Practice: Extend the number evaluation scheme to support grammar which has real numbers (rule number → sign list . list replaces number → sign list)

number → sign list	list.position ← 0
	if sign.negative
	then number.value ← - list.value
	else number.value ← list.value

sign → +
sign → -
	sign.negative ← false
	sign.negative ← true

list → bit	bit.position ← list.position
	list.value ← bit.value

$list_0$ → $list_1$ bit	$list_1$.position ← $list_0$.position + 1 bit.position ← $list_0$.position $list_0$.value ← $list_1$.value + bit.value

bit → 0
bit → 1
	bit.value ← 0
	bit.value ← $2^{\text{bit.position}}$

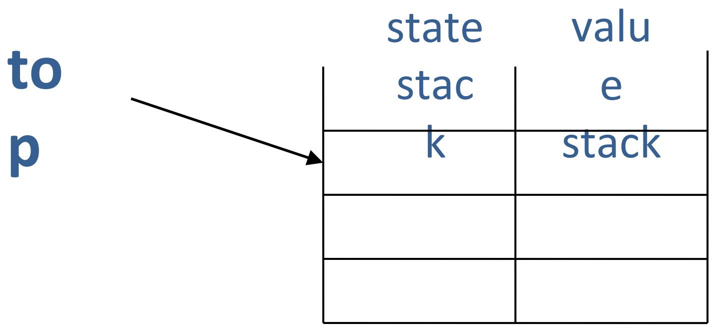
Compiler Design

Attribute Evaluation

Amey Karkare
Department of Computer Science and Engineering
IIT Kanpur
karkare@iitk.ac.in

Bottom-up evaluation of S-attributed definitions

- Can be evaluated while parsing
- Whenever reduction is made, value of new synthesized attribute is computed from the attributes on the stack
- Extend stack to hold the values also
- The current top of stack is indicated by top pointer



Bottom-up evaluation of S-attributed definitions

- Suppose semantic rule

  $$A.a = f(X.x, Y.y, Z.z)$$

  is associated with production

  $$A \rightarrow XYZ$$

- Before reducing XYZ to A, value of Z is in val(top), value of Y is in val(top-1) and value of X is in val(top-2)
- If symbol has no attribute then the entry is undefined
- After the reduction, top is decremented by 2 and state covering A is put in val(top)

Example: desk calculator

L → E $	Print (E.val)
E → E + T	E.val = E.val + T.val
E → T	E.val = T.val
T → T * F	T.val = T.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit.lexval

Example: desk calculator

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

Before reduction ntop = top - r +1
After code reduction top = ntop
r is the #symbols on RHS

Example: desk calculator

L → E$	print(val(top))
E → E + T	val(ntop) = val(top-2) + val(top)
E → T	
T → T * F	val(ntop) = val(top-2) * val(top)
T → F	
F → (E)	val(ntop) = val(top-1)
F → digit	

Before reduction	ntop = top - r +1
After code reduction	top = ntop

r is the #symbols on RHS

INPUT	**STATE**	**Val**	**PROD**
3*5+4$			
*5+4$	digit	3	
*5+4$	F	3	F → digit
*5+4$	T	3	T → F
5+4$	T*	3 □	
+4$	T*digit	3 □ 5	
+4$	T*F	3 □ 5	F → digit
+4$	T	15	T → T * F
+4$	E	15	E → T
4$	E+	15 □	
$	E+digit	15 □ 4	
$	E+F	15 □ 4	F → digit
$	E+T	15 □ 4	T → F
$	E	19	E → E +T

□ is a dummy placeholder.

YACC Terminology

$E \rightarrow E + T$ val(ntop) = val(top-2) + val(top)

In YACC

$E \rightarrow E + T$ $$ = $1 + $3

$$ maps to val[top – r + 1]

$k maps to val[top – r + k]

r=#symbols on RHS (here 3)

$$ = $1 is the *default* action in YACC

L-attributed definitions

- When translation takes place during parsing, order of evaluation is linked to the order in which nodes are created
- In S-attributed definitions parent’s attribute evaluated after child’s.
- A natural order in both top-down and bottom-up parsing is depth first-order
- L-attributed definition: where attributes can be evaluated in depth-first order

L attributed definitions …

- A syntax directed definition is L-attributed if each inherited attribute of X_j $(1 \le j \le n)$ at the right hand side of $A \to X_1 X_2 \ldots X_n$ depends only on
  - Attributes of symbols $X_1 X_2 \ldots X_{j-1}$ and
  - Inherited attribute of A
- Examples (i inherited, s synthesized)

$A \to LM$

$L.i = f_1(A.i)$
$M.i = f_2(L.s)$
$A.s = f_3(M.s)$

✓

$A \to QR$

$R.i = f4(A.i)$
$Q.i = f5(R.s)$
$A.s = f6(Q.s)$

✗

Translation schemes

- A CFG where semantic actions occur within the rhs of production
- Example: A translation scheme to map infix to postfix

  E→ T R

  R→ addop T R | ε

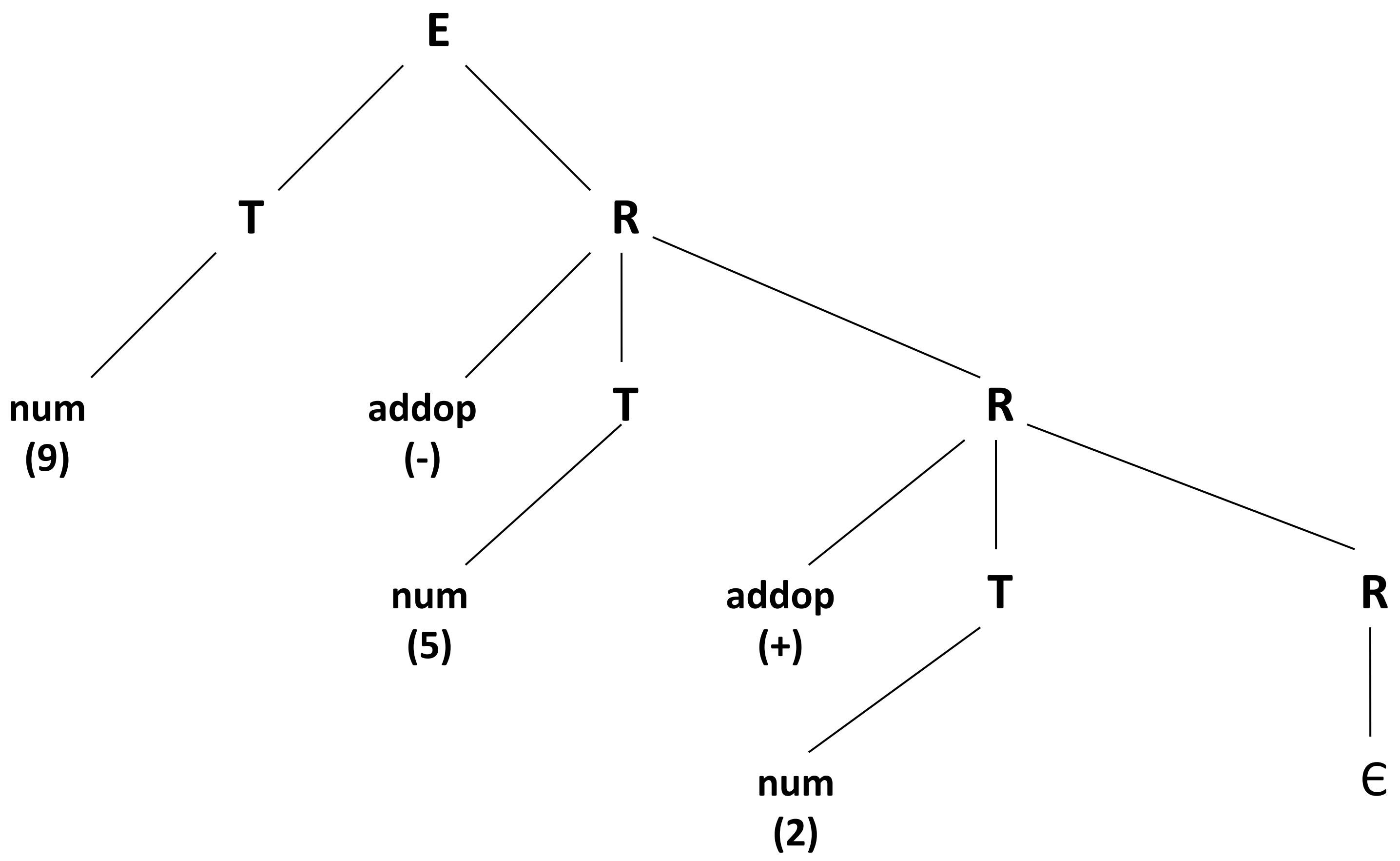
  T→ num

addop → + | –

Exercises: 1) Create Parse Tree for 9 – 5 + 2

2)Add actions to convert infix to postfix

Parse tree for 9-5+2



Translation schemes

- A CFG where semantic actions occur within the rhs of production
- Example: A translation scheme to map infix to postfix

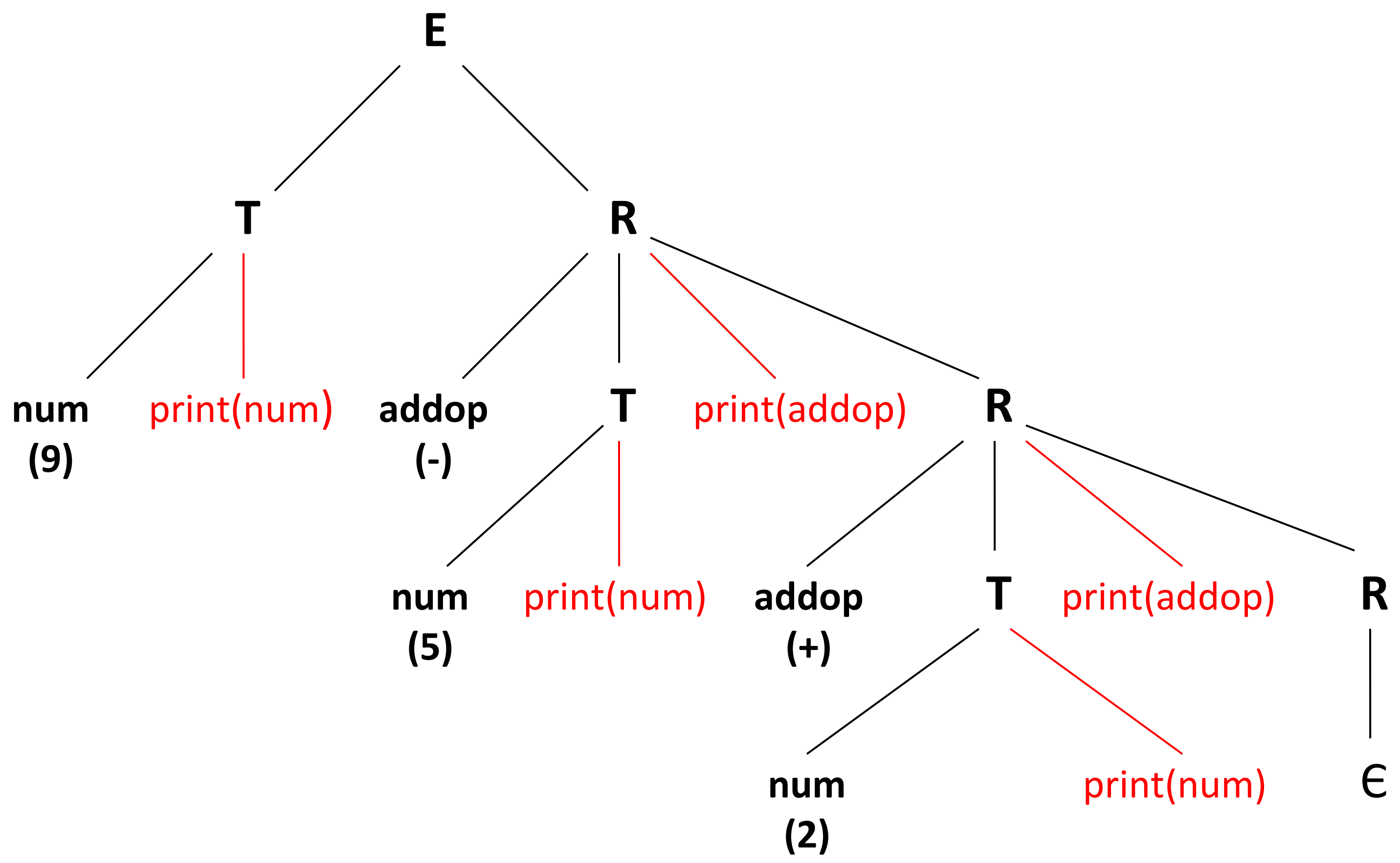
  $E \rightarrow T\ R$

  $R \rightarrow$ addop T {print(addop)} R | ε

  $T \rightarrow$ num {print(num)}

addop → + | –

Parse tree for 9-5+2



Evaluation of Translation Schemes

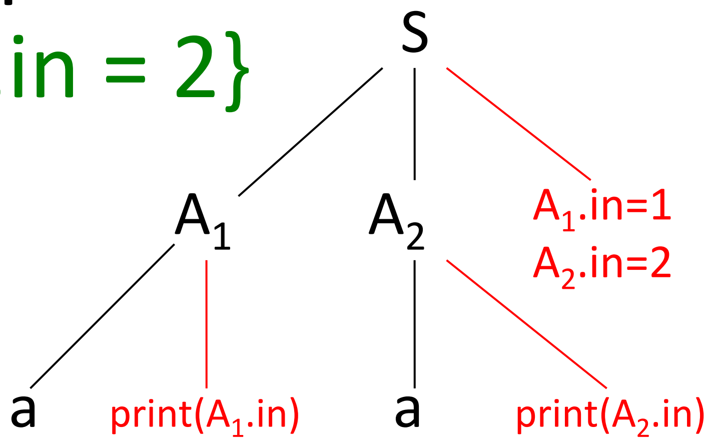
- Assume actions are terminal symbols
- Perform depth first order traversal to obtain 9 5 – 2 +

- When designing translation scheme, **ensure** attribute value is available when referred to
- In case of synthesized attribute it is trivial **(why ?)**

- An inherited attribute for a symbol on RHS of a production must be computed in an action before that symbol

  $S \rightarrow A_1\ A_2 \quad \{A_1.in = 1, A_2.in = 2\}$

  $A \rightarrow a \quad \{print(A.in)\}$



depth first order traversal gives error (*undef)*

- A synthesized attribute for the non terminal on the LHS is computed after all attributes it references, have been computed. **The action normally should be placed at the end of RHS.**

Example: Translation scheme for EQN (LaTeX like equations)

$S \rightarrow B$

$B \rightarrow B_1\ B_2$

$B \rightarrow B_1$ sub B_2

$B \rightarrow$ text

Example: Translation scheme for EQN (LaTeX like equations)

$S \to B$	$B.pts = 10$ $S.ht = B.ht$
$B \to B_1\ B_2$	$B_1.pts = B.pts$ $B_2.pts = B.pts$ $B.ht = max(B_1.ht, B_2.ht)$
$B \to B_1$ sub B_2	$B_1.pts = B.pts;$ $B_2.pts = shrink(B.pts)$ $B.ht = disp(B_1.ht, B_2.ht)$
$B \to$ text	$B.ht = text.h * B.pts$

After putting actions in the right place

$S \to$ {B.pts = 10} B
{S.ht = B.ht}

$B \to$ {B_1.pts = B.pts} B_1
{B_2.pts = B.pts} B_2
{B.ht = max(B_1.ht,B_2.ht)}

$B \to$ {B_1.pts = B.pts} B_1 sub
{B_2.pts = shrink(B.pts)} B_2
{B.ht = disp(B_1.ht,B_2.ht)}

$B \to$ text {B.ht = text.h * B.pts}

Bottom up evaluation of inherited attributes

- Remove embedded actions from translation scheme
- Make transformation so that embedded actions occur only at the ends of their productions
- Replace each action by a distinct marker non terminal M and attach action at end of $M \rightarrow \varepsilon$

```
E → T R
R → + T {print (+)} R
R → - T {print (-)} R
R → ϵ
T → num {print(num.val)}
```

transforms to

```
E → T R
R → + T M R
R → - T N R
R → ϵ
T → num         {print(num.val)}
M → ϵ           {print(+)}
N → ϵ           {print(-)}
```

Inheriting attribute on parser stacks

- bottom up parser reduces rhs of A → XY by removing XY from stack and putting A on the stack
- Suppose synthesized attributes of X is inherited by Y by using the copy rule Y.i=X.s
- X.s is already on the parser stack before any reductions take place in the sub-tree below Y
  - X.s can be used easily

Recall: SDD for Inherited Attributes

$D \rightarrow T\ L$	L.in = T.type
$T \rightarrow$ real	T.type = real
$T \rightarrow$ int	T.type = int
$L \rightarrow L_1$, id	L_1.in = L.in; addtype(id.entry, L.in)
$L \rightarrow$ id	addtype (id.entry,L.in)

Exercise: Convert to Translation Scheme

Inherited Attributes: Translation Scheme

D → T {L.in = T.type} L

T → int {T.type = integer}
T → real {T.type = real}

L → {L_1.in =L.in} L_1,id {addtype(id.entry,L_{in})}

L → id {addtype(id.entry,L_{in})}

Example: take string real p,q,r

State stack	INPUT	PRODUCTION
	real p,q,r	
real	p,q,r	
T	p,q,r	T → real
Tp	,q,r	
TL	,q,r	L → id
TL,	q,r	
TL,q	,r	
TL	,r	L → L,id
TL,	r	
TL,r	-	
TL	-	L → L,id
D	-	D →TL

Observation: Every time a string is reduced to L, T is just below it on the stack

Example …

- Every time a reduction to L is made, the value of T type is just below it
- Use the fact that T.type is at a known place in the value stack
- When production L → id is applied, id.entry is at the top of the stack and T.type is just below it, therefore,

addtype(id.entry,L.in) ⇔
addtype(val[top], val[top-1])

- Similarly when production L → L_1 , id is applied id.entry is at the top of the stack and T.type is three places below it, therefore,

addtype(id.entry, L.in) ⇔
addtype(val[top],val[top-3])

Example …

Therefore, the translation scheme becomes

D → T L	
T → int	val[top] =integer
T → real	val[top] =real
L → L,id	addtype(val[top], val[top-3])
L → id	addtype(val[top], val[top-1])

Simulating the evaluation of inherited attributes

- The scheme works only if grammar allows position of attribute to be predicted.
- Consider the grammar

$S \rightarrow aAC$	$C_i = A_s$
$S \rightarrow bABC$	$C_i = A_s$
$C \rightarrow c$	$C_s = g(C_i)$

- C inherits A_s
- there may or may not be a B between A and C on the stack when reduction by rule $C \rightarrow c$ takes place
- When reduction by $C \rightarrow c$ is performed the value of C_i is either in [top-1] or [top-2]

Simulating the evaluation …

- Insert a marker M just before C in the second rule and change rules to

$S \rightarrow aAC$	$C_i = A_s$
$S \rightarrow bABMC$	$M_i = A_s$; $C_i = M_s$
$C \rightarrow c$	$C_s = g(C_i)$
$M \rightarrow \varepsilon$	$M_s = M_i$

- When production $M \rightarrow \varepsilon$ is applied we have $M_s = M_i = A_s$
- Therefore value of C_i is always at val[top-1]

Simulating the evaluation …

- Markers can also be used to simulate rules that are **not copy rules**

$S \rightarrow aAC$	$C_i = f(A.s)$

- using a marker

$S \rightarrow aANC$	$N_i = A_s;\ C_i = N_s$
$N \rightarrow \varepsilon$	$N_s = f(N_i)$

- **Algorithm:** Bottom up parsing and translation with inherited attributes
- **Input**: L attributed definitions
- **Output**: A bottom up parser
- Assume every non terminal has one inherited attribute and every grammar symbol has a synthesized attribute
- For every production $A \rightarrow X_1 \dots X_n$ introduce n markers $M_1 \dots M_n$ and replace the production by

$$A \rightarrow M_1 X_1 \dots M_n X_n$$

$$M_1 \rightarrow \epsilon, \dots, M_n \; ? \; \epsilon$$

- Synthesized attribute $X_{j,s}$ goes into the value entry of X_j
- Inherited attribute $X_{j,i}$ goes into the value entry of M_j

Algorithm …

- If the reduction is to a marker M_j and the marker belongs to a production

$$A \rightarrow M_1\ X_1 \ldots\ M_n X_n$$

Then for computation of $X_{j.i}$

$X_{2.s}$ is in position top-2j+6

$X_{2.i}$ is in position top-2j+5

$X_{1.s}$ is in position top-2j+4

$X_{1.i}$ is in position top-2j+3

A_i is in position top-2j+2

Thus, all the attributes can be computed during the parsing.

Space for attributes at compile time

- Lifetime of an attribute begins when it is first computed
- Lifetime of an attribute ends when all the attributes depending on it, have been computed
- Space can be conserved by assigning space for an attribute only during its lifetime

Example

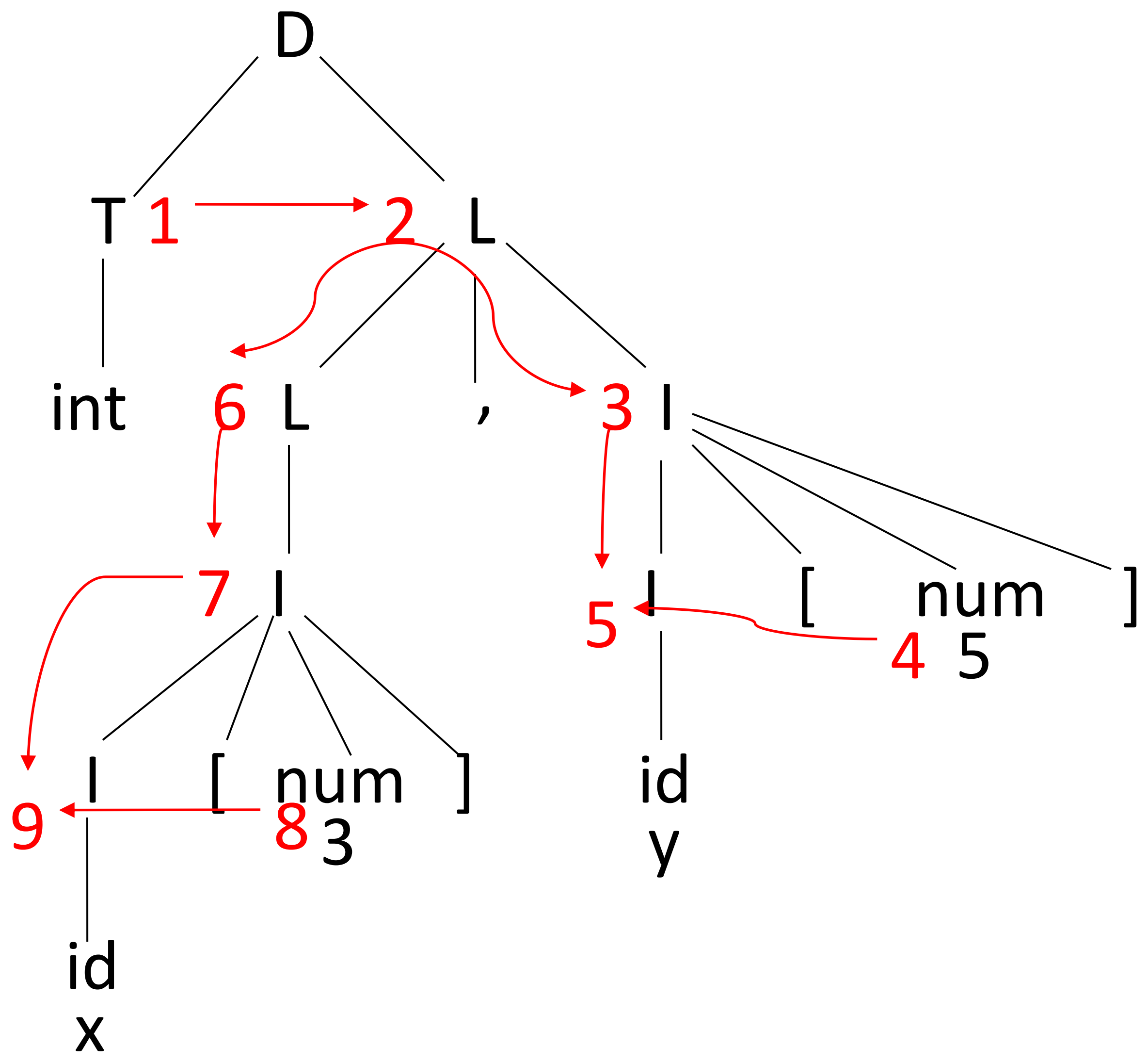
- Consider following definition

$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ real	T.type := real
$T \rightarrow$ int	T.type := int
$L \rightarrow L_1, I$	L_1.in :=L.in; I.in=L.in
$L \rightarrow I$	I.in = L.in
$I \rightarrow I_1$[num]	I_1.in=array(num, I.in)
$I \rightarrow$ id	addtype(id.entry,I.in)

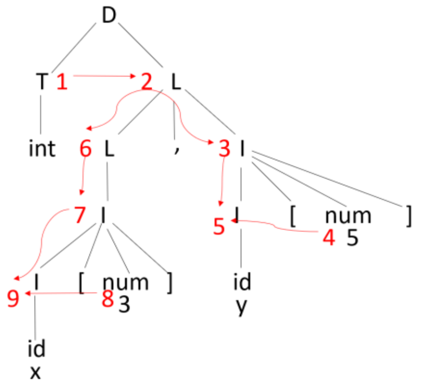
Consider string int x[3], y[5]

its parse tree and dependence graph

Consider string int x[3], y[5]



Resource requirement



$$\begin{tikzcd}
1 \arrow[r] & 2 \arrow[r] \arrow[rrr, bend left] \arrow[rrrr, bend left] & 3 & 4 \arrow[r] & 5 & 6 \arrow[r] & 7 \arrow[rr, bend left] & 8 \arrow[r] & 9
\end{tikzcd}$$

Allocate resources using
life time information

R1 R1 R2 R3 R2 R1 R1 R2 R1

Allocate resources using lifetime and copy information

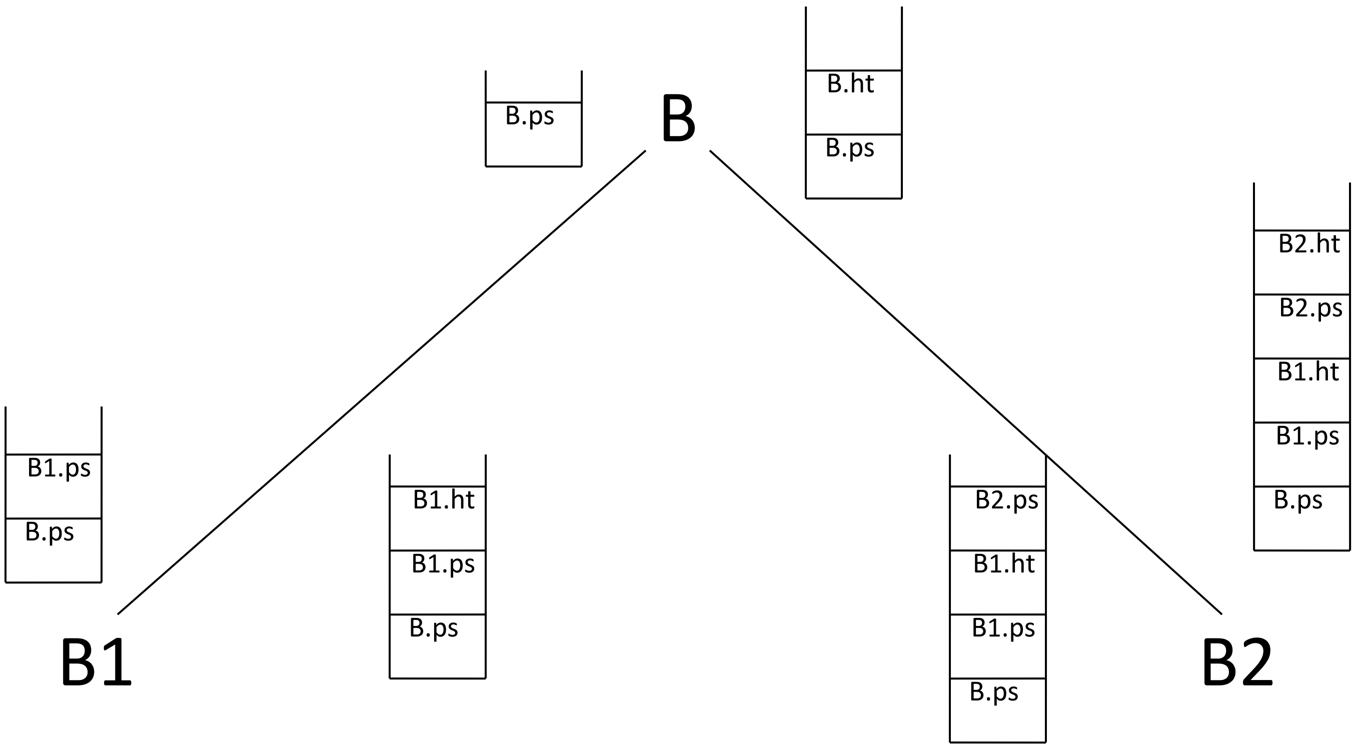
R1 =R1 =R1 R2 R2 =R1 =R1 R2 R1

Space for attributes at Compiler Construction time

- Attributes can be held on a single stack. However, lot of attributes are copies of other attributes
- For a rule like A →B C stack grows up to a height of five (assuming each symbol has one inherited and one synthesized attribute)
- Just before reduction by the rule A →B C the stack contains I(A) I(B) S(B) I(C) S(C)
- After reduction the stack contains I(A) S(A)
- Using multiple stacks can help in reducing space

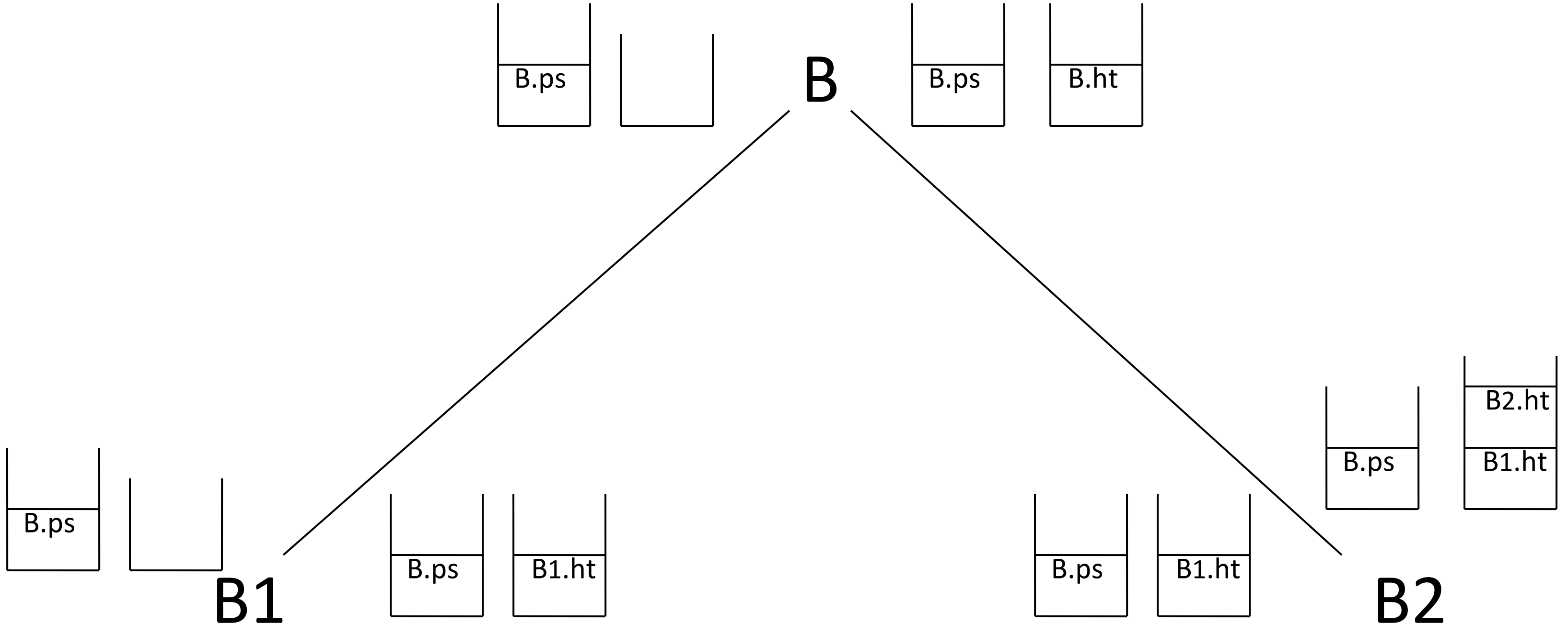
Example

- Consider rule B →B1 B2 with inherited attribute ps and synthesized attribute ht
- The parse tree for this string and a snapshot of the stack at each node appears as



Example …

- However, if different stacks are maintained for the inherited and synthesized attributes, the stacks will normally be smaller



Reading Assignment

Section 5.5, 5.9 of the OLD Dragonbook (3 authors: Aho, Sethi, Ullman).