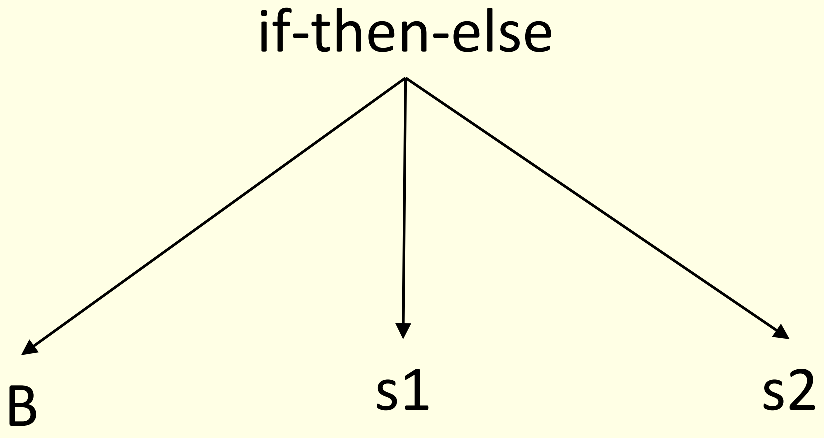
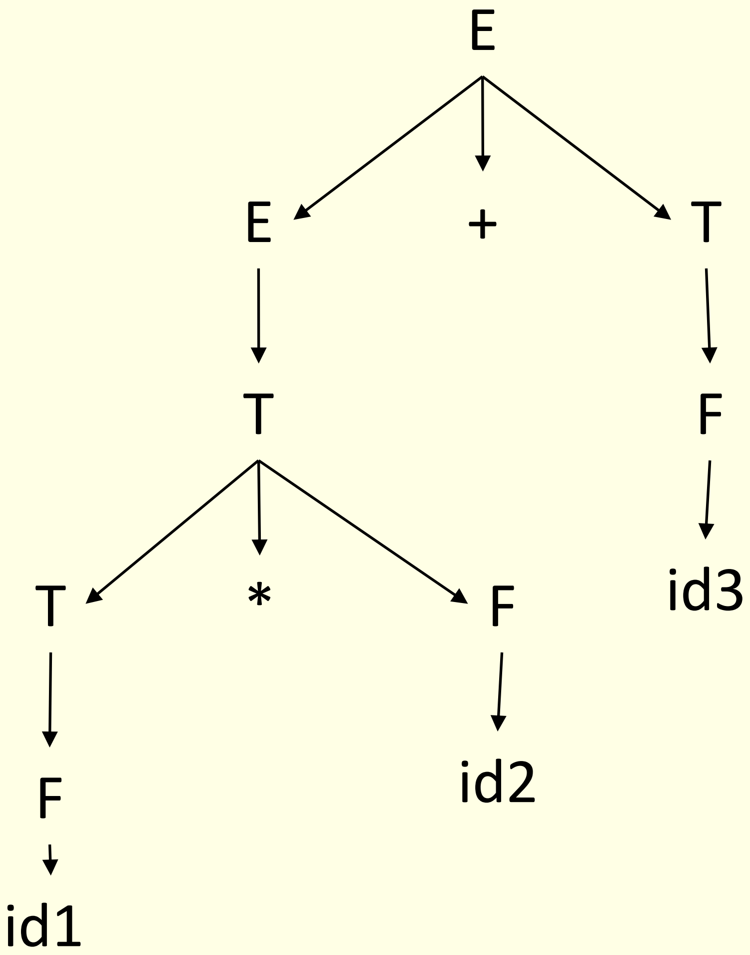
Abstract Syntax Tree

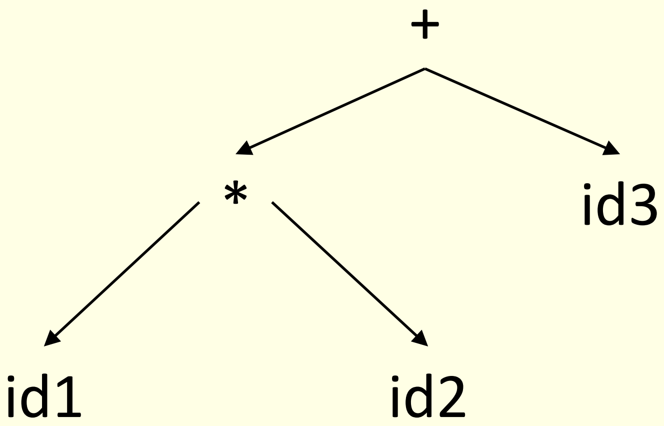
- Condensed form of parse tree,
- useful for representing language constructs.
- The production S → if B then s1 else s2
  may appear as



Abstract Syntax tree …

- Chain of single productions may be collapsed, and operators move to the parent nodes





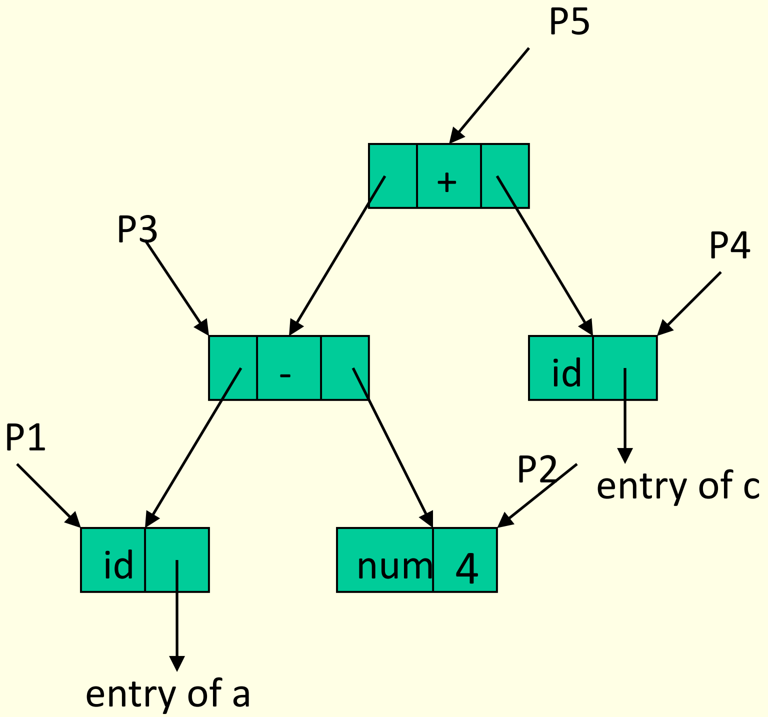
Constructing Abstract Syntax Tree for expression

- Each node can be represented as a record
- *operators*: one field for operator, remaining fields ptrs to operands
  - mknode(op,left,right)
- *identifier*: one field with label id and another ptr to symbol table
  - mkleaf(id,entry)
- *number*: one field with label num and another to keep the value of the number
  - mkleaf(num,val)

Example

the following sequence of function calls creates a parse tree for a- 4 + c

P_1 = mkleaf(id, entry.a)
P_2 = mkleaf(num, 4)
P_3 = mknode(-, P_1, P_2)
P_4 = mkleaf(id, entry.c)
P_5 = mknode(+, P_3, P_4)



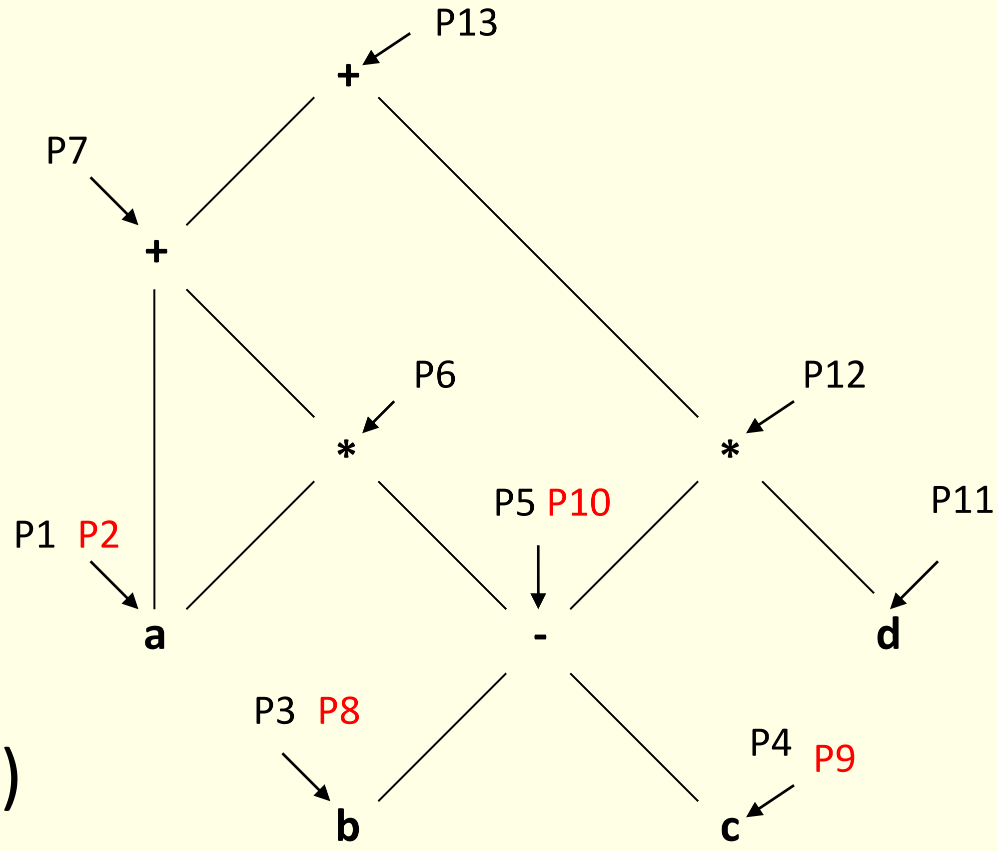
A syntax directed definition for constructing syntax tree

$E \rightarrow E_1 + T$	E.ptr = mknode(+, E_1.ptr, T.ptr)
$E \rightarrow T$	E.ptr = T.ptr
$T \rightarrow T_1 * F$	T.ptr := mknode(*, T_1.ptr, F.ptr)
$T \rightarrow F$	T.ptr := F.ptr
$F \rightarrow (E)$	F.ptr := E.ptr
$F \rightarrow \text{id}$	F.ptr := mkleaf(id, entry.id)
$F \rightarrow \text{num}$	F.ptr := mkleaf(num,val)

DAG for Expressions

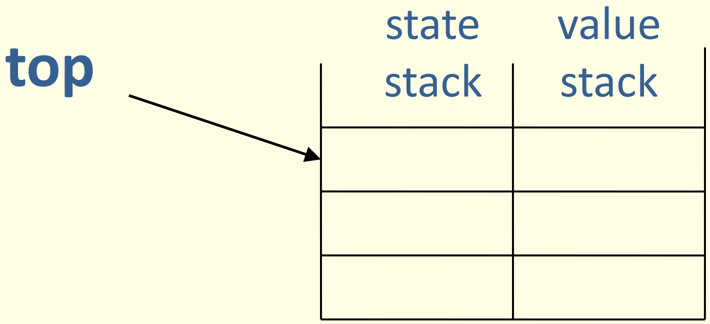
Expression a + a * (b – c) + (b - c) * d
make a leaf or node if not present,
otherwise return pointer to the existing node

P_1 = makeleaf(id,a)
P_2 = makeleaf(id,a)
P_3 = makeleaf(id,b)
P_4 = makeleaf(id,c)
P_5 = makenode(-,P_3,P_4)
P_6 = makenode(*,P_2,P_5)
P_7 = makenode(+,P_1,P_6)
P_8 = makeleaf(id,b)
P_9 = makeleaf(id,c)
P_{10} = makenode(-,P_8,P_9)
P_{11} = makeleaf(id,d)
P_{12} = makenode(*,P_{10},P_{11})
P_{13} = makenode(+,P_7,P_{12})



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

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

YACC Terminology

E → E + T   val(ntop) = val(top-2) + val(top)

In YACC
E → 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 ($1 \leq j \leq n$) at the right hand side of $A \rightarrow X_1\ X_2 \ldots X_n$ depends only on
  - Attributes of symbols $X_1\ X_2 \ldots X_{j-1}$ and
  - Inherited attribute of A
- Examples (i inherited, s synthesized)

$A \rightarrow LM$

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

✓

$A \rightarrow QR$

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

✗

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

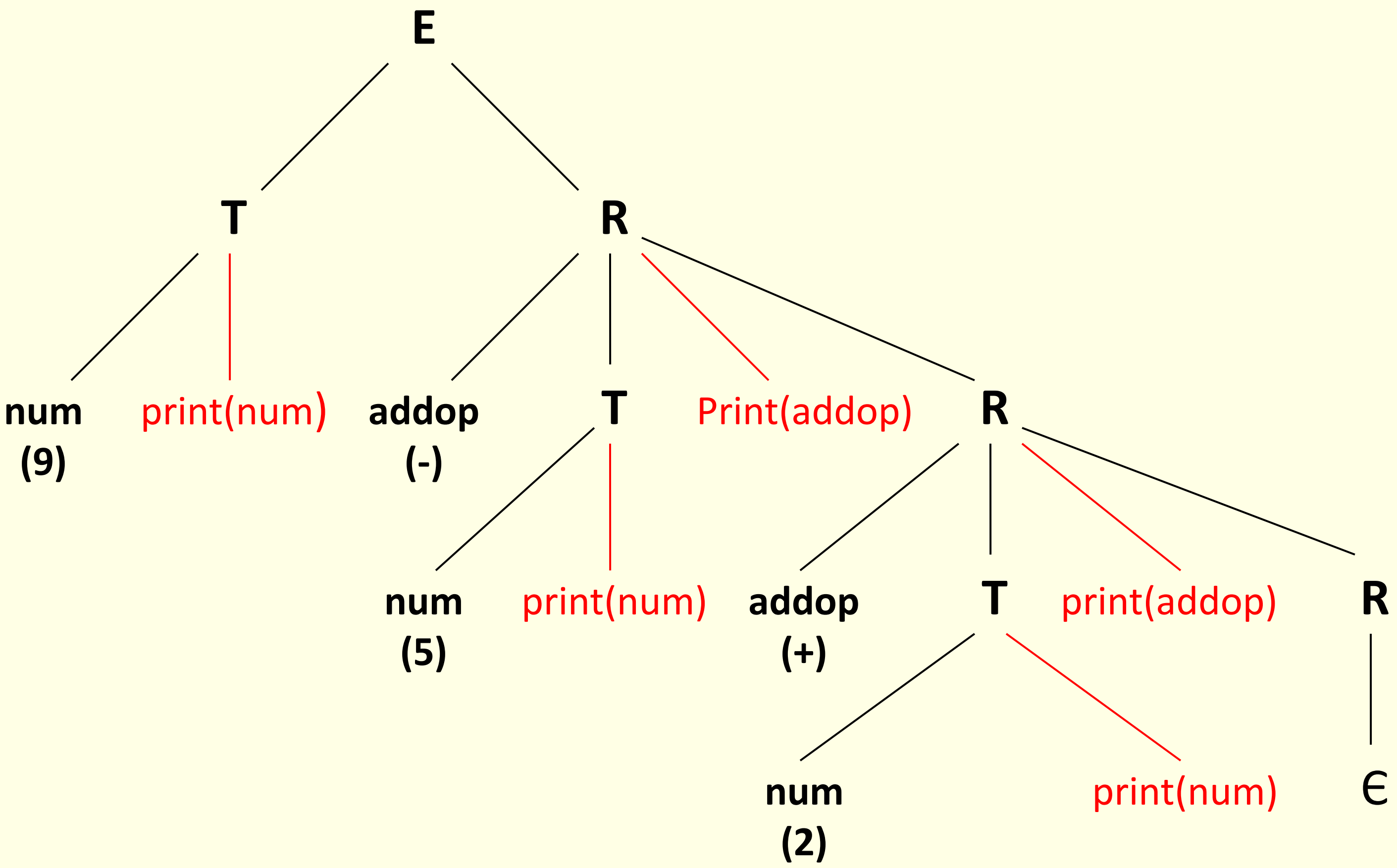
  R → addop T R | ε

  T→ num

addop → + | –

Exercise: Create Parse Tree for 9 – 5 + 2

Parse tree for 9-5+2



E

T

R

num
(9)

print(num)

addop
(-)

T

Print(addop)

R

num
(5)

print(num)

addop
(+)

T

print(addop)

R

num
(2)

print(num)

ϵ

Evaluation of Translation Schemes

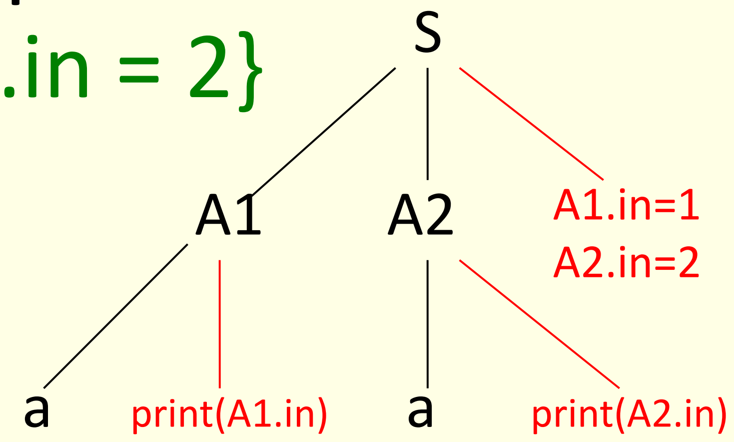
- Assume actions are terminal symbols
- Perform depth first order traversal to obtain 9 5 – 2 +

- 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 can be computed after all attributes it references, have been computed. **The action normally should be placed at the end of RHS.**

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 → ∈
T → num {print(num.val)}

transforms to

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

- synthesized attributes of Xs can be inherited by Y by using the copy rule Y.i=X.s

Inherited Attributes: SDD

$D \rightarrow T\ L$	L.in = T.type
$T \rightarrow$ real	T.type = real
T 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

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

Example …

- Every time a reduction to L is made value of T type is just below it
- Use the fact that T.val (type information) is at a known place in the 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→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

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

- using a marker

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

General algorithm

- **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 $A \rightarrow X_1 \ldots X_n$ introduce n markers $M_1 \ldots . M_n$ and replace the production by

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

- 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

  A_i is in position top-2j+2
  $X_{1.i}$ is in position top-2j+3
  $X_{1.s}$ is in position top-2j+4

- If reduction is to a non terminal A by production $A \rightarrow M_1\ X_1 \ldots M_n X_n$ then compute A_s and push on the stack

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

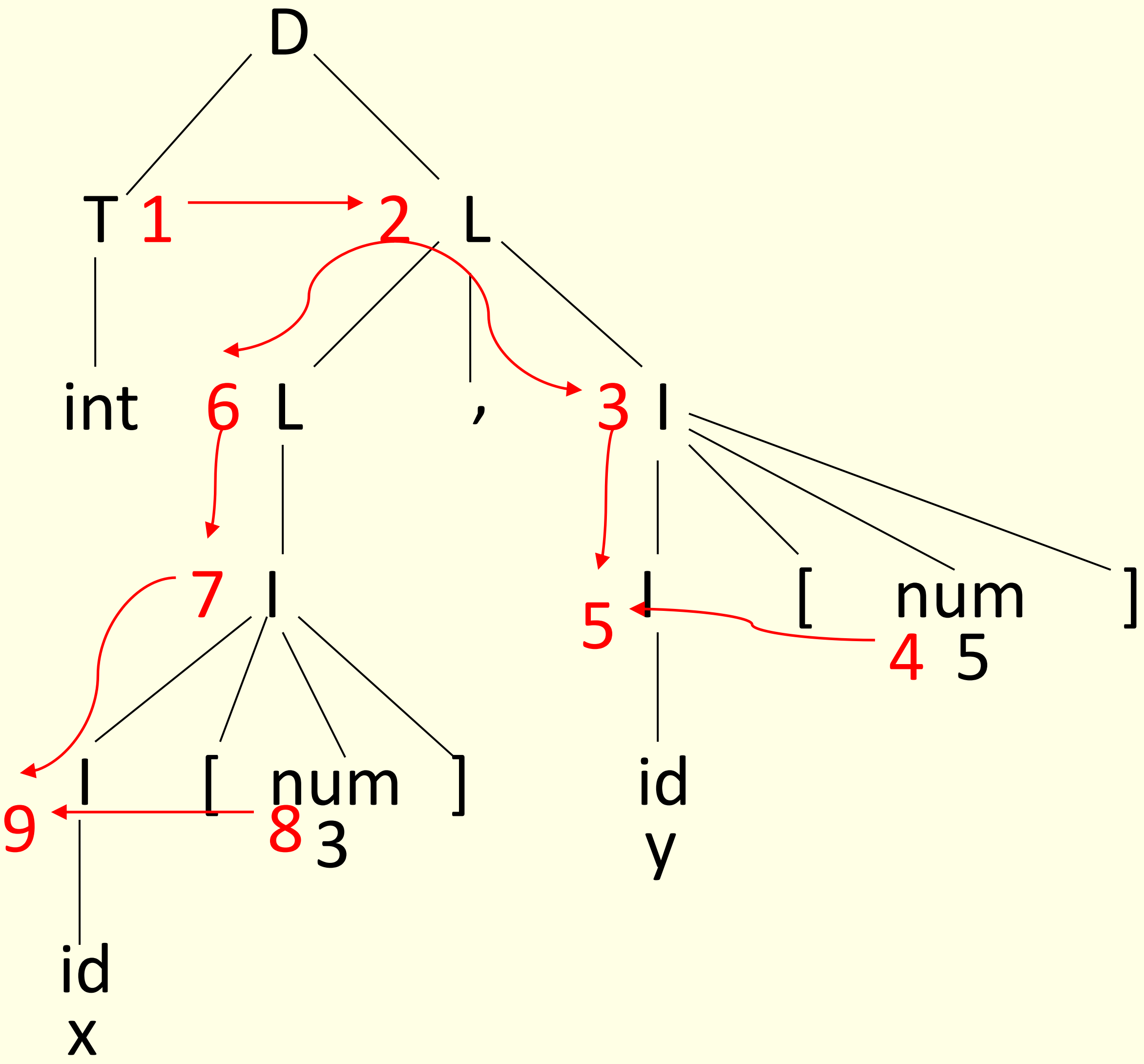
Example

- Consider following definition

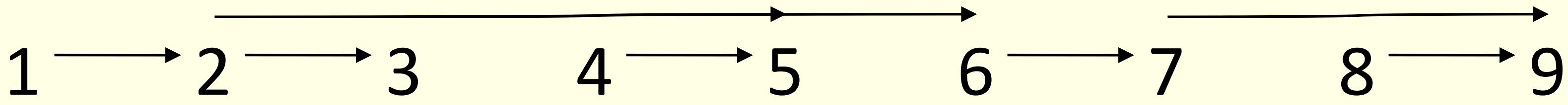
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ real	T.type := real
$T \rightarrow$ int	T.type := int
$L \rightarrow L_1,I$	L_1.in :=L.in; I.in=L.in
$L \rightarrow I$	I.in = L.in
$I \rightarrow I_1$[num]	I_1.in=array(numeral, I.in)
$I \rightarrow$ id	addtype(id.entry,I.in)

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

its parse tree and dependence graph



Resource requirement



Allocate resources using life time information

R1 R1 R2 R3 R2 R1 R1 R2 R1

Allocate resources using life time and copy information

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

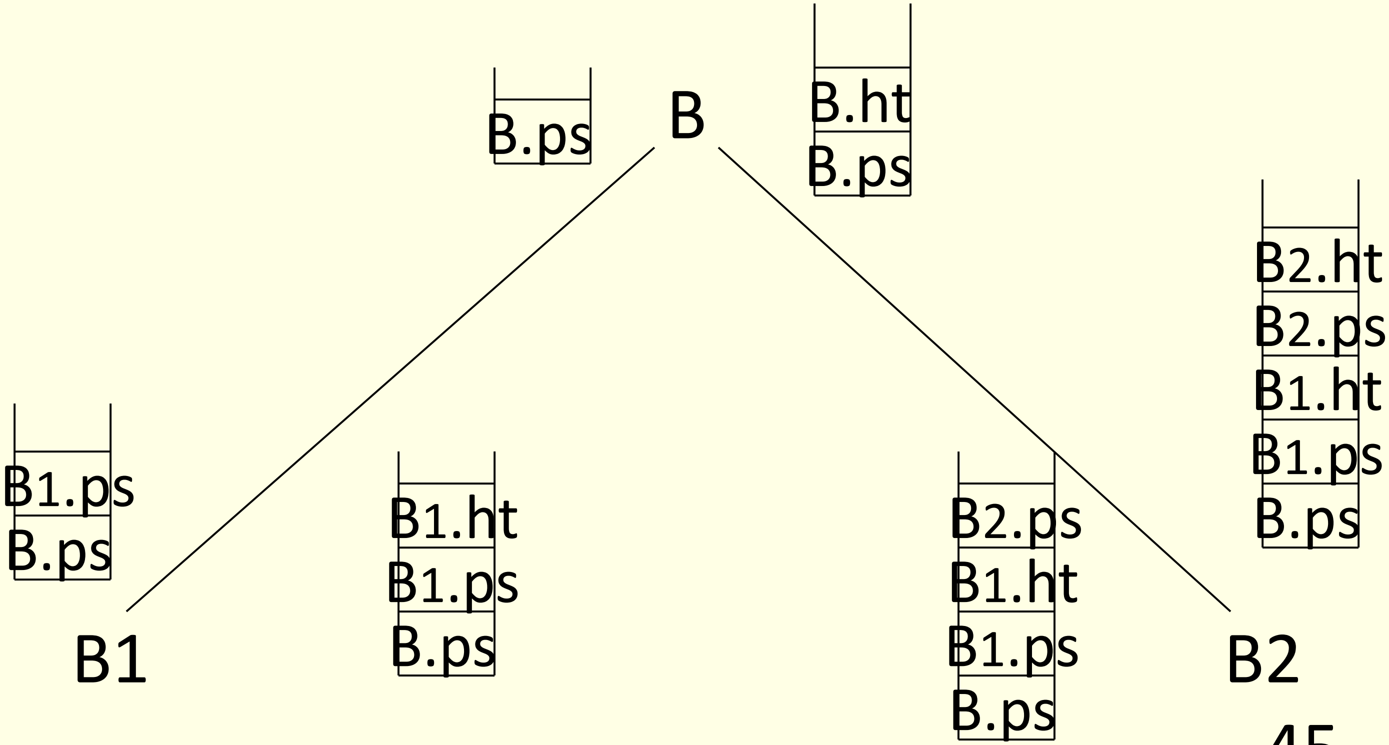
Space for attributes at compiler Construction time

- 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)
-

Example

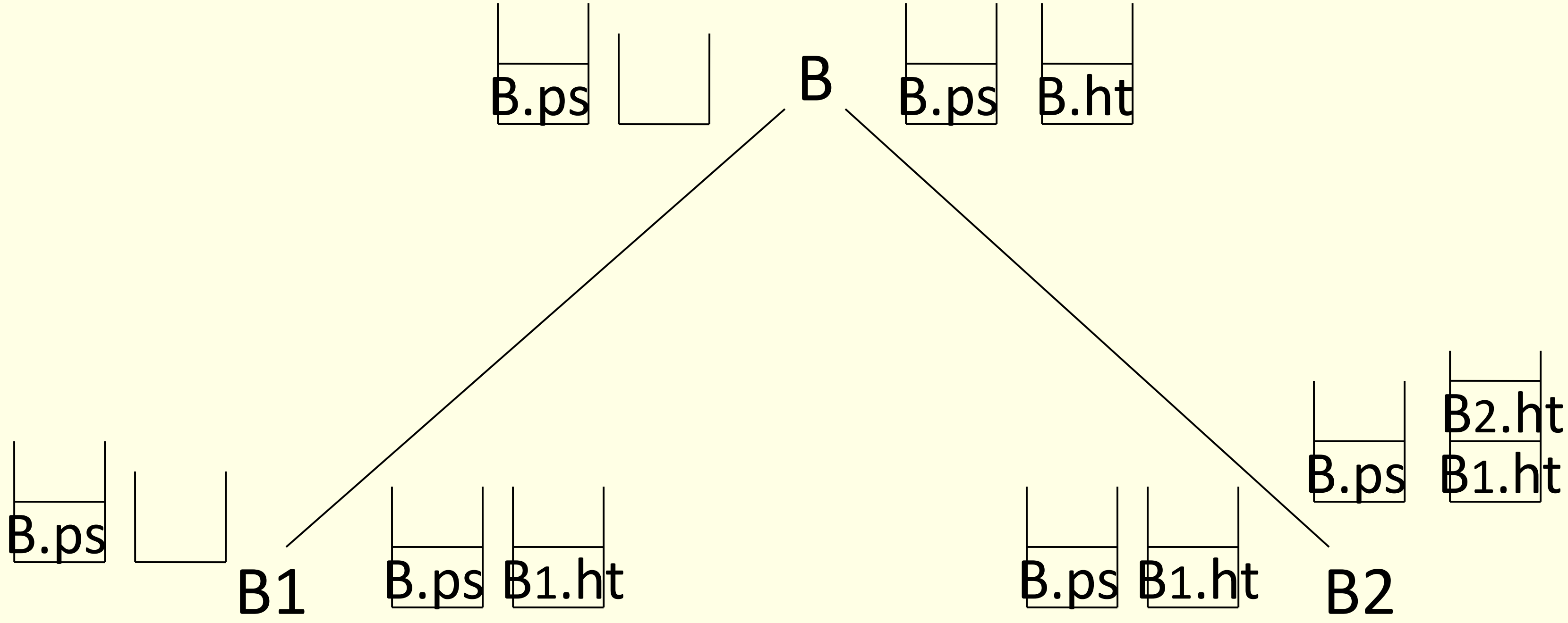
- Consider rule B →B1 B2 with inherited attribute ps and synthesized attribute ht

- 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



B.ps

B

B.ps

B.ht

B.ps

B1

B.ps

B1.ht

B.ps

B1.ht

B2

B.ps

B2.ht

B1.ht