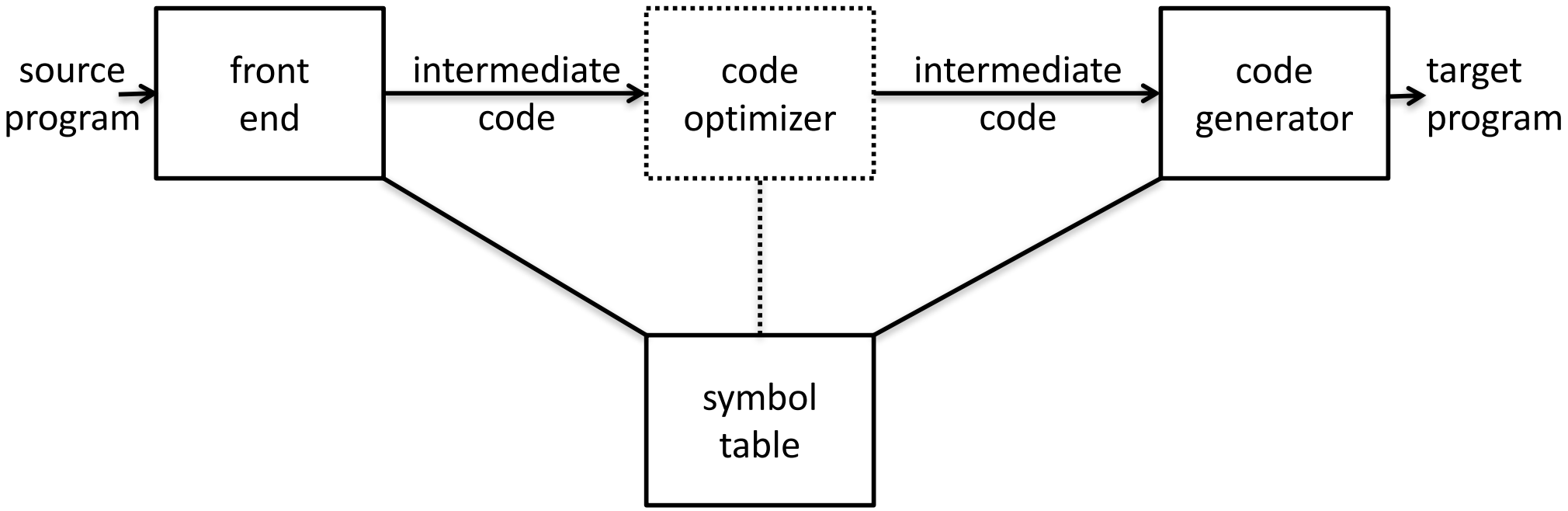
# ΕΠΛ323 - Θεωρία και Πρακτική Μεταγλωττιστών

Lecture 11b

## Code Generation

Elias Athanasopoulos
eliasathan@cs.ucy.ac.cy

# Final Stage!


source program
front end
intermediate code
code optimizer
intermediate code
code generator
target program
symbol table

# High-level Operation

- Input
  - Intermediate Code (AST, DAG, TAC)
- Output
  - Absolute Machine Code
  - Relocatable Machine Code
  - Assembly Code
- Memory management
  - Mapping names (symbols) into actual memory addresses

# Instruction Selection

- Quality of Code
  - Speed and size
- Machines understand specific instructions

```
MOV y, R0 /* load y into register R0.  */
ADD z, R0 /* add z to R0.  */
MOV R0, x /* store R0 into x  */
```

- TAC input

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

- Machine output (not efficient)

```
MOV b, R0
ADD c, R0
MOV R0, a
MOV a, R0
ADD e, R0
MOV R0, d
```

These instructions are redundant.

# Register Allocation

- During register allocation, we select the set of variables that will reside in registers at a point in the program.
- During a subsequent register assignment phase, we pick the specific register that a variable will reside in.
- Finding an optimal assignment of registers is NP-complete.
- For complications: hardware/OS specific register usage.

# Register Allocation

```
t := a + b
t := t * c
t := t / d
```

```
t := a + b
t := t + c
t := t / d
```

```
L     R1, a
A     R1, b
M     R0, c
D     R0, d
ST    R1, t
```

```
L     R0, a
A     R0, b
A     R0, c
SRDA  R0,32
D     R0, d
ST    R1, t
```

The optimal choice for the register into which `a` is to be loaded depends on what will ultimately happen to `t`.

# Choice of Evaluation Order

- The order in which computations are performed can affect the efficiency of the target code.
- The optimal order is also a difficult, NP-complete, problem.

# Approaches to Code Generation

- Target code may not be optimal, but it should be correct
- Given the premium of correctness, we try to design a code generator so that
  - It can be easily implemented
  - It can be easily tested
  - It can be easily maintained

# Basic Block

- A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end:

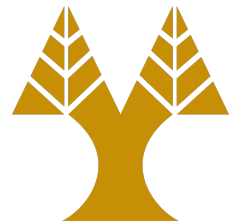
```
t1 := a * a
t2 := a * b
t3 := 2 * t2
t4 := t1 + t3
t5 := b * b
t6 := t4 + t5
```

- A three-address statement `x := y + z` *defines* `x` and *uses* (or *references*) `y` and `z`
- A name in a basic block is said to be *live* at a given point if its value is used after that point in the program, perhaps in another basic block

# Partition Code into Basic Blocks

- Input
  - A sequence of three-address statements
- Output
  - A list of basic blocks with each three-address statement in exactly one block
- Method
  1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are the following.
     1. The first statement is a leader.
     2. Any statement that is the target of a conditional or unconditional goto is a leader.
     3. Any statement that immediately follows a goto or conditional goto statement is a leader.
  2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

# Example



```
begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + a[i] * b[i];
    i:=i+1;
  end
  while i <= 20
end
```

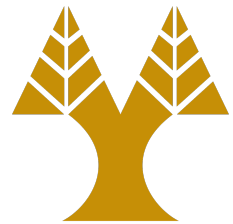
```
(1)  prod := 0
(2)  i  := 1
(3)  t1 := 4 * i
(4)  t2 := a[t1]
(5)  t3 := 4 * i
(6)  t4 := b[t3]
(7)  t5 := t2 * t4
(8)  t6 := prod + t5
(9)  prod := t6
(10) t7 := i+1
(11) i  := t7
(12) if i <= 20 goto (3)
(13) . . .
```

Statement (1) is a leader by Rule 1.
Statement (3) is a leader by Rule 2.
Statement (13) is a leader by Rule 3.

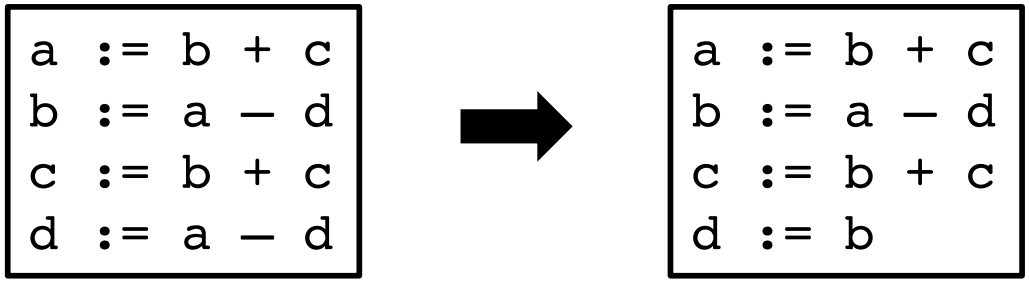
# Transformations on Basic Blocks

- Two basic blocks are said to be *equivalent* if they compute the same set of expressions
- Transformation can improve the quality of the produced code, without changing the set of expressions computed by a particular block
  - Structure-preserving Transformations
  - Algebraic Transformations

# Structure-preserving Transformations

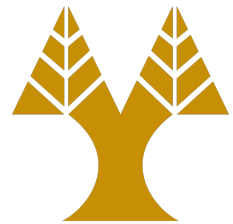


- Common subexpression elimination




- Dead-code elimination
  - Suppose that `x` is dead, that is, never subsequently used, at the point where the statement `x := y + z` appears in a basic block. Then this statement may be safely removed.

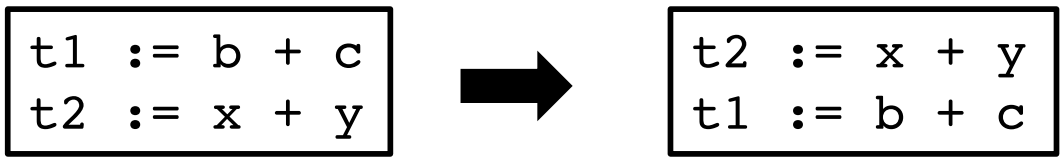
# Structure-preserving Transformations



- Renaming temporary variables

  `t := b + c` can become `u := b + c`

- Interchange of statements



# Algebraic Transformations

- Statements can be eliminated

```
x := x + 0
x := x * 1
```

- Statements can be replaced

```
x := y ** 2
```

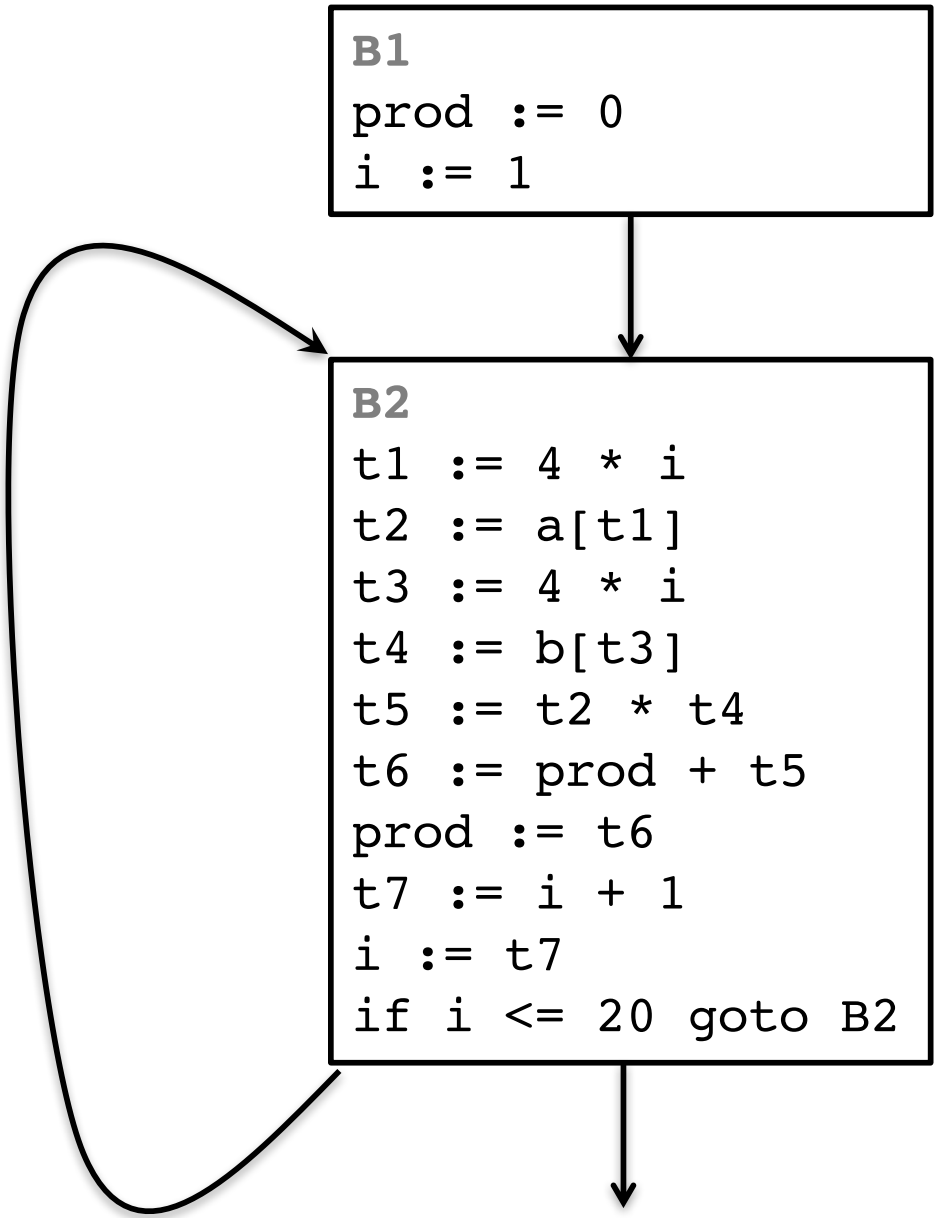
to

```
x := y * y
```

# Flow graphs

- We can add the flow-of-control information to the set of basic blocks making up a program by constructing a directed graph called a f*low graph*
- The *initial node* is the basic block whose leader is the first statement of the program
- There is a directed edge from block B1 to block B2:
  - Conditional or unconditional jump from last statement of B1 to first statement of B2
  - B2 immediately follows B1 in the order of the program and B1 does not end in an unconditional jump
- B1 is *predecessor* of B2 and B2 *successor* of B1

# Flow Graph


B1
prod := 0
i := 1
B2
t1 := 4 * i
t2 := a[t1]
t3 := 4 * i
t4 := b[t3]
t5 := t2 * t4
t6 := prod + t5
prod := t6
t7 := i + 1
i := t7
if i <= 20 goto B2

# Next-Use Information

- Next-use information dictates if a name in a basic block is going to be used again
- If the name is not going to be used again in the block, then the register holding the name can be released and used for holding other names

# Computing next uses

- Suppose we reach three-address statement *i*: `x := y op z`, in our backward scan. We then do the following
  1. Attach to statement *i* the information currently found in the symbol table regarding the next use and liveness of `x`, `y`, and `z`.
  2. In the symbol table, set `x` to “not live” and “no next use”
  3. In the symbol table, set `y` and `z` to “live” and the next uses of `y` and `z` to *i*. Note that the order of steps (2) and (3) may not be interchanged because `x` may be `y` or `z`.

# Example

| Code | Live/Dead | | | Next Use | | |
|---|---|---|---|---|---|---|
| | x | y | z | x | y | z |
| (4) x := z + y | F | T | T | | 4 | 4 |
| (3) y := z - 7 | F | F | T | | | 3 |
| (2) z := x * 5 | T | F | F | 2 | | |
| (1) x := y + z | F | T | T | | 1 | 1 |