第1章　プログラミング言語

問題 1 の回答例　　たとえば C プログラム：

```
#include <stdio.h>
int main(void){
    int i;
    float sum = 0;
    for(i = 1; i <= 10; i++)
            sum += i;
    printf("%f\n",sum);
}
```

を格納したファイルを test.c とする．このファイルについて

```
    > gcc -S test.c
```

などと実行すると test.s という名称のファイルが生成される．Intel プロセッサ版 Macintosh OSX ver.10.5 の gcc (ver.4.0.1) でコンパイルした場合の test.s は以下の通りである．

```
        .cstring
LC1:
        .ascii "%f\12\0"
        .literal4
        .align 2
LC0:
        .long        0
        .text
.globl _main
_main:
        pushl        %ebp
        movl        %esp, %ebp
        pushl        %ebx
        subl        $36, %esp
        call        L6
"L00000000001$pb":
L6:
        popl        %ebx
        leal        LC0-"L00000000001$pb"(%ebx), %eax
        movl        (%eax), %eax
        movl        %eax, -12(%ebp)
        movl        $1, -16(%ebp)
        jmp        L2
L3:
        cvtsi2ss        -16(%ebp), %xmm1
        movss        -12(%ebp), %xmm0
        addss        %xmm1, %xmm0
```

```
        movss        %xmm0, -12(%ebp)
        leal        -16(%ebp), %eax
        incl        (%eax)
L2:
        cmpl        $10, -16(%ebp)
        jle        L3
        cvtss2sd        -12(%ebp), %xmm0
        movsd        %xmm0, 4(%esp)
        leal        LC1-"L00000000001$pb"(%ebx), %eax
        movl        %eax, (%esp)
        call        L_printf$stub
        addl        $36, %esp
        popl        %ebx
        leave
        ret
        .section __IMPORT,__jump_table,symbol_stubs,self_modifying_code+pure_instructions,5
L_printf$stub:
        .indirect_symbol _printf
        hlt ; hlt ; hlt ; hlt ; hlt
        .subsections_via_symbols
```

このコードの詳細を説明することはしないが，コードの構造は，コード中のラベルと分岐命令を調べるだけでもおおまかに理解できる．その部分を抜き出すと以下の通り．後ろにコメントを付けた．

```
        ...
_main:                      // main 関数の開始位置
        ...
        jmp L2              // ループの条件判定部への分岐
L3:                         // ループ処理の先頭
        ...
L2:                         // ループの条件判定部開始位置
        ...
        jle        L3       // ループ本体への分岐
        ...
        call        L_printf$stub // printf の呼び出し
        ...
        ret                 // main 関数からの return
        ...
```

コード中の個々の命令，指示子（ディレクティブ）の詳しい意味は自身で調べよ（Intel のホームページから IA32 プロセッサのドキュメント（特に instruction set manual）を取得し，命令の意味を調べよ）．

問題 2 の回答例　問題 1 の回答例と同じプログラムを PowerPC プロセッサ版 Macintosh OSX ver.10.4 の gcc (ver.4.0.0) でコンパイルした場合のコードは以下の通り．もちろんアセンブリ命令自体は問題 1 の回答例と異なるが，ラベル L2，L3 の周辺のコードの構造は問題 1 の回答例と同じである．というのも，問題 1 と問題 2 のコンパイラはほとんど同じヴァージョンの gcc であり，同じ戦略の高水準最適化が施されるからである．個々の命令，指示子（ディレクティブ）の詳しい意味は自身で調べよ（IBM のホームページから PowerPC のドキュメント（特に instruction set manual）を取得し，命令の意味を調べよ）．

```
        .section __TEXT,__text,regular,pure_instructions
        .section __TEXT,__picsymbolstub1,symbol_stubs,pure...
        .machine ppc
        .cstring
        .align 2
LC0:
        .ascii "%f\12\0"
        .literal8
        .align 3
LC1:
        .long   1127219200
        .long   -2147483648
        .text
        .align 2
        .globl _main
_main:
        mflr r0
        stmw r30,-8(r1)
        stw r0,8(r1)
        stwu r1,-112(r1)
        mr r30,r1
        bcl 20,31,"L00000000001$pb"
"L00000000001$pb":
        mflr r31
        li r0,0
        stw r0,56(r30)
        li r0,1
        stw r0,60(r30)
        b L2
L3:
        lwz r0,60(r30)
        lis r2,0x4330
        addis r9,r31,ha16(LC1-"L00000000001$pb")
        lfd f13,lo16(LC1-"L00000000001$pb")(r9)
        xoris r0,r0,0x8000
        stw r0,76(r30)
        stw r2,72(r30)
        lfd f0,72(r30)
        fsub f0,f0,f13
        frsp f13,f0
        lfs f0,56(r30)
        fadds f0,f0,f13
        stfs f0,56(r30)
        lwz r2,60(r30)
        addi r0,r2,1
        stw r0,60(r30)
L2:
        lwz r0,60(r30)
        cmpwi cr7,r0,10
        ble cr7,L3
        lfs f0,56(r30)
        addis r2,r31,ha16(LC0-"L00000000001$pb")
        la r3,lo16(LC0-"L00000000001$pb")(r2)
```

```
        fmr f13,f0
        stfd f13,80(r30)
        lwz r9,80(r30)
        lwz r10,84(r30)
        mr r4,r9
        mr r5,r10
        fmr f1,f0
        bl "L_printf$LDBLStub$stub"
        lwz r1,0(r1)
        lwz r0,8(r1)
        mtlr r0
        lmw r30,-8(r1)
        blr
        .section __TEXT,__picsymbolstub1,symbol_stubs,pure_instructions,32
        .align 5
"L_printf$LDBLStub$stub":
        .indirect_symbol _printf$LDBLStub
        mflr r0
        bcl 20,31,"L00000000001$spb"
"L00000000001$spb":
        mflr r11
        addis r11,r11,ha16(L_printf$LDBLStub$lazy_ptr-...
        mtlr r0
        lwzu r12,lo16(L_printf$LDBLStub$lazy_ptr-"L000...
        mtctr r12
        bctr
        .lazy_symbol_pointer
L_printf$LDBLStub$lazy_ptr:
        .indirect_symbol _printf$LDBLStub
        .long   dyld_stub_binding_helper
        .subsections_via_symbols
```

問題 3 の回答例　Java プログラムを Java コンパイラ・コマンド javac でコンパイルし，得られた Test.class を hexdump -C でダンプした結果を以下に示す.

```
> hexdump -C Test.class
00000000  ca fe ba be 00 00 00 31  00 1b 0a 00 05 00 0e 09  |朋詐...1........|
00000010  00 0f 00 10 0a 00 11 00  12 07 00 13 07 00 14 01  |................|
00000020  00 06 3c 69 6e 69 74 3e  01 00 03 28 29 56 01 00  |..<init>...()V..|
00000030  04 43 6f 64 65 01 00 0f  4c 69 6e 65 4e 75 6d 62  |.Code...LineNumb|
00000040  65 72 54 61 62 6c 65 01  00 04 6d 61 69 6e 01 00  |erTable...main..|
00000050  16 28 5b 4c 6a 61 76 61  2f 6c 61 6e 67 2f 53 74  |.([Ljava/lang/St|
00000060  72 69 6e 67 3b 29 56 01  00 0a 53 6f 75 72 63 65  |ring;)V...Source|
00000070  46 69 6c 65 01 00 09 74  65 73 74 2e 6a 61 76 61  |File...test.java|
00000080  0c 00 06 00 07 07 00 15  0c 00 16 00 17 07 00 18  |................|
00000090  0c 00 19 00 1a 01 00 04  54 65 73 74 01 00 10 6a  |........Test...j|
000000a0  61 76 61 2f 6c 61 6e 67  2f 4f 62 6a 65 63 74 01  |ava/lang/Object.|
000000b0  00 10 6a 61 76 61 2f 6c  61 6e 67 2f 53 79 73 74  |..java/lang/Syst|
000000c0  65 6d 01 00 03 6f 75 74  01 00 15 4c 6a 61 76 61  |em...out...Ljava|
000000d0  2f 69 6f 2f 50 72 69 6e  74 53 74 72 65 61 6d 3b  |/io/PrintStream;|
000000e0  01 00 13 6a 61 76 61 2f  69 6f 2f 50 72 69 6e 74  |...java/io/Print|
000000f0  53 74 72 65 61 6d 01 00  07 70 72 69 6e 74 6c 6e  |Stream...println|
00000100  01 00 04 28 46 29 56 00  20 00 04 00 05 00 00 00  |...(F)V. .......|
```

```
00000110  00 00 02 00 00 00 06 00  07 00 01 00 08 00 00 00  |................|
00000120  1d 00 01 00 01 00 00 00  05 2a b7 00 01 b1 00 00  |.........*..=..|
00000130  00 01 00 09 00 00 00 06  00 01 00 00 00 01 00 09  |................|
00000140  00 0a 00 0b 00 01 00 08  00 00 00 49 00 02 00 03  |...........I....|
00000150  00 00 00 1d 0b 44 04 3d  1c 10 0a a3 00 0e 23 1c  |.....D.=... =..#.|
00000160  86 62 44 84 02 01 a7 ff  f2 b2 00 02 23 b6 00 03  |.bD... ==魏..#=..|
00000170  b1 00 00 00 01 00 09 00  00 00 1a 00 06 00 00 00  |=...............|
00000180  03 00 02 00 04 00 0a 00  05 00 0f 00 04 00 15 00  |................|
00000190  06 00 1c 00 07 00 01 00  0c 00 00 00 02 00 0d     |...............|
0000019f
```

ここに先頭の 4 バイトの 16 進数 cafe babe はこのファイルが Java クラスファイルであることを示すシグニチャである．そのシグニチャの後にバイトコードにコンパイルされたプログラム情報が続くが，その内容は javap -v コマンド（または javap -c コマンド）で見る方が分かり易い．以下はそのコマンドの実行結果である．

```
> javap -v Test
Compiled from "test.java"
class Test extends java.lang.Object
  SourceFile: "test.java"
  minor version: 0
  major version: 49
  Constant pool:
const #1 = Method        #5.#14;        //  java/lang/Object."<init>":()V
const #2 = Field        #15.#16;        //  java/lang/System.out:Ljava/io/PrintStream;
const #3 = Method        #17.#18;        //  java/io/PrintStream.println:(F)V
const #4 = class        #19;        //  Test
const #5 = class        #20;        //  java/lang/Object
const #6 = Asciz        <init>;
const #7 = Asciz        ()V;
const #8 = Asciz        Code;
const #9 = Asciz        LineNumberTable;
const #10 = Asciz        main;
const #11 = Asciz        ([Ljava/lang/String;)V;
const #12 = Asciz        SourceFile;
const #13 = Asciz        test.java;
const #14 = NameAndType        #6:#7;//  "<init>":()V
const #15 = class        #21;        //  java/lang/System
const #16 = NameAndType        #22:#23;//  out:Ljava/io/PrintStream;
const #17 = class        #24;        //  java/io/PrintStream
const #18 = NameAndType        #25:#26;//  println:(F)V
const #19 = Asciz        Test;
const #20 = Asciz        java/lang/Object;
const #21 = Asciz        java/lang/System;
const #22 = Asciz        out;
const #23 = Asciz        Ljava/io/PrintStream;;
const #24 = Asciz        java/io/PrintStream;
const #25 = Asciz        println;
const #26 = Asciz        (F)V;

{
Test();
  Code:
```

```
   Stack=1, Locals=1, Args_size=1
   0:        aload_0
   1:        invokespecial        #1; //Method java/lang/Object."<init>":()V
   4:        return
  LineNumberTable:
   line 1: 0

public static void main(java.lang.String[]);
  Code:
   Stack=2, Locals=3, Args_size=1
   0:        fconst_0
   1:        fstore_1
   2:        iconst_1
   3:        istore_2
   4:        iload_2
   5:        bipush        10
   7:        if_icmpgt        21
   10:        fload_1
   11:        iload_2
   12:        i2f
   13:        fadd
   14:        fstore_1
   15:        iinc        2, 1
   18:        goto        4
   21:        getstatic        #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   24:        fload_1
   25:        invokevirtual        #3; //Method java/io/PrintStream.println:(F)V
   28:        return
  LineNumberTable:
   line 3: 0
   line 4: 2
   line 5: 10
   line 4: 15
   line 6: 21
   line 7: 28

}
```

ファイル名，ヴァージョン情報，定数リスト等が出力され，最後にコンパイルして生成されたバイトコードの本体が出力されている．Java インタプリタはこのコードを実行する．なお，バイトコードのニーモニック表を調べると（詳細は各自で資料を調べよ），`fconst_0` 命令の 16 進表現は 0b，`fstore_1` のそれは 44 である．このことから，`main` 関数の先頭の 2 命令：

```
   0:        fconst_0
   1:        fstore_1
```

は，hexdump の出力の

```
00000150  00 00 00 1d 0b 44 04 3d  1c 10 0a a3 00 0e 23 1c
```

の行の 5，6 バイト目に相当することが分かる．

問題 4 の回答例　たとえば Gnu Common Lisp はフリーの Lisp インタプリタであり，様々な環境で動作する版が簡単にダウンロードできる．必要とされるハードディス容量も小さい．ソフトウェアをインストールして使ってみよ．

第2章　コンパイラの全体構成

問題 1 の回答　たとえば，1 章の章末問題 1 の回答のアセンブリコード（以下，オプションなしコード）を -O1，-O2，-O3 の最適化オプションでそれぞれコンパイルし，生成されたアセンブリコードを比較する．コンパイラは 1 章の章末問題 1 と同じとする．

まず，-O1 コードは以下の通りである．

```
        .cstring
LC1:
        .ascii "%f\12\0"
        .text
.globl _main
_main:
        pushl        %ebp
        movl        %esp, %ebp
        pushl        %ebx
        subl        $20, %esp
        call        L7
"L00000000001$pb":
L7:
        popl        %ebx
        xorps        %xmm1, %xmm1
        movl        $1, %eax
L2:
        cvtsi2ss        %eax, %xmm0
        addss        %xmm0, %xmm1
        incl        %eax
        cmpl        $11, %eax
        jne        L2
        cvtss2sd        %xmm1, %xmm1
        movsd        %xmm1, 4(%esp)
        leal        LC1-"L00000000001$pb"(%ebx), %eax
        movl        %eax, (%esp)
        call        L_printf$stub
        addl        $20, %esp
        popl        %ebx
        leave
        ret
        .section __IMPORT,__jump_table,symbol_stubs,self_modifying_code+pure_instructions,5
L_printf$stub:
        .indirect_symbol _printf
        hlt ; hlt ; hlt ; hlt ; hlt
        .subsections_via_symbols
```

このコードの構造は以下の通りであり，ループ処理の構造がオプションなしコード（前章，章末問題 1 参照）よりも簡素化している．

```
        ...
_main:                      // main 関数の開始位置
        ...
L2:                         // ループ処理の先頭
        ...
        jne         L2              // ループバックジャンプ
        ...
        call         L_printf$stub // printf の呼び出し
        ...
        ret                 // main 関数からの return
        ...
```

次に，-O2 オプションのコードのループ部分は以下の通りであった．

```
L2:
        cvtsi2ss    %eax, %xmm0
        incl        %eax
        cmpl        $11, %eax
        addss       %xmm0, %xmm1
        jne         L2
```

ループには -O1 オプションのコードと同じく 5 個の命令が含まれているが，-O2 オプションのコードでは -O1 オプションのコードと命令の並びが異なっている．このことから，-O2 オプションでは何らかの命令スケジューリングが実施されていると推定される．

次に，-O3 オプションのコードは -O2 オプションのコードと全く同じものであった．例題プログラムが簡単すぎて，より高度な最適化は効果がなかったと思われる．

問題 2 の回答　GNU gcc のサイトは http://gcc.gnu.org/ である．ここから最新ヴァージョンの gcc のソースファイル（.tar.gz ファイル）をダウンロードし，展開せよ．展開して得られたディレクトリ（たぶん gcc-*.*.* という名称である，'*' にはヴァージョン番号が入る）の下に多くのディレクトリを発見するだろう．README ファイルを参考にディレクトリの中を探索せよ．特に，gcc という名称のディレクトリへ入ると，そこに gcc コンパイラのソースプログラム・ファイル（C プログラム）が多数置かれている．ファイル名からその機能が予測できるものもある．ファイルの中を覗いてみよ．また各ファイルの大きさを調査してみるのも面白いだろう．

第3章　字句解析

問題 **1** の回答　“2..2” を字句解析するときの様子は以下の通り.

状態	LF	入力列の状態	注釈
1	0	$^{\vee}_{\wedge}$2 . . 2	
14	14	2$^{\vee}_{\wedge}$. . 2	
13	13	2 .$^{\vee}_{\wedge}$. 2	
0	13	2 .$^{\vee}$.$_{\wedge}$2	“2.” を REAL と認識
1	0	$^{\vee}_{\wedge}$. 2	
15	0	.$^{\vee}_{\wedge}$2	
13	13	. 2$^{\vee}_{\wedge}$	ファイルの終了位置に到達
0	13	. 2$^{\vee}_{\wedge}$	“.2” を REAL と認識

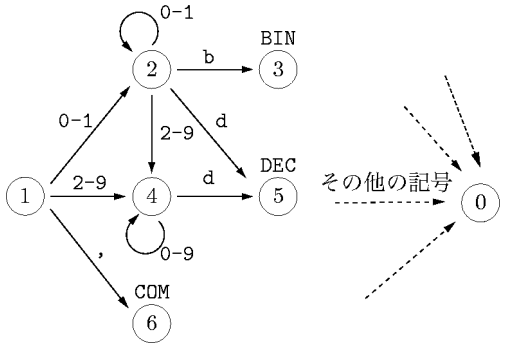
さらに必要ならば，この後，EoF トークンを構文解析器へ受け渡す.

“..2” を字句解析するときの様子は以下の通り.

状態	LF	入力列の状態	注釈
1	0	$^{\vee}_{\wedge}$. . 2	
15	0	.$^{\vee}_{\wedge}$. 2	
0	0	.$^{\vee}$.$_{\wedge}$2	字句解析エラー

問題 **2** の回答

1. 以下の図の通り．ただし，最小化されていない決定性オートマトンの形は唯一ではなく，またここでは最小化を要求しないから，以下の図は一例に過ぎない.



2. 以下のような動作になる.

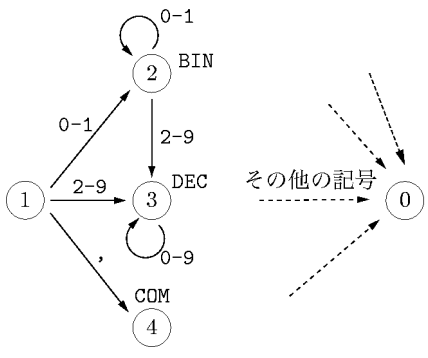
状態	LF	入力列の状態	注釈
1	0	$^{\vee}_{\wedge}$1 0 b , 2 3 d	
2	0	$^{\vee}$1$_{\wedge}$0 b , 2 3 d	
2	0	$^{\vee}$1 0$_{\wedge}$b , 2 3 d	
3	3	1 0 b$^{\vee}_{\wedge}$, 2 3 d	
0	3	1 0 b$^{\vee}$,$_{\wedge}$2 3 d	“10b” を BIN と認識
1	0	$^{\vee}_{\wedge}$, 2 3 d	
6	6	,$^{\vee}_{\wedge}$2 3 d	
0	6	,$^{\vee}$2$_{\wedge}$3 d	“,” を COM と認識
1	0	$^{\vee}_{\wedge}$2 3 d	
4	0	$^{\vee}$2$_{\wedge}$3 d	
4	0	$^{\vee}$2 3$_{\wedge}$d	
5	5	2 3 d$^{\vee}_{\wedge}$	ファイルの終了位置に到達
0	5	2 3 d$^{\vee}_{\wedge}$	“23d” を DEC と認識

3. 以下のような動作になる．

状態	LF	入力列の状態	注釈
1	0	$^{\vee}_{\wedge}$1 0 d , 2 3 b	
2	0	$^{\vee}$1$_{\wedge}$0 d , 2 3 b	
2	0	$^{\vee}$1 0$_{\wedge}$d , 2 3 b	
5	5	1 0 d$^{\vee}_{\wedge}$, 2 3 b	
0	5	1 0 d$^{\vee}$,$_{\wedge}$2 3 b	“10d” を DEC と認識
1	0	$^{\vee}_{\wedge}$, 2 3 b	
6	6	,$^{\vee}_{\wedge}$2 3 b	
0	6	,$^{\vee}$2$_{\wedge}$3 b	“,” を COM と認識
1	0	$^{\vee}_{\wedge}$2 3 b	
4	0	$^{\vee}$2$_{\wedge}$3 b	
4	0	$^{\vee}$2 3$_{\wedge}$b	
0	0	$^{\vee}$2 3 b$_{\wedge}$	字句解析エラー

問題 3 の回答

1. 以下の図の通り．以下の図の通り．ただし，最小化されていない決定性オートマトンの形は唯一ではなく，またここでは最小化を要求しないから，以下の図は一例に過ぎない．



2. 以下のような動作になる.

状態	LF	入力列の状態	注釈
1	0	$\overset{\vee}{\wedge}$1 0 , 2 3	
2	2	1$\overset{\vee}{\wedge}$0 , 2 3	
2	2	1 0$\overset{\vee}{\wedge}$, 2 3	
0	2	1 0$^{\vee}$,$_{\wedge}$2 3	“10” を BIN と認識
1	0	$\overset{\vee}{\wedge}$, 2 3	
4	4	,$\overset{\vee}{\wedge}$2 3	
0	4	,$^{\vee}$2$_{\wedge}$ 3	“,” を COM と認識
1	0	$\overset{\vee}{\wedge}$2 3	
3	3	2$\overset{\vee}{\wedge}$3	
3	3	2 3$\overset{\vee}{\wedge}$	ファイルの終了位置に到達
0	3	2 3$\overset{\vee}{\wedge}$	“23” を DEC と認識

3. 設問 1 の状態 2 には BIN トークンが付随する（状態 2 では BIN トークンが認識される）が, これは設問の元の記述順では DEC < BIN と仮定されるからであり, 優先度が変わるならば, 状態 2 には DEC トークンが付随する（状態 2 では DEC トークンが認識される）. 結果として, たとえば設問 2 の入力列についてはトークン列 DEC COM DEC が認識される.

第4章　字句解析器生成系 lex

問題 **1** の回答

1. トークン列 BIN COM DEC COM BIN COM DEC に変換される．
2. トークン列 BIN COM DEC COM DEC COM DEC に変換される．

問題 **2** の回答　省略する．各自で調べてみよ．

問題 **3** の回答　以下はその例である．プログラム作成の方針は以下の通りである．

1. 入力ファイルを変数 infile に設定する．以下のプログラムでは標準入力 stdin を設定している．
2. コメント，区切り記号にもトークン番号 COMMENT = -10，DELIM = -11 を割り付けておく．負の番号を割り付けておき，オートマトンが状態 0 へ移動したときに正の番号のトークンとは異なる処理を行う．
3. 関数 readone() において，一文字を入力から読み込む．ただし，読み込んだ文字は配列 inbuf へ一時格納し，1 回の字句解析処理終了後に，再度解析対象とする文字は入力ファイルへ ungetc を行う．
4. 字句解析関数 gettoken() には，オートマトンの全ての状態に対応するラベル L0，...，L29，L0 を用意し，そのラベルの後ろに，対応する状態での処理内容を記述する．状態遷移は，ラベルへの goto として実現する．
5. ラベル L0 の後ろには，トークンの認識処理を記述する．LF の値が正のときには そのトークン番号を return し，LF の値が負の場合には次の解析を行う（L1 へ移動する）．
6. 以下のプログラムには main 関数も加えた．デバッグに使用する．

なお，本問題に関連して本文中の該当箇所に訂正[1]がある．

```
#include <stdio.h>

enum {INT = 1, FLOAT, ID, NUM, REAL,
      COMMA, EQ, QU, SEMI, ADD, SUB, MUL, DIV, LPAR, RPAR, ERROR};
#define COMMENT -10
#define DELIM -11

FILE* infile;
int cur,LF,posLF;
```

[1]図 3.3「字句解析器を表す決定性有限状態オートマトン」の中に，状態 11 からピリオド “.” を読んで状態 13 へ遷移する枝を追加すること．

```
char inbuf[8192];
char gttext[8192];

char readone(){
        char c = fgetc(infile);
        if(c != EoF) inbuf[cur++] = c;
        return c;
}

int gettoken(void) {
        char cc;
L1:     cur = posLF = 0;
        LF = 0;
        cc = readone();
        if(cc == EoF) return 0;
        switch(cc){
        case 'f': goto L2;
        case 'i': goto L8;
        case '0': goto L11;
        case '.': goto L15;
        case '/': goto L16;
        case ' ':
        case '\t':
        case '\n': goto L20;
        case '=': goto L21;
        case '+': goto L22;
        case '-': goto L23;
        case '*': goto L24;
        case '(': goto L25;
        case ')': goto L26;
        case ',': goto L27;
        case ';': goto L28;
        }
        if(('a' <= cc)&&(cc <='z')) goto L7;
        if(('1' <= cc)&&(cc <='9')) goto L14;
        goto L29;

L2:     LF = ID; posLF = cur;
        cc = readone();
        if(('a' <= cc)&&(cc <= 'z')){
                if(cc == 'l') goto L3;
                else goto L7;
        }
        if(('0' <= cc)&&(cc <= '9')) goto L7;
        goto L0;

L3:     LF = ID; posLF = cur;
        cc = readone();
        if(('a' <= cc)&&(cc <= 'z')){
                if(cc == 'o') goto L4;
                else goto L7;
        }
```

```
        if(('0' <= cc)&&(cc <= '9')) goto L7;
        goto L0;

L4:     LF = ID; posLF = cur;
        cc = readone();
        if(('a' <= cc)&&(cc <= 'z')){
                if(cc == 'a') goto L5;
                else goto L7;
        }
        if(('0' <= cc)&&(cc <= '9')) goto L7;
        goto L0;

L5:     LF = ID; posLF = cur;
        cc = readone();
        if(('a' <= cc)&&(cc <= 'z')){
                if(cc == 't') goto L6;
                else goto L7;
        }
        if(('0' <= cc)&&(cc <= '9')) goto L7;
        goto L0;

L6:     LF = FLOAT; posLF = cur;
        cc = readone();
        if(('a' <= cc)&&(cc <= 'z')) goto L7;
        if(('0' <= cc)&&(cc <= '9')) goto L7;
        goto L0;

L7:     LF = ID; posLF = cur;
        cc = readone();
        if(('a' <= cc)&&(cc <= 'z')) goto L7;
        if(('0' <= cc)&&(cc <= '9')) goto L7;
        goto L0;

L8:     LF = ID; posLF = cur;
        cc = readone();
        if(('a' <= cc)&&(cc <= 'z')){
                if(cc == 'n') goto L9;
                else goto L7;
        }
        if(('0' <= cc)&&(cc <= '9')) goto L7;
        goto L0;

L9:     LF = ID; posLF = cur;
        cc = readone();
        if(('a' <= cc)&&(cc <= 'z')){
                if(cc == 't') goto L10;
                else goto L7;
        }
        if(('0' <= cc)&&(cc <= '9')) goto L7;
        goto L0;

L10:    LF = INT; posLF = cur;
        cc = readone();
```

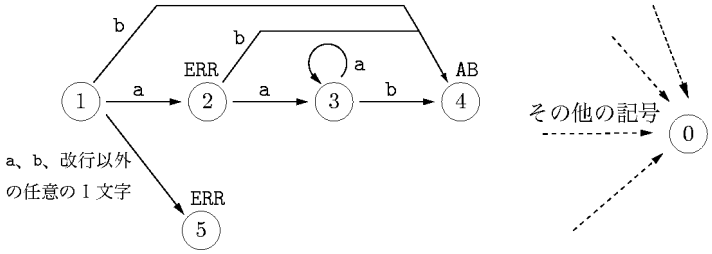
```
        if(('a' <= cc)&&(cc <= 'z')) goto L7;
        if(('0' <= cc)&&(cc <= '9')) goto L7;
        goto L0;

L11:    LF = NUM; posLF = cur;
        cc = readone();
        if(('0' <= cc)&&(cc <= '9')) goto L12;
        if(cc == '.') goto L13;   //本文訂正：図 3.3 に状態 11 から'.' を読んで
                                  //状態 13 へ遷移する枝を加えること.
        goto L0;

L12:    cc = readone();
        if(('0' <= cc)&&(cc <= '9')) goto L12;
        if(cc == '.') goto L13;
        goto L0;

L13:    LF = REAL; posLF = cur;
        cc = readone();
        if(('0' <= cc)&&(cc <= '9')) goto L13;
        goto L0;

L14:    LF = NUM; posLF = cur;
        cc = readone();
        if(('0' <= cc)&&(cc <= '9')) goto L14;
        if(cc == '.') goto L13;
        goto L0;

L15:    LF = ERROR; posLF = cur;cc = readone();
        if(('0' <= cc)&&(cc <= '9')) goto L13;
        goto L0;

L16:    LF = DIV; posLF = cur;
        cc = readone();
        if(cc == '*') goto L17;
        goto L0;

L17:    cc = readone();
        if(('a' <= cc)&&(cc <= 'z')) goto L17;
        if(('0' <= cc)&&(cc <= '9')) goto L17;
        if(cc == ' ') goto L17;
        if(cc == '*') goto L18;
        goto L0;

L18:    cc = readone();
        if(cc == '/') goto L19;
        goto L0;

L19:    LF = COMMENT; posLF = cur;
        cc = readone();
        goto L0;

L20:    LF = DELIM; posLF = cur;
        cc = readone();
```

```
        goto L0;

L21:    LF = EQ; posLF = cur;
        cc = readone();
        goto L0;

L22:    LF = ADD; posLF = cur;
        cc = readone();
        goto L0;

L23:    LF = SUB; posLF = cur;
        cc = readone();
        goto L0;

L24:    LF = MUL; posLF = cur;
        cc = readone();
        goto L0;

L25:    LF = LPAR; posLF = cur;
        cc = readone();
        goto L0;

L26:    LF = RPAR; posLF = cur;
        cc = readone();
        goto L0;

L27:    LF = COMMA; posLF = cur;
        cc = readone();
        goto L0;

L28:    LF = SEMI; posLF = cur;
        cc = readone();
        goto L0;

L29:    LF = ERROR; posLF = cur;
        goto L0;

L0:     if(LF > 0) {
                int i;
                for(i = 0; i < posLF; i++) gttext[i] = inbuf[i];
                gttext[i] = 0;
                for(i = cur-1; i >= posLF; i--) ungetc(inbuf[i],infile);
                return LF;
        }else{
                int i;
                for(i = cur-1; i >= posLF; i--) ungetc(inbuf[i],infile);
                goto L1;
        }
}

int main(){
        int k;
        infile = stdin;
```

```
        int t;
        while((t = gettoken()) != 0){
                printf("number = %d, string = \'%s\'\n",t,gttext);
        }

        return 0;
}
```

問題 4 の回答

1. 以下の図の通り．ただし，最小化されていない決定性オートマトンは唯一ではなく，またここでは最小化を要求しないから，以下の図は一例に過ぎない．



2. 以下のような動作になる．

状態	LF	入力列の状態	注釈
1	0	∨,∧a a a a	
2	2	a∨,∧a a a	
3	2	a∨a∧a a	
3	2	a∨a a∧a	
3	2	a∨a a a∧	ファイルの終了位置に到達
0	2	a∨a a a∧	“a” を ERR と認識
1	0	∨,∧a a a	
2	2	a∨,∧a a	
3	2	a∨a∧a	
3	2	a∨a a∧	ファイルの終了位置に到達
0	2	a∨a a∧	“a” を ERR と認識
1	0	∨,∧a a	
2	2	a∨,∧a	
3	2	a∨a∧	ファイルの終了位置に到達
0	2	a∨a∧	“a” を ERR と認識
1	0	∨,∧a	
2	2	a∨,∧	ファイルの終了位置に到達
0	2	a∨,∧	“a” を ERR と認識

3. 上の例が示しているように，n 個の a からなる入力列の先頭の 1 文字が ERR と認識されるまでの一連の処理で $n+1$ 回の入力動作を行う（ファイルの終了位置の発見を含む）．そ

して次回の解析は $n-1$ 個の a からなる入力列について再開する．これを繰り返すことで，n 個の a からなる入力列が n 個の ERR と解析されるまでに行われる入力動作回数は $(n+1)+n+(n-1)+\cdots+1=(n+2)(n+1)/2$ である．すなわち n^2 に比例する回数である．

補足　このように字句解析処理の最悪計算量は $O(n^2)$ である．しかし，上記の例は特殊であり，通常のプログラミング言語のトークンの解析が入力文字列の長さに依存することはありうるはずもなく，コンパイラ内の字句解析処理の計算量は $O(n)$ と考えて差し支えない．

第5章　構文解析の準備

問題 1 の回答　(a) から (g) のそれぞれについて正しい（○）/間違い（×）をつけると次の通りである．(a) ○ (b) × (c) × (d) ○ (e) × (f) × (g) ○

問題 2 の回答　以下の文法について問いに答えよ．ここに NUM，ID は終端記号である．

1: A → NUM	4: C → C ID
2: A → B C	5: D → NUM
3: B → ID	6: D → B D

1. 非終端記号 C から導出できる終端記号がひとつも存在しない．如何なる文形式にも非終端記号 D が含まれない．よって簡約形ではない．文法を簡約形にするには，非終端記号 C と D に関係する構文規則を排除すればよい．よって変換後の文法は唯一の構文規則を含む以下ものである．

   1: A → NUM

2. 如何なる文形式にも非終端記号 A，C が含まれない．よって簡約形ではない．文法を簡約形にするには，非終端記号 A と C に関係する構文規則を排除すればよい．よって変換後の文法は

   1: D → NUM
   2: D → B D
   3: B → ID

3. 簡約形の判定には個々の非終端記号 X について以下の二つの判定が必要である．

   (a) 少なくともひとつの文形式（開始記号から導出される記号列）に X が含まれるか？

   (b) X からの終端記号列が導出できるか？

   もし上の二つの質問のいずれかにでも NO と答える場合があれば，その文法は簡約形ではない．文法を簡約形にするには，もし最初の質問に NO となる場合，X を左辺とする構文規則を全て削除する（X が文形式に現れないため，関連する構文規則を文法から削除しても文法の生成する言語に影響しない）．もし二番目の質問に NO となる場合，右辺に X を含む構文規則を全て削除する（文形式に X が現れるような導出は言語に寄与しない）．それぞれの質問に答えるためには以下の手順を実行すればよい．

   (a) 各非終端記号 X に述語 reachable(X) を以下のように定義し，その値を求める．

      i. 開始記号 S について，reachable(S) = true とする．

      ii. 構文規則 $Y \to \alpha X \beta$ ($\alpha, \beta \in (T \cup N)^*$) が存在し，reachable($Y$) = true ならば，reachable($X$) = true とする．そのような構文規則が全く存在しないならば，reachable(X) = false とする．

もし reachable(X) が true ならば，最初の質問に YES と答える．さもなくば NO と答える．

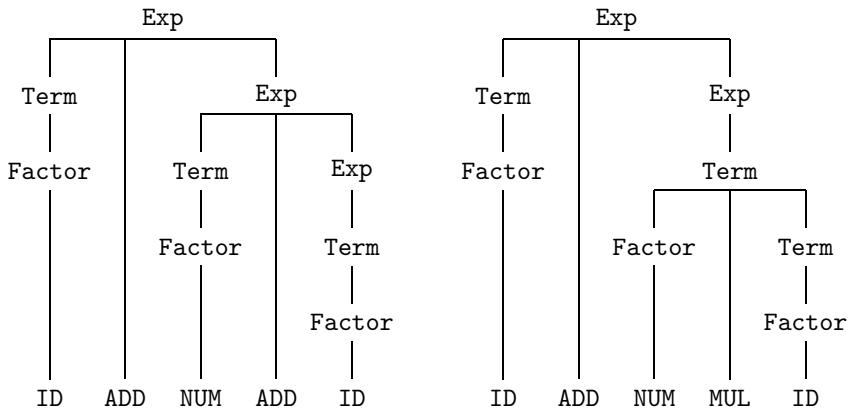
(b) 各非終端記号 X に述語 terminate(X) を以下のように定義し，値を求める．

i. 構文規則 $X \to u$（$u \in T^*$）が存在するならば，terminate(X) = true とする．

ii. 構文規則 $X \to \alpha$ が存在し，右辺 α に含まれる各非終端記号 Y について terminate(Y) = true ならば，terminate(X) = true とする．そのような構文規則が全く存在しないならば，terminate(X) = false とする．

もし terminate(X) が true ならば，二番目の質問に YES と答える．さもなくば NO と答える．

問題 3 の回答　四則演算が全て右結合性に変わる．よって，たとえば入力文字列　`a+3+b`，`a+3*b` の構文木は以下のようになる（図 5.7 との違いを比較せよ）．

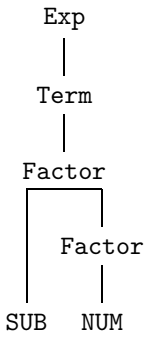


(a) `a+3+b`　　(b) `a+3*b`

問題 4 の回答　単項演算子を扱うことのできる文法は以下の通り．新たに `Factor → SUB Factor` を加えればよい．

1: `Exp → Exp ADD Term`
2: `Exp → Exp SUB Term`
3: `Exp → Term`
4: `Term → Term MUL Factor`
5: `Term → Term DIV Factor`
6: `Term → Factor`
7: `Factor → ID`
8: `Factor → NUM`
9: `Factor → LPAR Exp RPAR`
10: `Factor → SUB Factor`

`SUB NUM` の構文木は以下の通り．


Exp
Term
Factor
Factor
SUB
NUM

第6章　下向き構文解析

問題1の回答　トークン列 EX EoF を与えたときの動作は以下の表の通り．表の 5 行目の eat(NUM) の動作中に構文解析エラー（トークンの先頭が NUM ではないエラー）が起る．

	関数呼び出しの残り	入力記号の残り
1	Z();	EX EoF
2	Stmt();eOf();	EX EoF
3	eat(EX);Exp();eOf();	EX EoF
4	Exp();eOf();	EoF
5	eat(NUM);eOf();	EoF
6	eOf();	EoF

トークン列 NUM NUM EoF を与えたときの動作は以下の表の通り．表の 3 行目の eat(EX) の動作中に構文解析エラー（トークンの先頭が EX がはないエラー）が起る．

	関数呼び出しの残り	入力記号の残り
1	Z();	NUM NUM EoF
2	Stmt();eOf();	NUM NUM EoF
3	eat(EX);Exp();eOf();	NUM NUM EoF

問題2の回答　Director 集合は以下の通り．

構文規則	Director 集合
Z→A EoF	{NUM, SEMI, EoF}
A→SEMI	{SEMI}
A→B C	{NUM, SEMI, EoF}
B→ ε	{NUM, SEMI, EoF}
C→B	{NUM, EoF}
C→A NUM	{NUM, SEMI}

構文解析表は以下の通り．

	NUM	SEMI	EoF
Z	Z →A EoF	Z →A EoF	Z→A EoF
A	A→B C	A→SEMI A→B C	A→B C
B	B→ ε	B→ ε	B→ ε
C	C→B C→A NUM	C→A NUM	C→B

上の表にはひとつのエントリに複数の規則を含む箇所があるから，下向き構文解析できない．

問題 3 の回答　まず，本問題に関連して本文中の該当箇所に訂正[1]があることを注意する．

図 6.11 の文法の各非終端記号の 述語 nullable，First 集合，Follow 集合 は以下の通りである．

	nullable	First 集合	Follow 集合
Exp	false	{ID, NUM, LPAR}	{RPAR, EoF}
Exp2	true	{ADD, SUB}	{RPAR, EoF}
Term	false	{ID, NUM, LPAR}	{ADD, SUB, RPAR, EoF}
Term2	true	{MUL, DIV}	{ADD, SUB, RPAR, EoF}
Factor	false	{ID, NUM, LPAR}	{ADD, SUB, MUL, DIV, RPAR, EoF}

DIrector 集合，構文解析表は以下の通りである．

構文規則	Director 集合
0: Z → Exp EoF	{ID, NUM, LPAR}
1: Exp → Term Exp2	{ID, NUM, LPAR}
2: Exp2 → ADD Term Exp2	{ADD}
3: Exp2 → SUB Term Exp2	{SUB}
4: Exp2 → ε	{RPAR, EoF}
5: Term → Factor Term2	{ID, NUM, LPAR}
6: Term2 → MUL Factor Term2	{MUL}
7: Term2 → DIV Factor Term2	{DIV}
8: Term2 → ε	{ADD, SUB, RPAR, EoF}
9: Factor → ID	{ID}
10: Factor → NUM	{NUM}
11: Factor → LPAR Exp RPAR	{LPAR}

	ID	NUM	ADD	SUB	MUL	DIV	LPAR	RPAR	EoF
Z	0	0					0		
Exp	1	1					1		
Exp2			2	3				4	4
Term	5	5					5		
Term2			8	8	6	7		8	8
Factor	9	10					11		

以下は上記を基に作った下向き構文解析プログラムである．ここに fig06.h はこのプログラムをコンパイルするためのヘッダファイルである．サンプル・プログラム集に含まれている．このプログラムを実際に動作させるには，このプログラムを字句解析プログラムと結合する必要がある．

```
#include <stdio.h>
#include <stdlib.h>
#include "fig06.h"

void Z(void);
void Exp(void);
void Exp2(void);
```

[1]図 6.11 の構文規則 0 の右辺最左の tt を削除すること．正しい構文規則は Z → Exp EoF である．

```
void Term(void);
void Term2(void);
void Factor(void);

void error(void){ printf("syntax error\n"); exit(1); }
int  tok;
int  gettoken(void);
void advance(void){ tok = gettoken(); }
void eat(int t){ if(tok == t) advance(); else error(); }
void eOf(void){ if(tok != EoF) error(); }
int main(void){ advance(); Z(); }

void Z(void) {
    Exp(); eOf();
}
void Exp(void) {
    switch(tok){
    case ID:
    case NUM:
    case LPAR: Term(); Exp2(); break;
    default:   error();
    }
}
void Exp2(void) {
    switch(tok){
    case ADD:  eat(ADD); Term(); Exp2(); break;
    case SUB:  eat(SUB); Term(); Exp2(); break;
    case RPAR:
    case EoF:  break;
    default:   error();
    }
}
void Term(void) {
    switch(tok){
    case ID:
    case NUM:
    case LPAR: Factor(); Term2(); break;
    default:   error();
    }
}
void Term2(void) {
    switch(tok){
    case MUL:  eat(MUL); Factor(); Term2(); break;
    case DIV:  eat(DIV); Factor(); Term2(); break;
    case ADD:
    case SUB:
    case RPAR:
    case EoF:  break;
    default: error();
    }
}
void Factor(void) {
    switch(tok){
```

```
    case ID:   eat(ID); break;
    case NUM:  eat(NUM); break;
    case LPAR: eat(LPAR); Exp(); eat(RPAR); break;
    default:   error();
    }
}
```

問題 4 の回答　左再帰除去後の拡大文法は以下の通りである．

0: Z → Exp EoF
1: Exp → ID Exp2
2: Exp → NUM Exp2
3: Exp → LPAR Exp RPAR Exp2
4: Exp2 → ADD Exp Exp2
5: Exp2 → SUB Exp Exp2
6: Exp2 → MUL Exp Exp2
7: Exp2 → DIV Exp Exp2
8: Exp2 → ε

上の文法について各非終端記号の 述語 nullable，First 集合，Follow 集合 は以下の通りである．

	nullable	First 集合	Follow 集合
Exp	false	{ID, NUM, LPAR}	{ADD, SUB, MUL, DIV, RPAR, EoF}
Exp2	true	{ADD, SUB, NUL, DIV}	{ADD, SUB, MUL, DIV, RPAR, EoF}

Director 集合，構文解析表は以下の通りである．

構文規則	Director 集合
0: Z → Exp EoF	{ID, NUM, LPAR}
1: Exp → ID Exp2	{ID}
2: Exp → NUM Exp2	{NUM}
3: Exp → LPAR Exp RPAR Exp2	{LPAR}
4: Exp2 → ADD Exp Exp2	{ADD}
5: Exp2 → SUB Exp Exp2	{SUB}
6: Exp2 → MUL Exp Exp2	{MUL}
7: Exp2 → DIV Exp Exp2	{DIV}
8: Exp2 → ε	{ADD, SUB, MUL, DIV, RPAR, EoF}

	ID	NUM	ADD	SUB	MUL	DIV	LPAR	RPAR	EoF
Z	0	0					0		
Exp	1	2					3		
Exp2			4 8	5 8	6 8	7 8		8	8

構文解析表に衝突があるから，この文法は下向き構文解析できない．

問題 5 の回答

1. 左括り出し後の文法は以下の通り．

0: Z → Input EoF
1: Input → Seq
2: Seq → NUM Seq2
3: Seq2 → ε
4: Seq2 → Seq

2. 上の文法について各非終端記号の 述語 nullable，First 集合，Follow 集合 は以下の通りである．

	nullable	First 集合	Follow 集合
Input	false	{NUM}	{EoF}
Seq	false	{NUM}	{EoF}
Seq2	true	{NUM}	{EoF}

Director 集合，構文解析表は以下の通りである．

構文規則	Director 集合
0: Z → Input EoF	{NUM}
1: Input → Seq	{NUM}
2: Seq → NUM Seq2	{NUM}
3: Seq2 → ε	{EoF}
4: Seq2 → Seq	{NUM}

	NUM	EoF
Z	0	
Input	1	
Seq	2	
Seq2	4	3

問題 6 の回答　以下がそのプログラムの例である．このプログラムは以下の方針で作成した．

1. 元のプログラムの関数名 Z，Program，DeclStmts，DeclStmts2，DeclStmt，VarDefs，VarDefs2，PrintStmts，PrintStmt，VarRefs，VarRefs2，eOf をそのまま enum の定数名とする．ただし，トークン種別番号と重複しないように，Z 以下の定数値を 1000 以上の数値とする．
2. 関数名とトークン名をスタック stack に積んで行く．元のプログラムにおいて関数が f_1，f_2，...，f_n の順に実行されるとき，スタックには先に実行される関数名をスタックの上に積む．
3. プログラムの毎回の実行はスタックの先頭（頂上）の値に応じて switch 文で制御する．実際の実行内容は，元のプログラムの関数の動作をそのままスタック動作に翻訳する．
4. 関数名 eOf がスタックの先頭（頂上）にあるときに，トークン列の先頭が EoF ならば構文解析は成功したものと見なす．
5. トークンがスタックの先頭（頂上）にあるときには，そのトークンと同じトークンを入力トークン列から eat する．同じトークンを eat できないときには構文解析エラーと見なす．

```
#include <stdio.h>
#include <stdlib.h>
#include "fig06.h"
```

```
enum { Z = 1000,Program,DeclStmts,DeclStmts2,DeclStmt,VarDefs,
VarDefs2,PrintStmts,PrintStmt,VarRefs,VarRefs2,eOf};

int stack[1000];
int stacktop = -1;

void error(void){ printf("syntax error\n"); exit(1); }
int  tok;
int  gettoken(void);
void advance(void){ tok = gettoken(); }
void eat(int t){ if(tok == t) advance(); else error(); }

int main(void){
    int toptok;
    int i;

    advance();
    stack[++stacktop] = Z;

START:
    switch(toptok = stack[stacktop--]){
    case Z:
            stack[++stacktop] = eOf;
            stack[++stacktop] = Program;
            goto START;
    case Program:
            stack[++stacktop] = PrintStmts;
            stack[++stacktop] = DeclStmts;
            goto START;
    case DeclStmts:
            stack[++stacktop] = DeclStmts2;
            stack[++stacktop] = SEMI;
            stack[++stacktop] = DeclStmt;
            goto START;
    case DeclStmts2:
            switch(tok){
            case EX:
            case EoF:
                    break;
            case INT:
                    stack[++stacktop] = DeclStmts;
                    break;
            default: error();
            }
            goto START;
    case DeclStmt:
            stack[++stacktop] = VarDefs;
            stack[++stacktop] = INT;
            goto START;
    case VarDefs:
            stack[++stacktop] = VarDefs2;
            stack[++stacktop] = NUM;
            stack[++stacktop] = EQ;
```

```
        stack[++stacktop] = ID;
        goto START;
case VarDefs2:
        switch(tok){
        case COMMA:
                stack[++stacktop] = VarDefs2;
                stack[++stacktop] = NUM;
                stack[++stacktop] = EQ;
                stack[++stacktop] = ID;
                stack[++stacktop] = COMMA;
                break;
        case SEMI:
                break;
        default:    error();
        }
        goto START;
case PrintStmts:
        switch(tok){
        case EX:
                stack[++stacktop] = PrintStmts;
                stack[++stacktop] = SEMI;
                stack[++stacktop] = PrintStmt;
                break;
        case EoF:
                break;
        default:  error();
        }
        goto START;
case PrintStmt:
    stack[++stacktop] = VarRefs;
    stack[++stacktop] = EX;
    goto START;
case VarRefs:
    stack[++stacktop] = VarRefs2;
    stack[++stacktop] = ID;
    goto START;
case VarRefs2:
    switch(tok){
    case COMMA:
            stack[++stacktop] = VarRefs2;
            stack[++stacktop] = ID;
            stack[++stacktop] = COMMA;
            break;
    case SEMI:
            break;
    default: error();
    }
    goto START;
case eOf:
    if(tok != EoF) error();
    return;
default:
    eat(toptok);
```

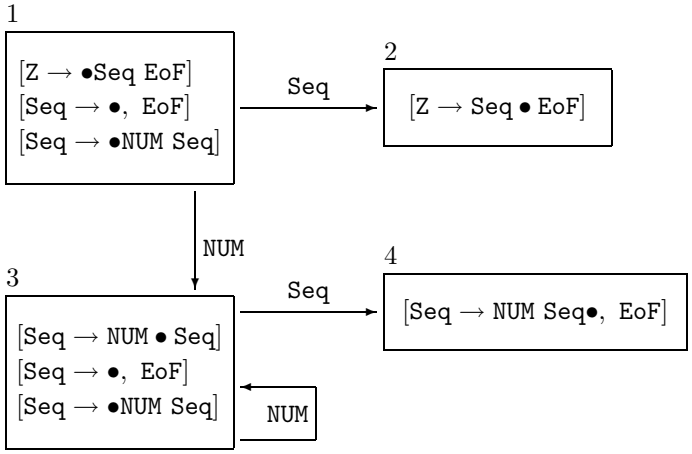
```
        goto START;
    }
}
```

第7章　上向き構文解析

問題 1 の回答　構文解析過程は表 7.2 と同じである（基本形の場合）．同じになることを読者自ら確認してほしい．

問題 2 の回答　ある LR 状態 s_i から記号 A を読んで LR 状態 s_j へ遷移すると仮定する．このとき，s_j が空でないならば，s_i には $[X \to \alpha \bullet A\beta]$ という形の LR(0) 項が存在し，s_j には $[X \to \alpha A \bullet \beta]$ という形の LR(0) 項が存在する．次に，s_i から記号 B を読んで LR 状態 s'_j へ遷移すると仮定する．このとき，もし $A \neq B$ ならば，s'_j には LR(0) 項 $[X \to \alpha A \bullet \beta]$ は決して含まれない．s'_j に含まれるその他の LR(0) 項から関係 $\downarrow$ によって $[X \to \alpha A \bullet \beta]$ が導出されることもない（$\downarrow$ によって導出される LR(0) 項の LR マーカーは必ず右辺の最左に付く）．よって s_i と s'_j は異なる．

問題 3 の回答　まず図 7.8 の文法の非終端記号 Seq の Follow 集合は {EoF} である．このことを用いて図 7.9 を修正すると，SLR オートマトンは以下の通りである．

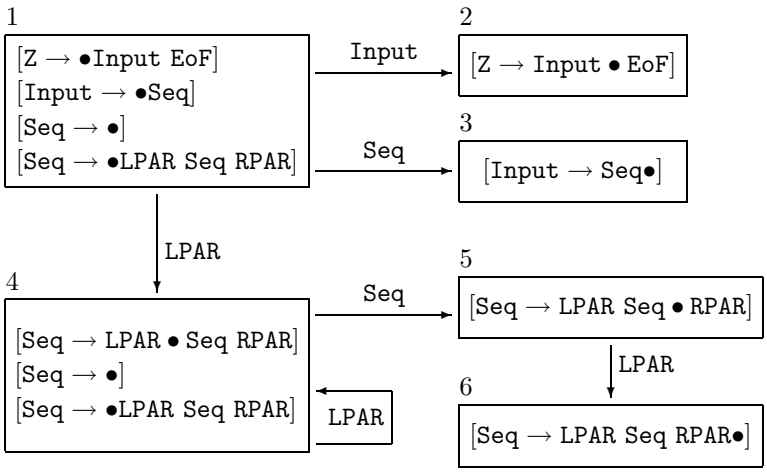


SLR 構文解析表は以下の通りである．

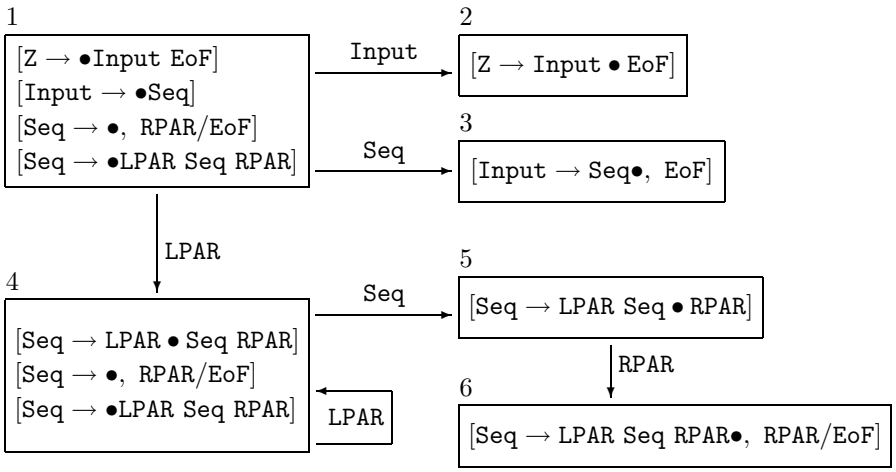
	NUM	EoF	Seq
1	S(3)	R[1]	S(2)
2		A	
3	S(3)	R[1]	S(4)
4		R[2]	

問題 4 の回答

1. 以下が LR(0) オートマトンである．状態 1，状態 4 にシフト/還元衝突があるから，この文法は LR(0) 文法ではない．

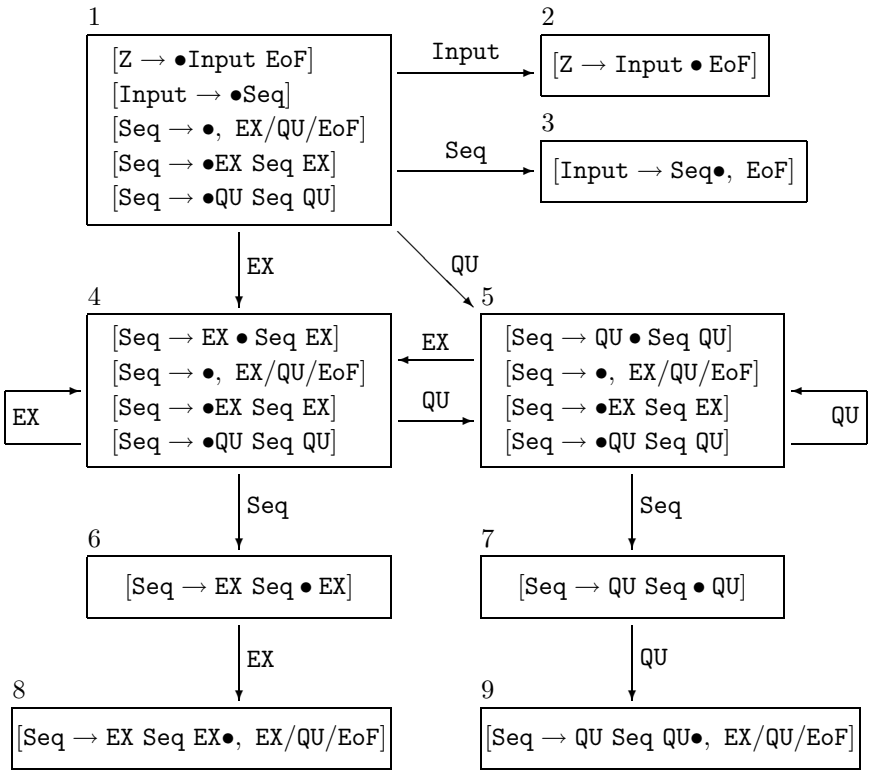


2. 以下が SLR オートマトンである．ここに，Follow(Input) = {EoF}，
Follow(Seq) = {RPAR, EoF} であることを用いた．いずれの状態にも衝突はないから，この文法は SLR 文法である．

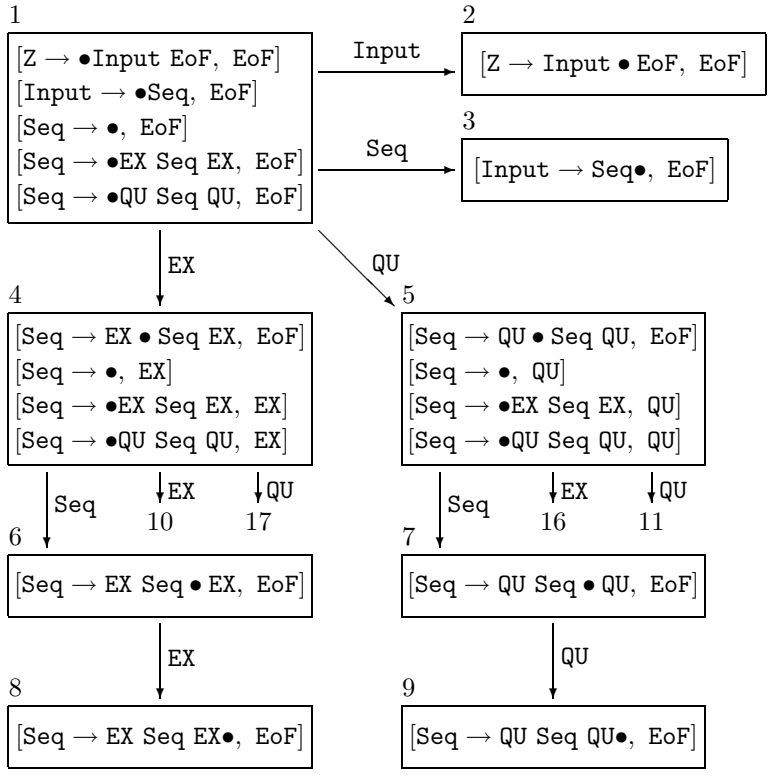


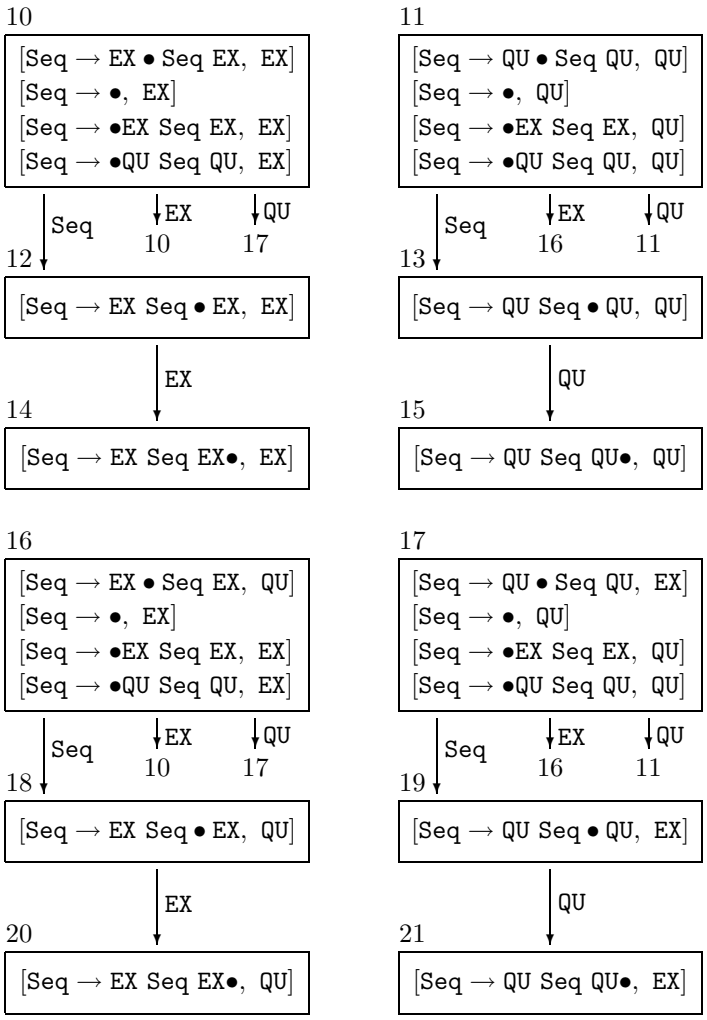
問題 **5** の回答

1. 以下が SLR オートマトンである．ここに，Follow(Input) = {EoF}，
Follow(Seq) = {EX, QU, EoF} であることを用いた．状態 1，状態 4，状態 5 に衝突があるから，この文法は SLR 文法ではない．



2. 以下が LR(1) オートマトンである．状態 4，5，10，11，16，17 に衝突があるから，この文法は LR(1) 文法ではない．





問題 **6** の回答

性質 **(1)**　LR(1) 項の核は LR(0) 項であり，その核によって LR 状態を構成しているから，LALR(1) オートマトンの状態は LR(0) オートマトンの状態と 1 対 1 対応が付く．よって，LALR(1) オートマトンの状態数は LR(0) オートマトンの状態数に等しい．SLR オートマトンの状態数は LR(0) オートマトンの状態数に等しいから，LALR(1) オートマトンの状態数は SLR オートマトンの状態数とも等しい．

性質 **(2)**　同じ核を持つ二つの LR(1) 状態が以下のように与えられたとしよう．

$$I_1 = \left\{ \begin{array}{l} [X \to \alpha \bullet s_0\beta,\ s_1/.../s_m], \\ [Y \to \gamma\bullet,\ t_1/.../t_n] \end{array} \right\}, \qquad I_2 = \left\{ \begin{array}{l} [X \to \alpha \bullet s_0\beta,\ s'_1/.../s'_{m'}], \\ [Y \to \gamma\bullet,\ t'_1/.../t'_{n'}] \end{array} \right\}$$

ここに s_0，$s_1, ..., s_m$，$s'_1, ..., s'_{m'}$，$t_1, ..., t_n$，$t'_1, ..., t'_{n'}$ は終端記号である．それぞれの状態にはシフト/還元衝突はないとする．衝突がないならば，s_0 と $t_1, ..., t_n$ は異なる終端記号であり，s_0 と $t'_1, ..., t'_{n'}$ も異なる終端記号である．さて，この二つの LR(1) 状態を合併した LALR(1) 状態は以下の通りである．

$$I_1 = \left\{ \begin{array}{l} [X \to \alpha \bullet s_0\beta,\ s_1/.../s_m/s'_1/.../s'_{m'}], \\ [Y \to \gamma\bullet,\ t_1/.../t_n/t'_1/.../t'_{n'}] \end{array} \right\}$$

s_0 と $t_1, ..., t_n$ は異なる終端記号であり，s_0 と $t'_1, ..., t'_{n'}$ も異なる終端記号であるから，上の状態にもシフト/還元衝突はない．この性質は，シフト項の数が複数の場合，還元項の数が複数の場合についても同様に示すことができる．

合併によって還元/還元衝突が起る場合のことは次の問題 7 の回答に示す．

問題 7 の回答　文法 G が LR(1) であり，LALR(1) ではないとする．そうすると，G の LR(1) オートマトンにはシフト/還元衝突も還元/還元衝突もないが，LALR(1) 文法の性質 (2) から LALR(1) オートマトンには還元/還元衝突がある．還元/還元衝突があるならば，その LALR(1) オートマトンには少なくとも二つの還元項を持つ状態が存在する．そして LR(1) オートマトンにはその衝突の元となる，同じ核を持つ二つの状態が存在する．その二つの状態を以下のように仮定しよう．ここに A，B，C は終端記号，X，Y は非終端記号である．α，β の列の形を具体化することは当面保留しておく．

$$I_1 = \left\{ \begin{array}{l} [\mathtt{X} \to \alpha\bullet,\ \mathtt{A}], \\ [\mathtt{Y} \to \beta\bullet,\ \mathtt{B}] \end{array} \right\}, \qquad I_2 = \left\{ \begin{array}{l} [\mathtt{X} \to \alpha\bullet,\ \mathtt{C}], \\ [\mathtt{Y} \to \beta\bullet,\ \mathtt{A}] \end{array} \right\}$$

これを合併すると以下のように，先読みが A のときに還元/還元衝突を持つ．

$$[I_1, I_2] = \left\{ \begin{array}{l} [\mathtt{X} \to \alpha\bullet,\ \mathtt{A/C}], \\ [\mathtt{Y} \to \beta\bullet,\ \mathtt{A/B}] \end{array} \right\}$$

そこで，LR(1) オートマトンが上の I_1，I_2 と同様の状態を持つような文法を作ればよい．以下はそのような例である．ここに D は終端記号である．

0: Z → S EoF
1: S → P Q
2: P → X A
3: P → Y B
4: Q → X C
5: Q → Y A
6: X → D
7: Y → D

この場合，上の I_1，I_2，$[I_1, I_2]$ に対応する各状態は以下の通りである．オートマトン全体の記述は省略する．読者自ら確認してほしい．

$$I_1 = \left\{ \begin{array}{l} [\mathtt{X} \to \mathtt{D}\bullet,\ \mathtt{A}], \\ [\mathtt{Y} \to \mathtt{D}\bullet,\ \mathtt{B}] \end{array} \right\}, \qquad I_2 = \left\{ \begin{array}{l} [\mathtt{X} \to \mathtt{D}\bullet,\ \mathtt{C}], \\ [\mathtt{Y} \to \mathtt{D}\bullet,\ \mathtt{A}] \end{array} \right\}, \qquad [I_1, I_2] = \left\{ \begin{array}{l} [\mathtt{X} \to \mathtt{D}\bullet,\ \mathtt{A/C}], \\ [\mathtt{Y} \to \mathtt{D}\bullet,\ \mathtt{A/B}] \end{array} \right\}$$

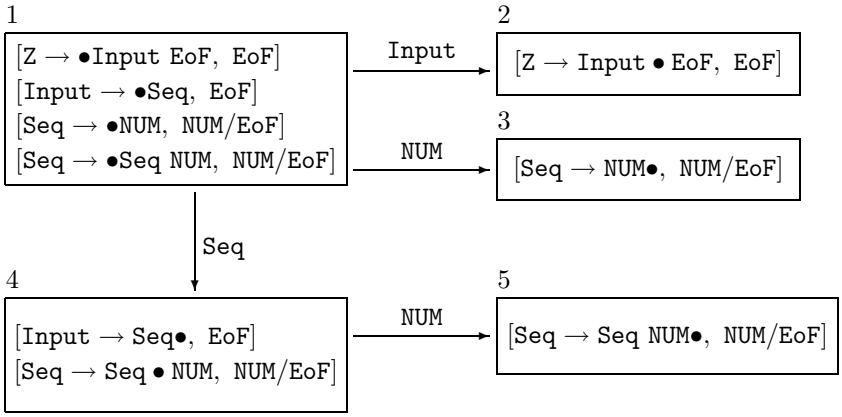
第8章　構文解析器生成系yacc

問題1の回答　次の左再帰的な文法を検討しよう．

0: Z → Input EoF　　2: Seq → NUM

1: Input → Seq　　3: Seq → Seq NUM

この文法のLALR(1)オートマトンは以下の通りである（なお，この文法ではLR(1)オートマトンとLALR(1)オートマトンは同一である）．

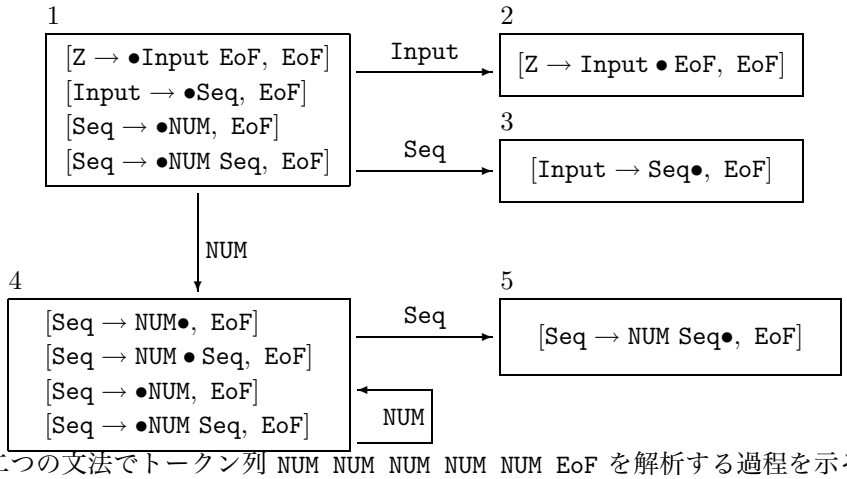


次に，上の文法と等価な，左再帰的でない（右再帰的な）な文法を以下のように想定しよう．

0: Z → Input EoF　　2: Seq → NUM

1: Input → Seq　　3: Seq → NUM Seq

この文法のLALR(1)オートマトンは以下の通りである（なお，この文法ではLR(1)オートマトンとLALR(1)オートマトンは同一である）．



この二つの文法でトークン列 NUM NUM NUM NUM NUM EoF を解析する過程を示そう．前者の場合は以下の通りである．

	状態スタック	入力記号の残り
1	1	NUM NUM NUM NUM NUM EoF
2	1 3	NUM NUM NUM NUM EoF
3	1	Seq NUM NUM NUM NUM EoF
4	1 4	NUM NUM NUM NUM EoF
5	1 4 5	NUM NUM NUM EoF
6	1	Seq NUM NUM NUM EoF
7	1 4	NUM NUM NUM EoF
8	1 4 5	NUM NUM EoF
9	1	Seq NUM NUM EoF
10	1 4	NUM NUM EoF
11	1 4 5	NUM EoF
12	1	Seq NUM EoF
13	1 4	NUM EoF
14	1 4 5	EoF
15	1	Seq EoF
16	1 4	EoF
17	1	Input EoF
18	1 2	EoF

後者の場合は以下の通りである.

	状態スタック	入力記号の残り
1	1	NUM NUM NUM NUM NUM EoF
2	1 4	NUM NUM NUM NUM EoF
3	1 4 4	NUM NUM NUM EoF
4	1 4 4 4	NUM NUM EoF
5	1 4 4 4 4	NUM EoF
6	1 4 4 4 4 4	EoF
7	1 4 4 4 4	Seq EoF
8	1 4 4 4 4 5	EoF
9	1 4 4 4	Seq EoF
10	1 4 4 4 5	EoF
11	1 4 4	Seq EoF
12	1 4 4 5	EoF
13	1 4	Seq EoF
14	1 4 5	EoF
15	1	Seq EoF
16	1 3	EoF
17	1	Input EoF
18	1 2	EoF

上の例から分かるように，前者ではひとつのトークン（NUM）を読むたびに還元が起きるからスタックの長さは一定の深さ（高さ）以上に伸びない．それに対して，後者では，まず全てのトーク

ンをスタックに載せるから，入力トークン列の長さに比例した長さのスタックが必要である．この性質は，この例題だけではなく，左再帰的な文法，非左再帰的な文法（と言うよりも右再帰的な文法）に共通した性質である．

コンピュータのメモリが高価であった時代には，空間効率の観点から左再帰的な文法が有効とされた．メモリが潤沢であっても，必要なスタックの長さが入力列に依存せず，$O(1)$ であることは大きな利点と考えられる．

問題 **2** の回答　以下の通りである．

```
%token ID, NUM, ADD, SUB, MUL, DIV, LPAR, RPAR, ERROR;
%%
Exp: Exp ADD Term {}
   | Exp SUB Term {}
   | Term {}

Term: Term MUL Factor {}
    | Term DIV Factor {}
    | Factor {}

Factor: ID {}
      | NUM {}
      | LPAR Exp RPAR {}
%%
#include "lex.yy.c"
int main(){
    if(!yyparse()) printf("successfully ended\n");
}
void yyerror(char* s){ fprintf(stderr,"%s\n",s); }
```

第9章　抽象構文木の構築

問題 **1** の回答　サンプルプログラム fig09_20.h，fig09_20.c に載せた．

問題 **2** の回答　サンプルプログラム fig09_09.c は cc -c でコンパイルに成功する．fig09_12.c は，fig09_10.l を lex でコンパイルした後（lex.yy.c を生成した後），cc -c でコンパイルに成功する．それらを組み合わせればよい．分割コンパイルもそれらファイル毎に行えばよい．

問題 **3** の回答　サンプルプログラム fig09_09.c は cc -c でコンパイルに成功する．fig09_16.y は，fig09_14.l を lex でコンパイルした後（lex.yy.c を生成した後），yacc でコンパイルし，さらに生成された y.tab.c を cc -c でコンパイルできる．それらを組み合わせればよい．分割コンパイルもそれらファイル毎に行えばよい．

問題 **4** の回答　読者みずから確認せよ．

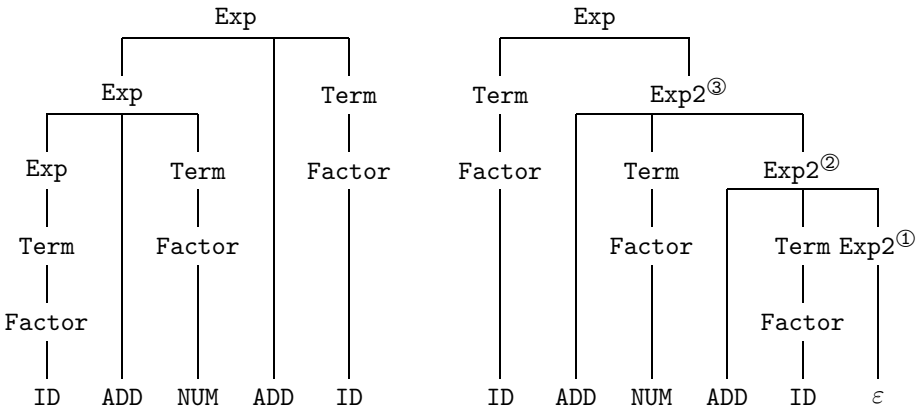
問題 **5** の回答　ここでは

1. 下向き構文解析プログラムに意味処理を追加すること，および
2. 上向き構文解析プログラムに意味処理を追加すること

の二つを要求されているが，この二つでは前者が難しい問題である．というのも，前者では元の文法に左再帰除去を適用して得られた文法を構文解析に使用している．その文法構造が抽象構文木の構造と一致しないからである．これに対して，後者の上向き構文解析プログラムに意味処理を追加することは 9 章の本文とほとんど同じ処理を行えばよい．そこで，この回答では，後者については回答を省略し，前者について詳しく解説する．

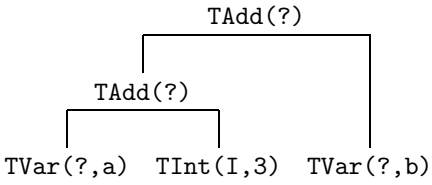
たとえば，入力文字列 a+3+b，そのトークン列 ID ADD NUM ADD ID に対応する図 5.6 の文法の構文木は下図の (a) の通りである．これに対して，同じトークン列に対応する図 6.11 の文法の構文木は下図の (b) の通りである．ただし，構文規則 0 は無視する．Exp2 の肩の①，②，③については後に述べる．

(a) の木では，木の左側の枝が深く茂っている．これは加算演算の左結合性を表しているためである．これに対して，(b) の木では，木の右側の枝が茂っている．これは左再帰除去の適用後，文法の形が変わったためである．



(a) (b)

次に，この入力文字列に対応する抽象構文木は以下の通りである．なお，構文木との比較のため，抽象構文木の枝は上から下へ伸びるように表記した（9 章本文では，紙面の都合上，枝は左から右へ伸びるように表記した）．

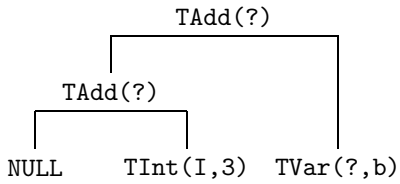


さて，上記 (a) の構文木と上記抽象構文木は同じ形状（左の枝が深い形状）をしており，構文解析と同時にその抽象構文木を作ることは 9 章の例題と同様に可能である．これに対して，上記 (b) の構文木と上記抽象構文木は異なる（枝の伸び方が異なる）形状をしており，構文解析と同時にその抽象構文木を作るのはそれほど簡単ではなく，9 章の関数 $F()$ を巧妙に作らねばならない．

そこで，上記 (b) の構文解析と同時に上記抽象構文木を作るために，以下のような関数を用意する．この関数では，関数の第二引数の抽象構文木の最左の葉は NULL であると仮定している．その葉を関数の第一引数で置換し，置換した木を return する．

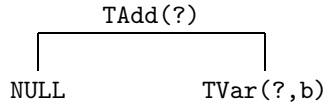
```
node* insertLeftMost(node *npl, node *npr){
        node *tmp;
        if(npr == NULL) return npl;
        for(tmp = npr; tmp->left != NULL; tmp = tmp->left) ;
        tmp->left = npl;
        return npr;
}
```

たとえば，第二引数が以下のような形状の木ならば，最左の葉の NULL を第一引数で置換する．

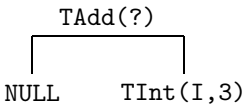


実は上の木は上記 (b) の③の Exp2 の意味値である．もし第一引数として TAdd(?) を与えたならば，置換後の木は求める抽象構文木になる．

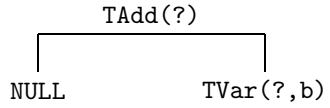
別の例として，第二引数が以下のような形状の木であったとしよう．



実は上の木は上記 (b) の②の Exp2 の意味値である．第一引数として以下の木を与えたならば，置換後の木は上記 (b) の③の Exp2 の意味値になる．



別の例として，第二引数が NULL であったとしよう．実はこの木は上記 (b) の①の Exp2 の意味値である．この木（NULL）に，第一引数として以下の木を与えたならば，置換後の木は第一引数そのものであり，それは上記 (b) の②の Exp2 の意味値である．



このように関数 insertLeftMost() を用いることで，半ば強引にではあるが，構文木とは形状の異なる抽象構文木を構築できる．以下は，上記の処理を行う意味記述を追加した下向き構文解析プログラムである．

```
#include <stdio.h>
#include <stdlib.h>
#include "fig06.h"
#include "fig09_04.h"
#include "fig09_09.h"
#include "fig09_20.h"

node* lval;
#include "lex.yy.c"

node *Z(void);
node *Exp(void);
node *Exp2(void);
node *Term(void);
node *Term2(void);
node *Factor(void);

void error(void){ printf("syntax error\n"); exit(1); }
int  tok;
int  gettoken(void);
void advance(void){ tok = yylex(); }
node *eat(int t){ node *p = lval; if(tok == t) advance(); else error(); return p; }
void eOf(void){ if(tok != EoF) error(); }
```

```
int main(void){
    node* np;
    advance();
    np = Z();
    print(np);
}

node* insertLeftMost(node *npl, node *npr);

node *Z(void) {
    node *np1;
    np1 = Exp();
    eOf();
    return np1;
}
node *Exp(void) {
    node *np1,*np2;
    switch(tok){
    case ID:
    case NUM:
    case LPAR:
        np1 = Term();
        np2 = Exp2();
        return insertLeftMost(np1,np2);
    default:   error();
    }
}
node *Exp2(void) {
    node *np2,*np3;
    switch(tok){
    case ADD:
        eat(ADD);
        np2 = Term();
        np3 = Exp2();
        return insertLeftMost(newTAdd(NULL,np2),np3);
    case SUB:
        eat(SUB);
        np2 = Term();
        np3 = Exp2();
        return insertLeftMost(newTSub(NULL,np2),np3);
    case RPAR:
    case EoF:
        return NULL;
    default:   error();
    }
}
node *Term(void) {
    node *np1,*np2;
    switch(tok){
    case ID:
    case NUM:
    case LPAR:
        np1 = Factor();
```

```
        np2 = Term2();
        return insertLeftMost(np1,np2);
    default:   error();
    }
}
node *Term2(void) {
    node *np2,*np3;
    switch(tok){
    case MUL:
        eat(MUL);
        np2 = Factor();
        np3 = Term2();
        return insertLeftMost(newTMul(NULL,np2),np3);
    case DIV:
        eat(DIV);
        np2 = Factor();
        np3 = Term2();
        return insertLeftMost(newTDiv(NULL,np2),np3);
    case ADD:
    case SUB:
    case RPAR:
    case EoF:
        return NULL;
    default: error();
    }
}
node *Factor(void) {
    node *np1,*np2;
    switch(tok){
    case ID:
        np1 = eat(ID);
        return np1;
    case NUM:
        np1 = eat(NUM);
        return np1;
    case LPAR:
        eat(LPAR);
        np2 = Exp();
        eat(RPAR);
        return np2;
    default:   error();
    }
}

node* insertLeftMost(node *npl, node *npr){
    node *tmp;
    if(npr == NULL) return npl;
    for(tmp = npr; tmp->left != NULL; tmp = tmp->left) ;
    tmp->left = npl;
    return npr;
}
```

第10章　中間木の構築

問題1の回答　以下がそのプログラムである.

```
#include <stdio.h>
#include <stdlib.h>
#include "fig09_04.h"
#include "fig09_09.h"
#include "fig10_01.h"
#include "fig10_03.h"

void checkSemType(node* np){
    if(np == NULL) return;
    switch(np->label){
    case TDecl:
        putVar(np->name,np->type);
        np->left = castType(np->left,np->type);
    case TInput: case TVar:
        np->type = getType(np->name);
        break;
    case TAssign:
        np->type = getType(np->name);
        checkSemType(np->left);
        np->left = castType(np->left,np->type);
        break;
    case TPrint:
        checkSemType(np->left);
        np->type = np->left->type;
        break;
    case TProgram:
    case TDeAsInSeq:
    case TPrintSeq:
        checkSemType(np->left);
        checkSemType(np->right);
        break;
    case TAdd:
    case TSub:
    case TMul:
    case TDiv:
        checkSemType(np->left);
        checkSemType(np->right);
        if(np->left->type == np->right->type) {
            np->type = np->left->type;
        } else {
            np->left = castType(np->left,TFLOAT);
            np->right = castType(np->right,TFLOAT);
            np->type = TFLOAT;
```

```
            }
        }
    }
```

第11章　インタプリタとコンパイラ

問題 1 の回答　ここでは lex と yacc を用いる方法を考えよう．

まずアセンブリコード中のトークンを以下のように lex で定義する．ただし，以下ではトークンの意味値はまだ考慮していない．

```
%%
0|[1-9][0-9]*        { return NUM; }     //整定数
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+) {
                       return REAL; }    //浮動小数点定数
"["[a-z][a-z0-9]*"]" { return MEMADD; }  //メモリアドレス
"r"(0|[1-9][0-9]*)   { return IR; }      //整数レジスタ
"f"(0|[1-9][0-9]*)   { return FR; }      //浮動小数点レジスタ
","                  { return COMMA; }
"="                  { return EQ; }
"\n"                 { return NL; } //改行．命令間には少なくともひとつの改行を置く．
"load.i"             { return LOADI; }   //以下，個々の命令名
"load.f"             { return LOADF; }
"store.i"            { return STOREI; }
"store.f"            { return STOREF; }
"const.i"            { return CONSTI; }
"const.f"            { return CONSTF; }
...
以下同様に全ての命令名をここに定義する．
...
" "|"\t"             { }                 //区切り記号
.                    { return ERROR; }   //エラートークン
%%
int yywrap(){ return 1; }
```

次に構文規則を以下のように yacc で定義する．ただし，以下ではまだ意味処理は考慮していない．

```
%token NUM, REAL, MEMADD, IR, FR, COMMA, EQ, NL,
       LOADI, LOADF, STOREI, STOREF, CONSTI, CONSTF, ...
       同様に全ての命令名トークンをここに宣言する ..., ERROR
%{
#include <string.h>
%}
%%
Program
    : InstSeq {}
InstSeq
    : Inst NL {}
    | InstSeq Inst NL {}
Inst
```

```
    : /* empty */ {}          //改行だけの行を許す
    | LOADI IR EQ MEMADD {}
    | LOADF FR EQ MEMADD {}
    | STOREI MEMADD EQ IR {}
    | STOREF MEMADD EQ FR {}
    | CONSTI IR EQ NUM {}
    | CONSTF FR EQ REAL {}
    | ... 以下同様に全ての命令形式を定義する ...

%%
#include "lex.yy.c"
int main(){
    if(!yyparse()) printf("successfully ended\n");
}
void yyerror(char* s){ fprintf(stderr,"%s\n",s); }
```

ここに，上の構文規則の InstSeq は左再帰的に定義されていることに注意する．その場合，トークン列中の先頭の命令 Inst に関する規則から順に還元が起きる．よって，以下の構文規則のアクション部に命令の実行内容を記述すれば，先頭の命令から後続の命令へ順に命令が実行されていく．

```
    | LOADI IR EQ MEMADD {}
    | LOADF FR EQ MEMADD {}
    | STOREI MEMADD EQ IR {}
    | STOREF MEMADD EQ FR {}
    | CONSTI IR EQ NUM {}
    | CONSTF FR EQ REAL {}
    | ... 以下同様に全ての命令形式を定義する ...
```

意味値の計算方法や意味処理の具体的な記述方法は読者に残しておく．

問題 2 の回答　様々な推定方法があろうが，以下はそのひとつである．まず，手順を示そう．

1. 測定に使用するベンチマークプログラムには以下の SL1 のプログラムを用いることにしよう．このプログラムは，1 個の宣言文と 20 個の積和演算の代入文からなる．

```
int x = 2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
```

```
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
x = 3*x+2;
```

このプログラムによって，主に演算の実行時間を比較できる．

2. プログラム中の特定部分の実行時間をプロセッサのマシン・マイクル単位で計測することを考える．IA-32 マシンの場合，プロセッサ内蔵の time stamp counter register の値を読み出す命令 `rdtsc` を用いればよい．たとえば以下はその計測部の抜粋である[1]．

```
#define RDTSC(x) asm volatile ("rdtsc" : "=a"(x))

    unsigned int before,after;

    RDTSC(before);

    /* RDTSC() に挟まれた，この部分の実行時間を計測する */

    RDTSC(after);
    printf("%u\n",after-before);
```

計測値はマシン・サイクル（MC）である．Power PC の場合には time base register の値を読み出す命令 `mftb` を用いればよい．プログラムは上と同様である．

3. インタプリタの実行時間を測定するために，表 11.2 の関数 `eval()` の実行時間を上の方法で計測する．

4. 本書では仮想的なプロセッサのコードを生成しているため，コンパイラによって生成されたコードの実行時間を計測することはできない．そこで類似の実行時間を求めるために，上の SL1 ベンチマークプログラムを以下の C プログラムに書き直す．

```
int main(){
    unsigned int before,after;
    int x = 2;
    RDTSC(before);
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
```

[1] カウンターは 64bit 長である．このプログラムでは下位 32bit のカウンター値を読み出しているため，稀に下位 32bit がオーバーフローし，計測値が負値になる場合がある．

```
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    x = 3*x+2;
    RDTSC(after);
    printf("%u\n",after-before);
}
```

そしてこのコードを C コンパイラでコンパイルし，IA-32 マシン上で実際に実行させて実行時間を得る．

さて，著者の手元にある IA-32 マシンで上記の実験を行った結果，インタプリタの実行時間は約 28000 MC であった．コンパイルしたコードの実行時間は約 270 MC であった．結果として，インタプリタの実行時間はコンパイルされたコードに比べ，約 100 倍遅いことになる．

この結果は，11 章のコラムに示している「インタプリタは 5 倍から 20 倍遅い」という範囲の外にある．その理由として，本書のインタプリタの作り方が簡便さを優先しており，効率を犠牲にしていることが挙げられる．表 11.1 はインタプリタの演算部のプログラムであるが，ひとつの演算を行う毎に `newTInt()` や `newTFloat()` を実行し，`node` 構造体をヒープ領域に生成している．この処理が全体の効率を落とす大きな要因になっていることは容易に想像される．もし本格的な使用に耐えうるインタプリタを作るならば，その辺りの改善が必須であろう．