Lecture 4

Local Optimizations

I. Basic blocks/Flow graphs

II. Abstraction 1: DAG

III. Abstraction 2: Value numbering

I. Basic Blocks & Flow Graphs

Basic block = a sequence of 3-address statements

- only the first statement can be reached from outside the block (no branches into middle of block)
- all the statements are executed consecutively if the first one is (no branches out or halts except perhaps at end of block)

- **We require basic blocks to be *maximal***, i.e., they cannot be made larger without violating the conditions

Flow graph

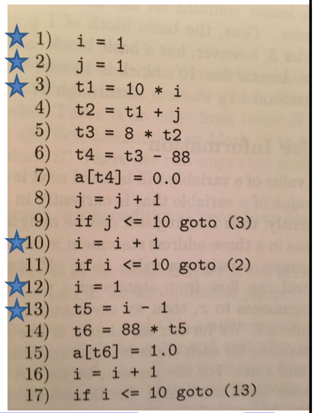
- **Nodes: basic blocks**
- **Edges: B_i -> B_j, iff B_j can follow B_i immediately in *some* execution**
  - Either first instruction of B_j is target of a goto at end of B_i
  - Or, B_j physically follows B_i, which does not end in an unconditional goto.

Partitioning into Basic Blocks

Identify the leader of each basic block

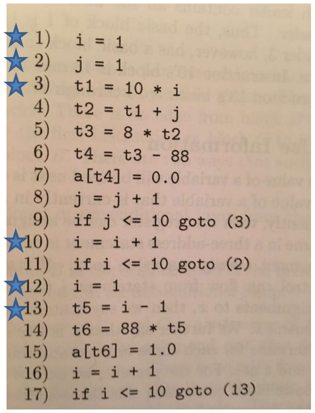
- **First instruction**
- **Any target of a jump**
- **Any instruction immediately following a jump**

Basic block starts at leader & ends at instruction immediately before a leader (or the last instruction)

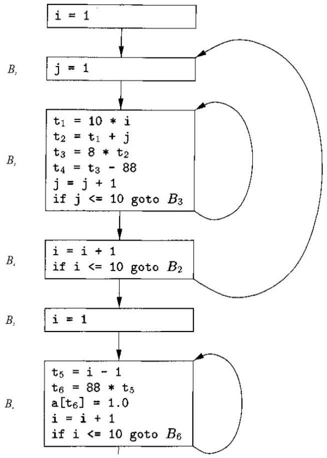


```
★ 1)  i = 1
★ 2)  j = 1
★ 3)  t1 = 10 * i
  4)  t2 = t1 + j
  5)  t3 = 8 * t2
  6)  t4 = t3 - 88
  7)  a[t4] = 0.0
  8)  j = j + 1
  9)  if j <= 10 goto (3)
★10)  i = i + 1
 11)  if i <= 10 goto (2)
★12)  i = 1
★13)  t5 = i - 1
 14)  t6 = 88 * t5
 15)  a[t6] = 1.0
 16)  i = i + 1
 17)  if i <= 10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

ALSU pp. 529-531



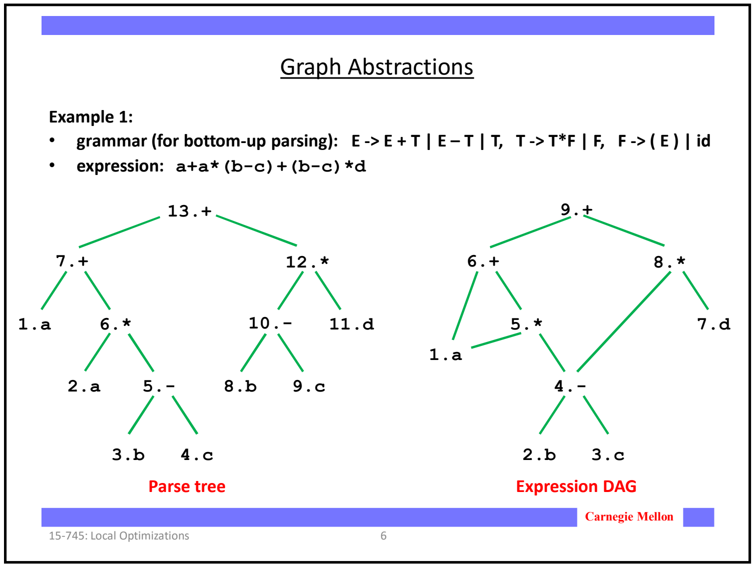
```
★ 1)  i = 1
★ 2)  j = 1
★ 3)  t1 = 10 * i
  4)  t2 = t1 + j
  5)  t3 = 8 * t2
  6)  t4 = t3 - 88
  7)  a[t4] = 0.0
  8)  j = j + 1
  9)  if j <= 10 goto (3)
★10)  i = i + 1
 11)  if i <= 10 goto (2)
★12)  i = 1
★13)  t5 = i - 1
 14)  t6 = 88 * t5
 15)  a[t6] = 1.0
 16)  i = i + 1
 17)  if i <= 10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



★ = Leader

II. Local Optimizations (within basic block)

- **Common subexpression elimination**
  - array expressions
  - field access in records
  - access to parameters

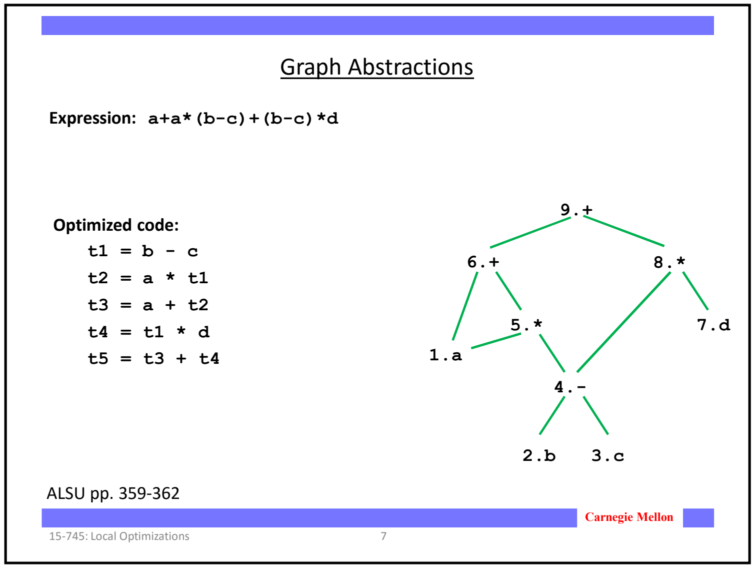


Graph Abstractions

Example 1:

- **grammar (for bottom-up parsing): E -> E + T | E – T | T, T -> T*F | F, F -> (E) | id**
- **expression: `a+a*(b-c)+(b-c)*d`**

Parse tree **Expression DAG**



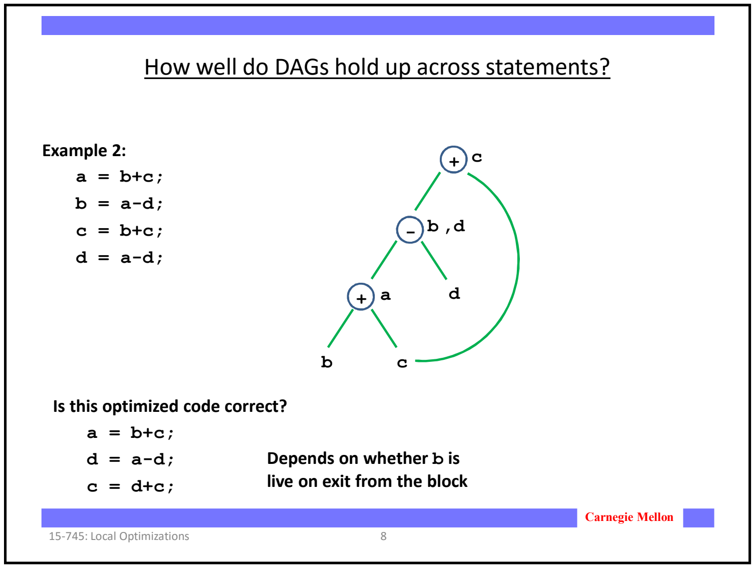
Graph Abstractions

Expression: `a+a*(b-c)+(b-c)*d`

Optimized code:

```
t1 = b - c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4
```

ALSU pp. 359-362



How well do DAGs hold up across statements?

Example 2:

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

Is this optimized code correct?

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

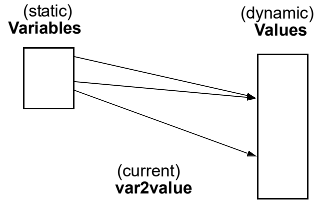
Depends on whether b is live on exit from the block

Critique of DAGs

- **Cause of problems**
  - Assignment statements
  - Value of variable depends on TIME
- **How to fix problem?**
  - build graph in order of execution
  - attach variable name to latest value
- **Final graph created is not very interesting**
  - Key: variable->value mapping across time
  - loses appeal of abstraction

III. Value Number: Another Abstraction

- John Cocke & Jack Schwartz in unpublished book: “Programming Languages and their Compilers”, (1970) (*ALSU pp. 360-362)*
- More explicit with respect to VALUES, and TIME



- **each value has its own “number”**
  - **common subexpression means same value number**
- var2value: current map of variable to value
  - used to determine the value number of current expression

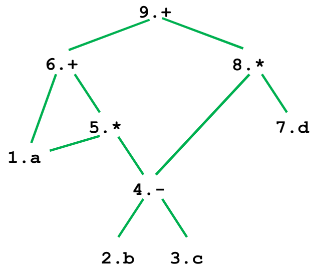
r1 + r2 => var2value(r1)+var2value(r2)

Value Numbering: Expression Example

Expression: `a+a*(b-c)+(b-c)*d`

Optimized code:

```
t4 = b - c
t5 = a * t4
t6 = a + t5
t8 = t4 * d
t9 = t6 + t8
```



Value Numbering Algorithm

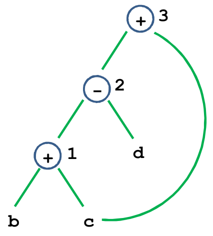
```
Data structure:
     VALUES = Table of
        expression      /* [OP, valnum1, valnum2] */
        var             /* name of variable currently holding expr */

For each instruction (dst = src1 OP src2) in execution order

  valnum1=var2value(src1); valnum2=var2value(src2)

  IF [OP, valnum1, valnum2] is in VALUES
     v = the index of expression
     Replace instruction with: dst = VALUES[v].var

  ELSE
     Add
        expression = [OP, valnum1, valnum2]
        var        = dst
     to VALUES
     v = index of new entry

  set_var2value (dst, v)
```

More Details

- **What are the initial values of the variables?**
  - values at beginning of the basic block
- **Possible implementations:**
  - Initialization: create “initial values” for all variables
  - Or dynamically create them as they are used
- **Implementation of VALUES and var2value: hash tables**

Value Numbering: Basic Block Example

```
a = b+c      t1 = b + c
             a = t1
b = a-d      t2 = t1 - d
             b = t2
c = b+c      t3 = t2 + c
             c = t3
d = a-d      d = t2
```



Q: Assigning to a temporary and then copying to the destination increases the number of instructions—so why do it?

A: If dst is overwritten later, would lose opportunity to eliminate common subexpression since no variable would hold the result

Value Numbering Algorithm

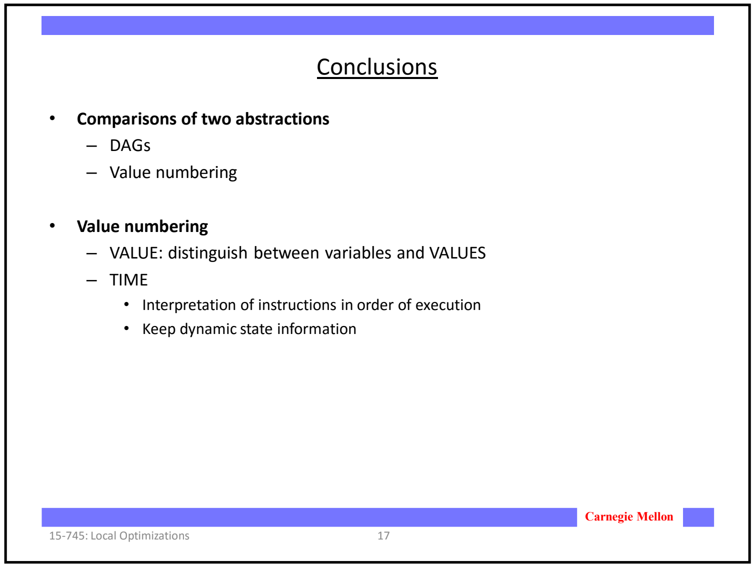
```
Data structure:
     VALUES = Table of
        expression     /* [OP, valnum1, valnum2] */
        var            /* name of variable currently holding expr */

For each instruction (dst = src1 OP src2) in execution order

  valnum1=var2value(src1); valnum2=var2value(src2)

  IF [OP, valnum1, valnum2] is in VALUES
     v = the index of expression
     Replace instruction with: dst = VALUES[v].var

  ELSE
     Add
        expression = [OP, valnum1, valnum2]
        var        = dst
     to VALUES
     v = index of new entry; tv is new temporary for v
     Replace instruction with: tv = VALUES[valnum1].var OP VALUES[valnum2].var
                               dst = tv

  set_var2value (dst, v)
```

Question

- **How do you extend value numbering to constant folding?**

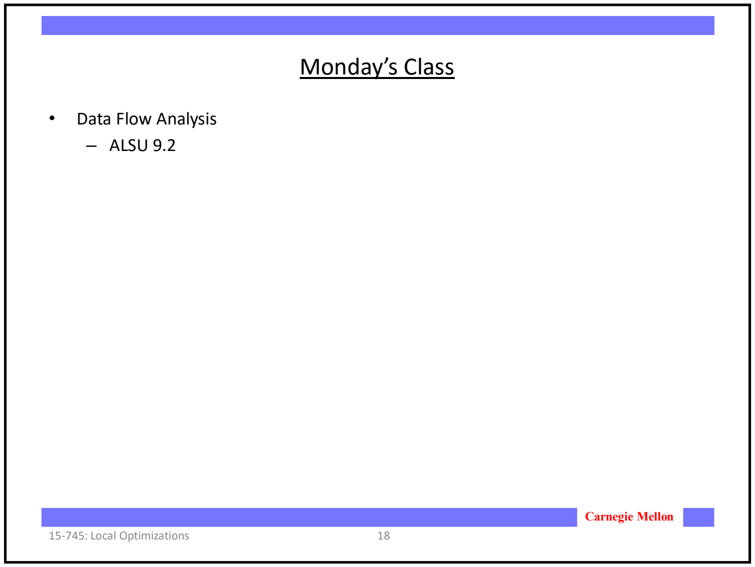
```
a = 1
b = 2
c = a+b
```

Answer: Can add a field to the VALUES table indicating when an expression is a constant and what its value is



Conclusions

- **Comparisons of two abstractions**
  - DAGs
  - Value numbering
- **Value numbering**
  - VALUE: distinguish between variables and VALUES
  - TIME
    - Interpretation of instructions in order of execution
    - Keep dynamic state information



Monday's Class

- Data Flow Analysis
  - ALSU 9.2