procedures is to assume that a call generates nothing, and that *a_kill*[*B*] for all blocks *B* is as computed above. As one does not expect many expressions to be generated by the typical procedure, this approach is good enough for most purposes.

A more complicated, and more accurate, alternative approach to the computation of available expressions is to compute *gen*[p] for each procedure p iteratively. We may initialize *gen*[p] to be the set of expressions available at the end of p's return node according to the method above. That is, no aliasing is permitted for generated expressions; a+b represents only itself, even if other variables could be aliases of a or b.

Now compute available expressions for all nodes of all procedures again. However, a call to q(a,b) generates those expressions in *gen*[q] with a and b substituted for the corresponding formals of q. *a_kill* remains as before. A new value of *gen*[p], for each procedure p, can be found by seeing what expressions are available at the end of p's return. This iteration may be repeated until we get no more changes in available expressions at any node.

10.9 DATA-FLOW ANALYSIS OF STRUCTURED FLOW GRAPHS

Gotoless programs have reducible flow graphs; so do programs encouraged by several programming methodologies. Several studies of large classes of programs have revealed that virtually all programs written by people have flow graphs that are reducible.[10] This observation is relevant for optimization purposes because we can find optimization algorithms that run significantly faster on reducible flow graphs. In this section we discuss a variety of flow-graph concepts, such as "interval analysis," that are primarily relevant to structured flow graphs. In essence, we shall apply the syntax-directed techniques developed in Section 10.5 to the more general setting where the syntax doesn't necessarily provide the structure, but the flow graph does.

Depth-First Search

There is a useful ordering of the nodes of a flow graph, known as *depth-first* ordering, which is a generalization of the depth-first traversal of a tree introduced in Section 2.3. A depth-first ordering can be used to detect loops in any flow graph; it also helps speed up iterative data-flow algorithms such as those discussed in Section 10.6. The depth-first ordering is created by starting at the initial node and searching the entire graph, trying to visit nodes as far away from the initial node as quickly as possible (*depth first*). The route of the search forms a tree. Before we give the algorithm, let us consider an example.

[10] "Written by people" is not redundant because we know of several programs that generate code with "rats' nests" of goto's. There is nothing wrong with this; the structure is in the input to these programs.

Example 10.30. One possible depth-first search of the flow graph in Fig. 10.45 is illustrated in Fig. 10.46. Solid edges form the tree; dashed edges are the other edges of the flow graph. The depth-first search of the flow graph corresponds to a preorder traversal of the tree, $1 \to 3 \to 4 \to 6 \to 7 \to 8 \to 10$, then back to 8, then to 9. We go back to 8 once more, retreating to 7, 6, and 4, and then forward to 5. We retreat from 5 back to 4, then back to 3 and 1. From 1 we go to 2, then retreat from 2, back to 1, and we have traversed the entire tree in preorder. Note that we have not yet explained how the tree is selected from the flow graph. □

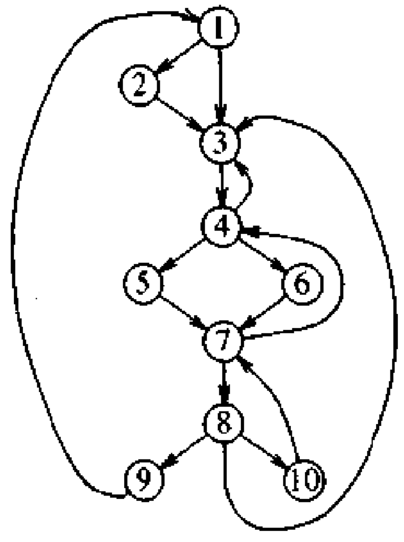

Fig. 10.45. Flow graph

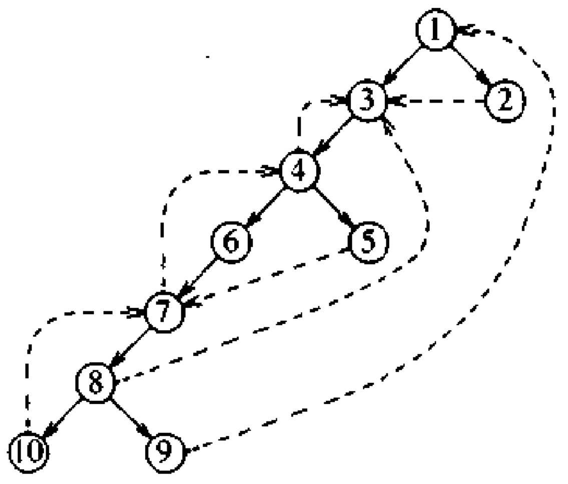

Fig. 10.46. Depth-first presentation

The *depth-first ordering* of the nodes is the reverse of the order in which we last visit the nodes in the preorder traversal.

Example 10.31. In Example 10.30, the complete sequence of nodes visited as we traverse the tree is

1, 3, 4, 6, 7, 8, 10, 8, 9, 8, 7, 6, 4, 5, 4, 3, 1, 2, 1.

In this list, mark the last occurrence of each number to get

$1, 3, 4, 6, 7, 8, \underline{10}, 8, \underline{9}, \underline{8}, \underline{7}, \underline{6}, 4, \underline{5}, \underline{4}, \underline{3}, 1, \underline{2}, \underline{1}$

The depth-first ordering is the sequence of marked numbers in reverse order. Here, this sequence happens to be 1, 2, . . . , 10. That is, initially the nodes were numbered in depth-first order. □

We now give an algorithm that computes a depth-first ordering of a flow graph by constructing and traversing a tree rooted at the initial node, trying to make paths in the tree as long as possible. Such a tree is called a *depth-first spanning tree* (*dfst*). It is this algorithm that was used to construct Fig. 10.46 from Fig. 10.45.

Algorithm 10.14. Depth-first spanning tree and depth-first ordering.

Input. A flow graph G.

Output. A *dfst* T of G and an ordering of the nodes of G.

Method. We use the recursive procedure *search*(n) of Fig. 10.47; the algorithm is to initialize all nodes of G to "unvisited," then call *search*(n_0), where n_0 is the initial node. When we call *search*(n), we first mark n "visited," to avoid adding n to the tree twice. We use i to count from the number of nodes of G down to 1, assigning depth-first numbers *dfn*[n] to nodes n as we go. The set of edges T forms the depth-first spanning tree for G, and they are called *tree edges.* □

```
     procedure search(n);
     begin
(1)      mark n "visited";
(2)      for each successor s of n do
(3)          if s is "unvisited" then begin
(4)              add edge n → s to T;
(5)              search(s)
             end;
(6)      dfn[n] := i;
(7)      i := i − 1
     end;

     /* main program follows */

(8)  T := empty;  /* set of edges */
(9)  for each node n of G do mark n "unvisited";
(10) i := number of nodes of G;
(11) search(n₀)
```

Fig. 10.47. Depth-first search algorithm.

Example 10.32. Consider Fig. 10.47. We set i to 10 and call *search*(1). At line (2) of *search* we must consider each successor of node 1. Suppose we consider $s = 3$ first. Then we add edge $1 \to 3$ to the tree and call *search*(3). In *search*(3) we add edge $3 \to 4$ to T and call *search*(4).

Suppose in *search*(4) we choose $s = 6$ first. Then we add edge $4 \to 6$ to T and call *search*(6). This in turn causes us to add $6 \to 7$ to T and call *search*(7). Node 7 has two successors, 4 and 8. But 4 was already marked "visited" by *search*(4), so we do nothing when $s = 4$. When $s = 8$ we add edge $7 \to 8$ to T and call *search*(8). Suppose we then choose $s = 10$. We add edge $8 \to 10$ and call *search*(10).

Now 10 has a successor, 7, but 7 is already marked "visited," so in *search*(10), we fall through to step (6) in Fig. 10.47, setting *dfn*[10] = 10 and

$i = 9$. This completes the call to *search*(10), so we return to *search*(8). We now set $s = 9$ in *search*(8), add edge $8 \rightarrow 9$ to T and call *search*(9). The only successor of 9, node 1, is already "visited," so we set $dfn[9] = 9$ and $i = 8$. Then we return to *search*(8). The last successor of 8, node 3, is "visited," so we do nothing for $s = 3$. At this point, we have considered all successors of 8, so we set $dfn[8] = 8$ and $i = 7$, returning to *search*(7).

All of 7's successors have been considered, so we set $dfn[7] = 7$ and $i = 6$, returning to *search*(6). Similarly, 6's successors have been considered, so we set $dfn[6] = 6$ and $i = 5$, and we return to *search*(4). Successor 3 of 4 has been "visited," but 5 has not, so we add $4 \rightarrow 5$ to the tree and call *search*(5), which results in no further calls, as successor 7 of 5 has been "visited." Thus, $dfn[5] = 5$, i is set to 4, and we return to *search*(4). We have completed consideration of the successors of 4, so we set $dfn[4] = 4$ and $i = 3$, returning to *search*(3). Then we set $dfn[3] = 3$ and $i = 2$ and return to *search*(1).

The final steps are to call *search*(2) from *search*(1), set $dfn[2] = 2$, $i = 1$, return to *search*(1), set $dfn[1] = 1$ and $i = 0$. Note that we chose a numbering of the nodes such that $dfn[i] = i$, but that relation need not hold for an arbitrary graph, or even for another depth-first ordering of the graph of Fig. 10.45. □

Edges in a Depth-First Presentation of a Flow Graph

When we construct a *dfst* for a flow graph, the edges of the flow graph fall into three categories.

1. There are edges that go from a node m to an ancestor of m in the tree (possibly to m itself). These edges we shall term *retreating* edges. For example, $7 \rightarrow 4$ and $9 \rightarrow 1$ are retreating edges in Fig. 10.46. It is an interesting and useful fact that if the flow graph is reducible, then the retreating edges are exactly the back edges of the flow graph,[11] independent of the order in which successors are visited in step (2) of Fig. 10.47. For any flow graph, every back edge is retreating, although if the graph is nonreducible there will be some retreating edges that are not back edges.

2. There are edges, called *advancing edges*, that go from a node m to a proper descendant of m in the tree. All edges in the *dfst* itself are advancing edges. There are no other advancing edges in Fig. 10.46, but, for example, if $4 \rightarrow 8$ were an edge, it would be in this category.

3. There are edges $m \rightarrow n$ such that neither m nor n is an ancestor of the other in the *dfst*. Edges $2 \rightarrow 3$ and $5 \rightarrow 7$ are the only such examples in Fig. 10.46. We call these edges *cross edges*. An important property of

[11] Recall, the back edges of a flow graph are those whose heads dominate their tails.

cross edges is that if we draw the *dfst* so children of a node are drawn from left to right in the order in which they were added to the tree, then all cross edges travel from right to left.

It should be noted that $m \rightarrow n$ is a retreating edge if and only if $dfn[m] \geq dfn[n]$. To see why, note that if m is a descendant of n in the *dfst*, then $search(m)$ terminates before $search(n)$, so $dfn[m] \geq dfn[n]$. Conversely, if $dfn[m] \geq dfn[n]$, then $search(m)$ terminates before $search(n)$, or $m = n$. But $search(n)$ must have begun before $search(m)$ if there is an edge $m \rightarrow n$, or else the fact that n is a successor of m would have made n a descendant of m in the *dfst*. Thus the time $search(m)$ is active is a subinterval of the time $search(n)$ is active, from which it follows that n is an ancestor of m in the *dfst*.

Depth of a Flow Graph

There is an important parameter of flow graphs called the *depth*. Given a depth-first spanning tree for the graph, the depth is the largest number of retreating edges on any cycle-free path.

Example 10.33. In Fig. 10.46, the depth is 3, since there is a path

$$10 \rightarrow 7 \rightarrow 4 \rightarrow 3$$

with three retreating edges, but no cycle-free path with four or more retreating edges. It is a coincidence that the "deepest" path here has only retreating edges; in general we may have a mixture of retreating, advancing, and cross edges in a "deepest" path. □

We can prove the depth is never greater than what one would intuitively call the depth of loop nesting in the flow graph. If a flow graph is reducible, we may replace "retreating" by "back" in the definition of "depth," since the retreating edges in any *dfst* are exactly the back edges. The notion of depth then becomes independent of the *dfst* actually chosen.

Intervals

The division of a flow graph into intervals serves to put a hierarchical structure on the flow graph. That structure in turn allows us to apply the rules for syntax-directed data-flow analysis whose development began in Section 10.5.

Intuitively, an "interval" in a flow graph is a natural loop plus an acyclic structure that dangles from the nodes of that loop. An important property of intervals is that they have *header* nodes that dominate all the nodes in the interval; that is, every interval is a region. Formally, given a flow graph G with initial node n_0, and a node n of G, the *interval with header* n, denoted $I(n)$, is defined as follows.

1. n is in $I(n)$.
2. If all the predecessors of some node $m \neq n_0$ are in $I(n)$, then m in is $I(n)$.
3. Nothing else is in $I(n)$.

We therefore may build I(n) by starting with n, and adding nodes m by rule (2). It does not matter in which order we add two candidates m because once a node's predecessors are all in I(n), they remain in I(n), and each candidate will eventually be added by rule (2). Eventually, no more nodes can be added to I(n), and the resulting set of nodes is the interval with header n.

Interval Partitions

Given any flow graph G, we can partition G into disjoint intervals as follows.

Algorithm 10.15. Interval analysis of a flow graph.

Input. A flow graph G with initial node n_0.

Output. A partition of G into a set of disjoint intervals.

Method. For any node n, we compute I(n) by the method sketched above:

```
I($n$) := {$n$};
while there exists a node $m \neq n_0$,
      all of whose predecessors are in I($n$) do
          I($n$) := I($n$) ∪ {$m$}
```

The particular nodes that are headers of intervals in the partition are chosen as follows. Initially, no nodes are "selected."

```
construct I($n_0$) and "select" all nodes in that interval;
while there is a node $m$, not yet "selected,"
      but with a selected predecessor do
          construct I($m$) and "select" all nodes in that interval
```
□

Once a candidate m has a predecessor p selected, m can never be added to some interval not containing p. Thus, candidate m's remain candidates until they are selected to head their own interval. Thus, the order in which interval headers m are picked in Algorithm 10.15 does not affect the final partition into intervals. Also, as long as all nodes are reachable from n_0, it can be shown by induction on the length of a path from n_0 to n that node n will eventually either be put in some other node's interval, or will become a header of its own interval, but not both. Thus, the set of intervals constructed in Algorithm 10.15 truly partition G.

Example 10.34. Let us find the interval partition for Fig. 10.45. We start by constructing I(1), because node 1 is the initial node. We can add 2 to I(1) because 2's only predecessor is 1. However, we cannot add 3 because it has predecessors, 4 and 8, that are not yet in I(1), and similarly, every other node except 1 and 2 has predecessors not yet in I(1). Thus, I(1) = {1,2}.

We may now compute I(3) because 3 has some "selected" predecessors, 1 and 2, but 3 is not itself in an interval. However, no node can be added to I(3), so I(3) = {3}. Now 4 is a header because it has a predecessor, 3, in an interval. We can add 5 and 6 to I(4) because these have only 4 as a predecessor but no other nodes can be added; e.g., 7 has predecessor 10.

Next, 7 becomes a header, and we can add 8 to I(7). Then, we can add 9 and 10, because these have only 8 as predecessor. Thus, the intervals in the partition of Fig. 10.45 are:

$$\begin{array}{ll} I(1) = \{1,2\} & I(4) = \{4,5,6\} \\ I(3) = \{3\} & I(7) = \{7,8,9,10\} \end{array}$$ □

Interval Graphs

From the intervals of one flow graph G, we can construct a new flow graph $I(G)$ by the following rules.

1. The nodes of $I(G)$ correspond to the intervals in the interval partition of G.
2. The initial node of $I(G)$ is the interval of G that contains the initial node of G.
3. There is an edge from interval I to a different interval J if and only if in G there is an edge from some node in I to the header of J. Note that there could not be an edge entering some node n of J other than the header, from outside J, because then there would be no way n could have been added to J in Algorithm 10.15.

We may apply Algorithm 10.15 and the interval graph construction alternately, producing the sequence of graphs G, $I(G)$, $I(I(G))$, ... Eventually, we shall come to a graph each of whose nodes is an interval all by itself. This graph is called the *limit flow graph* of G. It is an interesting fact that a flow graph is reducible if and only if its limit flow graph is a single node.[12]

Example 10.35. Fig. 10.48 shows the result of applying the interval construction repeatedly to Fig. 10.45. The intervals of that graph were given in Example 10.34, and the interval graph constructed from these is in Fig. 10.48(a). Note that the edge $10 \rightarrow 7$ in Fig. 10.45 does not cause an edge from the node representing {7,8,9,10} to itself in Fig. 10.48(a), because the interval graph construction explicitly excluded such loops. Also note that the flow graph of Fig. 10.45 is reducible because its limit flow graph is a single node. □

Node Splitting

If we reach a limit flow graph that is other than a single node, we can proceed further only if we split one or more nodes. If a node n has k predecessors, we may replace n by k nodes, $n_1, n_2, \ldots, n_k$. The ith predecessor of n becomes the predecessor of n_i only, while all successors of n become

[12] In fact, this definition is historically the original definition.

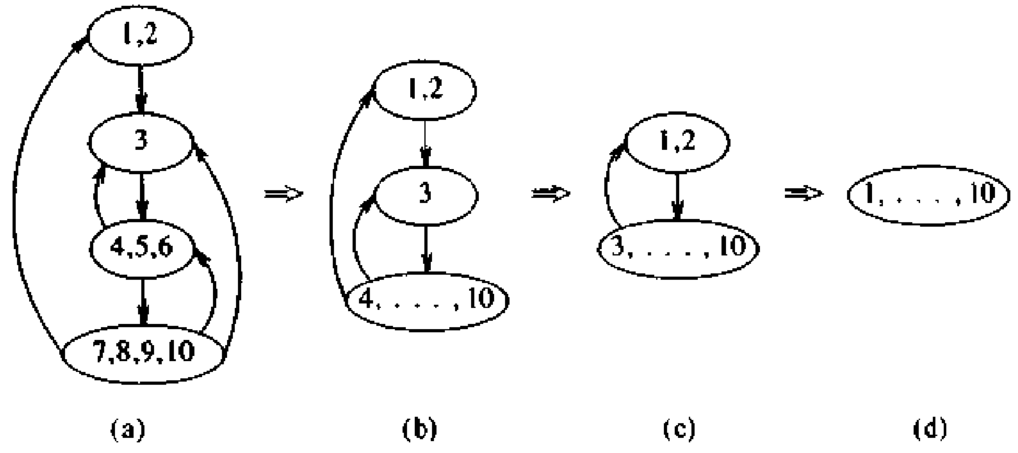

Fig. 10.48. Interval graph sequence.

successors of all of the n_i's.

If we apply Algorithm 10.15 to the resulting graph, each n_i has a unique predecessor, and so it will surely become part of that predecessor's interval. Thus, one node splitting plus one round of interval partitioning results in a graph with fewer nodes. As a consequence, the construction of interval graphs, interspersed when necessary with node splitting, must eventually attain a graph of a single node. The significance of this observation will become clear in the next section, when we design data-flow analysis algorithms that are driven by these two operations on graphs.

Example 10.36. Consider the flow graph of Fig. 10.49(a), which is its own limit flow graph. We may split node 2 into $2a$ and $2b$, with predecessors 1 and 3, respectively. This graph is shown in Fig. 10.49(b). If we apply interval partitioning twice, we get the sequence of graphs shown in Fig. 10.49(c) and (d), leading to a single node. □

T_1 - T_2 Analysis

A convenient way to achieve the same effect as interval analysis is to apply two simple transformations to flow graphs.

T_1: If n is a node with a loop, i.e., an edge $n \rightarrow n$, delete that edge.

T_2: If there is a node n, not the initial node, that has a unique predecessor, m, then m may *consume* n by deleting n and making all successors of n (including m, possibly) be successors of m.

Some interesting facts about the T_1 and T_2 transformations are:

1. If we apply T_1 and T_2 to a flow graph G in any order, until a flow graph results for which no applications of T_1 or T_2 are possible, then a unique flow graph results. The reason is that a candidate for loop removal by T_1

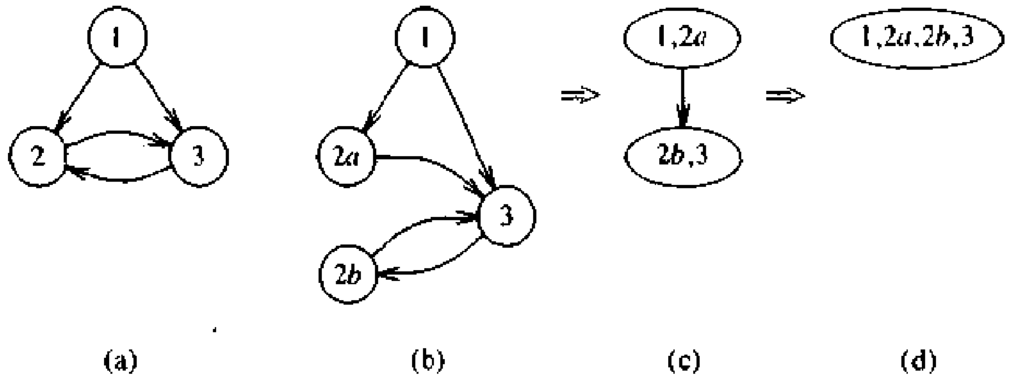

Fig. 10.49. Node splitting followed by interval partitioning.

or consumption by T_2 remains a candidate, even if some other application of T_1 or T_2 is made first.

2. The flow graph resulting from exhaustive application of T_1 and T_2 to G is the limit flow graph of G. The proof is somewhat subtle and left as an exercise. As a consequence, another definition of "reducible flow graph" is one that can be converted to a single node by T_1 and T_2.

Example 10.37. In Fig. 10.50 we see a sequence of T_1 and T_2 reductions starting from a flow graph that is a renaming of Fig. 10.49(b). In Fig. 10.50(b), c has consumed d. Note that the loop on cd in Fig. 10.50(b) results from the edge $d \rightarrow c$ in Fig. 10.50(a). That loop is removed by T_1 in Fig. 10.50(c). Also note that when a consumes b in Fig. 10.50(d), the edges from a and b to the node cd become a single edge. □

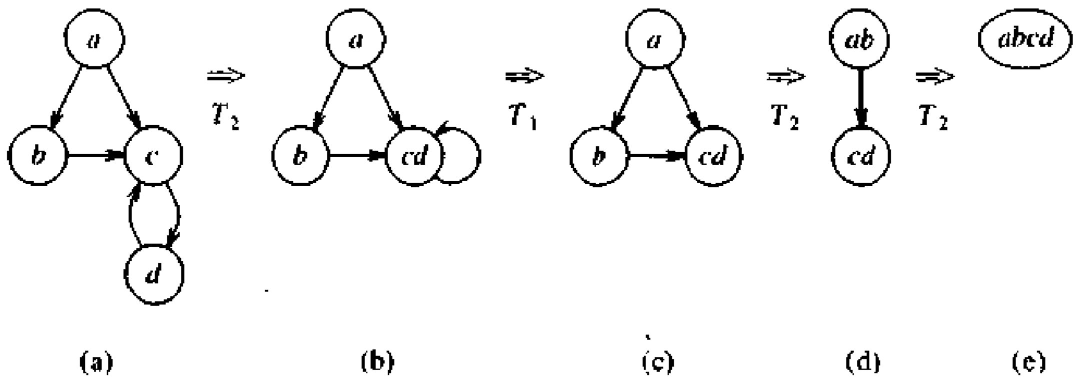

Fig. 10.50. Reduction by T_1 and T_2.

Regions

Recall from Section 10.5 that a region in a flow graph is a set of nodes N that includes a header, which dominates all the other nodes in a region. All edges between nodes in N are in the region, except (possibly) for some of those that enter the header. For example, every interval is a region, but there are regions that are not intervals because, for example, they may omit some nodes an interval would include, or they may omit some edges back to the header. There are also regions much larger than any interval, as we shall see.

As we reduce a flow graph G by T_1 and T_2, at all times the following conditions are true:

1. A node represents a region of G.
2. An edge from a to b represents a set of edges. Each such edge is from some node in the region represented by a to the header of the region represented by b.
3. Each node and edge of G is represented by exactly one node or edge of the current graph.

To see why these observations hold, notice first that they hold trivially for G itself. Every node is a region by itself, and every edge represents only itself. Suppose we apply T_1 to some node n representing a region R, while the loop $n \to n$ represents some set of edges E, all of which must enter the header of R. If we add the edges E to region R, it is still a region, so after removing the edge $n \to n$, the node n represents R and the edges of E, which preserves conditions (1)-(3) above.

If we instead use T_2 to consume node b by node a, let a and b represent regions R and S respectively. Also, let E be the set of edges represented by the edge $a \to b$. We claim R, S, and E together form a region whose header is the header of R. To prove this, we must verify that the header of R dominates every node in S. If not, then there must be some path to the header of S that does not end with an edge of E. Then the last edge of this path would have to be represented in the current flow graph by some other edge entering b. But there can be no such edge, or T_2 cannot be used to consume b.

Example 10.38. The node labeled cd in Fig. 10.50(b) represents the region shown in Fig. 10.51(a), which was formed by having c consume d. Note that the edge $d \to c$ is not part of the region; in Fig. 10.50(b) that edge is represented by the loop on cd. However, in Fig. 10.50(c), the edge $cd \to cd$ has been removed, and the node cd now represents the region shown in Fig. 10.51(b).

In Fig. 10.50(d), node cd still represents the region of Fig. 10.51(b), while node ab represents the region of Fig. 10.51(c). The edge $ab \to cd$ in Fig. 10.50(d) represents the edges $a \to c$ and $b \to c$ of the original flow graph in Fig. 10.50(a). When we apply T_2 to reach Fig. 10.50(e), the remaining node represents the entire flow graph, Fig. 10.50(a). □

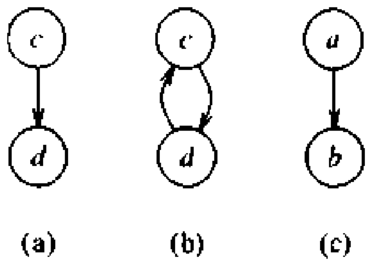

Fig. 10.51. Some regions.

We should observe that the property of T_1 and T_2 reduction mentioned above holds also for interval analysis. We leave as an exercise the fact that as we construct I(G), I(I(G)), and so on, each node in each of these graphs represents a region, and each edge a set of edges satisfying property (2) above.

Finding Dominators

We close this section with an efficient algorithm for a concept that we have used frequently, and will continue to use in developing the theory of flow graphs and data-flow analysis. We shall give a simple algorithm for computing the dominators of every node n in a flow graph, based on the principle that if $p_1, p_2, \ldots, p_k$ are all the predecessors of n, and $d \neq n$, then d *dom* n if and only if d *dom* p_i for each i. The method is akin to forward data-flow analysis with intersection as the confluence operator (e.g., available expressions), in that we take an approximation to the set of dominators of n and refine it by repeatedly visiting all the nodes in turn.

In this case, the initial approximation we choose has the initial node dominated only by the initial node, and everything dominating everything besides the initial node. Intuitively, the reason this approach works is that dominator candidates are ruled out only when we find a path that proves, say, m *dom* n is false. If we cannot find such a path, from the initial node to n avoiding m, then m really is a dominator of n.

Algorithm 10.16. Finding dominators.

Input. A flow graph G with set of nodes N, set of edges E and initial node n_0.

Output. The relation *dom*.

Method. We compute $D(n)$, the set of dominators of n, iteratively by the procedure in Fig. 10.52. At the end, d is in $D(n)$ if and only if d *dom* n. The reader may supply the details regarding how changes to $D(n)$ are detected; Algorithm 10.2 will serve as a model.

One can show that $D(n)$ computed at line (5) of Fig. 10.52 is always a subset of the current $D(n)$. Since $D(n)$ cannot get smaller indefinitely, we must eventually terminate the while-loop. A proof that, after convergence, $D(n)$ is

```
(1) $D(n_0) := \{n_0\}$;
(2) for $n$ in $N - \{n_0\}$ do $D(n) := N$;

    /* end initialization */

(3) while changes to any $D(n)$ occur do
(4)     for $n$ in $N - \{n_0\}$ do
(5)         $D(n) := \{n\} \cup \bigcap_{p \text{ a predecessor of } n} D(p)$;
```

Fig. 10.52. Dominator computing algorithm.

the set of dominators of n is left for the interested reader. The algorithm of Fig. 10.52 is quite efficient, as $D(n)$ can be represented by a bit vector and the set operations of line (5) can be done with logical **and** and **or**. □

Example 10.39. Let us return to the flow graph of Fig. 10.45, and suppose in the for-loop of line (4) nodes are visited in numerical order. Node 2 has only 1 for a predecessor, so $D(2) := \{2\} \cup D(1)$. Since 1 is the initial node, $D(1)$ was assigned $\{1\}$ at line (1). Thus, $D(2)$ is set to $\{1, 2\}$ at line (5).

Then node 3, with predecessors 1, 2, 4, and 8, is considered. Line (5) gives us $D(3) = \{3\} \cup (\{1\} \cap \{1,2\} \cap \{1,2, \ldots, 10\}) = \{1,3\}$. The remaining calculations are:

$$D(4) = \{4\} \cup (D(3) \cap D(7)) = \{4\} \cup (\{1,3\} \cap \{1,2, \ldots, 10\}) = \{1,3,4\}$$

$$D(5) = \{5\} \cup D(4) = \{5\} \cup \{1,3,4\} = \{1,3,4,5\}$$

$$D(6) = \{6\} \cup D(4) = \{6\} \cup \{1,3,4\} = \{1,3,4,6\}$$

$$\begin{aligned} D(7) &= \{7\} \cup (D(5) \cap D(6) \cap D(10)) \\ &= \{7\} \cup (\{1,3,4,5\} \cap \{1,3,4,6\} \cap \{1,2, \ldots, 10\}) = \{1,3,4,7\} \end{aligned}$$

$$D(8) = \{8\} \cup D(7) = \{8\} \cup \{1,3,4,7\} = \{1,3,4,7,8\}$$

$$D(9) = \{9\} \cup D(8) = \{9\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,9\}$$

$$D(10) = \{10\} \cup D(8) = \{10\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,10\}$$

The second pass through the while-loop is seen to produce no changes, so the above values yield the relation *dom*. □

10.10 EFFICIENT DATA-FLOW ALGORITHMS

In this section we shall consider two ways to use flow-graph theory to speed data-flow analysis. The first is an application of depth-first ordering to reduce the number of passes that the iterative algorithms of Sections 10.6 take, and the second uses intervals or the T_1 and T_2 transformations to generalize the syntax-directed approach of Section 10.5.

Depth-First Ordering in Iterative Algorithms

In all the problems studied so far, such as reaching definitions, available expressions, or live variables, any event of significance at a node will be propagated to that node along an acyclic path. For example, if a definition *d* is in *in*[*B*], then there is some acyclic path from the block containing *d* to *B* such that *d* is in the *in*'s and *out*'s all along that path. Similarly, if an expression x+y is not available at the entrance to block *B*, then there is some acyclic path that demonstrates that fact; either the path is from the initial node and includes no statement that kills or generates x+y, or the path is from a block that kills x+y and along the path there is no subsequent generation of x+y. Finally, for live variables, if x is live on exit from block *B*, then there is an acyclic path from *B* to a use of x, along which there are no definitions of x.

The reader should check that in each of these cases, paths with cycles add nothing. For example, if a use of x is reached from the end of block *B* along a path with a cycle, we can eliminate that cycle to find a shorter path along which the use of x is still reached from *B*.

If all useful information propagates along acyclic paths, we have an opportunity to tailor the order in which we visit nodes in iterative data-flow algorithms so that after relatively few passes through the nodes, we can be sure information has passed along all the acyclic paths. In particular, statistics gathered in Knuth [1971b] show that typical flow graphs have a very low *interval depth*, which is the number of times one must apply the interval partition to reach the limit flow graph; an average of 2.75 was found. Furthermore, it can be shown that the interval depth of a flow graph is never less than what we have called the "depth," the maximum number of retreating edges on any acyclic path. (If the flow graph is not reducible, the depth may depend on the depth-first spanning tree chosen.)

Recalling our discussion of the depth-first spanning tree in the previous section, we note that if $a \rightarrow b$ is an edge, then the depth-first number of *b* is less than that of *a* only when the edge is a retreating edge. Thus, replace line (5) of Fig. 10.26, which tells us to visit each block *B* of the flow graph for which we are computing reaching definitions, by:

```
for each block B in depth-first order do
```

Suppose we have a path along which a definition *d* propagates, such as

$$3 \rightarrow 5 \rightarrow 19 \rightarrow 35 \rightarrow 16 \rightarrow 23 \rightarrow 45 \rightarrow 4 \rightarrow 10 \rightarrow 17$$

where integers represent the depth-first numbers of the blocks along the path. Then the first time through the loop of lines (5)-(9) in Fig. 10.26, *d* will propagate from *out*[3] to *in*[5] to *out*[5], and so on, up to *out*[35]. It will not reach *in*[16] on that round, because as 16 precedes 35, we had already computed *in*[16] by the time *d* was put in *out*[35]. However, the next time we run through the loop of lines (5)-(9), when we compute *in*[16], *d* will be included because it is in *out*[35]. Definition *d* will also propagate to *out*[16], *in*[23], and so on, up to *out*[45], where it must wait because *in*[4] was already computed.

On the third pass, d travels to *in*[4], *out*[4], *in*[10], *out*[10], and *in*[17], so after three passes we establish that d reaches block 17.[13]

It should not be hard to extract the general principle from this example. If we use depth-first order in Fig. 10.26, then the number of passes needed to propagate any reaching definition along any acyclic path is no more than one greater than the number of edges along that path that go from a higher numbered block to a lower numbered block. Those edges are exactly the retreating edges, so the number of passes needed is one plus the depth. Of course Algorithm 10.2 does not detect the fact that all definitions have reached wherever they can reach for one more pass, so the upper bound on the number of passes taken by that algorithm with depth-first block ordering is actually two plus the depth, or 5 if we believe the results of Knuth [1971b] to be typical.

The depth-first order is also advantageous for available expressions (Algorithm 10.3), or any data flow problem that we solved by propagation in the forward direction. For problems like live variables, where we propagate backwards, the same average of five passes can be achieved if we chose the reverse of the depth-first order. Thus, we may propagate a use of a variable in block 17 backwards along the path

$$3 \rightarrow 5 \rightarrow 19 \rightarrow 35 \rightarrow 16 \rightarrow 23 \rightarrow 45 \rightarrow 4 \rightarrow 10 \rightarrow 17$$

in one pass to *in*[4], where we must wait for the next pass to in order reach *out*[45]. On the second pass it reaches *in*[16], and on the third pass it goes from *out*[35] to *out*[3]. In general, one plus the depth passes suffices to carry the use of a variable backward, along any acyclic path, if we choose the reverse of depth-first order to visit the nodes in a pass, because then, uses propagate along any decreasing sequence in a single pass.

Structure-Based Data-Flow Analysis

With a bit more effort, we can implement data-flow algorithms that visit nodes (and apply data-flow equations) no more times than the interval depth of the flow graph, and frequently the average node will be visited even fewer times than that. Whether the extra effort results in a true time savings has not been firmly established, but a technique like this one, based on interval analysis, has been used in several compilers. Further, the ideas exposed here apply to syntax-directed data-flow algorithms for all sorts of structured control statements, not just the **if** · · · **then** and **do** · · · **while** discussed in Section 10.5, and these have also appeared in several compilers.

We shall base our algorithm on the structure induced on flow graphs by the T_1 and T_2 transformations. As in Section 10.5, we are concerned with the definitions that are generated and killed as control flows through a region. Unlike the regions defined by if or while statements, a general region can have multiple exits, so for each block B in region R we shall compute sets

[13] Definition d also reaches *out*[17], but that is irrelevant to the path in question.

$gen_{R,B}$ and $kill_{R,B}$ of definitions generated and killed, respectively, along paths within the region from the header to the end of block B. These sets will be used to define a *transfer function* $trans_{R,B}(S)$ that tells for any set S of definitions, what set of definitions reach the end of block B by traveling along paths wholly within R, given that all and only the definitions in S reach the header of R.

As we have seen in Sections 10.5 and 10.6, the definitions reaching the end of block B fall into two classes.

1. Those that are generated within R and propagate to the end of B independent of S.

2. Those that are not generated in R, but that also are not killed along some path from the header of R to the end of B, and therefore are in $trans_{R,B}(S)$ if and only if they are in S.

Thus, we may write *trans* in the form:

$$trans_{R,B}(S) = gen_{R,B} \cup (S - kill_{R,B})$$

The heart of the algorithm is a way to compute $trans_{R,B}$ for progressively larger regions defined by some (T_1, T_2)-decomposition of a flow graph. For the moment, we assume that the flow graph is reducible, although a simple modification allows the algorithm to work for nonreducible graphs as well.

The basis is a region consisting of a single block, B. Here the transfer function of the region is the transfer function of the block itself, since a definition reaches the end of the block if and only if it is either generated by the block or is in the set S and not killed. That is,

$$gen_{B,B} = gen[B]$$
$$kill_{B,B} = kill[B]$$

Now, let us consider the construction of a region R by T_2; that is, R is formed when R_1 consumes R_2, as suggested in Fig. 10.53. First, note that within region R there are no edges from R_2 back to R_1 since any edge from R_2 to the header of R_1 is not a part of R. Thus any path totally within R goes (optionally) through R_1 first, then (optionally) through R_2, but cannot then return to R_1. Also note that the header of R is the header of R_1. We may conclude that within R, R_2 does not affect the transfer function of nodes in R_1; that is,

$$gen_{R,B} = gen_{R_1,B}$$
$$kill_{R,B} = kill_{R_1,B}$$

for all B in R_1.

For B in R_2, a definition can reach the end of B if any of the following conditions hold.

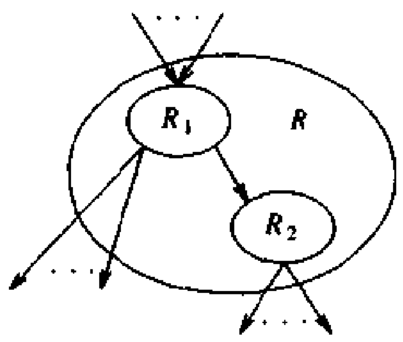

Fig. 10.53. Region building by T_2.

1. The definition is generated within R_2.
2. The definition is generated within R_1, reaches the end of some predecessor of the header of R_2, and is not killed going from the header of R_2 to B.
3. The definition is in the set S available at the header of R_1, not killed going to some predecessor of the header of R_2, and not killed going from the header of R_2 to B.

Hence, the definitions reaching the ends of those blocks in R_1 that are predecessors of the header of R_2 play a special role. In essence, we see what happens to a set S entering the header of R_1 as its definitions try to reach the header of R_2, via one of its predecessors. The set of definitions that reach one of the predecessors of the header of R_2 becomes the input set for R_2, and we apply the transfer functions for R_2 to that set.

Thus, let G be the union of $gen_{R_1,P}$ for all predecessors P of the header of R_2, and let K be the intersection of $kill_{R_1,P}$ for all those predecessors P. Then if S is the set of definitions that reach the header of R_1, the set of definitions that reach the header of R_2 along paths staying wholly within R is $G \cup (S-K)$. Therefore, the transfer function in R for those blocks B in R_2 may be computed by

$$gen_{R,B} = gen_{R_2,B} \cup (G - kill_{R_2,B})$$
$$kill_{R,B} = kill_{R_2,B} \cup (K - gen_{R_2,B})$$

Next, consider what happens when a region R is built from a region R_1 using transformation T_1. The general situation is shown in Fig. 10.54; note that R consists of R_1 plus some back edges to the header of R_1 (which is also the header of R, of course). A path going through the header twice would be cyclic and, as we argued earlier in the section, need not be considered. Thus, all definitions generated at the end of block B are generated in one of two ways.

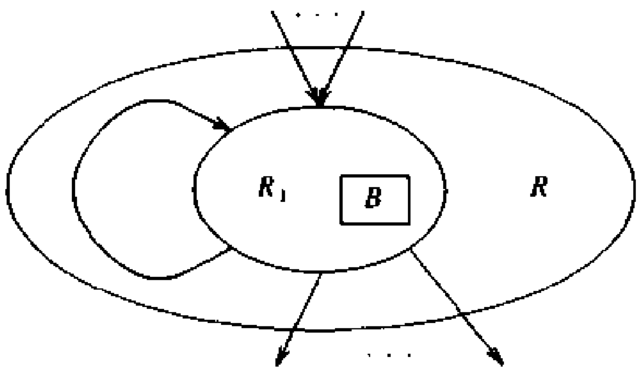

Fig. 10.54. Region building by T_1.

1. The definition is generated within R_1 and does not need the back edges incorporated into R in order to reach the end of B.
2. The definition is generated somewhere within R_1, reaches a predecessor of the header, follows one back edge, and is not killed going from the header to B.

If we let G be the union of $gen_{R_1,P}$ for all predecessors of the header in R, then

$$gen_{R,B} \;=\; gen_{R_1,B} \;\cup\; (G - kill_{R_1,B})$$

A definition is killed going from the header to B if and only if it is killed along all acyclic paths, so the back edges incorporated into R do not cause more definitions to be killed. That is,

$$kill_{R,B} \;=\; kill_{R_1,B}$$

Example 10.40. Let us reconsider the flow graph of Fig. 10.50, whose (T_1, T_2)-decomposition is shown in Fig. 10.55, with the regions of the decomposition named. We also show in Fig. 10.56 some hypothetical bit vectors representing three definitions and whether they are generated or killed by each of the blocks in Fig. 10.55.

Starting from the inside out, we note that for single-node regions, which we call A, B, C, and D, *gen* and *kill* are given by the table in Fig. 10.56. We may then proceed to region R, which is formed when C consumes D by T_2. Following the rules for T_2 above, we note that *gen* and *kill* do not change for C, that is,

$$gen_{R,C} = gen_{C,C} = 000$$
$$kill_{R,C} = kill_{C,C} = 010$$

For node D, we have to find in region C the union of *gen* for all the predecessors of the header of the region D. Of course the header of region D is node

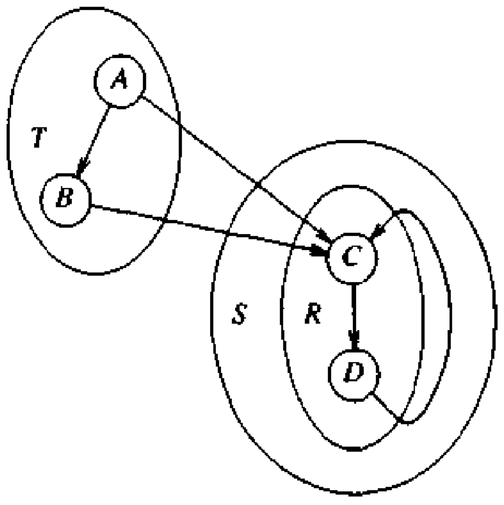

Fig. 10.55. Decomposition of a flow graph.

BLOCK	*gen*	*kill*
A	100	010
B	010	101
C	000	010
D	001	000

Fig. 10.56. *gen* and *kill* information for blocks in Fig. 10.55.

D, and there is only one predecessor of that node in region *C*, namely, the node *C*. Thus,

$$gen_{R,D} = gen_{D,D} \cup (gen_{C,C} - kill_{D,D}) = 001+(000-000) = 001$$
$$kill_{R,D} = kill_{D,D} \cup (kill_{C,C} - gen_{D,D}) = 000+(010-001) = 010$$

Now, we build region *S* from region *R* by T_1. The *kill*'s don't change, so we have

$$kill_{S,C} = kill_{R,C} = 010$$
$$kill_{S,D} = kill_{R,D} = 010$$

To compute the *gen*'s for *S* we note that the only back edge to the header of *S* that is incorporated going from *R* to *S* is the edge $D \rightarrow C$. Thus,

$$gen_{S,C} = gen_{R,C} \cup (gen_{R,D} - kill_{R,C}) = 000+(001-010) = 001$$
$$gen_{S,D} = gen_{R,D} \cup (gen_{R,D} - kill_{R,D}) = 001+(001-010) = 001$$

The computation for region *T* is analogous to that for region *R* and we obtain

$$\begin{aligned} gen_{T,A} &= 100 \\ kill_{T,A} &= 010 \\ gen_{T,B} &= 010 \\ kill_{T,B} &= 101 \end{aligned}$$

Finally, we compute *gen* and *kill* for region U, the entire flow graph. Since U is constructed when T consumes S by transformation T_2, the values of *gen* and *kill* for nodes A and B do not change from what was just given above. For C and D, we note that the header of S, node C, has two predecessors in region T, namely A and B. Therefore, we compute

$$\begin{aligned} G &= gen_{T,A} \cup gen_{T,B} = 110 \\ K &= kill_{T,A} \cap kill_{T,B} = 000 \end{aligned}$$

Then we may compute

$$\begin{aligned} gen_{U,C} &= gen_{S,C} \cup (G - kill_{S,C}) = 101 \\ kill_{U,C} &= kill_{S,C} \cup (K - gen_{S,C}) = 010 \\ gen_{U,D} &= gen_{S,D} \cup (G - kill_{S,D}) = 101 \\ kill_{U,D} &= kill_{S,D} \cup (K - gen_{S,D}) = 010 \end{aligned}$$ □

Having computed $gen_{U,B}$ and $kill_{U,B}$ for each block B, where U is the region consisting of the entire flow graph, we essentially have computed $out[B]$ for each block B. That is, if we look at the definition of $trans_{U,B}(S) = gen_{U,B} \cup (S - kill_{U,B})$, we note that $trans_{U,B}(\emptyset)$ is exactly $out[B]$. But $trans_{U,B}(\emptyset) = gen_{U,B}$. Thus, the completion of the structure-based reaching definition algorithm is to use the *gen*'s as the *out*'s, and compute the *in*'s by taking the union of the *out*'s of the predecessors. These steps are summarized in the following algorithm.

Algorithm 10.17. Structure-based reaching definitions.

Input. A reducible flow graph G and sets of definitions $gen[B]$ and $kill[B]$ for each block B of G.

Output. $in[B]$ for each block B.

Method.

1. Find the (T_1, T_2)-decomposition for G.
2. For each region R in the decomposition, from the inside out, compute $gen_{R,B}$ and $kill_{R,B}$ for each block B in R.
3. If U is the name of the region consisting of the entire graph, then for each block B, set $in[B]$ to the union, over all predecessors P of block B, of $gen_{U,P}$. □

Some Speedups to the Structure-Based Algorithm

First, notice that if we have a transfer function $G \cup (S - K)$, the function is not changed if we delete from K some members of G. Thus, when we apply T_2, instead of using the formulas

$$gen_{R,B} = gen_{R_2,B} \cup (G - kill_{R_2,B})$$

$$kill_{R,B} = kill_{R_2,B} \cup (K - gen_{R_2,B})$$

we can replace the second by

$$kill_{R,B} = kill_{R_2,B} \cup K$$

thus saving an operation for each block in region R_2.

Another useful idea is to notice that the only time we apply T_1 is after we have first consumed some region R_2 by R_1, and there are some back edges from R_2 to the header of R_1. Instead of first making changes in R_2 because of the T_2 operation, and then making changes in R_1 and R_2 due to the T_1 operation, we can combine the two sets of changes if we do the following.

1. Using the T_2 rule, compute the new transfer function for those nodes in R_2 that are predecessors of the header of R_1.
2. Using the T_1 rule, compute the new transfer function for all the nodes of R_1.
3. Using the T_2 rule, compute the new transfer function for all the nodes of R_2. Note that feedback due to the application of T_1 has reached the predecessors of R_2 and is passed to all of R_2 by the T_2 rule; there is no need to apply the T_1 rule for R_2.

Handling Nonreducible Flow Graphs

If the (T_1, T_2)-reduction of a flow graph stops at a limit flow graph that is not a single node, then we must perform a node splitting. Splitting a node of the limit flow graph corresponds to duplicating the entire region represented by that node. For example, in Fig. 10.57 we suggest the effect that node splitting might have on an original nine-node flow graph that was partitioned by T_1 and T_2 into three regions connected by some edges.

As mentioned in the previous section, by alternating splits with sequences of reductions, we are guaranteed to reduce the flow graph to a single node. The result of the splits is that some of the nodes of the original graph will have more than one copy in the region represented by the one-node graph. We may apply Algorithm 10.17 to this region with little change. The only difference is that when we split a node, the *gen*'s and *kill*'s for the nodes of the original graph in the region represented by the split node must be duplicated. For example, whatever the values of *gen* and *kill* are for the nodes in the two-node region of Fig. 10.57 on the left become *gen* and *kill* for each of the corresponding nodes in both two-node regions on the right. At the final step,

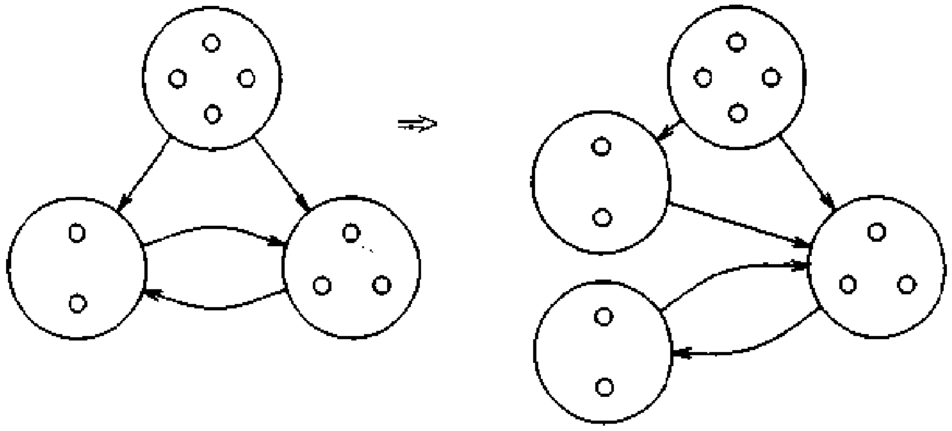
Fig. 10.57. Splitting a nonreducible flow graph.

when we compute the *in*'s for all the nodes, those nodes of the original graph that have several representatives in the final region have their *in*'s computed by taking the union of the *in*'s of all their representatives.

In the worst case, splitting of nodes could exponentiate the total number of nodes represented by all the regions. Thus, if we expect many flow graphs to be nonreducible, we probably should not use structure-based methods. Fortunately, nonreducible flow graphs are sufficiently rare that we can generally ignore the cost of node splitting.

10.11 A TOOL FOR DATA-FLOW ANALYSIS

As we have pointed out before, there are strong similarities among the various data-flow problems studied. The data-flow equations of Section 10.6 were seen to be distinguished by:

1. The transfer function used, which in each case studied was of the form $f(X) = A \cup (X - B)$. For example, A = *kill* and B = *gen* for reaching definitions.
2. The confluence operator, which in all cases so far has been either union or intersection.
3. The direction of propagation of information: forward or backward.

Since these distinctions are not great, it should not be surprising that all these problems can be treated in a unified way. Such an approach was described in Kildall [1973], and a tool to simplify the implementation of data-flow problems was implemented by Kildall and used by him in several compiler projects. It has not seen widespread use, probably because the amount of labor saved by the system is not as great as that saved by tools like parser generators. However, we should be aware of what can be done not only because it does suggest a simplification for implementers of optimizing