Module 08: CS31003: Compilers

Control Flow Graph and Local Optimization

Partha Pratim Das & Pralay Mitra

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in ; pralay@cse.iitkgp.ac.in

September 27 & October 04, 2021

Module Objectives

- What is code optimization and why is it needed?
- Types of optimizations
- Basic blocks and control flow graphs
- Local optimizations
- Building a control flow graph
- Directed acyclic graphs and value numbering

Module Outline

Machine-independent Code Optimization

- Intermediate code generation process introduces many inefficiencies
  - Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.
- Code optimization removes such inefficiencies and improves code
- Improvement may be time, space, or power consumption
- It changes the structure of programs, sometimes of beyond recognition
  - Inlines functions, unrolls loops, eliminates some programmer-defined variables, etc.
- Code optimization consists of a bunch of heuristics and percentage of improvement depends on programs (may be zero also)
- Optimizations may be classified as *local* and *global*

Example: Vector Product

Module 08

Partha Pratim Das & Pralay Mitra


```
int a[5], b[5], c[5];
int i, n = 5;

for(i = 0; i < n; i++) {
    if (a[i] < b[i])
        c[i] = a[i] * b[i];
    else
        c[i] = 0;
}
return;
```

```
// int i, n = 5;
100: t1 = 5
101: n = t1
// for(i = 0; i < n; i++) {
102: t2 = 0
103: i = t2
104: if i < n goto 109 // T
105: goto 129 // F
106: t3 = i
107: i = i + 1
108: goto 104
// if (a[i] < b[i])
109: t4 = 4 * i
110: t5 = a[t4]
111: t6 = 4 * i
112: t7 = b[t6]
113: if t5 < t7 goto 115 // T
114: goto 124 // F
```

```
// c[i] = a[i] * b[i];
115: t8 = 4 * i
116: t9 = c + t8
117: t10 = 4 * i
118: t11 = a[t10]
119: t12 = 4 * i
120: t13 = b[t12]
121: t14 = t11 * t13
122: *t9 = t14
123: goto 106 // next
// c[i] = 0;
124: t15 = 4 * i
125: t16 = c + t15
126: t17 = 0
127: *t16 = t17
// }
128: goto 106 // for
// return;
129: return
```

Peep-hole Optimization

- Eliminating redundant instructions
- Eliminating local def-use of temporary
- Eliminating unreachable code
- Eliminating jumps over jumps
- Algebraic simplifications
- Strength reduction

Local Optimization

Local optimization: within basic blocks

- Local Common Sub-Expression (LCSE) elimination
- Constant propagation and constant folding
- Eliminating local def-use of temporary
- Dead-code elimination
- Reordering computations using algebraic laws
- Eliminating redundant instructions

Global Optimization

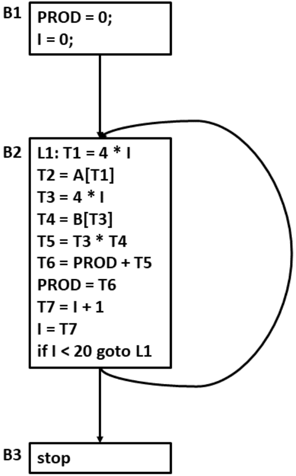
Global optimization: on whole procedures/programs

- Global Common Sub-Expression (GCSE) elimination
- Constant propagation and constant folding
- Eliminating unreachable code
- Eliminating jumps over jumps
- Eliminating jumps to jumps (chain of jumps)
- Eliminating def-use of temporary
- Eliminating redundant instructions
- Loop invariant code motion
- Partial redundancy elimination
- Loop unrolling and function inlining
- Vectorization and Concurrentization

Basic Blocks and Control-Flow Graphs

- **Basic Blocks** (BB) are sequences of intermediate code with a *single entry* and a *single exit*
- **Control Flow Graphs** (CFG) show control flow among basic blocks
- Basic blocks are represented as **Directed Acyclic Graphs** (DAGs), which are in turn represented using the value-numbering method applied on quadruples
- Optimizations on basic blocks

Example of Basic Blocks and Control Flow Graph




```
PROD = 0;
I = 0;
do {
   PROD = PROD + A[I] * B[I];
   ++I;
} while (I < 20);
```

```
PROD = 0;
I = 0;
L1: T1 = 4 * I
T2 = A[T1]
T3 = 4 * I
T4 = B[T3]
T5 = T3 * T4
T6 = PROD + T5
PROD = T6
T7 = I + 1
I = T7
if I < 20 goto L1
stop
```

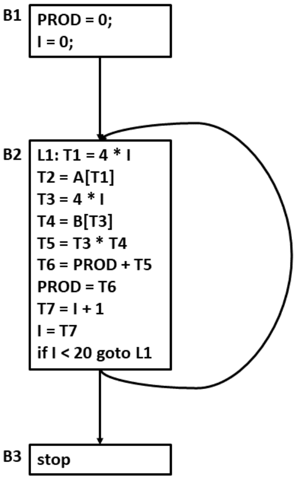
Algorithm for Partitioning into Basic Blocks

[1] Determine the set of *leaders*, the first statements of basic blocks
- The first statement is a leader
- Any statement which is the target of a conditional or unconditional *goto* is a leader
- Any statement which immediately follows a *conditional goto* is a leader

[2] A leader and all statements which follow it upto but not including the next leader (or the end of the procedure), is the basic block corresponding to that leader

[3] Any statements, not placed in a block, can never be executed, and may now be removed, if desired

Example of Basic Blocks and CFG




```
PROD = 0;
I = 0;
do {
    PROD = PROD + A[I] * B[I];
    ++I;
} while (I < 20);
```

```
PROD = 0;
I = 0;
L1: T1 = 4 * I
T2 = A[T1]
T3 = 4 * I
T4 = B[T3]
T5 = T3 * T4
T6 = PROD + T5
PROD = T6
T7 = I + 1
I = T7
if I < 20 goto L1
stop
```

Control Flow Graph

- The nodes of the CFG are basic blocks
- One node is distinguished as the initial node
- There is a directed edge $B1 \longrightarrow B2$, if B2 can immediately follow B1 in some execution sequence:
  - There is a conditional or unconditional jump from the last statement of B1 to the first statement of B2, or
  - B2 immediately follows B1 in the order of the program, and B1 does not end in an unconditional jump
- A basic block is represented as a record consisting of
  [1] a count of the number of quads in the block
  [2] a pointer to the leader of the block
  [3] pointers to the predecessors of the block
  [4] pointers to the successors of the block

Note: *Jump statements point to basic blocks and not* quads *so as to make code movement easy*

Example: Vector Product: Control Flow Graph

[1] **First** quad **of the program**

[2] quad**'s as target of some** goto

[3] quad**'s following a conditional** goto

```
100: n = 5                [1]
101: i = 0
102: if i < n goto 106    [2]
103: goto 124             [3]
104: i = i + 1            [2]
105: goto 102
106: t4 = 4 * i           [2]
107: t5 = a[t4]
108: t6 = 4 * i
109: t7 = b[t6]
110: if t5 >= t7 goto 120
111: t8 = 4 * i           [3]
112: t9 = c + t8
113: t10 = 4 * i
114: t11 = a[t10]
115: t12 = 4 * i
116: t13 = b[t12]
117: t14 = t11 * t13
118: *t9 = t14
119: goto 104
120: t15 = 4 * i          [2]
121: t16 = c + t15
122: *t16 = 0
123: goto 104
124: return               [2]
```

Example: Vector Product: Control Flow Graph

Control Flow Graph is shown below:

```
// Block B1
0: 100: n = 5
1: 101: i = 0
 :     : goto B2 [Fall through]

// Block B2
0: 102: if i < n goto B4 [106]
 : 103: goto B7 [124]

// Block B3
0: 104: i = i + 1
 : 105: goto B2 [102]

// Block B4
0: 106: t4 = 4 * i
1: 107: t5 = a[t4]
2: 108: t6 = 4 * i
3: 109: t7 = b[t6]
4: 110: if t5 >= t7 goto B6 [120]
 :     : goto B5 [Fall through]
```

```
// Block B5
0: 111: t8 = 4 * i
1: 112: t9 = c + t8
2: 113: t10 = 4 * i
3: 114: t11 = a[t10]
4: 115: t12 = 4 * i
5: 116: t13 = b[t12]
6: 117: t14 = t11 * t13
7: 118: *t9 = t14
 : 119: goto B3 [104]

// Block B6
0: 120: t15 = 4 * i
1: 121: t16 = c + t15
2: 122: *t16 = 0
 : 123: goto B3 [104]

// Block B7
0: 124: return
```

There is no unreachable quad to remove.

Example: Vector Product: Control Flow Graph: Graphical Depiction

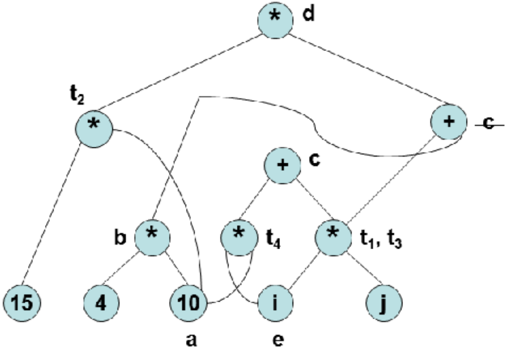
```
// Block B1
0: n = 5
1: i = 0
: goto B2

// Block B2
0: if i < n goto B4
: goto B7

// Block B7
0: return

// Block B4
0: t4 = 4 * i
1: t5 = a[t4]
2: t6 = 4 * i
3: t7 = b[t6]
4: if t5 >= t7 goto B6
 : goto B5

// Block B5
0: t8 = 4 * i
1: t9 = c + t8
2: t10 = 4 * i
3: t11 = a[t10]
4: t12 = 4 * i
5: t13 = b[t12]
6: t14 = t11 * t13
7: *t9 = t14
: goto B3

// Block B6
0: t15 = 4 * i
1: t16 = c + t15
2: *t16 = 0
 : goto B3

// Block B3
0: i = i + 1
 : goto B2
```

Optimization of Basic Blocks
Directed Acyclic Graph (DAG) Representation

```
1: a = 10
2: b = 4 * a
3: t1 = i * j
4: c = t1 + b
5: t2 = 15 * a
6: d = t2 * c
7: e = i
8: t3 = e * j
9: t4 = i * a
10: c = t3 + t4
```

Optimization of Basic Blocks
Directed Acyclic Graph (DAG) Representation

1: a = 10
2: b = 4 * a
3: t1 = i * j
4: c = t1 + b
5: t2 = 15 * a
6: d = t2 * c
7: e = i
8: t3 = e * j
9: t4 = i * a
10: c = t3 + t4



Value Numbering in Basic Blocks

- A simple way to represent DAGs is via *value-numbering*
- While searching DAGs represented using pointers etc., is inefficient, *value-numbering* uses hash tables and hence is very efficient
- Central idea is to assign numbers (called value numbers) to expressions in such a way that two expressions receive the same number if the compiler can prove that they are equal for all possible program inputs
- We assume quadruples with binary or unary operators
- The algorithm uses three tables indexed by appropriate hash values: *HashTable*, *ValnumTable*, and *NameTable*

Value Numbering in Basic Blocks

- Can be used to eliminate common sub-expressions, do constant folding, and constant propagation in basic blocks
- Can take advantage of commutativity of operators, addition of zero, and multiplication by one

Data Structures for Value Numbering

In the field *Namelist*, first name is the defining occurrence and replaces all other names with the same value number with itself (or its constant value)

ValueNumber Table (VNT) Entry

Indexed by Name Hash Value

Name	Value Number

Name Table (NT) Entry

Indexed by Value Number

Name List	Constant Value	Constant Flag

Hash Table (HT) Entry

Indexed by Expression Hash Value

Expression	Value Number

Example of Value Numbering

<table>
<tr><th>HLL Program</th><th>Quad's before value-numbering</th><th>Quad's after value-numbering</th></tr>
<tr><td><pre>a = 10
b = 4 * a
c = i * j + b
d = 15 * a * c
e = i
c = e * j + i * a</pre></td><td><pre>01. a = 10
02. b = 4 * a
03. t1 = i * j
04. c = t1 + b
05. t2 = 15 * a
06. d = t2 * c
07. e = i
08. t3 = e * j
09. t4 = i * a
10. c = t3 + t4</pre></td><td><pre>a = 10
b = 40
t1 = i * j
c = t1 + 40
t2 = 150
d = 150 * c
e = i
t3 = i * j
t4 = i * 10
c = t1 + t4</pre>Quad's 5 & 8 can be deleted</td></tr>
</table>

Example of Value Numbering

VN Table

Name	VN
a	1
b	2
i	3
j	4
t1	5
c	6, 10
t2	7
d	8
e	3
t3	5
t4	9

Name Table

Index	Name	Val
1	a	10
2	b	40
3	i, e	
4	j	
5	t1, t3	
6	c	
7	t2	150
8	d	
9	t4	
10	c	

Hash Table

Expr	VN
i * j	5
t1 + 40	6
150 * c	8
i * 10	9
t1 + t4	10

01. a = 10	a = 10
02. b = 4 * a	b = 40
03. t1 = i * j	t1 = i * j
04. c = t1 + b	c = t1 + 40
05. t2 = 15 * a	t2 = 150
06. d = t2 * c	d = 150 * c
07. e = i	e = i
08. t3 = e * j	t3 = i * j
09. t4 = i * a	t4 = i * 10
10. c = t3 + t4	c = t1 + t4

Running the algorithm through the example (1)

[1] $a = 10$:

- a is entered into *ValnumTable* (with a *vn* of 1, say) and into *NameTable* (with a constant value of 10)

[2] $b = 4 * a$:

- a is found in *ValnumTable*, its constant value is 10 in *NameTable*
  - We have performed *constant propagation*
  - $4 * a$ is evaluated to 40, and the quad is rewritten
  - We have now performed *constant folding*
  - b is entered into *ValnumTable* (with a *vn* of 2) and into *NameTable* (with a constant value of 40)

[3] $t1 = i * j$:

- i and j are entered into the two tables with new *vn* (as above), but with no constant value
- $i * j$ is entered into *HashTable* with a new *vn*
- $t1$ is entered into *ValnumTable* with the same *vn* as $i * j$

Running the algorithm through the example (2)

[4] Similar actions continue till $e = i$

- e gets the same vn as i

[5] $t3 = e * j$:

- e and i have the same vn
- hence, $e * j$ is detected to be the same as $i * j$
- since $i * j$ is already in the HashTable, we have found a *common subexpression*
- from now on, all uses of $t3$ can be replaced by $t1$
- quad $t3 = e * j$ can be deleted

[6] $c = t3 + t4$:

- $t3$ and $t4$ already exist and have vn
- $t3 + t4$ is entered into *HashTable* with a new vn
- this is a reassignment to c
- c gets a different vn, same as that of $t3 + t4$

[7] Quads are renumbered after deletions

Example of Value Numbering

If the same code snippet is translated by our automated scheme, we shall get a more verbose 3 address code. Here we show that this auto-translated code too gets optimized to the same as before

HLL Program	Quad's before value-numbering	Quad's after value-numbering
`a = 10`	`01. a = 10`	`01. a = 10`
`b = 4 * a`	`02. t1 = 4 * a`	`02. t1 = 40`
`c = i * j + b`	`03. b = t1`	`03. b = 40`
`d = 15 * a * c`	`04. t2 = i * j`	`04. t2 = i * j`
`e = i`	`05. t3 = t2 + b`	`05. t3 = t2 + 40`
`c = e * j + i * a`	`06. c = t3`	`06. c = t3`
	`07. t4 = 15 * a`	`07. t4 = 150`
	`08. t5 = t4 * c`	`08. t5 = 150 * t3`
	`09. d = t5`	`09. d = t5`
	`10. e = i`	`10. e = i`
	`11. t6 = e * j`	`11. t6 = i * j`
	`12. t7 = i * a`	`12. t7 = i * 10`
	`13. t8 = t6 + t7`	`13. t8 = t2 + t7`
	`14. c = t8`	`14. c = t8`

- Quad's 2, 6, 7 & 11 can be deleted
- Copy can be propagated (in reverse) to eliminate `t5` (between 8 & 9) and `t8` (between 13 & 14)
- Note that `e` in 10 cannot be removed as it may be used outside the block

Example of Value Numbering

VN Table

Name	VN
a	1
t1	2
b	2
i	3
j	4
t2	5
t3	6
c	6
t4	7
t5	8
d	8
e	3
t6	5
t7	9
t8	10
c	10

Hash Table

Expr	VN
i * j	5
t2 + 40	6
150 * t3	8
i * 10	9
t6 + t7	10

Name Table

Index	Name	Val
1	a	10
2	t1, b	40
3	i, e	
4	j	
5	t2, t6	
6	t3, c	
7	t4	150
8	t5, d	
9	t7	
10	t8, c	

Example: Vector Product: LCSE

We need to perform LCSE step for blocks:

```
B4:
// if (a[i] < b[i]) {
// Block B4
0: t4 = 4 * i
1: t5 = a[t4]
2: t6 = 4 * i
3: t7 = b[t6]
4: if t5 >= t7 goto B6
 : goto B5
```

and

```
B5:
// c[i] = a[i] * b[i];
// Block B5
0: t8 = 4 * i
1: t9 = c + t8
2: t10 = 4 * i
3: t11 = a[t10]
4: t12 = 4 * i
5: t13 = b[t12]
6: t14 = t11 * t13
7: *t9 = t14
 : goto B3
```

Example: Vector Product: LCSE (Block B4)

VN Table

Name	VN
i	1
t4	2
t5	3
t6	2
t7	4

Hash Table

Expr	VN
4 * i	2
a[t4]	3
b[t6]	2

Name Table

Index	Name	Val	Flag
1	i	?	No
2	t4, t6	?	No
3	t5	?	No
4	t7	?	No

Input:

```
// Block B4
0: t4 = 4 * i
1: t5 = a[t4]
2: t6 = 4 * i
3: t7 = b[t6]
4: if t5 >= t7 goto B6
 : goto B5
```

After LCSE:

```
// Block B4
0: t4 = 4 * i
1: t5 = a[t4]
2: t6 = t4 XXX
3: t7 = b[t4]
4: if t5 >= t7 goto B6
 : goto B5
```

After removal of useless quad's:

```
// Block B4
0: t4 = 4 * i
1: t5 = a[t4]
2: t7 = b[t4]
3: if t5 >= t7 goto B6
 : goto B5
```

Example: Vector Product: LCSE (Block B5)

Module 08

Partha Pratim Das & Pralay Mitra

Objectives & Outline

Optimization Issues

Basic Block

Control Flow Graph

Value Numbering

Extensional Handling

Extended Basic Blocks

VN Table

Name	VN
i	1
t8	2
t9	3
t10	2
t11	4
t12	2
t13	5
t14	6

Hash Table

Expr	VN
4 * i	2
t11 * t13	6

Name Table

Index	Name	Val	Flag
1	i	?	No
2	t8, t10, t12	?	No
3	t9	?	No
4	t11	?	No
5	t13	?	No
6	t14	?	No

Input:

```
// Block B5
0: t8 = 4 * i
1: t9 = c + t8
2: t10 = 4 * i
3: t11 = a[t10]
4: t12 = 4 * i
5: t13 = b[t12]
6: t14 = t11 * t13
7: *t9 = t14
 : goto B3
```

After LCSE:

```
// Block B5
0: t8 = 4 * i
1: t9 = c + t8
2: t10 = t8   XXX
3: t11 = a[t8]
4: t12 = t8   XXX
5: t13 = b[t8]
6: t14 = t11 * t13
7: *t9 = t14
 : goto B3
```

After removal of useless quad's:

```
// Block B5
0: t8 = 4 * i
1: t9 = c + t8
2: t11 = a[t8]
3: t13 = b[t8]
4: t14 = t11 * t13
5: *t9 = t14
 : goto B3
```

Example: Vector Product: CFG after LCSE

```
// Block B1
0: n = 5
1: i = 0
 : goto B2

// Block B2
0: if i < n goto B4
 : goto B7

// Block B3
0: i = i + 1
 : goto B2

// Block B4
0: t4 = 4 * i
1: t5 = a[t4]
2: t7 = b[t4]
3: if t5 >= t7 goto B6
 : goto B5
```

```
// Block B5
0: t8 = 4 * i
1: t9 = c + t8
2: t11 = a[t8]
3: t13 = b[t8]
4: t14 = t11 * t13
5: *t9 = t14
 : goto B3

// Block B6
0: t15 = 4 * i
1: t16 = c + t15
2: *t16 = 0
 : goto B3

// Block B7
0: return
```

Handling Commutativity etc.

- When a search for an expression $i + j$ in *HashTable* fails, try for $j + i$
- If there is a quad $x = i + 0$, replace it with $x = i$
- Any quad of the type, $y = j * 1$ can be replaced with $y = j$
- After the above two types of replacements, value numbers of x and y become the same as those of i and j, respectively
- Quads whose LHS variables are used later can be marked as *useful*
- All unmarked quads can be deleted at the end

Handling Array References

Consider the sequence of quads:

[1] $X = A[i]$

[2] $A[j] = Y$: i and j could be the same

[3] $Z = A[i]$: in which case, $A[i]$ is not a common subexpression here

- The above sequence cannot be replaced by: $X = A[i]$; $A[j] = Y$; $Z = X$
- When $A[j] = Y$ is processed during value numbering, ALL references to array A so far are searched in the tables and are marked KILLED - this kills quad 1 above
- When processing $Z = A[i]$, killed quads not used for CSE
- Fresh table entries are made for $Z = A[i]$
- However, if we know apriori that $i \neq j$, then $A[i]$ can be used for CSE

Handling Pointer References

Consider the sequence of quads:

[1] $X = *p$

[2] $*q = Y$: p and q could be pointing to the same object

[3] $Z = *p$: in which case, $*p$ is not a common sub-expression here

- The above sequence cannot be replaced by: $X = *p;\ *q = Y;\ Z = X$
- Suppose no pointer analysis has been carried out
  - p and q can point to *any* object in the basic block
  - Hence, When $*q = Y$ is processed during value numbering, ALL table entries created so far are marked KILLED - this kills quad 1 above as well
  - When processing $Z = *p$, killed quads not used for CSE
  - Fresh table entries are made for $Z = *p$

Handling Pointer References and Procedure Calls

- However, if we know apriori which objects p and q point to, then table entries corresponding to only those objects need to killed
- Procedure calls are similar
- With no dataflow analysis, we need to assume that a procedure call can modify any object in the basic block
  - changing call-by-reference parameters and global variables within procedures will affect other variables of the basic block as well
- Hence, while processing a procedure call, ALL table entries created so far are marked KILLED
- Sometimes, this problem is avoided by making a procedure call a separate basic block

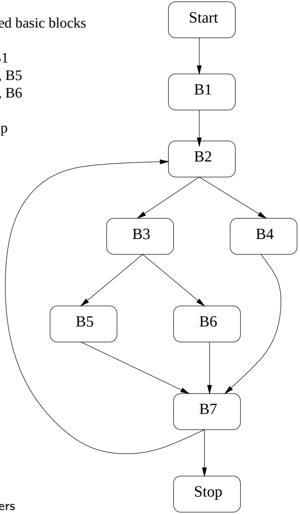
Extended Basic Blocks

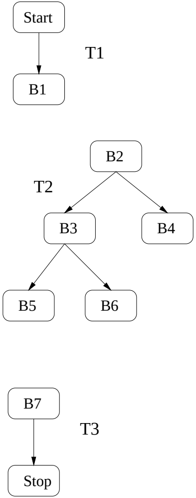
- A sequence of basic blocks $B_1, B_2, ..., B_k$, such that B_i is the unique predecessor of B_{i+1} $(i \leq i < k)$, and B_1 is either the start block or has no unique predecessor
- Extended basic blocks with shared blocks can be represented as a tree
- Shared blocks in extended basic blocks require scoped versions of tables
- The new entries must be purged and changed entries must be replaced by old entries
- Preorder traversal of extended basic block trees is used

Extended Basic Blocks and their Trees

Extended basic blocks

Start, B1
B2, B3, B5
B2, B3, B6
B2, B4
B7, Stop





Value Numbering with Extended Basic Blocks

function $visit\text{-}ebb\text{-}tree(e)$ // e is a node in the tree
begin
  // From now on, the new names will be entered with a new scope into the tables.
  // When searching the tables, we always search beginning with the current scope
  // and move to enclosing scopes. This is similar to the processing involved with
  // symbol tables for lexically scoped languages
  $value\text{-}number(e.B)$;
  // Process the block $e.B$ using the basic block version of the algorithm
  if $(e.left \neq null)$ then $visit\text{-}ebb\text{-}tree(e.left)$;
  if $(e.right \neq null)$ then $visit\text{-}ebb\text{-}tree(e.right)$;
  remove entries for the new scope from all the tables
  and undo the changes in the tables of enclosing scopes;
end

begin // main calling loop
  for each tree t do $visit\text{-}ebb\text{-}tree(t)$;
  // t is a tree representing an extended basic block
end