（若发现问题，请及时告知）

第 7 讲书面作业包括两部分。第一部分为 Lecture07.pdf 中课后作业题目中的第 4，5，和 10 题。第二部分为以下题目：

A1　参考 2.3.4 节采用短路代码进行布尔表达式翻译的 *L*-翻译模式片断及所用到的语义函数。若在基础文法中增加产生式 E → E ? E : E，试给出相应该产生式的语义动作集合。其语义可用其它逻辑运算定义为 P ? Q : T ≡ P and Q and (not T)。

A2　参考 2.3.6 节采用拉链与代码回填技术进行布尔表达式和控制语句翻译的 *S*-翻译模式片断及所用到的语义函数，重复题 A1 的工作。

A3 考虑一个简单的栈式虚拟机。该虚拟机维护一个存放整数的栈，并支持如下3条指令：

- `Push` *n*：把整数 *n* 压栈；
- `Plus`：弹出栈顶元素 n_1 和次栈顶元素 n_2，计算 $n_1 + n_2$ 的值，把结果压栈；
- `Minus`：弹出栈顶元素 n_1 和次栈顶元素 n_2，计算 $n_1 - n_2$ 的值，把结果压栈。

一条或多条指令构成一个指令序列。初始状态下，虚拟机的栈为空。

给定一个仅包含加法和减法的算术表达式语言：

$$A \to A + A \mid A - A \mid (A) \mid \underline{\text{int}}$$

终结符 int 表示一个整数，用 int.val 取得语法符号对应的语义值。

任何一个算术表达式都可以翻译为一个指令序列，使得该虚拟机执行完此指令序列后，栈中仅含一个元素，且它恰好为表达式的值。简单起见，我们用“||”来拼接两个指令序列。例如，算术表达式 1 + 2 – 3可翻译成指令序列

`Push` 3 || `Push` 2 || `Push` 1 || `Plus` || `Minus`

执行完成后，栈顶元素为0。

(1) 上述翻译过程可描述成如下S-翻译模式，其中综合属性 *A*.instr表示*A*对应的指令序列：

$A \to A_1 + A_2$　{ *A*.instr := ... }
$A \to A_1 - A_2$　{ *A*.instr := ... }
$A \to (A_1)$　{ *A*.instr := A_1.instr }
$A \to \underline{\text{int}}$　{ *A*.instr := `Push` int.val }

请补全其中两处空缺的部分。

给上述虚拟机新增一个变量表，支持读取和写入变量对应的整数值。新增如下指令：

- `Load` *x*：从表中读取变量 *x* 对应的值并压栈；
- `Store` *x*：把栈顶元素作为变量 *x* 的值写入表，并弹出栈顶元素；
- `Cmp`：若栈顶元素大于或等于 0，则修改栈顶元素为 1；否则，修改栈顶元素为 0；
- `Cond`：若栈顶元素非 0，则弹出栈顶元素；否则，弹出栈顶元素和次栈顶元素后，压入整数 0。

考虑一个仅支持赋值语句的简单语言*L*：

$$S \to \underline{\text{id}} := E \mid S\ ;\ S$$
$$E \to A \mid E\ \underline{\text{if}}\ B$$
$$A \to \ldots \mid \underline{\text{id}}$$
$$B \to A > A \mid B\ \&\ B \mid !B \mid \underline{\text{true}} \mid \underline{\text{false}}$$

终结符 id 表示一个变量，用 id.val 取得语法符号对应的语义值。算术表达式新增id，用来读取变量 id 的值。赋值语句 id := *E* 表示将表达式*E*的值写入变量 id。条件表达式 *E* if *B* 的语义为：若布尔/关系表达式 *B* 求值为真，则该表达式的值为 *E* 的值，否则为 0。布尔/关系表达式中，>为大于，&为逻辑与，!为逻辑非，true为真，false为假。

设 *P* 为 *L* 语言的一个程序，若 *P* 中所有被读取的变量，在读取之前都已经被赋过值，那么称 *P* 为合法程序。任何一个 *L* 语言的合法程序都可以翻译为一个指令序列，使得该虚拟机执行完此指令序列后，对任意程序中出现的变量，表中所存储的值等于程序执行后的实际值。

(2) 上述翻译过程可描述成如下S-翻译模式（与(1)中相同的部分已省略），综合属性 *E*.instr, *B*.instr, *S*.instr 分别表示 *E, B, S* 对应的指令序列：

$E \to E_1\ \underline{\text{if}}\ B$	{ *E*.instr := ... }
$A \to \underline{\text{id}}$	{ *A*.instr := `Load` id.val }
$B \to A_1 > A_2$	{ *B*.instr := ... }
$B \to B_1\ \&\ B_2$	{ *B*.instr := ... }
$B \to !B_1$	{ *B*.instr := ... }
$B \to \underline{\text{true}}$	{ *B*.instr := `Push` 1 }
$B \to \underline{\text{false}}$	{ *B*.instr := `Push` 0 }
$S \to \underline{\text{id}} := E$	{ *S*.instr := *E*.instr || `Store` id.val }
$S \to S_1\ ;\ S_2$	{ *S*.instr := S_1.instr || S_2.instr }

请补全其中四处空缺的部分。提示：在正确的实现中，任何表达式 *E, A, B* 翻译成的指令序列必须满足：虚拟机在初始状态下执行完此指令序列后，栈中仅含一个元素，且为表达式的值。

A4

（1）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二义性，即可以认为给出的文法是已经语法分析好的抽象语法。

$P \rightarrow D$ **;** S { *P.type := if D.type = ok and S.type = ok then ok else type_error* }

$S \rightarrow$ ***if*** E ***then*** S_1 ***end*** { *S.type := if E.type=bool then* S_1*.type else type_error* }

$S \rightarrow$ ***while*** E ***begin*** S_1 ***end*** { *S.type := if E.type=bool then* S_1*.type else type_error* }

$S \rightarrow S_1$ **;** S_2 { *S.type := if* S_1*.type=ok then* S_2*.type else type_error* }

$S \rightarrow$ *id* **:=** E { *S.type := if lookup_type (id.entry) = E.type then ok else type_error* }

$D \rightarrow$ ... { *省略与声明语句相关的全部规则* }

$E \rightarrow$ ... { *省略与整数/布尔表达式相关的全部规则* }

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）

$L \rightarrow$ *id* { ① }

$L \rightarrow$ *id* **,** L_1 { ② }

$R \rightarrow E$ { ③ }

$R \rightarrow E$ **,** R_1 { ④ }

$S \rightarrow$ ⑤ { ⑥ }

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) 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***

{ S_1*.next := S.next* } S_1 ***end***

{ *S.code := E.code* || *gen(E.true ":")* || S_1*.code* }

S → ***while*** { *E.true := newlabel; E.false := S.next* } *E* ***begin***

{ S_1*.next := newlabel* } S_1 ***end***

{ *S.code := gen(S1.next ":")* || *E.code* ||

gen(E.true ":") || S_1*.code* || *gen("goto"* S_1*.next)* }

S → { S_1*.next := newlabel* } S_1 ***;***

{ S_2*.next := S.next* } S_2

{ *S.code :=* S_1*.code* || *gen(*S_1*.next ":")* || S_2*.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。

$L \rightarrow id$ { ① }

$L \rightarrow id, L_1$ { ② }

$R \rightarrow E$ { ③ }

$R \rightarrow E, R_1$ { ④ }

$S \rightarrow$ 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) 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 $S \rightarrow$ ***begin*** S_1 ***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* ***:= true ;*** *x, y, a* ***:=*** *1* ***end default*** *a && b, 0*

其中有两个多余变量：*y* 和 *a*。这段代码等价为

e1 ***:=*** *a && b* ***;*** *e2* ***:=*** *0* ***;*** *b* ***:= 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 *gen(id.place " := _")* 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 *backpatch(list，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，*backpatch([id.place]，E.place)* 将会回填 *gen(id.place " := _")* 中的空白，将它变成 *gen(id.place " := " 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 → *D* ***;*** { *S.next := newlabel ; S.eint := []; S.ebool := []* }

S { *gen(S.next ":")* }

S → ***if*** { *E.true := newlabel ; E.false := S.next* } *E* ***then***

{ S_1*.next := S.next* } S_1 ***end***

{ *S.code := E.code || gen(E.true ":") ||* S_1*.code ;*

S.eint := S_1*.eint ; S.ebool :=* S_1*.ebool* }

S → ***while*** { *E.true := newlabel; E.false := S.next* } *E* ***begin***

{ S_1*.next := newlabel* } S_1 ***end***

{ *S.code := gen(*S_1*.next ":") || E.code ||*

gen(E. true ":") || S_1*.code || gen("goto"* S_1*.next);*

S.eint := S_1*.eint ; S.ebool :=* S_1*.ebool* }

S → { S_1*.next := newlabel* } S_1 ***;***

{ S_2*.next := S.next* } S_2

{ *S.code :=* S_1*.code || gen(*S_1*.next ":") ||* S_2*.code*

S.eint := S_1*.eint +* S_2*.eint ; S.ebool :=* S_1*.ebool +* S_2*.ebool* }

L → *id* { **…** }

L → *id,* L_1 { **…** }

R → *E* { **…** }

R → *E,* R_1 { **…** }

S → **…** { ①　}

S → ***begin*** S_1 ***end default*** E_1, E_2 { ②　}

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* }