Stage4 实验报告

姓名：丁俊辉

学号：2021011145

step7

实验过程

在 `namer.py` 中，加入了对 `ConditionExpression` 节点的访问：

```
def visitCondExpr(self, expr: ConditionExpression, ctx: ScopeStack) -> None:
    """
    1. Refer to the implementation of visitBinary.
    """
    expr.cond.accept(self, ctx)
    expr.then.accept(self, ctx)
    expr.otherwise.accept(self, ctx)
```

由于 `ConditionExpression` 的 `otherwise` 分支并不是 optional 的，直接按顺序访问三个子节点即可。

在 `tacgen.py` 中，加入了对 `ConditionExpression` 节点的访问：

```
def visitCondExpr(self, expr: ConditionExpression, mv: TACFuncEmitter) -> None:
    """
    1. Refer to the implementation of visitIf and visitBinary.
    """
    expr.cond.accept(self, mv)
    skipLabel = mv.freshLabel()
    exitLabel = mv.freshLabel()
    temp = mv.freshTemp()
    mv.visitCondBranch(
        tacop.CondBranchOp.BEQ, expr.cond.getattr("val"), skipLabel
    )

    expr.then.accept(self, mv)
    mv.visitAssignment(temp, expr.then.getattr("val"))
    mv.visitBranch(exitLabel)
    mv.visitLabel(skipLabel)

    expr.otherwise.accept(self, mv)
    mv.visitAssignment(temp, expr.otherwise.getattr("val"))
    mv.visitLabel(exitLabel)
    expr.setattr("val", temp)
```

同样，要直接访问三个子节点。除此之外，还要为整个表达式设置一个临时变量，用来存储这个表达式的值。

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。

A:

我们的实现框架中是将 else 与 if 就近匹配的，这是由语法的 production 定义规定出来的。

我认为重点在于 `ply_praser.py` 中 `p_if_else` 的声明：

```
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])
```

不论是 `statement_matched` 还是 `statement_unmatched` ，if 块中必须是 `statement_matched` 而不能是 `statement_unmatched`，这就使得 else 不能跨过就近的 if 和更上方的 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。

A:

在我当前的代码中，tacgen 的逻辑是先生成 branch 指令，然后在两个分支中生成两个表达式的计算，并分别赋值给三目表达式的临时变量，所以如果有分支没有执行到，其中的表达式也不会执行运算。

如果要让表达式不短路，我认为应该将 `expr.then.accept(self, mv)` 和 `expr.otherwise.accept(self, mv)` 提前。这样，就可以先生成计算这两个表达式的代码，然后只在分支中生成赋值的代码，就不会有短路了。

step8

实验过程

首先，在 `lex.py` 中添加 `for` 和 `continue` 保留字：

```
# Reserved keywords
reserved = {
    "return": "Return",
    "int": "Int",
    "if": "If",
    "else": "Else",
    "while": "While",
    "for": "For",
    "break": "Break",
    "continue": "Continue",
}
```

在 `visitor.py` 中，我加入了 `visitFor` 和 `visitContinue`：

```
class Visitor(Protocol[T, U]):  # type: ignore

    ...

    def visitWhile(self, that: While, ctx: T) -> Optional[U]:
        return self.visitOther(that, ctx)

    def visitFor(self, that: For, ctx: T) -> Optional[U]:
        return self.visitOther(that, ctx)

    def visitBreak(self, that: Break, ctx: T) -> Optional[U]:
        return self.visitOther(that, ctx)

    def visitContinue(self, that: Continue, ctx: T) -> Optional[U]:
        return self.visitOther(that, ctx)
```

在 `tree.py` 中加入 `For` 和 `Continue` 两个节点的定义：

```
class For(Statement):
    """
    AST node of for statement.
    """

    def __init__(
        self,
        init: Union[Declaration, Expression],
        cond: Expression,
        update: Expression,
        body: Statement,
    ) -> None:
        super().__init__("for")
        self.init = init
        self.cond = cond
        self.update = update
        self.body = body

    def __getitem__(self, key: int) -> Node:
        return (self.init, self.cond, self.update, self.body)[key]

    def __len__(self) -> int:
```

```
        return 4

    def accept(self, v: Visitor[T, U], ctx: T):
        return v.visitFor(self, ctx)

class Continue(Statement):
    """
    AST node of continue statement.
    """

    def __init__(self) -> None:
        super().__init__("continue")

    def __getitem__(self, key: int) -> Node:
        raise _index_len_err(key, self)

    def __len__(self) -> int:
        return 0

    def accept(self, v: Visitor[T, U], ctx: T):
        return v.visitContinue(self, ctx)

    def is_leaf(self):
        return True
```

加入定义后，就可以在 `ply_praser.py` 中加入 `p_for` 和 `p_continue`。从而使得 parser 可以按照 production 的规定解析出 `For` 和 `Continue` 节点。

```
def p_for(p):
    """
    statement_matched : For LParen opt_expression Semi opt_expression Semi
opt_expression RParen statement_matched
        | For LParen declaration Semi opt_expression Semi opt_expression RParen
statement_matched
    statement_unmatched : For LParen opt_expression Semi opt_expression Semi
opt_expression RParen statement_unmatched
        | For LParen declaration 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()
```

在 `scopestack.py` 中，我更新了 `ScopeStack` 的定义：

```
class ScopeStack:
    def __init__(self, globalscope: Scope) -> None:
        self.globalscope = globalscope
        self.stack: list[Scope] = [globalscope]
        self.loopCount = 0
```

```
    ...

    def openLoop(self) -> None:
        self.loopCount += 1

    def closeLoop(self) -> None:
        self.loopCount -= 1

    def inLoop(self) -> bool:
        return self.loopCount > 0
```

我加入了 `loopCount` 成员，用于记录栈中有几个 scope 是属于循环的。这样，当出现 break 或者 continue 时，就可以判断还有没有就近的 loop 作用域。

在 `namer.py` 中，加入了 while，for，break，continue 的解析：

```
class Namer(Visitor[ScopeStack, None]):

    ...

    def visitWhile(self, stmt: While, ctx: ScopeStack) -> None:
        stmt.cond.accept(self, ctx)
        ctx.openLoop()
        stmt.body.accept(self, ctx)
        ctx.closeLoop()

    def visitFor(self, stmt: For, ctx: ScopeStack) -> None:
        ctx.push(Scope(ScopeKind.LOCAL))
        if stmt.init is not NULL:
            stmt.init.accept(self, ctx)
        if stmt.cond is not NULL:
            stmt.cond.accept(self, ctx)
        if stmt.update is not NULL:
            stmt.update.accept(self, ctx)
        ctx.openLoop()
        stmt.body.accept(self, ctx)
        ctx.closeLoop()
        ctx.pop()
        ctx

    def visitBreak(self, stmt: Break, ctx: ScopeStack) -> None:
        if not ctx.inLoop():
            raise DecafBreakOutsideLoopError()

    def visitContinue(self, stmt: Continue, ctx: ScopeStack) -> None:
        if not ctx.inLoop():
            raise DecafContinueOutsideLoopError()
```

可以检查 break 和 continue 是否在循环中。

在 `tacgen.py` 中，加入 for 和 continue 的 tac 码生成：

```
class TACGen(Visitor[TACFuncEmitter, None]):
```

```
    ...

    def visitContinue(self, stmt: Continue, mv: TACFuncEmitter) -> None:
        mv.visitBranch(mv.getContinueLabel())

    ...

    def visitFor(self, stmt: For, mv: TACFuncEmitter) -> None:
        beginLabel = mv.freshLabel()
        loopLabel = mv.freshLabel()
        breakLabel = mv.freshLabel()
        mv.openLoop(breakLabel, loopLabel)

        if stmt.init is not NULL:
            stmt.init.accept(self, mv)
        mv.visitLabel(beginLabel)
        if stmt.cond is not NULL:
            stmt.cond.accept(self, mv)
            mv.visitCondBranch(tacop.CondBranchOp.BEQ, stmt.cond.getattr("val"),
breakLabel)
        stmt.body.accept(self, mv)
        mv.visitLabel(loopLabel)
        if stmt.update is not NULL:
            stmt.update.accept(self, mv)
        mv.visitBranch(beginLabel)
        mv.visitLabel(breakLabel)
        mv.closeLoop()
```

思考题

1. 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？

A:

第一种实现方式是直接按照 while 循环的语义生成 IR。

对于第二种实现方式，仔细观察可以发现，如果去掉开头的 cond 判断，就是一个 do...while 循环。也就是说，这种 while 循环可以再 do...while 循环上加一个分支判断来实现，即：

```
while(<cond>){
    <body>
}
```

等价于：

```
if(<cond>){
    do{
        <body>
    }while(<cond>)
}
```

如果已经实现了分支和 do...while 循环，那么第二种更容易设计，while 的语法树可以设计成 if 加上 do..while，这样的话 while 和 do..while 就可以一同实现。但是，while 的 IR 可能会更长一些，并且也不是很直观。

相比之下，第一种实现方式的好处在于简单明了，生成的 IR 更加符合 while 的语义。但是，while 和 do...while 要分别实现。

至于哪一个更好，我认为得看编译器具体的需求。如果只为了简便可以选第二种。

2. 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）

A:

双目标跳转灵活，如果想要两个分支都不直接跟在 branch 语句后面，或者执行完 if 跳到其他地方，双目标跳转都可以实现。

但是，双目标跳转指令需要两个用来跳出语句的范围的 jump；相比之下，单目标跳转指令只需要一个这样的 jump。

因此，如果不做优化，单目标跳转指令的性能会更好。并且由于目前的编译器不需要处理太多复杂的跳转，所以至少在 if 语句的实现上，我还是更倾向于使用单目标跳转指令。