9. Code Optimization

- "Optimizations" are only improvements w.r.t. code size, execution time, memory requirements. Code cannot be optimal in all cases.

- Target machine independent optimizations can be carried out in the front end.

- Target machine dependent optimization must be carried out in the back end.

- Hence optimizing compilers follow this – anyway recommendable – division into front end and back end. The interface is the syntax tree plus the symbol table.

Dead Code Elimination

- Based on following identities for if and similar for CASE:

  if true then A else B = A
  if false then A else B = B

  Examples:

  ```
  if debug then … else …
  case system of
      ios: … ;
      android: … ;
      …
  end
  ```

- Useful for languages where a preprocessor with conditional compilation is not available.

Algebraic Identities

- Based on arithmetic and boolean identities:

x + 0 = x	x * 0 = 0
x * 2 = x + x	x * 1 = x
b or false = b	b or true = true

  These also include replacing multiplications and divisions by powers of 2 with shifts, etc.

- Such expressions can either occur in the source code, typically with symbolic constants, or are generated by the compiler, e.g.
  - adding zero for accessing the first field of a record or
  - multiplying an array index by one for indexing strings.

- Constant folding is the simplification of constant expressions, done by the PO compiler as part of delayed code generation.

- Dead code elimination is a special case of an algebraic identity.

Instruction Selection

- CISC architecture typically feature instructions which combine
  - multiplication and addition (for indexing arrays of records)
  - addition, comparison, and conditional branch (for FOR loops).

  Selection of the best instruction or instruction sequence is a major issue for CISC architectures. If several alternatives are possible, individual instructions are assigned costs and the instruction(s) with the least total cost is (are) selected. This may involve optimization algorithms on an intermediate representation.

- Instructions supporting specific program constructs can be generated directly for those program constructs.

- Alternatively, instruction selection can be improved by delayed code generation and peephole optimization.

Peephole Optimization

- Peephole optimization examines a short sequence of instructions, the peephole, replacing those by more efficient ones, possibly enabling further optimization.

- Some compilers generate code without algebraic simplification and then offer (optional) peephole optimization.

- Peephole optimization can remove some redundant loads and stores:

```
x := e;          STW R.e,x,0     (* assuming e is in R.e *)
y := x + 2       LDW R.x,x,0     (* x is now in R.x *)
```

- Peephole optimization can also be used to eliminate jumps to jumps:

```
if a then
   if b then S else T
else U
```

Common Subexpression Elimination

- Example of a transformation which could be done by the programmer as well:

  x := (a + b) / c; y := (a + b) / d = u := a + b; x := u / c; y := u / d

- Example of a transformation which can be done only by the compiler:

  a[i, j] := a[i, j] + b[i, j]

  Here the computation of i * size(…) + j is repeated 3 times.

- Common subexpression elimination relies on representing the structure of the program not as a tree, but as a directed acyclic graph.

Loop Invariant Code Motion

- Repeated evaluation of a common subexpression occurs also in loops, even though the expression appears only once:

```
while i > x + z do          =          u := x + z;
    i := i – 1                         while i > u do
                                           i := i – 1
```

- This transformation could also be done by the programmer, but others can be done only by the compiler:

```
for i := 0 to n – 1 do
    for j := 0 to m – 1 do
        a[i, j] := 0
```

Here i * size(…) is calculated in the inner loop each n times.

Strength Reduction

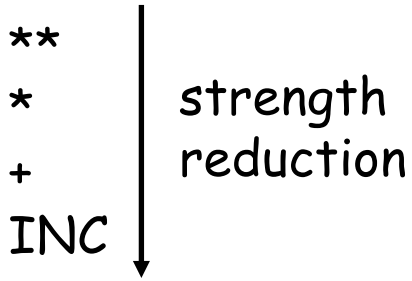
- The idea is to replace arithmetic operators within loops by "weaker" ones:

  for i := 0 to N – 1 do a[i] := x

  Here, rather than computing adr(a[i]) by

  adr(a[i]) = adr(a) + i * size(…)

  a variable u can be initialized to adr(a[0]) and incremented by size(…) in the loop. Hence * is reduced to +.



- Example where + can be reduced to INC:

```
for i := 0 to N – 1 do          =        u := x;
   a[i] := x + i                         for i := 0 to N - 1 do
                                             a[i] := u; u := u + 1
```

Register Allocation

- Major optimization technique for RISC architectures.
  Straightforward and optimized code generated for:

  z := (x – y) * (x + y); y := x

LDW 1,0,x	R1:=x	LDW 1,0,x	R1:=x
LDW 2,0,y	R2:=y	LDW 2,0,y	R2:=y
SUB 1,1,2	R1:=R1–R2	SUB 3,1,2	R3:=R1–R2
LDW 2,0,x	R2:=x	ADD 4,1,2	R4:=R1+R2
LDW 3,0,y	R3:=y	MUL 5,3,4	R5:=R3*R4
ADD 2,2,3	R2:=R2+R3	STW 5,0,z	z:=R5
MUL 1,1,2	R1:=R1*R2	STW 1,0,y	y:=R1
STW 1,0,z	z:=R1		
LDW 1,0,x	R1:=x		
STW 1,0,y	y:=R1		

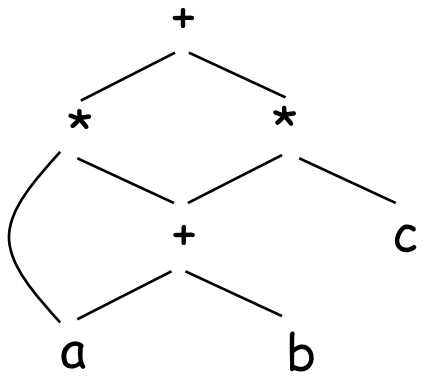
  Register allocation is based on determining the range of relevance of expressions. Further issues:

  - local variables, in particular loop indices, kept in register
  - parameters passed in registers

Common Subexpression Elimination …

- By using a directed acyclic graph instead of a tree for representing the abstract syntax, common subexpressions can be shared:

a * (a + b) + (a + b) * c



- For this, NewExp needs to be modified to return an existing node if one already exists:

$exp(n_0)$	$\rightarrow$	$term(n_1)$	$n_0 := n_1$
	|	$exp(n_1)$ "+" $term(n_2)$	NewExp(PlusSym,n_1,n_2,n_0)
	|	$exp(n_1)$ "-" $term(n_2)$	NewExp(MinusSym,n_1,n_2,n_0)
$term(n_0)$	$\rightarrow$	$factor(n_1)$	$n_0 := n_1$
	|	$term(n_1)$ "*" $factor(n_2)$	NewExp(TimesSym,n_1,n_2,n_0)
	|	$term(n_1)$ "/" $factor(n_2)$	NewExp(DivSym,n_1,n_2,n_0)
$factor(n_0)$	$\rightarrow$	identifier(id)	NewIdent(id, n_0)
	|	"(" $exp(n_1)$ ")"	$n_0 := n_1$

… Common Subexpression Elimination

- The steps in constructing the DAG are (last parameter is result):

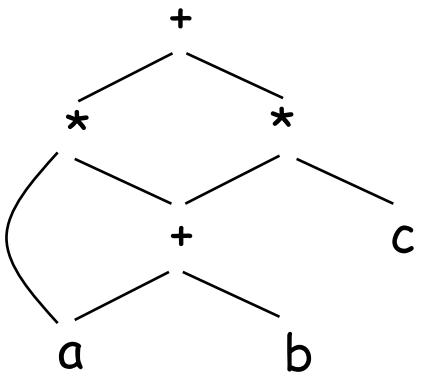
NewIdent(a, p_0)
NewIdent(a, p_0)
NewIdent(b, p_1)
NewExp(PlusSym, p_0, p_1, p_2)
NewExp(TimesSym, p_0, p_2, p_3)
NewIdent(a, p_0)
NewIdent(b, p_1)
NewExp(PlusSym, p_0, p_1, p_2)
NewIdent(c, p_4)
NewExp(TimesSym, p_2, p_4, p_5)
NewExp(PlusSym, p_3, p_5, p_6)

0	a		
1	b		
2	+	0	1
3	*	0	2
4	c		
5	*	2	4
6	+	3	5

- In order to look up existing nodes, the DAG can be stored in an array that is searched linearly. The entries in the array are either leafs or operators together with indices of arguments. In order to speed up the search, hashing can be used.

Three-Address Code

- Three-address code is a linearized representation of the DAG, where interior nodes are named:




```
t1 := a + b
t2 := a * t1
t3 := t1 * c
t4 := t2 + t3
```

- Three-address code can be used as an intermediate representation between the front-end and back-end. Besides assignments with unary operators (e.g t1 := -a) and binary operators, copy instructions (e.g. a := t1), three-address code has:
  - Unconditional jump "goto L"
  - Conditional jump "if x goto L", "ifnot x goto L", "if x relop y goto L",
  - Procedure call with parameters "param x", "call p", "x := call p"
  - Indexed copy "x[i] := y", "y := x[i]"
  - Pointer operations "x := &y", "x := *y", "*x := y"

Basic Blocks and Flow Graphs …

- Basic blocks are longest sequences of three-address code such that:
  - only the first instruction can be a jump target and
  - only the last instruction can be a jump,

  That is, basic blocks do not have a jump in the middle.

- The flow graph has basic blocks as nodes and an edge between blocks A and B if:
  - block B follows immediately block A in the code or
  - block A ends with a jump to block B.

  An empty ENTRY block with no incoming nodes and an empty EXIT block with no outgoing nodes are added.

… Basic Blocks and Flow Graphs …

```
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
```

This could have be code generated from a program that sets a to the identity matrix:

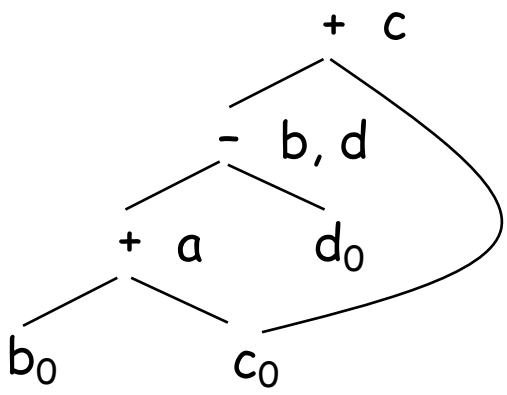
```
for i := 1 to 10 do
        for j := 1 to 10 do
                a[i, j] := 0.0;
for i := 1 to 10 do
        a[i, i] := 1.0
```

Determine the flow graph!

… Basic Blocks and Flow Graphs

- A number of optimization techniques can be applied. Common subexpression elimination can be extended to basic blocks. As variables may overwritten, their values need to be distinguished. In the DAG below, initial values have 0 subscripts:

```
a := b + c
b := a – d
c := b + c
d := a – d
```



- A global data flow analysis can determine which variables of a block are needed in it's successor blocks. Variables that are not needed do not need to be stored (e.g. if "b" above is not used in successors).
- Basic blocks can be linearized in any order. This can be used to minimize the number of jump instructions.

Data-Flow Analysis

- Given a representation in three-instruction form with basic blocks, each point before and after each statement is considered.
  - A statement assigning to variable x is said to define x.
  - A statement inspecting variable x is said to use x.
- The data-flow values before and after each statement s are denoted by IN[s] and OUT[s]. These are constrained by
  - The meaning of statements, as given by f_s:

    $OUT[s] = f_s(IN[s])$
  - The control flow within a basic block:

    $IN[s_{i+1}] = OUT[s_i]$ for a basic block with s1 … sn
  - The control flow between basic blocks:

    $IN[B] = \bigcup_{P:\text{predecessor}(B)} OUT[P]$

    or equivalently

    $OUT[B] = \bigcup_{S:\text{successor}(B)} IN[S]$

Reaching Definitions

- The data-flow values can be definitions of variables. Considering

  s1: a := b + c

  s2: a := 5

  we say that statement s1 generates a definition of variable a and statement s2 generates a new definition of a but kills the previous one. More generally:

  $f_d(x) = gen_d \cup (x - kill_d)$