

《编译原理》书面作业

截止日期：2025 年 12 月 30 日

若发现问题，或有任何想法（改进题目、调整任务量等等），请及时联系助教

Q1. 对于以下 C 语言函数片段，其运行时活动记录按照右图方式组织。

```
1 static int N, L;
2 int foo() {
3     int a[100];
4     int s = 3;
5     int b[N];
6     register int i;
7     for (i = 0; i < L; i++) {
8         b[i] = i * i;
9     }
10    a[10] = b[5];
11    return s;
12 }
```



其中， i 保存在寄存器 $T0$ 中（而非栈上）。本题的活动记录中首先保存返回地址 RA 与旧帧指针 FP ，之后按照定义顺序保存静态数组 a 和变量 s ，最后保存动态数组 b 。其中用 FP 保存分配动态数组（即 b ）之前的栈顶指针 SP 。本题的活动记录中数组中的元素根据编号由低地址向高地址存储在栈中，例如 $b[0]$ 存储在 SP 所指位置。假设全局变量 N 存放的内存位置为 $0x4000$ 。字长为 4 字节。

1. 该函数的栈帧（过程活动记录）总大小为多少字节？写成关于 N 的表达式。
2. 在进入函数时，需要先为所有确定大小的变量（数组 a 和变量 s ）和旧 FP 、 RA 分配栈空间，之后再为 b 动态分配空间。生成的 RISC-V 目标代码如下所示，请补全其中的偏移量。

```
1 func_prologue:
2 addi sp, sp, ----- # SP <- SP + ???
3 sw ra, ----- # Mem[SP + ???] <- RA
4 sw fp, ----- # Mem[SP + ???] <- FP
5 mv fp, sp # FP <- SP
6 lw a0, 0x4000(zero) # A0 <- Mem[0x4000]
7 slli a0, a0, 2 # A0 <- A0 * 4
8 sub sp, sp, a0 # SP <- SP - A0
```

3. 函数中 `a[10] = b[5]`; 这条语句对应如下的目标代码, 请补全其中的偏移量。

```
1 | lw  a0, _____(sp)  # A0 <- Mem[SP + ???]
2 | sw  a0, _____(fp)  # Mem[FP + ???] <- A0
```

4. 源程序中存在一个缓冲区溢出漏洞, 如果 L 的值大于 N 时会导致 `b` 数组访问越界, 此时可能会覆盖掉栈上的一些数据。覆盖数据可能会导致函数返回错误的值, 更严重的是, 如果被覆盖的数据恰好是返回地址, 函数返回时就会跳转到错误的地址。试问 L 取值大于等于多少时会覆盖函数的返回值? 取值大于等于多少时会覆盖函数的返回地址? 写成关于 N 的表达式。

参考答案:

1. $4N + 412$ 字节。

2. 如下所示。

```
1 | func_prologue:
2 | addi  sp, sp, -412
3 | sw    ra, 408
4 | sw    fp, 404
5 | mv    fp, sp
6 | lw    a0, 0x4000(zero)
7 | slli  a0, a0, 2
8 | sub   sp, sp, a0
```

3. 如下所示。

```
1 | lw  a0, 20(sp)
2 | sw  a0, 44(fp)
```

4. $L \geq N + 1$ 时覆盖返回值, $L \geq N + 103$ 时覆盖返回地址。

Q2. 以下是某简单语言的一段代码。其语法基本与 C 语言一致，但支持嵌套的函数声明，且允许全局语句（类似 Python）。简洁起见，我们目前讨论的函数都没有参数和返回值，每一个函数对应一个静态作用域，程序执行也遵循静态作用域规则。过程活动记录包括控制信息（静态链 SL、动态链 DL、返回地址 RA）和当前作用域声明的变量。右图是程序执行到 `foo()` 被第二次调用时的运行栈状态，其中变量名代表该位置存储了相应变量的值，左侧为单元编号（注意，通常来讲地址从高向低增长，但此处编号从低到高增长）。

		0	0 (SL)
		1	0 (DL)
		2	RA
		3	a
		4	b
1	<code>int a, b;</code>	5	0 (SL)
2	<code>void foo() {</code>	6	0 (DL)
3	<code>int x;</code>	7	RA
4	<code>void bar() {</code>	8	baz.x
5	<code>int x, a;</code>	9	0 (SL)
6	<code>a = 3;</code>	10	5 (DL)
7	<code>if (a > b) baz();</code>	11	RA
8	<code>x = ...;</code>	12	foo.x
9	<code>}</code>	13	9 (SL)
10	<code>bar();</code>	14	9 (DL)
11	<code>x = ...;</code>	15	RA
12	<code>}</code>	16	bar.x
13	<code>void baz() {</code>	17	bar.a
14	<code>int x;</code>	18	_____(SL)
15	<code>if (a < b) foo();</code>	19	_____(DL)
16	<code>x = ...;</code>	20	_____
17	<code>}</code>	21	_____
18	<code>a = 1;</code>	22	_____(SL)
19	<code>b = 2;</code>	23	_____(DL)
20	<code>baz();</code>	24	RA
		25	x

1. 补全运行栈中缺失的部分。
2. 采用 Display 表来代替静态链。假设采用只在活动记录保存一个 Display 表项的方法，且该表项占居图中 SL 的位置。试指出当前运行状态下 Display 表的内容，以及各活动记录中所保存的 Display 表项的内容（即图中所有 SL 位置的新内容）
3. 若采用动态作用域规则，该程序的执行效果与之前有何不同？（指出第一次控制流不同的时机即可）

参考答案：

1. 单元 18-23 的内容依次为：0, 13, RA, baz.x, 0, 18。
2. 当前 Display 表为： $D[0] = 0, D[1] = 22, D[2] = 13$ 。活动记录中单元 0, 5, 9, 13, 18, 22 中的内容分别为：无效, 无效, 5, 无效, 9, 18。
3. 在第二次执行到第 15 行时，若是静态作用域规则，则 $a = 1, b = 2$ ，因此会再次调用 `foo()`；若是动态作用域规则，则 $a = 3, b = 2$ ，因此不会调用 `foo()`。

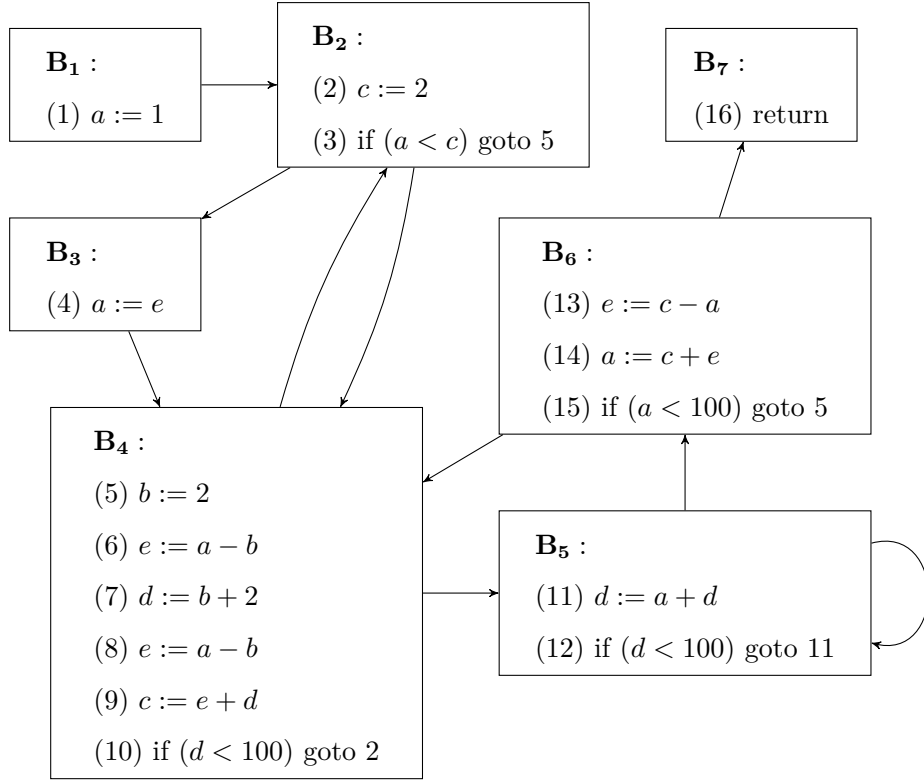
Q3. 考虑下面的中间代码。

1		a := 1	9		c := e + d
2		c := 2	10		if (d < 100) goto 2
3		if (a < c) goto 5	11		d := a + d
4		a := e	12		if (d < 100) goto 11
5		b := 2	13		e := c - a
6		e := a - b	14		a := c + e
7		d := b + 2	15		if (a < 100) goto 5
8		e := a - b	16		return

1. 请给出该代码的基本块划分和流图。使用 $B_1, B_2, \dots$ 命名基本块，其中基本块编号与块内第一条代码的行号顺序一致。
2. 请写出该流图中，包含代码行 5 的基本块的支配节点集合、始于此基本块的回边、以及基于该回边的自然循环。请按第 1 题中的命名方式命名基本块。
3. 采用迭代求解数据流方程的方法对活跃变量数据流信息进行分析，写出每个基本块的 Def、LiveUse、LiveIn、LiveOut 集合。假设出口基本块的 LiveOut 为空集。
4. 采用迭代求解数据流方程的方法对到达-定值数据流信息进行分析，写出每个基本块的 Gen、Kill、In、Out 集合。假设入口基本块的 In 为空集。
5. 指出在该流图范围内，变量 a 在语句 11 处的 UD 链。
6. 指出在该流图范围内，变量 c 在语句 2 处的 DU 链。
7. 画出寄存器相干图。
8. 按照课堂上介绍的图着色全局寄存器分配算法，若要保证图着色过程中不会出现将寄存器泄漏到内存中的情形，那么可供分配的物理寄存器的最小数目分别是多少？请给出一种合理的着色方案。

参考答案：

1. 如下所示。



2. B_4 的支配节点集合为 $\{B_1, B_2, B_4\}$ ，始于 B_4 的回边为 (B_4, B_2) ，基于该回边的自然循环包括 $\{B_2, B_3, B_4, B_5, B_6\}$ 。
3. 如下所示。

	Def	LiveUse	LiveIn	LiveOut
B_1	$\{a\}$	$\emptyset$	$\{e\}$	$\{a, e\}$
B_2	$\{c\}$	$\{a\}$	$\{a, e\}$	$\{a, e\}$
B_3	$\{a\}$	$\{e\}$	$\{e\}$	$\{a\}$
B_4	$\{b, c, d, e\}$	$\{a\}$	$\{a\}$	$\{a, c, d, e\}$
B_5	$\emptyset$	$\{a, d\}$	$\{a, c, d\}$	$\{a, c, d\}$
B_6	$\{e\}$	$\{a, c\}$	$\{a, c\}$	$\{a\}$
B_7	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

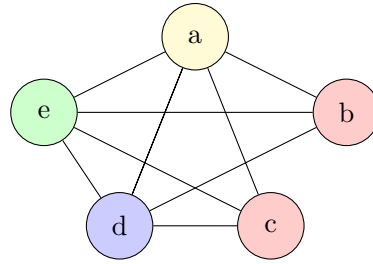
4. 如下所示。 $(B_6$ 的 $Kill$ 集合可以包含 6，也可以不包含，不会影响 In 和 Out 的计算)

	Gen	Kill	In	Out
B_1	$\{1\}$	$\emptyset$	$\emptyset$	$\{1\}$
B_2	$\{2\}$	$\{9\}$	$\{1, 4, 5, 7, 8, 9, 14\}$	$\{1, 2, 4, 5, 7, 8, 14\}$
B_3	$\{4\}$	$\{1, 14\}$	$\{1, 2, 4, 5, 7, 8, 14\}$	$\{2, 4, 5, 7, 8\}$
B_4	$\{5, 7, 8, 9\}$	$\{2, 11, 13\}$	$\{1, 2, 4, 5, 7, 8, 9, 11, 13, 14\}$	$\{1, 4, 5, 7, 8, 9, 14\}$
B_5	$\{11\}$	$\{7\}$	$\{1, 4, 5, 7, 8, 9, 11, 14\}$	$\{1, 4, 5, 8, 9, 11, 14\}$
B_6	$\{13, 14\}$	$\{1, 4, 8\}$	$\{1, 4, 5, 8, 9, 11, 14\}$	$\{5, 9, 11, 13, 14\}$
B_7	$\emptyset$	$\emptyset$	$\{5, 9, 11, 13, 14\}$	$\{5, 9, 11, 13, 14\}$

5. UD 链 $\{1, 4, 14\}$ 。

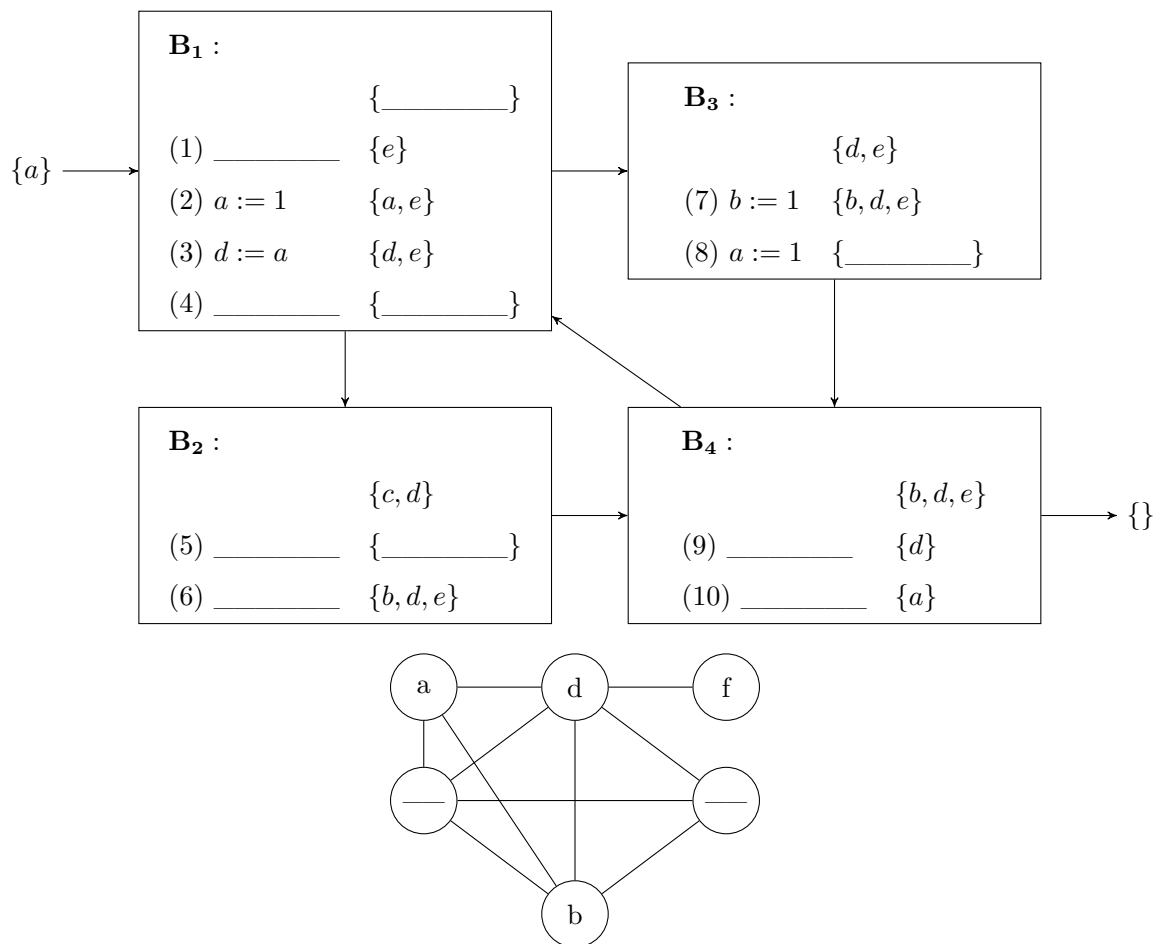
6. DU 链 {3}。

7. 如下所示。

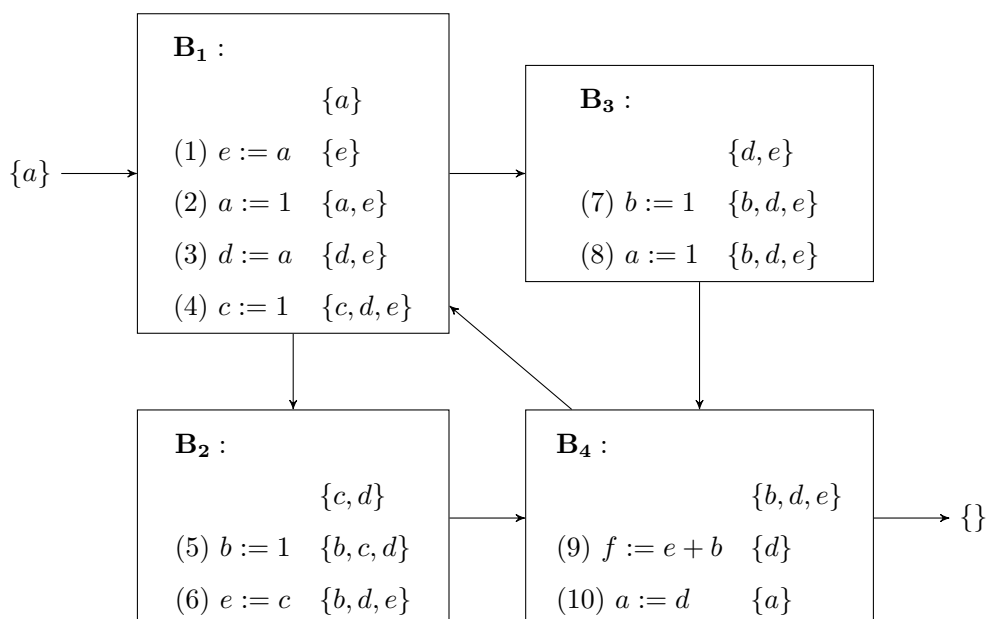


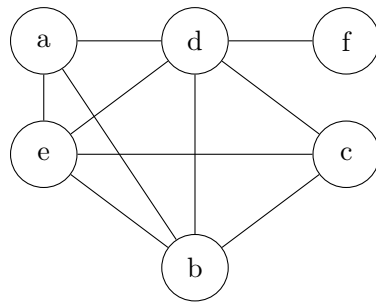
8. 可供分配的最小寄存器数目是 4。可行的染色方案已在上一问相干图中给出。

Q4. 下面给出了一个不完整的流图和寄存器相干图，流图中标注了每条语句执行过后的活跃变量列表，请补全图中缺失的部分。其中，缺失的语句请从 $x := 1$ 、 $x := y$ 、 $x := y + z$ 三种形式中选择。

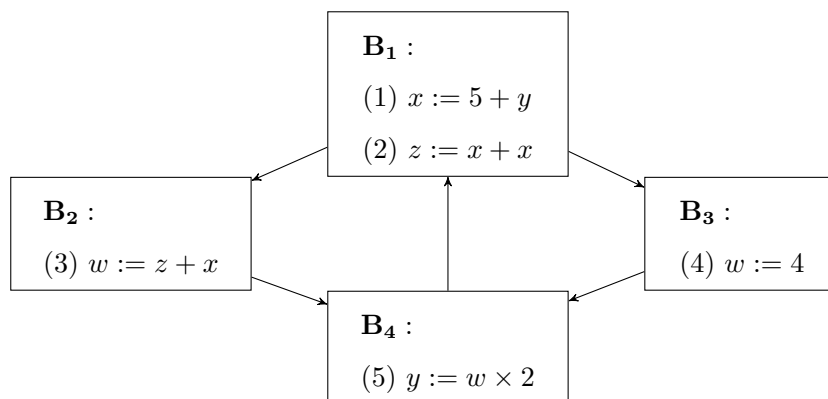


参考答案:





Q5. 在课堂上,我们学习了许多种数据流。现在,我们想设计一种新的数据流“Always-Defined”,帮助我们分析一个变量 x 在被语句 d 使用时是否一定被定义过。比如考虑下面的控制流图, x 在语句 2 和 3 中总是被定义的, w 在语句 5 中也总是被定义的,但 y 在语句 1 中并不总是被定义的。



为了实现这样的“Always-Defined”分析,我们用函数 $AD(d, x, OUT)$ 表示变量 x 在语句 d 刚好执行后是否一定被定义, $AD(d, x, IN)$ 表示变量 x 在语句 d 即将执行前是否一定被定义。 $AD(\cdot, \cdot, \cdot)$ 函数只有两个取值: True 和 False。

1. 应该用何值初始化 $AD(\cdot, \cdot, \cdot)$ 函数?
2. 对于语句 $d: x := e$ (e 为任意表达式), 如何计算 $AD(d, x, OUT)$?
3. 对于语句 $d: x := e$ (e 为任意表达式), 如何计算 $AD(d, x, IN)$?

参考答案:

1. 将所有 $AD(\cdot, \cdot, \cdot)$ 初始化为 False。
2. $AD(d, x, OUT) = \text{True}$ 。
3. $AD(d, x, IN) = \text{True}$ 当且仅当对于所有直接到达 d 的语句 p , $AD(p, x, OUT) = \text{True}$ 。

Q6. 考虑如下基本块。

1	$B := 32$	7	$I := A * C$
2	$D := A + C$	8	$J := H + I$
3	$E := A * C$	9	$K := B * 5$
4	$F := D + E$	10	$L := K + J$
5	$G := B * F$	11	$M := L$
6	$H := A + C$		

1. 依次对该基本块进行如下优化: 常量传播、强度削弱 (假定运算由快至慢分别为: 加减法、位移、乘法)、合并已知量、常量传播, 写出优化后的语句序列。(无需写出中间步骤)
2. 构造 DAG。(不考虑前一问的优化)

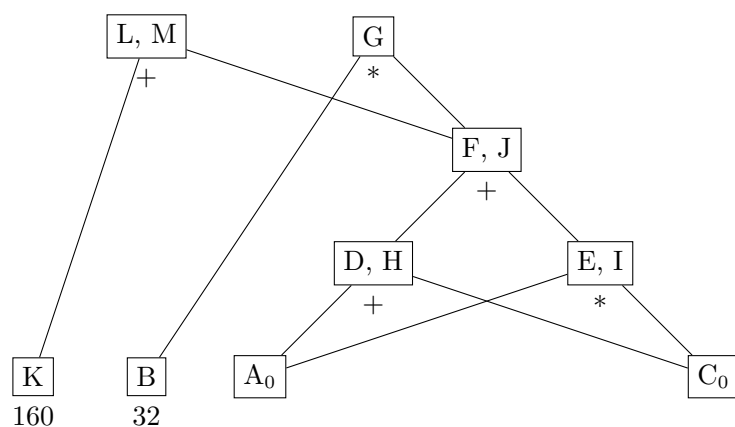
3. 基于上一问的 DAG，假设只有 L 在基本块后面还要被使用，写出优化后的语句序列。

参考答案：

1. 如下所示。

1	B := 32	7	I := A * C
2	D := A + C	8	J := H + I
3	E := A * C	9	K := 160
4	F := D + E	10	L := 160 + J
5	G := F << 5	11	M := L
6	H := A + C		

2. 如下所示。



3. 如下所示。

1	D := A + C
2	E := A * C
3	F := D + E
4	L := 160 + F

Principles and Practice of Compiler Construction HW

Due: 12/30/2025

If you have any suggestions, feel free to contact the TAs.

Q1. For the following C function section, its runtime activation record is organized as in the right figure.

```
1 static int N, L;
2 int foo() {
3     int a[100];
4     int s = 3;
5     int b[N];
6     register int i;
7     for (i = 0; i < L; i++) {
8         b[i] = i * i;
9     }
10    a[10] = b[5];
11    return s;
12 }
```

High address	
...	Old SP
Return address RA	FP
Old frame pointer FP	
Array a	
Variable s	FP
Array b	SP
Low address	

Where, `i` is stored in register `T0` rather than on the stack. The activation record first store the return address `RA` and old stack frame pointer `FP`, and then store array `a` and variable `s` in the order of definition, at last store the dynamic array `b`.

We use `FP` to store the stack top pointer `SP` before allocating dynamic array `b`. In this problem, array element are stored in the stack from low address to high address in the order of index, for example, `b[0]` is stored at the address pointed by `SP`.

Suppose the memory address of global variable `N` is `0x4000`. The word length is 4 bytes.

1. How many bytes is the total size of the stack frame (activation record) of the function? Write as expression of `N`.
2. When entering the function, we should first allocate stack space for all fixed-sized variables (array `a` and variable `s`) and old `FP`, `RA`. After that, allocate space for dynamic array `b`. The RISC-V code generated is as follows. Please fill the blanks.

```

1 func_prologue:
2 addi sp, sp, _____ # SP <- SP + ???
3 sw ra, _____ # Mem[SP + ???] <- RA
4 sw fp, _____ # Mem[SP + ???] <- FP
5 mv fp, sp # FP <- SP
6 lw a0, 0x4000(zero) # A0 <- Mem[0x4000]
7 slli a0, a0, 2 # A0 <- A0 * 4
8 sub sp, sp, a0 # SP <- SP - A0

```

3. Statement `a[10] = b[5];` in the function is translated into the following target code. Please fill the offsets.

```

1 lw a0, _____(sp) # A0 <- Mem[SP + ???]
2 sw a0, _____(fp) # Mem[FP + ???] <- A0

```

4. There is a buffer overflow vulnerability. If the value of L is greater than N will cause the array `b` to out-of-bound access, and may overwrite data on the stack.

Overwriting data may cause the function to return bad values. The more serious problem is, if the data overwritten is the return value, the function may return to bad value.

Please fill: return value will be overwritten when L is greater than _____, return address will be overwritten when greater than _____. Use expressions of N .

Solution:

1. $4N + 412$ bytes.

2. As follows.

```

1 func_prologue:
2 addi sp, sp, -412
3 sw ra, 408
4 sw fp, 404
5 mv fp, sp
6 lw a0, 0x4000(zero)
7 slli a0, a0, 2
8 sub sp, sp, a0

```

3. As follows.

```

1 lw a0, 20(sp)
2 sw a0, 44(fp)

```

4. Overwrite return value when $L \geq N + 1$, overwrite return address when $L \geq N + 103$.

Q2. Following is a code section of a simple language. Its grammar is almost the same with C, but it supports nested function declaration and global statements as in Python.

To simplify, all the functions we discuss here has no arguments nor return value. Each function has a static scope, program running uses static scoping.

The progress activation record consists of control information (static link SL, dynamic link DL, return value RA) and the variables declared in the current scope.

The right figure shows the runtime stack state when the program calls `foo()` the second time, where variable names stand for the position stores the value of the variable, left of the box is item index.

Notice: usually the address is expanded from high to low, but here the index is from low to high.

```

1  int a, b;
2  void foo() {
3      int x;
4      void bar() {
5          int x, a;
6          a = 3;
7          if (a > b) baz();
8          x = ...;
9      }
10     bar();
11     x = ...;
12 }
13 void baz() {
14     int x;
15     if (a < b) foo();
16     x = ...;
17 }
18 a = 1;
19 b = 2;
20 baz();

```

0	0 (SL)
1	0 (DL)
2	RA
3	a
4	b
5	0 (SL)
6	0 (DL)
7	RA
8	baz.x
9	0 (SL)
10	5 (DL)
11	RA
12	foo.x
13	9 (SL)
14	9 (DL)
15	RA
16	bar.x
17	bar.a
18	_____(SL)
19	_____(DL)
20	_____
21	_____
22	_____(SL)
23	_____(DL)
24	RA
25	x

1. Fill the blanks in the runtime stack.
2. We use Display table to replace the static link. Suppose we only store one Display table entry in the activation record, and the entry takes the place of SL in the figure.
Please point out the content of Display table in the current running state, and the Display

table entry stored in each activation record. *That is, the new contents of the SL positions.*

3. If we use dynamic scoping, what's the difference of the running result of the program?
Just point out the first time when the control flow is different.

Solution:

1. Item 18-23 has the value 0, 13, RA, baz.x, 0, 18, respectively.
2. The current Display table is: $D[0] = 0, D[1] = 22, D[2] = 13$.
Item 0, 5, 9, 13, 18, 22 has the value: invalid, invalid, 5, invalid, 9, 18, respectively.
3. When the program reaches line 15 for the second time, if we use static scoping, then $a = 1, b = 2$, `foo()` will be called again; if we use dynamic scoping, then $a = 3, b = 2$, `foo()` will not be called.

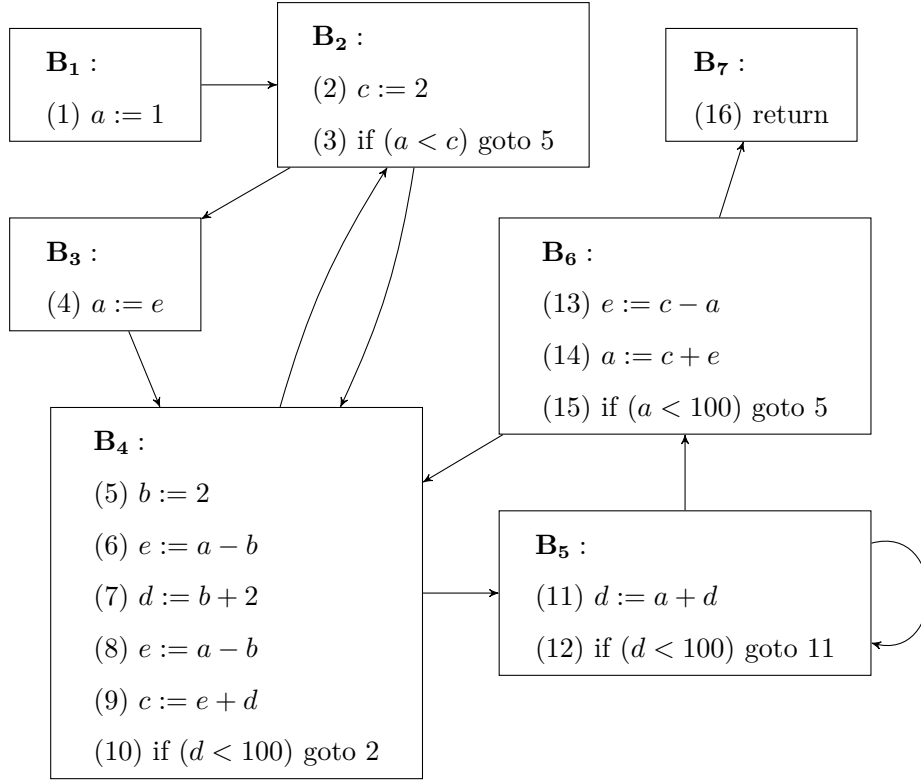
Q3. Consider the following IR.

<pre> 1 a := 1 2 c := 2 3 if (a < c) goto 5 4 a := e 5 b := 2 6 e := a - b 7 d := b + 2 8 e := a - b </pre>	<pre> 9 c := e + d 10 if (d < 100) goto 2 11 d := a + d 12 if (d < 100) goto 11 13 e := c - a 14 a := c + e 15 if (a < 100) goto 5 16 return </pre>
--	--

1. Give the basic block partitioning and flow graph of the code. Use $B_1, B_2, \dots$ to name the basic blocks, where the order should be consistent with the order of the first line of the basic block.
2. Please write down: (1) dominator set of the basic block containing Line 5, (2) back edge starting from this basic block, (3) natural loop based on the back edge.
Use the basic block name of the last problem.
3. Use the method of solving data flow equations iteratively to analyze live variable information of the code. Give the sets Def, LiveUse, LiveIn, LiveOut of each basic block. Suppose the LiveOut set of exit basic block is empty set.
4. Use the method of solving data flow equations iteratively to analyze reaching definitions information. Give the sets Gen, Kill, In, Out of each basic block. Suppose the In set of entry basic block is empty set.
5. Give the UD chain of variable a at Statement 11.
6. Give the DU chain of variable c at Statement 2.
7. Draw the register interference graph.
8. According to the graph coloring register allocating algorithm in the lecture. If we do not want to spill registers to the memory during the coloring process, what is the minimum number of physical register needed? Give a possible coloring scheme.

Solution:

1. As follows.



2. Dominator set of B_4 : $\{B_1, B_2, B_4\}$; back edge from B_4 : (B_4, B_2) ; natural loop : $\{B_2, B_3, B_4, B_5, B_6\}$.

3. As follows.

	Def	LiveUse	LiveIn	LiveOut
B_1	$\{a\}$	$\emptyset$	$\{e\}$	$\{a, e\}$
B_2	$\{c\}$	$\{a\}$	$\{a, e\}$	$\{a, e\}$
B_3	$\{a\}$	$\{e\}$	$\{e\}$	$\{a\}$
B_4	$\{b, c, d, e\}$	$\{a\}$	$\{a\}$	$\{a, c, d, e\}$
B_5	$\emptyset$	$\{a, d\}$	$\{a, c, d\}$	$\{a, c, d\}$
B_6	$\{e\}$	$\{a, c\}$	$\{a, c\}$	$\{a\}$
B_7	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

4. As follows.

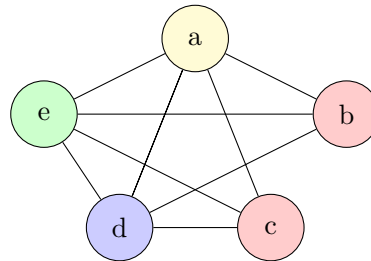
Whether the Kill set of B_6 includes 6 or not is both possible. It does not influence the computation of In and Out.

	Gen	Kill	In	Out
B_1	$\{1\}$	$\emptyset$	$\emptyset$	$\{1\}$
B_2	$\{2\}$	$\{9\}$	$\{1, 4, 5, 7, 8, 9, 14\}$	$\{1, 2, 4, 5, 7, 8, 14\}$
B_3	$\{4\}$	$\{1, 14\}$	$\{1, 2, 4, 5, 7, 8, 14\}$	$\{2, 4, 5, 7, 8\}$
B_4	$\{5, 7, 8, 9\}$	$\{2, 11, 13\}$	$\{1, 2, 4, 5, 7, 8, 9, 11, 13, 14\}$	$\{1, 4, 5, 7, 8, 9, 14\}$
B_5	$\{11\}$	$\{7\}$	$\{1, 4, 5, 7, 8, 9, 11, 14\}$	$\{1, 4, 5, 8, 9, 11, 14\}$
B_6	$\{13, 14\}$	$\{1, 4, 8\}$	$\{1, 4, 5, 8, 9, 11, 14\}$	$\{5, 9, 11, 13, 14\}$
B_7	$\emptyset$	$\emptyset$	$\{5, 9, 11, 13, 14\}$	$\{5, 9, 11, 13, 14\}$

5. UD chain : $\{1, 4, 14\}$.

6. DU chain : $\{3\}$.

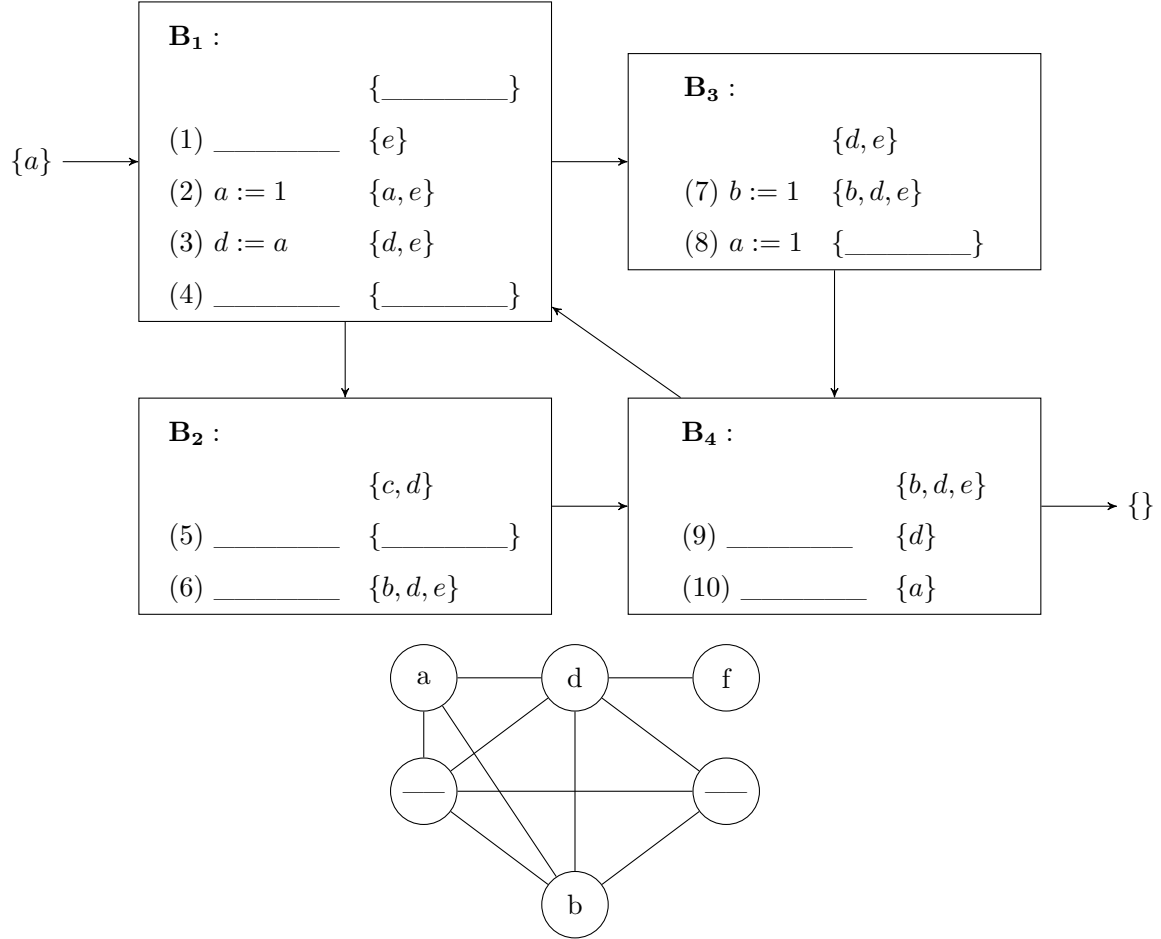
7. As follows.



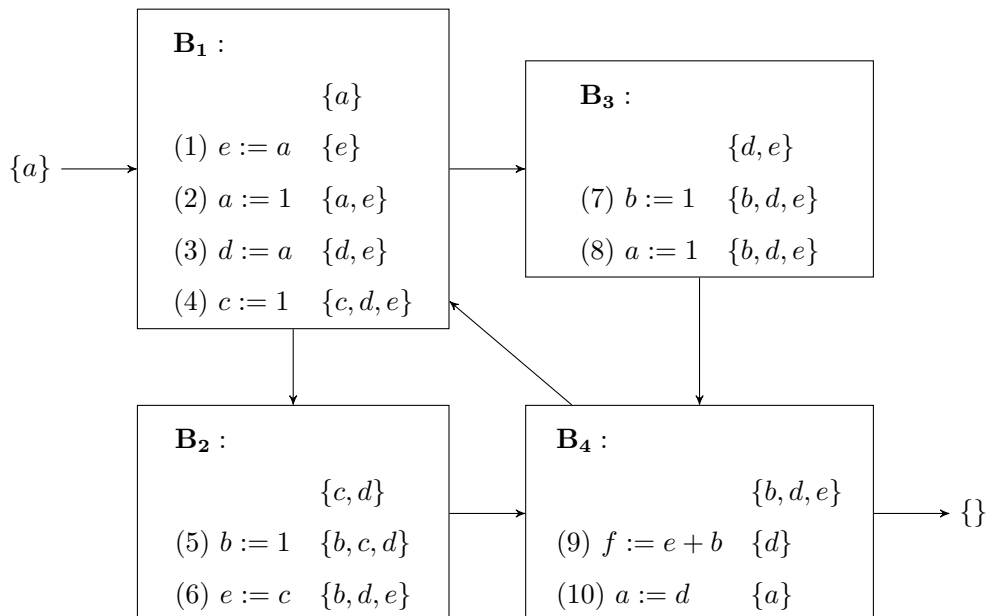
8. Minimum register number needed is 4. A possible coloring scheme is given in the solution of last problem.

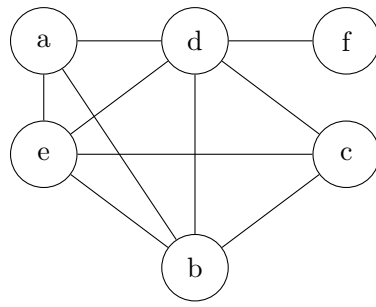
Q4. Here is an incomplete flow graph and register interference graph. The live variable list after each statement is attached to the graph. Fill in the missing parts.

The missing statements must be chosen from such forms: $x := 1$, $x := y$, $x := y + z$.



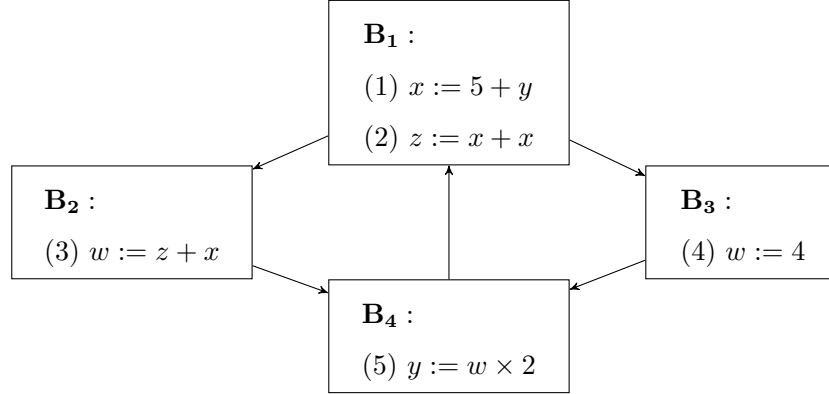
Solution:





Q5. In class, we've learned many types of data flow. Now we want to design a new data flow type "Always-Defined", to help us decide whether a variable x is always defined before used in statement d .

Consider the following flow graph, x is always defined in statements 2 and 3, w is always defined in statement 5, but y is not always defined in statement 1.



To implement such "Always-Defined" analysis, we use function $AD(d, x, OUT)$ to signify whether variable x is always defined after the execution of statement d , $AD(d, x, IN)$ to signify whether x is always defined before the execution of statement d .

Function $AD(\cdot, \cdot, \cdot)$ has only two possible values: True and False.

1. We should use which value to initialize function $AD(\cdot, \cdot, \cdot)$?
2. For statement $d : x := e$ (e is any expression), how to calculate $AD(d, x, OUT)$?
3. For statement $d : x := e$ (e is any expression), how to calculate $AD(d, x, IN)$?

Solution:

1. Initialize all $AD(\cdot, \cdot, \cdot)$ as False.
2. $AD(d, x, OUT) = \text{True}$.
3. $AD(d, x, IN) = \text{True}$ if and only if for any statement p that directly succeeded by d , $AD(p, x, OUT) = \text{True}$.

Q6. Consider the following basic block.

<pre> 1 B := 32 2 D := A + C 3 E := A * C 4 F := D + E 5 G := B * F 6 H := A + C </pre>	<pre> 7 I := A * C 8 J := H + I 9 K := B * 5 10 L := K + J 11 M := L </pre>
---	---

1. Perform the following optimizations on the basic block in sequence: constant propagation, strength reduction (assuming the operations from fastest to slowest are: addition/sub-

traction, shift, multiplication), combining known quantities, and constant propagation.
Write the optimized statement sequence.

Notice: No need to write the intermediate steps.

2. Construct the DAG.

Notice: Do not consider the optimization in the last problem.

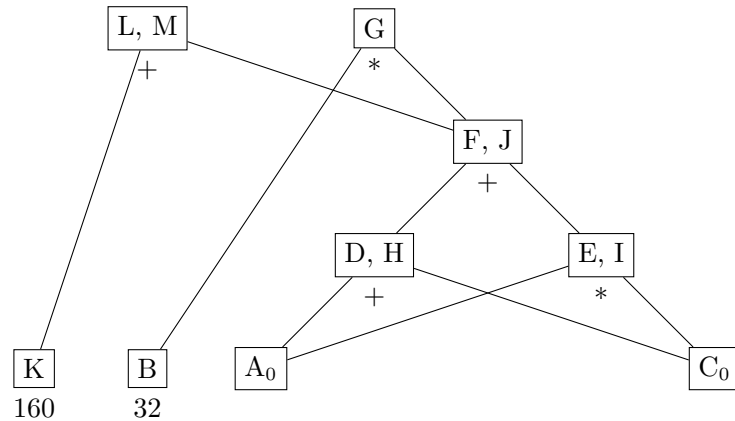
3. Based on the DAG of the last problem, suppose only L will be used after this basic block, give the optimized statement sequence.

Solution:

1. As following.

<pre> 1 B := 32 2 D := A + C 3 E := A * C 4 F := D + E 5 G := F << 5 6 H := A + C </pre>	<pre> 7 I := A * C 8 J := H + I 9 K := 160 10 L := 160 + J 11 M := L </pre>
--	---

2. As following.



3. As following.

```

1 | D := A + C
2 | E := A * C
3 | F := D + E
4 | L := 160 + F

```