Code Generation and Optimization

Lecture 11

Objectives

By the end of this lecture you should be able to:

1. Generate target code for a simple machine model.
2. Construct flow graphs.
3. Determine next-use information for variables.
4. Carry out simple optimizations on basic blocks.

Outline

Outline

Factors Affecting Code Generation

- The design, complexity, and optimality of a code generator depends on several factors:
  1. The intermediate representation.
  2. The target language.
  3. The optimality criterion.

Factors Affecting Code Generation

- The design, complexity, and optimality of a code generator depends on several factors:
  1. The intermediate representation.
  2. The target language.
  3. The optimality criterion.

Factors Affecting Code Generation

- The design, complexity, and optimality of a code generator depends on several factors:
  1. The intermediate representation.
  2. The target language.
  3. The optimality criterion.

Factors Affecting Code Generation

- The design, complexity, and optimality of a code generator depends on several factors:
  1. The intermediate representation.
  2. The target language.
  3. The optimality criterion.

Issues in Code Generation

1. Instruction selection.
   - For example, should we use an `inc` instruction or uniform addition?
2. Register allocation.
   - Which variables reside in registers.
   - Which variable resides in which register.
3. Evaluation order.
   - Finding the optimal order is **NP**-complete.

Target Machine Model

We assume a target machine with the following specifications.

- Byte-addressable.
- n general purpose registers: `R0`, `R1`, `R`$n-1$.
- Instruction set:
  - Loading: `LD` r, src; loads the value in src into register r.
  - Storing: `ST` loc, r; stores the value in register r into location loc.
  - Computing: OP dst, $src1$ (, $src2$)?.
  - Jumping: `BR` $label$.
  - Conditional Jumping: `BR`$cond$ r, $label$.
- In the above, a src is a register, a constant ($\#c$), or a location.

Target Machine Model

We assume a target machine with the following specifications.

- Byte-addressable.
- n general purpose registers: `R0`, `R1`, `R`$n-1$.
- Instruction set:
  - Loading: `LD` r, src; loads the value in src into register r.
  - Storing: `ST` loc, r; stores the value in register r into location loc.
  - Computing: OP dst, $src1$ (, $src2$)?.
  - Jumping: `BR` $label$.
  - Conditional Jumping: `BR`$cond$ r, $label$.
- In the above, a src is a register, a constant (#c), or a location.

Target Machine Model

We assume a target machine with the following specifications.

- Byte-addressable.
- n general purpose registers: `R0`, `R1`, `R`$n-1$.
- Instruction set:
  - Loading: `LD` *r*, *src*; loads the value in *src* into register *r*.
  - Storing: `ST` *loc*, *r*; stores the value in register *r* into location *loc*.
  - Computing: *OP dst*, *src*1 (, *src*2)?.
  - Jumping: `BR` *label*.
  - Conditional Jumping: `BR`*cond r*, *label*.
- In the above, a *src* is a register, a constant (#*c*), or a location.

Target Machine Model

We assume a target machine with the following specifications.

- Byte-addressable.
- n general purpose registers: `R0`, `R1`, `R`$n-1$.
- Instruction set:
  - Loading: `LD` *r*, *src*; loads the value in *src* into register *r*.
  - Storing: `ST` *loc*, *r*; stores the value in register *r* into location *loc*.
  - Computing: *OP dst*, *src*1 (, *src*2)?.
  - Jumping: `BR` *label*.
  - Conditional Jumping: `BR`*cond r*, *label*.
- In the above, a *src* is a register, a constant (#*c*), or a location.

Target Machine Model

We assume a target machine with the following specifications.

- Byte-addressable.
- n general purpose registers: `R0`, `R1`, `R`$n-1$.
- Instruction set:
  - Loading: `LD` *r*, *src*; loads the value in *src* into register *r*.
  - Storing: `ST` *loc*, *r*; stores the value in register *r* into location *loc*.
  - Computing: *OP dst*, *src*1 (, *src*2)?.
  - Jumping: `BR` *label*.
  - Conditional Jumping: `BR`*cond r*, *label*.
- In the above, a *src* is a register, a constant (#*c*), or a location.

Target Machine Model

We assume a target machine with the following specifications.

- Byte-addressable.
- n general purpose registers: `R0`, `R1`, `R`$n-1$.
- Instruction set:
  - Loading: `LD` *r*, *src*; loads the value in *src* into register *r*.
  - Storing: `ST` *loc*, *r*; stores the value in register *r* into location *loc*.
  - Computing: *OP dst*, *src*1 (, *src*2)?.
  - Jumping: `BR` *label*.
  - Conditional Jumping: `BR`*cond r*, *label*.
- In the above, a *src* is a register, a constant (#*c*), or a location.

Target Machine Model

We assume a target machine with the following specifications.

- Byte-addressable.
- n general purpose registers: `R0`, `R1`, `R`$n-1$.
- Instruction set:
  - Loading: `LD` *r*, *src*; loads the value in *src* into register *r*.
  - Storing: `ST` *loc*, *r*; stores the value in register *r* into location *loc*.
  - Computing: *OP dst*, *src*1 (, *src*2)?.
  - Jumping: `BR` *label*.
  - Conditional Jumping: `BR`*cond r*, *label*.
- In the above, a *src* is a register, a constant (#*c*), or a location.

Target Machine Model

We assume a target machine with the following specifications.

- Byte-addressable.
- n general purpose registers: `R0`, `R1`, `R`$n-1$.
- Instruction set:
  - Loading: `LD` *r*, *src*; loads the value in *src* into register *r*.
  - Storing: `ST` *loc*, *r*; stores the value in register *r* into location *loc*.
  - Computing: *OP dst*, *src*1 (, *src*2)?.
  - Jumping: `BR` *label*.
  - Conditional Jumping: `BR`*cond r*, *label*.
- In the above, a *src* is a register, a constant (#*c*), or a location.

Target Machine Model

We assume a target machine with the following specifications.

- Byte-addressable.
- n general purpose registers: `R0`, `R1`, `R`$n-1$.
- Instruction set:
  - Loading: `LD` *r*, *src*; loads the value in *src* into register *r*.
  - Storing: `ST` *loc*, *r*; stores the value in register *r* into location *loc*.
  - Computing: *OP dst*, *src*1 (, *src*2)?.
  - Jumping: `BR` *label*.
  - Conditional Jumping: `BR`*cond r*, *label*.
- In the above, a *src* is a register, a constant (#*c*), or a location.

Target Machine Model: Addressing Modes

In instructions, a location can be any of the following.

1. A variable name, standing for a memory location.
2. An indexed address $a(r)$, where a is a variable name or an integer and r is a register. Here, $a(r) = contents(a) + contents(r)$ or $a(r) = a + contents(r)$, respectively.
3. An indirect address $*r$, where r is a register. Here, $*r = contents(contents(r))$.
4. An indexed indirect address $*n(r)$, where n is an integer and r is a register. Here, $*n(r) = contents(n + contents(r))$.

Target Machine Model: Addressing Modes

In instructions, a location can be any of the following.

1. A variable name, standing for a memory location.
2. An indexed address $a(r)$, where a is a variable name or an integer and r is a register. Here, $a(r) = contents(a) + contents(r)$ or $a(r) = a + contents(r)$, respectively.
3. An indirect address $*r$, where r is a register. Here, $*r = contents(contents(r))$.
4. An indexed indirect address $*n(r)$, where n is an integer and r is a register. Here, $*n(r) = contents(n + contents(r))$.

Target Machine Model: Addressing Modes

In instructions, a location can be any of the following.

1. A variable name, standing for a memory location.
2. An indexed address $a(r)$, where a is a variable name or an integer and r is a register. Here, $a(r) = contents(a) + contents(r)$ or $a(r) = a + contents(r)$, respectively.
3. An indirect address $*r$, where r is a register. Here, $*r = contents(contents(r))$.
4. An indexed indirect address $*n(r)$, where n is an integer and r is a register. Here, $*n(r) = contents(n + contents(r))$.

Target Machine Model: Addressing Modes

In instructions, a location can be any of the following.

1. A variable name, standing for a memory location.
2. An indexed address $a(r)$, where a is a variable name or an integer and r is a register. Here, $a(r) = contents(a) + contents(r)$ or $a(r) = a + contents(r)$, respectively.
3. An indirect address $*r$, where r is a register. Here, $*r = contents(contents(r))$.
4. An indexed indirect address $*n(r)$, where n is an integer and r is a register. Here, $*n(r) = contents(n + contents(r))$.

Instruction Costs

- Assume instruction cost depends solely on the addressing mode.
- Instructions with operands involving constants or variables need one extra word for each constant or variable.
- The cost is the number of memory references needed to fetch and execute the instruction.
- For instruction with only register operands, the cost is 1.

Example

- $cost(\texttt{LD R1, R2}) = 1$.
- $cost(\texttt{LD R1, \#100}) = 2$.
- $cost(\texttt{ST x, R1}) = 3$.
- $cost(\texttt{LD R1, *10(R2)}) = 4$.

Instruction Costs

- Assume instruction cost depends solely on the addressing mode.
- Instructions with operands involving constants or variables need one extra word for each constant or variable.
- The cost is the number of memory references needed to fetch and execute the instruction.
- For instruction with only register operands, the cost is 1.

Example

- $cost($`LD R1, R2`$) = 1$.
- $cost($`LD R1, #100`$) = 2$.
- $cost($`ST x, R1`$) = 3$.
- $cost($`LD R1, *10(R2)`$) = 4$.

Instruction Costs

- Assume instruction cost depends solely on the addressing mode.
- Instructions with operands involving constants or variables need one extra word for each constant or variable.
- The cost is the number of memory references needed to fetch and execute the instruction.
- For instruction with only register operands, the cost is 1.

Example

- $cost(\texttt{LD R1, R2}) = 1$.
- $cost(\texttt{LD R1, \#100}) = 2$.
- $cost(\texttt{ST x, R1}) = 3$.
- $cost(\texttt{LD R1, *10(R2)}) = 4$.

Instruction Costs

- Assume instruction cost depends solely on the addressing mode.
- Instructions with operands involving constants or variables need one extra word for each constant or variable.
- The cost is the number of memory references needed to fetch and execute the instruction.
- For instruction with only register operands, the cost is 1.

Example

- $cost(\texttt{LD R1, R2}) = 1$.
- $cost(\texttt{LD R1, \#100}) = 2$.
- $cost(\texttt{ST x, R1}) = 3$.
- $cost(\texttt{LD R1, *10(R2)}) = 4$.

Instruction Costs

- Assume instruction cost depends solely on the addressing mode.
- Instructions with operands involving constants or variables need one extra word for each constant or variable.
- The cost is the number of memory references needed to fetch and execute the instruction.
- For instruction with only register operands, the cost is 1.

Example

- $cost(\texttt{LD R1, R2}) = 1$.
- $cost(\texttt{LD R1, \#100}) = 2$.
- $cost(\texttt{ST x, R1}) = 3$.
- $cost(\texttt{LD R1, *10(R2)}) = 4$.

Instruction Costs

- Assume instruction cost depends solely on the addressing mode.
- Instructions with operands involving constants or variables need one extra word for each constant or variable.
- The cost is the number of memory references needed to fetch and execute the instruction.
- For instruction with only register operands, the cost is 1.

Example

- *cost*(`LD R1, R2`) = 1.
- *cost*(`LD R1, #100`) = 2.
- *cost*(`ST x, R1`) = 3.
- *cost*(`LD R1, *10(R2)`) = 4.

Instruction Costs

- Assume instruction cost depends solely on the addressing mode.
- Instructions with operands involving constants or variables need one extra word for each constant or variable.
- The cost is the number of memory references needed to fetch and execute the instruction.
- For instruction with only register operands, the cost is 1.

Example

- *cost*(`LD R1, R2`) = 1.
- *cost*(`LD R1, #100`) = 2.
- *cost*(`ST x, R1`) = 3.
- *cost*(`LD R1, *10(R2)`) = 4.

Instruction Costs

- Assume instruction cost depends solely on the addressing mode.
- Instructions with operands involving constants or variables need one extra word for each constant or variable.
- The cost is the number of memory references needed to fetch and execute the instruction.
- For instruction with only register operands, the cost is 1.

Example

- *cost*(`LD R1, R2`) = 1.
- *cost*(`LD R1, #100`) = 2.
- *cost*(`ST x, R1`) = 3.
- *cost*(`LD R1, *10(R2)`) = 4.

Instruction Costs

- Assume instruction cost depends solely on the addressing mode.
- Instructions with operands involving constants or variables need one extra word for each constant or variable.
- The cost is the number of memory references needed to fetch and execute the instruction.
- For instruction with only register operands, the cost is 1.

Example

- *cost*(`LD R1, R2`) = 1.
- *cost*(`LD R1, #100`) = 2.
- *cost*(`ST x, R1`) = 3.
- *cost*(`LD R1, *10(R2)`) = 4.

Exercise 1

Example

Generate target code for the three-address instruction `b = a[i]`. What is the cost of your code?

Code.

```
LD R1, i
LD R2, a(R1)
ST b, R2
```

Cost. $3 + 3 + 3 = 9$.

Exercise 1

Example

Generate target code for the three-address instruction `b = a[i]`. What is the cost of your code?

Code.

```
LD R1, i
LD R2, a(R1)
ST b, R2
```

Cost. $3 + 3 + 3 = 9$.

Exercise 2

Example

Generate target code for the three-address instruction `*p = y`. What is the cost of your code?

Code.

```
LD R1, p
LD R2, y
ST 0(R1), R2
```

Cost. $3 + 3 + 3 = 9$.

Exercise 2

Example

Generate target code for the three-address instruction `*p = y`. What is the cost of your code?

Code.

```
LD R1, p
LD R2, y
ST 0(R1), R2
```

Cost. $3 + 3 + 3 = 9$.

Outline

A Graphical Representation of Intermediate Code

- Flow graphs constitute a graphical representation of the intermediate code.
- By considering a flow graph, we may optimize generated code.
  - The flow graph need not be explicitly constructed.
- A flow graph shows how values are defined and used.
- Nodes of a flow graph correspond to basic blocks of the intermediate code.

Basic Blocks

Definition

The set of leaders of a segment of intermediate code is the smallest set satisfying the following.

1. The first instruction of the code segment is a leader.
2. An instruction which is the target of a (conditional or unconditional) jump is a leader.
3. An instruction which immediately follows a (conditional or unconditional) jump is a leader.

Intuition: Control flows to a leader not simply by following the textual sequence of instructions.

Definition

A basic block of a segment of intermediate code is the sequence of instructions starting with a leader up to (i) but not including the next leader, or (ii) the end of the code segment.

Basic Blocks

Definition

The set of leaders of a segment of intermediate code is the smallest set satisfying the following.

1. The first instruction of the code segment is a leader.
2. An instruction which is the target of a (conditional or unconditional) jump is a leader.
3. An instruction which immediately follows a (conditional or unconditional) jump is a leader.

Intuition: Control flows to a leader not simply by following the textual sequence of instructions.

Definition

A basic block of a segment of intermediate code is the sequence of instructions starting with a leader up to (i) but not including the next leader, or (ii) the end of the code segment.

Basic Blocks

Definition

The set of leaders of a segment of intermediate code is the smallest set satisfying the following.

1. The first instruction of the code segment is a leader.
2. An instruction which is the target of a (conditional or unconditional) jump is a leader.
3. An instruction which immediately follows a (conditional or unconditional) jump is a leader.

Intuition: Control flows to a leader not simply by following the textual sequence of instructions.

Definition

A basic block of a segment of intermediate code is the sequence of instructions starting with a leader up to (i) but not including the next leader, or (ii) the end of the code segment.

Exercise

Example

```
1)  i = 1                          10)  i = i + 1
2)  j = 1                          11)  if i <= 10 goto (2)
3)  t1 = 10 * i                    12)  i = 1
4)  t2 = t1 + j                    13)  t5 = i - 1
5)  t3 = 8 * t2                    14)  t6 = 88 * t5
6)  t4 = t3 - 88                   15)  a[t6] = 1.0
7)  a[t4] = 0.0                    16)  i = i + 1
8)  j = j + 1                      17)  if i <= 10 goto (13)
9)  if j <= 10 goto (3)
```

©Aho et al. (2007)

Basic Blocks: $\langle 1 \rangle$, $\langle 2 \rangle$, $\langle 3, 4, 5, 6, 7, 8, 9 \rangle$, $\langle 10, 11 \rangle$, $\langle 12 \rangle$, $\langle 13, 14, 15, 16, 17 \rangle$.

The Flow Graph

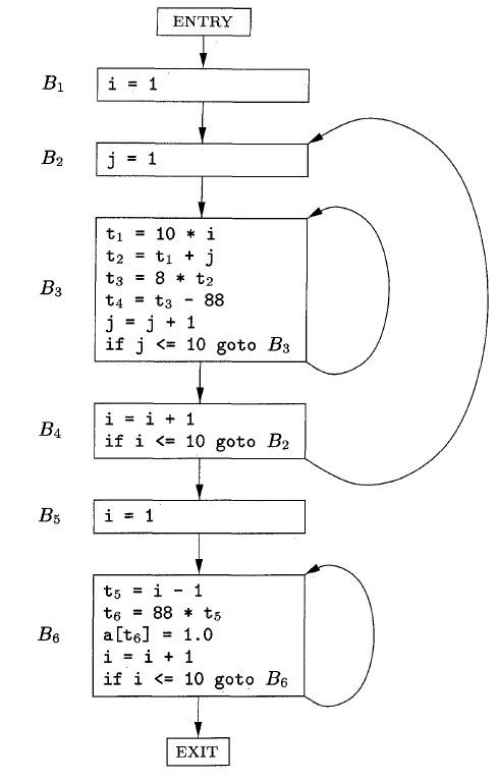
Definition

The flow graph of a segment S of intermediate code is a graph $F(S) = (\{\text{ENTRY}, \text{EXIT}\} \cup \mathcal{B}(S), E)$, where

1. $\mathcal{B}(S)$ is the set of basic blocks of S, and
2. $(B_1, B_2) \in E$ if and only if
   1. there is a (conditional or unconditional) jump from the last instruction of B_1 to the leader of B_2,
   2. the leader of B_2 immediately follows the last instruction of B_1 which is not an unconditional jump,
   3. $B_1 = \text{ENTRY}$ and B_2 is the block whose leader is the first instruction of S, or
   4. $B_2 = \text{EXIT}$ and B_1 is a block that ends with a jump to an instruction outside S or is the block containing the last instruction of S (which is not an unconditional jump).

Exercise

Example



©Aho et al. (2007)

Outline

Next-Use

- To optimize register allocation, it is important to collect information about the use of values stored in registers.
- For example, if a value stored in a register will never be used following some instruction, then the register may be allocated to another value.

Definition

Suppose that (i) instruction i assigns a value to variable x, (ii) instruction j uses x as an operand, and (iii) control can flow from statement i to statement j along a path that has no assignments to x. Then, we say that j uses the value of x computed at i and that x is alive at i.

Next-Use

- To optimize register allocation, it is important to collect information about the use of values stored in registers.
- For example, if a value stored in a register will never be used following some instruction, then the register may be allocated to another value.

Definition

Suppose that (i) instruction i assigns a value to variable x, (ii) instruction j uses x as an operand, and (iii) control can flow from statement i to statement j along a path that has no assignments to x. Then, we say that j uses the value of x computed at i and that x is alive at i.

Next-Use: Algorithm

- We would like to determine next-uses and liveness of variables within a basic block B.
- Assume the symbol table initially shows all non-temporary variables as alive.
- We start with the last instruction in B and scan backwards to the leader of B.
- For every instruction $i : x = y\ op\ z$, do the following.
  1. Attach to i the information in the symbol table about liveness and next-use of x, y, and z.
  2. In the symbol table, set x to "not live" and "no next use".
  3. In the symbol table, set y and z to "live" and the next-uses of y and z to i.

Exercise

Example

Determine the liveness and next-use information for the instructions in the following basic block.

```
1.  x = z + 1
2.  x = x * 4
3.  y = x + 1
4.  y = z * 3
5.  x = y + 2
6.  if (x > 0) goto L
```

Exercise

Example

Determine the liveness and next-use information for the instructions in the following basic block.

1. `x = z + 1`
   $x.liveness = 1; x.next-use = 2; z.liveness = 1; z.next-use = 4$
2. `x = x * 4`
   $x.liveness = 1; x.next-use = 3$
3. `y = x + 1`
   $y.liveness = 0; y.next-use = none; x.liveness = 0$
4. `y = z * 3`
   $y.liveness = 1; y.next-use = 5; z.liveness = 1$
5. `x = y + 2`
   $x.liveness = 1; y.liveness = 1$
6. `if (x > 0) goto L`

Optimization with the Next-Use Algorithm

- What is a simple modification to the Next-Use algorithm which admits some code optimization?
- Apply the new algorithm to the previous example.

DAG Representation of a Basic Block

- Many optimization techniques for basic blocks assume a DAG representation of the block.

DAG Representation

1. There are nodes for initial values of variables in the block.
2. There are nodes for constants appearing in the block.
3. There is a node $N(s)$ for each statement s in the block. $N(s)$ is labelled by the operator applied in s and attached to it is a set of variables for which it is the last definition within the block.
4. Children of $N(s)$ are nodes corresponding to statements which are the last definitions, before s, of operands used in s. The left-to-right ordering of children corresponds to the textual order of operands.

- The DAG representation allows us to carry out several block-optimizing transformations.

DAG Representation of a Basic Block

- Many optimization techniques for basic blocks assume a DAG representation of the block.

DAG Representation

1. There are nodes for initial values of variables in the block.
2. There are nodes for constants appearing in the block.
3. There is a node $N(s)$ for each statement s in the block. $N(s)$ is labelled by the operator applied in s and attached to it is a set of variables for which it is the last definition within the block.
4. Children of $N(s)$ are nodes corresponding to statements which are the last definitions, before s, of operands used in s. The left-to-right ordering of children corresponds to the textual order of operands.

- The DAG representation allows us to carry out several block-optimizing transformations.

DAG Representation of a Basic Block

- Many optimization techniques for basic blocks assume a DAG representation of the block.

DAG Representation

1. There are nodes for initial values of variables in the block.
2. There are nodes for constants appearing in the block.
3. There is a node $N(s)$ for each statement s in the block. $N(s)$ is labelled by the operator applied in s and attached to it is a set of variables for which it is the last definition within the block.
4. Children of $N(s)$ are nodes corresponding to statements which are the last definitions, before s, of operands used in s. The left-to-right ordering of children corresponds to the textual order of operands.

- The DAG representation allows us to carry out several block-optimizing transformations.

DAG Representation of a Basic Block

- Many optimization techniques for basic blocks assume a DAG representation of the block.

DAG Representation

1. There are nodes for initial values of variables in the block.
2. There are nodes for constants appearing in the block.
3. There is a node $N(s)$ for each statement s in the block. $N(s)$ is labelled by the operator applied in s and attached to it is a set of variables for which it is the last definition within the block.
4. Children of $N(s)$ are nodes corresponding to statements which are the last definitions, before s, of operands used in s. The left-to-right ordering of children corresponds to the textual order of operands.

- The DAG representation allows us to carry out several block-optimizing transformations.

DAG Representation of a Basic Block

- Many optimization techniques for basic blocks assume a DAG representation of the block.

DAG Representation

1. There are nodes for initial values of variables in the block.
2. There are nodes for constants appearing in the block.
3. There is a node $N(s)$ for each statement s in the block. $N(s)$ is labelled by the operator applied in s and attached to it is a set of variables for which it is the last definition within the block.
4. Children of $N(s)$ are nodes corresponding to statements which are the last definitions, before s, of operands used in s. The left-to-right ordering of children corresponds to the textual order of operands.

- The DAG representation allows us to carry out several block-optimizing transformations.

DAG Representation of a Basic Block

- Many optimization techniques for basic blocks assume a DAG representation of the block.

DAG Representation

1. There are nodes for initial values of variables in the block.
2. There are nodes for constants appearing in the block.
3. There is a node $N(s)$ for each statement s in the block. $N(s)$ is labelled by the operator applied in s and attached to it is a set of variables for which it is the last definition within the block.
4. Children of $N(s)$ are nodes corresponding to statements which are the last definitions, before s, of operands used in s. The left-to-right ordering of children corresponds to the textual order of operands.

- The DAG representation allows us to carry out several block-optimizing transformations.

DAG Representation of a Basic Block

- Many optimization techniques for basic blocks assume a DAG representation of the block.

DAG Representation

1. There are nodes for initial values of variables in the block.
2. There are nodes for constants appearing in the block.
3. There is a node $N(s)$ for each statement s in the block. $N(s)$ is labelled by the operator applied in s and attached to it is a set of variables for which it is the last definition within the block.
4. Children of $N(s)$ are nodes corresponding to statements which are the last definitions, before s, of operands used in s. The left-to-right ordering of children corresponds to the textual order of operands.

- The DAG representation allows us to carry out several block-optimizing transformations.

Detecting Common Sub-Expressions

Rule

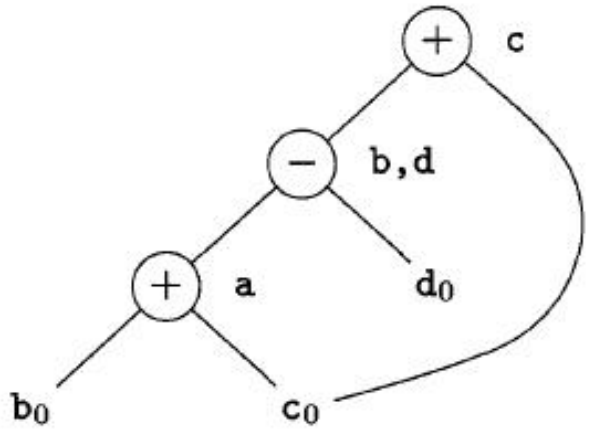
The DAG should never have two nodes with the same label and the same children.

- This is a sign of a common sub-expression.
- Instead of creating a new node, add a variable to the attached set.

An Example

Example

```
a = b + c
b = a - d
c = b + c
d = a - d
```



©Aho et al. (2007)

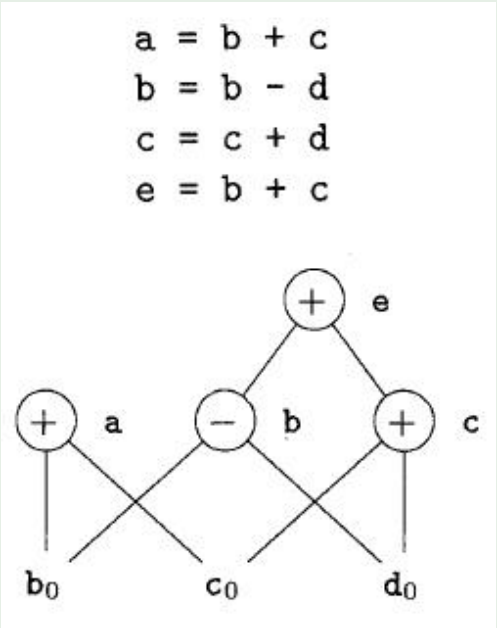
Detecting Dead Code

Rule

A node with no parents and no live variables attached may be eliminated.

An Example

Example



©Aho et al. (2007)

Supposing that *e* and *c* are not live, what is the result of applying the dead code rule?

Using Algebraic Identities

Rule

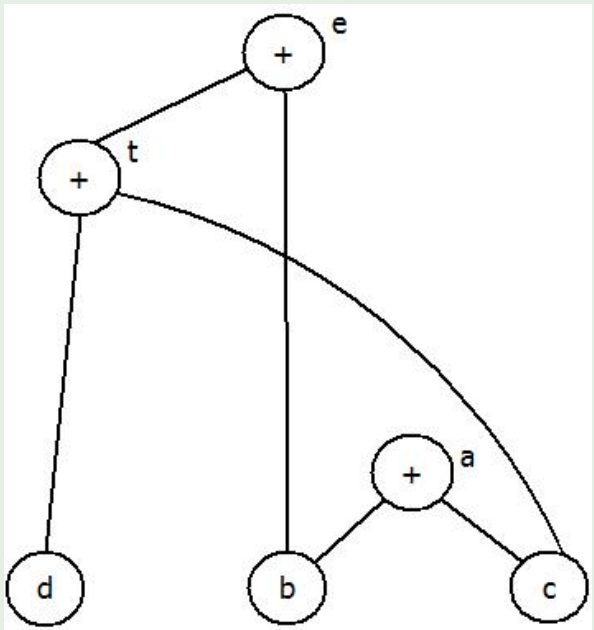
Whenever possible, use algebraic identities to simplify code:

- Constant folding: compute expressions involving only constants and replace them by their computed values.
- Use commutativity and associativity of operators wisely.
  - Use commutativity to discover equivalent common sub-expressions.
  - Use associativity to eliminate nodes.

An Example

Example

```
a = b + c
t = c + d
e = t + b
```



$\Longrightarrow$