

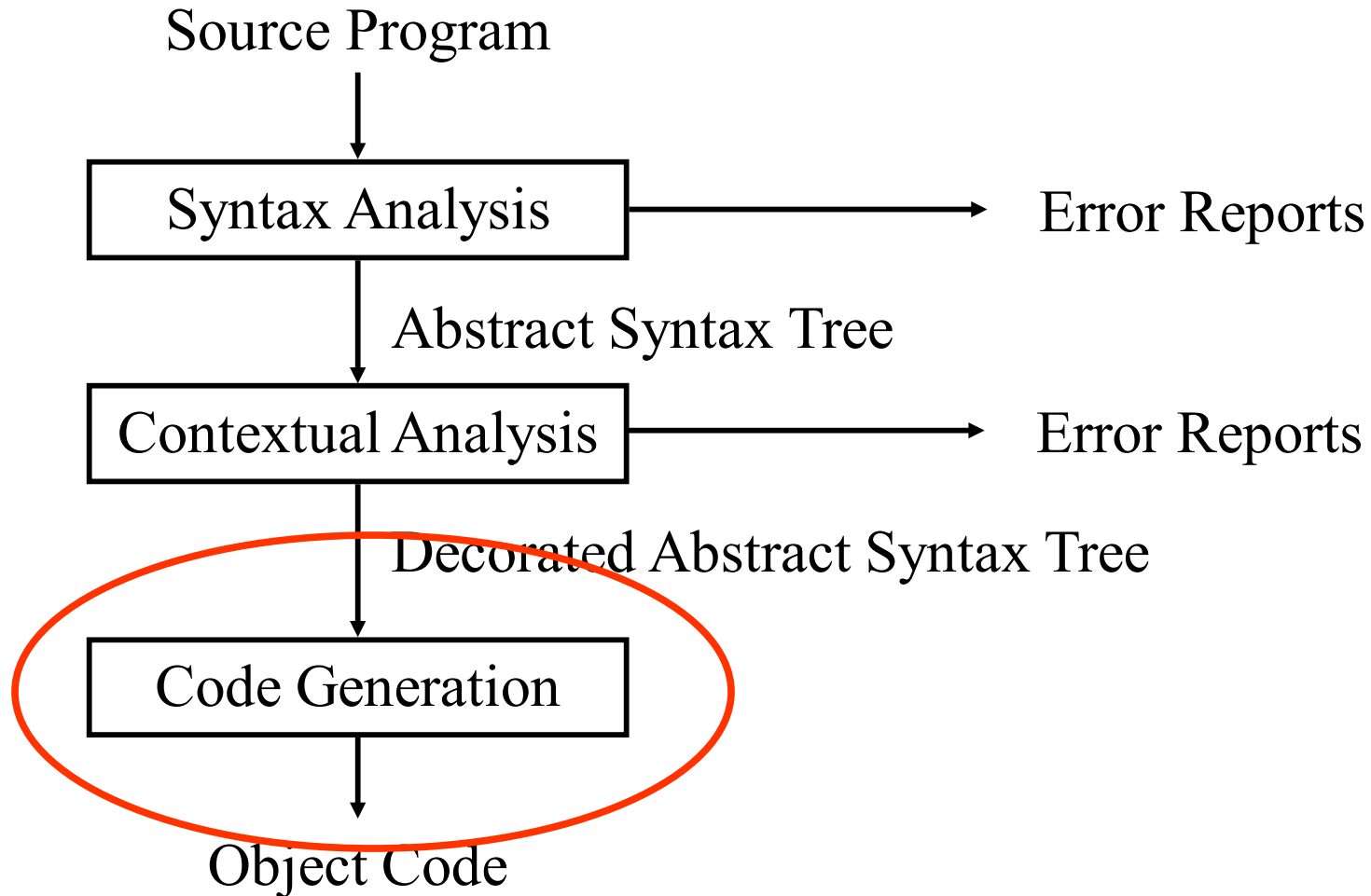
Languages and Compilers **(SProg og Oversættelse)**

Code Generation

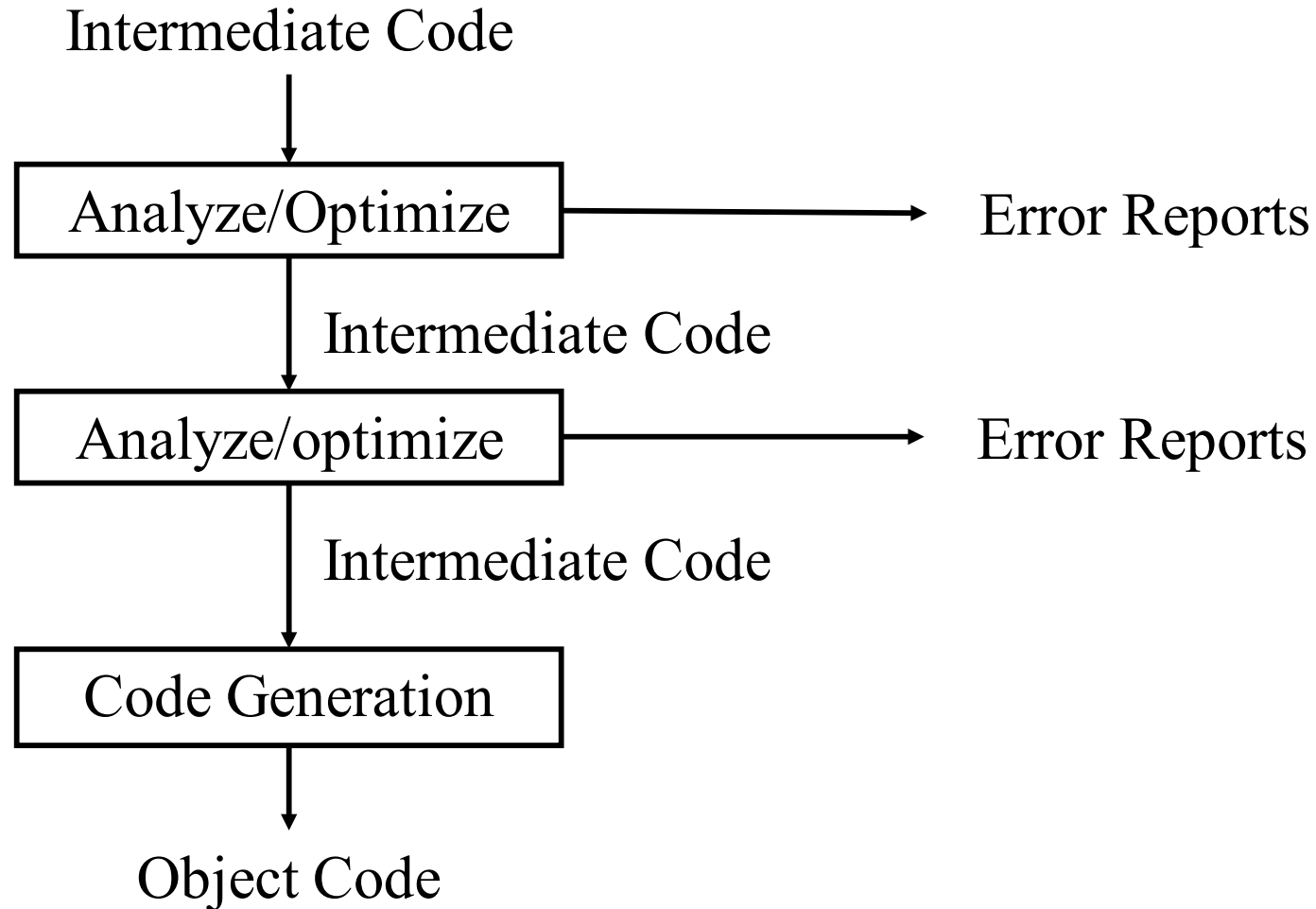
Code Generation

- a. Describe the purpose of the code generator
- b. Discuss Intermediate representations
- c. Describe issues in code generation
- d. Code templates and implementations
- e. Back patching
- f. Implementation of functions/procedures/methods
- g. Register Allocation and Code Scheduling
- h. Optimizations

The “Phases” of a Compiler



The “Phases” of a Compiler



Intermediate Representations

- Abstract Syntax Tree
 - Convenient for semantic analysis phases
 - We can generate code directly from the AST, but...
 - What about multiple target architectures?
- Intermediate Representation
 - "Neutral" architecture
 - Easy to translate to native code
 - Can abstracts away complicated runtime issues
 - Stack Frame Management
 - Memory Management
 - Register Allocation

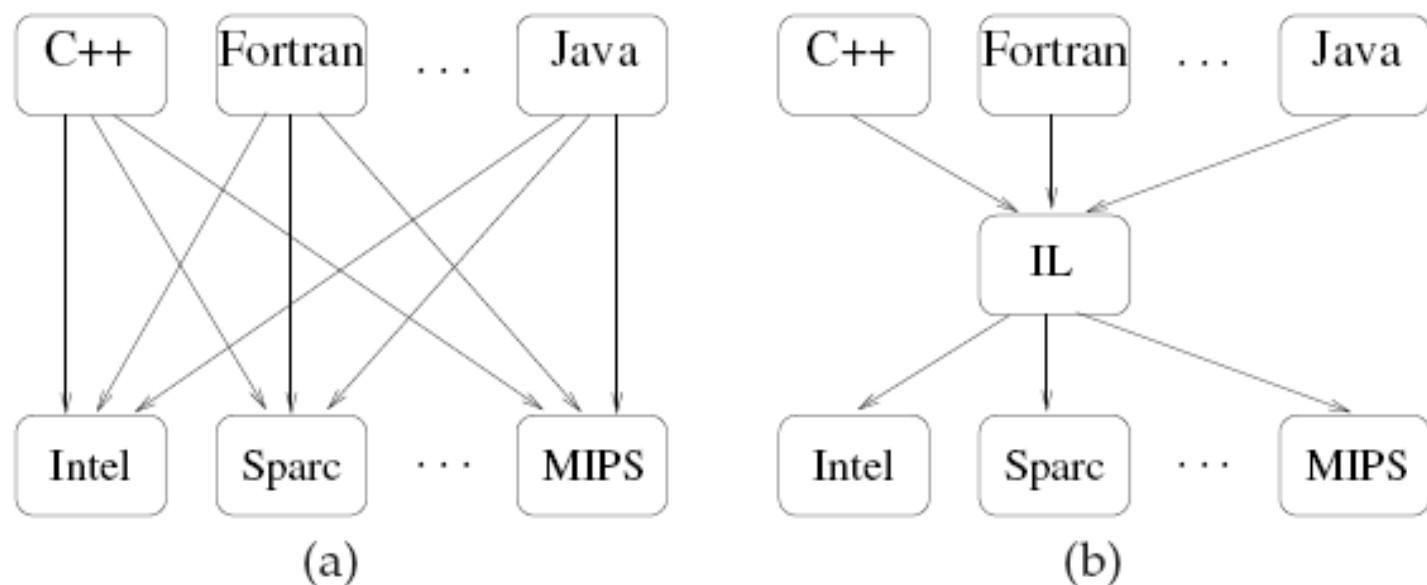


Figure 10.3: A middle-end and its ILs simplify construction of a compiler suite that must support multiple source languages and multiple target architectures.

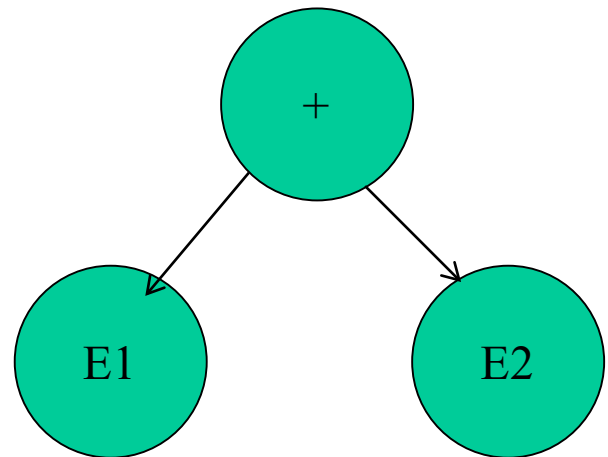
Issues in Code Generation

- **Code Selection:**
Deciding which sequence of target machine instructions will be used to implement each phrase in the source language.
- **Storage Allocation**
Deciding the storage address for each variable in the source program. (static allocation, stack allocation etc.)
- **Register Allocation (for register-based machines)**
How to use registers efficiently to store intermediate results.
- **Code Scheduling**
The order in which the generated instructions are executed

Code Emmission

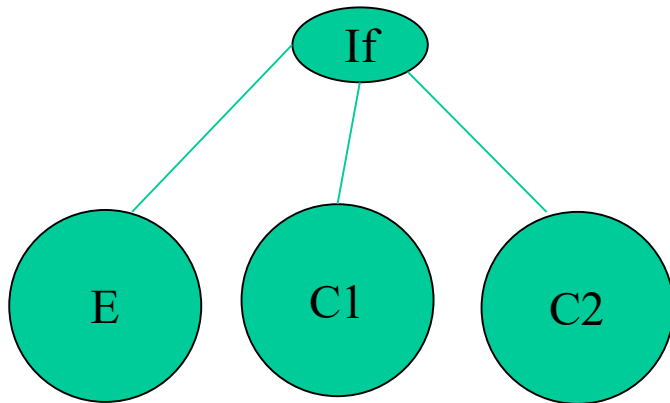
- Generating the actual instructions is usually called emission
 - a CodeGenVisitor emits instructions
- Example:
 - MethodBodyVisitor.visit(Plus)
 - visit(E1)
 - visit(E2)
 - emit("iadd\n")

```
/* Visitor code for Marker ⑦  
procedure VISIT( Computing n )  
  VISITCHILDREN( n )  
  loc ← ALLOCLocal()  
  n.SETRESULTLOCAL( loc )  
  call EMITOPERATION( n )  
end
```



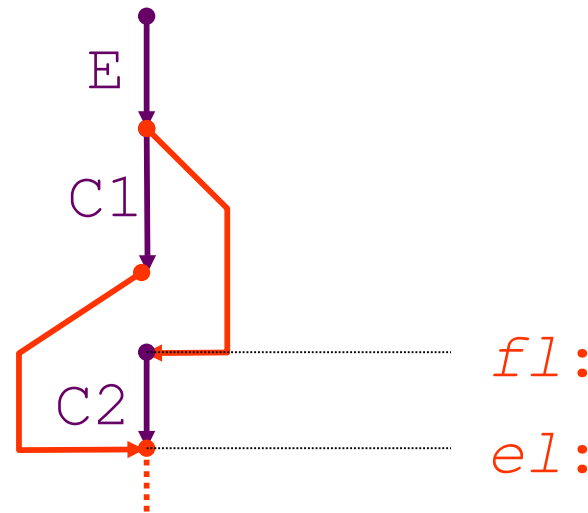
Code Templates

$visit[\text{if } E \text{ then } C1 \text{ else } C2] =$
 $visit[E]$
 JUMPIFFALSE fl
 $visit[C1]$
 JUMP el
 $fl:$ $visit[C2]$
 $el:$



```

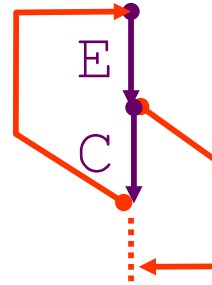
/* Visitor code for Marker 12
procedure VISIT(CondTesting n)
  falseLabel ← GENLABEL()
  endLabel ← GENLABEL()
  call n.GETPREDICATE().ACCEPT(this)
  predicateResult ← n.GETPREDICATE().GETRESULTLOCAL()
  call EMITBRANCHIFFALSE(predicateResult, falseLabel)
  call n.GETTRUEBRANCH().ACCEPT(this)
  call EMITBRANCH(endLabel)
  call EMITLABELDEF(falseLabel)
  call n.GETFALSEBRANCH().ACCEPT(this)
  call EMITLABELDEF(endLabel)
end
    
```



Code Templates

While Command:

$visit$ [**while** E **do** C] =
l: $visit$ [E]
 JUMPIFFALSE *d*
 $visit$ [C]
 JUMP *l*
d:

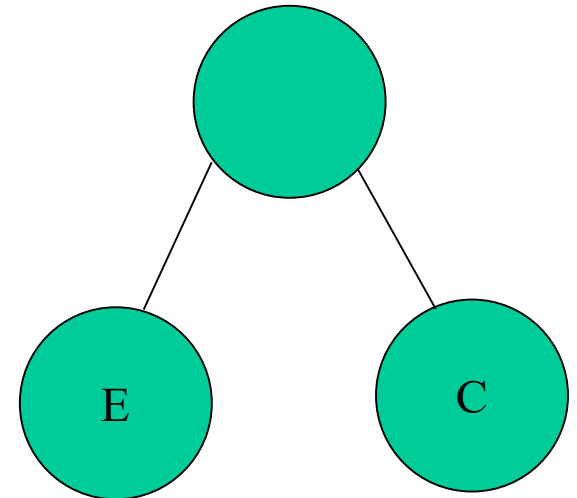
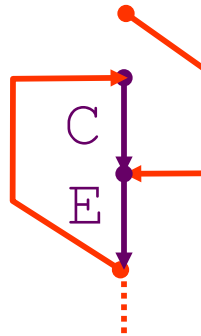


```

/* Visitor code for Marker ⑬
procedure visit( WhileTesting n)
  doneLabel ← GENLABEL()
  loopLabel ← GENLABEL()
  call EMITLABELDEF(loopLabel)
  n.GETPREDICATE().ACCEPT(this)
  predicateResult ← n.GETPREDICATE().GETRESULTLOCAL()
  call EMITBRANCHIFFALSE(predicateResult, doneLabel)
  n.GETLOOPBODY().ACCEPT(this)
  call EMITBRANCH(loopLabel)
  call EMITLABELDEF(doneLabel)
end
  
```

Alternative While Command code template:

$visit$ [**while** E **do** C] =
 JUMP *h*
l: $visit$ [C]
h: $visit$ [E]
 JUMPIFTRUE *l*



Backpatching Example

```
public Object WhileCommand (
    WhileCommand com, Object arg) {
    short j = nextInstrAddr;
    emit (Instruction.JUMPop, 0,
        Instruction.CBr, 0); dummy address
    short g = nextInstrAddr;
    com.C.visit(this, arg);
    short h = nextInstrAddr;
    code[j].d = h; backpatch
    com.E.visit(this, arg);
    emit (Instruction.JUMPIFop, 1,
        Instruction.CBr, g);
    return null;
}
```

execute [**while** E **do** C] =
 JUMP *h*
g: *execute* [C]
h: *evaluate*[E]
 JUMPIF (1) *g*

Code Template: Global Procedure

elaborate [**proc** I () $\sim C$] =

JUMP g

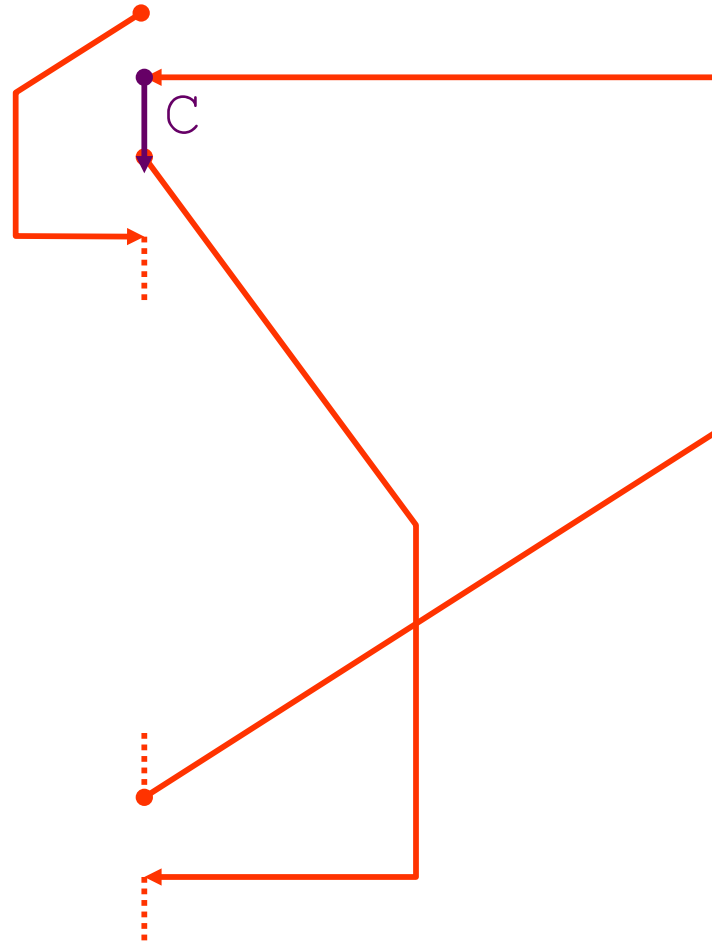
e : *execute* [C]

RETURN (0) 0

g :

execute [I ()] =

CALL (SB) e



```

subi    $sp,$sp,frameSz      # Push frame on stack
sw      $ra,0($sp)           # Save return address in frame
sw      $fp,4($sp)           # Save old frame pointer in frame
move    $fp,$sp              # Set $fp to access new frame
# Save callee-save registers (if any) here
# Body of method is here
# Restore callee-save registers (if any) here
lw      $ra,0($fp)           # Reload return address register
lw      $fp,4($fp)           # Reload old frame pointer
addi    $sp,$sp,frameSz      # Pop frame from stack
jr      $ra                  # Jump to return address

```

Figure 13.6: MIPS prologue and epilogue code

Register Allocation

- A compiler generating code for a register machine needs to pay attention to register allocation as this is a limited resource
- In routine protocol
 - Allocate arg1 in R1, arg2 in R2 .. Result in R0
 - But what if there are more args than regs?
- In evaluation of expressions
 - On MIPS all calculations take place in regs
 - Reduce traffic between memory and regs

Code scheduling

- Modern computers are pipelined
 - Instructions are processed in stages
 - Instructions take different time to execute
 - If result from previous instruction is needed but not yet ready then we have a **stalled pipeline**
 - Delayed load
 - Load from memory takes 2, 10 or 100 cycles
 - Also FP instructions takes time

Reg allocation and Code Scheduling

- Reg allocations algorithms try to minimize the number of regs used
- May conflict with pipeline architecture
 - Using more regs than strictly necessary may avoid pipeline stalls
- Solution
 - Integrated register allocator and code scheduler

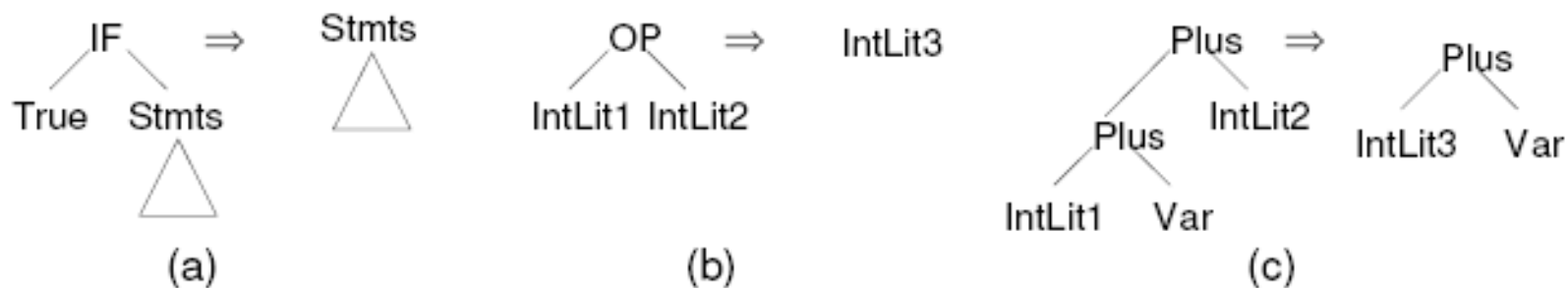


Figure 13.31: AST-Level Peephole Optimization

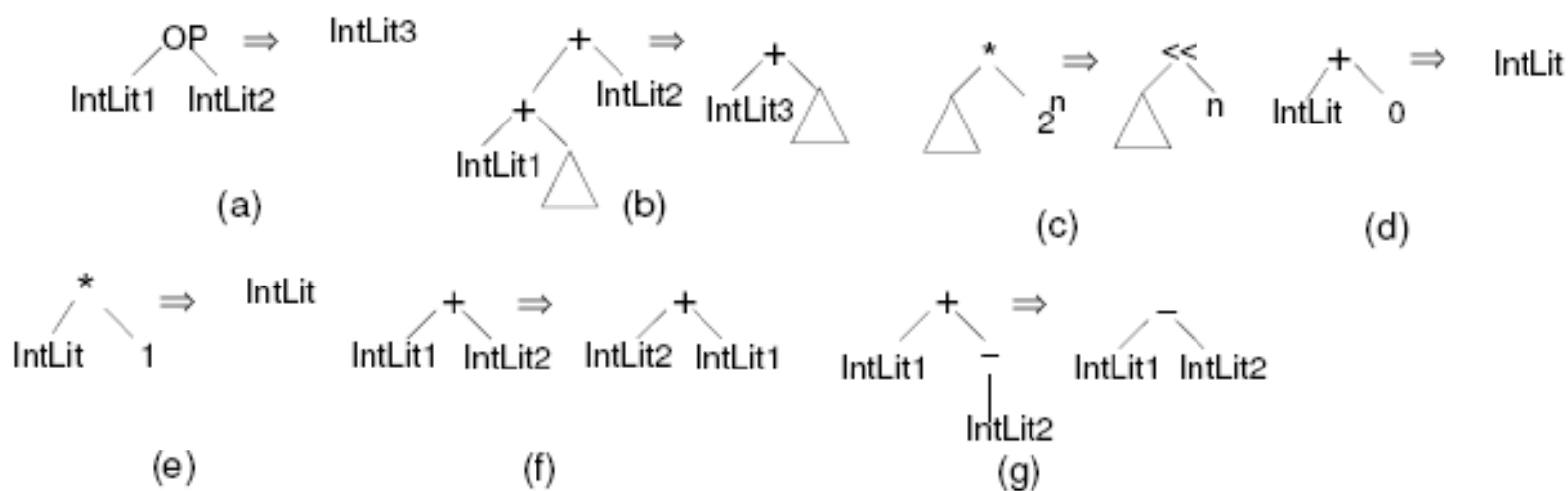


Figure 13.32: IR-Level Peephole Optimizations

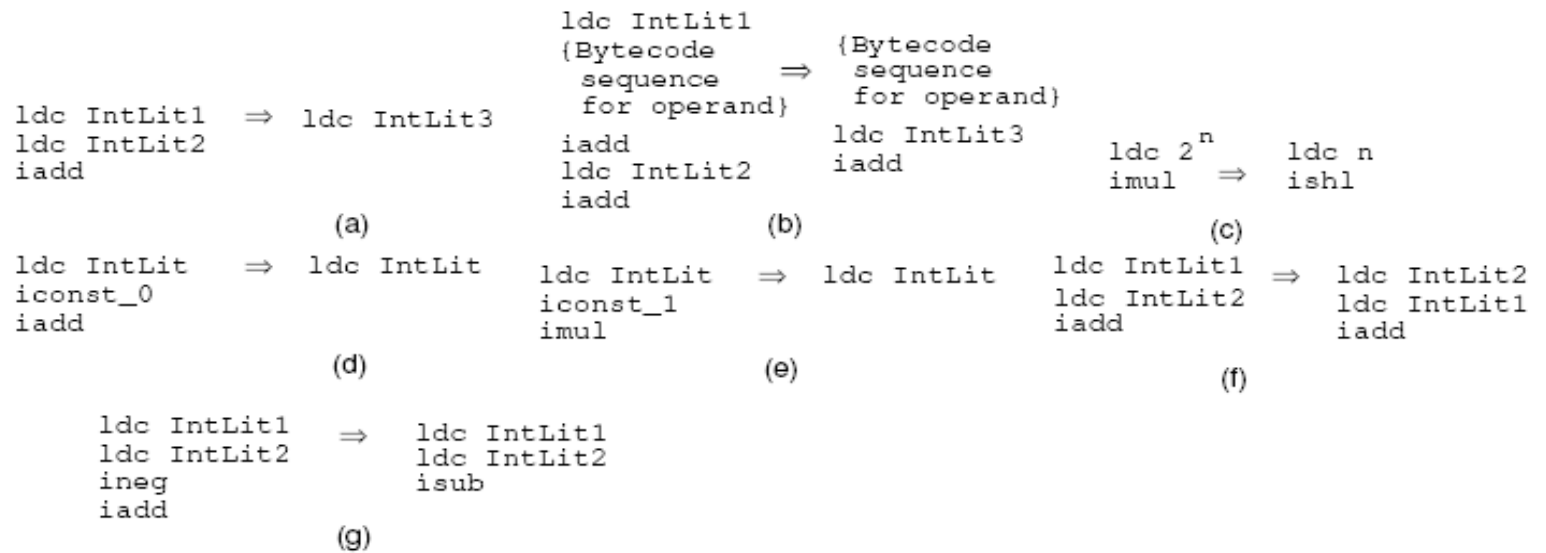


Figure 13.33: Bytecode-Level Peephole Optimizations

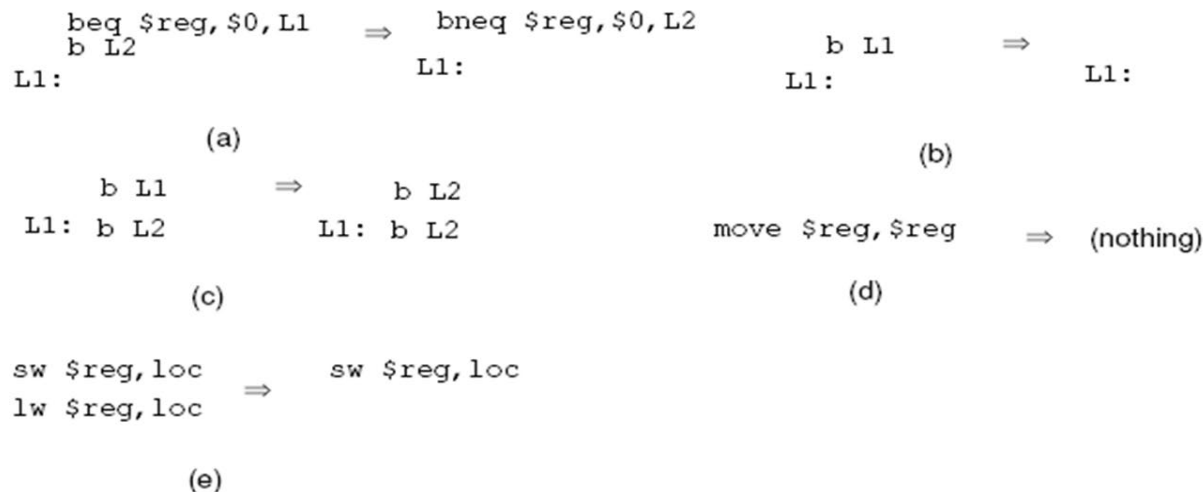


Figure 13.34: Code-Level Peephole Optimizations