Principles of Programming Languages

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

Prof. Andrea Corradini
Department of Computer Science, Pisa

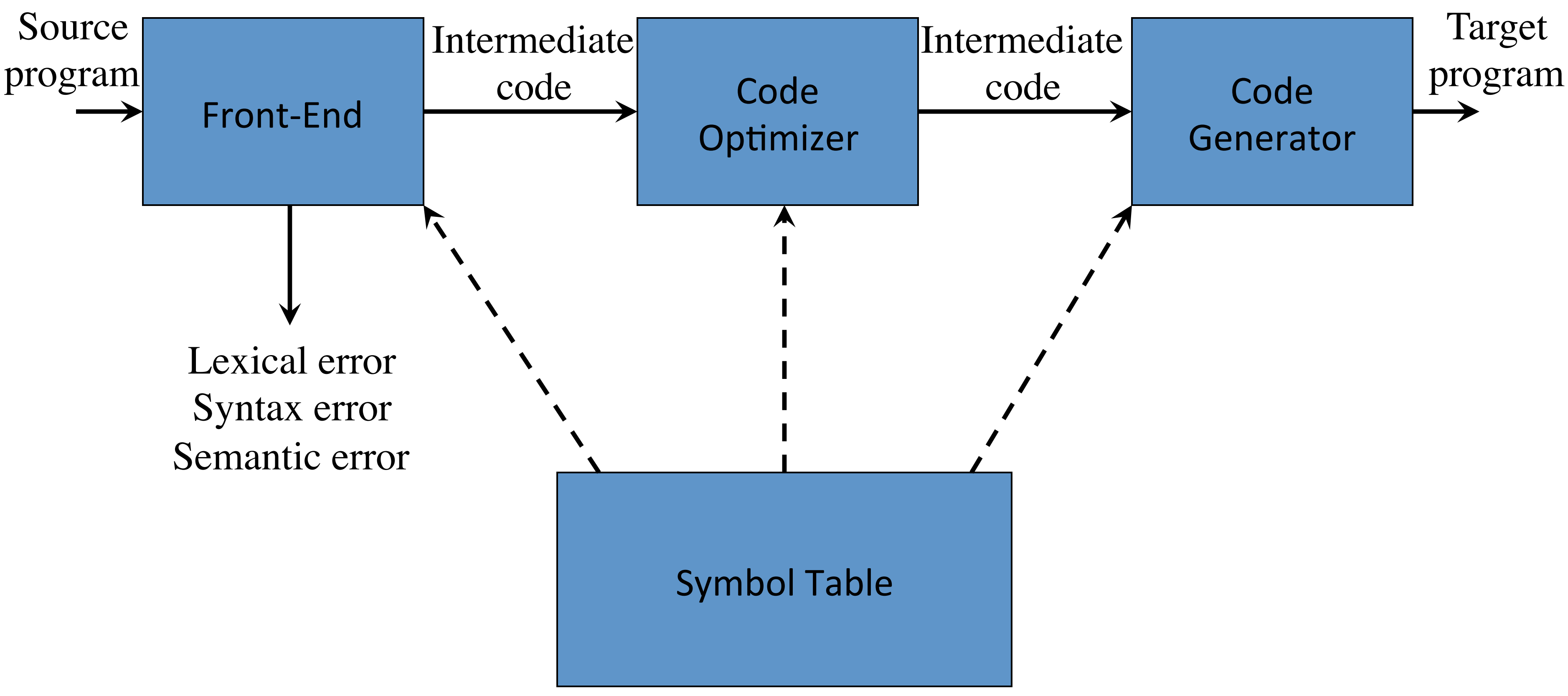
Lesson 31

- Code generation and optimization

On Code Generation

- Code produced by compiler must be correct
  - Source-to-target program transformation should be *semantics preserving*
- Code produced by compiler should be of high quality
  - Effective use of target machine resources
  - Heuristic techniques should be used to generate good but suboptimal code, because generating optimal code is undecidable

Position of a Code Generator in the Compiler Model



Code Generation: tasks

- Code generation has three primary tasks:
  – Instruction selection
  – Register allocation and assigment
  – Instruction ordering
- The compiler can include an optimization phase (mapping IR to optimized IR) before the code generation
- We consider some rudimentary optimizations only

Input of the Code Generator

- The input of code generation is the IR of the source program, with info in the symbol table
- Assumptions:
  - We assume that the IR is three-address code
  - Values and names in the IR can be manipulated directly by the target machine
  - The IR is free of syntactic and static semantic errors
  - Type conversion operators have been introduced where needed

Target Program Code

- The back-end code generator of a compiler may generate different forms of code, depending on the requirements:
  - Absolute machine code (executable code)
  - Relocatable machine code (object files for linker: allows separate compilation of subprograms)
  - Assembly language (facilitates debugging, but requires an assembly step)

Target Machine Architecture

- Defines the instruction-set, including addressing modes: high impact on the code generator
- **RISC** (*reduced instruction set computer*): single-clock instructions, many register, three address instructions, simple addressing modes
- **CISC** (*complex instruction set computer*): multi-clock instructions, complex addressing modes, several register classes, variable-length instructions operating in memory
- **Stack-based machines**: operands are put on the stack and operations act on top of stack (held in register). In general less efficient.
  - Revived thanks to bytecode forms for interpreters like the Java Virtual Machine

Our Target Machine

- We consider a RISC-like machine with some CISC-like addressing modes
- Assembly code as target language (for readability)
  - Variable names and constant are not translated
  - Absolute/relocatable target code requires to translate them using info from symbol table
- Our (hypothetical) machine:
  - Byte-addressable (word = 4 bytes)
  - Has *n* general purpose registers **R0**, **R1**, …, **R***n*-1
  - Simplified instruction-set: all operands are integer
  - Three-address instructions of the form

    op dest, *src1, src2*

The Target Machine: Instruction Set

- **LD r, x** (load operation: *r = x*)
- **ST x, r** (store operation: *x = r*)
- **OP dst, src1, src2** where OP = ADD, SUB, …: apply *OP* to src1 and src2, placing the result in *dst*).
- **BR *L*** (unconditional jump: *goto L*)
- **B*cond* r, L** (conditional jump: *if cond(r) goto L*)
  es: **BLTZ r, L** (*if (r < 0) goto L*)

The Target Machine: Addressing Modes

- Addressing modes and corresponding costs (*c* is an integer):

Mode	Form	Address	Added Cost
Absolute	**`M`**	**`M`**	1
Register	**`R`**	**`R`**	0
Indexed	*c*(**`R`**)	*c*+*contents*(**`R`**)	1
Indirect register	**`*R`**	*contents*(**`R`**)	0
Indirect indexed	**`*`***c*(**`R`**)	*contents*(*c*+*contents*(**`R`**))	1
Literal	**`#`***c*	N/A	1

Instruction Costs

- Machine is a simple, non-super-scalar processor with fixed instruction costs
- Realistic machines have deep pipelines, various kinds of caches, parallel instructions, etc.
- Define:

cost (**`OP dst, src1, src2`**) = 1
+ cost(*dst*-mode)
+ cost(*src1*-mode)
+ cost(*src2*-mode)

Examples

Instruction	Operation	Cost
`LD R0,R1`	Load *content*(**`R1`**) into register **`R0`**	1
`LD R0,M`	Load *content*(**`M`**) into register **`R0`**	2
`ST M,R0`	Store *content*(**`R0`**) into memory location **`R0`**	2
`BR 20(R0)`	Jump to address 20+*contents*(**`R0`**)	2
`ADD R0 R0 #1`	Increment **`R0`** by 1	2
`MUL R0,M,*12(R1)`	Multiply *contents*(**`M`**) by *contents*(12+*contents*(**`R1`**) and store the result in **`R0`**	3

Instruction Selection

- Instruction selection depends on (1) the level of the IR, (2) the instruction-set architecture, (3) the desired quality (e.g. efficiency) of the generated code
- Suppose we translate three-address code

  x:=y+z to:

```
LD R0,y          \\ R0=y
ADD R0,R0,z      \\ R0=R0+z
ST x,R0          \\ x=R0
```

- Then

`a:=a+1` →

```
LD R0,a
ADD R0,R0,#1
ST a,R0
```

Cost = 6

Better:

`ADD a,a,#1` Cost = 4

Best:

`INC a` Cost = 2

(if available)

Need for Global Machine-Specific Code Optimizations

- Suppose we translate three-address code
  x:=*y*+*z*   to:

```
LD R0,y        \\ R0=y
ADD R0,R0,z    \\ R0=R0+z
ST x,R0        \\ x=R0
```

- Then, we translate
  a:=b+c
  d:=a+e to:

```
LD R0,b
ADD R0,R0,c
ST a,R0
LD R0,a        ← Redundant
ADD R0,R0,e
ST d,R0
```

- We can choose among several equivalent instruction sequences → *Dynamic programming* algorithms

Register Allocation and Assignment

- Efficient utilization of the limited set of registers is important to generate good code
- Registers are assigned by
  - *Register allocation* to select the set of variables that will reside in registers at a point in the code
  - *Register assignment* to pick the specific register that a variable will reside in
- Finding an optimal register assignment in general is NP-complete

Choice of Instruction Ordering

- When instructions are independent, their evaluation order can be changed

```
a+b-(c+d)*e   =>   t1:=a+b          reorder   =>   t2:=c+d
                   t2:=c+d                          t3:=e*t2
                   t3:=e*t2                         t1:=a+b
                   t4:=t1-t3                        t4:=t1-t3
```

- The reordered sequence could lead to a better target code

Towards Flow Graphs

- In order to improve *instruction selection, register allocation and selection,* and *instruction ordering,* we structure the input three-address code as a *flow graph*
- This allows to make explicit certain dependencies among instructions of the IR
- Simple optimization techniques are based on the analysis of such dependencies
  - Better register allocation knowing how variables are defined and used
  - Better instruction selection looking at *sequences* of three-address code statements

Flow Graphs

- A *flow graph* is a graphical representation of a sequence of instructions with control flow edges
- A flow graph can be defined at the intermediate code level or target code level
- Nodes are *basic blocks*, sequences of instructions that are always executed together
- Arcs are execution order dependencies

Basic Blocks

- A *basic block* is a sequence of instructions s.t.:
  - – Control enters through the first instruction only
  - – Control leaves the block without branching, except possibly at the last instruction

```
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
```

```
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)
```

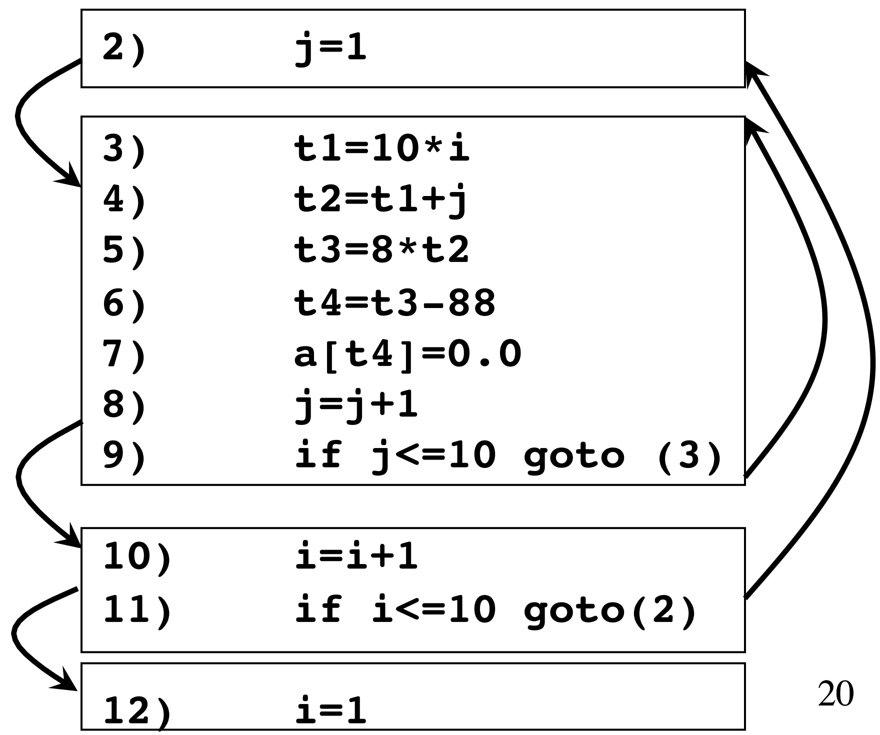
```
10)     i=i+1
11)     if i<=10 goto(2)
```

```
12)     i=1
```

Basic Blocks and Control Flow Graphs

- A control flow graph (CFG) is a directed graph with basic blocks B_i as vertices and with edges $B_i \rightarrow B_j$ iff B_j can be executed immediately after B_i
- Then B_i is a predecessor of B_j, B_j is a successor of B_i

```
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
```



Partition Algorithm for Basic Blocks

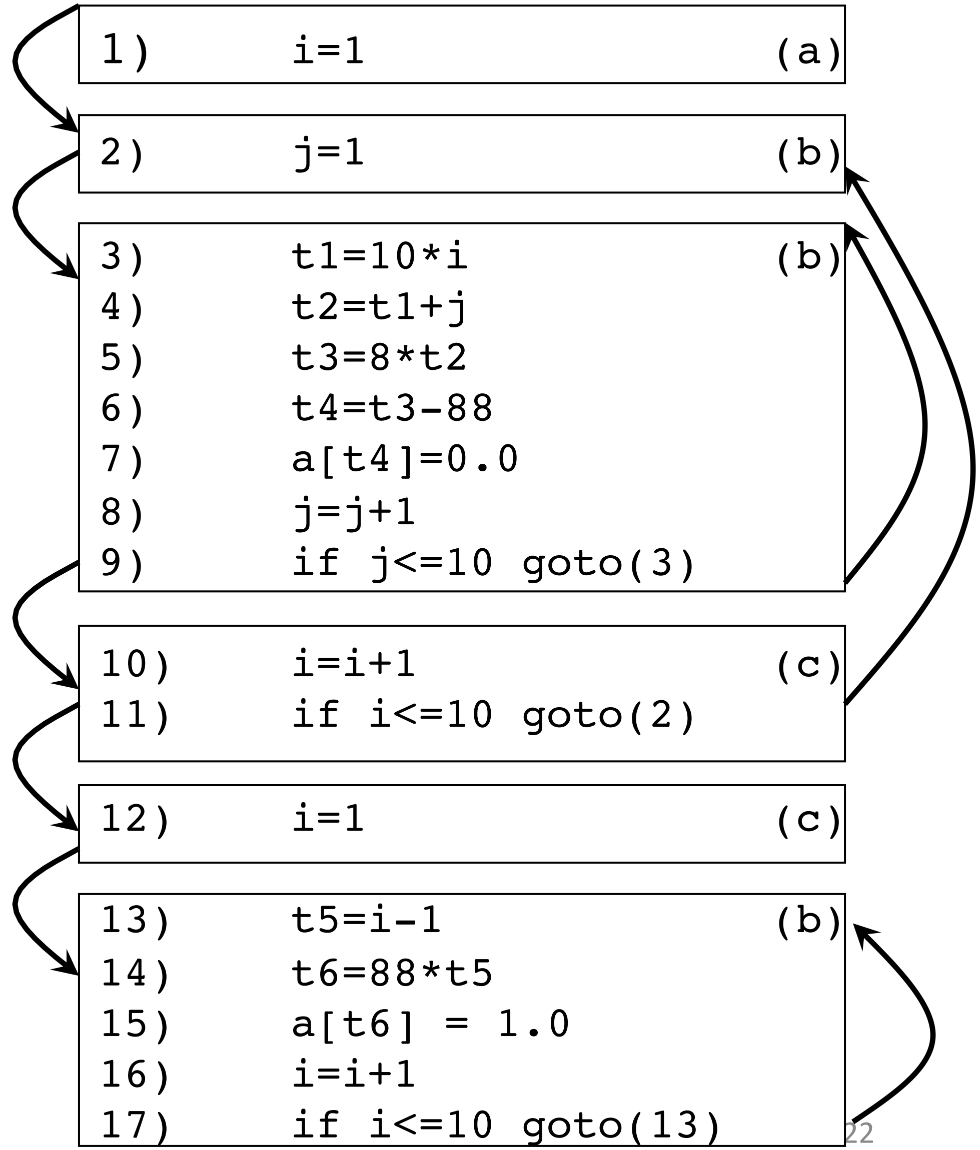
Input: A sequence of three-address statements
Output: A list of basic blocks with each three-address statement in exactly one block

1. Determine the set of *leaders*, the first statements in basic blocks
   a) The first statement is the leader
   b) Any statement that is the target of a *goto* is a leader
   c) Any statement that immediately follows a *goto* is a leader
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

Partition Algorithm for Basic Blocks: Example

1) i=1 (a)
2) j=1 (b)
3) t1=10*i (b)
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 (c)
11) if i<=10 goto(2)
12) i=1 (c)
13) t5=i-1 (b)
14) t6=88*t5
15) a[t6] = 1.0
16) i=i+1
17) if i<=10 goto(13)

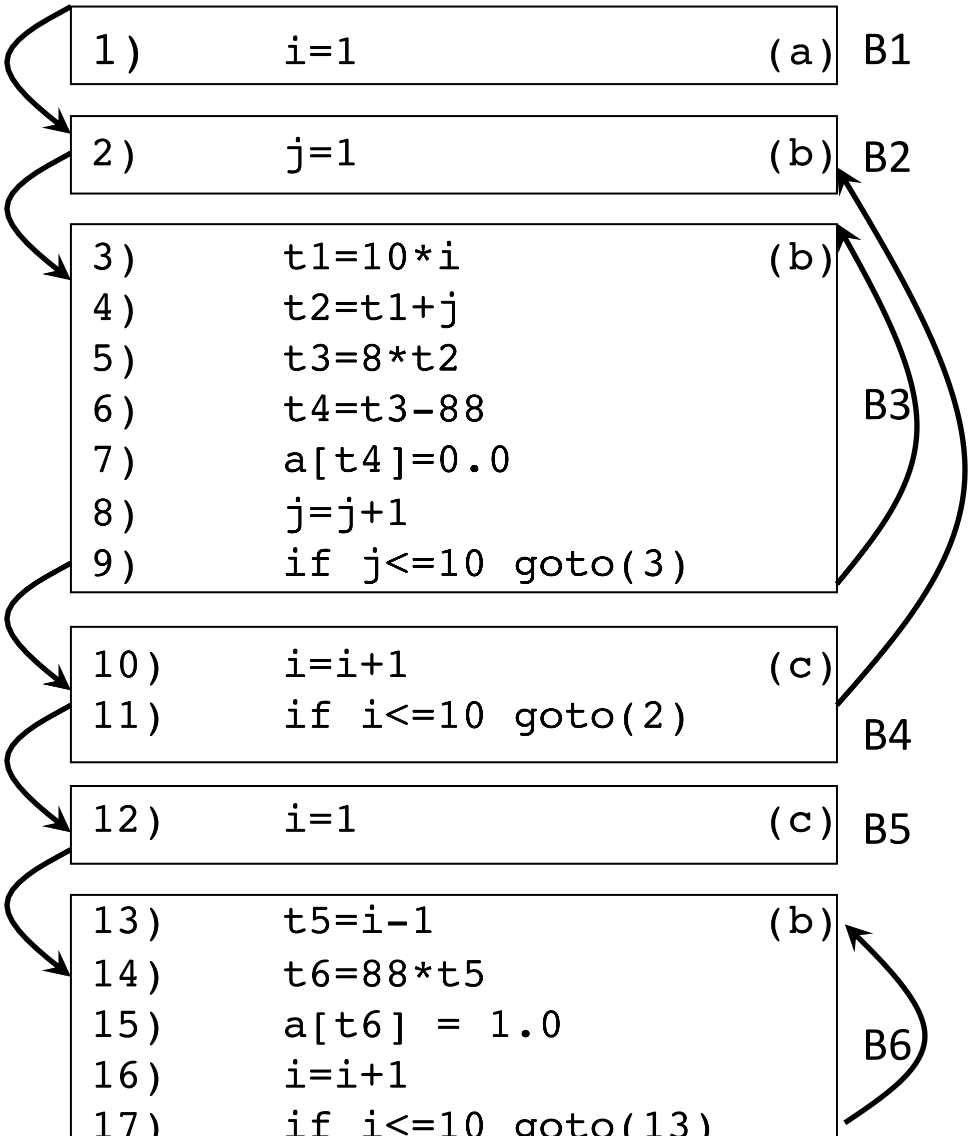
Leaders



Loops

- Programs spend most of the time executing loops
- Identifying and optimizing loops is important during code generation
- A *loop* is a collection of basic blocks, such that
  - All blocks in the collection are *strongly connected*
  - The collection has a unique *entry*, and the only way to reach a block in the loop is through the entry

Loops (Example)



Strongly connected components:

SCC={ {B2,B3,B4},
{B3}, {B6} }

Entries:
B2, B3, B6

Transformations on Basic Blocks

- A *code-improving transformation* is a code optimization to improve speed or reduce code size
- *Global transformations* are performed across basic blocks
- *Local transformations* are only performed on single basic blocks
- Transformations must be safe and preserve the meaning of the code
  - A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form
- We will sketch several local optimization techniques

Equivalence of Basic Blocks

- Two basic blocks are (semantically) *equivalent* if they compute the same set of expressions

```
b  := 0
t1 := a + b
t2 := c * t1
a  := t2
```

↓

```
a := c*a
b := 0
```

```
a  := c * a
b  := 0
```

↓

```
a := c*a
b := 0
```

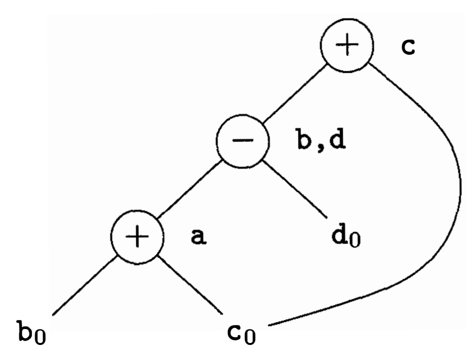
Blocks are equivalent, assuming **t1** and **t2** are *dead*: no longer used (no longer *live*)

DAG representation of basic blocks

1. One leaf for the initial value of each variable in the block
2. One node N for each statement *s*. Children are statements producing values of needed operands
3. Node N is labeled by the operator of *s*, and by the list of variables for which it defines the last value in the block
4. "Output nodes" are labeled by ***live on exit*** variables, determined with global analysis

Example:

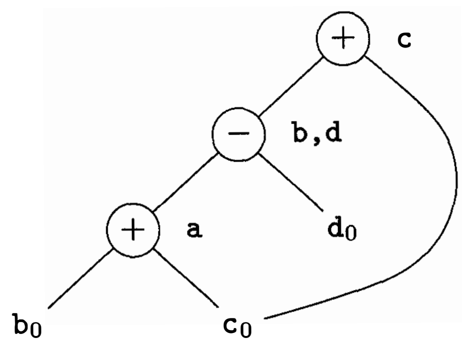
```
a := b + c
b := a - d
c := b + c
d := a - d
```



Common-Subexpression Elimination

- Remove redundant computations

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




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

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

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

Dead Code Elimination

- Remove unused statements

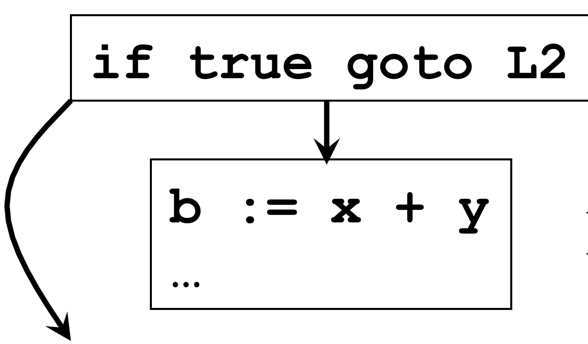
```
b := a + 1
a := b + c
…
```

⇒

```
b := a + 1
…
```

Assuming **a** is *dead* (not used)

- In the DAG: remove any root having no live variable attached, and iterate



Remove unreachable code

Algebraic Transformations

- Change arithmetic operations to transform blocks to algebraic equivalent forms

```
t1 := a - a
t2 := b + t1
t3 := 2 * t2
```

```
t1 := 0
t2 := b
t3 := t2 << 1
```

  - Algebraic identities (e.g. comm/assoc of operators) → has to conform the language specification
  - Reduction in strength
  - Constant folding

Renaming Temporary Variables

- Temporary variables that are dead at the end of a block can be safely renamed

```
t1 := b + c
t2 := a - t1
t1 := t1 * d
d := t2 + t1
```

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

Normal-form block

Interchange of Statements

- Independent statements can be reordered

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

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

Note that normal-form blocks permit all statement interchanges that are possible

(Local) Next-Use Information

- *Next-use information* is needed for dead-code elimination and register assignment
- Next-use is computed by a backward scan of a basic block and performing the following actions on statement

  i: *x* := *y* op *z*

  - Add liveness/next-use info on *x*, *y*, and *z* to statement *I*
    - This info can be stored in the symbol table
  - Before going up to the previous statement (scan up):
    - Set *x* info to "not live" and "no next use"
    - Set *y* and *z* info to "live" and the "next uses" of *y* and *z* to *i*

Next-Use (Step 1)

i: **`b := b + 1`**

j: **`a := b + c`**

k: **`t := a + b`** [*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`t`**) = true,
nextuse(**`a`**) = none, *nextuse*(**`b`**) = none, *nextuse*(**`t`**) = none]

Attach current live/next-use information
Because info is empty, assume variables are live
(Data flow analysis can provide accurate information)

Next-Use (Step 2)

i: **`b := b + 1`**

j: **`a := b + c`**

live(**`a`**) = true	*nextuse*(**`a`**) = *k*
live(**`b`**) = true	*nextuse*(**`b`**) = *k*
live(**`t`**) = false	*nextuse*(**`t`**) = none

k: **`t := a + b`** [*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`t`**) = true,
nextuse(**`a`**) = none, *nextuse*(**`b`**) = none, *nextuse*(**`t`**) = none]

Compute live/next-use information at *k*

Next-Use (Step 3)

i: **`b := b + 1`**

j: **`a := b + c`** [*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`c`**) = true,
nextuse(**`a`**) = *k*, *nextuse*(**`b`**) = *k*, *nextuse*(**`c`**) = none]

k: **`t := a + b`** [*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`t`**) = true,
nextuse(**`a`**) = none, *nextuse*(**`b`**) = none, *nextuse*(**`t`**) = none]

Attach current live/next-use information to *j*

Next-Use (Step 4)

live(**a**) = false	*nextuse*(**a**) = none
live(**b**) = true	*nextuse*(**b**) = *j*
live(**c**) = true	*nextuse*(**c**) = *j*
live(**t**) = false	*nextuse*(**t**) = none

i: `b := b + 1`

j: `a := b + c` [*live*(**a**) = true, *live*(**b**) = true, *live*(**c**) = true,
nextuse(**a**) = *k*, *nextuse*(**b**) = *k*, *nextuse*(**c**) = none]

k: `t := a + b` [*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true,
nextuse(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Compute live/next-use information *j*

Next-Use (Step 5)

i: **`b := b + 1`** [*live*(**`b`**) = true, *nextuse*(**`b`**) = *j*]

j: **`a := b + c`** [*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`c`**) = true,
nextuse(**`a`**) = *k*, *nextuse*(**`b`**) = *k*, *nextuse*(**`c`**) = none]

k: **`t := a + b`** [*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`t`**) = true,
nextuse(**`a`**) = none, *nextuse*(**`b`**) = none, *nextuse*(**`t`**) = none]

Attach current live/next-use information to *i*