stage-4实验报告

于新雨 计25 2022010841

step7

新增代码

在这一个step中，我们需要增加对条件语句if-else的支持

- namer 按照提示，类比于visitBinary,我们增加代码如下

```
        expr.cond.accept(self, ctx)
        expr.then.accept(self, ctx)
        expr.otherwise.accept(self, ctx)
```

tree.py中，表示条件语句的节点是由cond,then,otherwise(可能为空)组成，我们按顺序访问就可以了

- tacgen tacgen中，我们需要施工ConditiionalExpression的实现，即是三元运算符的实现
  我们主要参考`visitIf`函数实现，先讨论visitIf和visitCondExpr功能上的不同点，主要是在于if的`otherwise`分支可能为空，但是visitCondExpr则不会，此外条件表达式本身可能是有value的属性，就像是`int a=(3>5?6:7)`这样，**所以我们需要给节点增设属性value**
  代码如下

```
        expr.cond.accept(self, mv)
        expr.setattr("val",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(expr.getattr("val"), expr.then.getattr("val"))
        mv.visitBranch(exitLabel)
        mv.visitLabel(skipLabel)
        expr.otherwise.accept(self, mv)
        mv.visitAssignment(expr.getattr("val"), expr.otherwise.getattr("val"))
        mv.visitLabel(exitLabel)
```

- 我们把cond访问了，生成cond的值，并且为表达式本身设置一个临时变量来存值。
- 我们再仿照visitIf的设计生成条件跳转语句
- 我们再生成then和otherwise分支，但是和visitIf不同，我们在用accept生成完后要把值赋给整体表达式的value
- 我们再生成对应的结束标签就行了

- 没有新增riscv指令，后端基本不做改动

思考题

1. 处理dangling-else二义性是在ply-parser中指定 parser需要匹配的pattern来完成 该段代码如下：

```
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])
```

用上下文无关文法，可以一起写成下面这样

```
S->If ( expr ) S Else M
M->If ( expr ) S Else M | If ( expr ) statement
```

由形式语言与自动机的知识可知消除了二义性
2. 可知应该在前端进行修改，在then分支中增加对otherwise分支的访问但是不进行赋值，修改成下面这样：

```
        expr.cond.accept(self, mv)
        expr.setattr("val",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(expr.getattr("val"), expr.then.getattr("val"))
        expr.otherwise.accept(self,mv)
        mv.visitBranch(exitLabel)
        mv.visitLabel(skipLabel)
        expr.otherwise.accept(self, mv)
        mv.visitAssignment(expr.getattr("val"), expr.otherwise.getattr("val"))
        mv.visitLabel(exitLabel)
```

step8

新增代码

- lex 我们需要增加对于for和continue的解析，在保留字中增加以下两行

```
    "for":"For",
    "continue": "Continue",
```

- ply_parser 增加对for和continue语句的token匹配 for语句：

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

def p_for_init(p):
    """
    statement_matched : For LParen declaration Semi opt_expression Semi
opt_expression RParen statement_matched
    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])
```

continue语句生成较为直观

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

注意在实验指导里面给出的文法中，for的init一项可以是declaration，当时debug了很久这个，以及由于for的括号中的三项都是可有可无的，所以都是opt_expression类型

- scopestack 这个step的特殊需求是，在开启一个循环的时候，我们不仅要压入一个新的作用域，更是要记录循环层数，所以我们再搞一个stack叫做loop，同时增加push,pop,top方法，遇到循环时同时压作用域栈和loop

```
    def pushloop(self,scope:Scope):
        self.stack.append(scope)
```

```
        self.loop.append(scope)
    def poploop(self):
        self.stack.pop()
        return self.loop.pop()
    def toploop(self):
        return self.loop[-1]
```

- tree tree中需要我们增加Continue,For两个节点，分别如下设计

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

    def __init__(self, init: Statement, cond: Expression, after: Statement, body:
Statement) -> None:
        super().__init__("for")
        self.init = init
        self.cond = cond
        self.after = after
        self.body = body

    def __getitem__(self, key: int) -> Node:
        return (self.init, self.cond, self.after, 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 break 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
```

都比较简单，仿照其他几个节点实现一下接口就完成了

- visitor visitor.py里面没有visitContinue,我们给它加上

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

- namer 我们按照提示进行for语句生成,先压一个局部作用域，init访问，压一个Loop，访问cond,访问body,loop出栈，访问update，作用域出栈就行，和提示不太一样，但是这样感觉是对的

```
    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.
        ctx.push(Scope(ScopeKind.LOCAL))
        stmt.init.accept(self, ctx)
        ctx.pushloop(Scope(ScopeKind.LOCAL))
        stmt.cond.accept(self, ctx)
        stmt.body.accept(self,ctx)
        ctx.poploop()
        stmt.after.accept(self, ctx)
        ctx.pop()
```

对于break和continue只需检查现在loop栈是否为空

```
    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:

        """
        if not ctx.loop.__len__():
            raise DecafBreakOutsideLoopError()
```

- tacgen continue比较简单，直接加一个branch语句就行

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

我们参照文档中的示例中间码增加代码如下

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

        mv.openLoop(breakLabel, loopLabel)
        stmt.init.accept(self, mv)
        mv.visitLabel(beginLabel)
        stmt.cond.accept(self, mv)
        mv.visitCondBranch(tacop.CondBranchOp.BEQ, stmt.cond.getattr("val"),
breakLabel)
        stmt.body.accept(self, mv)
        mv.visitLabel(loopLabel)
        stmt.after.accept(self,mv)
        mv.visitBranch(beginLabel)
        mv.visitLabel(breakLabel)
        mv.closeLoop()
```

TAC:

```
    _T1 = 0
    _T0 = _T1                 # int i = 0;
_L1:                          # begin label
    _T2 = 5
    _T3 = LT _T0, _T2
    BEQZ _T3, _L3              # i < 5;
_L2:                          # loop label
    _T4 = 1
    _T5 = ADD _T0, _T4
    _T0 = _T5                 # i = i + 1;
    JUMP _L1
_L3:                          # break label
    # 后续指令 ...
```

思路就是：开一个Loop（循环最开始）（有个小技巧就是openLoop的形参是breakLabel和continueLabel,我们continueLabel和looplabel重叠就行啦）->访问初始语句->beginlabel生成->条件判断->条件跳转语句->块语句->loopLabel->after语句->跳转语句->生成breakLabel标记结束

- 没有新增riscv指令，后端基本不做改动

思考题

1. 第二种更好，第一种每一轮会多执行一条branch语句，因为如果设cond是x条语句，body是y条，则如果有n轮循环，方法一是n*(x+y+2)+2,方法二是2+n*(x+y+1)
2. 我认为单分支更合理，因为汇编语句为了利用cache的特性，往往会选择单分支的语句来增加局部性，而且现有分支预测也是往往基于单分支语句的类按顺序执行来处理的，从中间代码转汇编的角度上，用单分支的中间代码转为汇编更加直观。虽然表面上双分支可能可读性更强而且也被LLVM IR用了，但是个人还是决定单分支更加合理，虽然感觉这个问题本质上就是IR应该更贴近源码还是汇编的问题。