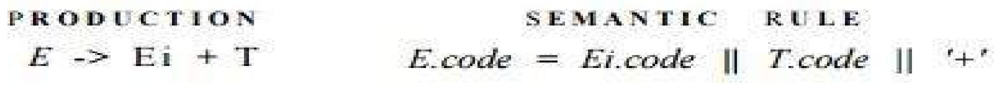
UNIT -5

Syntax Directed Translations

We associate information with a language construct by attaching attributes to the grammar symbol(s)representing the construct, A syntax-directed definition specifies the values of attributes by associatingsemantic rules with the grammar productions. For example, an infix-to-postfix translator might have a production and rule



PRODUCTION	SEMANTIC RULE
E -> Ei + T	$E.code$ = $Ei.code$ || $T.code$ || '+'

This production has two non terminals, E and T; the subscript in E1 distinguishes the occurrence of E inthe production body from the occurrence of E as the head. Both E and T have a string-valued attribute code. The semantic rule specifies that the string E. code is formed by concatenating E i. code, T.code, andthe character '+'. While the rule makes it explicit that the translation of E is built up from the translationsof E1, T, and '+', it may be inefficient to implement the translation directly by manipulating strings

a syntax-directed translation scheme embeds program fragments called semantic actions within production bodies
There are two notations for attaching semantic rules:

1. **Syntax Directed Definitions.** High-level specification hiding many implementation details (also called **Attribute Grammars**).

2. **Translation Schemes.** More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

Syntax Directed Definitions

Syntax Directed Definitions are a generalization of context-free grammars in which:

1. Grammar symbols have an associated set of **Attributes**;
2. Productions are associated with **Semantic Rules** for computing the values of attributes Such formalism generates **Annotated Parse-Trees** where each node of the tree is a record with a field for each attribute (e.g.,X.a indicates the attribute a of the grammar symbol X).

The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

We distinguish between two kinds of attributes:

1. **Synthesized Attributes.** They are computed from the values of the attributes of the children nodes.

2. **Inherited Attributes.** They are computed from the values of the attributes of both the siblings and the parent node

Syntax Directed Definitions: An Example

Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called *val*.

PRODUCTION	SEMANTIC RULE
PRODUCTION	SEMANTIC RULE
E -> Ei + T	E.code = Ei.code || T.code || '+'
$F \to (E)$	$F.val := E.val$
$F \to$ digit	$F.val :=$digit.*lexval*

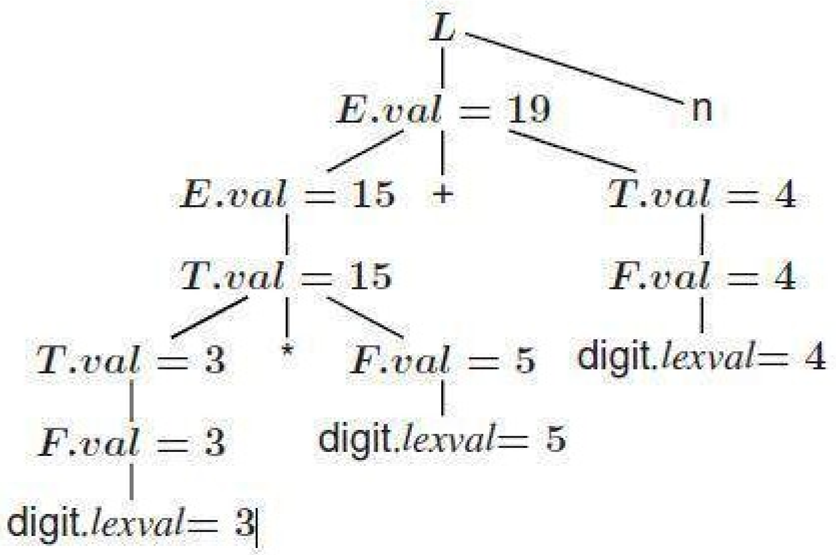
SDD of a simple desk calculator

S-ATTRIBUTED DEFINITIONS

Definition. An **S-Attributed Definition** is a Syntax Directed Definition that uses only synthesized attributes.

• **Evaluation Order.** Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or Post Order, traversal of the parse-tree.

• **Example.** The above arithmetic grammar is an example of an S-Attributed Definition. The annotated parse-tree for the input 3*5+4n is:



L-attributed definition

Definition: A SDD its *L-attributed* if each inherited attribute of Xi in the RHS of A ! X1 : :Xn depends only on

1. attributes of X1;X2; : : : ;Xi 1 (symbols to the left of Xi in the RHS)
2. inherited attributes of A.

Restrictions for translation schemes:

1. Inherited attribute of Xi must be computed by an action before Xi.
2. An action must not refer to synthesized attribute of any symbol to the right of that action.
3. Synthesized attribute for A can only be computed after all attributes it references have been completed (usually at end of RHS).

Evaluation order of SDTS

1 Dependency Graphs

2 Ordering the Evaluation of Attributes

3 S-Attributed Definitions

4 L-Attributed Definitions

"Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.

1 Dependency Graphs

A *dependency graph* depicts the flow of information among the attribute in-stances in a particular parse tree; an edge from one attribute instance to an-other means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules. In more detail:

Suppose that a semantic rule associated with a production *p* defines the value of inherited attribute *B.c* in terms of the value of *X.a.* Then, the dependency graph has an edge from *X.a* to *B.c.* For each node *N* labeled *B* that corresponds to an occurrence of this *B* in the body of production *p,* create an edge to attribute c at *N* from the attribute *a* at the node *M* that corresponds to this occurrence of *X.* Note that *M* could be either the parent or a sibling of *N.*

Since a node ***N*** can have several children labeled ***X,*** we again assume that subscripts distinguish among uses of the same symbol at different places in the production.

Example: Consider the following production and rule:

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

At every node *N* labeled *E,* with children corresponding to the body of this production, the synthesized attribute *val* at *N* is computed using the values of *val* at the two children, labeled *E* and *T.* Thus, a portion of the dependency graph for every parse tree in which this production is used looks like Fig. 5.6. As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid.

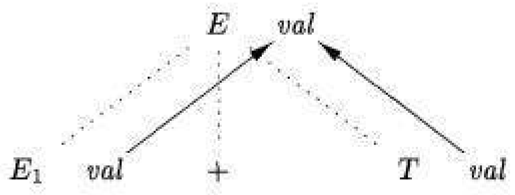


Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $E_2.val$

2. Ordering the Evaluation of Attributes

The dependency graph characterizes the possible orders in which we can evalu-ate the attributes at the various nodes of a parse tree. If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N. Thus, the only allowable orders of evaluation are those sequences of nodes N1, N2,... ,Nk such that if there is an edge of the dependency graph from Ni to Nj, then i < j. Such an ordering embeds a directed graph into a linear order, and is called a topological sort of the graph.

If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree. If there are no cycles, however, then there is always at least one topological sort

3. S-Attributed Definitions

An SDD is *S-attributed* if every attribute is synthesized. When an SDD is S-attributed, we can evaluate its attributes in any bottom-up order of the nodes of the parse tree. It is often especially simple to evaluate the attributes by performing a post order traversal of the parse tree and evaluating the attributes at a node *N* when the traversal leaves *N* for the last time.

S-attributed definitions can be implemented during bottom-up parsing, since a bottom-up parse corresponds to a post order traversal. Specifically, post order corresponds exactly to the order in which anLR parser reduces a production body to its head.

4 L-Attributed Definitions

The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left (hence "L-attributed"). More precisely, each attribute must be either

1. Synthesized, or
2. Inherited, but with the rules limited as follows. Suppose that there is a production A -> X1 X2 .... Xn, and that there is an inherited attribute Xi.a computed by a rule associated with this production.

Then the rule may use only:
Inherited attributes associated with the head A.
Either inherited or synthesized attributes associated with the occurrences of symbols X1, X2,... , X(i-1) located to the left of Xi.
Inherited or synthesized attributes associated with this occurrence of Xi itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X i

Application of SDTS

1 Construction of Syntax Trees
2 The Structure of a Type

The main application is the construction of syntax trees. Since some compilers use syntax trees as an intermediate representation, a common form of SDD turns its input string into a tree. To complete the translation to intermediate code, the compiler may then walk the syntax tree, using another set of rules that are in effect an SDD on the syntax tree rather than the parse tree.

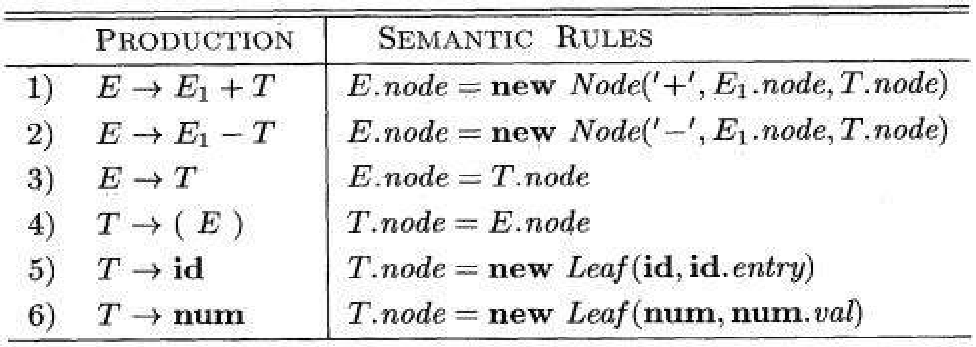
1 Construction of Syntax Trees

Each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct. A syntax-tree node representing an expression *E1* + E_2 has label + and two children representing the subexpressions *E1* and E_2
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,ci,c2,... ,ck) creates an object with first field op and k additional fields for the k children c1,... , .

Example



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

Figure 5.10: Constructing syntax trees for simple expressions

Figure 5.1 1 shows the construction of a syntax tree for the input *a* — 4 + *c*. The nodes of the syntax tree are shown as records, with the *op* field first. Syntax-tree edges are now shown as solid lines. The underlying parse tree, which need not actually be constructed, is shown with dotted edges. The third type of line, shown dashed, represents the values of *E.node* and *T-node;* each line points to the appropriate syntax-tree node.

.

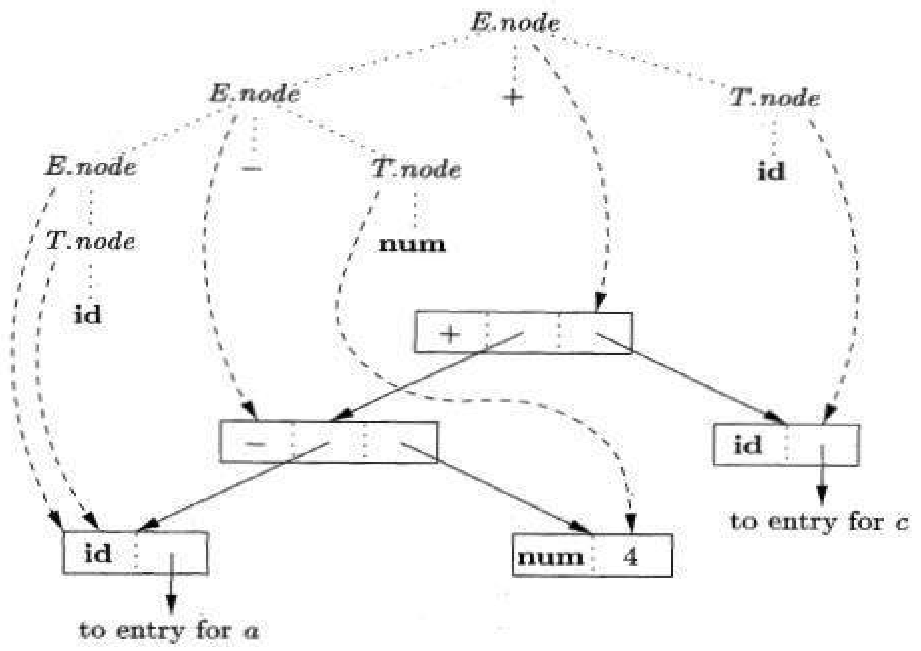


Figure 5.11: Syntax tree for $a - 4 + c$

1) p_1 = **new** *Leaf*(**id**, *entry-a*);
2) p_2 = **new** *Leaf*(**num**, 4);
3) p_3 = **new** *Node*('−', p_1, p_2);
4) p_4 = **new** *Leaf*(**id**, *entry-c*);
5) p_5 = **new** *Node*('+', p_3, p_4);

Figure 5.12: Steps in the construction of the syntax tree for $a - 4 + c$

2 The Structure of a Type

The type int [2][3] can be read as, "array of 2 arrays of 3 integers." The corresponding type expression array(2, array(3, integer)) is represented by the tree in Fig. 5.15. The operator array takes two parameters, a number and a type. If types are represented by trees, then this operator returns a tree node labeled array with two children for a number and a type.

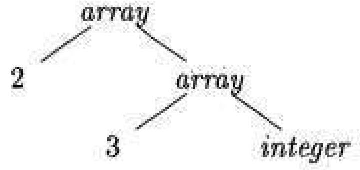


Figure 5.15: Type expression for **int**[2][3]

Nonterminal *B* generates one of the basic types **int** and **float.** *T* generates a basic type when *T* derives *B C* and *C* derives e. Otherwise, *C* generates array components consisting of a sequence of integers, each integer surrounded by brackets.

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

Figure 5.16: T generates either a basic type or an array type

An annotated parse tree for the input string **int [** 2] [3] is shown in Fig. 5.17. The corresponding type expression in Fig. 5.15 is constructed by passing the type *integer* from *B,* down the chain of C's through the inherited attributes *b.* The array type is synthesized up the chain of C's through the attributes *t.*

In more detail, at the root for *T* -» *B C,* nonterminal *C* inherits the type from *B,* using the inherited attribute *C.b.* At the rightmost node for C, the production is C e, so C.t equals C.6. The semantic rules for the production C [num] C1 form C.t by applying the operator array to the operands num.ua/ and C1.t.

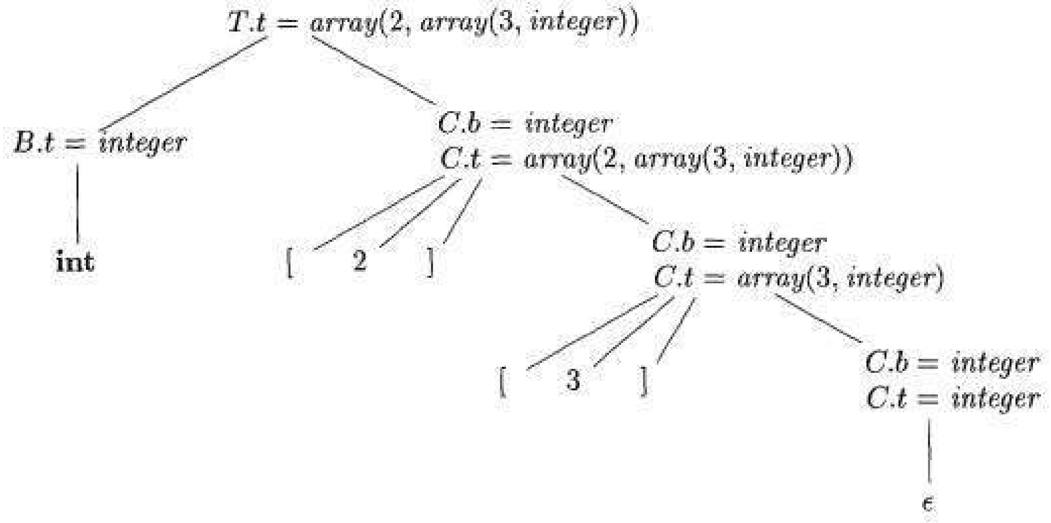


Figure 5.17: Syntax-directed translation of array types

Syntax Directed Translation Schemes.

syntax-directed translation scheme (SDT) is a context-free grammar with program fragments embedded within production bodies. The program fragments are called *semantic actions* and can appear at any position within a production body. By convention, we place curly braces around actions; if braces are needed as grammar symbols, then we quote them.SDT's are implemented during parsing, without building a parse tree.
Two important classes of SDD's are

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.

1 Postfix Translation Schemes

simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed. In that case, we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production. SDT's with all actions at the right ends of the production bodies are called postfix SDT's.

Example 5.14 : The postfix SDT in Fig. 5.18 implements the desk calculator SDD of Fig. 5.1, with one change: the action for the first production prints a value. The remaining actions are exact counterparts of the semantic rules. Since the underlying grammar is LR, and the SDD is S-attributed, these actions can be correctly performed along with the reduction steps of the parser.

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

Figure 5.18: Postfix SDT implementing the desk calculator

2 Parser-Stack Implementation of Postfix SDT's

The attribute(s) of each grammar symbol can be put on the stack in a place where they can be found during the reduction. The best plan is to place the attributes along with the grammar symbols (or the LR states that represent these symbols) in records on the stack itself.
In Fig. 5.19, the parser stack contains records with a field for a grammar symbol (or parser state) and, below it, a field for an attribute. The three grammar symbols *X YZ* are on top of the stack; perhaps they

are about to be reduced according to a production like *A* —> *X YZ.* Here, we show *X.x* as the one attribute of *X,* and so on. In general, we can allow for more attributes, either by making the records large enough or by putting pointers to records on the stack. With small attributes, it may be simpler to make the records large enough, even if some fields go unused some of the time. However, if one or more attributes are of unbounded size — say, they are character strings — then it would be better to put a pointer to the attribute's value in the stack record and store the actual value in some larger, shared storage area that is not part of the stack.

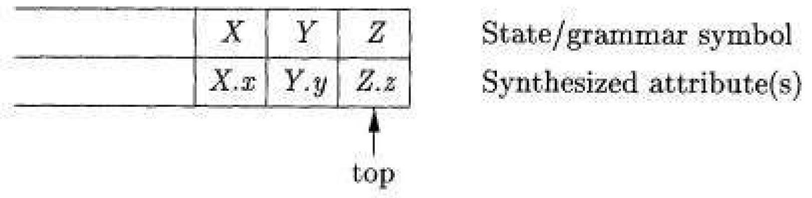


Figure 5.19: Parser stack with a field for synthesized attributes

3 SDT's With Actions Inside Productions

An action may be placed at any position within the body of a production.It is performed immediately after all symbols to its left are processed. Thus,if we have a production B -» X {a} Y, the action a is done after we have recognized X (if X is a terminal) or all the terminals derived from X (if X is a nonterminal).

More precisely,

• 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).

4 Eliminating Left Recursion From SDT's

First, consider the simple case, in which the only thing we care about is the order in which the actions in an SDT are performed. For example, if each action simply prints a string, we care only about the order in which the strings are printed. In this case, the following principle can guide us:

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

This principle is based on the idea that the grammar transformation preserves the order of the terminals in the generated string. The actions are therefore executed in the same order in any left-to-right parse, top-down or bottom-up.

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

A -> *A*a | b

that generate strings consisting of a *j3* and any number of en's, and replace them by productions that generate the same strings using a new nonterminal *R* (for "remainder") of the first production:

A->b*R*

R —»• a*R* | e

If *(3* does not begin with *A,* then *A* no longer has a left-recursive production. In regular-definition terms, with both sets of productions, *A* is defined by ***0****(a)**.

Example **5** . 1 7 : Consider the following E-productions from an SDT for translating infix expressions into postfix notation:

E -> E i + T { print('+'); }

E -> T

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

a = + T { print('-r'); }

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

E --> T R

R --> + T { printC-h'); } R

R -> e

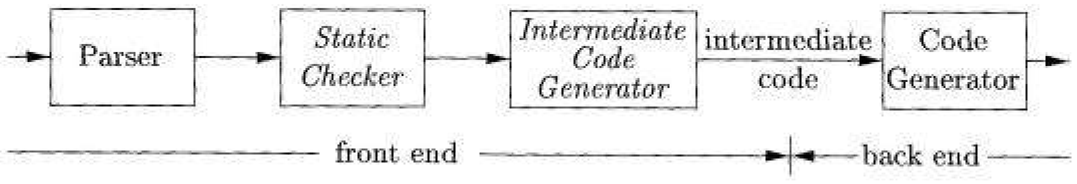
When the actions of an SDD compute attributes rather than merely printing output, we must be more careful about how we eliminate left recursion from a grammar. However, if the SDD is S-attributed, then we can always construct an SDT by placing attribute-computing actions at appropriate positions in the new production.

.

INTERMEDIATE-CODE GENERATION

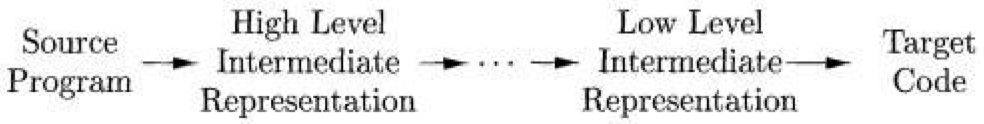
In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. This facilitates *retargeting*: enables attaching a back end for the new machine to an existing front end.

Logical Structure of a Compiler Front End



A compiler front end is organized as in figure above, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. All schemes can be implemented by creating a syntax tree and traversing the tree.

Static checking includes *type checking,* which ensures that operators are applied to compatible operands. In the process of translating a program in a given source language into code for a given target machine, a compiler construct a sequence of intermediate representations


Source Program
High Level Intermediate Representation
...
Low Level Intermediate Representation
Target Code

Sequence of intermediate representations

High-level representations are close to the source language and low-level representations are close to the target machine. A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection.

.

Variants of Syntax Trees

1 Directed Acyclic Graphs for Expressions

2 The Value-Number Method for Constructing DAG's

1. Directed Acyclic Graphs for Expressions

Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior codes corresponding to operators. The difference is that a node *N* in a DAG has more than one parent if *N* represents a com-mon subexpression; in a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression.

Example: Consider expression

a + a * (b - c) + (b - c) * d

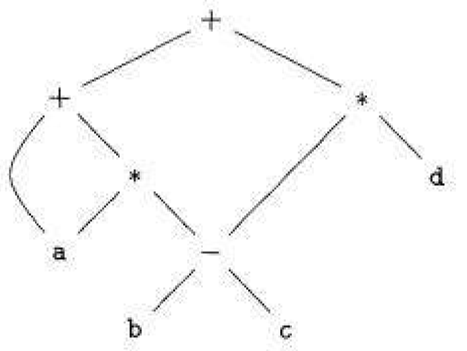
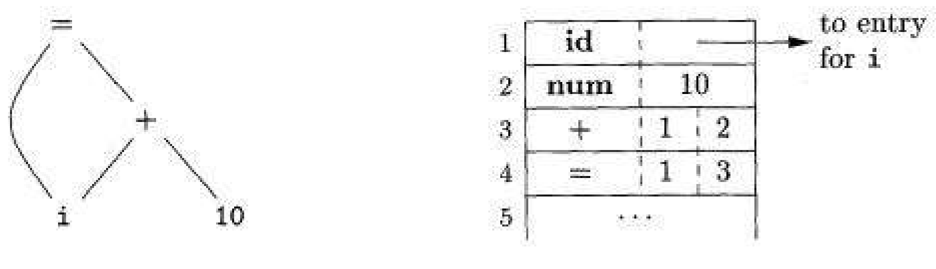


Figure 6.3: Dag for the expression a + a * (b - c) + (b - c) * d

2 The Value-Number Method for Constructing DAG's

The nodes of a syntax tree or DAG are stored in an array of records, as suggested by Fig. 6.6. Each row of the array represents one record, and therefore one node. In each record, the first field is an operation code, indicating the label of the node. In Fig. 6.6(b), leaves have one additional field, which holds the lexical value (either a symbol-table pointer or a constant, in this case), and interior nodes have two additional fields indicating the left and right children.



1	id		
2	num	10	
3	+	1	2
4	=	1	3
5	...		

(a) DAG (b) Array.

Figure 6.6: Nodes of a DAG for $i = i + 10$ allocated in an array

.

In this array, we refer to nodes by giving the integer index of the record for that node within the array. This integer is called the value number for the node.

Algorithm: The value-number method for constructing the nodes of a DAG.

INPUT : Label op, node /, and node r.

OUTPUT : The value number of a node in the array with signature (op, l,r).

METHOD : Search the array for a node M with label op, left child I, and right child r. If there is such a

node, return the value number of M. If not, create in the array a new node N with label op, left child I, and right child r, and return its value number.

Three-Address Code

1 Addresses and Instructions

2 Quadruples

3 Triples

In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted. Thus a source-language expression like x+y*z might be translated into the sequence of three-address instructions

$$t_1 = y * z$$
$$t_2 = x + t_1$$

where ti and t_2 are compiler-generated temporary names.

1 Addresses and Instructions

An address can be one of the following:

- *A name. S*ource-program names to appear as addresses in three-address code. In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.
- *A constant.* A compiler must deal with many different types of constants and variables.
- *A compiler-generated temporary.* Useful in optimizing com-pilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

common three-address instruction

1. **Assignment Statement:** x = y op z and x = op y
   Here,x, y and z are the operands. op represents the operator.
2. **Copy Statement:** x = y
3. **Conditional Jump:** If x relop y goto X

.

If the condition “x relop y” gets **satisfied,** then-

The control is sent directly to the location specified by label X.
All the statements in between are skipped.

If the condition “x relop y” **fails**, then-

The control is not sent to the location specified by label X.
The next statement appearing in the usual sequence is executed.

4. Unconditional Jump- goto X

On executing the statement,The control is sent directly to the location specified by label X.

All the statements in between are skipped.

5. Procedure Call- param x call p return y

Here, p is a function which takes x as a parameter and returns y.
For a procedure call p(x1, …, xn)

```
param x1
   …
param xn
call p, n
```

6. Indexed copy instructions: x = y[i] and x[i] = y
Left: sets x to the value in the location i memory units beyond y
Right: sets the contents of the location i memory units beyond x to y

7. Address and pointer instructions:

x = &y sets the value of x to be the location (address) of y.

x = *y, presumably y is a pointer or temporary whose value is a location. The value of x is set to the contents of that location.

*x = y sets the value of the object pointed to by x to the value of y.

Data Structure

Three address code is represented as record structure with fields for operator and operands. These records can be stored as array or linked list. Most common implementations of three address code are Quadruples, Triples and Indirect triples.

2. Quadruples

Quadruples consists of four fields in the record structure. One field to store operator op, two fields to store operands or arguments arg1and arg2 and one field to store result res.

res = arg1 op arg2

Example: a = b + c

b is represented as arg1, c is represented as arg2, + as op and a as res.

.
Unary operators like „-„do not use agr2. Operators like param do not use agr2 nor result. For conditional and unconditional statements res is label. Arg1, arg2 and res are pointers to symbol table or literal table for the names.

Example: a = -b * d + c + (-b) * d

Three address code for the above statement is as follows

t1 = - b

t2 = t1 * d

t3 = t2 + c

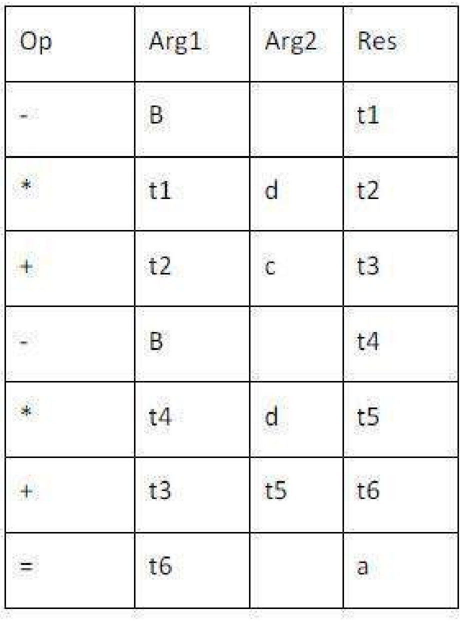
t4 = - b

t5 = t4 * d

t6 = t3 + t5

a = t6

Quadruples for the above example is as follows



Op	Arg1	Arg2	Res
-	B		t1
*	t1	d	t2
+	t2	c	t3
-	B		t4
*	t4	d	t5
+	t3	t5	t6
=	t6		a

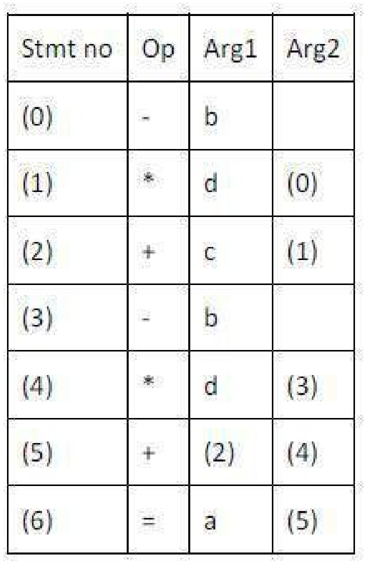
.

3 TRIPLES

Triples uses only three fields in the record structure. One field for operator, two fields for operands named as arg1 and arg2. Value of temporary variable can be accessed by the position of the statement the computes it and not by location as in quadruples.

Example: a = -b * d + c + (-b) * d

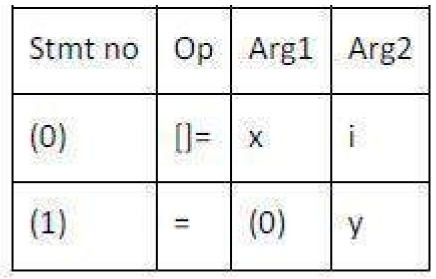
Triples for the above example is as follows



Stmt no	Op	Arg1	Arg2
(0)	-	b	
(1)	*	d	(0)
(2)	+	c	(1)
(3)	-	b	
(4)	*	d	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

Arg1 and arg2 may be pointers to symbol table for program variables or literal table for constant or pointers into triple structure for intermediate results.

Example: Triples for statement x[i] = y which generates two records is as follows



Stmt no	Op	Arg1	Arg2
(0)	[]=	x	i
(1)	=	(0)	y

.

Triples for statement x = y[i] which generates two records is as follows

Stmt no	Op	Arg1	Arg2
(0)	=[]	y	i
(1)	=	x	(0)

Triples are alternative ways for representing syntax tree or Directed acyclic graph
for program defined names.

Indirect Triples

Indirect triples are used to achieve indirection in listing of pointers. That is, it uses pointers to triples than listing of triples themselves.

Example: a = -b * d + c + (-b) * d

	Stmt no
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)

Stmt no	Op	Arg1	Arg2
(10)	-	b	
(11)	*	d	(0)
(12)	+	c	(1)
(13)	-	b	
(14)	*	d	(3)
(15)	+	(2)	(4)
(16)	=	a	(5)

.

RUNTIME ENVIRONMENT

By **runtime**, we mean a program in execution. **Runtime environment** is a state of the target machine, which may include software libraries, **environment** variables, etc., to provide services to the processes running in the system.

Storage Organization

- When the target program executes then it runs in its own logical address space in which the value of each program has a location.
- The logical address space is shared among the compiler, operating system and target machine for management and organization. The operating system is used to map the logical address into physical address which is usually spread throughout the memory.

The run-time representation of an object program in the logical address space consists of data and program areas as shown in Fig. 5.1

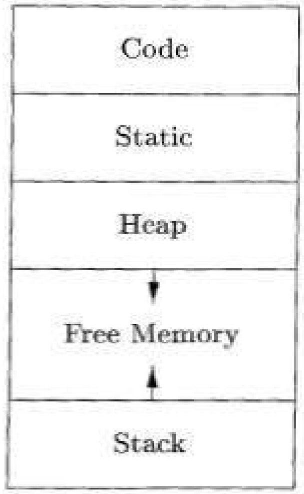


Figure 7.1: Typical subdivision of run-time memory into code and data areas

Storage needed for a name is determined from its type.

- Runtime storage comes into blocks, where a byte is used to show the smallest unit of addressable memory. Using the four bytes a machine word can form. Object of multibyte is stored in consecutive bytes and gives the first byte address.
- Run-time storage can be subdivide to hold the different components of an executing program:

.

1. Generated executable code
2. Static data objects
3. Dynamic data-object- heap
4. Automatic data objects- stack

Two areas, *Stack* and *Heap,* are at the opposite ends of the remainder of the address space. These areas are dynamic; their size can change as the program executes. Stack to support call/return policy for procedures.Heap to store data that can outlive a call to a procedure. . The heap is used to manage allocate and deallocate data.

Static Versus Dynamic Storage Allocation

The layout and allocation of data to memory locations in the run-time environment are key issues in storage management. The two terms *static* and *dynamic* distinguish between compile time and run time, respectively. We say that a storage-allocation decision is

Static:- if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes.
Dynamic:- if it can be decided only while the program is running.

Compilers use following two strategies for dynamic storage allocation:

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

Heap storage. Data that may outlive the call to the procedure that created it is usually allocated on a "heap" of reusable storage.The heap is an area of virtual memory that allows objects or other data elements to obtain storage when they are created and to return that storage when they are invalidated.

Stack allocation of space

1 Activation Trees

2 Activation Records

3 Calling Sequences

4 Variable-Length Data on the Stack

Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

1 Activation Trees

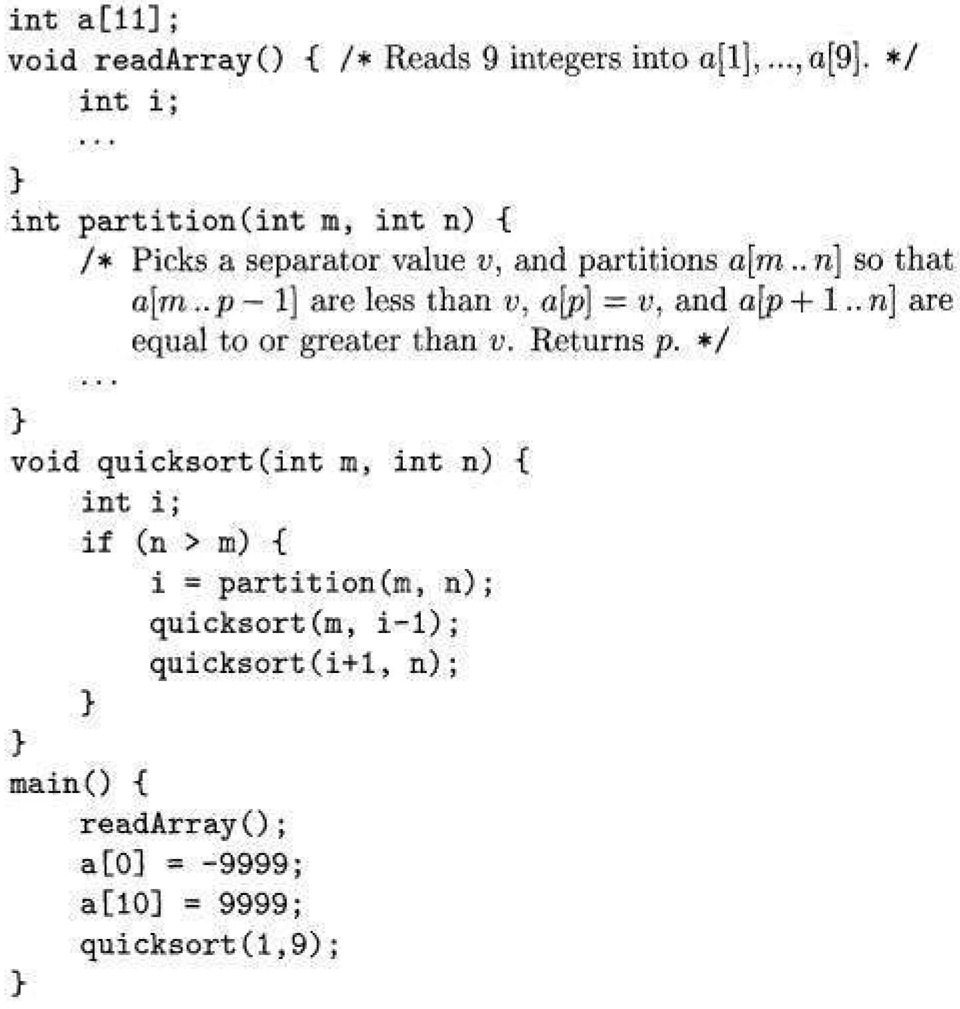
Stack allocation is a valid allocation for procedures since procedure calls are nested

Example: quicksort algorithm

.

The main function has three tasks. It calls *readArray,* sets the sentinels, and then calls *quicksort* on the entire data array.

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.



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

Figure 7.2: Sketch of a quicksort program

Represent the activations of procedures during the running of an entire program by a tree, called an activation tree. Each node corresponds to one activation, and the root is the activation of the "main" procedure that initiates execution of the program. At a node for an activation of procedure p, the children correspond to activations of the procedures called by this activation of p.

.

```
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()
```

Figure 7.3: Possible activations for the program of Fig. 7.2

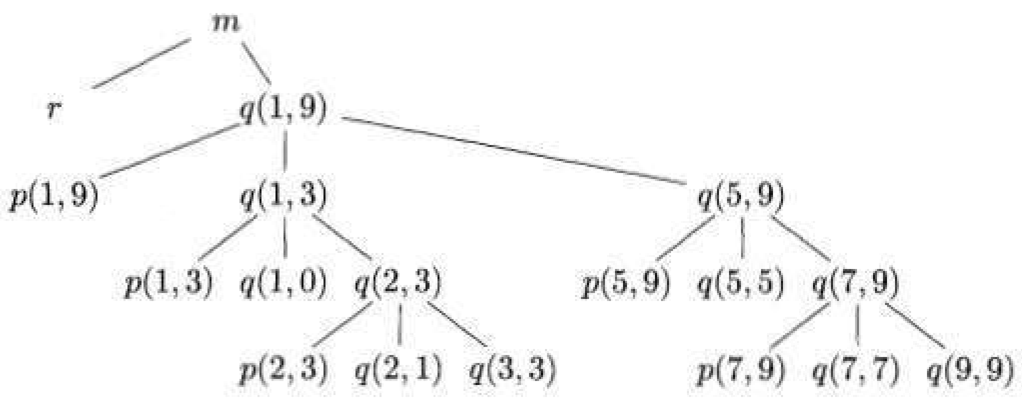


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

2 Activation Records

a. Procedure calls and returns are usually managed by a run-time stack called the control stack.
b. Each live activation has an activation record (sometimes called a frame)
c. The root of activation tree is at the bottom of the stack
d. The current execution path specifies the content of the stack with the last
e. Activation has record in the top of the stack.

.

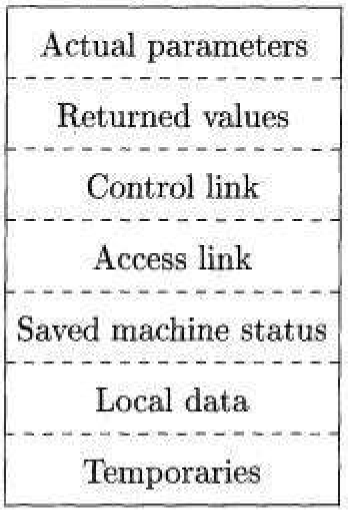


Figure 7.5: A general activation record

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.

An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.
Machine Status	Stores machine status such as Registers, Program Counter etc., before the procedure is called.
Control Link	Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.
Actual Parameters	Stores actual parameters, i.e., parameters which are used to send input to the called procedure.

.

Return Value	Stores return values.

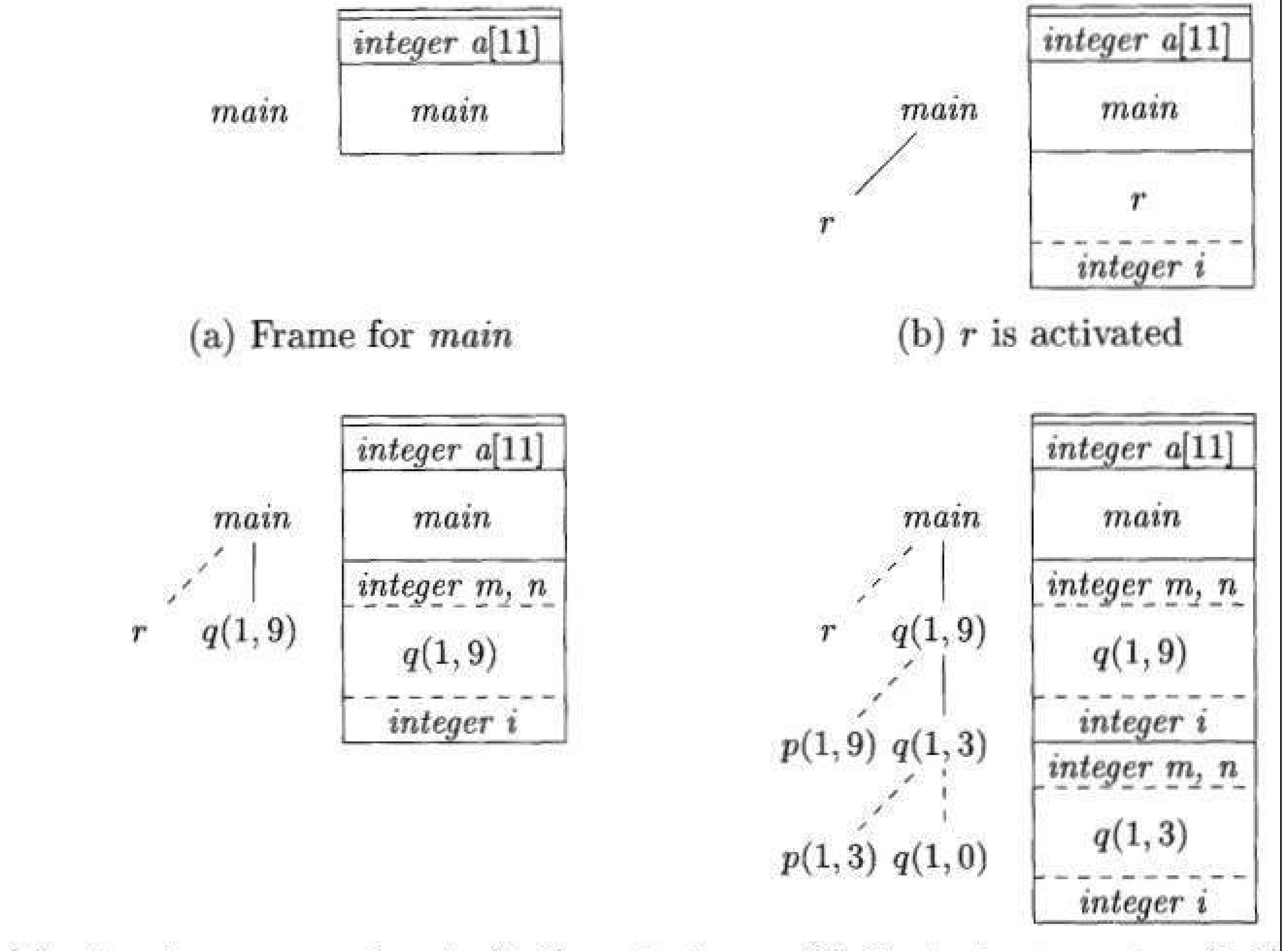


Figure 7.6: Downward-growing stack of activation records

3 Calling Sequences

Designing calling sequences and the layout of activation records, the following

1. Values communicated between caller and callee are generally placed at the beginning of callee's activation record
2. Fixed-length items: are generally placed at the middle. such items typically include the control link, the access link, and the machine status fields.
3. Items whose size may not be known early enough: are placed at the end of activation record
4. We must locate the top-of-stack pointer judiciously: a common approach is to have it point to the end of fixed length fields in the activation record.

.

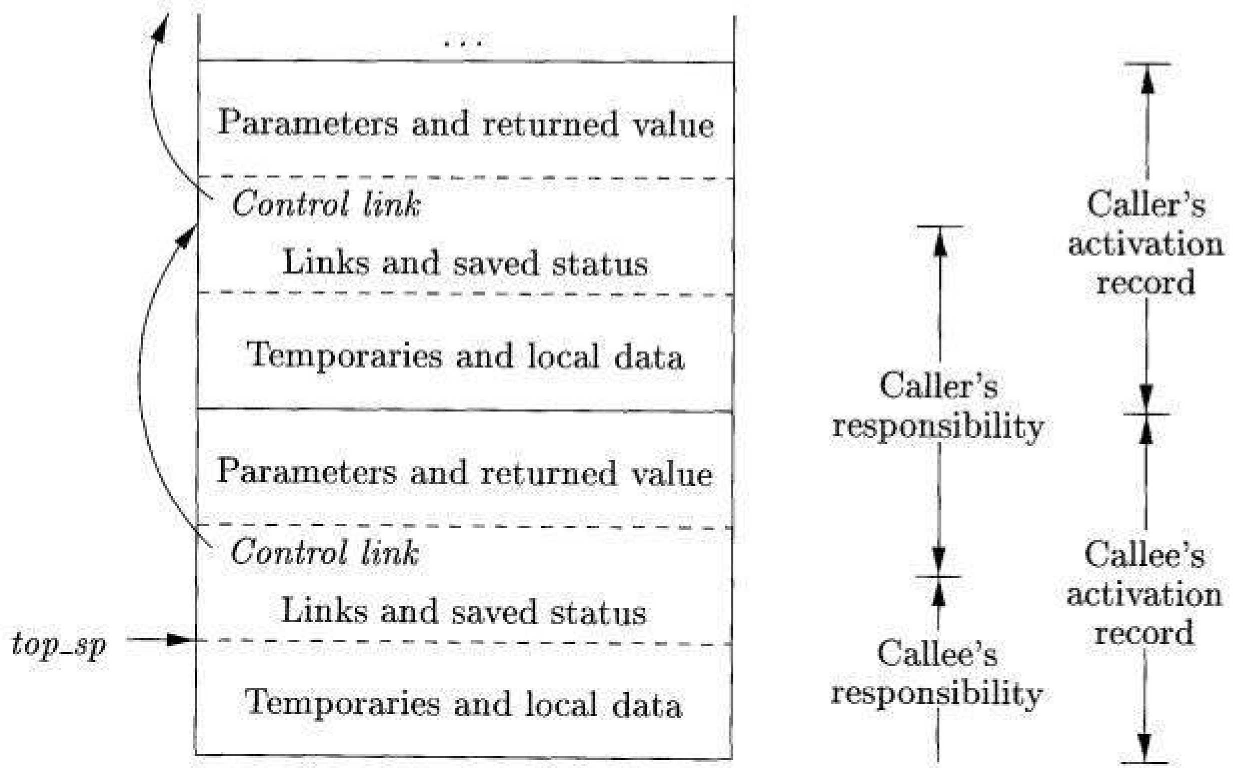

...
Parameters and returned value
Control link
Links and saved status
Temporaries and local data
Parameters and returned value
Control link
Links and saved status
Temporaries and local data
top_sp
Caller's responsibility
Callee's responsibility
Caller's activation record
Callee's activation record

Figure 7.7: Division of tasks between caller and callee

A register topsp points to the end of the machine-status field in the current top activation record. This position within the callee's activation record is known to the caller, so the caller can be made responsible for setting topsp before control is passed to the callee. The calling sequence and its division between caller and callee is as follows:

1. The caller evaluates the actual parameters.

The caller stores a return address and the old value of *topsp* into the callee's activation record. The caller then increments *topsp* to the position shown in Fig. 7.7. That is, *topsp* is moved past the caller's local data and temporaries and the callee's parameters and status fields.

The callee saves the register values and other status information.

The callee initializes its local data and begins execution.

A suitable, corresponding **return sequence** is:

1. The callee places the return value next to the parameters, as in Fig. 7.5.

2. Using information in the machine-status field, the callee restores *topsp* and other registers, and then branches to the return address that the caller placed in the status field.

3. Although *topsp* has been decremented, the caller knows where the return value is, relative to

.

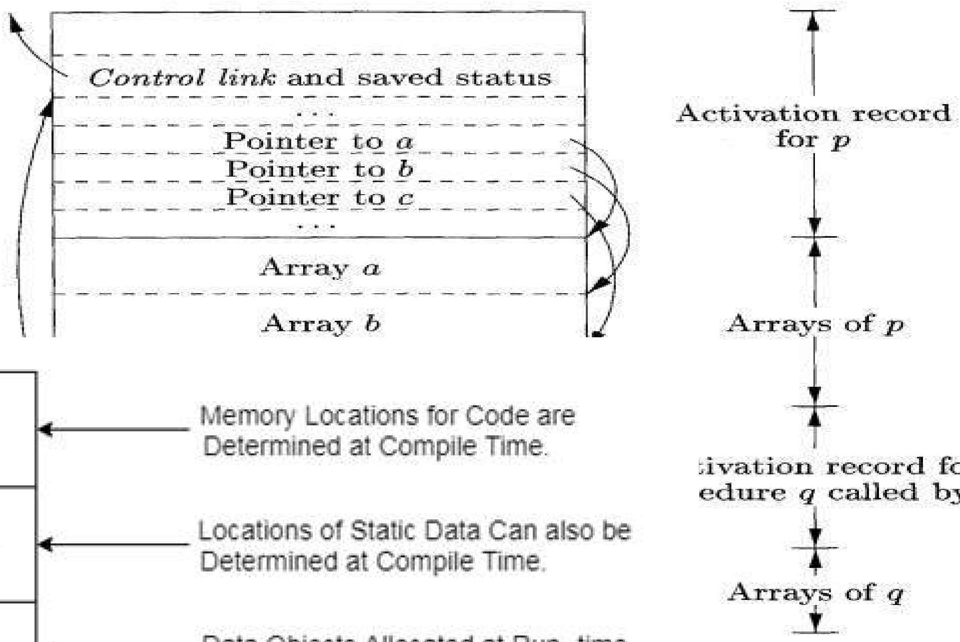
the current value of *topsp;* the caller therefore may use that value.

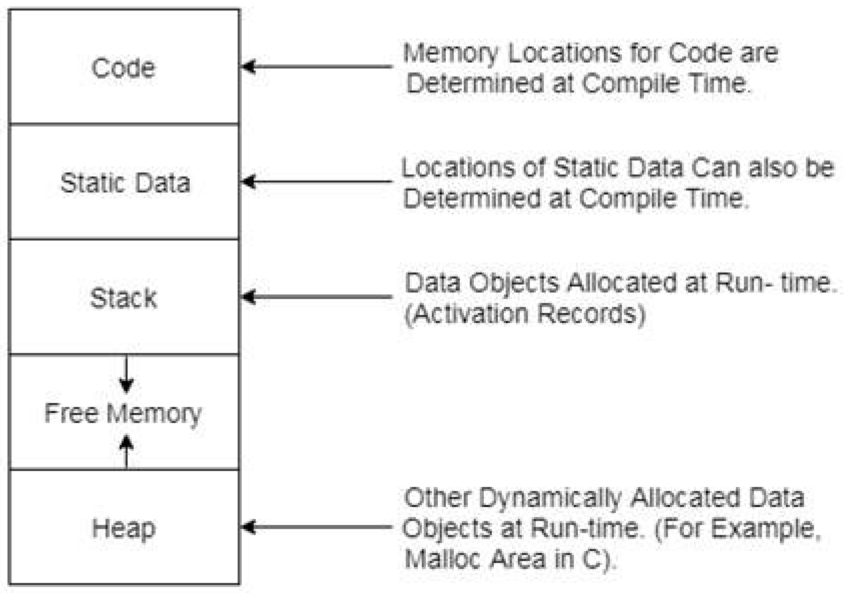
4. Variable-Length Data on the Stack

The run-time memory-management system must deal frequently with the allocation of space for objects the sizes of which are not known at compile time, but which are local to a procedure and thus may be allocated on the stack.

it is possible to allocate objects, arrays, or other structures of unknown size on the stack. The reason to prefer placing objects on the stack if possible is that we avoid the expense of garbage collecting their space. Note that the stack can be used only for an object if it is local to a procedure and becomes inaccessible when the procedure returns.

A common strategy for allocating variable-length arrays (i.e., arrays whose size depends on the value of one or more parameters of the called procedure) is shown in Fig. 7.8. The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.




Code
Static Data
Stack
Free Memory
Heap
Memory Locations for Code are Determined at Compile Time.
Locations of Static Data Can also be Determined at Compile Time.
Data Objects Allocated at Run- time. (Activation Records)
Other Dynamically Allocated Data Objects at Run-time. (For Example, Malloc Area in C).

ted arrays

d by p. The activation record for q
cated beyond that . Access to the

.

Access to Non Local data on the stack

Consider how procedures access their data. Especially im-portant is the mechanism for finding data used within a procedure *p* but that does not belong to *p*

1 Data Access Without Nested Procedures

Names are either local to the procedure in question or are declared globally.

1. For global names the address is known statically at compile time providing there is only one source file. If multiple source files, the linker knows. In either case no reference to the activation record is needed; the addresses are know prior to execution.
2. For names local to the current procedure, the address needed is in the AR at a known-at-compile-time constant offset from the sp. In the case of variable size arrays, the constant offset refers to a pointer to the actual storage.

2 Issues With Nested Procedures

Access becomes far more complicated when a language allows procedure dec-larations to be nested .The reason is that knowing at compile time that the declaration of *p* is immediately nested within *q* does not tell us the relative positions of their activation records at run time. In fact, since either *p* or *q* or both may be recursive, there may be several activation records of *p* and/or *q* on the stack.

Finding the declaration that applies to a nonlocal name *x in a* nested pro-cedure *p* is a static decision; it can be done by an extension of the static-scope rule for blocks. Suppose *x* is declared in the enclosing procedure *q*. Finding the relevant activation of *q* from an activation of *p* is a dynamic decision; it re-quires additional run-time information about activations. One possible solution is to use access links.

3. A Language With Nested Procedure Declarations

In various languages with nested procedures, one of the most influential is ML.

ML is a *functional language,* meaning that variables, once declared and initialized, are not changed. There are only a few exceptions, such as the array, whose elements can be changed by special function calls.

- Variables are defined, and have their unchangeable values initialized,

v a l (name) = (expression)

- Functions are defined using the syntax:

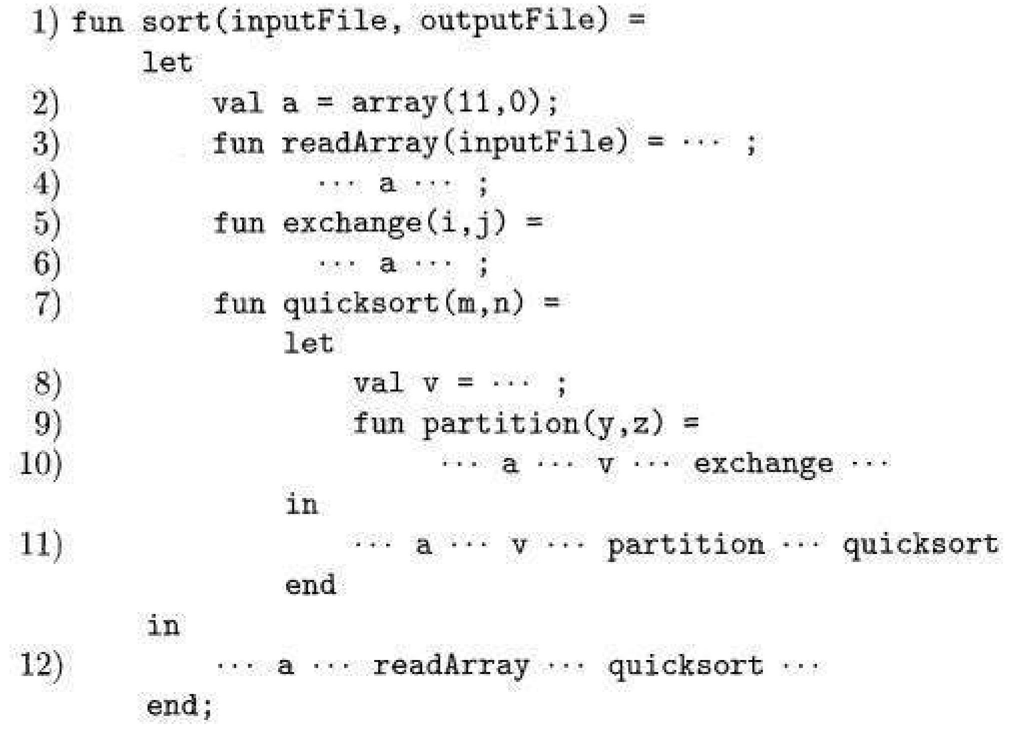
fun (name) ((arguments)) = (body)

- For function bodies, use let-statements of the form:

let (list of definitions) in (statements) end The definitions are normally v a l or fun statements. The scope of each such definition consists of all following definitions, up to the in, and all the statements up to the end. Most importantly, function definitions can be nested. For example, the body of a function p can contain a let-statement that includes the definition of another (nested) function q. Similarly, q can have function definitions within its own body, leading to arbitrarily deep nesting of function

4. Nesting Depth

Nesting depth is 1 to procedures that are not nested within any other procedure. For example, all C functions are at nesting depth 1. However, if a procedure *p* is defined immediately within a procedure at nesting depth *i,* then give *p* the nesting depth *i*



```
fun sort(inputFile, outputFile) =
      let
          val a = array(11,0);
          fun readArray(inputFile) = ··· ;
              ··· a ··· ;
          fun exchange(i,j) =
              ··· a ··· ;
          fun quicksort(m,n) =
              let
                  val v = ··· ;
                  fun partition(y,z) =
                      ··· a ··· v ··· exchange ···
              in
                  ··· a ··· v ··· partition ··· quicksort
              end
      in
          ··· a ··· readArray ··· quicksort ···
      end;
```

Figure 7.10: A version of quicksort, in ML style, using nested functions

+1

.

5. Access Links

A direct implementation of the normal static scope rule for nested functions is obtained by adding a pointer called the *access link* to each activation record. If procedure *p* is nested immediately within procedure *q* in the source code, then the access link in any activation of *p* points to the most recent activation of *q.* Note that the nesting depth of *q* must be exactly one less than the nesting depth of p. Access links form a chain from the activation record at the top of the stack to a sequence of activations at progressively lower nesting depths.

Figure 7.11 shows a sequence of stacks that might result from execution of the function *sort* of Fig. 7.10. In Fig. 7.11(a), we see the situation after *sort* has called *readArray* to load input into the array *a* and then called *quicksort(l,* 9) to sort the array. The access link from *quicksort(l,* 9) points to the activation record for *sort,* not because *sort* called *quicksort* but because *sort* is the most closely nested function surrounding *quicksort* in the program.

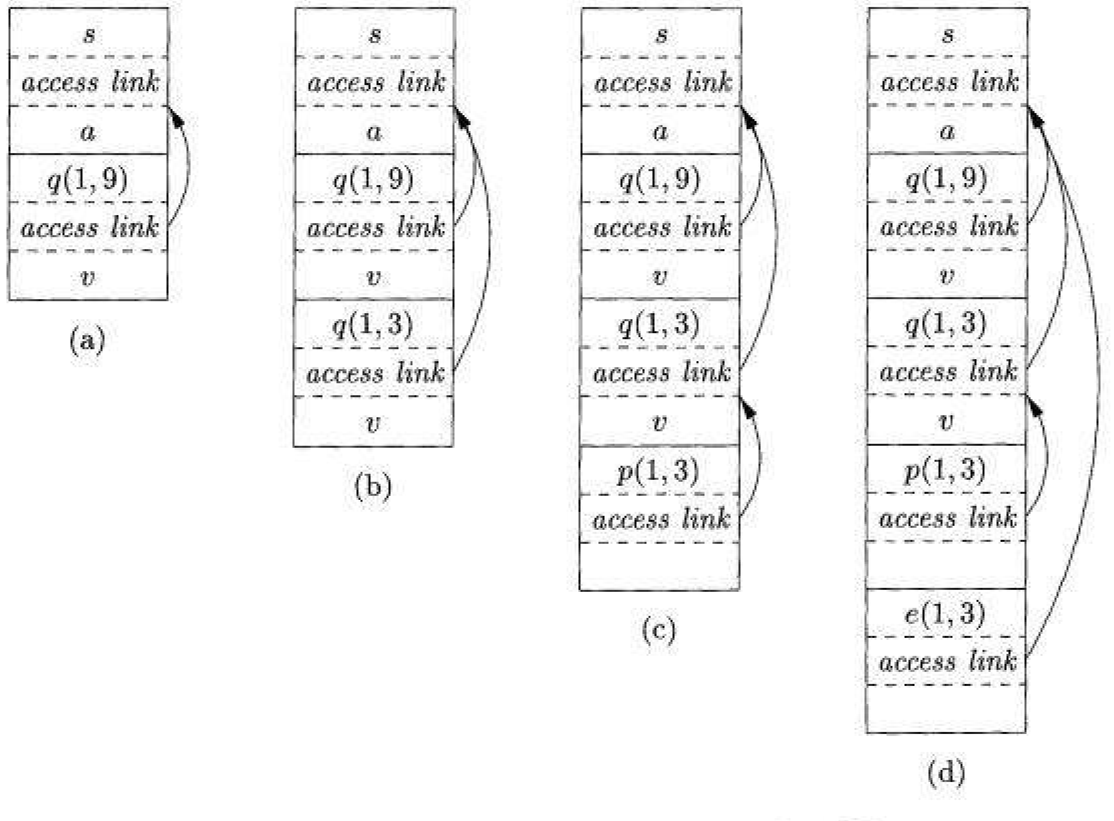


Figure 7.11: Access links for finding nonlocal data

see a recursive call to *quicksort(l,* 3), followed by a call to *partition,* which calls *exchange.* Notice that *quicksort(l,* 3)'s access link points to *sort,* for the same reason that *quicksort(l,* 9)'s does.

6. Manipulating Access Links

The harder case is when the call is to a procedure-parameter; in that case, the particular procedure being called is not known until run time, and the nesting depth of the called procedure may differ in different executions of the call. consider situation when a procedure *q* calls procedure *p,* explicitly. There are three cases:

1. Procedure *p* is at a higher nesting depth than *q*. Then p must be defined immediately within *q,* or the call by *q* would not be at a position that is within the scope of the procedure name *p.* Thus, the nesting depth of *p* is exactly one greater than that of *q,* and the access link from *p* must lead to *q.* It is a simple matter for the calling sequence to include a step that places in the access link for *p* a pointer to the activation record of *q.*

2. The call is recursive, that is, *p* = *q.*Then the access link for the new activation record is the same as that of the activation record below it.

3. The nesting depth np of p is less than the nesting depth nq of q. In order for the call within q to be in the scope of name p, procedure q must be nested within some procedure r, while p is a procedure defined immediately within r. The top activation record for r can therefore be found by following the chain of access links, starting in the activation record for q, for nq — np + 1 hops. Then, the access link for p must go to this activation of r.

7. Access Links for Procedure Parameters

When a procedure *p* is passed to another procedure *q* as a parameter, and *q* then calls its parameter (and therefore calls *p* in this activation of *q),* it is possible that *q* does not know the context in which *p* appears in the program. If so, it is impossible for *q* to know how to set the access link for *p.* The solution to this is, when procedures are used as parameters, the caller needs to pass, along with the name of the procedure-parameter, the proper access link for that parameter. The caller always knows the link, since if *p* is passed by procedure *r* as an actual parameter, then *p* must be a name accessible to r, and therefore, r can determine the access link for *p* exactly as if *p* were being called by *r* directly.

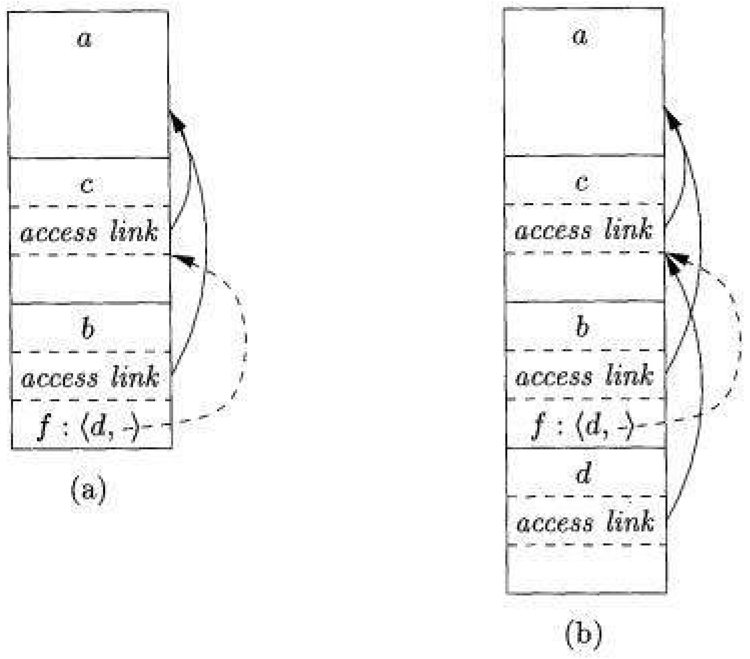


Figure 7.13: Actual parameters carry their access link with them

.

8. Displays

One problem with the access-link approach to nonlocal data is that if the nesting depth gets large, we may have to follow long chains of links to reach the data we need. A more efficient implementation uses an auxiliary array *d,* called the *display,* which consists of one pointer for each nesting depth. We arrange that, at all times, *d[i]* is a pointer to the highest activation record on the stack for any procedure at nesting depth *i.* Examples of a display are shown in Fig. 7.14.

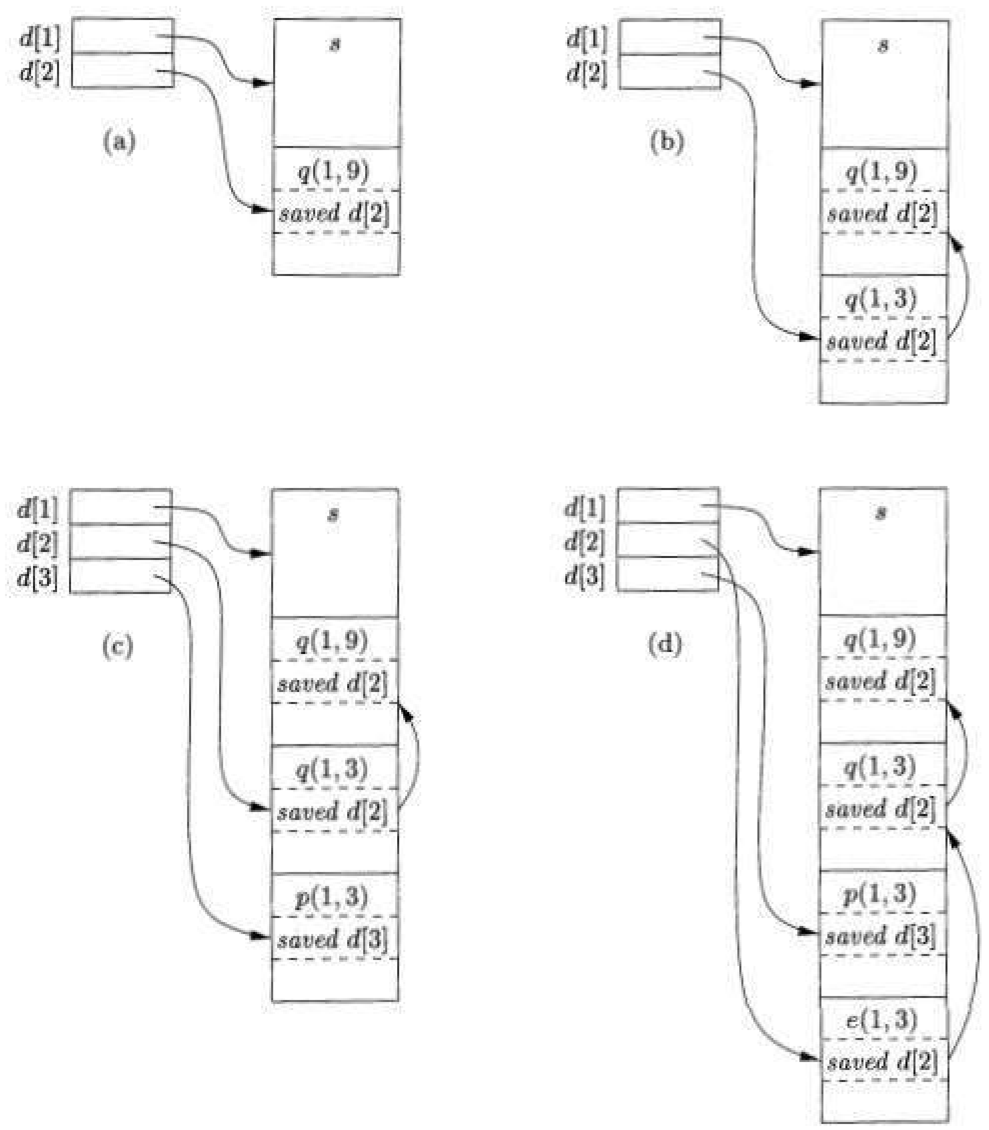


Figure 7.14: Maintaining the display

In order to maintain the display correctly, we need to save previous values of display entries in new activation records.

.

Heap Management

The heap is the portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes it.

1 The Memory Manager

2 The Memory Hierarchy of a Computer

3 Locality in Programs

4 Reducing Fragmentation

5 Manual Deallocation Requests

1 The Memory Manager

It performs two basic functions:

• **Allocation**. When a program requests memory for a variable or object,[3] the memory manager produces a chunk of contiguous heap memory of the requested size. If possible, it satisfies an allocation request using free space in the heap; if no chunk of the needed size is available, it seeks to increase the heap storage space by getting consecutive bytes of virtual memory from the operating system. If space is exhausted, the memory manager passes that information back to the application program.

• **Deallocation**. The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests. Memory managers typically do not return memory to the operating sys-tem, even if the program's heap usage drops.

Thus, the memory manager must be prepared to service, in any order, allo-cation and deallocation requests of any size, ranging from one byte to as large as the program's entire address space.

Here are the properties we desire of memory managers:

• **Space Efficiency**. A memory manager should minimize the total heap space needed by a program. Larger programs to run in a fixed virtual address space..

• **Program Efficiency.** A memory manager should make good use of the memory subsystem to allow programs to run faster.

• **Low Overhead**. Because memory allocations and deallocations are fre-quent operations in many programs, it is important that these operations be as efficient as possible. That is, we wish to minimize the *overhead*

2. The Memory Hierarchy of a Computer

The efficiency of a program is determined not just by the number of instructions executed, but also by how long it takes to execute each of these instructions. The time taken to execute an instruction can vary significantly, since the time taken to access different parts of memory can vary from nanoseconds to milliseconds. Data-intensive programs can therefore benefit significantly from optimizations that make good use of the memory subsystem.

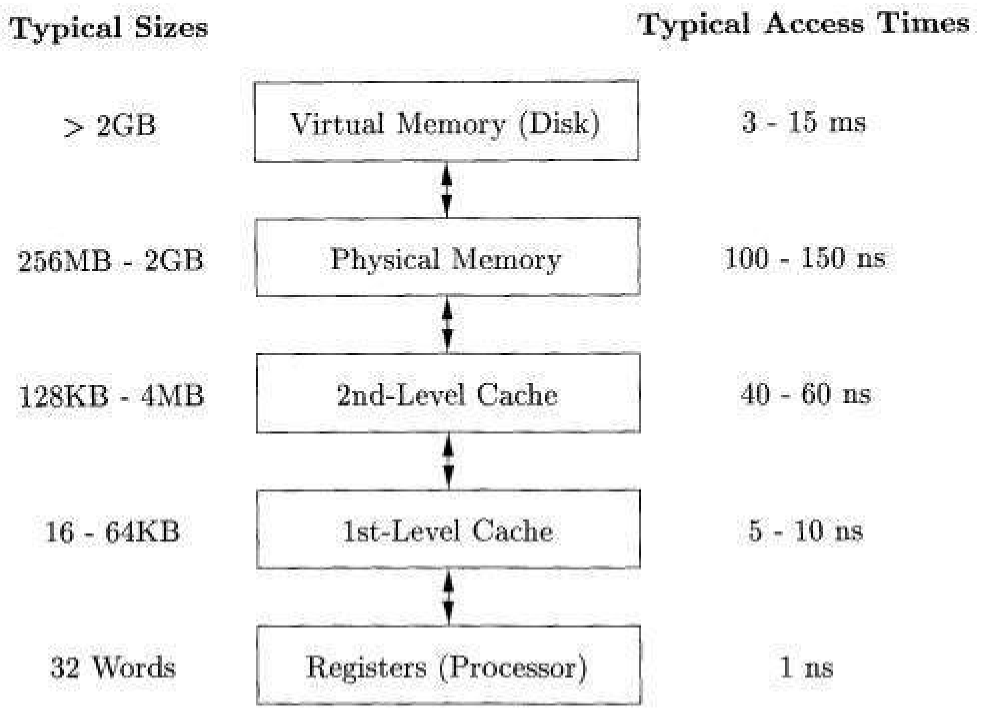


Figure 7.16: Typical Memory Hierarchy Configurations

3. Locality in Programs

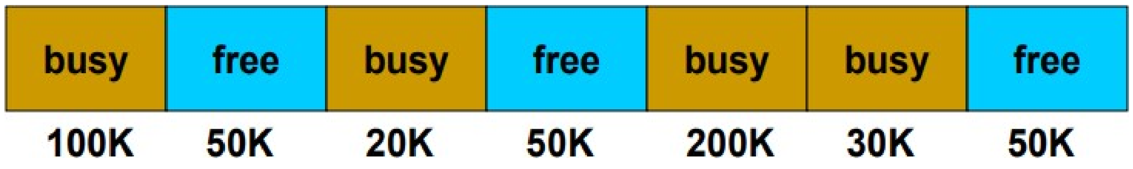
Most programs exhibit a high degree of locality; that is, they spend most of their time executing a relatively small fraction of the code and touching only a small fraction of the data. We say that a program has temporal locality if the memory locations it accesses are likely to be accessed again within a short period of time. We say that a program has *spatial locality* if memory locations close to the location accessed are likely also to be accessed within a short period of time.

Programs spend 90% of their time executing 10% of the code. Programs often contain many instructions that are never executed. Programs built with components and libraries use only a small fraction of the provided functionality.

The typical program spends most of its time executing innermost loops and tight recursive cycles in a program. By placing the most common instructions and data in the fast-but-small storage, while leaving the rest in the slow-but-large storage. Average memory-access time of a program can be lowered significantly.

.

4. Reducing Fragmentation



To begin with the whole heap is a single chunk of size 500K bytes

After a few allocations and deallocations, there are holes

In the above picture, it is not possible to allocate 100K or 150K even though total free memory is 150K

With each deallocation request, the freed chunks of memory are added back to the pool of free space. We coalesce contiguous holes into larger holes, as the holes can only get smaller otherwise. If we are not careful, the memory may end up getting fragmented, consisting of large numbers of small, noncontiguous holes. It is then possible that no hole is large enough to satisfy a future request, even though there may be sufficient aggregate free space.

Best - Fit and Next - Fit Object Placement

We reduce fragmentation by controlling how the memory manager places new objects in the heap. It has been found empirically that a good strategy for minimizing fragmentation for real life programs is to allocate the requested memory in the smallest available hole that is large enough. This *best-fit* algorithm tends to spare the large holes to satisfy subsequent, larger requests. An alternative, called *first-fit,* where an object is placed in the first (lowest-address) hole in which it fits, takes less time to place objects, but has been found inferior to best-fit in overall performance.

To implement best-fit placement more efficiently, we can separate free space into *bins,* according to their sizes.Binning makes it easy to find the best-fit chunk.

M a n a g i n g and Coalescing Free Space

When an object is deallocated manually, the memory manager must make its chunk free, so it can be allocated again. In some circumstances, it may also be possible to combine *(coalesce)* that chunk with adjacent chunks of the heap, to form a larger chunk. There is an advantage to doing so, since we can always use a large chunk to do the work of small chunks of equal total size, but many small chunks cannot hold one large object, as the combined chunk could.

Automatic garbage collection can eliminate fragmentation altogether if it moves all the allocated objects to contiguous storage.

.

5. Manual Deallocation Requests

In manual memory management, where the programmer must explicitly arrange for the deallocation of data, as in C and C + + . Ideally, any storage that will no longer be accessed should be deleted.

Problems with Manual Deallocation

1. Memory leaks ‰

   Failing to delete data that cannot be referenced ‰

   Important in long running or nonstop programs „

2. Dangling pointer dereferencing ‰

   Referencing deleted data „

Both are serious and hard to debug