This document contains solutions to the exercises in Chapters 1 through 8 of *Engineering a Compiler, 2nd Edition*.

The solution for exercise 7.13(a) is incomplete.

Engineering a Compiler, 2e

Solutions to Exercises, V1.0
Chapters 1 through 8

Keith D. Cooper

Linda Torczon

Rice University

Houston, Texas

CHAPTER 1

The questions for Chapter 1 are all "soft" questions, in the sense that they have no single correct answer. Instead, they are intended to provoke thought and discussion. Our answers are more detailed than we would expect from a beginning student.

1. Consider a simple web browser that takes as input a textual string in HTML format and displays the specified graphics on the screen. Is the display process one of compilation or interpretation?

   Answer: A browser can be viewed as a compiler that translates a web page from a textual language to a graphical language. It can also be viewed as an interpreter because it parses the input every time that it is displayed. Because the web page, viewed as a program, takes no input, the distinction between compilation and interpretation is blurred.

2. In designing a compiler, you will face many tradeoffs. What are the five qualities that you, as a user, consider most important in a compiler that you purchase? Does that list change when you are the compiler writer? What does your list tell you about a compiler that you would implement?

   Answer: The answers are subjective, so no single set of answers is correct. The authors' subjective answers are:

   From the buyer's perspective:

   (a) Compiled code is correct.
   (b) Compiled code adheres to system-wide calling conventions—that is, the code it produces operates correctly with code produced by other compilers and, if possible, other languages.
   (c) Compiler produces efficient code for my applications.
   (d) Compiler supports one or more of the standard interfaces for debug information, such as the DWARF standard.
   (e) Compiler produces clear, well-localized error messages.

   From the compiler writer's perspective:

   (a) Compiled code is correct.
   (b) Compiler produces efficient code for a broad variety of input programs.
   (c) Debugging individual passes in the compiler is relatively easy.
   (d) The compiler's intermediate representation is human-readable.
   (e) The compiler's design makes it easy to add and replace passes.

Insights lists: Optimization matters, as does compile time.

3. Compilers are used in many different circumstances. What differences might you expect in compilers designed for the following applications?

   Many answers are possible.

   (a) A *just-in-time* compiler used to translate user interface code downloaded over a network

   Answer: The time constraints on a just-in-time compiler make it unlikely that the compiler can afford to make many passes over the code. This may limit the amount of optimization that it can do.

   (b) A compiler that targets the embedded processor used in a cellular telephone

   Answer: A compiler for an embedded processor may focus on issues other than speed. In particular, generating code that is small and code that uses less power may be more important. However, it is not clear that speed and low power are, necessarily, competing issues.

   (c) A compiler used in an introductory programming course at a high school

   Answer: A compiler for an introductory programming course should give detailed feedback on syntax errors. It may also perform analysis to report on semantic errors that are not enforced by the language standard.

   (d) A compiler used to build wind-tunnel simulations that run on a massively parallel processor (where all processors are identical)

   Answer: A compiler for a massively parallel machine will concentrate on performance issues, especially finding sections of code that can execute in parallel.

   (e) A compiler that targets numerically intensive programs to a large number of diverse machines

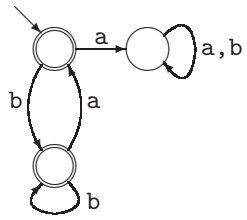
   Answer: A multi-platform compiler for numerically-intensive codes will focus on retargetability in its code generator and on machine-independent optimizations—particularly those that help improve array-element references in nested loops.

CHAPTER 2

Section 2.2

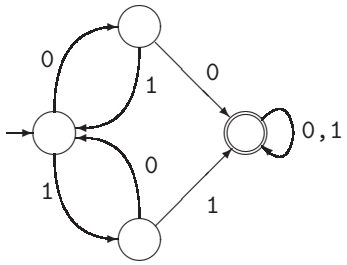
1. Describe informally the languages accepted by the following FAs:

a.



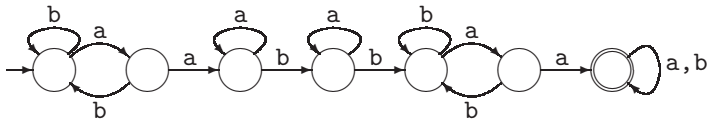
Answer: The set of strings in which every 'a' is preceded by a 'b'

b.



Answer: The set of strings that are not a concatenation of the strings '01' or '10'

c.

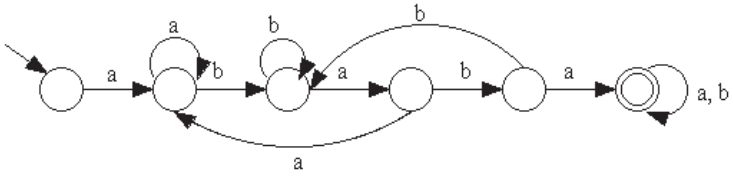


Answer: The set of strings that contain 'aab' as a substring and then 'baa' as a substring (not necessarily concatenated one with the other).

2. Construct an FA accepting each of the following languages:

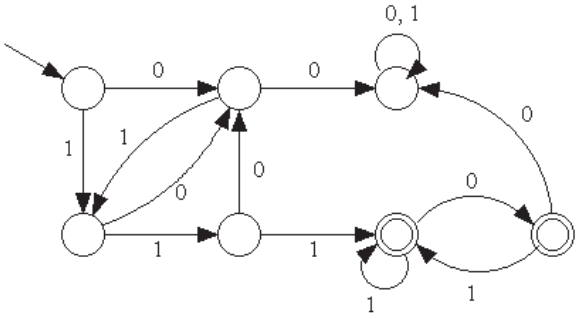
   a. $\{w \in \{a,b\}^* \mid w$ starts with 'a' and contains '$baba$' as a substring$\}$

   Answer:



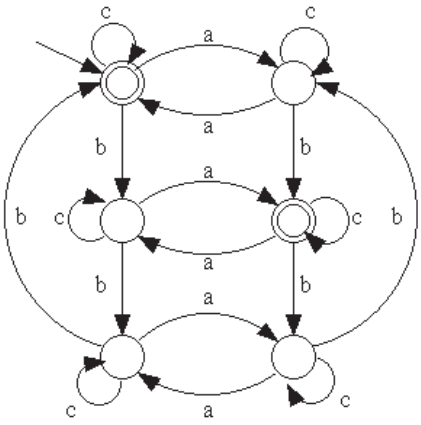
   b. $\{w \in \{0,1\}^* \mid w$ contains '111' as a substring and does not contain '00' as a substring$\}$

   Answer:



   c. $\{w \in \{a,b,c\}^* \mid$ in w the number of 'a's modulo 2 is equal to the number of 'b's modulo 3$\}$

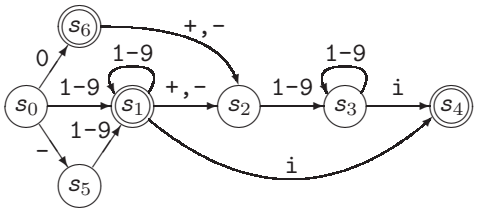
   Answer:



3. Create FAs to recognize

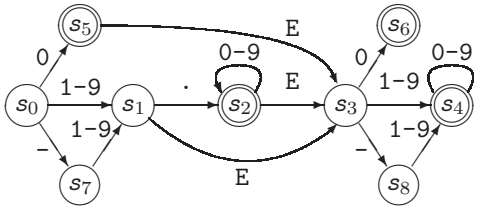
   a. words that represent complex numbers

   Answer:



   b. words that represent decimal numbers written in scientific notation.

   Answer:



Section 2.3

4. Different programming languages use different ways to represent integers. Construct a regular expression for each one of the following:

   a. Nonnegative integers in C represented in bases 10 and 16

   Answer: $[0-9][0-9]^*$ |
   0x0 | 0x$([1-9]|[a-f]|[A-F])([0-9]|[a-f]|[A-F])^*$

   b. Nonnegative integers in VHDL that may include underscores

   (An underscore cannot occur as the first or last character.)

   Answer: $[0-9](\_^*[0-9])^*$

   VHDL allows leading zeroes.

c. Currency, in dollars, represented as a positive decimal number rounded to the nearest one-hundredth. Such numbers begin with the character $, have commas separating each group of three digits to the left of the decimal point, and end with two digits to the right of the decimal point, for example, $8,937.43 and $7,777,777.77

Answer:

$\$(([1-9](\epsilon|[0-9])(\epsilon|[0-9])(\underline{,}[0-9][0-9][0-9])^*)|0)\,\underline{.}\,[0-9][0-9]$

(Underlines added to emphasize punctuation marks.)

Hint: not all the specifications describe regular languages.

5. Write a regular expression for each of the following languages.

a. Given an alphabet $\Sigma = \{0, 1\}$, L is the set of all strings of alternating pairs of 0s and pairs of 1s.

Answer: $(1100\,(1100)^*(11 \mid \epsilon)) \mid (0011\,(0011)^*(00 \mid \epsilon)$

b. Given an alphabet $\Sigma = \{0, 1\}$, L is the set of all strings of 0s and 1s that contain an even number of 0s or an even number of 1s.

Answer: $(1^*01^*01^*)^+ \mid (0^*10^*10^*)^+$

c. Given the lowercase English alphabet, L is the set of all strings in which the letters appear in ascending lexicographical order.

Answer: $a^*b^*c^*d^*e^*f^*g^*h^*i^*j^*k^*l^*m^*n^*o^*p^*q^*r^*s^*t^*u^*v^*w^*x^*y^*z^*$

d. Given an alphabet $\Sigma = \{a, b, c, d\}$, L is the set of strings $xyzwy$, where x and w are strings of one or more characters in Σ, y is any single character in Σ, and z is the character z, taken from outside the alphabet.

(Each string xyzwy contains two words xy and wy built from letters in Σ. The words end in the same letter, y. They are separated by z.)

Answer:
$(a|b|c|d)(a|b|c|d)^*a\ z\ (a|b|c|d)(a|b|c|d)^*a\ \mid$
$(a|b|c|d)(a|b|c|d)^*b\ z\ (a|b|c|d)(a|b|c|d)^*b\ \mid$
$(a|b|c|d)(a|b|c|d)^*c\ z\ (a|b|c|d)(a|b|c|d)^*c\ \mid$
$(a|b|c|d)(a|b|c|d)^*d\ z\ (a|b|c|d)(a|b|c|d)^*d$

e. Given an alphabet $\Sigma = \{+, -, \times, \div, (,), \texttt{id}\}$, L is the set of algebraic expressions using addition, subtraction, multiplication, division, and parentheses over `id`'s.

Answer: This language cannot be described in a regular expression because it requires matching parentheses.

6. Write a regular expression to describe each of the following programming language constructs:

   a. Any sequence of tabs and blanks (sometimes called *white space*)

      Answer: $(\not{b} \mid \backslash\mathtt{t})^*$

   b. Comments in the programming language C

      Answer: $/\!*\,[\,(\Sigma - *)^*\,(*^*(\Sigma - \{/, *\}))^*\,]^*\,*^*\;\; */$,

      where Σ represents the alphabet and $(\Sigma - x)$ represents the alphabet excluding the character 'x'.

   c. String constants (without escape characters)

      Answer: $"\,(\Sigma - ")^*\,"$

   d. Floating-point numbers

      Answer: The regular expression can become quite complex, depending on the specific notation used for exponents. See, for example, the FA in Section 2.2, question 3.b.

      For simple decimal numbers, without scientific notation, the following RE would work:

      $(0 \mid [1-9][0-9]^*)\,.\,[0-9]^*$

      It allows trailing zeros to indicate precision.

Section 2.4

7. Consider the three regular expressions:

$(ab \mid ac)^*$
$(0 \mid 1)^*\ 1100\ \ 1^*$
$(01 \mid 10 \mid 00)^*\ 11$

We present the solution for each RE in its entirety.

Answer: For the RE $(ab \mid ac)^*$:

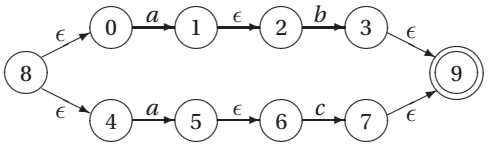
Part (a): Build an NFA for the RE using Thompson's construction.

Note: the NFAs can be built in any order that is consistent with the precedence among RE operators.

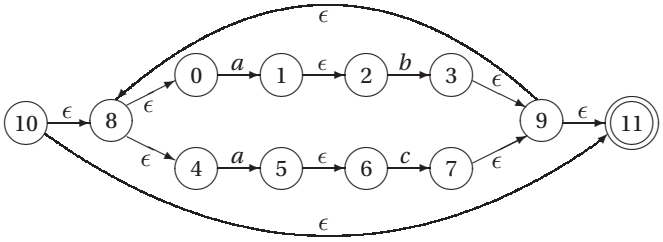
First, build DFAs for *a*, *b*, *a*, and *c*.

Next, combine them into NFAs for *ab* and *ac*

Combine the NFAs for *ab* and *ac* to form an NFA for $ab \mid ac$.



Finally, add two states and the ϵ-transitions for the closure to form an NFA for $(ab|ac)^*$.



Part (b): Convert the NFA to a DFA using the subset construction.

Initialization:

Note: the "core" states, those generated by δ, are set in bold. The states added by ϵ-closure are set in normal face.

The NFA's start state is 10.

$s_0 \leftarrow \epsilon\text{-closure}(10) = \{\mathbf{10}, 8, 0, 4, 11\}$

Round 1:

$s_1 \leftarrow \epsilon\text{-closure}(\delta(s_0,a)) = \{\mathbf{1}, \mathbf{5}, 2, 6\}$

There are no transitions out of s_0 on either *b* or *c*.

Round 2:

There is no transition out of s_1 on *a*.

$s_2 \leftarrow \epsilon\text{-closure}(\delta(s_1,b) = \{\mathbf{3}, 9, 8, 0, 4, 11\}$

$s_3 \leftarrow \epsilon\text{-closure}(\delta(s_1,c) = \{\mathbf{7}, 9, 8, 0, 4, 11\}$

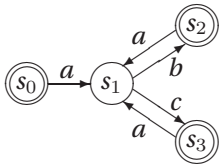
Round 3:

$\epsilon\text{-closure}(\delta(s_2,a) = s_1$

$\epsilon\text{-closure}(\delta(s_3,a) = s_1$

There are no transitions out of either s_2 or s_3 on *b* or *c*.

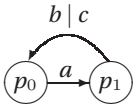
The worklist is empty. The resulting DFA has four states:



Part (c): Minimize the DFA:

Current Partition	Transition	Action
$\{ p_0 = \{s_0, s_2, s_3\}, p_1 = \{s_1\} \}$	—	—
	$\delta(p_0,a) = p_1$	*none*
	$\delta(p_0,b) = \emptyset$	*none*
	$\delta(p_0,c) = \emptyset$	*none*
	$\delta(p_1,a) = \emptyset$	*none*
	$\delta(p_1,b) = p_0$	*none*
	$\delta(p_1,c) = p_0$	*none*

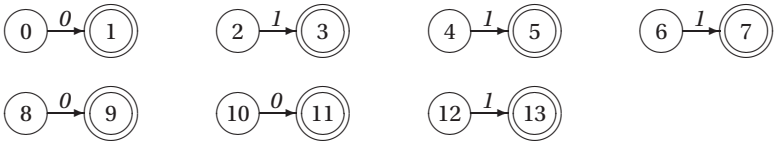
None of the potential transitions splits a set, so the minimal DFA has two states, as shown to the right.



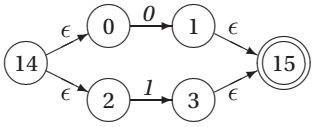
Answer: For the RE $(0|1)^*\ 1100\ 1^*$:

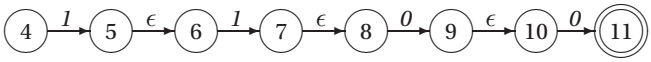
Part (a): Build an NFA for the RE using Thompson's construction.

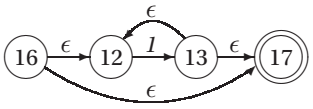
First build distinct DFAs for each of the digits:



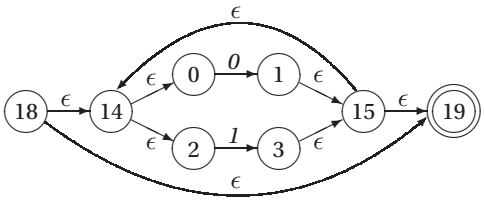
Next, combine these DFAs into NFAs for $0|1$, 1100, and 1^*:



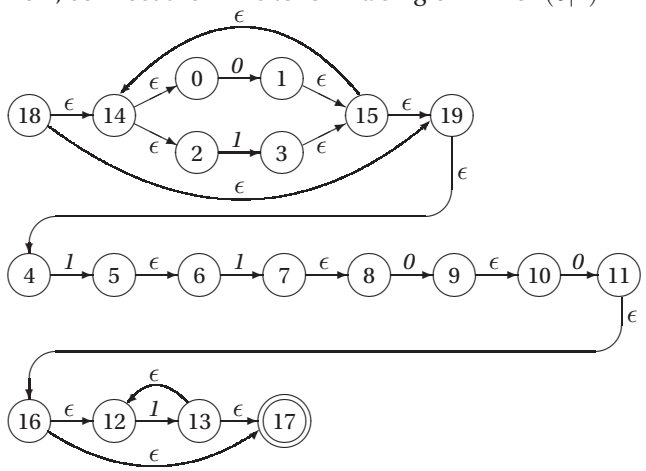




Next, build the NFA for $(0|1)^*$:



Now, connect the NFAs to form a single NFA for $(0|1)^*\ 1100\ 1^*$:



Part (b): Convert the NFA to a DFA using the subset construction.

Initialization:

The NFA's start state is 18.

$s_0 \leftarrow \epsilon\text{-closure}(18) = \{\mathbf{18}, 14, 0, 2, 19, 4\}$

Note: the "core" states, those generated by δ, are set in bold. The states added by ϵ-closure are set in normal face.

Round 1:

$s_1 \leftarrow \epsilon\text{-closure}(\delta(s_0,0)) = \{\mathbf{1}, 15, 14, 0, 2, 19, 4\}$

$s_2 \leftarrow \epsilon\text{-closure}(\delta(s_0,1)) = \{\mathbf{3}, \mathbf{5}, 15, 14, 0, 2, 19, 4, 6\}$

Round 2:

$\epsilon\text{-closure}(\delta(s_1,0)) = \{\mathbf{1}, \dots\} = s_1$

$\epsilon\text{-closure}(\delta(s_1,1)) = \{\mathbf{3}, \mathbf{5}, \dots\} = s_2$

$\epsilon\text{-closure}(\delta(s_2,0)) = \{\mathbf{1}, \dots\} = s_1$

$s_3 \leftarrow \epsilon\text{-closure}(\delta(s_2,1)) = \{\mathbf{3}, \mathbf{5}, \mathbf{7}, 15, 14, 0, 2, 19, 4, 6, 8\}$

Round 3:

$s_4 \leftarrow \epsilon\text{-closure}(\delta(s_3,0)) = \{\mathbf{1}, \mathbf{9}, 15, 14, 0, 2, 19, 4, 10\}$

$\epsilon\text{-closure}(\delta(s_3,1)) = \{\mathbf{3}, \mathbf{5}, \mathbf{7}, \dots\} = s_3$

Round 4:

$s_5 \leftarrow \epsilon\text{-closure}(\delta(s_4,0)) = \{\mathbf{1}, \mathbf{11}, 15, 14, 0, 2, 19, 4, 16, 12, 17\}$

$\epsilon\text{-closure}(\delta(s_4,1)) = \{\mathbf{3}, \mathbf{5}, \dots\} = s_2$

Round 5:

ϵ-closure($\delta(s_5,0)$) $= \{\mathbf{1}, \dots\} = s_1$

$s_6 \leftarrow \epsilon$-closure($\delta(s_5,1)$) $= \{\mathbf{3}, \mathbf{5}, \mathbf{13}, 15, 14, 0, 2, 19, 4, 6, 12, 17\}$

Round 6:

ϵ-closure($\delta(s_6,0)$) $= \{\mathbf{1}, \dots\} = s_1$

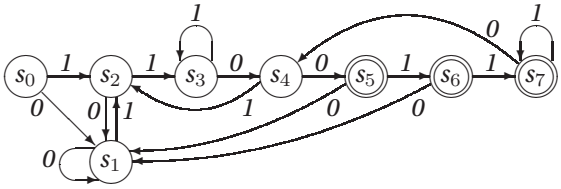
$s_7 \leftarrow \epsilon$-closure($\delta(s_6,1)$) $= \{\mathbf{3}, \mathbf{5}, \mathbf{7}, \mathbf{13}, 15, 14, 0, 2, 19, 4, 6, 8, 12, 17\}$

Round 7:

ϵ-closure($\delta(s_7,0)$) $= \{\mathbf{1}, \mathbf{9}, , \dots\} = s_4$

ϵ-closure($\delta(s_7,1)$) $= \{\mathbf{3}, \mathbf{5}, \mathbf{7}, \mathbf{13}, \dots\} = s_7$

At this point, the worklist is empty. The resulting DFA has eight states.

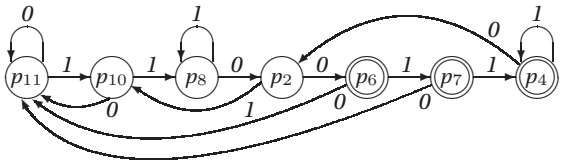


Part (c): Minimize the DFA.

Current Partition	Transition	Action
$\{\ p_f = \{s_5, s_6, s_7\},\ p_1 = \{s_0, s_1, s_2, s_3, s_4\}\ \}$		
	$\delta(p_0,0) = p_1$	*none*
	$\delta(p_0,1) = p_0$	*none*
	$\delta(p_1,0) = \{p_0,p_1\}$	split p_1
$\{\ p_0, p_2 = \{s_4\}, p_3 = \{s_0, s_1, s_2, s_3\}\ \}$		
	$\delta(p_0,0) = \{p_2, p_3\}$	split p_0
$\{\ p_4 = \{s_7\}, p_5 = \{s_5, s_6\}, p_2, p_3\ \}$		
	$\delta(p_5,0) = p_3$	*none*
	$\delta(p_5,1) = \{p_5,p_4\}$	split p_5
$\{p_4, p_6 = \{s_5\}, p_7 = \{s_6\}, p_2, p_3\ \}$		
	$\delta(p_3,0) = \{p_3,p_2\}$	split p_3
$\{p_4, p_6, p_7, p_2\ p_8 = \{s_3\}, p_9 = \{s_0, s_1, s_2\}\ \}$		
	$\delta(p_9,0) = p_9$	*none*
	$\delta(p_9,1) = \{p_9, p_8\}$	split p_9
$\{p_4, p_6, p_7, p_2, p_8\ p_{10} = \{s_2\}, p_{11} = \{s_0, s_1\}\ \}$		
	$\delta(p_{11},0) = p_{11}$	*none*
	$\delta(p_{11},1) = p_{10}$	*none*

Single element sets cannot split. We will omit the transitions for them.

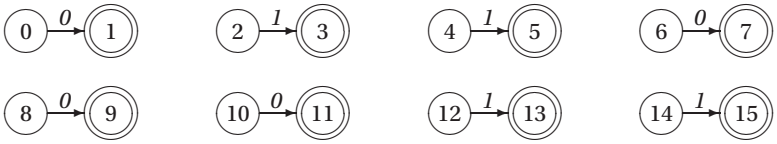
The minimization algorithm found seven distinct states. It combined states s_0 and s_1 in the original DFA. The minimal DFA is



Answer: For the RE $(01 \mid 10 \mid 00)^*\ 11$:

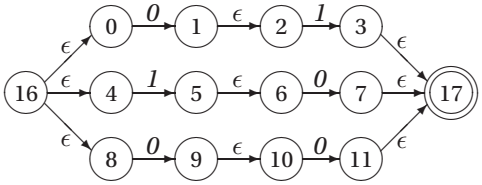
Part (a): Build an NFA for the RE using Thompson's construction.

First, build distinct DFAs for each of the digits:

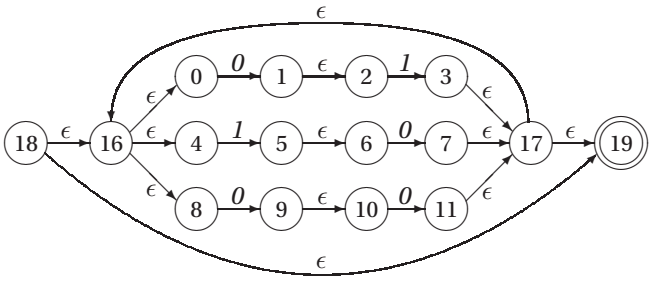


Next, combine the individual DFAs to form NFAs for the subterms 00, 10, 00, and 11:

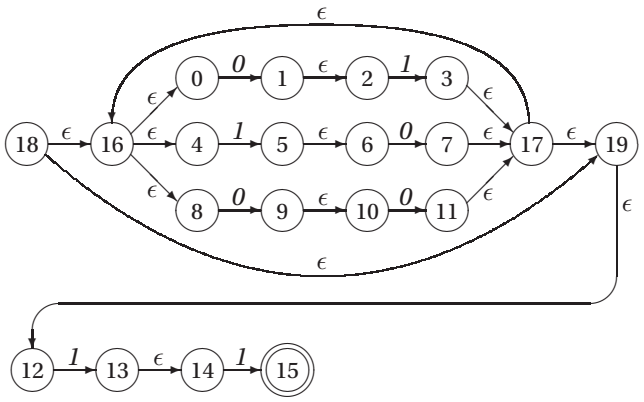
As the third step, combine the NFAs for 01, 10, and 00 to form an NFA for $(01 \mid 10 \mid 00)$



As the fourth step, form the NFA for $(01 \mid 10 \mid 00)^*$:



And, finally, concatenate the NFA for 11 to the end of the NFA for $(01 \mid 10 \mid 00)^*$:



Part (b): Convert the NFA to a DFA using the subset construction.

Initialization:

The initial state is 18.

$s_0 \leftarrow \epsilon\text{-closure}(\mathbf{18}) = \{\mathbf{18}, 16, 0, 4, 8, 19, 12\}$

Round 1:

$s_1 \leftarrow \epsilon\text{-closure}(\delta(s_0,0)) = \{\mathbf{1}, \mathbf{9}, 2, 10\}$

$s_2 \leftarrow \epsilon\text{-closure}(\delta(s_0,1)) = \{\mathbf{5}, \mathbf{13}, 6, 14\}$

Round 2:

$s_3 \leftarrow \epsilon\text{-closure}(\delta(s_1,0)) = \{\mathbf{11}, 17, 16, 0, 4, 8, 19, 12\}$

$s_4 \leftarrow \epsilon\text{-closure}(\delta(s_1,1)) = \{\mathbf{3}, 17, 16, 0, 4, 8, 19, 12\}$

$s_5 \leftarrow \epsilon\text{-closure}(\delta(s_2,0)) = \{\mathbf{7}, 17, 16, 0, 4, 8, 19, 12\}$

$s_6 \leftarrow \epsilon\text{-closure}(\delta(s_2,1)) = \{\mathbf{15}\}$

Round 3:

$\epsilon\text{-closure}(\delta(s_3,0)) = \{\mathbf{1}, \mathbf{9}, \ldots\} = s_1$

$\epsilon\text{-closure}(\delta(s_3,1)) = \{\mathbf{5}, \mathbf{13}, \ldots\} = s_2$

$\epsilon\text{-closure}(\delta(s_4,0)) = \{\mathbf{1}, \mathbf{9}, \ldots\} = s_1$

$\epsilon\text{-closure}(\delta(s_4,1)) = \{\mathbf{5}, \mathbf{13}, \ldots\} = s_2$

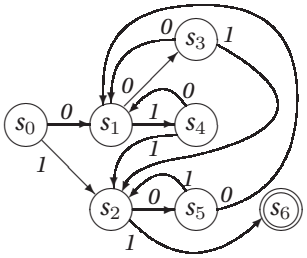
$\epsilon\text{-closure}(\delta(s_5,0)) = \{\mathbf{1}, \mathbf{9}, \ldots\} = s_1$

$\epsilon\text{-closure}(\delta(s_5,1)) = \{\mathbf{5}, \mathbf{13}, \ldots\} = s_2$

$\epsilon\text{-closure}(\delta(s_6,0)) = \emptyset$

ϵ-closure($\delta(s_6, 1)$) = $\emptyset$

At this point, the worklist is empty. The resulting DFA has seven states.

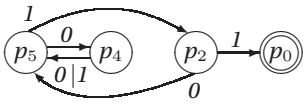


Part (c): Minimize the DFA.

Single element sets cannot split. We will omit the transitions for them.

Current Partition	Transition	Action
$\{ p_0 = \{s_6\}, p_1 = \{s_0, s_1, s_2, s_3, s_4, s_5\} \}$		
	$\delta(p_1, 0) = p_1$	*none*
	$\delta(p_1, 1) = \{p_0, p_1\}$	split p_1
$\{ p_0, p_2 = \{s_2\}, p_3 = \{s_0, s_1, s_3, s_4, s_5\} \}$		
	$\delta(p_3, 0) = p_3$	*none*
	$\delta(p_3, 1) = \{p_2, p_3\}$	split p_3
$\{ p_0, p_2, p_4 = \{s_1\}\ p_5 = \{s_0, s_3, s_4, s_5\} \}$		
	$\delta(p_5, 0) = p_4$	*none*
	$\delta(p_5, 1) = p_2$	*none*

The resulting DFA has four states.



8. One way of proving that two REs are equivalent is to construct their minimized DFAs and then to compare them. If they differ only by state names, then the REs are equivalent. Use this technique to check the following pairs of REs and state whether or not they are equivalent.

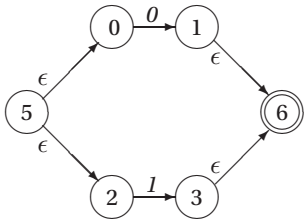
   a. $(0 \mid 1)^*$ and $(0^* \mid 10^*)^*$

   b. $(ba)^+ \ (a^* \ b^* \mid a^*)$ and $(ba)^* \ ba^+ \ (b^* \mid \epsilon)$

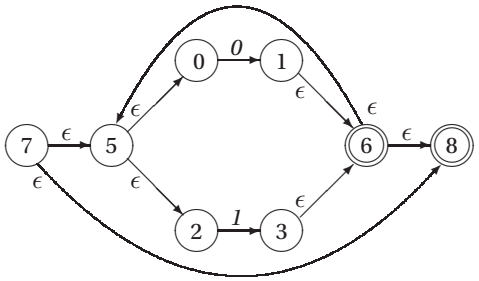
Answer: For $(0 \mid 1)^*$ and $(0^* \mid 10^*)^*$

First, build a DFA for $(0 \mid 1)^*$. We start with NFAs for 0 and 1

Next, we connect them to form an NFA for $0 \mid 1$.



Next, we add the states and edges for the closure, to form an NFA for $(0 \mid 1)^*$.



Now, we convert the NFA to a DFA with the subset construction, using the same notation as in the solution to Problem 2.8.

Initialization:

$$s_0 \leftarrow \epsilon\text{-closure}(7) = \{\mathbf{7}, 5, 0, 2, 8\}$$

Round 1:

$$s_1 \leftarrow \epsilon\text{-closure}(\delta(s_0, 0)) = \{\mathbf{1}, 6, 5, 0, 2, 8\}$$

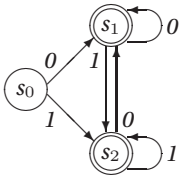
$s_2 \leftarrow \epsilon\text{-closure}(\delta(s_0,1a)) = \{\mathbf{3}, 6, 5, 0, 2, 8\}$

Round 2:

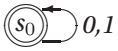
$\epsilon\text{-closure}(\delta(s_1,0)) = \{\mathbf{1}, \dots\} = s_1$
$\epsilon\text{-closure}(\delta(s_1,1)) = \{\mathbf{3}, \dots\} = s_2$

$\epsilon\text{-closure}(\delta(s_2,0)) = \{\mathbf{1}, \dots\} = s_1$
$\epsilon\text{-closure}(\delta(s_2,1)) = \{\mathbf{3}, \dots\} = s_2$

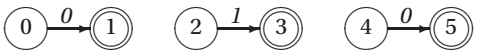
At this point the worklist is empty. The DFA has three states.



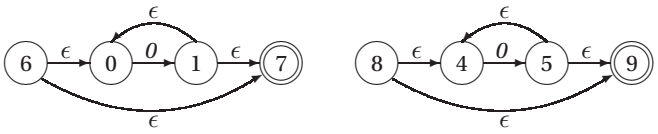
DFA minimization produces a single-state DFA, which a good compiler writer might have drawn from inspection.



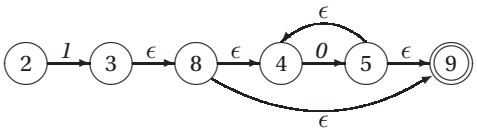
As the second step, we build the NFA for $(0^* \mid 10^*)^*$. We start with NFAs for 0, 1, and 0



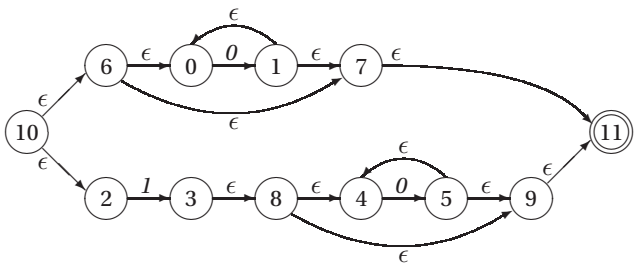
Next, we build the NFAs for 0^* and 0^*.



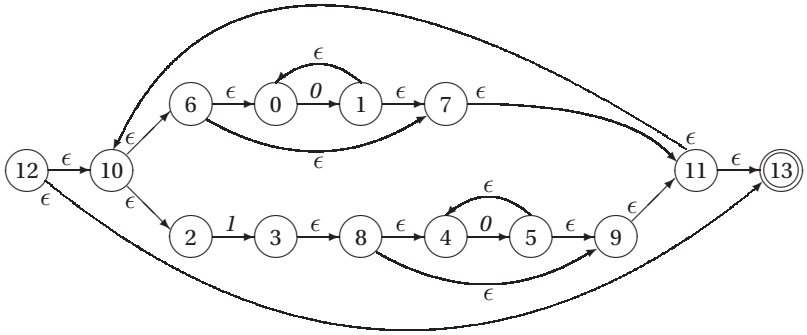
Now, we concatenate the NFAs for 1 and 0^* to form an NFA for 10^*.



Then, we form the NFA for $0^* \mid 10^*$.



And, finally, add the states to the NFA that implement the outermost closure.



This NFA is rather large, with lots of ϵ transitions. First, we apply the subset construction to find the equivalent DFA. Then, we will minimize the DFA.

Initialization:

$$s_0 = \epsilon\text{-closure}(12) = \{\mathbf{12}, 10, 6, 0, 7, 11, 13, 2\}$$

Round 1:

$$s_1 = \epsilon\text{-closure}(\delta(s_0,0)) = \{\mathbf{1}, 7, 11, 10, 6, 0, 2, 13\}$$
$$s_2 = \epsilon\text{-closure}(\delta(s_0,1)) = \{\mathbf{3}, 8, 4, 9, 11, 10, 6, 0, 2, 13\}$$

Round 2:

$$\epsilon\text{-closure}(\delta(s_1,0)) = \{\mathbf{1}, \dots\} = s_1$$
$$\epsilon\text{-closure}(\delta(s_1,1)) = \{\mathbf{3}, \dots\} = s_2$$

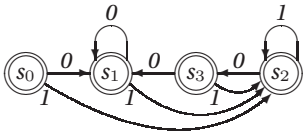
$$s_3 = \epsilon\text{-closure}(\delta(s_2,0)) = \{\mathbf{1}, \mathbf{5}, 7, 11, 10, 6, 0, 2, 9, 13\}$$
$$\epsilon\text{-closure}(\delta(s_2,1)) = \{\mathbf{3}, \dots\} = s_2$$

Round 3:

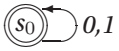
$$\epsilon\text{-closure}(\delta(s_3,0)) = \{\mathbf{1}, \dots\} = s_1$$

ϵ-closure($\delta(s_3,1)$) $= \{\mathbf{3}, \dots\} = s_2$

At this point, the worklist is empty. The resulting DFA has four states.

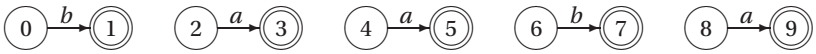


Minimization discovers that states s_0, s_1, s_2, and s_3 form an equivalence class: each state is a final state and each has a transition back into the class on *0* and on *1*. Thus, the minimized DFA has one state:

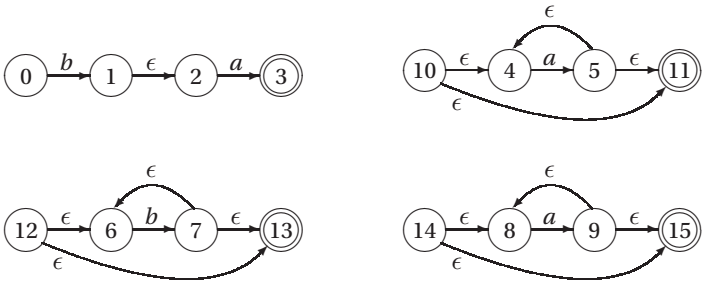


Answer: For $(ba)^+\ (a^*\ b^* \mid a^*)$ and $(ba)^*\ ba^+\ (b^* \mid \epsilon)$

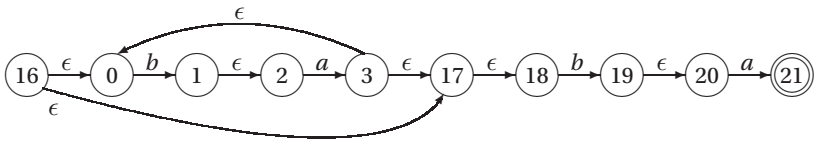
First, we build the NFA for $(ba)^+\ (a^*\ b^* \mid a^*)$. We start with individual NFAs for each of the letters.



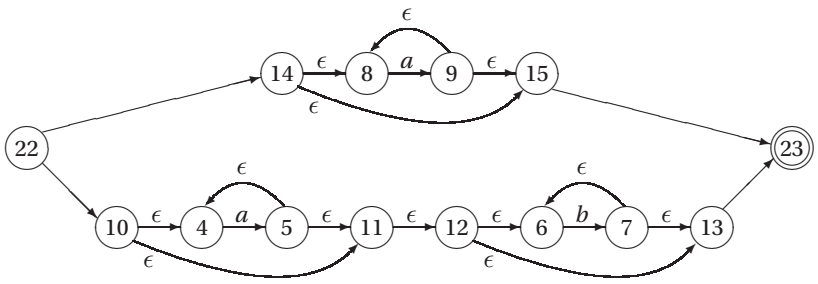
Next, we build NFAs for ba, a^*, b^*, and a^*.



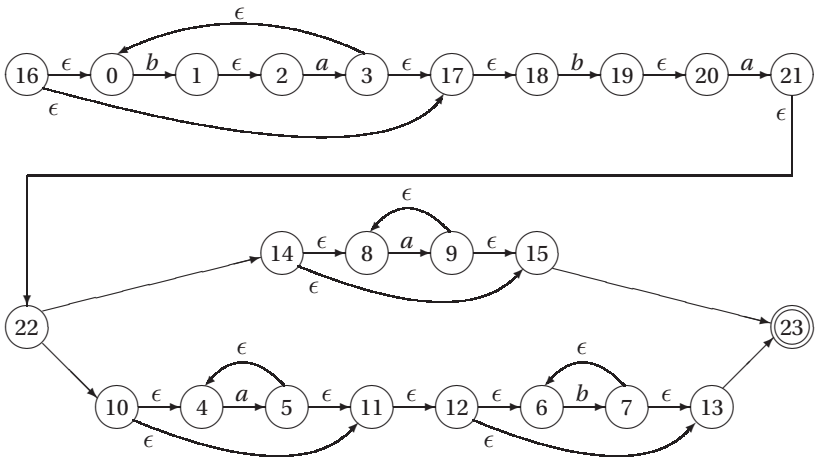
Thompson's construction does not give us a form for $(ba)^+$. We can either model it as $(ba)^*\ ba$, or we can build a more concise form by adding a single ϵ transition from state 0 to state 3. Since these solutions are intended to be explanatory, we will choose the former approach. Thus, we next produce NFAs for $(ba)^*\ ba$ and for $a^*\ b^*$



Next is the NFA for $(a^*\ b^* \mid a^*)$.



Finally, we tie together the NFAs for $(ba)^*ba$ and $(a^*\ b^* \mid a^*)$ by adding an ϵ transition from state 21 to state 22, to produce the final NFA.



The next step is to use the subset construction to convert the NFA to a DFA.

Initialization

The initial state is 16.

$s_0 \leftarrow \epsilon\text{-closure}(16) = \{\mathbf{16}, 0, 17, 18\}$

Round 1:

$\epsilon\text{-closure}(\delta(s_0,a)) = \emptyset$

$s_1 \leftarrow \epsilon\text{-closure}(\delta(s_0,b)) = \{\mathbf{1}, \mathbf{19}, 2, 20\}$

Round 2:

$s_2 \leftarrow \epsilon\text{-closure}(\delta(s_1,a)) = \{\mathbf{3}, \mathbf{21}, 0, 17, 18, 22, 14, 8, 15, 23, 10, 4, 11, 12, 6, 13\}$

$\epsilon\text{-closure}(\delta(s_1,b) = \emptyset$

Round 3:

$s_3 \leftarrow \epsilon\text{-closure}(\delta(s_2,a)) = \{\mathbf{9}, \mathbf{5}, 8, 15, 23, 4, 11, 12, 6, 13\}$

$s_4 \leftarrow \epsilon\text{-closure}(\delta(s_2,b)) = \{\mathbf{1}, \mathbf{19}, \mathbf{7}, 2, 20, 6, 13, 23\}$

Round 4:

$\epsilon\text{-closure}(\delta(s_3,a)) = \{\mathbf{9}, \mathbf{5}, 8, 15, 23, 4, 11, 12, 6, 13\} = s_3$

$s_5 \leftarrow \epsilon\text{-closure}(\delta(s_3,b)) = \{\mathbf{7}, 6, 13, 23\}$

$\epsilon\text{-closure}(\delta(s_4,a)) = \{\mathbf{3}, \mathbf{21}, 0, 17, 18, 22, 14, 8, 15, 23, 10, 4, 11, 12, 6, 13\}$
$= s_2$

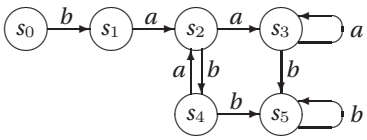
$\epsilon\text{-closure}(\delta(s_4,b)) = \{\mathbf{7}, 6, 13, 23\} = s_5$

Round 5:

$\epsilon\text{-closure}(\delta(s_5,a)) = \emptyset$

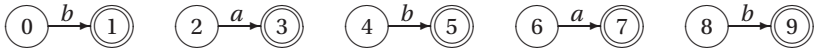
$\epsilon\text{-closure}(\delta(s_5,b)) = \{\mathbf{7}, 6, 13, 23\} = s_5$

The resulting DFA is:

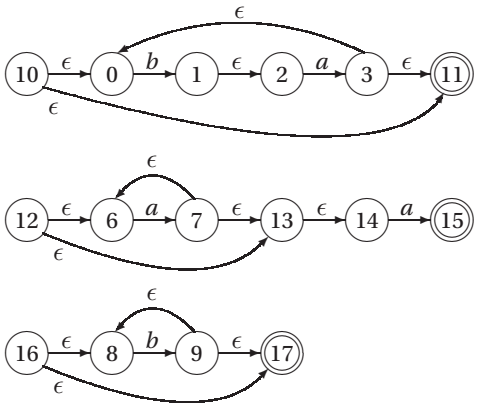


We will defer minimization, another intricate step, until we see the DFA for the other RE.

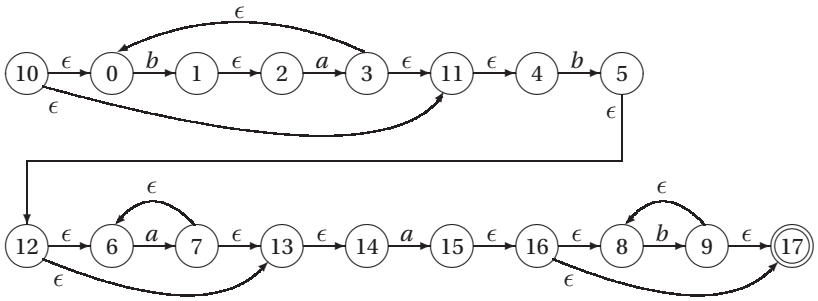
Next, we build the NFA for $(ba)^*ba^+(b* \mid \epsilon)$. We will make one simplification, noting that the term $(b^* \mid \epsilon)$ is equivalent to b^*. We begin by building individual NFAs for each letter



Next, we build NFAs for the terms $(ba)^*$, a^+, and b^*. (We skipped building ba. It occurs as shown for the first RE in this question. We will treat a^+ as $a^*\ a$.)



Finally, we tie these three NFAs together, along with the NFA for *b* (states 4 and 5 earlier) to form the final NFA



Next, we convert the NFA to a DFA with the subset construction.

Initialization

The initial state is 10.

$s_0 \leftarrow \epsilon\text{-closure}(10) = \{\mathbf{10}, 0, 11, 4\}$

Round 1

$\epsilon\text{-closure}(\delta(s_0,a)) = \emptyset$

$s_1 = \epsilon\text{-closure}(\delta(s_0,a)) = \{\mathbf{1}, \mathbf{5}, 2, 12, 6, 13, 14\}$

Round 2

$s_2 \leftarrow \epsilon\text{-closure}(\delta(s_1,a)) = \{\mathbf{3}, \mathbf{7}, \mathbf{15}, 0, 11, 4, 6, 13, 14, 16, 8, 17\}$

$\epsilon\text{-closure}(\delta(s_1,b)) = \emptyset$

Round 3

$s_3 \leftarrow \epsilon\text{-closure}(\delta(s_2,a)) = \{\mathbf{7}, \mathbf{15}, 6, 13, 14, 16, 8, 17\}$

$s_4 \leftarrow \epsilon\text{-closure}(\delta(s_2,b)) = \{\mathbf{1}, \mathbf{5}, \mathbf{9}, 2, 12, 6, 13, 14, 8, 17\}$

Round 4

$\epsilon\text{-closure}(\delta(s_3,a)) = \{\mathbf{7}, \mathbf{15}, 6, 13, 14, 16, 8, 17\} = s_3$

$s_5 \leftarrow \epsilon\text{-closure}(\delta(s_3,b)) = \{\mathbf{9}, 8, 17\}$

$\epsilon\text{-closure}(\delta(s_4,a)) = \{\mathbf{3}, \mathbf{7}, \mathbf{15}, 0, 11, 4, 6, 13, 14, 16, 8, 17\} = s_2$

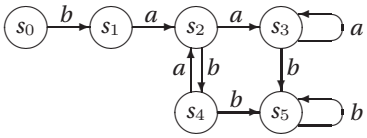
$\epsilon\text{-closure}(\delta(s_4,b)) = \{\mathbf{9}, 8, 17\} = s_5$

Round 5

$\epsilon\text{-closure}(\delta(s_5,a)) = \emptyset$

$\epsilon\text{-closure}(\delta(s_5,b)) = \{\mathbf{9}, 8, 17\} = s_5$

The resulting DFA is:



Notice that this DFA is identical to the DFA for the other RE. Thus, their minimized forms will be identical, within a renaming of states and we have shown that the two REs describe the same language.

9. In some cases, two states connected by an ϵ-move can be combined.

   a. Under what set of conditions can two states connected by an ϵ-move be combined?

      Answer: Two states $s_i \xrightarrow{\epsilon} s_j$ can be combined if the ϵ-move is the only transition that enters s_j and the transitions leaving s_i and s_j, other than $s_i \xrightarrow{\epsilon} s_j$ are compatible. That is, if $s_i \xrightarrow{c} s_k$, then either s_k has no transition on c or $s_j \xrightarrow{c} s_k$.

      These conditions hold for the ϵ transitions in the initial NFAs formed in Thompson's construction.

   b. Give an algorithm for eliminating ϵ-moves.

      Answer: The following algorithm removes ϵ transitions of the kind found in NFAs created by Thompson's construction.

```
for each edge e ∈ E
    if e = ⟨q_i, q_j, ϵ⟩ then
        add e to WorkList
while (WorkList ≠ ∅)
    remove e = ⟨q_i, q_j, α⟩ from WorkList
    if α = ϵ then
        for each ⟨q_j, q_k, β⟩ in E
            add ⟨q_i, q_k, β⟩ to E
            if β = ϵ then
                add ⟨q_i, q_k, β⟩ to WorkList
        delete ⟨q_i, q_j, ϵ⟩ from E
        if q_j ∈ F then
            add q_i to F
for each state q_i ∈ N
    if q_i has no entering edge then
        delete q_i from N
```

      If $s_i \xrightarrow{\epsilon} s_j$, the algorithm copies transitions that leave s_j to s_i to create an equivalent path through the transition graph that avoids the ϵ-move. After all such transitions have been copied, it deletes the ϵ-move. A final pass over the transition graph eliminates any unreachable states – states that were reachable only with an ϵ-move will be removed in this pass.

c. How does your algorithm relate to the ϵ-closure function used to implement the subset construction?

Answer: Using the ϵ-closure function, the algorithm could directly discover the set of outbound transitions needed at each node. In some sense, the algorithm that we gave for part (b) computes ϵ-closure(s_i) on the fly, as it copies the edges.

An algorithm that used ϵ-closure directly would still need to create all the new edges and make a final pass to delete unreachable states.

10. Show that the set of regular languages is closed under intersection.

Answer: We can prove that regular languages are closed under union and under complement.

(Closure under union, sketch of proof: Take the DFAs for L_1 and L_2, add a new start state with ϵ-moves to the start states of the DFAs for L_1 and L_2. This new NFA accepts a string if it is accepted by the DFA for L_1 or if it is accepted by the DFA for L_2, so it accepts $L_1 \cup L_2$.)

(Closure under complement, sketch of proof: Take the DFA for L_1. Make all non-final states in the original DFA be final states of a new DFA and all the final states of the original DFA be non-final states in the new DFA. Now, the new DFA accepts any string except those that the original DFA would accept, so it accepts $\overline{L_1}$.)

Now, we know from DeMorgan's laws that $L_1 \cap L_2$ is equivalent to $\overline{\overline{L_1} \cup \overline{L_2}}$. Thus, we can compute $L_1 \cap L_2$ by finding the regular languages that are the complements of L_1 and L_2, finding the language that is the union of those new languages, and taking its complement. At each step, the languages are regular because of closure under complement and union, so the resulting language is regular. This procedure guarantees a regular language from $L_1 \cap L_2$, so the set of regular languages is closed under intersection.

11. The DFA minimization algorithm given in Figure 2.9 is formulated to enumerate all the elements of P and all of the characters is Σ on each iteration of the while loop.

    a. Recast the algorithm so that it uses a worklist to hold the sets that must still be examined.

    Answer:

```
Worklist ← { FinalStates, States - FinalStates }
Partition ← { FinalStates, States - FinalStates }

while (Worklist ≠ ∅) do
    remove a set w from Worklist
    temp ← Split(w)
    if temp ≠ w then                    // Split() refined w
        remove w from Partition         // Update Partition
        Partition ← Partition ∪ temp
        Worklist ← Worklist ∪ temp      // Update Worklist
```

    b. Recast the *Split* function so that it partitions the set around all of the characters in Σ.

    Answer:

```
Split(S)  {
    Result ← { S }
    for each c ∈ Σ do {
        for each s ∈ Result do {
            T ← ∅
            if c splits s into s1 and s2 then
                remove s from Result
                T ← T ∪ { s1, s2 }
        }
        Result ← Result ∪ T
    }
    return Result
}
```

c. How does the expected case complexity of your modified algorithm compare to the expected case complexity of the original algorithm?

Answer: The total number of splits that the algorithm can perform is bounded by the number of states. Each state is in exactly one partition, so the number of splits cannot exceed the number of states.

However, the algorithm in Figure 2.19 can examine a set many more times that the modified algorithm. The original algorithm tries to split every set in every partition on each iteration of the outer while loop, so it does something like $\mathbf{O}(S^2)$ calls to *Split*, where S is the number of states in the minimal DFA.

Ignore, for the moment, the modifications to *Split* in part (b) of the question. The worklist version of the algorithm, in part (a), calls *Split* once on each set that appears in *Worklist*. How many sets appear in *Worklist*? We know that every state in the minimal DFA appears in *Worklist*. Those states are either one of the two initial sets, or are split from some other set. If you think of the splitting as a binary tree that has S leaves, you can see that *Worklist* has, potentially, $\mathbf{O}(S^2)$ distinct entries over time.

Are the two algorithms the same? No. The modified algorithm considers $\frac{1}{2}S^2$ sets where the original algorithm considered S^2 sets. In practice, that savings is worthwhile.

What about the modifications to *Split*? While the modifications would reduce the number of calls to *Split*, they would not reduce the total number of sets that the algorithm touches.

Section 2.5

12. Construct a DFA for each of the following C language constructs, and then build the corresponding table for a table-driven implementation for each of them:

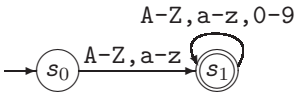
a. Integer constants

Answer:

δ	*0–9*	*other*
s_0	s_0	s_e
s_e	s_e	s_e

b. Identifiers

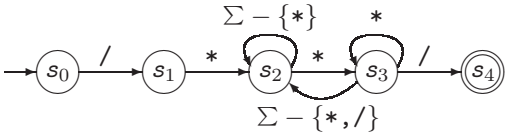
Answer:



δ	*A-Z, a-z*	*0-9*	*other*
s_0	s_1	s_e	s_e
s_1	s_1	s_1	s_e
s_e	s_e	s_e	s_e

c. Comments

Answer:



δ	/	*	*other*
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_2	s_3	s_2
s_3	s_4	s_3	s_2
s_4	s_e	s_e	s_e
s_e	s_e	s_e	s_e

Note: the transitions for s_4 reflect the fact that the DFA only accepts comments. It does not accept any text following the comment.

13. For each of the DFAs from the previous exercise, build the corresponding direct-coded scanner.

Answer:

```
    goto s0
s0: char ← NextCharacter
    if ('0' ≤ char ≤ '9')
      then goto s0
      else if char = eof
        then report acceptance
        else goto se
se: print error message
    report failure
```

(a) Integer Constants

```
    goto s0
s0: char ← NextCharacter
    if (char = '_') or
      ('a' ≤ char ≤ 'z') or
      ('A' ≤ char ≤ 'Z')
      then goto s1
      else goto se
s1: char ← NextCharacter
    if (char = '_') or
      ('a' ≤ char ≤ 'z') or
      ('A' ≤ char ≤ 'Z') or
      ('0' ≤ char ≤ '9')
      then goto s1
      else if char = eof
        then report acceptance
        else goto se
se: print error message
    report failure
```

(b) Identifiers

```
goto s0
s0: char ← NextCharacter
    if (char = '/')
      then goto s1
      else goto se
s1: char ← NextCharacter
    if (char = '*')
      then goto s2
      else goto se
s2: char ← NextCharacter
    if (char = '*')
      then goto s3
      else goto s2
s3: char ← NextCharacter
    if (char = '*')
      then goto s3
      else if (char = '/')
        then goto s4
        else goto s2
s4: report acceptance
se: print error message
    report failure
```

(c) Comments in C

14. This chapter described two styles of DFA implementations: a table-driven implementation that uses a case or switch statement, and a direct-coded scanner that uses branches and jumps. A third alternative is to use mutually recursive functions to implement a scanner. Discuss the advantages and disadvantages of such an implementation.

Answer: The primary advantage of implementing a DFA as a set of mutually recursive is that the implementation is conceptually simple.

The primary disadvantage is efficiency. Each procedure call must, at least, include a transfer of control that is equivalent to the one that occurs in the direct-coded scanner. In most implementations, the call will also allocate an activation record (see Chapter 6) and establish bindings for parameters. All of these make the cost per character higher than in the direct-coded scanner. (Space costs for activation records will be $\mathbf{O}(n)$ where n is the number of characters. The extra overhead from parameter binding and such will be constant per call, with one call for each character.)

15. To reduce the size of the transition table, the scanner generator can use a character classification scheme. Generating the classifier table, however, seems expensive. The obvious algorithm would require $\mathbf{O}(|\Sigma|^2 \cdot |states|)$ time. Derive an asymptotically faster algorithm for finding identical columns in the transition table.

Answer: Many solutions are possible to this problem. One simple solution is to perform a radix sort, state by state, on the columns, from s_0 through s_e. Characters whose columns end up in the same bucket belong in the same character class. The complexity of this approach should be $\mathbf{O}(|\Sigma| \cdot |states|)$, a factor of $|\Sigma|$ faster than the naive algorithm. Many other approaches will work.

16. Figure 2.15 shows a scheme that avoids quadratic roll back behavior in a scanner built by simulating a DFA. Unfortunately, that scheme requires that the scanner know in advance the length of the input stream and that it maintain a bit-matrix, *Failed*, of size $|states| \times |input|$. Devise a scheme that avoids the need to know the size of the input stream in advance. Can you use the same scheme to reduce the size of the *Failed* table in cases where the worst case input does not occur?

    Answer: In fact, this question is worded incorrectly. We do not know of a scheme that avoids, in general, the need to know the length of the input in the worst case. We can reduce the space needed by the worst case syntax by only storing every k^{th} state; that reduces the needed space by a factor of k while increasing the cost of scanning—the scanner may repeat k states before detecting that it is repeating a failed path in the DFA.

    See the paper "'Maximal-munch' tokenization in linear time" by Thomas Reps, in ACM Transactions on Programming Languages and Systems (TOPLAS), 20(2), March 1998, pages 259–273.

CHAPTER 3

Section 3.2

1. Write a context-free grammar for the syntax of regular expressions.

   Answer:

   $$\begin{array}{rcl} \mathrm{R} & \rightarrow & \mathrm{R\,R} \\ & | & \mathrm{R}^{*} \\ & | & (\,\mathrm{R}\,) \\ & | & \mathrm{R} \mid \mathrm{R} \\ & | & \texttt{char} \end{array}$$

   where `char` represents a single character

2. Write a context-free grammar for the Backus-Naur Form (BNF) notation for context-free grammars.

   Answer:

   $$\begin{array}{rcl} \textit{BNF} & \rightarrow & \textit{LHSList} \\ \textit{LHSList} & \rightarrow & \textit{Production}\ \ \textit{LHSList} \\ & | & \textit{Production} \\ \textit{Production} & \rightarrow & \texttt{NonTerm Derives}\ \textit{RHSList} \\ \textit{RHSList} & \rightarrow & \textit{RHS}\ \ \texttt{Also}\ \ \textit{RHSList} \\ & | & \textit{RHS} \\ \textit{RHS} & \rightarrow & \texttt{NonTerm}\ \ \textit{RHS} \\ & | & \texttt{Term}\ \ \textit{RHS} \\ & | & \texttt{NonTerm} \\ & | & \texttt{Term} \end{array}$$

   where `NonTerm` is a nonterminal symbol, `Term` is a terminal symbol, `Derives` represents '$\rightarrow$', and `Also` represents '|'.

3. When asked about the definition of an *unambiguous context-free grammar* on an exam, two students gave different answers. The first defined it as "a grammar where each sentence has a unique syntax tree by leftmost derivation." The second defined it as "a grammar where each sentence has a unique syntax tree by any derivation." Which one is correct?

   Answer: The two definitions are equivalent, as can be proved by induction. The basic idea is to show that any syntax tree generated by any derivation can be generated by a left-most derivation.

Section 3.3

4. The following grammar is not suitable for a top-down predictive parser. Identify the problem and correct it by rewriting the grammar. Show that your new grammar satisfies the LL(1) condition.

$$\begin{array}{lcl} L & \to & R\,\mathtt{a} \\ & | & Q\,\mathtt{ba} \end{array} \qquad \begin{array}{lcl} R & \to & \mathtt{aba} \\ & | & \mathtt{caba} \\ & | & R\,\mathtt{bc} \end{array} \qquad \begin{array}{lcl} Q & \to & \mathtt{bbc} \\ & | & \mathtt{bc} \end{array}$$

Answer: The grammar, as written, contains a left-recursion on the non-terminal symbol R. We must rewrite to remove that left recursion. The productions

$$\begin{array}{lcl} R & \to & \mathtt{aba} \\ & | & \mathtt{caba} \\ & | & R\,\mathtt{bc} \end{array} \quad \text{can be rewritten as} \quad \begin{array}{lcl} R & \to & \mathtt{aba}S \\ & | & \mathtt{caba}S \end{array} \qquad \begin{array}{lcl} S & \to & \mathtt{bc}S \\ & | & \epsilon \end{array}$$

At that point, the remaining problem lies in the productions for Q, which both begin with b. Left factoring the productions will cure that problem:

$$\begin{array}{lcl} Q & \to & \mathtt{bbc} \\ & | & \mathtt{bc} \end{array} \quad \text{can be rewritten as} \quad \begin{array}{lcl} Q & \to & \mathtt{b}T \\ T & \to & \mathtt{bc} \\ & | & \mathtt{c} \end{array}$$

To show that the resulting grammar is LL(1), we need to compute, for each non-terminal symbol, the FIRST set of each alternative right-hand side. The grammar is, after the transformations:

$$\begin{array}{lcl} L & \to & R\,\mathtt{a} \\ & | & Q\,\mathtt{ba} \end{array} \qquad \begin{array}{lcl} R & \to & \mathtt{aba}S \\ & | & \mathtt{caba}S \end{array} \qquad \begin{array}{lcl} S & \to & \mathtt{bc}S \\ & | & \epsilon \end{array} \qquad \begin{array}{lcl} Q & \to & \mathtt{b}T \end{array} \qquad \begin{array}{lcl} T & \to & \mathtt{bc} \\ & | & \mathtt{c} \end{array}$$

To show that this grammar is LL(1), we must show that the FIRST sets of the alternative right hand sides for each non-terminal symbol are disjoint:

L:	$\text{FIRST}(R\mathtt{a}) = \{\mathtt{a},\mathtt{c}\}$	$\text{FIRST}(Q\mathtt{ba}) = \{\mathtt{b}\}$	$\{\mathtt{a},\mathtt{c}\} \cap \{\mathtt{b}\} = \emptyset$
R:	$\text{FIRST}(\mathtt{aba}S) = \{\mathtt{a}\}$	$\text{FIRST}(\mathtt{caba}S) = \{\mathtt{c}\}$	$\{\mathtt{a}\} \cap \{\mathtt{c}\} = \emptyset$
S:	$\text{FIRST}(\mathtt{bc}S) = \{\mathtt{b}\}$	$\text{FIRST}(\epsilon) = \{\mathtt{a}\}$	$\{\mathtt{b}\} \cap \{\mathtt{a}\} = \emptyset$
T:	$\text{FIRST}(\mathtt{bc}) = \{\mathtt{b}\}$	$\text{FIRST}(\mathtt{c}) = \{\mathtt{c}\}$	$\{\mathtt{b}\} \cap \{\mathtt{c}\} = \emptyset$

5. Consider the following grammar:

$$\begin{array}{lcl} A & \to & B\,\mathtt{a} \\ B & \to & \mathtt{dab} \\ & \mid & C\,\mathtt{b} \end{array} \qquad \begin{array}{lcl} C & \to & \mathtt{c}\,B \\ & \mid & A\,\mathtt{c} \end{array}$$

Does this grammar satisfy the LL(1) condition? Justify your answer. If it does not, rewrite it as an LL(1) grammar for the same language.

Answer: The grammar has an indirect left recursion from the productions $A \to B\mathtt{a}$, $B \to C\mathtt{b}$, and $C \to A\mathtt{c}$.

The algorithm for removal of indirect left recursion, in Figure 3.6, cures this problem. Conceptually, it operates in two steps. First, it converts the indirect left recursion to a direct left recursion:

$$\begin{array}{lcl} A & \to & \mathtt{daba} \\ & \mid & \mathtt{c}B\mathtt{ba} \\ & \mid & A\mathtt{cba} \end{array} \qquad \begin{array}{lcl} B & \to & \mathtt{dab} \\ & \mid & C\mathtt{b} \end{array} \qquad \begin{array}{lcl} C & \to & \mathtt{c}B \\ & \mid & A\mathtt{c} \end{array}$$

Second, it eliminates the direct left recursion on A by rewriting the productions for A as follows:

$$\begin{array}{lcl} A & \to & \mathtt{daba}D \\ & \mid & \mathtt{c}B\mathtt{ba}D \end{array} \qquad \begin{array}{lcl} D & \to & \mathtt{cba}D \\ & \mid & \epsilon \end{array}$$

The algorithm, as written, interleaves these steps. Combining them produces the following grammar.

$$\begin{array}{lcl} A & \to & \mathtt{daba}D \\ & \mid & \mathtt{c}B\mathtt{ba}D \end{array} \qquad \begin{array}{lcl} B & \to & \mathtt{dab} \\ & \mid & C\mathtt{b} \end{array} \qquad \begin{array}{lcl} C & \to & \mathtt{c}B \\ & \mid & A\mathtt{c} \end{array} \qquad \begin{array}{lcl} D & \to & \mathtt{cba}D \\ & \mid & \epsilon \end{array}$$

The alternative right-hand sides for A and D have disjoint FIRST sets. Both B and C, however, have problems in this regard.

B: $\text{FIRST}(\mathtt{dab}) = \{\mathtt{d}\}$ $\text{FIRST}(C\,\mathtt{b}) = \{\mathtt{c},\mathtt{d}\}$ $\{\mathtt{d}\} \cap \{\mathtt{c},\mathtt{d}\} = \{\mathtt{d}\}$

C: $\text{FIRST}(\mathtt{c}\,B) = \{\mathtt{c}\}$ $\text{FIRST}(A\,\mathtt{c}) = \{\mathtt{d},\mathtt{c}\}$ $\{\mathtt{c}\} \cap \{\mathtt{d},\mathtt{c}\} = \{\mathtt{c}\}$

Left factoring can cure this problem. To see this point, substitute the two right-hand sides of A into the right-hand side for C, then substitute the right-hand sides of C into the right-hand side for B, to yield:

$$
\begin{array}{rcl}
B & \rightarrow & \texttt{dab} \\
& | & \texttt{c}\,B \\
& | & \texttt{daba}\,D\,\texttt{c} \\
& | & \texttt{c}\,B\,\texttt{ba}\,D\,\texttt{c}
\end{array}
$$

This grammar fragment has two right-hand sides that begin with d and two that begin with c. In each case, we can introduce a new non-terminal symbol for the common prefix, to yield:

$$
\begin{array}{rcl}
B & \rightarrow & \texttt{c}\,B\,E \\
& | & \texttt{dab}\,F
\end{array}
\qquad
\begin{array}{rcl}
E & \rightarrow & \texttt{ba}\,D\,\texttt{c} \\
& | & \epsilon
\end{array}
\qquad
\begin{array}{rcl}
F & \rightarrow & \texttt{a}\,D\,\texttt{c} \\
& | & \epsilon
\end{array}
$$

Combining these new fragments with the remaining fragments of the grammar produces the following final grammar.

$$
\begin{array}{rcl}
A & \rightarrow & \texttt{daba}D \\
& | & \texttt{c}B\texttt{ba}D
\end{array}
\quad
\begin{array}{rcl}
B & \rightarrow & \texttt{c}\,B\,E \\
& | & \texttt{dab}\,F
\end{array}
\quad
\begin{array}{rcl}
D & \rightarrow & \texttt{cba}D \\
& | & \epsilon
\end{array}
\quad
\begin{array}{rcl}
E & \rightarrow & \texttt{ba}\,D\,\texttt{c} \\
& | & \epsilon
\end{array}
\quad
\begin{array}{rcl}
F & \rightarrow & \texttt{a}\,D\,\texttt{c} \\
& | & \epsilon
\end{array}
$$

Inspecting the alternative right-hand sides for each non-terminal symbol shows that they have distinct FIRST and FIRST^+ sets.

6. Grammars that can be parsed top-down, in a linear scan from left to right, with a k word lookahead are called LL(k) grammars. In the text, the LL(1) condition is described in terms of FIRST sets. How would you define the FIRST sets necessary to describe an LL(k) condition?

   Answer: In an llone grammar, the parser needs to look just one word ahead, so the FIRST sets have elements that are terminal symbols of the grammar. For an LL(k) grammar, the corresponding FIRST sets, called FIRST_k sets would need to contain elements representing combinations of up to k terminal symbols. Thus, $\text{FIRST}_k(\alpha)$ would be a subset of the set of all combinations of terminals that form words of length $\leq k$.

7. Suppose an elevator is controlled by two commands: $\uparrow$ to move the elevator up one floor and $\downarrow$ to move the elevator down one floor. Assume that the building is arbitrarily tall and that the elevator starts at floor x.

   Write an LL(1) grammar that generates arbitrary command sequences that (1) never cause the elevator to go below floor x and (2) always return the elevator to floor x at the end of the sequence. For example, $\uparrow\uparrow\downarrow\downarrow$ and $\uparrow\downarrow\uparrow\downarrow$ are

valid command sequences, but ↑↓↓↑ and ↑↓↓ are not. For convenience, you may consider a null sequence as valid. Prove that your grammar is LL(1).

Answer: This problem is analogous to the parenthesis matching problem. Thus, one solution is

$$\begin{array}{lcl} E & \rightarrow & \uparrow E \downarrow E \\ & | & \epsilon \end{array}$$

The grammar is trivially LL(1)

Section 3.4

8. Top-down and bottom-up parsers build syntax trees in different orders. Write a pair of programs, `TopDown` and `BottomUp`, that take a syntax tree and print out the nodes in order of construction. `TopDown` should display the order for a top-down parser, while `BottomUp` should show the order for a bottom-up parser.

   Answer: A top-down parser constructs its parse tree in a root-first, left-child first order. Thus a preorder traversal, with left-to-right handling of children, should produce the order of construction for a top-down parser. A bottom-up parser constructs its parse tree in a root-last, left-child first order. Thus, a postorder traversal, with left-to-right handling of children, should produce the order of construction of a bottom-up parser.

9. The *ClockNoise* language (*CN*) is represented by the following grammar:

$$\begin{array}{lcl} \textit{Goal} & \rightarrow & \textit{ClockNoise} \\ \textit{ClockNoise} & \rightarrow & \textit{ClockNoise}\ \texttt{tick tock} \\ & | & \texttt{tick tock} \end{array}$$

a. What are the LR(1) items of *CN*?

Answer:

[*Goal* → • *ClockNoise*, `EOF`]
[*Goal* → • *ClockNoise*, `tick`]
[*Goal* → • *ClockNoise*, `tock`]
[*Goal* → *ClockNoise* •, `EOF`]
[*Goal* → *ClockNoise* •, `tick`]
[*Goal* → *ClockNoise* •, `tock`]
[*ClockNoise* → • *ClockNoise* `tick tock, EOF`]
[*ClockNoise* → • *ClockNoise* `tick tock, tick`]
[*ClockNoise* → • *ClockNoise* `tick tock, tock`]
[*ClockNoise* → *ClockNoise* • `tick tock, EOF`]
[*ClockNoise* → *ClockNoise* • `tick tock, tick`]
[*ClockNoise* → *ClockNoise* • `tick tock, tock`]
[*ClockNoise* → *ClockNoise* `tick` • `tock, EOF`]
[*ClockNoise* → *ClockNoise* `tick` • `tock, tick`]
[*ClockNoise* → *ClockNoise* `tick` • `tock, tock`]
[*ClockNoise* → *ClockNoise* `tick tock` •, `EOF`]
[*ClockNoise* → *ClockNoise* `tick tock` •, `tick`]
[*ClockNoise* → *ClockNoise* `tick tock` •, `tock`]
[*ClockNoise* → • `tick tock, EOF`]
[*ClockNoise* → • `tick tock, tick`]
[*ClockNoise* → • `tick tock, tock`]
[*ClockNoise* → `tick` • `tock, EOF`]
[*ClockNoise* → `tick` • `tock, tick`]
[*ClockNoise* → `tick` • `tock, tock`]
[*ClockNoise* → `tick tock` •, `EOF`]
[*ClockNoise* → `tick tock` •, `tick`]
[*ClockNoise* → `tick tock` •, `tock`]

b. What are the FIRST sets of *CN*?

Answer:

Symbol	FIRST
Goal	`tick`
ClockNoise	`tick`
`tick`	`tick`
`tock`	`tock`
`EOF`	`EOF`

c. Construct the Canonical Collection of Sets of LR(1) Items for *CN*.

Answer:

CC_0: { [*Goal* → • *ClockNoise*,`EOF`],
[*ClockNoise* → • *ClockNoise tick tock*,{`EOF`,`tick`}],
[*ClockNoise* → • *tick tock*,{`EOF`,`tick`}] }

CC_1: { [*Goal* → *ClockNoise* •,`EOF`],
[*ClockNoise* → *ClockNoise* • *tick tock*,{`EOF`,`tick`}] }

CC_2: { [*ClockNoise* → *tick* • *tock*,{`EOF`,`tick`}] }

CC_3: { [*ClockNoise* → *ClockNoise tick* • *tock*,{`EOF`,`tick`}] }

CC_4: { [*ClockNoise* → *tick tock* •,{`EOF`,`tick`}] }

CC_5: { [*ClockNoise* → *ClockNoise tick tock* •,{`EOF`,`tick`}] }

d. Derive the `Action` and `Goto` tables.

Answer:

ACTION

	`EOF`	`tick`	`tock`
CC_0		s 2	
CC_1	acc	s 3	
CC_2			s 4
CC_3			s 5
CC_4	r 3	r 3	
CC_5	r 2	r 2	

GOTO

	ClockNoise
CC_0	1
CC_1	0
CC_2	0
CC_3	0
CC_4	0
CC_5	0

10. Consider the following grammar:

$$
\begin{array}{lcl}
\mathit{Start} & \rightarrow & S \\
S & \rightarrow & A\ \mathtt{a} \\
A & \rightarrow & B\ C \\
 & \mid & B\ C\ \mathtt{f} \\
B & \rightarrow & \mathtt{b} \\
C & \rightarrow & \mathtt{c}
\end{array}
$$

a. Construct the canonical collection of sets of LR(1) items for this grammar.

Answer:

CC_0: { [*Start* → • *S*,EOF], [*S* → • *A a*,EOF], [*A* → • *B C*,a], [*A* → • *B C f*,a], [*B* → • *b*,c] }

CC_1: { [*Start* → *S* •,EOF] }

CC_2: { [*S* → *A* • *a*,EOF] }

CC_3: { [*A* → *B* • *C*,a], [*A* → *B* • *C f*,a], [*C* → • *c*,{a,f}] }

CC_4: { [*B* → *b* •,c] }

CC_5: { [*S* → *A a* •,EOF] }

CC_6: { [*A* → *B C* •,a], [*A* → *B C* • *f*,a] }

CC_7: { [*C* → *c* •,{a,f}] }

CC_8: { [*A* → *B C f* •,a] }

b. Derive the `Action` and `Goto` tables.

Answer:

ACTION

	EOF	a	f	b	c
CC_0				s 4	
CC_1	acc				
CC_2		s 5			
CC_3					s 7
CC_4					r 5
CC_5	r 2				
CC_6		r 3	s 8		
CC_7		r 6	r 6		
CC_8		r 4			

GOTO

	S	*A*	*B*	*C*
CC_0	1	2	3	0
CC_1	0	0	0	0
CC_2	0	0	0	0
CC_3	0	0	0	6
CC_4	0	0	0	0
CC_5	0	0	0	0
CC_6	0	0	0	0
CC_7	0	0	0	0
CC_8	0	0	0	0

c. Is the grammar LR(1)?

Answer: Yes. the table construction succeeds.

11. Consider a robot arm that accepts two commands: $\triangledown$ puts an apple in the bag and $\triangle$ takes an apple out of the bag. Assume the robot arm starts with an empty bag.

A valid command sequence for the robot arm should have no prefix that contains more $\triangle$ commands than $\triangledown$ commands. As examples, $\triangledown\triangledown\triangle\triangle$ and $\triangledown\triangle\triangledown$ are valid command sequences, but $\triangledown\triangle\triangle\triangledown$ and $\triangledown\triangle\triangledown\triangle\triangle$ are not.

a. Write an LR(1) grammar that represents all the value command sequences for the robot arm.

Answer: A valid command sequence is a list of self-contained sets of commands, where each set consists of a $\triangledown$, an optional set, and an optional $\triangle$. We will assume that a command sequence must have at least one command, a $\triangledown$. We will assume that the grammar needs a unique start symbol. From that description, the grammar is straightforward.

$$\begin{array}{lcl} \textit{Start} & \rightarrow & \textit{List} \\ \\ \textit{List} & \rightarrow & \textit{Set List} \\ & | & \triangledown\ \textit{List} \\ & | & \epsilon \\ \\ \textit{Set} & \rightarrow & \triangledown\ \textit{Tail} \\ \\ \textit{Tail} & \rightarrow & \textit{Set} \triangle \\ & | & \triangle \end{array}$$

b. Prove that the grammar is LR(1).

Answer: The easiest way to tell if a grammar is LR(1) is to build the canonical LR(1) tables for it. That can be done by hand or using a tool. Personally, I would accept a yacc or bison input file, along with the y.output file that showed the outcome. Students, however, tend to construct the table by hand.

First, we construct the FIRST Sets.

Symbol	FIRST
Start	$\triangledown\ \epsilon$
List	$\triangledown\ \epsilon$
Set	$\triangledown$
Tail	$\triangledown\ \triangle$
$\triangledown$	$\triangledown$
$\triangle$	$\triangle$
EOF	EOF

Next, we construct the Canonical Collection of Sets of LR(1) Items:

CC_0: [*Start* $\rightarrow \bullet$ *List*, EOF] [*List* $\rightarrow \bullet$ *Set List*, EOF] [*List* $\rightarrow \bullet \triangledown$ *List*, EOF]
[*List* $\rightarrow \bullet\ \epsilon$, EOF] [*Set* $\rightarrow \bullet \triangledown$ *Tail*, EOF] [*Set* $\rightarrow \bullet \triangledown$ *Tail*, $\triangledown$]

CC_1: [*Start* $\rightarrow$ *List* $\bullet$, EOF]

CC_2: [*List* $\rightarrow \bullet$ *Set List*, EOF] [*List* $\rightarrow$ *Set* $\bullet$ *List*, EOF]
[*List* $\rightarrow \bullet \triangledown$ *List*, EOF] [*List* $\rightarrow \bullet\ \epsilon$, EOF] [*Set* $\rightarrow \bullet \triangledown$ *Tail*, EOF]
[*Set* $\rightarrow \bullet \triangledown$ *Tail*, $\triangledown$]

CC_3: [*List* $\rightarrow \bullet$ *Set List*, EOF] [*List* $\rightarrow \bullet \triangledown$ *List*, EOF]
[*List* $\rightarrow \triangledown \bullet$ *List*, EOF] [*List* $\rightarrow \bullet\ \epsilon$, EOF] [*Set* $\rightarrow \bullet \triangledown$ *Tail*, EOF]
[*Set* $\rightarrow \bullet \triangledown$ *Tail*, $\triangledown$] [*Set* $\rightarrow \bullet \triangledown$ *Tail*, $\triangle$] [*Set* $\rightarrow \triangledown \bullet$ *Tail*, EOF]
[*Set* $\rightarrow \triangledown \bullet$ *Tail*, $\triangledown$] [*Tail* $\rightarrow \bullet$ *Set* $\triangle$, EOF] [*Tail* $\rightarrow \bullet$ *Set* $\triangle$, $\triangledown$]
[*Tail* $\rightarrow \bullet\ \triangle$, EOF] [*Tail* $\rightarrow \bullet\ \triangle$, $\triangledown$]

CC_4: [*List* $\rightarrow \epsilon\ \bullet$, EOF]

CC_5: [*List* $\rightarrow$ *Set List* $\bullet$, EOF]

CC_6: [*List* $\rightarrow \triangledown$ *List* $\bullet$, EOF]

CC_7: [*List* $\rightarrow \bullet$ *Set List*, EOF] [*List* $\rightarrow$ *Set* $\bullet$ *List*, EOF]
[*List* $\rightarrow \bullet \triangledown$ *List*, EOF] [*List* $\rightarrow \bullet\ \epsilon$, EOF] [*Set* $\rightarrow \bullet \triangledown$ *Tail*, EOF]
[*Set* $\rightarrow \bullet \triangledown$ *Tail*, $\triangledown$] [*Tail* $\rightarrow$ *Set* $\bullet\ \triangle$, EOF] [*Tail* $\rightarrow$ *Set* $\bullet\ \triangle$, $\triangledown$]

CC_8: [*List* → • *Set List*, EOF] [*List* → • ▽ *List*, EOF]
[*List* → ▽ • *List*, EOF] [*List* → • ϵ, EOF] [*Set* → • ▽ *Tail*, EOF]
[*Set* → • ▽ *Tail*, ▽] [*Set* → • ▽ *Tail*, △] [*Set* → ▽ • *Tail*, EOF]
[*Set* → ▽ • *Tail*, ▽] [*Set* → ▽ • *Tail*, △] [*Tail* → • *Set* △, EOF]
[*Tail* → • *Set* △, ▽] [*Tail* → • *Set* △, △] [*Tail* → • △, EOF]
[*Tail* → • △, ▽] [*Tail* → • △, △]

CC_9: [*Set* → ▽ *Tail* •, EOF] [*Set* → ▽ *Tail* •, ▽]

CC_{10}: [*Tail* → △ •, EOF] [*Tail* → △ •, ▽]

CC_{11}: [*Tail* → *Set* △ •, EOF] [*Tail* → *Set* △ •, ▽]

CC_{12}: [*List* → • *Set List*, EOF] [*List* → *Set* • *List*, EOF]
[*List* → • ▽ *List*, EOF] [*List* → • ϵ, EOF] [*Set* → • ▽ *Tail*, EOF]
[*Set* → • ▽ *Tail*, ▽] [*Tail* → *Set* • △, EOF] [*Tail* → *Set* • △, ▽]
[*Tail* → *Set* • △, △]

CC_{13}: [*Set* → ▽ *Tail* •, EOF] [*Set* → ▽ *Tail* •, ▽] [*Set* → ▽ *Tail* •, △]

CC_{14}: [*Tail* → △ •, EOF] [*Tail* → △ •, ▽] [*Tail* → △ •, △]

CC_{15}: [*Tail* → *Set* △ •, EOF] [*Tail* → *Set* △ •, ▽] [*Tail* → *Set* △ •, △]

And, finally, we build the ACTION and GOTO tables:

	EOF	$\triangledown$	ϵ	$\triangle$
cc_0		s 3	s 4	
cc_1	acc			
cc_2		s 3	s 4	
cc_3		s 8	s 4	s 10
cc_4	r 4			
cc_5	r 2			
cc_6	r 3			
cc_7		s 3	s 4	s 11
cc_8		s 8	s 4	s 14
cc_9	r 5	r 5		
cc_{10}	r 7	r 7		
cc_{11}	r 6	r 6		
cc_{12}		s 3	s 4	s 15
cc_{13}	r 5	r 5		r 5
cc_{14}	r 7	r 7		r 7
cc_{15}	r 6	r 6		r 6

	List	Set	Tail
cc_0	1	2	0
cc_1	0	0	0
cc_2	5	2	0
cc_3	6	7	9
cc_4	0	0	0
cc_5	0	0	0
cc_6	0	0	0
cc_7	5	2	0
cc_8	6	12	13
cc_9	0	0	0
cc_{10}	0	0	0
cc_{11}	0	0	0
cc_{12}	5	2	0
cc_{13}	0	0	0
cc_{14}	0	0	0
cc_{15}	0	0	0

The construction succeeds, so the grammar is LR(1).

12. The following grammar has no known LL(1) equivalent:

0	*Start*	→	*A*
1		|	*B*
2	*A*	→	(*A*)
3		|	a
4	*B*	→	(*B* >
5		|	b

Show that the grammar is LR(1).

Answer: The easiest way to show that a grammar is LR(1) is to build the canonical LR(1) tables for it. That can be done by hand or using a tool. Personally, I would accept a yacc or bison input file, along with the y.output file that showed the outcome. Students, however, tend to construct the table by hand.

First, we construct FIRST sets.

Symbol	FIRST
EOF	EOF
Start	(a b
A	(a
B	(b
(	(
)	)
a	a
>	>
b	b

Next, we construct the Canonical Collection of Sets of LR(1) Items:

CC_0: $[Start \rightarrow \bullet\ B, \texttt{EOF}]$ $[B \rightarrow \bullet\ \texttt{(}\ B\ \texttt{>}, \texttt{EOF}]$ $[B \rightarrow \bullet\ b, \texttt{EOF}]$

CC_1: $[Start \rightarrow B\ \bullet, \texttt{EOF}]$

CC_2: $[B \rightarrow \bullet\ \texttt{(}\ B\ \texttt{>}, \texttt{>}]$ $[B \rightarrow \texttt{(}\ \bullet\ B\ \texttt{>}, \texttt{EOF}]$ $[B \rightarrow \bullet\ b, \texttt{>}]$

CC_3: $[B \rightarrow b\ \bullet, \texttt{EOF}]$

CC_4: $[B \rightarrow \texttt{(}\ B\ \bullet\ \texttt{>}, \texttt{EOF}]$

CC_5: $[B \rightarrow \bullet\ \texttt{(}\ B\ \texttt{>}, \texttt{>}]$ $[B \rightarrow \texttt{(}\ \bullet\ B\ \texttt{>}, \texttt{>}]$ $[B \rightarrow \bullet\ b, \texttt{>}]$

CC_6: $[B \rightarrow b\ \bullet, \texttt{>}]$

CC_7: $[B \rightarrow \texttt{(}\ B\ \texttt{>}\ \bullet, \texttt{EOF}]$

CC_8: $[B \rightarrow \texttt{(}\ B\ \bullet\ \texttt{>}, \texttt{>}]$

CC_9: $[B \rightarrow \texttt{(}\ B\ \texttt{>}\ \bullet, \texttt{>}]$

And, finally, we build the ACTION and GOTO tables:

ACTION

	EOF	(	)	a	>	b
cc_0		s 2				s 3
cc_1	acc					
cc_2		s 5				s 6
cc_3	r 6					
cc_4					s 7	
cc_5		s 5				s 6
cc_6					r 6	
cc_7	r 5					
cc_8					s 9	
cc_9					5	

GOTO

	A	*B*
cc_0	0	1
cc_1	0	0
cc_2	0	4
cc_3	0	0
cc_4	0	0
cc_5	0	8
cc_6	0	0
cc_7	0	0
cc_8	0	0
cc_9	0	0

The fact that the table construction succeeds (i.e., it does not define any table location twice) implies that the grammar is LR(1).

Section 3.6

13. Write a grammar for expressions that can include binary operators (+ and x), unary minus (-), autoincrement (++), and autodecrement (- -) with their customary precedence. Assume that repeated unary minuses are not allowed, but that repeated autoincrement and autodecrement operators are allowed.

Answer:

Expr → *Expr* + *Term*
| *Expr* - *Term*
| *Term*

Term → *Term* × *Factor*
| *Term* ÷ *Factor*
| *Factor*

Factor → (*Expr*)
| - (*Expr*)
| - num
| num
| - id
| - *Auto*
| *Auto*

Auto → ++ *Auto*
| - - *Auto*
| *Auto* ++
| *Auto* - -
| id

Section 3.7

14. Consider the task of building a parser for the programming language Scheme. Contrast the effort required for a top-down recursive-descent parser with that needed for a table-driven LR(1) parser. (Assume that you already have an LR(1) table generator.)

Answer: While the answer is subjective, a good case can be made for either technique. The syntax of Scheme is sufficiently simple that a recursive descent parser with be simple, compact, and fast. By the same argument, writing an LR(1) grammar for Scheme is easy.

Most of the difficulty lies in deriving a grammar that is suitable for the selected parsing technique. For the top-down parser, the grammar must avoid left recursion. That simple fact may shift the balance slightly in favor of the LR(1) solution.

15. The text describes a manual technique for eliminating useless productions in a grammar.

    a. Can you modify the LR(1) table-construction algorithm so that it automatically eliminates the overhead from useless productions?

    Answer: The author of the table constructor can easily add code that recognizes any LR(1) item of the form $[A \rightarrow \bullet B]$, for a non-terminal symbol B, and replaces it with a series of LR(1) items that expand B into its various alternative right-hand sides. This transformation has the effect of eliminating productions of the form $A \rightarrow B$.

    b. Even though a production is syntactically useless, it may serve a practical purpose. For example, the compiler writer might associate a syntax-directed action (see Chapter 4) with the useless production. How should your modified table-construction algorithm handle an action associated with a useless production?

    Answer: The transformation needs to check for the presence of a syntax-directed action before it eliminates a production. If a syntactially-useless production has an action that occurs when the parser reduces by it, the production serves a purpose and should not be eliminated.

CHAPTER 4

Section 4.2

1. In Scheme, the + operator is overloaded. Given that Scheme is dynamically typed, describe a method to type check an operation of the form `(+ a b)` where `a` and `b` may be of any type that is valid for the + operator.

   Answer: The implementation can associate a tag with the run-time representation of each value. The implementation of + can verify the types of its operands by checking their tag fields. The results of those checks allow + to select the correct sequence of actions for any set of operands.

2. Some languages, such as APL or PHP, neither require variable declarations nor enforce consistency between assignments to the same variable. (A program can assign the integer 10 to × and later assign the string value "book" to × in the same scope.) This style of programming is sometimes called *type juggling*.

   Suppose that you have an existing implementation of a language that has no declarations but requires type-consistent uses. How could you modify it to allow type juggling?

   Answer: The implementation must consider each assignment as defining both a new value and, potentially, a new type. When a variable is used, its type is the type last assigned to the variable.

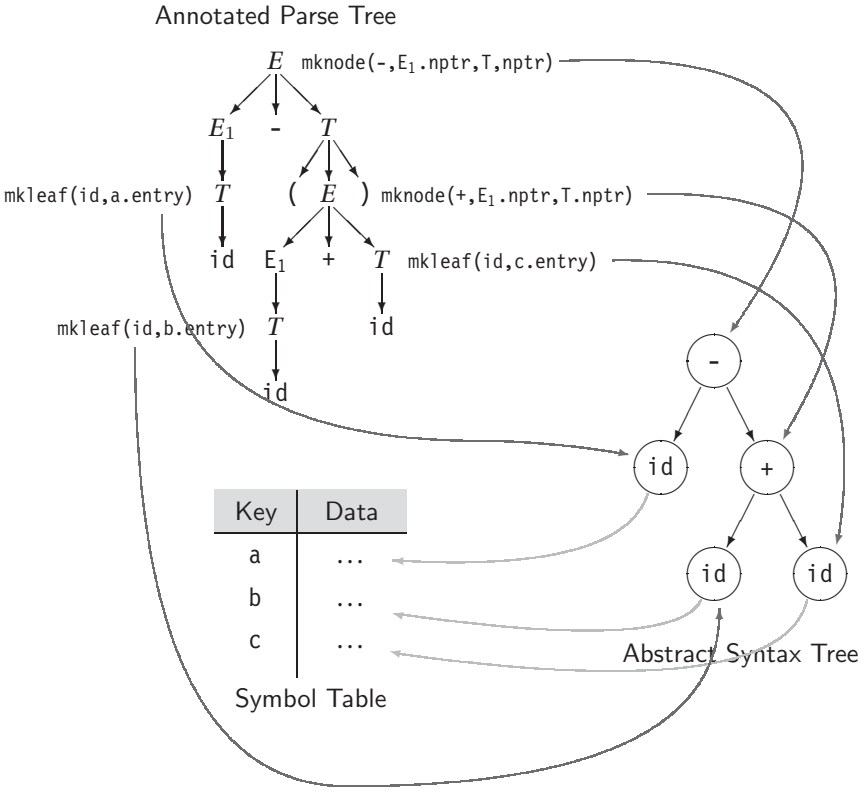
   With control-flow, the programmer can write code that assigns distinct types to a single variable along different control-flow paths. Such use patterns may require run-time checking. The implementation should attach a tag (a run-time location that holds a value denoting the item's data type) to each variable. The implementation can set the tag's value appropriately at each assignment and explicitly check the tag's value at each use. By tracking the type information at runtime, theimplementation can ensure that each value is handled correctly.

Section 4.3

3. Based on the following evaluation rules, draw an annotated parse tree that shows how the syntax tree for `a - (b + c)` is constructed.

Production	Evaluation Rules
$E_0 \rightarrow E_1 + T$	{ $E_0.nptr \leftarrow mknode($ `+`, $E_1.nptr, T.nptr)$ }
$E_0 \rightarrow E_1 - T$	{ $E_0.nptr \leftarrow mknode($ `-`, $E_1.nptr, T.nptr)$ }
$E_0 \rightarrow T$	{ $E_0.nptr \leftarrow T.nptr$ }
$T \rightarrow (E)$	{ $T.nptr \leftarrow E.nptr$ }
$T \rightarrow$ `id`	{ $T.nptr \leftarrow mkleaf($ `id,id` $.entry)$ }

Answer:



4. Use the attribute-grammar paradigm to write an interpreter for the classic expression grammar. Assume that each `name` has a `value` attribute and a `lexeme` attribute. Assume that all attributes are already defined and that all values will always have the same type.

Answer: Assume the following grammar:

Expr → *Expr* + *Term*	*Term* → *Term* × *Factor*	*Factor* → (*Expr*)
| *Expr* - *Term*	| *Term* ÷ *Factor*	| `number`
| *Term*	| *Factor*	| `name`

Now, a set of attribution rules:

	Production	Attribution Rule
1	*Expr* → *Expr* + *Term*	$Expr_0$.`value` ← $Expr_1$.`value` + *Term*.`value`
2	*Expr* → *Expr* - *Term*	$Expr_0$.value ← $Expr_1$.value - *Term*.`value`
3	*Expr* → *Term*	*Expr*.`value` ← *Term*.`value`
4	*Term* → *Term* × *Factor*	$Term_0$.`value` ← $Term_1$.`value` × *Factor*
5	*Term* → *Term* ÷ *Factor*	$Term_0$.`value` ← $Term_1$.`value` ÷ *Factor*
6	*Term* → *Factor*	*Term*.`value` ← *Factor*.`value`
7	*Factor* → (*Expr*)	*Factor*.`value` ← *Expr*.`value`
8	*Factor* → `number`	*Factor*.`value` ← *ValueOf*(`number.lexeme`)
9	*Factor* → `name`	*Factor*.`value` ← `name.value`

Here, *ValueOf* is a simple function that computes a value from a lexeme—the text representation of the number.

5. Write a grammar to describe all binary numbers that are multiples of four. Add attribution rules to the grammar that will annotate the start symbol of a syntax tree with an attribute `value` that contains the decimal value of the binary number.

Answer: To simplify the grammar, we will assume that the scanner can recognize the final double zero in a binary number that is a multiple of four as a single word. (If we try to recognize it as two consecutive zeros in the grammar, we hit shift reduce problems that seriously complicate the grammar.) To recognize *DZero* requires two characters of lookahead, but it has a small constant additional cost. With that caveat, we can use the following LR(1) grammar:

$$
\begin{array}{lcl}
\mathit{Number} & \rightarrow & \mathtt{0} \\
 & | & \mathtt{1}\ \ \mathit{List} \\
\mathit{List} & \rightarrow & \mathtt{1}\ \ \mathit{List} \\
 & | & \mathtt{0}\ \ \mathit{List} \\
 & | & \mathit{DZero}
\end{array}
$$

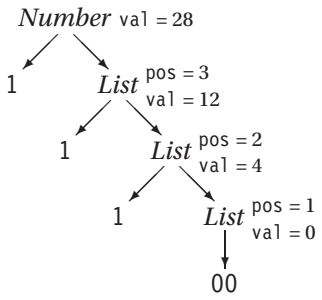
Next, we add attribution rules to the grammar:

	Production	Attribution Rule
1	$\mathit{Number} \rightarrow \mathtt{0}$	$\mathit{Number}.\mathtt{val} \leftarrow 0$
2	$\mathit{Number} \rightarrow \mathtt{1}\ \ \mathit{List}$	$\mathit{Number}.\mathtt{val} \leftarrow 2^{(\mathit{List}.\mathtt{pos}+1)} + \mathit{List}.\mathtt{val}$
3	$\mathit{List}_0 \rightarrow \mathtt{1}\ \ \mathit{List}_1$	$\mathit{List}_0.\mathtt{pos} \leftarrow \mathit{List}_1.\mathtt{pos} + 1$ $\mathit{List}_0.\mathtt{val} \leftarrow 2^{\mathit{List}_0.\mathtt{pos}} + \mathit{List}_1.\mathtt{val}$
4	$\mathit{List}_0 \rightarrow \mathtt{0}\ \ \mathit{List}_1$	$\mathit{List}_0.\mathtt{pos} \leftarrow \mathit{List}_1.\mathtt{pos} + 1$ $\mathit{List}_0.\mathtt{val} \leftarrow \mathit{List}_1.\mathtt{val}$
5	$\mathit{List} \rightarrow \mathit{DZero}$	$\mathit{List}.\mathtt{val} \leftarrow 0$ $\mathit{List}.\mathtt{pos} \leftarrow 1$

6. Using the grammar defined in the previous exercise, build the syntax tree for the binary number 11100.

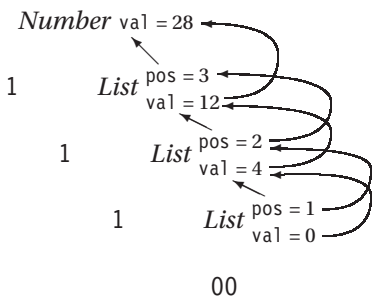
   (a) Show all the attributes in the tree with their corresponding values.

   Answer:



   (b) Draw the attribute dependence graph for the syntax tree and classify all attributes as being either synthesized or inherited.

   Answer:



   All attributes are synthesized.

Section 4.4

7. A Pascal program can declare two integer variables `a` and `b` with the syntax

```
var a, b: int
```

   This declaration might be described with the following grammar:

$$
\begin{array}{lcl}
\textit{VarDecl} & \rightarrow & \texttt{var}\ \textit{IDList}\ \texttt{:}\ \textit{TypeID} \\
\textit{IDList} & \rightarrow & \textit{IDList}, \textit{ID} \\
 & | & \textit{ID}
\end{array}
$$

   where *IDList* derives a comma-separated list of variable names and *TypeID* derives a valid Pascal type. You may find it necessary to rewrite the grammar.

(a) Write an attribute grammar that assigns the correct data type to each declared variable.

Answer:

VarDecl → `var` *IDList* : *TypeID*	*TypeID*.`type` ← type of *TypeID* ; *IDList*.`type` ← *TypeID*.`type`
$IDList_0$ → $IDList_1$, *ID*	*ID*.`type` ← $IDList_0$.`type` ; $IDList_1$.`type` ← $IDList_0$.`type`
| *ID*	*ID*.`type` ← $IDList_0$.`type` ;

(b) Write an ad hoc syntax-directed translation scheme that assigns the correct data type to each declared variable.

Answer:

VarDecl → `var` *IDList* : *TypeID*	for each ID_k in \$1 ID_k ← type of *TypeID*
$IDList_0$ → $IDList_1$, *ID*	\$\$ ← `cons`(\$1,`makenode(`*ID*`)`
| *ID*	\$\$ ← `makenode(`*ID*`)`

The iterator used in the action for the first production is somewhat abstract. This can be done with an iterative list traversal, an abstract function such as Scheme's `map`, or a hand-coded recursive function

(c) Can either scheme operate in a single pass over the syntax tree?

Answer: With an appropriate evaluation order (e.g., a right-to-left preorder walk), an attribute gramar evaluator can process the rules from part (a) in a single pass.

The ad-hoc syntax-directed translation scheme must parse the list of *ID*s first, before it sees the declared type. The scheme in part (b) of this solution builds a list of those elements, discovers the type, and walks the list to set their types. Thus, it takes two passes over the syntax tree.

8. Sometimes, the compiler writer can move an issue across the boundary between context-free and context-sensitive analysis. Consider, for example, the classic ambiguity that arises between function invocation and array references in FORTRAN 77 (and other languages). These constructs might be added to the classic expression grammar using the productions:

$$\begin{array}{lcl} \textit{Factor} & \rightarrow & \texttt{name (} \textit{ExprList} \texttt{)} \\ \textit{ExprList} & \rightarrow & \textit{ExprList} \texttt{ , } \textit{Expr} \\ & | & \textit{Expr} \end{array}$$

Here, the only difference between a function invocation and an array reference lies in how the `name` is declared.

In previous chapters, we have discussed using cooperation between the scanner and the parser to disambiguate these constructs. Can the problem be solved during context-sensitive analysis? Which solution is preferable?

Answer: To solve the problem during context sensitive analysis, the compiler can use a grammar that contains only a single derivation for the two constructs (function invocations and array references). The parser can build a representation that preserves all the information. During context-sensitive analysis, when the type information is naturally available, the compiler can rewrite the intermediate representation as a function invocation or an array reference, as appropriate.

Both approaches (scanner-parser cooperation and rewriting during context-sensitive analysis) work. The scanner-parser scheme isolates the problem in the syntax-oriented portions of the front end. It may add a small overhead to processing every name. The context-sensitive analysis scheme simplifies the scanner and parser, defers the issue until complete type information is available, and pushes the complication later into compilation.

Neither approach is a clear win. The best answer is to change the syntax of the source language and avoid the ambiguity. Unfortunately, that is not an option for FORTRAN. We are at least 40 years too late!

9. Sometimes, a language specification uses context-sensitive mechanisms to check properties that can be tested in a context-free way. Consider the grammar fragment in Figure 4.16 on page 208. It allows an arbitrary number of *StorageClass* specifiers when, in fact, the standard restricts a declaration to a single *StorageClass* specifier.

   (a) Rewrite the grammar to enforce the restriction grammatically.

   Answer:

   $$\begin{aligned} \textit{SpecifierList} &\rightarrow \textit{StorageClass TypeList} \\ &\mid \textit{TypeList} \\ \textit{TypeList} &\rightarrow \textit{TypeSpecifier TypeList} \\ &\mid \textit{TypeSpecifier} \end{aligned}$$

   (b) Similarly, the language allows only a limited set of combinations of *TypeSpecifier*. `long` is allowed with either `int` or `float`; `short` is allowed only with `int`. Either `signed` or `unsigned` can appear with any form of `int`. `signed` may also appear on `char`. Can these restrictions be written into the grammar?

   Answer: Yes. They can be added to the grammar at the cost of additional productions and states in the parser.

   Hint: The scanner returned a single token type for any of the *StorageClass* values and another token type for any of the *TypeSpecifiers*.

   (c) Propose an explanation for why the authors structured the grammar as they did.

   Answer: Reducing the set of terminals (by combining all *StorageClass* specifiers into a single terminal and all *TypeSpecifiers* into a single terminal) shrinks the tables and produces a more compact parser, at the cost of checking the restrictions during context-sensitive analysis.

   (d) Do your revisions to the grammar change the overall speed of the parser? In building a parser for C, would you use the grammar like the one in Figure 4.16, or would you prefer your revised grammar? Justify your answer.

   Answer: The version shown in Figure 4.16 also avoids one reduction on each use of a *StorageClass* or *TypeSpecifier*, so it makes the parser marginally faster. As to which version of the grammar is preferred, that is a subjective matter.

   Encoding the restriction in the grammar avoids the need to write code that checks the restrictions. Doing the check during context-sensitive analysis allows the compiler writer to produce a more useful diagnostic message when an error is discovered.

Section 4.5

10. Object-oriented languages allow operator and function overloading. In these languages, the function name is not always a unique identifier, since you can have multiple related definitions, as in

```
void Show(int);
void Show(char *);
void Show(float);
```

For lookup purposes, the compiler must construct a distinct identifier for each function. Sometimes, such overloaded functions will have different return types, as well. How would you create distinct identifiers for such functions?

Answer: Assume that the compiler has a character, such as $, that is not allowed in an identifier. The compiler can create distinct function names by concatenating the function name, the return type, and the parameter types, separating each with $. Applying this scheme to the declarations for `Show` yields three distinct identifiers:

```
Show$void$int
Show$void$char*
Show$void$float
```

11. Inheritance can create problems for the implementation of object-oriented languages. When object type A is a parent of object type B, a program can assign a "pointer to B" to a "pointer to A," with syntax such as `a` $\leftarrow$ `b`. This should not cause problems since everything that A can do, B can also do. However, one cannot assign a "pointer to A" to a "pointer to B," since object class B can implement methods that object class A does not.

Design a mechanism that can use ad hoc syntax-directed translation to determine whether or not a pointer assignment of this kind is allowed.

Answer: If the declarations of A and B must be present at compile time, the compiler can build a tree to represent the inheritance hierarchy. Given an inheritance tree and an assignment "$x \leftarrow y$" the compiler can determine if the type of x is a parent of the type of y by walking up the inheritance tree from the node for type of y.

Section 5.2

1. A parse tree contains much more information than an abstract syntax tree.

   a. In what circumstances might you need information that is found in the parse tree but not the abstract syntax tree?

      Answer: In any application where the compiler needs to generate output that the programmer will read, the parse tree may contain information that helps produce results that more closely resemble the original input program.

   b. What is the relationship between the size of the input program and its parse tree? Its abstract syntax tree?

      Answer: The parse tree has, to a first approximation, size proportional to the grammatical derivation of the input program (recall Chapter 3). That is, it has (roughly) a node for each word in the input program, along with a node for every reduction in the parse. Of course, some multi-word constructs, such as a C `for` loop, may be represented with fewer nodes than they have words.

      The abstract syntax tree has, to a first approximation, a node for each word in the input program. It may have more nodes if it expands source-level abstractions, such as an array reference (see Chapter 7), into an address calculation. It may have more nodes if it represents control-flow structures implicity—for example, a single node for a loop construct such as the C `for` loop, rather than a node for each of the parentheses and semicolons.

   c. Propose an algorithm to recover a program's parse tree from its abstract syntax tree.

      Answer: The simplest algorithm would be to regenerate the syntax from the AST and to parse it again.

2. Write an algorithm to convert an expression tree into a DAG.

Answer: To facilitate comparisons among subexpressions, we need a hash table that maps subexpressions (or nodes in the expression tree) into unique integers (which can be as simple as the index of the corresponding table entry.) The algorithm traverses the expression tree in a post-order walk. Assume, without loss of generality, that each node has no more than two children.

```
BuildDAG( root ) {
    initialize hash table entries to null
    DAG ← Walk( root )
    return DAG
}

Walk( node ) {
    if node is a leaf then
        if lexeme(node) is not in the hash table then
            create an entry for it in the table
            create a node x for it in the new DAG
            store x in the table entry
        else let x be the DAG node stored in the table entry

    else {
        if LeftChild(node) exists
            then Left ← Walk(LeftChild(node))
            else Left ← null

        if RightChild(node) exists
            then Right ← Walk(RightChild(node))
            else Right ← null

        if the key "op(node),Left,Right" is not in the hash table then
            create an entry for it in the table
            create a node x in the new DAG
            set LeftChild(x) to be Left
            set RightChild(x) to be Right

        else let x be the DAG node stored in the table entry
    }
    return x
}
```

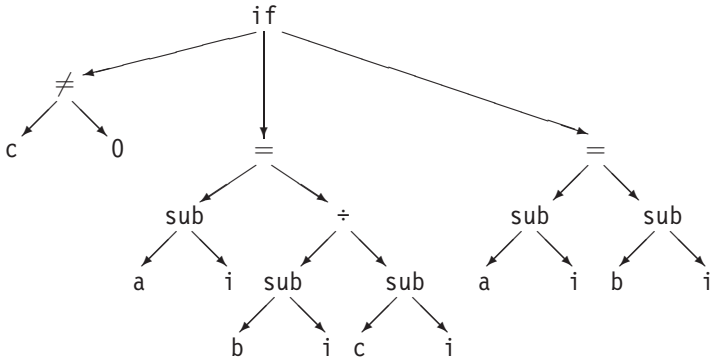
Section 5.3

3. Show how the following code fragment

```
if (c[i] ≠ 0)
   then a[i] ← b[i] ÷ c[i];
   else a[i] ← b[i];
```

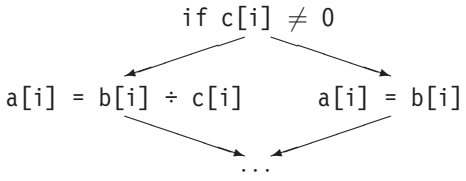
might be represented in an abstract syntax tree, in a control-flow graph, and in quadruples. Discuss the advantages of each representation. For what applications would one representation be preferable to the others?

Answer: *Abstract syntax tree:* The abstract syntax tree has a number of advantages. Because it is close to the actual syntax of the original code, it is easy to regenerate source code in a walk of the AST. The AST typically represents code at a near-source level of abstraction, which makes it a good representation for some kinds of analysis and transformation. For example, it is easier to move a string-copy operation as a compact subtree in an AST than as a long sequence of near-assembly level operations.



Among the disadvantages of an AST are the fact that it does not represent control-flow in any explicit way.

Control-flow graph: The control-flow graph is useful for reasoning about the flow of control in the original code. In the example, the CFG makes it obvious that one of the two assignments executes, where the AST does not.



Among the disadvantages of a CFG are the fact that it does not represent the computation in blocks in any explicity way.

Quadruples: A low-level representation such as quadruples has the advantage that it exposes *all* of the computation. The representation is easy to understand and easy to manipulate. Simple data structures can be used to represent quadruples.

```
        r1        ←  i × 4
        r2        ←  r1 + @c
        r3        ←  Mem(r2)
        r4        ←  0
        cmp           r3 = r4
        bneq      →  L1, L2

L1:     r5        ←  i × 4
        r6        ←  r5 + @b
        r7        ←  Mem(r6)
        r8        ←  r5 + @c
        r9        ←  Mem(r8)
        r10       ←  r7 ÷ r9
        r11       ←  r5 + @a
        Mem(r11)  ←  r10
        b         →  L3

L2:     r12       ←  i × 4
        r13       ←  r12 + @b
        r14       ←  Mem(r13)
        r15       ←  r12 + @a
        Mem(r15)  ←  r14
        b         →  L3

L3:     nop
```

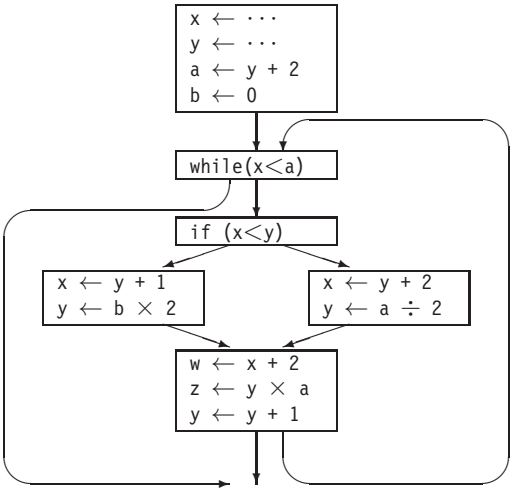
One disadvantage of a low-level linear representation such as quadruples is the fact that all detail is exposed. Transformations that try to manipulate groups of operations, such as moving an entire assignment statement or a string copy operation, must manipulate each operation. In the example, the block of operations starting at L_1 is a single source-level statement, as is the block at L_2.

```
...
x ← ...
y ← ...
a ← y + 2
b ← 0
while(x < a)
   if (y < x)
      x ← y + 1
      y ← b × 2
   else
      x ← y + 2
      y ← a ÷ 2;
   w ← x + 2
   z ← y × a
   y ← y + 1
```

FIGURE 5.13 Code Fragment for Exercise 4

4. Examine the code fragment shown in Figure 5.13. Draw its CFG and show its SSA form as a linear code.

Answer:



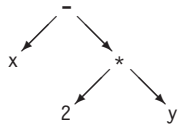
Control-flow Graph

$x_0 \leftarrow \cdots$
$y_0 \leftarrow \cdots$
$a_0 \leftarrow y_0$
$b_0 \leftarrow 0$
if ($x_0 > a_0$) goto l2
l1: $x_1 \leftarrow \phi(x_0, x_4)$
$y_1 \leftarrow \phi(y_0, y_5)$
if ($x_1 < y_1$)
$x_2 \leftarrow y_1 + 1$
$y_2 \leftarrow b_0 \times 2$
else
$x_3 \leftarrow y_1 + 2$
$y_3 \leftarrow a_0 / 2$
$x_4 \leftarrow \phi(x_2, x_3)$
$y_4 \leftarrow \phi(y_2, y_3)$
$w_0 \leftarrow x_4 + 2$
$z_0 \leftarrow y_4 \times a_0$
$y_5 \leftarrow y_4 + 1$
if ($x_4 < a_0$) goto l1
l2: ...

SSA Form

5. Show how the expression x - 2 × y might be translated into an abstract syntax tree, one-address code, two-address code, and three-address code.

Answer:



AST

```
push  2
push  y
mul
push  x
sub
```

1 Address

```
t  ←  2
t  ←  t × y
t  ←  x - t
```

2 Address

```
r0  ←  2
r1  ←  y
r2  ←  r0 × r1
r3  ←  x
r4  ←  r3 - r2
```

3 Address

Note that the 1-address code assumes that `sub` computes `stack[top] - stack[top-1]`.

6. Given a linear list of ILOC operations, develop an algorithm that finds the basic blocks in the ILOC code. Extend your algorithm to build a control-flow graph to represent the connections between blocks.

Taken from § 9.2 of EaC, 1^{st} Edition.

Answer: At its simplest, the CFG identifies the beginning and the end of each basic block and connects the resulting blocks with edges that describe the possible transfers of control among blocks. Initially, let's assume that the CFG-builder receives, as input, a simple, linear IR that represents a procedure in a classic, Algol-like language. At the end of this section, we briefly explore some of the complications that can arise in CFG construction.

To begin, the compiler must find the beginning and the end of each basic block. In a linear IR, the initial operation of a block is sometimes called a *leader*. An operation is a leader if it is the first operation in the procedure, *or* if it has a label that is, potentially, the target of some branch. The compiler can identify all the leaders in a single pass over the IR, shown on the left-hand side of the unnumbered Figure labelled "Building a Control-flow Graph". It iterates over the operations in the program, in order, finds the labelled statements, and records them as leaders.[1]

The second pass finds every operation that ends a block. It assumes that every block ends with a branch or a jump, and that branches specify labels for both outcomes— a "branch taken" label and a "branch not taken" label. This simplifies the handling of blocks and allows the compiler's back end to

1. If the code contains spurious labels, this may unnecessarily split blocks. A more complex algorithm would only add branch and jump targets to the leader set. If the code contains any ambiguous jumps—that is, a jump to an address in a register—then it must include all labelled statements as leaders anyway.

```
next ← 1
leader[next++] ← 1
for i ← 1 to n
  if op_i has a label l_i then
    leader[next++] ← i
    create a CFG node for l_i
```

Finding Leaders

```
for i ← 1 to next - 1
  j ← leader[i] + 1
  while (j ≤ n and op_j ∉ leader)
    j ← j + 1
  j ← j - 1
  last[i] ← j

  if op_j is "cbr r_k→l_1,l_2" then
    add edge from j to node for l_1
    add edge from j to node for l_2
  else if op_j is "jumpI→l_1" then
    add edge from j to node for l_1
  else if op_j is "jump→r_1" then
    add edges from j to all other nodes
```

Finding Last & Adding Edges

Building a Control-flow Graph

choose which path will follow the "fall through" case of a branch. (For the moment, assume branches have no delay slots.)

To find the end of each block, the algorithm iterates through the blocks, in order of their appearance in the *leader* array. It walks forward through the IR until it finds the leader of the following block. The operation immediately before that leader ends the current block. The algorithm records that operation's index in *last[i]*, so that the pair $\langle$*leader[i]*,*last[i]*$\rangle$ describes block i. It adds the needed edges to the CFG. This pass takes time proportional to the size of the IR program.

As we have assumed throughout the book, the CFG should have a unique entry node n_0 and a unique exit node n_f. The underlying code should have this shape. If it does not, a simple postpass over the graph can create n_0 and n_f.

Complications in CFG Construction Features of the IR, the target architecture, and even the source language can complicate this process. For example, an ambiguous jump—ILOC's `jump` operation—may force the compiler to add an edge from the jump to every labelled block in the procedure. This can add edges to the CFG that never occur at run time, degrade the quality of the data-flow information that the compiler derives, and reduce the effectiveness of optimization. In general, the compiler writer should avoid creating ambiguous jumps when the targets can be known, even if this requires

extra analysis during IR generation. The ILOC examples in this book avoid using the jump-to-register operation, `jump`, in favor of the immediate jump, `jumpI`.

The compiler writer can improve this situation, to some extent, by including features in the IR that record potential jump targets. ILOC includes the `tbl` pseudo-operation to let the compiler record the potential targets of an ambiguous jump. Anytime that the compiler generates a `jump`, it should follow the branch with a set of `tbl` operations that record the possible targets for the branch. When it builds the CFG, the compiler can use these hints to limit the number of spurious edges. Any `jump` followed by a set of `tbl`s gets an edge to each block named in those `tbl` operations. Any `jump` that has no `tbl`s generates an edge to each labelled block.

The target architecture can complicate the process. The algorithm given in the unnumbered figure assumes that all leaders, except the first, are labelled. If the target machine has fall-through branches, the algorithm must be extended to recognize unlabeled statements that receive control on a fall-through path. If the code has been scheduled and it models branch-delay slots, the problem becomes harder. A labelled statement that sits in a branch delay slot is a member of two distinct blocks. The compiler can cure this by replication—creating a new (unlabeled) copy of the operations in the delay slots.

Delay slots also complicate finding the end of a block. If a branch or jump can occur in a branch delay slot, the CFG builder must walk forward from the leader to find the block-ending branch—the first branch it encounters. Branches in the delay slot of a block-ending branch can, themselves, be pending on entry to the target block. They can split the target block and force creation of new blocks and new edges. The analysis required to create a CFG in these circumstances is much more complex.

Some languages, such as Pascal and Algol, allow jumps to labels outside the current procedure. This transfer can be modelled in the current procedure with a branch to a CFG node created to represent the target. The complication arises on the other end of the branch, where a transfer from an unknown source can reach a block. For this reason, non-local gotos are usually restricted to lexically nested procedures, where the compiler can see the CFGs for all related procedures at once. In that situation, it can insert the correct edge and model the effects correctly. Otherwise any block whose label can be the target of a non-local goto must be treated as if it has an unknown predecessor in the CFG.

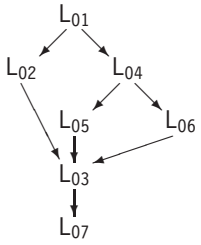
```
L01: add    ra,rb   ⇒ r1
     add    rc,rd   ⇒ r2
     add    r1,r2   ⇒ r3
     add    ra,rb   ⇒ r4
     cmp_LT r1,r2   ⇒ r5
     cbr    r5      → L02,L04
L02: add    ra,rb   ⇒ r6
     multI  r6,17   ⇒ r7
     jumpI          → L03
L03: add    ra, rb  ⇒ r22
     multI  r22,17  ⇒ r23
     jumpI          → L07
L04: add    rc,rd   ⇒ r8
     i2i    ra      ⇒ r9
     cmp_LT r9,rd   ⇒ r10
     cbr    r10     → L05,L06
L05: add    r9,rb   ⇒ r11
     add    ra,rb   ⇒ r12
     add    rc,rd   ⇒ r13
     i2i    ra      ⇒ r13
     add    r13,rb  ⇒ r14
     multI  r12,17  ⇒ r15
     jumpI          → L03
L06: add    r1,r2   ⇒ r16
     i2i    r2      ⇒ r17
     i2i    r1      ⇒ r18
     add    r17,r18 ⇒ r19
     add    r18,r17 ⇒ r20
     multI  r1,17   ⇒ r21
     jumpI          → L03
L07: nop
```

FIGURE 5.14 Code Fragment for Exercise 7

Section 5.4

7. For the code shown in Figure 5.14 find the basic blocks and construct the CFG.

   Answer: The basic blocks are easy to find. Each labelled statement is a leader. Each block ends at the first `cbr` or `jumpI`. The CFG is shown below.



```
static int max = 0;
void A(int b, int e)
{
  int a, c, d, p;
  a = B(b);
  if (b > 100) {
     c = a + b;
     d = c * 5 + e;
  }
  else
     c = a * b;
  *p = c;
  C(&p);
}
```

```
int B(int k)
{
  int x, y;
  x = pow(2, k);
  y = x * 5;
  return y;
}

void C(int *p)
{
  if (*p > max)
     max = *p;
}
```

FIGURE 5.15 Code for Exercise 8

8. Consider the three C procedures shown in Figure 5.15.

   a. Suppose a compiler uses a register-to-register memory model. Which variables in procedures A, B, and C would the compiler be forced to store in memory? Justify your answers.

      Answer: In a register-to-register model, the compiler would need to store p in procedure A in memory, because A passes p's address to C. The other variables are all used locally or passed as values to other procedures. Thus, the compiler can safely keep them in registers.

   b. Suppose a compiler uses a memory-to-memory model. Consider the execution of the two statements that are in the `if` clause of the `if-else` construct. If the compiler has two registers available at that point in the computation, how many loads and stores would the compiler need to issue in order to load values in registers and store them back to memory during execution of those two statements? What if the compiler has three registers available?

      Answer: With two registers, the compiler might generate the following code:

for c = a + b

```
load  a      ⇒ r1
load  b      ⇒ r2
add   r1,r2  ⇒ r3
store r1     ⇒ c
```

for d = c * 5 + e

```
loadI 5      ⇒ r1
load  c      ⇒ r2
mult  r1,r2  ⇒ r1
load  e      ⇒ r2
add   r1,r2  ⇒ r1
store r1     ⇒ d
```

With a third register, the compiler could keep the value of c alive, avoiding the load before the second use (in the computation of c * 5). It cannot avoid the store to c because of the later use.

9. In FORTRAN, two variables can be forced to begin at the same storage location with an equivalence statement. For example, the following statement forces a and b to share storage:

```
equivalence (a,b)
```

Can the compiler keep a local variable in a register throughout the procedure if that variable appears in an equivalence statement? Justify your answer.

Answer: In general, the compiler cannot keep either a or b in a register if they appear together in an equivalence statement. More specifically, it cannot keep a in a register across an assignment to b (and vice-versa) because the assignment must change the value of both variables.

Typically, compilers handle this problem by keeping in memory any variable mentioned in an equivalence statement, as any other variable that might have an ambiguous name. (Examples include call-by-reference parameters and variables involved in pointer-based references.)

Section 5.5

10. Some part of the compiler must be responsible for entering each identifier into the symbol table.

    a. Should the scanner or the parser enter identifiers into the symbol table? Each has an opportunity to do so.

    Answer: Either the scanner or the parser can create the initial symbol-table entry for an identifier. The scanner can create the entry but cannot enter any information about its type. Depending on the syntax of the language, the parser may be able to enter type information. (For example, in C's declaration syntax, when the parser encounters an identifier in a declaration, it has already seen the syntax that assigns it a base type.)

b. Is there an interaction between this issue, declare-before-use rules, and disambiguation of subscripts from function calls in a language with the FORTRAN 77 ambiguity?

Answer: If the scanner has access to the symbol table, it can use type information from the symbol table to disambiguate certain kinds of ambiguity. For example, the scanner can return a different syntactic category for a function name than for an array name in FORTRAN, removing the ambiguity in `fee(a,b)`.

11. The compiler must store information in the IR version of the program that allows it to get back to the symbol table entry for each name. Among the options open to the compiler writer are pointers to the original character strings and subscripts into the symbol table. Of course, the clever implementor may discover other options.

    What are the advantages and disadvantages of each of these representations for a name? How would you represent the name?

    Answer: Pointers to the character string allow the various parts of the compiler to directly access the strings without further address calculations. This produces a lower overhead on access to names. Unless the compiler can guarantee that it only keeps one copy of each string, it cannot use the pointer for an equality test, so equality tests are expensive. Symbol table lookups would require a hash-computation on each lookup.

    Symbol table indices provide direct access to the information in the symbol table, without the need for hash-computation on lookup. They also provide for an inexpensive equality test, which pointers may not provide. Access to the actual name of the symbol (its lexeme) requires more work under this scheme than it would in an implementation that stored the pointer.

    To make this choice, the compiler should consider which form of access occurs more often: a symbol table access or an access to the lexeme. If space is not an overriding concern, the compiler writer might consider storing both the pointer and the index.

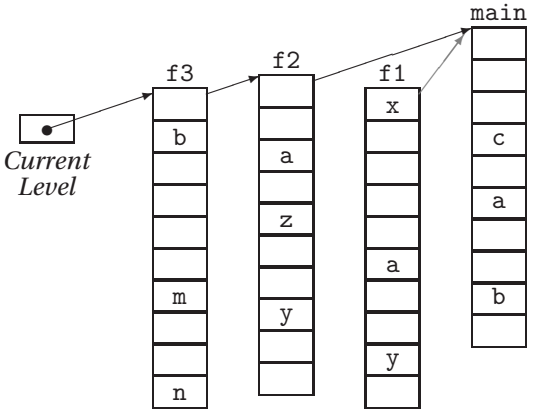
12. You are writing a compiler for a simple lexically-scoped language. Consider the example program shown in Figure 5.16.

    a. Draw the symbol table and its contents at line 11.

    Answer: Many answers are possible for this question, depending on the scheme that the student proposes for implementing a lexically scoped table. The simplest scheme is the "sheaf of tables" implementation, which might produce the following state. (`f1` has been discarded.)

```
 1   procedure main
 2      integer a, b, c;
 3      procedure f1(w,x);
 4         integer a,x,y;
 5         call f2(w,x);
 6         end;
 7      procedure f2(y,z)
 8         integer a,y,z;
 9         procedure f3(m,n)
10            integer b, m, n;
11            c = a * b * m * n;
12            end;
13         call f3(c,z);
14         end;
15      ...
16      call f1(a,b);
17      end;
```

FIGURE 5.16 Program for Exercise 12



b. What actions are required for symbol table management when the parser enters a new procedure and when it exits a procedure?

Answer: In the "sheaf of tables" implementation, entering a new procedure (or scope) entails creating a new table and linking it into the chain of tables. (The compiler sets the new table's chain to the contents of *Current Level* and then sets *Current Level* to point at the new table.) On exit from a procedure (or scope), it removes the first table from the

chain of tables (setting *Current Level* to the chain leading from the *Current Level* table to the next scope). The de-linked table may be freed; alternately, the compiler may keep it around for use in the debugger or later stages of compilation.

In other table organizations, the detailed actions will be different. However, the basic actions are: on entry to a scope, mark the "pre-scope" state of the table, and on exit from a scope, restore it to the appropriate "pre-scope" state.

13. The most common implementation technique for a symbol table uses a hash table, where insertion and deletion are expected to have **O**(1) cost.

    a. What is the worst-case cost for insertion and for deletion in a hash table?

    Answer: The worst case cost for insertion depends on the implementation. In an open-hashing table, the worst case cost might be **O**(1), if the insertion code does not perform a lookup first. If it does perform a lookup, the worst case for insertion becomes **O**(n), for a table that contains n names.

    In an open-addressing table, the worst-case cost will be **O**(n) for a table that contains n names.

    If deletion requires a lookup to find the record for the name being deleted, then the worst-case cost is **O**(n) for a table that contains n names.

    b. Suggest an alternative implementation scheme that guarantees **O**(1) insertion and deletion.

    Answer: See, for example, the sidebar labelled *An Alternative to Hashing* on page 256 of Chapter 5, or the discussion in § 2.6.3 of closure-free regular expressions that starts on page 77.

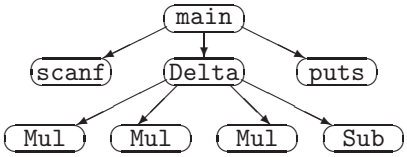
Section 6.2

1. Show the call tree and execution history for the following C program:

```
int Sub(int i, int j) {
   return i - j;
}
int Mul(int i, int j) {
   return i * j;
}
int Delta(int a, int b, int c) {
   return Sub(Mul(b,b), Mul(Mul(4,a),c));
}
void main() {
   int a, b, c, delta;
   scanf("%d %d %d", &a, &b, &c);
   delta = Delta(a, b, c);
   if (delta == 0)
     puts("Two equal roots");
   else if (delta > 0)
     puts("Two different roots");
   else
     puts("No root");
}
```

Answer:

Call Tree:



Note that `main` calls `puts` at most once, even though it has three distinct call sites that can invoke `puts`.

Execution History:

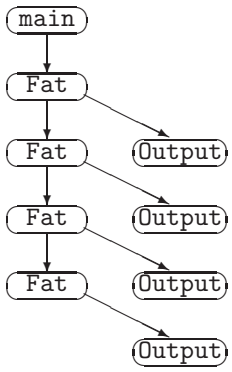
1. `main` calls `scanf`
2. `scanf` returns to `main`
3. `main` calls `Delta`
4. `Delta` calls `Mul`
5. `Mul` returns to `Delta`
6. `Delta` calls `Mul`
7. `Mul` returns to `Delta`
8. `Delta` calls `Mul`
9. `Mul` returns to `Delta`
10. `Delta` calls `Sub`
11. `Sub` returns to `Delta`
12. `Delta` returns to `main`
13. `main` calls `puts` (from 1 of 3 sites)
14. `puts` returns to `main`

2. Show the call tree and execution history for the following C program:

```
void Output(int n, int x) {
   printf("The value of %d! is %s.\n", n, x);
}
int Fat(int n) {
   int x;
     if (n > 1)
   x = n * Fat(n - 1);
   else
     x = 1;
   Output(n, x);
   return x;
}
void main() {
   Fat(4);
}
```

Answer:

Call Tree



Execution History:

1. `main` calls `Fat(4)`
2. `Fat(4)` calls `Fat(3)`
3. `Fat(3)` calls `Fat(2)`
4. `Fat(2)` calls `Fat(1)`
5. `Fat(1)` calls `Output`
6. `Output` returns to `Fat(1)`
7. `Fat(1)` returns to `Fat(2)`
8. `Fat(2)` calls `Output`
9. `Output` returns to `Fat(2)`
10. `Fat(2)` returns to `Fat(3)`
11. `Fat(3)` calls `Output`
12. `Output` returns to `Fat(3)`
13. `Fat(3)` returns to `Fat(4)`
14. `Fat(4)` calls `Output`
15. `Output` returns to `Fat(4)`
16. `Fat(4)` returns to `main`

Section 6.3

3. Consider the following Pascal program, in which only procedure calls and variable declarations are shown:

```
1   program Main(input, output);
2     var a, b, c : integer;
3     procedure P4; forward;
4     procedure P1;
5       procedure P2;
6         begin
7         end;
8       var b, d, f : integer;
9       procedure P3;
10        var a, b : integer;
11        begin
12          P2;
13          end;
14      begin
15        P2;
16        P4;
17        P3;
18        end;
19    var d, e : integer;
20    procedure P4;
21      var a, c, g : integer;
22      procedure P5;
23        var c, d : integer;
24        begin
25          P1;
26          end;
27      var d : integer;
28      begin
29        P1;
30        P5;
31        end;
32    begin
33      P1;
34      P4;
35      end.
```

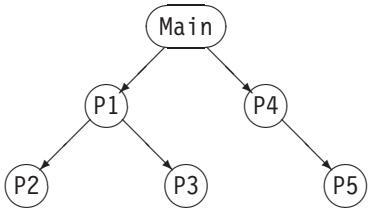
a. Construct a static coordinate table, similar to the one in Figure 6.3.

Answer:

Scope	a	b	c	d	e	f	g
Main	⟨1,0⟩	⟨1,4⟩	⟨1,8⟩	⟨1,12⟩	⟨1,16⟩	—	—
P1	⟨1,0⟩	⟨2,0⟩	⟨1,8⟩	⟨2,4⟩	—	⟨2,8⟩	—
P2	⟨1,0⟩	⟨1,4⟩	⟨1,8⟩	—	—	—	—
P3	⟨3,0⟩	⟨3,4⟩	⟨1,8⟩	⟨2,4⟩	—	⟨2,8⟩	—
P4	⟨2,0⟩	⟨1,4⟩	⟨2,4⟩	⟨1,12⟩	⟨1,16⟩	—	⟨2,8⟩
P5	⟨2,0⟩	⟨1,4⟩	⟨3,0⟩	⟨3,4⟩	⟨1,16⟩	—	⟨2,8⟩

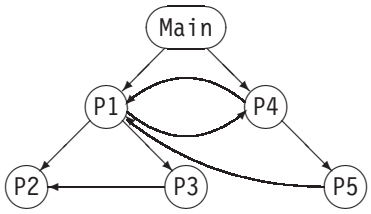
b. Construct a graph to show the nesting relationships in the program.

Answer:



c. Construct a graph to show the calling relationships in the program.

Answer:



4. Some programming languages allow the programmer to use functions in the initialization of local variables but not in the initialization of global variables.

   a. Is there an implementation rationale to explain this seeming quirk of the language definition?

      Answer: Most implementations use static initialization (with an assembly-language directive) to initialize global variables. Since a function must execute to return its value, this mechanism excludes any initial value that is not known at compile time.

   b. What mechanisms would be needed to allow initialization of a global variable with the result of a function call?

      Answer: If an implementation allowed a function call as the initializer of a global variable, it would need to defer initialization to runtime, when it could execute the function. It would also need to restrict the domain of the functions used in the initialization expression.

      For example, the following program is not well formed because the implementation cannot resolve the initial value of `fee`.

```
int fee = foe();

int foe() {
   return fee + 5;
}
```

5. The compiler writer can optimize the allocation of ARs in several ways. For example, the compiler might:

   a. Allocate ARs for leaf procedures statically.

   b. Combine the ARs for procedures that are always called together. (When α is called, it always calls β.)

   c. Use an arena-style allocator in place of heap allocation of ARs.

   For each scheme, consider the following questions:

   a. What fraction of the calls might benefit? In the best case? In the worst case?

      Answer: Estimating these fractions is tenuous, but that is part of the point of the question. People make sweeping statements about the benefits of various call-site optimizations without much data as to how often the phenomenon in question occurs.

      Leaf procedures: Estimating the fraction of calls that invoke leaf procedures is a tenuous business, at best. It can range from none (a program

can have no leaf procedures—think of recursive fibonnaci) to all of the calls.

Always called together: If p always calls q, then every call to p will benefit from combining p and q's ARs. However, the fraction of all calls that benefit will depend on how often p is called.

Arena allocation: This option benefits all calls to procedures implemented with heap-allocated ARs. If all procedures have heap-allocated ARs, then all calls benefit.

b. What is the impact on runtime space utilization?

Answer: Leaf procedures: This scheme requires one activation record, sized large enough to hold the largest AR of a leaf procedure. (Only one leaf procedure can be active at a given point in execution.) That storage is permanently allocated, so it forms a constant overhead on the program's space requirement.

The extra space can be bounded by the size of the statically allocated AR.

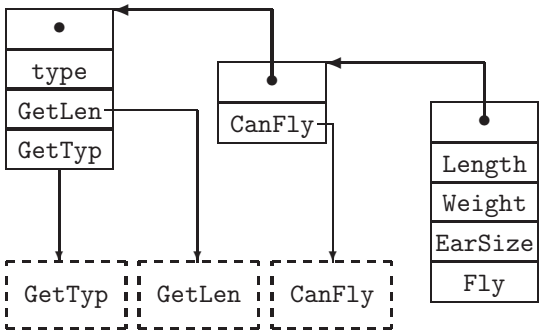
Always called together: Assume the p and q have their ARs merged and p always calls q. This scheme requires no extra space, since the original program would allocate ARs for both p and q.

Arena allocation: This scheme might require significant amounts of extra space. The arena allocators have cheap allocation and bulk deallocation. The latter may be a poor match for AR allocation, since the dynamic behavior of the program will determine when ARs may be freed. With a contrary program, deallocation is never possible, so all ARs remain allocated from the point of their creation until the end of execution.

6. Draw the structures that the compiler would need to create to support an object of type Dumbo, defined as follows:

```
class Elephant {
   private int Length;
   private int Weight;
   static int type;

   public int GetLen();
   public int GetTyp();
}

class Dumbo extends Elephant {
   private int EarSize;
   private boolean Fly;

   public boolean CanFly();
}
```

Answer:



Dashed boxes represent executable code

7. In a programming language with an open class structure, the number of method invocations that need runtime name resolution, or dynamic dispatch, can be large. A method cache, as described in Section 6.3.4, can reduce the runtime cost of these lookups by short-circuiting them.

   As an alternative to a global method cache, the implementation might maintain a single entry method cache at each call site—an inline method cache that records record the address of the method most recently dispatched from that site, along with its class.

   Develop pseudocode to use and maintain such an inline method cache. Explain the initialization of the inline method caches and any modifications to the general method lookup routine required to support inline method caches.

   Answer:

   The inline method cache consists of a pair ⟨*class tag*,*code*⟩. It is initialized to a known illegal value in the cache tag. Each method invocation is replaced with the following pseudo-code:

```
                                  if obj.class_tag = inline_cache.class_tag
                                     then inline_cache.code(obj, arguments)
                                  else {
                                     code ← lookup(obj.method)
                                     inline_cache.class_tag ← obj.class_tag
                                     inline_cache.code ← code
code ← lookup(obj.method)            inline_cache.code(obj, arguments)
invoke code(obj, arguments )      }
```

   Original Code | With Inline Cache

   As long as the general lookup routine returns a reference to the executable method is some compact form, such as a code pointer, this scheme should work. It would be good if the general lookup routine handled errors, such as an attempt to lookup a method that does not map to any known implementation. If, instead, that error check must be performed on the result returned by the lookup routine, it adds code to every method invocation, causing the code to be larger. (In either scheme, the code executes. If the code is replicated at each call, then the executable will have myriad copies of that check instead of one.)

```
1   procedure main;
2     var a : array[1...3] of int;
3         i : int;
4     procedure p2(e : int);
5       begin
6         e := e + 3;
7         a[i] := 5;
8         i := 2;
9         e := e + 4;
10        end;
11    begin
12      a := [1, 10, 77];
13      i := 1;
14      p2(a[i]);
15      for i := 1 to 3 do
16        print(a[i]);
17      end.
```

FIGURE 6.13 Program for Problem 8

Section 6.4

8. Consider the program written in Pascal-like pseudo code shown in Figure 6.13. Simulate its execution under call-by-value, call-by-reference, call-by-name, and call-by-value-result parameter binding rules. Show the results of the print statements in each case.

   Answer:

   Call-by-value should print 5, 10, and 77.

   Call-by-reference should print 9, 10, and 77.

   Call-by name should print 5, 14, and 77.

   Call-by-value-result should print 8, 10, and 77.

9. The possibility that two distinct variables refer to the same object (memory area) is considered undesirable in programming languages. Consider the following Pascal procedure, with parameters passed by reference:

```
procedure mystery(var x, y : integer);
  begin
    x := x + y;
    y := x - y;
    x := x - y;
  end;
```

If no overflow or underflow occurs during the arithmetic operations:

a. What result does `mystery` produce when it is called with two distinct variables, `a` and `b`?

   Answer: As long as no overflow or underflow occurs, `mystery` exchanges the values of `a` and `b`.

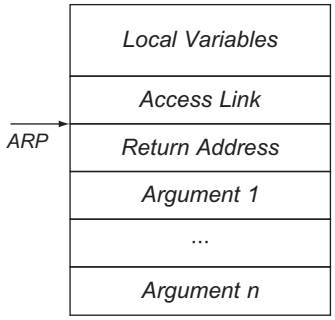
b. What would be the expected result if `mystery` is invoked with a single variable `a` passed to both parameters? What is the actual result in this case?

   Answer: Someone unfamiliar with the implementation of call-by-reference parameters might reasonably expect that a "swap" routine would leave its argument's value unchanged when it is passed the same name as both actual parameters. For example, the standard three-variable sequence (`t` $\leftarrow$ `x`, `x` $\leftarrow$ `y`, and `y` $\leftarrow$ `t`) has that behavior.

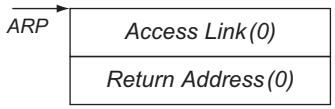
   However, `mystery` sets its actual parameter to zero when it is passed the same argument in both parameter positions, assuming that no overflow or underflow occurs.

```
1   program main(input, output);
2     procedure P1( function g(b: integer): integer);
3       var a: integer;
4       begin
5         a := 3;
6         writeln(g(2))
7         end;
8     procedure P2;
9       var a: integer;
10      function F1(b: integer): integer;
11        begin
12          F1 := a + b
13          end;
14      procedure P3;
15        var a:integer;
16        begin
17          a := 7;
18          P1(F1)
19          end;
20      begin
21        a := 0;
22        P3
23        end;
24    begin
25      P2
26      end.
```

(a) Example Pascal Program



(b) Activation Record Structure



(c) Initial Activation Record

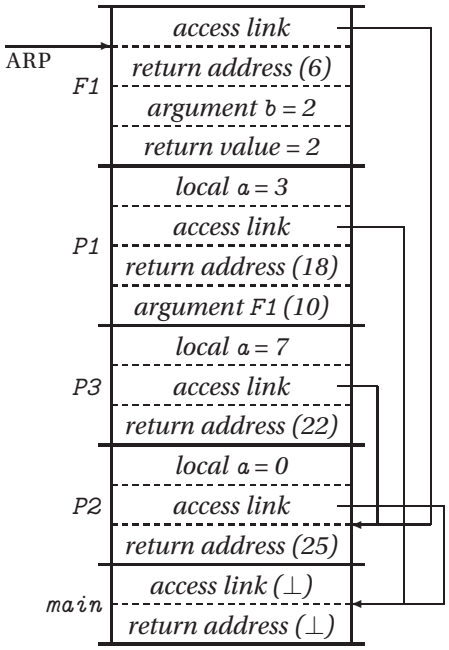
FIGURE 6.14 Program for Problem 10

Section 6.5

10. Consider the Pascal program shown in Figure 6.14a. Suppose that the implementation uses ARs as shown in Figure 6.14b. (Some fields have been omitted for simplicity.) The implementation stack allocates the ARs, with the stack growing toward the top of the page. The ARP is the only pointer to the AR, so access links are previous values of the ARP. Finally, Figure 6,14c shows the initial AR for a computation.

    For the example program in Figure 6.14a draw the set of its ARs just prior to the return from function F1. Include all entries in the ARs. Use line numbers for return addresses. Draw directed arcs for access links. Label the values of local variables and parameters. Label each AR with its procedure name.

Answer:


ARP
F1
access link
return address (6)
argument b = 2
return value = 2
P1
local a = 3
access link
return address (18)
argument F1 (10)
P3
local a = 7
access link
return address (22)
P2
local a = 0
access link
return address (25)
main
access link (⊥)
return address (⊥)

11. Assume that the compiler is capable of analyzing the code to determine facts such as "*from this point on, variable* v *is not used again in this procedure*" or "*variable* v *has its next use in line 11 of this procedure*," and that the compiler keeps all local variables in registers for the following three procedures:

```
procedure main
   integer a, b, c
   b = a + c;
   c = f1(a,b);
   call print(c);
   end;
procedure f1(integer x, y)
   integer v;
   v = x * y;
   call print(v);
   call f2(v);
   return -x;
   end;
procedure f2(integer q)
   integer k, r;
   ...
   k = q / r;
   end;
```

a. Variable x in procedure f1 is live across two procedure calls. For the fastest execution of the compiled code, should the compiler keep it in a caller-saves or callee-saves register? Justify your answer.

Answer: Keeping x in a callee-saves register would let the two callees, print and f2, determine whether or not to save its value. Were the compiler to assign x to a caller-saves register in f1, the linkage sequence would save its value and restore it at each call site, even if the callee did not use the full set of registers. Thus, callee-saves is (probably) the best choice for x.

Of course, if the compiler knew that both print and f2 saved their full complement of callee-saves registers, it might choose to keep x in a caller-saves register, save it before the call to print and restore it after the call to f2. This would produce fewer loads and stores, but requires knowledge about the behavior of both print and f2 during compilation of f1.

b. Consider variables a and c in procedure main. Should the compiler keep them in caller-saves or callee-saves registers, again assuming that the

compiler is trying to maximize the speed of the compiled code? Justify your answer.

Answer: Neither `a` nor `c` has a value that must be preserved across the call to `f1`. The value of `a` has no use after `main` evaluates it as an actual parameter.

The situation with `c` is slightly different. The use before the call to `f1` is disjoint from the definition and use after the call. However, no value passes from the first part to the second.

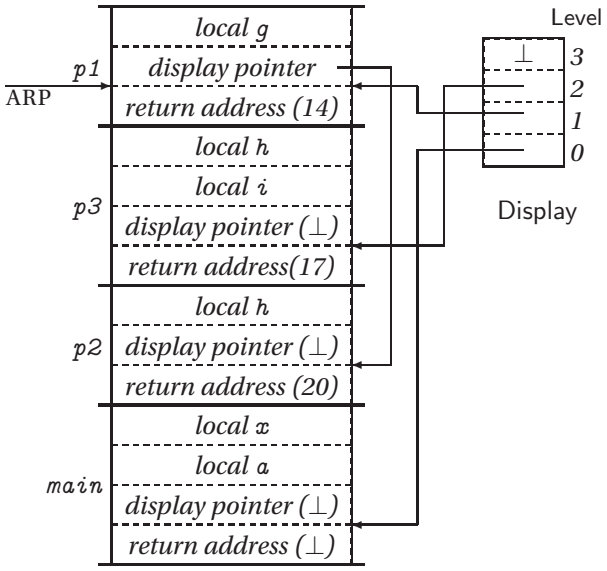
Since neither value must be preserved across the call, the compiler should assign them both to caller-saves registers. In this way, it can avoid the stores to save them and the loads to restore them.

12. Consider the following Pascal program. Assume that the ARs follow the same layout as in problem 10,with the same initial condition, *except* that the implementation uses a global display rather than access links.

```
 1  program main(input, output);
 2    var x : integer;
 3        a : float;
 4    procedure p1();
 5      var g:character;
 6      begin
 7        ...
 8        end;
 9    procedure p2();
10      var h:character;
11      procedure p3();
12        var h,i:integer;
13        begin
14          p1();
15          end;
16      begin
17        p3();
18        end;
19    begin
20      p2();
21      end
```

Draw the set of ARs that are on the runtime stack when the program reaches line 7 in procedure `p1`.

Answer:


local g
p1
display pointer
ARP
return address (14)
local h
local i
p3
display pointer (⊥)
return address(17)
local h
p2
display pointer (⊥)
return address (20)
local x
local a
main
display pointer (⊥)
return address (⊥)
Level
⊥
3
2
1
0
Display

Section 7.2

1. Memory layout affects the addresses assigned to variables. Assume that character variables have no alignment restriction, short integer variables must be aligned to halfword (2 byte) boundaries, integer variables must be aligned to word (4 byte) boundaries, and long integer variables must be aligned to doubleword (8 byte) boundaries. Consider the following set of declarations:

```
char a;
long int b;
int c;
short int d;
long int e;
char f;
```

Draw a memory map for these variables:

(a) Assuming that the compiler cannot reorder the variables

Answer:

a	1 byte
empty	7 bytes
b	8 bytes
c	4 bytes
d	2 bytes
empty	2 bytes
e	8 bytes
f	1 byte

(b) Assuming the compiler can reorder the variables to save space

Answer: Many answers are possible. The general algorithm would produce the layout shown on the left; a greedy rearrangement might produce the layout shown on the right.

b	8 bytes
e	8 bytes
c	4 bytes
d	2 bytes
a	1 byte
f	1 byte

a	1 byte
f	1 byte
d	2 bytes
c	4 bytes
b	8 bytes
e	8 bytes

2. As demonstrated in the previous question, the compiler needs an algorithm to lay out memory locations within a data area. Assume that the algorithm receives as input a list of variables, their lengths, and their alignment restrictions, such as

$$\langle \mathtt{a}, 4, 4\rangle, \langle \mathtt{b}, 1, 3\rangle, \langle \mathtt{c}, 8, 8\rangle, \langle \mathtt{d}, 4, 4\rangle, \langle \mathtt{e}, 1, 4\rangle, \langle \mathtt{f}, 8, 16\rangle, \langle \mathtt{g}, 1, 1\rangle.$$

The algorithm should produce, as output, a list of variables and their offsets in the data area. The goal of the algorithm is to minimize unused, or wasted, space.

(a) Write down an algorithm to lay out a data area with minimal wasted space.

Answer: The following pseudo-code assumes the presence of a global integer variable, *Offset*, and that it can destroy a copy of the list, stored in *List*. (It could, of course, mark list elements as processed rather than removing them from the list.)

```
Assign() {
    Offset ← 0
    NextSize ← FindBoundary( List )        // find largest size
    while (NextSize > 0)
        LayoutClass( List, NextSize )      // assign offsets
        NextSize ← FindBoundary( List )    //find next size
}

FindBoundary( List ) {
    boundary ← 0
    for each element, e, in List
        if alignment(e) > boundary
            then boundary ← alignment(e)
    return boundary
}

LayoutClass( List, Size ) {
    for each element, e, in List
        if alignment(e) = Size then
            assign name(e) the storage at Offset
            print name(e), Offset
            Offset ← Offset + Size
            remove element from List
}
```

(b) Apply your algorithm to the example list above and two other lists that you design to demonstrate the problems that can arise in storage layout.

Answer: For the list in the problem, the algorithm produces the following output

```
f,0, c,16, a,24, d,28, e,32, b,36, g,39
```

(c) What is the complexity of your algorithm?

Answer: The algorithm takes time proporational to the length of the list and the number of distinct alignment-boundaries present in the list. It makes two passes over the list for each distinct boundary size. In the worst case, each list element has a distinct boundary size and the algorithm will require $\mathbf{O}(N^2)$ where N is the number of list elements. In practice, the number of boundary sizes will be limited to five or six.

3. For each of the following types of variable, state where in memory the compiler might allocate the space for such a variable. Possible answers include registers, activation records, static data areas (with different visibilities), and the runtime heap.

   a. A variable local to a procedure

   Answer: The compiler can allocate it to a register or in the activation record.

   b. A global variable

   Answer: The compiler can allocate it in a static global data area.

   c. A dynamically allocated global variable

   Answer: The compiler can allocate it on the run-time heap.

   d. A formal parameter

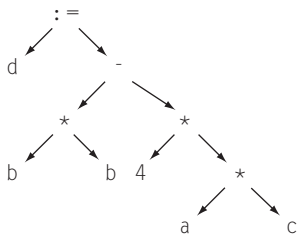
   Answer: The compiler can allocate it to a register or in the activation record.

   e. A compiler-generated temporary variable

   Answer: The compiler can allocate it to a register or in the activation record.

Section 7.3

4. Use the treewalk code-generation algorithm from Section 7.3 to generate naive code for the following expression tree. Assume an unlimited set of registers.

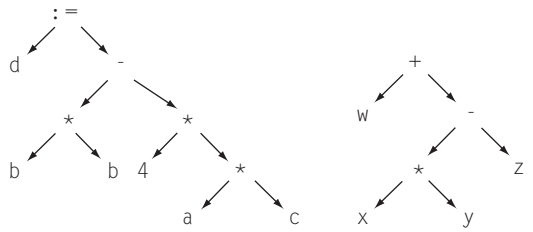


Answer: There are many possible answers. The one on the left follows the style of the book, while the one on the right is more concise.

```
loadI  @b      ⇒ r1
loadAO rarp,r1 ⇒ r2
mult   r2,r2   ⇒ r3
loadI  4       ⇒ r4
loadI  @a      ⇒ r5
loadAO rarp,r5 ⇒ r6
loadI  @c      ⇒ r7
loadAO rarp,r7 ⇒ f8
mult   r6,r8   ⇒ r9
mult   r4,r9   ⇒ r10
sub    r3,r10  ⇒ r11
loadI  @d      ⇒ r12
store  r11     ⇒ r12
```

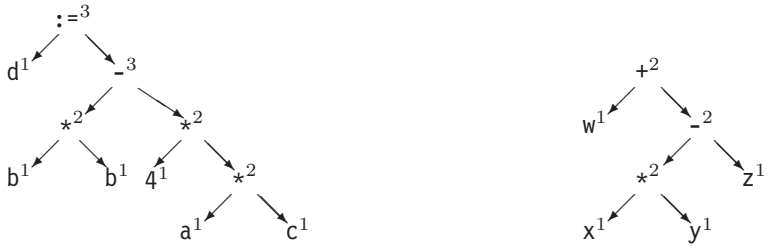
```
loadAI rarp,@b ⇒ r1
mult   r1,r1   ⇒ r2
loadI  4       ⇒ r3
loadAI rarp,@a ⇒ r4
loadAI rarp,@c ⇒ r5
mult   r4,r5   ⇒ r6
mult   r3,r6   ⇒ r7
sub    r2,r7   ⇒ r8
loadI  @d      ⇒ r9
store  r8      ⇒ r9
```

5. Find the minimum number of registers required to evaluate the following trees using the ILOC instruction set. For each nonleaf node, indicate which of its children must be evaluated first in order to achieve this minimum number of registers.



(a) (b)

Answer: Nodes are superscripted with the number of registers required to evaluate the subtree rooted at that node.



(a) (b)

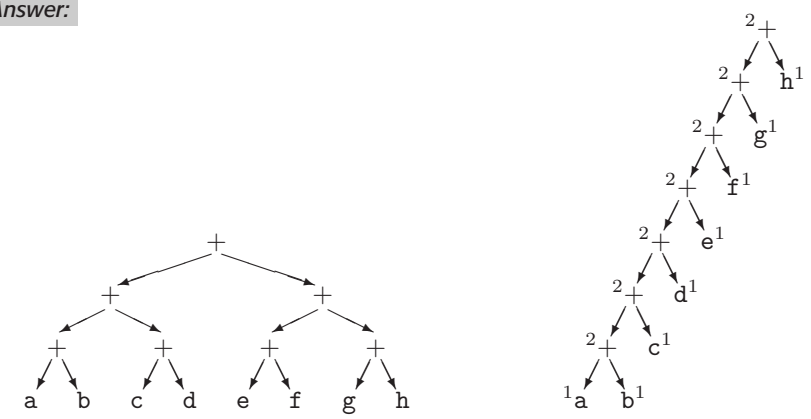
In each case, the code should evaluate the more demanding subtree—the one requiring more registers—first. In tree (a), notice that an immediate multiply would eliminate the register needed to hold 4, but would not decrease the overall demand for registers.

6. Build expression trees for the following two arithmetic expressions, using standard precedence and left-to-right evaluation. Compute the minimum number of registers required to evaluate each of them using the ILOC instruction set.

   (a) `((a + b) + (c + d)) + ((e + f) + (g + h))`

   (b) `a + b + c + d + e + f + g + h`

Answer:



(a) (b)

Section 7.4

7. Generate predicated ILOC for the following code sequence. (No branches should appear in the solution.)

```
if (x < y)
   then z = x * 5;
   else z = y * 5;
w = z + 10;
```

Answer:

```
          cmp_LT  rx,ry  ⇒ r1
          not     r1     ⇒ r2
(r1)?     loadI   5      ⇒ r3
(r1)?     mult    rx,r3  ⇒ rz
(r2)?     loadI   5      ⇒ r3
(r1)?     mult    ry,r3  ⇒ rz
          loadI   10     ⇒ r4
          add     rz,r4  ⇒ rw
```

Note that the immediate load into r_3 is identical on both paths. The predicate could be dropped and one fewer operation generated. One way to catch this inefficiency is *code hoisting* (see § 10.3.2).

8. As mentioned in Section 7.4, short-circuit code for the following expression in C avoids a potential division-by-zero error:

```
a != 0 && b / a > 0.5
```

If the source-language definition does not specify short-circuit evaluation for boolean-valued expressions, can the compiler generate short-circuit code as an optimization for such expressions? What problems might arise?

Answer: If short-circuit code is generated as an optimization and the source-language definition requires that the compiler preserve exceptions, the optimization cannot be applied to any expression that might generate an exception. In the expression shown above, the subexpression `b / a` can generate an exception, so the short-circuit code will (illegally) alter the behavior when `a` is zero.

Section 7.5

9. For a character array `A[10...12,1...3]` stored in row-major order, calculate the address of the reference `A[i,j]`, using at most four arithmetic operations in the generated code.

Answer: Using the "false zero" scheme, $@A_0$ becomes `@A - 31`, so the address of `A[i,j]` is $@A_0 + i \times 3 + j$.

10. What is a dope vector? Give the contents of the dope vector for the character array in the previous question. Why does the compiler need a dope vector?

Answer: A dope vector is a structure that the compiler builds to pass information about an array's size at a call site. The dope vector typically contains the array's base address and the sizes of each dimension. It may, depending on language issues, need to contain the number of dimensions, too.

For the array `A[10...12,1...3]` in the previous exercise, the dope vector would contain $\langle @A_0, 3, 3 \rangle$. If the implementation needed the number of dimensions, then the dope vector would be $\langle @A_0, 2, 3, 3 \rangle$

Dope vectors are necessary because a single procedure can be invoked from many different call sites with different array-valued actual parameters. Using a dope-vector lets the callee generate appropriate code without knowing the size of each dimension or, in some cases, the number of such dimensions.

11. When implementing a C compiler, it might be advisable to have the compiler perform range checking for array references. Assuming range checks are used and that all array references in a C program have successfully passed them, is it possible for the program to access storage outside the range of an array, for example, accessing A[-1] for an array declared with lower bound zero and upper bound N?

    Answer: *This question is not a particularly useful question. It will be replaced in EaC3e.*

    Range checking is useful for debugging and to detect unexpected or malicious behavior. Good techniques exist to decrease the runtime cost of range checks; see, for example, reference [257] in the bibliography.

    In an implementation that includes range checking, a program can use pointer arithmetic to access storage outside the valid range of an array. For example, with an array declared as A[0:N], the code could load the value from the storage that would be A[-1] into i with a code sequence such as:

```
int i, *p;
p = &A[0];
p-;
i = *p;
```

Section 7.6

12. Consider the following character-copying loop from Section 7.6.2:

```
do {
  *a++ = *b++;
} while (*b!='\0')
```

```
    loadI  @b     ⇒ r@b   // get pointers
    loadI  @a     ⇒ r@a
    loadI  NULL   ⇒ r1    // terminator
L1: cload  r@b    ⇒ r2    // get next char
    cstore r2     ⇒ r@a   // store it
    addI   r@b, 1 ⇒ r@b   // bump pointers
    addI   r@a, 1 ⇒ r@a
    cmp_NE r1, r2 ⇒ r4
    cbr    r4     → L1, L2
L2: nop                   // next stmt
```

    Modify the code so that it branches to an error handler at L_{sov} on any attempt to overrun the allocated length of a. Assume that the allocated length of a is stored as an unsigned four-byte integer at an offset of –8 from the start of a.

Answer:

```
        loadI  @b          ⇒ r@b     // get pointers
        loadI  @a          ⇒ r@a
        loadAI r@a,-8      ⇒ rlen(a) // bound on a
        add    r@a,rlen(a) ⇒ rub
        loadI  NULL        ⇒ r1      // terminator
   L1:  cload  r@b         ⇒ r2      // get next char
        cstore r2          ⇒ r@a     // store it
        addI   r@b,1       ⇒ r@b     // bump pointers
        addI   r@a,1       ⇒ r@a
        cmp_GT r@a,rub     ⇒ r4      // check overflow
        cbr    r4                    → Lsov,L2
        cmp_NE r1,r2       ⇒ r5
        cbr    r5          → L1,L3
   L3:  nop                         // next stmt
```

This solution introduces a second compare and branch into the loop body. If branch overhead is a concern, and it should be, the following version might be worth generating.

```
        loadI  @b          ⇒ r@b     // get pointers
        loadI  @a          ⇒ r@a
        loadAI r@a,-8      ⇒ rlen(a) // bound on a
        add    r@a,rlen(a) ⇒ rub
        loadI  NULL        ⇒ r1      // terminator
   L1:  cload  r@b         ⇒ r2      // get next char
        cstore r2          ⇒ r@a     // store it
        addI   r@b,1       ⇒ r@b     // bump pointers
        addI   r@a,1       ⇒ r@a
        cmp_LE r@a,rub     ⇒ r4      // check overflow
        cmp_NE r1,r2       ⇒ r5      // check terminator
        and    r4,r5       ⇒ r6      // combine them
        cbr    r6          → L1,L2
   L2:  cbr    r4          → L3,Lsov // overflow exit?
   L3:  nop                         // next stmt
```

This version has the same number of operations inside the loop, but replaces the `cbr` for overflow with a logical `and`. On exit from the loop, at L_2, the code tests to determine if the exit occurred because the loop encountered a `NULL` or because the loop tried to overrun the allocated length of `a`. Finally, the first statement after the loop has moved to L_3.

13. Arbitrary string assignments can generate misaligned cases.

    (a) Write the ILOC code that you would like your compiler to emit for an arbitrary PL/I-style character assignment, such as

    ```
    fee(i:j) = fie(k:l);
    ```

    where `j-i = l-k`. This statement copies the characters in `fie`, starting at location `k` and running through location `l` into the string `fee`, starting at location `i` and running through location `j`.

    Include versions using character-oriented memory operations and versions using word-oriented memory operations. You may assume that `fee` and `fie` do not overlap in memory.

    Answer: Assume that the values of `i`, `j`, `k`, and `l` are already in local variables with the obvious names, r_i, r_j, r_k, and r_l. We assume that (`j - i`) = (`k - l`), so the code will not check that condition.

    ```
            sub       rj, ri          ⇒ rcount
            loadI     0               ⇒ rzero
            loadI     0               ⇒ roffset
            loadI     @fee            ⇒ r@fee
            loadI     @fie            ⇒ r@fie
            cmp       rcount,rzero    ⇒ r1
            cbr_LE    r1              → Lexit
    L0:     cloadAI   rfee, roffset   ⇒ rchar
            cstoreAI  rchar           ⇒ rfie, roffset
            addI      roffset, 1      roffset
            subI      rcount, 1       rcount
            cmp       rcount,rzero    ⇒ r1
            cbr_GT    r1              → L0
    Lexit:  nop
    ```

    Character-oriented operations

    Improvements on this code are possible. For example, we could shift the loop-counter tests and branches from r_{count} to r_{offset} (see, for example, the discussion of *linear function test replacement* in § 10.7.2, starting on page 589).

    To implement the same functionality with word oriented operations, we need an initial loop to read characters up to a word boundary, then an loop that loads a word, adjusts its alignments, and writes a word.

The version that uses full-word load and store is only slightly more complex, if we assume that load and store can access arbitrarily-aligned locations in memory. With that assumption, the code simply performs full-word load and store operations until less than a word remains. At that point, it performs one, two, or three character-oriented operations.

The truly complex case occurs with full-word load and store operations that only work on word-aligned addresses. We will provide ILOC code for that case in the next release of the solutions — the set that includes all thirteen chapters.

(b) The programmer can create character strings that overlap. In PL/I, the programmer might write

```
fee(i:j) = fee(i+1:j+1);
```

or, even more diabolically,

```
fee(i+k:j+k) = fee(i:j);
```

How does this complicate the code that the compiler must generate for the character assignment?

Answer: First, the language must define what behavior is expected from assignments involving overlapping storage. We will assume that the language expects overlapped assignment to behave in the naïve way, that is, to read data from a location before writing new data to it.

(a). The easiest way, albeit the least efficient, would be to copy the source to an internally allocated buffer, and then to copy the data from the buffer to its destination location.

(b). The more complex version breaks into two separate cases: the destination string runs ahead of the source string, which is easy; and the destination string runs behind the source string (so that we must defer writing to it).

In the first case, we can use the character-copying loop and let it run. It should work. If the distance between the start of the source and destination is large enough, the word-oriented version will also work.

In the second case, we need to buffer the characters in the overlap region. The simplest approach is to use an internally-allocated, full-size buffer, as in (a).

(c) Are there optimizations that the compiler could apply to the various character-copying loops that would improve runtime behavior? How would they help?

Answer: If the target machine supports a *load multiple*/*store multiple* instruction pair, then the loops might do better by clearing out a set of adjacent registers and using the multiword memory operations.

Because the loops contain a fair amount of code, the compiler might reduce code size by generating calls to a library routine, such as the Linux `memcopy` and `memmove` routines. Those library implementations should be optimized for the target machine.

Section 7.7

14. Consider the following type declarations in C:

```
struct S2 {
  int i;
  int f;
};

union U {
   float r;
   struct S2;
};

struct S1 {
   int a;
   double b;
   union U;
   int d;
};
```

Build a structure-element table for S1. Include in it all the information that a compiler would need to generate references to elements of a variable of type S1, including the name, length, offset, and type of each element.

Answer:

Name	*Length*	*Offset*	*Type*
a	4	0	int
b	8	4	double
r	4	12	float
i	4	12	int
f	4	16	int
d	4	20	int

Structure Element Table for S1

This table assumes that the compiler introduces no padding between structure elements. In practice, the compiler might add padding between a and b to ensure doubleword alignment of b (required on some machines). Such padding would, of course, change the offsets of many of the names.

15. Consider the following declarations in C:

```
struct record {
  int StudentId;
  int CourseId;
  int Grade;
} grades[1000];
int g, i;
```

Show the code that a compiler would generate to store the value in variable `g` as the grade in the i^{th} element of `grades`, assuming the following:

(a) The array `grades` is stored as an array of structures.

Answer:

```
multI r_i,12       ⇒ r_1   // record is 12 bytes long
addI  r_1,8        ⇒ r_2   // offset of Grade is 8
addI  r_2,@grades  ⇒ r_3   // add base address
store r_g          ⇒ r_3
```

(b) The array `grades` is stored as a structure of arrays.

Answer:

```
loadI @grades ⇒ r_1   // get base address
addI  r_1,8000 ⇒ r_2   // start of grade array
multI r_i,4    ⇒ r_3   // offset of ith element
add   r_2,r_3  ⇒ r_4   // add base address
store r_g      ⇒ r_4
```

Section 7.8

16. As a programmer, you are interested in the efficiency of the code that you produce. You recently implemented, by hand, a scanner. The scanner spends most of its time in a single `while` loop that contains a large case statement.

    (a) How would the different case statement implementation techniques affect the efficiency of your scanner?

    Answer:

    Linear Search: The order in which the case labels appear may greatly affect the efficiency of the scanner. If there are a large number of cases that appear with roughly equal frequency, linear search may produce worse results than the other implementation techniques.

    Binary Search: This technique provides roughly equal access times for all the labels. If there are a large number of equi-probable labels, and they form a sparse set, then this technique is a good choice.

    Jump Table: In this case, the compiler arranges for the code to directly compute an offset into a table of labels, sometimes called a *jump table*. If the set of labels forms a dense set, this technique can eliminate many of the table lookups required by the binary search method (without wasting space). The case lookup takes constant time, but requires a memory reference and a jump to register. For extremely small label sets, linear search may be faster. (In general, this technique is faster than the other methods, but it requires a dense set of labels.)

    (b) How would you change your source code to improve the runtime performance under each of the case statement implementation strategies?

    Answer:

    Linear Search: Since the compiler tests the labels in order of appearance, the programmer should rearrange them so that commonly occurring labels appear first, with rarely used labels appearing at the end.

    Binary Search: Since the compiler must reorder the labels into lexicographic order, the order in which they appear in the code is irrelevant to the actual performance.

    Jump Table: This technique has a constant cost per execution, so it is insensitive to the order in which the labels appear in the source code.

17. Convert the following C tail-recursive function to a loop:

```
List * last(List *l) {
  if (l == NULL)
     return NULL;
  else if (l->next == NULL)
     return l;
  else
     return last(l->next); }
```

Answer:

```
List * last(List *l) {
  while(1) {
     if (l == NULL)
        break;
     else if (l->next == NULL)
        break;
     else l = l->next;
     }
  return l;
}
```

Section 7.9

18. Assume that *x* is an unambiguous, local, integer variable and that *x* is passed as a call-by-reference actual parameter in the procedure where it is declared. Because it is local and unambiguous, the compiler might try to keep it in a register throughout its lifetime. Because it is passed as a call-by-reference parameter, it must have a memory address at the point of the call.

    (a) Where should the compiler store *x*?

    Answer: Ideally, the compiler should place x in a caller-saves register, as long as there are enough such registers to accommodate x. In a caller-saves register, it will be spilled at the call.

    (b) How should the compiler handle *x* at the call site?

    Answer: If x is in a caller-saves register, the compiler should emit the code to save x into the register save area of the current AR. When it evaluates x as an actual parameter, it can use the address in the register save area as x's memory address. After the call, it should restore x from the register save area. (There is no need to create a second copy of x for the call site.)

    (c) How would your answers change if *x* was passed as a call-by-value parameter?

    Answer: With a call-by-value parameter, the calling sequence needs a value for x rather than an address, so the sequence need not (necessarily) store x. Thus, the bias in favor of a caller-saves register is removed. The compiler should assign x to a register based on whatever other criteria it uses to select caller-saves versus callee-saves registers.

    To evaluate x as an argument, it can copy the value from x into the appropriate register or AR location. There is no need to restore the value after the call.

19. The linkage convention is a contract between the compiler and any outside callers of the compiled code. It creates a known interface that can be used to invoke a procedure and obtain any results that it returns (while protecting the caller's runtime environment). Thus, the compiler should only violate the linkage convention when such a violation cannot be detected from outside the compiled code.

    (a) Under what circumstances can the compiler be certain that using a variant linkage is safe? Give examples from real programming languages.

    Answer: If the procedure cannot be invoked from outside the current unit of compilation, then the compiler can safely use a non-standard linkage. Examples include `static` procedures in C and nested local procedures in Pascal. Since these procedures cannot be named outside the compilation unit, they cannot be invoked from outside.

    (b) In these circumstances, what might the compiler change about the calling sequence and the linkage convention?

    Answer: The compiler can change almost anything in such a case. For example, it might pass all of the parameters to a procedure in registers. It might collect information about the number of registers used in each procedure in the compilation unit and use that knowledge to avoid unnecessary register save & restore code at some calls. It might merge the ARs of some of the procedures, saving a few operations involved in allocation and deallocation of the ARs.

Section 8.4

1. Apply the algorithm from Figure 8.4 to each of the following blocks:

$$
\begin{array}{lll}
t_1 & \leftarrow & a + b \\
t_2 & \leftarrow & t_1 + c \\
t_3 & \leftarrow & t_2 + d \\
t_4 & \leftarrow & b + a \\
t_5 & \leftarrow & t_3 + e \\
t_6 & \leftarrow & t_4 + f \\
t_7 & \leftarrow & a + b \\
t_8 & \leftarrow & t_4 - t_7 \\
t_9 & \leftarrow & t_8 * t_6
\end{array}
$$

Block b_0

$$
\begin{array}{lll}
t_1 & \leftarrow & a \times b \\
t_2 & \leftarrow & t_1 \times 2 \\
t_3 & \leftarrow & t_2 \times c \\
t_4 & \leftarrow & 7 + t_3 \\
t_5 & \leftarrow & t_4 + d \\
t_6 & \leftarrow & t_5 + 3 \\
t_7 & \leftarrow & t_4 + e \\
t_8 & \leftarrow & t_6 + f \\
t_9 & \leftarrow & t_1 + 6
\end{array}
$$

Block b_1

Answer: Value numbers are shown as superscripts. An asterisk before an operation indicates that it should be replaced with the operation listed to its right.

$$
\begin{array}{lll}
 & t_1{}^3 \leftarrow a^1 + b^2 & \\
 & t_2{}^5 \leftarrow t_1{}^3 + c^4 & \\
 & t_3{}^7 \leftarrow t_2{}^5 + d^6 & \\
* & t_4{}^3 \leftarrow b^2 + a^1 & t_4{}^3 \leftarrow t_1{}^3 \\
 & t_5{}^9 \leftarrow t_3{}^7 + e^8 & \\
 & t_6{}^{11} \leftarrow t_4{}^3 + f^{10} & \\
* & t_7{}^3 \leftarrow a^1 + b^2 & t_7{}^3 \leftarrow t_1{}^3 \\
* & t_8{}^{12} \leftarrow t_4{}^3 - t_7{}^3 & t_8{}^{12} \leftarrow 0^{12} \\
* & t_9{}^{13} \leftarrow t_8{}^{12} * t_6{}^{11} & t_9{}^{13} \leftarrow 0^{12}
\end{array}
$$

Block b_0

$$
\begin{aligned}
t_1{}^3 &\leftarrow a^1 \times b^2 \\
t_2{}^5 &\leftarrow t_1{}^3 \times 2^4 \\
t_3{}^7 &\leftarrow t_2{}^5 \times c^6 \\
t_4{}^9 &\leftarrow 7^8 + t_3{}^7 \\
t_5{}^{11} &\leftarrow t_4{}^9 + d^{10} \\
t_6{}^{13} &\leftarrow t_5{}^{11} + 3^{12} \\
t_7{}^{15} &\leftarrow t_4{}^9 + e^{14} \\
t_8{}^{17} &\leftarrow t_6{}^{13} + f^{16} \\
t_9{}^{19} &\leftarrow t_1{}^3 + 6^{18}
\end{aligned}
$$

Block b_1

Local value numbering finds no improvements in block b_1.

2. Consider a basic block, such as b_0 or b_1 in question 1 above. It has n operations, numbered 0 to $n-1$.

   a. For a name x, USES(x) contains the index in b of each operation that uses x as an operand. Write an algorithm to compute the USES set for every name mentioned in block b. If $x \in$ LIVEOUT(b), then add two dummy entries ($> n$) to USES(x).

   Answer: Assumptions: (1) We have a hash table, as in local value numbering. (2) The *lookup()* routine creates an entry for its argument if no entry exists. (3) *Uses* is a field in the hash table. (4) *LeftOp(i)* returns the first operand in operation *i*. *RightOp(i)* returns the second operand. *Target(i)* returns the name defined by operation *i*.

```
initialize the hash table
for i ← 0 to n-1 do
    index ← lookup(LeftOp(i))
    add i to Uses[index]

    index ← lookup(RightOp(i))
    add i to Uses[index]

for each i in the block's LIVEOUT set do
    index ← lookup(i)
    add n+1 to Uses[index]
    add n+2 to Uses[index]
```

b. Apply your algorithm to blocks b_0 and b_1 above.

Answer: To make the drawing understandable, assume that *lookup* allocates space in the table in a simple way. The first entry is at index 0, the second at 1, and so on. Since the question supplies no LIVEOUT sets, we will assume that LIVEOUT is empty.

The algorithm produces the following table for block b_0:

Index	Name	Uses
0	a	1, 4, 7
1	b	1, 4, 7
2	t_1	2,
3	c	2,
4	d	3
5	t_2	3
6	t_3	5
7	t_4	6, 8

Index	Name	Uses
8	e	5
9	t_5	
10	f	6
11	t_6	9
12	t_7	8
13	t_8	9
14	t_9	

Applied to block b_1, the algorithm produces the following table:

Index	Name	Uses
0	a	1
1	b	2
2	t_1	2, 9
3	t_2	3
4	c	3
5	t_3	4
6	t_4	5, 7
7	d	5

Index	Name	Uses
8	t_5	6
9	t_6	8
10	e	7
11	t_7	
12	f	8
13	t_8	
14	t_9	

Note that the algorithm is defined for *names*. For that reason, we did not create entries for the literal constants.

c. For a reference to x in operation i of block b, DEF(x,i) is the index in b where the value of x visible at operation i was defined. Write an algorithm to compute DEF(x,i) for each reference x in b. If x is upward exposed at i, then DEF(x,i) should be -1.

Answer: We will assume a hash table, as in part (a) of the question, with an additional field, `LastDef`. When `lookup` creates an entry, it initializes `LastDef` to -1.

```
initialize the hash table
for i ← 0 to n-1 do
    index ← lookup(LeftOp(i))                    DEF(LeftOp(i),i)
    mark LeftOp(i) with LastDef(index)             ← LastDef(index)
    index ← lookup(RightOp(i))                   DEF(RightOp(i),i)
    mark RightOp(i) with LastDef(index)            ← LastDef(index)
    index ← lookup(Target(i))
    LastDef(index) ← i
```

d. Apply your algorithm to blocks b_0 and b_1 above.

Answer: The DEF information is harder to represent, because x in DEF(x,i) refers to a specific reference—that is, a name and an operation. To simplify the issue, we will annotate the code with numerical superscripts. The superscript refers to the operation number in which the value is defined. Thus, c^2 indicates that this specific reference to c uses a value defined in operation 2. The superscript -1 indicates an upward exposed use.

1	t_1	←	a^{-1} + b^{-1}	1	t_1	←	a^{-1} × b^{-1}
2	t_2	←	$t_1{}^1$ + c^{-1}	2	t_2	←	$t_1{}^1$ × 2
3	t_3	←	$t_2{}^2$ + d^{-1}	3	t_3	←	$t_2{}^2$ × c^{-1}
4	t_4	←	b^{-1} + a^{-1}	4	t_4	←	7 + $t_3{}^3$
5	t_5	←	$t_3{}^3$ + e^{-1}	5	t_5	←	$t_4{}^4$ + d^{-1}
6	t_6	←	$t_4{}^4$ + f^{-1}	6	t_6	←	$t_5{}^5$ + 3
7	t_7	←	a^{-1} + b^{-1}	7	t_7	←	$t_4{}^4$ + e^{-1}
8	t_8	←	$t_4{}^4$ - $t_7{}^7$	8	t_8	←	$t_6{}^6$ + f^{-1}
9	t_9	←	$t_8{}^8$ * $t_6{}^6$	9	t_9	←	$t_1{}^1$ + 6

Block b_0 Block b_1

3. Apply the tree-height balancing algorithm from Figures 8.7 and 8.8 to the two blocks in problem 1. Use the information computed in problem 2b above. In addition, assume that LIVEOUT(b_0) is $\{t_3, t_9\}$, that LIVEOUT(b_1) is $\{t_7, t_8, t_9\}$, and that the names a through f are upward-exposed in the blocks.

Answer: For block b_0, we first compute the USES set for each name, as shown in the table on the left below. Notice that t_5 is unused and that we have used an asterisk to indicate an arbitrary large integer—added to the USES set of any name in LIVEOUT.

t_5 is unused, or dead. Watch its progress through the algorithm. Similarly, t_4 and t_7 are identical and redundant with t_1 which is not a root. The algorithm performs tree-height balancing, not common subexpression elimination.

The next step is to find the roots of subexpressions that must be balanced and scheduled. The algorithm enqueues the ⟨*root*,*priority*⟩ pairs ⟨t_3,+⟩, ⟨t_4,+⟩, ⟨t_6,+⟩, ⟨t_7,+⟩, ⟨t_8,-⟩, and ⟨t_9,×⟩. The roots and their trees are shown in the table to the right below.

#	Uses	Code
1	2	$t_1 \leftarrow a + b$
2	3	$t_2 \leftarrow t_1 + c$
3	*, *, 5	$t_3 \leftarrow t_2 + d$
4	6, 8	$t_4 \leftarrow b + a$
5	—	$t_5 \leftarrow t_3 + e$
6	9	$t_6 \leftarrow t_4 + f$
7	8	$t_7 \leftarrow a + b$
8	9	$t_8 \leftarrow t_4 - t_7$
9	*, *	$t_9 \leftarrow t_8 * t_6$

Root	Tree
t_9	(× t_8, t_6)
t_3	(+ a, b, c, d)
t_4	(+ a, b)
t_6	(+ t_4, f)
t_7	(+ a, b)
t_8	(- t_4, t_7)

Next, the algorithm pulls trees out of the queue, in precedence order. It flattens the tree and rebuilds it. If it encounters a root in this process, it recurs on the root to ensure that the result of that subtree is available in the current computation.

For block b_0. the first subtree dequeued is t_9. It recurs on t_8 (which recurs on t_4 and t_7) and t_6 (which recurs on t_4 and discovers that it has already been rebuilt). Finally, it dequeues t_3 and processes it. The algorithm produces the following code:

	Generated Code
1	$t_4 \leftarrow a + b$
2	$t_7 \leftarrow a + b$
3	$t_8 \leftarrow t_4 - t_7$
4	$t_6 \leftarrow t_4 + f$
5	$t_9 \leftarrow t_8 \times t_6$
6	$temp_0 \leftarrow a + b$
7	$temp_1 \leftarrow c + d$
8	$t_3 \leftarrow temp_0 + temp_1$

A good scheduler could issue operations (1, 2, 6, and 7) in parallel, (3, 4, and 8) in parallel, and finish the block with operation 5.

For block b_1, we first compute the USES set for each name, as shown in the table to the left below. Again, we have used an asterisk to indicate an arbitrary large integer—added to the USES set of any name in LIVEOUT.

The next step is to find the roots of subexpressions that must be balanced and scheduled. The algorithm enqueues the ⟨*root,priority*⟩ pairs $\langle t_1,\times\rangle$, $\langle t_3,\times\rangle$, $\langle t_4,+\rangle$, $\langle t_7,+\rangle$, $\langle t_8,+\rangle$, and $\langle t_9,+\rangle$. The roots and their trees are shown in the table to the right below.

#	Uses	Code
1	2, 9	$t_1 \leftarrow a \times b$
2	3	$t_2 \leftarrow t_1 \times 2$
3	4	$t_3 \leftarrow t_2 \times c$
4	5, 7	$t_4 \leftarrow 7 + t_3$
5	6	$t_5 \leftarrow t_4 + d$
6	8	$t_6 \leftarrow t_5 + 3$
7	*, *	$t_7 \leftarrow t_4 + e$
8	*, *	$t_8 \leftarrow t_6 + f$
9	*, *	$t_9 \leftarrow t_1 + 6$

Root	Tree
t_1	$(\times a, b)$
t_3	$(\times t_1, 2, c)$
t_4	$(+ 7, t_3)$
t_7	$(+ t_4, e)$
t_8	$(+ t_4, d, 3, f)$
t_9	$(+ t_1, 6)$

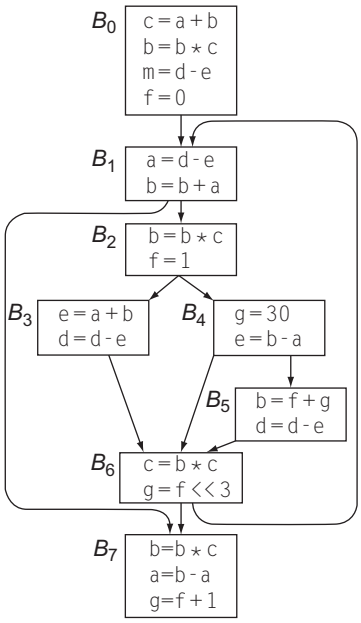
Next, the algorithm pulls trees out of the queue, in precedence order. It flattens the tree and rebuilds it. If it encounters a root in this process, it recurs on the root to make the result of that subtree available.

For block b_1, the first subtree dequeued is t_1. Next, the algorithm dequeues t_3, which recurs on t_1. It then dequeues the addition subtrees, t_4, t_7, t_8, and t_9. That process produces the following code:

	Generated Code
1	$t_1 \leftarrow a \times b$
2	$temp_0 \leftarrow 2 \times c$
3	$t_3 \leftarrow t_1 \times temp_0$
4	$t_4 \leftarrow 7 + t_3$
5	$t_7 \leftarrow t_4 + e$
6	$temp_1 \leftarrow t_4 + d$
7	$temp_2 \leftarrow 3 + f$
8	$t_8 \leftarrow temp_1 + temp_2$
9	$t_9 \leftarrow t_1 + 6$

Section 8.5

4. Consider the following control-flow graph:



a. Find the extended basic blocks and list their distinct paths.

Answer: The EBBs are $\{A\}$, $\{B, C, D, E, F\}$, $\{G\}$, and $\{H\}$.

b. Apply local value numbering to each block.

Answer:

B_0:	$c_3 = a_1 + b_2$	*no redundancies in this block*
	$b_4 = b_2 * c_3$	
	$m_7 = d_5 - e_6$	
	$f_8 = 0_8$	
B_1:	$a_3 = d_1 - e_2$	*no redundancies in this block*
	$b_5 = b_4 + a_3$	
B_2:	$b_3 = b_1 * c_1$	
	$f_4 = 1_4$	

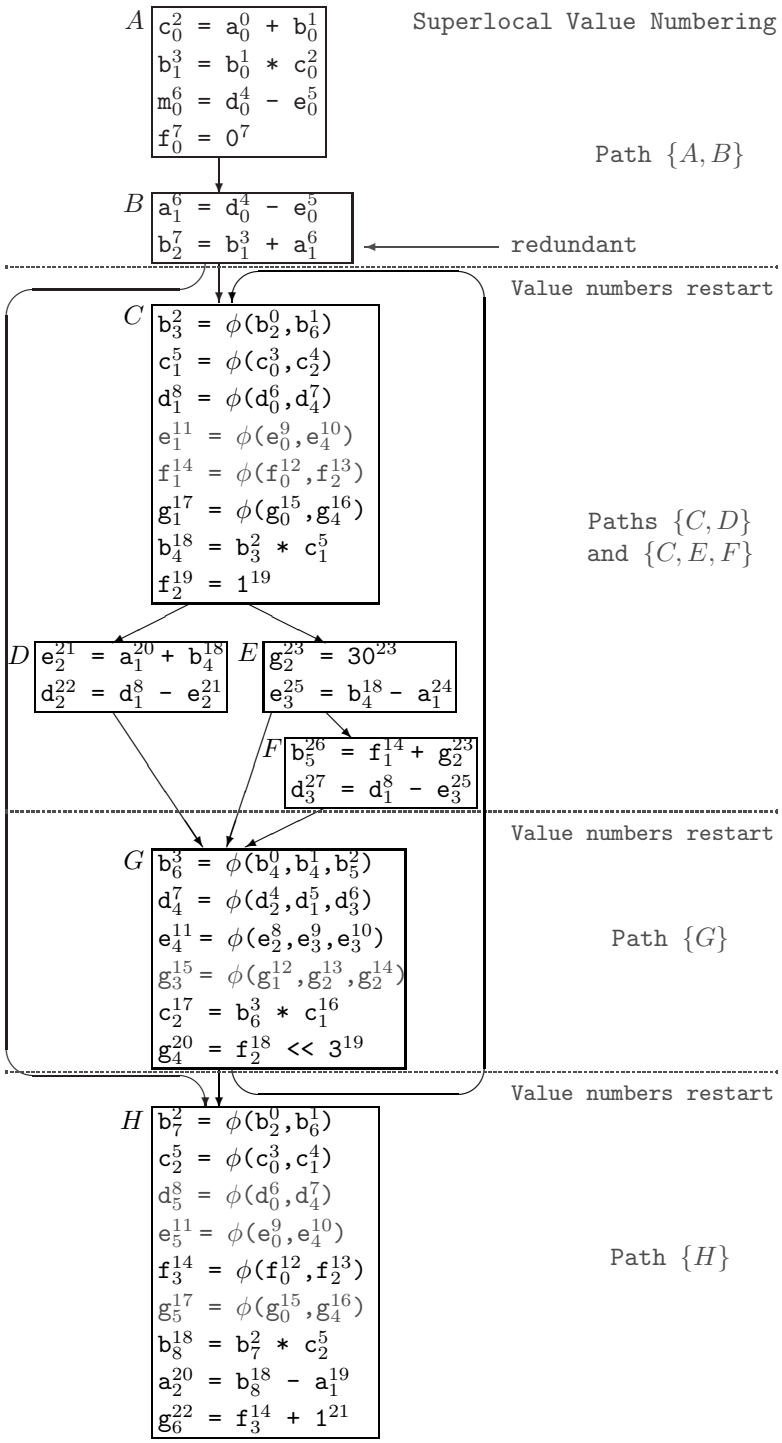
B_3:	$e_3 = a_1 + b_2$	*no redundancies in this block*
	$d_5 = d_4 - e_3$	
B_4:	$g_1 = 30_1$	*no redundancies in this block*
	$e_4 = b_2 - a_3$	
B_5:	$b_3 = f_1 + g_2$	*no redundancies in this block*
	$d_6 = d_4 - e_5$	
B_6:	$c_3 = b_1 * c_2$	*no redundancies in this block*
	$g_6 = f_4 << 3_5$	
B_7:	$b_3 = b_1 * c_2$	*no redundancies in this block*
	$a_5 = b_3 - a_4$	
	$g_8 = f_6 + 1_7$	

c. Apply superlocal value numbering to the EBBs and note any improvements that it finds beyond those found by local value numbering.

Answer: First, convert the code to SSA form. The answer (on the next page) uses minimal SSA form. Dead ϕ-functions are shown in gray. Some of the ϕ-functions that are live have their sole uses in other ϕ-functions, including dead ones.

Superlocal value numbering finds the extended basic blocks $\{A, B\}$, $\{C, D, E, F\}$, $\{G\}$, and $\{H\}$. All of these EBBs except $\{C, D, E, F\}$ have a single path, while $\{C, D, E, F\}$ has two paths ($\{C, D\}$ and $\{C, E, F\}$). Thus, the algorithm processed five paths: $\{A, B\}$, $\{C, D\}$, $\{C, E, F\}$, $\{G\}$, and $\{H\}$.

SSA names are indicated with subscripts. Value numbers appear as superscripts.


A
c$_0^2$ = a$_0^0$ + b$_0^1$
b$_1^3$ = b$_0^1$ * c$_0^2$
m$_0^6$ = d$_0^4$ - e$_0^5$
f$_0^7$ = 0^7
Superlocal Value Numbering
Path {A, B}
B
a$_1^6$ = d$_0^4$ - e$_0^5$
b$_2^7$ = b$_1^3$ + a$_1^6$
redundant
Value numbers restart
C
b$_3^2$ = φ(b$_2^0$,b$_6^1$)
c$_1^5$ = φ(c$_0^3$,c$_2^4$)
d$_1^8$ = φ(d$_0^6$,d$_4^7$)
e$_1^{11}$ = φ(e$_0^9$,e$_4^{10}$)
f$_1^{14}$ = φ(f$_0^{12}$,f$_2^{13}$)
g$_1^{17}$ = φ(g$_0^{15}$,g$_4^{16}$)
b$_4^{18}$ = b$_3^2$ * c$_1^5$
f$_2^{19}$ = 1^{19}
Paths {C, D}
and {C, E, F}
D
e$_2^{21}$ = a$_1^{20}$ + b$_4^{18}$
d$_2^{22}$ = d$_1^8$ - e$_2^{21}$
E
g$_2^{23}$ = 30^{23}
e$_3^{25}$ = b$_4^{18}$ - a$_1^{24}$
F
b$_5^{26}$ = f$_1^{14}$ + g$_2^{23}$
d$_3^{27}$ = d$_1^8$ - e$_3^{25}$
Value numbers restart
G
b$_6^3$ = φ(b$_4^0$,b$_4^1$,b$_5^2$)
d$_4^7$ = φ(d$_2^4$,d$_1^5$,d$_3^6$)
e$_4^{11}$ = φ(e$_2^8$,e$_3^9$,e$_3^{10}$)
g$_3^{15}$ = φ(g$_1^{12}$,g$_2^{13}$,g$_2^{14}$)
c$_2^{17}$ = b$_6^3$ * c$_1^{16}$
g$_4^{20}$ = f$_2^{18}$ << 3^{19}
Path {G}
Value numbers restart
H
b$_7^2$ = φ(b$_2^0$,b$_6^1$)
c$_2^5$ = φ(c$_0^3$,c$_1^4$)
d$_5^8$ = φ(d$_0^6$,d$_4^7$)
e$_5^{11}$ = φ(e$_0^9$,e$_4^{10}$)
f$_3^{14}$ = φ(f$_0^{12}$,f$_2^{13}$)
g$_5^{17}$ = φ(g$_0^{15}$,g$_4^{16}$)
b$_8^{18}$ = b$_7^2$ * c$_2^5$
a$_2^{20}$ = b$_8^{18}$ - a$_1^{19}$
g$_6^{22}$ = f$_3^{14}$ + 1^{21}
Path {H}

5. Consider the following simple five-point stencil computation:

```
      do 20 i = 2, n-1, 1
         t1 = A(i,j-1)
         t2 = A(i,j)
         do 10 j = 2, m-1, 1
            t3 = A(i,j+1)
            A(i,j) = 0.2 × (t1 + t2 + t3 + A(i-1,j) + A(i+1,j))
            t1 = t2
            t2 = t3
10       continue
20    continue
```

Each iteration of the loop executes two copy operations.

a. Loop unrolling can eliminate the copy operations. What unroll factor is needed to eliminate all copy operations in this loop?

Answer: The inner loop should be unrolled by three, as shown in the following code:

```
      do 20 i = 2, n-1, 1
         t1 = A(i,j-1)
         t2 = A(i,j)
         do 10 j = 2, m-1, 3
            t3     = A(i,j+1)
            t4     = A(i-1,j)
            t5     = A(i+1,j)
            A(i,j) = (t1+t2+t3+t4+t5)/5
            t1     = A(i,j+2)
            t4     = A(i-1,j+1)
            t5     = A(i+1,j+1)
            A(i,j) = (t1+t2+t3+t4+t5)/5
            t2     = A(i,j+3)
            t4     = A(i-1,j+2)
            t5     = A(i+1,j+2)
            A(i,j) = (t1+t2+t3+t4+t5)/5
10       continue
20    continue
```

b. In general, if a loop contains multiple cycles of copy operations, how can you compute the unroll factor needed to eliminate all of the copy operations?

Answer: Find the length of each cycle of copy operations. The desired unroll factor is the least common multiple of the complete set of unique cycle lengths.

Section 8.6

6. At some point p, LIVE(p) is the set of names that are *live* at p. LIVEOUT(b) is just the LIVE set at the end of block b.

a. Develop an algorithm that takes as input a block b and its LIVEOUT set and produces as output the LIVE set for each operation in the block.

Answer: Assume that block $b_c$onsists of a list of three-address operations, numbered from 0 to n-1. For the sake of illustration, we will compute a *LiveNow* set for each operation, plus one for a virtual n^{th} operation.

```
LiveNow(n) ← LIVEOUT(b)

for i = n-1 to 0 do
    LiveNow(i) ← LiveNow(i+1)
    remove Target(i) from LiveNow(i)
    add LeftOp(i) to LiveNow(i)
    add RightOp(i) to LiveNow(i)
```

In practice, most algorithms that use *LiveNow* compute it and use it incrementally, so that they do not need to store a set of each operation.

b. Apply your algorithm to blocks b_0 and b_1 in problem 1, using LIVEOUT(b_0) = {t_3, t_9} and LIVEOUT(b_1) = {t_7, t_8, t_9}.

Answer:

	Code	*LiveNow*
1	$t_1 \leftarrow a + b$	{ a, b, c, d, e, f }
2	$t_2 \leftarrow t_1 + c$	{ a, b, c, d, e, f }
3	$t_3 \leftarrow t_2 + d$	{ a, b, d, e, f }
4	$t_4 \leftarrow b + a$	{ t_3, a, b, e, f }
5	$t_5 \leftarrow t_3 + e$	{ t_3, t_4, a, b, e, f }
6	$t_6 \leftarrow t_4 + f$	{ t_3, t_4, a, b, f }
7	$t_7 \leftarrow a + b$	{ t_3, t_4, t_6, a, b }
8	$t_8 \leftarrow t_4 - t_7$	{ t_3, t_4, t_6, t_7 }
9	$t_9 \leftarrow t_8 * t_6$	{ t_3, t_6, t_8 }
10	LIVEOUT(b_0)	{ t_3, t_9 }

Results for Block b_0

	Code	*LiveNow*
1	$t_1 \leftarrow a \times b$	{ t_1, a, b, c, d, e, f}
2	$t_2 \leftarrow t_1 \times 2$	{ t_1, c, d, e, f }
3	$t_3 \leftarrow t_2 \times c$	{ t_1, t_2, c, d, e, f }
4	$t_4 \leftarrow 7 + t_3$	{ t_1, t_3, d, e, f }
5	$t_5 \leftarrow t_4 + d$	{ t_1, t_4, d, e, f }
6	$t_6 \leftarrow t_5 + 3$	{ t_1, t_4, t_5, e, f }
7	$t_7 \leftarrow t_4 + e$	{ t_1, t_4, t_6, e, f }
8	$t_8 \leftarrow t_6 + f$	{ t_1, t_6, t_7, f }
9	$t_9 \leftarrow t_1 + 6$	{ t_1, t_7, t_8 }
10	LIVEOUT(b_1)	{ t_7, t_8, t_9 }

Results for Block b_1

7. Figure 8.16 shows an algorithm for constructing hot paths in the CFG.

   a. Devise an alternate hot-path construction that pays attention to ties among equal-weight edges.

   Answer: Many solutions are possible. In our solution, we will assume a function, *MaxOut(⟨i,j⟩)* that, applied to an edge ⟨i,j⟩ *e*, returns the highest frequency count of an edge whose source is *j*. *MaxOut* is straightforward to write.

   One tie-breaking variant of the algorithm from Figure 8.16 is as follows:

```
E ← |edges|
for each block b
    make a degenerate chain, d, for b
    priority(d) ← E
// Create a priority queue, EdgeQ, of edges ordered by frequency
EdgeQ ← new priority queue
for each edge ⟨x,y⟩
    Enqueue(EdgeQ, ⟨x,y⟩, Frequency(⟨x,y⟩))
P ← 0
// Create a priority queue, Q, of edges with same frequency,
// ordered by MaxOut
⟨x,y⟩ ← Dequeue(EdgeQ)
while EdgeQ is not empty do
    Q ← new priority queue
    ⟨a,b⟩ ← Dequeue(EdgeQ)
    while Frequency(⟨a,b⟩) = Frequency(⟨x,y⟩) do
        Enqueue(Q, ⟨a,b⟩, MaxOut(⟨a,b⟩))
        if EdgeQ is empty
            then break
    x ← a    // set up next iteration of outer loop
    y ← b
    // process edges in Q, in order by MaxOut values
    while Q is not empty do
        ⟨m,n⟩ ← Dequeue(Q)
        if m is the tail of chain c_a and n is the head of chain c_b then
            t ← priority(c_a)
            append c_b onto c_a
            priority(c_a) ← min(t, priority(c_b), P++)
```

The version in Figure 8.16 takes equal-priority edges in the order that the queue presents them. This version constructs, for each distinct edge frequency count, a priority queue that contains all edges with that count. For each such edge, $\langle x,y \rangle$, it finds the largest frequency count of the edges that leave y and it uses that value to order the queue.

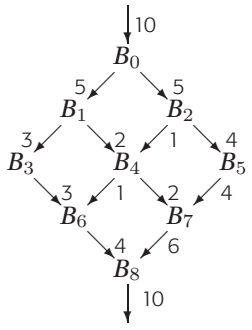
Thus, it will break ties between edges of equal frequency count by considering the frequencies of the edges successors in the CFG, and choosing the remaining edge with the "hottest" successor to process next.

b. Construct two examples where your algorithm leads to a code layout that improves on the layout produced by the book's algorithm. Use the code-layout algorithm from Figure 8.17 with the chains constructed by your algorithm and those built by the book's algorithm.

Answer:

Of course, myriad examples are possible. The CFG shown to the right is one example. The first edge chosen by the algorithm will be $\langle B_7,B_8 \rangle$.

At that point, it must choose one of the edges leaving B_0. Without tie breaking, the choice will be made on some arbitrary, but typically deterministic, criteria, such as the name of the destination node. If it chooses $\langle B_0,B_1 \rangle$ instead of $\langle B_0,B_2 \rangle$, then the hot path, (B_0,B_2,B_5,B_7,B_8) will contain one or more taken branches. With the other choice, all those jumps will be fall-through cases.



Example CFG

Section 8.7

8. Consider the following code fragment. It shows a procedure fee and two call sites that invoke fee.

```
static int A[1000,1000], B[1000];
   ...
x = A[i,j] + y;
call fee(i,j,1000);
   ...
call fee(1,1,0);

   ...
fee(int row; int col; int ub) {
   int i, sum;

   sum = A[row,col];

   for (i=0; i<ub; i++) {
      sum = sum + B[i];
   }
}
```

a. What optimization benefits would you expect from inlining fee at each of the call sites? Estimate the fraction of fee's code that would remain after inlining and subsequent optimization.

Answer: At the first call site, the compiler would realize some benefits from eliminating the call. For example, immediately before the call, the code references the value of A[i,j] while the first executable statement inside fee refers to the same location. After inlining, the compiler should recognize that it can reuse the value loaded for the first statement in the second. Of course, some of the overhead of the procedure call and return will vanish, as well. The larger part of the work, however, occurs in the loop. The only impact that inlining will have on the loop is to make the upper bound explicit.

At the second call site, the compiler would realize that the loop is dead code. Since the only use of sum is in the loop, it might eliminate that assignment as well and eliminate all of the code in fee.

Of course, fee is, in some sense, dead code. It has no externally visible effects. Thus, neither call can affect the answers produced by the overall program, except by making the application run longer. However, the compiler cannot tell, at the calls, that fee has no effect. Unless the compiler performs fairly sophisticated interprocedrual analysis, it cannot declare those calls dead.

Inlining, at either call site, would move all of the facts needed for dead code elimination into a single procedure. Thus, a compiler with a reasonable dead code eliminator, such as the *Dead* and *Clean* techniques described in Chapter 10, would eliminate all of the code that originated in `fee` after it inlined the calls.

b. Based on your experience in part a, sketch a high-level algorithm for estimating the benefits of inlining a specific call site. Your technique should consider both the call site and the callee.

Answer: See answer for question nine, below. We asked the same basic question twice.

9. In Problem 8, features of the call site and its context determined the extent to which the optimizer could improve the inlined code. Sketch, at a high level, a procedure for estimating the improvements that might accrue from inlining a specific call site. (With such an estimator, the compiler could inline the call sites with the highest estimated profit, stopping when it reached some threshold on procedure size or total program size.)

Answer: The "profit" from inline substitution arises from several distinct sources.

- First, inlining a procedure eliminates the overhead of the actual calling sequence—the pre-call sequence, the prolog, the epilog, and the post-return sequence. Most of the work entailed in those sequences is eliminated.
- Second, if any of the actual parameters are constants, they can be folded into the body of the callee after inlining. Two kinds of effects occur: direct simplification of computations and simplification of control flow. If all operands of a computation are (now) known constants, the computation can be folded away. If some operands are known constants and others are not, algebraic simplifications may still be possible. (See, for example, Figure 8.3 on page 424.)
  Control-flow constructs that depend on parameters with known constant values can sometimes be simplified. The loop in procedure `fee` of the example for Question 8.8 (above) does not execute when `ub` < 1.

We can calculate a rough estimate of the amount of computation that the compiler can eliminate by considering the savings in the procedure linkage, the savings in direct computations, and the simplifications of control flow. The estimate can be computed in an ad-hoc fashion. A slightly more formal methodology can be found in reference [31] in the EaC2e bibliography.

10. When the procedure placement algorithm, shown in Figure 8.21, considers an edge $\langle p,q \rangle$ it always places p before q.

    a. Formulate a modification of the algorithm that would consider placing the sink of an edge before its source.

    Answer: In principle, the solution is simple. In the graph, each node represents one or more procedures. Initially, each procedure holds one node. When the algorithm coalesces together nodes that are connected by edges in the graph, the resulting node represents all the procedures in the two nodes that it coalesced.

    We will call the first procedure in each node its head. We will keep a table of all procedures that contains, for each procedure, the head of its current node, the size of the procedure, and the procedure's distance from the head (from the first procedure in the node). With the entry for each procedure, we will keep a cumulative length for all the procedures in a node headed by that procedure.

    These data structures are easy to initialize. The algorithm must update them when it merges two nodes.

    Now, when the placement algorithm considers an edge $\langle x,y \rangle$, it should compare the distance between x and y with the node for x placed first against the distance with the node for x placed last. The calculation is easy, given that we know the distance from x to its head and the length of the code associated with that head, along with the same information for y. If

    $$dist(x) + length(head(y)) - dist(y) \leq dist(y) + length(head(x)) - dist(x)$$

    then x should follow y, otherwise y should follow x.

    All that remains is to discuss how to maintain the data structures. When the algorithm merges two nodes, say i and j, and places i before j, it must update the information for every procedure contained in j. For each p in j, it must set:

    $$head(p) \leftarrow head(i)$$
    $$dist(p) \leftarrow dist(p) + length(head(i))$$

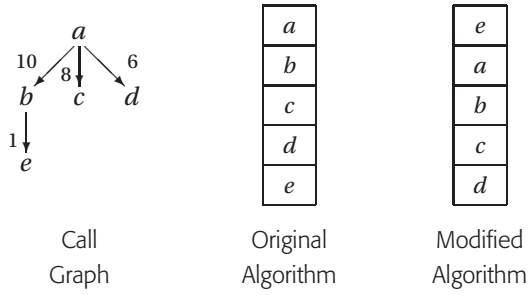
    It must also update the length of *head(i)*. If we update the code from Figure 8.21 with these changes, it might look as follows:

```
build the call multi-graph G                    // Initialization work
Q ← a new priority queue

for each node x ∈ G                              // Set up new structures
    list(x) ← { x }
    head(x) ← x
    dist(x) ← 0
    size(x) ← actual code length                 // size is for x
    length(x) ← size(x)                          // cumulative length of node x

for each edge (x,y) ∈ G                          // Add weights to the edges
    if (x = y)                                   // Self loop is irrelevant
        then delete (x,y) from G
        else weight( (x,y) ) ← estimated execution frequency for (x,y)

for each node x ∈ G
    list(x) ← { x }                              // Initialize placement lists
    if multiple edges exist from x to y
        then combine them and their weights
    for each edge (x,z) ∈ G                      // Put each edge into Q
        Enqueue(Q, (x,z), weight( (x,z) )

// Iterative reduction of the graph
while Q is not empty
    (x,y) ← Dequeue(Q)                           // Take highest priority edge
    if dist(x) + length(head(y)) - dist(y) ≤
        dist(y) + length(head(x)) - dist(x)
        then src ← x and dst ← y
        else src ← y and dst ← x

    for each edge (dst,z) ∈ G                    // Move source from dst to src
        ReSource( (dst,z), stc)
    for each edge (z,dst) ∈ G                    // Move target from dst to src
        ReTarget( (z,dst), src)

    for each procedure p in list(dst)
        head(p) ← head(src)
        dist(p) ← dist(p) + length(head(src))

    append list(dst) to list(src)                // Update the placement list
    delete dst and its edges from G              // Clean up G
```

More efficient schemes can be constructed, for example, using the disjoint-set union-find algorithm.

b. Construct an example where this approach places two procedures closer together than the original algorithm. Assume that all procedures are of uniform size.

Answer: The number of correct solutions is, essentially, unbounded. Here is one simple graph where considering placement before and placement afterwards results in a shorter distance between two procedures connected by a call.



The original algorithm produces the order shown in the center panel. The modified algorithm produces the order shown in the right panel. If we assume that all procedures have unit size, then the modified algorithm puts *b* closer to *e*.