Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-15/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 32

- Code generation and optimization (2)

Summary

- A Code Generator
  - Register allocation and assignment
    - Graph coloring
  - Instruction selection
    - Tree transducer
- An overview on Dataflow Analysis and some Global Optimization techniques

A Simple Code Generator

- Algorithm for generating target code for a basic block (sequence of three-address statements) using next-use information
- Critical issue: how to use registers. Several competing uses:
  - To store operands of a target code operation
  - Registers make good temporaries
  - To hold (global) values computed in a block and used in another
  - To help runtime storage management (stack pointer, …)
- The algorithm will check if operands of three-address code are available in registers to avoid unnecessary stores and loads.

A Simple Code Generator (2)

- We assume that
  - A set of register can be used for values used within the block
  - The order of statements in the block is fixed
  - Each three-address operator corresponds to a single machine instruction
  - Machine instructions take operands in registers and leave the result in a register
- The algorithm makes use of ***address*** and ***register descriptors***, and of function ***getreg()*** such that *getreg(x = y OP z)* returns the three registers to be used for x, y and z.

Register and Address Descriptors

- A *register descriptor RD* keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.
- An *address descriptor AD* keeps track of the location where the current value of the name can be found at run time, e.g. a register, stack location, memory address, etc.
- Eg:

`LD R0,a`	$RD(\mathtt{R0}) = \{\mathtt{a}\}$, $AD(\mathtt{a}) = AD(\mathtt{a}) \cup \{\mathtt{R0}\}$
`ST a,R0`	$RD(\mathtt{R0}) = RD(\mathtt{R0}) \cup \{\mathtt{a}\}$, $AD(\mathtt{a}) = \{\mathtt{R0}\}$

The Code Generation Algorithm

For each statement *x* := *y* op *z*

1. Use *getreg*(x := *y* OP *z*) to get registers *Rx*, *Ry* and *Rz*
2. If *Ry* $\notin$ *AD*(y) then emit **LD** *Ry*, *y'*
   where *y'* $\in$ *AD*(y), preferably a register
3. If *Rz* $\notin$ *AD*(z) then emit **LD** *Rz*, *z'*
   where *z'* $\in$ *AD*(z), preferably a register
4. emit **OP** *Rx, Ry, Rz*
5. *Update the descriptors for the* **LD** *statements*
6. *RD(Rx) = {x}, AD(x) = {Rx}, remove Rx from other AD's*

The Code Generation Algorithm

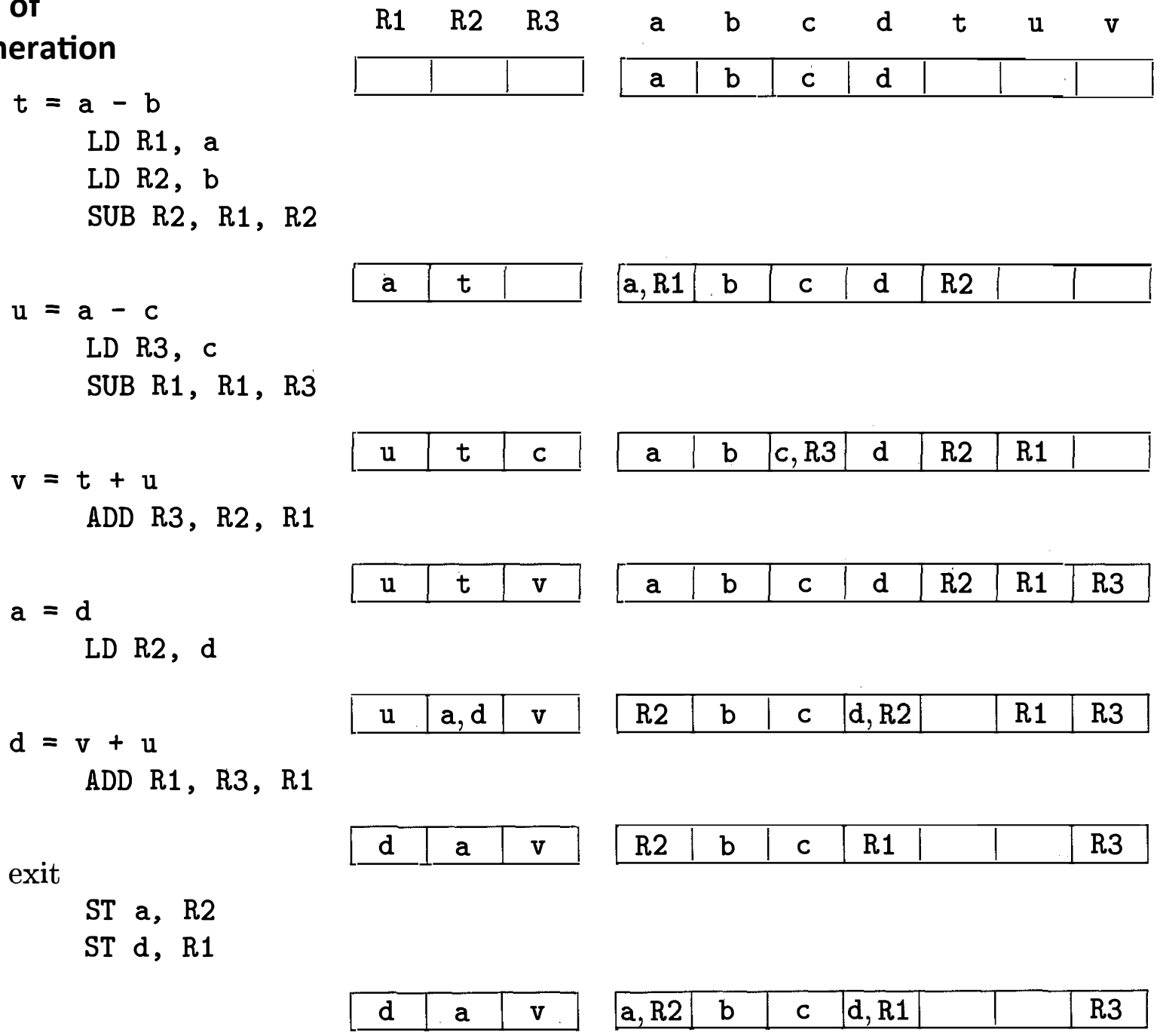
For each copy statement *x :=y*

1. Use *getreg*(x := *y*) to get register *Ry* (= *Rx*)
2. If *Ry* $\notin$ *AD*(y) then emit **LD** *Ry*, *y'*
   where *y'* $\in$ *AD*(y), preferably a register
3. *Update the descriptors for operation* **LD**
4. *RD(Ry) = RD(Ry)* $\cup${*x*}*, AD(x) = {Ry}*

At the end of the basic block

1. For each live variable x, if *x* $\notin$ *AD*(x)
   emit **ST x,R**, where R $\in$ *AD*(x)

Example of code generation



The *getreg* algorithm

To compute *getreg*(x := *y* OP *z*)

1. If *y* is stored in a register *R*, return it as *Ry*
2. If *y* is not in a register, but exists *R* empty, return it as *Ry*
3. If y is not in a register and no register is empty, consider *R* and check any variable $v \in RD(R)$
   a. If *AD(v)* does not contain only *R*, OK.
   b. If *v = x* and *x* is not an operand in this instruction, OK.
   c. If *v* is not used later, then OK.
   d. Otherwise emit **`ST v,R`** this is a *spill*

   Choose *R* that minimizes the number of *spills* and return it as *Ry*
4. Same algorithm for determining *Rz*
5. For *Rx*, similar algorithm, but
   a. Any register containing *x* only is OK
   b. It is possible to return *Ry* for *Rx* if y is no more used and if RD(*Ry*) *={y}.* Similarly for *Rz.*

To compute *getreg*(x := *y*)

1. Choose *Ry* as above
2. Choose *Rx = Ry*

Peephole Optimization

- Examines a short sequence of target instructions (not necessarily contiguous) in a sliding window (*peephole*)
- Replaces the instructions by a faster and/or shorter sequence when possible
- Each improvement may enable other improvements
- Applied to intermediate code or target code
- Typical optimizations:
  - Redundant instruction elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms

Peephole Opt: Eliminating Redundant Loads and Stores

- Consider

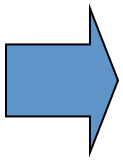
```
LD R0,a
ST a,R0
```

- The second instruction can be deleted, but only if it is not labeled with a target label
  - Thus both instructions should be in the same block
- In general a peephole represents a sequence of instructions with at most one entry point

Peephole Optimization: Deleting Unreachable Code

```
debug := 0
…

if debug != 1 goto L2
…
Print debug info
…
L2:
```



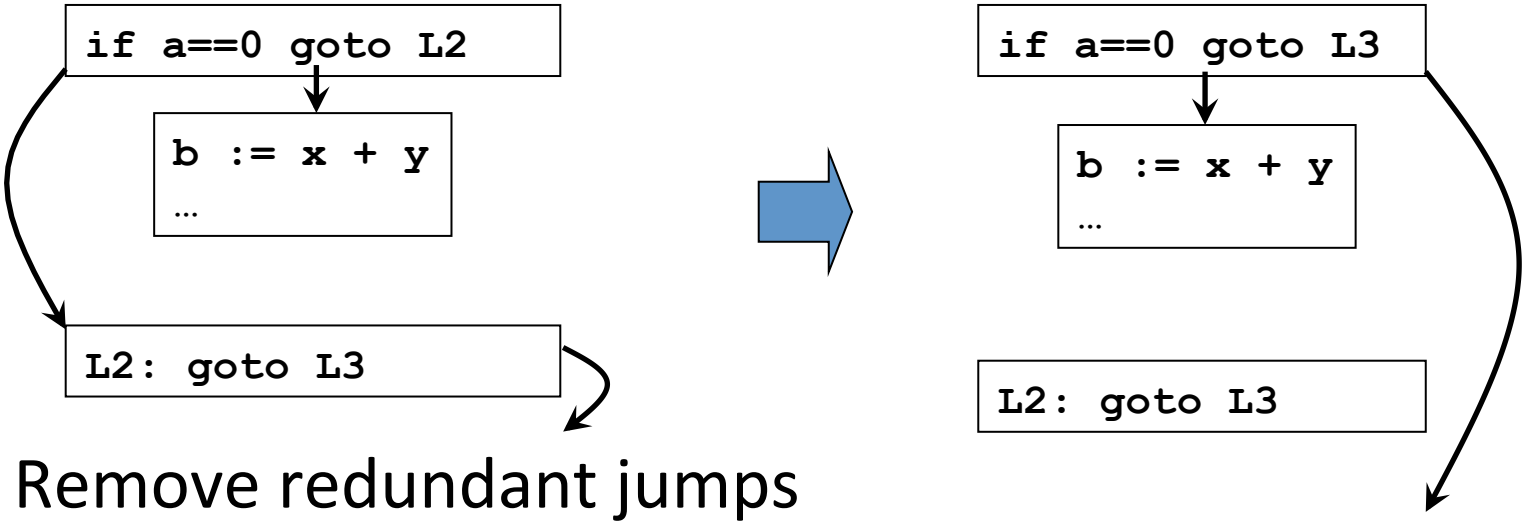
```
debug := 0
…

if 0 != 1 goto L2
…
Print debug info
…
L2:
```

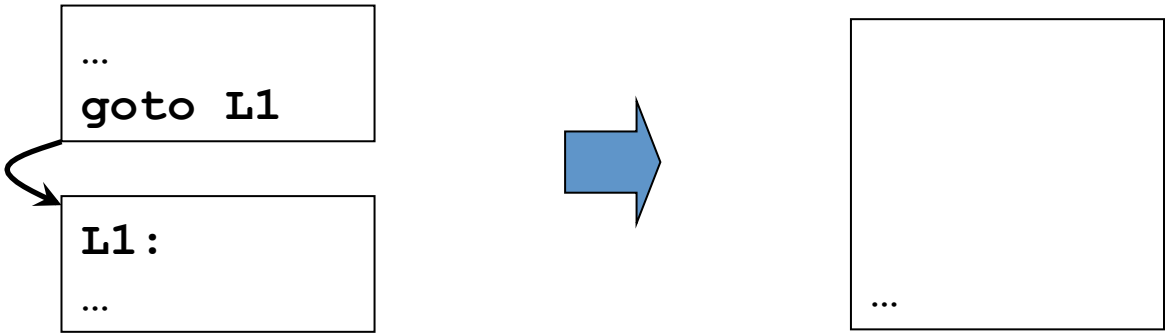
- Constant propagation allows to evaluate the guard
- Unreachable statements can be deleted one at a time

Peephole Optimization: Flow-of-Control Optimizations

- Branch Chaining: Shorten chain of branches by modifying target labels



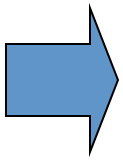
- Remove redundant jumps



Other Peephole Optimizations

- *Reduction in strength*: replace expensive arithmetic operations with cheaper ones

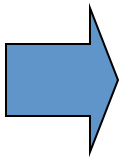
```
…
a := x ^ 2
b := y / 8
```



```
…
a := x * x
b := y >> 3
```

- Utilize machine idioms

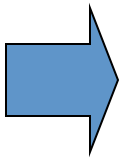
```
…
a := a + 1
```



```
…
inc a
```

- Algebraic simplifications

```
…
a := a + 0
b := b * 1
```



```
…
```

Register Allocation and Assignment

- The code generation algorithm based on *getreg()* is not optimal
  - All live variables in registers are stored (flushed) at the end of a block: this could be not necessary
- *Global register allocation* assigns variables to limited number of available registers and attempts to keep these registers consistent across basic block boundaries
  - Keeping variables in registers in looping code can result in big savings
- Groups of registers can be dedicated to certain values (base addresses, arithmetics, stack pointer)
  - Simpler code generator, but less efficient use of registers

Allocating Registers in Loops: Usage Counts

- Suppose
  - not storing a variable *x* has a benefit of 2
  - accessing a variable in register instead of in memory has benefit 1
- Let
  - *use*(*x*, *B*) = number of uses of *x* in *B* before assignment
  - *live*(*x*, *B*) = *1* if *x* is assigned in *B* and live on exit from *B*
- Then the (approximate) *benefit* of allocating a register to a variable *x* within a loop *L* is

$$\sum_{B \in L} (\mathit{use}(x, B) + 2\ \mathit{live}(x, B))$$

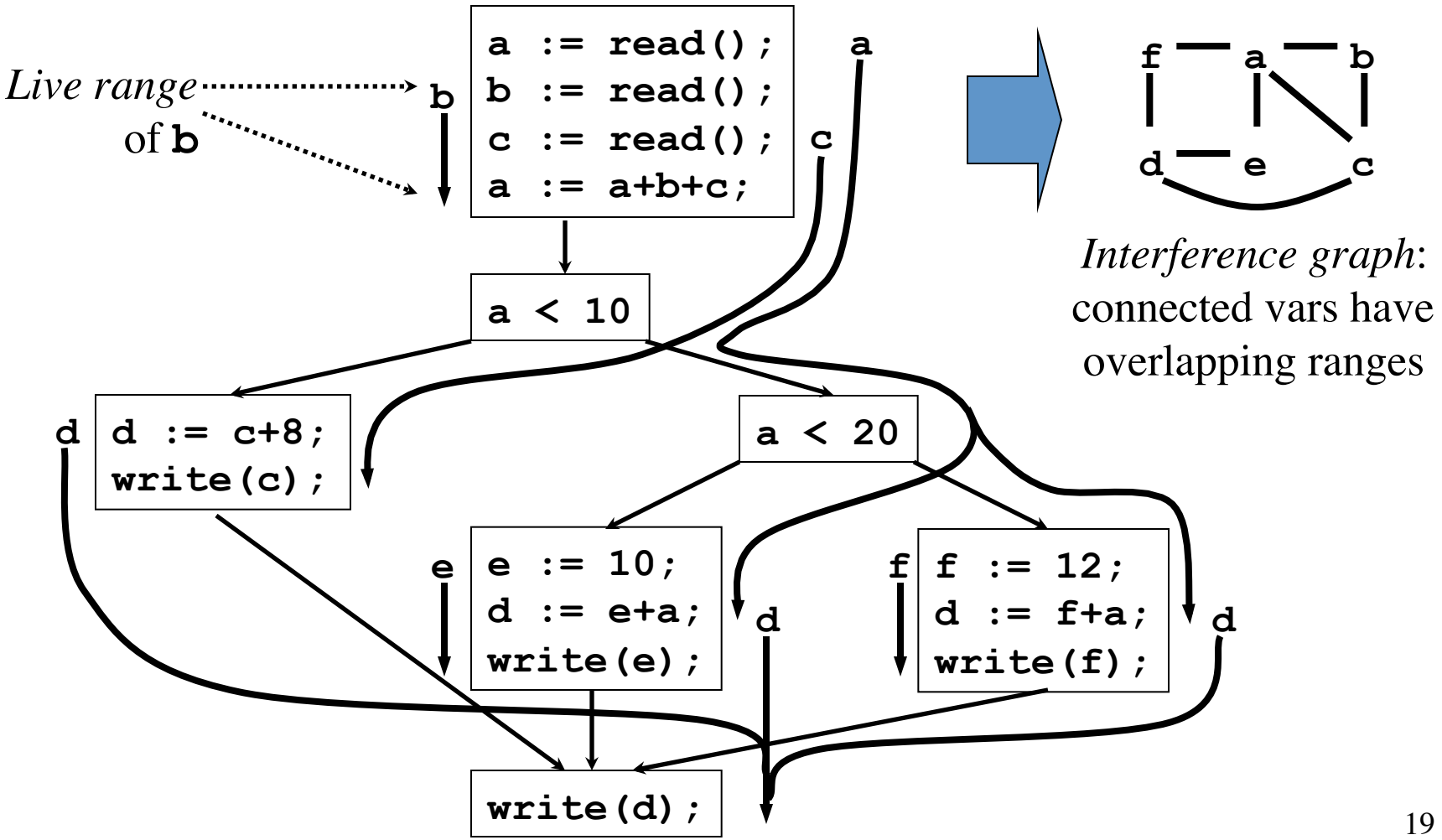
Global Register Allocation with Graph Coloring

- When a register is needed but all available registers are in use, the content of one of the used registers must be stored (*spilled*) to free a register
- Graph coloring allocates registers and attempts to minimize the cost of spills
- Build a *conflict graph* (*interference graph*): two variables have an edge if one is live where the other is defined
- Find a *k*-coloring for the graph, with *k* the number of registers

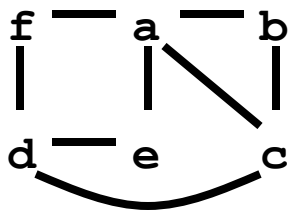
Register Allocation with Graph Coloring: Example

```
a := read();
b := read();
c := read();
a := a + b + c;
if (a < 10) {
    d := c + 8;
    write(c);
} else if (a < 20) {
    e := 10;
    d := e + a;
    write(e);
} else {
    f := 12;
    d := f + a;
    write(f);
}
write(d);
```

Register Allocation with Graph Coloring: Live Ranges

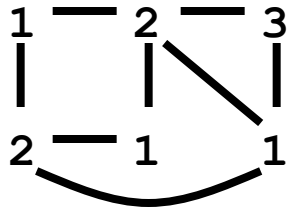


Register Allocation with Graph Coloring: Solution



Interference graph

Solve



Three registers:

```
a = r2
b = r3
c = r1
d = r2
e = r1
f = r1
```

```
r2 := read();
r3 := read();
r1 := read();
r2 := r2 + r3 + r1;
if (r2 < 10) {
    r2 := r1 + 8;
    write(r1);
} else if (r2 < 20) {
    r1 := 10;
    r2 := r1 + r2;
    write(r1);
} else {
    r1 := 12;
    r2 := r1 + r2;
    write(r1);
}
write(r2);
```

On Instruction Selection by Tree Rewriting

- Our simple algorithm uses a trivial Instruction Selection
- In practice it is a difficult problem, mainly for CISC machines with rich addressing mode
- Tree-rewriting rules can be used effectively for specifying the translation from IR to target code
- Tree-translation schemes can be handled with techniques similar to syntax-directed definitions: can be the basis of code-generator generators
- Start with a tree representing the IR code, and reduce it using the rules, producing the target code as associated actions

Code generation using a Tree Translation Scheme

1)	$R_i \leftarrow C_a$	{ LD Ri, #a }
2)	$R_i \leftarrow M_x$	{ LD Ri, x }
3)	$M \leftarrow$ = (M_x, R_i)	{ ST x, Ri }
4)	$M \leftarrow$ = (ind(R_i), R_j)	{ ST *Ri, Rj }
5)	$R_i \leftarrow$ ind(+(C_a, R_j))	{ LD Ri, a(Rj) }
6)	$R_i \leftarrow$ +(R_i, ind(+(C_a, R_j)))	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \leftarrow$ +(R_i, R_j)	{ ADD Ri, Ri, Rj }
8)	$R_i \leftarrow$ +(R_i, C_1)	{ INC Ri }

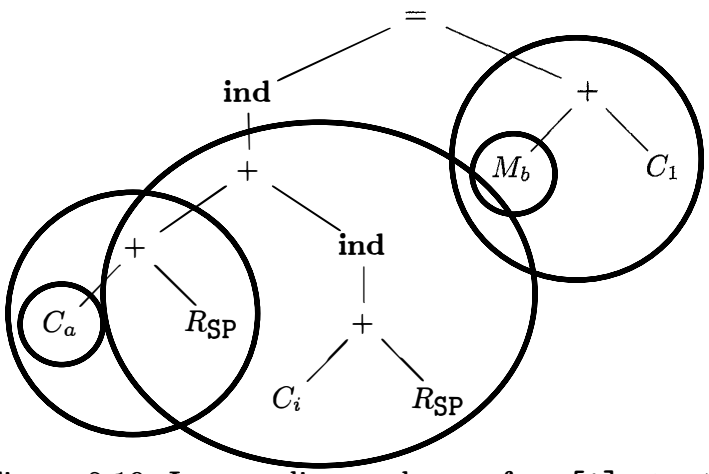


Figure 8.19: Intermediate-code tree for `a[i] = b + 1`

```
LD   R0, #a
ADD  R0, R0, SP
ADD  R0, R0, i(SP)
LD   R1, b
INC  R1
ST   *R0, R1
```

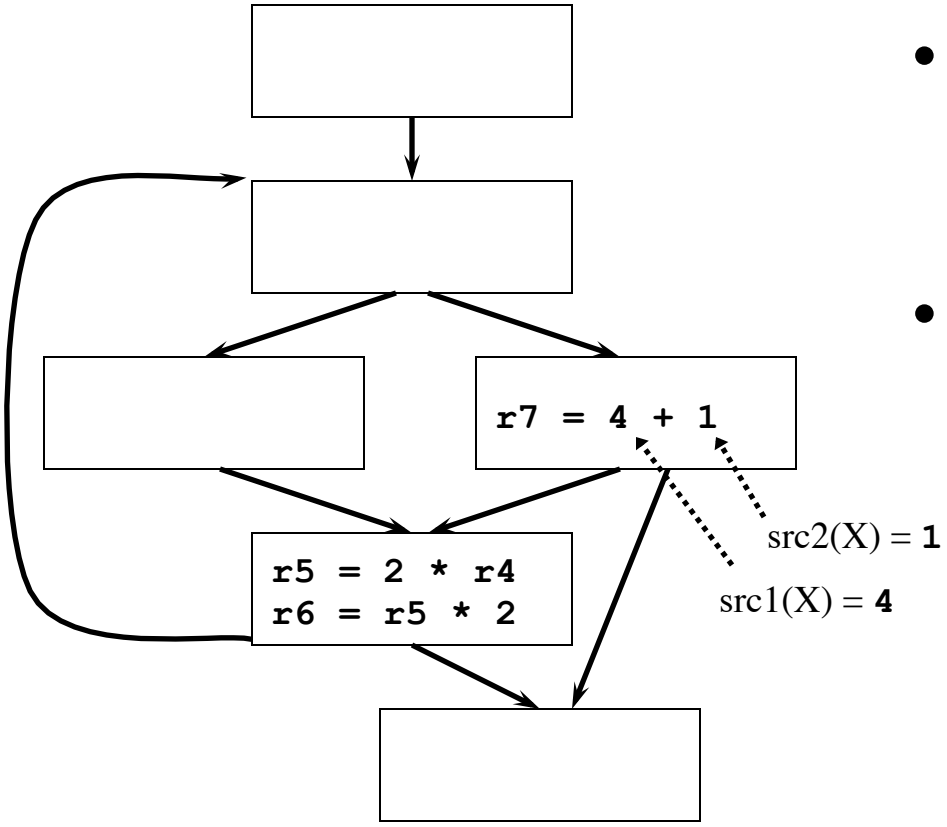
Tree matching

- Various algorithms for pattern matching and general tree matching
- LR parsing on a string (prefix) representation of trees
  - Uses a syntax-directed translation scheme encoding the tree-translation scheme
  - Highly ambiguous. In absence of costs, *reduce-reduce conflicts* choose the longer reduction, *shift-reduce* prefer shift
- General tree matching transforming templates into sets of strings, and by matching multiple strings in parallel
- We can associate costs with the tree-rewriting rules and apply dynamic programming to obtain an optimal instruction selection

Classic Examples of Local and Global Code Optimizations

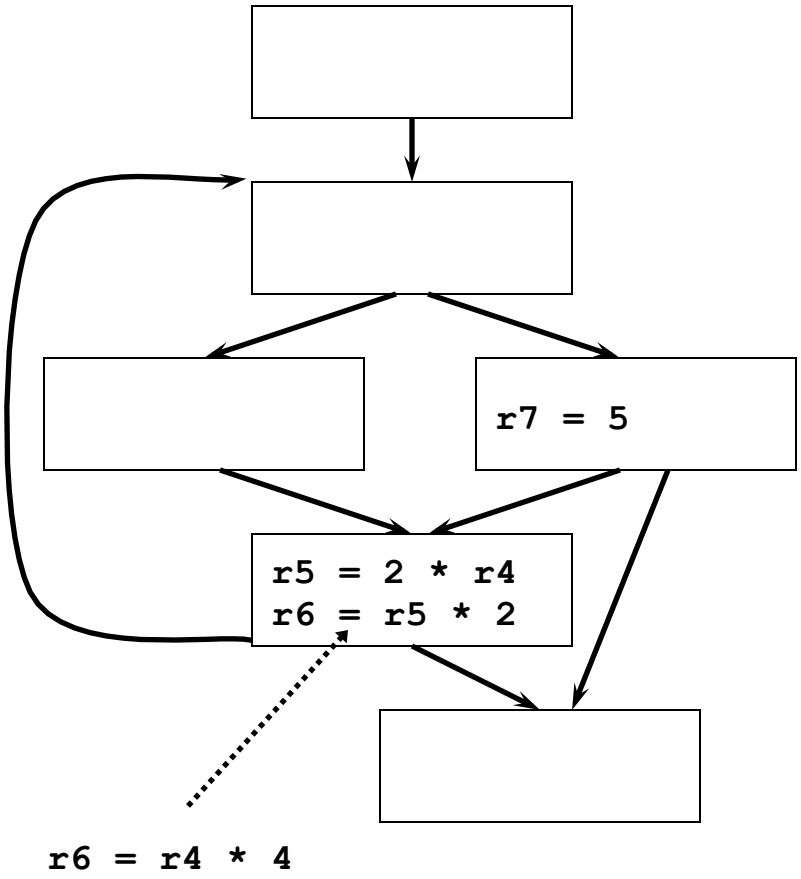
- Local
  - Constant folding
  - Constant combining
  - Strength reduction
  - Constant propagation
  - Common subexpression elimination
  - Backward copy propagation
- Global – based on data flow analysis
  - Dead code elimination
  - Constant propagation
  - Forward copy propagation
  - Common subexpression elimination
  - Code motion
  - Loop strength reduction
  - Induction variable elimination

Local: Constant Folding



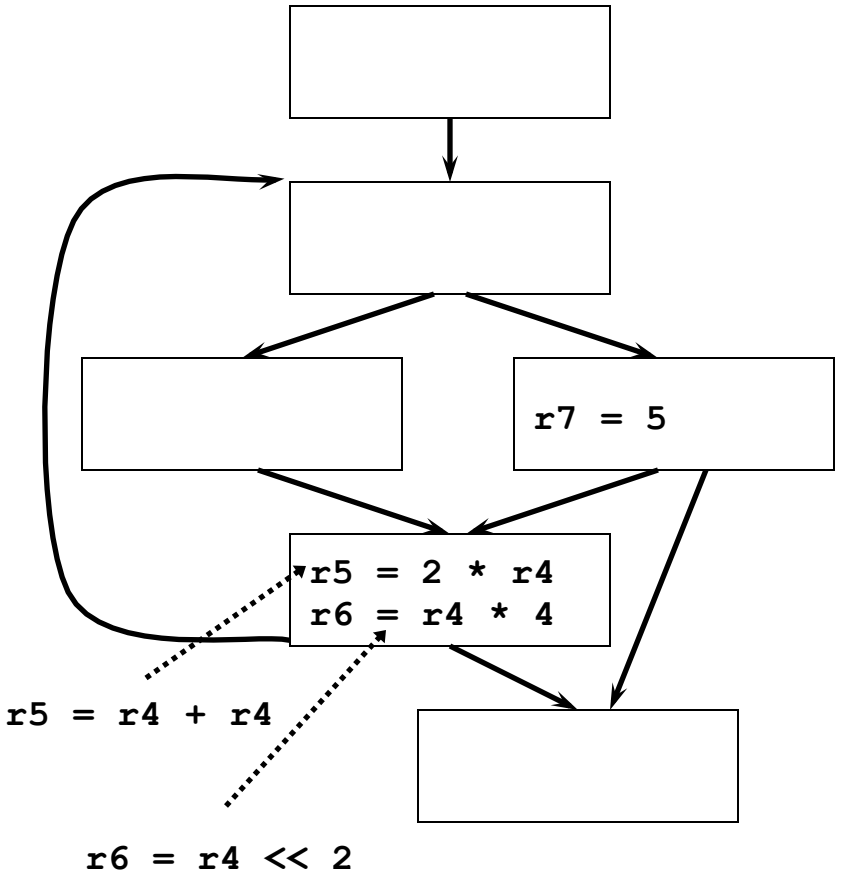
- Goal: eliminate unnecessary operations
- Rules:
  1. X is an arithmetic operation
  2. If src1(X) and src2(X) are constant, then change X by applying the operation

Local: Constant Combining



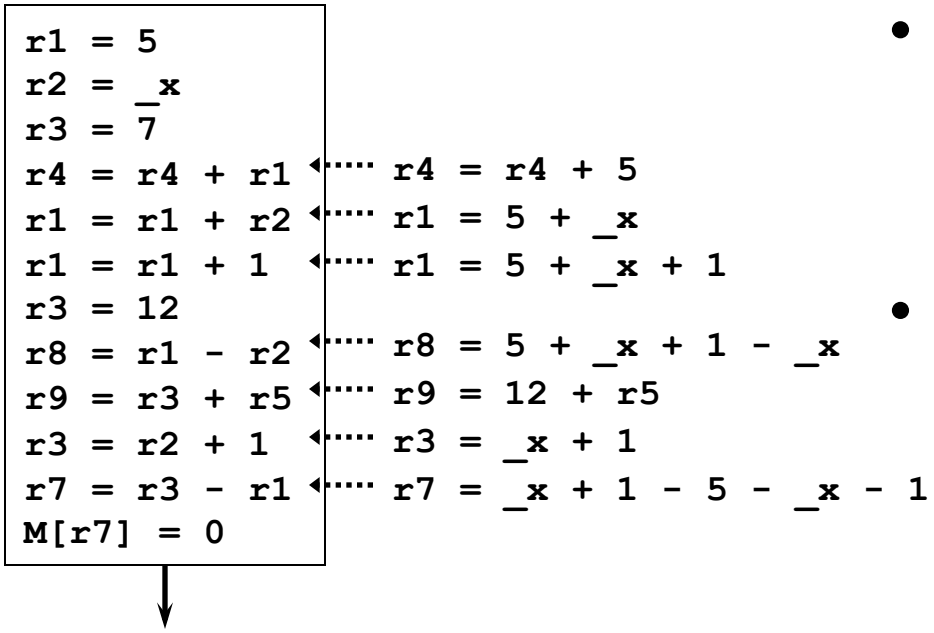
- Goal: eliminate unnecessary operations
  - First operation often becomes dead after constant combining
- Rules:
  1. Operations X and Y in same basic block
  2. X and Y have at least one literal src
  3. Y uses dest(X)
  4. None of the srcs of X have defs between X and Y (excluding Y)

Local: Strength Reduction



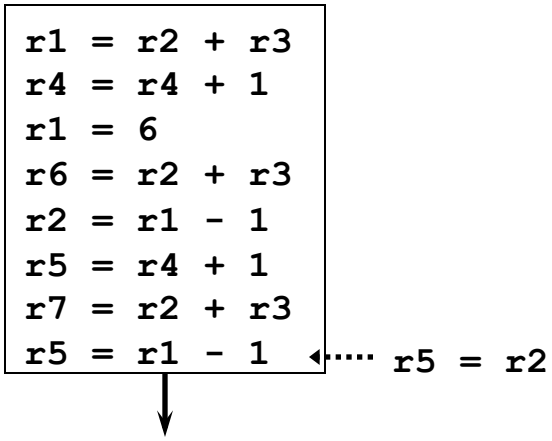
- Goal: replace expensive operations with cheaper ones
- Rules (common):
  1. X is an multiplication operation where src1(X) or src2(X) is a const 2^k integer literal
  2. Change X by using shift operation
  3. For k=1 can use add

Local: Constant Propagation



- Goal: replace register uses with literals (constants) in a single basic block
- Rules:
  1. Operation X is a move to register with src1(X) literal
  2. Operation Y uses dest(X)
  3. There is no def of dest(X) between X and Y (excluding defs at X and Y)
  4. Replace dest(X) in Y with src1(X)

Local: Common Subexpression Elimination (CSE)



- Goal: eliminate re-computations of an expression
  - More efficient code
  - Resulting moves can get copy propagated (see later)
- Rules:
  1. Operations X and Y have the same opcode and Y follows X
  2. src(X) = src(Y) for all srcs
  3. For all srcs, no def of a src between X and Y (excluding Y)
  4. No def of dest(X) between X and Y (excluding X and Y)
  5. Replace Y with
     dest(Y) = dest(X)

Dataflow Analysis

- A data-flow analysis schema defines a value at each point in the program, IN[s] and OUT[s] for each statement s
- Statements of the program have associated *transfer functions* that relate the value before the statement to the value after
  - Forward OUT[s] = *f* (IN[s]) or backward IN[s] = *f* (OUT[s])
- Statements with more than one predecessor must have their value defined by combining the values at the predecessors, using a *meet* (or *confluence*) operator.
- Often *basic blocks* are annotated with values instead of individual statements: OUT[B] and IN[B]
- Useful for annotating the code with info needed for local or global optimization.

Dataflow analysis for Reaching Definitions and Live Variables

- ***Reaching Definitions***: Each statement is associated with the set of definitions that are active.
- The transfer function for a block kills definitions of variables that are redefined in the block and adds definitions of variables that occur in the block.
- The confluence operator is union.
- ***Live Variables***: computes the variables that are *live* (will be used before redefinition) at each point.
- Similar to *reaching definitions*, but the transfer function runs backward. A variable is *live* at the beginning of a block if it is either used before definition in the block or is live at the end of the block and not redefined in the block.

Dataflow analysis for Reaching Definitions

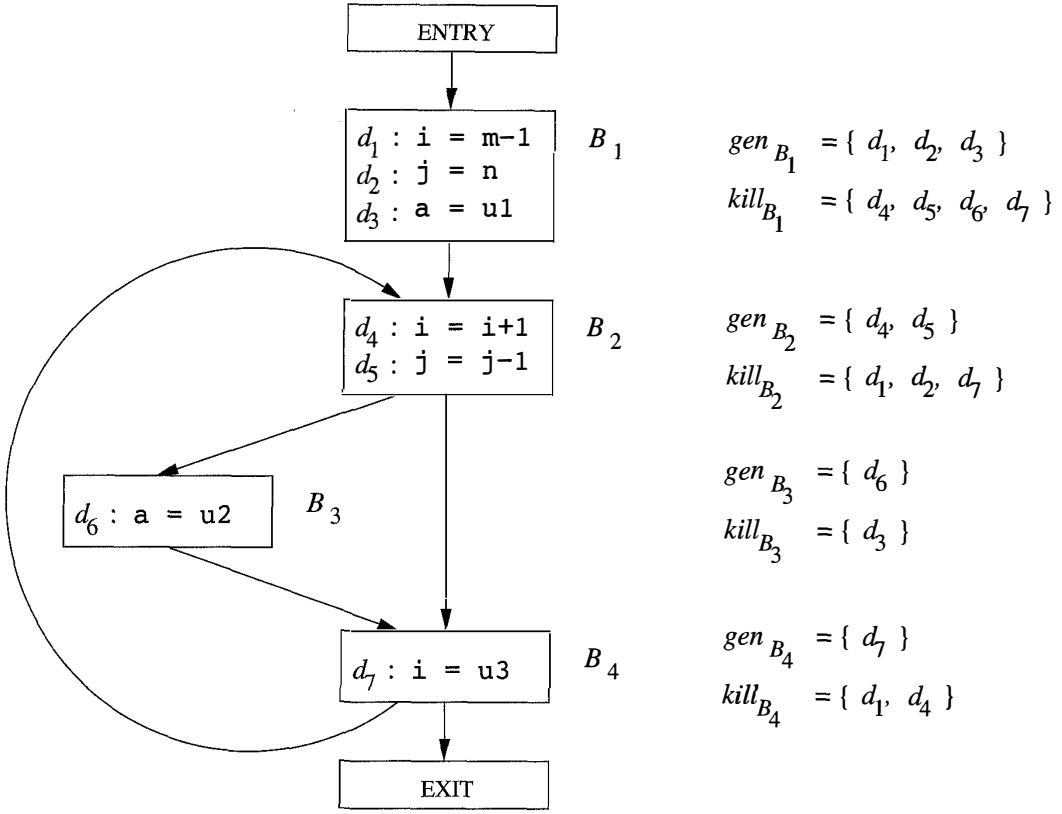


Figure 9.13: Flow graph for illustrating reaching definitions

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$$