Important Errata and Additions for for the First and Second Printings of Advanced Compiler Design and Implementation

Steven S. Muchnick

30 June 1998

Chapter 3. Symbol Table Structure

P. 53, Figure 3.8(a): Change the pointer from the top entry on the block stack to point to the top entry on the symbol stack.

P. 54: In line 3, change "reordered" to "extended"; and, in line 7, change "entries above" to "entries at or above".

P. 55: In line -3, change "`28`" to "`56`", and, in lines -5 in Figure 3.10(b) and line 2 of Figure 3.10(c), change "`24`" to "`60`".

P. 57: In lines -7 and -6 of the text, change "`28`" to "`56`".

P. 59: Add item 8 to the numbered list as follows:

8. `Find_Opcode(`*stype*`)` returns the index in the array `LdStType` of the entry with type *stype*.

P. 61, Figure 3.14, lines 8, 16, 19, and 20: Change "`reg`" to "`symreg`".

P. 62, Figure 3.15, line 23: Change "`type`" to "`symtype`" and "`symtype`" to "`type`".

P. 63, Figure 3.15*(continued)*: In line 1, change "`disp`" to "`symdisp`" and replace lines -9 through -3 by

```
if Short_Const(symdisp) & stflag then
   ||  generate store
   Gen_Inst(Opcode,regc ⊕ ",[" ⊕ reg1 ⊕ "+" ⊕ Int_Char(symdisp) ⊕ "]"); return
elif Short_Const(symdisp) then
   ||  generate load
   Gen_Inst(Opcode,"[" ⊕ reg1 ⊕ "+" ⊕ Int_Char(symdisp) ⊕ "]," ⊕ regc); return
fi
   ||  generate sethi
Gen_Inst("sethi","%hi(" ⊕ Int_Char(symdisp) ⊕ ")," ⊕ reg1)
if stflag then
   ||  generate store
   Gen_Inst(Opcode,regc ⊕ ",[" ⊕ reg1 ⊕ ⊕ "+%lo(" ⊕ Int_Char(symdisp) ⊕ ")]")
else
   ||  generate load
   Gen_Inst(Opcode,"[" ⊕ reg1 ⊕ ⊕ "+%lo(" ⊕ Int_Char(symdisp) ⊕ ")]," ⊕ regc)
fi
```

Chapter 4. Intermediate Representations

P. 81: In line 14, after "`IROper`" insert " and its synonym "`Operator`"; and, in line 15, after "`=`" insert "`Operator =`".

Chapter 5. Run-Time Support

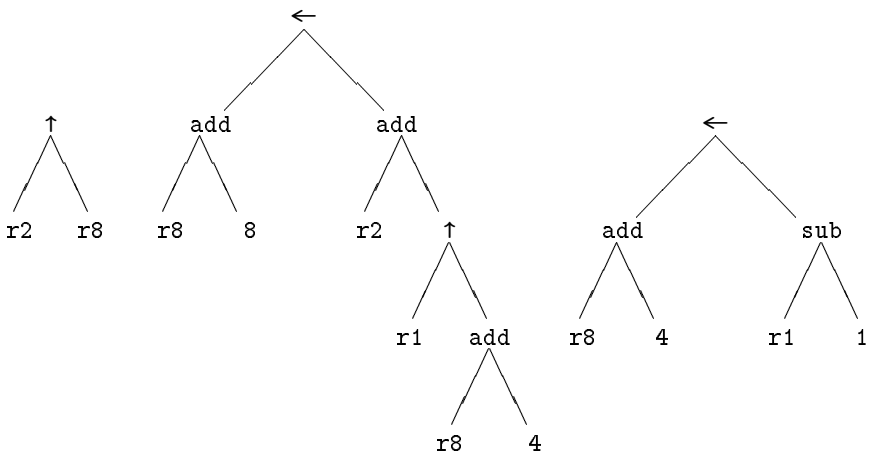
P. 129, in the first block of code, line 1: Change "`GOT_off + 8`" to "`GOT_off - 4`".

P. 135: In lines 1 and 2 of Exercise 5.1, change "MIR" to "LIR"; and in line 2, insert "byte" before "address". Also, delete Exercise 5.2 and renumber the following ones as 5.2 through 5.12.

Chapter 6. Producing Code Generators Automatically

P. 140, line 1 of the text: Change "`mov r1 ↑`" to "`↑ r1`".

P. 141: Replace the content of Figure 6.3 by the following:



and replace the content of Figure 6.4 by the following:

```
↑ r2 r8 ← + r8 8 + r2 ↑ r1 + r8 4 ← + r8 4 - r1 1
 ϵ                                          ld    [r8,0],r2
← + r8 8 + r2 ↑ r1 + r8 4 ← + r8 4 - r1 1
               r1                           ld    [r8,4],r1
← + r8 8 + r2 r1 ← + r8 4 - r1 1
          r3                                add   r2,r1,r3
← + r8 4 r3 ← + r8 4 - r1 1
 ϵ                                          st    r3,[r8,4]
← + r8 4 - r1 1
          r4                                sub   r1,1,r4
← + r8 4 r4
 ϵ                                          st    r4,[r8,4]
```

(a) (b)

P. 144, line -15: Change "`MGrammar`" to "`MGrammar & rule.lt = ϵ})`" and move the text left 21 spaces.

P. 145, Fig. 6.6 *(continued)*, lines -10 through -8: Replace these three lines by

```
& (item.lt = 'ϵ' ∨ (v ∈ Follow(item.lt)
& ∀item1 ∈ ItemSet[s] (item1 = item ∨ |item1.rt| ≤ |item.rt|
   ∨ item1.pos ≤ |item1.rt| ∨ v ∉ Follow(item1.lt))))}
```

P. 147: In lines 11, 12, and 15, change "*DottedPrefix*" to "*Prefix*"; in line 11, replace "$Nonterminal \cup \{\text{'}\epsilon\text{'}\}$" by *Vocab*; in lines 12 and 13, delete "each with a dot inserted in its right-hand side"; and in line 16, change "left" to "leftmost".

P. 149, Figure 6.9: Delete all the *First* and *Last* entries that have `'ϵ'` as an operand.

P. 153, Figure 6.13: In line 4, change "`CMG`" to "`C, MG`"; and, in line -6, change "`r.lt`" to "`r1.lt`".

P. 154: Change lines 5 through 17 of Figure 6.14 to

```
R := {r}, oldR:  set of Nonterminal
|| determine the set of nonterminals reachable from r
|| without generating any terminal symbols
repeat
    oldR := R
    for each r1 ∈ MGrammar do
        if r1.lt ∈ R & |r1.rt| = 1 & r1.rt↓1 ∈ Nonterminal & r1.rt↓1 ∉ R then
            R ∪= {r1.rt}
        fi
    od
until R = oldR
```

In line -7, change "symbol x" to "symbol x and returns a set containing only the resulting rule, except that if the resulting rule has identical left- and right-hand sides, it returns $\emptyset$".

P. 166: In Exercise 6.1, line 2, change "6.1" to "6.8"; and, in Exercise 6.2, line 2, change "constant" to "register".

P. 167, line 7: Change the first occurrence of "`fr.1`" to "`↑ r.1`" and its second occurrence to "`[r.1]`".

Chapter 7. Control-Flow Analysis

P. 176, Figure 7.6: Delete lines -16, -13, -12, -8, -5, and -2.

P. 178, line 2: Change "siblings" to "siblings that are not also its descendants".

P. 181: Move line 6 ("`i := 2:  integer`") to precede line 1 on the page.

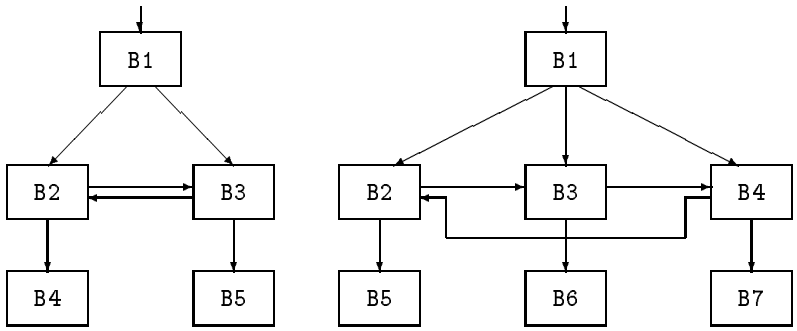
P. 192, Figure 7.21: In line 10, change "`=`" to "`≠`"; and change line 11 to "`Stack ⊕= [m]`".

P. 195, Figure 7.26: Before line 1, insert the line "`N: set of Node`"; in line 7, insert "`in `" after "`Succ: `"; in line 16, insert "`,Succ`" after "`y`"; in line 17, insert "`& ∃i ∈ integer (y = Stack↓i)`" after "`Dfn(x)`"; and change lines -12 through -9 to the following:

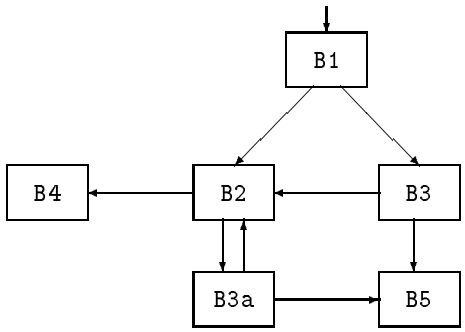
```
    Dfn(x) := LowLink(x) := 0
od
NextDfn := 0; Stack := []
```

and insert the line "`All_SCC ∪= {{Stack↓1}}`" following line -2.

P. 197, Figure 7.28: Change the contents of Figure 7.27 to the following:



and change the contents of Figure 7.28 to the following:



See the document "Errata for Structural Analysis in Advanced Compiler Design and Implementation" for replacement text for most of Section 7.7. Structural Analysis.

Chapter 8. Data-Flow Analysis

P. 224, line 3 of the text: Change the first occurrence of "$(x \sqcap z) \sqcup (y \sqcap z)$" to "$(x \sqcup z) \sqcap (y \sqcup z)$".

P. 226, line -17: Change "$\langle 11x_3x_4x_5x_6x_7x_8 \rangle$" to "$\langle 111x_4x_500x_8 \rangle$" (as shown in Table 8.2 on p. 233).

P. 232: In Figure 8.6, in line 1, change "`FP`" to "`F`"; in line 4, change "`FP: in Node`" to "`F: in Node × L`"; in line -5, change "`{B}`" to "`Succ(B)`"; and move lines -7 through -4 to follow line -3, so that lines -8 through -3 read as follows:

```
      totaleffect ⊓= effect
od
if dfin(B) ≠ totaleffect then
   dfin(B) := totaleffect
   Worklist ∪= Succ(B)
fi
```

Also, in line -7 of the text, change "whose" to "whose predecessors' ".

P. 233–234: Replace the last 10 lines on p. 233 and the first 21 on p. 234 by the following:

> Entering the `repeat` loop, the initial value of `B` is `B1`, with the worklist becoming `{B2,B3, B4,B5,B6,exit}`. `B1`'s only predecessor is `P` = `entry`, and the result of computing `effect` and `totaleffect` is ⟨00000000⟩, unchanged from the initial value of `dfin(B1)`, so `B1`'s successors are not put into the worklist.
>
> Next, we get `B` = `B2` and the worklist becomes `{B3,B4,B5,B6,exit}`. The only predecessor of `B2` is `P` = `B1`, and the result of computing `effect` and `totaleffect` is ⟨11100000⟩, which becomes the new value of `dfin(B2)`, and `exit` is added to the worklist to produce `{B3,B4,B5,B6,exit}`.
>
> Next, we get `B` = `B3`, and the worklist becomes `{B4,B5,B6,exit}`. `B3` has one predecessor, namely `B1`, and the result of computing `effect` and `totaleffect` is ⟨11100000⟩, which becomes the new value of `dfin(B3)`, and `B4` is put onto the worklist.
>
> Then we get `B` = `B4`, and the worklist becomes `{B5,B6,exit}`. `B4` has two predecessors, `B3` and `B6`, with `B3` contributing ⟨11110000⟩ to `effect`, `totaleffect` and `dfin(B4)`, and `B6` contributing ⟨00001111⟩, so that the final result of this iteration is `dfin(B4)` = ⟨11111111⟩ and the worklist becomes `{B5,B6,exit}`.
>
> Next, `B` = `B5`, and the worklist becomes `{B6,exit}`. `B5` has one predecessor, `B4`, which contributes ⟨11111111⟩ to `effect`, `totaleffect`, and `dfin(B5)`, and `exit` is added to the worklist.
>
> Next, `B` = `B6`, and the worklist becomes `{exit}`. `B6`'s one predecessor, namely `B4`, contributes ⟨11111111⟩ to `dfin(B6)` and `B4` is put back onto the worklist.
>
> Now `exit` is removed from the worklist, resulting in `{B4}`, and its two predecessors, `B2` and `B5`, result in `dfin(exit)` = ⟨11111111⟩.
>
> The reader can check that the body of the `repeat` loop is executed twice more for each element of the worklist, but that no further changes result in the `dfin( )` values computed in the last iteration. One can also check that the results are identical to those computed in Section 8.1.

P. 236, line 2: Change "*distributive*" to "*strongly distributive*".

P. 243: In line 14, change "$in(\texttt{B4a}) = F_{\texttt{B3}}(in(\texttt{B4a}))$" to "$in(\texttt{B4a}) = F_{\texttt{B3}}(in(\texttt{B3a}))$".

P. 254: In line 8, change "$idom(y) \neq x$" to "$idom(z) = x \ \&\ idom(y) \neq x$"; in line 11 (under the set-union symbol), change "$idom(z) = x$" to "$(idom(z) = x)$"; in line 14, change "dominates" to "dominates (i.e., $idom(x) = y$ if and only if $x \in$ `IDom(`y`)`)"; and, change the sentence that begins in line 15 to "The function `Post_Order(`N`,`$IDom$`)` returns a sequence of nodes that is a postorder traversal of the dominator tree represented by N and $IDom$.".

P. 255, Figure 8.19: Replace the body of the figure by the following:

```
IDom, Succ, Pred: Node ⟶ set of Node

procedure Dom_Front(N,E,r) returns Node ⟶ set of Node
    N: in set of Node
    E: in set of (Node × Node)
    r: in Node
begin
    y, z: Node
    P: sequence of Node
```

```
        i: integer
        DF: Node ⟶ set of Node
        Domin_Fast(N,r,IDom)
        P := Post_Order(N,IDom)
        for i := 1 to |P| do
            DF(P↓i) := ∅
            for each y ∈ Succ(P↓i) do
                if y ∉ IDom(P↓i) then
                    DF(P↓i) ∪= {y}
                fi
            od
            for each z ∈ IDom(P↓i) do
                for each y ∈ DF(z) do
                    if y ∉ IDom(P↓i) then
                        DF(P↓i) ∪= {y}
                    fi
                od
            od
        od
        return DF
    end    || Dom_Front
```

P. 265: In Exercise 8.2, insert "strongly" before "distributive".

Chapter 9. Dependence Analysis & Dependence Graphs

P. 272: In line 2, change "`MaxCycles:`" to "`MaxCycles, IssueLatency:`".

P. 274, Figure 9.6: In line 4, delete "`set of`"; in line -7, change "`⟨k,j⟩`" to "`k→j`"; and in line -6, change "`Inst[j]`" to "`IssueLatency+1,Inst[j],1`".

P. 286: In Figure 9.19, replace the "`Y`" label on the edges from node `B1` to both `B3` and `B4` by "`N`", and, in Figure 9.20, interchange the labels on the dashed edges from `B1` to `R1` and `R2`.

Chapter 10. Alias Analysis

P. 317, Figure 10.13: In line -10, change "`k == 0`" to "`k = 0`", and, in line -9, change "`q = &p[k]`" to "`q == &p[k]`".

Chapter 11. Introduction to Optimization

P. 328: Delete Exercise 11.3.

Chapter 12. Early Optimizations

P. 330: In line -13 of Figure 12.1, change "`opd1`" to "`opd`"; and, in each of lines 5 and 6 of the text, change the second occurrence of "*opd*1" to "*opd*".

P. 342: Replace lines 3 through 12 by the following:

can be applied safely to them. For example, the ANSI/IEEE floating-point standard includes zeroes with both positive and negative signs, i.e., `+0.0` and `-0.0`, and x`/+0.0 = +`∞ while x`/-0.0 = -`∞ for any positive finite value x. Also, x`+0.0` and x are not necessarily equal, since, if x is a signaling NaN, the first of them causes an exception when the arithmetic operation is executed, while the second generally would not.

Let MF denote the maximal finite floating-point value representable in a given precision. Then

In lines -19, -18, and -13, change the five occurrences of "`1`" to "`1.0`" and, in line -13, the single occurrence of "`0`" to "`0.0`".

P. 343: In lines 3 through 5 of the second paragraph in Section 12.4, change the text

> all four optimizations are, in fact, incomparable with each other. The examples in Figure 12.11 show situations where each is better than

to

> value numbering is, in fact, incomparable with the three others. The examples in Figure 12.11 show situations that distinguish value numbering from each of

in lines -5 through -3, delete the words

> while we have not shown that common-subexpression elimination and partial-redundancy elimination differ in power—which is easy, but we defer it to Section 13.3—

and append to the end of that paragraph the sentence

> As we shall see in Section 13.3, partial-redundancy elimination subsumes common-subexpression elimination.

in Figure 12.11(c), move the line "`l ← 2 * i`" to follow the line "`read(i)`" and, in the figure's caption, insert "and" before "common-" and delete ", and partial-redundancy elimination".

P. 345, Figure 12.13: In line -12, change "`nblocks,m,i`" to "`m,i,nblocks`".

P. 351: In line 12, change "five" to "four" and delete item 5 from the numbered list below.

P. 352, Figure 12.21: Delete line 2; in line 9, change "`i, m`" to "`i, j1, k1, m`"; in line 10, change "`j, j1, k, k1, p`" to "`j, k, p`"; and delete lines -17 and -8.

P. 353: In item 3, change "integer k" to "node y" and delete the text " and $y \in B[k]$".

P. 354, Figure 12.22: Delete lines 16 and 24; change "`[k]`" in line 25 to "`[i]`" and, in line 27, change "`{k}`" to "`{i}`"; move lines 25 through 27 to follow line 15; in line -12, change "`integer`" to "`Node`"; in line -7, change "`Node × integer × integer`" to "`Node × integer × Node`"; and, in line -4, change "`NtoB(el@3)`" to "`el@3`".

Pp. 357–358, Figure 12.24: Replace lines 5 through -4 of `Local_Copy_Prop( )` by the following code:

```
ACP := ∅: set of (Var × Var)
i: integer
for i := 1 to n do
   || replace operands that are copies
   case Exp_Kind(Block[m][i].kind) of
```

```
binexp:     Block[m][i].opd1.val := Copy_Value(Block[m][i].opd1,ACP)
            Block[m][i].opd2.val := Copy_Value(Block[m][i].opd2,ACP)
unexp:      Block[m][i].opd.val := Copy_Value(Block[m][i].opd,ACP)
listexp:    for j := 1 to |Block[m][i].args| do
               Block[m][i].args↓j@1.val := Copy_Value(Block[m][i].args↓j@1,ACP)
            od
default: esac
         || delete pairs from ACP that are invalidated by the current
         || instruction if it is an assignment
         if Has_Left(Block[m][i].kind) then
            Remove_ACP(ACP,Block[m][i].left)
         fi
         ||  insert pairs into ACP for copy assignments
         if Block[m][i].kind = valasgn & Block[m][i].opd.kind = var
            & Block[m][i].left ≠ Block[m][i].opd.val then
            ACP ∪= {⟨Block[m][i].left,Block[m][i].opd.val⟩}
```

and the routine `Replace_Copy( )` by the following two routines:

```
procedure Remove_ACP(ACP,v)
   ACP: inout set of (Var × Var)
   v: in Var
begin
   T := ACP: set of (Var × Var)
   acp: Var × Var
   for each acp ∈ T do
      if acp@1 = v ∨ acp@2 = v then
         ACP -= {acp}
      fi
   od
end    || Remove_ACP

procedure Copy_Value(opnd,ACP) returns Var
   opnd: in Operand
   ACP: in set of (Var × Var)
begin
   acp: Var × Var
   for each acp ∈ ACP do
      if opnd.kind = var & opnd.val = acp@1 then
         return acp@2
      fi
   od
   return opnd.val
end    || Copy_Value
```

P. 364: In line 17, change "edge $\langle a,b\rangle$ in the flowgraph" to "flowgraph edge $a \rightarrow b$" and delete the text "node n that defines an ("; in line 18, change "form)" to "form" and delete "n,"; in line 19, change the text "node n; since each node defines at most" to "the node that defines it."; delete line 20; in line 24, change "Five" to "Four"; delete item 1 in the numbered list and renumber the remaining items as "1" through "4"; in the new item 2, change "(k,x)" to "( )" and delete "where x ranges

over the possible variables"; in the new item 3, change "⟨*k*,*i*⟩" to "*k*→*i*"; and, in the new item 4, change "edges `e` in E such that `e@2 =` b" to "executable edges e such that e`@2 =` b".

P. 365, Figure 12.31: Insert after line 3, the line "`Succ: integer ⟶ set of integer`"; replace line 5 with the line

```
procedure Sparse_Cond_Const(ninsts,Inst,E,EL,entry)
```

insert the line "`EL: in (integer × integer) ⟶ enum {Y,N}`" after line 8; in line 12, delete "`c, d`"; in line 16, delete "`,V`"; in line 25, delete "`b,`"; in lines 27 and -5, replace "`])`" by "`],EL)`"; and delete lines -17 through -15.

P. 366, Figure 12.32: In line 15, delete "`,V`"; delete lines 19 and 23; and replace lines -5 and -4 by

```
        if Has_Left(Inst[i].kind) then
            LatCell(Inst[i].left) := ⊤
```

Pp. 366–367: In the text, change each ordered pair "⟨x,y⟩" to the form "x→y"; change the text on pp. 366–367, from line -2 on p. 366 through line -3 of p. 367 to the following:

> `Visit_Inst(B3,"`a_1 `<` b_1`")`, and so on. The result is that the lattice cells are set to `LatCell(`a_1`) = 2`, `LatCell(`b_1`) = 3`, `LatCell(`c_1`) = 4`, `LatCell(`c_2`) = ⊤`, and `LatCell(`c_3`) = 4`. Note that `LatCell(`c_2`)` is never changed because the algorithm determines that the edge

P. 367, Figure 12.32 *(continued)*: Replace line 1 by "`procedure Visit_Inst(k,inst,EL)`"; replace line 4 by "`EL: in (integer × integer) ⟶ enum {Y,N}`"; delete line 7; and replace lines 9 through -4 of the definition of the procedure `Visit_Inst( )` by

```
      val := Lat_Eval(inst): ConstLat
      if Has_Left(inst.kind) & val ≠ LatCell(inst.left) then
         LatCell(inst.left) ⊓= val
         SSAWL ∪= SSASucc(k)
      fi
      case Exp_Kind(inst.kind) of
   binexp, unexp:
      if val = ⊤ then
         for each i ∈ Succ(k) do
            FlowWL ∪= {k→i}
         od
      elif val ≠ ⊥ then
         if |Succ(k)| = 2 then
            for each i ∈ Succ(k) do
               if (val & EL(k,i) = Y) ∨ (!val & EL(k,i) = N) then
                  FlowWL ∪= {k→i}
            od
         elif |Succ(k)| = 1 then
            FlowWL ∪= {k→◆Succ(k)}
         fi
```

Chapter 13. Redundancy Elimination

Pp. 386–387: In the table of *EVAL*( ) and *KILL*( ) sets, change the following entries as shown:

$$\begin{aligned} EVAL(\texttt{B2}) &= \{\texttt{a+b,c>d}\} \\ EVAL(\texttt{B5}) &= \{\texttt{i>10}\} \\ KILL(\texttt{B1}) &= \{\texttt{c*2,c>d,a*c,d*d,i+1,i>10}\} \\ KILL(\texttt{B5}) &= \emptyset \end{aligned}$$

In the definition of $AEin(i)$ in line 12 on p. 387, change "$AEin(j)$" to "$AEout(j)$"; change "U_{exp}" to "{a+b,a*c,d*d,c>d,i>10}"; in the table of *AEin*( ) sets, add "a*c," to $AEin$(B2); add "c>d," to each of $AEin$(B3), $AEin$(B4), $AEin$(B5), and $AEin$(exit); and add ",i>10" to $AEin$(exit).

P. 388: Add to the end of item 4 the text "where $Replace(inst, exp, tj)$ returns the instruction *inst* with *exp* replaced by *tj*.".

P. 391: In the table of *EVALP*( ) and *KILLP*( ) sets, change the following entries as shown:

$$\begin{aligned} EVALP(\texttt{B2}) &= \{\langle\texttt{a+b,B1,1}\rangle, \langle\texttt{c>d,B2,3}\rangle\} \\ EVALP(\texttt{B5}) &= \{\langle\texttt{i>10,B5,2}\rangle\} \\ KILLP(\texttt{B1}) &= \{\langle\texttt{a*c,B3,1}\rangle, \langle\texttt{c*2,B2,2}\rangle, \langle\texttt{c>d,B2,3}\rangle, \langle\texttt{d*d,B1,2}\rangle, \langle\texttt{i+1,B5,1}\rangle, \langle\texttt{i>10,B5,2}\rangle\} \end{aligned}$$

In the table of *AEinP*( ) sets, add "⟨c>d,B2,3⟩" to each of $AEinP$(B3), $AEinP$(B4), $AEinP$(B5), and $AEinP$(exit), and add "⟨i>10,B5,2⟩" to $AEinP$(exit).

P. 392, Figure 13.11: Insert before line 1 the line

```
BinExp = Operand × Operator × Operand
```

In lines 1 and 2, replace "Operand × Operator × Operand" by "BinExp"; in line 7, change "(set of BinExp" to "BinExpIntPairSet"; and delete line 8.

P. 393, line 25: Change "set of BinExpIntPairSet" to "BinExpIntPairSet".

Pp. 403–404: Change the occurrence of "append_preheader(*inst*,*bset*)" in item 4 of the list on p. 403 to "append_preheader(*bset*,*ninsts*,*Block*,*inst*)" and change each occurrence of the function on p. 404 correspondingly. Also, on p. 404, in line 8, delete "j, ".

P. 417, line 4: Change "earliest" to "latest".

P. 423: In Exercise 13.3, in line 1, after "..." insert "and a set of optimizations covered in Chapters 12 and 13"; in lines 2 and 3, change "local and global common-subexpression elimination" to "the optimizations". Also, delete Exercise 13.9 and renumber the following two as 13.9 and 13.10.

Chapter 14. Loop Optimizations

P. 431: In item 3 of the enumerated list, change "in the loop" to "outside the loop".

P. 437, Figure 14.10: In line 5, change "MIRInst" to "array [··] of MIRInst"; and, in line -11, change "val:iv1.fctr" to "val:iv1.diff".

P. 445, block `B3`: Replace the line "`t23 ← t23 + 4 * t21`" by the two lines

```
t27 ← 4 * t21
t23 ← t23 + t27
```

P. 450, Figure 14.22: Change line 14 from "`v, tj: Var`" to "`tj: Var`"; and insert after line 14 the line "`v: Const`".

P. 452, Figure 14.22 *(continued)*, line -1: Delete this line.

P. 453, Figure 14.22 *(continued)*: In line 1, change "`if`" to "`elif`" and, in line -2, change "`od`" to "`L1: od`".

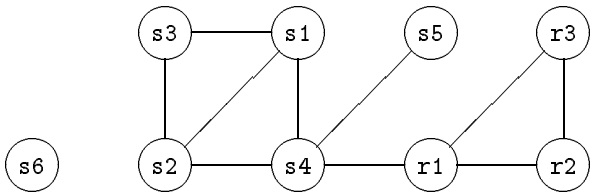
P. 460, Exercise 14.2: Add "**ADV**" in the left margin.

Chapter 15. Procedure Optimizations

P. 474: In the equations for $RANTout(i)$, change "$RANTin(i)$" to "$RANTin(j)$", and, in the equations for $RAVin(i)$, change "$RAVout(i)$" to "$RAVout(j)$".

Chapter 16. Register Allocation

P. 486, Figure 16.2: Replace the content of the figure by the following:



P. 492, Figure 16.8: In basic block `B4`, change "y_1" to "x_5".

P. 493, Figure 16.9: Change line 11 to the following:

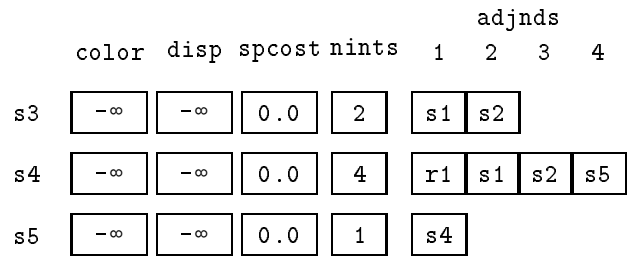
```
j := 1
while j ≤ ninsts[i] do
```

and insert the line "`j += 1`" after line -4.

P. 496, Figure 16.11: Remove the "`f`"s at positions `[s1,s1]`, `[s2,s2]`, through `[s5,s5]` and change the "`t`" at `[s5,s3]` to "`f`" and the "`f`" at `[s5,s4]` to "`t`".

P. 497: In Figure 16.12, after line 3, insert "`def: UdDu`", and, in line -1 of the text, change "that" to "that either".

Pp. 498: In line 1 of the text, change "other" to "other or neither *si* nor *sj* is stored into between the copy assignment and the end of the routine." In Figure 16.13, change the entries for `s3`, `s4`, and `s5` as follows:



In line 10 of the text, change "two" to "three"; add item 3 to the enumerated list, as follows:

> **3.** `Non_Store(,LBlock,`k`,`l`,`i`,`j`)` returns `true` if neither symbolic register `s`k nor `s`l is stored into between the assignment `s`k ← `s`l in `LBlock[`i`][`j`]` and the end of the routine containing that copy assignment and `false` otherwise.

and, in line 2 of footnote 6, change "si and sj may each" to "each of si and sj may".

P. 500, Figure 16.15: Replace lines 8 through -4 by the following:

```
for i := 1 to nblocks do
   for j := 1 to ninsts[i] do
      || if this instruction is a coalescable copy, adjust
      || assignments to its source to assign to its target instead
      if LBlock[i][j].kind = regval then
         k := Reg_to_Int(LBlock[i][j].left)
         l := Reg_to_Int(LBlock[i][j].opd.val)
         if !AdjMtx[max(k,l),min(k,l)] ∨ Non_Store(LBlock,k,l,i,j) then
            for p := 1 to nblocks do
               for q := 1 to ninsts[p] do
                  if LIR_Has_Left(LBlock[p][q])
                     & LBlock[p][q].left = LBlock[i][j].opd.val then
                     LBlock[p][q].left := LBlock[i][j].left
                  fi
               od
            od
            || remove the copy instruction and combine Symregs
            delete_inst(i,j,ninsts,LBlock,Succ,Pred)
            Symreg[k].defs ∪= Symreg[l].defs
            Symreg[k].uses ∪= Symreg[l].uses
            || adjust adjacency matrix to reflect removal of the copy
            Symreg[l] := Symreg[nwebs]
            for p := 1 to nwebs do
               if AdjMtx[max(p,l),min(p,l)] then
                  AdjMtx[max(p,k),min(p,k)] := true
               fi
               AdjMtx[max(p,l),min(p,l)] := AdjMtx[nwebs,p]
            od
            nwebs -= 1
         fi
      fi
```

P. 502, Figure 16.16: In line 24, change "`valasgn`" to "`regval`".

P. 509, Figure 16.22: Change the last line in basic block `B5` from "`use x`" to "`def x`".

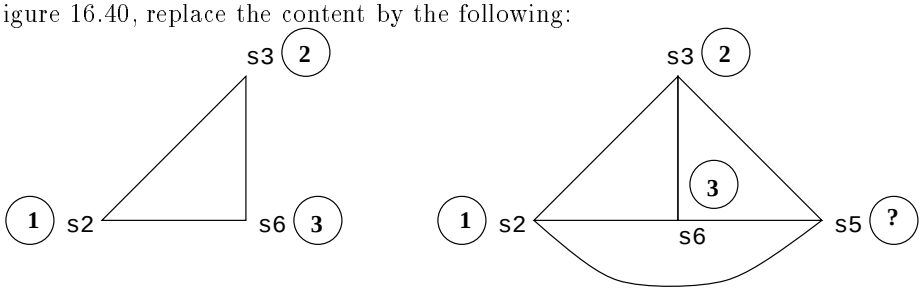
P. 519: In Figure 16.37(a), add an arc connecting `s2` and `s5`; in Figure 16.37(b), change the "`f`" at `[s5,s2]` to "`t`" and the "`t`" at `[s6,s4]` to "`f`"; change lines 5 and 6 of the text to say "and the resulting interference graph is shown in Figure 16.39(a). Removing node `s8` and pushing it onto the stack results in the stack"; in line 7, delete "`s2`" from the stack; in line 9, change "degree three" to "degree greater than or equal to three"; and, replace the last sentence on the page by "So we again select a node with minimal spill cost divided by current degree, namely, `s5`, and push it onto the stack. The resulting interference graph is shown in Figure 16.40(a). Now all nodes have degree less than three, so we push them all onto the stack, resulting in the stack".

P. 520: In Figure 16.39(a) add an arc from `s2` to `s5`; in (b) and (c), retain the node `s2` from part (a) and add arcs connecting `s2` with each of `s3`, `s5`, and `s6`; in the caption, change "nodes `s2` and `s8`" to "node `s8`" and change "pushing them" to "pushing it"; replace the stack diagram that is the first line of text on the page, by the following one:

```
s2 s3 s6 s5 s7 s8 s9 s4 r4 r3 r2 s1
```

in line 2, change "to pop" to "popping"; and, in line -2, change "16.40" to "16.40(b)".

P. 521: In Figure 16.40, replace the content by the following:

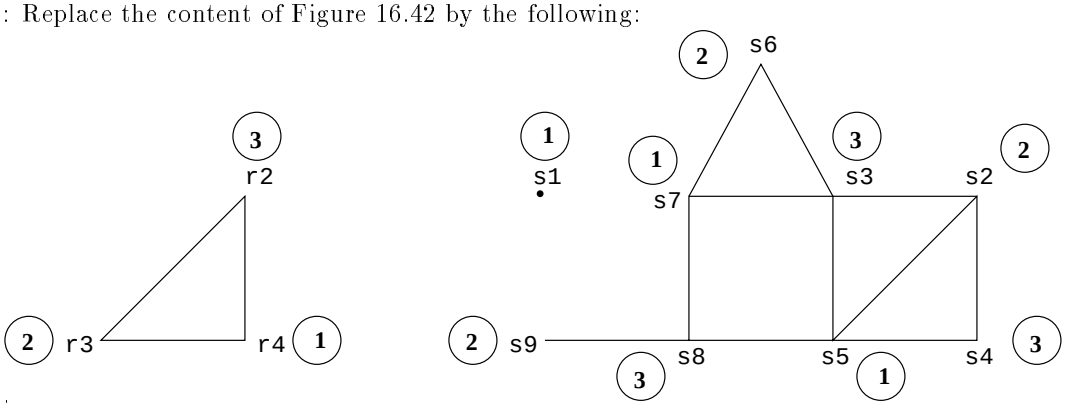


in the caption, change "after" to "(a) after", "four" to "three", and "stack" by "stack and (b) after popping the fourth node".

In Figure 16.41, insert the line "`[r10+4]` ← `s5`" after the third line in each of blocks `B2` and `B3`; insert the line "`s5` ← `[r10+4]`" after the third line in block `B4`; and insert the text "`s5` and" after "for" in the caption.

In line 1 of the text, change "`s7`" to "`s7` and `s5`", and, in line 2, change "`Disp = 0`" to "`Disp = 0` and `Disp = 4`, respectively".

P. 522: Replace the content of Figure 16.42 by the following:



and, in Figure 16.43, insert the line "`[r10+4] ← r4`" after the third line in each of blocks `B2` and `B3` and the line "`r4 ← [r10+4]`" after the third line in block `B4`.

P. 529: In Exercises 16.1, 16.2, and 16.3, change "Modify the code ... in Figure 16.15" to "Explain how `Build_AdjMtx( )` in Figure 16.12 and `Coalesce_Registers( )` in Figure 16.15 can be used". Also, in each, delete the last sentence (beginning "Assume").

P. 530, Exercise 16.8: Append the sentence "Note that, in Figure 16.41, we would have produced better code if, instead of spilling `s7`, we had rematerialized it before the `print( )` call."

Chapter 17. Code Scheduling

P. 538: In line 6 of the text, change "`Latency(`n`,`m`)`" to "`Latency(LInst(`n`),2,LInst(`m`),1)`" and, in line -11, change "DAG" to "and `LInst(`i`)` is the LIR instruction represented by node i in the DAG".

P. 539: In Figure 17.5, change line 3 to the two lines

```
LInst: Node ⟶ LIRInst
DagSucc: (Node × Dag) ⟶ set of Node
```

and change lines -8 and -7 to the following:

```
for each n ∈ DagSucc(PostOrd[i],Dag) do
   ld := Latency(LInst(PostOrd[i]),2,LInst(n),1) + Delay(n)
```

Also, change the first three lines in the Node-`Delay` table at the bottom of the page to

```
1      5
2      4
3      4
```

P. 540: In Figure 17.6, in line 3, delete "`set of`". In numbered item 2, change "`2.`" to "`1.`".

P. 541: In Figure 17.6, change each occurrence of "`Etime(`i`)`" to "`Etime[`i`]`"; change lines 12–15 to

```
elif |ECands| > 1 then
   n := Heuristics(ECands)
else
   n := Heuristics(MCands)
```

and, in line -5, change "`Inst(n),1,Inst(m)`" to "`LInst(n),2,LInst(m)`".

P. 553: In the second numbered item 2, change "sets `Moved` accordingly" to "returns `true` if it moved the window and `false` otherwise".

P. 555: In line 2 of the text, change the second occurrence of "n to "$n-1$" and, in line 3, change "$n+1$" to "n".

P. 558: In Figure 17.27, delete line 10; in line 16, delete "`,ninsts[i]`"; and, in line 18, change "`Dag.Nodes`" to "`|Dag.Nodes|`". In lines 2 and 3 of the text, change "m" to "i"; in line -7, change "m" to "k"; in line -6, change "i" to "j"; in line -5, change "m" to "k" and "i" to "j".

P. 566, Figure 17.33: In line 6, change "`integer` $\longrightarrow$" to "`array [1··m] of`"; change line 19 to

```
i := 1
while i ≤ ninsts[m] do
```

in line -13, change "`(m)`" to "`[m]`"; and, after line -3, insert the line "`i += 1`".

Chapter 18. Control-Flow and Low-Level Optimizations

P. 581, Figure 18.1: Replace lines 1 through -9 by the following:

```
procedure Elim_Unreach_Code(en,nblocks,ninsts,Block,Pred,Succ)
   en: in integer
   nblocks: inout integer
   ninsts: inout array [1··nblocks] of integer
   Block: inout array [1··nblocks] of array [··] of Instruction
   Pred, Succ: inout integer → set of integer
begin
   again: boolean
   i: integer
   repeat
      again := false
      i := ◆Succ(en)
      while i ≤ nblocks do
         if No_Path(en,i) then
            ninsts[i] := 0
```

In line 2 of the text, change ". If any of the deleted block's successors have become unreachable," to "and"; and in line -9 of the text, delete the sentence "We assume that `Block[1]` is the entry block."

P. 593: Combine numbered items 5 and 6 as numbered item 5 with the text

> **5.** Remove from the procedure each instruction that is unmarked, and remove each basic block that has become empty.

In lines -12 through -10, delete "routine `If_Inst(`k`)` returns `true` if an instruction of kind k is a conditional and `false` otherwise and"; in line -10, change "used" to "used as operands"; and, in line -9, change "`Block[`x`@1][`x`@2]`" to "`Block[`x`@1][`x`@2]`, and `Delete_Unmarked_Insts( )` uses `delete_inst( )` to delete each unmarked instruction and adjusts `Mark[ ][ ]` to reflect the deletions."

P. 594, Figure 18.19: Replace lines -10 and -9 by

```
if !Mark[y@1][y@2] & Block[y@1][y@2].kind
   ∈ {binif,unif,valif,bintrap,untrap,valtrap} then
```

and, in line -2, change "`Empty_Blocks`" to "`Unmarked_Insts`" and "`Pred)`" to "`Pred,Mark)`".

P. 596: Replace the text from the words "So we check" in line 11 to the end of the page by the following text:

> So we set `v = i` and check each member of `UD(i,⟨B4,1⟩) = {⟨B2,1⟩}`. We set `Mark[B2][1] = true` and add `⟨B2,1⟩` to the worklist, resulting in `Worklist = {⟨B2,2⟩, ⟨B2,1⟩}`. Next we set `v = j`, and find that `UD(j,⟨B4,1⟩) = {⟨B2,3⟩}` resulting in setting `Mark[B2][3] = true` and adding `⟨B2,3⟩` to the worklist, so `Worklist = {⟨B2,2⟩,⟨B2,1⟩,⟨B2,3⟩}`. Next we determine that `Has_Left(Block[B4][1].kind) = false`, so we move on to the next element of the worklist.
>
> Now we remove `x = ⟨B2,2⟩` from the worklist, leaving it with the value `Worklist = {⟨B2,1⟩,⟨B2,3⟩}`, and find that `Vars_ Used(Block,⟨B2,2⟩) = {j}`. So we set `v = j` and check each member of `UD(j,⟨B2,2⟩) = {⟨B1,2⟩,⟨B2,3⟩}`. This results in setting `Mark[B1][2] = true` (`Mark[B2][3]` is already `true`) and adding `⟨B1,2⟩` to the worklist, i.e., `Worklist = {⟨B2,1⟩,⟨B2,3⟩,⟨B1,2⟩}`.
>
> Next we remove `⟨B2,1⟩` from the worklist, leaving `Worklist = {⟨B2,3⟩,⟨B1,2⟩}`. `Vars_Used(Block,⟨B2,1⟩) = {i,j}`, so we examine `UD(i,⟨B2,1⟩) = {⟨B1,1⟩,⟨B2,1⟩}`. We set `Mark[B1][1] = true` and put `⟨B1,1⟩` on the worklist, resulting in `Worklist = {⟨B2,3⟩,⟨B1,2⟩,⟨B1,1⟩}`. `Mark[B2][1]` is already set to `true`, so we proceed to the next element of the worklist. Now `UD(j,⟨B2,1⟩) = {⟨B1,2⟩,⟨B2,3⟩}`, both of which are already marked, so we move on.
>
> Next we remove `⟨B2,3⟩` from the worklist, leaving it with the value `Worklist = {⟨B1,2⟩}`. Now `Vars_Used(Block,⟨B2,3⟩) = {j}`, so we check whether either of `Mark[B1][2]` and `Mark[B2][3]` is marked. Both are already marked, so we proceed to determine that `Has_Left(Block[B2][3].left) = true`, so we check each of `Block[B2][1]`, `Block[B2][2]`, `Block[B2][3]`, `Block[B2][4]`, `Block[B3][1]`, and `Block[B4][1]` to determine whether the instructions in any of those positions are conditionals and unmarked. Only `Block[B2][4]` is, so we set `Mark[B2][4] = true` and add `⟨B2,4⟩` to the worklist, resulting in `Worklist = {⟨B1,2⟩,⟨B2,4⟩}`.
>
> Now we remove `⟨B1,2⟩` from the worklist, leaving it with the value `Worklist = {⟨B2,4⟩}`. Now `Vars_Used(Block,⟨B1,2⟩) = ∅` and `DU(Block[B1][2],j) = {⟨B2,1⟩,⟨B2,2⟩, ⟨B2,3⟩}`, all of which are already marked.
>
> Finally, we remove `⟨B2,4⟩` from the worklist, leaving it empty. `Mark[B2][4]` is already `true`, so the algorithm terminates with the value of `Mark` being as shown in Figure 18.21(a) and the flowgraph resulting from performing dead-code elimination is shown in Figure 18.21(b).

P. 605, Exercise 18.4: Change "Extend" to "Show that"; change "to recognize" to "recognizes"; and change "eliminate" to "eliminates".

Chapter 19. Interprocedural Analysis & Optimization

P. 618: In line 2 of the text, change "`(f)`" to "`(f,Inst,numinsts)`".

P. 630, Figure 19.18: In line 1, change "`(r)`" to "`( )`"; delete line 2; and, in line 5, change "`n:`" to "`n, r:`".

P. 638, Figure 19.21: In line 13, change "`Var`" to "`(Var ∪ Const)`".

P. 644, Figure 19.25: In line 4, change "`call`" to "`a :=`".

P. 645, Figure 19.26: Delete line -11 and, in line -7, change "`u,v`" to "`v,u`".

P. 646: In numbered item 5, in line 1, delete "R,"; in line 2, delete ", roots R"; and, in line 3, append the sentence "(Note that there may be more than one root.)". Also, in line 1 of the following text, change "Note that if" to "If".

P. 648: In Table 19.11, change the following entries as shown:

```
A(i) = {a,z}            A(j) = {a,z}            A(k) = {a,b1,w}
A(l) = {b1,w}           A(m) = {b1,c,w}         A(r) = {a,b2,j,z}
A(s) = {b1,c,b2}        A(u) = {a,b2,j}         A(v) = {a,k}

   ALIAS(b1,p) = ALIAS(b2,n) = ALIAS(c,p) = ALIAS(b2,p) = ∅
   ALIAS(b1,t) = {s,k,l}
```

and, in the caption for Figure 19.29, change "nonlocal variables aliased to formal parameters" to "formal parameters aliased to each nonlocal variable".

P. 653, Table 19.12: In the caption, change "Formal-parameter alias sets" to "Alias sets for formal parameters", and, in the body of the table change "`ALIAS(s,p) = {b1,r}`" to "`ALIAS(s,p) = {b1,c,r}`" and "`ALIAS(r,p) = {a,s}`" to "`ALIAS(r,p) = {a,b2,s}`".

P. 654, Figure 19.36: Replace lines 8 through -7 by the following:

```
i: integer
v: Var
for each p ∈ P do
    for i := 1 to numinsts(p) do
        || initialize MOD with direct MOD
        MOD(p,i) := DMOD(p,i)
        || add formal-parameter aliases
        for each v ∈ Formals(p) do
            if v ∈ DMOD(p,i) then
                MOD(p,i) ∪= ALIAS(v,p)
            fi
        od
        || add nonlocal-variable aliases
```

P. 655: In Figure 19.37, change "`p'`" to "`P'`" and "`p`" to "`P`", and, in lines 2 and 3 of the text, change "`P`" to "`P'`".

P. 656: In the first paragraph, change "P_1" to "P_1'", "P_k" to "P_k'", and both occurrences of "P_i" to "P_i'".

Chapter 20. Optimization for the Memory Hierarchy

P. 671: In Figure 20.1, in part (c) replace lines 4 through 7 by the following:

```
do jj = j,min(j+t-1,N)
  do kk = k,min(k+t-1,N)
    C(i,jj) = C(i,jj)
       + A(i,kk) * B(kk,jj)
```

and in part (d) replace lines 4 through 7 by

```
do jj = j,min(j+t-1,N)
  do kk = k,min(k+t-1,N)
    C(i,jj) = C(i,jj)
       + A(kk,i) * B(jj,kk)
```

Also, in numbered item 3, change "`N`" to "`t`".

P. 674: In Figure 20.2, in line 7, change "`A:`" to "`A := ∅:`"; in line 10, delete "`origwt, `"; delete line 12; in line 14, change "`A`" to "`T := A`"; in line 15, delete "`origwt([e@1,e@2]) := `" and change "`weight(e)`" to "`weight(e@1,e@2)`"; and, in line -4, change "`origwt`" to "`weight`". In line 1 of the text, change "*origwt,weight*" to "*weight,psweight*", and, in line 3, change "*weight*" to "*psweight*".

P. 675, Figure 20.3: Move the comment in line -16 to precede line 18; replace line -18 by "`T := (T ∪ {padd}) - {p1,p2}`"; in line -15, change "`T`" to "`T (p ≠ padd)`"; and delete lines -15, -4, and -2.

P. 676, Figure 20.4: In the body of procedure `Remove( )`, replace lines 2 through -3 by the following:

```
   A: in set of ProcSeq
   s: in ProcSeq
begin
   t: ProcSeq
   for each t ∈ T do
      if Same(s,t) then
         A -= {s}
         return A
      fi
   od
```

P. 680, Figure 20.6 *(continued)*: In line 24, change "`[c]`" to "`c`".

P. 684: In line 7 of the text, change both occurrences of "*elem*" to "*A*`[`*i*`]`" ; in line 8, after "references" insert "to *A*`[`*i*`+`*j*`*`*s*`]`, where *s* is the stride of sequential accesses to *A*," and change "$t_0 \leftarrow$ *elem* or *elem* $\leftarrow t_0$" to "$t_j \leftarrow$ `A[`*i*`+`*j*`*`*s*`]` or `A[`*i*`+`*j*`*`*s*`]` $\leftarrow t_j$".