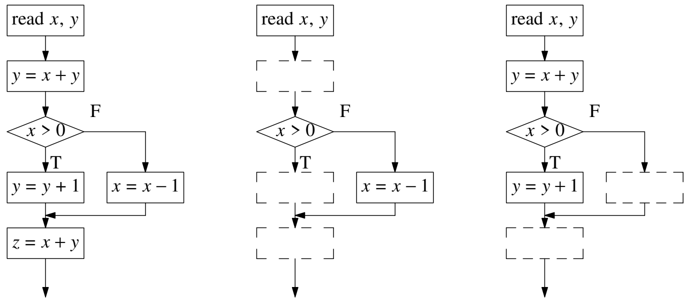
PROGRAM SLICE

Program Slice: A *program slice* for a variable x at a statement s in a program P is a subprogram $P' = P^+(x, s)$ of P which produces the same sequence of computations that affect the value of x *after* the execution of s in P for all inputs. (Here, P' is obtained by deleting parts of P, say, deleting nodes in the flowchart for P, but not modifying P otherwise for contents of other nodes.)

Example 1. Consider the flowchart form of P to avoid the issues of programming language syntax. If "read x, y" is decomposed into two parts "read x" and "read y", then "read y" will not be part of the slice for x in (iii), but "read x" will still be part of the slice for y in (iv).

1. read x and y
2. $y = x + y$
3. if $(x > 0)$ then $y = y + 1$
4. else $x = x - 1$
5. $z = x + y$
6. ⋯

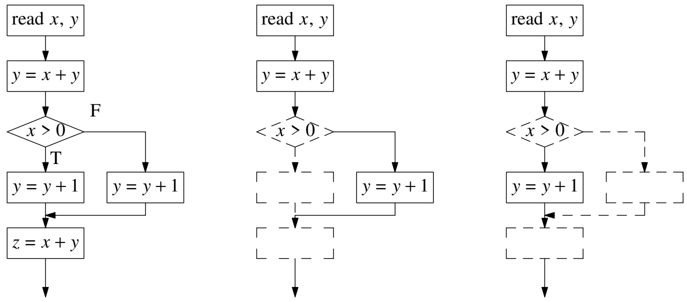


(i) A code-segment. (ii) The flowchart for (i) and a slice $P^+(z, 5)$. (iii) A slice $P^+(x, 5)$. (iv) A slice $P^+(y, 5)$.

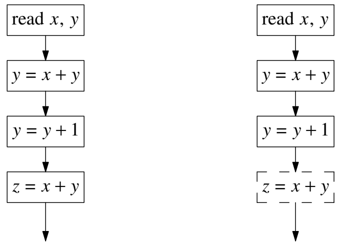
Question: What would be the benefit of a definition of slice where "after the execution of s" is replaced by "before the execution of s" (which may be denoted by $P^-(x, s)$)?

Example 2. Deletion of different parts of P may lead to the same resulting subprogram P' – a pathological example.

1. read x and y
2. $y = x + y$
3. if $(x > 0)$ then $y = y + 1$
4. else $y = y + 1$
5. $z = x + y$
6. ⋯



(i) A code-segment. (ii) The flowchart for (i). (iii) A slice $P^+(y, 5)$. (iv) Another slice $P^+(y, 5)$.



(v) An equivalent flowchart for (ii). (vi) The unique slice for y at the same place.

THEORETICAL DIFFICULTY IN GETTING AN EQUIVALENT MINIMAL SUBPROGRAM

Simple cases are doable:

```
...                                                          ...
z = y+2;                                                     z = y+2;
if (x > 0) then y = y+1;         is equivalent to            y = y+1;
else y = (y+z)/2;
...                                                          ...
```

General case is not possible (because of inherent mathematical difficulty):

```
if (x > 0)
   for (i=1; i<=2*n; i=i+2) //compute sum = C_1^{2n} + C_3^{2n} + ... + C_{2n-1}^{2n}
       ...
else for (i=0; i<=2*n; i=i+2) //compute sum = C_0^{2n} + C_2^{2n} + ... + C_{2n}^{2n}
         ...
```

is equivalent to

```
for (i=1; i<=2*n; i=i+2) //compute sum = C_1^{2n} + C_3^{2n} + ... + C_{2n-1}^{2n}
    ...
```

What we can compute is the union of all minimal subprograms P' each of which is a slice for x at s.

Def(s): This is the set of variables defined (i.e., a value is assigned to them, directly or via a side-effect) in a given statement s.

Use(s): This is the set of variables used in a given statement s.

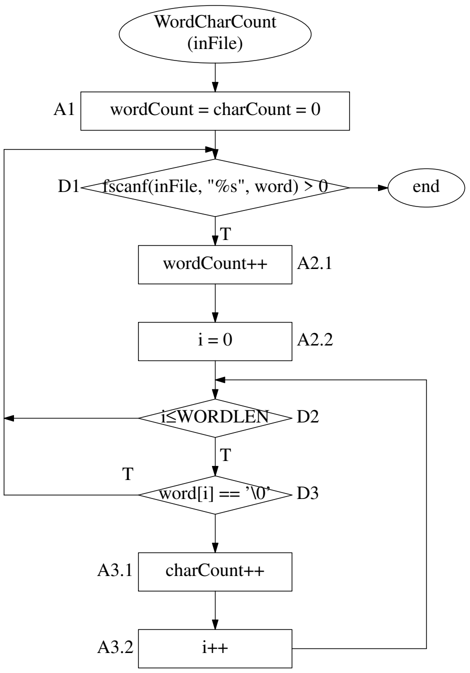
Example: s: $i = 0$; Def(s) $= \{i\}$, Use(s) $= \varnothing$
s: $i++$; Def(s) $= \{i\}$, Use(s) $= \{i\}$

Theorem: If $s' \in P^+(x, s)$

DEFINITION, USE, AND REACHING-RELATION

Example. Shown below is a C-function and its flowchart.

```
//scans inFile and sets values for global variables wordCount and charCount;
//blanks and tabs are not counted in charCount
void WordCharCount(FILE *inFile)
{int i;
 char word[WORDLEN+1];
 wordCount = charCount = 0;
 while (fscanf(inFile, "%s", word) > 0) {
       wordCount++;
       for (i=0; i<=WORDLEN; i++)
           if (word[i] == '\0') break;
           else charCount++;
       //printf("wordCount = %d, charCount = %d0, wordCount, charCount);
 }
}
```



The (unstructured) flowchart of the code.

Reaching-relation: A definition of the variable x at a node s is said to *reach* a node s' (which may equal s) if there is an ss'-path with one or more arcs such that x is not defined on any node on the path in between s and s'.

Notation: $(x, s, s') \in$ Reaching-relation.

We say $(x, s, s') \in$ DefUse-relation if $(x, s, s') \in$ Reaching-relation and $x \in$ Use(s').

Node s	Def(s)	Use(s)	Nodes s' such that $(x, s, s') \in$ Reaching-relation
Start	inFile	–	A1, D1
A1	wordCount,	charCount	A2.1, D1, end
	charCount		A2.1, A2.2, A3.1, D1, D2, D3, end
D1	word,	inFile	A2.1, A2.2, A3.1, A3.2, D1, D2, D3, end
	inFile		A2.1, A2.2, A3.1, A3.2, D1, D2, D3, end
A2.1	wordCount	wordCount	A2.1, A2.2, A3.1, A3.2, D1, D2, D3, end
A2.2	i	–	A2.1, A2.2, A3.1, A3.2, D1, D2, D3, end
D2	–	i	
D3	–	i, word	
A3.1	charCount	charCount	A2.1, A2.2, A3.1, A3.2, D1, D2, D3, end
A3.2	i	i	A2.1, A2.2, A3.1, A3.2, D1, D2, D3, end

COMPUTING A SLICE

Two Types of Nodes in $P^+(x, s)$:

- Nodes that have a direct role, i.e., which defines one or more variables that are used at other nodes that are already in the slice.
- Branch-nodes which dominate one or more nodes that are already in the slice.

Key Idea:

- Use a back-propagation algorithm to connect definition of variables to their uses, starting with the variables used at s if x defined at s or s is a branch-node.

Implementation:

- For each node s, keep a list $L(s)$ of variables x such that Slice(x, s) has been called to prevent repeated calls for the same $\langle x, s\rangle$ pair. Initially, each $L(s)$ is empty (make these lists static-local in the function Slice($\cdot$)). A node s is considered marked if $L(s) \neq$ empty.
- The top-level recursion (i.e., the initial call) begins by computing the Reaching-relation, DefUse-relation, and the domination-relation. It ends by computing $P^+(x, s) = \{s'\colon L(s') \neq \text{empty}\}$. (The control-flow among the nodes in the slice is simply that in the original flowchart. This is obtained by simply pretending "going through" the nodes that are not in the slice.)

Algorithm Slice(x, s):

Input: The control-flow graph of program, which may be unstructured.
Output: $P^+(x, s)$.

1. If (($x \notin$ Def(s)) and (s is not a branch-node)), //this can happen only in the initial-call
   then for each $(x, s', s) \in$ Reaching-relation, call Slice(x, s').
   else (($x \in$ Def(s)) or (s is a branch-node)), then add x to $L(s)$ and for each $(y, s', s) \in$ DefUse-relationship, call Slice(y, s') provided $y \notin L(s')$.
2. If (there are branch-nodes, i.e., if-nodes or loop-test nodes s'' which are unmarked and which dominate one or more marked nodes), then call Slice(?, s'') for each such s'', where "?" represents a fictitious variable name not defined or used in P.
3. Go back to step (2) if some s'' is were found.

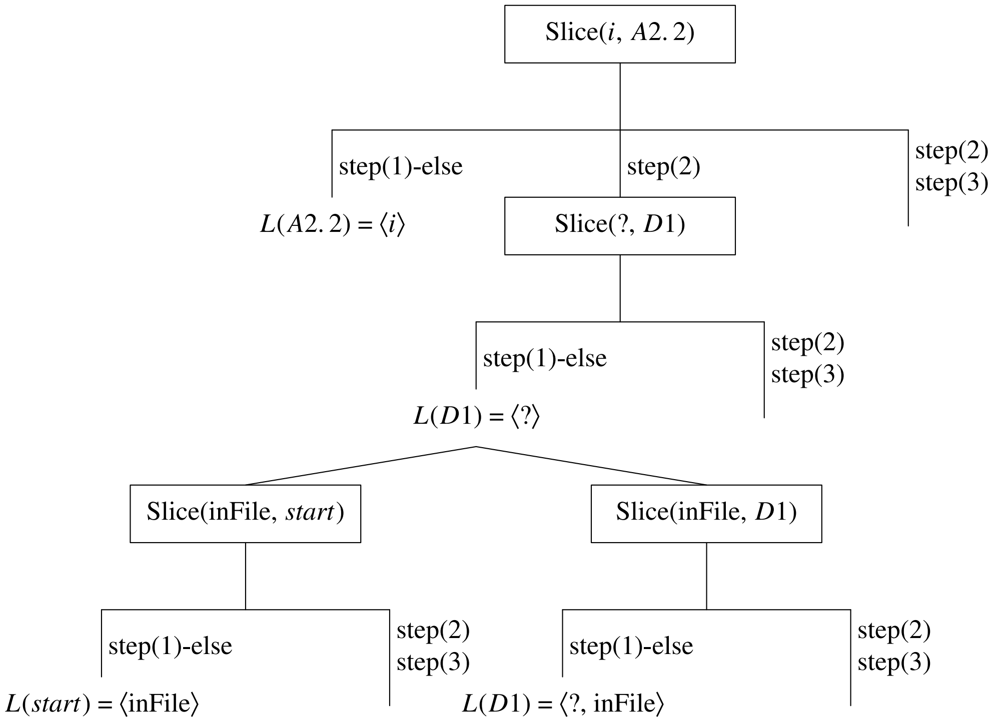
Note: Some slight modification is needed if s corresponds to a break-node. (See Exercises next page.)

If $s' \in P^+(x, s)$, then $P^-(y, s') \subseteq P^+(x, s)$ for all $y \in$ Use(s').

EXERCISE

1. A good coding practice is to filter out as much as possible the common parts of "then" and "else" components of an if-statement (either before the if-statement or after the if-statement as the case may be). Explain with an example the beneficial impact of this on a slice $P^+(x, s)$.
2. Another good coding practice is to eliminate an and/or component of a loop-test condition if none of the variables in that component is modified in the loop-body. What is its impact on $P^+(x, s)$?
3. Yet another good coding practice is to make sure that every variable definition is used somewhere. What is the role of such a definition in $P^-(x, \text{end})$ for any x?

ILLUSTRATION OF SLICE-ALGORITHM



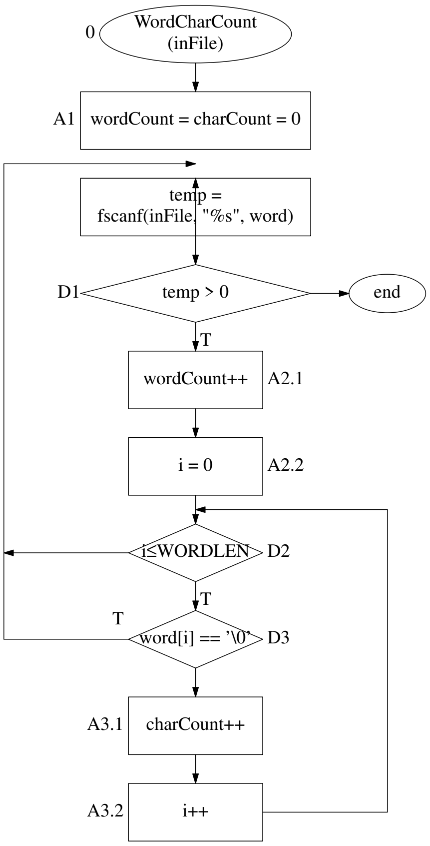
Slice(i, A2.2) = {A2.2, D1, start}; this does not correspond to a part of the program as such because A2.2 is not a statement in itself.

EXERCISE

1. Show the tree of calls generated in computing the slice $P^+(word, D3)$ using the flowchart of Word-CharCount-function. (You do not need to show the details of all intermediate steps statement by statement - just indicate which calls generate which calls.)
2. Suppose we add the break-node A3.0 immediately to the left of D3 in the flowchart and replace the true-branch (D3, D1) at D3 by the arcs (D3, A3.0) and (A3.0, D1). Note that a break-node s does not define or use any variable. Should the slice $P^+(word, D3)$ be any different now? Should slice $P^+(word, A3.0)$ differ from $P^+(word, D3)$?
3. There is a small change needed in the algorithm for computing $P^+(x, s)$ when s corresponds to a break-node. Describe the change and illustrate the need for the change.

THE NORMALIZED FORM OF A FLOWCHART AND ITS IMPACT ON SLICES

- A branch-node does not assign value to a variable as a side effect.
- An action-node may define values of more than one variable (e.g., as in a function-call).



The normalized form of the flowchart for WordCharCount-function.

- The normalization of a structured flowchart may make it unstructured.

EXERCISE

1. Is it true that a slice $P^{+}(x, s)$ for the above normalized flowchart is basically the same as that for the original flowchart of WordCharCount-function, when s corresponds to a node in the original flowchart? Explain with an example.
2. Is the computation of a slice for a normalized flowchart easier in some way than a non-normalized flowchart?