LALR

26/10/2012

LALR
Look Ahead

più usato non tutti i linguaggi sono SLR

$LL(1) \to LL(k)$
$SLR(1) \to SLR(k)$

G: $S \to AB \mid ac \mid xAc$
$A \to a$
$B \to b \mid \varepsilon$

- not ambiguous
- finite
- $L(G) = \{a, ab, ac, xac\}$

	FIRST	FOLLOW
S	a x	$
A	a	c b $
B	b ε	$

I_0:
$S' \to .S$
$S \to .AB$
$S \to .ac$
$S \to .xAc$
$A \to .a$

$goto(I_0, a) = I_2$:
$S \to a.c$
$A \to a.$

$goto(I_2, c) = I_6$

shift/reduce conflict!!

$T[2, c] = \{$ Shift 6 / Reduce "$A \to a$" $\}$

So G is not SLR (even if the grammar is really simple!)

We could "fix" the grammar

G': $S \to A_1B \mid ac \mid xA_2c$
$A_1 \to a$
$A_2 \to a$
$B \to b \mid \varepsilon$

the same A was used for two different purposes
Now it's clear of which A we are talking about.

	FIRST	FOLLOW
S	a x	$
A_1	a	b $
A_2	a	c
B	b ε	$

I_0:
$S' \to .S$
$S \to .A_1B$
$S \to .ac$
$S \to .xA_2c$
$A_1 \to .a$

$goto(I_0, a) = I_2$:
$S \to a.c$
$A_1 \to a.$

$T[2, c] =$ shift 6

(c is no more in the follow of A_1)

Efficient LALR (canonical LR is more powerful but really more expensive and poorer)

SLR $\Rightarrow$ $A \to \alpha . \beta$ LR(0)

↖ IS NOT THE LOOKAHEAD AS LL(1)

LALR $\Rightarrow$ $[A \to \alpha . \beta , X]$ LR(1)

↑ terminal or $ (extra information Lookahead)

We need to redefine CLOSURE and GOTO to take into account the fact that now items are different

CLOSURE(I) (for LR(1) items)

repeat foreach $[A \to \alpha . B\beta , x] \in I$

foreach $B \to \gamma$ in G'

foreach $b \in FIRST(\beta x)$

add $[B \to . \gamma , b]$ to I

until saturation

return I

GOTO(I, X) {

$J = \emptyset$

foreach $[A \to \alpha . X \beta , x] \in I$

add $[A \to \alpha X . \beta , x]$ to J

return J

}

CONSTRUCTION OF LALR LOOKAHEAD PROPAGATION GRAPH

Kernel LR(0) item: $S' \to .S$ and any other LR(0) item where the dot occurs in some position different from the very beginning

$A \to \alpha .$ } Kernels
$A \to \alpha . \beta$ }

$A \to . \alpha$ ($A \neq S'$) is **NOT** kernel

nodes given by pairs of the shape $(I_J, \text{Kernel LR(0) item in } I_J)$ for every I_J, and for every kernel item in I_J

labels (of the nodes) each node v has an attached $fw(v)$:
at the beginning: $fw((I_0, S' \to .S)) = \{\$\}$
and $fw(v) = \phi$ for all other nodes

edges (and the updated values of $fw(v)$)

```
Foreach  v = (I_K , A -> α.β)
    let # be a new symbol                          "Fake" lookahead
    compute J = CLOSURE( { [A -> α.β , #] } )
    foreach [B -> γ.Yδ, x] ∈ J
        let v' = (goto(I_K , Y) , B -> γY.δ)
        if x = #
            add an edge from v to v' // PROPAGATION
        else
            add x to fw(v') // SPONTANEOUS GENERATION OF LOOKAHEADS
```

CONSTRACTION OF KERNEL LALR(1) items

1. Compute the kernels of the LR(0) set of items (or directly all the collection of LR(0) items, then take the kernels)

2. Construct the lookahead propagation graph

3. evaluate graph:

```
repeat
    saturation := true
    foreach edge(v, v') do
        temp := fw(v')
        fw(v') := fw(v') ∪ fw(v)
        if fw(v') ≠ temp
            saturation := false
until saturation
```

We're ready for LALR parsing table

SLR: - fill in the shift entries))=> in LALR is the same

- fill in the accept
- fill in the reduction))=> in LALR we use info from propagation graph

- check conflicts
- goto's

30/10/2012

in SLR, parsing table construction

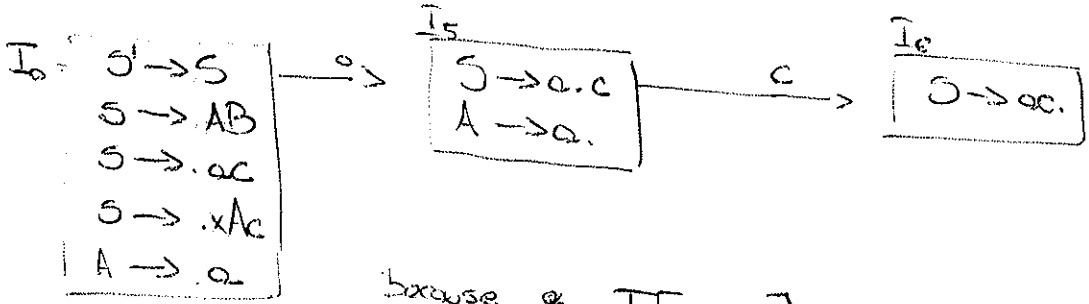
- if $A \to \alpha . a \beta \in I_k$ and $goto(I_k, a) = I_j$ then $T[k, a] = shift\ j$
- if $A \to \alpha . \in I_k$ with $A \neq S'$ then $\forall x \in Follow(A)\ T[k, x] = reduce\ "A \to \alpha"$

$G: S \to AB \mid ac \mid xAc$

$A \to a$

$B \to b \mid \varepsilon$

G is not ambiguous, $L(G)$ is composed of 4 word configs, each of them has a unique derivation tree



because * $T[S, c] = S_6$

because **
$c \in Follow(A)$ $T[S, c] = R\ "A \to a"$

LR(0)
$A \to \alpha . \beta$

LR(1) Length of Lookahead
$[A \to \alpha . \beta, l]$

closure(I) = { repeat { foreach $[A \to \alpha . B \beta, x] \in I$
do foreach $B \to \gamma \in G'$
do foreach $b \in first(\beta x)$
do add $[B \to . \gamma, b]$ to I }
while satisfaction
}
return I

G: $S \to AB \mid ac \mid xAc$
$A \to a$
$B \to b \mid \varepsilon$

	FIRST	
S	a x	
A	a	
B	b ε	

I_0:

$S' \to .S \; , \$$
$S \to .AB \; , \$$
$S \to .ac \; , \$$
$S \to .xAc \; , \$$
$A \to .a \; , b$
$A \to .a \; , \$$

(sul libro sostituito for $A \to .a \; , b/\$$)

$I_0 \xrightarrow{a}$

$S \to a.c \; , \$$
$A \to a. \; , b$
$A \to a. \; , \$$

the reduction step when constructing the LR(1) table would read this way

if $[A \to \alpha. \; , \ell] \in I_k$ with $A \notin S'$
then $T[k, \ell]$ = reduce "$A \to \alpha$"

RIPETIZIONE

GOTO for LALR

```
GOTO(I, X) {
    J = ∅
    foreach [A -> α.Xβ , x] ∈ I
        add [A -> αX.β , x] to J
    return J
}
```

CONSTRUCTION OF THE LOOKAHEAD PROPAGATION GRAPH

RIPETIZIONE

S' --> .S , #
S --> .AB , $
S --> .aC , $
S --> .xAc , #
A --> .a , b
A --> .a , $

nodes ~~pairs~~ are of the shape (state, items)

$I_0, I_1, \ldots$ $A \to \alpha.\beta$
LR(0) item

states are taken from the collection of set of LR(0) items for G
items are the so called kernel items contained in the states,
namely $S' \to .S$ and any other item where the "dot" does not occur at the very beginning of the body of the production

OBS $B \to .$ is kernel, read the dot as if it is at the end of the body

labels of nodes $\forall$ node v set $f_w(v) = \phi$;
set $f_w($ (initial state, $S' \to .S)) = \{\$\}$

foreach $v = (I_h, A \to \alpha.\beta)$ do
{ compute $J = \text{closure}(\{[A \to \alpha.\beta, \chi]\})$ with χ is a new symbol;
foreach $[B \to \gamma., x] \in J$ do {let $v' = (I_k, B \to \gamma.)$;
if $x = \chi$ then add edge (v, v') % propagates
else add x to $f_w(v')$ % generating }

foreach $[B \to \gamma.Y\delta, x] \in J$ do
{let $v' = (goto(I_k, Y), B \to \gamma Y.\delta)$;
if $x = \chi$ then add edge (v, v')
else add x to $f_w(v')$ }
}

OBS let $v=(s, B \to \gamma . a \delta)$

$closure(\{[B \to \gamma . a\delta, \&]\}) = \{[B \to \gamma . a\delta, \&]\}$

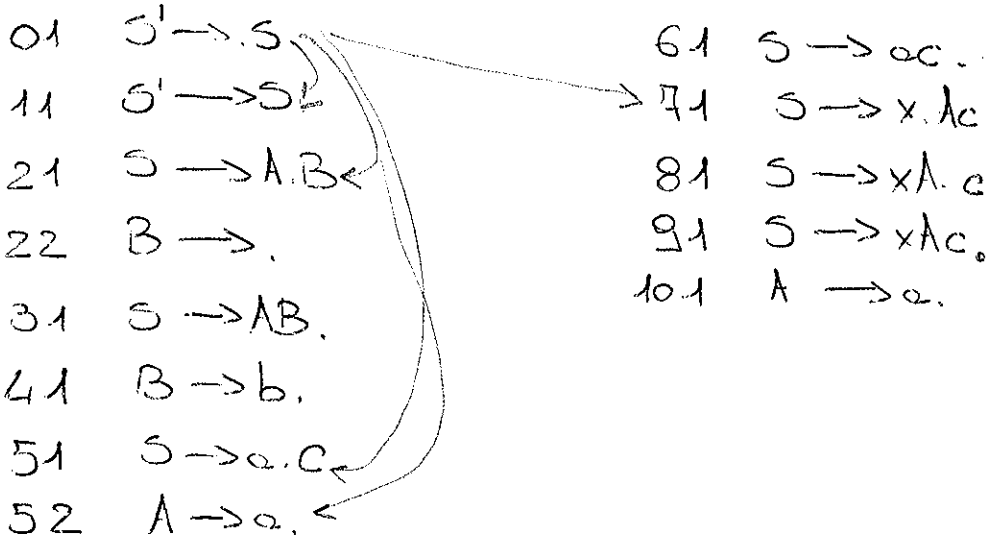
then v always propagates (and only propagates) to $(goto(s,a), B \to \gamma a . \delta)$

OBS Let $v=(s,i)$ with $i = B \to \alpha .$

$closure(\{[i, \&]\}) = \{[i, \&]\}$

and no goto ~~is~~ defined from i

then there is nothing to do



NOTE

$closure(\{[A \to \beta . , \&]\}) = \{[A \to \beta . , \&]\}$

↑ for every one that ends with the dot I do nothing

$closure(\{[A \to \alpha . a\beta , \&]\}) = \{[A \to \alpha . a\beta , \&]\}$

if after the dot there is a non terminal I do propagation

EXAMPLE 5, $S \to a . c$

add an edge to 6, $S \to ac.$

01 S' → .S , $
S → .AB , $
S → .ac , $
S → .xAc , $
A → .a , $
A → .a , b

← CLOSURE di 01 Propagati. on

from	to
01 :	11, 21, 51, 71, 52
21 :	31, 41, 22
51 :	61
71 :	81

81 : 91

generati. ons

b generated by 01 for 52
c generated by 71 for 101

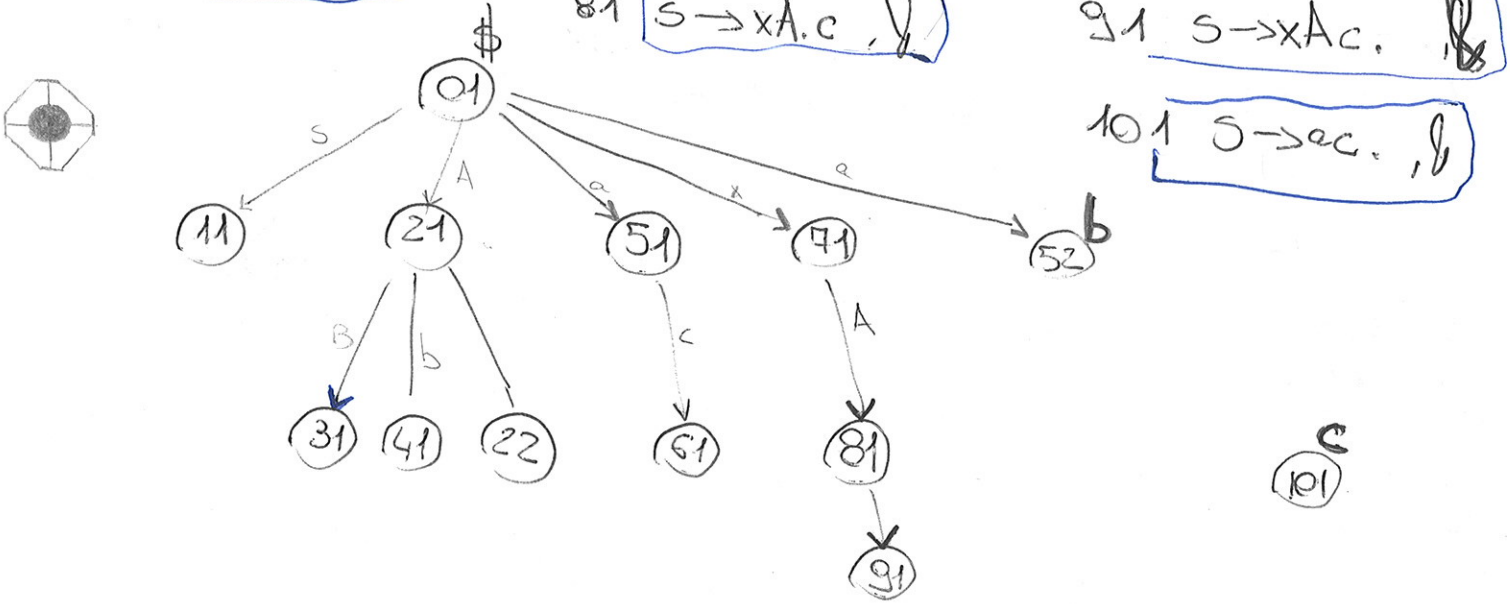
11 S' → S. , $ ← CLOSURE di 11

21 S → A.B , $ ← CLOSURE di 21
B → .b , $
B → .ε , $

22 B → . , $
31 S → AB. , $
41 B → b. , $
51 S → a.c , b
61 S → ac. , b

71 S → x.Ac , $
A → .a , c

81 S → xA.c , $
91 S → xAc. , $
101 S → ac. , b



6/11/2012

LALR

- collections of sets of LR(0) items (as in the case of SLR)
- kernels of LR(0) items
- bookkeeping propagation graph (construction & execution)
- check the set $fw(v)$ for the nodes v such that $v=(i, B\to\alpha.)$

LALR Parsing Table

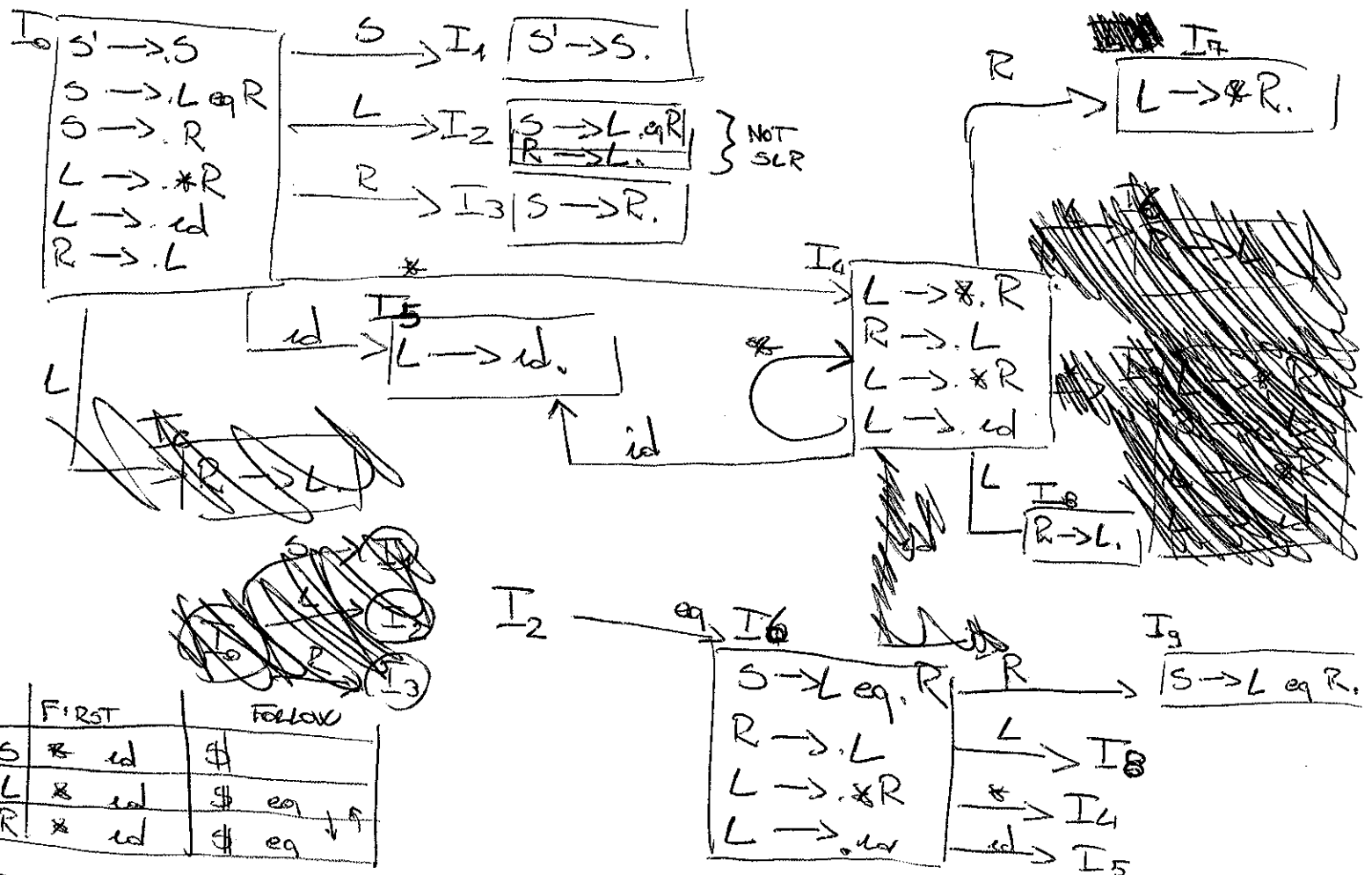
same alg. as for SLR with one single exception relative to reduction entries

"IF $B\to\alpha. \in I_i$ with $B \neq S'$
and $x \in fw((i, B\to\alpha.))$ then $T[i,x] = R$"$B\to\alpha$" "

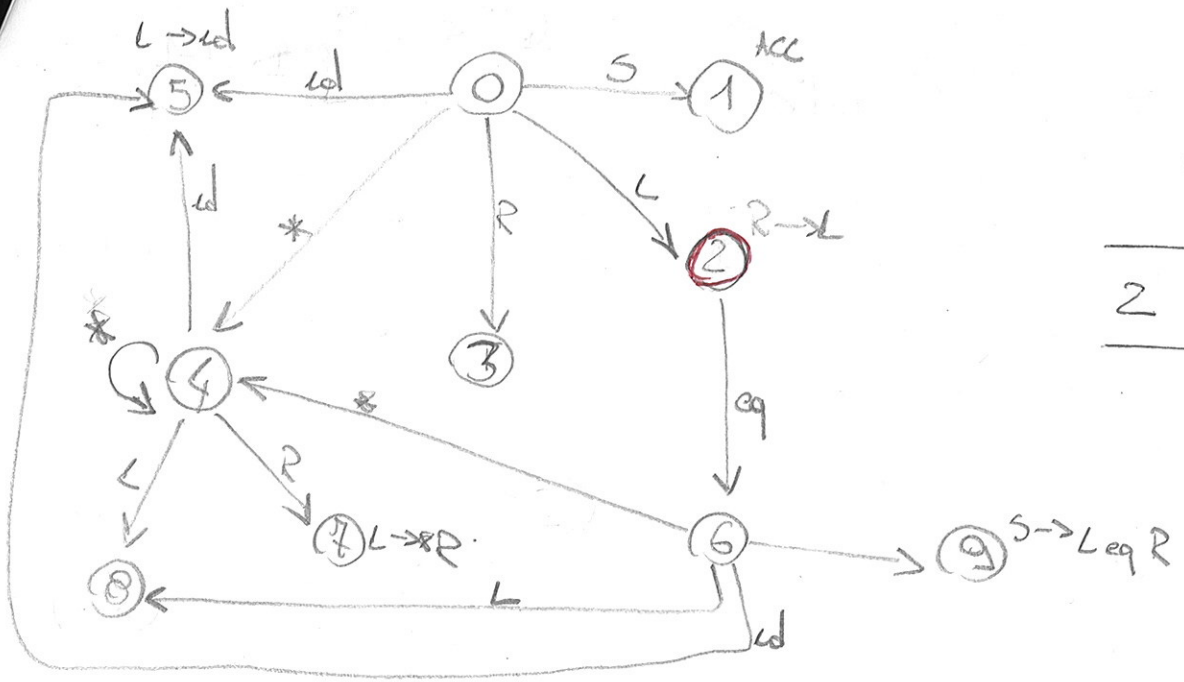
EXERCISE

$S \to L$ eq R
$S \to R$
$L \to *R$
$L \to$ id
$R \to L$

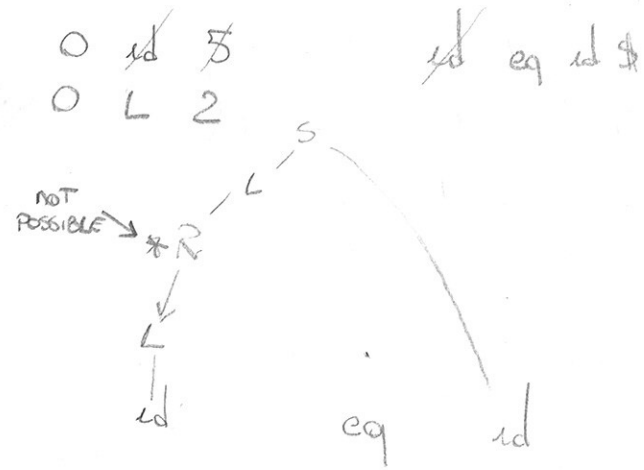
say wether it's SLR ?



	FIRST	FOLLOW
S	* id	$
L	* id	$ eq
R	* id	$ eq



	eq
2	S_6 r: "R → L"



0 1 S' → .S
1 1 S' → S'.
2 1 S → L. eq R
2 2 R → L.
3 1 S → R.
4 1 L → *. R
5 1 L → id.
6 1 S → L eq. R
7 1 L → *R.
8 1 R → L.
9 1 S → L eq R.

0 1	S' → .S, $
	S → .L eq R, $
	S → .R, $
	L → .*R, eq/$
	L → .id, eq/$
	R → .L, $

01 propagates to

11, 21, 31, 41, 51, 22

61 propagates to

91, 81, 41, 51

21 propagates to

61

01 eq is generated for 41, 51

41 propagates to

71

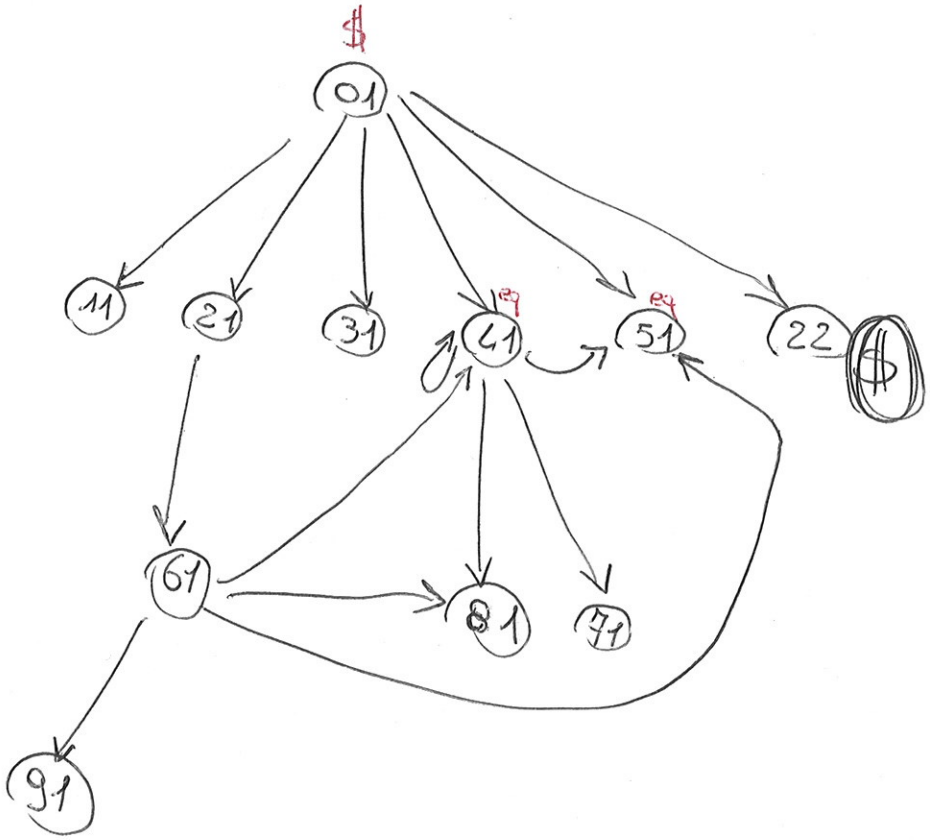
81

41

51

41
- L → *.R, $
- R → .L, $
- L → .*R, $
- L → .id, $

61
- S → L eq.R, $
- R → .L, $
- L → .*R, $
- L → .id, $



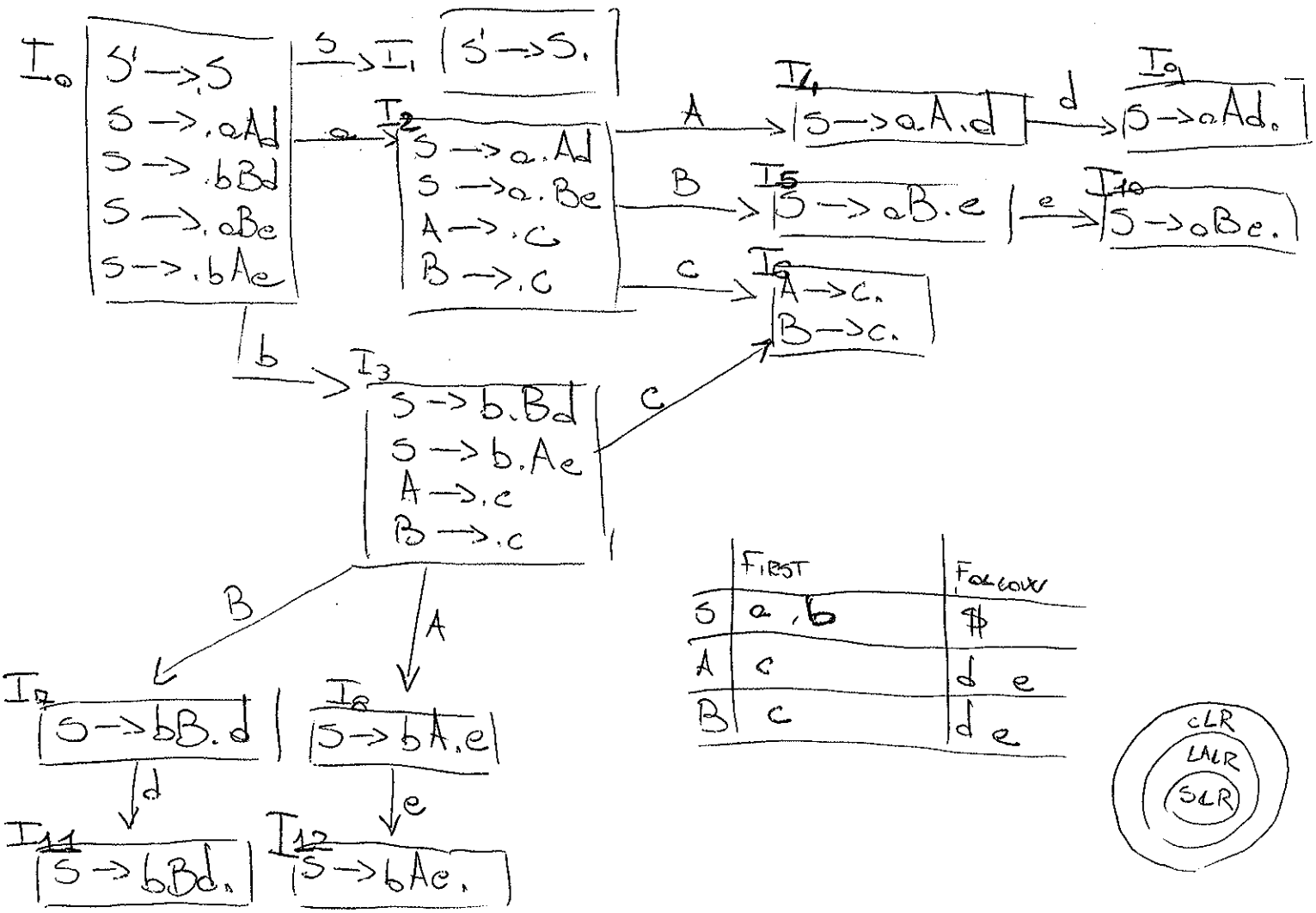
7/11/2012

$S \to aAd \mid bBd \mid aBe \mid bAe$

$A \to c$

$B \to c$

S
|



	FIRST	FOLLOW
S	a, b	$
A	c	d e
B	c	d e

	a	b	c	d	e	$	S	A	B
0	s_2	s_3					1		
1						acc			
2			s_6					4	5
3			s_6					8	7
4				s_9					
5					s_{10}				
6				r: A→c r: B→c	r: A→c r: B→c				
7									
8									
9									
10									
11									
12									

Ex

$S \to aaAaa \mid abAba \mid T \mid aca \mid bcb \mid c$

$T \to bbBbb \mid baBab \mid S$

$A \to S \mid aca \mid bcb \mid c$

$B \to T \mid aca \mid bcb \mid c$

$wcw^R \mid w = (a \text{ or } b)^*$

$\mathcal{L}(G)$? G is LL(1)?

$\mathcal{L}(G) = \{ w c w^R \mid w = (a \mid b)^* \}$

G is AMBIGUOUS, so $\mathcal{L}$ is NOT LL(1)

Ex

$\mathcal{L} = \{ a^i b^j \mid 0 \leq j < i \}$

- define non ambigue grammar G such that $\mathcal{L}(G) = \mathcal{L}$
- define a LALR grammar G such that $\mathcal{L}(G) = \mathcal{L}$

~~$S \to aA$~~

~~$A \to aAb \mid aA \mid \varepsilon$~~

AMBIGUOUS

$G: S \to aSb \mid A$

$A \to aA \mid a$

$S \to a^k S b^k \to a^k A b^k \to a^k a^m b^k$

$a^m, m > 0$

9/11/2012

$S \to AB$

$A \to aA \mid a$

$B \to aBb \mid \varepsilon$

read a_1

shift a_1 on the stack

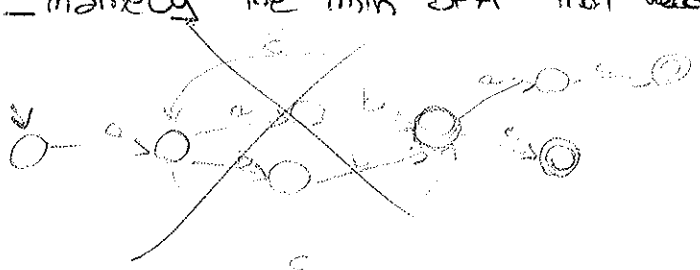
read a_2 — shift / reduce

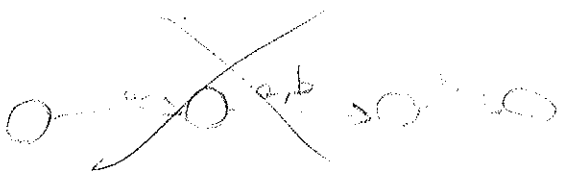
$S \to aS \mid aB$

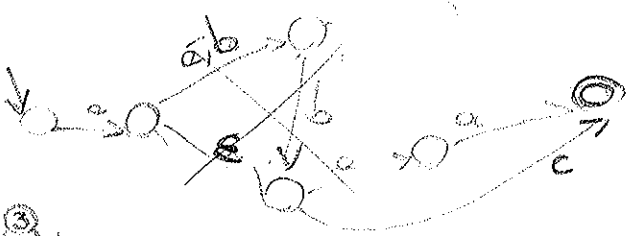
$B \to aBb \mid \varepsilon$

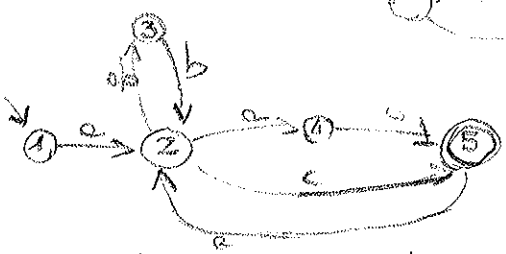
Let $r = r_1 r_1^*$ with $r_1 = (a\ (ab \mid bb)^*\ (aa \mid c))$

- provide the min DFA that recognizes the complement of $\mathcal{L}(r)$ over the alphabet $\{a, b, c\}$
- namely the min DFA that recognizes $\mathcal{L}((a|b|c)^*) \setminus \mathcal{L}(r)$

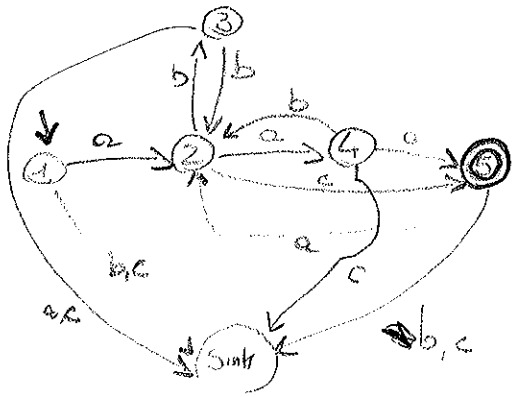






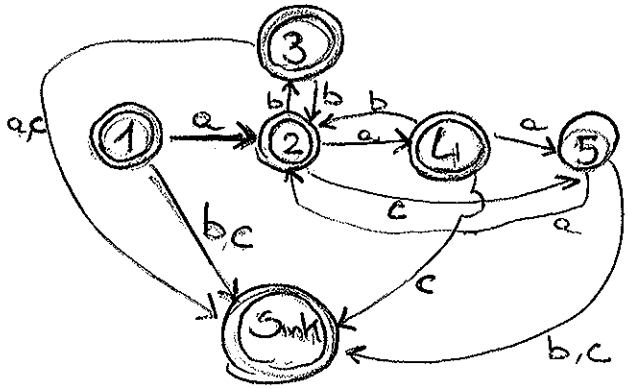


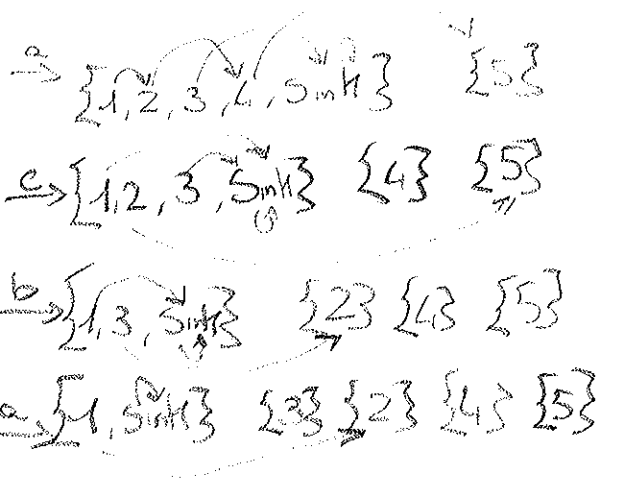
NON ancora corretto



Determinizzato

Trasformo gli stati finali in non finali, e i non finali in finali





G: B → B or B | B and B | not B | (B) | tt | ff

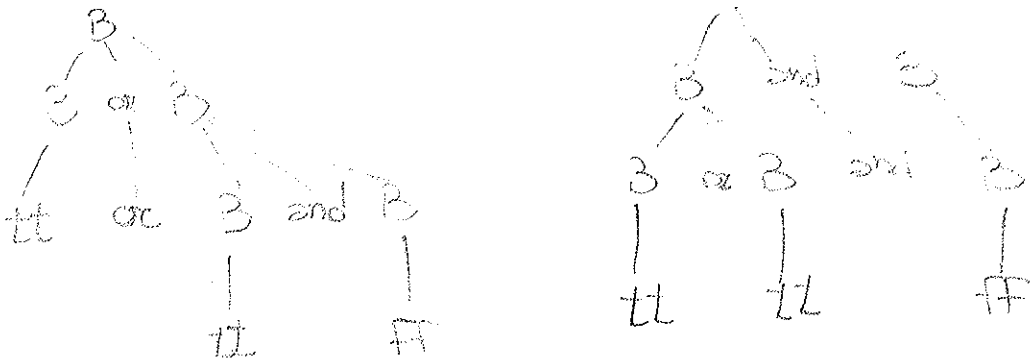
1) Trovare che G è AMBIGUA

2) trovare una G LALR(1)

Precedenze NOT più preced dee
AND " " "
OR

deve essere left-associative

1)



So G is AMBIGUOUS

2)

B → B or B | C
C → C and C | D
D → not G | G
G → (B) | tt | ff

we can't have "not not ff"

D → (B) | tt | ff | not B

↑ AMBIGUOUS

BOTTOM-UP PARSING

HANDLE substring β that matches the right side of some $A \to \beta$ and such that

$S \xrightarrow[rm]{*} \alpha A w \xrightarrow[rm]{} \alpha \beta w \xrightarrow[rm]{*} z$

rm = rightmost

is a rightmost derivation of z from S

when this happens β starting at $|\alpha|+1$ is a handle of $\alpha\beta w$ because $A \to \beta$

example

$S \to aABe$
$A \to Abc \mid b$
$B \to d$

$S \xrightarrow[rm]{} aABe$
$\xrightarrow[rm]{} aAde$
$\xrightarrow[rm]{} aAbcde$
$\xrightarrow[rm]{} abbcde$

$S \to aSb \mid ab$

$S \to aSb \to aaSbb \to aaabbb$

$	aaabbb$	SHIFT	
$a	aabbb$		
$aa	abbb$		
$aaa	bbb$	SHIFT	
$aaab	bb$	REDUCE	$S \to ab$
$aaS	bb$	SHIFT	
$aaSb	b$	REDUCE	$S \to aSb$
$aS	b$	SHIFT	
$aSb	$	REDUCE	$S \to aSb$
$S	$	–	

what's happen with an ambiguous grammar

$E \rightarrow E+E \mid E*E \mid id$

$id_1 + id_2 * id_3$

Stack	Input
$	$id_1 + id_2 * id_3$ $
$ id_1	$+ id_2 * id_3$ $
$ E	$+ id_2 * id_3$ $
$ $E + id_2$	$* id_3$ $
$ E+E	$* id_3$ $
$ E	$* id_3$ $
$ $E * id_3$	$
$ E*E	$
$ E	

le parser non sa se effettuare uno shift (caso 2) o un reduce (caso 1)

or

Stack	Input
$ E+E	$* id_3$ $
$ $E+E * id_3$	$
$ E+E*E	$
$ E+E	$
$ E	

$E \rightarrow E*E$
$\rightarrow E * id_3$
$\rightarrow E+E * id_3$
$\rightarrow E + id_2 * id_3$
$\rightarrow id_1 + id_2 * id_3$

$E \rightarrow E+E$
$\rightarrow E+E*E$
$\rightarrow E+E* id_3$
$\rightarrow E+ id_2 * id_3$
$\rightarrow id_1 + id_2 * id_3$

S → id (Parameter_list)

Parameter → id

Parameter_list → Parameter | Parameter_list ; Parameter

Expr → id | id (Expr_list)

Expr_list → Expr | Expr_list ; Expr

id(id, id)

19/10/2012

SLR

1

Simple

input w$

stack

parsing table

states	Symbols of G, $

M[s,X] =

- shift k — shift one element from the input buffer to the stack and the new state is k
- goto k — the next state is k
- reduce "A→β" — reduce j, if productions of the grammar enumerated and A→β is the j-th productions
- accept — input string accepted
- error — error

LR Parsing

- SLR
- LALR — Look ahead LR
- canonical LR

LR Parsing Alg

(the core for all the LR parsing techniques)

input w, G, parsing table for G

output rightmost derivations of w if $w \in \mathcal{L}(G)$, error otherwise

init starting state (s_0) on the stack with input buffer

let b be the first symbol of w$

while true do

{ let s be top of stack;

if M[s,b] = sn (shift n) { push b onto the stack;
push n onto the stack;
let b be the next input symbol }

else if (M[s,b] = R "A → β") { pop 2*|β| symbols from the stack;
let m be the top of the stack;
let n be such that M[m, A] = gn;
push A,
push n;
output "A → β" }

else if (M[s,b] = accept) { break }

else error()

COLLECTION OF SETS OF ITEM

ITEM a production of the given grammar with a dot ('.') occurring at some position in the right side of the production

given a grammar G, with a starting symbol S,
to define the collection of sets of items for G
we refere to an enriched grammar G'
$S' \to S$ added to P with S' new non terminal
$S' \to .S$ is the first item that we consider

closure(I) closure of a set of item
goto(I,X) goto function depending on a set of items and a symbol of G

$S' \to S$
$S \to aSb \mid ab$

$S' \to .S$
$S \to .aSb$
$S \to .ab$

$A \to \alpha . B\beta$ in I add $B \to .\gamma$
$B \to \gamma$ in G

Function closure
J := I
repeat foreach $A \to \alpha . B\beta$ in J & foreach $B \to \gamma$ in G do
{ if $B \to .\gamma \notin J$ then add it to J }
until saturation % nothing more can be added
return J

goto(I,X) = closure(K)
where K is the set of items of the shape
$A \to \alpha X . \beta$ such that $A \to \alpha . X\beta$ is in I

EXAMPLE
$S' \to S$
$S \to aSb \mid ab$

$I = closure(\{S' \to .S\}) = S' \to .S$
$S \to .aSb$
$S \to .ab$

goto(I,a) = $S \to a.Sb$
$S \to a.b$
$S \to .aSb$
$S \to .ab$

generation of the collection of items

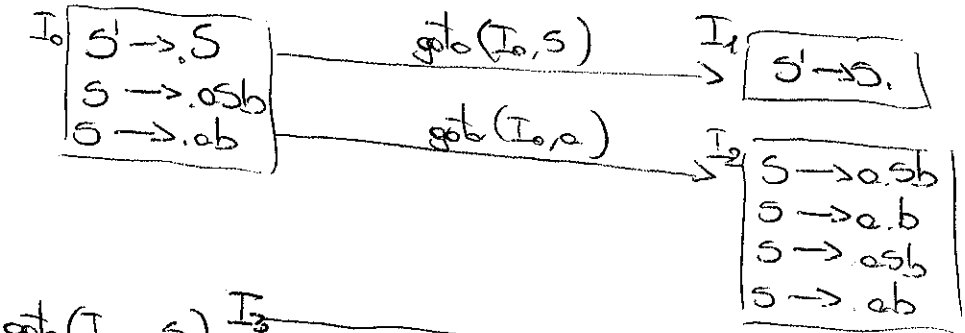
procedure C_Items(G')

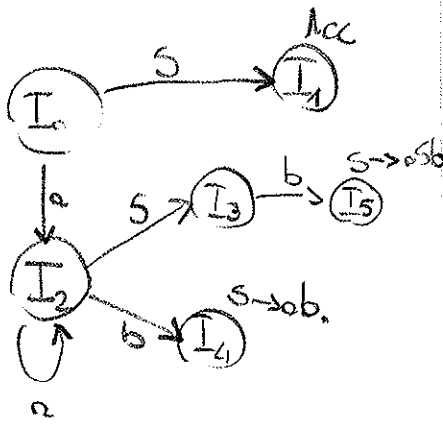
C := closure({S' → .S})

repeat (foreach set I in C) & (foreach symbol X of G such that goto(I,X) ≠ ∅ & goto(I,X) ∉ C yet) do

add goto(I,X) to C

until saturation





goto(I2, S) = I3 | S → aS.b |

goto(I2, a) = I2

goto(I2, b) = I4 | S → ab. |

goto(I3, b) = I5 | S → aSb. |

CONSTRUCTION OF SLR Parsing Table

input enriched grammar G'

output SLR parsing table for G

	FIRST
S	a

	FOLLOW
S	$ b

1. find the collection of sets of items LR(0) for G', say I0, ... , In
2. define state j from Ij
   - if A → α.aβ ∈ Ij and goto(Ij, a) = Ik then M[j, a] = sk (shift k)
   - if A → α. ∈ Ij then ∀x ∈ FOLLOW(A) M[j, x] = Reduce "A → α"
   - if S' → S. ∈ Ij then M[j, $] = acc

   if at this stage there are multiply-defined entries, then STOP, the grammar cannot be parsed using SLR (the grammar is not SLR
3. if goto(Ij, A) = Ik then M[j, A] = gk
4. fill by error all the entries still empty
5. set 0 state of the parser is given by the set of items containing S' → .S

SLR Parsing Table

	a	b	$	S
S_0	S_2			1
S_1			acc	
S_2	S_2	S_4		3
S_3		S_5		
S_4		R S→ab	R S→ab	
S_5		R S→aSb	R S→aSb	

0	aaaabbbb$	shift 2
0 a2	aaabbbb$	shift 2
0 a2 a2	aabbbb$	shift 2
0 a2 a2 a2	abbbb$	shift 2
0 a2 a2 a2 a2	bbbb$	shift 4
0 a2 a2 a2 a2 b4	bbb$	r: S→ab
0 a2 a2 a2 S3	bb$	shift 5
0 a2 a2 a2 S3 b5	bb$	r: S→aSb
0 a2 a2 S3	b$	shift 5
0 a2 a2 S3 b5	b$	r: S→aSb
0 a2 S3	$	shift 5
0 a2 S3 b5	$	S→aSb
0 S1	$	accept

23/10/2012

EXERCISE

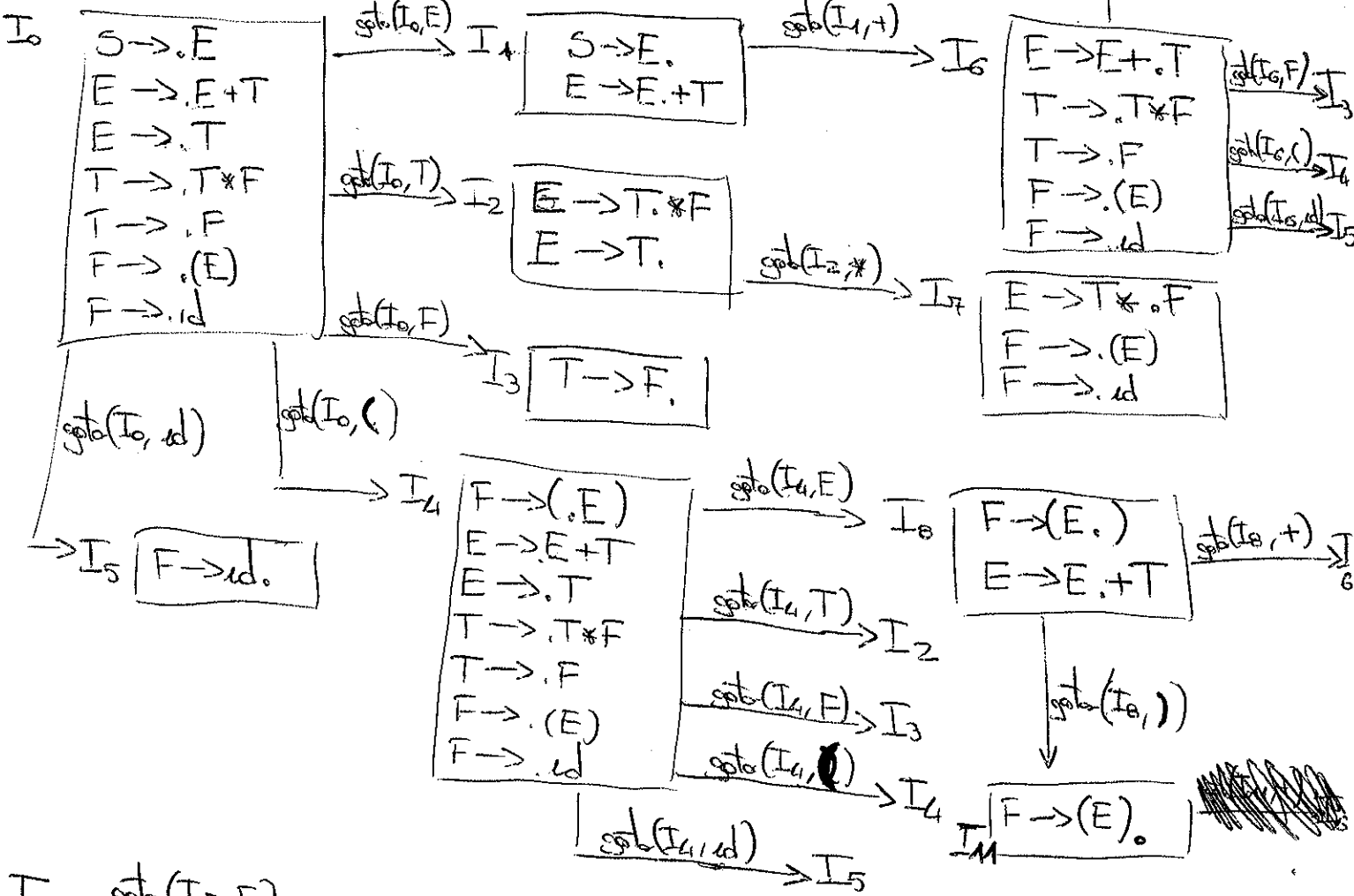
- collection of sets of items
- algorithm SLR parsing table
- algorithm of SLR parsing (parse id+id*id)

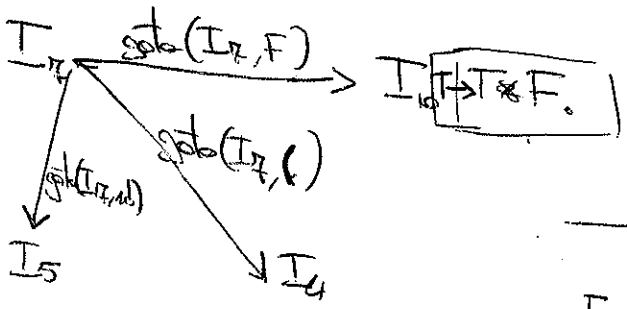
$E \to E+T \mid T$

$T \to T*F \mid F$

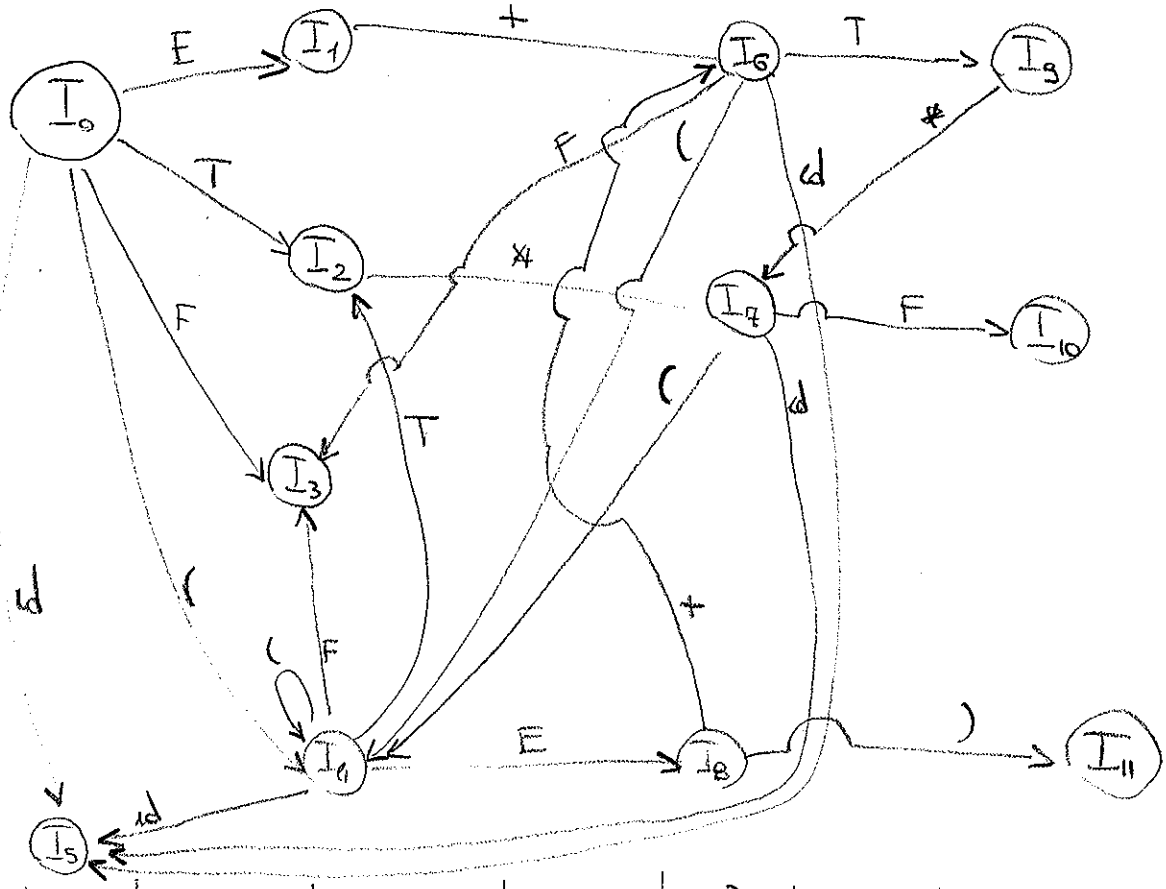
$F \to (E) \mid id$

$S \to E$ ← enrich



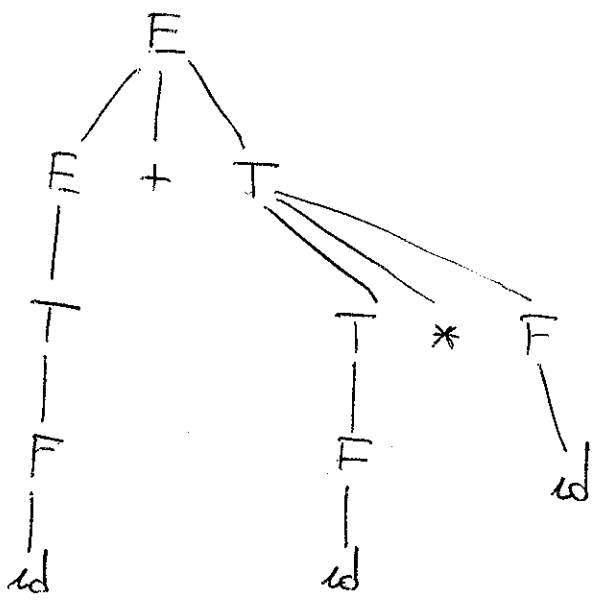


	First	Follow
E	(id	$ +)
T	(id	* $ +)
F	(id	* $ +)



	+	*	id	(	)	$	E	T	F
I_0			S_5	S_4			g_1	g_2	g_3
I_1	S_6					acc			
I_2	r: E→T	S_7			r: E→T	r: E→T			
I_3	r: T→F	r: T→F			r: T→F	r: T→F			
I_4			S_5	S_4			g_8	g_2	g_3
I_5	r: F→id	r: F→id			r: F→id	r: F→id			
I_6			S_5	S_4				g_9	g_3
I_7			S_5	S_4					g_{10}
I_8	S_6				S_{11}				
I_9	r: E→E+T	S_7			r: E→E+T	r: E→E+T			
I_{10}	r: T→T*F	r: T→T*F			r: T→T*F	r: T→T*F			
I_{11}	r: F→(E)	r: F→(E)			r: F→(E)	r: F→(E)			

STACK	INPUT	
0	id + id * id $	
0 id 5	+ id * id $	Reduce F → id
0 F 3	+ id * id $	Reduce T → F
0 T 2	+ id * id $	Reduce E → T
0 E 1	+ id * id $	
0 E 1 + 6	id * id $	
0 E 1 + 6 id 5	* id $	Reduce F → id
0 E 1 + 6 F 3	* id $	Reduce T → F
0 E 1 + 6 T 9	* id $	
0 E 1 + 6 T 9 * 7	id $	
0 E 1 + 6 T 9 * 7 id 5	$	Reduce F → id
0 E 1 + 6 T 9 * 7 F 10	$	Reduce T → T * F
0 E 1 + 6 T 9	$	Reduce E → E + T
0 E 1	$	ACC



24/10/2012

Which is the difference between this two grammars?

G_1: $A \to A\ op\ b \mid b$

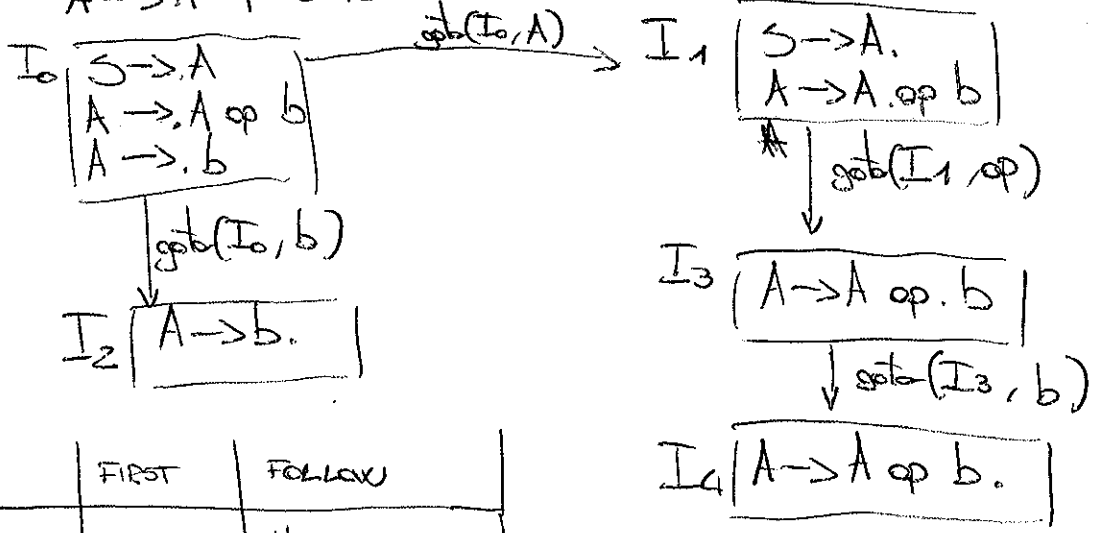
G_2: $A \to b\ op\ A \mid b$

$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$
"special grammar" has associativity

- G_1 is left recursive
- G_2 is left factoring
- neither of them is LL(1)

G_1 $S \to A$

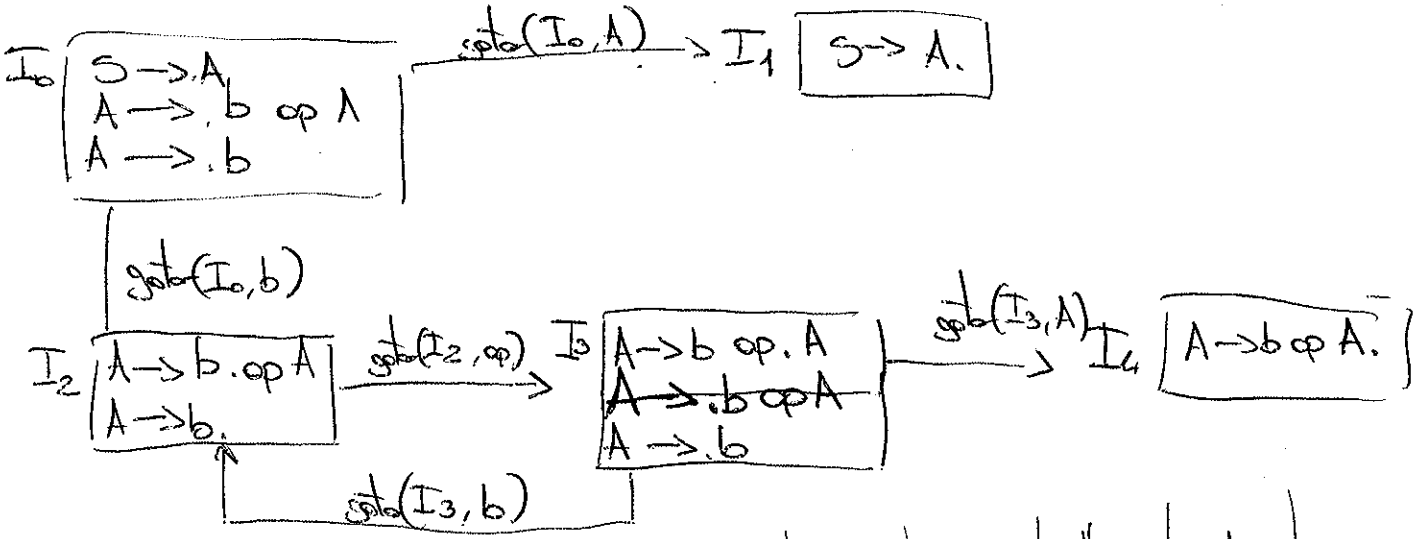
$A \to A\ op\ b \mid b$



	FIRST	FOLLOW
A	b	$ op

#	b	op	$	A
I_0	S_2			g_1
I_1		S_3	acc	
I_2		r: A->b	r: A->b	
I_3	S_4			
I_4		r: A->A op b	r: A op b	

G_2 $S \to A$
$A \to b\ op\ A \mid b$

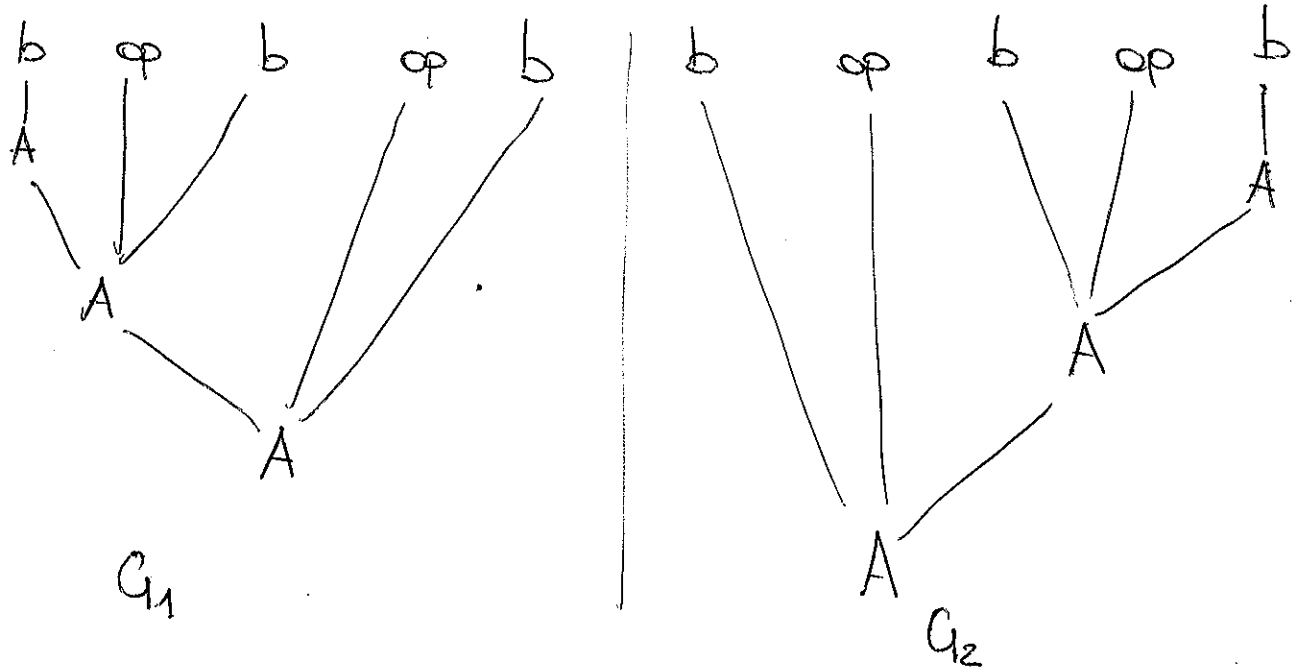


	FIRST	FOLLOW
A	b	$

	op	b	$	A
I_0		S_2		g_1
I_1			acc	
I_2	S_3		r: A → b	
I_3		S_2		g_4
I_4			r: A → b op A	

G_1 has associativity on the left
G_2 " " " " right

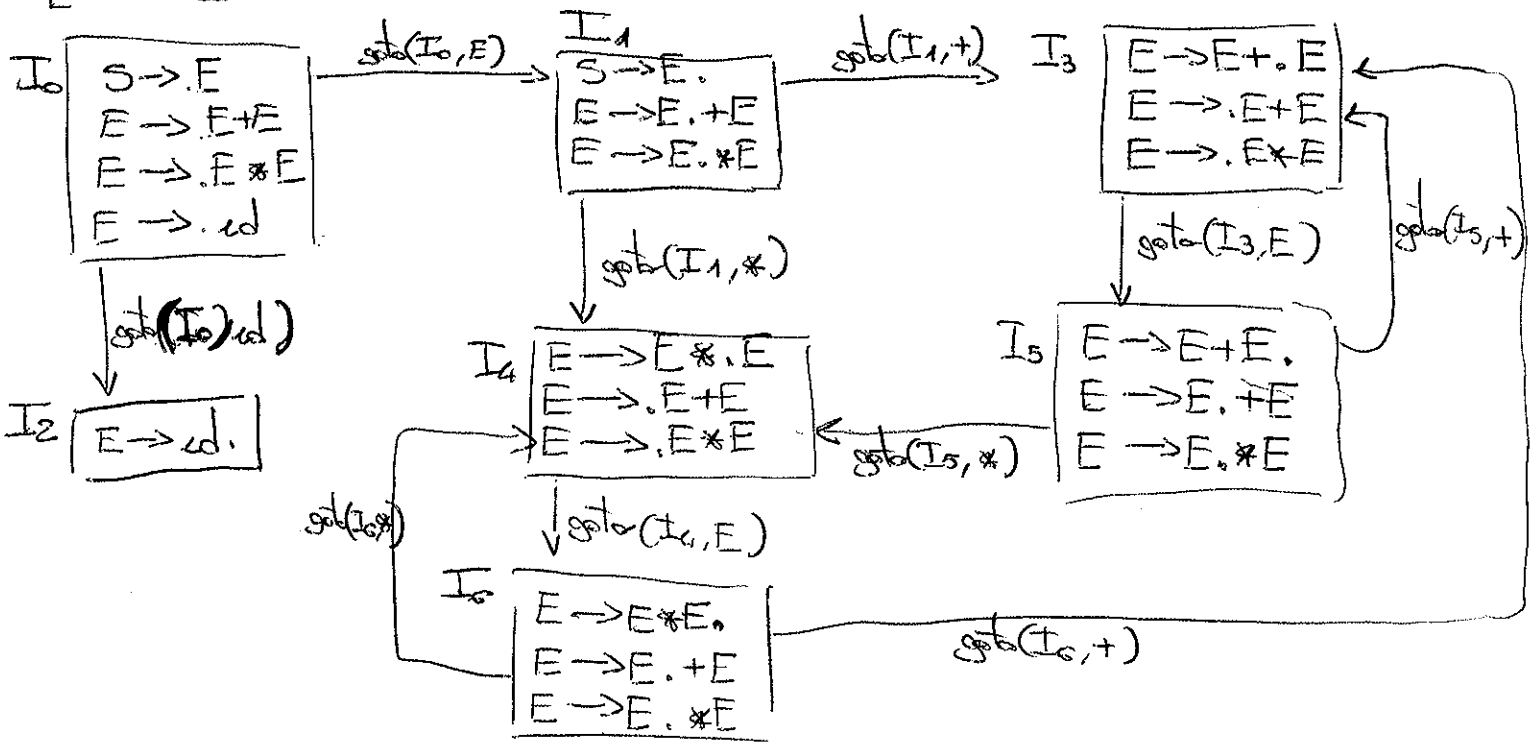
EXAMPLE b op b op b

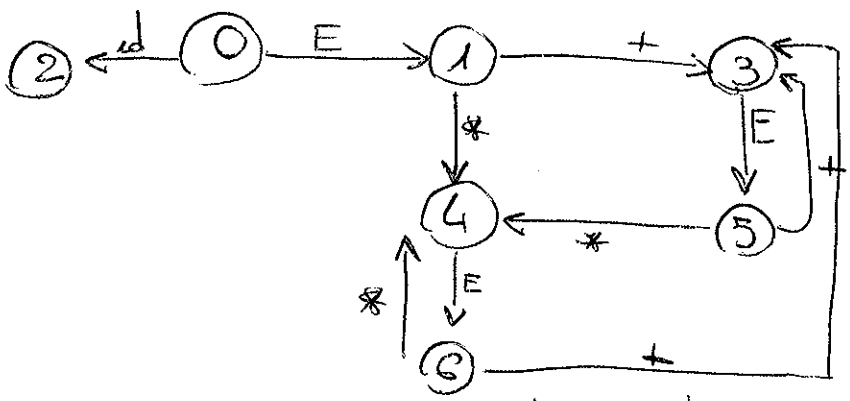


$E \to E+E \mid E*E \mid id$ ← IS AMBIGUOUS

⇓

$S \to E$
$E \to E+E$
$E \to E*E$
$E \to id$





	FIRST	FOLLOW
E	id	$ + *

	+	*	id	$	E
I_0			S_2		g_1
I_1	S_3	S_4		acc	
I_2	r:E→id	r:E→id		r:E→id	
I_3					g_5
I_4					g_6
I_5	r:E→E+E S3	r:E→E+E S4		r:E→E+E	
I_6	r:E→E*E S3	r:E→E*E S4		r:E→E*E	

Here I do reduce
(I read something like E+E and then another +)

Here I shift
(I read E+E and then *, I shift to continue reading)

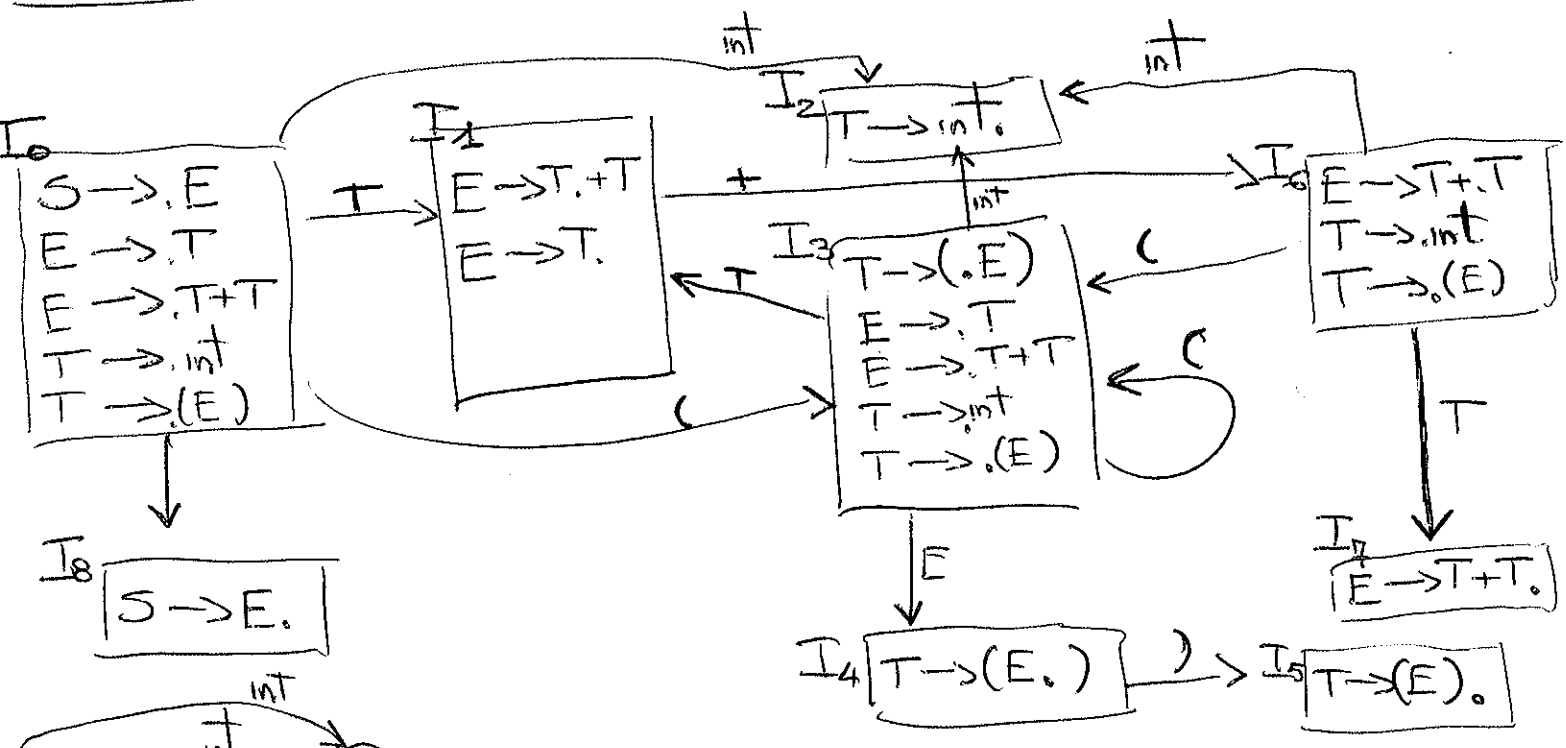
multiple entries!!
but we can choose what to do

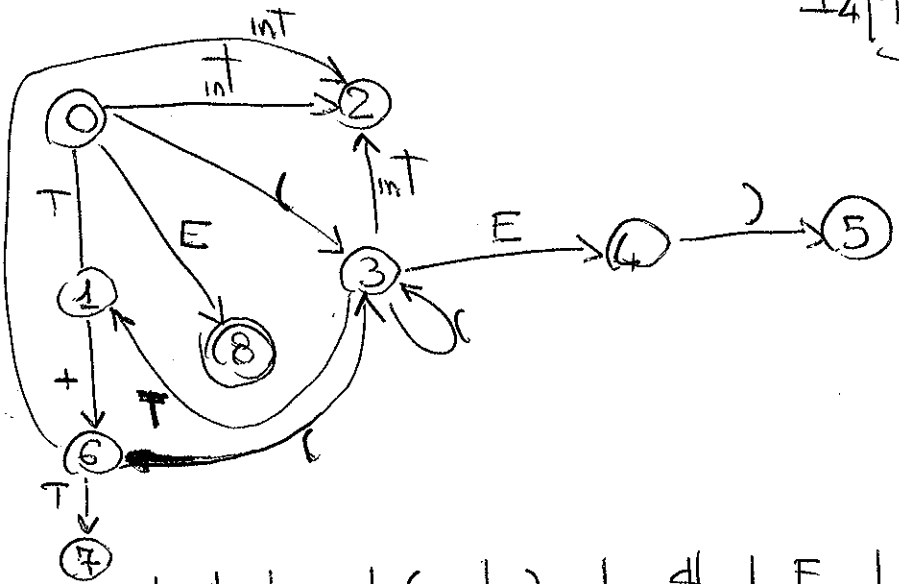
Here I do Reduce
(~~Here~~ I read E+E and then + or * in both cases I associate to the left)

Esercizio

1. $E \to T$
2. $E \to T + T$
3. $T \to int$
4. $T \to (E)$

	First	Follow
E	int (	$)
T	int (	$) +





	int	+	(	)	$	E	T
0	s_2		s_3			g_8	g_1
1		s_6		r: E→T	r: E→T		
2		r: T→int		r: T→int	r: T→int		
3	s_2	s_3				g_4	g_1
4				s_5			
5		r T→(E)		r T→(E)	r T→(E)		
6	s_2		s_3				g_7
7				r: E→T+T	r: E→T+T		
8					acc		

PROPERTIES OF CONTEXT-FREE LANG

9/10/2012

$G=(V,T,S,P)$

$\omega \in \mathcal{L}(G)$

DERIVATION TREE FOR $\omega \in \mathcal{L}(G)$ IS a labelled tree such that
- the root is labelled by S
- a node labelled by $A \in V \setminus T$ has immediate (ordered from left or right) descendants $X_1 \dots X_m$ iff $A \to X_1 \dots X_m \in P$
- the border of the tree is ω

$S \to aSb \mid ab$

$\omega = aaabbb$

PUMPING LEMMA FOR CONTEXT-FREE LANGUAGE

Let $\mathcal{L}$ be a context-free language, then exist $p \in \mathbb{N}^+$ such that
$\forall z \in \mathcal{L}: |z| > p \quad \exists\ u,v,w,x,y$ such that

1. $z = uvwxy$
2. $|vwx| \leq p$
3. $|vx| > 0$
4. $\forall i \in \mathbb{N}.\ uv^iwx^iy \in \mathcal{L}$

Proof By $\mathcal{L}$ is context-free, $\exists$ G context-free such that $\mathcal{L} = \mathcal{L}(G)$

[G in Chomsky normal-form]

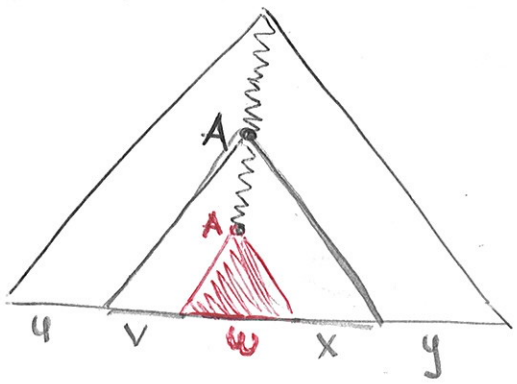
what is p?

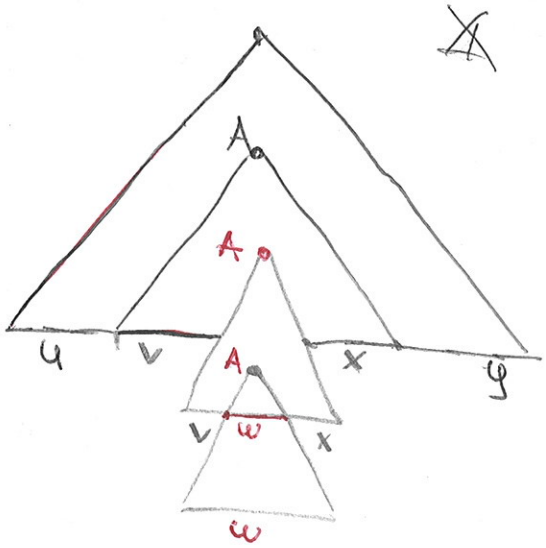
$G=(V,T,S,P)$ $n = |V \setminus T|$ (non-terminal)

p is the length of the longest word in the set of all the words in $\mathcal{L}(G)$ that can be generated by derivation tree whose depth is less or equal n

$S \to aSb \mid ab$ $n=1$ $p=2$

1



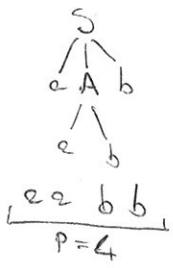


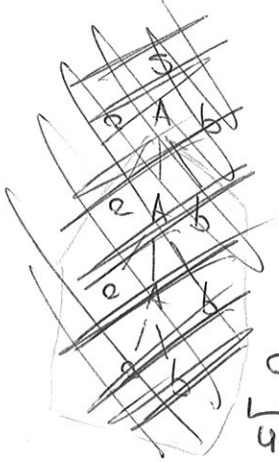
$S \rightarrow aAb$

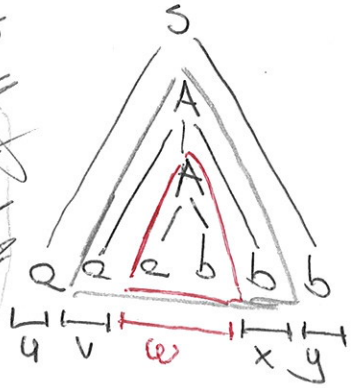
$A \rightarrow aAb \mid ab$

non terminal $n = 2$

$p = 4$







$\{a^m b^m c^m \mid m \geq 0\}$ is context-free

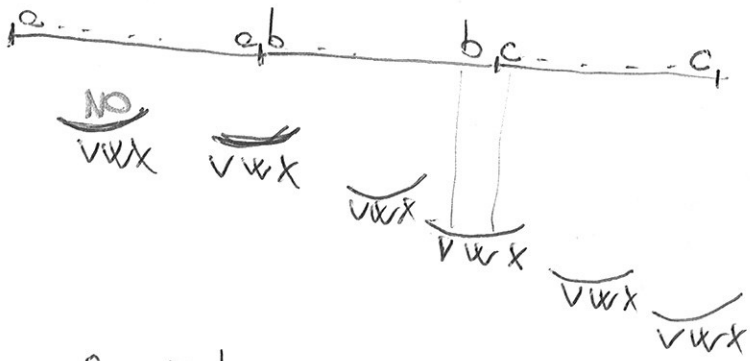
Suppose it is context-free

Choose an arbitrary p (p is in the pumping lemma c.f.)

Take $z = a^p b^p c^p$

$z \in L$, $|z| > p$

c does not occur at all vwx



if $z = uvwxy$ & $|vwx| \leq p$ & $|vx| > 0$

then $|vwx|$ has no occurrences of 'a' or no occurrences of 'c'

then for $i = 2$ uv^2wx^2y is either $a^p b^j c^k$ with possibly $j = k$ or $a^j b^k c^p$, with possibly $j = k$

Closure Properties

PROP CFree-languages are closed under union

[Let L_1, L_2 be Free languages then $L = L_1 \cup L_2$ is C-Free]

L_1, L_2 are context-Free, then $\exists\ G_1, G_2$ C-F-grammars : $L_1 = L(G_1)$ and $L_2 = L(G_2)$

$G_i = (V_i, T_i, S_i, P_i)$ $i = 1,2$ $\quad (V_1' \cup V_2' \cup \{S\} \mid T_1 \cup T_2, S, P_1' \cup P_2' \cup \{S \to S_1' \mid S_2'\})$

~~choose a new symbol S~~

1- refresh the non-terminals so that there is no clash between the two grammars and obtain, say V_1', V_2'

2- choose new symbol S

PROP C Free-language are closed under concatenation

$\{w \mid w = w_1 w_2 \ \&\ w_1 \in L_1 \ \&\ w_2 \in L_2\}$

$V_1' \cup V_2' \cup \{S\}, T_1 \cup T_2, S, P_1' \cup P_2' \cup \{S \to S_1' \mid S_2'\}$

PROP CFree-Langs are NOT close under intersections

choose L_1, L_2 cf

show that $L_1 \cap L_2$ is not context-Free $\qquad \{a^n b^n c^n \mid n \geq 0\}$

$L_1 = \{a^n b^n c^i \mid n, i \geq 0\}$

$L_2 = \{a^i b^n c^n \mid n, i \geq 0\}$

$L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$

are free (show grammars) but $L_1 \cap L_2$ is not

lexical analysis

↓

syntax analysis) PARSING

- TOP-DOWN PARSING (left-most derivation of w from S)
- BOTTOM-UP PARSING (right-most derivation of w from S)

TOP DOWN PARSING

backtrack

$S \to cAd \longrightarrow c\,\cancel{a}\,bd \Rightarrow cAd \to cad$ SUCCESS

$S \to cAd$

$A \to ab \mid a$

$w = cad$

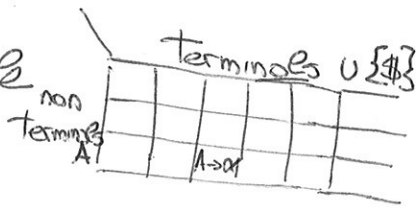
PREDICTIVE TOP-DOWN PARSING → no backtrack needed
→ suitable grammar

PREDICTIVE TOP-DOWN PARSING

input buffer: [w$]

stack

parsing table



algoritm

input: string w$, top-down parsing table for G

output: left-most derivation of w if $w \in L(G)$, error() otherwise

init: w$ in the buffer, $S in the stack

repeat

X := top stack

y := ↑ip — first symbol of w

if X is a terminal or $

then if X = y then { pop X; move ip forward }

else error()

else

if $T[X,y] = X \to Y_1 \dots Y_k$

then { pop X; push $Y_k, \dots Y_1$; output "$X \to Y_1 \dots Y_k$" }

else error()

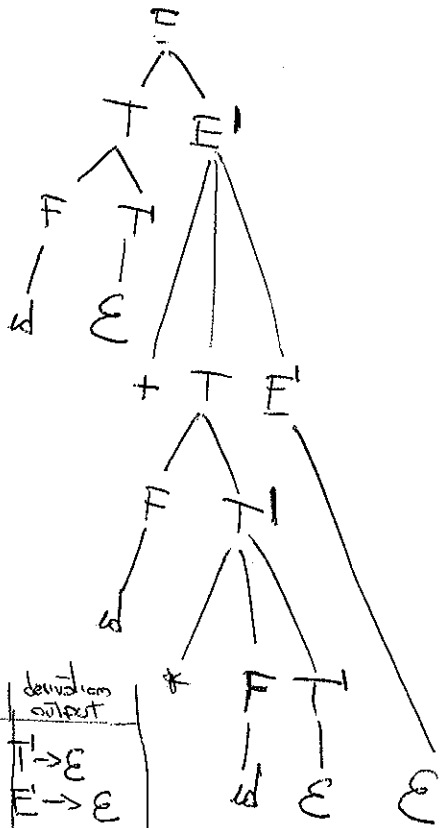
until X = $

if ↑ip = $ then success

G:
$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid id$

Table	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

stack	input	derivation output
$E	id+id*id$	
$E'T	id+id*id$	E→TE'
$E'T'F	id+id*id$	T→FT'
$E'T'id	id+id*id$	F→id
$E'T'	+id*id$	match
$E'	+id*id$	T'→ε
$E'T+	+id*id$	E'→+TE'
$E'T	id*id$	match
$E'T'F	id*id$	T→FT'
$E'T'id	id*id$	F→id
$E'T'	*id$	match
$E'T'F*	*id$	T'→*FT'
$E'T'F	id$	match
$E'T'id	id$	F→id
$E'T'	$	match
$E'	$	T'→ε
$	$	E'→ε
		accepted



	b
A	A→α

the idea is
$[A, b] = A \to \alpha$
if b is the first symbol of either α
or some string derivable from α if $\alpha = C\alpha'$

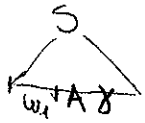
TOP-DOWN PARSING ✳ 12/10/2012

general idea

given $A \to \alpha$

if $\alpha \to^* b\beta$ then $M[A, b] = A \to \alpha$ FIRST

if $\alpha \to^* \varepsilon$ FOLLOW



FIRST(α)

base case $\alpha = X$

1. if X is a terminal FIRST(X) = X
2. if X is a non-terminal
   (i) if $X \to \varepsilon \in P$ then add ε to FIRST(X)
   (ii) if $X \to Y_1 \dots Y_k \in P$ then if $\forall j = 1..k \ \varepsilon \in$ FIRST(Y_j) then add ε to FIRST(X)
   if $\varepsilon \in$ FIRST(Y_1) $\cap \dots \cap$ FIRST(Y_{j-1}) & $b \in$ FIRST(Y_j) for some $2 \le j \le k$ then add b to FIRST(X)

inductive case $\alpha = X_1 \dots X_m$, $m > 1$

- assumed $X_0 = \varepsilon$, add to FIRST(α) all the symbols b such that $(\exists i : 1 \le i \le m \ \ b \in$ FIRST(X_i)) & ($\forall j : 0 \le j \le i-1 . \ \varepsilon \in$ FIRST(X_j))
- add ε to FIRST(α) if $\forall j \ 1 \le j \le m, \ \varepsilon \in$ FIRST(X_j)

$E \to T E'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid id$

	first
E	id (
E'	+ ε
T	id (
T'	* ε
F	.(id

FOLLOW(A) * A must be a non-terminal

{terminals that can occour immediately at the right of A}

1. add $ to FOLLOW(S)
2. foreach $B \to \alpha A \beta \in P$ add FIRST(β) $\setminus \{\varepsilon\}$ to FOLLOW(A)
3. foreach $B \to \alpha A \in P$ add FOLLOW(B) to FOLLOW(A) [OBS if B=A then this is void]
4. foreach $B \to \alpha A \beta \in P$, if $\varepsilon \in$ FIRST(β) then add FOLLOW(B) to FOLLOW A [OBS if B=A then SKIP]

	FOLLOW
E	$)
E'	$)
T	+ $)
T'	+ $)
F	* + $)

Construction of PREDICTIVE PARSING TABLE

Alg

input G

output predictive parsing table

foreach $A \to \alpha$ do

{ $\forall b \in$ first(α), insert $A \to \alpha$ in M[A,b];

if $\varepsilon \in$ first(α) then $\forall x \in$ FOLLOW(A), insert $A \to \alpha$ in M[A,x]

}

insert error() in all the empty entries

If the predictive parsing table for grammar G has no multiply defined entry then G is said a LL(1) grammar

- we read the input string from the left
- we derive a leftmost derivation
- the length of the lookhead

Define a LL(1) grammar G such that ✗

$L(G) = \{a^n b c^n \mid n > 0\}$ and apply the parsing procedure to $w = aabcc$

- provide G
- show G is LL(1) i.e. it has a predictive parsing table without multiply defined entries

$S \rightarrow aBc$
$B \rightarrow aBc \mid b$

	FIRST	FOLLOW
S	a	$
B	a, b	c

	a	b	c	$
S	S→aBc			
B	B→aBc	B→b		

G is LL(1) because there are not multiply entry on the predictive parsing table

STACK	INPUT	DERIVATION OUTPUT
$S	aabcc$	
$cBa	aabcc$	with S→aBc
$cB	abcc$	B→aBc
$ccBa	abcc$	with B→b
$cc	bcc$	match
c	cc$	match
$	c$	match
$	$	accept

Provide Predictive Parsing table for ✳

G_1: $E \to E+E \mid E*E \mid id$

G_2: $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

G_1:

	First
E	id

	+	*	id	$
E			E→E+E E→E*E E→id	

G_1 is not LL(1)

G_2:

	First
E	(id
T	(id
F	(id

	+	*	(	)	id	$
E			E→E+T E→T		E→E+T E→T	
T			T→T*F T→F		T→T*F T→F	
F			F→(E)		F→id	

G_2 is not LL(1)

http://dis.uniroma1.it/marchetti/teaching/PLC-2013
use unitn.it address

16/10/2012

GRAMMAR PROBLEMS FOR TOP-DOWN PARSING
- Ambiguity
- Left recursion
- Left factoring

Consider Ambiguous
$E \to E+E \mid E*E \mid (E) \mid id$
$S \to$ if b then S | if b then S else S | c

Left recursion $A \to A\alpha$

Ambiguity is a problem for TOP-DOWN PARSING

Left Factorization
$A \to \alpha\beta_1 \mid \alpha\beta_2$

es $S \to aSb \mid ab$

	a
S	S→aSb
	S→ab

AMBIGUITY (there are no algorithm to individuate ambiguous grammars I must try)
- Parentesis

Parentesis ( ) can disambiguate every grammar

- constraint: each else has to refere to the closest unmatched then at its left

$S \to M \mid U$
$M \to$ if b then M else M | c
$U \to$ if b then S | if b then M else U

$E \to E + E \mid E*E \mid id$

1 + 2 * 3
id id id

LEFT RECURSION

* sul libro a pag 212
sul pdf a pag 235

$A \to^{+} A\alpha$

$A \to A\alpha$ è IR immediate left recursion

$A \to B\beta$, $B \to A\alpha$ } $A \to B\beta \to A\alpha\beta$ not immediate left recursion

immediate left recursion

$A \to A\alpha_1 | \dots | A\alpha_m | \beta_1 | \dots | \beta_n$ with $\alpha_i \neq \varepsilon$, prefix $\beta_i \neq A$ $\forall i$
if $\alpha_i = \varepsilon$ then $A \to A\alpha_i$ is $A \to A$

$A \to A\alpha | \beta$

replace $A \to A\alpha | \beta$

by $\begin{cases} A \to \beta A' \\ A' \to \alpha A' | \varepsilon \end{cases}$ A' is a new symbol

replace $A \to A\alpha_1 | \dots | A\alpha_m | \beta_1 | \dots | \beta_n$

by $\begin{cases} A \to \beta_1 A' | \dots | \beta_n A' \\ A' \to \alpha_1 A' | \dots | \alpha_m A' | \varepsilon \end{cases}$

Algorithm for elimination of general left-recursion
Does if G has cycles ($A \to^{+} A$) or if G has ε-productions
the alg does not guarantee elimination]

1. for an ordering of non-terminals of G, say $A_1, \dots, A_n$
2. for $i = 1$ to n do { for $j = 1$ to $i-1$ do
   { let $A_j \to \delta_1 | \dots | \delta_k$ be all the productions for A_j
   replace $A_i \to A_j\gamma$ by $A_i \to \delta_1\gamma | \dots | \delta_k\gamma$ }
   eliminate immediate left recursion for A_i
   }

example of application

$A \to Ba \mid b$

$B \to Bc \mid Ad \mid b$

$A = A_1$

$B = A_2$

$A_1 \to A_2 a \mid b$

$A_2 \to A_2 c \mid \underbrace{A_1 d}_{\beta_1} \mid \underbrace{b}_{\beta_1}$

$i=1$ $j=1$ NO

$i=1$ $j=2$ $A_1 \to A_2 c a \mid A_1 d a \mid b a$

$i = 1$ ✓

$i=2$ $j=1$

$A \to Ba \mid b$
$A_1 \to A_2 a \mid b$

replace $B \to \underbrace{A}_{A_2} \underbrace{T}_{\alpha}$ by $B \to \underbrace{Ba}_{A_2 a} d \mid \underbrace{bd}_{b}$

$B \to \underbrace{Bc}_{A_2 \alpha_1} \mid \underbrace{Bad}_{A_2 \alpha_2} \mid \underbrace{bd}_{\beta_1} \mid \underbrace{b}_{\beta_2}$

Final grammar

$A \to Ba \mid b$

$B \to bdB' \mid bB'$

$B' \to aB' \mid adB' \mid \varepsilon$

is it LL(1)?

No because it's left factorization

	b
B	B→bdB' B→bB'

Exercise

eliminate left recursion from

$E \to E+T \mid T$ $\quad E = A_1$ $\quad A_1 \to A_1 + A_2 \mid A_2$

$T \to T * F \mid F$ $\quad T = A_2$ $\quad A_2 \to A_2 * A_3 \mid A_3$

$F \to (E) \mid id$ $\quad F = A_3$ $\quad A_3 \to (A_1) \mid id$

$i=1$ $A_1 \to A_2 A_1'$

$j=1$ $A_1' \to +A_2 A_1' \mid \varepsilon$

$j=2$ $A_1 \to$

LEFT FACTORING

$A \to \alpha\beta_1 \mid \alpha\beta_2$

replace $A \to \alpha\beta_1 \mid \alpha\beta_2$

by $A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2$

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_m \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k$

α is the longest common prefix to all the productions

by $A \to \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_k$

$A' \to \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_m$

Left factoring the following grammar

G: $S \to$ if b then S | if b then S else S | c

G: $S \to$ if b then S S' | c

$S' \to \varepsilon$ | else S

G' is ambiguous, just as G is (Left factoring is not a cure against ambiguity)

	first	follow
S	if c	$ else
S'	else ε	$ else

$ else

	if	b	then	c	else	$
S	S→if b then S S'			S→c		
S'					S'→else S S'→ε	S'→ε

G' is not LL(1)

17/10/2012

PROPERTIES OF LL(1) grammars

1) No ambiguous grammar is LL(1)
2) No left recursive grammar is LL(1)
3) No grammar that can undergo left-factoring is LL(1)
4) G is LL(1) iff
   if G has productions $A \to \alpha | \beta$ then
   - $First(\alpha) \cap First(\beta) = \emptyset$
   - $\varepsilon \in First(\alpha) \to First(\beta) \cap Follow(A) = \emptyset$ and (vice versa) $\varepsilon \in First(\beta) \to First(\alpha) \cap Follow(A) = \emptyset$

LL(k) $k > 1$

example LL(2)

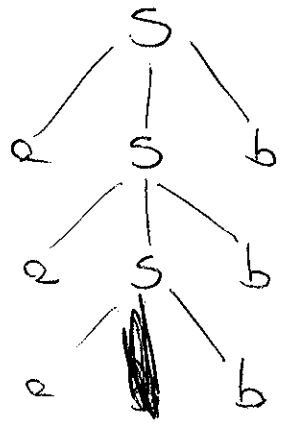
$S \to aSb \mid ab$

string: aaabbb

	FIRST
S	a

	aa	ab	ba	bb	$
S	S→aSb	S→ab			

S	aaabbb$	S→aSb
bS	aaabbb$	match
bSb	aabbb$	S→aSb
$bbSb$	abbb$	match
$bbaSbb$	bbb$	S→ab
$bbabb$	bb$	match
bbb	b$	match
bb	$	match
b		accepted
$		



TOP-DOWN
l'albero si costruisce dalla radice verso le foglie

SYNTAX-DIRECTED DEFINITIONS (SDDs)

16/11/2012

context free grammar &
attribute for the symbols of the grammar &

rules	semantic actions
$L \to E$ enter (symbol finale)	$\{L.val = E.val\}$
$E \to E_1 + T$	$\{E.val = E_1.val + T.val\}$
$E \to T$	$\{E.val = T.val\}$
$T \to T_1 * F$	$\{T.val = T_1.val * F.val\}$
$T \to F$	$\{T.val = F.val\}$
$F \to (E)$	$\{F.val = E.val\}$
$F \to digit$	$\{F.val = digit.lexval\}$

symbol attributes:
- L.val
- E.val
- T.val
- F.val
- digit.lexval

$A \to X_1 \dots X_j Y X_{j+1} \dots X_k$

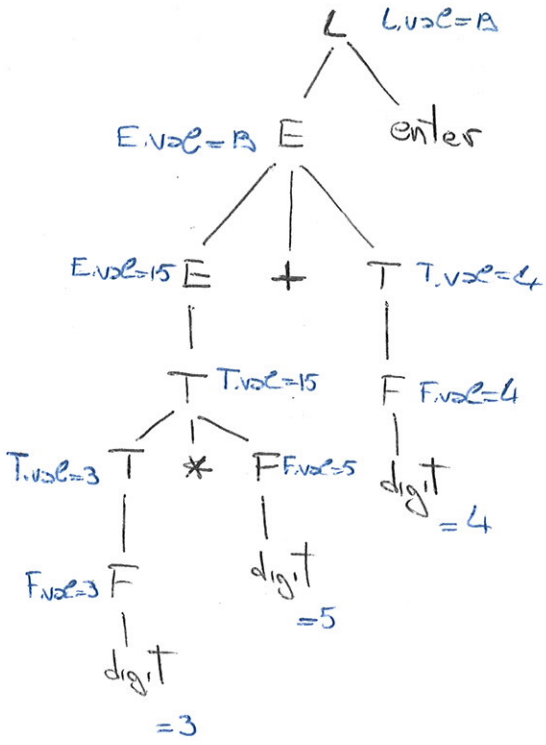
SYNTHESIZED ATTRIBUTE : its an attribute $A.a$ which is defined as a function of attributes of $A, X_1, X_2 \dots, Y, \dots, X_k$

$B \to X_1 \dots X_j, A, X_{j+1}, \dots X_k$

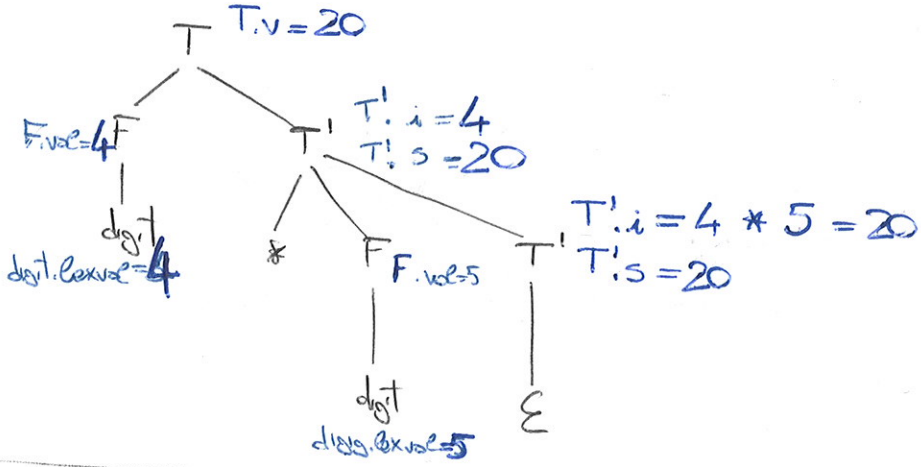
INHERITED ATTRIBUTE : its an attribute $A.a$ defined as a function of $B, X_1, \dots X_k$

3 * 5 + 4 enter

L → E enter
E → E + T
E → T
T → T * F
T → F
F → (E)
F → digit



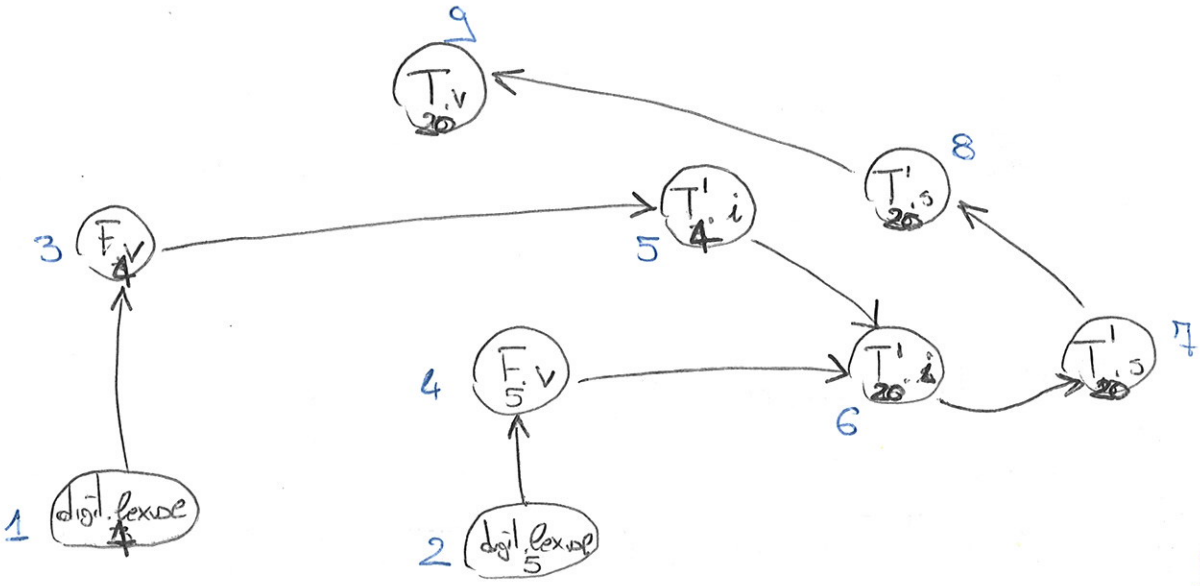
T → F T'
T' → * F T'
T' → ε
F → digit



Productions	Semantic rules
T → F T'	T'.i = F.v T.v = T'.s
T' → * F T'1	T'1.i = T'.i * F.v T'.s = T'1.s
T' → ε	T'.s = T'.i
F → digit	F.v = digit.lexval

DEPENDENCY GRAPH FOR THE EVALUTATION OF ATTRIBUTES

- for each node of the parse tree and for each attribute of the symbol associated to that node, set a node of the DG
- for each attribute A.a and for each attribute X.x used to define A.a, set an edge from X.x to A.a
  (the edge from X.x to A.a means X.x is needed to compute A.a)



in blue topological sort

TOPOLOGICAL SORT OF A GRAPH

number the nodes $N_1 \dots N_K$ of the graph in such a way that if there is an edge $N_i \to N_j$ then $i < j$

L attributed definitions

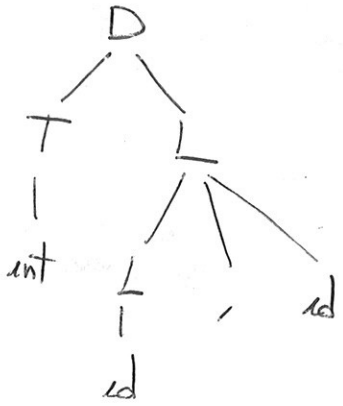
definition whose attributes are
- either synthesized or
- inherited and such that for each productions of the shape $A \to X_1 \dots X_K$ and for each attribute $X_j.i$ the rule for $X_j.i$ only uses
  - inherited att of A
  - inherited or synthesized attribute of $X_1 \dots X_{j-1}$ located to the left of X_j

S-attributed

An SDD is S-attributed if every attribute is synthesized

20/11/2012

$D \rightarrow T\ L$
$T \rightarrow int$
$T \rightarrow float$
$L \rightarrow L, id$
$L \rightarrow id$

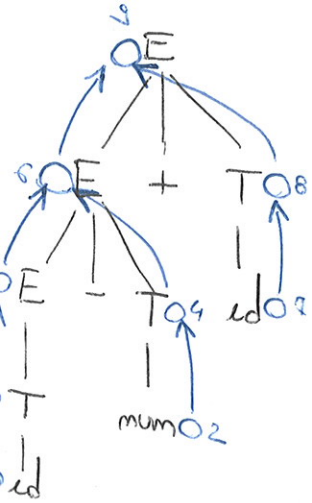


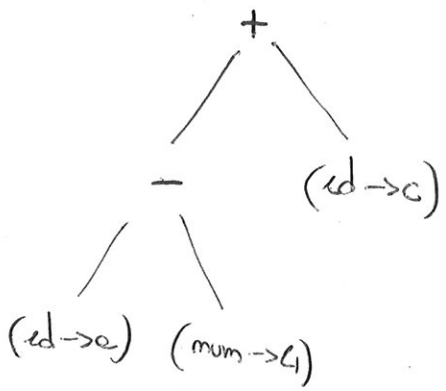
$D \rightarrow TL$	$\{L.i = T.s\}$
$T \rightarrow int$	$\{T.s = integer\}$
$T \rightarrow float$	$\{T.s = float\}$
$L \rightarrow L_1, id$	$\{L_1.i = L.i\ ,\ addType(id.entry, L.i)\}$
$L \rightarrow id$	$\{addType(id.entry, L.i$

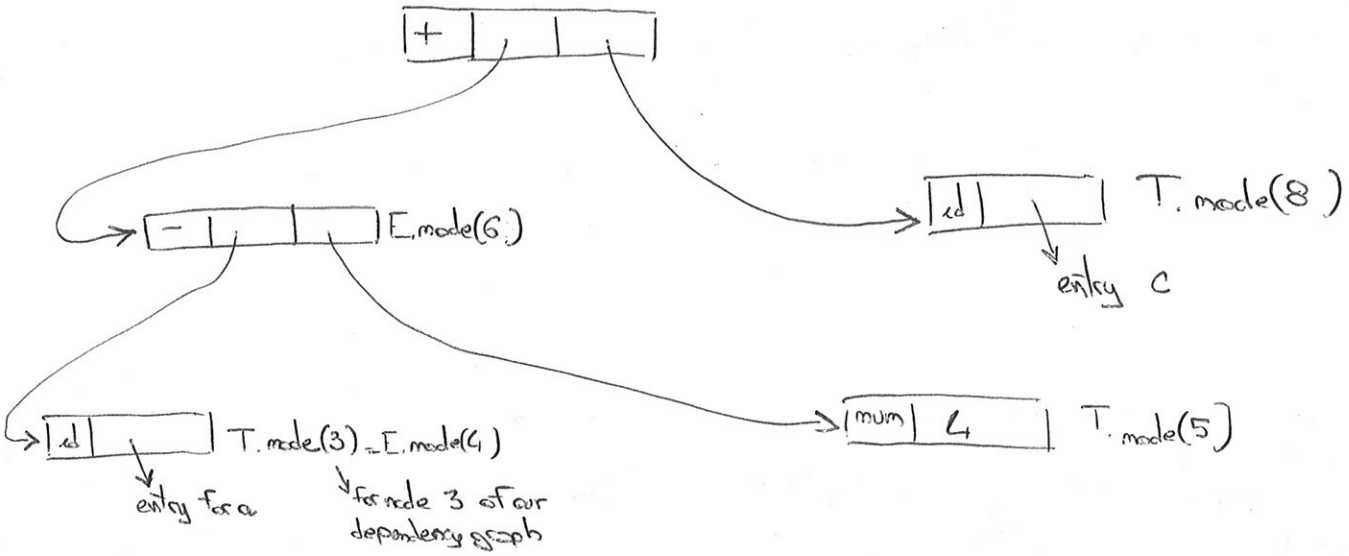
SDT (Syntax Directed Translation)

a - 4 + c | id - num + id (usata per le type checking e la generazione delle intermediate code)

$E \rightarrow E_1 + T$	$E.node = newNode("+", E_1.node, T.node)$
$E \rightarrow E_1 - T$	$E.node = newNode("-", E_1.node, T.node)$
$E \rightarrow T$	$E.node = T.node$
$T \rightarrow (E)$	$T.node = E.node$
$T \rightarrow id$	$T.node = newLeaf(id, id.entry)$
$\rightarrow num$	$T.node = newLeaf(num, num.val)$







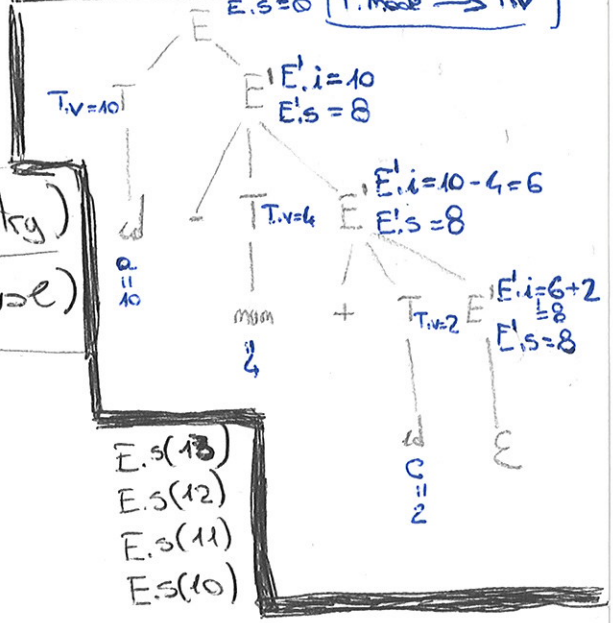
21/11/2012

$E \to TE'$	$E.s = E'.s \quad E'.i = T.node$
$E' \to +TE'_1$	$E'_1.i = newNode('+', E'.i, T.node)$, $E'.s = E'_1.s$
$E' \to -TE'_1$	$E'_1.i = newNode('-', E'.i, T.node)$, $E'.s = E'_1.s$
$E' \to \varepsilon$	$E'.s = E'.i$
$T \to (E)$	$T.node = E.s$
$T \to id$	$T.node = newLeaf(id, id.entry)$
$T \to num$	$T.node = newLeaf(num, num.val)$

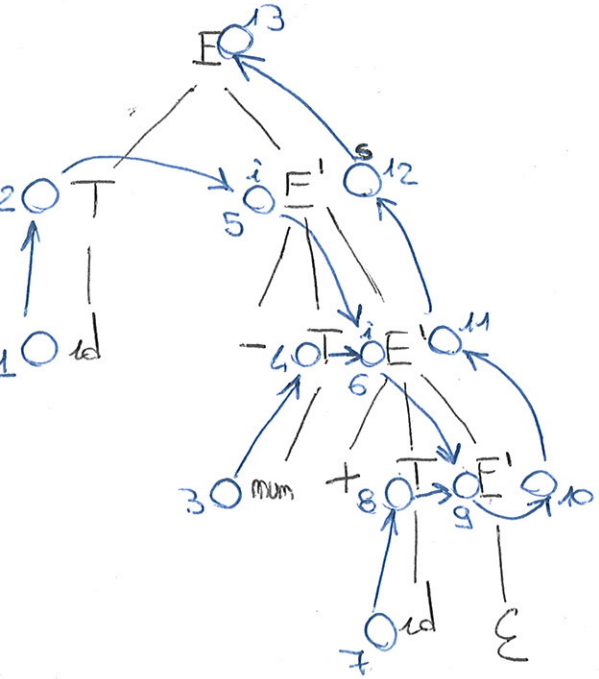
Input: a − 4 + c

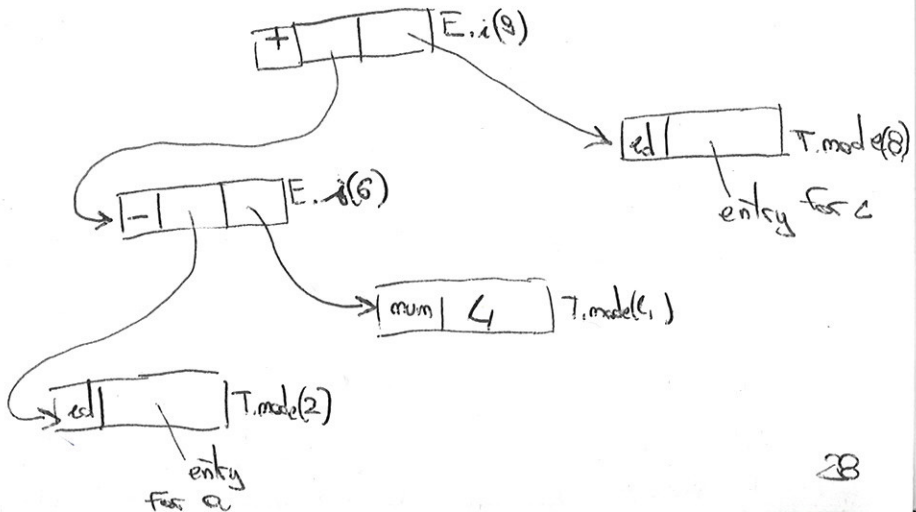
In quest'albero ~~E.node~~ T.node → T.v

E.s = 8



E.s(13)
E.s(12)
E.s(11)
E.s(10)





input : string of digits
output : represented decimal number corresponding to the string

S → Digits
Digits → Digits d | d

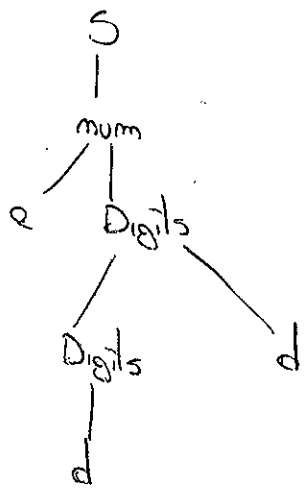
1336

S → Digits print(Digits.v)
Digits → Digits$_1$ d Digits.v = Digits$_1$.v * 10 + d.lexval
Digits → d Digits.v = d.lexval

(σDigits : indica una cifra da zero a 7 (in base 8))

23/11/2012

S → num
num → e Digits
num → Digits
Digits → Digits d
Digits → d

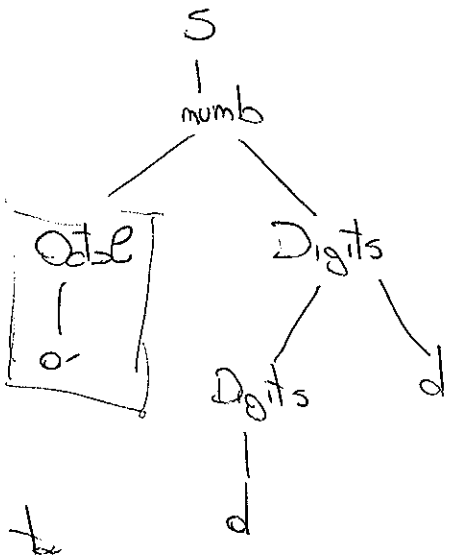


1) Come productions to have distinct semantic actions for distinct productions

S → num
num → e ODigits
num → DDigits
DDigits → DDigits d
ODigits → ODigits d
DDigits → d
ODigits → d

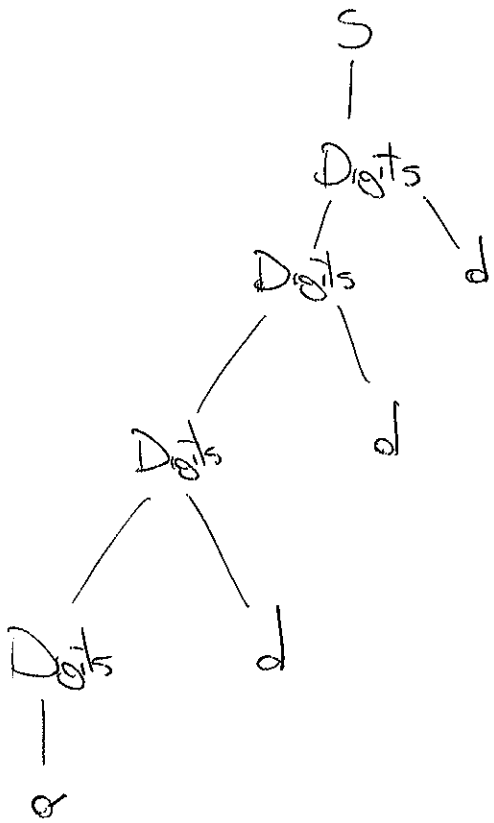
2) Forcing reduction step by introducing extra productions
(either unit productions A → X or ε-productions A → ε)
+ global variable to be used in semantic actions.

S → num	print (num.val)
num → OctalDigits	num.val = Digits.val
num → DecimalDigits	num.val = Digits.val
Octal → o	base 8
Decimal → ε	base 10
Digits → Digits$_1$ d	Digits.val = Digits$_1$.val * base + d.lexval
Digits → d	Digits.val = d.lexval



3) aggressive modifications to the grammar to the needed infor in the right places.
Careful G' has still to be LALR
hint: think of the derivation three first

S → Digits	print (Digits.val)
Digits → Digits$_1$ d	{Digits.val = Digits$_1$.val * Digits.base + d.lexval ; Digits.base = Digits$_1$.base}
Digits → Base	{Digits.val = 0 ; Digits.base = Base.val}
Base → o	Base.val = 8
Base → ε	Base.val = 10



Exercise

Given $V \to num \mid (E)$

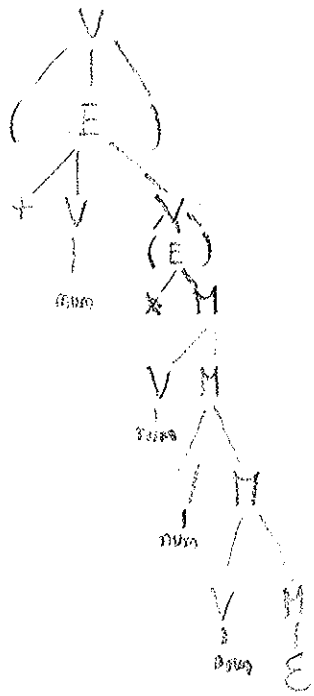
$E \to +VV \mid *M$

$M \to VM \mid \varepsilon$

provide a SDD that determines, as attribute of V, the numerical value of the expression

E.g. for the expression (+1 (* 2 3 4)) the attribute should be 25

$S \to V$	print (V.val)
$V \to num$	V = num.lexval
$V \to (E)$	V.val = E.val
$E \to +V_1 V_2$	E.val = V_1.val + V_2.val
$E \to *M$	E.val = M.val
$M \to VM_1$	M.val = V.val * M_1.val
$M \to \varepsilon$	M.val = 0



Given the grammar

$T \to BC$

$B \to int \mid float$

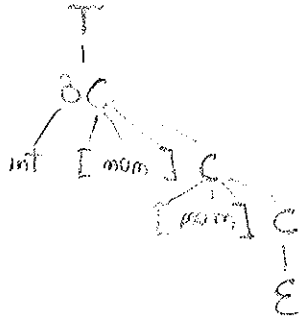
$C \to [num]C \mid \varepsilon$

provide an SDD that determines the type expression of terms written in the language generated by $TE \to integer \mid float \mid array(n', TE)$ where n stands for any positive integer

For instance, as in C, the type int [2][3] denotes an "array of 2 array of 3 integers" and in the fixed languages for type expressions, the type of int [2] [3] is

(array(2, array(3, integer)))

$T \to BC$	T.str = C.start ‖ B.str ‖ C.end
$B \to int$	B.str = 'integer'
$B \to float$	B.str = 'float'
$C \to [num]C_1$	C.start = 'array(' ‖ str(num.lexval) ‖ ',' ‖ C_1.start ; C.end = C_1.end ‖ ')'
$C \to \varepsilon$	C.start = '' , C.end = ''



int [num] [num]

'' = empty string

27/11/2012

Parsing TOP-DOWN con attributi ereditati è molto complicato

SDT (Syntax Driven Translation)

Tecnica per semplificare il parsing di grammatiche con attributi ereditati.

- L-attribute definitions
- Bottom up parsing LL grammar

Top Down Parsing with inherited attribute

SDD SYNTAX DRIVEN TRASLATION

$A \to \alpha \{ \dots \} \beta$ (action)

Esempio SDD (type)

$T \to BC$ $\quad T.t = C.t$

$C.b = B.t$

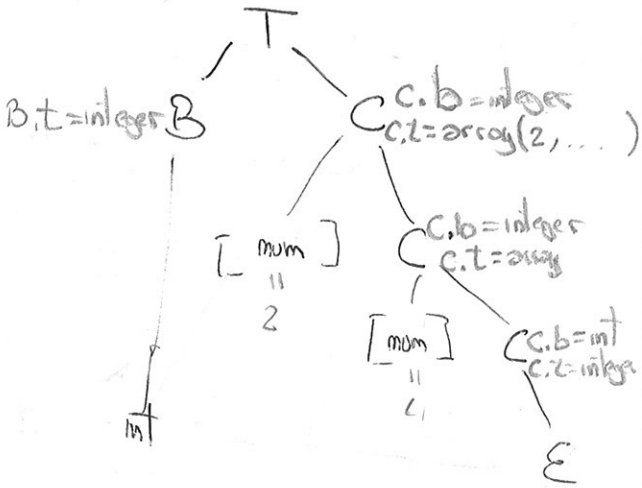
$B \to int$ $\quad B.t = integer$

$B \to float$ $\quad B.t = float$

$C \to [num] C_1$ $\quad C.t = array(num.val, C_1.t)$

$C_1.b = C.b$

$C_1 \to \varepsilon$ $\quad C.t = C.b$



in SDT

$T \to B \{C.b = B.t\} C$ $\quad T.t = C.t$

$B.t = integer$

$B \to int$

$B.t = float$

$B \to float$

$C \to [num] \{C_1.b = C.b\} C_1$ $\quad C.t = array(num.val, C_1.t)$

$C_1 \to \varepsilon$ $\quad C.t = C.b$

$T \to B\ M_1\ C$ $\quad T.t = C.t$

$M_1 \to \varepsilon$ $\quad$ "C.b = B.t" $\quad M_1.b = B.t$; $M_1.s = M_1.b$

$B \to int$ $\quad B.t = integer$

$B \to float$ $\quad B.t = float$

$C \to [num]\ M_2\ C_1$ $\quad C.t = array(num.val, C_1.t)$

$M_2 \to \varepsilon$ $\quad$ " $C_1.b = C.b$ " $\quad M_2.b = M_1.b$; $M_2.s = M_2.b$

$C \to \varepsilon$ $\quad C.t = \cancel{C.b}\ M_1.b$

Parsing Bottom-Up

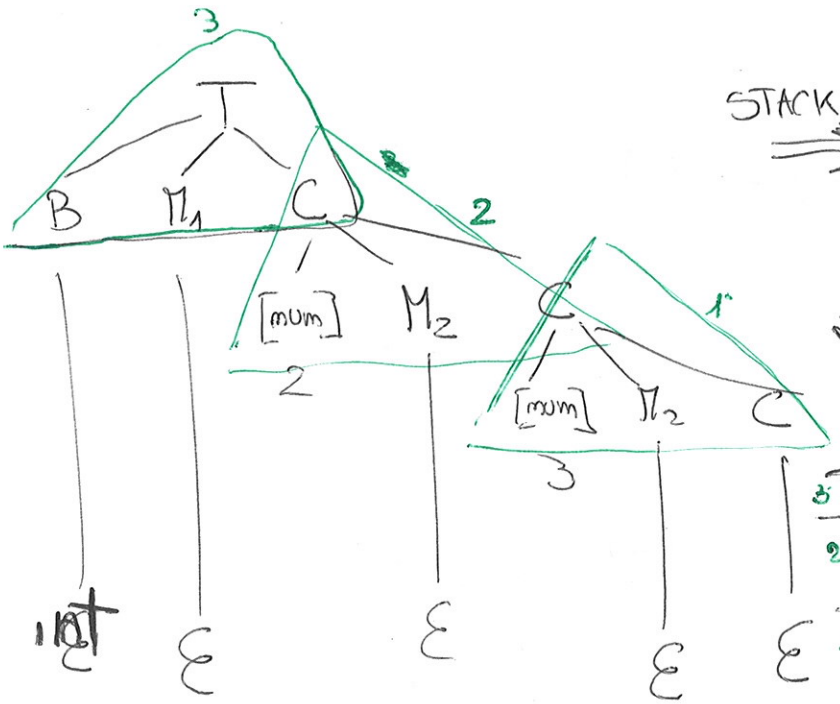
& continua sotto ↓

* ↓

$A \to \{B.i = f(x)\}\ B\ C$

$A \to M\ B\ C$

$M \to \varepsilon$ $\quad \{ M.i = X\ ,\ M.s = f(M.i) \}$



[num] $\quad num.val = 3$

$M_2.s = M_2.b = integer$

M_2 $\quad M_2.b = M_1.b = integer$

[num] $\quad num.val = 2$

$M_1.s = M_1.b = integer$

STACK ⇒ M_1 $\quad M_1.b = B.t = integer$

B $\quad B.t = integer$

↑ int

3° T $\quad T.t = array(2, array(3, integer))$

2° C $\quad C.t = array(2, array(3, integer))$

1° C $\quad C.t = array(3, integer)$

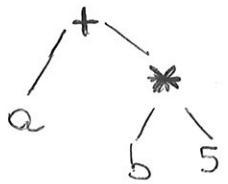
C $\quad C.t = integer$

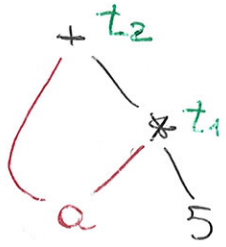
$M_2.s = integer$

↑ M_2 $\quad M_2.b = integer$

INTERMEDIATE CODE GENERATION

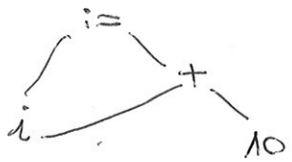
$i := i + 5$ $\quad a + b * 5$



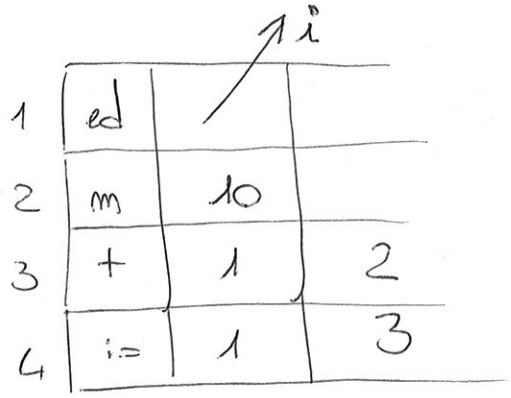


$t_1 = a * 5$
$t_2 = a + t_1$

value numbers



$i := i + 10$

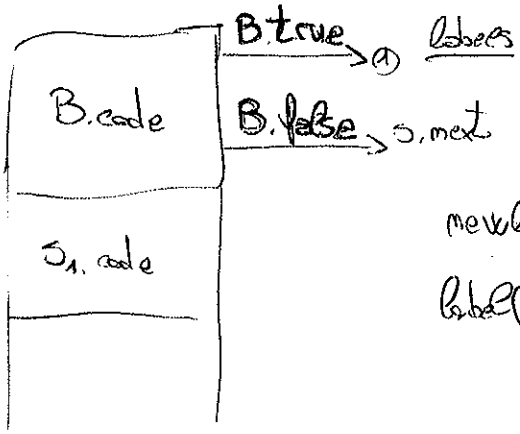


28/11/2012

```
Public class AB{
  Public static void main(String[] args){
    int a = 1;
    int b = 1;
    if((a==b) || (a++ > 140))
      System.out.println(a);
  }}
```

```
Publi class BA {
  Public static void main(String[] args){
    int a = 1;
    int b = 1;
    if((a++ > 140) || (a==b))
      System.out.println(a);
  }
}
```

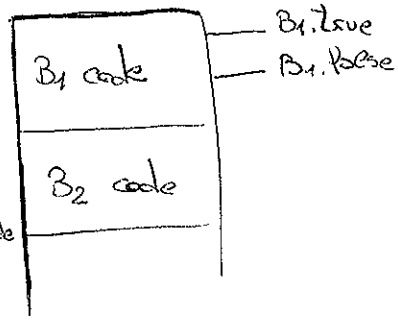
$P \rightarrow S$ S.next = newLabel()
P.code = S.code ▶ label(S.next) ▶ ...



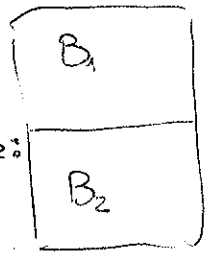
$S \rightarrow \text{if}(B)\ S_1$ B.true = newLabel()
B.false = S.next
S_1.next = S.next
S.code = B.code ▶ label(B.true) ▶ S_1.code

newLabel()
label(L) ▶ "code"

$B \rightarrow B_1 \,||\, B_2$ B_1.true = B.true
B_1.false = newLabel()
B_2.true = B.true
B_2.false = B.false
B.code = B_1.code ▶ label(B_1.false) ▶ B_2.code



$B \rightarrow B_1 \,\&\&\, B_2$ B_1.true = newLabel()
B_1.false = B.false
B_2.true = B.true
B_2.false = B.false
B.code = B_1.code ▶ label(B_1.true) ▶ B_2.code



$B \rightarrow \text{true}$ B.code = gen("GOTO" B.true)

$B \rightarrow \text{false}$ B.code = gen("GOTO" B.false)

$B \rightarrow E_1\ \text{rel}\ E_2$ E_1.code ▶ E_2.code ▶ gen(IF (E_1.V rel.op E_2.V) GOTO B.true) ▶ gen(GOTO B.false)

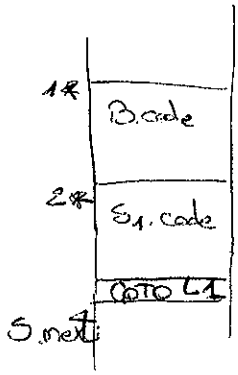
---

if(x<100 || (x>200 && x!=y)) x=2;

	Code(x<100)	if x < 100 GOTO L2 GOTO L3
L3	Code(x>200)	if x > 200 GOTO L4 GOTO L1
L4	Code x!=y	if x!=y GOTO L2 GOTO L1
L2	Code (x=2)	x = 2
L1		

$S \rightarrow$ WHILE (B) DO S_1

$S_1.next$ = newlabel() ①
$B.true$ = newlabel() ②
$B.false$ = $S.next$
$S.code$ = label($S_1.next$) ▶ $B.code$ ▶
label($B.true$) ▶ $S_1.code$ ▶
gen(goto $S_1.next$)



$B \rightarrow B_1 \,||\, B_2$

$B_1.false$ = redundant
~~$B_1.true = B.true$~~
$B_2.true$ = $B.true$
$B_2.false$ = $B.false$
$B.code$ = ~~$B_1.code$ ▶ $B_2.code$~~

$B_1.true$ = if $B.true \neq$ redundant
then $B.true$
else newlabel()

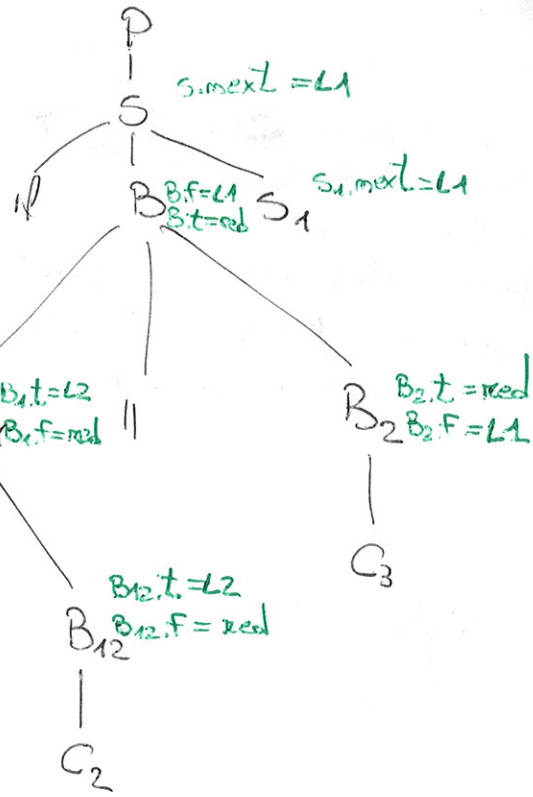
$B.code$ = if $B.true \neq$ redundant
then $B_1.code$ ▶ $B_2.code$
else $B_1.code$ ▶ $B_2.code$ ▶ label($B.true$)

$B \rightarrow B_1 \,\&\&\, B_2$

$B_1.true$ = redundant
$B_1.false$ = if $B.false \neq$ redundant
then $B.false$
else newlabel()

$B_2.true$ = $B.true$
$B_2.false$ = $B.false$
$B.code$ = if $B.false \neq$ redundant
then $B_1.code$ ▶ $B_2.code$
else $B_1.code$ ▶ $B_2.code$ ▶ label($B_1.false$)

if (C_1 && C_2 || C_3) { assignment }



P → S S.next = L1
S.code ▶ label(L1)

S → if (B) S_1 B.true = redundant
B.false = S.next
S_1.next = S.next
B.code ▶ S_1.code

B → B_1 || B_2 B_1.true = L2
B_1.false = red
B_2.true = red
B_2.false = L1
B_1.code ▶ B_2.code ▶ label(L2)

B_1 → B_{11} && B_{12} B_{11}.true = red
B_{11}.false = L3
B_{12}.true = L2
B_{12}.false = red
B_{11}.code ▶ B_{12}.code ▶ label(L3)

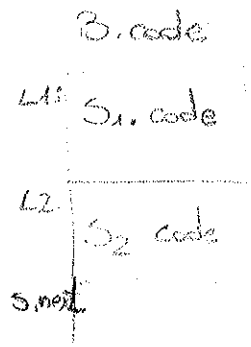
B_{11}.code
B_{12}.code
L3: B_2.code
L2: S_1.code
L1:

IFFALSE C_1 GOTO L3
IF C_2 GOTO L2
L3: IFFALSE C_3 GOTO L1
L2:
: S_1 ass
:
L1:

http://sites.google.com/site/cacoffotec2012/

$S \longrightarrow$ if (B) S_1 else S_2

$B.true$ = newlabel $L1$ ← redundant
$B.false$ = newlabel $L2$
$S_1.next = S.next$
$S_2.next = S.next$



$S.code = B.code \,||\, S_1.code \,||\, gen(goto\ S.next) \,||\, label(B.false) \,||\, S_2.code$

The following grammar generates a language for assignments

$S \rightarrow id := A \mid id := B$

$A \rightarrow A \,\&\, P \mid P$

$P \rightarrow id \mid (A)$

$B \rightarrow id$

Assume that:

* the symbol table for identifier records the type of identifiers
* there is a function newtemp() that generates a new temporary address (if needed in the 3-address code)

Write a translation ~~scheme~~ schema that generates the 3-address code for $L(G)$ and computes in the attribute $S.type$ the "type" of S defined to be int if S has the shape $id := A$ / $id := B$ and the type of id is int and the type A is int

id bool

bool

error otherwise

$S \longrightarrow id := A$ $\quad S.code = id.add = A.add$

$S \longrightarrow id := B$ $\quad S.type = \underline{if}\ B.type = bool\ \underline{then}\ bool\ \underline{else}\ error$ $\quad S.code = id.add = B.add$

$A \longrightarrow A_1 * P$ $\quad A.type = \underline{if}\ A.type = int\ AND\ P.type = int\ \underline{then}\ int\ \underline{else}\ error$ $\quad A.add = newtemp()$, $A.code = A_1.code \blacktriangleright gen(A.add = A_1.add * P.add)$

$A \longrightarrow P$ $\quad A.type = P.type$

$P \longrightarrow id$ $\quad P.type = id.type$

$P \longrightarrow (A)$ $\quad P.type = A.type$

$B \longrightarrow id$ $\quad B.type = id.type$ $\quad B.add = id.add$