Stage-4 Report

曹伦郗 *2020011020*

实验内容

step9

词法分析

在*lex.py*中新增了逗号，用于函数声明和定义中的形参列表和函数调用时的参数列表。

```
t_Comma = ','
```

语法分析

在*tree.py*中：

- 为*Program*类新增方法*functions*，以字典形式返回所有函数节点（定义和声明同时存在时，只返回定义）；
- 完善了函数类*Function*，为其新增了成员形参列表*parameter_list*；
- 新增了形参类*Parameter*，继承*Node*类，新增成员有参数类型*var_t*、参数名称*ident*；
- 新增了函数调用类*Call*，继承*Expression*类，新增成员有函数名称*ident*、参数列表*argument_list*。

```
    def functions(self) -> dict[str, Function]:
        funcs = {}
        for func in self:
            if isinstance(func, Function):
                funcs[func.ident.value] = func
        return funcs
```

```
class Function(Node):
    """
    AST node that represents a function.
    """

    def __init__(
        self,
        ret_t: TypeLiteral,
        ident: Identifier,
        parameter_list: list[Parameter],
        body: Block,
    ) -> None:
        super().__init__("function")
        self.ret_t = ret_t
        self.ident = ident
        self.parameter_list = parameter_list
        self.body = body

    def __getitem__(self, key: int) -> Node:
        return (
            self.ret_t,
```

```
            self.ident,
            self.parameter_list,
            self.body,
        )[key]

    def __len__(self) -> int:
        return 4

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

```
class Parameter(Node):
    """
    AST node that represents a parameter.
    """

    def __init__(self, var_t: TypeLiteral, ident: Identifier) -> None:
        super().__init__('parameter')
        self.var_t = var_t
        self.ident = ident

    def __getitem__(self, key: int) -> Node:
        return (self.var_t, self.ident)[key]

    def __len__(self) -> int:
        return 2

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

```
class Call(Expression):
    """
    AST node of function call expression.
    """

    def __init__(self, ident: Identifier, argument_list: list[Expression]) ->
None:
        super().__init__('call')
        self.ident = ident
        self.argument_list = argument_list

    def __getitem__(self, key: int) -> Node:
        return (self.ident, self.argument_list)[key]

    def __len__(self) -> int:
        return 2

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

而在*Visitor*类下添加了*visitFunction*、*visitParameter*、*visitCall*。

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

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

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

在*ply_parser.py*中：

- 修改了*p_program*，新增了*p_program_one*，以实现多个函数，产生式为

  ```
  program : function | program function
  ```

  ```
  def p_program(p):
      """
      program : program function
      """
      p[1].children.append(p[2])
      p[0] = p[1]


  def p_program_one(p):
      """
      program : function
      """
      p[0] = Program(p[1])
  ```

- 新增了*p_function_decl*，修改了*p_function_def*，添加了形参列表，产生式为

  ```
  function : type Identifier LParen parameter_list RParen LBrace block RBrace
  | type Identifier LParen parameter_list RParen Semi
  ```

  ```
  def p_function_decl(p):
      """
      function : type Identifier LParen parameter_list RParen Semi
      """
      p[0] = Function(p[1], p[2], p[4], NULL)


  def p_function_def(p):
      """
      function : type Identifier LParen parameter_list RParen LBrace block
  RBrace
      """
      p[0] = Function(p[1], p[2], p[4], p[7])
  ```

- 新增了*p_parameter_list*、*p_parameter_list_empty*、*p_parameter_list_nonempty*、*p_parameter_list_one*，以产生不同个数的形参列表，产生式为

```
parameter_list : parameter_list_nonempty | empty
parameter_list_nonempty : parameter_list_nonempty Comma parameter |
parameter
```

```
def p_parameter_list(p):
    """
    parameter_list : parameter_list_nonempty
    """
    p[0] = p[1]


def p_parameter_list_emtpy(p):
    """
    parameter_list : empty
    """
    p[0] = []


def p_parameter_list_nonempty(p):
    """
    parameter_list_nonempty : parameter_list_nonempty Comma parameter
    """
    p[1].append(p[3])
    p[0] = p[1]


def p_parameter_list_one(p):
    """
    parameter_list_nonempty : parameter
    """
    p[0] = [p[1]]
```

- 新增了*p_parameter*，产生式为

```
parameter : type Identifier
```

```
def p_parameter(p):
    """
    parameter : type Identifier
    """
    p[0] = Parameter(p[1], p[2])
```

- 新增了*p_call*，产生式为

```
postfix : Identifier LParen argument_list RParen
```

```
def p_call(p):
    """
    postfix : Identifier LParen argument_list RParen
    """
    p[0] = Call(p[1], p[3])
```

- 新增了*p_argument_list*、*p_argument_list_empty*、*p_argument_list_nonempty*、*p_argument_list_one*，以实现不同个数的参数列表，产生式为

```
argument_list : argument_list_nonempty | empty
argument_list_nonempty : argument_list_nonempty Comma expression |
expression
```

```
def p_argument_list(p):
    """
    argument_list : argument_list_nonempty
    """
    p[0] = p[1]


def p_argument_list_empty(p):
    """
    argument_list : empty
    """
    p[0] = []


def p_argument_list_nonempty(p):
    """
    argument_list_nonempty : argument_list_nonempty Comma expression
    """
    p[1].append(p[3])
    p[0] = p[1]


def p_argument_list_one(p):
    """
    argument_list_nonempty : expression
    """
    p[0] = [p[1]]
```

语义分析

修改了*Namer.visitProgram*，以访问各个函数，而不只访问*main*函数。

```
    def visitProgram(self, program: Program, ctx: ScopeStack) -> None:
        # Check if the 'main' function is missing
        if not program.hasMainFunc():
            raise DecafNoMainFuncError
        for func in program:
            func.accept(self, ctx)
```

为*FuncSymbol*新增了成员*defined*，修改了*Namer.visitFunction*，根据*Function*节点的函数体成员*body*是否为空，分为函数声明或函数定义两种：

- 声明：

  ①检查所声明的函数是否有同名函数已经声明，若已声明且参数列表不同，抛出*DecafDeclConflictError*，若已声明但参数列表相同，即函数重复声明，不再继续执行；

②为声明的新函数创建函数符号，且成员*defined*设置为*False*，代表该函数尚未被定义，添加到符号表内；

- 定义：

  ①检查所定义的函数是否有同名函数已经声明：

  若已声明：若已定义或参数列表不同，抛出*DecafDeclConflictError*，否则将该函数符号的成员*defined*设置为*True*；

  若未声明，则为定义的新函数创建函数符号，且成员*defined*设置为*True*，代表该函数已被定义，添加到符号表内；

  ②开启新的局部作用域；

  ③依次访问形参列表中的每个形参；

  ④访问函数体；

  ⑤关闭局部作用域。

```
class FuncSymbol(Symbol):
    def __init__(self, name: str, type: DecafType, scope: Scope, defined: bool)
-> None:
        super().__init__(name, type)
        self.scope = scope
        self.para_type = []
        self.defined = defined
```

```
    def visitFunction(self, func: Function, ctx: ScopeStack) -> None:
        if func.body != NULL: # defination
            declaredFuncSymbol = ctx.findConflict(func.ident.value)
            if declaredFuncSymbol: # declared
                if declaredFuncSymbol.defined or len(func.parameter_list) !=
declaredFuncSymbol.parameterNum:
                    # declared and defined or conflict declaration
                    raise DecafDeclConflictError(func.ident.value)
                # declared but not defined
                declaredFuncSymbol.defined = True
            else: # not declared, declare and define at the same time
                funcSymbol = FuncSymbol(func.ident.value, func.ret_t.type,
ctx.currentScope(), True)
                for param in func.parameter_list:
                    funcSymbol.addParaType(param.var_t)
                ctx.declare(funcSymbol)
            ctx.open(Scope(ScopeKind.LOCAL))
            for param in func.parameter_list:
                param.accept(self, ctx)
            func.body.func_body = True
            func.body.accept(self, ctx)
            ctx.close()
        else: # declaration
            declaredFuncSymbol = ctx.findConflict(func.ident.value)
            if declaredFuncSymbol: # declared
                if len(func.parameter_list) != declaredFuncSymbol.parameterNum:
                    raise DecafDeclConflictError(func.ident.value)
                return # multi-declaration
            # first declaration
```

```
            funcSymbol = FuncSymbol(func.ident.value, func.ret_t.type,
ctx.currentScope(), False)
            for param in func.parameter_list:
                funcSymbol.addParaType(param.var_t)
            ctx.declare(funcSymbol)
```

修改了*Namer.visitBlock*。

由于访问形参要在函数所开启的局部作用域中进行，以使得形参的变量符号声明在该作用域的符号表内，因此访问函数体时，不能为这个*block*再额外开启一个局部作用域。

故为*Block*类新增一成员*func_body*，以表示该*block*语句是否是某函数的函数体，从而选择是否需要开启新的局部作用域。

```
class Block(Statement, ListNode[Union["Statement", "Declaration"]]):
    """
    AST node of block "statement".
    """

    def __init__(self, *children: Union[Statement, Declaration], func_body: bool
= False) -> None:
        super().__init__("block", list(children))
        self.func_body = func_body

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

    def is_block(self) -> bool:
        return True
```

```
    def visitBlock(self, block: Block, ctx: ScopeStack) -> None:
        if block.func_body:
            for child in block:
                child.accept(self, ctx)
            return
        ctx.open(Scope(ScopeKind.LOCAL))
        for child in block:
            child.accept(self, ctx)
        ctx.close()
```

新增了*Namer.visitParameter*，即：

①检查所声明的形参是否被重复声明，若重复声明，抛出*DecafDeclConflictError*；

②为声明的新形参创建变量符号，添加到符号表内；

③将该变量符号同*param*关联。

```
    def visitParameter(self, param: Parameter, ctx: ScopeStack) -> None:
        if ctx.findConflict(param.ident.value):
            raise DecafDeclConflictError(param.ident.value)
        symbol = VarSymbol(param.ident.value, param.var_t.type)
        ctx.declare(symbol)
        param.setattr('symbol', symbol)
```

新增了*Namer.visitCall*，即：

①根据函数名称检查调用的函数是否有对应符号；

②若无对应符号或对应符号不是函数符号，则抛出*DecafUndefinedFuncError*；

③检查该函数符号的形参个数是否和调用时的参数个数相同，若不同则抛出*DecafBadFuncCallError*；

④依次访问参数列表中的每个参数。

```
    def visitCall(self, call: Call, ctx: ScopeStack) -> None:
        funcSymbol = ctx.lookup(call.ident.value)
        if not funcSymbol or not isinstance(funcSymbol, FuncSymbol):
            raise DecafUndefinedFuncError(call.ident.value)
        if len(call.argument_list) != funcSymbol.parameterNum:
            raise DecafBadFuncCallError(call.ident.value)
        for argument in call.argument_list:
            argument.accept(self, ctx)
```

中间代码生成

新增了*Param*、*Call*两个指令类。

```
# Parameter setting instruction.
class Param(TACInstr):
    def __init__(self, parameter: Temp) -> None:
        super().__init__(InstrKind.SEQ, [], [parameter], None)
        self.parameter = parameter

    def __str__(self) -> str:
        return 'param %s' % (self.parameter)

    def accept(self, v: TACVisitor) -> None:
        return v.visitParam(self)


# Calling a function.
class Call(TACInstr):
    def __init__(self, ret_v: Temp, target: Label, args: list[Temp]) -> None:
        super().__init__(InstrKind.SEQ, [ret_v], [], target)
        self.ret_v = ret_v
        self.target = target
        self.args = args

    def __str__(self) -> str:
        return '%s = call %s' % (self.ret_v, self.target)

    def accept(self, v: TACVisitor) -> None:
        return v.visitCall(self)
```

在*tacvisitor.py*中添加了相应的两个函数。

```
    def visitParam(self, instr: Param) -> None:
        self.visitOther(instr)

    def visitCall(self, instr: Call) -> None:
        self.visitOther(instr)
```

在*FuncVisitor*中添加相应的两个函数，其中*visitCall*需要为函数返回值分配新的临时变量。

```
    def visitParam(self, parameter: Temp) -> None:
        self.func.add(Param(parameter))

    def visitCall(self, target: Label, args: list[Temp]) -> Temp:
        temp = self.freshTemp()
        self.func.add(Call(temp, target, args))
        return temp
```

修改了*TACGen.transform*，以为各个函数生成中间代码，而不只对*main*函数工作，即：

①以所有函数节点的列表（定义和声明同时存在时，只返回定义）为参数构造*ProgramWriter*类的对象*pw*；

②对每个函数定义，生成对应*TACFunc*（不再保存成员*numArgs*，转为以成员*params*直接保存形参的临时变量的列表），并为其形参们分配临时变量，并将该临时变量列表作为成员*params*；

③访问函数体后，结束该函数的中间代码生成；

④完成对每个函数的中间代码生成后，结束。

```
    # Entry of this phase
    def transform(self, program: Program) -> TACProg:
        pw = ProgramWriter([func for func in program.functions()])
        for func_name, func in program.functions().items():
            if func.body != NULL:
                mv = pw.visitFunc(func_name)
                for param in func.parameter_list:
                    temp = mv.freshTemp()
                    param.getattr('symbol').temp = temp
                    mv.func.params.append(temp)
                func.body.accept(self, mv)
                # Remember to call mv.visitEnd after the translation a function.
                mv.visitEnd()
```

新增了*TACGen.visitCall*，即：

①依次访问每个参数，以其值为参数调用*mv.visitParam*添加*PARAM*指令；

②调用*mv.visitCall*，添加*CALL*指令；

③将*visitCall*返回的临时变量作为节点的值。

```
    def visitCall(self, call: Call, mv: FuncVisitor) -> None:
        for argument in call.argument_list:
            argument.accept(self, mv)
            mv.visitParam(argument.getattr('val'))
        args = [argument.getattr('val') for argument in call.argument_list]
        call.setattr('val', mv.visitCall(mv.ctx.getFuncLabel(call.ident.value),
args))
```

目标平台汇编代码生成

新增了*Riscv.Call*、*Riscv.FPUpdate*两条指令。

```
    class Call(TACInstr):
        def __init__(self, ret_v: Temp, target: Label, args: list[Temp]) ->
None:
            super().__init__(InstrKind.SEQ, [], args, target)
            self.ret_v = ret_v
            self.target = target
            self.args = args

        def __str__(self) -> str:
            return 'call ' + str(self.target.name)
```

```
    class FPUpdate(NativeInstr):
        def __init__(self, offset: int) -> None:
            super().__init__(InstrKind.SEQ, [Riscv.FP], [Riscv.SP], None)
            self.offset = offset

        def __str__(self) -> str:
            assert -2048 <= self.offset <= 2047  # Riscv imm [11:0]
            return "addi " + Riscv.FMT3.format(
                str(Riscv.FP), str(Riscv.SP), str(self.offset)
            )
```

在*RiscvInstrSelector*下实现了*visitParam*和*visitCall*。由于我计划将传参的汇编指令生成放在为*Riscv.Call*生成的*Loc*分配寄存器时，故visitParam并无实际操作。

```
        def visitParam(self, instr: Param) -> None:
            pass

        def visitCall(self, instr: Call) -> None:
            self.seq.append(Riscv.Call(instr.ret_v, instr.target, instr.args))
```

在*RiscvSubroutineEmitter*的初始化中，由于保存*FP*的需要，成员*nextLocalOffset*的初值要再增加*4*。

此外，添加了成员*params*，其来源为对应的*TACFunc*的成员*params*，保存了该函数的形参们的临时变量，并为所有通过栈传递的参数，计算出其同栈帧起始位置*FP*的偏移量，保存在成员*paramFPOffsets*中。

```
        # + 4 is for the RA reg, + 4 is for the FP reg, so + 8
        self.nextLocalOffset = 4 * len(Riscv.CalleeSaved) + 8
```

```
        # parameters of this function
        self.params = params
        # in step9, step11 you can compute the offset of local array and
parameters here
        # offset to FP of each parameter
        self.paramFPOffsets = {param.index : 4 * i for i, param in
enumerate(self.params[8:])}
```

并为*RiscvSubroutineEmitter*新增了*changeOffset*方法，以便当*SP*变化时，调整所有相对*SP*的偏移量。

```
    def changeOffset(self, delta: int):
        self.nextLocalOffset += delta
        for i in self.offsets:
            self.offsets[i] += delta
```

修改了*RiscvSubroutineEmitter.emitLoadFromStack*，当加载的*temp*是栈上的形参时，根据*FP*和*paramFPOffsets*来生成*LW*指令。

```
    # load some temp from stack
    # usually happen when using a temp which is stored to stack before
    # in step9, you need to think about the fuction parameters here
    def emitLoadFromStack(self, dst: Reg, src: Temp):
        if src.index not in self.offsets:
            if src.index not in self.paramFPOffsets:
                raise IllegalArgumentException()
            else:
                self.buf.append(Riscv.NativeLoadWord(dst, Riscv.FP,
self.paramFPOffsets[src.index]))
        else:
            self.buf.append(
                Riscv.NativeLoadWord(dst, Riscv.SP, self.offsets[src.index])
            )
```

在*emitEnd*中，在*prologue*和*epilogue*部分，分别对*FP*、*RA*、*calleeSaveRegs*进行了保存和恢复，并在保存了原*FP*值之后将其更新。

```
        # save RA
        self.printer.printInstr(Riscv.NativeStoreWord(Riscv.RA, Riscv.SP, 4 + 4
* len(Riscv.CalleeSaved)))
        # save FP
        self.printer.printInstr(Riscv.NativeStoreWord(Riscv.FP, Riscv.SP, 4 *
len(Riscv.CalleeSaved)))
        # update FP
        self.printer.printInstr(Riscv.FPUpdate(self.nextLocalOffset))
```

```
        # reload FP
        self.printer.printInstr(Riscv.NativeLoadWord(Riscv.FP, Riscv.SP, 4 *
len(Riscv.CalleeSaved)))
        # reload RA
        self.printer.printInstr(Riscv.NativeLoadWord(Riscv.RA, Riscv.SP, 4 + 4 *
len(Riscv.CalleeSaved)))
```

寄存器分配

在*BruteRegAlloc.localAlloc*中，若该基本块是函数的初始基本块，则将通过寄存器传递的形参的临时变量们绑定到对应的*ArgReg*中。

```
        self.bindings.clear()
        for reg in self.emitter.allocatableRegs:
            reg.occupied = False
        # the first basic block
        if bb.id == 0:
            for i, param in enumerate(subEmitter.params[:8]):
                self.bind(param, Riscv.ArgRegs[i])
```

新增了*allocForCall*，用于在访问到由*Riscv.Call*生成的*Loc*时取代*allocForLoc*。

该方法内部逻辑如下：

①保存*callerSaveRegs*；

②通过*SPAdd(-4)*和*NativeStoreWord*，将所有参数从右往左压栈，其中由于*SP*变化，需要调用*subEmitter.changeOffset*调整相对*SP*的偏移量；

③通过*SPAdd(4)*和*NativeLoadWord*，将通过寄存器传递的参数从左往右依次从栈中弹出，而其余的通过栈传递的参数，已于上一步就已经在栈中保存好了，其中由于*SP*变化，需要调用*subEmitter.changeOffset*调整相对*SP*的偏移量；

④为该条由*Riscv.Call*生成的*Loc*分配寄存器；

⑤将栈复原，释放掉用于传递参数的内存；

⑥保存*a0*中的返回值；

⑦恢复*callerSaveRegs*。

```
# in step9, you may need to think about how to store callersave regs here
        for loc in bb.allSeq():
            subEmitter.emitComment(str(loc.instr))
            if isinstance(loc.instr, Riscv.Call):
                self.allocForCall(loc, subEmitter)
            else:
                self.allocForLoc(loc, subEmitter)
```

```
    def allocForCall(self, loc: Loc, subEmitter: SubroutineEmitter):
        # store caller save regs
        callerSaveRegs = []
        for reg in self.emitter.callerSaveRegs:
            if reg.occupied and reg.temp.index in loc.liveOut:
                subEmitter.emitStoreToStack(reg)
                callerSaveRegs.append(reg)
        # pass the parameters by stack
        for arg in reversed(loc.instr.args):
            reg = self.allocRegFor(arg, True, loc.liveIn, subEmitter)
            # push the parameters
            subEmitter.emitNative(Riscv.SPAdd(-4))
            subEmitter.emitNative(Riscv.NativeStoreWord(reg, Riscv.SP, 0))
            # change offsets due to the change of SP
```

```
            subEmitter.changeOffset(4)
        # pass the parameters by regs
        for i in range(len(loc.instr.args[:8])):
            # pop the parameters to arg regs
            subEmitter.emitNative(Riscv.NativeLoadWord(Riscv.ArgRegs[i],
Riscv.SP, 0))
            subEmitter.emitNative(Riscv.SPAdd(4))
            # change offsets due to the change of SP
            subEmitter.changeOffset(-4)
        # call instr
        self.allocForLoc(loc, subEmitter)
        # if there are parameters passed by stack
        if len(loc.instr.args) > 8:
            size = 4 * (len(loc.instr.args) - 8)
            # free stack memory
            subEmitter.emitNative(Riscv.SPAdd(size))
            # change offsets due to the change of SP
            subEmitter.changeOffset(-size)
        # store return value
        self.saveReturnValue(loc, subEmitter)
        # load caller save regs
        for reg in callerSaveRegs:
            subEmitter.emitLoadFromStack(reg, reg.temp)
```

新增了*saveReturnValue*，用于保存*a0*中的值入栈，即：

①若*a0*已经绑定临时变量，记录并解绑；

②将*a0*同中间代码中用于存储返回值的临时变量绑定；

③将返回值存入栈中；

④若*a0*曾解绑，则重新绑定原临时变量。

```
    def saveReturnValue(self, loc: Loc, subEmitter: SubroutineEmitter):
        tempForA0 = None
        if Riscv.A0.occupied:
            tempForA0 = Riscv.A0.temp
            self.unbind(Riscv.A0.temp)
        self.bind(loc.instr.ret_v, Riscv.A0)
        subEmitter.emitStoreToStack(Riscv.A0)
        self.unbind(Riscv.A0.temp)
        if tempForA0:
            self.bind(tempForA0, Riscv.A0)
```

step10

语法分析

修改了*Program*类，其子节点不再只包含函数*Function*类，还包括变量声明*Declaration*类（此处即全局变量声明），并为其新增了*Program.declarations*方法，用于返回所有的全局变量声明语句。

```
class Program(ListNode[Union["Function", 'Declaration']]):
    """
    AST root. It should have only one children before step9.
    """
```

```
    def __init__(self, *children: Union[Function, Declaration]) -> None:
        super().__init__("program", list(children))

    def functions(self) -> dict[str, Function]:
        funcs = {}
        for func in self:
            if isinstance(func, Function):
                funcs[func.ident.value] = func
        return funcs

    def declarations(self) -> list[Declaration]:
        return [decl for decl in self if isinstance(decl, Declaration)]
```

修改了非终结符*program*的产生式，以包含全局变量声明。

```
def p_program(p):
    """
    program : program function
        | program declaration Semi
    """
    p[1].children.append(p[2])
    p[0] = p[1]
def p_program_one(p):
    """
    program : function
        | declaration Semi
    """
    p[0] = Program(p[1])
```

语义分析

修改了*Namer.visitProgram*(仅修改了代码中变量名称，实际行为不变)，访问所有函数和全局变量声明。

```
    def visitProgram(self, program: Program, ctx: ScopeStack) -> None:
        # Check if the 'main' function is missing
        if not program.hasMainFunc():
            raise DecafNoMainFuncError
        for funcOrDecl in program:
            funcOrDecl.accept(self, ctx)
```

修改了*Namer.visitDeclaration*，根据当前作用域是否是全局作用域，执行不同操作，并将该信息存入新创建的变量符号中。若不是全局作用域，保持原先行为逻辑不变；若是全局作用域，在发现重复声明时需抛出*DecafGlobalVarDefinedTwiceError*，在有初始值时需要检查其是否是常量。

```
    def visitDeclaration(self, decl: Declaration, ctx: ScopeStack) -> None:
        """
        1. Use ctx.findConflict to find if a variable with the same name has been
declared.
        2. If not, build a new VarSymbol, and put it into the current scope using
ctx.declare.
        3. Set the 'symbol' attribute of decl.
        4. If there is an initial value, visit it.
        """
```

```
        isGlobal = ctx.isGlobalScope()
        if ctx.findConflict(decl.ident.value):
            if isGlobal:
                raise DecafGlobalVarDefinedTwiceError(decl.ident.value)
            raise DecafDeclConflictError(decl.ident.value)
        symbol = VarSymbol(decl.ident.value, decl.var_t.type, isGlobal)
        ctx.declare(symbol)
        decl.setattr('symbol', symbol)
        if decl.init_expr != NULL:
            decl.init_expr.accept(self, ctx)
            if isGlobal and not isinstance(decl.init_expr, IntLiteral):
                raise DecafGlobalVarBadInitValueError(decl.ident.value)
```

中间代码生成

新增了*LoadSymbol*、*Store*、*Load*三个指令类，分别用于加载全局变量地址，向指定内存地址写入值，从指定内存地址处加载值。

```
# Load address of symbol.
class LoadSymbol(TACInstr):
    def __init__(self, dst: Temp, symbolName: str) -> None:
        super().__init__(InstrKind.SEQ, [dst], [], None)
        self.dst = dst
        self.symbolName = symbolName

    def __str__(self) -> str:
        return '%s = load_symbol %s' % (self.dst, self.symbolName)

    def accept(self, v: TACVisitor) -> None:
        return v.visitLoadSymbol(self)


# Store data in address = base + offset.
class Store(TACInstr):
    def __init__(self, src: Temp, base: Temp, offset: int) -> None:
        super().__init__(InstrKind.SEQ, [], [src, base], None)
        self.src = src
        self.base = base
        self.offset = offset

    def __str__(self) -> str:
        return 'store %s, %s, %s' % (self.src, self.base, self.offset)

    def accept(self, v: TACVisitor) -> None:
        return v.visitStore(self)


# Load data in address = base + offset.
class Load(TACInstr):
    def __init__(self, dst: Temp, base: Temp, offset: int) -> None:
        super().__init__(InstrKind.SEQ, [dst], [base], None)
        self.dst = dst
        self.base = base
        self.offset = offset
```

```
    def __str__(self) -> str:
        return '%s = load %s, %s' % (self.dst, self.base, self.offset)

    def accept(self, v: TACVisitor) -> None:
        return v.visitLoad(self)
```

在*tacvisitor.py*中添加了相应的三个函数。

```
    def visitLoadSymbol(self, instr: LoadSymbol) -> None:
        self.visitOther(instr)

    def visitStore(self, instr: Store) -> None:
        self.visitOther(instr)

    def visitLoad(self, instr: Load) -> None:
        self.visitOther(instr)
```

在*FuncVisitor*中添加相应的三个函数，其中*visitLoadSymbol*和*visitLoadInMem*分别需要为全局变量地址、加载的值分配新的临时变量。

```
    def visitLoadSymbol(self, symbolName: str) -> Temp:
        temp = self.freshTemp()
        self.func.add(LoadSymbol(temp, symbolName))
        return temp

    def visitLoadInMem(self, base: Temp, offset: int) -> Temp:
        temp = self.freshTemp()
        self.func.add(Load(temp, base, offset))
        return temp

    def visitStoreInMem(self, src: Temp, base: Temp, offset: int) -> None:
        self.func.add(Store(src, base, offset))
```

修改了*TACGen.transform*，新增了对全局变量声明的处理。对每个全局变量声明，不像局部变量声明那样调用*accept*来访问，而只需要对有初始值的全局变量符号通过*setInitValue*方法设置初始值即可。此外，以一个字典保存全局变量名称及初始值，留给后端产生相应的目标代码。

```
        globalSymbolNameValues = {}
        for decl in program.declarations():
            if decl.init_expr != NULL:
                decl.getattr('symbol').setInitValue(decl.init_expr.value)
            globalSymbol = decl.getattr('symbol')
            globalSymbolNameValues[globalSymbol.name] = globalSymbol.initValue
        # Remember to call pw.visitEnd before finishing the translation phase.
        return pw.visitEnd(globalSymbolNameValues)
```

修改了*TACGen.visitIdentifier*，与局部变量在声明时直接分配有新的临时变量不同，要读全局变量的值，其临时变量需要通过从内存中全局变量所在地址处加载获得。

```
    def visitIdentifier(self, ident: Identifier, mv: FuncVisitor) -> None:
        """
        1. Set the 'val' attribute of ident as the temp variable of the 'symbol'
attribute of ident.
        """
        symbol = ident.getattr('symbol')
        if symbol.isGlobal:
            base = mv.visitLoadSymbol(symbol.name)
            symbol.temp = mv.visitLoadInMem(base, 0)
        ident.setattr('val', symbol.temp)
```

修改了*TACGen.visitAssignment*，与局部变量在声明时直接分配有新的临时变量不同，要对全局变量赋值，需要向内存中全局变量所在地址处写入要赋的值。

```
    def visitAssignment(self, expr: Assignment, mv: FuncVisitor) -> None:
        """
        1. Visit the right hand side of expr, and get the temp variable of left
hand side.
        2. Use mv.visitAssignment to emit an assignment instruction.
        3. Set the 'val' attribute of expr as the value of assignment
instruction.
        """
        symbol = expr.lhs.getattr('symbol')
        expr.rhs.accept(self, mv)
        if symbol.isGlobal:
            base = mv.visitLoadSymbol(symbol.name)
            mv.visitStoreInMem(expr.rhs.getattr('val'), base, 0)
            expr.setattr('val', expr.rhs.getattr('val'))
        else:
            expr.setattr('val', mv.visitAssignment(symbol.temp,
expr.rhs.getattr('val')))
```

目标平台汇编代码生成

新增了*Riscv.LoadAddress*、*Riscv.Store*、*Riscv.Load*三条指令。

```
    class LoadAddress(TACInstr):
        def __init__(self, dst: Temp, symbolName: str) -> None:
            super().__init__(InstrKind.SEQ, [dst], [], None)
            self.symbolName = symbolName

        def __str__(self) -> str:
            return 'la ' + Riscv.FMT2.format(str(self.dsts[0]),
str(self.symbolName))

    class Store(TACInstr):
        def __init__(self, src: Temp, base: Temp, offset: int) -> None:
            super().__init__(InstrKind.SEQ, [], [src, base], None)
            self.offset = offset

        def __str__(self) -> str:
            assert -2048 <= self.offset <= 2047  # Riscv imm [11:0]
            return 'sw ' + Riscv.FMT_OFFSET.format(str(self.srcs[0]),
str(self.offset), str(self.srcs[1]))
```

```
    class Load(TACInstr):
        def __init__(self, dst: Temp, base: Temp, offset: int) -> None:
            super().__init__(InstrKind.SEQ, [dst], [base], None)
            self.offset = offset

        def __str__(self) -> str:
            assert -2048 <= self.offset <= 2047  # Riscv imm [11:0]
            return 'lw ' + Riscv.FMT_OFFSET.format(str(self.dsts[0]),
str(self.offset), str(self.srcs[0]))
```

在*RiscvInstrSelector*下实现了*visitLoadSymbol*、*visitStore*、*visitLoad*。

```
        def visitLoadSymbol(self, instr: LoadSymbol) -> None:
            self.seq.append(Riscv.LoadAddress(instr.dst, instr.symbolName))

        def visitStore(self, instr: Store) -> None:
            self.seq.append(Riscv.Store(instr.src, instr.base, instr.offset))

        def visitLoad(self, instr: Load) -> None:
            self.seq.append(Riscv.Load(instr.dst, instr.base, instr.offset))
```

在*RiscvAsmEmitter*的初始化中，根据字典*globalSymbolNameValues*，加入*data*数据段及各个全局变量声明（因为框架下未初始化的全局变量初始值为默认值*0*，我将其视为以*0*初始化的全局变量）。

此处的*globalSymbolNameValues*是中间代码生成时创建的一个字典，保存了全局变量名称及初始值，它被作为一个新的参数，添加到了这些函数的参数列表中：*ProgramWriter.visitEnd*、*TACProg.__init__*、*RiscvAsmEmitter.__init__*，从而从中端传递到了后端，在后端被用于添加目标代码中的*data*数据段。

```
        # int step10, you need to add the declaration of global var here
        if globalSymbolNameValues:
            self.printer.println('.data')
            for symbolName, initValue in globalSymbolNameValues.items():
                self.printer.println('.global %s' % (symbolName))
                self.printer.printLabel(Label(LabelKind.TEMP, symbolName))
                self.printer.println('.word %s' % (initValue))
                self.printer.println("")
```

```
    def visitEnd(self, globalSymbolNameValues: dict[str, int]) -> TACProg:
        return TACProg(self.ctx.funcs, globalSymbolNameValues)
```

```
class TACProg:
    def __init__(self, funcs: list[TACFunc]) -> None:
    def __init__(self, funcs: list[TACFunc], globalSymbolNameValues: dict[str,
int]) -> None:
        self.funcs = funcs
        self.globalSymbolNameValues = globalSymbolNameValues
```

```
# Target code generation stage: Three-address code -> RISC-V assembly code
def step_asm(p: TACProg):
    riscvAsmEmitter = RiscvAsmEmitter(Riscv.AllocatableRegs, Riscv.CallerSaved,
p.globalSymbolNameValues)
    asm = Asm(riscvAsmEmitter, BruteRegAlloc(riscvAsmEmitter))
    prog = asm.transform(p)
    return prog
```

```
class RiscvAsmEmitter(AsmEmitter):
    def __init__(
        self,
        allocatableRegs: list[Reg],
        callerSaveRegs: list[Reg],
        globalSymbolNameValues: dict[str, int]
    ) -> None:
```

思考题

step9

```
int add(int lhs, int rhs){
    return lhs + rhs;
}
int main(){
    int x = 1;
    return add(x = 2, x);
}
```

若参数求值顺序为从左到右，则*add*函数返回值为*4*，若从右到左，则为*3*。

step10

```
la v0, a
```

- *PIC case*

  其中*delta = GOT[a] - pc*

  ```
  auipc v0, delta[31:12] + delta[11]
  lw v0, delta[11:0](v0)
  ```

- *non-PIC case*

  其中*delta = a - pc*

  ```
  auipc v0, delta[31:12] + delta[11]
  addi v0, v0, delta[11:0]
  ```