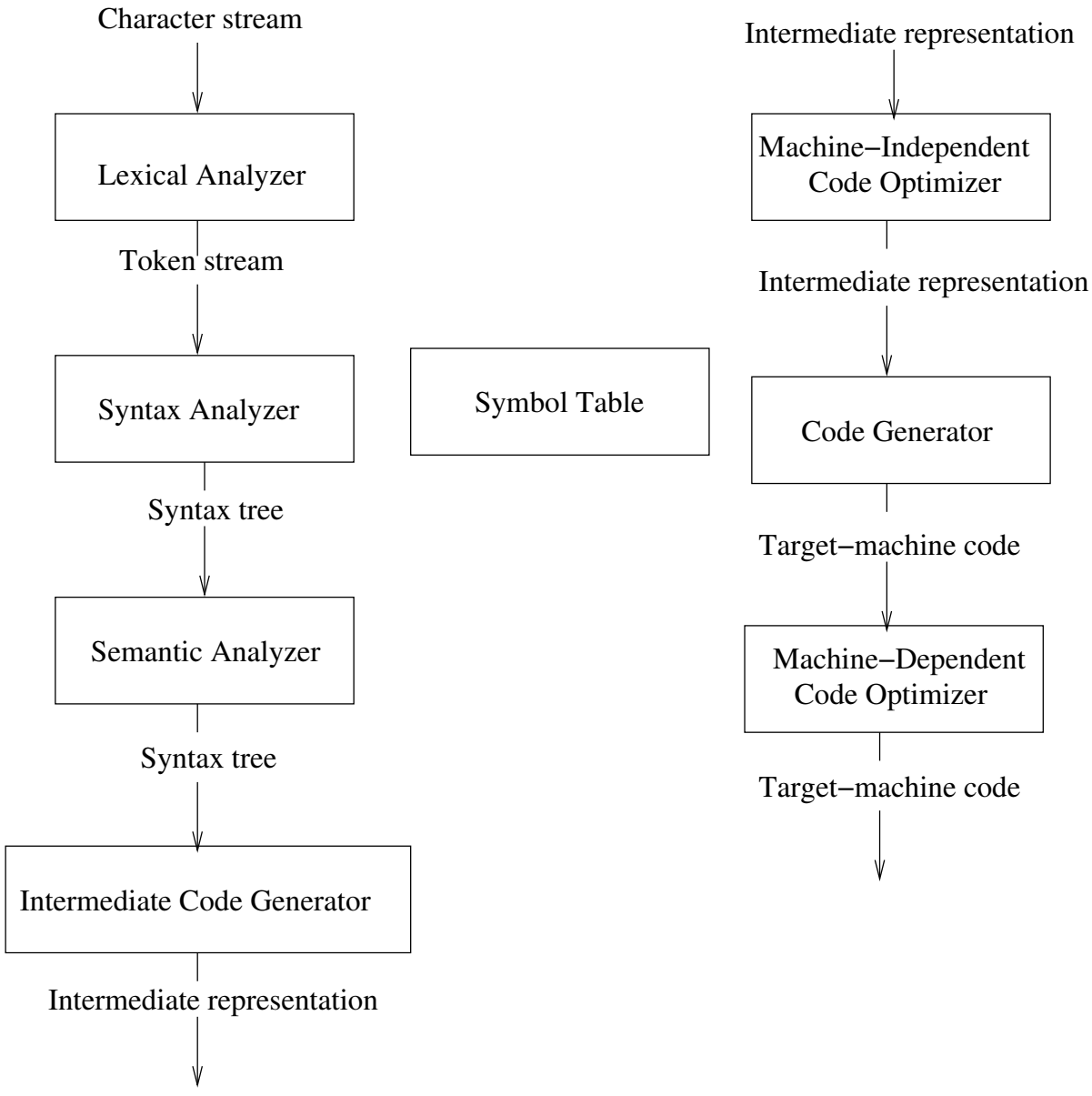
Machine-independent Optimizations

Sudakshina Dutta

IIT Goa

29^{th} April, 2022

The Phases of a Compiler



Graph representation of three-address code

- Graph representation of three-address code are helpful code optimization and code generation
- **Basic block :** Partition the intermediate code into *basic blocks*, which is a sequence of instructions with following properties
  1. The flow of control can only enter the basic block through the first instruction and there are no jumps into middle of basic blocks
  2. Control will leave the block without halting or branching, except the last instruction
- **Flow graph :** The basic blocks become nodes of *flow graph*, whose edges indicate which blocks can follow which other blocks

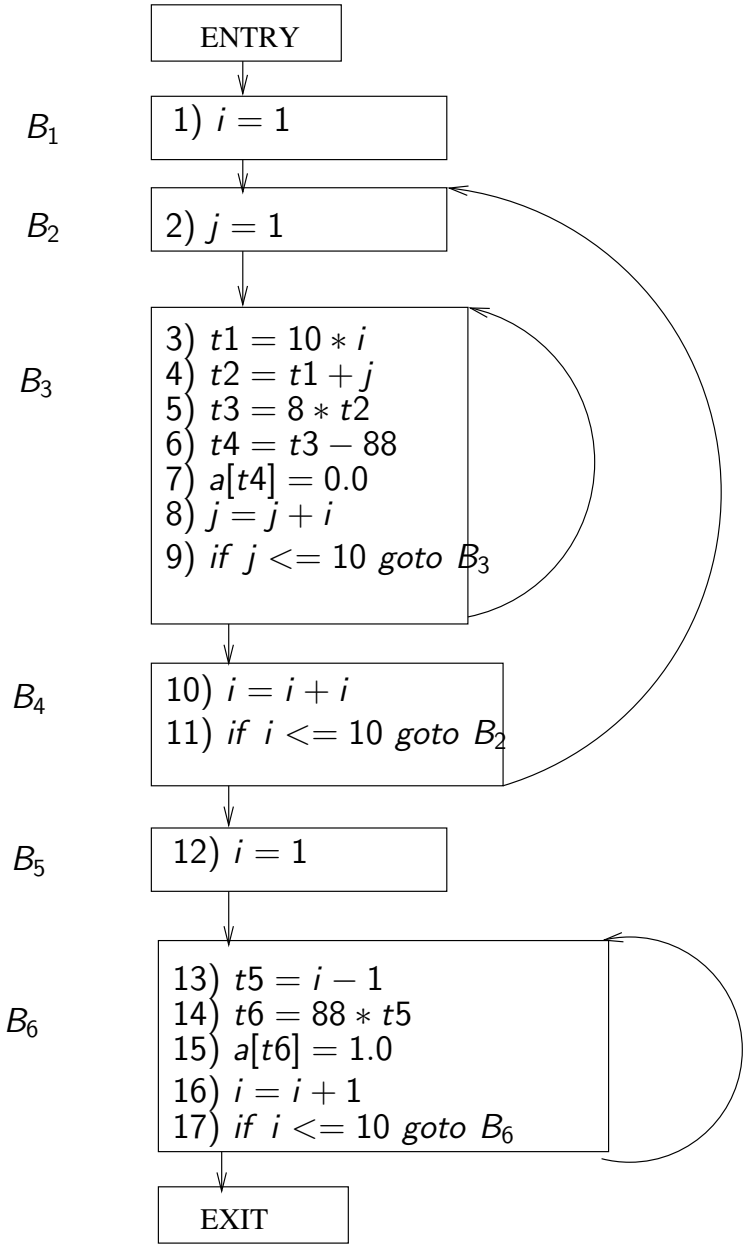
How to construct basic block

- Finding leaders
  - The first three-address instruction is a leader

    —The first instruction of any basic block is called a *leader*
  - Any instruction that is the target of a conditional or unconditional jump is a leader
  - Any instruction that immediately follows a conditional or unconditional jump is a leader
- For each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program

Basic block and flow graph

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) $j = j + i$
9) *if* $j <= 10$ *goto* (3)
10) $i = i + i$
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)



Compiler optimization

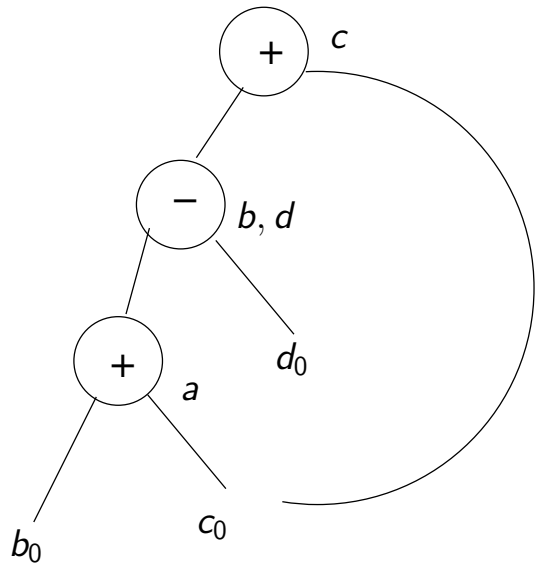
- Compiler generates “naive” intermediate code and then improves the quality of target code by applying optimizing transformations on it
- There is no guarantee that the resulting code is optimal under any mathematical measure
- Many simple transformations significantly improve the running time and space requirement of the target program

Local optimizations

- Within the basic block itself
- Substantial improvement possible
- **Local common subexpression elimination** : While adding a new node M, it takes place by noticing whether there is an existing node N with the same children, in the same order, and with the same operator
- If yes, N can be used in place of M

Finding local common subexpressions

$a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$



Algebraic identities

- **Arithmetic identities** : Important class of optimization inside basic blocks
  1. $x + 0 = 0 + x = x$
  2. $x \times 1 = 1 \times x = x$
  3. $x - 0 = x$
  4. $x/1 = x$
- **Strength reduction** : Replacing a more expensive operation by a cheaper one
  1. $2 \times x$ by $x + x$
- **Constant folding** : Evaluating constant expressions at compile time and replace by values e.g., 2×3 by 6

Peephole optimization

- A simple way to improve the code
- A small, sliding window is called *peephole*
- It is done by replacing instruction sequence within the peephole by a faster sequence, if possible

Peephole optimizations

- **Eliminating redundant load and store** Replace
  $LD\ R_0, a$
  $ST\ a, R_0$
  with the first instruction
- **Eliminating unreachable code**
  1. An unlabeled instruction immediately following an unconditional jump can be removed
  2. Optimizing the following code fragment in presence of the assignment $debug = 0$
     if $debug == 1$ *goto* $L1$
     goto $L2$
     $L1$: print debugging information
     L2:

Flow-of-control optimization

- Intermediate code generation algorithms often produces jumps to jumps
- They can be eliminated by peephole optimization
  $if\ a < b\ goto\ L1$
  ...
  $L1 : goto\ L2$
  can be replaced by
  $if\ a < b\ goto\ L2$
  ...
  $L1 : goto\ L2$
- and also **Algebraic simplification and reduction in strength**

Global code optimization

- Across basic block
- Mostly based on *data-flow analysis*
- For each instruction, they specify the properties that must hold every the instruction is executed e.g., whether the value of a variable is constant for *constant propagation optimization*
- Example : Global common subexpression elimination, copy propagation, code motion, induction variable elimination and strength reduction, etc.

Copy propagation

if($a > b$)
$\{ \quad a = d + e; \}$
else
$\{ \; b = d + e; \}$
$c = d + e;$

The above can be changed to

if($a > b$)
$\{ \; t = d + e; \; a = t; \}$
else
$\{ \; t = d + e; \; b = t; \}$
$c = t;$

Code motion

while($a < b$)
{ $x = y * 5;$
$a = a + x;$
}

The above can be changed to

$x = y * 5;$
while($a < b$)
{ $a = a + x;$
}

- The evaluation of x is loop invariant and it can be moved out of the loop

Dead-code elimination

- A variable is *live* its value can be used subsequently; otherwise it is *dead*
- Dead code is a segment of code that does not get executed. It gets introduced unintentionally

$a = true;$
if$(a == false)$
$\{$
    $b = f(a);$
$\}$

Cross-compilers

- A cross-compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running
  - Example - Clang
  - clang -c -target mips prog.c
  - clang -c -target armv7a prog.c
- A cross compiler is necessary to compile code for multiple platforms from one development host

Cross-compilers

- In GCC world, every host/target combination has its own set of binaries, headers, libraries, etc.
  - It is usually simple to download a package with all files in, unzip to a directory and point the build system to that compiler
- On the other hand, Clang/LLVM is natively a cross-compiler
  - It means that one set of programs can compile to all targets by setting the -target option
  - It makes it a lot easier for programmers wishing to compile to different platforms and architectures, and for compiler developers that only have to maintain one build system, and for OS distributions, that need only one set of main packages