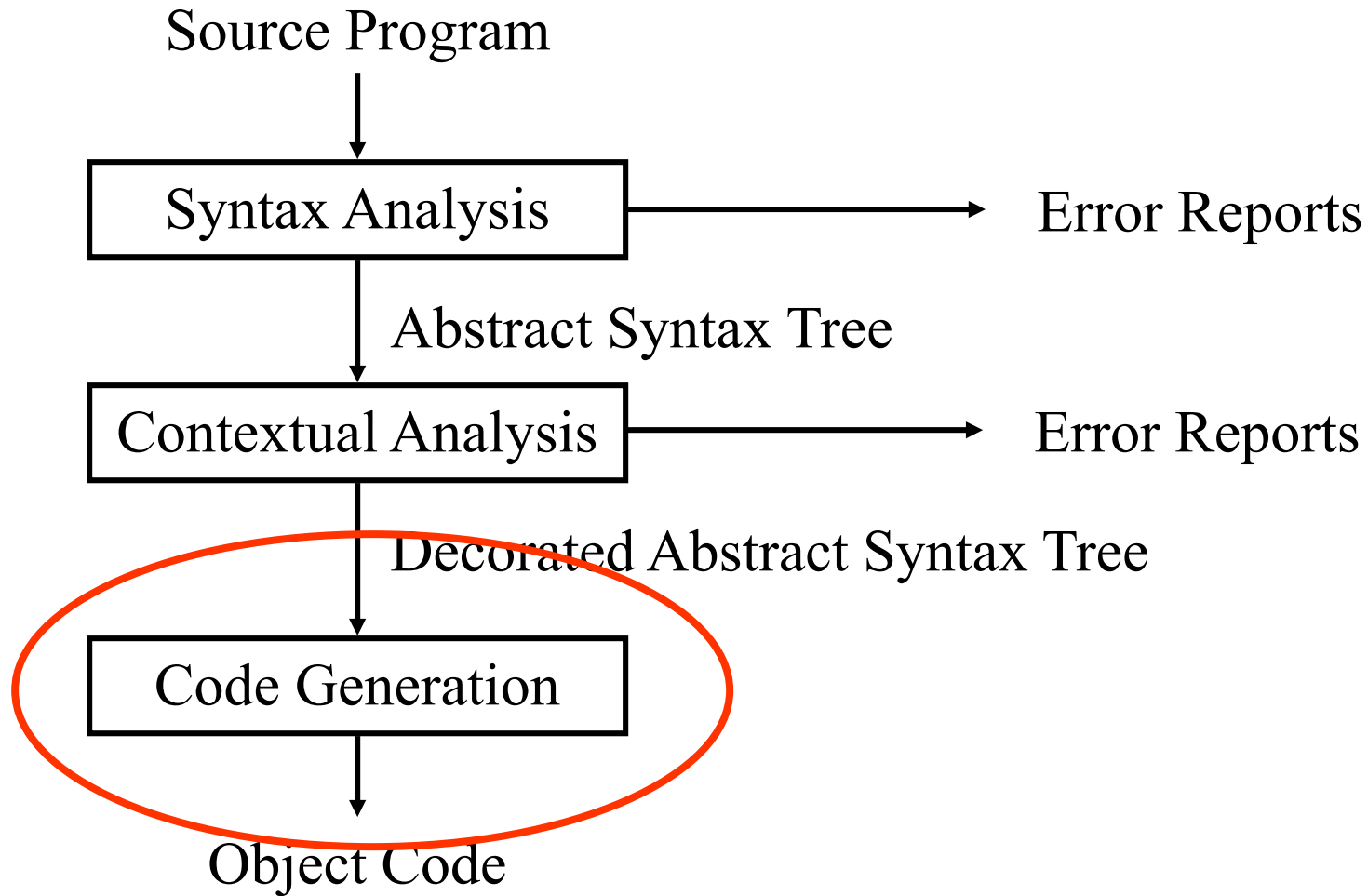# Languages and Compilers
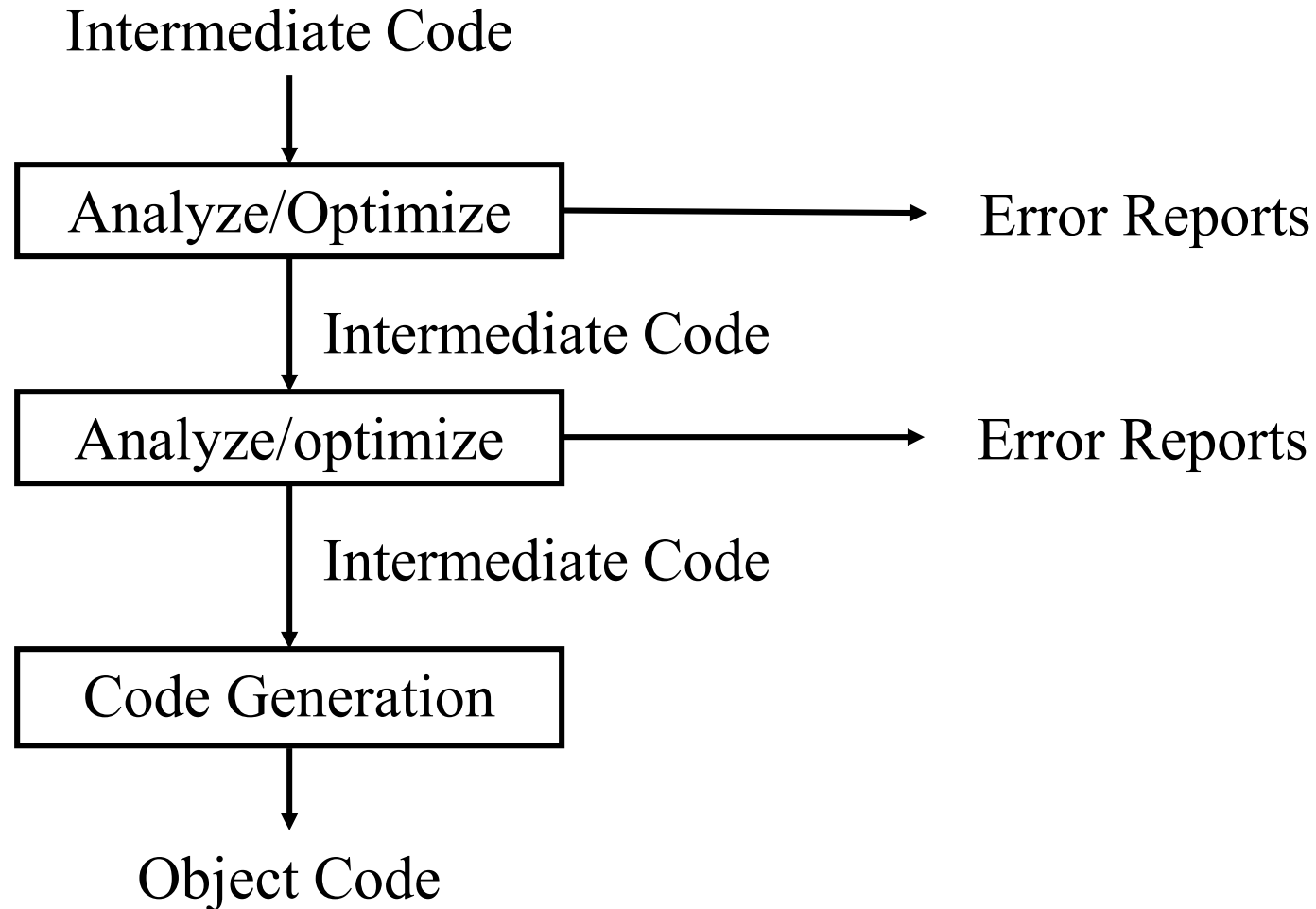# (SProg og Oversættere)

Code Generation

Code Generation

– Describe the purpose of the code generator

– Discuss Intermediate representations

– Describe issues in code generation

– Code templates and implementations

– Back patching

– Implementation of functions/procedures/methods

– Register Allocation and Code Scheduling

The "Phases" of a Compiler

Source Program

↓

| Syntax Analysis | → Error Reports |

↓ Abstract Syntax Tree

| Contextual Analysis | → Error Reports |

↓ Decorated Abstract Syntax Tree

| Code Generation |

↓

Object Code

3

The "Phases" of a Compiler

Intermediate Code

↓

| Analyze/Optimize | → Error Reports

↓

Intermediate Code

↓

| Analyze/optimize | → Error Reports

↓

Intermediate Code

↓

| Code Generation |

↓

Object Code

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
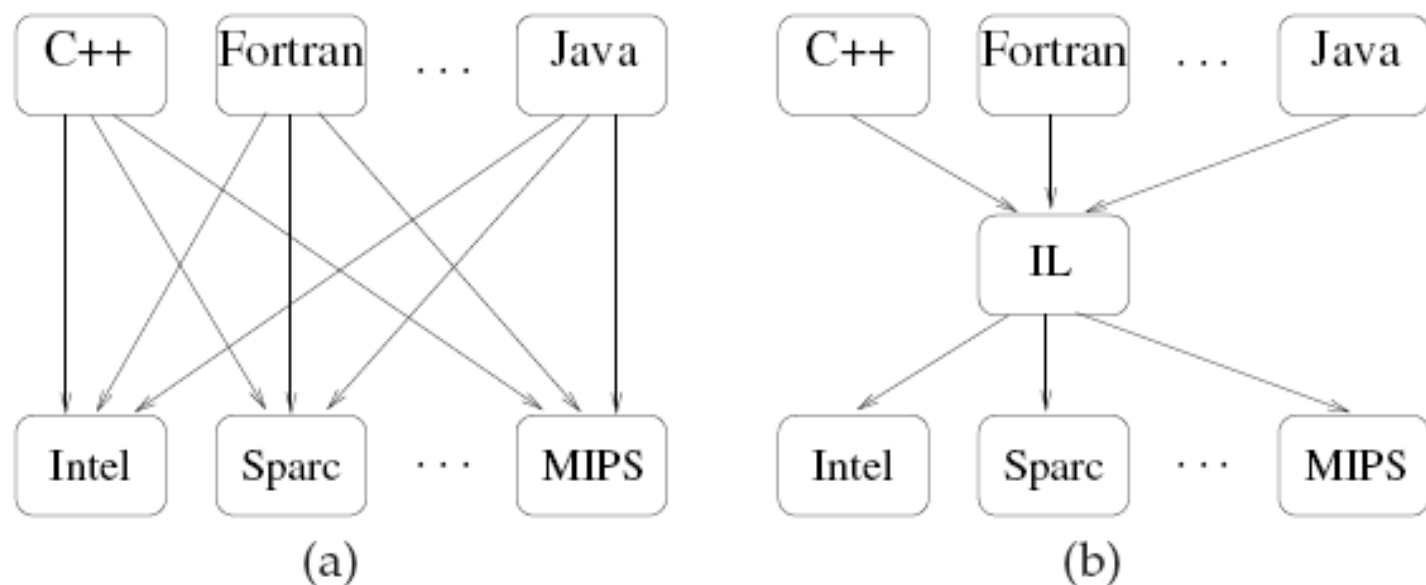   - Memory Management
    - Register Allocation

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 Emmision

- 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
```
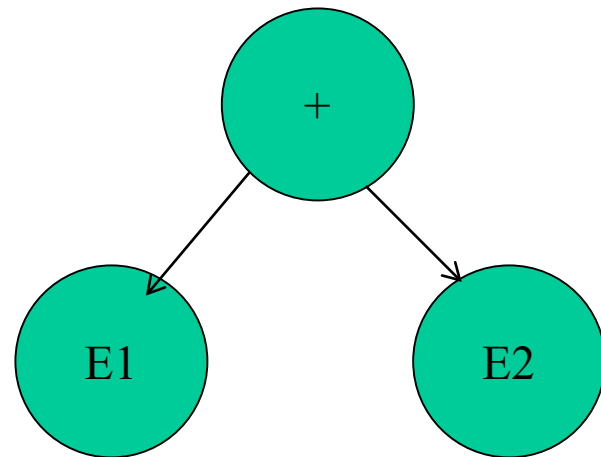
Code Templates

$visit\,[\textbf{if}\ \ \text{E}\ \ \textbf{then}\ \text{C1}\ \textbf{else}\ \text{C2}] =$

      $visit\,[\text{E}]$

      JUMPIFFALSE g

      $visit\,[\text{C1}]$

      JUMP h

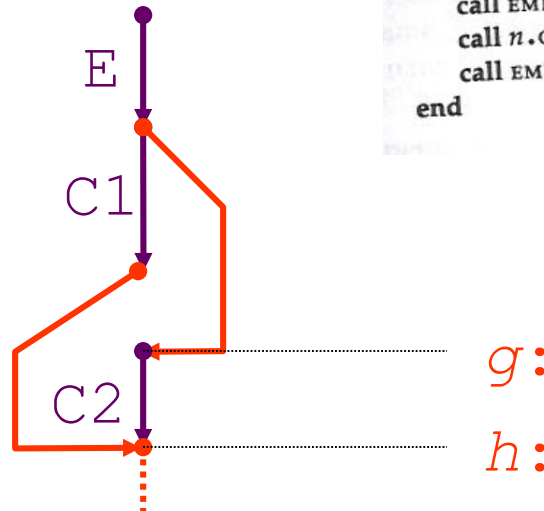g: $visit\,[\text{C2}]$
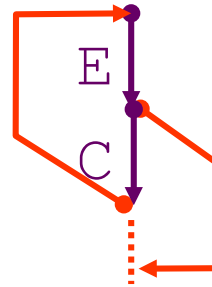
h:

```
/*    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
```

E

C1

g:

C2

h:

Code Templates

While Command:

$visit\,[\textbf{while}\ \text{E}\ \textbf{do}\ \text{C}] =$

    $l:\ visit\,[\text{E}]$

       `JUMPIFFALSE` d

     $visit[\text{C}]$

     `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:

$visit\,[\textbf{while}\ \text{E}\ \textbf{do}\ \text{C}] =$

     `JUMP` h

   $l:\ visit\,[\text{C}]$

   $h:\ visit[\text{E}]$

     `JUMPIFTRUE` l
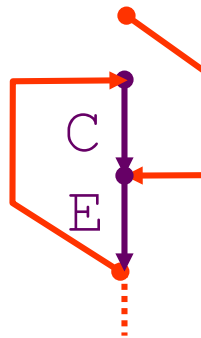
Backpatching Example

```
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 **()** **~** C] =
      JUMP *g*
   e: *execute* [C]
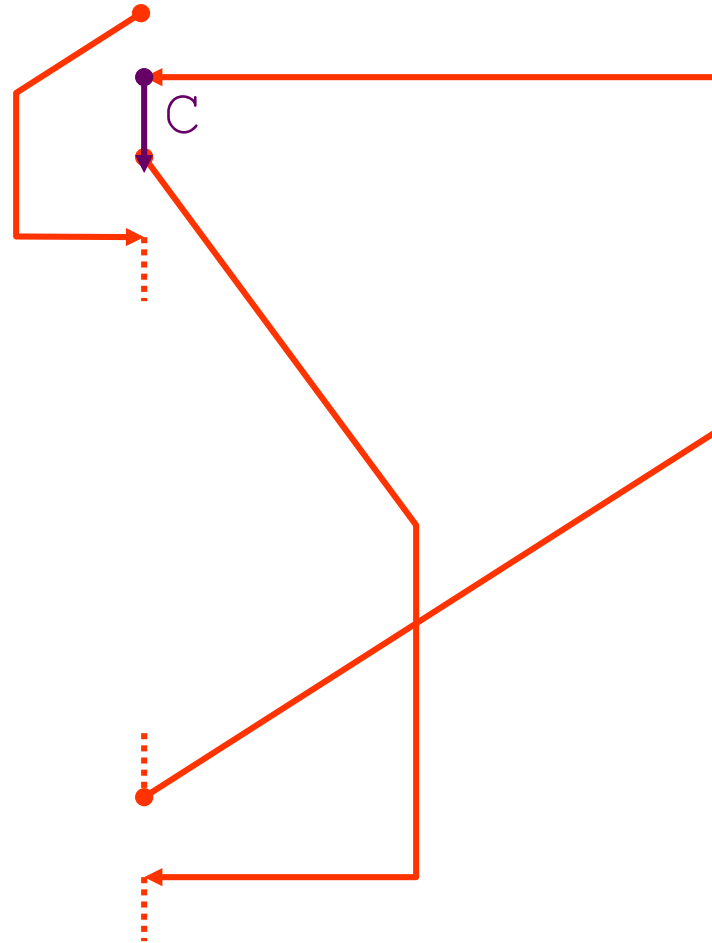   RETURN(0) 0
   g:

                 C

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 ressource

- 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 Scheluling

- 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
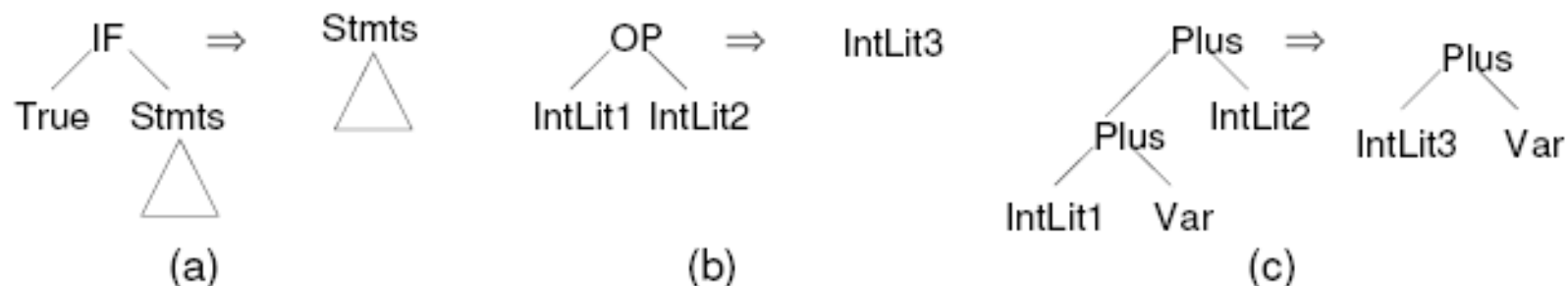 - Integrated register allocator and code scheduler
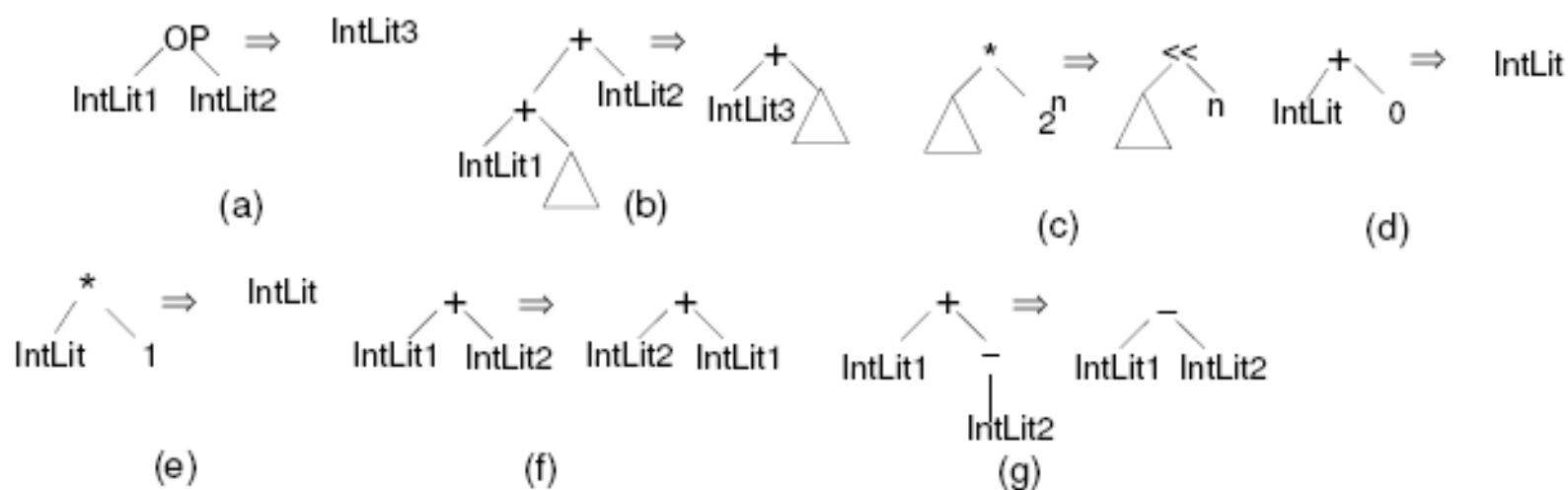
Figure 13.31: AST-Level Peephole Optimization

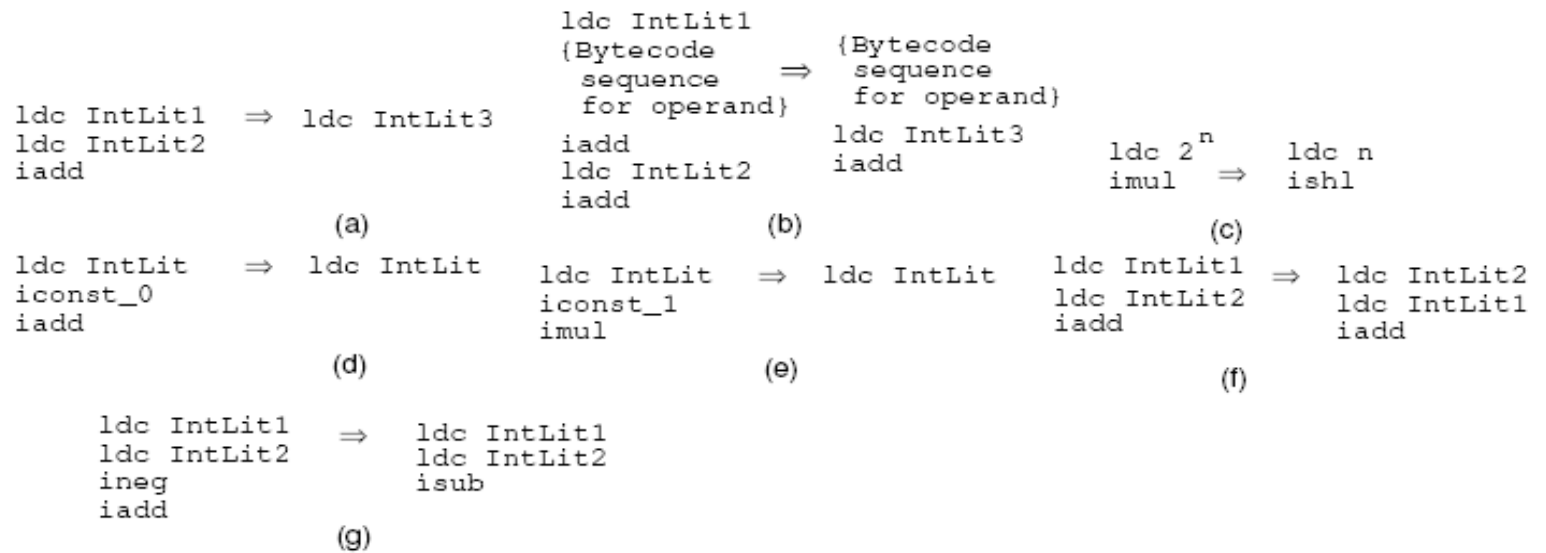Figure 13.32: IR-Level Peephole Optimizations

```
ldc IntLit1     ⇒  ldc IntLit3
ldc IntLit2
iadd
```
(a)

```
ldc IntLit1
{Bytecode            {Bytecode
  sequence      ⇒     sequence
  for operand}         for operand}
iadd                 ldc IntLit3
ldc IntLit2          iadd
iadd
```
(b)

```
ldc 2^n   ⇒  ldc n
imul         ishl
```
(c)

```
ldc IntLit    ⇒  ldc IntLit
iconst_0
iadd
```
(d)

```
ldc IntLit    ⇒  ldc IntLit
iconst_1
imul
```
(e)

```
ldc IntLit1    ⇒  ldc IntLit2
ldc IntLit2       ldc IntLit1
iadd              iadd
```
(f)

```
ldc IntLit1    ⇒  ldc IntLit1
ldc IntLit2       ldc IntLit2
ineg              isub
iadd
```
(g)

Figure 13.33: Bytecode-Level Peephole Optimizations

```
beq $reg,$0,L1  ⇒  bneq $reg,$0,L2
   b L2
L1:                L1:
```
(a)

```
  b L1    ⇒
L1:            L1:
```
(b)

```
  b L1      ⇒    b L2
L1: b L2       L1: b L2
```
(c)

```
move $reg,$reg  ⇒  (nothing)
```
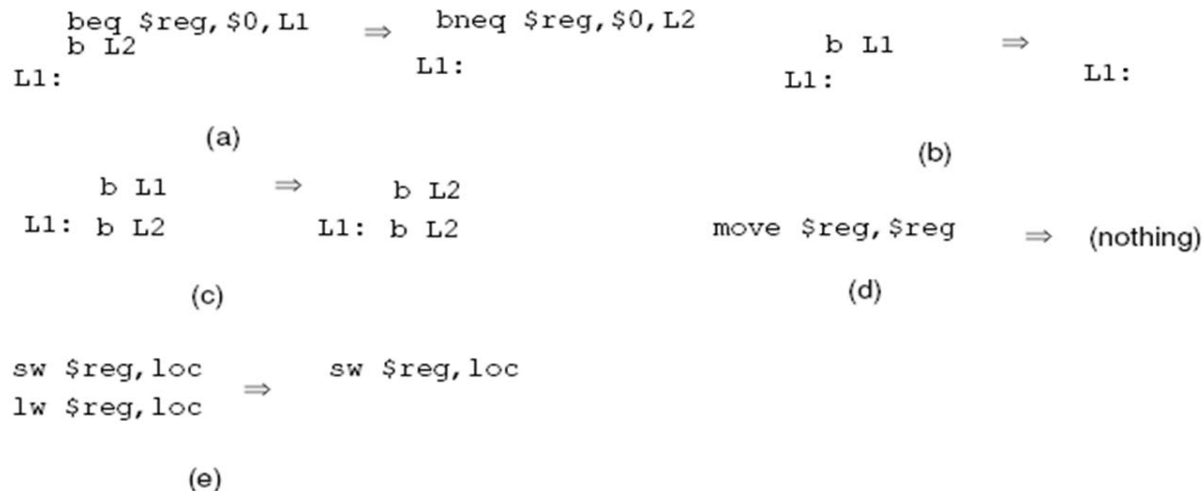(d)

```
sw $reg,loc       sw $reg,loc
lw $reg,loc   ⇒
```
(e)

Figure 13.34: Code-Level Peephole Optimizations