

Local Code Optimization

Code Optimization

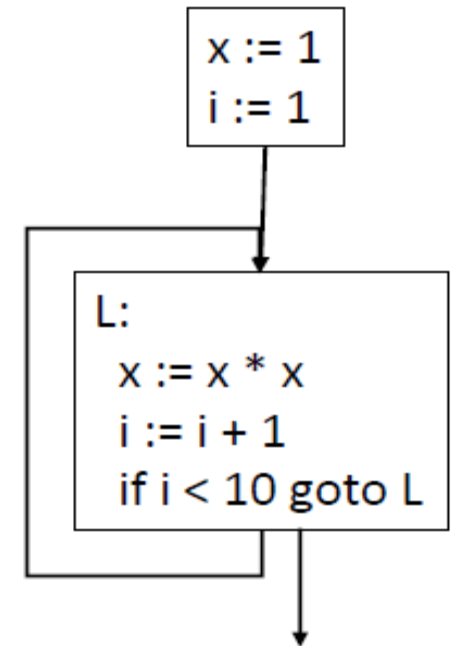
- Most complexity in modern compilers is in the optimizer
 - Also by far the largest phase
- Optimizations can be performed on
 - AST / DAG
 - Machine independent optimization
 - Assembly code
 - Target dependent optimization

Basic Blocks

- A basic block is a maximal sequence of instructions with:
 - no labels (except at the first instruction), and
 - no jumps (except in the last instruction)
- Idea:
 - Cannot jump into a basic block (except at beginning)
 - Cannot jump out of a basic block (except at end)
 - A basic block is a single-entry, single-exit, straight-line code segment
- *The property of sequential control flow can be useful for many optimizations.*

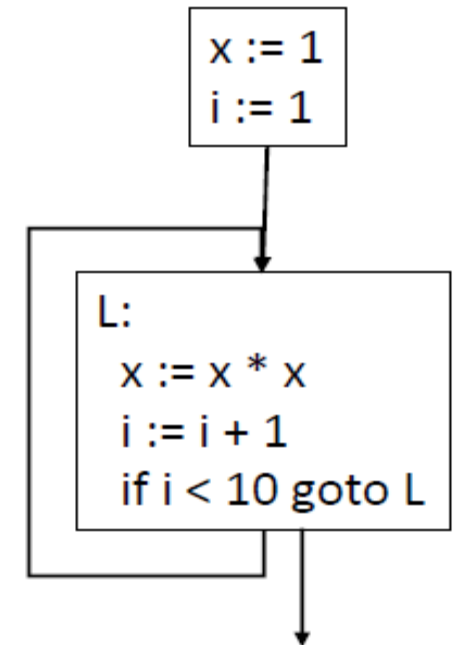
Control Flow Graphs

- Control-Flow Graph (CFG)
 - Models the way that the code transfers control between blocks in the procedure.
 - *Node*: a single basic block
 - *Edge*: transfer of control between basic blocks.
 - All *return* nodes are terminal



Generating Control Flow Graphs

- 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
- 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
- Any statement, not placed in a block, can never be executed, and may now be removed, if desired



Code Optimization

- Optimization seeks to improve a program's resource utilization
 - Execution time (most often)
 - Code size
 - Network messages sent
 - Memory Usages
 - Disk Accesses
 - Power
- Optimization should not alter what the program computes
 - *The answer before and after optimization must remain same*

Code Optimization

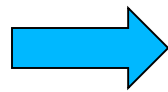
- There are three granularities of optimizations
 - Local optimizations
 - Apply to a basic block in isolation
 - Global optimizations
 - Apply to a control-flow graph (method body) in isolation
 - Inter-procedural optimizations
 - Apply across method boundaries
- *Most compilers do local and global optimizations but not the third*
- Often a conscious decision is made not to implement the fanciest optimization known
 - Goal: Maximum benefit for minimum cost

Local Optimization

■ Constant Folding

- Evaluation at compile-time of expressions whose operands are known to be constant
- Example

$a = 10 * 5 + 6 - b;$



```
t0 = 10 ;  
t1 = 5 ;  
t2 = t0 * t1 ;  
t3 = 6 ;  
t4 = t2 + t3 ;  
t5 = t4 - b ;  
a = t5 ;
```



```
t0 = 56 ;  
t1 = t0 - b ;  
a = t1 ;
```


Local Optimization

- Constant Propagation

- If a variable is assigned a constant value, then subsequent uses of that variable can be replaced by the constant as long as no intervening assignment has changed the value of the variable.
- Example:

```
t0 = 12 ;  
t1 = arr + t0 ;  
t2 = *(t1) ;
```

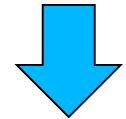


```
li $t0, 12  
lw $t1, 8($fp)  
add $t2, $t1, $t0  
lw $t3, 0($t2)
```



Constant propagation +
rearrangement cuts no. of
regs. and insns. from 4 to 2

```
t0 = *(arr + 12) ;
```



```
lw $t0, 8($fp)  
lw $t1, 12($t0)
```

Local Optimization

- Algebraic simplification and Reassociation
 - Simplifications use algebraic properties or particular operator-operand combinations to simplify expressions.
 - Reassociation refers to using properties such as associativity, commutativity and distributivity to rearrange an expression to enable other optimizations

Local Optimization

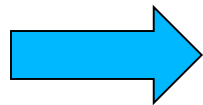
- Algebraic simplification and Re-association

- Simplification Examples:

$x+0 = x$ $0+x = x$ $x*1 = x$ $1*x = x$ $0/x = 0$ $x-0 = x$
 $b \ \&\& \ \text{true} = b$ $b \ \&\& \ \text{false} = \text{false}$
 $b \ || \ \text{true} = \text{true}$ $b \ || \ \text{false} = b$

- Example: Re-arrangement + constant folding

```
b = 5 + a + 10 ;
```



```
t0 = 5 ;  
t1 = t0 + a ;  
t2 = t1 + 10 ;  
b = t2 ;
```



```
t0 = 15 ;  
t1 = a + t0 ;  
b = t1 ;
```

Local Optimization

- Operator Strength Reduction
 - Replaces an operator by a "less expensive" one
 - Often performed as part of *loop-induction variable elimination*
 - Example:

```
while (i < 100) {  
    arr[i] = 0;  
    i = i + 1;  
}
```



```
t1 = i; t2 = arr;  
L0: If t1 > 100 Goto L1;  
t3 = 4 * t1 ;  
t4 = t2 + t3 ;  
*(t4) = 0 ;  
t1 = t1 + 1 ;  
Jump L0;  
L1:
```



```
t1 = i; t2 = arr;  
L0: If t1 > 100 Goto L1;  
* t2 = 0;  
t2 = t2 + 4;  
t1 = t1 + 1 ;  
Jump L0;  
L1:
```

Local Optimization

- Copy Propagation

- Similar to constant propagation, but generalized to non-constant values
- For $a = b$, we can replace later occurrences of a with b (assuming there are no changes to either variable in-between)
- Example

```
t2 = t1;  
t3 = t2 * t1;  
t4 = t3;  
t5 = t3 * t2;  
c = t5 + t4;
```



```
t3 = t1 * t1;  
t5 = t3 * t1;  
c = t5 + t3;
```

Dead Code Elimination

- Remove code which does not affect the program results.
- Benefits:
 - It shrinks program size
 - It allows the running program to avoid executing irrelevant operations and thus reduces running time.
- Dead code includes:
 - Code that is never be executed (unreachable code)
 - Code that only affects **dead variables**, that is, variables whose definition remains unused.

Dead Code Elimination

```
int foo(void) {  
    int a = 24;  
    int b = 25; /* Assignment to dead variable */  
    int c;  
    c = a << 2;  
    return c;  
    b = 24;      /* Unreachable code */  
    return 0; /* Unreachable code */  
}
```

Common Sub-expression Elimination (CSE)

- Two operations are *common* if they produce the same result.
- An expression is *alive* if the operands used to compute the expression have not been changed.
 - An expression not alive is *dead*.
- *CSE* searches for instances of expressions that evaluate to the same value and analyses if it may be replaced with a single variable holding the computed value.

```
a = b * c + g;  
d = b * c * d;
```



CSE done by using “tmp” in the definition of “d”

```
tmp = b * c;  
a = tmp + g;  
d = tmp * d;
```


Local Optimization Example

a := x ** 2

b := 3

c := x

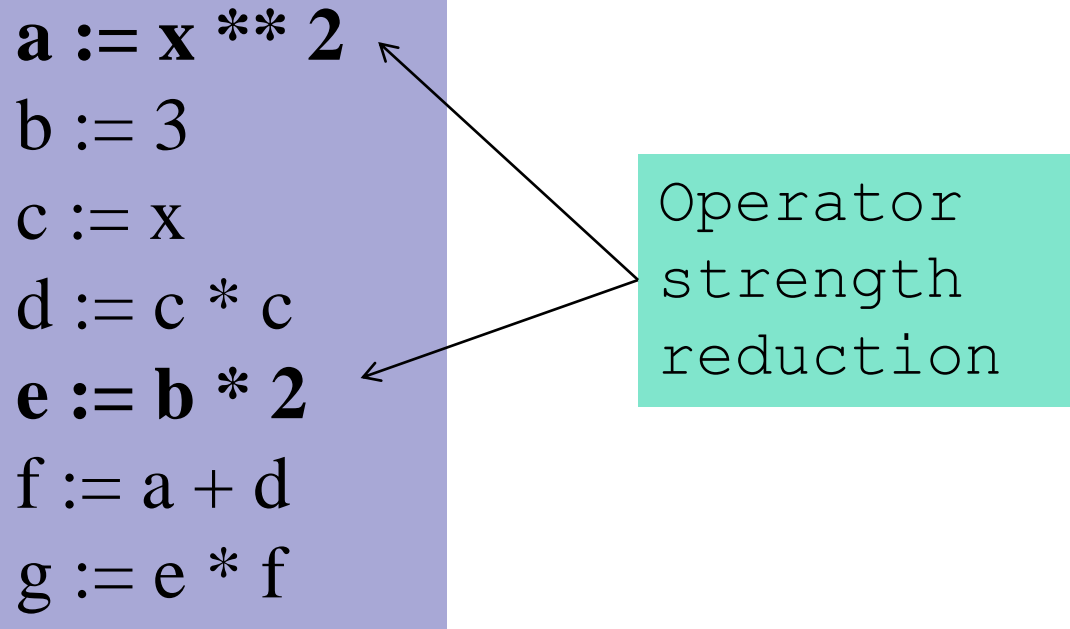
d := c * c

e := b * 2

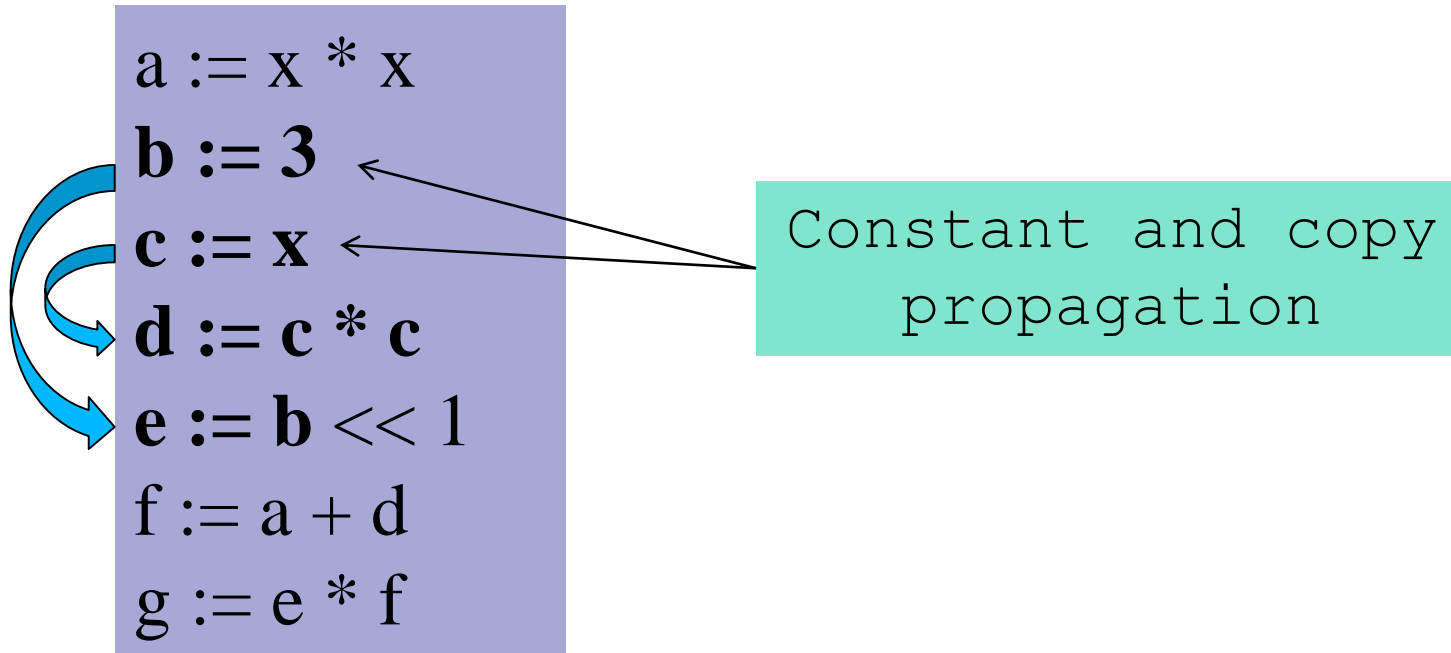
f := a + d

g := e * f

Operator
strength
reduction



Local Optimization Example



Local Optimization Example

```
a := x * x  
b := 3  
c := x  
d := x * x  
e := 3 << 1  
f := a + d  
g := e * f
```

Constant folding



Local Optimization Example

a := x * x

b := 3

c := x

d := x * x

e := 6

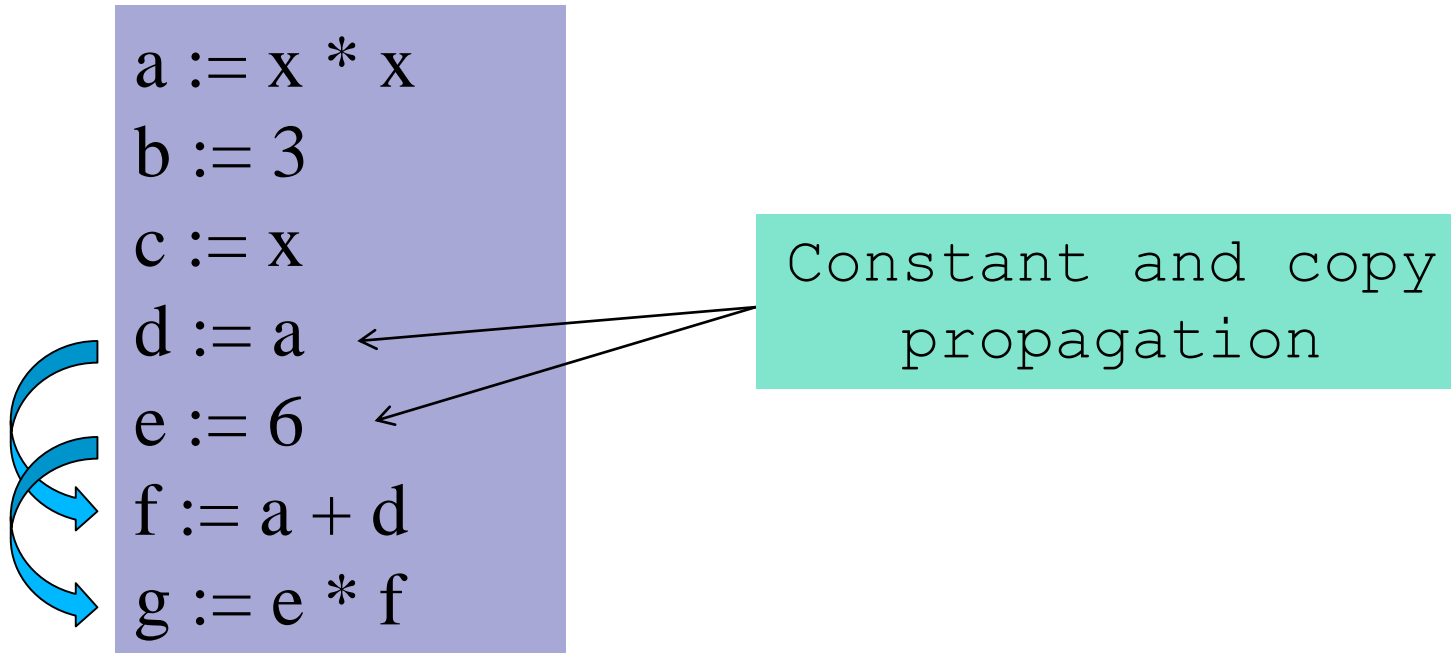
f := a + d

g := e * f

Common
Subexpression
Elimination



Local Optimization Example



Local Optimization Example

`a := x * x`

`b := 3`

`c := x`

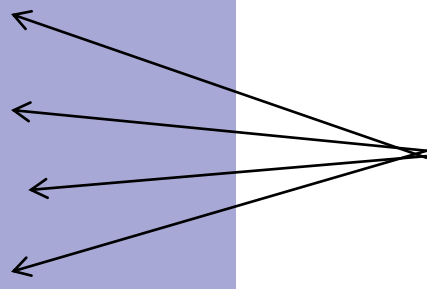
`d := a`

`e := 6`

`f := a + a`

`g := 6 * f`

Dead code
elimination



Local Optimization Example

```
a := x * x  
f := a + a  
g := 6 * f
```

Optimized form

An Automated Local Optimization Algorithm

- **Value Numbering** – Central idea:
 - **Assign numbers (called value numbers (vn)) to expressions** in such a way that two expressions receive the same VN if the compiler can prove that they are equal for all possible program inputs
- Can be used to eliminate common sub-expressions, do constant folding, and constant propagation **in basic blocks**
- Can take advantage of algebraic simplification and re-association

Value Numbering

- The algorithm uses three tables indexed by appropriate hash values:
 - **HashTable**: holds vn for expressions
 - Format: <expression, vn>
 - Indexed by expression hash value
 - **ValnumTable**: holds vn for named identifiers
 - Format: <name, vn>
 - Indexed by name hash value
 - **NameTable**: holds the values for names
- In case of constants
 - Format: <name list, constant value, constflag>
 - Indexed by vn
 - In the field Namelist, first name is the defining occurrence and replaces all other names with the same vn with itself (or its constant value)

Value Numbering Example

HLL Program	Quadruples before Value-Numbering	Quadruples after Value-Numbering
$a = 10$ $b = 4 * a$ $c = i * j + b$ $d = 15 * a * c$ $e = i$ $c = e * j + i * a$	<ol style="list-style-type: none"> 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. $c = t3 + t4$ 	<ol style="list-style-type: none"> 1. $a = 10$ 2. $b = 40$ 3. $t1 = i * j$ 4. $c = t1 + 40$ 5. $t2 = 150$ 6. $d = 150 * c$ 7. $e = i$ 8. $t3 = i * j$ 9. $t4 = i * 10$ 10. $c = t1 + t4$ <p>(Instructions 5 and 8 can be deleted)</p>

Value Numbering Example

- $a = 10$
 - a is entered into Valnum Table (with a vn of 1, say) and into Name Table (with a constant value of 10)
- $b = 4 * a$
 - a is found in Valnum Table, its constant value of 10 in Name Table
 - We perform constant propagation and folding
 - $4 * a$ is evaluated to 40
 - b is entered into Valnum Table (with a vn of 2) and into Name Table (with a constant value of 40)
- $t1 = i * j$
 - i and j are entered into the two tables with new vn (as above), but with no constant value
 - $i * j$ is entered into Hash Table with a new vn
 - $t1$ is entered into Valnum Table with the same vn

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

Hash Table		
Expression	VN	
$i * j$	5	
$t1 + 40$	6	
$150 * c$	8	
$i * 10$	9	
$t3 + t4$	10	

Name Table		
Name	CV	CF
a	10	T
b	40	T
i, e	-	
j	-	
t1, t3	-	
t2	150	T
d	7	
c	8	

Value Numbering Example

- Similar actions for “ $c = t1 + b$ ”, “ $t2 = 15 * a$ ”, “ $d = t2 * c$ ”
- $e = i$
 - e gets the same vn as i
- $t3 = e * j$
 - e and i have the same vn
 - $e * j$ is detected to be the same as $i * j$
 - since $i * j$ is already in the Hash Table, we have found a *common subexpression*
 - from now on, all uses of $t3$ can be replaced by $t1$
 - The instruction $t3 = e * j$ can be deleted
- $t4 = i * a$
 - Similar
- $c = t3 + t4$
 - $t3$ and $t4$ already exist and have vn
 - $t3 + t4$ is entered into Hash Table with a new vn
 - Reassignment to c ; c gets a different vn, same as that of $t3 + t4$

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

Hash Table		
Expression	VN	
$i * j$	5	
$t1 + 40$	6	
$150 * c$	8	
$i * 10$	9	
$t3 + t4$	10	

Name Table		
Name	CV	CF
a	10	T
b	40	T
i, e	-	
j	-	
t1, t3	-	
t2	150	T
d	-	
c	-	

Value Numbering Example

- When a search for an expression $i + j$ in HashTable fails, try for $j + i$
- If there is an instruction $x = i + 0$, 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 *alive*
- All unmarked quads can be deleted at the end

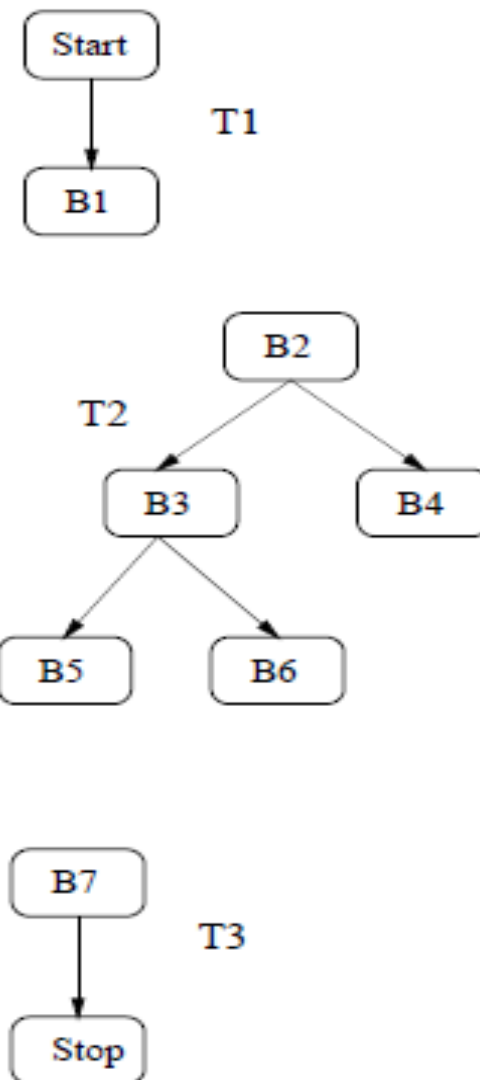
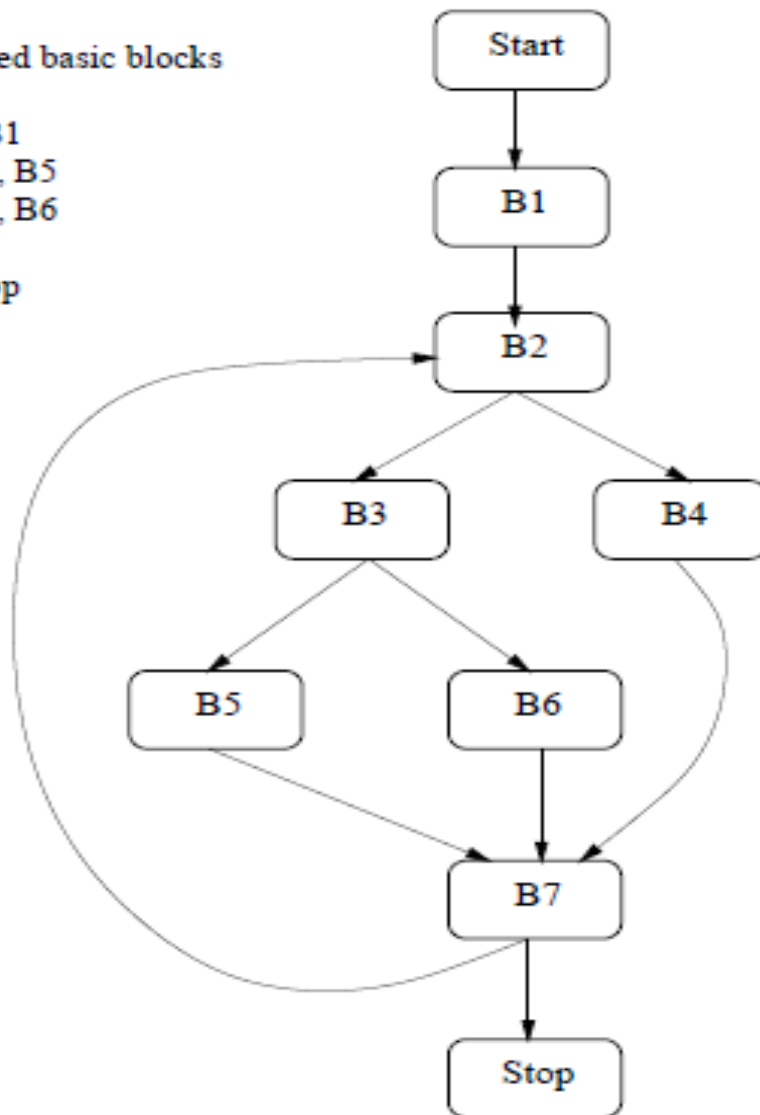
Extended Basic Blocks

- A sequence of basic blocks B_1, B_2, \dots, B_k , such that B_i is the unique predecessor of B_{i+1} ($1 \leq i < k$), 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

Extended basic blocks

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



Extended Basic Blocks

```
function visit-ebb-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-number(e.B);
    // Process the block e.B using the basic block version of the algorithm
    if (e.left  $\neq$  null) then visit-ebb-tree(e.left);
    if (e.right  $\neq$  null) then visit-ebb-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-ebb-tree(t);
    // t is a tree representing an extended basic block
end
```