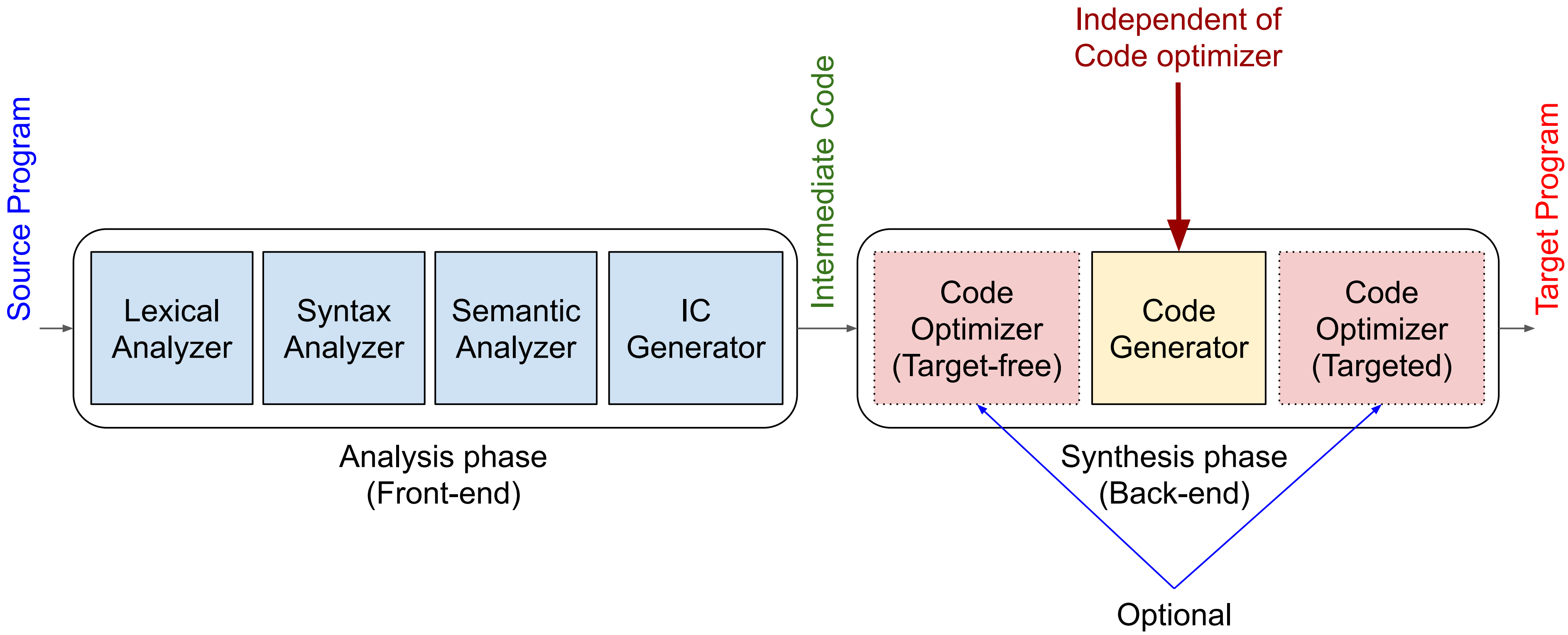
Code Generation

Md Shad Akhtar
Assistant Professor
IIIT Dharwad

Where are we?


Independent of Code optimizer
Source Program
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
IC Generator
Intermediate Code
Code Optimizer (Target-free)
Code Generator
Code Optimizer (Targeted)
Target Program
Analysis phase (Front-end)
Synthesis phase (Back-end)
Optional

Expectation from Code generation

- Target program
  - Must preserve the semantic of the source program (Most important!)
  - Should be of high quality
    - Ensure the effective use of available resources of the target machine
- Code generation itself must run efficiently
  - Generating an optimal target program for a given source program is undecidable;
  - Register allocation is intractable.

Code Generation

- Three primary tasks
  - Instruction selection
    - Choose the appropriate target-machine instructions to perform the desired operation
  - Register allocation and assignment
    - What values to keep in which registers. (Registers are limited!)
  - Instruction ordering
    - Scheduling the order of execution of instructions

Issues in Code Generator

- Input
  - Three address code, Syntax Trees, DAG etc
- Target Program
  - RISC architecture
    - 3 address instructions
    - Many registers
    - Simple addressing modes
  - CISC architecture
    - 2 address instructions
    - A few registers
    - Variety of addressing modes
  - Stack-based machine
    - Push operands onto stack
    - Operates on the operands at the top of the stack

Instruction selection

- Depends on
  - Level of IR
  - Instruction set architecture
  - Desired quality

- Instruction selection can be straight-forward
  - For each three-address instruction, define a code snippet to generate the target code
  - E.g.:
    - $x = y + z$

$\Rightarrow$

```
LD   R0,  y          // Load y into register R0
ADD  R0,  R0,  z     // Add z to register R0
ST   x,   R0         // Store R0 to x
```

Instruction selection

- However, such approach can be inefficient and may produce redundant codes
  - E.g.:

$$a = b + c$$
$$d = a + e$$

$$\Rightarrow \quad \begin{array}{llll}
\text{LD} & \text{R}_0, & b & \\
\text{ADD} & \text{R}_0, & \text{R}_0, & c \\
\text{ST} & a, & \text{R}_0 & \\
\text{LD} & \text{R}_0, & a & \\
\text{ADD} & \text{R}_0, & \text{R}_0, & e \\
\text{ST} & d, & \text{R}_0 &
\end{array}$$

Do we need these statements?

Instruction selection

- On most machines, a giver IR can be implemented by many different code sequences.
  - E.g.:

$a = a + 1$

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

OR

⇒        INC  a
```

Register Allocation

- When to use registers?
  - Utilize as many as possible; however, they are expensive and usually limited
- Among many values what values should be stored in register?
- Two subproblems
  - Register Allocation
    - Selection of set of variables that will reside in register at each point in the program
  - Register Assignment
    - Selection of a specific register that a variable will reside in
    - NP-complete problem

Evaluation Order

- Evaluation order can affect the efficiency of the target code
  - Some order are efficient than others
- Selecting an optimal evaluation order is NP-complete

Target Code

- Target code has dependency on the target machine and the instruction set available
    - A generic discussion is not possible
- A simple target-machine model
    - Three-address machine
    - *N* registers
    - Operations
        - Load, Store, Arithmetic, Jump, etc.
    - Addressing modes

MODE	**FORM**	**ADDRESS**
absolute	M	M
Register	R	R
Indexed	c(R)	c + contents(R)
indirect register	*R	contents(R)
indirect indexed	*c(R)	contents(c + contents(R))
MODE	**FORM**	**CONSTANT**
literal	#c	c

Instruction Costs

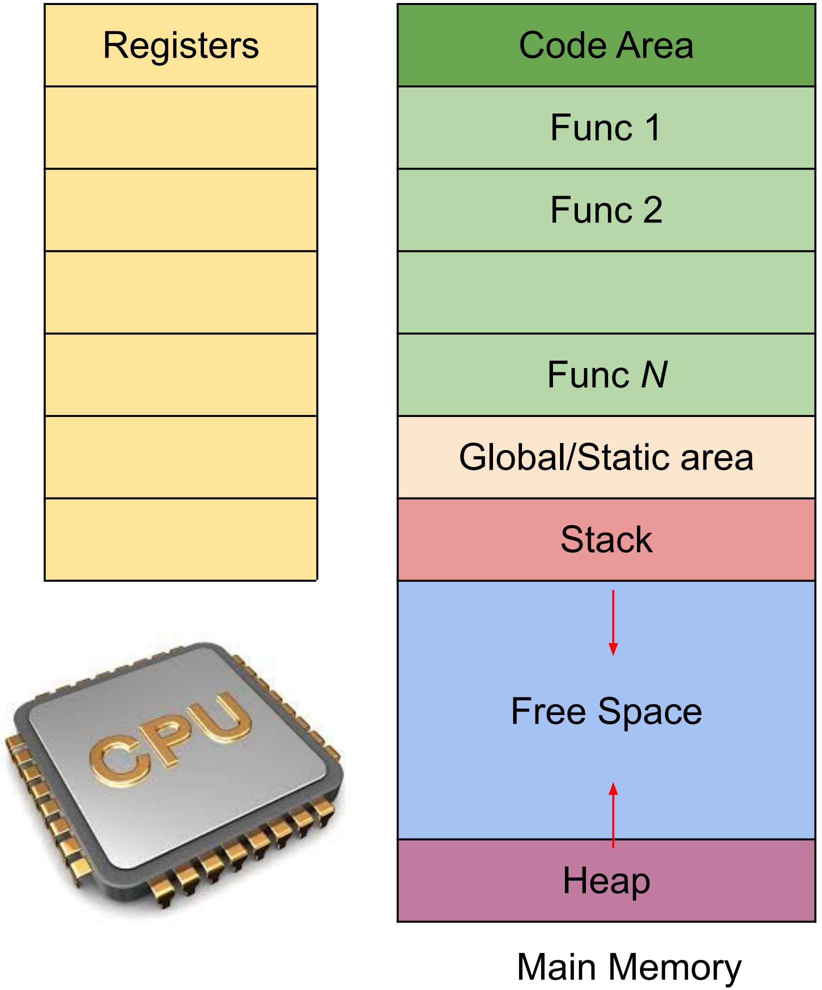
- We are interested in the efficient code
- How to define a code is efficient?
  - Each instruction has an associated cost
  - For simplicity, let us assume that each instruction has cost
    - 1 + cost(addressing mode)

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
MODE	FORM	CONSTANT	ADDED COST
literal	#c	c	1

- E.g.:
  - LD R0, R1 ⇒ 1
  - LD R0, M ⇒ 2
  - LD R0, *100(R1) ⇒ 2

Addresses in Target Code

- Logical address space of a program
  - Registers
    - Faster access, limited space
  - Main memory
    - Slower access, larger space



A simple code generator

- Code generation within a basic block
- In most machine architectures, some or all of the operands of an operator must be in registers
- Assumption
  - LD *reg*, *mem*
  - ST *mem*, *reg*
  - OP *reg*, *reg*, *reg*

Register and Address Descriptors

- For each available register, a *register descriptor* is maintained
  - It keeps track of the variable names whose current value is in the register
- For each program variable, an *address descriptor* is maintained
  - It keeps track of the location where the current value of that variable can be found
  - Location may be register, memory address, etc.

The code-generation algorithm

- An special function $getReg(I)$ selects the registers for each memory location associated with the three-address instruction I.
- Machine Instructions for Operations
  - For a three-address instruction $x = y + z$,
    - $getReg(x = y + z)$ to select registers for $x, y,$ and z. Call these R_x, R_y, and R_z
    - If y is not in R_y (according to register descriptor for R_y), then
      - LD R_y, y' (where y' is the memory location for y as per the address descriptor of y)
    - Similarly for R_z
    - ADD R_x, R_y, R_z
- Machine Instructions for Copy
  - For a three-address instruction $x = y$,
    - We assume that $getReg(x = y)$ selects same register R_y for both x and y.
    - If y is not in R_y, that call LD R_y, y'
- At the end of a basic block if a variable x is live, then
  - ST x, R_x

The code-generation algorithm….. contd

- Managing Register and Address Descriptor
  - For LD R_x, x
    - Change the register descriptor for register R_x so it holds only x
    - Change the address descriptor for x by adding register R_x as an additional location
  - For ST x, R_x
    - Change the address descriptor for x to include its own memory location
  - For a three-address instruction ADD R_x, R_y, R_z,
    - Change the register descriptor for register R_x so it holds only x
    - Change the address descriptor for x so that its only location is R_x
    - Remove R_x from the address descriptor of any variables other than x
  - For a three-address instruction $x = y$, after the LD R_y, y
    - Update the register descriptor for R_y to include x as one of the values
    - Change the address descriptor for x so that its only location is R_x

Code Generation: An example

- Generate target code for the following three-address codes for a basic block

  $t = a - b$

  $u = a - c$

  $v = t + u$

  $a = d$

  $d = v + u$

  Where

  - t, u, and v are the temporary variables
  - a, b, c, and d are the program variables that are live on exit from the basic block
  - Registers are unlimited; however, reuse any register if its value is not required

Code Generation: An example

	R1	*R2*	*R3*	*a*	*b*	*c*	*d*	*t*	*u*	*v*
				a	b	c	d			
t = *a* - *b* LD R1, *a* LD R2, *c* SUB R2, R1, R2	a	t		a, R1	b	c	d	R2		
u = *a* - *c* LD R3, *c* SUB R1, R1, R3	u	t	c	a	b	c, R3	d	R2	R1	
v = *t* + *u* ADD R3, R2, R1	u	t	v	a	b	c	d	R2	R1	R3
a = *d* LD R2, *d*	u	a,d	v	R2	b	c	d, R2		R1	R3
d = *u* + *v* ADD R1, R3, R1	d	a	v	R2	b	c	R1			R3
exit ST *a*, R2 ST *d*, R1	d	a	v	a, R2	b	c	d, R1			R3

Register Allocation and Assignment

- How to ensure the minimum number of registers required for a particular code?
- A potential solution:
    - Graph coloring problem
        - No adjacent nodes in the graph can have same color
        - Each node must be colored
        - The number of unique colors can be used as minimum number of register required for a code.

Peephole Optimization

- For the local improvement of the target code
- Examine a sliding window (aka, peephole) of target instructions and replacing instructions sequences within the peephole by a shorter/faster sequence, whenever possible.
  - Redundant-instruction elimination
    - LD R0, a
    - ST a, R0
  - Flow-of-control optimization
    - goto L1
    - L1: goto L2
  - Algebraic simplification

Summary

- Three primary tasks
  - Instruction selection, register allocation and instruction ordering
- Input → Three-address code
- Output → Target code
  - Depends on the target machine and the instruction set available
- Code generation algorithm
  - Address descriptor
  - Register descriptor
- Peephole optimization