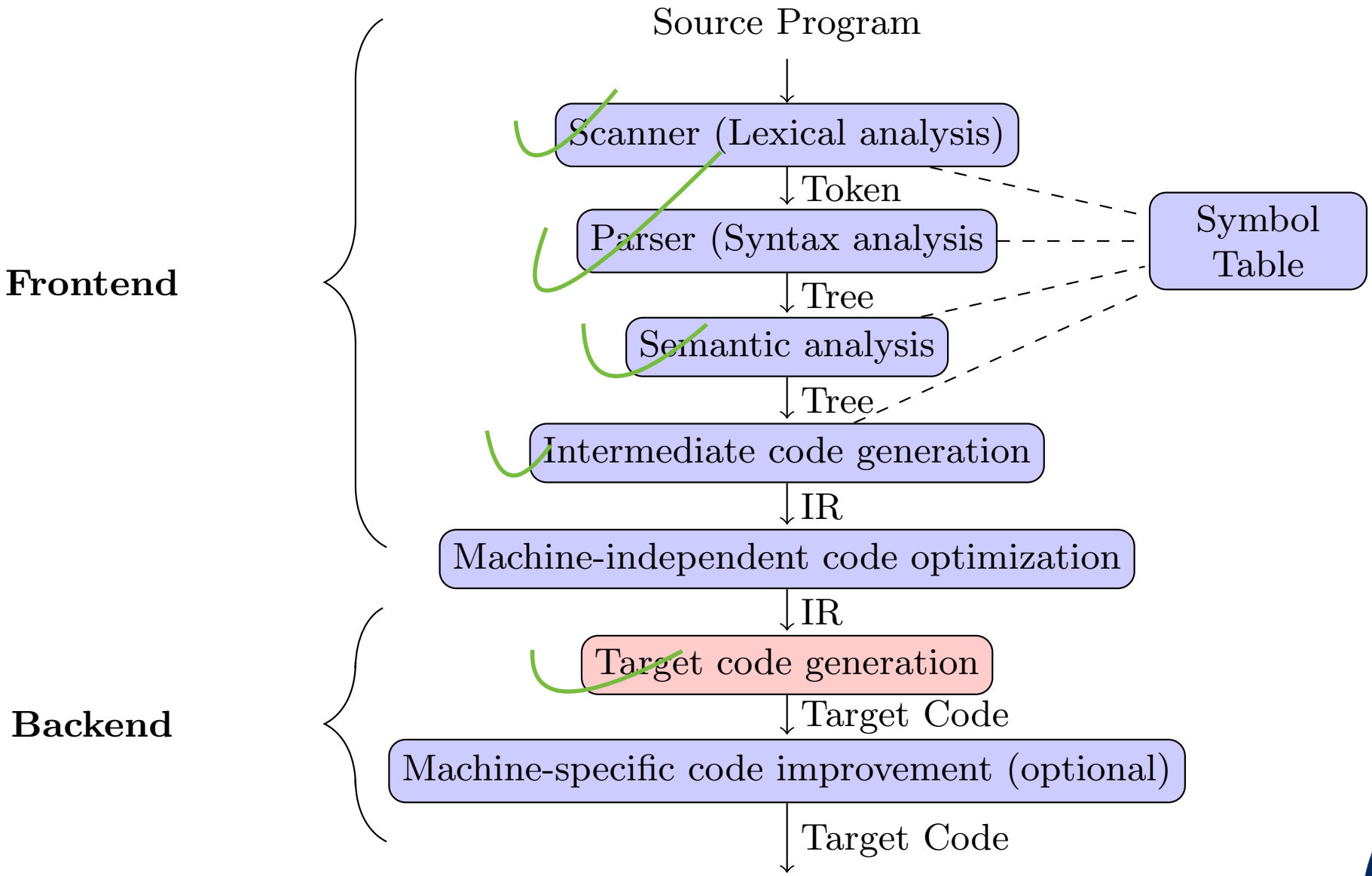
Code Generation

Lecture 20
Section 8.1 - 8.4

Rokan Uddin Faruqui

Associate Professor
Dept of Computer Science and Engineering
University of Chittaong, Bangladesh
Email: *rokan@cu.ac.bd*


Source Program
Scanner (Lexical analysis)
Token
Parser (Syntax analysis
Tree
Semantic analysis
Tree
Intermediate code generation
IR
Machine-independent code optimization
IR
Target code generation
Target Code
Machine-specific code improvement (optional)
Target Code
Symbol Table
Frontend
Backend

The compiler Backend

What have we obtained by the front-end transformations?

Assuming we have produced relatively low-level IR (like 3-address code), we can assume that all **syntactic** and **semantic** errors have been eliminated.

Outline

What code generation issues are important?

- Correctness (preserving the semantics) of the program.
- Speed of the target program (efficient use of the target machine's resources)
- Speed and structure of the code generator (length, easy to implement, test and maintain).

What code generation issues are important?

- Correctness (preserving the semantics) of the program.
- Speed of the target program (efficient use of the target machine's resources)
- Speed and structure of the code generator (length, easy to implement, test and maintain).

What code generation issues are important?

- Correctness (preserving the semantics) of the program.
- Speed of the target program (efficient use of the target machine's resources)
- Speed and structure of the code generator (length, easy to implement, test and maintain).

What code generation issues are important?

Hilbert's 10th Problem

1936 — Alonzo Church — Computable Function
Alan Turing —> Turing Machine

L = { a^nb^nc^n | n > 0 }

Generating an optimal target program for a given source program is **undecidable**. Many subprogroblems encountered in code generation are computationally **intractable**

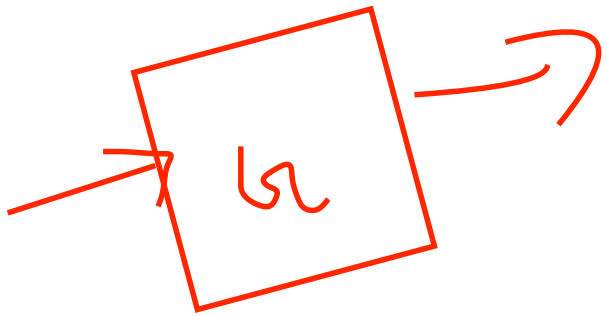
- there are heuristics to generate good code
- optimization on the IR helps

What code generation issues are important?

Generating an optimal target program for a given source program is **undecidable**. Many subprogroblems encountered in code generation are computationally **intractable**

- there are heuristics to generate good code
- optimization on the IR helps

Code Generator: Input



1. Three-address presentations (quadruples, triples, . . .)
2. Virtual machine presentations (bytecode, stack-machine, . . .)
3. Linear presentation (postfix, . . .)
4. Graphical presentation (syntax trees, DAGs, . . .)

Text

Code Generator: Input

1. Three-address presentations (quadruples, triples, . . .)
2. Virtual machine presentations (bytecode, stack-machine, . . .)
3. Linear presentation (postfix, . . .)
4. Graphical presentation (syntax trees, DAGs, . . .)

Code Generator: Input

1. Three-address presentations (quadruples, triples, ...)
2. Virtual machine presentations (bytecode, stack-machine, ...)
3. Linear presentation (postfix, ...)
4. Graphical presentation (syntax trees, DAGs, ...)

Code Generator: Input

1. Three-address presentations (quadruples, triples, ...)
2. Virtual machine presentations (bytecode, stack-machine, ...)
3. Linear presentation (postfix, ...)
4. Graphical presentation (syntax trees, DAGs, ...)

Code Generator : Target Program

1. Instruction set architecture (RISC, CISC)

   CISC MANY, SPECIALIZED instructions, FEW registers, variety of addressing-modes (CISC requires micro code)

   RISC FEW, SIMPLE, ORTHOGONAL instructions, MANY registers, simple addressing-modes (RISK drives hardware directly)

   Stack Machine Instruction set is organized with NO REGISTERS, all operations done on stack (e.g. ADD instuctions whence no arguments..)

2. Producing absolute machine-language program
3. Producing relocatable machine-language program
4. Producing assembly language programs

Code Generator : Target Program

1. Instruction set architecture (RISC, CISC)
   - **CISC** MANY, SPECIALIZED instructions, FEW registers, variety of addressing-modes (CISC requires micro code)
   - **RISC** FEW, SIMPLE, ORTHOGONAL instructions, MANY registers, simple addressing-modes (RISK drives hardware directly)

   Stack Machine Instruction set is organized with NO REGISTERS, all operations done on stack (e.g. ADD instuctions whence no arguments..)
2. Producing absolute machine-language program
3. Producing relocatable machine-language program
4. Producing assembly language programs

Code Generator : Target Program

1. Instruction set architecture (RISC, CISC)

   CISC MANY, SPECIALIZED instructions, FEW registers, variety of addressing-modes (CISC requires micro code)

   RISC FEW, SIMPLE, ORTHOGONAL instructions, MANY registers, simple addressing-modes (RISK drives hardware directly)

   Stack Machine Instruction set is organized with NO REGISTERS, all operations done on stack (e.g. ADD instuctions whence no arguments..)

2. Producing absolute machine-language program
3. Producing relocatable machine-language program
4. Producing assembly language programs

Code Generator : Target Program

1. Instruction set architecture (RISC, CISC)

   CISC MANY, SPECIALIZED instructions, FEW registers, variety of addressing-modes (CISC requires micro code)

   RISC FEW, SIMPLE, ORTHOGONAL instructions, MANY registers, simple addressing-modes (RISK drives hardware directly)

   Stack Machine Instruction set is organized with NO REGISTERS, all operations done on stack (e.g. ADD instuctions whence no arguments..)

2. Producing absolute machine-language program
3. Producing relocatable machine-language program
4. Producing assembly language programs

Main Tasks of Code Generator

t1 = a + b
t2 = t1 -c

Instruction selection choosing appropriate target-machine instructions to implement the IR statements

Registers allocation and assignment deciding what values to keep in which registers

Instruction ordering deciding in what order to schedule the execution of instructions

Instruction Selection

The complexity of mapping IR program into code-sequence for target machine depends on:

1. Level of IR (high-level or low-level)
2. Nature of instruction set (data type support)
3. Desired quality of generated code (speed and size)

Instruction Selection

The complexity of mapping IR program into code-sequence for target machine depends on:

1. Level of IR (high-level or low-level)
2. Nature of instruction set (data type support)
3. Desired quality of generated code (speed and size)

Instruction Selection

The complexity of mapping IR program into code-sequence for target machine depends on:

1. Level of IR (high-level or low-level)
2. Nature of instruction set (data type support)
3. Desired quality of generated code (speed and size)

Instruction Selection

1. How closely does the IR match the target language?
2. How good can the code be?
3. What kind of target architecture are we considering?

Instruction Selection

1. How closely does the IR match the target language?
2. How good can the code be?
3. What kind of target architecture are we considering?

Instruction Selection

1. How closely does the IR match the target language?
2. How good can the code be?
3. What kind of target architecture are we considering?

Register allocation and assignment

A register is the FASTEST COMPUTATIONAL UNIT on the target machine. When not enough registers to hold all values, the remaining values reside in memory

Allocation : Selecting the set of variables that will reside in registers at each point in the program

Assignment : Picking the specific register that a variable will reside in

- Hard! (NP complete).
- Operating system conventions.
- Hardware restrictions:
  - Special purpose registers (stack, index, frame, etc.)
  - Register pair (even/odd registers)

Evaluation Order

- Selecting the order in which computations are performed
- Affects the efficiency of the target code
- Picking a best order is NP-complete
- Some orders require fewer registers than others

Cost?

Everything is a trade-off when generating efficient code.

- Instruction code size.
- Instruction memory access time.
- Instruction execution time.

Outline

Simple Target-Machine

- Load/store operations

  $LD\ dst,\ addr$

  $ST\ x,\ r$

- Computation operations:

  $OP\ dst,\ src1,\ src2$

- Jump operations:

  $BR\ L$

- Conditional jumps:

  $Bcond\ r,\ L$

- Byte addressable
- n registers: $R0,\ R1, \ldots\ Rn-1$

8-bit Microprocessor

a=1000

a[4]=

b=a[i]

Simple Target-Machine

*a(r) = contents(contents(a + contents(r))

Addressing modes:

- variable name
- a(r) means: *contents(a + contents(r))*
- *a(r) means: *contents(contents(a + contents(r)))*
- immediate: *#constant (e.g. LD R1, #100)*

Cost

- cost of an instruction $=$ 1 $+$ cost of operands
- cost of register operand $=$ 0
- cost involving memory and constants $=$ 1
- cost of a program $=$ sum of instruction costs

Example: $x = y - z$

Code	Instructions	Notes Effects
$x = y - z$	`LD R1, y`	$R1 = y$
	`LD R2, z`	$R2 = z$
	`SUB R1, R1, R2`	$R1 = R1 - R2$
	`ST x, R1`	$x = R1$

Example: $b = a[i]$

a[4]

Code	Instructions	Notes Effects
$b = a[i]$	`LD R1, i`	$R1 = i$
	`MUL R1, R1, 8`	$R1 = R1 * 8$
	`LD R2, a(R1)`	$R2 = contents(a + contents(R1))$
	`ST b, R2`	$b = R2$

Example: $x = *p$

Code	Instructions	Notes Effects

$x = *p$	`LD R1, p`
	`LD R2, 0(R1)`
	`ST x, R2`
		$R1 = p$
		$R2 = contents(0 + contents(R1))$
		$x = R2$

Outline

Idea

1. **Basic Blocks** are maximal sequences of consecutive three-address instructions,
    1. control flow can only enter through first instruction,
    2. control flow can only leave from last instruction.

Basic Blocks

- Graph presentation of intermediate code
- Nodes of the graph are called basic blocks
- Edges indicate which block follows which other block.
- The graph is useful for doing better job in:
  - Register allocation
  - Instruction selection

Algorithm: Determine Basic Blocks

Input: A (numbered) sequence of three-address instructions.

Output: A mapping from the instruction (numbers) to basic blocks.

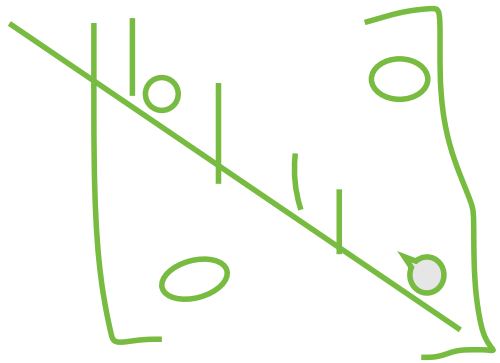
Method: Generate set of leaders.

1. The **first** instruction in sequence is the leader.
2. Any instruction that is the target of branch is a leader.
3. Any instruction that follows a branch is a leader.

Now create a basic block per leader, and associate the leader with it. Associate all non-leaders with the basic block of the nearest proceeding leader.

Example: 10×10 Identity Matrix

```
1  for i from 1 to 10 {
2    for j from 1 to 10 {
3
4          a[i,j] = 0.0;
5  } }
6    for i from 1 to 10 {
7
8          a[i,i] = 1.0; }
```

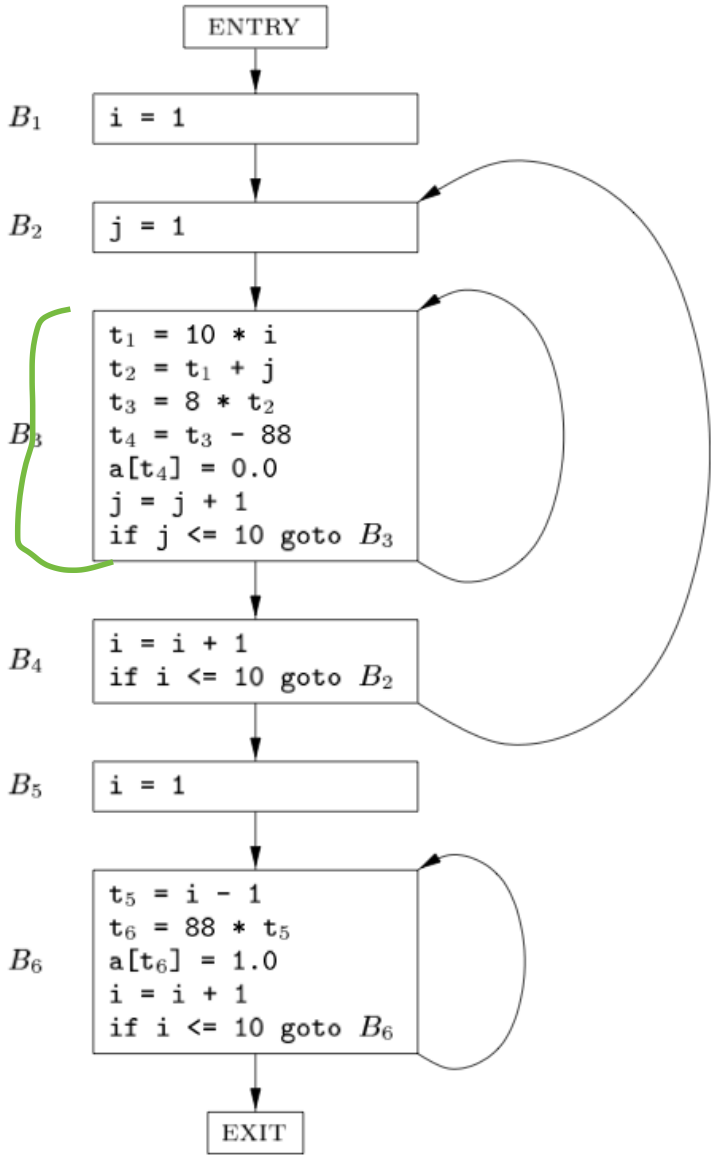


- Numbers are doubles (of size 8 bytes)

Three-address code:

```
1.  i = 1
2.  j = 1
3.  t1 = 10 * i
4.  t2 = t1 + j
5.  t3 = 8 * t2 // each element a double
6.  t4 = t3 - 88 // unused, allocated space, used
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
13.  t5 = i - 1
14.  t6 = 88 * t5 // strength reduction (reducing complexity)
15.  a[t6] = 1.0
16.  i = i + 1
17.  if i <= 10 goto 13
```

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 + 1
9. if j ¡= 10 goto 3
10. i = i + 1
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


ENTRY
B_1
i = 1
B_2
j = 1
B_3
t_1 = 10 * i
t_2 = t_1 + j
t_3 = 8 * t_2
t_4 = t_3 - 88
a[t_4] = 0.0
j = j + 1
if j <= 10 goto B_3
B_4
i = i + 1
if i <= 10 goto B_2
B_5
i = 1
B_6
t_5 = i - 1
t_6 = 88 * t_5
a[t_6] = 1.0
i = i + 1
if i <= 10 goto B_6
EXIT

Flow graphs

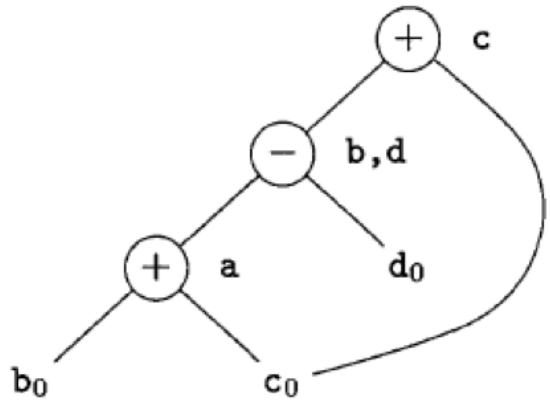
- **Flow Graph** is the the graph with basic blocks as nodes and branches as directed edges
- Picture of basic blocks with arrows for possible control flows.
- A loop is sequence of distinct basic blocks where
  1. Entry to first block is loop entry is only label where control comes from basic blocks outside the sequence.
  2. It is possible to reach the loop entry from any block in the sequence.

DAG Representation of Basic Blocks

- Leaves for initial values of variables (we may not know the values so we use a0, b0, ...)
- Node for each expression
- Node label is the expression operation
- Next to the node we put the variable(s) for which the node produced last definition
- Children of a node consist of nodes producing last definition of operands

Finding Local Common Subexpressions

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



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

Construct the DAG for the basic block

```
d = b * c
e = a + b
b = b * c
a = e - d
```

Dead Code Elimination

From the basic block DAG:

- Remove any root node that has no live variables
- Repeat until no nodes can be removed

```
a = b + c;
b = b - d
c = c + d
e = b + c
```