stage-4

计13 沈佳茗 2021010745

step 7

一、实验内容

1. 前端

   本阶段前端没有改动

2. 中端

   在 Namer.visitCondExpr 中依次访问 expr.cond，expr.then，expr.otherwise 来检查是否有语义错误。

```
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)
    # raise NotImplementedError
```

   在 TACGen.visitCondExpr 中，仿照 visitIf 当中 `stmt.otherwise != NULL` 的情况，在此基础上新增一个 Temp 用来保存 expr 的值，在访问 expr.then 和 expr.otherwise 之后，添加 exit_Label 之前，增加一句赋值语句，在最后将 Temp 设为 expr 的 val 即可。

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

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

3. 后端

   本阶段后端无需修改。

二、思考题

1. 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。

```
"""
statement_matched : If LParen expression RParen statement_matched Else
statement_matched
statement_unmatched : If LParen expression RParen statement_matched Else
statement_unmatched
"""
"""
statement_unmatched : If LParen expression RParen statement
"""
```

   在 frontend/parser/ply_parser.py 中，statement 分为 statement_matched 和 statement_unmatched 两种，在 if-else 语法中，statement_matched 一定为 具有完整的 if-else 的模式，statement_unmatched 在 if 内的部分一定每个 if 与 else 都完成了配对，if 与 else 中间一定是 statement_matched，所以悬吊的 else（即还没有与 if 配对的 else）就会与最近的没有与 else 配对的 if 结合，以此来解决悬吊 else 的问题。

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。

   在进入判断跳转之前先依次访问 expr.cond，expr.then，expr.otherwise

```
def visitCondExpr(self, expr: ConditionExpression, mv: TACFuncEmitter) -> None:
    """
    1. Refer to the implementation of visitIf and visitBinary.
    """
    # 先依次访问 expr.cond，expr.then，expr.otherwise 完成三个表达式的操作
    expr.cond.accept(self, mv)
    expr.then.accept(self, mv)
    expr.otherwise.accept(self, mv)
    # 再进入判断跳转
    skipLabel = mv.freshLabel()
    exitLabel = mv.freshLabel()
    value = mv.freshTemp()
    mv.visitCondBranch(
        tacop.CondBranchOp.BEQ, expr.cond.getattr("val"), skipLabel
    )
```

```
    mv.visitAssignment(value, expr.then.getattr("val"))
    mv.visitBranch(exitLabel)
    mv.visitLabel(skipLabel)

    mv.visitAssignment(value, expr.otherwise.getattr("val"))
    mv.visitLabel(exitLabel)
    expr.setattr( "val", value)
```

step 8

由于在写 step 8 时，stage-6 中还存在 do while 的测例，所以代码中包含了 do while 的实现。

一、实验内容

1. 前端

   首先在 `frontend/lexer/lex.py` 的保留字 `reserved` 中添加 `"do": "Do", "for": "For","continue": "Continue"`

   随后，在 `frontend/ast/tree.py` 中定义 AST 节点 `DoWhile`，`For`，`Continue` 类，其中 `DoWhile`，`For` 类仿照 `While` 实现， `Continue` 类仿照 `Break` 实现。

   最后，根据语法规范中新增的部分完成 `frontend/parser/ply_parser.py` 中新增语法的部分。

```
def p_dowhile(p):
    """
    statement_matched : Do statement_matched While LParen expression RParen Semi
    statement_unmatched : Do statement_unmatched While LParen expression RParen
Semi
    """
    p[0] = DoWhile(p[7], p[2])

def p_for_init(p):
    """
    for_init : declaration
    for_init : opt_expression
    """
    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_return(p):
    """
    statement_matched : Return expression Semi
```

```
    """
    p[0] = Return(p[2])


def p_expression_statement(p):
    """
    statement_matched : opt_expression Semi
    """
    p[0] = p[1]


def p_block_statement(p):
    """
    statement_matched : LBrace block RBrace
    """
    p[0] = p[2]


def p_break(p):
    """
    statement_matched : Break Semi
    """
    p[0] = Break()

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

2. 中端

在类型检查中，对 For，DoWhile 来说依次检查每个组成部分，需要注意的是在每个 body 开始前需要对 Scopestack 的 loop 加一（由函数 beginloop 完成），在离开 body 后，对 Scopestack 的 loop 减一（由函数 endloop 完成），方便 Break 和 Continue 对是否处于循环内的判断。在检查 for 的元素前还需开启一个新的作用域，在结束的时候将它关上。Break 和 Continue 则只需检查它们是否位于循环内，如果不在循环内则抛出错误。

在三地址码生成中，根据 visitWhile 的实现，调整循环体/循环条件/跳转/跳转标签等部分的顺序完成 `visitDoWhile, visitFor`。在 visitFor 判断条件中需检查 cond 的 val 属性是否为空（None），防止 `for (int i = 0; ; i = i + 1)` 语句生成时 cond 的值实际为 None 在生成 tac 时出现 None 的情况。

比如说对于程序

```
int main() {
    int sum = 0;
    for (int i = 0; ; i = i + 1) {
        sum = sum + 1
    }
    return sum;
}
```

在tac 码中会出现 `if (None == 0) branch _L3` 的语句，在之后后端的处理中就会报 `AttributeError: 'NoneType' object has no attribute 'index'` 的错误。

```
FUNCTION<main>:
    _T1 = 0
    _T0 = _T1
    _T3 = 0
    _T2 = _T3
_L1:
    if (None == 0) branch _L3
    _T4 = 1
    _T5 = (_T0 + _T4)
    _T0 = _T5
    _T6 = 10
    _T7 = (_T0 == _T6)
    if (_T7 == 0) branch _L4
    branch _L3
_L4:
_L2:
    _T8 = 1
    _T9 = (_T2 + _T8)
    _T2 = _T9
    branch _L1
_L3:
    return _T0
```

tacgen.py 中的代码改动如下：

```
def visitDoWhile(self, stmt: DoWhile, mv: TACFuncEmitter) -> None:
    beginLabel = mv.freshLabel()
    loopLabel = mv.freshLabel()
    breakLabel = mv.freshLabel()
    mv.openLoop(breakLabel, loopLabel)

    mv.visitLabel(beginLabel)
    stmt.body.accept(self, mv)
    # mv.visitLabel(loopLabel)

    stmt.cond.accept(self, mv)
    mv.visitCondBranch(tacop.CondBranchOp.BEQ, stmt.cond.getattr("val"),
breakLabel)
    mv.visitLabel(loopLabel)
    mv.visitBranch(beginLabel)
    mv.visitLabel(breakLabel)

    mv.closeLoop()

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

```
    stmt.init.accept(self, mv)

    mv.openLoop(breakLabel, loopLabel)
    mv.visitLabel(beginLabel)

    stmt.cond.accept(self, mv)
    if stmt.cond.getattr("val") is not None:
        mv.visitCondBranch(tacop.CondBranchOp.BEQ, stmt.cond.getattr("val"),
breakLabel)
    stmt.body.accept(self, mv)

    mv.visitLabel(loopLabel)
    stmt.incr.accept(self, mv)
    mv.visitBranch(beginLabel)
    mv.visitLabel(breakLabel)

    mv.closeLoop()
```

3. 后端

   本阶段后端无需修改

二、思考题

1. 第一种翻译方式在从 body 到 cond 时使用了跳转，而第二种翻译方式通过开头插入 cond 指令的方式避免了这次跳转，相当于 从 body->cond 改为了 cond->body。对于相同的一段代码，循环执行到 cond 为假结束时，第二种翻译方式生成的程序会比第一种翻译方式生成的程序少执行 1 条跳转指令，所以第二种方式更好。（以只执行循环体一次为例，第一种翻译方式会执行 `cond 的 IR，beqz BREAK_LABEL，body 的 IR， br BEGINLOOP_LABEL，cond 的 IR，beqz BREAK_LABEL`，而第二种翻译方式会执行`cond 的 IR，beqz BREAK_LABEL，body 的 IR， cond 的 IR，beqz BREAK_LABEL` 少一条 `br BEGINLOOP_LABEL` 指令）
2. 我认为单目标更合理。在程序执行过程中，如果需要跳转，则流水线会被打断，效率下降。假设 CPU 在没有对跳转进行优化的情况下，使用单目标跳转相比双目标跳转可以减少一次流水线的冲刷，效率更高，所以我认为单目标更合理。