

(若发现问题, 请及时告知)

A2, A3, A4, A6 是选自 Lecture07 文档中的题目

.....

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 $E \rightarrow E ? E : E$, 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 $P ? Q : T \equiv P \text{ and } Q \text{ and } (\text{not } T)$ 。

A2 参考 2.3.5 节进行语句翻译的 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 $S \rightarrow \text{repeat } S \text{ until } E$, 试给出相应该产生式的语义动作集合。

注: 控制语句 $\text{repeat } \langle \text{循环体} \rangle \text{ until } \langle \text{布尔表达式} \rangle$ 的语义为: 至少执行 $\langle \text{循环体} \rangle$ 一次, 直到 $\langle \text{布尔表达式} \rangle$ 成真时结束循环。

A3

(a) 以下是一个 L -翻译模式片断, 描述了某小语言中语句相关的一种类型检查工作:

$$\begin{aligned} P \rightarrow D ; S & \quad \{ P.type := \text{if } D.type = ok \text{ and } S.type = ok \text{ then } ok \text{ else } type_error \} \\ S \rightarrow \text{if } E \text{ then } S_1 & \quad \{ S.type := \text{if } E.type = bool \text{ then } S_1.type \text{ else } type_error \} \\ S \rightarrow \text{while } E \text{ then } S_1 & \quad \{ S.type := \text{if } E.type = bool \text{ then } S_1.type \text{ else } type_error \} \\ S \rightarrow S_1 ; S_2 & \quad \{ S.type := \text{if } S_1.type = ok \text{ and } S_2.type = ok \text{ then } ok \text{ else } type_error \} \end{aligned}$$

其中, $type$ 属性以及类型表达式 ok , $type_error$, $bool$ 等的含义与讲稿中一致。

(此外, 假设在语法制导处理的分析过程中遇到的冲突问题均可以按照某种原则处理, 这里不必考虑基础文法是否 LR 文法。)

下面叙述本小题的要求:

若在基础文法中增加关于“带条件的继续循环”语句的产生式

$$S \rightarrow \text{continue when } E$$

语句 $\text{continue when } E$ 只能出现在某个循环语句内, 即至少有一个包围它的 while ; 同时 E 必须为 $bool$ 类型。

试在上述翻译模式片段基础上增加相应的语义处理内容 (要求是 L -翻译模式), 以实现针对“带条件的继续循环”语句的这一类型检查任务。(提示: 可以引入 S 的一个继承属性)

(b) 以下是一个L-翻译模式片断，可以产生控制语句的 TAC 语句序列：

$$\begin{aligned}
 &P \rightarrow D ; \{ S.next := newlabel \} S \{ gen(S.next ':') \} \\
 &S \rightarrow \text{if } \{ E.true := newlabel; E.false := S.next \} E \text{ then} \\
 &\quad \{ S_1.next := S.next \} S_1 \{ S.code := E.code // gen(E.true ':') // S_1.code \} \\
 &S \rightarrow \text{while } \{ E.true := newlabel; E.false := S.next \} E \text{ do} \\
 &\quad \{ S_1.next := newlabel \} S_1 \\
 &\quad \{ S.code := gen(S_1.next ':') // E.code // gen(E.true ':') // \\
 &\quad \quad S_1.code // gen('goto' S_1.next) \} \\
 &S \rightarrow \{ S_1.next := newlabel \} S_1 ; \\
 &\quad \{ S_2.next := S.next \} S_2 \\
 &\quad \{ S.code := S_1.code // gen(S_1.next ':') // S_2.code \}
 \end{aligned}$$

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：继承属性 $E.true$ 和 $E.false$ 分别代表 E 为真和假时控制要转移到的程序位置，即标号；综合属性 $E.code$ 表示对 E 进行求值的 TAC 语句序列；综合属性 $S.code$ 表示对应于 S 的 TAC 语句序列；继承属性 $S.next$ 代表退出 S 时控制要转移到的语句标号。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。

（此外，假设在语法制导处理的分析过程中遇到的冲突问题均可以按照某种原则处理，这里不必考虑基础文法是否 LR 文法。）

下面叙述本小题的要求：

若在基础文法中增加关于“带条件的继续循环”语句的产生式

$$S \rightarrow \text{continue when } E$$

语句 **continue when E** 只能出现在某个循环语句内，即至少有一个包围它的 **while** 语句，其执行语义为：当 E 为真时，跳出直接包含该语句的 **while** 循环体，并回到 **while** 循环的开始处判断循环条件重新执行该循环；当 E 为假时，不做任何事情，转下一条语句。

试在上述 L-翻译模式片段基础上增加针对 **continue when E** 语句的语义处理内容（不改变 L-翻译模式的特征，不考虑“是否出现在 **while** 语句内”的语义检查工作）。

注：可设计引入新的属性或删除旧的属性，必要时给出解释。

A4 设有开关语句

$$\begin{aligned}
 &\text{switch } A \text{ of} \\
 &\quad \text{case } d_1 : S_1 ; \\
 &\quad \text{case } d_2 : S_2 ; \\
 &\quad \dots\dots \\
 &\quad \text{case } d_n : S_n ; \\
 &\quad \text{default } S_{n+1}
 \end{aligned}$$

end

假设其具有如下执行语义：

- (1) 对算术表达式 A 进行求值；
- (2) 若 A 的取值为 d_1 ，则执行 S_1 ，转 (3)；
否则，若 A 的取值为 d_2 ，则执行 S_2 ，转 (3)；
.....
否则，若 A 的取值为 d_n ，则执行 S_n ，转 (3)；
否则，执行 S_{n+1} ，转 (3)；
- (3) 结束该开关语句的执行。

若在基础文法中增加关于开关语句的下列产生式

$$\begin{aligned} S &\rightarrow \text{switch } A \text{ of } L \text{ end} \\ L &\rightarrow \text{case } V : S ; L \\ L &\rightarrow \text{default } S \\ V &\rightarrow d \end{aligned}$$

其中，终结符 d 代表常量，其属性值可由词法分析得到，以 $d.lexval$ 表示。 A 是生成算术表达式的非终结符（对应2.3.1中的 E ）。

试参考 2.3.5节进行控制语句（不含 **break**）翻译的 L -翻译模式片断及所用到的语义函数，给出相应的语义处理部分（不改变 L -翻译模式的特征）。

注：可设计增加新的属性，必要时给出解释。

A5 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：

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

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

$$\text{Push } 3 \parallel \text{Push } 2 \parallel \text{Push } 1 \parallel \text{Plus} \parallel \text{Minus}$$

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：

$$\begin{aligned} A &\rightarrow A_1 + A_2 & \{ A.instr := \dots \} \\ A &\rightarrow A_1 - A_2 & \{ A.instr := \dots \} \\ A &\rightarrow (A_1) & \{ A.instr := A_1.instr \} \\ A &\rightarrow \underline{int} & \{ A.instr := \text{Push } \underline{int}.val \} \end{aligned}$$

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 ：

$$\begin{aligned} S &\rightarrow \underline{id} := E \mid S ; S \\ E &\rightarrow A \mid E \underline{if} B \\ A &\rightarrow \dots \mid \underline{id} \\ B &\rightarrow A > A \mid B \& B \mid !B \mid \underline{true} \mid \underline{false} \end{aligned}$$

终结符 \underline{id} 表示一个变量，用 $\underline{id}.val$ 取得语法符号对应的语义值。算术表达式新增 \underline{id} ，用来读取变量 \underline{id} 的值。赋值语句 $\underline{id} := E$ 表示将表达式 E 的值写入变量 \underline{id} 。条件表达式 $E \underline{if} B$ 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， \underline{true} 为真， \underline{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：

$$\begin{aligned} E &\rightarrow E_1 \underline{if} B & \{ E.instr := \dots \} \\ A &\rightarrow \underline{id} & \{ A.instr := \text{Load } \underline{id}.val \} \\ B &\rightarrow A_1 > A_2 & \{ B.instr := \dots \} \\ B &\rightarrow B_1 \& B_2 & \{ B.instr := \dots \} \\ B &\rightarrow !B_1 & \{ B.instr := \dots \} \\ B &\rightarrow \underline{true} & \{ B.instr := \text{Push } 1 \} \end{aligned}$$

$$\begin{aligned}
B &\rightarrow \underline{\text{false}} & \{ B.\text{instr} &:= \text{Push } 0 \} \\
S &\rightarrow \underline{\text{id}} := E & \{ S.\text{instr} &:= E.\text{instr} \parallel \text{Store } \underline{\text{id}}, \text{val} \} \\
S &\rightarrow S_1 ; S_2 & \{ S.\text{instr} &:= S_1.\text{instr} \parallel S_2.\text{instr} \}
\end{aligned}$$

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。

A6 以下是语法制导生成 TAC 语句的一个 L -属性文法：

$$\begin{aligned}
S &\rightarrow \text{if } E \text{ then } S_1 \\
&\quad \{ E.\text{case} := \text{false}; \\
&\quad \quad E.\text{label} := S.\text{next}; \\
&\quad \quad S_1.\text{next} := S.\text{next}; \\
&\quad \quad S.\text{code} := E.\text{code} \parallel S_1.\text{code} \parallel \text{gen}(S.\text{next} ':') \\
&\quad \} \\
S &\rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2 \\
&\quad \{ E.\text{case} := \text{false}; \\
&\quad \quad E.\text{label} := \text{newlabel}; \\
&\quad \quad S_1.\text{next} := S.\text{next}; \\
&\quad \quad S_2.\text{next} := S.\text{next}; \\
&\quad \quad S.\text{code} := E.\text{code} \parallel S_1.\text{code} \parallel \text{gen}(\text{'goto' } S.\text{next}) \parallel \text{gen}(E.\text{label} ':') \\
&\quad \quad \quad \parallel S_2.\text{code} \parallel \text{gen}(S.\text{next} ':') \\
&\quad \} \\
S &\rightarrow \text{while } E \text{ do } S_1 \\
&\quad \{ E.\text{case} := \text{false}; \\
&\quad \quad E.\text{label} := S.\text{next}; \\
&\quad \quad S_1.\text{next} := \text{newlabel}; \\
&\quad \quad S.\text{code} := \text{gen}(S_1.\text{next} ':') \parallel E.\text{code} \parallel S_1.\text{code} \parallel \text{gen}(\text{'goto' } S_1.\text{next}) \parallel \text{gen}(S.\text{next} ':') \\
&\quad \} \\
S &\rightarrow S_1 ; S_2 \\
&\quad \{ S_1.\text{next} := \text{newlabel}; \\
&\quad \quad S_2.\text{next} := S.\text{next}; \\
&\quad \quad S.\text{code} := S_1.\text{code} \parallel S_2.\text{code} \\
&\quad \} \\
E &\rightarrow E_1 \text{ or } E_2 \\
&\quad \{ E_2.\text{label} := E.\text{label}; \\
&\quad \quad E_2.\text{case} := E.\text{case}; \\
&\quad \quad E_1.\text{case} := \text{true}; \\
&\quad \quad \text{if } E.\text{case} \{ \\
&\quad \quad \quad E_1.\text{label} := E.\text{label}; \\
&\quad \quad \quad E.\text{code} := E_1.\text{code} \parallel E_2.\text{code} \} \\
&\quad \}
\end{aligned}$$

```

else {
    E1.label := newlabel ;
    E.code := E1.code || E2.code || gen(E1.label ':') }
}

```

```

E → E1 and E2
{ E2.label := E.label ;
  E2.case := E.case ;
  E1.case := false ;
  if E.case {
      E1.label := newlabel ;
      E.code := E1.code || E2.code || gen(E1.label ':') }
  else {
      E1.label := E.label;
      E.code := E1.code || E2.code }
}

```

```

E → not E1
{ E1.label := E.label;
  E1.case := not E.case;
  E.code := E1.code
}

```

```

E → (E1)
{ E1.label := E.label;
  E1.case := E.case;
  E.code := E1.code
}

```

```

E → id1 rop id2
{
    if E.case {
        E.code := gen('if'  id1.place rop.op id2.place 'goto' E.label)  }
    else {
        E.code := gen('if'  id1.place rop.not-op id2.place 'goto' E.label)  }
}
// 这里，rop.not-op 是 rop.op 的补运算，例=和≠，< 和 ≥ ， > 和 ≤ 互为补运算

```

```

E → true
{
    if E.case {
        E.code := gen( 'goto' E.label)  }
}

```

```

E → false
{

```

```

    if not E.case {
        E.code := gen( 'goto' E.label)  }
}

```

其中，属性 $S.code$, $E.code$, $S.next$, 语义函数 $newlabel$, gen , 以及所涉及到的TAC 语句与讲稿中一致，“//”表示TAC语句序列的拼接；如下是对属性 $E.case$ 和 $E.label$ 的简要说明：

$E.case$ ：取逻辑值 **true** 和 **false**之一（**not** 是相应的“非”逻辑运算）

$E.label$ ：布尔表达式 E 的求值结果为 $E.case$ 时，应该转去的语句标号

（此外，假设在语法制导处理过程中遇到的二义性问题可以按照某种原则处理（比如规定优先级，else 匹配之前最近的 if，运算的结合性，等等），这里不必考虑基础文法的二义性。）

(a) 若在基础文法中增加产生式 $E \rightarrow E \uparrow E$ ，其中“ \uparrow ”代表“与非”逻辑运算符，试参考上述布尔表达式的处理方法，给出相应的语义处理部分。

注：“与非”逻辑运算的语义可用其它逻辑运算定义为 $P \uparrow Q \equiv \text{not} (P \text{ and } Q)$

(b) 若在基础文法中增加产生式 $S \rightarrow \text{repeat } S \text{ until } E$ ，试参考上述控制语句的处理方法，给出相应的的语义处理部分。

注：repeat <循环体> until <布尔表达式> 至少执行<循环体>一次，直到<布尔表达式>成真时结束循环