stage-4

计11班 周韧平 2021010699

实验内容

step7要求支持条件语句

首先仿照if的实现修改 `Namer` 和 `TACGen` 中的 `visitCondExpr` 函数，区别在于三目运算符不存在像 `if` 一样缺少 `else` 的情况，只需要顺序执行即可

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

    # raise NotImplementedError


```

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

step8要求实现For语句和While语句

首先需要在 `frontend/ast/tree.py` 中模仿 `If` 和 `break` 的实现方式， 手动增加 `For`，`Continue` 节点， 并在 `frontend/ast/visitor.py` 中提供对应的 Vistor 默认函数

```
'''tree.py'''
class For(Statement):
    '''
    AST node of for statement
```

```
    '''

    def __init__(self, init: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
```

```
'''visitor.py'''
def visitFor(self, that: For, ctx: T) -> Optional[U]:
    return self.visitOther(that, ctx)
def visitContinue(self, that: Continue, ctx: T) -> Optional[U]:
    return self.visitOther(that, ctx)
```

由于这部分的文法也没有实现，因此，我们需要在 `lex.py` 中为它们增加保留字，并在 `ply_parser.py` 中实现for和continue的文法，需要注意的是 declaration 和 expression 在这里要区分开来，因为for的第一个部分既有可能是对变量的定义，也有可能仅仅只是一个赋值，这两种操作应该在语法上都被允许

```
def p_for(p):
    """
```

```
    statement_matched : For LParen expression Semi expression Semi
expression RParen statement_matched
    statement_matched : For LParen declaration Semi expression Semi
expression RParen statement_matched
    statement_unmatched : For LParen expression Semi expression Semi
expression RParen statement_unmatched
    statement_unmatched : For LParen declaration Semi expression Semi
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()

```

至于中段，需要仿照IF和Break的实现，在`Namer`中实现`visitFor`和`visitContinue`函数，需要注意的是，由于for自带一个作用域，因此需要模仿visitBlock，打开/关闭对应的作用域。而`visitBreak`和`visitBreak`在发现并不在 loop内时，则抛出对应报错

```
def visitFor(self, stmt: For, ctx: ScopeStack) -> None:
    ctx.open(Scope(ScopeKind.LOCAL))
    stmt.init.accept(self,ctx)
    stmt.cond.accept(self,ctx)
    stmt.update.accept(self,ctx)
    ctx.visit_loop()
    stmt.body.accept(self, ctx)
    ctx.close()
    ctx.close_loop()

def visitBreak(self, stmt: Break, ctx: ScopeStack) -> None:
    """
    You need to check if it is currently within the loop.
    To do this, you may need to check 'visitWhile'.

    if not in a loop:
        raise DecafBreakOutsideLoopError()
    """
    if not ctx.is_inloop():
        raise DecafBreakOutsideLoopError()
    return

def visitContinue(self, stmt: Continue, ctx: ScopeStack)->None:
    """
    def visitContinue(self, stmt: Continue, ctx: Scope) -> None:

    1. Refer to the implementation of visitBreak.
    """
    if not ctx.is_inloop():
        raise DecafBreakOutsideLoopError()
```

此外，由于对于loop的判定通过 ctx 中的计数器实现，因此每进入一层For都要使该计数器加一，反之退出需要减一，由于这一过程也应包括While，因此我在原有的While里面也增加了这一计数器的操作

```
def visitWhile(self, stmt: While, ctx: ScopeStack) -> None:
    stmt.cond.accept(self, ctx)
    ctx.visit_loop()
    stmt.body.accept(self, ctx)
    ctx.close_loop()
```

思考题

> 你使用语言的框架里是如何处理悬吊 else 问题的？请简要描述。

正如实验指导书所述，解决的悬吊else问题在于只让else和最近的if匹配，也就是对于if-if-else结构在没有大括号的情况下必须让else和第二个if匹配，具体来说，框架中规定了这样的文法

```
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])
```

他将if和else下的语句做了区分，分为match和unmatch两种类型，使得if-else的匹配必须满足if下是一个statement_matched而非statement_unmatched，即如果想要按照 **if**-if-**else** 的方式处理，则需要使内层的孤立if和if-else的if块儿相匹配，而文法通过规定 p_if 的产生式左端为 statement_unmatched 从而避免了这一错误匹配，换言之，if-else通过规定if块儿和else块儿下的非终结符一个只能接收matched而另一个则都可以接收，从而使得所有的孤立if都被放到了else块儿下，解决了悬吊if问题

> 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。

要求条件表达式不短路，需要修改tac处理Cond表达式时的处理逻辑，之前实现的逻辑是首先判断条件是否成立，再根据条件跳转至对应的代码块儿执行代码，这一操作逻辑使得当条件不成立时，对应branch内的语句并不会被执行，也就是短路的情况，具体来说，从原代码生成三地址码可以看出由于_T3=0恒为False，因此_T0并不会执行赋值为2的操作，而是直接返回，也就是发生了短路。

```
2021010699@compiler-lab:~/minidecaf-2021010699$ python main.py --input example.c --tac
FUNCTION<main>:
    _T1 = 0
    _T0 = _T1
    _T3 = 1
    if (_T3 == 0) branch _L1
    _T5 = 1
    _T4 = _T5
    branch _L2
_L1:
    _T6 = 2
    _T0 = _T6
    _T4 = _T6
_L2:
    _T2 = _T4
    return _T0
```

为了修改使得表达式不断路，只需要在真正跳转前先完成每一个块儿中的语句，然后再根据条件去决定返回结果逻辑，具体来说，在tacgen中将expr.then.accept和expr.otherwise.accept移到靠前处执行即可

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

可以看到，新的三地直码首先执行了块儿中的操作，然后再根据条件决定返回结果，这样就避免了短路

```
FUNCTION<main>:
    _T1 = 0
    _T0 = _T1
    _T3 = 1
    _T4 = 1
    _T5 = 2
    _T0 = _T5
    if (_T3 == 0) branch _L1
    _T6 = _T4
    branch _L2
_L1:
    _T6 = _T5
_L2:
    _T2 = _T6
    return _T0
```

> 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？

第二种方法更好，在假设每个循环体至少执行一次的情况下，两种方法都会在进入循环体前进行条件判断，然后执行循环体，区别在于第一种会先跳回BEGINLOOP_LABEL再执行条件判断，第二种则会先执行条件判断再决定是否跳回BEGINLOOP_LABEL，那么对于终止那次的迭代，第二种会比第一种少执行一条“跳转至BEGINLOOP_LABEL”的操作，而是直接退出循环（跳转至BREAK_LABEL）而第一种需要先跳回去再执行条件判断发现终止循环再跳转至BREAK_LABEL，因此第二种比第一种执行的代码条数要少一条，翻译的更好

> 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：
>
> ```
> 1 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）

我认为采用双目标分支指令设计更为合理，对程序员编写和维护代码来说，双目标指令能够更直接地表达条件为真和为假时的执行流程，便于程序员维护代码，降低出错的可能性，而对于编译器来说，这样做也可以更容易地对分支进行静态分析，从而应用更多的优化技术，如指令调度、分支预测等，从而提高程序地运行效率