Exercises for Section 1.1

1.1.1

What is the difference between a compiler and an interpreter?

Answer

A compiler is a program that can read a program in one language - the source language - and translate it into an equivalent program in another language – the target language and report any errors in the source program that it detects during the translation process.

Interpreter directly executes the operations specified in the source program on inputs supplied by the user.

1.1.2

What are the advantages of:
(a) a compiler over an interpreter
(b) an interpreter over a compiler?

Answer

a. The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs.

b. An interpreter can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

1.1.3

What advantages are there to a language-processing system in which the compiler
produces assembly language rather than machine language?

Answer

The compiler may produce an assembly-language program as its output, because
assembly language is easier to produce as output and is easier to debug.

1.1.4

A compiler that translates a high-level language into another high-level language is called a *source-to-source* translator. What advantages are there to using C as a target language for a compiler?

Answer

For the C language there are many compilers available that compile to almost every hardware.

1.1.5

Describe some of the tasks that an assembler needs to perform.

Answer

It translates from the assembly language to machine code. This machine code is
relocatable.

Exercises for Section 1.3

1.3.1

Indicate which of the following terms:

a. imperative
b. declarative
c. von Neumann
d. object-oriented
e. functional
f. third-generation
g. fourth-generation
h. scripting

apply to which of the following languages:

1. C
2. C++
3. Cobol
4. Fortran
5. Java
6. Lisp
7. ML
8. Perl
9. Python
10. VB.

Answer

imperative: C, C++

object-oriented: C++, Java

functional: ML

scripting: Perl, Python

Exercises for Section 1.6

1.6.1

For the block-structured C code below, indicate the values assigned to w, x, y, and z.

```c
int w, x, y, z;
int i = 4; int j = 5;
{
  int j = 7;
  i = 6;
  w = i + j;
}
x = i + j;
{
  int i = 8;
  y = i + j;
}
z = i + j;
```

Answer

w = 13, x = 11, y = 13, z = 11.

1.6.2

Repeat Exercise 1.6.1 for the code below.

```
int w, x, y, z;
int i = 3; int j = 4;
{
  int i = 5;
  w = i + j;
}
x = i + j;
{
  int j = 6;
  i = 7;
  y = i + j;
}
z = i + j;
```

Answer

w = 9, x = 7, y = 13, z = 11.

1.6.3

For the block-structured code of Fig. 1.14, assuming the usual static scoping of declarations, give the scope for each of the twelve declarations.

Answer

```
Block B1:
    declarations:  ->   scope
        w               B1-B3-B4
        x               B1-B2-B4
        y               B1-B5
        z               B1-B2-B5
Block B2:
    declarations:  ->   scope
        x               B2-B3
        z               B2
Block B3:
    declarations:  ->   scope
        w               B3
        x               B3
Block B4:
    declarations:  ->   scope
```

```
        w               B4
        x               B4
Block B5:
    declarations:  ->   scope
        y               B5
        z               B5
```

1.6.4

What is printed by the following C code?

```
#define a (x + 1)
int x = 2;
void b() { x = a; printf("%d\n", x); }
void c() { int x = 1; printf("%d\n", a); }
void main () { b(); c(); }
```

Answer

3

2

Exercises for Section 2.2

2.2.1

Consider the context-free grammar:

S -> S S + | S S * | a

1. Show how the string `aa+a*` can be generated by this grammar.
2. Construct a parse tree for this string.
3. What language does this grammar generate? Justify your answer.

Answer

1. `S` -> `S` S * -> `S` S + S * -> a `S` + S * -> a a + `S` * -> a a + a *
2.
3. L = {Postfix expression consisting of digits, plus and multiple signs}

2.2.2

What language is generated by the following grammars? In each case justify your answer.

1. S -> 0 S 1 | 0 1
2. S -> + S S | - S S | a
3. S -> S (S) S | ε
4. S -> a S b S | b S a S | ε
5. S -> a | S + S | S S | S * | (S)

Answer

1. L = {0^n1^n | n>=1}
2. L = {Prefix expression consisting of plus and minus signs}
3. L = {Matched brackets of arbitrary arrangement and nesting, includes ε}
4. L = {String has the same amount of a and b, includes ε}
5. L = {Regular expressions used to describe regular languages} refer to wiki

2.2.3

Which of the grammars in Exercise 2.2.2 are ambiguous?

Answer

1. No
2. No
3. Yes
4. Yes
5. Yes

2.2.4

Construct unambiguous context-free grammars for each of
the following languages. In each case show that your grammar is correct.

1. Arithmetic expressions in postfix notation.
2. Left-associative lists of identifiers separated by commas.
3. Right-associative lists of identifiers separated by commas.
4. Arithmetic expressions of integers and identifiers with the four binary operators +, -, *, /.
5. Add unary plus and minus to the arithmetic operators of 4.

Answer

```
1. E -> E E op | num

2. list -> list , id | id

3. list -> id , list | id

4. expr -> expr + term | expr - term | term
   term -> term * factor | term / factor | factor
   factor -> id | num | (expr)
```

```

5. expr -> expr + term | expr - term | term
   term -> term * unary | term / unary | unary
   unary -> + factor | - factor | factor
   factor - > id | num | (expr)
```

2.2.5

1. Show that all binary strings generated by the following grammar have values divisible by 3. Hint. Use induction on the number of nodes in a parse tree.

   num -> 11 | 1001 | num 0 | num num

2. Does the grammar generate all binary strings with values divisible by 3?

Answer

1. Proof

   Any string derived from the grammar can be considered to be a sequence consisting of 11 and 1001, where each sequence element is possibly suffixed with a 0.

   Let `n` be the set of positions where `11` is placed. `11` is said to be at position `i` if the first `1` in `11` is at position `i`, where `i` starts at 0 and grows from least significant to most significant bit.

   Let `m` be the equivalent set for `1001`.

   The sum of any string produced by the grammar is:

   sum

   $= \Sigma_n (2^1 + 2^0) * 2^n + \Sigma_m (2^3 + 2^0) * 2^m$

   $= \Sigma_n 3 * 2^n + \Sigma_m 9 * 2^m$

   This is clearly divisible by 3.

2. No. Consider the string "10101", which is divisible by 3, but cannot be derived from the grammar.

   Readers seeking a more formal proof can read about it below:

   Proof:

Every number divisible by 3 can be written in the form `3k`. We will consider `k > 0` (though it would be valid to consider `k` to be an arbitrary integer).

Note that every part of num(11, 1001 and 0) is divisible by 3, if the grammar could generate all the numbers divisible by 3, we can get a production for binary k from num's production:

```
3k = num   -> 11 | 1001 | num 0 | num num
 k = num/3 -> 01 | 0011 | k 0   | k k
 k         -> 01 | 0011 | k 0   | k k
```

It is obvious that any value of `k` that has more than 2 consecutive bits set to 1 can never be produced. This can be confirmed by the example given in the beginning:

10101 is 3*7, hence, k = 7 = 111 in binary. Because 111 has more than 2 consecutive 1's in binary, the grammar will never produce 21.

2.2.6

Construct a context-free grammar for roman numerals.

Note: we just consider a subset of roman numerals which is less than 4k.

Answer

wikipedia: Roman_numerals

- via wikipedia, we can categorize the single roman numerals into 4 groups:

```
I, II, III | I V | V, V I, V II, V III | I X
```

then get the production:

```
digit -> smallDigit | I V | V smallDigit | I X
smallDigit -> I | II | III | ε
```

- and we can find a simple way to map roman to arabic numerals. For example:

    - XII => X, II => 10 + 2 => 12
    - CXCIX => C, XC, IX => 100 + 90 + 9 => 199
    - MDCCCLXXX => M, DCCC, LXXX => 1000 + 800 + 80 => 1880
- via the upper two rules, we can derive the production:

  romanNum -> thousand hundred ten digit

  thousand -> M | MM | MMM | ε

  hundred -> smallHundred | C D | D smallHundred | C M

  smallHundred -> C | CC | CCC | ε

  ten -> smallTen | X L | L smallTen | X C

  smallTen -> X | XX | XXX | ε

  digit -> smallDigit | I V | V smallDigit | I X

  smallDigit -> I | II | III | ε

2.3 Exercises for Section 2.3

2.3.1

Construct a syntax-directed translation scheme that translates arithmetic expressions from infix notation into prefix notation in which an operator appears before its operands; e.g. , -xy is the prefix notation for x - y. Give annotated parse trees for the inputs 9-5+2 and 9-5*2.

Answer

productions:

```
expr -> expr1 + term
      | expr1 - term
      | term
term -> term1 * factor
      | term1 / factor
      | factor
factor -> digit | (expr)
```

translation schemes:

```
expr -> {print("+")} expr + term
      | {print("-")} expr - term
      | term
term -> {print("*")} term * factor
      | {print("/")} term / factor
      | factor
factor -> digit {print(digit)}
        | (expr)
```

2.3.2

Construct a syntax-directed translation scheme that translates arithmetic expressions from postfix notation into infix notation. Give annotated parse trees for the inputs 95-2* and 952*-.

Answer

productions:

```
expr -> expr expr +
      | expr expr -
      | expr expr *
      | expr expr /
      | digit
```

translation schemes:

```
expr -> expr {print("+")} expr +
      | expr {print("-")} expr -
      | {print("(")} expr {print(")*(")} expr {print(")")} *
      | {print("(")} expr {print(")/(")} expr {print(")")} /
      | digit {print(digit)}
```

Another reference answer

```
E -> {print("(")} E {print(op)} E {print(")"}} op | digit
{print(digit)}
```

2.3.3

Construct a syntax-directed translation scheme that translates integers into roman numerals.

Answer

assistant function:

```
repeat(sign, times) // repeat('a',2) = 'aa'
```

translation schemes:

```
num -> thousand hundred ten digit
       { num.roman = thousand.roman || hundred.roman ||
ten.roman || digit.roman;
         print(num.roman)}
thousand -> low {thousand.roman = repeat('M', low.v)}
```

```
hundred -> low {hundred.roman = repeat('C', low.v)}
         | 4 {hundred.roman = 'CD'}
         | high {hundred.roman = 'D' || repeat('X', high.v - 
5))}
         | 9 {hundred.roman = 'CM'}
ten -> low {ten.roman = repeat('X', low.v)}
     | 4 {ten.roman = 'XL'}
     | high {ten.roman = 'L' || repeat('X', high.v - 5))}
     | 9 {ten.roman = 'XC'}
digit -> low {digit.roman = repeat('I', low.v)}
       | 4 {digit.roman = 'IV'}
       | high {digit.roman = 'V' || repeat('I', high.v - 5))}
       | 9 {digit.roman = 'IX'}
low -> 0 {low.v = 0}
     | 1 {low.v = 1}
     | 2 {low.v = 2}
     | 3 {low.v = 3}
high -> 5 {high.v = 5}
      | 6 {high.v = 6}
      | 7 {high.v = 7}
      | 8 {high.v = 8}
```

2.3.4

Construct a syntax-directed translation scheme that trans lates roman numerals into integers.

Answer

productions:

```
romanNum -> thousand hundred ten digit
thousand -> M | MM | MMM | ε
hundred -> smallHundred | C D | D smallHundred | C M
smallHundred -> C | CC | CCC | ε
ten -> smallTen | X L | L smallTen | X C
smallTen -> X | XX | XXX  | ε
digit -> smallDigit | I V | V smallDigit | I X
smallDigit -> I | II | III | ε
```

translation schemes:

```
romanNum -> thousand hundred ten digit {romanNum.v = thousand.v || hundred.v || ten.v || digit.v; print(romanNun.v)}
thousand -> M {thousand.v = 1}
          | MM {thousand.v = 2}
          | MMM {thousand.v = 3}
          | ε {thousand.v = 0}
hundred -> smallHundred {hundred.v = smallHundred.v}
         | C D {hundred.v = smallHundred.v}
         | D smallHundred {hundred.v = 5 + smallHundred.v}
         | C M {hundred.v = 9}
smallHundred -> C {smallHundred.v = 1}
              | CC {smallHundred.v = 2}
              | CCC {smallHundred.v = 3}
              | ε {hundred.v = 0}
ten -> smallTen {ten.v = smallTen.v}
     | X L  {ten.v = 4}
     | L smallTen  {ten.v = 5 + smallTen.v}
     | X C  {ten.v = 9}
smallTen -> X {smallTen.v = 1}
          | XX {smallTen.v = 2}
          | XXX {smallTen.v = 3}
          | ε {smallTen.v = 0}
digit -> smallDigit {digit.v = smallDigit.v}
       | I V  {digit.v = 4}
       | V smallDigit  {digit.v = 5 + smallDigit.v}
       | I X  {digit.v = 9}
 smallDigit -> I {smallDigit.v = 1}
             | II {smallDigit.v = 2}
             | III {smallDigit.v = 3}
             | ε {smallDigit.v = 0}
```

2.3.5

Construct a syntax-directed translation scheme that translates postfix arithmetic expressions into equivalent prefix arithmetic expressions.

Answer

production:

```
expr -> expr expr op | digit
```

translation scheme:

```
expr -> {print(op)} expr expr op | digit {print(digit)}
```

Exercises for Section 2.4

2.4.1

Construct recursive-descent parsers, starting with the following grammars:

1. S -> + S S | - S S | a
2. S -> S (S) S | ε
3. S -> 0 S 1 | 0 1

Answer

See 2.4.1.1.c, 2.4.1.2.c, and 2.4.1.3.c for real implementations in C.

1) S -> + S S | - S S | a

```
void S(){
  switch(lookahead){
    case "+":
      match("+"); S(); S();
      break;
    case "-":
      match("-"); S(); S();
      break;
    case "a":
      match("a");
      break;
    default:
      throw new SyntaxException();
  }
}
void match(Terminal t){
  if(lookahead = t){
    lookahead = nextTerminal();
  }else{
    throw new SyntaxException()
  }
}
```

2) S -> S (S) S | ε

```
void S(){
  if(lookahead == "("){
    match("("); S(); match(")"); S();
  }
}
```

3) S -> 0 S 1 | 0 1

```
void S(){
  switch(lookahead){
    case "0":
      match("0"); S(); match("1");
      break;
    case "1":
      // match(epsilon);
      break;
    default:
      throw new SyntaxException();
  }
}
```

Exercises for Section 2.6

2.6.1

Extend the lexical analyzer in Section 2.6.5 to remove comments, defined as follows:

1. A comment begins with // and includes all characters until the end of that line.
2. A comment begins with /* and includes all characters through the next occurrence of the character sequence */.

2.6.2

Extend the lexical analyzer in Section 2.6.5 to recognize the relational operators <, <=, ==, ! =, >=, >.

2.6.3

Extend the lexical analyzer in Section 2.6.5 to recognize floating point numbers such as 2., 3.14, and . 5.

Answer

Source code: commit 8dd1a9a

Code snippet(src/lexer/Lexer.java)：

```
public Token scan() throws IOException, SyntaxException{
  for(;;peek = (char)stream.read()){
    if(peek == ' ' || peek == '\t'){
      continue;
    }else if(peek == '\n'){
      line = line + 1;
    }else{
      break;
    }
  }
```

```java

    // handle comment
    if(peek == '/'){
      peek = (char) stream.read();
      if(peek == '/'){
        // single line comment
        for(;;peek = (char)stream.read()){
          if(peek == '\n'){
            break;
          }
        }
      }else if(peek == '*'){
        // block comment
        char prevPeek = ' ';
        for(;;prevPeek = peek, peek = (char)stream.read()){
          if(prevPeek == '*' && peek == '/'){
            break;
          }
        }
      }else{
        throw new SyntaxException();
      }
    }

    // handle relation sign
    if("<=!>".indexOf(peek) > -1){
      StringBuffer b = new StringBuffer();
      b.append(peek);
      peek = (char)stream.read();
      if(peek == '='){
        b.append(peek);
      }
      return new Rel(b.toString());
    }

    // handle number, no type sensitive
    if(Character.isDigit(peek) || peek == '.'){
      Boolean isDotExist = false;
      StringBuffer b = new StringBuffer();
      do{
        if(peek == '.'){
          isDotExist = true;
        }
        b.append(peek);
        peek = (char)stream.read();
      }while(isDotExist == true ? Character.isDigit(peek) : Character.isDigit(peek) || peek == '.');
      return new Num(new Float(b.toString()));
    }

    // handle word
    if(Character.isLetter(peek)){
      StringBuffer b = new StringBuffer();
      do{
        b.append(peek);
        peek = (char)stream.read();
      }while(Character.isLetterOrDigit(peek));
      String s = b.toString();
      Word w = words.get(s);
      if(w == null){
        w = new Word(Tag.ID, s);
        words.put(s, w);
      }
      return w;
    }

    Token t = new Token(peek);
    peek = ' ';
    return t;
  }
```

Exercises for Section 2.8

2.8.1

For-statements in C and Java have the form:

for (exprl ; expr2 ; expr3) stmt

The first expression is executed before the loop; it is typically used for initializing the loop index. The second expression is a test made before each iteration of the loop; the loop is exited if the expression becomes 0. The loop itself can be thought of as the statement {stmt expr3 ; }. The third expression is executed at the end of each iteration; it is typically used to increment the loop index. The meaning of the for-statement is similar to

expr1 ; while (expr2) {stmt expr3 ; }

Define a class For for for-statements, similar to class If in Fig. 2.43.

Answer

```
class For extends Stmt {
  Expr E1;
  Expr E2;
  Expr E3;
  Stmt S;
  public For(Expr expr1, Expr expr2, Expr expr3, Stmt stmt){
    E1 = expr1;
    E2 = expr2;
    E3 = expr3;
    S = stmt;
  }
  public void gen(){
    E1.gen();
    Label start = new Label();
    Label end = new Label();
    emit("ifFalse " + E2.rvalue().toString() + " goto " + end);
    S.gen();
    E3.gen();
    emit("goto " + start);
    emit(end + ":")
  }
}
```

2.8.2

The programming language C does not have a boolean type. Show how a C compiler might translate an if-statement into three-address code.

Answer

Replace

```
emit("ifFalse " + E.rvalue().toString() + " goto " + after);
```

with

```
emit("ifEqual " + E.rvalue().toString() + " 0 goto " + after);
```

or

```
emit("ifEqualZero " + E.rvalue().toString() + " goto " + after);
```

第2章要点

1. 文法、语法制导翻译方案、语法制导的翻译器

以一个仅支持个位数加减法的表达式为例

1. 文法

   list -> list + digit | list - digit | digit

   digit -> 0 | 1 | ... | 9

2. （消除了左递归的）语法制导翻译方案

   expr -> term rest

   rest -> + term { print('+') } rest | - term { print('+') } rest | ε

   term -> 0 { print('0') } | 1 { print('1') } | ... | 9 { print('9') }

3. 语法制导的翻译器

   java代码见 p46

2. 语法树、语法分析树

以 2 + 5 - 9 为例

3. 正则文法、上下文无关文法、上下文相关文法?

文法缩写：

- RG：正则文法
- CFG：上下文无关文法
- CSG：上下文相关文法

正则文法

wiki

正则文法在标准之后所有产生式都应该满足下面三种情形中的一种：

```
B -> a
B -> a C
B -> epsilon
```

关键点在于：

1. 产生式的左手边必须是一个非终结符。
2. 产生式的右手边可以什么都没有，可以有一个终结符，也可以有一个终结符加一个非终结符。

从产生式的角度看，这样的规定使得每应用一条产生规则，就可以产生出零或一个终结符，直到最后产生出我们要的那个字符串。

从匹配的角度看，这样的规定使得每应用一条规则，就可以消耗掉一个非终结符，直到整个字符串被匹配掉。

这样定义的语言所对应的自动机有一种性质：有限状态自动机。

简单来说就是只需要记录当前的一个状态，和得到下一个输入符号，就可以决定接下来的状态迁移。

正则文法和上下文无关文法

CFG 跟 RG 最大的区别就是，产生式的右手边可以有零或多个终结符或非终结符，顺序和个数都没限制。

想像一个经典例子，括号的配对匹配：

expr -> '(' expr ')' | epsilon

这个产生式里（先只看第一个子产生式），右手边有一个非终结符 expr，但它的左右两侧都有终结符，这种产生式无法被标准化为严格的 RG 。这就是CFG的一个例子。

它对应的自动机就不只要记录当前的一个状态，还得外加记录到达当前位置的历史，才可以根据下一个输入符号决定状态迁移。所谓的“历史”在这里就是存着已匹配规则的栈。

CFG 对应的自动机为 PDA(下推自动机)。

RG 的规定严格，对应的好处是它对应的自动机非常简单，所以可以用非常高效且简单的方式来实现。

上下文相关文法

CSG 在 CFG的基础上进一步放宽限制。

产生式的左手边也可以有终结符和非终结符。左手边的终结符就是“上下文”的来源。也就是说匹配的时候不能光看当前匹配到哪里了，还得看当前位置的左右到底有啥（也就是上下文是啥），上下文在这条规则应用的时候并不会被消耗掉，只是“看看”。

CSG 的再上一层是 PSG，phrase structure grammar。

基本上就是CSG的限制全部取消掉。

左右两边都可以有任意多个、任意顺序的终结符和非终结符。

反正不做自然语言处理的话也不会遇到这种文法，所以具体就不说了。

4. 为什么有 n 个运算符的优先级，就对应 n+1 个产生式?

优先级的处理可以在纯文法层面解决，也可以在parser实现中用别的办法处理掉。

纯文法层面书上介绍的，有多少个优先级就有那么多加1个产生式。

书上介绍的四则运算的文法，会使得加减法离根比较近，乘除法离根比较远。

语法树的形状决定了节点的计算顺序，离根远的节点就会先处理，这样看起来就是乘除法先计算，也就是乘除法的优先级更高。

参考：http://rednaxelafx.iteye.com/blog/492667

5. 避免二义性文法的有效原则?

二义性问题主要是跟 CFG 的特性有关系的。

CFG 的选择结构（"|"）是没有规定顺序或者说优先级的，
同时，多个规则可能会有共同前缀，
这样才会有二义性问题。

PEG 是跟CFG类似的一种东西，语言的表达力上跟CFG相似。
但文法层面没有二义性，因为它的选择结构（"|"）是有顺序或者说有优先级的。

6. 避免预测分析器因左递归文法造成的无限循环

产生式：

A -> A x | y

语法制导翻译伪代码片段：

```
void A(){
    switch(lookahead){
        case x:
            A();match(x);break;
        case y:
            match(y):break;
        default:
            report("syntax error")
    }
}
```

当语句符合 A x 形式时， A() 运算会陷入死循环，可以通过将产生式改为等价的非左递归形式来避免：

B -> y C

C -> x C | ε

7. 为什么在右递归的文法中，包含了左结合运算符的表达式翻译会比较困难？

8. 中间代码生成时的左值和右值问题。

看了书上 lvalue() 和 rvalue() 的伪代码，感觉可以做左值也可以做右值的都由 lvalue() 处理，而对于右值的处理，要么自己处理掉了，对于可以作为左值的右值则调用 lvalue()。

为什么不直接弄个 value() 就结了？

Exercises for Section 3.1

3.1.1

Divide the following C++ program:

```
float limitedSquare(x){float x;
  /* returns x-squared, nut never more than 100 */
  return (x <= -10.0 || x >= 10.0) ? 100 : x*x;
}
```

into appropriate lexemes, using the discussion of Section 3.1.2 as a guide. Which lexemes should get associated lexical values? What should those values be?

Answer

```
<float> <id, limitedSquaare> <(> <id, x> <)> <{>
  <float> <id, x>
  <return> <(> <id, x> <op,"<="> <num, -10.0> <op, "||"> <id, x> <op, ">="> <num, 10.0> <)> <op, "?"> <num, 100> <op, ":"> <id, x> <op, "*"> <id, x>
<}>
```

3.1.2

Tagged languages like HTML or XML are different from conventional programming
languages in that the punctuation (tags) are either very numerous (as in HTML) or a user-definable set (as in XML). Further, tags can often have parameters. Suggest how to divide the following HTML document:

```
Here is a photo of <b>my house</b>;
<p><img src="house.gif"/><br/>
see <a href="morePix.html">More Picture</a> if you
liked that one.</p>
```

into appropriate lexemes. Which lexemes should get associated lexical values, and what should those values be?

Answer

```
<text, "Here is a photo of"> <nodestart, b> <text, "my house"> <nodeend, b>
<nodestart, p> <selfendnode, img> <selfendnode, br>
<text, "see"> <nodestart, a> <text, "More Picture"> <nodeend, a>
<text, "if you liked that one."> <nodeend, p>
```

Consult the language reference manuals to determine

1. the sets of characters that form the input alphabet (excluding those that may only appear in character strings or comments)
2. the lexical form of numerical constants, and
3. the lexical form of identifiers,

for each of the following languages:

1. C
2. C++
3. C#
4. Fortran
5. Java
6. Lisp
7. SQL

Describe the languages denoted by the following regular expressions:

1. a(a|b)*a
2. ((ε|a)b*)*
3. (a|b)*a(a|b)(a|b)
4. a*ba*ba*ba*
5. !! (aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*

1. String of a's and b's that start and end with a.
2. String of a's and b's.
3. String of a's and b's that the character third from the last is a.
4. String of a's and b's that only contains three b.

5. String of a's and b's that has a even number of a and b.

In a string of length n, how many of the following are there?

1. Prefixes.
2. Suffixes.
3. Proper prefixes.
4. ! Substrings.
5. ! Subsequences.

1. n + 1
2. n + 1
3. n - 1
4. C(n+1,2) + 1 (need to count epsilon in)
5. Σ(i=0,n) C(n, i)

Most languages are case sensitive, so keywords can be written only one way, and the regular expressions describing their lexeme is very simple. However, some languages, like SQL, are case insensitive, so a keyword can be written either in lowercase or in uppercase, or in any mixture of cases. Thus, the SQL keyword SELECT can also be written select, Select, or sElEcT, for instance. Show how to write a regular expression for a keyword in a case insensitive language. Illustrate the idea by writing the expression for "select" in SQL.

```
select -> [Ss][Ee][Ll][Ee][Cc][Tt]
```

Write regular definitions for the following languages:

1. All strings of lowercase letters that contain the five vowels in order.
2. All strings of lowercase letters in which the letters are in ascending

lexicographic order.
3. Comments, consisting of a string surrounded by /* and */, without an intervening */, unless it is inside double-quotes (")
4. !! All strings of digits with no repeated digits. Hint: Try this problem first with a few digits, such as {O, 1, 2}.
5. !! All strings of digits with at most one repeated digit.
6. !! All strings of a's and b's with an even number of a's and an odd number of b's.
7. The set of Chess moves,in the informal notation,such as p-k4 or kbp*qn.
8. !! All strings of a's and b's that do not contain the substring abb.
9. All strings of a's and b's that do not contain the subsequence abb.

1

```
want -> other* a (other|a)* e (other|e)* i (other|i)* o (other|o)* u (other|u)*
other -> [bcdfghjklmnpqrstvwxyz]
```

2

```
a* b* ... z*
```

3

```
\/\*([^*"]*|".*"|\*+[^/])*\*\/
```

4

```
want -> 0|A?0?1(A0?1|01)*A?0?|A0?
A -> 0?2(02)*
```

Steps:

step1. Transition diagram

step2. GNFA

step3. Remove node 0 and simplify

step4. Remove node 2 and simplify

step5. Remove node 1 and simplify

5

```
want -> (FE*G|(aa)*b)(E|FE*G)
E -> b(aa)*b
F -> a(aa)*b
G -> b(aa)*ab|a
F -> ba(aa)*b
```

Steps:

step1. Transition diagram

step2. GNFA

step3. Remove node A and simplify

step4. Remove node D and simplify

step5. Remove node C and simplify

8

```
b*(a+b?)*
```

9

```
b* | b*a+ | b*a+ba*
```

Write character classes for the following sets of characters:

1. The first ten letters (up to "j") in either upper or lower case.
2. The lowercase consonants.
3. The "digits" in a hexadecimal number (choose either upper or lower case

for the "digits" above 9).
4. The characters that can appear at the end of alegitimate English sentence (e.g. , exclamation point) .

1. [A-Ja-j]
2. [bcdfghjklmnpqrstvwxzy]
3. [0-9a-f]
4. [.?!]

Note that these regular expressions give all of the following symbols (operator characters) a special meaning:

```
\ " . ^ $ [ ] * + ? { } | /
```

Their special meaning must be turned off if they are needed to represent themselves in a character string. We can do so by quoting the character within a
string of length one or more; e.g., the regular expression "**" matches the string ** . We can also get the literal meaning of an operator character by preceding it by a backslash. Thus, the regular expression ** also matches the string **. Write a regular expression that matches the string "\.

```
\"\\
```

The regular expression r{m, n} matches from m to n occurrences of the pattern r.
For example, a [1 , 5] matches a string of one to five a's. Show that for every regular expression containing repetition operators of this form, there is an equivalent regular expression without repetition operators.

r{m,n} is equals to r.(m).r | r.(m + 1).r | ... | r.(n).r

The operator ^ matches the left end of a line, and

Double exponent: use braces to clarify matches any complete line that does not contain a lowercase vowel.

1. How do you tell which meaning of ^ is intended?
2. Can you always replace a regular expression using the ^ and $ operators by an equivalent expression that does not use either of these operators?

1. if ^ is in a pair of brakets, and it is the first letter, it means complemented classes, or it means the left end of a line.

3.4 节的练习

3.4.1

给出识别练习 3.3.2 中各个正则表达式所描述的语言状态转换图。

解答

解答步骤：NFA -> DFA -> 最少状态的 DFA（状态转换图）

1. a(a|b)*a

   NFA:

   DFA:

NFA	DFA	a	b
{0}	A	B	
{1,2,3,5,8}	B	C	D
{2,3,4,5,7,8,**9**}	**C**	C	D
{2,3,5,6,7,8}	D	C	D

```
最少状态的 DFA(状态转换图):

合并不可区分的状态 B 和 D

![3 4 1-1](https://f.cloud.github.com/assets/340282/155878/fd81a78c-7674-11e2-9cdc-8097e665161f.gif)
```

2. ((ε|a)b*)*

3. (a|b)*a(a|b)(a|b)

   NFA:

```
DFA:

<table>
    <thead>
        <tr>
            <th>NFA</th>
            <th>DFA</th>
            <th>a</th>
            <th>b</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>{0,1,2,4,7}</td>
            <td>A</td>
            <td>B</td>
            <td>C</td>
        </tr>
        <tr>
            <td>{1,2,3,4,6,7,8,9,11}</td>
            <td>B</td>
            <td>D</td>
            <td>E</td>
        </tr>
        <tr>
            <td>{1,2,4,5,6,7}</td>
            <td>C</td>
            <td>B</td>
            <td>C</td>
        </tr>
        <tr>
            <td>{1,2,3,4,6,7,8,9,10,11,13,14,16}</td>
            <td>D</td>
            <td><b>F</b></td>
            <td><b>G</b></td>
        </tr>
        <tr>
            <td>{1,2,4,5,6,7,12,13,14,16}</td>
            <td>E</td>
            <td><b>H</b></td>
            <td><b>I</b></td>
        </tr>
        <tr>
            <td>{1,2,3,4,6,7,8,9,10,11,13,14,15,16,<b>18</b>}
</td>
            <td><b>F</b></td>
            <td><b>F</b></td>
            <td><b>G</b></td>
        </tr>
        <tr>
            <td>{1,2,4,5,6,7,12,13,14,16,17,<b>18</b>}</td>
            <td><b>G</b></td>
            <td><b>H</b></td>
            <td><b>I</b></td>
        </tr>
        <tr>
            <td>{1,2,3,4,6,7,8,9,11,15,<b>18</b>}</td>
            <td><b>H</b></td>
            <td>D</td>
            <td>E</td>
        </tr>
        <tr>
            <td>{1,2,4,5,6,7,17,<b>18</b>}</td>
            <td><b>I</b></td>
            <td>B</td>
            <td>C</td>
        </tr>
    </tbody>
</table>

最少状态的 DFA(状态转换图):

合并不可区分的状态 A 和 C

![3 4 1-3](https://f.cloud.github.com/assets/340282/412536/700de2e0-abbb-11e2-9f34-1a2605c8eff4.gif)
```

4. a*ba*ba*ba*

3.4.2

给出识别练习 3.3.5 中各个正则表达式所描述语言的状态转换图。

3.4.3

构造下列串的失效函数。

1. ababababaab
2. aaaaaa
3. abbaabb

解答

代码详见：src/failure-function.js

1. [0, 0, 1, 2, 3, 4, 5, 1, 2]
2. [0, 1, 2, 3, 4, 5]
3. [0, 0, 0, 1, 1, 2, 3]

3.4.4 !

对 s 进行归纳，证明图 3-19 的算法正确地计算出了失效函数。

图 3-19：计算关键字 b_1b_2...b_n 的失效函数

```
01  t = 0;
02  f(1) = 0;
03  for (s = 1; s < n; s ++){
04      while( t > 0 && b_s+1 != b_t+1) t = f(t);
05      if(b_s+1 == b_t+1){
06          t = t + 1;
07          f(s + 1) = t;
08      }else{
09          f(s + 1) = 0;
10      }
11  }
```

证明

1. 已知 f(1) = 0

2. 在第 1 次 for 循环时，计算 f(2) 的值，当第5行代码 b_2 == b_1 成立时，代码进入到第7行得出 f(2) = 1，不成立时，则代码进入第9行得出 f(2) = 0。显然，这次循环正确的计算出了 f(2) 。
3. 假设在第 i-1 次进入循环时，也正确的计算出了 f(i)，也有 f(i) = t (无论 t 是大于 0 还是等于 0)
4. 那么在第 1 次进入循环时，分两种情况进行考虑：
    1. t == 0

       这种情况比较简单，直接从第 5 行开始，当 b_i+1 == b_1 时，f(i+1) = 1，否则 f(i+1) = 0
    2. t > 0
       while 循环会不断缩小 t 值，试图找出最大可能的使得 b_i+1 == b_t+1 成立的 t 值，如果找到了，则进入第 5 行执行，得到 f(i+1) = t+1；或者直到 t == 0 时也没有找到，则跳出循环，这时进入第 5 行执行，过程类似于前一种情况。

3.4.5！！

说明图 3-19 中的第 4 行的复制语句 t = f(t) 最多被执行 n 次。进而说明整个算法的时间复杂度是 O(n)，其中 n 是关键字长度。

解答

详见 matrix 的博文 KMP算法详解。

3.4.6

应用 KMP 算法判断关键字 ababaa 是否为下面字符串的子串：

1. abababaab
2. abababbaa

解答

代码详见：src/failure-function.js

1. true
2. false

3.4.7！！

说明图 3-20 中的算法可以正确的表示输入关键字是否为一个给定字符串的子串。

图 3-20：KMP 算法在 O(m + n) 的时间内检测字符串a_1a_3...a_n 中是否包含单个关键字 b1b2...bn

```
s = 0;
for(i = 1; i <= m; i ++){
    while(s > 0 && a_i != b_s+1) s = f(s);
    if(a_i == b_s+1) s = s + 1;
    if(s == n) return "yes";
}
return "no";
```

3.4.8

假设已经计算得到函数 f 且他的值存储在一个以 s 为下标的数字中，说明图 3-20 中算法的时间复杂度为 O(m + n)。

解答

详见 matrix 的博文 KMP算法详解。

3.4.9

Fibonacci 字符串的定义如下：

1. s1 = b
2. s2 = a
3. 当 k > 2 时， $s_k = s_{k-1}\ s_{k-2}$

例如：s_3 = ab, s_4 = aba, s_5 = abaab

1. s_n 的长度是多少?
2. 构造 s_6 的失效函数。
3. 构造 s_7 的失效函数。
4. ！！ 说明任何 s_n 的失效函数都可以被表示为：f(1) = f(2) = 0，且对于 2 < j <= $|s_n|$, f(j) = j - $|s_{k-1}|$，其中 k 是使得 $|s_k|$ <= j + 1 的最大整数。
5. ！！ 在 KMP 算法中，当我们试图确定关键字 s_k 是否出现在字符串 s_{k+1} 中，最多会连续多少次调用失效函数?

解答

1. 见 维基百科
2. s_6 = abaababa

   failure = [0, 0, 1, 1, 2, 3, 2, 3]
3. s_7 = abaababaabaab

   failure = [0, 0, 1, 1, 2, 3, 2, 3, 4, 5, 6, 4, 5]

Exercises for Section 3.5

3.5.1

Describe how to make the following modifications to the Lex
program of Fig. 3.23:

1. Add the keyword while.
2. Change the comparison operators to be the C operators of that kind.
3. Allow the underscore (_) as an additional letter.
4. ! Add a new pattern with token STRING. The pattern consists of a double quote (") , any string of characters and a final double-quote. However, if a double-quote appears in the string, it must be escaped by preceding it with a backslash () , and therefore a backslash in the string must be represented by two backslashes. The lexical value, which is the string without the surrounding double-quotes, and with backslashes used to es.,. cape a character removed. Strings are to be installed in a table of strings.

source

3.5.2

Write a Lex program that copies a file, replacing each non
empty sequence of white space by a single blank

3.5.3

Write a Lex program that copies a C program, replacing each
instance of the keyword f loat by double.。

3.5.4 !

Write a Lex program that converts a file to "Pig latin."
Specifically, assume the file is a sequence of words (groups . of letters)
separated
by whitespace. Every time you encounter a word:

1. If the first letter is a consonant, move it to the end of the word and then add ay!
2. If the first letter is a vowel, just add ay to the end of the word.

All nonletters are copied int act to the output.

source

3.5.5 !

In SQL, keywords and identifiers are case-insensitive. Write
a Lex program that recognizes the keywords SELECT, FROM, and WHERE (in any combination of capital and lower-case letters) , and token ID, which for the purposes of this exercise you may take to be any sequence of letters and digits, beginning with a letter. You need not install identifiers in a symbol table, but tell how the "install" function would differ from that described for case-sensitive
identifiers as in Fig. 3.23.

source

3.6 Exercises for Section 3.6

3.6.1 !

Figure 3.19 in the exercises of Section 3.4 computes the failure function for the KMP algorithm. Show how, given that failure function, we can construct, from a keyword b1b2...bn an n + 1-state DFA that recognizes .*b1b2...bn, where the dot stands for "any character." Moreover, this DFA can be constructed in O(n) time.

Answer

Take the string "abbaabb" in exercise 3.4.3-3 as example, the failure function is:

- n : 1, 2, 3, 4, 5, 6, 7
- f(n): 0, 0, 0, 1, 1, 2, 3

The DFA is：

Pseudocode of building the DFA：

```
for (i = 0; i< n; i ++) {
  move[s[i], c] = {
    if ( c == b1b2…bn[i] ) {
      goto s[i+1]
    } else {
      goto s[f(i)]
    }
  }
}
```

It is obviously that with the known f(n), this DFA can be constructed in O(n) time.

3.6.2

Design finite automata (deterministic or nondeterministic) for each of the languages of Exercise 3.3.5.

3.6.3

For the NFA of Fig. 3.29, indicate all the paths labeled aabb.
Does the NFA accept aabb?

Answer

- (0) -a-> (1) -a-> (2) -b-> (2) -b-> ((3))
- (0) -a-> (0) -a-> (0) -b-> (0) -b-> (0)
- (0) -a-> (0) -a-> (1) -b-> (1) -b-> (1)
- (0) -a-> (1) -a-> (1) -b-> (1) -b-> (1)
- (0) -a-> (1) -a-> (2) -b-> (2) -b-> (2)
- (0) -a-> (1) -a-> (2) -b-> (2) -ε-> (0) -b-> (0)
- (0) -a-> (1) -a-> (2) -ε-> (0) -b-> (0) -b-> (0)

This NFA accepts "aabb"

3.6.4

Repeat Exercise 3.6.3 for the NFA of Fig. 3.30.

3.6.5

Give the transition tables for the NFA of:

1. Exercise 3.6.3.
2. Exercise 3.6.4.
3. Figure 3.26.

Answer

Table 1

state	a	b	ε
0	{0,1}	{0}	∅
1	{1,2}	{1}	∅
2	{2}	{2,3}	{0}
3	∅	∅	∅

Table 2

state	a	b	ε
0	{1}	∅	{3}
1	∅	{2}	{0}
2	∅	{3}	{1}
3	{0}	∅	{2}

Table 3

state	a	b	ε
0	∅	∅	{1,2}
1	{2}	∅	∅
2	{2}	∅	∅
3	∅	{4}	∅
4	∅	{4}	∅

Exercises for Section 3.7

3.7.1

Convert to DFA's the NFA's of:

1. Fig. 3.26.
2. Fig. 3.29.
3. Fig. 3.30.

Answer

1、

Transition table

NFA State	DFA State	a	b
{0,1,3}	A	B	C
{2}	B	B	∅
{4}	C	∅	C

DFA

2、

Transition table

NFA State	DFA State	a	b
{0}	A	B	A
{0,1}	B	C	B
{0,1,2}	C	C	D
{0,2,3}	D	C	D

DFA

3、

Transition table

NFA State	DFA State	a	b
{0,1,2,3}	A	A	A

DFA

3.7.2

use Algorithm 3.22 to simulate the NFA's:

1. Fig. 3.29.
2. Fig. 3.30.

on input aabb.

Answer

1. -start->{0}-a->{0,1}-a->{0,1,2}-b->{0,2,3}-b->{0,2,3}
2. -start->{0,1,2,3}-a->{0,1,2,3}-a->{0,1,2,3}-b->{0,1,2,3}-b->{0,1,2,3}

3.7.3

Convert the following regular expressions to deterministic finite automata, using algorithms 3.23 and 3.20:

1. (a|b)*
2. (a*|b*)*
3. ((ε|a)|b*)*
4. (a|b)*abb(a|b)*

Answer

1、

NFA

Transition table

NFA State	DFA State	a	b
{0,1,2,3,7}	A	B	C
{1,2,3,4,6,7}	B	B	C
{1,2,3,5,6,7}	C	B	C

DFA

2、

NFA

Transition table

NFA State	DFA State	a	b
{0,1,2,3,4,5,8,9,10,11}	A	B	C
{1,2,3,4,5,6,8,9,10,11}	B	B	C
{1,2,3,4,5,7,8,9,10,11}	C	B	C

DFA

3、

NFA

Transition table

NFA State	DFA State	a	b
{0,1,2,3,4,6,7,9,10}	A	B	C
{1,2,3,4,5,6,7,9,10}	B	B	C
{1,2,3,4,6,7,8,9,10}	C	B	C

DFA

4、

NFA

Transition table

NFA State	DFA State	a	b
{0,1,2,4,7}	A	B	C
{1,2,3,4,6,7,8}	B	B	D
{1,2,4,5,6,7}	C	B	C
{1,2,4,5,6,7,9}	D	B	E
{1,2,4,5,6,7,10,11,12,14,17}	E	F	G
{1,2,3,4,6,7,8,11,12,13,14,16,17}	F	F	H
{1,2,4,5,6,7,11,12,13,15,16,17}	G	F	G
{1,2,4,5,6,7,9,11,12,14,15,16,17}	H	F	I
{1,2,4,5,6,7,10,11,12,14,15,16,17}	I	F	G

DFA

Exercises for Section 3.8

3.8.1

Suppose we have two tokens: (1) the keyword if, and (2)
identifiers, which are strings of letters other than if. Show:

1. The NFA for these tokens, and
2. The DFA for these tokens.

Answer

1. NFA

   NOTE: this NFA has potential conflict, we can decide the matched lexeme by 1. take the longest 2. take the first listed.

2. DFA

3.8.2

Repeat Exercise 3.8.1 for tokens consisting of (1) the keyword
while, (2) the keyword when, and (3) identifiers consisting of strings of letters and digits, beginning with a letter.

Answer

1. NFA

2. DFA

   bother to paint

3.8.3 !

Suppose we were to revise the definition of a DFA to allow
zero or one transition out of each state on each input symbol (rather than exactly one such transition, as in the standard DFA definition). Some regular expressions would then have smaller "DFA's" than they do under the standard

definition of a DFA. Give an example of one such regular expression.

Answer

Take the language defined by regular expression "ab" as the example, assume that the set of input symbols is {a, b}

Standard DFA

Revised DFA

Obviously, the revised DFA is smaller than the standard DFA.

3.8.4 !!

Design an algorithm to recognize Lex-lookahead patterns of
the form rl/r2, where rl and r2 are regular expressions. Show how your algorithm works on the following inputs:

1. (abcd|abc)/d
2. (a|ab)/ba
3. aa*/a*

Exercises for Section 3.9

3.9.1

Extend the table of Fig. 3.58 to include the operators

1. ?
2. +

Answer

node n	nullable(n)	firstpos(n)
n = c_1 ?	true	firstpos(c_1)
n = c_1 +	nullable(c_1)	firstpos(c_1)

3.9.2

Use Algorithm 3.36 to convert the regular expressions of Exercise 3.7.3 directly to deterministic finite automata.

Answer

1. (a|b)*
   - Syntax tree
   - firstpos and lastpos for nodes in the syntax tree

```
- The function followpos

    <table>
        <thead>
            <tr>
                <th>node n</th>
                <th>followpos(n)</th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td>1</td>
                <td>{1, 2, 3}</td>
            </tr>
            <tr>
                <td>2</td>
                <td>{1, 2, 3}</td>
            </tr>
            <tr>
                <td>3</td>
                <td>∅</td>
            </tr>
        </tbody>
    </table>

 - Steps

     The value of firstpos for the root of the tree is {1, 2, 3}, so this set is the start state of D. Call this set of states A. We compute Dtran[A, a] and Dtran[A, b]. Among the positions of A, 1 correspond to a, while 2 correspond to b. Thus Dtran[A, a] = followpos(1) = {1, 2, 3},  Dtran[A, b] = followpos(2) = {1, 2, 3}. Both the results are set A, so dose not have new state, end the computation.

 - DFA

     ![3 9 2-1-dfa](https://f.cloud.github.com/assets/340282/457270/916fb2b6-b38f-11e2-9ad2-d3445e758b5e.gif)
```

2. (a*|b*)*
3. ((ε|a)|b*)*
4. (a|b)*abb(a|b)*

3.9.3 !

We can prove that two regular expressions are equivalent by
showing that their minimum-state DFA's are the same up to renaming of states.
Show in this way that the following regular expressions: (a|b)*, (a*|b*)*, and
((ε|a)b*)* are all equivalent. Note: You may have constructed the DFA's for

these expressions in response to Exercise 3.7.3.

Answer

Refer to the answers of 3.7.3 and 3.9.2-1

3.9.4 !

Construct the minimum-state DFA's for the following regular expressions:

1. (a|b)*a(a|b)
2. (a|b)*a(a|b)(a|b)
3. (a|b)*a(a|b)(a|b)(a|b)

Do you see a pattern?

3.9.5 !!

To make formal the informal claim of Example 3.25, show
that any deterministic finite automaton for the regular expression

(a|b)*a(a|b)...(a|b)

where (a|b) appears n - 1 times at the end, must have at least 2n states. Hint: Observe the pattern in Exercise 3.9.4. What condition regarding the history of inputs does each state represent?

第3章要点

1. 从 NFA、DFA 到正则表达式的转换

http://courses.engr.illinois.edu/cs373/sp2009/lectures/lect_08.pdf

2. KMP 及其扩展算法(p87)

参考 matrix 的博文 KMP算法详解。文中提供了例子，比较容易理解。

3. 字符串处理算法的效率(p103)

对于每个构造得到的 DFA 状态，我们最多必须构造 4|r| 个新状态

4. DFA 模拟中的时间和空间的权衡(p116)

图 3-66 表示的算法

5. 最小化一个 DFA 的状态数量（p115）

注意图 3-64 的第 4 行：“状态 s 和 t 在 a 上的转换都到达 Π 中的同一组”，而不是到达同一个状态。如果通过是否到达同一个状态来判定，那么如果 s 和 t 在 a 上的转换到了两个不同但不能区分的状态时，就会认为 s 和 t 是可区分的。

Exercises for Section 4.2

4.2.1

Consider the context-free grammar:

```
S -> S S + | S S * | a
```

and the string aa + a*.

1. Give a leftmost derivation for the string.
2. Give a rightmost derivation for the string.
3. Give a parse tree for the string.
4. ! Is the grammar ambiguous or unambiguous? Justify your answer.
5. ! Describe the language generated by this grammar.

Answer

1. S =lm=> SS* => SS+S* => aS+S* => aa+S* => aa+a*
2. S =rm=> SS* => Sa* => SS+a* => Sa+a* => aa+a*
   3.

3. Unambiguous
4. The set of all postfix expressions consist of addition and multiplication

4.2.2

Repeat Exercise 4 . 2 . 1 for each of the following grammars and strings:

1. S -> 0 S 1 | 0 1 with string 00011l.
2. S -> + S S | * S S | a with string + * aaa.
3. ! S -> S (S) S | ε with string (()())
4. ! S -> S + S | S S | (S) | S * | a with string (a+a)*a
5. ! S -> (L) | a 以及 L -> L, S | S with string ((a,a),a,(a))

6. !! S -> a S b S | b S a S | ε with string aabbab
7. The following grammar for boolean expressions:

```
bexpr -> bexpr or bterm | bterm
bterm -> bterm and bfactor | bfactor
bfactor -> not bfactor | (bexpr) | true | false
```

Answer

1. S =lm=> 0S1 => 00S11 => 000111
2. S =rm=> 0S1 => 00S11 => 000111
3. Omit
4. Unambiguous
5. The set of all strings of 0s and followed by an equal number of 1s

2、

1. S =lm=> +SS => +*SSS => +*aSS => +*aaS => +*aaa
2. S =rm=> +SS => +Sa => +*SSa => +*Saa => +*aaa
3. Omit
4. Unambiguous
5. The set of all prefix expressions consist of addition and multiplication.

3、

1. S =lm=> S(S)S => (S)S => (S(S)S)S => ((S)S)S => (()S)S => (()S(S)S)S => (()(S)S)S => (()()S)S => (()())S => (()())
2. S =rm=> S(S)S => S(S) => S(S(S)S) => S(S(S)) => S(S()) => S(S(S)S()) => S(S(S)()) => S(S()()) => S(()()) => (()())
3. Omit
4. Ambiguous
5. The set of all strings of symmetrical parentheses

4、

1. S =lm=> SS => S*S => (S)*S => (S+S)*S => (a+S)*S => (a+a)*S => (a+a)*a
2. S =rm=> SS => Sa => S*a => (S)*a => (S+S)*a => (S+a)*a => (a+a)*a
3. Omit
4. Ambiguous
5. The set of all string of plus, mupplication, 'a' and symmetrical parentheses, and plus is not the beginning and end of the position, multiplication is not

the beginning of the position

5、

1. S =lm=> (L) => (L, S) => (L, S, S) => ((S), S, S) => ((L), S, S) => ((L, S), S, S) => ((S, S), S, S) => ((a, S), S, S) => ((a, a), S, S) => ((a, a), a, S) => ((a, a), a, (L)) => ((a, a), a, (S)) => ((a, a), a, (a))
2. S =rm=> (L) => (L, S) => (L, (L)) => (L, (a)) => (L, S, (a)) => (L, a, (a)) => (S, a, (a)) => ((L), a, (a)) => ((L, S), a, (a)) => ((S, S), a, (a)) => ((S, a), a, (a)) => ((a, a), a, (a))
3. Omit
4. Unambiguous
5. Something like tuple in Python

6、

1. S =lm=> aSbS => aaSbSbS => aabSbS => aabbS => aabbaSbS => aabbabS => aabbab
2. S =rm=> aSbS => aSbaSbS => aSbaSb => aSbab => aaSbSbab => aaSbbab => aabbab
3. Omit
4. Ambiguous
5. The set of all strings of 'a's and 'b's of the equal number of 'a's and 'b's

7、 Unambiguous, boolean expression

4.2.3

Design grammars for the following languages:

1. The set of all strings of 0s and 1s such that every 0 is immediately followed by at least one 1.
2. ! The set of all strings of 0s and 1s that are palindromes; that is, the string reads the same backward as forward.
3. ! The set of all strings of 0s and 1s with an equal number of 0s and 1s.
4. !! The set of all strings of 0s and 1s with an unequal number of 0s and 1s.
5. ! The set of all strings of 0s and as in which 011 does not appear as a substring.
6. !! The set of all strings of 0s and 1s of the form xy, where x<>y and x and y are of the same length.

Answer

1、

```
S -> (0?1)*
```

2、

```
S -> 0S0 | 1S1 | 0 | 1 | ε
```

3、

```
S -> 0S1S | 1S0S | ε
```

5、

```
S -> 1*(0+1?)*
```

4.2.4

There is an extended grammar notation in common use.
In this notation, square and curly braces in production bodies are metasymbols (like -> or |) with the following meanings:

1. Square braces around a grammar symbol or symbols denotes that these constructs are optional. Thus, production A -> X[Y]Z has the same effect as the two productions A -> XYZ and A -> XZ.
2. Curly braces around a grammar symbol or symbols says that these symbols
may be repeated any number of times, including zero times. Thus,
A -> X{YZ} has the same effect as the infinite sequence of productions
A -> X, A -> XYZ, A -> XYZYZ, and so on.

Show that these two extensions do not add power to grammars; that is, any language that can be generated by a grammar with these extensions can be generated by a grammar without the extensions.

Proof

extended grammar	not extended grammar
A -> X[Y]Z	A -> XZ | XYZ
A -> X{YZ}	A -> XB B -> YZB | ε

4.2.5

Use the braces described in Exercise 4.2.4 to simplify the following grammar for statement blocks and conditional statements:

```
stmt -> if expr then stmt else stmt
      | if stmt them stmt
      | begin stmtList end
stmtList -> stmt; stmtList | stmt
```

Answer

```
stmt -> if expr then stmt [else stmt]
      | begin stmtList end
stmtList -> stmt [; stmtList]
```

4.2.6

Extend the idea of Exercise 4.2.4 to allow any regular expression of grammar symbols in the body of a production. Show that this extension does not allow grammars to define any new languages.

Proof

Every regular grammar has a corresponding not extended grammar

4.2.7 !

A grammar symbol X (terminal or nonterminal) is useless if there is no derivation of the form S =*=> wXy =*=> wxy. That is, X can never appear in the derivation of any sentence.

1. Give an algorithm to eliminate from a grammar all productions containing useless symbols.
2. Apply your algorithm to the grammar:

```
S -> 0 | A
A -> AB
B -> 1
```

4.2.8

The grammar in Fig. 4.7 generates declarations for a single numerical identifier; these declarations involve four different, independent properties of numbers.

```
stmt -> declare id optionList
optionList -> optionList option | ε
option -> mode | scale | precision | base
mode -> real | complex
scale -> fixed | floating
precision -> single | double
base -> binary | decimal
```

1. Generalize the grammar of Fig. 4.7 by allowing n options Ai, for some fixed n and for i = 1,2... ,n, where Ai can be either ai or bi· Your grammar should use only 0(n) grammar symbols and have a total length of productions that is O(n).
2. ! The grammar of Fig. 4.7 and its generalization in part (a) allow declarations
   that are contradictory and/or redundant, such as

   declare foo real fixed real floating

   We could insist that the syntax of the language forbid such declarations; that is, every declaration generated by the grammar has exactly one value for each of the n options. If we do, then for any fixed n there is only a finite number of legal declarations. The language of legal declarations thus has a grammar (and also a regular expression), as any finite language does. The obvious grammar, in which the start symbol has a production for every legal declaration has n! productions and a total production length of O(n x n!). You must do better: a total production length that is O(n2^n)

3. !! Show that any grammar for part (b) must have a total production length of at least 2n.
4. What does part (c) say about the feasibility of enforcing nonredundancy and noncontradiction among options in declarations via the syntax of the programming language?

Answer

1、

```
stmt -> declare id optionList
optionList -> optionList option | ε
option -> A_1 | A_2 | … | A_n
A_1 -> a_1 | b_1
A_2 -> a_2 | b_2
…
A_n -> a_n | b_n
```

4.3 节的练习

4.3.1

下面是一个只包含符号 a 和 b 的正则表达式文法。它使用 + 替代表示并运算的字符 | ，以避免和文法中作为元符号使用的竖线相混淆：

```
rexpr -> rexpr + rterm | rterm
rterm -> rterm rfactor | rfactor
rfactor -> rfactor * | rprimary
rprimary -> a | b
```

1. 对这个文法提取左公因子。
2. 提取左公因子的变换能使这个文法适用于自顶向下的语法分析技术吗?
3. 提取左公因子之后，从原文法中消除左递归。
4. 得到的文法适用于自顶向下的语法分析吗?

解答

1. 无左公因子

2. 不适合

3. 消除左递归

   ```
   rexpr -> rterm A
       A -> + rterm A | ε
   rterm -> rfactor B
       B -> rfactor B | ε
   ```

   rfactor -> rprimary C
       C -> * C | ε
   rprimary -> a | b

4. 适合?

4.3.2

对下面的文法重复练习 4.3.1

1. 练习 4.2.1 的文法
2. 练习 4.2.2-1 的文法
3. 练习 4.2.2-3 的文法
4. 练习 4.2.2-5 的文法
5. 练习 4.2.2-7 的文法

解答

1. S -> S S + | S S * | a
    1. 提取左公因子

        ```
        S -> S S A | a
        A -> + | *
        ```

    2. 不适合
    3. 消除左递归

        ```
        // initial status
        1) S -> S S A | a
        2) A -> + | *

        // i = 1
        1) S -> a B
        2) B -> S A B | ε
        3) A -> + | *

        // i = 2, j = 1
        1) S -> a B
        2) B -> a B A B | ε
        3) A -> + | *

        // i = 3, j = 1 ~ 2
        // nothing changed
        ```

    4. 适合
2. S -> 0 S 1 | 0 1
    1. 提取左公因子

```
S -> 0 A
A -> S 1 | 1
```

   2. 不适合，有间接左递归
   3. 消除左递归

```
// initial status
1) S -> 0 A
2) A -> S 1 | 1

// i = 1
// nothing changed

// i = 2, j = 1
1) S -> 0 A
2) A -> 0 A 1 | 1
```

   4. 合适
3. S -> S (S) S | ε
   1. 无左公因子
   2. 不合适
   3. 消除左递归

```
// initial status
1) S -> S (S) S | ε

// i = 1
1) S -> A
2) A -> (S) S A | ε

// i = 2, j = 1
// nothing changed
```

   4. 合适
4. S -> (L) | a 以及 L -> L, S | S
   1. 无左公因子
   2. 不合适

3. 消除左递归

   ```
   // initial status
   1) S -> (L) | a
   2) L -> L, S | S

   // i = 1
   // nothing changed

   // i = 2, j = 1
   1) S -> (L) | a
   2) L -> (L) A | a A
   3) A -> , S A | ε

   // i = 3, j = 1~2
   // nothing changed
   ```

4. 合适

4.3.3 !

下面文法的目的是消除 4.3.2 节中讨论的 “悬空-else 二义性”：

```
stmt -> if expr then stmt
        | matchedStmt
matchedStmt -> if expr then matchedStmt else stmt
               | other
```

说明这个文法仍然是二义性的。

解答

看一段示范代码，我们通过缩进来表示代码解析的层次结构

```
if expr
then
    if expr
    then matchedStmt
    else
        if expr
        then matchedStmt
else stmt
```

这段代码还可以被解析成

```
if expr
then
    if expr
    then matchedStmt
    else
        if expr
        then matchedStmt
        else stmt
```

所以这仍然是一个二义性的文法。原因在于 `matchedStmt -> if expr then matchedStmt else stmt` 中的最后一个 `stmt`，如果包含 `else` 语句的话，既可以认为是属于这个 `stmt` 的，也可以认为是属于包含这个 `matchedStmt` 的语句的。

4.4 节的练习

4.4.1

为下面的每个文法设计一个预测分析器，并给出预测分析表。你可能先要对文法进行提取左公因子或者消除左递归的操作。

练习 4.2.2 中 1 - 7 中的文法。

解答

1. S -> 0 S 1 | 0 1

   step1. 提取左公因子

   ```
   S -> 0 A
   A -> S 1 | 1
   ```

   step2. 消除左递归

   ```
   S -> 0 A
   A -> 0 A 1 | 1
   ```

   step3. 预测分析表

非终结符号	输入符号		
	0	**1**	**$**
S	S -> 0 A		
A	A -> 0 A 1	A -> 1	

2. S -> + S S | * S S | a

   step1. 无左公因子

   step2. 无左递归

   step3. 预测分析表

非终结符号	输入符号			
	+	*****	**a**	**$**
S	S -> + S S	S -> * S S	S -> a	

3. ! S -> S (S) S | ε

   step1. 无左公因子

   step2. 消除左递归

   ```
   S -> A
   A -> (S) S A | ε
   ```

   step3. 预测分析表

非终结符号	**输入符号**		
	(	**)**	**$**
S	S -> A	S -> A	S -> A
A	A -> (S) S A A -> ε	A -> ε	A -> ε

4. ! S -> S + S | S S | (S) | S * | a

   step1. 提取左公因子

   ```
   S -> SA | (S) | a
   A -> +S | S | *
   ```

   进一步提取终结符

   ```
   S -> SA | T
   A -> +S | S | *
   T -> (S) | a
   ```

   step2. 消除左递归(根据 p135 的算法 4.19)

```
i = 1
        S -> TB
        B -> AB | ε

i = 2
    j = 1
        A -> +S | TB | *

i = 3
    j = 1
        无需处理
    j = 2
        无需处理
```

得到最终的产生式

```
S -> TB
B -> AB | ε
A -> +S | TB | *
T -> (S) | a
```

step3. first && follow

```
first(T) = [(, a]
first(A) = [+, *] + first(T) =[+, *, (, a]
first(B) = [ε] + first(A) = [ε, +, *, (, a]
first(S) = first(T) = [(, a]

follow(T) = [$, +, *, (, a]
follow(A) = [$, +, *, (, ), a]
follow(B) = [$]
follow(S) = [$, +, *, (, ), a]
```

step4. 预测分析表

非终结符号	**输入符号**					
	(	**)**	**+**	*****	**a**	**$**
S	S -> TB				S -> TB	
B	B -> AB		B -> AB	B -> AB	B -> AB	B -> ε
A	A -> TB		A -> +S	A -> *	A -> TB	
T	T -> (S)				T -> a	

5. S -> (L) | a 以及 L -> L, S | S

   step1. 无左公因子

   step2. 消除左递归

```
S -> (L) | a
L -> SA
A -> ,SA | ε
```

   step3. 预测分析表

6. grammar for boolean expressions:

```
bexpr -> bexpr or bterm | bterm
bterm -> bterm and bfactor | bfactor
bfactor -> not bfactor | ( bexpr ) | true | false
```

   step1. 无左公因子

   step2. 消除左递归

```
bexpr -> bterm bexpr'
bexpr' -> or bterm bexpr' | ε
bterm -> bfactor bterm'
bterm' -> and bfactor bterm' | ε
bfactor -> not bfactor | (bexpr) | true | false
```

   step3. first && follow

```
first(bexpr) = first(bterm) = first(bfactor) = [not, (, true, false]
first(bexpr') = [or, ε]
first(bterm') = [and, ε]

follow(bexpr) = follow(bexpr') = [), $]
follow(bterm) = follow(bterm') = [or, $]
follow(bfactor) = [and, $]

```

   step4. 预测分析表

非终结符号	输入符号							
	and	**or**	**not**	**(**	**)**	**true**	**false**	**$**
bexpr			bexpr -> bterm bexpr'	bexpr -> bterm bexpr'		bexpr -> bterm bexpr'	bexpr -> bterm bexpr'	
bexpr'		bexpr' -> or bterm bexpr'			bexpr' -> ε			bexpr' -> ε
bterm			bterm -> bfactor bterm'	bterm -> bfactor bterm'		bterm -> bfactor bterm'	bterm -> bfactor bterm'	
bterm'		bterm' -> and bfactor bterm'			bterm' -> ε			bterm' -> ε
bfactor			bfactor -> not bfactor	bfactor -> (bexpr)		bfactor -> true	bfactor -> false	

4.4.2！！

有没有可能通过某种方法修改练习 4.2.1 中的文法，构造出一个与该练习中的语言（运算分量为 a 的后缀表达式）对应的预测分析器?

解答

```
S -> SS+ | SS* | a
```

step1. 提取左公因子

```
S -> SSA | a
A -> + | *
```

step2. 消除左递归

```
i = 1
        S -> aB
        B -> SAB | ε
        A -> + | *
i = 2
    j = 1
        S -> aB
        B -> aBAB | ε
        A -> + | *
```

step3. 预测分析表

非终结符号	**输入符号**			
	+	*****	**a**	**$**
S			S -> aB	
A	A -> +	A -> *		
B	B -> ε	B -> ε	B -> SAB	B -> ε

4.4.3

计算练习 4.2.1 的文法的 FIRST 和 FOLLOW 集合。

解答

- first(S) = [a]
- follow(S) = [a, +, *]

4.4.4

计算练习 4.2.2 中各个文法的 FIRST 和 FOLLOW 集合。

解答

1. S -> 0 S 1 | 0 1
    - first(S) = [0]
    - follow(S) = [1, $]
2. S -> + S S | * S S | a
    - first(S) = [+, *, a]

    - follow(S) = [+, *, a, $]
3. S -> S (S) S | ε
    - first(S) = [(, ε]
    - followS(S) = [), $]
4. S -> S + S | S S | (S) | S * | a
    - first(S) = [(, a]
    - follow(S) = [+, (,), a, *, $]
5. S -> (L) | a 以及 L -> L, S | S
    - first(S) = [(, a]
    - follow(S) = [",", $]
    - first(L) = first(S) = [(, a]
    - follow(L) = [), ",", $]
6. S -> a S b S | b S a S | ε
    - first(S) = [a, b, ε]
    - follow(S) = [a, b, $]
7. 下面的布尔表达式对应的文法：

```
bexpr -> bexpr or bterm | bterm
bterm -> bterm and bfactor | bfactor
bfactor -> not bfactor | (bexpr) | true | false
```

4.4.5

文法 S -> aSa | aa 生成了所有由 a 组成的长度为偶数的串。我们可以为这个文法设计一个带回溯的递归下降分析器。如果我们选择先用产生式 S -> aa 展开，那么我们只能识别串 aa。因此，任何合理的递归下降分析器将首先尝试 S -> aSa。

1. ！ 说明这个递归下降分析器识别输入 aa，aaaa 和 aaaaaaaa，但识别不了 aaaaaa。
2. ！！ 这个递归下降分析器识别什么样的语言?

注意

以下题目请参考 Aho 本人的讲义：Aho: Properties of Context-Free Languages，本地副本

此外还有另一篇内容相似的文章，本地副本

关于 CNF 和 CYK 算法，有较多相关资料，自行搜索

4.4.6 !

如果一个文法没有产生式体为 ε 的产生式，那么这个文法就是无 ε 产生式的。

1. 给出一个算法，他的功能是把任何文法转变成一个无 ε 产生式的生成相同语言的文法（唯一可能的例外是空串——没有哪个无 ε 产生式的文法能生成 ε）。提示：首先找出所有可能为空的非终结符号。非终结符号可能为空是指它（可能通过很长的推导）生成 ε。
2. 将你的算法应用于文法 S -> aSbS | bSaS | ε

4.4.7 ！

单产生式是指其产生式体为单个非终结符号的产生式，即形如 A -> B 的产生式。

1. 给出一个算法，它可以把任何文法转变成一个生成相同语言（唯一可能的例外是空串）的、无 ε 产生式、无单产生式的文法。提示：首先消除 ε 产生式，然后找出所有满足下列条件的非终结符号对 A 和 B：存在 A =*=> B。
2. 将你的算法应用于 4.1.2 节的算法。
3. 说明作为 （1） 的一个结果，我们可以把一个文法转换成一个没有环的等价文法。

4.4.8 ！ ！

如果一个文法的每个产生式要么形如 A -> BC，要么形如 A -> a，那么这个文法就成为 Chomsky 范式（Chomsky Normal Form， CNF）。说明如何将任意文法转变成一个生成相同语言（唯一可能的例外是空串——没有 CNF 文法可以生成 ε）的 CNF 文法。

4.4.9 ！

对于每个具有上下文无关的语法，其长度为 n 的串可以在 O(n^3) 的时间内完成识别。完成这种识别工作的一个简单方法称为 Cocke-Younger-Kasami（CYK）算法。该算法基于动态规划技术。也就是说，给定一个串 a_1a_2...a_n，我们构造出一个 nxn 的表 T 使得 T_ij 是可以生成子串 a_ia_i+1...aj 的非终结符号的集合。如果基础文法是 CNF 的，那么只要我们按照正确的顺序来填表：先填 j-i 值最小的条目，则表中的每一个条目都可以在 O(n) 时间内填写完毕。给出一个能够正确填写这个表的条目的算法，并说明你的算法的时间复杂度为 O(n^3)。填完这个表之后，你如何判断 a_1a_2...a_n 是否在这个语言中?

4.4.10 ！

说明我们如何能够在填好练习 4.4.9 中的表之后，在 O(n) 的时间内获得 a_1a_2...a_n 对应的一颗语法分析树？提示：修改练习 4.4.9 中的表 T，使得对于表的每个条目 T_ij 中的每个非终结符号 A，这个表同时记录了其他条目中的哪两个非终结符号组成的对偶使得我们将 A 放到 T_ij 中。

4.4.11 ！

修改练习 4.4.9 中的算法，使得对于任意符号串，他可以找出至少需要执行多少次插入、删除和修改错误（每个错误是一个字符）的操作才能将这个串变成基础文法的语言的句子。

4.4.12 !

```
stmt -> if e then stmt stmtTail
      | while e do stmt
      | begin list end
      | s
stmtTail -> else stmt
          | ε
list -> stmt listTail
listTail -> ; list
          | ε
```

上面的代码给出了对应于某些语句的文法。你可以将 e 和 s 当做分别代表条件表达式和“其他语句”的终结符号。如果我们按照下列方法来解决因为展开可选“else”（非终结符号 stmtTail）而引起的冲突：当我们从输入中看到一个 else 时就选择消耗掉这个 else。使用 4.4.5 节中描述的同步符号的思想：

1. 为这个文法构造一个带有错误纠正信息的预测分析表。
2. 给出你的语法分析器在处理下列输入时的行为：
    1. if e then s; if e then s end
    2. while e do begin s; if e then e; end

4.5 节的练习

4.5.1

对于练习 4.2.2（a）中的文法 S -> 0 S 1 | 0 1，指出下面各个最右句型的句柄。

1. 000111
2. 00S11

解答

1. 01
2. 0S1

4.5.2

对于练习 4.2.1 的文法 S -> S S + | S S * | a 和下面各个最右句型，重复练习 4.5.1 。

1. SSS+a*+
2. SS+a*a+
3. aaa*a++

解答

1. SS+
2. SS+
3. a

4.5.3

对于下面的输入符号串和文法，说明相应的自底向上语法分析过程。

1. 练习 4.5.1 的文法的串 000111 。
2. 练习 4.5.2 的文法的串 aaa*a++ 。

解答

1、 000111

栈	输入	句柄	动作
$	000111$		移入
$0	00111$		移入
$00	0111$		移入
$000	111$		移入
$0001	11$	01	规约：S -> 01
$00S	11$		移入
$00S1	1$	0S1	规约：S -> 0S1
$0S	1$		移入
$0S1	$	0S1	规约：S -> 0S1
$S	$		接受

2、 aaa*a++

栈	输入	句柄	动作
$	aaa*a++$		移入
$a	aa*a++$	a	规约: S -> a
$S	aa*a++$		移入
$Sa	a*a++$	a	规约: S -> a
$SS	a*a++$		移入
$SSa	*a++$	a	规约: S -> a
$SSS	*a++$		移入
$SSS*	a++$	SS*	规约: S -> SS*
$SS	a++$		移入
$SSa	++$	a	规约: S -> a
$SSS	++$		移入
$SSS+	+$	SS+	规约: S -> SS+
$SS	+$		移入
$SS+	$	SS+	规约: S -> SS+
$S	$		接受

4.6 节的练习

4.6.1

描述下列文法的所有可行前缀

1. 练习4.2.2-1的文法 S->0S1|01
2. ！ 练习4.2.1的文法 S->SS+|SS*|a
3. ！ 练习4.2.2-3的文法 S->S(S)S|ε

解答

以下提取左公因子和消除左递归后的文法均由练习 4.3.2 得到

1. 提取左公因子和消除左递归后的增广文法

   ```
   0) S' -> S
   1) S -> 0 A
   2) A -> 0 A 1
   3) A -> 1
   ```

   LR(0) 自动机

   可行前缀为 `0+A?1?`

2. 提取左公因子和消除左递归后的增广文法

   ```
   0) S' -> S
   1) S -> a B
   2) B -> a B A B
   3) B -> ε
   4) A -> +
   5) A -> *
   ```

   LR(0) 自动机

   可行前缀为 `aB?|a{2,∞}(BAa+)*(B|B+|B*|BA|BAB)?`

3. 提取左公因子和消除左递归后的增广文法

```
0) S' -> S
1) S -> A
2) A -> (S) S A
3) A -> ε
```

LR(0) 自动机

箭头太复杂，懒得归纳了

4.6.2

为练习4.2.1中的（增广）文法构造SLR项集。计算这些项集的GOTO函数。给出这个函数的语法分析表。这个文法是SLR文法吗?

解答

该文法的项集和 GOTO 函数见 4.6.1-2

FOLLOW 函数如下：

```
FOLLOW(S) = [$]
FOLLOW(A) = [a, $]
FOLLOW(B) = [+, * ,$]
```

语法分析表如下：

状态	ACTION				GOTO		
	a	**+**	*****	**$**	**S**	**A**	**B**
0	s2				s1		
1				acc			
2	s4	r3	r3	r3			s3
3				r1			
4	s4	r3	r3	r3			s5
5		s7	s8			s6	
6	s4	r3	r3	r3			s9
7	r4			r4			
8	r5			r5			
9		r2	r2	r2			

无冲突，这显然是一个 SLR 文法

4.6.3

利用练习4.6.2得到的语法分析表，给出处理输入aa*a+时的各个动作。

解答

	栈	符号	输入	动作
1)	0		aa*a+$	移入
2)	02	a	a*a+$	移入
3)	024	aa	*a+$	根据 B -> ε 规约
4)	0245	aaB	*a+$	移入
5)	02458	aaB*	a+$	根据 A -> * 规约
6)	02456	aaBA	a+$	移入
7)	024564	aaBAa	+$	根据 B -> ε 规约
8)	0245645	aaBAaB	+$	移入
9)	02456457	aaBAaB+	$	根据 A -> + 规约
9)	02456456	aaBAaBA	$	根据 B -> ε 规约
10)	024564569	aaBAaBAB	$	根据 B -> aBAB 规约
11)	024569	aaBAB	$	根据 B -> aBAB 规约
12)	023	aB	$	根据 S -> aB 规约
13)	01	S	$	接受

4.6.4

对于练习4.2.2-1~4.2.2-7中的各个（增广）文法：

1. 构造SLR项集和他们的GOTO函数
2. 指出你的项集中的所有动作冲突
3. 如果存在SLR语法分析表，构造出这个语法分析表

4.6.5

说明下面的文法

```
S->AaAb|BbBa
A->ε
B->ε
```

是LL(1)的，但不是SLR(1)的。

解答

1. 该文法是 LL(1) 的

   见 4.4.3 节，p142 的判定标准

2. 该文法不是 SLR(1) 的

   ```
   I_0

   S' -> .S
   S -> .AaAb
   S -> .BbBa
   A -> .
   B -> .
   ```

   由于 FOLLOW(A) = FOLLOW(B) = [a, b]，所以当 I_0 后输入为 a 或 b 时，就会发生规约冲突。

4.6.6

说明下面的文法

```
S->SA|A
A->a
```

是SLR(1)的，但不是LL(1)的

解答

1. 该文法不是 LL(1) 的

   `S -> SA` 和 `S -> A` 均能推导出以 a 开头的串，所以不是 LL(1) 的

2. 该文法是 SLR(1) 的

   该文法生成的语法分析表是没有冲突的

4.6.7!!

考虑按照下面的方式定义的文法族 G_n：

```
S -> A_i b_i           其中1<=i<=n
A_i-> a_j A_j | a_j     其中1<=i,j<=n 且i<>n
```

说明：

1. G_n有 2n^2-n 个产生式
2. G_n有 $2^{n+n}2+n$ 个 LR(0) 项集
3. G_n是 SLR(1) 的

关于LR语法分析器的大小，这个分析结果说明了什么？

4.6.8!

我们说单个项可以看做一个 NFA 的状态，而有效项的集合就是一个 DFA 的状态。对于练习4.2.1的文法 S->SS+|SS*|a

1. 根据“将项看作一个NFA的状态”部分中的规则，画出这个文法的有效的转换图（NFA）
2. 将子集构造算法（算法3.20）应用于在（1）部分构造得到的NFA。得到的DFA和这个文法的LR(0)项集比有什么关系
3. ！！ 说明在任何情况下，将子集构造算法应用于一个文法的有效项的NFA所得到的就是该文法的 LR(0) 项集

4.6.9!

下面是一个二义性的文法

```
S->AS|b
A->SA|a
```

构造出这个文法的规范LR(0)项集族。如果我们试图为这个文法构造出一个LR语法分析表，必然会存在某些冲突动作。都有哪些冲突动作？假设我们使用这个语法分析表，并且在出现冲突时不确定地选择一个动作。给出输入abab时所有可能的动作序列

4.7 节的练习

4.7.1

为练习 4.2.1 的文法 S -> S S + | S S * | a 构造

1. 规范 LR 项集族
2. LALR 项集族

4.7.2

对练习 4.2.2-1 ~ 4.4.2-7 的各个文法重复练习 4.7.1

! 4.7.3

对练习 4.7.1 的文法，使用算法 4.63，根据该文法的 LR(0) 项集的内核构造出它的 LALR 项集族

! 4.7.4

说明下面的文法

```
S -> A a | b A c | d c | b d a
A -> d
```

是 LALR(1) 的，但不是 SLR(1) 的

! 4.7.5

说明下面的文法

```
S -> A a | b A c | B c | b B a
A -> d
B -> d
```

是 LR(1) 的，但不是 LALR(1) 的

第4章要点

！LR(0), SLR, LR, LALR 之间的区别

p157: LR(0) 自动机是如何做出移入-规约决定的？假设文法符号串 γ 使得 LR(0) 自动机从开始状态 0 运行到某个状态 j，那么如果下一个输入符号为 a 且状态 j 有一个在 a 上的转换，就移入 a，否则就进行规约。

这种方法会导致一些错误的规约，假定规约后的符号为 X，但 a 并不在 FOLLOW(X) 中，这种情况下就会有问题。所以 SLR 在这方面进行了改进。

p161：构造一个 SLR 分析表时，如果 [A -> α.] 在 I_i 中，那么对于 FOLLOW(A) 中的所有 a，将 ACTION[i, a] 设置为 “规约 A -> α”

SLR 一定程度上解决了错误规约的问题，但没有完全解决。因为虽然 a 在 FOLLOW(A) 中才会选择规约，但是就当前所处的状态 I_i 而言，并不是每个 FOLLOW(A) 中的终结符都可以出现在状态 I_i 中的 A 后面。

p166: 用更正式一点的语言来讲，必须要为 I_i 精确得指明哪些输入符号可以更在句柄 α 后面，从而使 α 可以被规约为 A。

LR 通过在项中加入第二个分量，即向前看符号来解决这个问题。但新的问题是 LR 会使得状态表及其庞大，而 LALR 就是一种比较经济的做法，它具有和 SLR 一样多的状态。

p170：一般地说，通过将具有相同核心项集的 LR 项集合并，可以得到 LALR 项集。虽然 LALR 可能会进行一些错误的规约，但最终会在输入任何新的符号之前发现这个错误。

消除二义性 （p134）

图 4-10，如何得出这个消除方法的？

消除左递归 （p135）

为什么图 4-11 的算法能消除文法中的左递归?

消除递归需满足两个条件：

1. 不存在立即左递归，即不存在形似这样的产生式 A -> Aα 。
2. 不存在由多步推导可产生的左递归。

算法 3~5 行循环的结果使得形如 A_i -> A_m α 的产生式一定满足 m >= i ，就消除了形如 S => Aa => Sda 这样的转换可能，也就是说由 A_m 一定推导不出以 A_i 开头的产生式，A_m α 就不存在产生 A-i 左递归的可能。

同时需要注意的是： 只需要处理 A_i -> A_j α 这样的产生式，而不需要处理形如 A_i -> α A_j β 这样的产生式

循环完成后，第 6 行消除了替换后的产生式中的立即左递归。

使用 LR(0) 创建出 LALR(1) 项集的内核 （p173）

自发生成的和传播的向前看符号

CNF 和 BNF

- Chomsky normal form
- Backus Naur Form

5.1 节的练习

5.1.1

对于图 5-1 中的 SDD，给出下列表达式对应的注释语法分析树

1. (3+4)*(5+6)n
2. 1*2*3*(4+5)n
3. (9+8*(7+6)+5)*4n

解答

1. (3+4)*(5+6)n

2. 1*2*3*(4+5)n

5.1.2

扩展图 5-4 中的 SDD，使它可以像图 5-1 所示的那样处理表达式

解答

	产生式	语法规则
1)	L -> En	L.val = E.val
2)	E -> TE'	E'.inh = T.val E.val = E'.syn
3)	E' -> +TE_1'	E_1'.inh = E'.inh + T.val E'.syn = E_1'.syn
4)	E' -> ε	E'.syn = E'.inh
5)	T -> FT'	T'.inh = F.val T.val = T'.syn
6)	T' -> *FT_1'	T_1'.inh = T'.inh * F.val T'.syn = T_1'.syn
7)	T' -> ε	T'.syn = T'.inh
8)	F -> (E)	F.val = E.val
9)	F -> digit	F.val = digit.lexval

5.1.3

使用你在练习 5.1.2 中得到的 SDD，重复练习 5.1.1

解答

1. (3+4)*(5+6)n

2. 1*2*3*(4+5)n

5.2 节的练习

5.2.1

图 5-7 中的依赖图的全部拓扑顺序有哪些

解答

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
[ 1, 2, 3, 5, 4, 6, 7, 8, 9 ],
[ 1, 2, 4, 3, 5, 6, 7, 8, 9 ],
[ 1, 3, 2, 4, 5, 6, 7, 8, 9 ],
[ 1, 3, 2, 5, 4, 6, 7, 8, 9 ],
[ 1, 3, 5, 2, 4, 6, 7, 8, 9 ],
[ 2, 1, 3, 4, 5, 6, 7, 8, 9 ],
[ 2, 1, 3, 5, 4, 6, 7, 8, 9 ],
[ 2, 1, 4, 3, 5, 6, 7, 8, 9 ],
[ 2, 4, 1, 3, 5, 6, 7, 8, 9 ]
```

算法见 5.2.1.js

5.2.2

对于图 5-8 中的 SDD，给出下列表达式对应的注释语法分析树：

1. int a, b , c
2. float w, x, y, z

解答

1. int a, b, c

5.2.3

假设我们有一个产生式 A -> BCD。A, B, C, D 这四个非终结符号都有两个属性，综合属性 s 和继承属性 i。对于下面的每组规则，指出（1）这些规则是否满足 S 属性定义的要求（2）这些规则是否满足 L 属性定义的要求（3）是否存在和这些规则一致的求值过程？

1. A.s = B.i + C.s
2. A.s = B.i + C.s , D.i = A.i + B.s
3. A.s = B.s + D.s
4. ! A.s = D.i , B.i = A.s + C.s , C.i = B.s , D.i = B.i + C.i

解答

1. 否， ？
2. 否， 是
3. 是， 是
4. 否， 否

5.2.4 !

这个文法生成了含“小数点”的二进制数：

```
S -> L.L|L
L -> LB|B
B -> 0|1
```

设计一个 L 属性的 SDD 来计算 S.val，即输入串的十进制数值。比如，串 101.101 应该被翻译为十进制数 5.625。

解答

	产生式	语法规则

1)	S -> L_1.L_2	L_1.isLeft = true
		L_2.isLeft = false
		S.val = L_1.val + L_2.val

2)	S -> L	L.isLeft = true
		S.val = L.val

3)	L -> L_1B	L_1.isLeft = L.isLeft
		L.len = L_1.len + 1
		L.val = L.isLeft ? L_1.val * 2 + B.val : L_1.val + B.val * 2^(-L.len)

4)	L -> B	L.len = 1
		L.val = L.isLeft ? B.val : B.val/2

5)	B -> 0	B.val = 0
6)	B -> 1	B.val = 1

其中：

- isLeft 为继承属性，表示节点是否在小数点的左边
- len 为综合属性，表示节点包含的二进制串的长度
- val 为综合属性

5.2.5！！

为练习 5.2.4 中描述的文法和翻译设计一个 S 属性的 SDD。

解答

	产生式	语法规则
1)	S -> L_1.L_2	S.val = L_1.val +　L_2.val/L_2.f
2)	S -> L	S.val = L.val

3)	L -> L_1B	L.val = L_1.val*2 + B.val
		L.f = L_1.f * 2

4)	L -> B	L.val = B.val
		L.f = 2

5)	B -> 0	B.val = 0
6)	B -> 1	B.val = 1

5.2.6！！

使用一个自顶向下的语法分析文法上的 L 属性 SDD 来实现算法 3.23。这个算法把一个正则表达式转换为一个 NFA。假设有一个表示任意字符的词法单元 char，并且 char.lexval 是它所表示的字符。你可以假设存在一个函数 new()，该函数范围一个新的状态页就是一个之前尚未被这个函数返回的状态。使用任何方便的表示来描述这个 NFA 的翻译。

5.3 节的练习

5.3.1

下面是涉及运算符 + 和整数或浮点运算分量的表达式的文法。区分浮点数的方法是看它有无小数点。

```
E -> E + T | T
T -> num.num | num
```

1. 给出一个 SDD 来确定每个项 T 和表达式 E 的类型
2. 扩展这个得到的 SDD，使得它可以把表达式转换成为后缀表达式。使用一个单目运算符 intToFloat 把一个整数转换为相等的浮点数。

解答

1.

	产生式	语法规则
1)	E -> E_1 + T	E.type = E_1.type === float || T.type === float ? float : int
2)	E -> T	E.type = T.type
3)	T -> num.num	T.type = float
4)	T -> num	T.type = int

5.3.2 !

给出一个 SDD，将一个带有 + 和 * 的中缀表达式翻译成没有冗余括号的表达式。比如因为两个运算符都是左结合的，并且 * 的优先级高于 +，所以 ((a*(b+c))*(d)) 可翻译为 a*(b+c)*d

解答

几个属性设置：

- wrapped: 表达式最外层是否有括号。
- precedence: 令 +，*，() 和单 digit 的优先级分别为 0，1，2，3。 如果表达式最外层有括号，则为去掉括号后最后被计算的运算符的优先级，否则为表达式最后被计算的运算符的优先级。
- expr: 表达式。
- cleanExpr: 去除了冗余括号的表达式。

	产生式	语法规则
1)	L -> En	L.cleanExpr = E.wrapped ? E.cleanExpr : E.expr

2)	E -> E_1 + T	E.wrapped = false
		E.precedence = 0
		E.expr = E_1.expr || "+" || T.expr
		E.cleanExpr = (E_1.wrapped ? E_1.cleanExpr : E_1.expr) || "+" || (T.wrapped ? T.cleanExpr : T.expr)

3)	E -> T	E.wrapped = T.wrapped
		E.precedence = T.precedence
		E.expr = T.expr
		E.cleanExpr = T.cleanExpr

4)	T -> T_1 * F	T.wrapped = false
		T.precedence = 1
		T.expr = T_1.expr || "*" || F.expr
		T.cleanExpr = (T_1.wrapped && T_1.precedence >= 1 ? T_1.cleanExpr : T_1) || * || (F.wrapped && F.precedence >= 1 ? F.cleanExpr : F.expr)

5)	T -> F	T.wrapped = F.wrapped
		T.precedence = F.precedence
		T.expr = F.expr
		T.cleanExpr = F.cleanExpr

6)	F -> (E)	F.wrapped = true
		F.precedence = E.precedence
		F.expr = "(" || E.expr || ")"
		F.cleanExpr = E.expr

7)	F -> digit	F.wrapped = false
		F.precedence = 3
		F.expr = digit
		F.cleanExpr = digit

5.3.3 !

给出一个 SDD 对 x*(3*x+x*x) 这样的表达式求微分。表达式中涉及运算符 + 和 * 、变量 x 和常量。假设不进行任何简化，也就是说，比如 3*x 将被翻译为 3*1+0*x。

5.4 节的练习

5.4.1

我们在 5.4.2 节中提到可能根据语法分析栈中的 LR 状态来推导出这个状态表示了什么文法符号。我们如何推导这个信息?

解答

见算法 4.44

5.4.2

改写下面的 SDT:

```
A -> A {a} B | A B {b} | 0
B -> B {c} A | B A {d} | 1
```

使得基础文法变成非左递归的。

5.4.3！

下面的 SDT 计算了一个由 0 和 1 组成的串的值。它把输入的符号串当做正二进制数来解释。

```
B -> B_1 0 {B.val = 2 * B_1.val}
   | B_1 1 {B.val = 2 * B_1.val + 1}
   | 1 {B.val = 1}
```

改写这个 SDT，使得基础文法不再是左递归的，但仍然可以计算出整个输入串的相同的 B.val 的值。

解答

提取左公因子

```
B -> B_1 digit {B.val = 2 * B_1.val + digit.val}
   | 1 {B.val = 1}
digit -> 0 {digit.val = 0}
       | 1 {digit.val = 1}
```

在形如 `A = A a | b` 的左递归产生式中， a 为 `digit {B.val = 2 * B_1.val + digit.val}`， b 为 `1`

消除左递归后得

```
B -> 1 {A.i = 1} A
A -> digit {A_1.i = 2 * A.i + digit.val} A_1 {A.val = A_1.val}
   | ε {A.val = A.i}
digit -> 0 {digit.val = 0}
       | 1 {digit.val = 1}
```

5.4.4 !

为下面的产生式写出一个和例 5.19 类似的 L 属性 SDD。这里的每个产生式表示一个常见的 C 语言那样的控制流结构。你可能需要生成一个三地址语句来跳转到某个标号 L，此时你可以生成语句 goto L。

1. S -> if (C) S_1 else S_2
2. S -> do S_1 while (C)
3. S -> '{' L '}'; L -> L S | ε

请注意，列表中的任何语句都可能包含一条从它的内部跳转到下一个语句的跳转指令，因此简单地为各个语句按顺序生成代码是不够的。

解答

1. S -> if (C) S_1 else S_2

   ```
   L_1 = new()
   C.false = L_1
   S_1.next = S.next
   S.code = C.code || S_1.code || label || L_1 || S_2.code
   ```

2. S -> do S_1 while (C)

```
L_1 = new()
C.true = L_1
S.code = label || L_1 || S_1.code || C.code
```

5.4.5

按照例 5.19 的方法，把在练习 5.4.4 中得到的各个 SDD 转换成一个 SDT。

解答

1. S -> if (C) S_1 else S_2

```
S -> if (      {new L_1; C.false = L_1}
     C )       {S_1.next = S.next}
     S_1 else
     S_2       {S.code = C.code || S_1.code || label || L_1 || S_2.code}
```

2. S -> do S_1 while (C)

```
S -> do             {new L_1}
     S_1 while (    {C.true = L_1}
     C )            {S.code = label || L_1 || S_1.code || C.code}
```

5.4.6

修改图 5.25 中的 SDD，使它包含一个综合属性 B.le，即一个方框的长度。两个方框并列后得到的方框的长度是这两个方框的长度和。然后把你的新规则加入到图 5.26 中 SDT 的合适位置上。

5.4.7

修改图 5.25 中的 SDD，使它包含上标，用方框之间的运算符 sup 表示。如果方框 B_2 是方框 B_1 的一个上标，那么将 B_2 的基线放在 B_1 的基线上方，两条基线的距离是 0.6 乘以 B_1 的大小。把新的产生式和规则加入到图 5.26 的 SDT 中去。

5.4.6 和 5.4.7 的解答

```
1) S -> B                B.ps = 10
```

```
                                B.wd = 

2) S -> B_1 B_2                 B_1.ps = B.ps
                                B_2.ps = B.ps
                                B.wd = B_1.wd + B_2.wd
                                B.ht = max(B_1.ht, B_2.ht)
                                B.dp = max(B_1.dp, B_2.dp)

3) B -> B_1 sub B_2             B_1.ps = B.ps
                                B_2.ps = 0.7 * B.ps
                                B.wd = B_1.wd + B_2.wd
                                B.ht = max(B_1.ht, B_2.ht - 0.25 * B.ps)
                                B.dp = max(B_1.dp, B_2.dp + 0.25 * B.ps)

4) B -> B_1 sup B_2             B_1.ps = B.ps
                                B_2.ps = 0.6 * B.ps
                                B.wd = B_1.wd + B_2.wd
                                B.ht = max(B_1.ht, B_2.ht + 0.6 * B.ps)
                                B.dp = max(B_1.dp, B_2.dp - 0.6 * B.ps)

5) B -> ( B_1 )                 B_1.ps = B.ps
                                B.wd = B_1.wd
                                B.ht = B_1.ht
                                B.dp = B_1.dp

6) B -> text                    B.wd = getWd(B.ps, text.lexval)
                                B.ht = getHt(B.ps, text.lexval)
                                B.dp = getDp(B.ps, text.lexval)
```

5.5 节的练习

5.5.1

按照 5.5.1 节的风格，将练习 5.4.4 中得到的每个 SDD 实现为递归下降的语法分析器。

5.5.2

按照 5.5.2 节的风格，将练习 5.4.4 中得到的每个 SDD 实现为递归下降的语法分析器。

5.5.3

按照 5.5.3 节的风格，将练习 5.4.4 中得到的每个 SDD 和一个 LL 语法分析器一起实现。它们应该边扫描输入边生成代码。

5.5.4

按照 5.5.3 节的风格，将练习 5.4.4 中得到的每个 SDD 和一个 LL 语法分析器一起实现，但是代码（或者指向代码的指针）存放在栈中。

5.5.5

按照 5.5.4 节的风格，将练习 5.4.4 中得到的每个 SDD 和一个 LR 语法分析器一起实现。

5.5.6

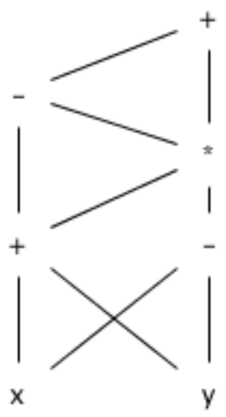
按照 5.5.1 节 的风格实现练习 5.2.4 中得到的 SDD。按照 5.5.2 节的风格得到的实现和这个实现相比有什么不同吗?

6.1 节的练习

为下面的表达式构造 DAG

```
((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))
```

解答

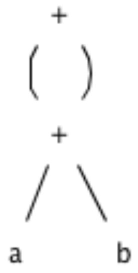


为下列表达式构造 DAG，且指出他们每个子表达式的值编码。假定 + 是左结合的。

1. a+b+(a+b)
2. a+b+a+b
3. a+a+(a+a+a+(a+a+a+a))

解答

1. a+b+(a+b)



1	id	a	
2	id	b	
3	+	1	2
4	+	3	3

2. a+b+a+b

1	id	a	
2	id	b	
3	+	1	2
4	+	3	1
5	+	4	2

3. a+a+(a+a+a+(a+a+a+a))

1	id	a	
2	+	1	1
3	+	2	1
4	+	3	1
5	+	3	4
6	+	2	5

6.2 节的练习

6.2.1

将算数表达式 a+-(b+c) 翻译成

1. 抽象语法树
2. 四元式序列
3. 三元式序列
4. 间接三元式序列

解答

1. 抽象语法树

2. 四元式序列

	op	arg1	arg2	result
0	+	b	c	t1
1	minus	t1		t2
2	+	a	t2	t3

3. 三元式序列

	op	arg1	arg2
0	+	b	c
1	minus	(0)	
2	+	a	(1)

4. 间接三元式序列

	op	arg1	arg2
0	+	b	c
1	minus	(0)	
2	+	a	(1)

	instruction
0	(0)
1	(1)
2	(2)

参考

- 间接三元式更详细的讲解

6.2.2

对下列赋值语句重复练习 6.2.1

1. a = b[i] + c[j]
2. a[i] = b*c - b*d
3. x = f(y+1) + 2
4. x = *p + &y

解答

1. a = b[i] + c[j]
   - 四元式

     ```
     0) =[]    b    i    t1
     1) =[]    c    j    t2
     2) +      t1   t2   t3
     3) =      t3        a
     ```

   - 三元式

```
0) =[]   b    i
1) =[]   c    j
2) +     (0)  (1)
3) =     a    (2)
```

- 间接三元式

```
0) =[]   b    i
1) =[]   c    j
2) +     (0)  (1)
3) =     a    (2)

0)
1)
2)
3)
```

2. a[i] = b*c - b*d
   - 四元式

```
0) *    b    c    t1
1) *    b    d    t2
2) -    t1   t2   t3
3) []=  a    i    t4
4) =    t3        t4
```

   - 三元式

```
0) *    b    c
1) *    b    d
2) -    (0)  (1)
3) []=  a    i
4) =    (3)  (2)
```

   - 间接三元式

```
0) *    b    c
1) *    b    d
2) -    (0)  (1)
3) []=  a    i
4) =    (3)  (2)

0)
1)
2)
3)
4)
```

3. x = f(y+1) + 2

    - 四元式

    ```
    0) +        y    1    t1
    1) param    t1
    2) call     f    1    t2
    3) +        t2    2   t3
    4) =        t3        x
    ```

    - 三元式

    ```
    0) +        y     1
    1) param    (0)
    2) call     f     1
    3) +        (2)   2
    4) =        x     (3)
    ```

    - 间接三元式

```
0) +        y     1
1) param    (0)
2) call     f     1
3) +        (2)   2
4) =        x     (3)

0)
1)
2)
3)
4)
```

参考

- 数组元素的取值和赋值

6.2.3 !

说明如何对一个三地址代码序列进行转换，使得每个被定值的变量都有唯一的变量名。

6.3 节的练习

6.3.1

确定下列声明序列中各个标识符的类型和相对地址。

```
float x;
record {float x; float y;} p;
record {int tag; float x; float y;} q;
```

解答

SDT

```
S ->                    {top = new Evn(); offset = 0;}
     D
D -> T id;              {top.put(id.lexeme, T.type, offset);
                         offset += T.width}
     D1
D -> ε
T -> int                {T.type = interget; T.width = 4;}
T -> float              {T.type = float; T.width = 8;}
T -> record '{'
                        {Evn.push(top), top = new Evn();
                         Stack.push(offset), offset = 0;}
     D '}'              {T.type = record(top); T.width = offset;
                         top = Evn.top(); offset = Stack.pop();}
```

标识符类型和相对地址

```
line id      type        offset   Evn

  1) x       float       0        1

  2) x       float       0        2
  2) y       float       8        2
  2) p       record()    8        1

  3) tag     int         0        3
  3) x       float       4        3
  3) y       float       12       3
  3) q       record()    24       1
```

6.3.2 !

将图 6-18 对字段名的处理方法扩展到类和单继承的层次结构。

1. 给出类 Evn 的一个实现。该实现支持符号表链，使得子类可以重定义一个字段名，也可以直接引用某个超类中的字段名。
2. 给出一个翻译方案，该方案能够为类中的字段分配连续的数据区域，这些字段中包含继承而来的域。继承而来的字段必须保持在对超类进行存储分配时获得的相对地址。

6.4 节的练习

6.4.1

向图 6-19 的翻译方案中加入对应于下列产生式的规则：

1. E -> E1 * E2
2. E -> +E1

解答

```
产生式              语义规则

E -> E1 * E2       { E.addr = new Temp();
                     E.code = E1.code || E2.code ||
                              gen(E.addr '=' E1.addr '*' E2.addr);
}

    | +E1          { E.addr = E1.addr;
                     E.code = E1.code; }
```

6.4.2

使用图 6-20 的增量式翻译方案重复练习 6.4.1

解答

```
产生式              语义规则

E -> E1 * E2       { E.addr =  new Temp();
                     gen(E.addr '=' E1.addr '*' E2.addr; }

    | +E1          { E.addr = E1.addr; }
```

6.4.3

使用图 6-22 的翻译方案来翻译下列赋值语句：

1. x = a[i] + b[j]
2. x = a[i][j] + b[i][j]
3. ! x = a[b[i][j]][c[k]]

解答

1. x = a[i] + b[j]

   语法分析树:

   三地址代码

   ```
   t_1 = i * awidth
   t_2 = a[t_1]
   t_3 = j * bwidth
   t_4 = b[t_3]
   t_5 = t_2 + t_4
   x = t_5
   ```

2. x = a[i][j] + b[i][j]

   语法分析树：

   三地址代码：

   ```
   t_1 = i * ai_width
   t_2 = j * aj_width
   t_3 = t_1 + t_2
   t_4 = a[t_3]
   t_5 = i * bi_width
   t_6 = j * bj_width
   t_7 = t_5 + t_6
   t_8 = b[t_7]
   t_9 = t_4 + t_8
   x = t_9
   ```

3. ! x = a[b[i][j]][c[k]]

6.4.4 !

修改图 6-22 的翻译方案，使之适合 Fortran 风格的数据引用，也就是说 n 维数组的引用为 id[E1, E2, ..., En]

解答

仅需修改 L 产生式（同图 6-22 一样，未考虑消除左递归）

```
L -> id[A]   { L.addr = A.addr;
               global.array = top.get(id.lexeme); }

A -> E       { A.array = global.array;
               A.type = A.array.type.elem;
               A.addr = new Temp();
               gen(A.addr '=' E.addr '*' A.type.width; }

A -> A1,E    { A.array = A1.array;
               A.type = A1.type.elem;
               t = new Temp();
               A.addr = new Temp();
               gen(t '=' E.addr '*' A.type.length);
               gen(A.addr '=' A1.addr '+' t); }
```

注意

令 a 表示一个 i*j 的数组，单个元素宽度为 w

```
a.type = array(i, array(j, w))
a.type.length = i
a.type.elem = array(j, w)
```

6.4.5

将公式 6.7 推广到多维数据上，并指出哪些值可以被存放到符号表中并用来计算偏移量。考虑下列情况：

1. 一个二维数组 A，按行存放。第一维的下标从 l_1 到 h_1，第二维的下标从 l_2 到 h_2。单个数组元素的宽度为 w。
2. 其他条件和 1 相同，但是采用按列存放方式。
3. ！ 一个 k 维数组 A，按行存放，元素宽度为 w，第 j 维的下标从 l_j 到 h_j。
4. ！ 其他条件和 3 相同，但是采用按列存放方式。

解答

令 n_i 为第 i 维数组的元素个数，计算公式：n_i = h_i - l_i + 1

```
3. A[i_1]]…[i_k] = base +
                   (
                       (i_1 - l_1) * n_2 * … * n_k +
                       … +
                       (i_k-1 - l_k-1) * n_k +
                       (i_k - l_k)
                   ) * w

4. A[i_1]]…[i_k] = base +
                   (
                       (i_1 - l_1) +
                       (i_2 - l_2) * n_1 +
                       … +
                       (i_k - l_k) * n_k-1 * n_k-2 * … * n_1
                   ) * w
```

6.4.6

一个按行存放的整数数组 A[i, j] 的下标 i 的范围为 1~10，下标 j 的范围为 1~20。每个整数占 4 个字节。假设数组 A 从 0 字节开始存放，请给出下列元素的位置：

1. A[4, 5]
2. A[10, 8]
3. A[3, 17]

解答

计算公式：((i-1) * 20 + (j-1)) * 4

1. (3 * 20 + 4) * 4 = 256
2. (9 * 20 + 7) * 4 = 748
3. (2 * 20 + 16) * 4 = 224

6.4.7

假定 A 是按列存放的，重复练习 6.4.6

解答

计算公式：((j-1) * 10 + (j-1)) * 4

1. (4 * 10 + 3) * 4 = 172
2. (7 * 10 + 9) * 4 = 316
3. (16 * 10 + 2) * 4 = 648

6.4.8

一个按行存放的实数型数组 A[i, j, k] 的下标 i 的范围为 1~4，下标 j 的范围为 0~4，且下标 k 的范围为 5~10。每个实数占 8 个字节。假设数组 A 从 0 字节开始存放，计算下列元素的位置：

1. A[3, 4, 5]
2. A[1, 2, 7]
3. A[4, 3, 9]

解答

计算公式：((i-1) * 5 * 6 + j * 6 + (k-5)) * 8

1. ((3-1) * 5 * 6 + 4 * 6 + (5-5)) * 8 = 672
2. ((1-1) * 5 * 6 + 2 * 6 + (7-5)) * 8 = 112
3. ((4-1) * 5 * 6 + 3 * 6 + (9-5)) * 8 = 896

6.4.9

假定 A 是按列存放的，重复练习 6.4.8

解答

计算公式：((i-1) + j * 4 + (k-5) * 5 * 4) * 8

1. ((3-1) + 4 * 4 + (5-5) * 5 * 4) * 8 = 144
2. ((1-1) + 2 * 4 + (7-5) * 5 * 4) * 8 = 384
3. ((4-1) + 3 * 4 + (9-5) * 5 * 4) * 8 = 760

6.5 节的练习

6.5.1

假定图 6-26 中的函数 widen 可以处理图 6-25a 的层次结构中的所有类型，翻译下列表达式。假定 c 和 d 是字符类型，s 和 t 是短整型， i 和 j 为整型， x 是浮点型。

1. x = s + c
2. i = s + c
3. x = (s + c) * (t + d)

解答

1. x = s + c

   ```
   t1 = (int) s
   t2 = (int) c
   t3 = t1 + t2
   x = (float) t3
   ```

2. i = s + c

   ```
   t1 = (int) s
   t2 = (int) c
   i = t1 + t2
   ```

3. x = (s + c) * (t + d)

   ```
   t1 = (int) s
   t2 = (int) c
   t3 = t1 + t2
   t4 = (int) t
   t5 = (int) d
   t6 = t4 + t5
   t7 = t3 + t6
   x = (float) t7
   ```

6.5.2

像 Ada 中那样，我们假设每个表达式必须具有唯一的类型，但是我们根据一个子表达式本身只能推导出一个可能类型的集合。也就是说，将函数 E1 应用于参数 E2（文法产生式为 E -> E1(E2)）有如下规则：

```
E.type = {t | 对 E2.type 中的某个 s, s -> t 在 E1.type 中}
```

描述一个可以确定每个字表达式的唯一类型的 SDD。它首先使用属性 type，按照自底向上的方式综合得到一个可能类型的集合。在确定了整个表达式的唯一类型之后，自顶向下地确定属性 unique 的值，整个属性表示各子表达式的类型。

6.6 节的练习

6.6.1

在图 6-36 的语法制导定义中添加处理下列控制流构造的规则：

1. 一个 repeat 语句：repeat S while B
2. ！ 一个 for 循环语句：for (S1; B; S2) S3

解答

```
Production                      Syntax Rule

S -> repeat S1 while B          S1.next = newlabel()
                                B.true = newlabel()
                                B.false = S.next
                                S.code = label(B.true) || S1.code
                                    || label(S1.next) || B.code

S -> for (S1; B; S2) S3         S1.next = newlabel()
                                B.true = newlabel()
                                B.false = S.next
                                S2.next = S1.next
                                S3.next = newlabel()
                                S.code = S1.code
                                    || lable(S1.next) || B.code
                                    || lable(B.true) || S3.code
                                    || label(S3.next) || S2.code
                                    || gen('goto', S1.next)
```

6.6.2

现代计算机试图在同一个时刻执行多条指令，其中包括各种分支指令。因此，当计算机投机性地预先执行某个分支，但实际控制流却进入另一个分支时，付出的代价是很大的。因此我们希望尽可能地减少分支数量。请注意，在图 6-35c 中 while 循环语句的实现中，每个迭代有两个分支：一个是从条件 B 进入到循环体中，另一个分支跳转回 B 的代码。基于尽量减少分支的考虑，我们通常更倾向于将 while(B) S 当作 if(B) {repeat S until !(B)} 来实现。给出这种翻译方法的代码布局，并修改图 6-

36 中 while 循环语句的规则。

解答

```
Production                   Syntax Rule

S -> if(B) {                 B.true = newlabel()
         repeat S1           B.false = S.next
         until !(B)          S1.next = newlabel()
     }                       S.code = B.code
                                 || label(B.true) || S1.code
                                 || label(S1.next) || B.code
```

6.6.3!

假设 C 中存在一个异或运算。按照图 6-37 的风格写出这个运算符的代码生成规则。

解答

B1 ^ B2 等价于 !B1 && B2 || B1 && !B2 (运算符优先级 ! > && > ||)

```
Production      Syntax Rule

B -> B1 ^ B2    B1.true = newlabel()
                B1.false = newlabel()

                B2.true = B.true
                B2.false = B1.true

                b3 = newboolean()
                b3.code = B1.code
                b3.true = newlabel()
                b3.false = B.false

                b4 = newboolean()
                b4.code = B2.code
                b4.true = B.false
                b4.false = B.true

                S.code = B1.code
                    || label(B1.false) || B2.code
                    || label(B1.true) || b3.code
```

```
                                  || label(b3.true) || b4.code
```

6.6.4

使用 6.6.5 节中介绍的避免 goto 语句的翻译方案，翻译下列表达式：

1. if (ab && cd || e==f) x == 1
2. if (ab || cd || e==f) x == 1
3. if (ab || cd && e==f) x == 1

解答

1. if (ab && cd || e==f) x == 1

   ```
   ifFalse a==b goto L3
   if c==d goto L2
   ```

   L3: ifFalse e==f goto L1
   L2: x == 1
   L1:

2. if (ab || cd || e==f) x == 1

   ```
   if a==b goto L2
   if c==d goto L2
   ifFalse e==f goto L1
   ```

   L2: x==1
   L1:

3. if (ab || cd && e==f) x == 1

   ```
   if a==b goto L2
   ifFalse c==d goto L1
   ifFalse e==f goto L1
   ```

   L2: x==1
   L1:

6.6.5

基于图 6-36 和图 6-37 中给出的语法制导定义，给出一个翻译方案。

6.6.6

使用类似于图 6-39 和图 6-40 中的规则，修改图 6-36 和图 6-37 的语义规则，使之允许控制流穿越。

解答

仅补充完毕书中未解答部分

```
Production              Syntax Rule

S -> if(B) S1 else S2   B.true = fall
                        B.false = newlabel()
                        S1.next = S.next
                        S2.next = S.next
                        S.code = B.code
                            || S1.code
                            || gen('goto' S1.next)
                            || label(B.false) || S2.code

S -> while(B) S1        begin = newlabel()
                        B.true = fall
                        B.false = S.next
                        S1.next = begin
                        S.code = label(begin) || B.code
                            || S1.code
                            || gen('goto' begin)

S -> S1 S2              S1.next = fall
                        S2.next = S.next
                        S.code = S1.code || S2.code

B -> B1 && B2           B1.true = fall
                        B1.false = if B.false == fall
                                   then newlabel()
                                   else B.false
                        B2.true = B.true
                        B2.false = B.false
                        B.code = if B.false == fall
                                 then B1.code || B2.code || label(B1.false)
                                 else B1.code || B2.code
```

6.6.7！

练习 6.6.6 中的语义规则产生了一些不必要的标号。修改图 6-36 中语句的规则，使之只创建必要的标号。你可以使用特殊符号 deferred 来表示还没有创建的一个标号。你的语义规则必须能生成类似于例 6.21 的代码。

6.6.8！！

6.6.5 节中讨论了如何使用穿越代码来尽可能减少生成的中间代码中跳转指令的数据。然而，它并没有充分考虑将一个条件替换为它的补的方法，例如将 `if a < b goto L1; goto L2` 替换成 `ifFalse a >= b goto L2; goto L1`。给出语法制导定义，它在需要时可以利用这种替换方法。

6.7 节的练习

6.7.1

使用图 6-43 中的翻译方案翻译下列表达式。给出每个子表达式的 truelist 和 falselist。你可以假设第一条被生成的指令的地址是 100.

1. ab && (cd || e==f)
2. (ab || cd) || e==f
3. (ab && cd) && e==f

解答

1. ab && (cd || e==f)

6.7.2

解答

1. E3.false = i1
2. S2.next = i7
3. E4.false = i7
4. S1.next = i3
5. E2.true = i3

6.7.3

当使用图 6-46 中的翻译方案对图 6-47 进行翻译时，我们为每条语句创建 S.next 列表。一开始是赋值语句 S1, S2, S3，然后逐步处理越来越大的 if 语句，if-else 语句，while 语句和语句块。在图 6-47 中有 5 个这种类型的结构语句：

- S4: while (E3) S1
- S5: if(E4) S2
- S6: 包含 S5 和 S3 的语句块
- S7: if(E2) S4 else S6
- S8: 整个程序

对于这些结构语句，我们可以通过一个规则用其他的 Sj.next 列表以及程序中的表达式的列表 Ek.true 和 Ek.false 构造出 Si.next。给出计算下列 next 列表的规则：

1. S4.next
2. S5.next
3. S6.next
4. S7.next
5. S8.next

解答

(该题解答不是很肯定)

1. S4.next = S3.next
2. S5.next = S2.next
3. S6.next = S3.next
4. S7.next = S3.next
5. S8.next = E1.false

Exercises for Section 7.2

7.2.1

Suppose that the program of Fig.7.2 uses a partition function that always picks a[m] as the separator v. Also, when the array a[m], ... , a[n] is reordered, assume that the order is preserved as much as possible. That is, first come all the elements less than v, in their original order, then all elements equal to v, and finally all elements greater than v, in their original order.

1. Draw the activation tree when the numbers 9,8,7,6,5,4,3,2,1 are sorted.
2. What is the largest number of activation records that ever appear together on the stack?

Answer

1. Draw the activation tree when the numbers 9,8,7,6,5,4,3,2,1 are sorted.

2. What is the largest number of activation records that ever appear together on the stack?

   9

7.2.2

Repeat Exercise 7.2.1 when the initial order of the numbers is 1,3,5,7,9,2,4,6,8.

7.2.3

In Fig. 7.9 is C code to compute Fibonacci numbers recursively. Suppose that the activation record for f includes the following elements in order: (return value, argument n, local s, local t); there will normally be other elements in the activation record as well. The questions below assume that the initial call is f(5).

```
int f(int n) {
    int t, s;
    if (n < 2) return 1;
    s = f(n-1);
    t = f(n-2);
    return s+t;
}

Figure 7.9: Fibonacci program for Exercise 7.2.3
```

1. Show the complete activation tree.
2. What dose the stack and its activation records look like the first time f(1) is about to return?
3. ! What does the stack and its activation records look like the fifth time f(1) is about to return?

Answer

1. Show the complete activation tree.

2. What dose the stack and its activation records look like the first time f(1) is about to return?

3. ! What does the stack and its activation records look like the fifth time f(1) is about to return?

7.2.4

Here is a sketch of two C functions f and g:

```
int f(int x){int i;...return i+1;...}
int g(int y) {int j;...f(j+1). ..}
```

That is, function g calls f. Draw the top of the stack, starting with the activation record for g, after g calls f, and f is about to return. You can consider only return values, parameters, control links, and space for local variables; you do not have to consider stored state or temporary or local values not shown in the code sketch. However, you should indicate:

1. Which function creates the space on the stack for each element?
2. Which function writes the value of each element?
3. To which activation record does the element belong?

Answer

7.2.5

In a language that passes parameters by reference, there is a function f(x, y) that does the following:

```
x = x + 1;
y = y + 2;
return x+y;
```

If a is assigned the value 3, and then f(a, a) is called, what is returned?

Answer

```
x = x + 1  ->  a = a + 1  ->  now a is 4
y = y + 2  ->  a = a + 2  ->  now a is 6
x + y  ->  a + a  ->  6 + 6  ->  12
```

f(a, a) is 12

7.2.6

The C function f is defined by:

```
int f(int x, *py, **ppz) {
    **ppz += 1;
    *py += 2;
    x += 3;
    return x+y+z;
}
```

Variable a is a pointer to b; variable b is a pointer to c, and c is an integer currently with value 4. If we call f(c, b, a) , what is returned?

Answer

f(c, b, a) is 21

view source code

mind that c is passed by value, so the process is:

```
sentence          x in f()    x out of f()   *py      **ppz

**ppz += 1;       4           5              5        5
*py += 2;         4           7              7        7
x += 3;           7           7              7        7
```

Exercises for Section 7.3

7.3.1

In Fig. 7.15 is a ML function main that computes Fibonacci numbers in a nonstandard way. Function fibO will compute the nth Fibonacci number for any n >= O. Nested within in is fib1, which computes the nth Fibonacci number on the assumption n >= 2, and nested within fib1 is fib2, which assumes n >= 4. Note that neither fib1 nor fib2 need to check for the basis cases. Show the stack of activation records that result from a call to main, up until the time that the first call (to fibO(1)) is about to return. Show the access link in each of the activation records on the stack.

```
fun main() {
    let
        fun fibO(n)
            let
                fun fib1(n) =
                    let
                        fun fib2(n) = fib1(n-l) + fib1(n-2)
                    in
                        if n >= 4 then fib2(n)
                        else fibO(n-l) + fibO(n-2)
                    end
            in
                if n >= 2 then fib1(n) else 1
            end
    in
        fibO(4)
    end ;
```

Figure 7.15: Nested functions computing Fibonacci numbers

Answer

activation tree:

activation stack when first call to fib0(1) is about to return:

7.3.2

Suppose that we implement the functions of Fig. 7.15 using a display. Show the display at the moment the first call to fibO(1) is about to return. Also, indicate the saved display entry in each of the activation records on the stack at that time.

Answer

Exercises for Section 7.4

7.4.1

Suppose the heap consists of seven chunks, starting at address 0. The sizes of the chunks, in order, are 80, 30, 60, 50, 70, 20, 40 bytes. When we place an object in a chunk, we put it at the high end if there is enough space remaining to form a smaller chunk (so that the smaller chunk can easily remain on the linked list of free space) . However , we cannot tolerate chunks of fewer that 8 bytes, so if an object is almost as large as the selected chunk, we give it the entire chunk and place the object at the low end of the chunk. If we request space for objects of the following sizes: 32, 64, 48, 16, in that order, what does the free space list look like after satisfying the requests, if the method of selecting chunks is

1. First fit.
2. Best fit.

Answer

values in parentheses are sizes actually in use

1. First fit.

   48, 32(32), 14, 16(16), 60, 50(48), 70(64), 20, 40

2. Best fit.

   80, 30, 60, 50(48), 70(64), 20(16), 8, 32(32)

Exercises for Section 7.5

7.5.1

What happens to the reference counts of the objects in Fig. 7.19 if:

1. The pointer from A to B is deleted.
2. The pointer from X to A is deleted.
3. The node C is deleted.

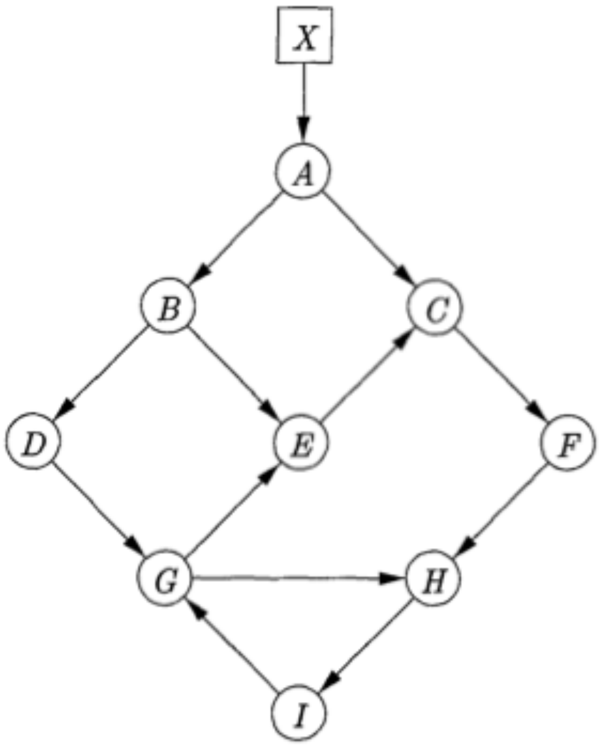
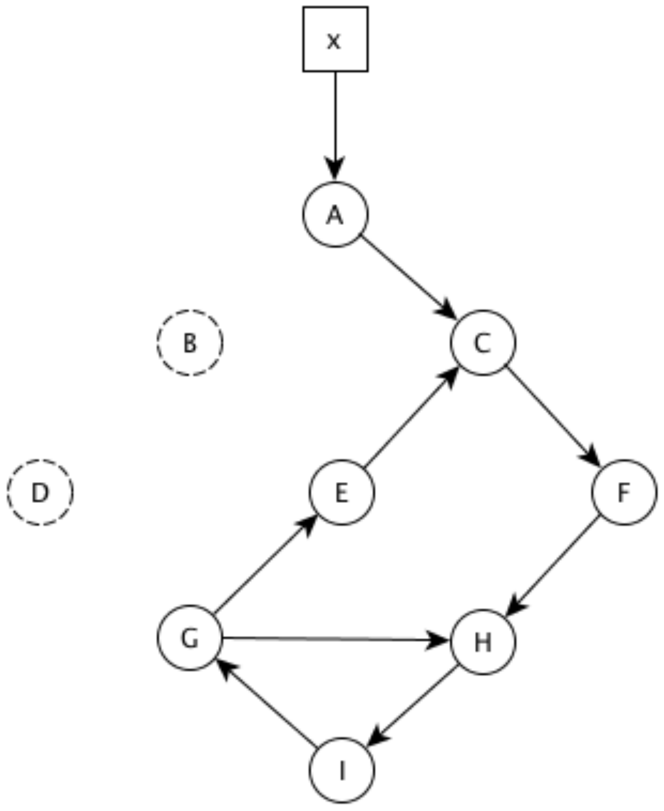


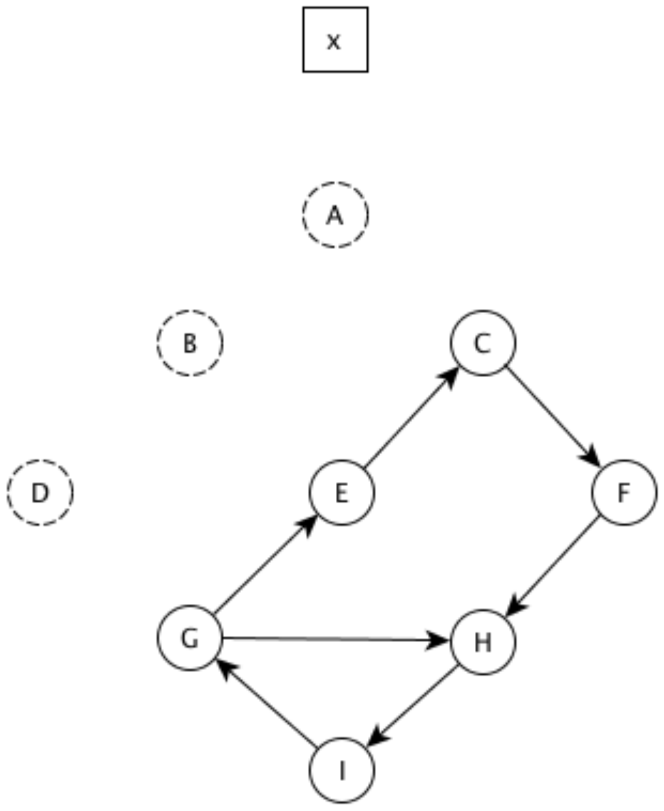
Figure 7.19: A network of objects

Answer

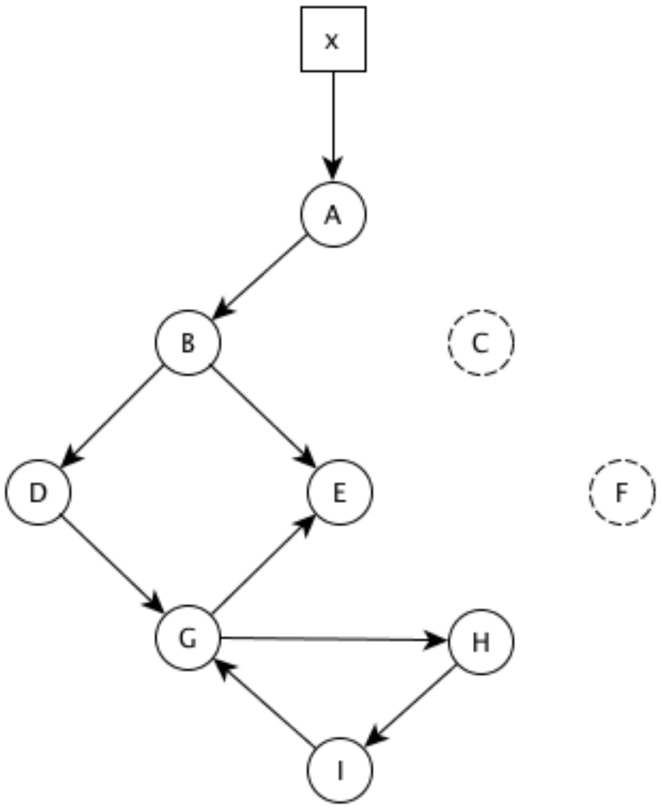
1. The pointer from A to B is deleted.



2. The pointer from X to A is deleted.



3. The node C is deleted.



7.5.2

What happens to reference counts when the pointer from A to D in Fig. 7.20 is deleted?

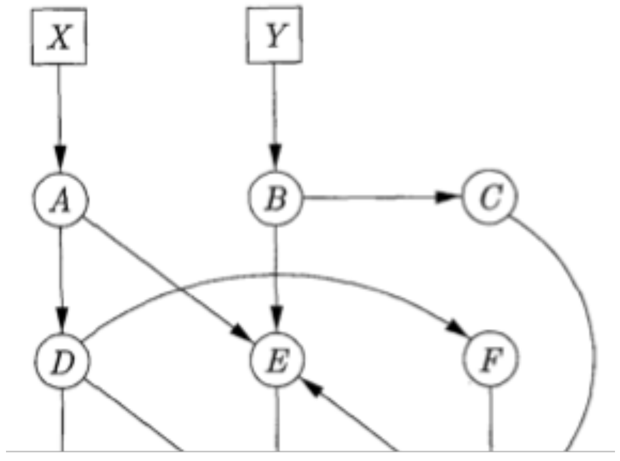
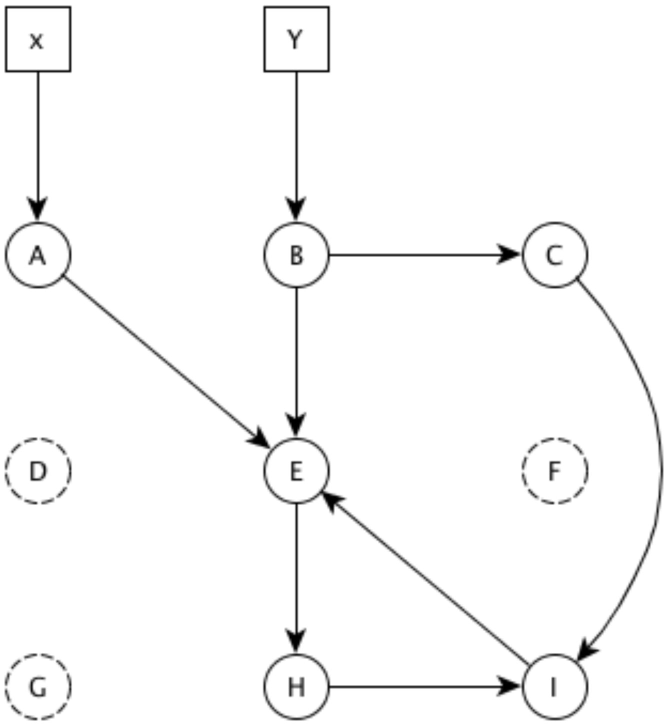


Figure 7.20: Another network of objects

Answer


x
Y
A
B
C
D
E
F
G
H
I

Exercises for Section 7.6

7.6.1

Show the steps of a mark-and-sweep garbage collector on

1. Fig. 7.19 with the pointer A to B deleted.
2. Fig. 7.19 with the pointer A to C deleted.
3. Fig. 7.20 with the pointer A to D deleted.
4. Fig. 7.20 with the object B deleted.

Answer

1. Fig. 7.19 with the pointer A to B deleted.

```
before:     A.reached =  … = I.reached = 0
            Unscanned = []
```

```
line1:      A.reached = 1
            Unscanned.push(A)

line2~7:

    loop1:  Unscanned.shift()
            C.reached = 1
            Unscanned.push( C )

    loop2:  Unscanned.shift()
            F.reached = 1
            Uncanned.push(F)

    loop3:  Unscanned.shift()
            H.reached = 1
            Uncanned.push(H)

    loop4:  Unscanned.shift()
            I.reached = 1
            Uncanned.push(I)

    loop5:  Unscanned.shift()
            G.reached = 1
            Uncanned.push(G)

    loop6:  Unscanned.shift()
            E.reached = 1
            Uncanned.push(E)

    loop7:  Unscanned.shift()
            // no more object add to list Unscanned
            // now it is empty, loop end

line8:      Free = []

line9~11:   Free = [B, D]
            A.reached = C.reached = E.reached = … =
I.reached = 0
```

7.6.2

The Baker mark-and-sweep algorithm moves objects among four lists: Free, Unreached, Unscanned, and Scanned. For each of the object networks of Exercise 7.6.1, indicate for each object the sequence of lists on which it finds itself from just before garbage collection begins until just after it finishes.

Answer

1. Fig. 7.19 with the pointer A to B deleted.

```
line1:      Free = [] // assume it is empty
            Unreached = [A, B, C, D, E, F, G, H, I]
            Unscanned = []
            Scanned = []

line2:      Unscanned = [A]
            Unreached = [B, C, D, E, F, G, H, I]

line3~7:

    loop1:  Scanned = [A]
            Unscanned = [C]
```

```
               Unreached = [B, D, E, F, G, H, I]

     loop2:    Scanned = [A, C]
               Unscanned = [F]
               Unreached = [B, D, E, G, H, I]

     loop3:    Scanned = [A, C, F]
               Unscanned = [H]
               Unreached = [B, D, E, G, I]

     loop4:    Scanned = [A, C, F, H]
               Unscanned = [I]
               Unreached = [B, D, E, G]

     loop5:    Scanned = [A, C, F, H, I]
               Unscanned = [G]
               Unreached = [B, D, E]

     loop6:    Scanned = [A, C, F, H, I, G]
               Unscanned = [E]
               Unreached = [B, D]

     loop7:    Scanned = [A, C, F, H, I, G, E]
               Unscanned = []
               Unreached = [B, D]

line8:         Free = [B, D]

line9:         Unreached = [A, C, F, H, I, G, E]
```

7.6.3

Suppose we perform a mark-and-compact garbage collection on each of the networks of Exercise 7.6.1. Also, suppose that

1. Each object has size 100 bytes, and
2. Initially, the nine objects in the heap are arranged in alphabetical order, starting at byte 0 of the heap.

What is the address of each object after garbage collection?

Answer

1. Fig. 7.19 with the pointer A to B deleted.

    ```
    A(0), C(100), E(200), F(300), G(400), H(500), I(600)
    ```

7.6.4

Suppose we execute Cheney's copying garbage collection algorithm on each of the networks of Exercise 7.6.1. Also, suppose that

1. Each object has size 100 bytes,
2. The unscanned list is managed as a queue, and when an object has more than one pointer, the reached objects are added to the queue in alpha betical order, and
3. The From semispace starts at location 0, and the To semispace starts at location 10,000.

What is the value of NewLocation(o) for each object o that remains after garbage collection?

Answer

1. Fig. 7.19 with the pointer A to B deleted.

    ```
    A(10000), C(10100), F(10200), H(10300), I(10400), G(10500),
    E(10600)
    ```

Exercises for Section 7.7

7.7.1

Suppose that the network of objects from Fig.7.20 is managed by an incremental algorithm that uses the four lists Unreached, Unscanned, Scanned, and Free, as in Baker's algorithm. To be specific, the Unscanned list is managed as a queue, and when more than one object is to be placed on this list due to the scanning of one object, we do so in alphabetical order. Suppose also that we use write barriers to assure that no reachable object is made garbage. Starting with A and B on the Unscanned list, suppose the following events occur:

1. A is scanned.
2. The pointer A -> D is rewritten to be A -> H.
3. B is scanned.
4. D is scanned.
5. The pointer B -> C is rewritten to be B -> I.

Simulate the entire incremental garbage collection, assuming no more pointers are rewritten. Which objects are garbage? Which objects are placed on the Free list?

Answer

0. init

```
Free = []
Unreached = [C, D, E, F, G, H, I]
Uscanned = [A, B]
Scanned = []
```

1. A is scanned.

```
Unreached = [C, F, G, H, I]
Uscanned = [B, D, E]
Scanned = [A]
```

2. The pointer A -> D is rewritten to be A -> H.

```
Unreached = [C, F, G, I]
Uscanned = [B, D, E, H]
Scanned = [A]
```

3. B is scanned.

```
Unreached = [F, G, I]
Uscanned = [D, E, H, C]
Scanned = [A, B]
```

4. D is scanned.

```
Unreached = [F, G, I]
Uscanned = [E, H, C]
Scanned = [A, B, D]
```

5. The pointer B -> C is rewritten to be B -> I.

```
![7 7 1-2]
(https://f.cloud.github.com/assets/340282/1313847/144a01e4-3263-
11e3-8037-b09e2c3b03f4.gif)

    Unreached = [F, G]
    Uscanned = [E, H, C, I]
    Scanned = [A, B, D]
```

7. E is scanned.

```
Unreached = [F, G]
Uscanned = [H, C, I]
Scanned = [A, B, D, E]
```

8. H is scanned.

```
Unreached = [F, G]
Uscanned = [C, I]
Scanned = [A, B, D, E, H]
```

9. C is scanned.

```
Unreached = [F, G]
Uscanned = [I]
Scanned = [A, B, D, E, H, C]
```

10. I is scanned.

```
Unreached = [F, G]
Uscanned = []
Scanned = [A, B, D, E, H, C, I]
```

11. end

```
Free = [F, G]
Unreached = [A, B, D, E, H, C, I]
Unscanned = []
Scanned = []
```

so, `[C, D, F, G]` is garbage, Free list is `[F, G]`.

7.7.2

Repeat Exercise 7.7.1 on the assumption that

1. Events (2) and (5) are interchanged in order.
2. Events (2) and (5) occur before (1), (3), and (4).

Answer

1. Events (2) and (5) are interchanged in order.

   omit

2. Events (2) and (5) occur before (1), (3), and (4).

   0. init

   ```
   Free = []
   Unreached = [C, D, E, F, G, H, I]
   Uscanned = [A, B]
   Scanned = []
   ```

   1. The pointer A -> D is rewritten to be A -> H.

```
Unreached = [C, D, E, F, G, I]
Uscanned = [A, B, H]
```

2. The pointer B -> C is rewritten to be B -> I.

```
    ![7 7 1-2]
(https://f.cloud.github.com/assets/340282/1313847/144a01e4-
3263-11e3-8037-b09e2c3b03f4.gif)

        Unreached = [C, D, E, F, G]
        Uscanned = [A, B, H, I]

3. A is scanned.

        Unreached = [C, D, F, G]
        Unscanned = [B, H, I, E]
        Scanned = [A]

4. B is scanned.

        Unreached = [C, D, F, G]
        Unscanned = [H, I, E]
        Scanned = [A, B]

5. H is scanned.

        Unreached = [C, D, F, G]
        Unscanned = [I, E]
        Scanned = [A, B, H]

5. I is scanned.

        Unreached = [C, D, F, G]
        Unscanned = [E]
        Scanned = [A, B, H, I]

5. E is scanned.

        Unreached = [C, D, F, G]
        Unscanned = []
        Scanned = [A, B, H, I, E]

```

```
6. end

        Free = [C, D, F, G]
        Unreached = [A, B, H, I, E]
        Unscanned = []
        Scanned = []

so, `[C, D, F, G]` is garbage, Free list also is `[C, D, F, G]`.
```

7.7.3

Suppose the heap consists of exactly the nine cars on three trains shown in Fig. 7.30 (i.e., ignore the ellipses). Object o in car 11 has references from cars 12, 23, and 32. When we garbage collect car 11, where might o wind up?

Answer

```
if any room in trains 2 and 3
    o can go in some existing car of either trains 2 and 3.
else
    o can go in a new, last car of either trains 2 and 3.
```

7.7.4

Repeat Exercise 7.7.3 for the cases that o has

1. Only references from cars 22 and 31.
2. No references other than from car 11.

Answer

1. Only references from cars 22 and 31.

   The same with Exercise 7.7.3.

2. No references other than from car 11.

```
if there is room in car 12
    o can go in car 12
else if there is room in other cars of train 1
    o can go in any car has room
else
    o can go in a new, last car of train 1
```

7.7.5

Suppose the heap consists of exactly the nine cars on three trains shown in Fig. 7.30 (i.e., ignore the ellipses). We are currently in panic mode. Object o1 in car 11 has only one reference, from object o2 in car 12. That reference is rewritten. When we garbage collect car 11, what could happen to o1?

Answer

It is not important which train we move it to, as long as it is not the first train?

Exercises for Section 8.2

8.2.1

Generate code for the following three-address statements assuming all variables are stored in memory locations.

1. x = 1
2. x = a
3. x = a + 1
4. x = a + b
5. The two statements
   - x = b * c
   - y = a + x

answer

```
1.  LD R1, #1
    ST x, R1

2.  LD R1, a
    ST x, R1

3.  LD R1, a
    ADD R1, R1, #1
    ST x, R1

4.  LD R1, a
    LD R2, b
    ADD R1, R1, R2
    ST x, R1

5.  LD R1, b
    LD R2, c
    MUL R1, R1, R2
    LD R3, a
    ADD R3, R3, R1
    ST y, R3
```

Note：第 5 小题，可以在生成的汇编码第三行后插入 `ST x, R1` 和 `LD R1, x` 两句，这两句属于冗余代码（redundant store-load）。使用简易代码生成策略很容易生成这种冗余代码，慢是慢一些但是也是正确的，有专门处理这种问题的优化（redundant store-load elimination），所以生不生成在这题的答案里感觉都行。

8.2.2

Generate code for the following three-address statements assuming a and b are arrays whose elements are 4-byte values.

1. The four-statement sequence

   ```
   x = a[i]
   y = b[j]
   a[i] = y
   b[j] = x
   ```

2. The three-statement sequence

   ```
   x = a[i]
   y = b[i]
   z = x * y
   ```

3. The three-statement sequence

   ```
   x = a[i]
   y = b[x]
   a[i] = y
   ```

answer

```
1.  LD R1, i
    MUL R1, R1, #4
    LD R2, a(R1)
    LD R3, j
    MUL R3, R3, #4
    LD R4, b(R3)
    ST a(R1), R4
```

```
    ST b(R3), R2

2.  LD R1, i
    MUL R1, R1, #4
    LD R2, a(R1)
    LD R1, b(R1)
    MUL R1, R2, R1
    ST z, R1

3.  LD R1, i
    MUL R1, R1, #4
    LD R2, a(R1)
    MUL R2, R2, #4
    LD R2, b(R2)
    ST a(R1), R2
```

8.2.3

Generate code for the following three-address sequence assuming that p and q are in memory locations:

```
y = *q
q = q + 4
*p = y
p = p + 4
```

answer

```
LD R1, q
LD R2, 0(R1)
ADD R1, R1, #4
ST q, R1
LD R1, p
ST 0(R1), R2
ADD R1, R1, #4
ST p, R1
```

8.2.4

Generate code for the following sequence assuming that x, y, and z are in memory locations:

```
    if x < y goto L1
    z = 0
    goto L2
L1: z = 1
```

answer

```
    LD R1, x
    LD R2, y
    SUB R1, R1, R2
    BLTZ R1, L1
    LD R1, #0
    ST z, R1
    BR L2
L1: LD R1, #1
    ST z, R1
```

Note：实际生成代码时会把标签对应到具体的数字地址上，但这小节还没到那一步，把原本题目里的标签名拿来随便写写就好啦。

8.2.5

Generate code for the following sequence assuming that n is in a memory location:

```
    s = 0
    i = 0
L1: if i > n goto L2
    s = s + i
    i = i + 1
    goto L1
L2:
```

answer

```
Long version:

    LD R1, #0
    ST s, R1
    ST i, R1
L1: LD R1, i
```

```
    LD R2, n
    SUB R2, R1, R2
    BGTZ R2, L2
    LD R2, s
    ADD R2, R2, R1
    ST s, R2
    ADD R1, R1, #1
    ST i, R1
    BR L1
L2:

Short version:

    LD R2, #0
    LD R1, R2
    LD R3, n
L1: SUB R4, R1, R3
    BGTZ R4, L2
    ADD R2, R2, R1
    ADD R1, R1, #1
    BR L1
L2:
```

Note：短版本的优化 1）消除冗余存-读 2）循环不变代码外提 3）然后外加寄存器分配

8.2.6

Determine the costs of the following instruction sequences:

```
1.  LD R0, y
    LD R1, z
    ADD R0, R0, R1
    ST x, R0

2.  LD R0, i
    MUL R0, R0, 8
    LD R1, a(R0)
    ST b, R1

3.  LD R0, c
    LD R1, i
    MUL R1, R1, 8
    ST a(R1),R0

4.  LD R0, p
    LD R1, 0(R0)
    ST x, R1

5.  LD R0, p
    LD R1, x
    ST 0(R0), R1

6.  LD R0, x
    LD R1, y
    SUB R0, R0, R1
    BLTZ *R3, R0
```

answer

1. 2 + 2 + 1 + 2 = 7
2. 2 + 2 + 2 + 2 = 8
3. 2 + 2 + 2 + 2 = 8
4. 2 + 2 + 2 = 6
5. 2 + 2 + 2 = 6
6. 2 + 2 + 1 + 1 = 6

Note：这本书用的指令集没明确定义所有指令的细节，但看起来所谓用变量名来指定内存地址实际上隐含着这些变量是静态分配的假设，也就是说在真正生成完的指令里这些变量名都会被替换为它们对应的数字形式的地址常量，而地址存在指令后的一个额外的word里，这就算多一单位的开销。

Note

1. 很明显本节内容写得非常随意，推荐数字常量是应该都加#前缀的，除了放在地址里用。比如 `LD R1, #1` 和 `ADD R1, R1, #1`。
2. 本书中 Ri 表示第 i 号寄存器。
   1. 在翻译成汇编码的过程中，是可以随意指定 i 的值（比如 R3, R4, R1000）呢还是会有某种限制？

      回答：现在暂时随意。等后面说寄存器个数有限制的时候再考虑有限制的情况。

2. 另外，如果代码中所示的 R1 在后面的代码中用不着了，那么新的值是不是可以被加载到 R1 中？如果可以的话，如何知道之前的 R1 用不着了？

   回答：可以覆盖。至于如何知道前面的值死了就要看 def-use 链。这是优化的重要问题。例如9.2.5小节讲 live variable 就跟这个有关。

3. b = a[i] 对应的汇编码：

   ```
   LD R1, i
   MUL R1, R1, 8
   LD R2, a(R1)
   ...
   ```

   其中 a 为什么不需要先 load 到寄存器?

   回答：这里隐含一个假设：变量是静态分配存储的。后面涉及不是静态变量的时候情况会有变化。

Exercises for Section 8.3

8.3.1

Generate code for the following three-address statements assuming stack allocation where register SP points to the top of the stack.

```
call p
call q
return
call r
return
return
```

Answer

```
100:  LD SP, #stackStart
108:  ADD SP, SP, #psize
116:  ST *SP, #132
124:  BR pStart
132:  SUB SP, SP, #psize
140:  ADD SP, SP, #qsize
148:  ST *SP, #164
156:  BR qStart
164:  SUB SP, SP, #qsize
172:  BR **SP
```

8.3.2

Generate code for the following three-address statements assuming stack allocation where register SP points to the top of the stack.

1. x = 1
2. x=a
3. x = a + 1
4. x = a+b

5. The two statements
    - x = b * c
    - y = a + x

8.3.3

Generate code for the following three-address statements again assuming stack
allocation and assuming a and b are arrays whose elements are 4-byte values.

1. The four-statement sequence

    ```
    x = a[i]
    y = b[j]
    a[i] = y
    b[j] = x
    ```

2. The three-statement sequence

    ```
    x = a[i]
    y = b[i]
    z = x * y
    ```

3. The three-statement sequence

    ```
    x = a[i]
    y = b[x]
    a[i] = y
    ```

Note

1. 指令长度

```
120:  ST 364, #140
132:  BR 200
140:  ACTION2
```

图 8-4 部分代码

- 每行指令前面的标号代表了这行代码的起始位置（即偏移量），和下一行指令的标号差代表这行指令的长度。
- 第一行有 1 个指令和 2 个常量，所以指令长度是 12，同理第二行有 1 个指令和 1 个常量，所以长度为 8.

```
100:  LD, SP, #600
108:  ACTION1
128:  ADD SP, SP, #msize
136:  ST *SP, #152
```

图 8-6 部分代码

- 由于 SP 不占空间，所以上图中的几行指令长度均为 8。

Exercises for Section 8.4

8.4.1

Figure 8.10 is a simple matrix-multiplication program.

1. Translate the program into three-address statements of the type we have been using in this section. Assume the matrix entries are numbers that require 8 bytes, and that matrices are stored in row-major order.
2. Construct the flow graph for your code from (a).
3. Identify the loops in your flow graph from (b).

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0.0;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

Figure 8.10: A matrix-multiplication algorithm

Answer

1. three-address statements

```
B1        1)  i = 0

B2        2)  if i >= n goto(13)

B3        3)  j = 0

B4        4)  if j >= n goto(11)

B5        5)  t1 = n * i
          6)  t2 = t1 + j
          7)  t3 = t2 * 8
          8)  c[t3] = 0.0
          9)  j = j + 1
         10)  goto(4)

B6       11)  i = i + 1
         12)  goto(2)

B7       13)  i = 0

B8       14)  if i >= n goto(40)

B9       15)  j = 0

B10      16)  if j >= n goto(38)

B11      17)  k = 0

B12      18)  if k >= n goto(36)

B13      19)  t4 = n * i
         20)  t5 = t4 + j
         21)  t6 = t5 * 8
         22)  t7 = c[t6]
         23)  t8 = n * i
         24)  t9 = t8 + k
         25)  t10 = t9 * 8
         26)  t11 = a[t10]
         27)  t12 = n * k
         28)  t13 = t12 + j
         29)  t14 = t13 * 8
         30)  t15 = b[t14]
         31)  t16 = t11 * t15
         32)  t17 = t7 + t16
         33)  c[t6] = t17
         34)  k = k + 1
         35)  goto(18)

B14      36)  j = j + 1
         37)  goto(16)

B15      38)  i = i + 1
         39)  goto(14)
```

2. flow graph

3. loops

    - {B2, B3, B4, B6}
    - {B4, B5}
    - {B8, B9, B10, B15}
    - {B10, B11, B12, B14}
    - {B12, B13}

8.4.2

Figure 8.11 is code to count the number of primes from 2 to n, using the sieve method on a suitably large array a. That is, a[i] is TRUE at the end only if there is no prime i^0.5 or less that evenly divides i. We initialize all a[i] to TRUE and then set a[j] to FALSE if we find a divisor of j.

1. Translate the program into three-address statements of the type we have been using in this section. Assume integers require 4 bytes.
2. Construct the flow graph for your code from (a).
3. Identify the loops in your flow graph from (b).

```
for (i=2; i<=n; i++)
    a[i] = TRUE;
count = 0;
s = sqrt(n);
for (i=2; i<=s; i++)
if (a[i]) 1* i has been found to be a prime *1 {
    count++ ;
    for (j=2*i; j<=n; j = j+i)
        a[j] = FALSE; 1* no multiple of i is a prime *1
    }
```

Figure 8.11: Code to sieve for primes

Answer

1. three-address statements

```
B1       1)  i = 2

B2       2)  if i > n goto(7)
```

```

B3        3)  t1 = i * 4
          4)  a[t1] = TRUE
          5)  i = i + 1
          6)  goto(2)

B4        7)  count = 0
          8)  s = sqrt(n)
          9)  i = 2

B5       10)  if i > s goto(22)

B6       11)  t2 = i * 4
         12)  ifFalse a[t2] goto(20)

B7       13)  count = count + 1
         14)  j = 2 * i

B8       15)  if j > n goto(20)

B9       16)  t3 = j * 4
         17)  a[t3] = FALSE
         18)  j = j + i
         19)  goto(15)

B10      20)  i = i + 1
         21)  goto(10)
```

2. flow graph
3. loops
   - {B2, B3}
   - {B5, B6, B10}
   - {B5, B6, B7, B8, B10}
   - {B8, B9}

Note

1. A demo for algorithm 8.7: Determining the liveness and next-use information foreach statement in a basic block.

```
init:

three-address statements                   symbol table

                                            symbol  live   nextuse
    i)  a = b + c                          [a,      true,  null]
    j)  t = a + b                          [b,      true,  null]
                                           [c,      true,  null]
                                           [t,      true,  null]

step1:
Attach to statement j the information currently found in the
symbol table

                                            symbol  live   nextuse
    i)  a = b + c                          [a,      true,  null]
    j)  t = a + b  [t, true, null]         [b,      true,  null]
                   [a, true, null]         [c,      true,  null]
                   [b, true, null]         [t,      true,  null]

step2:
In the symbol table, set x.live = false and
                         x.nextuse = null

                                            symbol  live   nextuse
    i)  a = b + c                          [a,      true,  null]
    j)  t = a + b  [t, true, null]         [b,      true,  null]
                   [a, true, null]         [c,      true,  null]
                   [b, true, null]         [t,      false, null]

step3:
In the symbol table, set a.live = true, b.live = true and
                         a.nextuse = j, b.nextuse = j

                                            symbol  live   nextuse
    i)  a = b + c                          [a,      true,  j   ]
    j)  t = a + b  [t, true, null]         [b,      true,  j   ]
                   [a, true, null]         [c,      true,  null]
                   [b, true, null]         [t,      false, null]

step4:

```

```

                                                     symbol  live   nextuse
     i)  a = b + c  [a, true, j   ]      [a,      true,  j   ]
                    [b, true, j   ]      [b,      true,  j   ]
                    [c, true, null]      [c,      true,  null]
                                         [t,      false, null]
     j)  t = a + b  [t, true, null]
                    [a, true, null]
                    [b, true, null]

step5:

                                                     symbol  live   nextuse
     i)  a = b + c  [a, true, j   ]      [a,      false, null]
                    [b, true, j   ]      [b,      true,  j   ]
                    [c, true, null]      [c,      true,  null]
                                         [t,      false, null]
     j)  t = a + b  [t, true, null]
                    [a, true, null]
                    [b, true, null]

step6:

                                                     symbol  live   nextuse
     i)  a = b + c  [a, true, j   ]      [a,      false, null]
                    [b, true, j   ]      [b,      true,  i   ]
                    [c, true, null]      [c,      true,  i   ]
                                         [t,      false, null]
     j)  t = a + b  [t, true, null]
                    [a, true, null]
                    [b, true, null]

```

2. Three ways to generate code for "for(i = 0; i < n ; i++)" statement

```
 1) i = 0
 2) if i >= n goto(9)
 3)
    ...
 7) i = i + 1
 8) if i < n goto(3)
```

```
9)


1) i = 0
2) goto(8)
3)
   ...
7) i = i + 1
8) if i < n goto(3)
9)


1) i= 0
2) if i >= n goto(9)
   ...
7) i = i + 1
8) goto(2)
9)
```

更多可参考 RednaxelaFX 的 对C语义的for循环的基本代码生成模式

Exercises for Section 8.5

8.5.1

Construct the DAG for the basic block

```
d = b * c
e = a + b
b = b * c
a = e - d
```

Answer

8.5.2

Simplify the three-address code of Exercise 8.5.1, assuming

1. Only a is live on exit from the block.
2. a, b, and c are live on exit from the block.

Answer

1. Only a is live on exit from the block.

   ```
   e = a + b
   d = b * c
   a = e - d
   ```

2. a, b, and c are live on exit from the block.

   ```
   e = a + b
   b = b * c
   a = e - b
   ```

8.5.3

Construct the basic block for the code in block B6 of Fig. 8.9. Do not forget to include the comparison i <= 10.

Answer

疑问

- “Construct the basic block” 被翻译成 “构造 DAG”，是这个意思吗?
- 如何为一个 “if goto” 语句 construct the basic block?

8.5.4

Construct the DAG for the code in block B3 of Fig. 8.9.

Answer

8.5.5

Extend Algorithm 8.7 to process three-statements of the form

1. a[i] = b
2. a = b[i]
3. a = *b
4. *a = b

8.5.6

Construct the DAG for the basic block

```
a[i] = b
*p = c
d = a[j]
e = *p
*p = a[i]
```

on the assumption that

1. p can point anywhere.
2. p can point only to b or d.

疑问

8.5.6 节讲指针赋值这里又没有 demo 啊！！！

- `*p = c` 和 `c = *p`翻译成 DAG 是不是这样的：
- `*p = a[i]` 这样的语句用 DAG 如何表示?
- 8.5.6 节讲到：the operator =* must take all nodes that are currently associated with identifiers as arguments。这句话再 DAG 中如何表示?

8.5.7 !

If a pointer or array expression, such as a[i] or *p is assigned and then used, without the possibility of being changed in the interim, we can take advantage of the situation to simplify the DAG. For example, in the code of Exercise 8.5.6, since p is not assigned between the second and fourth statements,the statement e = *p can be replaced by e = c, regardless of what p points to. Revise the DAG-construction algorithm to take advantage of such situations, and apply your algorithm to the code of Example 8.5.6.

8.5.8

Suppose a basic block is formed from the C assignment statements

```
x = a + b + c + d + e + f;
y = a + c + e;
```

1. Give the three-address statements (only one addition per statement) for this block.
2. Use the associative and commutative laws to modify the block to use the fewest possible number of instructions, assuming both x and y are live on exit from the block.

Answer

1. three-address statements

```
t1 = a + b
t2 = t1 + c
t3 = t2 + d
t4 = t3 + e
t5 = t4 + f
x = t5
t6 = a + c
t7 = c + e
y = t6 + t7
```

2. optimized statments

```
t1 = a + c
t2 = t1 + e
y = t2
t3 = t2 + b
t4 = t3 + d
t5 = t4 + f
x = t5
```