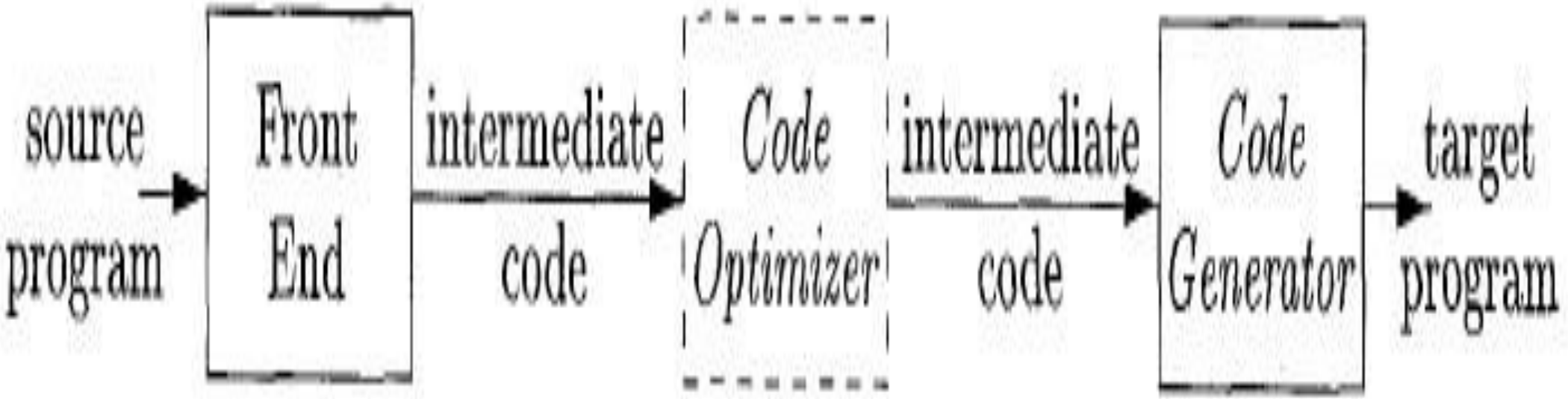
Code Generation

Code Generation



Code Generation

Requirements

- Preserve semantic meaning of source program
- Make effective use of available resources of target machine
- Code generator itself must run efficiently

Challenges

- Problem of generating optimal target program is undecidable
- Many subproblems encountered in code generation are computationally intractable

Main Tasks of Code Generator

- **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

Issues in the Design of a Code Generator

- **Input to the Code Generator**
- **The Target Program**
- **Instruction Selection**
- **Register Allocation**
- **Evaluation Order**

Input to the Code Generator

- The input to the code generator is the intermediate representation of the source program

Three-address representations	**quadruples, triples, indirect triples;**
Virtual machine representations	**Bytecodes and stack-machine code**
Linear representations	**Postfix notation**
Graphical representations	**Syntax trees and DAG's**

- Here it is assume that all syntactic and static semantic errors have been detected, that the necessary type checking has taken place, and that type conversion operators have been inserted wherever necessary.
- The code generator can therefore proceed on the assumption that its **input is free of these kinds of errors**.

The Target Program

- The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code.
- The most common target-machine architectures are
  - RISC (reduced instruction set computer)
    - It has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture
  - CISC (complex instruction set computer),
    - It has few registers, two-address instructions, a variety of addressing modes, several

register classes, variable-length instructions, and instructions with side effects

  - Stack based
    - operations are done by pushing operands onto a stack and then performing the

operations on the operands at the top of the stack

The Target Program (Cont)

- To overcome the high performance penalty of interpretation, *just-in-time* (JIT) Java compilers have been created.
- These JIT compilers translate bytecodes during run time to the native hardware instruction set of the target machine
- Producing an **absolute machine-language** program as output has the advantage that it can be placed in a fixed location in memory and immediately executed
- Producing a **relocatable machine-language** program (often called an *object module)* as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader.

Instruction Selection

- The code generator must map the IR program into a code sequence that can be executed by the target machine.
- The complexity of performing this mapping is determined by a factors such as
  - The level of the IR
  - The nature of the instruction-set architecture
  - The desired quality of the generated code.

Instruction Selection

- If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates. Such statement by-statement code generation, however, often produces poor code that needs
- If the IR reflects some of the low-level details of the underlying machine, then the code generator can use this information to generate more efficient code sequences.
- The quality of the generated code is usually determined by its speed and size.
- On most machines, a given IR program can be implemented by many different code sequences, with significant cost differences between the different implementations

Register Allocation

- A key problem in code generation is deciding what values to hold in what registers.
- Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values.
- Values not held in registers need to reside in memory.
- Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.
- The use of registers is often subdivided into two sub-problems:
  - **Register allocation**, during which we select the set of variables that will reside in registers at each

point in the program.

  - **Register assignment**, during which we pick the specific register that a variable will reside in.
- Finding an optimal assignment of registers to variables is difficult, even with single-register machines.
- Mathematically, the problem is **NP-complete**.
- The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register-usage conventions be observed.

Evaluation Order

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.
- However, picking a best order in the general case is a **difficult NP-complete problem**.
- Initially, we shall **avoid** the problem **by generating code for the three- address statements** in the order in which they have been produced by the intermediate code generator.

Simple Code Generator

Content..

- ✓ Introduction(Simple Code Generator)
- ✓ Register and Address Descriptors
- ✓ A Code-GenerationAlgorithm
- ✓ The Function getreg
- ✓ Generating Code for Other Types of Statements

Introduction

- **Code Generator:**

✓ Code generator can be considered as the final phase of compilation.

✓ It generates a target code for a sequence of three-address statement.

✓ It consider each statement in turn, remembering if any of the operands of the statement are currently in registers, and taking advantage of that fact if possible.

Introduction

- We assume that computed results can be left in registers an long as possible.
- Storing them only

a) if their register is needed for another computation (or)

b) Just before a procedure call, jump, or labeled statement

- Condition (b)implies that everything must be stored just before the end of a basic block.

Introduction

- The reason we must do so is that, after leaving a basic block, we may be able to go to several different blocks, or we may go to one particular block that can be reached from several others.
- In either case, we cannot, without extra effort, assume that a datum used by a block appears in the same register no matter how control reached that block.
- Thus, to avoid a possible error, our simple code- generator algorithm stores everything when moving across basic block boundaries as well as when procedure calls are made.

Introduction

- We can produce reasonable code for a three-address statement **a:=b+c**
- If we generate the single instruction **ADD Rj,Ri**
- with cost one
- If Ri contain b but c is in a memory location, we can generate the sequence

Add c,Ri **cost=2**

(or)

Mov c,Rj
Add Rj, Ri **cost=3**

Register and Address Descriptors

- The code generator has to track both the registers (for availability) and addresses (location of values) while generating the code. For both of them, the following two descriptors are used:
  - **Register descriptor**
  - **Address descriptor**

Register descriptor

- Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.

  - Keep track of what is currently in each register.
  - Initially all the registers are empty

Address descriptor

- Values of the names (identifiers) used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations.
  - Keep track of location where current value of the name can be found at runtime
  - The location might be a register, stack, memory address or a set of those

A Code-Generation Algorithm

- Basic blocks comprise of a sequence of three- address instructions. Code generator takes these sequence of instructions as input.
- For each three-address statement of the form

x:=y op z

we perform the following actions

I. Call function getReg, to decide the location of L.

II. Determine the present location (register or memory) of **y** by consulting the Address Descriptor of **y**. If **y** is not presently in register **L**, then generate the following instruction to copy the value of **y** to **L**

MOV y', L

where **y'** represents the copied value of **y**.

III. Determine the present location of **z** using the same method used in step 2 for **y** and generate the following instruction:

OP z', L

where **z'** represents the copied value of **z**.

IV. Now L contains the value of y OP z, that is intended to be assigned to **x**. So, if L is a register, update its descriptor to indicate that it contains the value of **x**. Update the descriptor of **x** to indicate that it is stored at location **L**.

If y and z has no further use, they can be given back to the system.

The Function getreg

- Code generator uses *getReg* function to determine the status of available registers and the location of name values. *getReg* works as follows:
- If variable Y is already in register R, it uses that register.
- Else if some register R is available, it uses that register.
- Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions.

The Function getreg

1. If Y is in register (that holds no other values) and Y is not live and has no next use after

   X = Y op Z

   then return register of Y for L.

2. Failing (1) return an empty register

3. Failing (2) if X has a next use in the block or op requires register then get a register R, store its content into M (by Mov R, M) and use it.

4. else select memory location X as L

Example

- For example, the assignment d := (a - b) + (a - c) + (a - c) might be translated into the following three-address code sequence:
- $t_1 = a - b$
- $t_2 = a - c$
- $t_3 = t_1 + t_2$
- $d = t_3 + t_2$

Example

Stmt	code	reg desc	addr desc
$t_1=a-b$	mov a,R_0	R_0 contains t_1	t_1 in R_0
	sub b,R_0		
$t_2=a-c$	mov a,R_1	R_0 contains t_1	t_1 in R_0
	sub c,R_1	R_1 contains t_2	t_2 in R_1
$t_3=t_1+t_2$	add R_1,R_0	R_0 contains t_3	t_3 in R_0
		R_1 contains t_2	t_2 in R_1
$d=t_3+t_2$	add R_1,R_0	R_0 contains d	d in R_0
	mov R_0,d		d in R_0 and memory

Generating Code for Other Types of Statements

- Indexed assignment

  a:=b[i]

  and

  a[i]:=b

- Pointer assignment

  a:=*p

  and

  *p:=a

⑥ Assignment statements

- can be a type of integer, real, array & record.

Syntax-directed translation scheme for generation of 3-address code for assignment stmts

$S \rightarrow id = E$

$E \rightarrow E_1 + E_2$

~~$E \rightarrow E_1 * E_2$~~

$E \rightarrow -E_1$

$E \rightarrow (E_1)$

$E \rightarrow id$

- Following syntax-directed defn builds up 3-addr code for an assignment stmt S using attribute 'code' for S and attribute 'code' and 'addr' for expression E.

MAY 2015
M T W T F S S
13 14 15 16

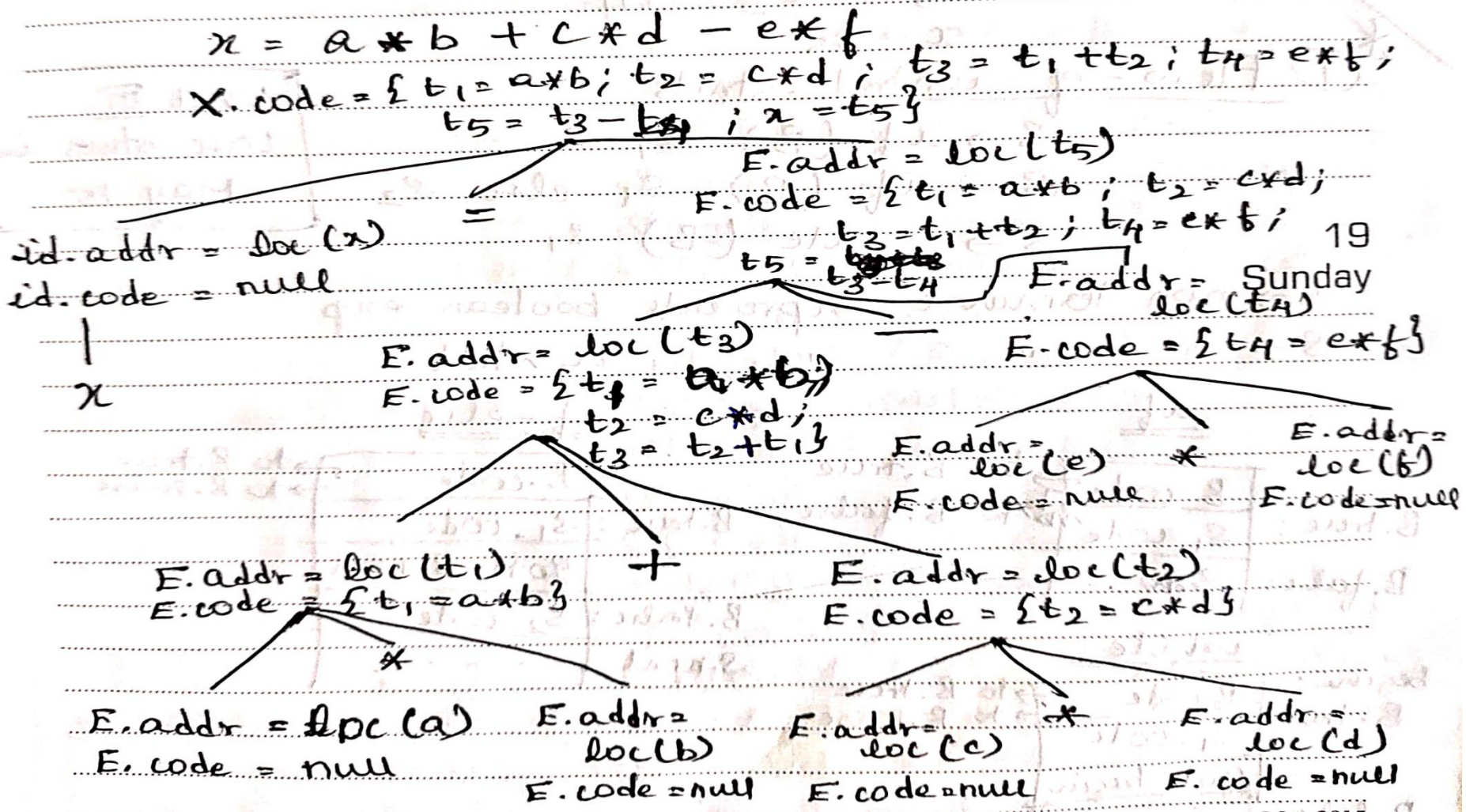
Friday

- Attributes S.code and E.code denote 3-addr code for S and E.
- Att E.addr denotes addr that will hold the value of E.
addr can be a name, a constant or a compiler-generated temporary.

Production	Semantic rules
$S \to id = E;$	S.code = E.code || gen(top.get('id.lexeme') '=' E.addr)
$E \to E_1 + E_2$	E.addr = new Temp() E.code = E_1.code || E_2.code || gen(E.addr '=' E_1.addr '+' E_2.addr)
$\mid -E_1$	E.addr = new Temp() E.code = E_1.code || gen(E.addr '=' 'minus' E_1.addr)
$\mid (E_1)$	E.addr = E_1.addr E.code = E_1.code
$\mid id$	E.addr = top.get('id.lexeme') E.code = ''

E → id

- Consider E → id. When an exp is a single identifier, say x, then x itself holds the value of the expression.
- Semantic rules for this prodn define E.addr to point to symbol-table entry for this instance of id. Let 'top' denote current symbol table.



∴ 3-addr code for above stmt is

$t_1 = a * b$

$t_2 = c * d$

$t_3 = t_1 + t_2$

$t_4 = e * f$

$t_5 = t_3 - t_4$

$x = t_5$