stage-5

计11班 周韧平 2021010699

实验内容

前端

首先在 `./frontend/ast/tree.py` 中增加 `Parameter` 和 `Call` 类的定义，并为 Function 添加参数列表，并在 `./frontend/ast/visitor.py` 中为其添加对应的访问函数

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

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

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

    def __len__(self) -> int:
        return 3 + len(self.param_list)

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

class Parameter(Node):
    """
    AST node that represents a paramteter.
    """
    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(Node):
    """
    AST node that represents call.
    """
    def __init__(
        self,
        ident: Identifier,
        parameter_list : list[Expression]
    ) -> None:
        super().__init__("call")
        self.ident = ident
        self.parameter_list  = parameter_list

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

    def __len__(self) -> int:
        return len(self.parameter_list)+1

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

然后是在 `./frontend/parser/ply_parser.py` 中为函数的定义和调用设计对应的文法，在函数定义设计文法时需要注意参数列表可能为空的情况。同时还需要更改Program的文法，使允许一个 Program 中有多个函数

```
def p_program(p):
    """
    program : function_list
    """
    p[0] = Program(p[1])


def p_function_list(p):
```

```
    """
    function_list : function function_list
    """
    p[0] = [p[1]] + p[2]

def p_function_list_empty(p):
    """
    function_list : empty
    """
    p[0] = []

```

```
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_function_declare(p):
    """
    function : type Identifier LParen parameter_list RParen Semi
    """
    p[0] = Function(p[1], p[2], p[4], None)

def p_parameter_list(p):
    """
    parameter_list : parameter comma_parameter_list
    """
    p[0] = [p[1]] + p[2]

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

def p_comma_parameter_list(p):
    """
    comma_parameter_list : Comma parameter comma_parameter_list
    """
    p[0] = [p[2]] + p[3]

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

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

用相似的方法，参考实验文档规范，定义了函数调用的文法

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

def p_expression_list(p):
    """
    expression_list : expression comma_expression_list
    """
    p[0] = [p[1]] + p[2]

def p_empty_comma_expression_list(p):
    """
    comma_expression_list : empty
    """
    p[0] = []

def p_comma_expression_list(p):
    """
    comma_expression_list : Comma expression comma_expression_list
    """
    p[0] = [p[2]] + p[3]

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

```

中端

首先增加 `visitFunction` ，检查是否出现函数重定义、存在声明非函数变量冲突等，并为符号表添加新的符号 `Funcsymbol` ，如果函数有定义的话，还需要在访问函数参数后，再访问函数体body中的内容，此外，还需要还需要定义 `visitParameter` 和 `visitCall` ， `visitParameter` 检查了每个参数是否存在重定义报错

```
def visitFunction(self, func: Function, ctx: ScopeStack) -> None:
    func_name = func.ident.value
    func_type = func.ret_t.type
    conflict_symbol = ctx.lookup(func_name)
    if conflict_symbol is not None:
        if not isinstance(conflict_symbol, FuncSymbol):
            raise DecafDeclConflictError(f"Non-function type definition of
{func.name}")
        if func.body is not None and conflict_symbol.defined == True:
            raise DecafDeclConflictError(f"Multi-defined function
{func.name}")
        if func.body:

```

```
            conflict_symbol.defined = True
    else:
        func_symbol = FuncSymbol(func_name, func_type,
Scope(ScopeKind.LOCAL))
        for param in func.param_list:
            func_symbol.addParaType(param.var_t)
        func.setattr("symbol",func_symbol)
        ctx.declare(func_symbol)
        if func.body is not None:
            func_symbol.defined = True


    if func.body is None:
        return
    ctx.open(Scope(ScopeKind.LOCAL))

    for paramter in func.param_list:
        paramter.accept(self, ctx)
    if func.body is not None:
        for children in func.body:
            children.accept(self,ctx)

    ctx.close()

        # func.body.accept(self, ctx)

def visitParameter(self, parameter:Parameter, ctx: ScopeStack) -> None:
    if ctx.lookup(parameter.ident.value, only_top = True):
        raise DecafDeclConflictError
    parameter_symbol = VarSymbol(parameter.ident.value,
parameter.var_t.type)
    parameter.setattr("symbol",parameter_symbol)
    ctx.declare(parameter_symbol)
    # import pdb; pdb.set_trace()

def visitCall(self, call:Call, ctx:ScopeStack) -> None:
    func: FuncSymbol = ctx.lookup(call.ident.value)
    if func is None or not func.defined:
        raise DecafUndefinedFuncError

    if func.parameterNum != len(call.parameter_list):
        raise DecafBadFuncCallError(call.ident.value)

    call.ident.setattr("symbol",func)
    for paramter in call.parameter_list:
        paramter.accept(self,ctx)


```

针对 TAC 三地址码的生成，我才用的是“传参和调用分离”的策略，这需要在 `./utils/tac/tacinstr.py` 中为这两类地址添加对应的类，并在 `./utils/tac/tacvisitor.py` 中完善其访问函数

```
```

```python
class Param(TACInstr):
    def __init__(self, parameter:Temp,index:int) -> None:
        super().__init__(InstrKind.SEQ, [parameter], [], None)
        self.parameter = parameter
        self.index = index

    def __str__(self) -> str:
        return "PARAM %s" % str(self.parameter)

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

class Call(TACInstr):
    def __init__(self, output_temp:Temp, parameter_list: list[Temp], func_label: Label) -> None:
        super().__init__(InstrKind.SEQ, [output_temp], parameter_list, func_label)
        self.output = output_temp
        self.parameter_list = parameter_list
        self.function = func_label

    def __str__(self) -> str:
        # import pdb; pdb.set_trace()
        return "%s = CALL %s" % (str(self.output), str(self.function))

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


```

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

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

在 `./frontend/tacgen/tacgen.py` 中，修改 `transform` 函数使其可以遍历所有函数，并增加 `visitCall`，`visitFunction` 和 `visitParamter` 函数，同时为 `LabelManager` 添加 `label` 属性，用于函数label增加和查找

```python
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
        if astFunc.body is None:
            continue
        labelManager.putFuncLabel(funcName,len(astFunc.param_list))
        emitter = TACFuncEmitter(labelManager.labels[funcName],
len(astFunc.param_list), labelManager)
        astFunc.accept(self, emitter)
        tacFuncs.append(emitter.visitEnd())
    return TACProg(tacFuncs)
```

```python
def visitFunction(self, func: Function, mv: TACFuncEmitter) -> None:
    for parameter in func.param_list:
        # import pdb; pdb.set_trace()
        parameter.accept(self, mv)
    func.body.accept(self,mv)

def visitParameter(self, parameter: Parameter, mv: TACFuncEmitter ) -> None:
    # import pdb; pdb.set_trace()
    temp = mv.freshTemp()
    symbol = parameter.getattr("symbol")
    symbol.temp = temp
    return temp

def visitCall(self, call:Call, mv:TACFuncEmitter) -> None:
    temp_list = []
    for parameter in call.parameter_list:
        parameter.accept(self,mv)
        temp_list.append(parameter.getattr("val"))
    for i in range(len(temp_list)):
        mv.visitParameter(temp_list[i],i)
    output = mv.visitCall(mv.labelManager.getFuncLabel(call.ident.value),
temp_list)
    call.setattr("val",output)
```

```python
class LabelManager:
    """
    A global label manager (just a counter).
    We use this to create unique (block) labels accross functions.
    """

    def __init__(self):
        self.nextTempLabelId = 0
        self.labels = {}

    def freshLabel(self) -> BlockLabel:
        self.nextTempLabelId += 1
        return BlockLabel(str(self.nextTempLabelId))

    def getFuncLabel(self, name:str) -> FuncLabel:
        return self.labels[name]
```

```
    def putFuncLabel(self, name:str, parameter_num:int) -> None:
        self.labels[name] = FuncLabel(name,parameter_num)
```

后端

后端是本实验的难点，关键在于要理解 `riscvasmemitter.py` 中的 `RiscvSubroutineEmitter` 类对函数栈的作用，以及 `bruteregalloc.py` 中传参，调用函数和返回值是如何对应到 Risc-V 汇编以及函数栈上的操作。

具体来说，首先给出函数栈设计，和常规设计不同，我在设计时采用了将所有参数都放到函数栈上传参的策略

```
++++++++++++++++++++++
(Caller-saved registers)
++++++++++++++++++++++
Arg n-1
Arg n-2
...
Arg 1
Arg 0
++++++++++++++++++++++ FP
(Callee-saved registers)
++++++++++++++++++++++ SP
```

具体操作上，我首先在 `riscv.py` 中增加了 `Param`，`Call` 和 `FPSet` 三个类，前两个类和前端中端一脉相承，可以理解为 Risc-v 伪代码，第三个类则是用来设定FP，确定栈帧位置。同时在 `riscvasmemitter.py` 中设置对应的访问函数调用 `Param` 和 `Call` 类 。接下来，我重构了 `BruteRegAlloc` 类的 `allocForLoc` 方法，在这里我实现的是将前面提到的伪代码翻译为真正的 Risc-V 代码，实现 Caller 侧的函数调用工作，对于 `Param` ，用 `mv` 指令将参数放到栈对应的位置，对于 `Call` ，需要依次翻译 Store Caller-saved registers ， 函数调用， Load Caller-saved registers 和将存储在寄存器 `A0` 中的结果返回这四步。最后，在 `riscvasmemitter.py` 的 `RiscvSubroutineEmitter` 类中，通过重构 `emitLoadFromStack` 和 `emitEnd` 函数，实现 Callee 侧的参数读取和 Callee-saved registers 维护（其中callee-saved registers的维护代码大部分由框架给出，我们只需要特别增加对 `ra` 和 `fp` 的维护就行了）

```
class Param(TACInstr):
    def __init__(self, parameter: Temp, index: int) -> None:
        super().__init__(InstrKind.SEQ, [], [parameter], None)
        self.index = index
        self.parameter = parameter

    def __str__(self) -> str:
        return "param " + Riscv.FMT2.format(
            str(self.index),
            str(self.parameter)
        )

class Call(TACInstr):
    def __init__(self, output: Temp, func_label: Label) -> None:
        super().__init__(InstrKind.SEQ, [output], [], func_label)
        self.func_label = func_label

```

```
    def __str__(self) -> str:
        return "call " + str(self.func_label.func)

class FPSet(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)
        )
```

```
class RiscvAsmEmitter(AsmEmitter):
    ...
# in step9, you need to think about how to pass the parameters and how to store and restore callerSave regs
def visitParam(self, instr:Param) -> None:
    self.seq.append(Riscv.Param(instr.parameter,instr.index))

# def visitParamPop(self, instr:Param)
def visitDirectCall(self, call : Call) -> None:
    self.seq.append(Riscv.Call(call.output, call.function))

```

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

    if isinstance(instr,Riscv.Param):
        # 将所有参数都放到函数栈上
        offset = 4 * instr.index
        reg  = self.allocRegFor(instr.srcs[0],True,loc.liveIn,subEmitter)
        subEmitter.emitNative(Riscv.NativeStoreWord(reg, Riscv.SP, offset))

    elif isinstance(instr, Riscv.Call):
        temp_list = []
        # 保存 caller-saved register
        for i in range(len(Riscv.CallerSaved)):
            reg = Riscv.CallerSaved[i]
            if reg.isUsed():
                temp_list.append(reg.temp)
                subEmitter.emitStoreToStack(reg)
            else:
                temp_list.append(None)

        temp = instr.dsts[0]
        if isinstance(temp, Reg):
            dstRegs.append(temp)
        else:
            dstRegs.append(self.allocRegFor(temp, False, loc.liveIn, subEmitter))
```

```
28
29          # 调用函数
30          subEmitter.emitNative(instr.toNative(dstRegs, srcRegs))
31
32          # 恢复 caller-saved register
33          for i in range(len(temp_list)):
34              if temp_list[i] is not None:
35                  subEmitter.emitLoadFromStack(Riscv.CallerSaved[i],
    temp_list[i])
36
37          # 返回结果
38          subEmitter.emitNative(Riscv.Move(dstRegs[0],
    Riscv.A0).toNative(dstRegs, [Riscv.A0]))
39
40      else:
41          for i in range(len(instr.srcs)):
42              temp = instr.srcs[i]
43              if isinstance(temp, Reg):
44                  srcRegs.append(temp)
45              else:
46                  srcRegs.append(self.allocRegFor(temp, True, loc.liveIn,
    subEmitter))
47
48          for i in range(len(instr.dsts)):
49              temp = instr.dsts[i]
50              if isinstance(temp, Reg):
51                  dstRegs.append(temp)
52              else:
53                  dstRegs.append(self.allocRegFor(temp, False, loc.liveIn,
    subEmitter))
54          subEmitter.emitNative(instr.toNative(dstRegs, srcRegs))
```

```
1   class RiscvSubroutineEmitter(SubroutineEmitter):
2       ...
3       def emitLoadFromStack(self, dst: Reg, src: Temp):
4           if src.index < self.parameter_num:
5               self.buf.append(
6                   Riscv.NativeLoadWord(dst, Riscv.FP, src.index*4)
7               )
8               return
9           if src.index not in self.offsets:
10              raise IllegalArgumentException()
11          else:
12              self.buf.append(
13                  Riscv.NativeLoadWord(dst, Riscv.SP, self.offsets[src.index])
14              )
15      def emitEnd(self)
16          ...
17          self.printer.printInstr(
18              Riscv.NativeStoreWord(Riscv.RA, Riscv.SP,
    4*len(Riscv.CalleeSaved))
19          )
20
21          self.printer.printInstr(
```

```
            Riscv.NativeStoreWord(Riscv.FP, Riscv.SP,
4*len(Riscv.CalleeSaved)+4)
        )

        self.printer.printInstr(
            Riscv.FPSet(self.nextLocalOffset)
        )
        ...

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

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

```

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
   ```

   我更倾向于采用“传参和调用分离”的策略，因为在我看来这样的作法将函数操作相比”一整条函数调用 “拆分的粒度更细，一般来说这样做可以提高代码的灵活性，为之后更多功能的开发提供便利。比如说，在函数调用的场景下，当我们需要给函数传递默认值的时候，”传参和调用分离“ 只需要对某一条 PARAM 指令进行修改就可以，而 “一整条函数调用” 则可能需要更加复杂的更改才能实现这一操作。但更细粒度可能导致的问题是参数和函数容易出现不匹配的情况，可以想见，随着开发功能的增多，参数和函数二者的 TAC 代码之间会增加更多的处理逻辑，如果设计不当的话，参数和函数调用很容易出现不匹配的情况。

   相比之下，”一整条函数调用“在代码简洁性和易读性上更有优势，同时原子化的操作也有助于减少前面提到的参数和函数不匹配的问题，但缺点在于粗粒度的代码往往不够灵活，比如说当我们只需要修改某个参数的传参方式时，却需要为此修改整个一条函数调用语句，这样做在一定程度上增加了维护的复杂度。

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？

保存寄存器的原则是应该保存尽量少的寄存器，因为对 memory 的访问本身是十分耗时的，减少寄存器的保存可以降低函数调用的时间开销和空间开销

如果寄存器全部都由一方保存的话，每次函数调用的空间开销和时间开销都会严格等于一个理论上的最大值。而如果将他们分成两类，对于 caller-saved 寄存器，往往不建议 caller 长期使用（或者说至少不建议在函数调用前后使用，因此它们被称为易失性寄存器），在实际操作中这些寄存器可能存了一些临时变量，会被迅速使用迅速丢弃，只有在一些不可避免要用的情况下，caller才去保存这部分寄存器。而另一类在 caller 中被长期使用的寄存器，如果这些寄存器也交给caller来保存的话，那么每次的开销又会很大，因此，这部分寄存器callee应该尽量少地去用，如果callee一定要用，则需要由callee来维护这些寄存器的值，也就是callee-saved寄存器。总之，将寄存器拆成两类来保存，可以最大程度上减小维护寄存器带来的开销。

ra 是 caller-saved 寄存器，跳转指令（如 `jalr`）会在进入函数之前就修改 `ra` 的值，因此ra的值需要由caller来维护，而不能交给callee在函数开始时保存。尽管修改编译器让每次函数的调用ra都遵循callee-saved的规范，的确不会出现ra被覆盖出错的问题，但由于ra具有易失性，因此在常规的caller-saved和callee-saved规范中，ra应该被划分为caller-saved寄存器更符合大部分编译器的逻辑以及Risc-V的设计规范