * int + int

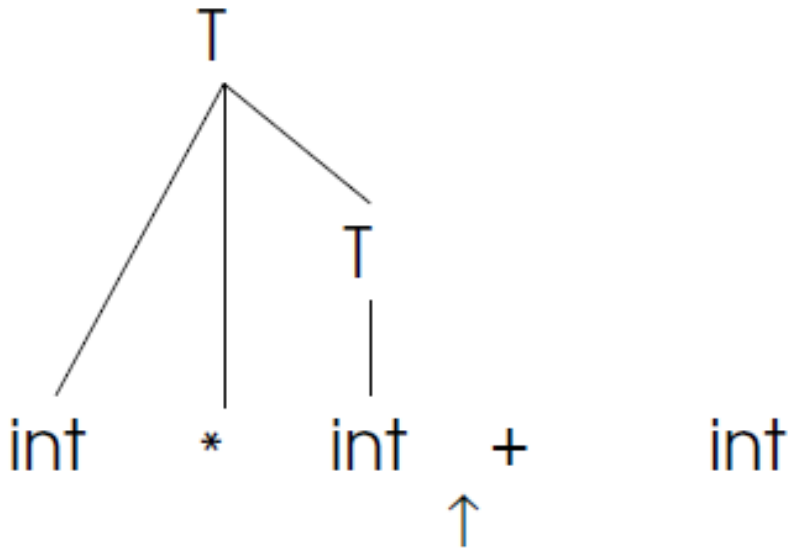
int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int



Costruzione dell'albero

In dettaglio (7)

|int * int + int

int | * int + int

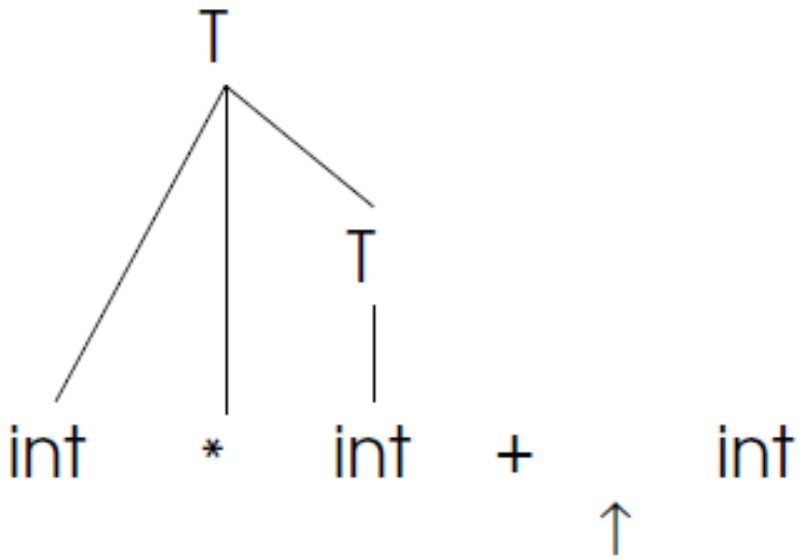
int * | int + int

int * int | + int

int * T | + int

T | + int

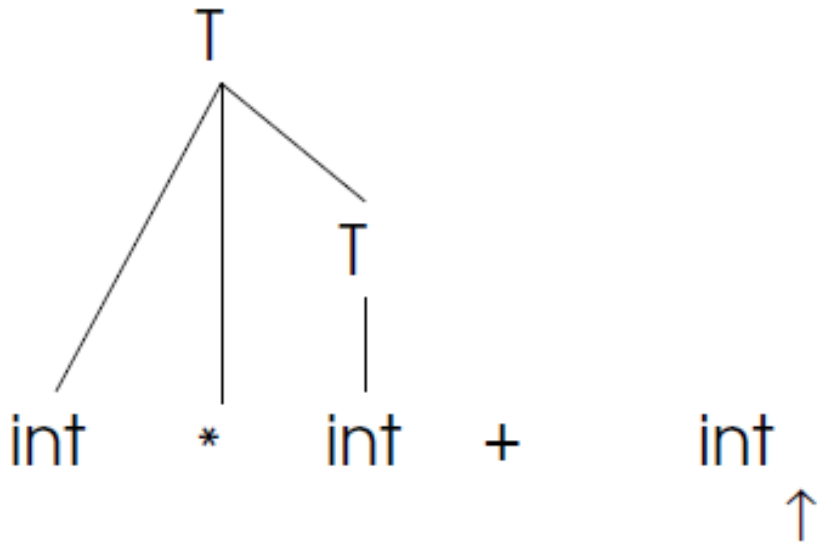
T + | int



Costruzione dell'albero

In dettaglio (8)

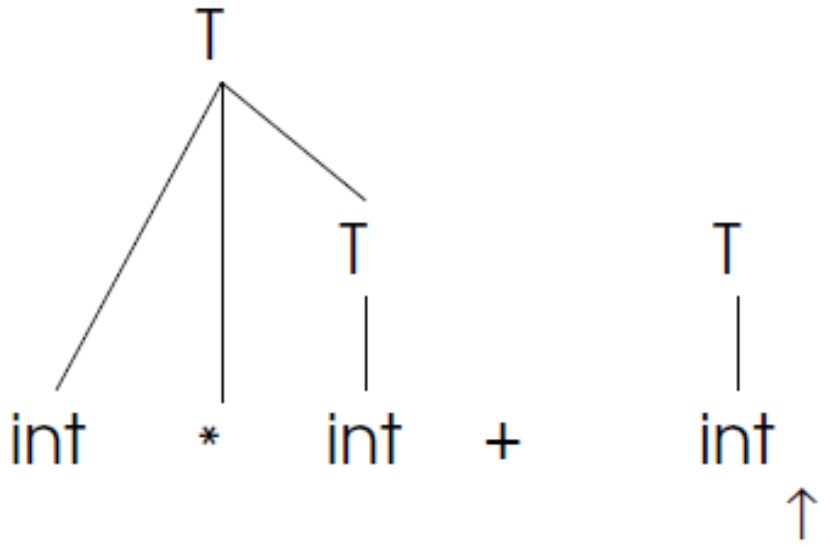
|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |



Costruzione dell'albero

In dettaglio (9)

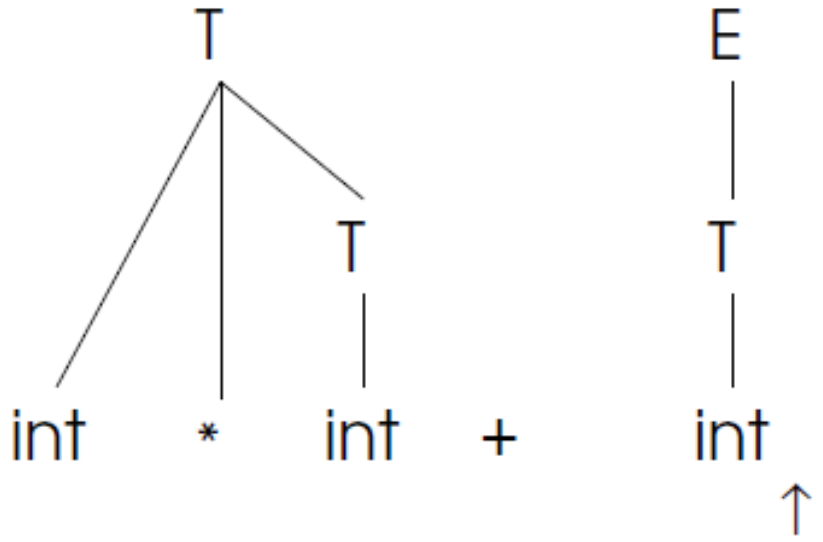
|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |



Costruzione dell'albero

In dettaglio (10)

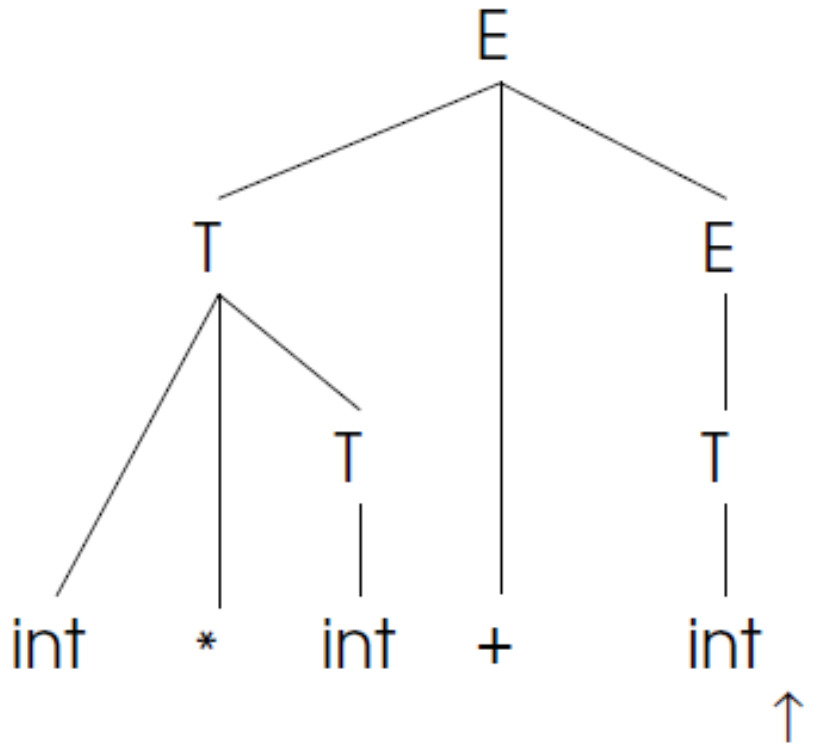
|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
T + E |



Costruzione dell'albero

In dettaglio (11)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
T + E |
E |

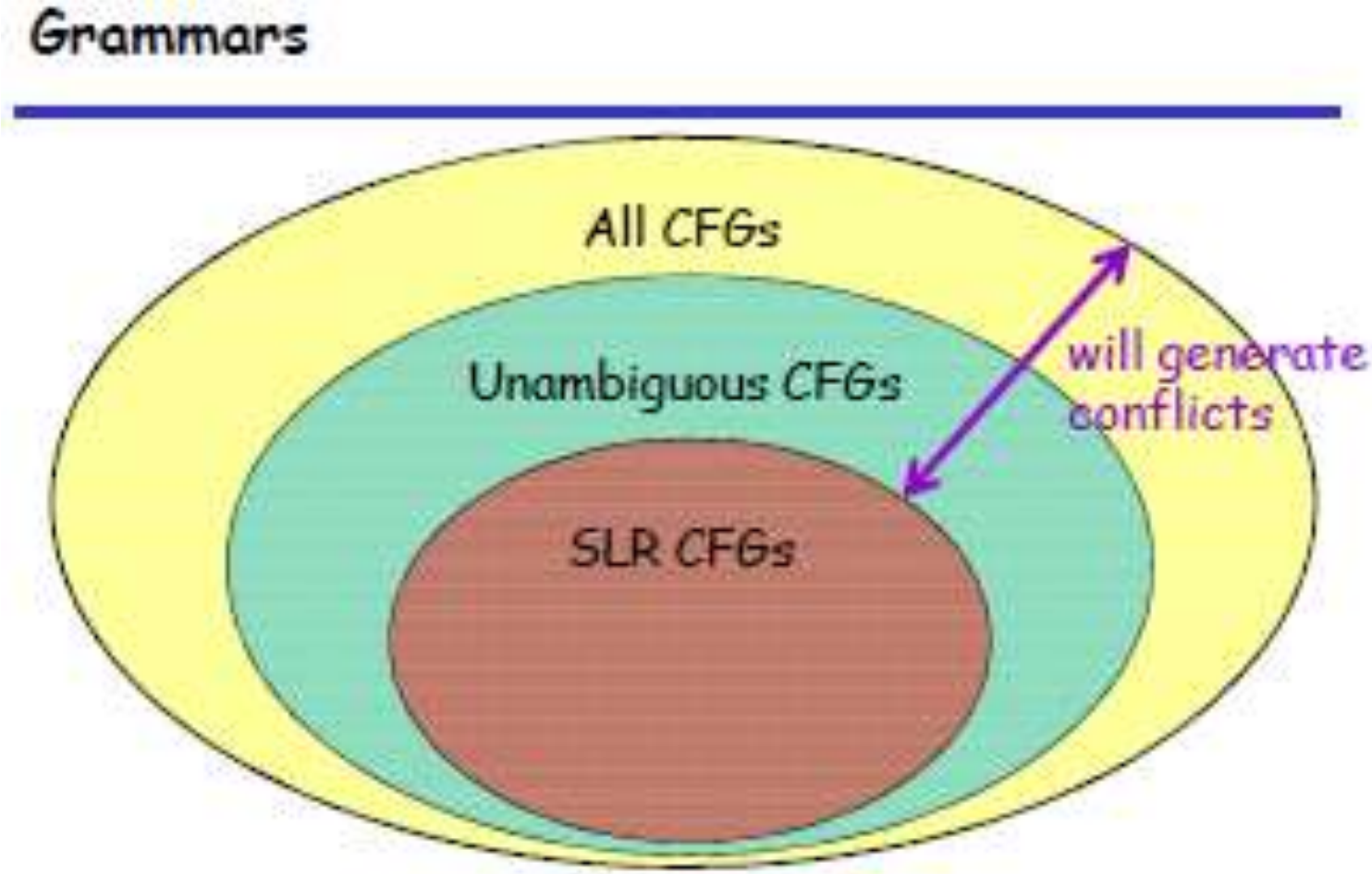


Conflitti

- Il parsing shift-reduce non funziona con tutte le grammatiche libere da contesto
- Per alcune grammatiche, qualunque parser shift-reduce arriva a dei conflitti, anche conoscendo tutta la pila e i prossimi k simboli in input
  - **Conflitto shift-reduce**: il parser non sa decidere se fare uno shift o una reduce
  - **Conflitto reduce-reduce**: il parser non sa quale riduzione fare
- Le grammatiche in questione non sono LR(k) (esse sono dette non-LR); una grammatica ambigua e' non-LR
- **Di solito i compilatori usano, invece, grammatiche LR(1)**

Simple LR parsing (SLR)

Anche se l'analizzatore LR viene costruito facendo uso di un tool automatico, è opportuno conoscere i concetti di base di tale analisi, almeno nel caso dei più semplici parser di questo tipo.

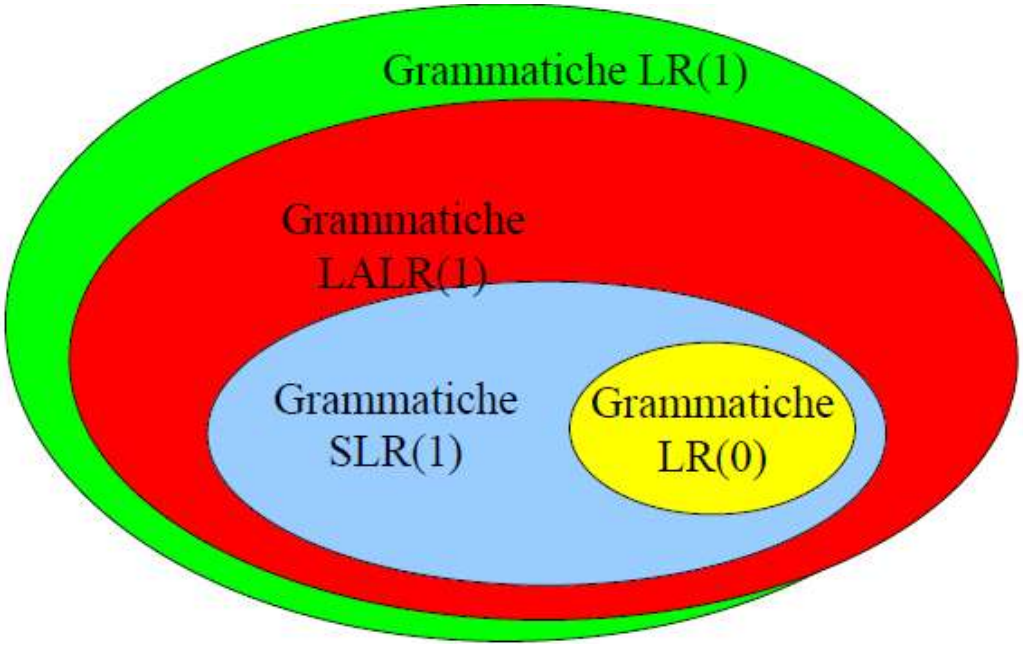


PARSER BOTTOM-UP: Shift-reduce parsing

Introduciamo l'algoritmo generale eseguito da un parser LR(K), K=0 oppure K=1.

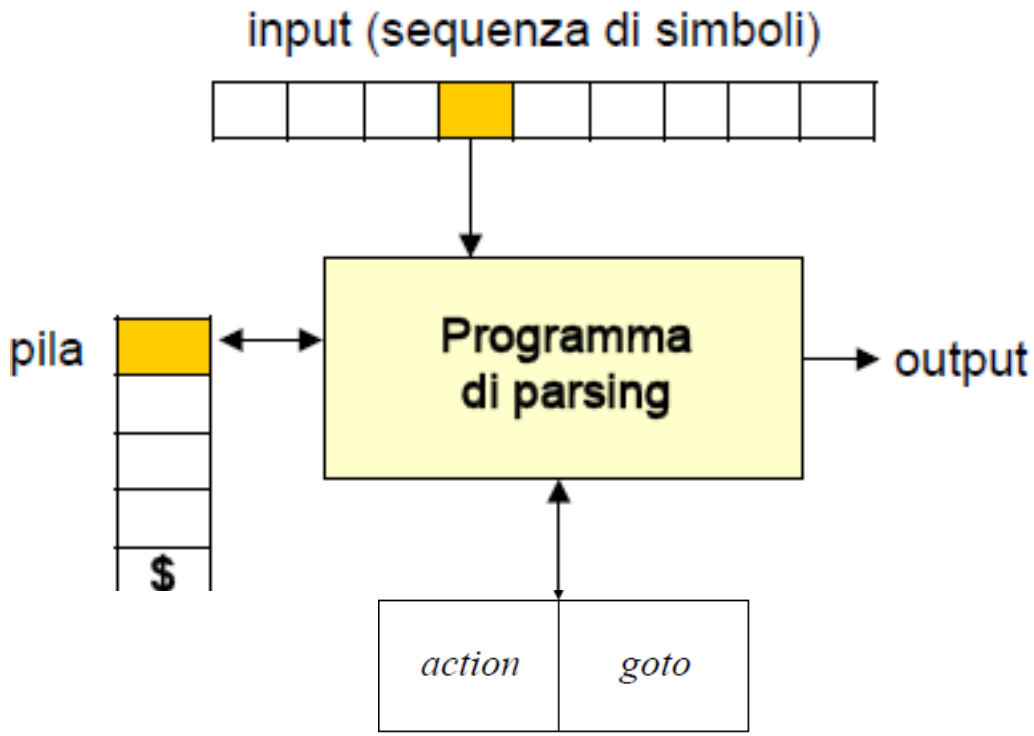
Esaminiamo il metodo più semplice per la costruzione della tabella di un parser LR, che viene detto simple LR (abbreviato in SLR)

Considerando il metodo LR canonico (che è il metodo più potente, ma anche il più costoso, per costruire la tabella) e semplificandolo un po', si può definire il metodo lookahead LR (abbreviato come LALR) che si trova ad un livello intermedio fra gli altri due.



Il parser LR

Lo schema di un parser LR è quello riportato di seguito, in cui il programma driver fa uso di una parsing table costituita da due parti (**action** e **goto**).



Il programma di parsing è lo stesso per tutti i parser LR; solo la parsing table cambia da parser a parser.

Il parser LR

La pila contiene una sequenza di stati $s_0\ s_1\ s_2\ \ldots\ s_{m-1}\ s_m$, dove s_m è sul top.

Ogni stato rappresenta un insieme di elementi che indicano quanta parte di una produzione è stata già riscontrata all'attuale punto dell'analisi.

Per costruzione ad ogni stato, eccezion fatta per lo stato 0, è associato uno ed un solo simbolo della grammatica, mentre non è vero il contrario.

L'input rimanente, ancora da analizzare, è dato da $a_i\ a_{i+1}\ \ldots\ a_n\ \$$.

In tal caso il parser ha la **configurazione** rappresentata dalla coppia

$$s_0\ s_1\ s_2\ \ldots\ s_{m-1}\ s_m,\ a_i\ a_{i+1}\ \ldots\ a_n\ \$$$

Invece di un simbolo, il parser fa shift di uno stato. Ogni stato riassume l'informazione contenuta nella parte sottostante dello stack.

Ad ogni passo il parser decide se fare shift o reduce, in base allo stato che si trova in cima allo stack e al simbolo di lookahead corrente.

La parsing table: la funzione **action**

La parsing table è costituita, come anzidetto, da due funzioni (**action** e **goto**).

La funzione **action** ha due argomenti: uno stato **i** e un terminale **a** (o **$**, il carattere di fine della stringa di input). Il valore di **action[i, a]** può essere:

- **Shift j** per fare lo shift di **a** (prossimo simbolo nell'input) sul top dello stack, anche se viene usato lo stato **j** per rappresentare **a**
- **Reduce** $A \rightarrow \beta$ per operare la riduzione di β (la parte destra della stringa che si trova al top dello stack) con il non terminale **A**
- **Accept** per accettare l'input (sostituzione della stringa con l'assioma) e terminare l'analisi corrente
- **Error** per indicare la rilevazione di un errore e avviare un'opportuna azione

La parsing table: la funzione **goto**

La funzione **goto**, definita su insiemi di elementi ed estesa agli stati, ha la seguente forma:

$$\mathbf{goto}(I_i, A) = I_j$$

che ha il significato che c'è una transizione dallo stato i (con non terminale **A**) allo stato j.

Da quanto visto deriva che la configurazione

$$s_0\ s_1\ s_2\ \ldots\ s_{m-1}\ s_m,\ a_i\ a_{i+1}\ \ldots\ a_n\ \$$$

corrisponde alla forma sentenziale destra

$$X_1\ X_2\ \ldots\ X_{m-1}\ X_m,\ a_i\ a_{i+1}\ \ldots\ a_n\ \$$$

dove il generico X_i corrisponde al simbolo della grammatica rappresentato dallo stato s_i.

Si noti che s_0 non rappresenta un simbolo della grammatica, ma indica solo il fondo dello stack.