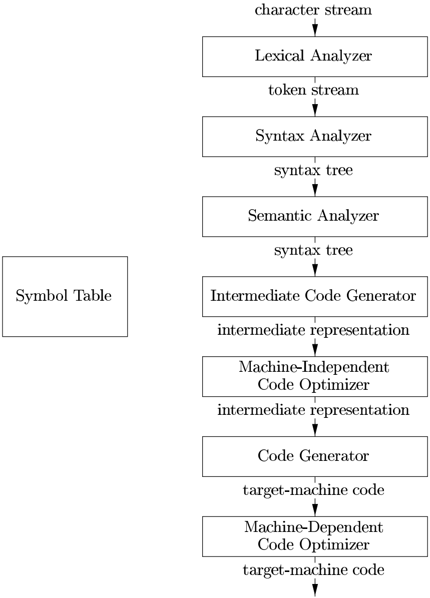
Basic Block Optimizations

The Compiler



Overview of Optimizations

- Optimizations are program transformations that seek to improve a program's resource utilization
  - Execution time (most often)
  - Space
  - Code size
  - Network messages sent, etc.
- Optimizations should not alter what the program computes.
  - The observable behaviour of the program must stay the same.

Classification of Optimizations

For imperative languages like `C`, `C++`, `Java`, etc. there are three granularities of optimizations

1. Local optimizations
   - Apply to a basic block in isolation
2. Global optimizations
   - Apply to a control-flow graph (of a method) in isolation
3. Inter-procedural optimizations
   - Apply across method boundaries.

Most compilers do 1, many do 2, few do 3.

Cost of Optimizations

- In practice, a conscious decision is made not to implement the fanciest optimization known.
- Why?
  - Some optimizations are hard to implement.
  - Some optimizations are costly in compilation time.
  - Some optimizations have low benefit.
  - Many fancy optimizations are all three!
- **Goal:** Maximum benefit for minimum cost
- The term ‘program optimization’ is a slight misnomer: we don’t necessarily get the ‘optimal’ code.
  - Program improvement is a more appropriate term.

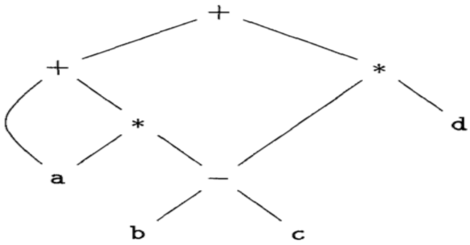
Local Optimizations

- The simplest form of optimizations.
- No need to analyze the entire procedure code, just look at a basic block.
  - It is a linear piece of code.
  - Analyzing and optimizing is easier.
  - Has local scope - and hence effect is limited.
- Inspite of being simple, it can often provide substantial benefits.

DAG representation of basic blocks

Recall: DAG representation of expressions

- leaves corresponding to atomic operands, and interior nodes corresponding to operators.
- A node N has multiple parents - N is a common subexpression.
- Example: `(a + a * (b - c)) + ((b - c) * d)`



DAG construction for a basic block

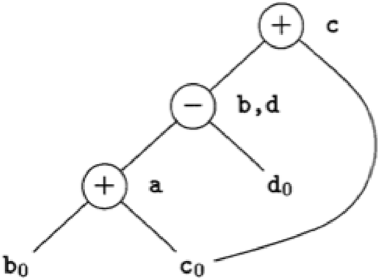
- There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
- There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.
- Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.
- Certain nodes are designated output nodes. These are the nodes whose variables are live on exit from the block.

Optimizations on the DAG

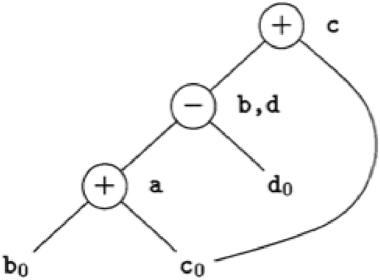
- Common sub-expression elimination.
- Eliminate dead code.
- Copy propagation
- Algebraic optimizations.

Finding common sub-expressions

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



Example (contd)

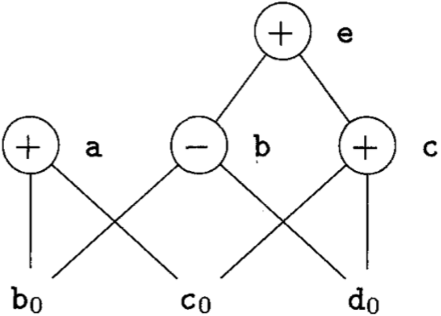



```
a = b + c
d = a - d
c = d + c

// if  b is live
b = d
```

Limitations of the DAG based CSE

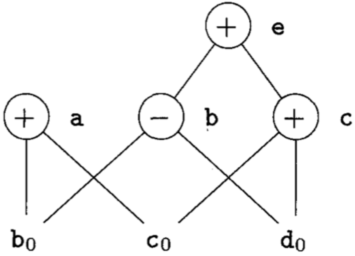
```
a = b + c
b = b - d
c = c + d
e = b + c
```



- The two occurrences of the sub-expressions `b + c` compute the same value.
- Value computed by `a` and `e` are the same.
- How to handle the algebraic identities?

Dead code elimination

- Delete any root from DAG that has no ancestors and is not live out (has no live out variable associated).
- Repeat previous step till no change.



- Assume `a` and `b` are live out.
- Remove first `e` and then `c`.
- `a` and `b` remain.

CSE via Algebraic identities

- Recall: In common sub-expression elimination, we want to reuse nodes that compute the same value.
- Recall: We mainly focussed on syntactic similarities.
- Can we go beyond that?

Similarities in the semantics - identity, inverse, zero

```
x + 0 = 0 + x = x

x * 1 = 1 * x = x

a && true = true && a = a

a || false = false || a = a

x * 0 = 0 * x = 0

0 / x = 0
```

Goal: apply arithmetic identities to eliminate computation.

Similarities in the semantics - strength reduction

```
x^2 = x * x

2 * x = x + x = x << 1

x/2 = x * 0.5 = x >> 1
```

Constant folding:

```
a = 5 * 2
->
a = 10
```

Goal: identify equivalence modulo <u>strength reduction</u> operations.

Algebraic properties

- Commutative: Say the operator * is commutative. x * y = y * x
- Associative: a + (b - c) = (a + b) - c

```
a = b + c
e = c + d + b
->
a = b + c
t = c + d
a = t + b
-> (assuming t is not used anywhere else)
a = b + c
e = a + d
```

- `a = b - 1; c = a + 1` $\rightarrow$ `c = b`

Copy Propagation

if `w = x` appears in a basic block, replace subsequent uses of `w` with `x`, until the next definition of `w`.

```
b = z + y
a = b
x = 2 * a
->
b = z + y
a = b
x = 2 * b
```

Only useful for enabling other optimizations

- Constant folding
- Dead code elimination
- Common sub-expression elimination

Copy Propagation and Constant Folding

```
a = 5
x = 2 * a
y = x + 6
t = x * y
->
a = 5
x = 10
y = 16
t = 160
```

Applying Local Optimizations

- Each local optimization does little by itself.
- Typically optimizations interact with each other.
    - Performing one optimization enables another.
- Optimizing compilers repeat optimizations until no improvement is possible.

An Example

Initial Code:

```
a = x ^ 2
b = 3
c = x
d = c * c
e = b * 2
f = a + d
g = e * f
```

An Example

Algebraic Properties (Strength Reduction):

```
a = x ^ 2
b = 3
c = x
d = c * c
e = b * 2
f = a + d
g = e * f
```

An Example

Algebraic Properties (Strength Reduction):

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

An Example

Copy Propagation:

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

An Example

Copy Propagation + Constant Folding:

```
a = x * x
b = 3
c = x
d = x * x
e = 6
f = a + d
g = e * f
```

An Example

Common Sub-expression Elimination:

```
a = x * x
b = 3
c = x
d = x * x
e = 6
f = a + d
g = e * f
```

An Example

Common Sub-expression Elimination:

```
a = x * x
b = 3
c = x
d = a
e = 6
f = a + d
g = e * f
```

An Example

Copy Propagation (again):

```
a = x * x
b = 3
c = x
d = a
e = 6
f = a + d
g = e * f
```

An Example

Copy Propagation (again):

```
a = x * x
b = 3
c = x
d = a
e = 6
f = a + a
g = 6 * f
```

An Example

Dead Code Elimination:

```
a = x * x
b = 3
c = x
d = a
e = 6
f = a + a
g = 6 * f
```

An Example

Final Code:

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

Representing Array accesses in the DAG

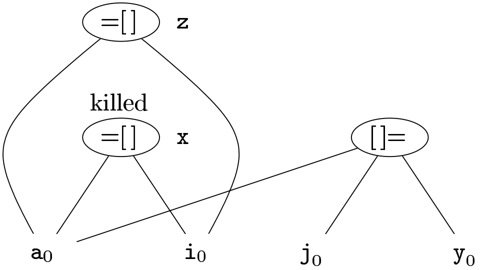
```
x = a[i]
a[j] = y
z = a[i]
```

Q: Is a[i] a common sub-expression?

- To represent assignment from an array, we will create a node with operator $=[]$ with two children representing the array name and index.
- To represent assignment to an array, we will create a node with operator $[]=$ with 3 children, representing the array name, index and RHS variable.
- An assignment to an array <u>kills</u> all previous nodes associated with the array.
- A <u>killed</u> node cannot receive any more labels; it cannot become a common sub-expression.

Representing Array accesses in the DAG

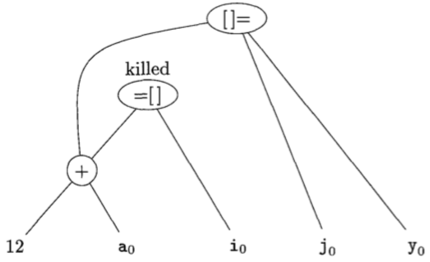
```
x = a[i]
a[j] = y
z = a[i]
```



Array representation (2)

```
b = a + 12
x = b[i]
b[j] = y
```

Assume that elements of 'a' are 4 bytes size



Home reading: How to handle pointers.

Peephole optimization

- A local optimization technique.
- Simplistic in nature, but effective in practise.
- Idea:
  - Keep a sliding window (called peephole)
  - Replace instruction sequences within the peephole by an efficient (shorter / faster / ...) sequence.

Peephole optimization

- The “peephole” is typically small.
- The code in the peephole need not be contiguous.
- Each improvement may lead to additional improvements.
- In general, we may have to make multiple passes.

Eliminating redundant loads and stores

```
Load a, R0
Store R0, a
```

Delete the pair of instructions. Always?

What if there is a label on the store instruction?

We need to be sure that the `Store` instruction and `Load` are executed as a pair.

Why would we have such stupid code?

Eliminating unreachable code

- An unlabelled statement after an unconditional jump – can be removed.

```
goto L2
INCR R0
L2:
```

Flow-of-control optimizations

- Naive code generation creates many jumps.
- Jumps to jumps can be short circuited.

  ```
  goto L1
  ...
  L1: goto L2
  ```

  Can be replaced with

  ```
  goto L2
  ...
  L1: goto L2
  ```

  Further optimizations on `L1` are possible.

  Similar situation with conditional jumps

  ```
  if (cond) goto L1
  ...
  L1: goto L2
  ```

Algebraic simplification and strength reduction

- Eliminate identity operations.
- Replace x^2 by $x * x$, and so on.
- Replace multiplication by a power of two (by left-shift) and division by a power of two (by right shift).

Peephole procedure

- First make a list of patterns that you want to replace with a list of target patterns.
- Identify the pattern in the code and do the replacement.
- Iterate till you are done.
- Can be efficiently done on an DAG.
- No guarantees about optimality.
- Most of the peephole optimizations guarantee improvement.