STAGE4 实验报告

致理-数理 1 赵宇峰 2021013376

STEP7

1.实验代码任务

在 namer.py 加入 visitCondExpr，和 if 语句基本一致：

```
def visitCondExpr(self, expr: ConditionExpression, ctx :ScopeStack) -> None:
        expr.cond.accept(self, ctx)
        expr.then.accept(self, ctx)
        expr.otherwise.accept(self, ctx)
```

相应地，模仿 if 语句，在 tacgen 加入：

```
def visitCondExpr(self, expr: ConditionExpression, mv: TACFuncEmitter) -> None:
    # 访问条件部分
    expr.cond.accept(self, mv)

    # 创建跳转标签和临时变量
    skipLabel = mv.freshLabel()
    exitLabel = mv.freshLabel()
    tempValue = mv.freshTemp()

    # 条件分支，如果条件不成立，跳转到 skipLabel
    mv.visitCondBranch(tacop.CondBranchOp.BEQ, expr.cond.getattr("val"), skipLabel)

    # 访问真值部分
    expr.then.accept(self, mv)

    # 赋值真值给临时变量
    mv.visitAssignment(tempValue, expr.then.getattr("val"))

    # 跳转到 exitLabel
    mv.visitBranch(exitLabel)

    # 标记 skipLabel
    mv.visitLabel(skipLabel)

    # 访问假值部分
    expr.otherwise.accept(self, mv)

    # 赋值假值给临时变量
    mv.visitAssignment(tempValue, expr.otherwise.getattr("val"))

    # 标记 exitLabel
    mv.visitLabel(exitLabel)
```

```
    # 将临时变量的值赋给条件表达式
    expr.setattr('val', tempValue)
```

2.思考题

1. 我们的实验框架里是如何处理悬吊 else 问题的？请简要描述。

可以观察 ply_parser.py

```
def p_if_else(p):
    """
    statement_matched : If LParen expression RParen statement_matched Else statement_matched
    statement_unmatched : If LParen expression RParen statement_matched Else statement_unmatched
    """
    p[0] = If(p[3], p[5], p[7])


def p_if(p):
    """
    statement_unmatched : If LParen expression RParen statement
    """
    p[0] = If(p[3], p[5])
```

因此，如果出现 if then else if then else 时，如果将中间的 if then 读取为 statement_unmatched，会因为找不到 If LParen expression RParen statement_unmatched Else 这种语法而报错。换言之，要求了 else 之后一定是 statement_matched : If LParen expression RParen statement_matched Else statement_matched 句型，从而保证了 else 一定与最近的 if 匹配。

2.

在实验要求的语义规范中，条件表达式存在短路现象。即：

```
int main() {
        int a = 0;
        int b = 1 ? 1 : (a = 2);
        return a;
    }
```

会返回 0 而不是 2。如果要求条件表达式不短路，在你的实现中该做何种修改？简述你的思路。

可以在生成 TAC 时改变条件表达式的访问顺序。在访问条件表达式时，顺序访问条件、真值和假值部分，然后根据条件进行跳转。这样，无论条件是否满足，都会执行条件的两个分支，从而实现了不短路的效果。最终，给表达式赋值的时候，根据条件选择相应的值。

STEP8

1. 实验代码任务

依据规范

| 'for' '(' expression? ';' expression? ';' expression? ')' statement

| 'for' '(' declaration expression? ';' expression? ')' statement

在 ply_parser.py 中加入下列内容

```
def p_for_init(p):
    """
    for_init : opt_expression
    for_init : declaration
    """
    p[0] = p[1]


def p_for(p):
    """
    statement_matched : For LParen for_init Semi opt_expression Semi opt_expression RParen statement_matched
    statement_unmatched : For LParen for_init Semi opt_expression Semi opt_expression RParen statement_unmatched
    """
    p[0] = For(p[3], p[5], p[7], p[9])



def p_continue(p):
    """
    statement_matched : Continue Semi
    """
    p[0] = Continue()
```

“namer.py”在语义分析中添加对 For 的分析

(参考 visitBlock)

```
    def visitFor(self, stmt: For, ctx :ScopeStack) -> None:
        """
        1. Open a local scope for stmt.init.
        2. Visit stmt.init, stmt.cond, stmt.update.
        3. Open a loop in ctx (for validity checking of break/continue)
        4. Visit body of the loop.
        5. Close the loop and the local scope.
        """
        with ctx.local()://开启了本地的作用域
            stmt.init.accept(self, ctx)
            stmt.cond.accept(self, ctx)
            stmt.update.accept(self, ctx)
            with ctx.loop():
```

```
                stmt.body.accept(self, ctx)
```
而 visitContinue visitBreak 只需加入这一句：
```
    if not ctx.inLoop():
            raise DecafBreakOutsideLoopError()
```
“tacgen.py”
```
def visitContinue(self, stmt: Continue, mv: TACFuncEmitter) -> None:
        mv.visitBranch(mv.getContinueLabel())
def visitFor(self, stmt: For, mv: TACFuncEmitter) -> None:
    """
    示例：
    _T1 = 0
    _T0 = _T1
    _L1:                                #beginLabel
        _T2 = 5
        _T3 = LT _T0, _T2            # Condition Test
        BEQZ _T3, _L3                 # 如果 _T3 为 0（假），跳转到循环结束标签 _L3
    _L2:                                # loopLabel
        _T4 = 1                         # 将常数 1 赋值给临时变量 _T4
        _T5 = ADD _T0, _T4           # 将 _T0 和 _T4 相加，将结果存储在 _T5 中
        _T0 = _T5                       # 将 _T5 的值赋给循环变量 i，即 i = i + 1
        JUMP _L1                        # 无条件跳转到循环开始标签 _L1
    _L3:                                # breaklabel
    """
    beginLabel = mv.freshLabel()
    loopLabel = mv.freshLabel()
    breakLabel = mv.freshLabel()
    stmt.init.accept(self, mv)        # 访问循环初始化语句，例如初始化循环变量 i
    mv.openLoop(breakLabel, loopLabel)  # 打开循环，设置循环结束标签和循环体标签
    mv.visitLabel(beginLabel)         # 访问循环开始标签
    stmt.cond.accept(self, mv)        # 访问循环条件表达式，例如 i < 5
    if stmt.cond.getattr("val") is not None:
        mv.visitCondBranch(tacop.CondBranchOp.BEQ,                stmt.cond.getattr("val"),
breakLabel)
        # 如果条件成立，跳转到循环结束标签 _L3
    stmt.body.accept(self, mv)        # 访问循环体语句
    mv.visitLabel(loopLabel)          # 访问循环体标签
    stmt.update.accept(self, mv)     # 访问循环更新语句，例如 i = i + 1
    mv.visitBranch(beginLabel)        # 无条件跳转到循环开始标签 _L1，实现循环
    mv.visitLabel(breakLabel)         # 访问循环结束标签
    mv.closeLoop()
```
思考题

将循环语句翻译成 IR 有许多可行的翻译方法，例如 while 循环可以有以下两种翻译方式：

第一种（即实验指导中的翻译方式）：

label BEGINLOOP_LABEL：开始下一轮迭代
cond 的 IR
beqz BREAK_LABEL：条件不满足就终止循环
body 的 IR
label CONTINUE_LABEL：continue 跳到这
br BEGINLOOP_LABEL：本轮迭代完成
label BREAK_LABEL：条件不满足，或者 break 语句都会跳到这儿

第二种：

cond 的 IR

beqz BREAK_LABEL：条件不满足就终止循环
label BEGINLOOP_LABEL：开始下一轮迭代
body 的 IR
label CONTINUE_LABEL：continue 跳到这
cond 的 IR
bnez BEGINLOOP_LABEL：本轮迭代完成，条件满足时进行下一次迭代
label BREAK_LABEL：条件不满足，或者 break 语句都会跳到这儿

从执行的指令的条数这个角度（label 不算做指令，假设循环体至少执行了一次），请评价这两种翻译方式哪一种更好？

第二种更好，因为在代码末尾就进行了条件是否满足的判断，如果不满足条件就自然地进入下一段代码而避免了跳转指令的执行。

我们目前的 TAC IR 中条件分支指令采用了单分支目标（标签）的设计，即该指令的操作数中只有一个是标签；如果相应的分支条件不满足，则执行流会继续向下执行。在其它 IR 中存在双目标分支（标签）的条件分支指令，其形式如下：

br cond, false_target, true_target
其中 cond 是一个临时变量，false_target 和 true_target 是标签。其语义为：如果 cond 的值为 0（假），则跳转到 false_target 处；若 cond 非 0（真），则跳转到 true_target 处。它与我们的条件分支指令的区别在于执行流总是会跳转到两个标签中的一个。

你认为中间表示的哪种条件分支指令设计（单目标 vs 双目标）更合理？为什么？（言之有理即可）

我认为双目标分支更合理，在编译上提供了很大的灵活度，而且可以通过默认的方式退化成单目标分支，给编译器的设计者多一种选择。双目标分支很有可能能够减少指令的执行。