STAGE-5

计13 占海川 2021050009

STEP-9

实验内容

抽象语法树

`frontend/ast/tree.py`

- 向Program列表加入Function

```
def addFunc(self, Func: Function) -> Program:
    self.children.append(Func)
    return self
```

- 添加Function类定义

```
class Function(Node):
    def __init__(
        self,
        ret_t:  TypeLiteral,        # 返回类型
        ident:  Identifier,         # 标识符
        params: List[Parameter],    # 参数列表
        body:   Block,              # 函数体
    )
```

- 添加Parameter类定义

```
class Parameter(Declaration):
    def __init__(
        self,
        var_t: TypeLiteral,         # 类型
        ident: Identifier,          # 标识符
)
```

- 添加Call类定义

```
class Call(Expression):
    def __init__(
        self,
        ident: Identifier,                # 标识符
        argument_list: List[Expression]   # 参数列表
    )
```

`frontend/ast/visitor.py`

- 加入对应visit方法

语法分析

`frontend/parser/ply_parser.py`

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

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


# 参数列表
def p_param(p):
    """
    parameter : type Identifier
    """
    p[0] = Parameter(p[1], p[2])

def p_params_none(p):
    """
    params : empty
    """
    p[0] = []

def p_params_single(p):
    """
    params : parameter
    """
    p[0] = [p[1]]

def p_params(p):
    """
    params : parameter Comma params
    """
    p[0] = [p[1]] + p[3]


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

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


# 函数调用参数列表
def p_argument_list_none(p):
    """
    argument_list : empty
    """
    p[0] = []

def p_argument_list_single(p):
    """
    argument_list : expression
    """
    p[0] = [p[1]]

def p_argument_list(p):
    """
    argument_list : argument_list Comma expression
    """
    p[0] = p[1] + [p[3]]


# 函数调用
def p_function_call(p):
    """
    postfix : Identifier LParen argument_list RParen
    """
    p[0] = Call(p[1], p[3])
```

语义分析

`frontend/typecheck/namer.py`

- visitFunction

```
def visitFunction(self, func: Function, ctx: ScopeStack) -> None:
    # 新建函数符号
    funcSym = FuncSymbol(func.ident.value, func.ret_t.type, ctx.globalScope)
    for param in func.params:
        funcSym.addParaType(param.var_t.type)
    # 检查声明冲突
    conflict = ctx.lookup(func.ident.value)
    if conflict:
        if (conflict.isFunc & conflict.type == funcSym.type):
            funcSym = conflict
        else:
            raise DecafDeclConflictError(func.ident.value)
    else:
        ctx.globalScope.declare(funcSym)
    func.setattr('symbol', funcSym)
    # 检查定义冲突
    if func.body is NULL:
        return
    else:
        if (funcSym.isDefined):
            raise DecafFuncDefConflictError(func.ident.value)
        else:
            funcSym.define
    # 开启函数体局部作用域
    ctx.push(Scope(ScopeKind.LOCAL))
    # 添加参数列表符号
    for param in func.params:
        var = VarSymbol(param.ident.value, param.var_t.type)
        ctx.top().declare(var)
        param.setattr("symbol", var)

    for child in func.body:
        child.accept(self, ctx)
    ctx.pop()
```

- visitFuncCall

```
def visitFuncCall(self, call: Call, ctx: ScopeStack) -> None:
    # 检查函数定义
    func = ctx.globalScope.lookup(call.ident.value)
    if not (func.isFunc & func.isDefined):
        raise DecafUndefinedFuncError(func.ident.value)
    if not (len(call.argument_list) == len(func.params)):
        raise DecafBadFuncCallError(call.ident.value)
    call.ident.setattr('symbol', func)
    for argument in call.argument_list:
        argument.accept(self, ctx)
```

中间代码分析

`frontend/tacgen/tacgen.py`

- visitFunction

```
def visitFunction(self, func: Function, mv: TACFuncEmitter) -> None:
    for parameter in func.params:
        parameter.accept(self, mv)
    func.body.accept(self, mv)
```

- visitCall

```
def visitCall(self, call: Call, mv: TACFuncEmitter) -> None:
    params = []
    for argument in call.argument_list:
        argument.accept(self, mv)
        params += [argument.getattr('val')]

    dst = mv.freshTemp()
    call.setattr('val', dst)
    target = FuncLabel(call.ident.value)
    mv.visitCall(dst, params, target)
```

- visitParameter

```
def visitParameter(self, param: Parameter, mv: TACFuncEmitter) -> None:
    var = param.getattr("symbol")
    var.temp = mv.freshTemp()
    mv.func.addTemp(var.temp)
```

后端代码

`backend/reg/bruteregalloc.py`

- 在分析CFG中每条指令前,首先将绑定参数列表与参数寄存器,将所有参数放到栈上

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

    for temp, argReg in zip(subEmitter.info.temps, Riscv.ArgRegs):
        self.bind(temp, argReg)
    if len(graph.reachable) > 0:
        for tempIndex in graph.nodes[0].liveIn:
            if tempIndex in self.bindings:
                subEmitter.emitStoreToStack(self.bindings.get(tempIndex))
    ...
```

- 为指令分配物理寄存器时, 根据指令是否为`CALL`决定不同处理方法, 最后将仍然活跃的寄存器放到栈上

```
def localAlloc(self, bb: BasicBlock, subEmitter: SubroutineEmitter):
    self.bindings.clear()
    for reg in self.emitter.allocatableRegs:
        reg.occupied = False

    # in step9, you may need to think about how to store callersave regs here
    for loc in bb.allSeq():
        if loc.instr.isCall:
            # Call
            self.allocForCall(loc, bb.liveIn, subEmitter)
        else:
            subEmitter.emitComment(str(loc.instr))
            self.allocForLoc(loc, subEmitter)
        # bb.liveOut不更新???
        for tempindex in loc.liveOut:
            if tempindex in self.bindings:
                subEmitter.emitStoreToStack(self.bindings.get(tempindex))
    ...
```

- 为`CALL`指令分配物理寄存器的处理函数

```
def allocForCall(self, loc: Loc, liveIn: set[int], subEmitter: SubroutineEmitter):
    # 保存活跃临时变量
    for i in range(len(Riscv.CallerSaved)):
        temp = Riscv.CallerSaved[i].temp
        if temp != None:
            if temp.index in liveIn:
                subEmitter.emitComment(
                    "store {} to {}".format(
                        str(temp), str(self.bindings.get(temp.index))
                    )
                )
                if self.bindings.get(temp.index) is not None:
                    subEmitter.emitStoreToStack(self.bindings.get(temp.index))
            self.unbind(temp)

    # 参数/寄存器绑定
    call = loc.instr
    for temp, argReg in zip(call.srcs, Riscv.ArgRegs):
        subEmitter.emitComment(
            "CALL allocate {} to {}".format(
                str(temp), str(argReg)
            )
        )
        if temp.index in self.bindings:
            self.unbind(temp)
        subEmitter.printer.printComment("load " + str(subEmitter.offsets))
        subEmitter.emitLoadFromStack(argReg, temp)
        self.bind(temp, argReg)

    # 函数调用
    subEmitter.emitComment(str(call))
    subEmitter.emitNative(NativeInstr(InstrKind.SEQ, call.dsts, call.dsts, call.label, cal

    # 处理函数返回值, target绑定a0
    self.bind(call.dsts[0], Riscv.A0)
    subEmitter.emitStoreToStack(Riscv.A0)
    subEmitter.printer.printComment("store2 " + str(subEmitter.offsets))
```

`backend/riscv/riscvasmemitter.py`

- 加入对`CALL`指令的riscv指令翻译, 通过TACInstr构造函数传递参数

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

class Call(TACInstr):
    @classmethod
    def construct(cls, call: Call) -> Call:
        return cls(call.dsts[0], call.srcs, call.label)

    def __init__(self, dst: Temp, params: List[Temp], target: Label) -> None:
        super().__init__(InstrKind.SEQ, [dst], params, target, True)

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

- 修改SubroutineEmitter类, 加入成员函数Temps以传递函数列表

```
def selectInstr(self, func: TACFunc) -> tuple[list[str], SubroutineInfo]:

    selector: RiscvAsmEmitter.RiscvInstrSelector = (
        RiscvAsmEmitter.RiscvInstrSelector(func.entry)
    )
    for instr in func.getInstrSeq():
        instr.accept(selector)
    info = SubroutineInfo(func.entry, func.temps)
```

- 保存/加载 ra

```
# save ra
self.printer.printInstr(
    Riscv.NativeStoreWord(Riscv.RA, Riscv.SP, 4 * len(Riscv.CalleeSaved))
)
# load ra
self.printer.printInstr(
    Riscv.NativeLoadWord(Riscv.RA, Riscv.SP, 4 * len(Riscv.CalleeSaved))
)
```

思考题

1. 我更倾向于第一种中间表示, 前者在TACInstr翻译到后端指令时只需一步visitCall, 但参数传递相对麻烦; 后者一条指令对应一个寄存器,参数传递简单,但翻译为后端代码需要多步完成
2.
   - 若完全由一方保存, 则需要保存所有需要用到的通用寄存器, 但实际使用的只是一部分, 全部保存开销过大
   - 在调用函数时，新的返回地址被jal等指令保存到RA寄存器,因此在进入被调用函数后,RA的值已经被修改，被调用者无法保存一个已经在调用过程中被修改的值。