Stage5 实验报告

姓名：丁俊辉

学号：2021011145

step9

实验过程

在 `lex.py` 中，增添逗号运算符：

```
t_Comma = ","
```

在 `tree.py` 中，重新定义 Function 节点，加入函数参数。新定义 AST 节点，用来描述函数形参，形参列表，函数声明，函数调用和实参列表：

```
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)

    def __str__(self) -> str:
        return f"{self.var_t.name} {self.ident.value}"


class ParameterList(ListNode[Parameter]):
    """
    AST node that represents a list of parameters.
    """

    def __init__(self, *children: Parameter) -> None:
        super().__init__("parameter_list", list(children))

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

    def __str__(self) -> str:
        return ", ".join(map(str, self.children))
```

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

    def __init__(
        self,
        ret_t: TypeLiteral,
        ident: Identifier,
        params: ParameterList,
        body: Block,
    ) -> None:
        super().__init__("function")
        self.ret_t = ret_t
        self.ident = ident
        self.params = params
        self.body = body

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

    def __len__(self) -> int:
        return 4

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

    def __str__(self) -> str:
        return f"{self.ret_t.name} {self.ident.value}({self.params})"


class ExpressionList(ListNode[Expression]):
    """
    AST node of expression list.
    """

    def __init__(self, *children: Expression) -> None:
        super().__init__("expression_list", list(children))

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

    def __str__(self) -> str:
        return ", ".join(map(str, self.children))


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

```
    """

    def __init__(self, ident: Identifier, args: ExpressionList) -> None:
        super().__init__("call")
        self.ident = ident
        self.args = args

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

    def __len__(self) -> int:
        return 2

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

    def __str__(self) -> str:
        return "{}({})".format(
            self.ident.value,
            self.args,
        )
```

在 `ply_parser.py` 中，令程序可以有多个函数声明，并且新加入生成带参函数和函数调用的语法：

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


def p_program_empty(p):
    """
    program : empty
    """
    p[0] = Program()
```

这是对 `program` 的语法更改，目的是让一个程序可以包含多个函数。

```
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])


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

```
def p_comma_parameter(p):
    """
    comma_parameter : Comma parameter
    """
    p[0] = p[2]


def p_comma_parameter_list(p):
    """
    comma_parameter_list : comma_parameter_list comma_parameter
    """
    p[1].children.append(p[2])
    p[0] = p[1]


def p_comma_parameter_list_empty(p):
    """
    comma_parameter_list : empty
    """
    p[0] = ParameterList()


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


def p_parameter_list(p):
    """
    parameter_list : parameter comma_parameter_list
    """
    p[2].children.insert(0, p[1])
    p[0] = p[2]
```

这一部分主要是在函数定义中加入 `parameter_list`，并设计 `parameter_list` 的产生式。

```
def p_comma_expression(p):
    """
    comma_expression : Comma expression
    """
    p[0] = p[2]


def p_comma_expression_list(p):
    """
    comma_expression_list : comma_expression_list comma_expression
    """
    p[1].children.append(p[2])
    p[0] = p[1]


def p_comma_expression_list_empty(p):
    """
```

```
    comma_expression_list : empty
    """
    p[0] = ExpressionList()


def p_expression_list(p):
    """
    expression_list : expression comma_expression_list
    """
    p[2].children.insert(0, p[1])
    p[0] = p[2]


def p_expression_list_empty(p):
    """
    expression_list : empty
    """
    p[0] = ExpressionList()


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

这一部分主要是设计 `expression_list` 的产生式，然后定义函数调用。

至此，可以正确生成函数定义和调用的 AST。

在 `namer.py` 中，我加入了函数的类型检查：

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

    def visitProgram(self, program: Program, ctx: ScopeStack) -> None:
        ...

        for func in program.children: # 不对函数名称强行映射，以检查重名
            func.accept(self, ctx)

    def visitParameter(self, param: Parameter, ctx: ScopeStack) -> None:
        if ctx.top().lookup(param.ident.value) is not None:
            raise DecafDeclConflictError(param.ident.value)
        symbol = VarSymbol(param.ident.value, param.var_t.type)
        ctx.top().declare(symbol)
        param.setattr("symbol", symbol)

    def visitParameterList(self, params: ParameterList, ctx: ScopeStack) -> None:
        for param in params:
            param.accept(self, ctx)

    def visitFunction(self, func: Function, ctx: ScopeStack) -> None:
        if ctx.globalscope.lookup(func.ident.value) is not None:
            raise DecafDeclConflictError(func)
        symbol = FuncSymbol(func.ident.value, func.ret_t.type, ctx.globalscope)
```

```
        for param in func.params.children:
            symbol.addParaType(param.var_t.type)
        ctx.globalscope.declare(symbol)
        func.setattr("symbol", symbol)

        ctx.push(Scope(ScopeKind.LOCAL))
        func.params.accept(self, ctx)
        for child in func.body:
            child.accept(self, ctx) # 不直接 visitBlock，保证在同一个作用域
        ctx.pop()

    def visitExpressionList(self, exprs: ExpressionList, ctx: ScopeStack) ->
None:
        for expr in exprs:
            expr.accept(self, ctx)

    def visitCall(self, expr: Call, ctx: ScopeStack) -> None:
        if not ctx.lookup(expr.ident.value).isFunc:
            raise DecafBadFuncCallError(expr.ident.value)
        func: FuncSymbol = GlobalScope.lookup(expr.ident.value)
        if func is None:
            raise DecafUndefinedFuncError(expr.ident.value)
        expr.setattr("symbol", func)

        expr.args.accept(self, ctx)
        if len(expr.args) != func.parameterNum:
            raise DecafBadFuncCallError(expr.ident.value)
        for i in range(len(expr.args.children)):
            if not func.getParaType(i) == INT:
                raise DecafBadFuncCallError(expr.ident.value)
```

当前的代码中我只检查了参数是否为 INT 类型，因为这是唯一的类型。之后可能会做更改。

在 `tacop.py` 中，我加入了函数调用的 `InstrKind`：

```
@unique
class InstrKind(Enum):
    ...

    # Call instruction.
    CALL = auto()
```

在 `tacinstr.py` 中，加入函数调用的 `TACInstr`：

```
# Function call.
class Call(TACInstr):
    def __init__(self, dst: Temp, label: FuncLabel, args: list[Temp]) -> None:
        super().__init__(InstrKind.CALL, [dst], args, label)
        self.dst = dst
        self.label = label
        self.args = args

    def __str__(self) -> str:
        args_str = ", ".join(map(str, self.args))
        return "%s = CALL %s(%s)" % (self.dst, self.label, args_str)
```

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

在 `tacgen.py` 中，我加入了函数声明和调用的 TAC 码生成：

```
class TACFuncEmitter(TACVisitor):
    ...

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

这部分是 `TACInstr` 的生成

```
class TACGen(Visitor[TACFuncEmitter, None]):
    # Entry of this phase
    def transform(self, program: Program) -> TACProg:
        labelManager = LabelManager()
        tacFuncs = []
        for funcName, astFunc in program.functions().items():
            # in step9, you need to use real parameter count
            emitter = TACFuncEmitter(FuncLabel(funcName), len(astFunc.params),
labelManager)
            astFunc.params.accept(self, emitter)
            astFunc.body.accept(self, emitter)
            tacFuncs.append(emitter.visitEnd())
        return TACProg(tacFuncs)

    def visitParameter(self, param: Parameter, mv: TACFuncEmitter) -> None:
        temp = mv.freshTemp()
        param.getattr('symbol').temp = temp
        mv.func.argTemps.append(temp)

    def visitParameterList(self, params: ParameterList, mv: TACFuncEmitter) ->
None:
        for param in params.children:
            param.accept(self, mv)

    ...

    def visitExpressionList(self, exprs: ExpressionList, mv: TACFuncEmitter) ->
None:
        for expr in exprs.children:
            expr.accept(self, mv)

    def visitCall(self, expr: Call, mv: TACFuncEmitter) -> None:
        expr.args.accept(self, mv)
        argTemps = [arg.getattr("val") for arg in expr.args.children]
        expr.setattr("val", mv.visitCall(FuncLabel(expr.getattr("symbol").name),
argTemps))
```

这部分是让在 visit 函数体之前，先 visit 参数列表。并且实现函数调用。

到这一步，已经可以生成正确的 TAC 代码。但是除此之外，我还修改了 `tacfunc.py` 中 `TACFunc` 的定义：

```
class TACFunc:
    def __init__(self, entry: FuncLabel, numArgs: int) -> None:
        self.entry = entry
        self.numArgs = numArgs
        self.argTemps: list[Temp] = []
        self.instrSeq = []
        self.tempUsed = 0
```

我加入了 `argTemps` 属性，用来记录一个函数被作为参数的 `Temp`。这样特定记下来是因为我无法完全保证参数就是前 `numArgs` 个 `Temp`（虽然看起来约定是这样的），我希望将这些 `Temp` 直接传递到寄存器绑定阶段。

为了将这些 `argTemps` 传递到寄存器绑定阶段，我还需要将 `argTemps` 传递给 `subroutineinfo.py` 中的 `SubroutineInfo`，为此我也在 `SubroutineInfo` 增添了属性：

```
class SubroutineInfo:
    def __init__(self, funcLabel: FuncLabel, args: list[Temp]) -> None:
        self.funcLabel = funcLabel
        self.args = args

    def __str__(self) -> str:
        return "funcLabel: {}".format(
            self.funcLabel.name,
        )
```

在 `asm.py` 中，我完成了这些参数 `Temp` 的传递：

```
class Asm:
    def __init__(self) -> None:
        pass

    def transform(self, prog: TACProg):
        analyzer = LivenessAnalyzer()

        emitter = RiscvAsmEmitter(Riscv.AllocatableRegs, Riscv.CallerSaved)
        reg_alloc = BruteRegAlloc(emitter)
        for func in prog.funcs:
            pair = emitter.selectInstr(func)
            builder = CFGBuilder()
            cfg: CFG = builder.buildFrom(pair[0])
            analyzer.accept(cfg)
            reg_alloc.accept(cfg, pair[1])

        return emitter.emitEnd()
```

此外，我也对 ASM 代码生成的过程进行更改。因为 `RiscvAsmEmitter` 要维护 `AsmCodePrinter` 的 buf，如果每次循环重新构造 `RiscvAsmEmitter`，就会将 buf 清空，所以我将 `RiscvAsmEmitter` 和 `RegAlloc` 的构造放到循环体外面。

在 `riscv.py` 中，定义 `Call` 的 `BackendInstr`：

```
class Riscv:
    ...

    class Call(BackendInstr):
        def __init__(self, dst: Temp, label: FuncLabel, args: list[Temp]) ->
None:
            super().__init__(InstrKind.CALL, [dst], args, label)
            self.args = args

        def __str__(self) -> str:
            return "call " + str(self.label.name)
```

在 `bruteregalloc.py` 中，我完成了传递参数的过程：

```
class BruteRegAlloc(RegAlloc):
    ...

    def accept(self, graph: CFG, info: SubroutineInfo) -> None:
        subEmitter = RiscvSubroutineEmitter(self.emitter, info)

        args = info.args
        argRegs = Riscv.ArgRegs
        for idx, arg in enumerate(args):
            if idx < len(argRegs):
                self.bind(arg, Riscv.ArgRegs[idx])
            else:
                pass

        for bb in graph.iterator():
            # you need to think more here
            # maybe we don't need to alloc regs for all the basic blocks
            if not graph.isReachable(bb.id):
                continue
            if bb.label is not None:
                subEmitter.emitLabel(bb.label)
            self.localAlloc(bb, subEmitter)
        subEmitter.emitFunc()

    def allocForLoc(self, loc: Loc, subEmitter: RiscvSubroutineEmitter):
        instr = loc.instr
        srcRegs: list[Reg] = []
        dstRegs: list[Reg] = []

        if instr.isCall():
            instr: Riscv.Call
            savedCallerRegs = [reg for reg in self.emitter.callerSaveRegs if
reg.occupied]
            for reg in savedCallerRegs:
                subEmitter.emitStoreToStack(reg)
                self.unbind(reg.temp)

            for idx, arg in enumerate(instr.args):
                if idx < len(Riscv.ArgRegs):
                    reg = Riscv.ArgRegs[idx]
```

```
                    if reg.occupied:
                        self.unbind(reg.temp)
                    self.bind(arg, reg)
                    subEmitter.emitComment("  allocate {} to {} (read:
{})".format(str(arg), str(reg), str(True)))
                    subEmitter.emitLoadFromStack(reg, arg)
                else:
                    pass

        for i in range(len(instr.srcs)):
            temp = instr.srcs[i]
            if isinstance(temp, Reg):
                srcRegs.append(temp)
            else:
                srcRegs.append(self.allocRegFor(temp, True, loc.liveIn,
subEmitter))

        for i in range(len(instr.dsts)):
            temp = instr.dsts[i]
            if isinstance(temp, Reg):
                dstRegs.append(temp)
            else:
                dstRegs.append(self.allocRegFor(temp, False, loc.liveIn,
subEmitter))
        instr.fillRegs(dstRegs, srcRegs)
        subEmitter.emitAsm(instr)
```

这里我没有实现栈传参。如果要进行栈传参，应当将函数调用的 `allocForLoc` 单独分出来，否则按照我的 `Call` 类型定义，栈中参数也会在 src 中，会有冲突。

在 `riscvasmemitter.py` 中，我加入了 RA 寄存器的存储：

```
class RiscvSubroutineEmitter():
    ...

    def emitFunc(self):
        self.printer.printComment("start of prologue")
        self.printer.printInstr(Riscv.SPAdd(-self.nextLocalOffset))

        # in step9, you need to think about how to store RA here
        # you can get some ideas from how to save CalleeSaved regs
        for i in range(len(Riscv.CalleeSaved)):
            if Riscv.CalleeSaved[i].isUsed():
                self.printer.printInstr(
                    Riscv.NativeStoreWord(Riscv.CalleeSaved[i], Riscv.SP, 4 * i)
                )

        self.printer.printInstr(Riscv.NativeStoreWord(Riscv.RA, Riscv.SP,
len(Riscv.CalleeSaved) * 4))

        self.printer.printComment("end of prologue")
        self.printer.println("")

        self.printer.printComment("start of body")
```

```
        # in step9, you need to think about how to pass the parameters here
        # you can use the stack or regs

        # using asmcodeprinter to output the RiscV code
        for instr in self.buf:
            self.printer.printInstr(instr)

        self.printer.printComment("end of body")
        self.printer.println("")

        self.printer.printLabel(
            Label(LabelKind.TEMP, self.info.funcLabel.name +
Riscv.EPILOGUE_SUFFIX)
        )
        self.printer.printComment("start of epilogue")

        for i in range(len(Riscv.CalleeSaved)):
            if Riscv.CalleeSaved[i].isUsed():
                self.printer.printInstr(
                    Riscv.NativeLoadWord(Riscv.CalleeSaved[i], Riscv.SP, 4 * i)
                )

        self.printer.printInstr(Riscv.NativeLoadWord(Riscv.RA, Riscv.SP,
len(Riscv.CalleeSaved) * 4))

        self.printer.printInstr(Riscv.SPAdd(self.nextLocalOffset))
        self.printer.printComment("end of epilogue")
        self.printer.println("")

        self.printer.printInstr(Riscv.NativeReturn())
        self.printer.println("")
```

至此，可以正确生成所有测例的 ASM 代码。

思考题

1. 你更倾向采纳哪一种中间表示中的函数调用指令的设计（一整条函数调用 vs 传参和调用分离）？写一些你认为两种设计方案各自的优劣之处。

   具体而言，某个“一整条函数调用”的中间表示大致如下：

   ```
   _T3 = CALL foo(_T2, _T1, _T0)
   ```

   对应的“传参和调用分离”的中间表示类似于：

   ```
   PARAM _T2
   PARAM _T1
   PARAM _T0
   _T3 = CALL foo
   ```

A:

从代码的形式来看，第一种更加贴近源代码，第二种更加贴近 ASM 代码。

第一种的好处是对于参数的计数更加明了，可以通过一次 `visitCall` 直接完成函数调用。并且在这个过程中，应该会有很多优化的机会。

后者也有好处，我认为对于后者，只要额外维护参数的次序信息，就可以很好地呈现传参过程的数据流。比如当参数超过寄存器数量的时候，就很方便单拿出来处理，而不是像我当前的实现这样函数调用和参数处理有所耦合。

对于我目前的实现而言（即，如果不考虑优化），我会觉得第二种更好用。

2. 为何 RISC-V 标准调用约定中要引入 callee-saved 和 caller-saved 两类寄存器，而不是要求所有寄存器完全由 caller/callee 中的一方保存？为何保存返回地址的 ra 寄存器是 caller-saved 寄存器？

A:

通过区分 callee-saved 和 caller-saved 寄存器，可以减少不必要的寄存器保存和恢复操作，从而提高性能：

- caller-saved 寄存器：调用者在调用函数前保存这些寄存器，并在函数返回后恢复它们。这些寄存器通常用于短期的临时值，因为调用者知道哪些寄存器需要保存。
- callee-saved 寄存器：被调用者在函数开始时保存这些寄存器，并在函数结束时恢复它们。这些寄存器通常用于长期保存的值，因为被调用者知道哪些寄存器需要保存。
- 通过只保存和恢复必要的寄存器，还可以减少栈空间的使用，从而提高内存利用率。

关于 `ra` 寄存器是 caller-saved 寄存器的原因：

- `ra` 寄存器保存返回地址，调用者在调用函数前将返回地址保存在 `ra` 寄存器中。由于调用者知道何时需要保存和恢复返回地址，因此将 `ra` 设为 caller-saved 寄存器是合理的。
- 在递归调用中，每次调用都会生成新的返回地址。如果 `ra` 是 callee-saved 寄存器，那么每个被调用者都需要保存和恢复 `ra`，这会增加额外的开销。将 `ra` 设为 caller-saved 寄存器，可以让调用者在需要时保存和恢复返回地址，从而简化递归调用的实现。