

---

**CMPSC 160**  
**Translation of Programming Languages**

**Lecture 14: Code Generation: Stack  
Machine Code**

---

# Code Generation

---

- We will discuss code generation for stack machine code and three-address code.
  - We will discuss how to deal with Boolean values, control flow (such as loops and if-then-else statements), functions, and etc.
-

# Register-based VS. Stack-based Machines

---

A **register**-based machine has a number of registers used for calculations.  $2 + 3$  would work something like this:

- `LOADI R4,#2;` : Load immediate 2 into register 4
- `LOADI R5,#3;` : Load immediate 3 into register 5
- `ADD R4,R5;` : Add R4 and R5, storing result in R4

On a **stack**-based machine, computation would work like this

- `PUSHI #2;` : Push immediate 2 onto stack
  - `PUSHI #3;` : Push immediate 3 onto stack
  - `ADD;` : Pop top two numbers, add them, and push results to the top of the stack.
-

# Key Challenge in Code Generation

---

- Different hardware backends.
  - Good abstraction is needed.
  - We first need to understand general code generation principles.
-

# General Code Generation Principles

---

- Each different type of computer has a different type of assembly code, so what are the **general principles**?
  - Key: the code generation method will usually need refining to produce code of a reasonable quality: the interplay between **main memory** and the **arithmetic logic unit (ALU)** in the CPU:
    - some computers put values from the main memory into a **small, fast memory** within the CPU before they can be used
    - most computers can normally use **one operand** in main memory per instruction; some computers can use more.
-

# General Code Generation Principles

---

- Fast memory within the CPU used for arithmetic:
    - several arithmetic registers (can access different registers)
    - a stack memory (only access the top parts)
    - a hybrid mode: stack + registers (the most common case is to use a single accumulator register)
  - ALU can only operate on fast memory or also main memory
    - stack-based machine: only stack + register (if in a hybrid mode). That is to say, data has to be loaded into fast memory before execution
    - register-based machine: most of the computations and temporary results are stored in register. Some machine also allows arithmetic instructions with operands of data in memory.
-

# Our Focus

---

- We first focus on generating code for a stack machine (w/ and w/o accumulator) with syntax-based translation.
  - We will simulate the execution of resulting code on a real machine, e.g., the MIPS/x86 processor (or simulator)
    - We simulate stack machine instructions using MIPS/x86 instructions and registers.
  - Code generation for function calls (especially the control transfer with specialized activation record designs).
-

# Why Use a Stack Machine ?

---

- Easy to describe & understand: Each operation takes operands from the same place and puts results in the same place, In particular,
    - Location of the operands is implicit
    - Always on the top of the stack
    - No need to specify operands explicitly
    - No need to specify the location of the result
    - Instruction “**add**” as opposed to “**add r1, r2**” ⇒ Smaller encoding of instructions ⇒ More **compact** programs
  - This is one reason why **Python** and **Java Bytecodes** use a stack evaluation model
-



# Why Use a Stack Machine ?

---

- Easy to generate from a compiler's perspective: code generation for stack machines is much simpler than for register machines, since e.g., no **register allocation** is needed (we'll talk about register allocation later in this course).
  - Compact object code, which saves memory.
    - The reason for this is that machine commands have no, or only one argument, unlike instructions for register machines.
  - Simple CPUs (= cheap, easy to manufacture).
  - Used in e.g., the JVM and WebAssembly.
  - But stack machines also have disadvantages, primarily that they are **slow** (see e.g., the Wikipedia page on stack machines), but for us here simplicity of code generation is key.
-

# Stack Machine Instructions

---

- Instructions of a typical stack machine
  - These are similar to instructions of JVM

`multiply, divide, add, subtract`

pop top two values from the operand stack perform the operation on them  
and **push the result back to the operand stack**

`and, or`

pop top two values from the operand stack perform the bitwise “*and*” or  
“*or*” operation on them and push the result back to the operand stack

`push <constant>`

push the constant value to the stack

`load <location>`

push the value at the given memory location to the stack

`store <location>`

pop the value at the top of the stack and store it to the given memory  
location

---

# Stack Machine Instructions

---

- Control flow

`goto <label>`

jumps to the instruction with the given label (address of instructions)

`ifeq <label>`

pop the top element from the operand stack and jump to the instruction with the given label if it is equal to 0

`ifne <label>`

pop the top element from the operand stack and jump to the instruction with the given label if it is not equal to 0

`if_cmplt (or iflt) <label>`

pop the top two elements from the operand stack and jump to the instruction with the given label if the one popped second is less than the one popped first

Similar instructions:

`if_cmpeq, if_icmpne, if_cmpgt, if_cmple, if_cmpge`

---

# A simple language

---

## Step1: Language only with Arithmetic Expressions

$S \rightarrow \text{id} := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow (E_1)$

$E \rightarrow -E_1$

$E \rightarrow \text{id}$

$E \rightarrow \text{num}$

## Step2a: Extended with control statements

$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$

$S \rightarrow \text{while } E \text{ do } S_1$

$S \rightarrow S_1 ; S_2$

## Step2b: Boolean Expressions to work with control statements

$E \rightarrow E_1 \text{ relop } E_2$

$E \rightarrow E_1 \text{ and } E_2$

$E \rightarrow E_1 \text{ or } E_2$

$E \rightarrow \text{not } E_1$

$E \rightarrow \text{true}$

$E \rightarrow \text{false}$

---

# Stack Machine Code

```
if (x < y)
  x = 5*y + 5*y/3;
else
  y = 5;
x = x+y;
```

pushes the value  
at the location x to  
the stack

```
load x
load y
iflt L1
push 0
goto L2
L1: push 1
L2: ifne L3:
goto L4
L3: push 5
load y
multiply
push 5
load y
multiply
push 3
divide
add
store x
goto L5
L4: push 5
store y
L5: load x
load y
add
store x
```

pops the top  
two elements and  
compares them

pops the top  
element and  
compares it to 0

pops the top two  
elements, multiplies  
them, and pushes the  
result back to the stack

stores the value at the  
top of the stack to the  
location x

# Stack-Based Code Generation for Expressions and Assignment

Attributes:	$E.code$ : sequence of instructions that are generated for $E$ <i>(no place for an expression is needed since the result of an expression is stored in the operand stack)</i>
Procedures:	gen(): Generates instruction <i>(have to call it with appropriate arguments)</i> lookup(id.name): Returns the location of id from the symbol table    denotes concatenation

## Productions

$S \rightarrow \text{id} := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow ( E_1 )$

$E \rightarrow - E_1$

$E \rightarrow \text{id}$

$E \rightarrow \text{num}$

# Stack-Based Code Generation for Expressions and Assignment

Attributes:	$E.code$ : sequence of instructions that are generated for $E$ <i>(no place for an expression is needed since the result of an expression is stored in the operand stack)</i>
Procedures:	gen(): Generates instruction <i>(have to call it with appropriate arguments)</i> lookup(id.name): Returns the location of id from the symbol table    denotes concatenation

## Productions

$S \rightarrow id := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow ( E_1 )$

$E \rightarrow - E_1$

$E \rightarrow id$

$E \rightarrow num$

## Semantic Rules

$id.place \leftarrow lookup(id.name);$

$S.code \leftarrow E.code \parallel gen('store' id.place);$

$E.code \leftarrow E_1.code \parallel E_2.code \parallel gen('add');$

*(arguments for the add instruction are at the top of the stack)*

$E.code \leftarrow E_1.code \parallel E_2.code \parallel gen('multiply');$

$E.code \leftarrow E_1.code;$

$E.code \leftarrow gen('push 0') \parallel E_1.code \parallel gen('subtract');$

*(if there is an instruction to negate the value at the top, then we can use that)*

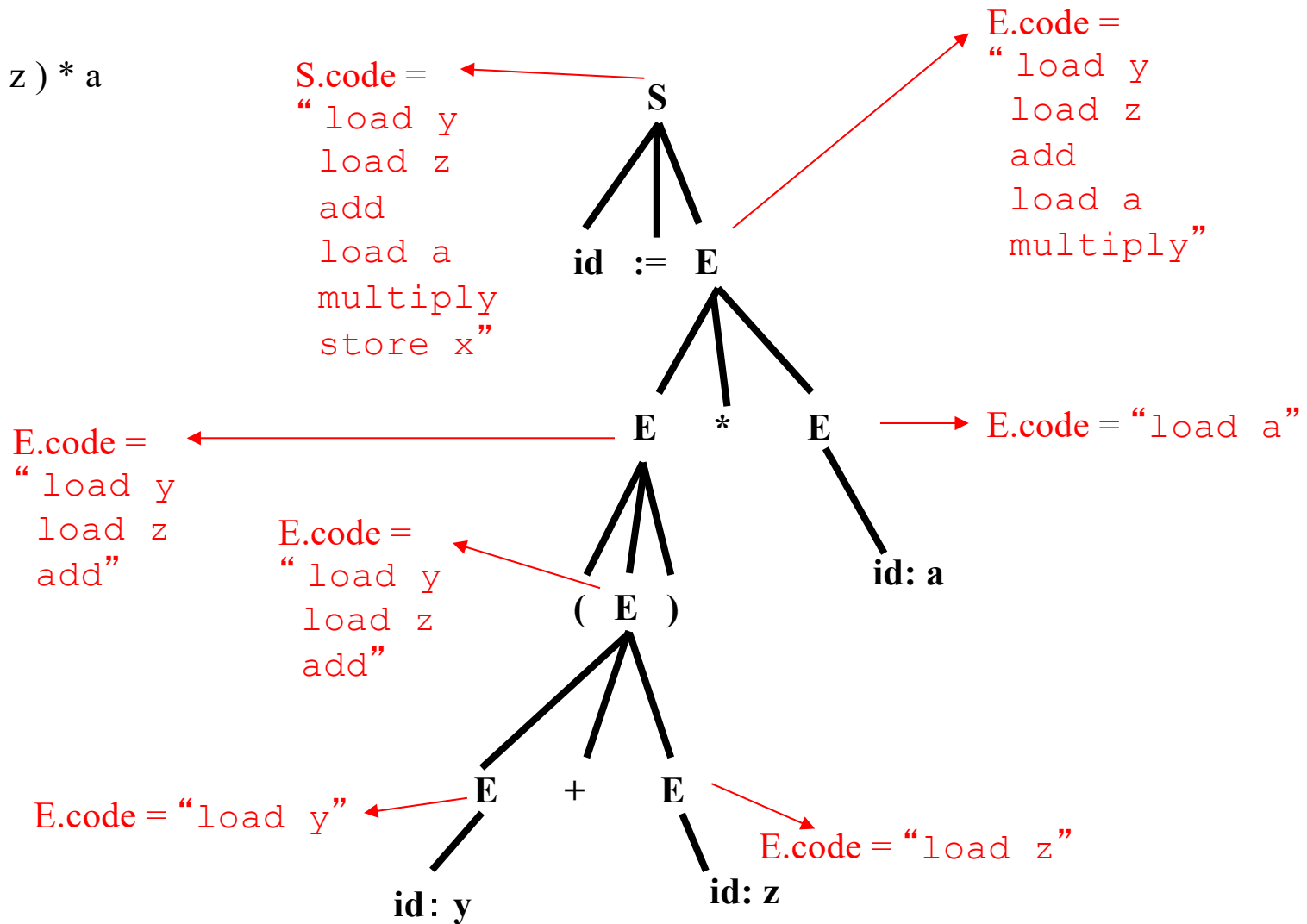
$id.place \leftarrow lookup(id.name);$

$E.code \leftarrow gen('load' id.place);$

$E.code \leftarrow gen('push' num.value);$

# Example

$x := (y + z) * a$





# Note

---

- The code for  $e1 + e2$  and other arithmetic expression unit is a template with “holes” for code for  $e1$  and  $e2$ .
  - Stack machine code generation is **recursive**.
  - Code for  $e1 + e2$  consists of code for  $e1$  and  $e2$  glued together.
  - Code generation—at least for expressions—can be written as a postorder transversal of the AST.
    - suppose we have already used ad-hoc syntax based translation for address translation for different variables and store them in the symbol table (see lecture 12)
-

# Code Generation for Boolean Expressions

---

- Two approaches
    - Numerical representation
    - Implicit representation
  - Numerical representation
    - Use 1 to represent true, use 0 to represent false
    - For stack machine code store this result in the stack
    - For three-address code store this result in a temporary
  - Implicit representation
    - For the boolean expressions which are used in flow-of-control statements (such as if-statements, while-statements etc.) boolean expressions do not have to explicitly compute a value, they just need to branch to the right instruction
    - Generate code for boolean expressions which branch to the appropriate instruction based on the result of the boolean expression
-

# Boolean Expressions: Numerical Representation, Stack Machine Code

---

Attributes :	$E.code$ : sequence of instructions that are generated for $E$ relop. $op$ : operator can be ==, !=, <=, >=, >, <
Procedures:	newlabel(): generates a new label

## Productions

$E \rightarrow E_1 \text{ relop } E_2$

$E \rightarrow E_1 \text{ and } E_2$

$E \rightarrow E_1 \text{ or } E_2$

$E \rightarrow \text{not } E_1$

$E \rightarrow \text{true}$

$E \rightarrow \text{false}$

## Semantic Rules

# Boolean Expressions: Numerical Representation, Stack Machine Code

Attributes :	$E.code$ : sequence of instructions that are generated for $E$ $relop.op$ : operator can be $=$ , $!=$ , $<=$ , $>=$ , $>$ , $<$
Procedures:	$newlabel()$ : generates a new label

## Productions

$E \rightarrow E_1 \text{ relop } E_2$

$E \rightarrow E_1 \text{ and } E_2$

$E \rightarrow E_1 \text{ or } E_2$

$E \rightarrow \text{not } E_1$

$E \rightarrow \text{true}$

$E \rightarrow \text{false}$

## Semantic Rules

$L1 \leftarrow newlabel();$

$L2 \leftarrow newlabel();$

$E.code \leftarrow E_1.code \parallel E_2.code$

$\parallel gen('if\_cmp' \text{ relop.op } L1)$

$\parallel gen('push 0') \parallel gen('goto' L2)$

$\parallel gen('L1 :') \parallel gen('push 1') \parallel gen('L2 :');$

$E.code \leftarrow E_1.code \parallel E_2.code \parallel gen('and');$


$E.code \leftarrow E_1.code \parallel E_2.code \parallel gen('or');$

$E.code \leftarrow E_1.code \parallel gen('negate');$

$E.code \leftarrow gen('push 1');$

$E.code \leftarrow gen('push 0');$

This will be the label of the instruction that comes after this one



# Numerical Representation of Boolean Expressions

---

Input boolean expression:  $x < y$  and  $a == b$

Generated stack machine code:

```
        load x
        load y
        if_cmplt L1
        push 0
        goto L2
L1:     push 1
L2:     load a
        load b
        if_cmpeq L3
        push 0
        goto L4
L3:     push 1
L4:     and
```

code generated for  $x < y$

code generated for  $a == b$