清华大学本科生考试试题专用纸

考试课程 《编译原理》 （B 卷） 2023 年 1 月 5 日

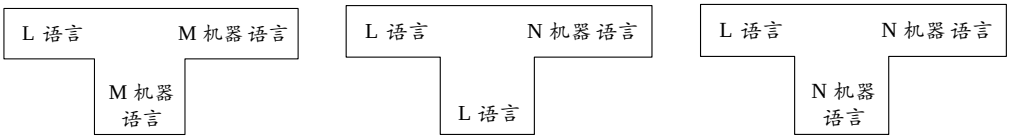
学号：_____ 姓名：_____ 班级：_____

（注：解答请写在自己准备的答题纸上。）

一.（15分）简答题

1. (3 分) 填空：

设单源单目标编译器 X ，Y 和 Z 的 T 型图从左到右分别为：

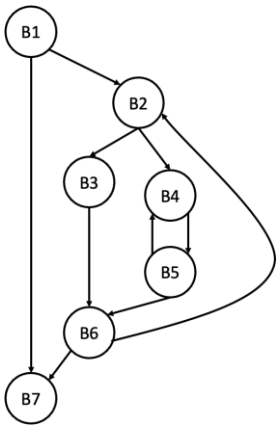


假设已经存在编译器 X 和 Y，并限定不能用 Y 直接编译包含自身在内的任何编译器（注: 这样会有，A 不是 Y），那么可以通过如下步骤实现编译器 Z：

（1）用 _____ 编译 _____ 得到编译器 A；

（2）再用 A 编译 ____ 得到编译器 Z .

2. (2 分) 根据以下流图（B1 为入口基本块）回答问题：



在以上流图中，有多少个自然循环？

3. (3 分) 对于下列文法 G[*S*]，试给出右句型 *aaSabab* 的所有短语和句柄；

$$S \to a\,S\,b \mid S\,a \mid c$$

4. (7 分) 给定 LL(1) 文法 *G*[*S*]：

$$S \to A\,b\,B$$
$$A \to a\,A \mid c\,A \mid \varepsilon$$
$$B \to b\,B \mid \varepsilon$$

如下是以 *G*[*S*] 作为基础文法设计的一个 *L* 翻译模式：

$$S \to A\,b\ \{\ B.in\_num := A.num - 1\}\ B\ \{\ print(B.num)\ \}$$
$$A \to a\,A_1\ \{\ A.num := A_1.num + 1\ \}$$

$A \rightarrow c\ A_1$ { $A.\ num := A_1.num - 1$ }
$A \rightarrow \varepsilon$ { $A.\ num := 0$ }
$B \rightarrow b$ { $B_1.in\_num := B.in\_num$ } B_1 { $B.\ num := B_1.num - 1$ }
$B \rightarrow \varepsilon$ { $B.\ num := B.in\_num$ }

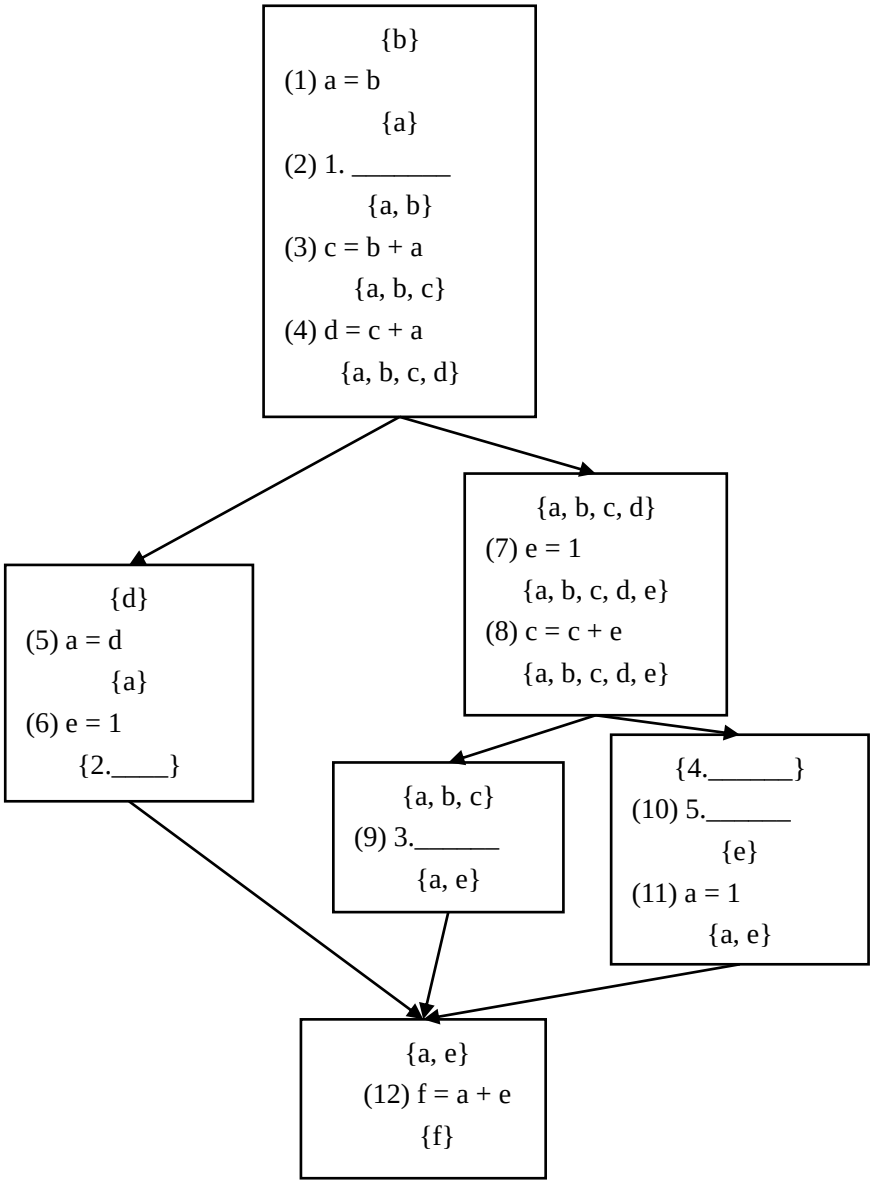
针对该翻译模式构造相应的递归下降（预测）翻译程序。该翻译程序中 (1)-(7) 的部分未给出，试填写之。

```
void ParseS( )                          // 主函数
{
    A_num := ParseA();
    MatchToken('b');
    (1)            ;
    (2)            ;
    print(B_num)
}

int ParseA( )
{
    switch (lookahead)  {                   // lookahead为下一个输入符号
        case 'a' :
            MatchToken('a');
            A1_num := ParseA();
            (3)            ;
            break;
        case 'c':
            MatchToken('c');
            A_num := A1_num - 1;
            break;
        case (4)            :
            A_num := 0;
            break;
        default:
            printf("syntax error \n")
            exit(0);
    }
    (5)          ;
}

int ParseB( int B_in_num )
{
    switch (lookahead)  {
        case 'b' :
            (6)          ;
            B1_in_num := B_in_num;
            B1_num := ParseB(B1_in_num);
            B_num := B1_num-1;
            break;
        case '#' :
            (7)          ;
            break;
        default:
            printf("syntax error \n")
            exit(0);
    }
    return B_num;
}
```

二. **(11分)** 下图是一个流图的一部分，{} 中标注的是当前位置的活跃变量集合。



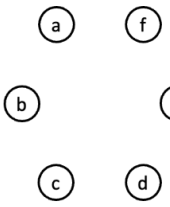
1. **(5分)** 请补充空余部分的语句或者活跃变量集合，空缺的语句只有以下三种形式：

   x=1, x=y, 和 x=y+z,

   其中 x, y, z 是 a-z 中的字母，并且所有被赋值的变量均在当前定值点后的活跃变量集合中。如果一个空有多种可能情况写出一种即可。

2. **(2分)** 请指出该流图范围内，变量 d 在 (4) 的 DU 链。

3. **(2分)** 请画出上述流图的寄存器相干图。(2分)



4. **(2分)** 在上述流图范围内，若采用课堂介绍的启发式算法实现的图着色寄存器分配，想要不出现寄存器spill到内存的情况，最少需要多少个物理寄存器？

三.（**13 分**）左下是某 C 风格语言的一段代码（斜体为关键字），该语言与 C 的主要不同点为：

- 若变量被声明后未赋值就被使用，则认为初始值为 0；
- 允许嵌套的函数定义，且函数在其被定义的整个作用域中均可见，例如示例代码中 func1 里可以调用后定义的 func2，这一点与 C++中类的成员函数类似。
- 新关键字 *print_address* 可用于打印一个变量的地址。

已知该语言中的函数遵循静态作用域规则，活动记录中的控制信息包括静态链 SL、动态链 DL，以及返回地址 RA。

```
(01) int x, y;
(02) void func0() {
(03)     int x = x, y;
(04)     void func1() {
(05)         int x = y + 2;
(06)         print_address y;
(07)         func2();
(08)     }
(09)     void func2() {
(10)         int y = x + 2;
(11)         if (y % 2 == 0 || y < 9)
(12)             func1();
(13)     }
(14)     y = x + 2;
(15)     func1();
(16) }
(17) x = 1;
(18) func0();
```

25	y	
24	?	RA
23		DL
22		SL
21	x	
20	?	RA
19		DL
18		SL
17	y	
16	?	RA
15	10	DL
14	5	SL
13	x	
12	?	RA
11	5	DL
10	5	SL
9	y	
8	x	
7	?	RA
6	0	DL
5	0	SL
4	y	
3	x	
2	?	RA
1	0	DL
0	0	SL

1. **(6分)** 若该语言的变量也遵循静态作用域规则，实现时采用多符号表结构，每个静态作用域均对应一个符号表（函数和变量共用，在静态分析前已经创建完毕，作用域内全部符号已在表中）。试指出：分析至语句11时，开作用域有哪些？都分别包含哪些符号？

2. **(4分)** 左边的代码第二次执行到语句11时，运行栈的当前状态如右半部分所示（栈顶指向单元26），其中变量的名字用于代表相应的值。试补齐该运行状态下，单元18、19、22和23中的内容。

3. **(3分)** 为探究该语言中变量遵循的是静态作用域规则还是动态作用域规则，Chisato 同学尝试运行上述代码，发现程序不会终止，据此能否做出判断？请简要说明理由。

Takina同学发现，执行过程中语句06输出的信息也能用于判断变量的作用域规则，请简要解释一下原理。

四. (12分) 给定文法 $G[S]$：

$$S \to E\ b$$
$$E \to a\ E\ b \mid E\ c\ E \mid \varepsilon$$

1. **(5分)** 针对文法 $G[S]$，请计算出各产生式的预测集合 *PS*，各产生式右部文法符号串的 *First* 集合，以及各产生式左部非终结符的 *Follow* 集合。即完成下表：

（表中的 *rhs(r)* 表示产生式 *r* 右部的文法符号串，*lhs(r)* 表示产生式 *r* 左部的非终结符。）

G 中的规则 r	*First* (*rhs*(*r*))	*Follow* (*lhs*(*r*))	*PS* (*r*)
$S \to E\ b$			
$E \to a\ E\ b$			
$E \to E\ c\ E$		此处不填	
$E \to \varepsilon$		此处不填	

2. **(4分)** 以下是文法 $G[S]$ 的预测分析表，请完成表中的内容：

	a	*b*	*c*	#
S	$S \to Eb$	$S \to Eb$	$S \to Eb$	
E				

3. **(3 分)** 该文法 $G[S]$ 是否为 LL(1) 文法，为什么？

五. (10 分)

以下是二值布尔表达式计算的一个文法 G(E):

$S \to S_1 \vee T$	{ S.val = S_1.val ∨ T.val; }
$S \to T$	{ S.val = T.val; }
$T \to T_1 \wedge F$	{ T.val = T_1.val ∧ F.val; }
$T \to F$	{ T.val = F.val; }
$F \to \neg F$	{ T.val = ¬F.val; }
$F \to \underline{false}$	{ F.val = false; }
$F \to \underline{true}$	{ F.val = true; }

1. **(2分)** 请给出表达式 $\underline{true} \wedge \underline{false} \vee \neg\ \underline{true}$ 的语法分析树和相应的带标注语法分析树。

2. **(3分)** S-翻译模式在形式上与 S-属性文法是一致的，可以采取同样的语义计算方法。如果在 LR 分析过程中根据该翻译模式进行自底向上语义计算，试填写出在按

每个产生式归约时实现语义计算的一个代码片断，可以体现语义栈上的操作(设语义栈由向量 v 表示， 归约前栈顶位置为 top，不用考虑对 top 的维护)。

$S \to S_1 \vee T$	{ (1)_____ }
$S \to T$	{ v[top].val:=v[top].val; }
$T \to T_1 \wedge F$	{ (2)_____ }
$T \to F$	{ v[top].val:=v[top].val; }
$F \to \neg F$	{ (3)_____ }
$F \to \underline{false}$	{ v[top].val = false; }
$F \to \underline{true}$	{ v[top].val = true; }

3. (5分) 该文法 G(E) 其含有左递归，因而不能用 LL(1) 方法。给出将

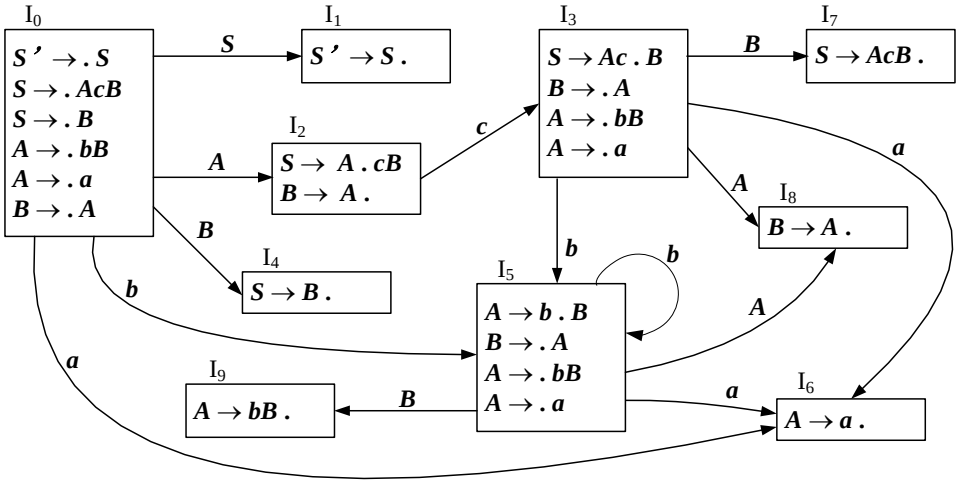
$S \to S_1 \vee T$	{ S.val = S_1.val $\vee$ T.val; }
$S \to T$	{ S.val = T.val; }

中的关于 S 的直接左递归消去后，变换得到的一个 L-翻译模式。

六. (24分) 给定文法 $G[S]$：

(1) $S \to AcB$
(2) $S \to B$
(3) $A \to bB$
(4) $A \to a$
(5) $B \to A$

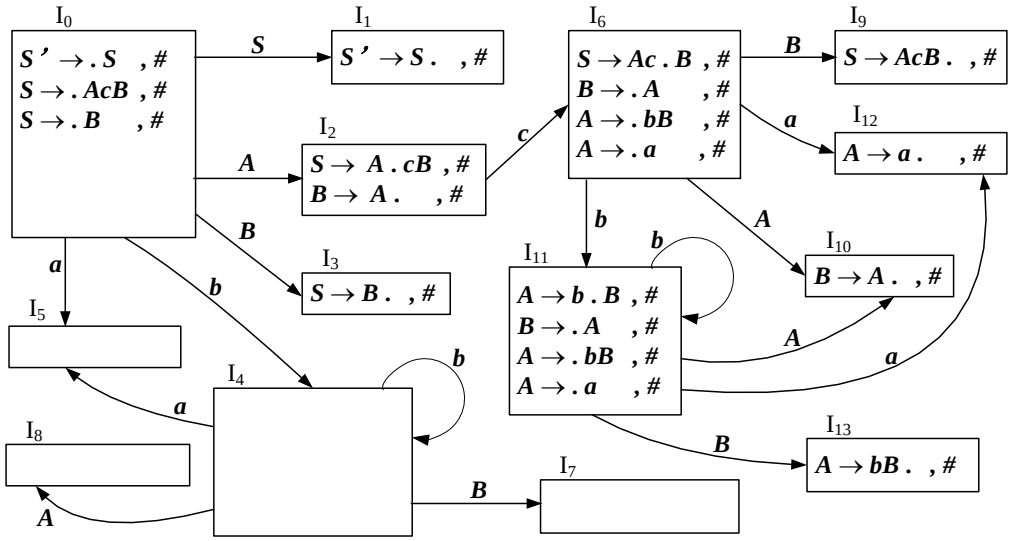
设 $G'[S']$ 为 $G[S]$ 的增广文法。下图表示 $G[S]$ 的 LR(0) 有限状态机：



1. (2分) 从上述 LR(0) 有限状态机可以看出， $G[S]$ 不是 LR(0) 文法。试指出该 LR(0) 有限状态机中哪一个或哪几个状态是有冲突的？同时请指出所包含的每一个冲突的类别，即是移进-归约冲突，还是归约-归约冲突？

2. (3分) $G[S]$ 也不是SLR(1) 文法，请简单解释原因。

3. (3分) 下图是相应于 $G[S]$ 的 LR(1) 有限状态机，但部分状态对应的 LR(1) 项目集信息并未完善，试补齐之（分别补全状态 I_0，I_4，I_5，I_7 和 I_8 对应的项目集）。



4. **(3分)** 若依据上述的 LR(1) 有限状态机进行 LR 分析，当到达状态 I_{10} 时，分析栈（含符号栈）上的符号串（对应某个活前缀）可能有哪些？（如必要，结果可以写成正规表达式的形式）

5. **(4分)** 从上述的 LR(1) 有限状态机出发，通过合并同芯（同心）状态可得到 *G*[S] 的 LALR(1) 有限状态机。该 LALR(1) 有限状态机共有多少个状态？从 LALR(1) 有限状态机可看出 *G*[S] 为 LALR(1) 文法。LR(1) 有限状态机中状态 I_2 合并后有无变化？简述该状态为什么不冲突。

6. **(4分)** 给出 *G*[S] 的 LALR(1) 分析表前 4 行（分别对应于 LR(1) 有限状态机中状态 I_0，I_1，I_2 和 I_3 合并后的 LALR(1) 有限状态机状态）的内容。（即完成下表）

状态	ACTION				GOTO		
	a	*b*	*c*	#	*S*	*A*	*B*
0							
1							
2							
3							

7. **(5分)** 根据上述的 LR(1) 和 LALR(1) 有限状态机，分别进行 LR 分析。对于一个有语法错误的输入符号串，一般哪一种分析方法更快一些？所谓“更快”是指发现错误时经过的步数较少，而每一“步”是指进行一次“移进”或者完成一次“归约”的动作。请给出一个会发生语法错误的输入符号串，用以支持你的结论。

七. (15分)

1. **(5 分)** 在 Rust，C++11 等现代编程语言中存在“移动语义（move semantics）”。其大意是指在变量赋值和初始化时，我们并不是简单的将等号右边的变量所代表的资源复制（copy）到等号左边的变量，而是把等号右边的变量所代表的资源转移（move）到等号左边的变量。为此我们定义如下文法：

$P \rightarrow S$
$S \rightarrow S_1 ; S_2$
$S \rightarrow id := E$
$E \rightarrow E_1 + E_2$
$E \rightarrow val$
$E \rightarrow move\ id$
$E \rightarrow copy\ id$

其中 *id* 代表了变量名，*val* 代表了一种类型的字面量，我们这里可以认为是整数。其中 *id := move E* 即代表了移动语义，*id := copy E* 代表了一般的基于复制的赋值和初始化。移动语义最基本的要求是一个变量不能被移动一次以上，同时一个变量被移动后，就不能再访问，例如下面就是一段合法的程序

```
a := 10;
b := move a;
b := 30
```

但是下面这段程序就不合法，因为 *a* 在第二行已经被移动了，在第三行的读是非法的，同时最后一行的再次移动也是非法的。

```
a := 10;
b := move a;
c := copy a;
b := 10;
c := move a
```

下面我们设计一个 L– 翻译模式来检查一段程序是否符合移动语义，我们使用 *s = makeset()* 来生成一个空集合，用 *s.insert(x), s.delete(x), s.has(x)* 来进行插入、删除和查询，使用 *if ... then ... else ...* 来执行条件语句。属性 *ok* 为布尔变量，表示对应的语句是否符合移动语义，*in_active* 表示对应语句前可以被移动的变量，*active* 表示对应语句执行完后可以被移动的变量，我们使用变量名的 *entry* 属性作为操作集合的符号。请你补全缺失的部分。

$P \rightarrow \{\ S.in\_active = makeset();\ \}\ S\ \{\ P.ok = S.ok\ \}$
$S \rightarrow \{\ S_1.in\_active = S.in\_active\ \}\ S_1\ ;\ \{\ \underline{\quad\quad (1) \quad\quad}\ \}\ S_2$
$\{\ S.active = S_2.active;\ S.ok = S_1.ok\ and\ S_2.ok\ \}$
$S \rightarrow id := \{\ E.in\_active = S.in\_active\ \}\ E\ \{\ S.ok = E.ok;\ S.active = E.active;$
$S.active.insert(id.entry)\ \}$
$E \rightarrow \{\ E_1.in\_active = E.active\ \}\ E_1 + \{\ \underline{\quad\quad (2) \quad\quad}\ \}\ E_2$
$\{\ E.active = E_2.active;\ E.ok = E_1.ok\ and\ E_2.ok\ \}$
$E \rightarrow val\ \{\ E.active = E.in\_active;\ E.ok = true\ \}$
$E \rightarrow copy\ id\ \{\ \underline{\quad\quad (3) \quad\quad}\ \}$
$E \rightarrow move\ id\ \{\ \underline{\quad\quad (4) \quad\quad}\}$

2. (4分) 基于上面的文法，我们把 *val* 视为一维整数向量，定义 *val.ptr* 为向量的基地址，*val.len* 为向量的长度。一个向量的字面表示可以为 $[a_0, a_1, \ldots, a_n]$，例如 [*1, 2, 3*] 表示了一个长度为 3 的向量。我们定义向量的加法为逐个相加，且需要满足两者的长度相同。

同时我们增加拼接（concatenate）表达式的支持，即

$E \rightarrow concat\ E_1, E_2$

该表达式可以把两个向量前后拼接在一起，例如 *concat* [*1, 2, 3*], [*4, 5*] 的结果应该为 [*1, 2, 3, 4, 5*]。我们用 *set_len* 来在一个全局的数据结构中设置一个对应向量的长度，用 *lookup_len* 来读取一个对应向量的长度; *type* 属性用于检查类型是否正确，其值若为 *ok* 则表示没问题，若为 *error* 则表示检查过程中存在错误；其余定义同上。请你补全下面的 *S* – 翻译模式，来进行向量运算长度的检查。

P → *S* { *P.type = S.type* }
S → S_1 ; S_2 { *if* S_1*.type = ok and* S_2*.type =ok then S.type = ok else S.type = error* }
S → *id := E* { *if E.type = ok then S.type = ok; set_len(id.entry, E.len)*
else S.type = error }
E → E_1 + E_2 { _____(1)_____ }
E → *concat* E_1, E_2 {_____(2)_____ }
E → *val* { *E.len = val.len; E.type = ok* }
E → *move id* { *set_len(id.entry, id.len); E.len = id.len; E.type = ok* }
E → *copy id* { *set_len(id.entry, id.len); E.len = id.len; E.type = ok* }

3. (6 分) 基于上面的文法，我们增加中间表达 *ptr = alloc len* 表示分配长度为 *len* 的空间，将基地址赋给 *ptr* ；以及 *memcpy* ptr_1, ptr_2, *len* 表示将 [ptr_1, ptr_1 + *len -1*] 这部分的内容复制到 [ptr_2, ptr_2 + *len -1*]；以及 *add* ptr_1, ptr_2, ptr_3, *len* 表示将 [ptr_1, ptr_1 + *len -1*] 以及 [ptr_2, ptr_2 + *len -1*] 相加的结果写到 [ptr_3, ptr_3 + *len -1*]，其中 ptr_1 和 ptr_2 与 ptr_3 可以相等。

你可以使用 *newtemp()* 来得到一个临时变量，*gen("...")* 来生成中间表达的代码，代码之间可以通过 || 来连接；*code* 属性表示生成的中间表达， *ptr* 属性为一个中间表达的变量，存储对应向量的基地址， *writable* 属性表示对应的向量在内存中是否是可写的；其余定义同上。

请你补全下面的 S – 翻译模式，生成合适的中间表达，使得分配的空间尽可能小。

P → *S* { *P.code = S.code* }
S → S_1 ; S_2 { *S.code =* S_1*.code ||* S_2*.code* }
S → *id* := *E* { *set_len(id.entry, E.len); id.ptr = newtemp(); if E.writable then*
S.code = E.code || gen("id.ptr = E.ptr") else S.code = E.code || gen("id.ptr = alloc E.len") || gen("memcpy E.ptr, id.ptr, E.len") }
E → E_1 + E_2 { _____(1)_____}
E → *concat* E_1, E_2 {_____(2)_____ }
E → *val* {*E.ptr = newtemp(); E.len = val.len; E.writable = true;*
E.code = gen("E.ptr = alloc E.len") || gen("memcpy val.ptr, E.ptr, E.len")}
E → *move id* {_____(3)_____ }
E → *copy id* { *E.ptr = newtemp(); E.len = lookup_len(id.entry); E.writable = false;*
E.code = gen("E.ptr = id.ptr") }