

清华大学本科生考试试题专用纸

考试课程 《编译原理》 (A 卷) 2019 年 12 月 31 日

学号: _____ 姓名: _____ 班级: _____

(注: 解答可以写在答题纸上, 也可以写在试卷上; 交卷时二者均需上交。)

一. (16分) 简答题:

1. (2分) 设文法 $G[A]$ 为:

$A \rightarrow aB$

$A \rightarrow Bb$

$B \rightarrow Ac$

$B \rightarrow d$

试变换该文法, 得到一个等价的不含左递归的文法。

2. (3分) 已知文法 $G(E)$:

$E \rightarrow T \mid E + T$

$T \rightarrow F \mid T * F$

$F \rightarrow (E) \mid i$

给出句型 $(T * F + i)$ 的短语、直接短语、句柄。

3. (4分) 对于以下 Decaf/Mind 程序, 在*****处有哪些开作用域? 其中分别有哪些符号?

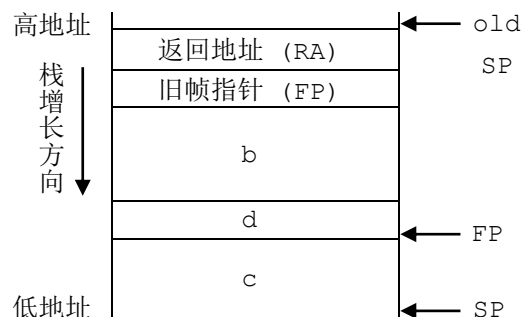
```
class Animal {
    int id;
    string name;
    void touch() { Print("unimplement!"); }
    void foo(int bar) {}
}
class Cat extends Animal {
    void touch() { Print("Touching cat ", name, "\n"); }
}
class Fish extends Animal {
    int type;
    void setid(int id) {
        int newId = id + type;
        this.id = newId;
        // *****
    }
    void touch() { Print("Touching fish ", name, "\n"); }
}
class Main {
    static void main() {
        class Fish fish = new Fish();
        fish.touch();
        fish.setid(233);
    }
}
```

4. (7分) 对于以下 C 函数片段，其运行时的活动记录按右图方式组织：

```

1) static int N;
2) void foo(int a) {
3)     short b[2020];
4)     int c[2 * N];
5)     unsigned long d = 996;
6)     c[2020] = a;
7)     b[1231] = 2019;
8) }

```



其中 b, d 的大小是确定的，可直接确定其在栈内的偏移。而 c 是一个动态数组，编译器并不能确定将需要多少存储空间。这种组织方式将 c 存到 b, d 的下方（如图所示），用 FP 保存分配 c 之前的栈顶指针（SP），当分配完 c 时，再对栈顶指针做相应的调整（无需保存内情向量）。

设全局变量 N 存放的内存位置为 0x4000，参数 a 保存在寄存器 A0。一个 long 类型占 4 个字节，一个 int 类型占 4 个字节，一个 short 类型占 2 个字节。由该函数生成的某种 32 位计算机上的目标代码如下（如果不熟悉汇编指令，右侧给出了详细的注释帮助你理解）：

```

foo_prologue:
    addi sp, sp, ①      # SP <- SP + ??? (allocate new stack frame)
    sw   ra, ②(sp)      # M[SP + offset_RA] <- RA (store old RA)
    sw   fp, ③(sp)      # M[SP + offset_FP] <- FP (store old FP)
    mv   fp, sp         # FP <- SP (set new frame point)

foo_body:
    lw   t0, 0x4000(zero) # T0 <- M[0x4000] (load `N`)
    slli t0, t0, 0x3      # T0 <- T0 * 8
    sub  sp, sp, t0       # SP <- SP - T0 (allocate `c` on stack)

    li   t0, 996          # T0 <- 996
    sw   t0, ④(fp)       # M[FP + offset_d] <- T0 (d = 996)

    sw   a0, ⑤(sp)       # M[SP + offset_c] <- A0 (c[2020] = a)

    li   t0, 2019         # T0 <- 2019
    sh   t0, ⑥(fp)       # M[FP + offset_b] <- T0 (b[1231] = 2019)

foo_epilogue:
    ⑦                    # (deallocate `c`)
    ⑧                    # ???
    ⑨                    # ???
    ⑩                    # ???
    jr   ra               # PC <- RA (return)

```

回答以下问题：

- (1) 试补全目标代码中缺失的偏移量，并参考函数调用起始阶段 foo_prologue 完成相应的函数调用收尾阶段 foo_epilogue（每空填一个数字或一条指令，意思正确即可，不需要考虑指令格式是否规范）。（5分）
- (2) 源程序的第 6 行存在一个缓冲区溢出漏洞，即 N 取太小时会导致 c 数组访问越界，此时可能会覆盖掉栈上的一些数据。覆盖一般的数据影响不大，而如果被覆盖

的数据恰好是函数返回地址，函数返回时就会跳转到错误的地址。通过精心构造该地址可使得程序执行流程发生更改，带来恶意代码执行等极其严重的后果。试问 N 取何值的时候恰好能覆盖该函数的返回地址？(2分)

二 (17 分) 给定某类表达式文法 $G[S]$:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow TG \\ G &\rightarrow +TG \mid \varepsilon \\ T &\rightarrow FM \\ M &\rightarrow *FM \mid \varepsilon \\ F &\rightarrow (E) \mid \underline{a} \end{aligned}$$

1. (3分) 针对文法 $G[S]$ ，下表给出各产生式右部文法符号串的 First 集合，各产生式左部非终结符的 Follow 集合。试填充其中空白表项（共3处）的内容：

$G[S]$ 的规则 r	$First(rhs(r))$	$Follow(lhs(r))$
$S \rightarrow E$	$\underline{a}, ($	$\#$
$E \rightarrow TG$	$\underline{a}, ($	$), \#$
$G \rightarrow +TG$	$+$	$), \#$
$G \rightarrow \varepsilon$	ε	此处不填
$T \rightarrow FM$		$+,), \#$
$M \rightarrow *FM$	$*$	
$M \rightarrow \varepsilon$	ε	此处不填
$F \rightarrow (E)$	$($	
$F \rightarrow \underline{a}$	$\underline{a}$	此处不填

表中， $rhs(r)$ 为产生式 r 右部的文法符号串， $lhs(r)$ 为产生式 r 左部的非终结符。

2. (4分) 以下是文法 $G[S]$ 的预测分析表，一些表项的内容未给出，试补充完成：

	$\underline{a}$	$+$	$*$	$($	$)$	$\#$
S	$S \rightarrow E$			$S \rightarrow E$		
E						
G		$G \rightarrow +TG$			$G \rightarrow \varepsilon$	$G \rightarrow \varepsilon$
T	$T \rightarrow FM$					
M		$M \rightarrow \varepsilon$			$M \rightarrow \varepsilon$	$M \rightarrow \varepsilon$
F	$F \rightarrow \underline{a}$			$F \rightarrow (E)$		

3. (2分) $G[S]$ 是否是 $LL(1)$ 文法？为什么？

4. (8分) 如下是以 $G[S]$ 为基础改造的一个 L 翻译模式：

$$\begin{aligned} S &\rightarrow \{ E.i := 0 \} E \{ S.s := E.s \} \\ E &\rightarrow \{ T.i := E.i \} T \{ G.i := T.s \} G \{ E.s := G.s \} \\ G &\rightarrow + \{ T.i := G.i \} T \{ G_1.i := T.s \} G_1 \{ G.s := G_1.s \} \\ G &\rightarrow \varepsilon \{ G.s := G.i \} \\ T &\rightarrow \{ F.i := T.i \} F \{ M.i := F.s \} M \{ T.s := M.s \} \\ M &\rightarrow * \{ F.i := -M.i \} F \{ M_1.i := F.s \} M_1 \{ M.s := M_1.s \} \\ M &\rightarrow \varepsilon \{ M.s := M.i \} \\ F &\rightarrow (\{ E.i := F.i \} E \{ F.s := E.s \} \\ F &\rightarrow \underline{a} \{ F.s := F.i + 1 \} \end{aligned}$$

针对该 L 翻译模式构造如下自上而下的递归下降（预测）翻译程序：

```

void ParseS ( )                // 主函数
{
    _____ ① _____;
    es := ParseE(ei);
    ss := es;
    return ss;
}

int ParseE(int i)
{
    _____ ② _____;
    ts := ParseT(ti);
    gi := ts;
    gs := ParseG(gi);
    es := gs;
    return es;
}

int ParseG(int i)
{
    switch (lookahead) {                // lookahead 为下一个输入符号
        case '+':
            MatchToken('+');
            ti := i;
            ts := ParseT(ti);
            g1i := ts;
            _____ ③ _____;
            gs := g1s;
            break;
        case _____ ④ _____:
            gs := i;
            break;
        default:
            printf("syntax error \n")
            exit(0);
    }
    return gs;
}

int ParseT(int i)
{
    .....                            //略
}

int ParseM(int i)
{
    .....                            //略
}

int ParseF(int i)
{
    .....                            //略
}

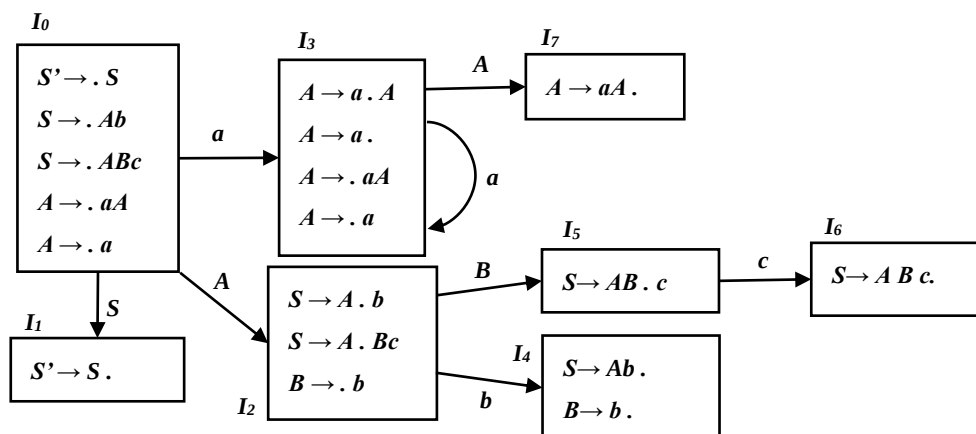
```

其中使用了与课程中所给的 MatchToken 函数。
 该翻译程序中 ①~④ 的部分未给出，试填写之。

三 (15 分) 给定文法 $G[S]$:

- (1) $S \rightarrow Ab$
- (2) $S \rightarrow ABc$
- (3) $A \rightarrow aA$
- (4) $A \rightarrow a$
- (5) $B \rightarrow b$

1. (4分) 下图是相应于 $G[S]$ 的 LR(0) 自动机:



文法 $G[S]$ 不是 LR(0) 文法。试指出 $G[S]$ 的 LR(0) 自动机中存在哪些冲突的状态? 并指出这些状态的冲突类别, 即是移进-归约冲突还是归约-归约冲突?

2. (5分) 完善 $G[S]$ 的 SLR (1) 分析表, 并说明 $G[S]$ 是 SLR (1) 文法:

状态	ACTION				GOTO		
	a	b	c	#	S	A	B
0	s3				1		
1				acc			
2		s4					5
3							
4							
5							
6							
7		r3					

3. (2分) 基于上面的 SLR(1) 分析表进行 LR 分析, 若分析过程中设有符号栈, 则栈中存放的整个符号串 (初始栈顶符号 # 除外) 对应于文法 $G[S]$ 的 _____。

- A. 某个活前缀 B. 某个可归约串 C. 某个句柄 D. 某个 LR(0) 项目

4. (2分) 基于上面的 SLR(1) 分析表进行 LR 分析, 若处于某个正常状态 (未出错状态) 时在栈顶已形成句柄, 且当前所面临的输入符号是 b , 那么这样的状态可能有 _____ 个。

- A. 1个 B. 2个 C. 3个 D. 4个

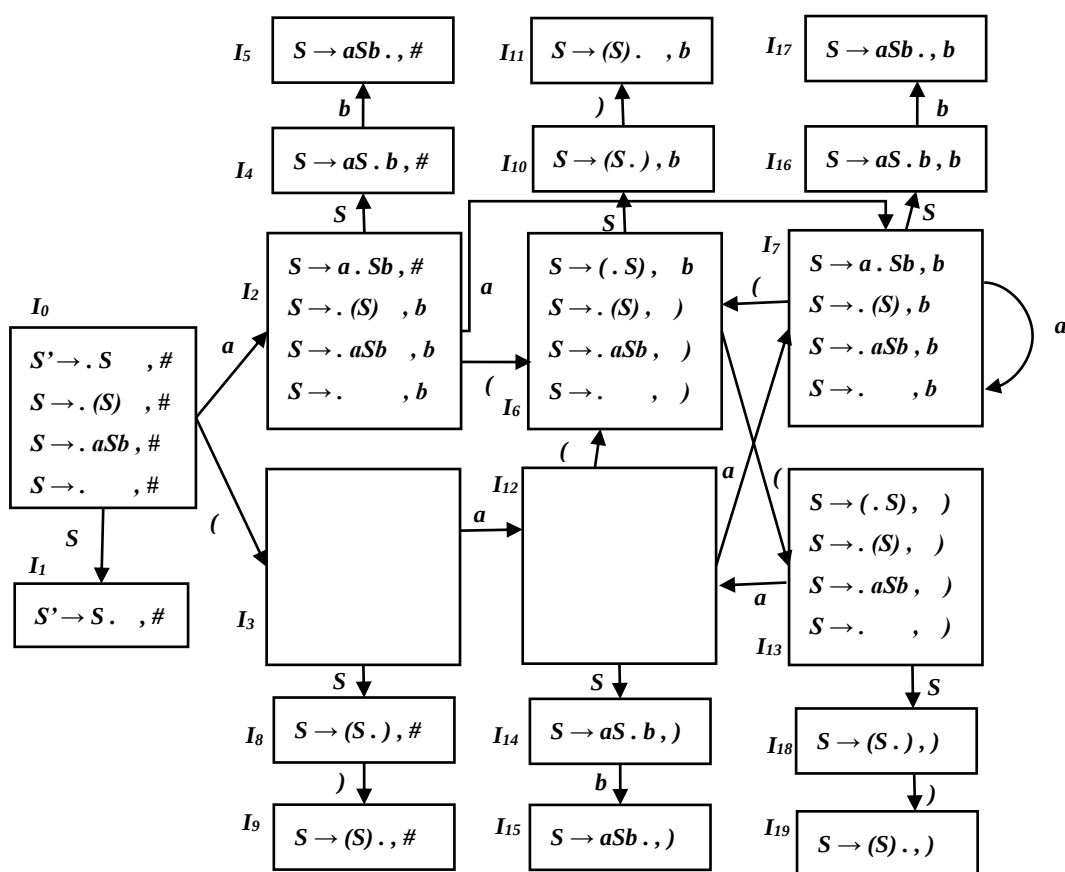
5. (2分) 基于上面的 SLR(1) 分析表进行 LR 分析, 若处于某个出错状态时所面临的输入符号是 b , 那么这样的状态可能有 _____ 个。

- A. 1个 B. 2个 C. 3个 D. 4个

四 (7 分) 给定文法 $G[S]$:

- (1) $S \rightarrow (S)$
 (2) $S \rightarrow aSb$
 (3) $S \rightarrow \varepsilon$

1. (5分) 下图是相应于 $G[S]$ 的 LR(1) 自动机, 但自动机的状态和转移边并未完善, 试补齐之 (分别给出状态 I_3 和 I_{12} 对应的项目集, 并补充自动机中缺少的状态转移关系)。



2. (2分) 基于上面的 LR(1) 自动机进行语法分析, 若输入串是 $a(b)\#$, 则到达第一个出错状态时, 最不可能的报错信息是 _____。

- A. 多 “(” B. 多 “)” C. 缺 “(” D. 缺 “)”

五 (10 分) 给定用来描述某种命题逻辑公式的文法 $G[S]$:

- (1) $S \rightarrow P$
- (2) $P \rightarrow P P \wedge$
- (3) $P \rightarrow P P \vee$
- (4) $P \rightarrow P \neg$
- (5) $P \rightarrow \underline{id}$

其中，终结符 $\wedge$ 、 $\vee$ 、 $\neg$ 分别代表三种逻辑连接词，id 是标识符（在计算逻辑公式真值的时候，代表一个命题变元）。

如下是以 $G[S]$ 为基础文法的一个 S 翻译模式:

- | | |
|------------------------------------|----------------------------------|
| (1) $S \rightarrow P$ | $\{ \text{print}(P.s) \}$ |
| (2) $P \rightarrow P_1 P_2 \wedge$ | $\{ P.s := f_1(P_1.s, P_2.s) \}$ |
| (3) $P \rightarrow P_1 P_2 \vee$ | $\{ P.s := f_2(P_1.s, P_2.s) \}$ |
| (4) $P \rightarrow P_1 \neg$ | $\{ P.s := f_3(P_1.s) \}$ |
| (5) $P \rightarrow \underline{id}$ | $\{ P.s := g(\underline{id}) \}$ |

其中, $print$ 为打印函数, f_1, f_2, f_3, g 为其他语义函数。

1. (4分) 如果在 LR 分析过程中根据这一翻译模式进行自下而上语义计算, 试写出在按每个产生式归约时语义处理的一个代码片断 (设语义栈由向量 *val* 表示, 归约前栈顶位置为 *top*, 终结符不对应语义值, 而每个非终结符的综合属性都只对应一个语义值, 本题中可用 *val[i].s* 表示; 不用考虑对 *top* 的维护)。

2. (2分) 指定语义函数

```
f1(x, y) = if (x = 1 and y = 1) then 1
           else if (x = 0 and y = 0) then 1
           else 0
```

```
f2(x, y) = if (x = 0 and y = 1) then 1
           else if (x = 1 and y = 0) then 1
           else 0
```

$$f_3(x) = \text{if } x = 0 \text{ then } 1 \text{ else } 0$$
$$g(a) = 1 \qquad g(b) = 1 \qquad g(c) = 0$$

那么对于合法输入串 $ab \neg \wedge abc \wedge \vee a \neg cb \wedge \vee \vee \vee$ ，翻译模式的语义计算结果是什么？（即 *print* 的打印结果，你无需给出计算过程）

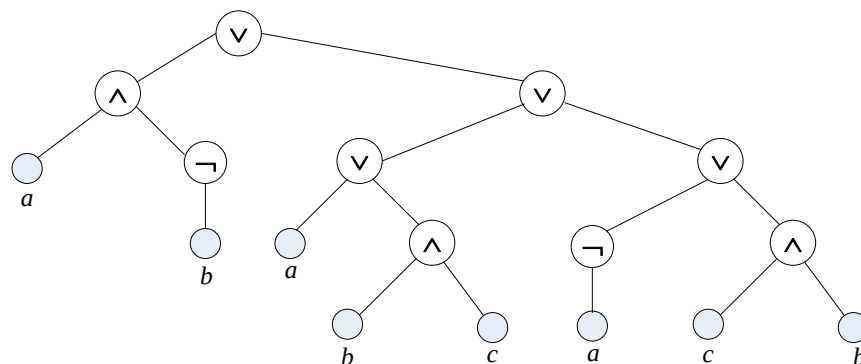
3. (4分) 文法 $G[S]$ 可用于识别后缀形式（逆波兰式）的命题逻辑公式。输入串

$$ab \neg \wedge abc \wedge \vee a \neg cb \wedge \vee \vee \vee$$

对应于中缀式

$$(a \wedge \neg b) \vee ((a \vee (b \wedge c)) \vee (\neg a \vee (c \wedge b))) \quad ,$$

以下是该公式对应的表达式树:

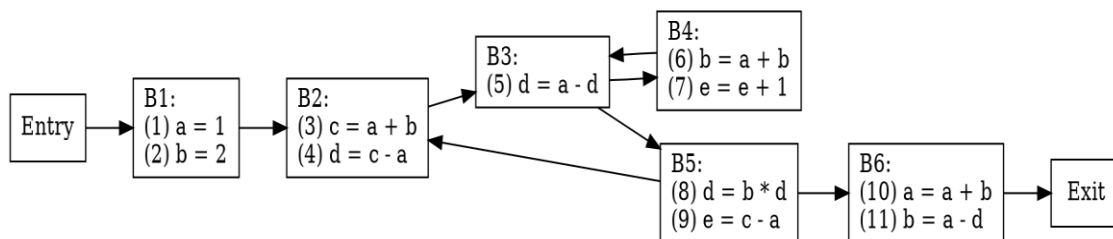


如果在翻译模式中, $print(P.s)$ 的含义是打印出由 P 识别的逻辑公式对应的表达式树根节点对应的 Ershov 数 (Ershov number), 请给出四个语义函数 f_1 , f_2 , f_3, g 的具体定义。(提示: f_1 与 f_2 定义相同)

六 (11 分)

数据流分析在编译器中后端的应用非常广泛, 除了课堂上讲到的两种数据流分析 (活跃变量分析和到达-定值分析) 之外还有很多, 它们的具体定义和功能不尽相同, 但是大致过程都是对数据流方程的求解, 且求解方式都是类似的。一般来说, 只要给出数据流方程, 以及迭代求解方程的初始值, 就可以采用统一的迭代算法来求解方程。下面的问题中, 我们都不会给出求解方程的迭代算法, 请大家自己根据课上所学知识进行类比。

在下面的两个小题中都使用这个数据流图来进行计算。流图中的 **Entry** 和 **Exit** 是两个虚拟的基本块, **Entry** 是程序的唯一入口, **Exit** 是程序的唯一出口; 除此以外, 你可以认为它们就是两个不包含任何指令的普通基本块。流图省略了基本块间的跳转语句, 它们是否存在对本题的作答无影响。



1. (4分)

课堂上定义了数据流图中的“支配节点”的概念。为了计算数据流图中每个基本块的支配节点集, 我们可以应用数据流分析的方法。定义数据流方程如下:

$$\begin{aligned} Out(B) &= In(B) \cup \{B\} \\ In(B) &= \bigcap Out(B'), \quad B' \in pred(B) \end{aligned}$$

其中 $pred(B)$ 为 B 的前驱基本块的集合。

方程的初值为:

$$\begin{aligned} In(B) &= \emptyset \\ Out(B) &= U, \quad B \neq Entry \\ Out(Entry) &= \{Entry\} \end{aligned}$$

其中 $\emptyset$ 为空集; U 为全集, 即所有基本块组成的集合。

试填空：

(a) 这个数据流分析属于 _____ 数据流分析。（填：前向/后向）

(b) $In(B_4) = ______$ ， $Out(B_5) = ______$ 。

(c) 每个基本块的支配节点集就是它的 _____ 集合。（填：In/Out）。

2. (7分)

实际编译器中，一种常见且重要的优化手段是部分冗余消除(PRE, Partial Redundancy Elimination)。为了实现 PRE，需要使用多种数据流分析，其中一种称为预期执行的表达式分析 (Anticipated Expression Analysis)。

对于一个程序点，如果从这一点处开始的所有路径都会计算某个表达式的值，并且从这一点到表达式的计算点的路径上，这个表达式的所有操作数都没有被重新定值，则说这个表达式在这一点处被预期执行。预期执行的表达式分析就是希望找出每个程序点处预期执行的表达式的集合。

定义数据流方程如下：

$$\begin{aligned} In(B) &= Gen(B) \cup (Out(B) - Kill(B)) \\ Out(B) &= \bigcap In(B'), B' \in succ(B) \end{aligned}$$

其中 $succ(B)$ 为 B 的后继基本块的集合。

方程的初值为：

$$\begin{aligned} Out(B) &= \emptyset \\ In(B) &= U, B \neq Exit \\ In(Exit) &= \emptyset \end{aligned}$$

其中 $\emptyset$ 为空集； U 为全集，即程序中出现过的所有表达式的集合。在本题中，

$$U = \{ a + b, c - a, a - d, e + 1, b * d \}$$

方程中涉及到的四个集合的定义如下：

- **Gen(B)**: B 中计算过，且从计算点到 B 的入口都没有操作数被重新定值的表达式的集合；对于一条语句 $x = x + y$ ，应该把它看作先计算 $x + y$ ，再对 x 定值
- **Kill(B)**: 以 B 中定值的任一变量作为操作数，且在程序中出现过的表达式的集合
- **In(B)**: B 入口处预期执行的表达式的集合
- **Out(B)**: B 出口处预期执行的表达式的集合

试填空：

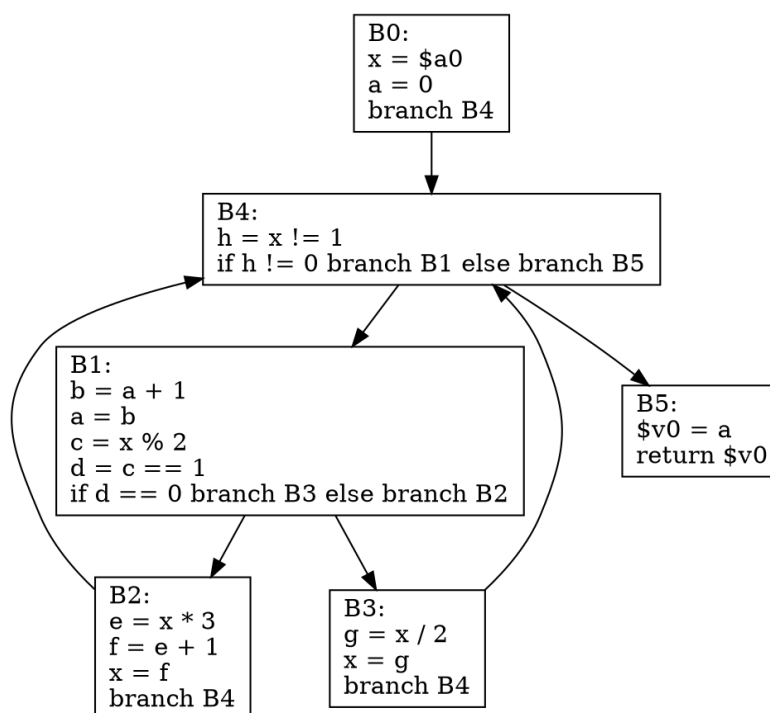
(a) 这个数据流分析属于 _____ 数据流分析。（填：前向/后向）

(b) $Gen(B_2) = ______$ ， $Kill(B_3) = ______$ ， $In(B_4) = ______$ ， $Out(B_1) = ______$ 。

(c) 在 PRE 中，预期执行的表达式分析可以用来确定哪些位置可以放置一个表达式。可以放置的条件是：这个表达式在这一点处被预期执行（否则可能会导致原程序中没有执行过的计算在优化后的程序中执行了）。据此，判断 B_3 中 (5) 后面是否可以放置表达式 $a + b$ ：____， B_5 中 (8) 和 (9) 之间是否可以放置表达式 $a - d$ ：____。（填：是/否）

七 (4分)

假设经过指令选择，生成了如下的包含虚拟寄存器和物理寄存器的伪汇编代码：



伪汇编代码的规范如下：

- (1) 其中以\$开头的标识符都是物理寄存器的名字，其它标识符都是虚拟寄存器的名字；
- (2) B0 为入口基本块；
- (3) 指令中没有隐含的数据依赖关系，即每条指令所读取和写入的物理/虚拟寄存器都已经明确的表示在指令中了；
- (4) 指令的格式与课堂上和实验中使用的 TAC 的格式类似，因此不再解释每条语句的具体语义；

对这段代码进行基于图着色的寄存器分配。要求构造出干涉图（寄存器相干图），并在干涉图上给出一个合法的着色，你的着色只能使用 \$v0, \$a0 和 \$a1 这三个物理寄存器。

我们要求拓展课堂上描述的相关算法，让干涉图的节点由物理寄存器(不包括 \$a1，因为程序中没有使用过它)和虚拟寄存器组成。物理寄存器是“预着色”的，即在着色算法执行之前，每个物理寄存器就已经拥有了自己的颜色，且着色算法不能再赋予它别的颜色。除了这一点特殊之处外，物理寄存器应该看作与虚拟寄存器没有区别。

八 (20分)

1. (6分) 以下是一个 S-翻译模式片断，描述了某小语言部分特性的类型检查工作。

其中，*type* 属性以及类型表达式 *ok*、*type_error*、*bool*、以及所涉及到的语义函数（如 *lookup_type*）等的含义与讲稿中一致；加黑的单词为保留字；声明语句、布尔表达式以及算术表达式相关的部分已全部略去。额外地，我们提醒：*lookup_type* 对于未声明变量，返回 *nil* 而非 *type_error*。

本题中，我们不考虑语法分析的歧义性，即可以认为给出的文法是已经语法分析好的抽象语法。

$$\begin{aligned}
 P &\rightarrow D ; S \quad \{ P.type := \text{if } D.type = \text{ok and } S.type = \text{ok then ok else type_error} \} \\
 S &\rightarrow \text{if } E \text{ then } S_1 \text{ end} \quad \{ S.type := \text{if } E.type = \text{bool then } S_1.type \text{ else type_error} \} \\
 S &\rightarrow \text{while } E \text{ begin } S_1 \text{ end} \quad \{ S.type := \text{if } E.type = \text{bool then } S_1.type \text{ else type_error} \} \\
 S &\rightarrow S_1 ; S_2 \quad \{ S.type := \text{if } S_1.type = \text{ok then } S_2.type \text{ else type_error} \} \\
 S &\rightarrow id := E \quad \{ S.type := \text{if lookup_type}(id.entry) = E.type \text{ then ok else type_error} \} \\
 D &\rightarrow \dots \quad \{ \text{省略与声明语句相关的全部规则} \} \\
 E &\rightarrow \dots \quad \{ \text{省略与整数/布尔表达式相关的全部规则} \}
 \end{aligned}$$

下面叙述本小题的要求：

在基础文法中增加对“批量赋值” (bulk assignment) 的支持，原来的 $id := E$ 赋值语句变为批量赋值语句：

$$id(, id)^* := E(, E)^*$$

其中 $*$ 表示 Kleene 星闭包（即符号串出现零次或多次）。

批量赋值语句是合法的，当且仅当 $:=$ 左侧每个 id 均在 D 中出现（使用 $lookup_type$ 检查），并且右侧每个 E 都通过类型检查。而且还要求， $:=$ 左边 id 数目和右边 E 数目相等，且对应元素类型（ id 的声明类型和 E 的类型）相同。

试在上述 S-翻译模式片段的基础上，新增对批量赋值语句进行类型检查的片段，填写以下空缺的部分。（提示：需要增加一个综合属性）

$$\begin{aligned}
 L &\rightarrow id \quad \{ \underline{\quad ① \quad} \} \\
 L &\rightarrow id, L_1 \quad \{ \underline{\quad ② \quad} \} \\
 R &\rightarrow E \quad \{ \underline{\quad ③ \quad} \} \\
 R &\rightarrow E, R_1 \quad \{ \underline{\quad ④ \quad} \} \\
 S &\rightarrow \underline{\quad ⑤ \quad} \quad \{ \underline{\quad ⑥ \quad} \}
 \end{aligned}$$

除了课程所讲以及上文例子中有的属性动作 / 语义函数外，你还可以在属性动作中使用如下操作：

- $[x]$ 初始化一个只有 x 一个元素的列表，类似讲义中的 *makelist*。
- 加号表示列表拼接。例如 $[x] + [z, w]$ 的结果是 $[x, z, w]$ ，类似讲义中的 *merge*。
- $list1 = list2$ 判断列表相等。仅当长度相等，且对应位置的元素都相等的时候，两个列表才相等。
- 列表的长度使用 $len(list)$ 获得。
- 列表的第 i 个元素使用 $list[i]$ 获得，注意：列表下标从 0 开始。

请不要使用其他操作。

2. (7 分) 以下是一个 L-翻译模式片断，可以对原语言的程序产生相应的 TAC 语句序列：

```

P → D ; { S.next := newlabel } S { gen(S.next “:”) }

S → if { E.true := newlabel ; E.false := S.next } E then
    { S1.next := S.next } S1 end
    { S.code := E.code || gen(E.true “:”) || S1.code }

S → while { E.true := newlabel; E.false := S.next } E begin
    { S1.next := newlabel } S1 end
    { S.code := gen(S1.next “:”) || E.code ||
      gen(E.true “:”) || S1.code || gen(“goto” S1.next) }

S → { S1.next := newlabel } S1 ;
    { S2.next := S.next } S2
    { S.code := S1.code || gen(S1.next “:”) || S2.code }

S → id := E
    { S.code := E.code || gen(id.place “:=” E.place) }

D → ... { 省略与声明语句相关的全部规则 }

E → ... { 省略与布尔表达式相关的全部规则 }

```

其中，属性 $S.code$ 、 $E.code$ 、 $S.next$ 、 $E.true$ 、 $E.false$ 、语义函数 $newlabel$ 、 $gen()$ 以及所涉及到的 TAC 语句与讲稿中一致。语义函数 $newtemp$ 的作用是在符号表中新建一个从未使用过的名字，并返回该名字的存储位置；语义函数 gen 的结果是生成一条 TAC 语句；“||”表示 TAC 语句序列的拼接。所有符号的 $place$ 综合属性也均与讲稿中一致。

下面叙述本小题的要求：

考虑增加批量赋值的 TAC 生成功能。我们要求，批量赋值的几个赋值是一次性完成的。例如 $a, b := b, a$ 可以交换 a 和 b 的值。

试在上述 L-翻译模式片段的基础上，增加针对批量赋值的 TAC 代码生成片段，填写以下空缺的部分。要求不改变 L-翻译模式的特征。假设输入已经通过第 1 小题的类型检查。

```

L → id { ① }
L → id, L1 { ② }
R → E { ③ }
R → E, R1 { ④ }
S → ⑤ { ⑥ }

```

除了第 1 小题中允许的操作外，你可以在属性动作中：

- 使用形如 “for $i = A$ to B do 循环体 end” 的循环，其中 i 的值可以取到循环上下界 A, B 。如果 $B < A$ 那么循环等于空语句（但 $B=A$ 的时候不是）。

- 用 "" 表示空 TAC 串。

请不要使用其他操作。

3. (7 分) 我们不妨允许批量赋值语句中, $:=$ 左侧的变量个数比右侧的表达式个数多, 如 $a, b, c := 1$ 。对于多出来的变量 (称之为“多余变量”), 我们按照它们的类型是 *bool* 还是 *int* 给一个默认值。

因此我们加入新的保留字 **default**, 并且加入语法 $S \rightarrow \text{begin } S_1 \text{ end default } E_1, E_2$ 表示 S_1 中所有的“多余变量”, 如果是 *bool* 类型, 那么默认值是 E_1 , 如果是 *int* 类型, 那么默认值是 E_2 ; 假设没有其他类型。 E_1, E_2 中可以有变量, 但它们的值在 **begin** 的时候就被求得 (按照先 E_1 后 E_2 的顺序), 无论 **begin ... end** 中对 E_1, E_2 使用的变量如何修改, E_1 和 E_2 的值都不变。

例如, 假设变量 x, y, z 均为 *int* 类型, 而变量 a, b, c 均为 *bool* 类型。下面代码

begin $b := \text{true}; x, y, a := 1$ **end default** $a \ \&\& \ b, 0$

其中有两个多余变量: y 和 a 。这段代码等价于

$e1 := a \ \&\& \ b; e2 := 0; b := \text{true}; x, y, a := 1, e2, e1$

提示: 因为多余变量的值要到后来看到 **default** 才知道, 所以在批量赋值时我们还不知道它们的默认值。因此我们需要采用“代码回填”技术, 将额外变量的 *place* 先留空, 等看到 **default** 中再填空。使用 $\text{gen}(id.place \text{ “:= ”})$ 表示 (注意是 *gen* 而非 *emit*): id 是额外变量, 但它的 *place* (即 “_”) 还不确定, 具体是什么由在后面看到 **default** 的时候回填。使用 $\text{backpatch}(\text{list}, \text{place})$ 表示回填的过程。其中 *list* 列表包含一系列如上待回填的 “:= _”, 而 *place* 回填的值。例如, $\text{backpatch}([id.place], E.place)$ 将会回填 $\text{gen}(id.place \text{ “:= ”})$ 中的空白, 将它变成 $\text{gen}(id.place \text{ “:= ” } E.place)$ 。

下面叙述本小题的要求:

增加针对批量赋值默认值的 TAC 代码生成片段, 填写以下空缺的部分。要求不改变 L-翻译模式的特征。假设输入已经通过第 1 小题的类型检查, 且保证 E_1 类型是 *bool*, E_2 类型是 *int*。

你可以使用前面两个小题中对 L 和 R 定义的所有属性。本题答案可能会和第 2 小题答案重复, 为方便, 请将所有需要“在此处复制第 2 小题的对应的那个空”之处用 “...” 表示。

$P \rightarrow D; \{ S.next := \text{newlabel}; S.eint := []; S.ebool := [] \}$

$S \{ \text{gen}(S.next \text{ “:= ”}) \}$

$S \rightarrow \text{if } \{ E.true := \text{newlabel}; E.false := S.next \} E \text{ then}$

$\{ S_1.next := S.next \} S_1 \text{ end}$

$\{ S.code := E.code \parallel \text{gen}(E.true \text{ “:= ”}) \parallel S_1.code ;$

$S.eint := S_1.eint; S.ebool := S_1.ebool \}$

$S \rightarrow \text{while } \{ E.true := \text{newlabel}; E.false := S.next \} E \text{ begin}$

$\{ S_1.next := \text{newlabel} \} S_1 \text{ end}$

$\{ S.code := \text{gen}(S_1.next \text{ “:= ”}) \parallel E.code \parallel$

$\text{gen}(E.true \text{ “:= ”}) \parallel S_1.code \parallel \text{gen}(\text{“goto” } S_1.next);$

$S.eint := S_1.eint; S.ebool := S_1.ebool \}$

$$\begin{aligned}
S &\rightarrow \{ S_1.next := newlabel \} S_1 ; \\
&\quad \{ S_2.next := S.next \} S_2 \\
&\quad \{ S.code := S_1.code \parallel gen(S_1.next \text{ “:”}) \parallel S_2.code \\
&\quad \quad S.eint := S_1.eint + S_2.eint ; S.ebool := S_1.ebool + S_2.ebool \} \\
L &\rightarrow id \{ \dots \} \\
L &\rightarrow id, L_1 \{ \dots \} \\
R &\rightarrow E \{ \dots \} \\
R &\rightarrow E, R_1 \{ \dots \} \\
S &\rightarrow \dots \{ \quad \text{①} \quad \} \\
S &\rightarrow \textit{begin } S_1 \textit{ end default } E_1, E_2 \quad \{ \quad \text{②} \quad \} \\
D &\rightarrow \dots \{ \text{省略与声明语句相关的全部规则} \} \\
E &\rightarrow \dots \{ \text{省略与布尔表达式相关的全部规则} \}
\end{aligned}$$

