第 9 讲书面作业包括两部分。第一部分为 Lecture09.pdf 中课后作业题目中的第 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)。

参考解答：

$E \rightarrow$ { E_1.false := E.false; E_1.true := *newlabel* ;} E_1 ?
{ E_2. false:= E.false ; E_2. true:= *newlabel* ; } E_2 :
{ E_3.true := E.false ; E_3.false := E.true ; } E_3
{ E.*code* := E_1.*code* || *gen* (E_1.true ':') || E_2.*code*
|| *gen* (E_2.true ':') || E_3.*code* }

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。

参考解答：

$E \rightarrow E_1$? M_1 E_2 : M_2 E_3 { *backpatch*(E_1.*truelist*,M_1.*gotostm*) ;
backpatch(E_2.*truelist*,M_2.*gotostm*) ;
E.*truelist* := E_3 . *falselist* ;
E.*falselist* := *merge*(*merge*(E_1.*falselist*, E_2.*falselist*), E_3 .
truelist) ;
}

A3　在 2.3.5 节所讨论的编程语言基础上，新增 do 语句。

S → do { W }
W → E : S　W

对于形如

do {
E_1 : S_1
E_2 : S_2
……
E_n : S_n
}

的 do 语句，其语义为：

(1) 令 i=1；
(2) 判断 E_i 是否为真，若是，则执行对应的 S_i，然后转(3)， 否则，直接转(3)；
(3) 令 i=i+1，若 i>n 则退出，否则回到(2)。

请写出一个 *L*-翻译模式片断，可以产生 do 语句的 TAC 语句序列。

参考解答：

$S \rightarrow$ do{ { *W.next* := *S.next* } *W* }

$W \rightarrow$ { *E.true* := *newlabel*; *E.false* := *newlabel* } *E* ：

{ *S.next* := *E.false* } *S* { W_1*.next* := *W.next* } W_1

{ *W.code* := *E.code* || *gen*(*E.true* ‘:’) || *S.code* ||*gen*(*E.false* ‘:’) || W_1*.code* }

A4 在广泛使用的开源工具 LLVM 中，采用三段式编译方式：

c/C++/rust -> LLVM IR -> RISCV/X86/ARM machine code

前端将各种语言翻译成 SSA 格式的 LLVM IR。在 LLVM IR 上进行一系列的机器无关优化后，将 IR 翻译为 不同的后端指令。

为了能够将 IR 翻译为不同的架构的机器指令，LLVM 首先会将 LLVM IR 转化为 SelectionDAG，将 IR 中的操作和变量在 DAG 中的不同的节点表示，之后再在 SelectionDAG 上进一步地对操作进行指令选择，对变量进行寄存器分配。

考察 LLVM IR 语法的缩略版，假设有如下描述 IR 命令的文法：

Instruction→ AddInst | SDivInst

AddInst → "add" Value1 "," Value2 // 注：Value1 和 Value2 同 Value

SDivInst → "sdiv" Value1 "," Value2 // 注：Value1 和 Value2 同 Value

Value → IntVal | FloatVal | Null

试依据下面的描述，给出将其翻译为 DAG 图的翻译模式片段：

1) DAG 图中的每个语法节点包括两个属性，type 和 node。

2) type 代表当前节点的类型，可能是 int， float 或 null, type 属性用来进行类型检查。

3) node 代表一个操作，以及该操作对应的操作数，其格式为(Opcode, Op1...)，即（操作码 ,操作数 1,...)。getNode(OpCode, Op1, ...) 函数可以用来生成 DAG node。

4) 生成一个变量的操作码是 def，唯一操作数是其值所 Val 对应的变量（表示为 Val.var）或 null。

5) AddInst 的操作码是 add，SDivInst 的操作码为 sdiv。Add 指令语义为：将 Value1 和 Value2 的值进行相加。SDiv 指令语义为：将 Value1 的值除以 Value2。两条指令的操作数的类型不能是 null。

6) 错误节点的操作码为 error，操作数维持不变。错误节点可用 errornode 表示。

要求：翻译模式要进行类型检查，通过类型检查后生成对应的 DAG 图。add 指令以及 sdiv 指令的两个操作数类型可以是 float 或者 int 类型（可以不相同，但不可一是不能是 null）。

参考答案：

Instruction → AddInst { return if AddInst.node!= errornode then AddInst.node
else error ; }
| SDivInst { return if SDivInst.node!= errornode then SDivInst.node
else error ; }

AddInst → "add" Value1 "," Value2 { if (Value1.type = int or Value1.type = float) and (Value2.type = int or Value2.type = float)
Then AddInst.node := getNode(add, Value1.node, Value2.node)
else AddInst.node := getNode(error, Value1.node, Value2.node) }

SDivInst → "sdiv" Value1 "," Value2 { if (Value1.type = int or Value1.type = float) and (Value2.type = int or Value2.type = float)
then SDivInst.node := getNode(sdiv, Value1.node, Value2.node)
else SDivInst.node := getNode(error, Value1.node, Value2.node) }

Value → IntVal {Value.type := int ; Value.node := getNode(def, IntVal.var)}
| FloatVal {Value.type := float ; Value.node := getNode(def, FloatVal.var)}
| Null {Value.type := null ; Value.node = getNode(def, null)}

或者

Instruction → AddInst { return if AddInst.node!= errornode then AddInst.node
else error ; }
| SDivInst { return if SDivInst.node!= errornode then SDivInst.node
else error ; }

AddInst → "add" Value1 "," Value2 { if (Value1.type != null and Value2.type != null)
Then
AddInst.node := getNode(add, Value1.node, Value2.node)
else AddInst.node := getNode(error, Value1.node, Value2.node) }

SDivInst → "sdiv" Value1 "," Value2 { if (Value1.type != null and Value2.type != null)
then SDivInst.node := getNode(sdiv, Value1.node, Value2.node)
else SDivInst.node := getNode(error, Value1.node, Value2.node) }

Value → IntVal {Value.type := int ; Value.node := getNode(def, IntVal.var)}
| FloatVal {Value.type := float ; Value.node := getNode(def, FloatVal.var)}
| Null {Value.type := null ; Value.node = getNode(def, null)}

A5 简单类型λ演算 (Simply-typed λ-calculus)

语法 简单类型λ演算的**项** (term) 定义如下

$t ::= x \mid \lambda x{:}T.\, t \mid t_1\ t_2 \mid \text{true} \mid \text{false} \mid \text{ite}(t_1, t_2, t_3)$

其中x为变量，本题用符号$x, x_1, x_2, \ldots, y, y_1, y_2, \ldots$ 表示具体的变量。$\lambda x{:}T.\, t$被称为λ抽象。

$t_1\ t_2$被称为λ应用，遵守左结合：$t_1\ t_2\ t_3 = ((t_1\ t_2)\ t_3)$。true和false为常量。$\text{ite}(t_1, t_2, t_3)$

相当于 C++中的三目运算表达式 $t_1 ? t_2 : t_3$。

此外，T被称为**类型** (type)，本题仅引入Bool类型和箭头类型：

$T ::= \text{Bool} \mid T_1 \to T_2$

注：下标仅用来区分同一个非终结符出现在不同位置，如t, t_1, t_2都代表“项”这一个非终结符。上述文法仅t和T是非终结符。

值和上下文　规定下列项为**值** (value)：①true；②false；③任意λ抽象。用谓词$\mathrm{Value}(t)$表示项t是值。

定义**上下文** (context) 为变量到值的映射σ，我们用$\sigma(x)$取得变量x对应的值。若σ中不含有变量x，那么$\sigma(x) = \bot$。例如，给定上下文$\sigma = \{x_1 \mapsto \mathsf{true}, x_2 \mapsto \lambda x{:}\,\mathsf{Bool}.\, x\}$，那么$\sigma(x_1) = \mathsf{true}$，$\sigma(x_2) = \lambda x{:}\,\mathsf{Bool}.\, x$，但$\sigma(x_3) = \bot$。

语义　定义项上的二元关系 (binary relation) $t_1 \Longrightarrow t_2$，读作“t_1可**一步归约**到t_2”。规则如下：

(1) 值不可归约：若$\mathrm{Value}(t)$，则不存在t'使得$t \Longrightarrow t'$。

(2) 变量x可一步归约到它在上下文σ中对应的值（如果存在），该规则记为

$$\text{Var}\frac{\sigma(x) \neq \bot}{x \Longrightarrow \sigma(x)}$$

横线上为前提（条件），横线下为结论，横线左方为规则的名称。规则 Var 说，若$\sigma(x) \neq \bot$，则$x \Longrightarrow \sigma(x)$。

(3) λ应用$t_1\ t_2$的规则：

$$\text{App-1}\frac{t_1 \Longrightarrow t_1'}{t_1\ t_2 \Longrightarrow t_1'\ t_2} \qquad \text{App-2}\frac{\mathrm{Value}(t_1)\quad t_2 \Longrightarrow t_2'}{t_1\ t_2 \Longrightarrow t_1\ t_2'} \qquad \text{App-Abs}\frac{\mathrm{Value}\,(t_2)}{(\lambda x{:}T.t)\ \ t_2 \Longrightarrow t[x \mapsto t_2]}$$

规则 App-1 说，如果t_1可以一步归约，那么优先归约t_1；

规则 App-2 说，如果t_1已经归约到了某个值（或本身就已经是值），但是t_2可以一步归约，那么就归约t_2；

规则 App-Abs 说，当t_2也已经归约到了某个值（或本身就已经是值），并且t_1是一个λ抽象$\lambda x{:}T.\,t$，那么执行λ应用。直观来说，λ应用就是所谓的“函数调用”，而λ抽象就是“定义了一个函数”。执行过程就是在“函数体” t中把**自由** (free) 出现的“形参” x统统替换成 “实参” t_2，记作$t[x \mapsto t_2]$，递归定义如下（s代表一个项）：

$$
\begin{array}{rcll}
x[x \mapsto s] & = & s & \\
y[x \mapsto s] & = & y & \text{if } x \neq y \\
(\lambda x{:}T.\,t)[x \mapsto s] & = & \lambda x{:}T.\,t & \\
(\lambda y{:}T.\,t)[x \mapsto s] & = & \lambda y{:}T.\,(t[x \mapsto s]) & \text{if } x \neq y \\
(t_1\ t_2)[x \mapsto s] & = & (t_1[x \mapsto s]\ t_2[x \mapsto s]) & \\
\text{true}[x \mapsto s] & = & \text{true} & \\
\text{false}[x \mapsto s] & = & \text{false} & \\
\text{ite}(t_1, t_2, t_3)[x \mapsto s] & = & \text{ite}(t_1[x \mapsto s], t_2[x \mapsto s], t_3[x \mapsto s]) &
\end{array}
$$

简而言之，自由出现的变量就是没有被声明为“形参”的变量。例如，$\lambda x{:}\,\text{Bool}.\,y$中，$y$是自由变量，但$x$不是自由变量（$x$被$\lambda x{:}\,\text{Bool}$声明为了“形参”）。因此，$(\lambda x{:}\,\text{Bool}.\,y)[y \mapsto s] = \lambda x{:}\,\text{Bool}.\,s$，但$(\lambda x{:}\,\text{Bool}.\,y)[x \mapsto s] = \lambda x{:}\,\text{Bool}.\,y$。

(4)$\text{ite}(t_1, t_2, t_3)$的规则：

$$\text{ITE-1}\ \frac{t_1 \Rightarrow t_1'}{\text{ite}(t_1, t_2, t_3) \Rightarrow \text{ite}(t_1', t_2, t_3)}$$

$$\text{ITE-True}\ \frac{}{\text{ite}(\text{true}, t_2, t_3) \Rightarrow t_2} \qquad \text{ITE-False}\ \frac{}{\text{ite}(\text{false}, t_2, t_3) \Rightarrow t_3}$$

规则 ITE-1 说，如果t_1可以一步归约，那么优先归约t_1；

规则 ITE-True 说，如果t_1归约到了true（或本身就已经是true），那么整个$\text{ite}(\text{true}, t_2, t_3)$就可以一步归约到$t_2$；规则 ITE-False 类似。

一步归约关系可扩展至**多步归约**关系$t_1 \Rightarrow_* t_2$，递归定义如下：

- 任何项t可多步归约（实际是零步）到它自己：$t \Rightarrow_* t$；
- 若$t_1 \Rightarrow t_2$，且$t_2 \Rightarrow_* t_3$，则$t_1 \Rightarrow_* t_3$。

对于任何项t，若$t \Longrightarrow_* t'$且t'不可再归约下去，我们称t'是t的**范式** (normal form)。可以证明，按照上述规则，任何项都有且仅有一个范式。

Q1：给定上下文$\sigma = \{x \mapsto \text{true}\}$，请求出项

$$(\lambda x{:}\,\text{Bool} \to \text{Bool}.\ \lambda y{:}\,\text{Bool}.\ (x\ y))\ (\lambda y{:}\,\text{Bool}.\ \text{ite}(y, \text{false}, \text{true}))\ x$$

的范式，要求写出每一步归约的过程。

Q2：并非所有的范式都是值，请举例说明。

Q3：用 *L*-翻译模式可以写出求范式的过程：

$$\begin{array}{lll}
t ::= & x & \{\, t.\text{nf} := \text{if } \sigma(x) = \bot \text{ then } x \text{ else } \sigma(x) \,\} \\
\mid & \lambda x{:}\,T.\,t_1 & \{\, \ldots \,\} \\
\mid & t_1\ t_2 & \{\, \ldots \,\} \\
\mid & \text{true} & \{\, t.\text{nf} := \text{true} \,\} \\
\mid & \text{false} & \{\, t.\text{nf} := \text{false} \,\} \\
\mid & \text{ite}(t_1, t_2, t_3) & \{\, \ldots \,\}
\end{array}$$

其中综合属性$t.\text{nf}$记录项t的范式。请补全空缺部分的伪代码。提示：可以使用之前定义过的$t[x \mapsto s]$。

类型推导　　定义**类型上下文** (typing context) 为变量到类型的映射Γ，我们用$\Gamma(x)$取得变量x对应的类型。若Γ中不含有变量x，那么$\Gamma(x) = \bot$。$\Gamma[x \mapsto T']$表示对Γ的更新，更新后的类型上下文中，x映射到类型T'。用$\Gamma \vdash t{:}\,T$表示项t在类型上下文Γ中具有类型T，规则如下：

(1) 变量的类型由类型上下文确定：T-Var$\dfrac{\Gamma(x) \neq \bot}{\Gamma \vdash x{:}\,\Gamma(x)}$

(2) λ抽象的类型由指定的“参数”类型T和求出的“函数体”t具有的类型决定，该类型是一个箭头类型$T \to T'$：T-Abs$\dfrac{\Gamma[x \mapsto T] \vdash t{:}\,T'}{\Gamma \vdash (\lambda x{:}\,T.\,t){:}\,T \to T'}$

(3) 对一个λ应用$t_1\ t_2$，仅当t_1具有类型$T \to T'$，且t_2恰好具有类型T时（“实参”类型与“形参”类型一致），$t_1\ t_2$才具有类型T'：T-App$\dfrac{\Gamma \vdash t_1 : T \to T' \quad \Gamma \vdash t_2 : T}{\Gamma \vdash (t_1\ t_2) : T'}$

(4) true和false总具有Bool类型：T-True$\dfrac{}{\Gamma \vdash \text{true} : \text{Bool}}$ T-False$\dfrac{}{\Gamma \vdash \text{false} : \text{Bool}}$

(5) 当t_1具有Bool类型，且t_2和t_3的类型一致，都为T时，$\text{ite}(t_1, t_2, t_3)$也具有类型$T$：

T-ITE$\dfrac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{ite}(t_1, t_2, t_3) : T}$

Q4：用 *L*-翻译模式可以写出类型推导的过程：

t	::=	x	$\{ t.\text{ty} := \text{if } \Gamma(x) = \bot \text{ then } \bot \text{ else } \Gamma(x) \}$
	|	$\lambda x{:}\,T$	$\{ \dots \}$
		$.\,t_1$	$\{ \dots \}$
	|	$t_1\ t_2$	$\{ \dots \}$
	|	true	$\{ t.\text{ty} := \text{Bool} \}$
	|	false	$\{ t.\text{ty} := \text{Bool} \}$
	|	$\text{ite}(t_1, t_2, t_3)$	$\{ \dots \}$

其中综合属性$t.\text{ty}$记录项t的范式，$t.\Gamma$表示与项t所在的类型上下文。若根据上述规则无法得知项t的类型，记$t.\text{ty} = \bot$。请补全空缺部分的伪代码。

在空类型上下文中，若项t具有某个类型T，则称t well-typed；反之称 ill-typed。可以证明，对于任何 well-typed 项t，其范式t'一定是值。该性质表明，我们在求某个项的范式之前，如果先判断它是否 well-typed，就能提前把那些归约不出值的项给排除掉。但遗憾地是，我们也同时把一些本身可以归约到值的项判断为 ill-typed。

Q5：请举出一个 ill-typed 项t，其范式t'是值。

参考解答　　（朱俸民）

Q1：根据规则，λ抽象中的“函数体”部分不能先归约，要先归约“实参”，并将其代入到“函数体”。直至当前项不是一个λ应用时，才能开始归约“函数体”部分。

$$
\begin{array}{lll}
 & (\lambda x{:}\,\mathrm{Bool} \to \mathrm{Bool}.\ \lambda y{:}\,\mathrm{Bool}.\ (x\ y))\ (\lambda y{:}\,\mathrm{Bool}.\ \mathrm{ite}(y, \mathrm{false}, \mathrm{true}))\ x & \\
\Longrightarrow & \Big(\lambda y{:}\,\mathrm{Bool}.\ \big((\lambda y{:}\,\mathrm{Bool}.\ \mathrm{ite}(y, \mathrm{false}, \mathrm{true}))\ y\big)\Big)\ x & \text{by App-Abs} \\
\Longrightarrow & \Big(\lambda y{:}\,\mathrm{Bool}.\ \big((\lambda y{:}\,\mathrm{Bool}.\ \mathrm{ite}(y, \mathrm{false}, \mathrm{true}))\ y\big)\Big)\ \mathrm{true} & \text{by Var} \\
\Longrightarrow & (\lambda y{:}\,\mathrm{Bool}.\ \mathrm{ite}(y, \mathrm{false}, \mathrm{true}))\ \mathrm{true} & \text{by App-Abs} \\
\Longrightarrow & \mathrm{ite}(\mathrm{true}, \mathrm{false}, \mathrm{true}) & \text{by App-Abs} \\
\Longrightarrow & \mathrm{false} & \text{by ITE-True}
\end{array}
$$

Q2：举出一个根据规则无法再进行归约，但又不是值的项，如

$\mathrm{ite}(\lambda x{:}\,\mathrm{Bool}.\ x, \mathrm{false}, \mathrm{true})$

Q3：完整的语义函数如下

$$
\begin{array}{lll}
t ::= & x & \{\, t.\mathrm{nf} := \text{if } \sigma(x) = \bot \text{ then } x \text{ else } \sigma(x) \,\} \\
\mid & \lambda x{:}\,T.\ t_1 & \{\, t.\mathrm{nf} := \lambda x{:}\,T.\ t \,\} \\
\mid & t_1\ t_2 & \{\, t.\mathrm{nf} := \text{ if } t_1.\mathrm{nf} = \lambda x{:}\,T.\ t \text{ and } \mathrm{Value}(t_2) \text{ then } t[x \mapsto t_2] \text{ else } \bot \,\} \\
\mid & \mathrm{true} & \{\, t.\mathrm{nf} := \mathrm{true} \,\} \\
\mid & \mathrm{false} & \{\, t.\mathrm{nf} := \mathrm{false} \,\} \\
\mid & \mathrm{ite}(t_1, t_2, t_3) & \{\, t.\mathrm{nf} := \text{if } t_1.\mathrm{nf} = \mathrm{true} \text{ then } t_2.\mathrm{nf} \text{ else} \\
 & & \quad \text{if } t_1.\mathrm{nf} = \mathrm{false} \text{ then } t_3.\mathrm{nf} \text{ else } \mathrm{ite}(t_1.\mathrm{nf}, t_2, t_3) \,\}
\end{array}
$$

Q4：完整的语义函数如下

$$
\begin{array}{lll}
t ::= & x & \{\, t.\mathrm{ty} := \text{if } \Gamma(x) = \bot \text{ then } \bot \text{ else } \Gamma(x) \,\} \\
\mid & \lambda x{:}\,T & \{\, t_1.\Gamma := t.\Gamma[x \mapsto T] \,\} \\
 & .\ t_1 & \{\, t.\mathrm{ty} := \text{if } t_1.\mathrm{ty} = \bot \text{ then } \bot \text{ else } T \to t_1.\mathrm{ty} \,\} \\
\mid & t_1\ t_2 & \{\, t.\mathrm{ty} := \text{if } t_1.\mathrm{ty} = \bot \text{ or } t_2.\mathrm{ty} = \bot \text{ then } \bot \text{ else} \\
 & & \quad \text{if } t_1.\mathrm{ty} = t_2.\mathrm{ty} \mapsto T \text{ then } T \text{ else } \bot \,\} \\
\mid & \mathrm{true} & \{\, t.\mathrm{ty} := \mathrm{Bool} \,\} \\
\mid & \mathrm{false} & \{\, t.\mathrm{ty} := \mathrm{Bool} \,\} \\
\mid & \mathrm{ite}(t_1, t_2, t_3) & \{\, t.\mathrm{ty} := \text{if } t_1.\mathrm{ty} = \mathrm{Bool} \text{ and } t_2.\mathrm{ty} = t_3.\mathrm{ty} \text{ then } t_2.\mathrm{ty} \text{ else } \bot \,\}
\end{array}
$$

Q5：如$\mathrm{ite}(\mathrm{true}, \mathrm{true}, \lambda x{:}\,\mathrm{Bool}.\ x)$。注：在 C++中，该项相当于

```
true ? true : [](auto x) { return x; }
```

该表达式无法通过 C++编译器的类型检查。但是，在 Python 等编译期不会检查类型的语言中，表达式

```
True if True else lambda x: x
```

可以编译通过，且不会报运行时错误。

以下是 Lecture09 文档中的题目

---

5. 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（不含 break）翻译的 *S*-翻译模式片断及所用到的语义函数。设在该翻译模式基础上增加下列两条产生式及相应的语义动作集合：

$S \rightarrow S'$ { *S.nextlist := S'.nextlist* }

$S' \rightarrow id := E'$ { *S'.nextlist* := *"; emit (id .place ':='E'. place)* }

其中，*E'* 是生成算术表达式的非终结符（对应 2.3.1 中的 *E*）。若在基础文法中增加对应 for-循环语句的产生式 $S \rightarrow for\ (S'; E\ ;\ S')\ S$，试给出相应该产生式的语义动作集合。

注：for-循环语句的控制语义类似 C 语言中的for-循环语句。

参考解答：

$S \rightarrow for\ (\ S'_1\ ;\ M_1\ E\ ;\ M_2\ S'_2\ N_1\ )\ M_3\ S_1\ N_2$

{

backpatch(E.truelist, M_3.gotostm) ;

backpatch(S_1.nextlist, M_2.gotostm) ;

backpatch(S'_1.nextlist, M_1.gotostm) ;

backpatch(S'_2.nextlist, M_1.gotostm) ;

backpatch(N_1.nextlist, M_1.gotostm) ;

backpatch(N_2.nextlist, M_2.gotostm) ;

S.nextlist := E.falselist;

}

10. 以下是语法制导生成 *TAC* 语句的一个 *L*-属性文法：

$S \rightarrow if\ E\ then\ S_1$

{ *E .case := **false** ;*

E .label:= S .next ;

S_1.next := S .next ;

S .code := E .code || S_1 .code || gen(S .next ':')

}

$S \rightarrow if\ E\ then\ S_1\ else\ S_2$

{ *E .case := **false** ;*

E .label:= newlabel;

S_1 .next := S .next ;

S_2 .next := S .next ;

S .code := E .code || S_1 .code || gen('goto'S.next) || gen(E .label ':')

|| S_2 .code || gen(S .next ':')

}

$S \rightarrow while\ E\ do\ S_1$

{ *E .case* := ***false*** ;
E .label:= *S .next* ;
S_1 *.next* := *newlabel* ;
S .code := *gen*(S_1 *.next* ':') || *E .code* || S_1 *.code* || *gen*('goto' S_1 *.next*) || *gen*(*S .next* ':')
}

$S \rightarrow S_1; S_2$
{ S_1 *.next* := *newlabel* ;
S_2 *.next* := *S .next* ;
S .code := S_1 *.code* || S_2 *.code*
}

$E \rightarrow E_1$ *or* E_2
{ E_2 *.label* := *E .label* ;
E_2 *.case* := *E .case* ;
E_1 *.case* := ***true*** ;
if *E .case* {
E_1 *.label* := *E .label*;
E .code := E_1 *.code* || E_2 *.code* }
else {
E_1 *.label* := *newlabel* ;
E .code := E_1 *.code* || E_2 *.code*|| *gen*(E_1 *.label* ':') }
}

$E \rightarrow E_1$ *and* E_2
{ E_2 *.label* := *E .label* ;
E_2 *.case* := *E .case* ;
E_1 *.case* := ***false*** ;
if *E .case* {
E_1 *.label* := *newlabel* ;
E .code := E_1 *.code* || E_2 *.code*|| *gen*(E_1 *.label* ':') }
else {
E_1 *.label* := *E .label*;
E .code := E_1 *.code* || E_2 *.code* }
}

$E \rightarrow$ not E_1
{ E_1 *.label* := *E .label*;
E_1 *.case* := ***not*** *E .case*;
E .code := E_1 *.code*
}

$E \rightarrow (E_1)$
{ E_1 *.label* := *E .label*;
E_1 *.case* := *E .case*;
E .code := E_1 *.code*
}

$E \rightarrow \underline{id_1}$ rop $\underline{id_2}$

{

if *E .case* {

E .code := gen('if' $\underline{id_1}$*.place* rop.op $\underline{id_2}$*.place* 'goto' *E .label)* }

else {

E .code := gen('if' $\underline{id_1}$*.place* rop.not-op $\underline{id_2}$*.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

$E \rightarrow$ true

{

if *E .case* {

E .code := gen('goto' *E .label)* }

}

$E \rightarrow$ 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"↑"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 $S \rightarrow$ repeat S 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

参考解答：

（*a*）

$E \rightarrow E_1 \uparrow E_2$

{ E_2 *.label := E .label ;*

E_2 .case := **not** E .case ;
E_1 .case := **false** ;
if E .case {
E_1 .label := E .label;
E .code := E_1 .code || E_2 .code }
else {
E_1 .label := newlabel ;
E .code := E_1 .code || E_2 .code|| gen(E_1 .label ':') }
}

（b）

$S \rightarrow$ repeat S_1 until E
{ S_1 .next := newlabel ;
E .case := **false** ;
E .label:= S_1 .next ;
S .code := gen(S_1 .next ':') || S_1 .code || E .code || gen(S .next ':')
}