Syntax directed translation

DR PARKAVI A

Syntax directed Translation

- Attributes attached to grammar symbols
- Semantic actions
- Example

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$

SDT

- **EXAMPLE**

a syntax-directed translation scheme embeds program fragments called semantic actions within production bodies, as in

$$E \rightarrow E_1 + T \quad \{ \text{ print } '+' \ \}$$

By convention, semantic actions are enclosed within curly braces.

SDT

- Translation can be done during parsing
- Can be shown in parse tree
- L-Attributed translations
  - Left to right
  - Top down parsing
- S-Attributed
  - Synthesized
  - Bottom up parsing

Syntax directed definitions: SDD

- CFG with rules and attributes
- Attributes are associated with grammar symbols
- Rules are attached to production rules
- What is what ???
  - X.a
  - Attributes may be strings, numbers, types, table references, instances

Inherited and synthesized attributes

1. A *synthesized attribute* for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head. A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

2. An *inherited attribute* for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body. An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

Continued

- Attributes for terminals
  - Their lexical values

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

Syntax-directed definition of a simple desk calculator

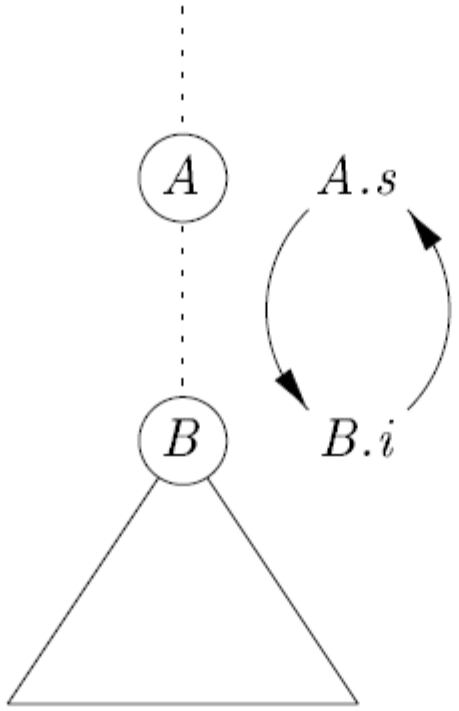
Evaluating SDD at the nodes of a parse tree

- A Parse tree with attributes and values → annotated parse tree
- Post order traversal of parse tree
- Bottom up order
- Evaluation of S-attributed tree
- Evaluation order
- Circular dependency

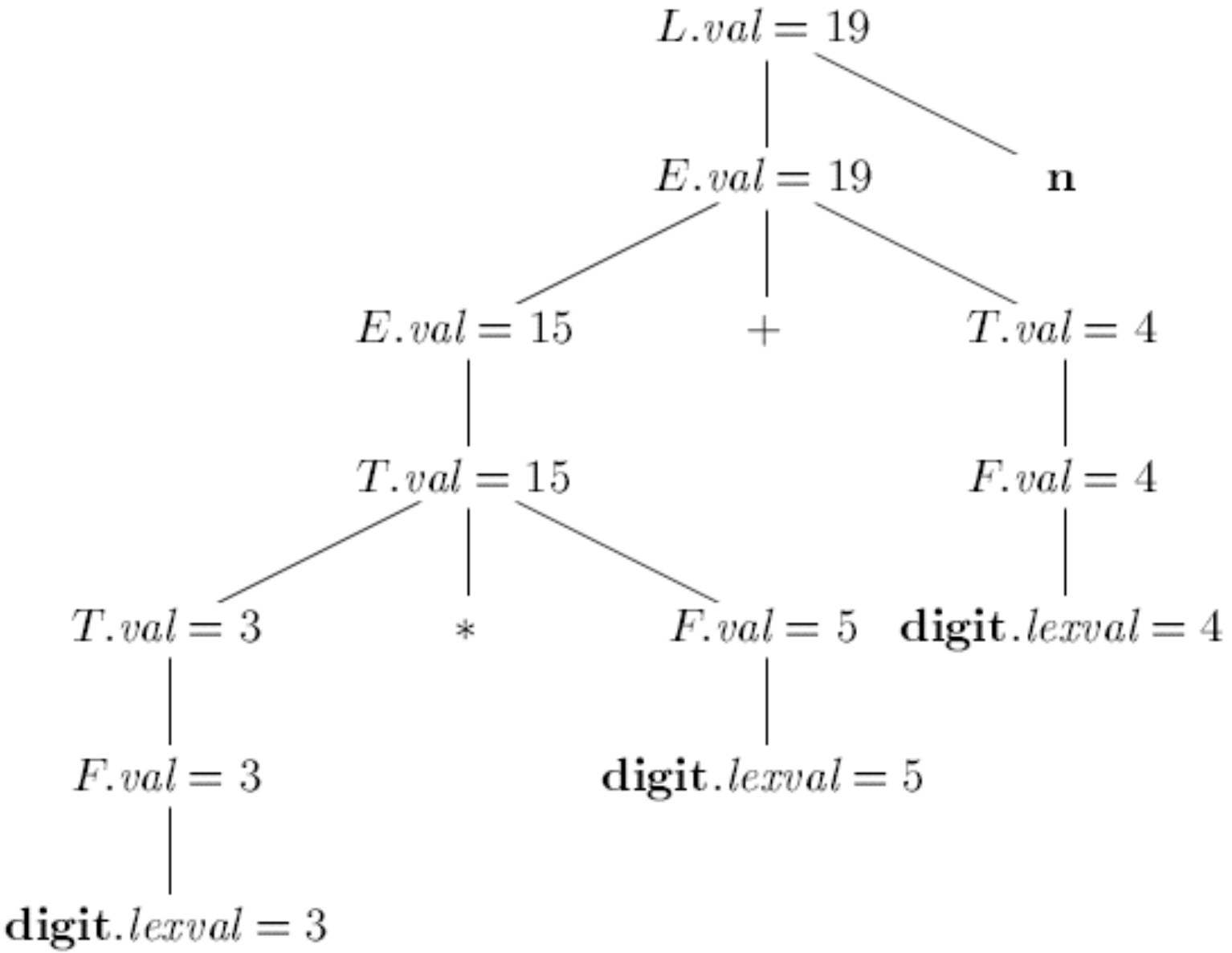
Example of S and L- attributes in a grammar

PRODUCTION	SEMANTIC RULES
$A \rightarrow B$	$A.s = B.i;$
	$B.i \mid= A.s + 1$

Continued



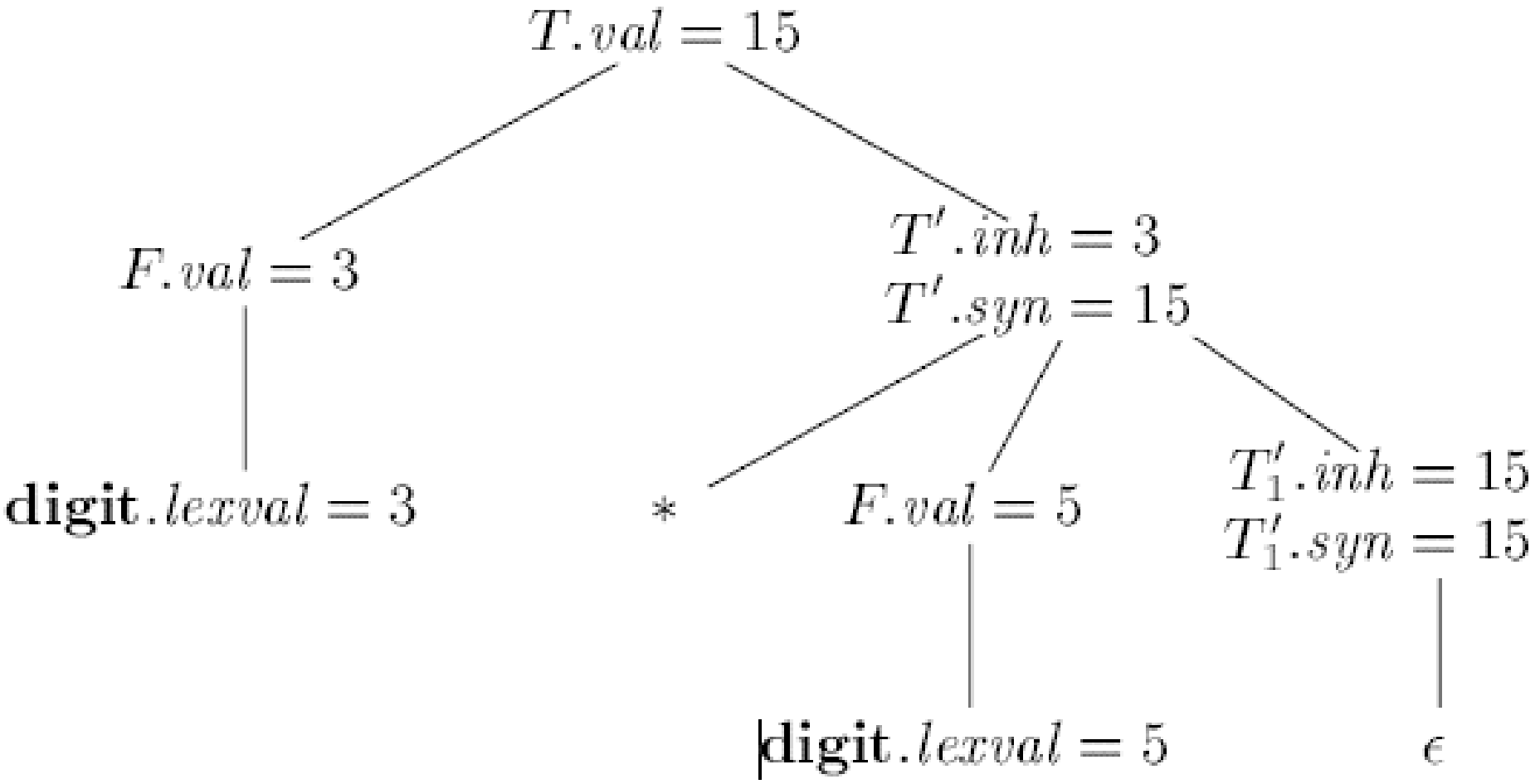
The circular dependency of $A.s$ and $B.i$ on one another



Annotated parse tree for $3 * 5 + 4$ **n**

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

An SDD based on a grammar suitable for top-down parsing



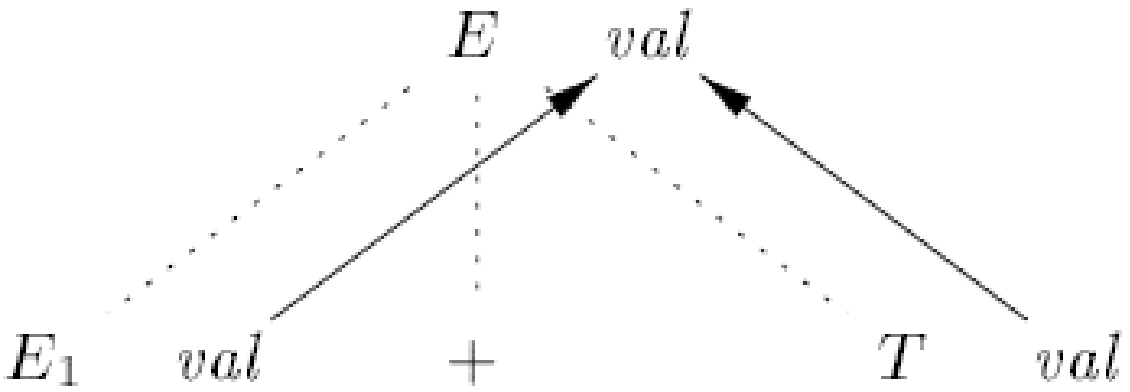
Annotated parse tree for $3 * 5$

Evaluation order for SDD's

- Dependency graph
- Each nodes is associated with attributes
- Edges between nodes
- Annotated parse tree

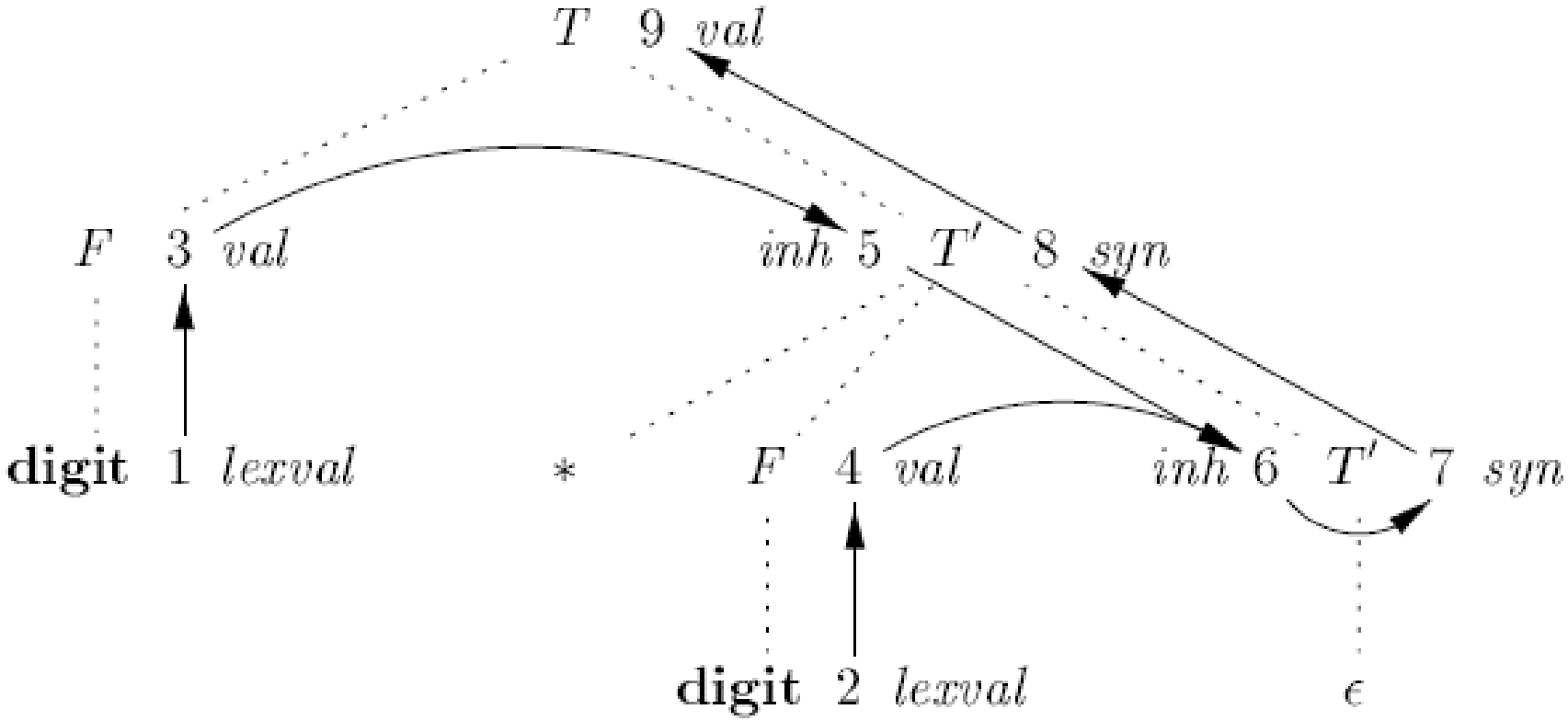
Continued

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$



$E.val$ is synthesized from $E_1.val$ and $T.val$

Continued…



Dependency graph for the annotated parse tree

Ordering the evaluation of attributes

- Topological sorting
- SDD : S-attributed

SDT

DR PARKAVI A

S-Attributed definitions

- SDD is S-attributed

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

Contd…

```
postorder(N) {
        for ( each child C of N, from the left ) postorder(C);
        evaluate the attributes associated with node N;
}
```

L-attributed definition

1. Synthesized, or

2. Inherited, but with the rules limited as follows. Suppose that there is a production $A \rightarrow X_1 X_2 \cdots X_n$, and that there is an inherited attribute $X_i.a$ computed by a rule associated with this production. Then the rule may use only:

   (a) Inherited attributes associated with the head A.

   (b) Either inherited or synthesized attributes associated with the occurrences of symbols $X_1, X_2, \ldots, X_{i-1}$ located to the left of X_i.

Contd..

(c) Inherited or synthesized attributes associated with this occurrence of X_i itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i.

PRODUCTION	SEMANTIC RULE
$T \rightarrow F\,T'$	$T'.inh = F.val$
$T' \rightarrow *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$

SDD

Example Any SDD containing the following production and rules cannot be L-attributed:

PRODUCTION	SEMANTIC RULES
$A \rightarrow B\ C$	$A.s = B.b;$
	$B.i = f(C.c, A.s)$

Semantic rules with controlled side effects

- Permit incidental side effects that do not constrain attribute evaluation. In other words, permit side effects when attribute evaluation based on any topological sort of the dependency graph produces a "correct" translation, where "correct" depends on the application.
- Constrain the allowable evaluation orders, so that the same translation is produced for any allowable order. The constraints can be thought of as implicit edges added to the dependency graph.

Contd….

	PRODUCTION	SEMANTIC RULE
1)	$L \rightarrow E$ **n**	$print(E.val)$

	PRODUCTION	SEMANTIC RULES
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow \mathbf{int}$	$T.type = \text{integer}$
3)	$T \rightarrow \mathbf{float}$	$T.type = \text{float}$
4)	$L \rightarrow L_1\ ,\ \mathbf{id}$	$L_1.inh = L.inh$ $addType(\mathbf{id}.entry, L.inh)$
5)	$L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry, L.inh)$

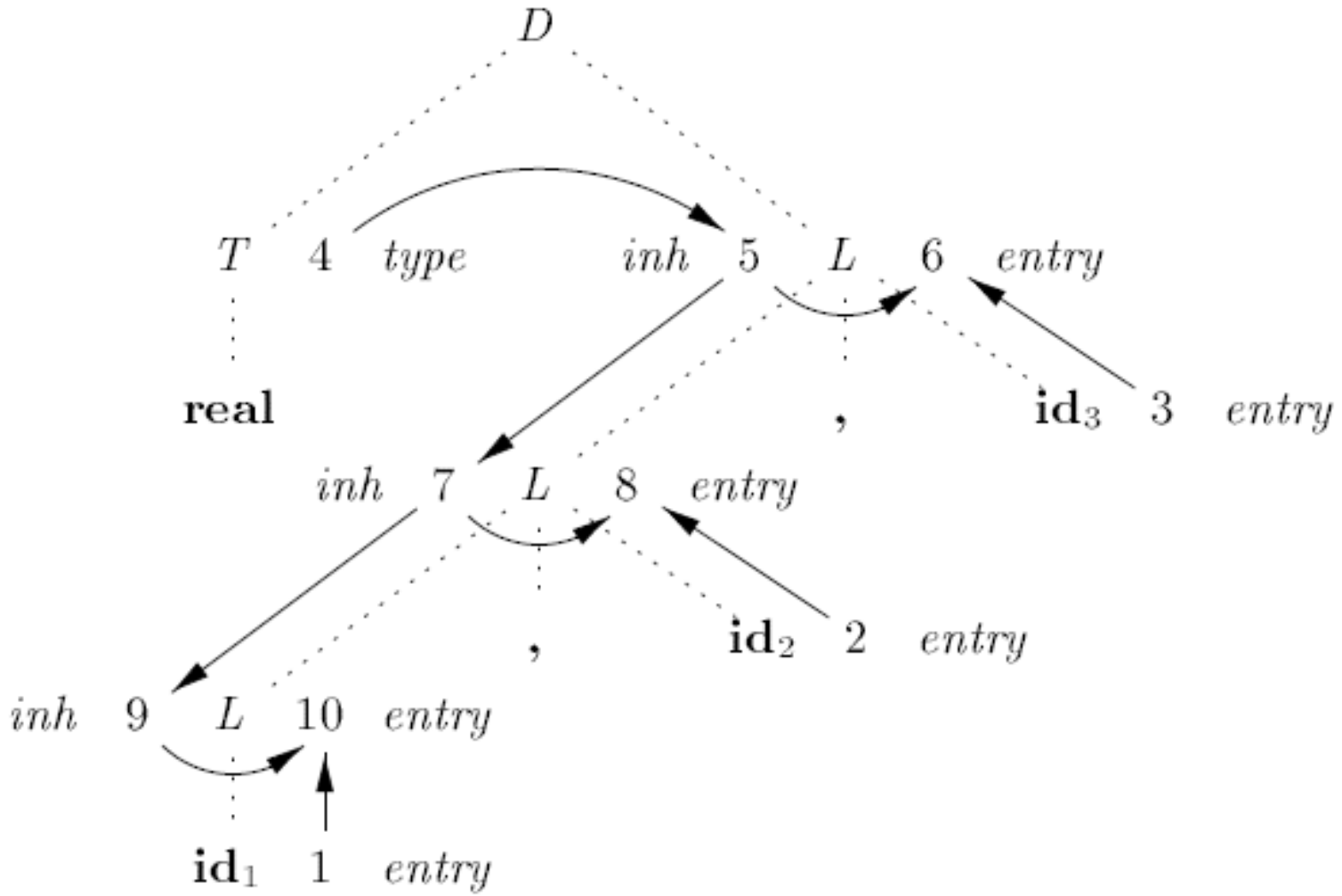
Syntax-directed definition for simple type declarations

Contd..

Productions 4 and 5 also have a rule in which a function *addType* is called with two arguments:

1. **id**.*entry*, a lexical value that points to a symbol-table object, and

2. *L.inh*, the type being assigned to every identifier on the list.



Dependency graph for a declaration **float** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$

Problem to Solve : H/W Tutorial

- Draw dependency graph for the following inputs:

int a,b,c,d

float k,l,m,n

Applications of SDT

We shall implement the nodes of a syntax tree by objects with a suitable number of fields. Each object will have an *op* field that is the label of the node. The objects will have additional fields as follows:

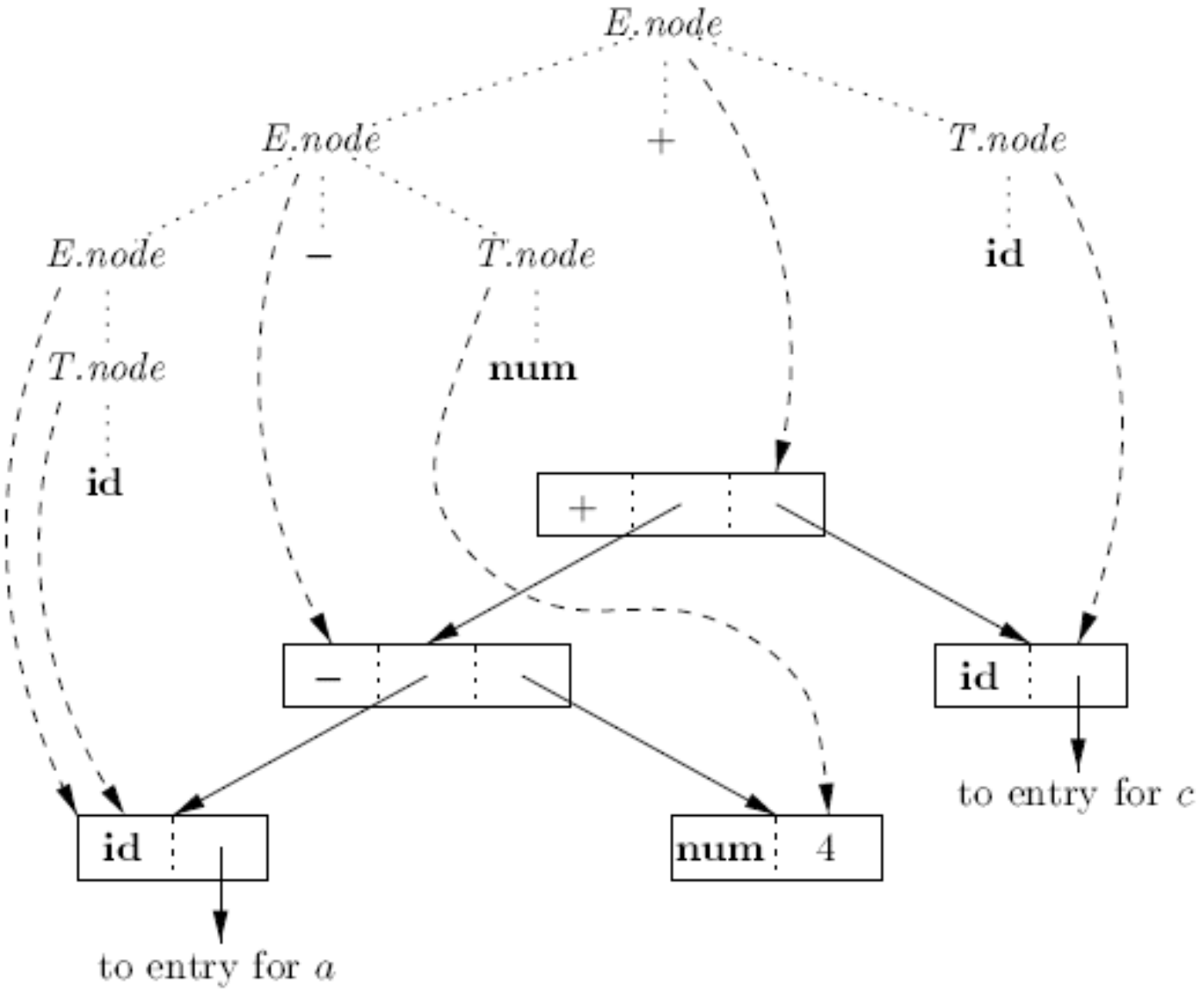
- If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function $Leaf(op, val)$ creates a leaf object. Alternatively, if nodes are viewed as records, then *Leaf* returns a pointer to a new record for a leaf.

- If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function *Node* takes two or more arguments: $Node(op, c_1, c_2, \ldots, c_k)$ creates an object with first field *op* and k additional fields for the k children $c_1, \ldots, c_k$.

Contd..

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Constructing syntax trees for simple expressions



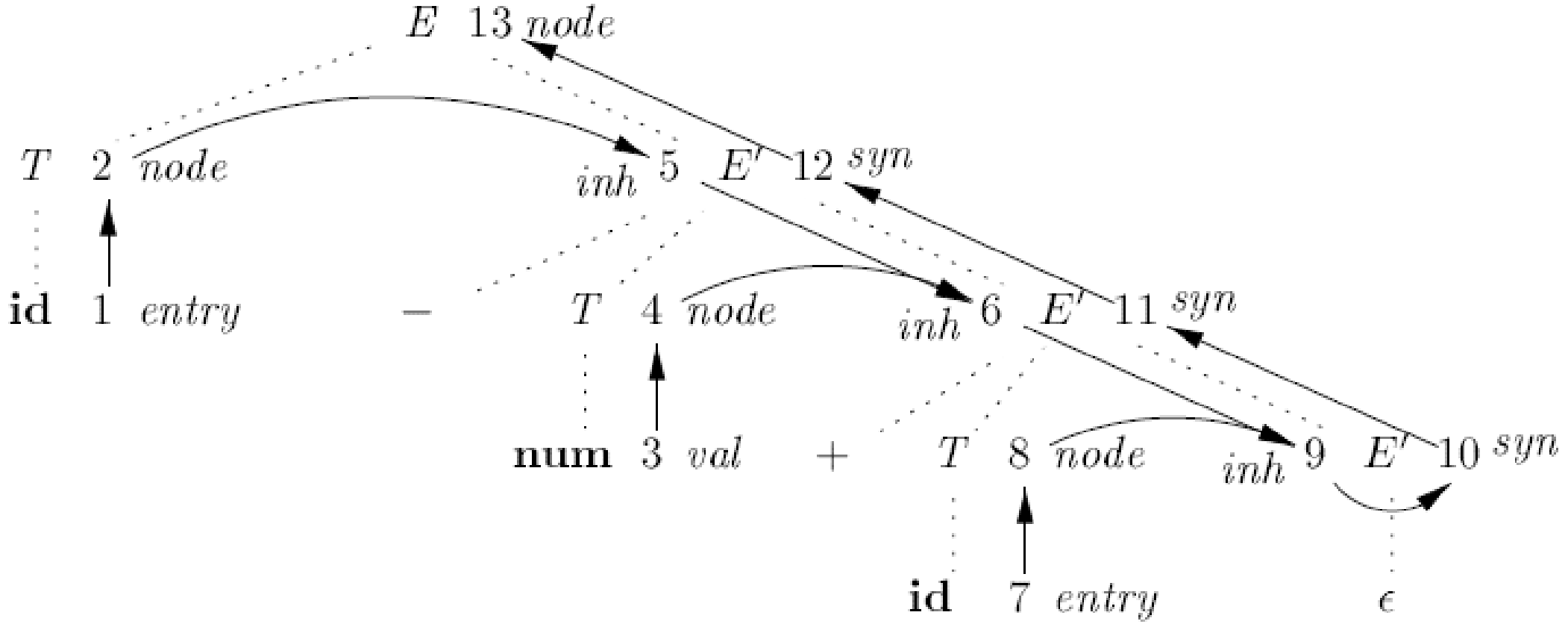
Syntax tree for $a - 4 + c$

1) $p_1 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry}\text{-}a);$
2) $p_2 = \textbf{new}\ \mathit{Leaf}(\textbf{num}, 4);$
3) $p_3 = \textbf{new}\ \mathit{Node}('-', p_1, p_2);$
4) $p_4 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry}\text{-}c);$
5) $p_5 = \textbf{new}\ \mathit{Node}('+', p_3, p_4);$

Steps in the construction of the syntax tree for $a - 4 + c$

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2)	$E' \rightarrow +\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \rightarrow -\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
5)	$T \rightarrow (\ E\ )$	$T.node = E.node$
6)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
7)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Constructing syntax trees during top-down parsing

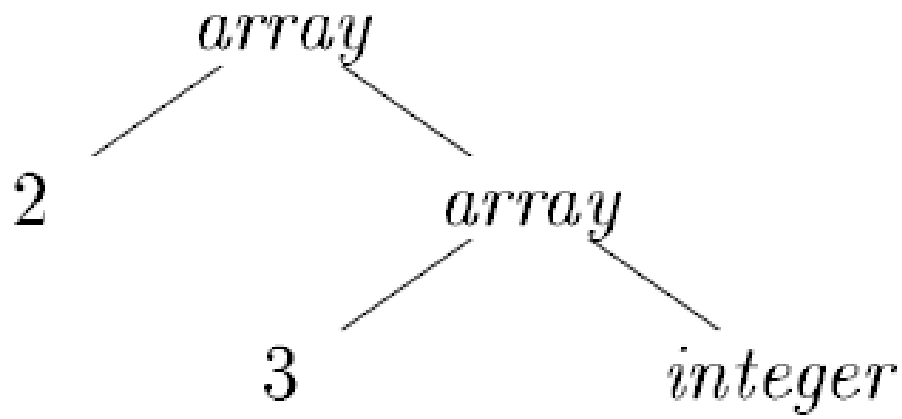


Dependency graph for $a - 4 + c$, with the SDD

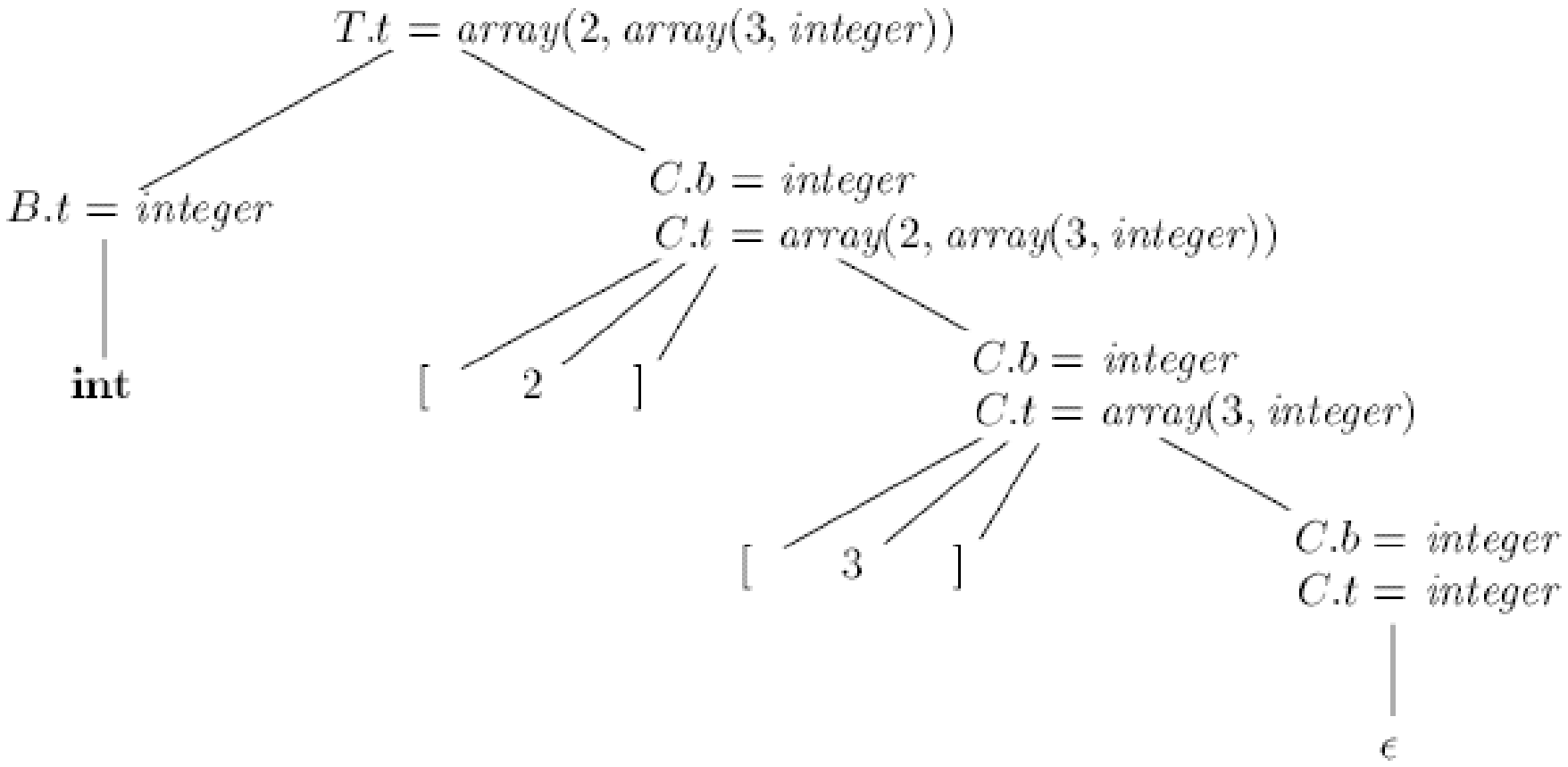
Structure of a type

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow \mathbf{int}$	$B.t = integer$
$B \rightarrow \mathbf{float}$	$B.t = float$
$C \rightarrow [\ \mathbf{num}\ ]\ C_1$	$C.t = array\,(\mathbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

T generates either a basic type or an array type



Type expression for **int**[2][3]



Syntax-directed translation of array types

Tutorial

! Exercise 4.7.5 : Show that the following grammar

$$
\begin{array}{lcl}
S & \rightarrow & A\ a \mid b\ A\ c \mid B\ c \mid b\ B\ a \\
A & \rightarrow & d \\
B & \rightarrow & d
\end{array}
$$

is LR(1) but not LALR(1).

SDT

DR PARKAVI A

SDT schemes

1. The underlying grammar is LR-parsable, and the SDD is S-attributed.
2. The underlying grammar is LL-parsable, and the SDD is L-attributed.

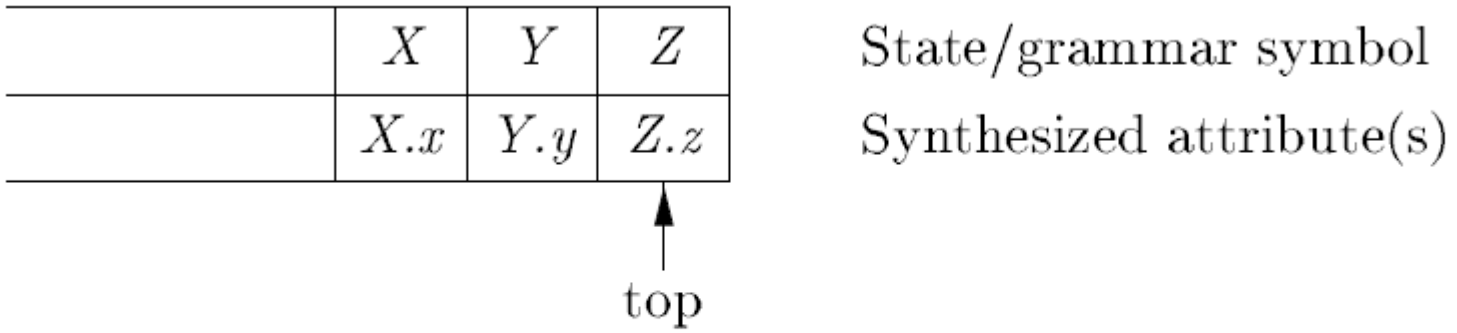
Postfix translation schemes

- SDD : S- attributed
- Action placed at end of production rule
- Post fix SDD's

$$
\begin{array}{lcll}
L & \rightarrow & E\ \mathbf{n} & \{\ \mathrm{print}(E.val);\ \} \\
E & \rightarrow & E_1 + T & \{\ E.val = E_1.val + T.val;\ \} \\
E & \rightarrow & T & \{\ E.val = T.val;\ \} \\
T & \rightarrow & T_1 * F & \{\ T.val = T_1.val \times F.val;\ \} \\
T & \rightarrow & F & \{\ T.val = F.val;\ \} \\
F & \rightarrow & (\ E\ ) & \{\ F.val = E.val;\ \} \\
F & \rightarrow & \mathbf{digit} & \{\ F.val = \mathbf{digit}.lexval;\ \}
\end{array}
$$

Postfix SDT implementing the desk calculator

Parser stack implementation of postfix SDT's

- A→XYZ
- X.x is attribute
- 

Parser stack with a field for synthesized attributes

PRODUCTION	ACTIONS
$L \to E\ \mathbf{n}$	{ print($stack[top-1].val$); $top = top - 1$; }
$E \to E_1 + T$	{ $stack[top-2].val = stack[top-2].val + stack[top].val$; $top = top - 2$; }
$E \to T$	
$T \to T_1 * F$	{ $stack[top-2].val = stack[top-2].val \times stack[top].val$; $top = top - 2$; }
$T \to F$	
$F \to (\ E\ )$	{ $stack[top-2].val = stack[top-1].val$; $top = top - 2$; }
$F \to \mathbf{digit}$	

Implementing the desk calculator on a bottom-up parsing stack

SDT's with actions inside productions

- If the parse is bottom-up, then we perform action a as soon as this occurrence of X appears on the top of the parsing stack.
- If the parse is top-down, we perform a just before we attempt to expand this occurrence of Y (if Y a nonterminal) or check for Y on the input (if Y is a terminal).

Contd..

1)	L	$\rightarrow$	E **n**
2)	E	$\rightarrow$	{ print('+'); } $E_1 + T$
3)	E	$\rightarrow$	T
4)	T	$\rightarrow$	{ print('*'); } $T_1 * F$
5)	T	$\rightarrow$	F
6)	F	$\rightarrow$	(E)
7)	F	$\rightarrow$	**digit** { print(**digit**.*lexval*); }

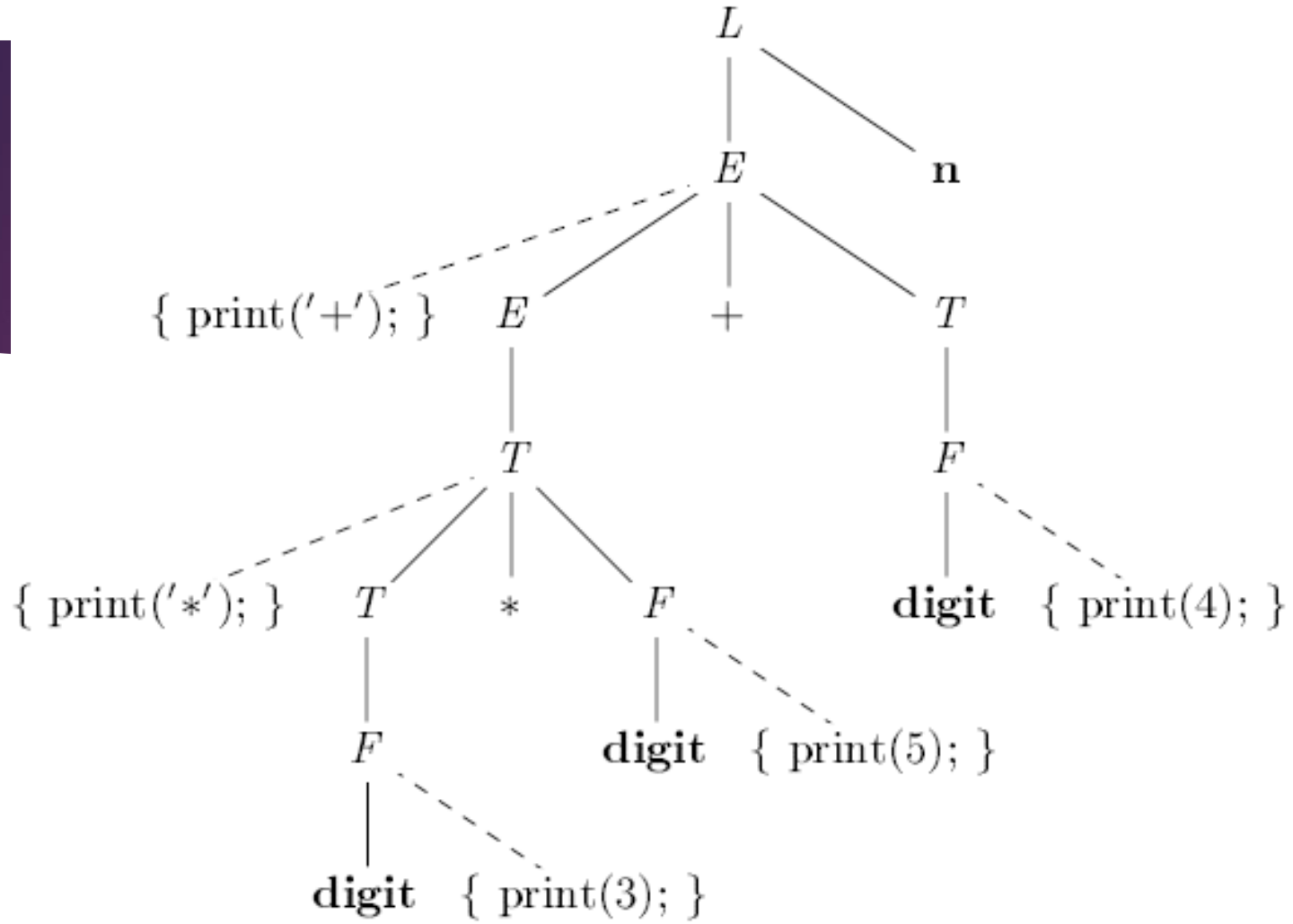
Problematic SDT for infix-to-prefix translation during parsing

Contd..

Any SDT can be implemented as follows:

1. Ignoring the actions, parse the input and produce a parse tree as a result.
2. Then, examine each interior node N, say one for production $A \to \alpha$. Add additional children to N for the actions in α, so the children of N from left to right have exactly the symbols and actions of α.
3. Perform a preorder traversal of the tree, and as soon as a node labeled by an action is visited, perform that action.

Contd..



Parse tree with actions embedded

SDT

DR PARKAVI A

Eliminating left recursion from SDTs

- When transforming the grammar, treat the actions as if they were terminal symbols.

The "trick" for eliminating left recursion is to take two productions

$$A \to A\alpha \mid \beta$$

that generate strings consisting of a β and any number of α's, and replace them by productions that generate the same strings using a new nonterminal R (for "remainder") of the first production:

$$A \to \beta R$$
$$R \to \alpha R \mid \epsilon$$

Contd....

$$
\begin{array}{lcll}
E & \rightarrow & E_1 + T & \{ \text{ print}('+'); \ \} \\
E & \rightarrow & T &
\end{array}
$$

If we apply the standard transformation to E, the remainder of the left-recursive production is

$$\alpha \ = \ + \ T \ \{ \text{ print}('+'); \ \}$$

and β, the body of the other production is T. If we introduce R for the remainder of E, we get the set of productions:

$$
\begin{array}{lcl}
E & \rightarrow & T \ R \\
R & \rightarrow & + \ T \ \{ \text{ print}('+'); \ \} \ R \\
R & \rightarrow & \epsilon
\end{array}
$$

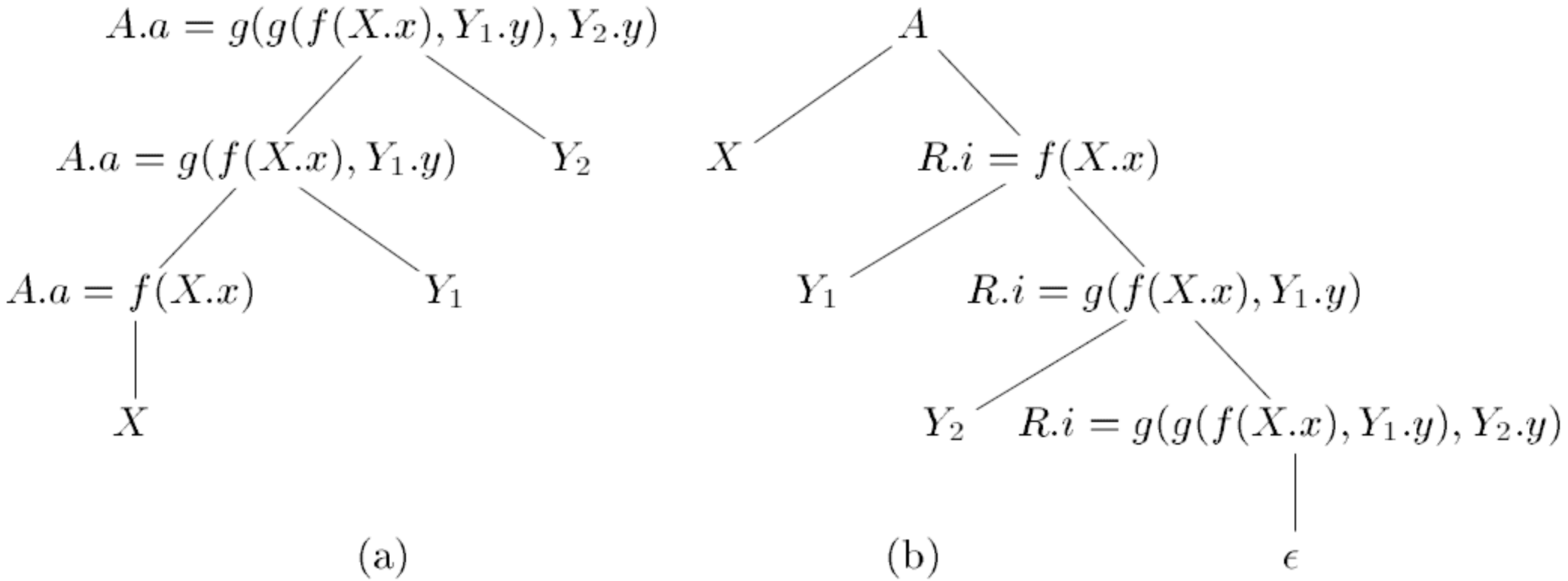
Example
Construct Annotated parse tree for the input : XYY

$$A \rightarrow A_1\ Y\ \{A.a = g(A_1.a, Y.y)\}$$
$$A \rightarrow X\ \{A.a = f(X.x)\}$$

Contd… LRE applied
Construct Annotated parse tree for the input : XYY

$$
\begin{aligned}
A &\rightarrow X\ R \\
R &\rightarrow Y\ R \mid \epsilon
\end{aligned}
$$

Contd..



Eliminating left recursion from a postfix SDT

SDT formation

$$
\begin{array}{lcl}
A & \rightarrow & X \ \{R.i = f(X.x)\} \ R \ \{A.a = R.s\} \\
R & \rightarrow & Y \ \{R_1.i = g(R.i, Y.y)\} \ R_1 \ \{R.s = R_1.s\} \\
R & \rightarrow & \epsilon \ \{R.s = R.i\}
\end{array}
$$

SDT for L-attributed definition

The rules for turning an L-attributed SDD into an SDT are as follows:

1. Embed the action that computes the inherited attributes for a nonterminal A immediately before that occurrence of A in the body of the production. If several inherited attributes for A depend on one another in an acyclic fashion, order the evaluation of attributes so that those needed first are computed first.

2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production.

Example : This example is motivated by languages for typesetting mathematical formulas. `Eqn` is an early example of such a language; ideas from `Eqn` are still found in the TEX typesetting system, which was used to produce this book.

We shall concentrate on only the capability to define subscripts, subscripts of subscripts, and so on, ignoring superscripts, built-up fractions, and all other mathematical features. In the `Eqn` language, one writes `a sub i sub j` to set the expression a_{i_j}. A simple grammar for *boxes* (elements of text bounded by a rectangle) is

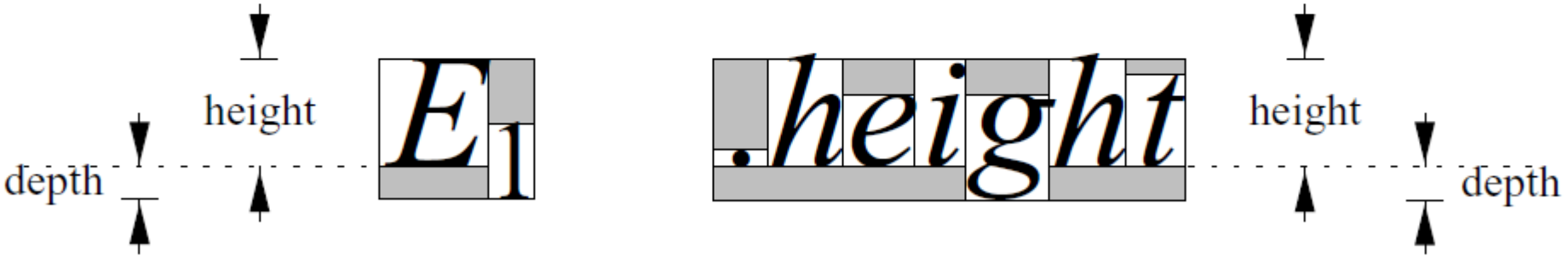
Contd

$$B \rightarrow B_1 \ B_2 \mid B_1 \ \textbf{sub} \ B_2 \mid (\ B_1 \ ) \mid \textbf{text}$$

Corresponding to these four productions, a box can be either

1. Two boxes, juxtaposed, with the first, B_1, to the left of the other, B_2.
2. A box and a subscript box. The second box appears in a smaller size, lower, and to the right of the first box.
3. A parenthesized box, for grouping of boxes and subscripts. Eqn and TEX both use curly braces for grouping, but we shall use ordinary, round parentheses to avoid confusion with the braces that surround actions in SDT's.
4. A text string, that is, any string of characters.

The left box for E_1 is itself constructed from the box for E and the subscript 1. The subscript 1 is handled by shrinking its box by about 30%, lowering it, and placing it after the box for E. Although we shall treat *.height* as a text string, the rectangles within its box show how it can be constructed from boxes for the individual letters.



Constructing larger boxes from smaller ones

Contd....

The values associated with the vertical geometry of boxes are as follows:

a) The *point size* is used to set text within a box. We shall assume that characters not in subscripts are set in 10 point type, the size of type in this book. Further, we assume that if a box has point size p, then its subscript box has the smaller point size $0.7p$. Inherited attribute $B.ps$ will represent the point size of block B. This attribute must be inherited, because the context determines by how much a given box needs to be shrunk, due to the number of levels of subscripting.

Contd..

b) Each box has a *baseline*, which is a vertical position that corresponds to the bottoms of lines of text, not counting any letters, like "g" that extend below the normal baseline. In Fig. 5.24, the dotted line represents the baseline for the boxes E, *.height*, and the entire expression. The baseline for the box containing the subscript 1 is adjusted to lower the subscript.

c) A box has a *height*, which is the distance from the top of the box to the baseline. Synthesized attribute $B.ht$ gives the height of box B.

d) A box has a *depth*, which is the distance from the baseline to the bottom of the box. Synthesized attribute $B.dp$ gives the depth of box B.

	PRODUCTION	SEMANTIC RULES
1)	$S \rightarrow B$	$B.ps = 10$
2)	$B \rightarrow B_1\ B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = \max(B_1.ht, B_2.ht)$ $B.dp = \max(B_1.dp, B_2.dp)$
3)	$B \rightarrow B_1\ \textbf{sub}\ B_2$	$B_1.ps = B.ps$ $B_2.ps = 0.7 \times B.ps$ $B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps)$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps)$
4)	$B \rightarrow (\ B_1\ )$	$B_1.ps = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
5)	$B \rightarrow \textbf{text}$	$B.ht = getHt(B.ps, \textbf{text}.lexval)$ $B.dp = getDp(B.ps, \textbf{text}.lexval)$

SDD for typesetting boxes

SDD to SDT

	PRODUCTION	ACTIONS
1)	$S \rightarrow B$	$\{\ B.ps = 10;\ \}$
2)	$B \rightarrow B_1\ B_2$	$\{\ B_1.ps = B.ps;\ \}$ $\{\ B_2.ps = B.ps;\ \}$ $\{\ B.ht = \max(B_1.ht, B_2.ht);$ $B.dp = \max(B_1.dp, B_2.dp);\ \}$
3)	$B \rightarrow B_1\ \textbf{sub}\ B_2$	$\{\ B_1.ps = B.ps;\ \}$ $\{\ B_2.ps = 0.7 \times B.ps;\ \}$ $\{\ B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps);$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps);\ \}$
4)	$B \rightarrow (\ B_1\ )$	$\{\ B_1.ps = B.ps;\ \}$ $\{\ B.ht = B_1.ht;$ $B.dp = B_1.dp;\ \}$
5)	$B \rightarrow \textbf{text}$	$\{\ B.ht = getHt(B.ps, \textbf{text}.lexval);$ $B.dp = getDp(B.ps, \textbf{text}.lexval);\ \}$

SDT for typesetting boxes

Example

- S→ while (c) S1

Contd

We use the following attributes to generate the proper intermediate code:

1. The inherited attribute $S.next$ labels the beginning of the code that must be executed after S is finished.

2. The synthesized attribute $S.code$ is the sequence of intermediate-code steps that implements a statement S and ends with a jump to $S.next$.

3. The inherited attribute $C.true$ labels the beginning of the code that must be executed if C is true.

4. The inherited attribute $C.false$ labels the beginning of the code that must be executed if C is false.

5. The synthesized attribute $C.code$ is the sequence of intermediate-code steps that implements the condition C and jumps either to $C.true$ or to $C.false$, depending on whether C is true or false.

Contd..

- The function *new* generates new labels.
- The variables $L1$ and $L2$ hold labels that we need in the code. $L1$ is the beginning of the code for the while-statement, and we need to arrange

  that S_1 jumps there after it finishes. That is why we set $S_1.next$ to $L1$. $L2$ is the beginning of the code for S_1, and it becomes the value of $C.true$, because we branch there when C is true.
- Notice that $C.false$ is set to $S.next$, because when the condition is false, we execute whatever code must follow the code for S.
- We use || as the symbol for concatenation of intermediate-code fragments. The value of $S.code$ thus begins with the label $L1$, then the code for condition C, another label $L2$, and the code for S_1.

$S \rightarrow \textbf{while}\ (\ C\ )\ S_1$	$L1 = new();$ $L2 = new();$ $S_1.next = L1;$ $C.false = S.next;$ $C.true = L2;$ $S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code$

SDD for while-statements

Tutorial Problem
Construct SDT for the following grammars

a) $S \rightarrow \textbf{if}\ (\ C\ )\ S_1\ \textbf{else}\ S_2$

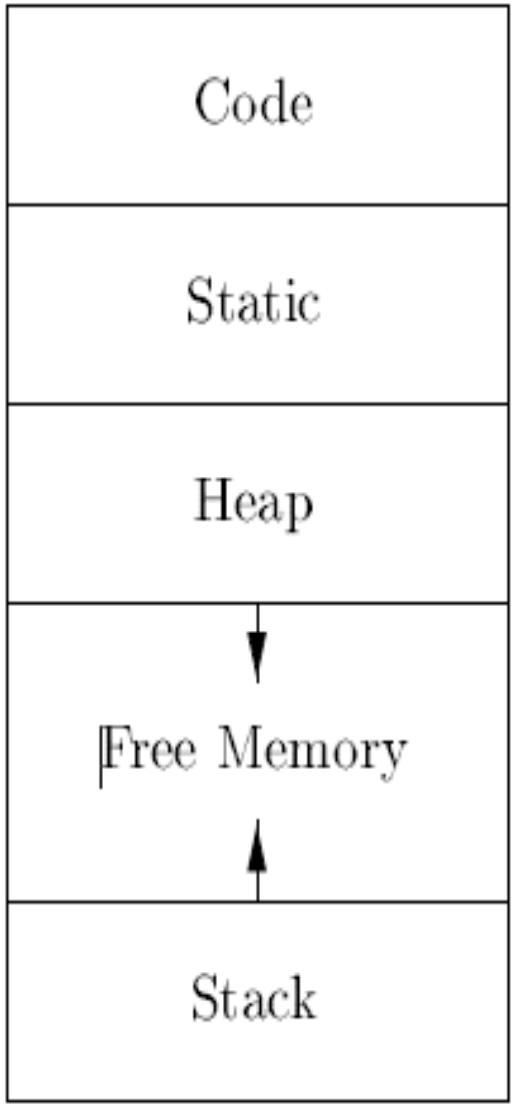
b) $S \rightarrow \textbf{do}\ S_1\ \textbf{while}\ (\ C\ )$

Run time Environment

Dr Parkavi A

Storage Organization

- Logical address space
- Physical address space



Typical subdivision of run-time memory into code and data areas

Contd..

- Alignment of bytes: address divisible by 4
- Padding
- Area : Code
- Static area
- Dynamic
  - Stack area
  - Heap area

Contd..

an activation record is used to store information about the status of the machine, such as the value of the program counter and machine registers, when a procedure call occurs. When control returns from the call, the activation of the calling procedure can be restarted after restoring the values of relevant registers and setting the program counter to the point immediately after the call. Data objects whose lifetimes are contained in that of an activation can be allocated on the stack along with other information associated with the activation.

Static vs Dynamic Storage allocation

- Compile time and runt time

Many compilers use some combination of the following two strategies for dynamic storage allocation:

1. *Stack storage.* Names local to a procedure are allocated space on a stack. The stack supports the normal call/return policy for procedures.

2. *Heap storage.* Data that may outlive the call to the procedure that created it is usually allocated on a "heap" of reusable storage.

Contd..

- Garbage collection
- Stack allocation of space
  - Procedures
  - Local variables
- Activation trees
  - Activation of procedures
  - Stack allocation

Contd..

In this example, as is true in general, procedure activations are nested in time. If an activation of procedure p calls procedure q, then that activation of q must end before the activation of p can end. There are three common cases:

1. The activation of q terminates normally. Then in essentially any language, control resumes just after the point of p at which the call to q was made.

2. The activation of q, or some procedure q called, either directly or indirectly, aborts; i.e., it becomes impossible for execution to continue. In that case, p ends simultaneously with q.

Contd..

3. The activation of q terminates because of an exception that q cannot handle. Procedure p may handle the exception, in which case the activation of q has terminated while the activation of p continues, although not necessarily from the point at which the call to q was made. If p cannot handle the exception, then this activation of p terminates at the same time as the activation of q, and presumably the exception will be handled by some other open activation of a procedure.

Contd..

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
```

Contd…

```
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Sketch of a quicksort program

Contd..

The use of a run-time stack is enabled by several useful relationships between the activation tree and the behavior of the program:

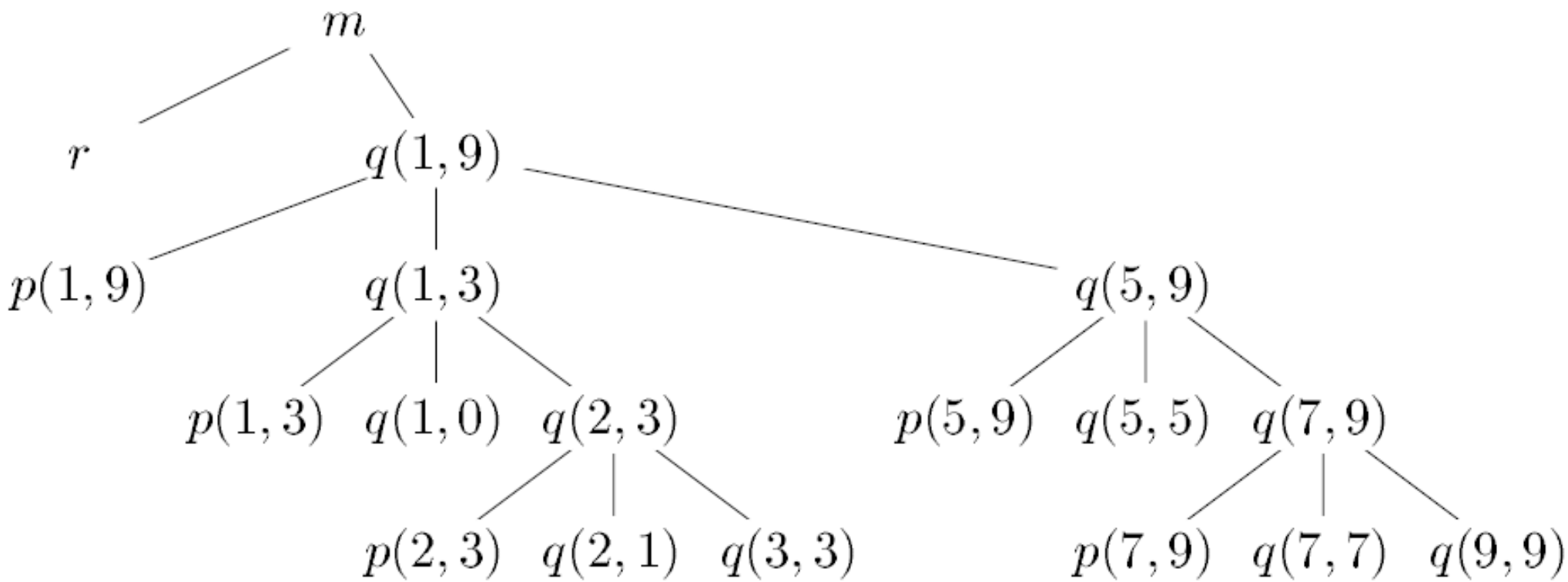
1. The sequence of procedure calls corresponds to a preorder traversal of the activation tree.

2. The sequence of returns corresponds to a postorder traversal of the activation tree.

3. Suppose that control lies within a particular activation of some procedure, corresponding to a node N of the activation tree. Then the activations that are currently open (*live*) are those that correspond to node N and its ancestors. The order in which these activations were called is the order in which they appear along the path to N, starting at the root, and they will return in the reverse of that order.

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Possible activations for the program

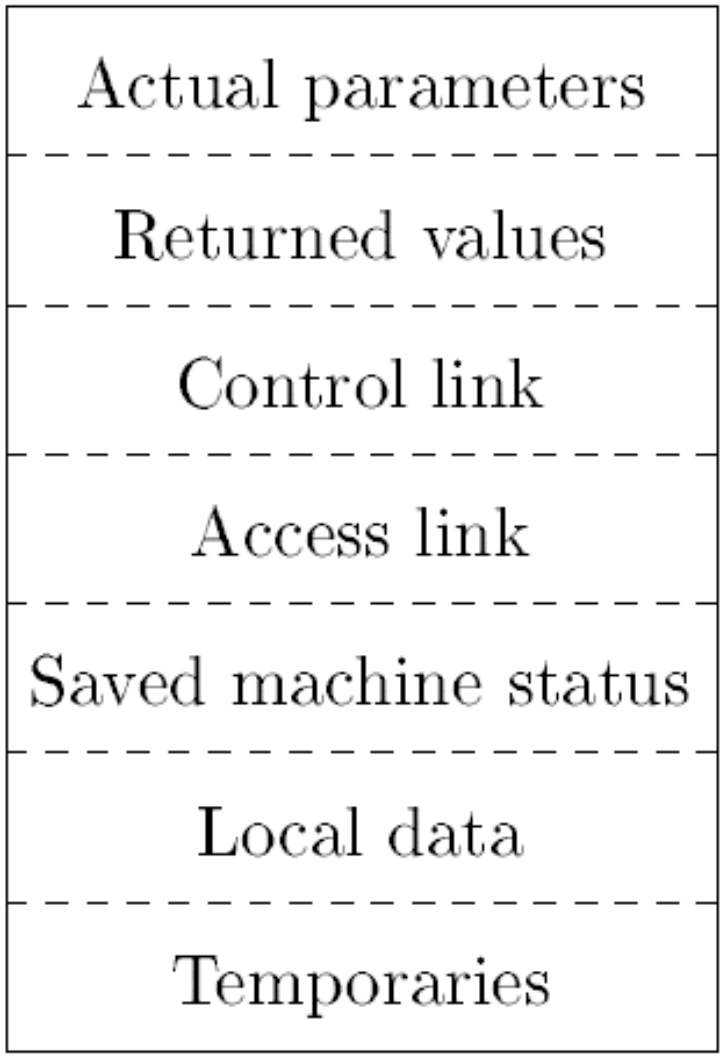


Activation tree representing calls during an execution of *quicksort*

Activation record

- Procedure calls and returns
- Control stack
- Activation record will be stored in control stack

Contd…



A general activation record

1. Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.

2. Local data belonging to the procedure whose activation record this is.

3. A saved machine status, with information about the state of the machine just before the call to the procedure. This information typically includes the *return address* (value of the program counter, to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.

4. An "access link" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.

5. A *control link*, pointing to the activation record of the caller.

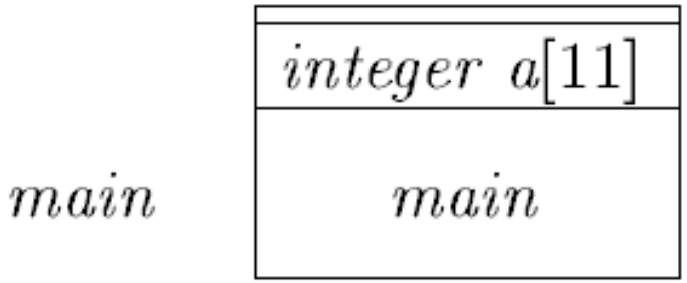
6. Space for the return value of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

Contd..

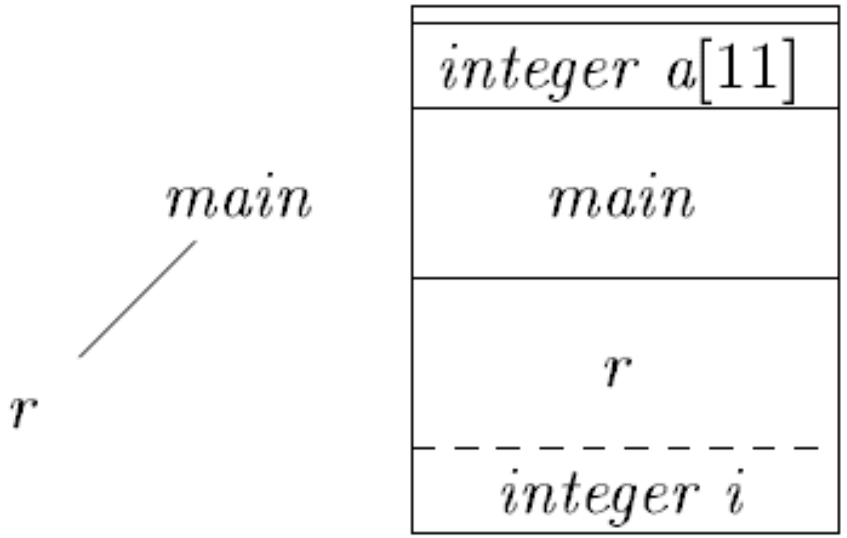
6. Space for the return value of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

7. The actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency. However, we show a space for them to be completely general.

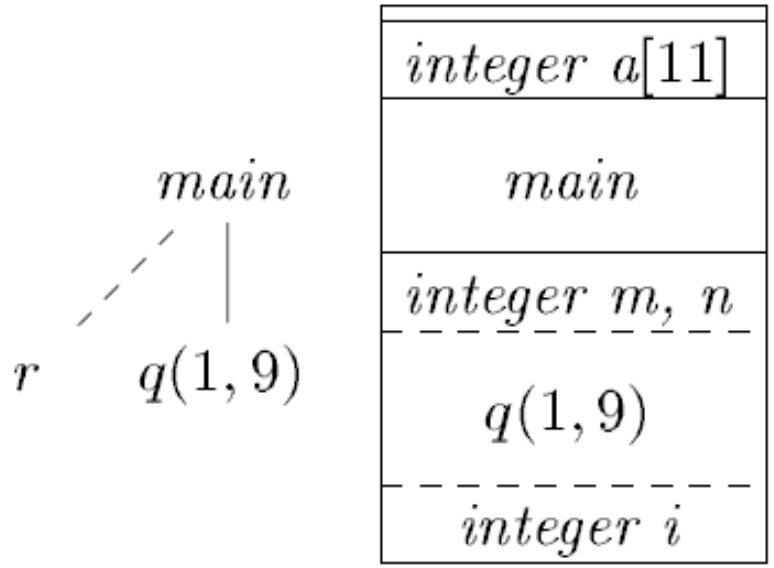
Contd..



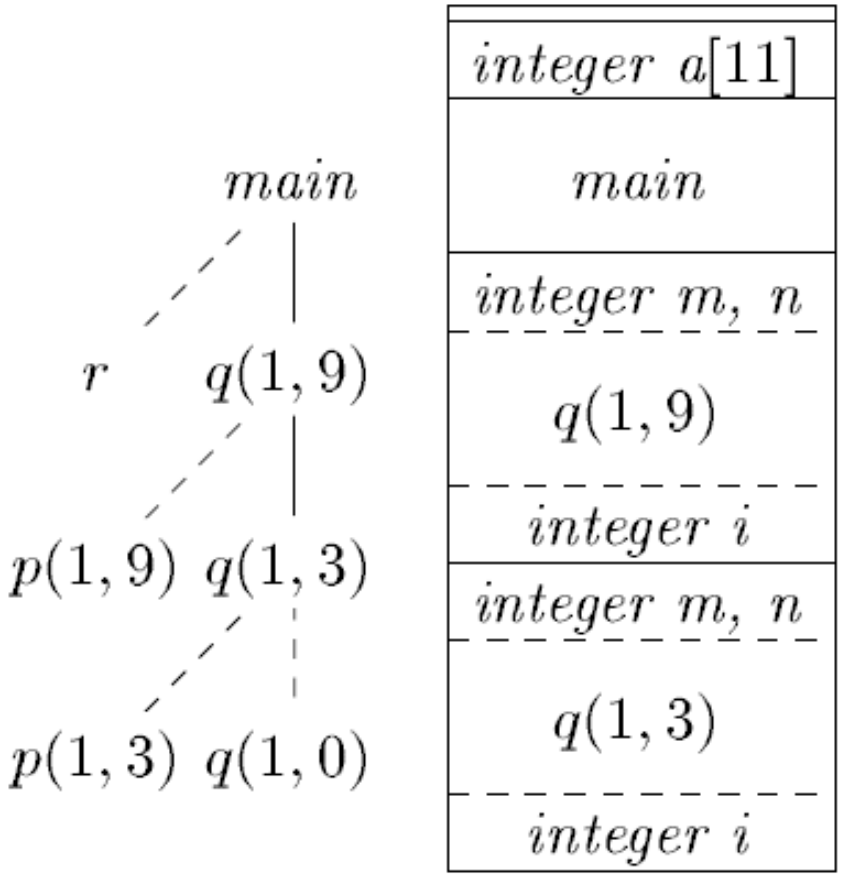
(a) Frame for *main*



(b) r is activated



(c) r has been popped and $q(1,9)$ pushed



(d) Control returns to $q(1,3)$

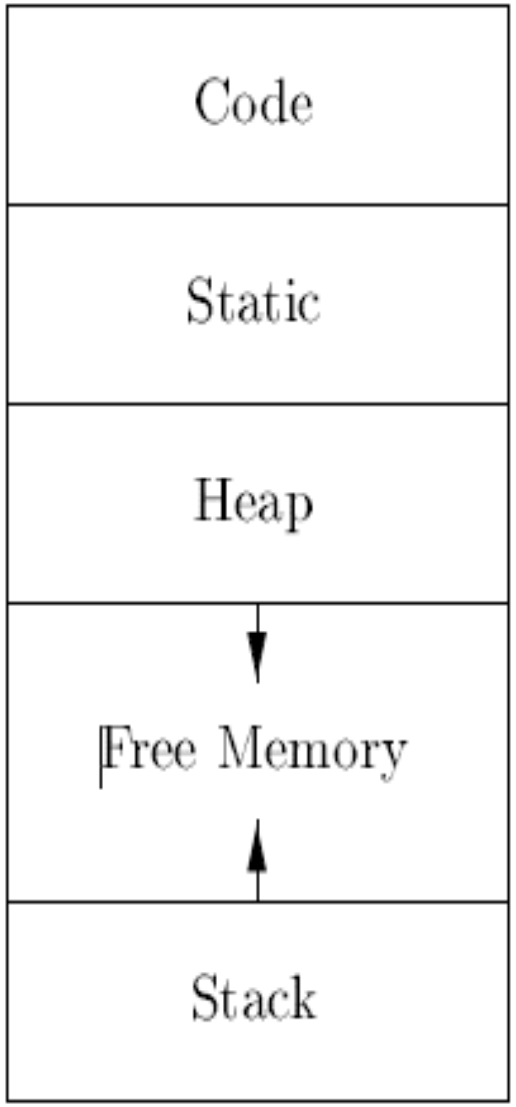
Downward-growing stack of activation records

Run time Environment

DR PARKAVI A

Storage Organization

- Logical address space
- Physical address space



Typical subdivision of run-time memory into code and data areas

Contd..

- Alignment of bytes: address divisible by 4
- Padding
- Area : Code
- Static area
- Dynamic
  - Stack area
  - Heap area

Contd..

an activation record is used to store information about the status of the machine, such as the value of the program counter and machine registers, when a procedure call occurs. When control returns from the call, the activation of the calling procedure can be restarted after restoring the values of relevant registers and setting the program counter to the point immediately after the call. Data objects whose lifetimes are contained in that of an activation can be allocated on the stack along with other information associated with the activation.

Static vs Dynamic Storage allocation

Many compilers use some combination of the following two strategies for dynamic storage allocation:

1. *Stack storage.* Names local to a procedure are allocated space on a stack. The stack supports the normal call/return policy for procedures.

2. *Heap storage.* Data that may outlive the call to the procedure that created it is usually allocated on a "heap" of reusable storage.

Contd..

- Garbage collection
- Stack allocation of space
  - Procedures
  - Local variables
- Activation trees
  - Activation of procedures
  - Stack allocation

Contd..

In this example, as is true in general, procedure activations are nested in time. If an activation of procedure p calls procedure q, then that activation of q must end before the activation of p can end. There are three common cases:

1. The activation of q terminates normally. Then in essentially any language, control resumes just after the point of p at which the call to q was made.

2. The activation of q, or some procedure q called, either directly or indirectly, aborts; i.e., it becomes impossible for execution to continue. In that case, p ends simultaneously with q.

Contd..

3. The activation of q terminates because of an exception that q cannot handle. Procedure p may handle the exception, in which case the activation of q has terminated while the activation of p continues, although not necessarily from the point at which the call to q was made. If p cannot handle the exception, then this activation of p terminates at the same time as the activation of q, and presumably the exception will be handled by some other open activation of a procedure.

Contd..

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
```

Contd...

```
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Sketch of a quicksort program

Contd..

The use of a run-time stack is enabled by several useful relationships between the activation tree and the behavior of the program:

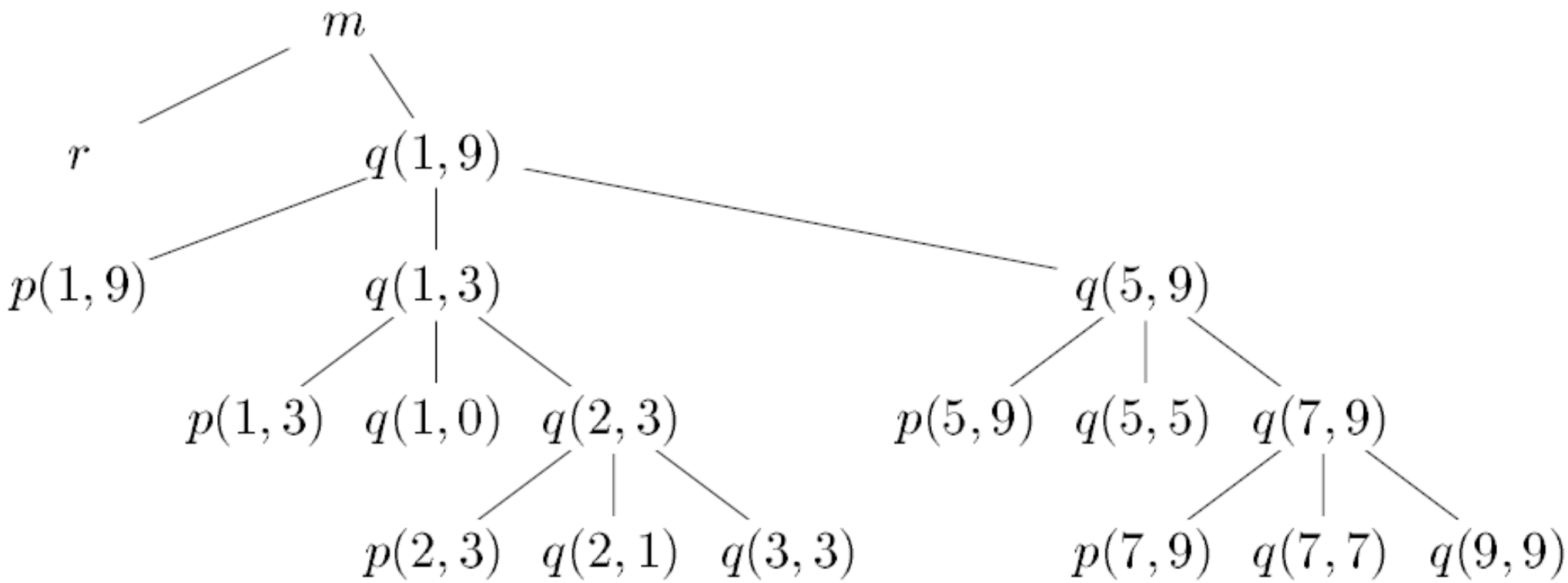
1. The sequence of procedure calls corresponds to a preorder traversal of the activation tree.

2. The sequence of returns corresponds to a postorder traversal of the activation tree.

3. Suppose that control lies within a particular activation of some procedure, corresponding to a node N of the activation tree. Then the activations that are currently open (*live*) are those that correspond to node N and its ancestors. The order in which these activations were called is the order in which they appear along the path to N, starting at the root, and they will return in the reverse of that order.

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Possible activations for the program

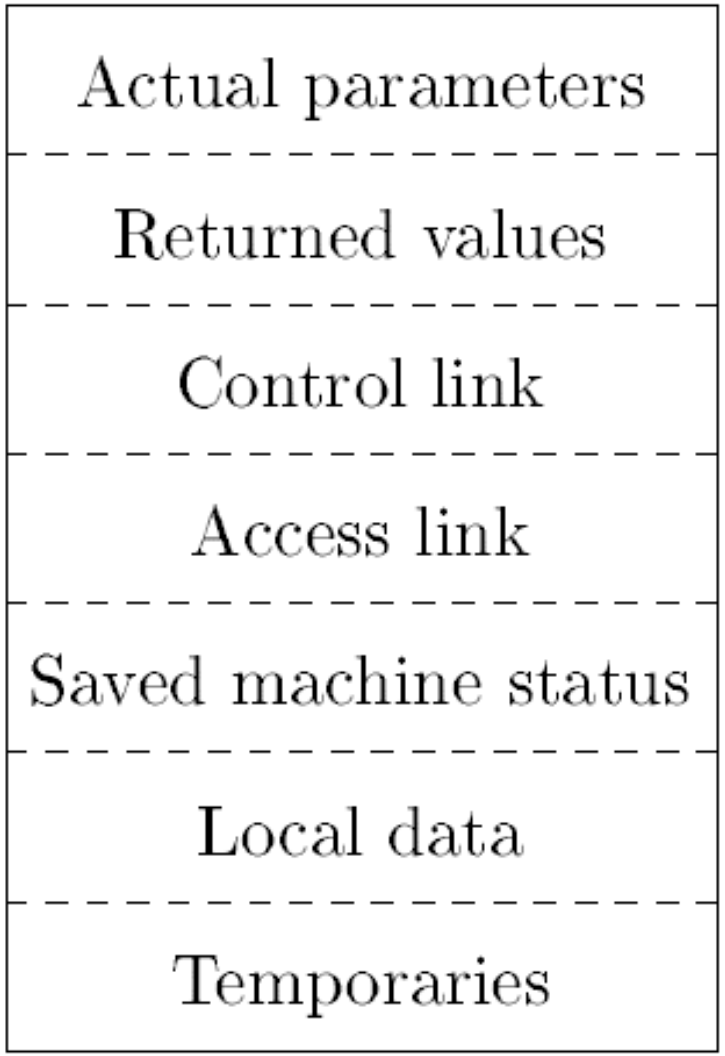


Activation tree representing calls during an execution of *quicksort*

Activation record

- Procedure calls and returns
- Control stack
- Activation record will be stored in control stack

Contd...



A general activation record

1. Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.

2. Local data belonging to the procedure whose activation record this is.

3. A saved machine status, with information about the state of the machine just before the call to the procedure. This information typically includes the *return address* (value of the program counter, to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.

4. An "access link" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.

5. A *control link*, pointing to the activation record of the caller.

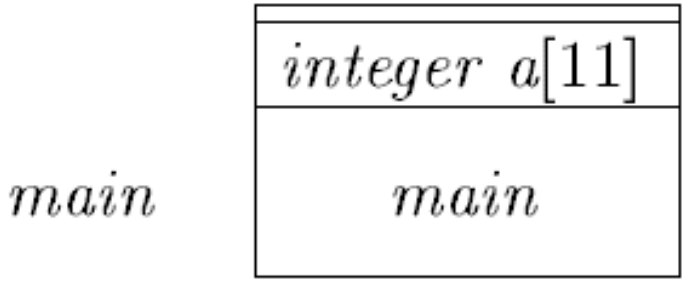
6. Space for the return value of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

Contd..

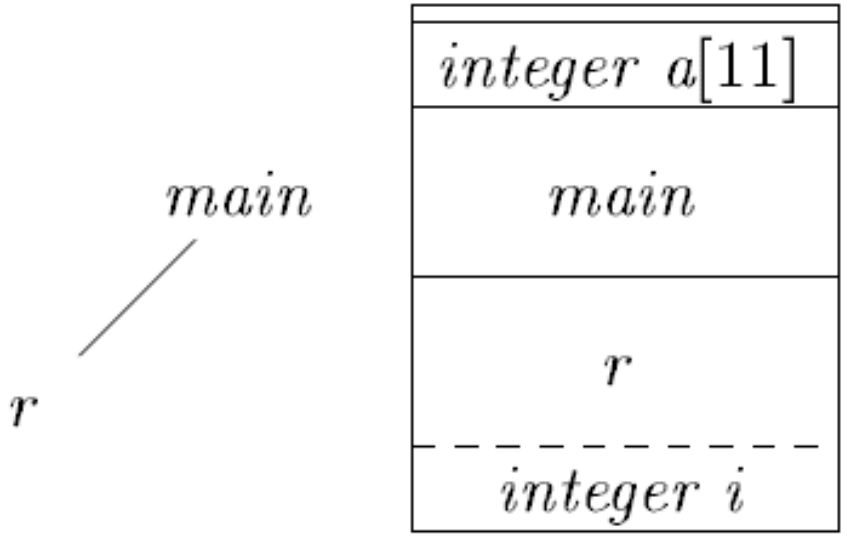
6. Space for the return value of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

7. The actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency. However, we show a space for them to be completely general.

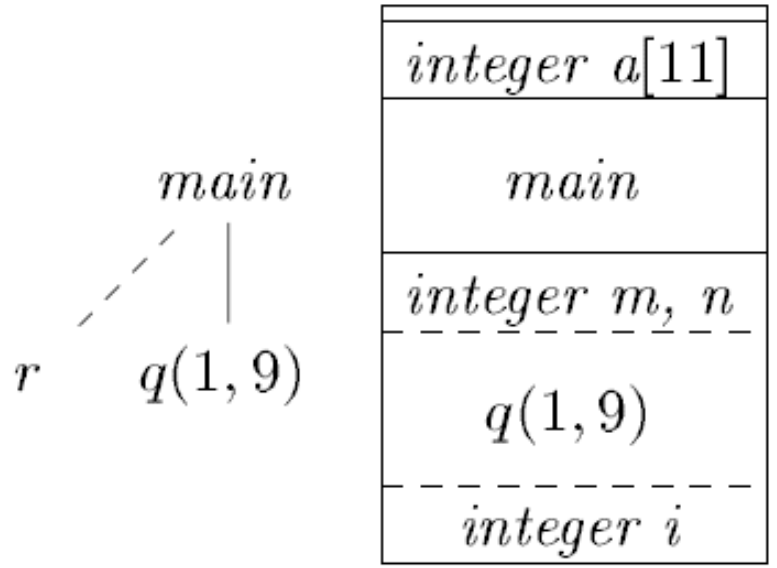
Contd..



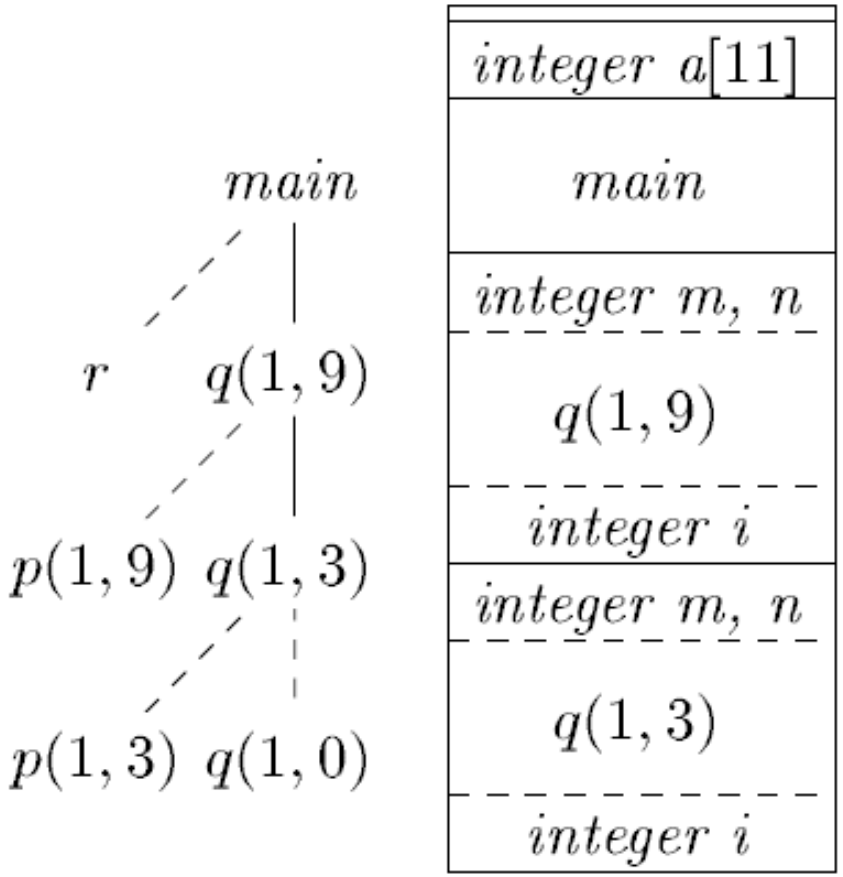
(a) Frame for *main*



(b) r is activated



(c) r has been popped and $q(1,9)$ pushed



(d) Control returns to $q(1,3)$

Downward-growing stack of activation records

Run time environment

DR PARKAVI A

Run time environment

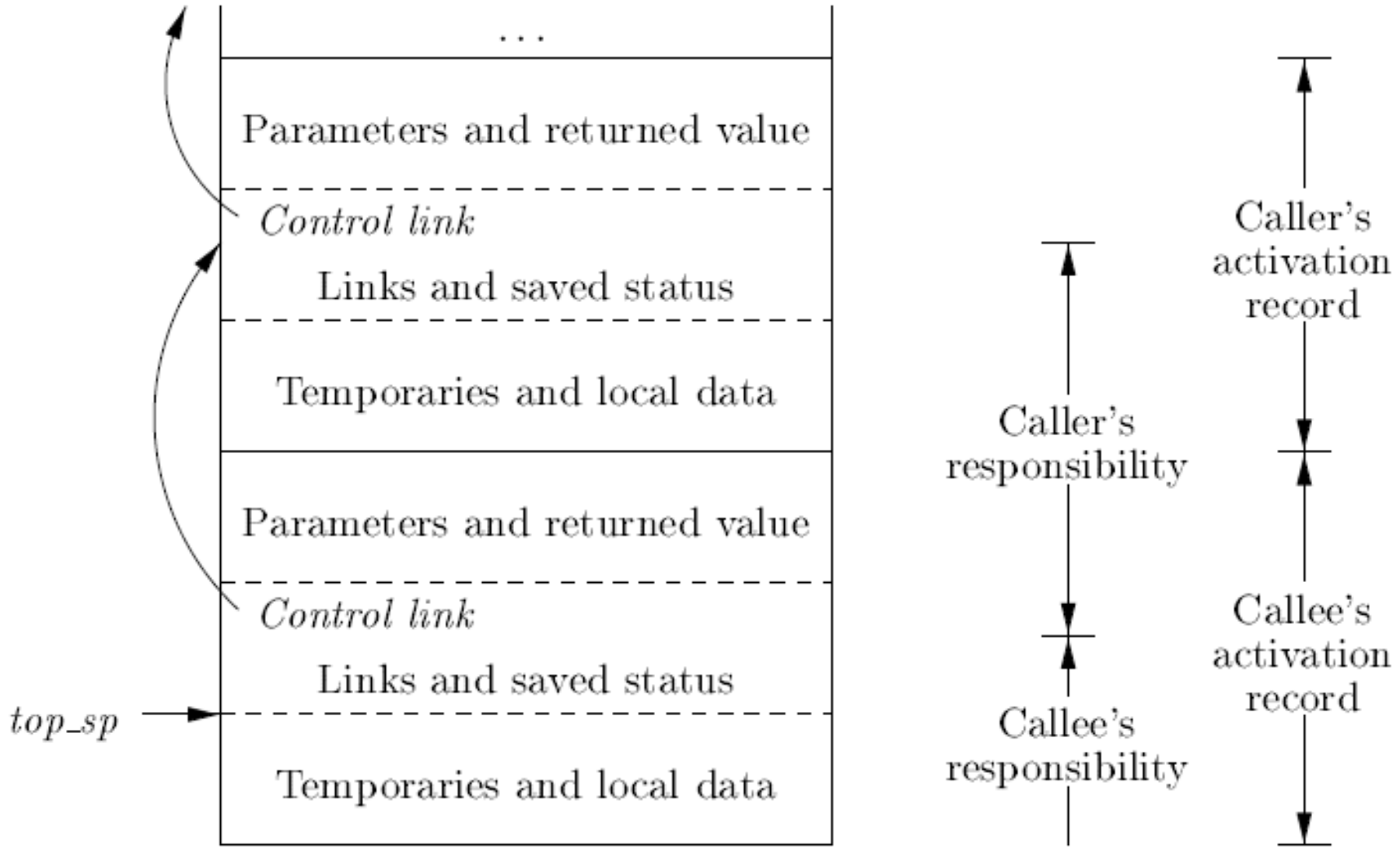
- Calling sequence
- Return sequence
- Caller
- Callee

Contd....

1. Values communicated between caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.

2. Fixed-length items are generally placed in the middle.
items typically include the control link, the access link, and the machine status fields. If exactly the same components of the machine status are saved for each call, then the same code can do the saving and restoring for each.

3. Items whose size may not be known early enough are placed at the end of the activation record.



Division of tasks between caller and callee

Contd..

The calling sequence and its division between caller and callee are as follows:

1. The caller evaluates the actual parameters.

2. The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments *top_sp* to the position shown That is, *top_sp* is moved past the caller's local data and temporaries and the callee's parameters and status fields.

3. The callee saves the register values and other status information.

4. The callee initializes its local data and begins execution.

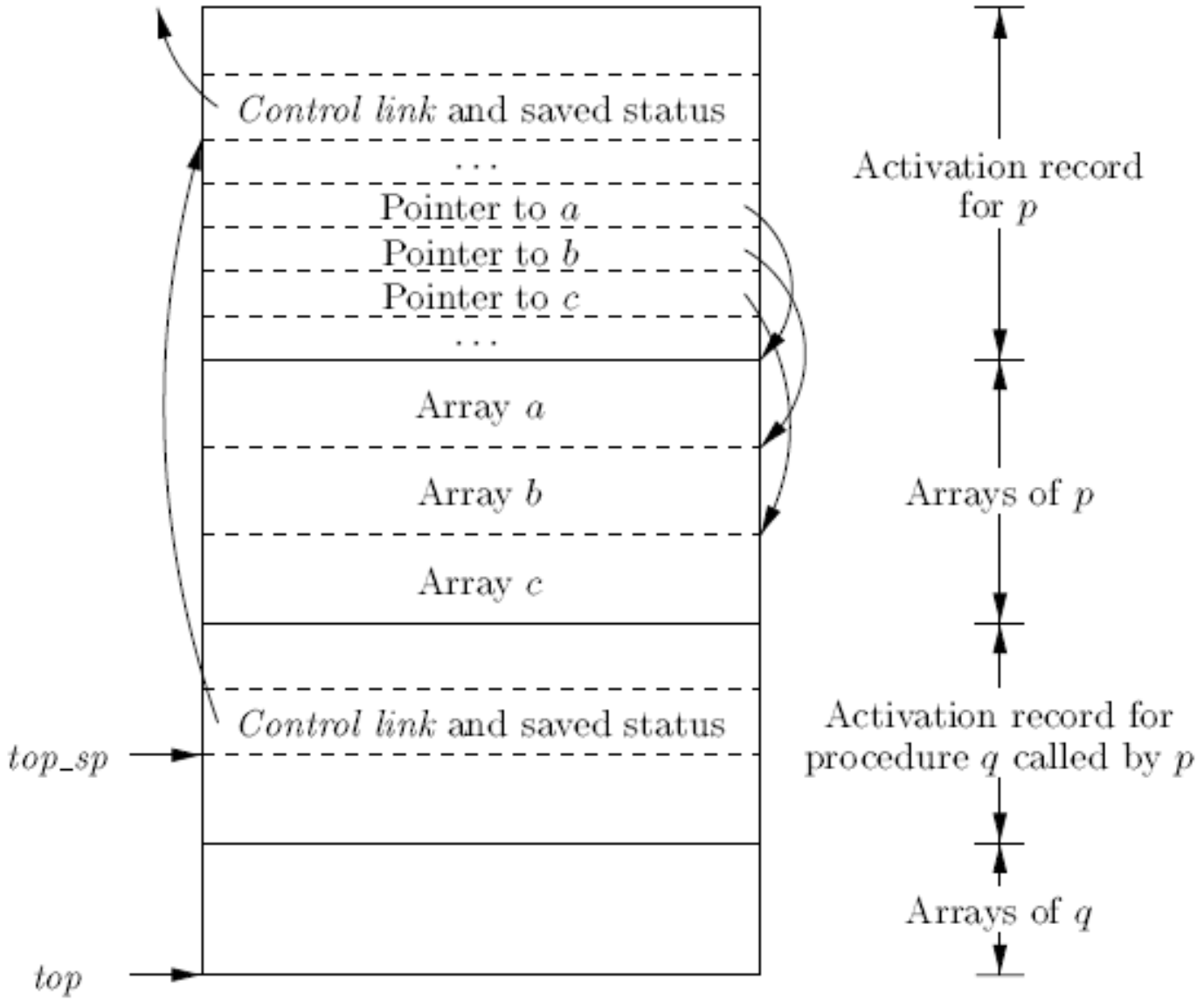
Contd..

A suitable, corresponding return sequence is:

1. The callee places the return value next to the parameters.
2. Using information in the machine-status field, the callee restores *top_sp* and other registers, and then branches to the return address that the caller placed in the status field.
3. Although *top_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top_sp*; the caller therefore may use that value.

Variable length data on the stack

- Run time memory management system
- Allocate space
- Variable length arrays
  - On stack
  - Pointer to beginning of arrays



Access to dynamically allocated arrays

Access to non local data on the stack

For languages that do not allow nested procedure declarations, allocation of storage for variables and access to those variables is simple:

1. Global variables are allocated static storage. The locations of these variables remain fixed and are known at compile time. So to access any variable that is not local to the currently executing procedure, we simply use the statically determined address.

2. Any other name must be local to the activation at the top of the stack. We may access these variables through the *top_sp* pointer of the stack.