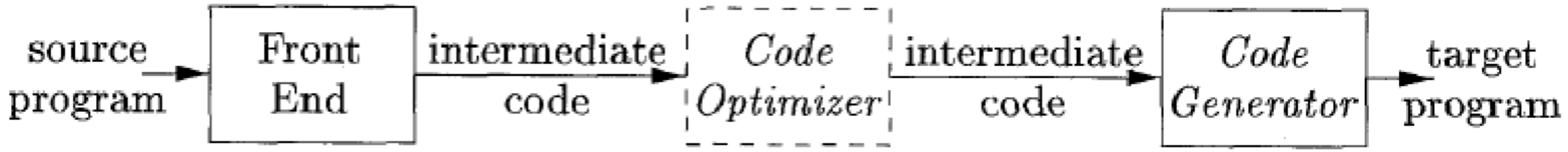
Chapter-08

Code Generation

Code Generation



Position of code generator

Issues in the Design of a Code Generator

- While the details are dependent on the specifics of
  - the intermediate representation,
  - the target language and
  - the run-time system

tasks such as

- instruction selection,
- register allocation and assignment and
- instruction ordering

are encountered in the design of almost all code generators

Issues in the Design of a Code Generator(Cont…)

- The most important criterion for a code generator is that it produces correct code
- Correctness takes on special significance because of the number of special cases that a code generator might face
- Given the premium on correctness, designing a code generator so it can be easily implemented, tested and maintained is an important design goal

Input to the Code Generator

- The input to the code generator is
  - the intermediate representation of the source program produced by the front end
  - along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR

Input to the Code Generator(Cont…)

- The many choices for the IR include
  - three-address representations such as quadruples, triples, indirect triples
  - virtual machine representations such as byte codes and stack-machine code
  - linear representations such as postfix notation and
  - graphical representations such as syntax trees and DAG’s

Input to the Code Generator(Cont…)

- We assume that the front end has
  - scanned,
  - parsed and
  - translated the source program into a relatively low-level IR

so that the values of the names appearing in the IR can be represented by quantities that the target machine can directly manipulate such as integers and floating-point numbers

Input to the Code Generator(Cont…)

- We also assume that
  - all syntactic and static semantic errors have been detected
  - the necessary type checking has taken place and
  - 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), CISC (complex instruction set computer), and stack based

The Target Program(Cont…)

- A RISC machine typically has
  - many registers
  - three-address instructions
  - simple addressing modes and
  - a relatively simple instruction-set architecture

The Target Program(Cont…)

- In contrast, a CISC machine typically has
  - few registers,
  - two-address instructions,
  - a variety of addressing modes,
  - several register classes,
  - variable-length instructions and
  - instructions with side effects

The Target Program(Cont…)

- In a stack-based machine operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack
- To achieve high performance the top of the stack is typically kept in registers
- Stack-based machines almost disappeared because it was felt that the stack organization was too limiting and required too many swap operations

The Target Program(Cont…)

- However, stack-based architectures were revived with the introduction of the Java Virtual Machine (JVM)
- The JVM is a software interpreter for Java byte codes, an intermediate language produced by Java compilers
- The interpreter provides software compatibility across multiple platforms, a major factor in the success of Java

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 byte codes during run time to the native hardware instruction set of the target machine
- Another approach to improving Java performance is to build a compiler that compiles directly into the machine instructions of the target machine by passing the Java byte codes entirely

The Target Program(Cont…)

- 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
- Programs can be compiled and executed quickly

The Target Program(Cont…)

- 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
- Must pay the added expense of linking and loading if we produce relocatable object modules

The Target Program(Cont…)

- But we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module
- If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program modules

The Target Program(Cont…)

- Producing an assembly-language program as output makes the process of code generation somewhat easier
- We can generate symbolic instructions and use the macro facilities of the assembler to help generate code

The Target Program(Cont…)

- We shall use a very simple RISC-like computer as our target machine
- We add to it some CISC-like addressing modes so that we can also discuss code-generation techniques for CISC machines
- For readability, we use assembly code as the target language
- As long as addresses can be calculated from offsets and other information stored in the symbol table the code generator can produce relocatable or absolute addresses for names

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(Cont…)

- 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 further optimization
- 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

Instruction Selection(Cont…)

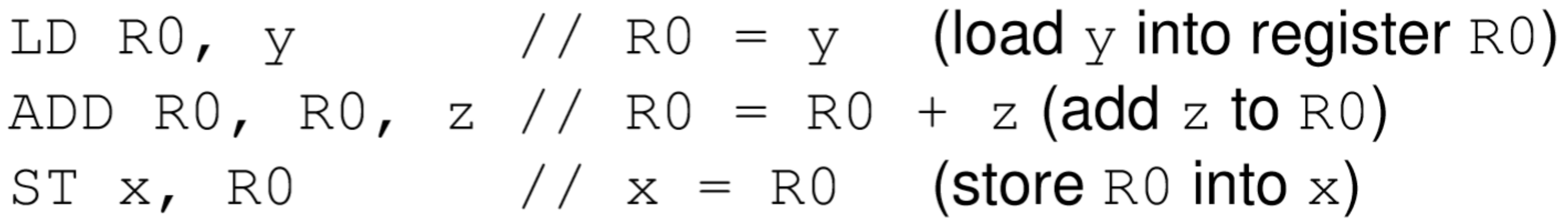
- The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection
- For example, the uniformity and completeness of the instruction set are important factors
- If the target machine does not support each data type in a uniform manner then each exception to the general rule requires special handling
- On some machines, for example, floating-point operations are done using separate registers

Instruction Selection(Cont…)

- Instruction speed is another important factor
- If we do not care about the efficiency of the target program then instruction selection is straightforward
- For each type of three-address statement we can design a code skeleton that defines the target code to be generated for that construct

Instruction Selection(Cont…)

- For example, every three-address statement of the form **x = y + z**, where **x**, **y** and **z** are statically allocated, can be translated into the code sequence



```
LD R0, y          // R0 = y       (load y into register R0)
ADD R0, R0, z     // R0 = R0 + z  (add z to R0)
ST x, R0          // x = R0       (store R0 into x)
```

Instruction Selection(Cont…)

- This strategy often produces redundant loads and stores
- For example, the sequence of three-address statements

  a = b +c

  d = a +e

would be translated into

```
LD R0, b        // R0 = b
ADD R0, R0, c // R0 = R0 + c
ST a, R0        // a = R0
LD R0, a        // R0 = a
ADD R0, R0, e // R0 = R0 + e
ST d, R0        // d = R0
```

Here, the fourth statement is redundant since it loads a value that has just been stored

Instruction Selection(Cont…)

- 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
- A naive translation of the intermediate code may therefore lead to correct but unacceptably inefficient target code
- We need to know instruction costs in order to design good code sequences but unfortunately accurate cost information is often difficult to obtain

Instruction Selection(Cont…)

- For example, if the target machine has an “increment” instruction (INC) then the three-address statement **a = a + 1** may be implemented more efficiently by the single instruction **INC a**
- Rather than by a more obvious sequence that loads a into a register, adds one to the register and then stores the result back into a:

```
LD R0, a          // R0 = a
ADD R0, R0, #1    // R0 = R0 + 1
ST a, R0          // a = R0
```

The Target Language

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator
- We shall use as a target language assembly code for a simple computer that is representative of many register machines

A Simple Target Machine Model

- Our target computer models a three-address machine with load and store operations, computation operations, jump operations and conditional jumps
- The underlying computer is a byte-addressable machine with n general-purpose registers R0, R1, ..., Rn−1
- A full-fledged assembly language would have scores of instructions

A Simple Target Machine Model(Cont…)

- We use a very limited set of instructions and assume that all operands are integers
- Most instructions consists of an operator followed by a target followed by a list of source operands
- A label may precede an instruction

A Simple Target Machine Model(Cont…)

We assume the following kinds of instructions are available:

- **Load Operations:** The instruction **LD dst, addr** loads the value in location addr into location dst
- This instruction denotes the assignment **dst = addr**
- The most common form of this instruction is **LD r, x** which loads the value in location x into register r
- An instruction of the form **LD r1,r2** is a register-to-register copy in which the contents of register r2 are copied into register r1

A Simple Target Machine Model(Cont…)

We assume the following kinds of instructions are available:

- **Store Operations:** The instruction **ST x,r** stores the value in register r into the location x
- This instruction denotes the assignment **x = r**

A Simple Target Machine Model(Cont…)

We assume the following kinds of instructions are available:

- **Computation Operations** of the form **OP dst,src1,src2** where OP is a operator like ADD or SUB and dst, src1, and src2 are locations, not necessarily distinct
- The effect of this machine instruction is to apply the operation represented by OP to the values in locations src1 and src2 and place the result of this operation in location dst
- For example, **SUB r1,r2,r3** computes **r1 = r2 –r3**
- Any value formerly stored in r1 is lost but if r1 is r2 or r3, the old value is read first
- Unary operators that take only one operand do not have a src2

A Simple Target Machine Model(Cont…)

We assume the following kinds of instructions are available:

- **Unconditional jumps:** The instruction **BR L** causes control to branch to the machine instruction with label L (BR stands for branch)

A Simple Target Machine Model(Cont…)

We assume the following kinds of instructions are available:

- **Conditional jumps** of the form **Bcond r,L** where r is a register, L is a label and cond stands for any of the common tests on values in the register r
- For example, **BLTZ r,L** causes a jump to label L if the value in register r is less than zero and allows control to pass to the next machine instruction if not

A Simple Target Machine Model(Cont…)

We assume our target machine has a variety of addressing modes:

- In instructions, a location can be a **variable name x** referring to the memory location that is reserved for x (that is l-value of x)

A Simple Target Machine Model(Cont…)

We assume our target machine has a variety of addressing modes:

- A location can also be an **indexed address of the form a(r)** where a is a variable and r is a register
- The memory location denoted by a(r) is computed by taking the l-value of a and adding to it the value in register r

A Simple Target Machine Model(Cont…)

We assume our target machine has a variety of addressing modes:

- For example, the instruction **LD R1, a(R2)** has the effect of setting **R1 = contents(a +contents(R2))** where contents(x) denotes the contents of the register or memory location represented by x
- This addressing mode is useful for accessing arrays, where a is the base address of the array (that is the address of the first element) and r holds the number of bytes past that address we wish to go to reach one of the elements of array a

A Simple Target Machine Model(Cont…)

We assume our target machine has a variety of addressing modes:

- A memory location can be an **integer indexed by a register**
- For example, **LD R1, 100(R2)** has the effect of setting **R1 = contents(100+contents(R2))** that is of loading into R1 the value in the memory location obtained by adding 100 to the contents of register R2

A Simple Target Machine Model(Cont…)

We assume our target machine has a variety of addressing modes:

- We also allow two **indirect addressing modes**
- **$*$r** means the memory location found in the location represented by the contents of register r and
- **$*$100(r)** means the memory location found in the location obtained by adding 100 to the contents of r

A Simple Target Machine Model(Cont…)

We assume our target machine has a variety of addressing modes:

- For example, **LD R1, *100(R2)** has the effect of setting **R1 = contents(contents(100+contents(R2)))** that is of loading into R1 the value in the memory location stored in the memory location obtained by adding 100 to the contents of register R2

A Simple Target Machine Model(Cont…)

We assume our target machine has a variety of addressing modes:

- Finally, we allow an **immediate constant addressing mode**
- The constant is prefixed by **#**
- The instruction **LD R1, #100** loads the integer 100 into register R1 and
- **ADD R1, R1, #100** adds the integer 100 into register R1

- Comments at the end of instructions are preceded by //

Example 8.2

- The three-address statement **x = y - z** can be implemented by the machine instructions:

```
LD R1, y          // R1 = y
LD R2, z          // R2 = z
SUB R1, R1, R2 // R1 = R1 - R2
ST x, R1          // x = R1
```

- We can do better perhaps
- One of the goals of a good code-generation algorithm is to avoid using all four of these instructions whenever possible

Example 8.2(Cont…)

- The three-address statement **x = y - z** can be implemented by the machine instructions:

```
LD R1, y        // R1 = y
LD R2, z        // R2 = z
SUB R1, R1, R2  // R1 = R1 - R2
ST x, R1        // x = R1
```

- For example, y and/or z may have been computed in a register and if so we can avoid the LD step(s)
- Likewise, we might be able to avoid ever storing x if its value is used within the register set and is not subsequently needed

Example 8.2(Cont…)

- Suppose a is an array whose elements are 8-byte values
- Also assume elements of a are indexed starting at 0
- We may execute the three-address instruction **b = a [i]** by the machine instructions:

```
LD R1, i      // R1 = i
MUL R1, R1, 8// R1 = Rl * 8
LD R2, a(R1) // R2 = contents(a + contents(R1))
ST b, R2      // b = R2
```

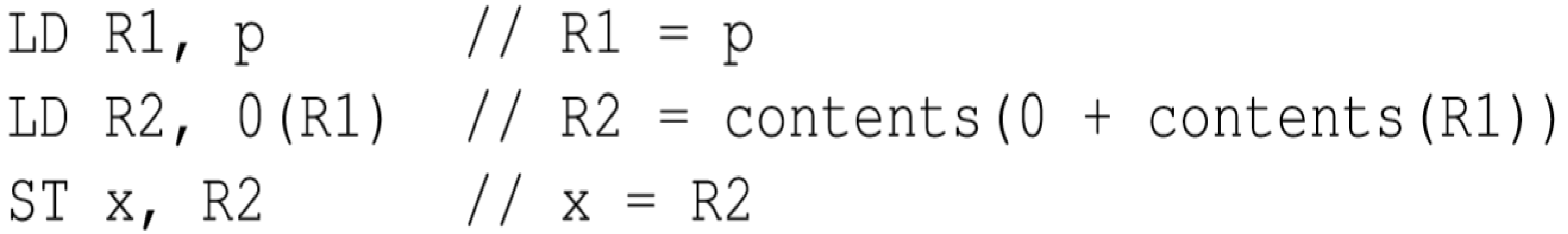
Example 8.2(Cont…)

- Similarly, the assignment into the array a represented by three-address instruction **a[j] = c** is implemented by:

```
LD R1, c        // R1 = c
LD R2, j        // R2 = j
MUL R2, R2, 8 // R2 = R2 * 8
ST a(R2), R1  // contents(a + contents(R2)) = R1
```

Example 8.2(Cont…)

- To implement a simple pointer indirection, such as the three-address statement **x = *p**, we can use machine instructions like:



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

Example 8.2(Cont…)

- The assignment through a pointer ***p = y** is similarly implemented in machine code by:

```
LD R1, p      // R1 = p
LD R2, y      // R2 = y
ST 0(R1), R2 // contents(0 + contents(R1)) =  R2
```

Example 8.2(Cont…)

- Finally, consider a conditional jump three-address instruction like **if x < y goto L**
- The machine-code equivalent would be something like:

```
LD R1, x         // R1 = x
LD R2, y         // R2 = y
SUB R1, R1, R2 // R1 = R1 - R2
BLTZ R1, M      // if R1 < 0 jump to M
```

- Here, M is the label that represents the first machine instruction generated from the three-address instruction that has label L

Example 8.2(Cont…)

- As for any three-address instruction we hope that we can save some of these machine instructions because the needed operands are already in registers or because the result need never be stored

Program and Instruction Costs

- We often associate a cost with compiling and running a program
- Depending on what aspect of a program we are interested in optimizing, some common cost measures are the length of compilation time and the size, running time and power consumption of the target program

Program and Instruction Costs(Cont…)

- Determining the actual cost of compiling and running a program is a complex problem
- Finding an optimal target program for a given source program is an undecidable problem in general and many of the sub problems involved are NP-hard
- As we have indicated, in code generation we must often be content with heuristic techniques that produce good but not necessarily optimal target programs

Program and Instruction Costs(Cont…)

- We shall assume each target-language instruction has an associated cost
- For simplicity, we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands
- This cost corresponds to the length in words of the instruction
- Addressing modes involving registers have zero additional cost while those involving a memory location or constant in them have an additional cost of one because such operands have to be stored in the words following the instruction

Program and Instruction Costs(Cont…)

Some examples:

- The instruction **LD R0, R1** copies the contents of register R1 into register R0
- This instruction has a cost of one because no additional memory words are required

Program and Instruction Costs(Cont…)

Some examples:

- The instruction **LD R0, M** loads the contents of memory location M into register R0
- The cost is two since the address of memory location M is in the word following the instruction

Program and Instruction Costs(Cont…)

Some examples:

- The instruction **LD R1, *100(R2)** loads into register R1 the value given by **contents(contents(100+contents(R2)))**
- The cost is four because the constant 100 is stored in the word following the instruction

Program and Instruction Costs(Cont…)

- We assume the cost of a target-language program on a given input is the sum of costs of the individual instructions executed when the program is run on that input
- Good code-generation algorithms seek to minimize the sum of the costs of the instructions executed by the generated target program on typical inputs

Addresses in the Target Code

- We show how names in the IR can be converted into addresses in the target code by looking at code generation for simple procedure calls and returns using static and stack allocation
- Each executing program runs in its own logical address space that was partitioned into four code and data areas:
  - A statically determined area **Code** that holds the executable target code
  - The size of the target code can be determined at compile time

Addresses in the Target Code(Cont…)

- Each executing program runs in its own logical address space that was partitioned into four code and data areas:
  - A statically determined data area **Static** for holding global constants and other data generated by the compiler
  - The size of the target code can be determined at compile time

Addresses in the Target Code(Cont…)

- Each executing program runs in its own logical address space that was partitioned into four code and data areas:
    - A dynamically managed area **Heap** for holding data objects that are allocated and freed during program execution
    - The size of the Heap cannot be determined at compile time

Addresses in the Target Code(Cont…)

- Each executing program runs in its own logical address space that was partitioned into four code and data areas:
    - A dynamically managed area **Stack** for holding activation records as they are created and destroyed during procedure calls and returns
    - Like the Heap, the size of the Stack cannot be determined at compile time

Static Allocation

- To illustrate code generation for simplified procedure calls and returns we shall focus on the following three-address statements:
    - **call callee**
    - **return**
    - **halt**
    - **action**, which is a placeholder for other three-address statements

Static Allocation(Cont…)

- The size and layout of activation records are determined by the code generator via the information about names stored in the symbol table
- We shall first illustrate how to store the return address in an activation record on a procedure call and how to return control to it after the procedure call
- For convenience, we assume the first location in the activation holds the return address

Static Allocation(Cont…)

- Let us consider the code needed to implement the simplest case static allocation
- Here, a call callee statement in the intermediate code can be implemented by a sequence of two target-machine instructions:
    - **ST callee.staticArea, #here + 20**
    - **BR callee.codeArea**

Static Allocation(Cont…)

ST callee.staticArea, #here + 20

BR callee.codeArea

- The **ST** instruction saves the return address at the beginning of the activation record for callee and
- the **BR** transfers control to the target code for the called procedure callee

Static Allocation(Cont…)

ST callee.staticArea, #here + 20

BR callee.codeArea

- The attribute before **callee.staticArea** is a constant that gives the address of the beginning of the activation record for callee and
- the attribute **callee.codeArea** is a constant referring to the address of the first instruction of the called procedure callee in the Code area of the run-time memory

Static Allocation(Cont…)

ST callee.staticArea, #here + 20

BR callee.codeArea

- The operand **#here + 20** in the ST instruction is the literal return address
- It is the address of the instruction following the BR instruction
- We assume that **#here** is the address of the current instruction and that the three constants plus the two instructions in the calling sequence have a length of **5 words or 20 bytes**

Static Allocation(Cont…)

```
ST callee.staticArea, #here + 20
BR callee.codeArea
```

- The code for a procedure ends with a return to the calling procedure
- Except that the first procedure has no caller, so its final instruction is **HALT** which returns control to the operating system

Static Allocation(Cont…)

- A return callee statement can be implemented by a simple jump instruction **BR *callee.staticArea** which transfers control to the address saved at the beginning of the activation record for callee

Example 8.3

Suppose we have the following three-address code:

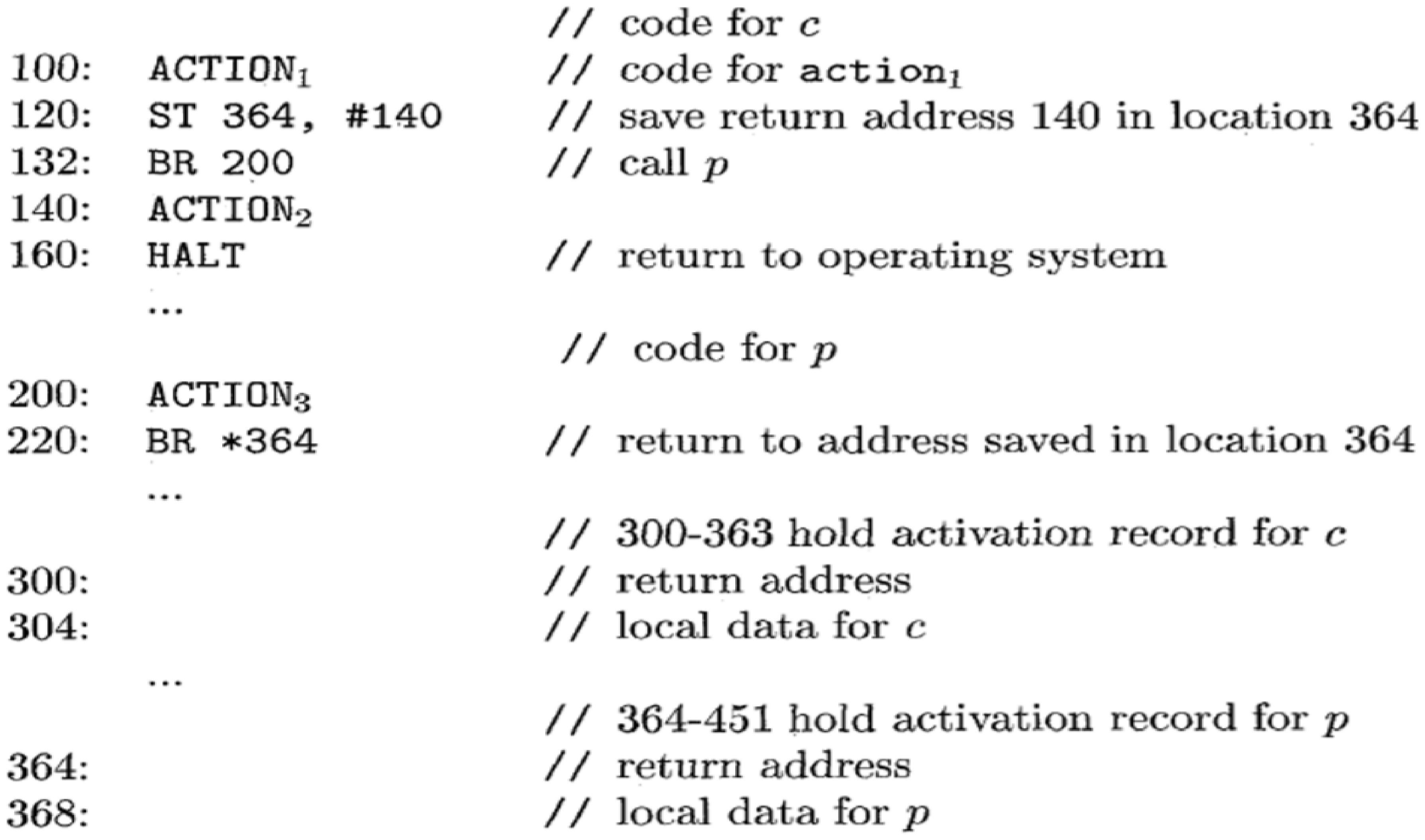
```
// code for c
     action1
     call p
     action2
     halt
// code for p
     action3
      return
```

Example 8.3(Cont…)

- We use the pseudo-instruction **ACTION** to represent the sequence of machine instructions to execute the statement action
- This represents three-address code that is not relevant for this discussion
- We arbitrarily start the **code** for **procedure c** at address **100** and for **procedure p** at address **200**
- We assume that each **ACTION** instruction takes **20 bytes**
- We further assume that the **activation records** for these procedures are **statically allocated** starting at locations **300** and **364** respectively

Example 8.3(Cont…)

```
// code for c
    action1
    call p
    action2
    halt
// code for p
    action3
     return
```



```
                                  // code for c
100:   ACTION_1                   // code for action_1
120:   ST 364, #140               // save return address 140 in location 364
132:   BR 200                     // call p
140:   ACTION_2
160:   HALT                       // return to operating system
       ...
                                  // code for p
200:   ACTION_3
220:   BR *364                    // return to address saved in location 364
       ...
                                  // 300-363 hold activation record for c
300:                              // return address
304:                              // local data for c
       ...
                                  // 364-451 hold activation record for p
364:                              // return address
368:                              // local data for p
```

Target code for static allocation

Thank
you