Stage-3 Report

玄镇 智00 2020010762

step 7

step7中，我完成了作用域和块语句。在符号表构建的扫描中，我们实现了动态维护作用栈。每当开启一个代码块时，我们新建一个作用域并压栈；而当退出代码块时，要弹栈并关闭此作用域。

具体说来，在符号表构建时，我们重写了扫描AST根节点的`Namer.visitBlock`函数：

遇到一个代码块，我们先开启一个局部代码块`ctx.open(Scope(ScopeKind.LOCAL))`，在退出时，我们关闭这个局部代码块`ctx.close()`

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

在后端中，我们在CFG中实现了`unreachable`函数，来判断一个基本块是否可达。具体逻辑是利用`getPrev`函数，判断其是否能通过其他基本块到达：

```
def unreachable(self,id):
        if not self.getPrev(id):
            return True
        return False
```

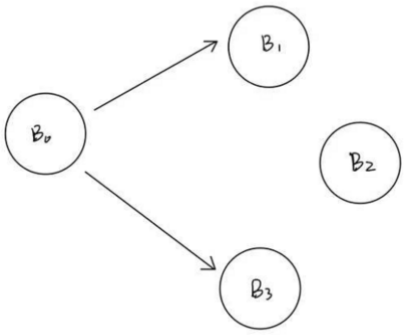
同时在寄存器分配算法中，对于不可到达的基本块，直接跳过，不分配寄存器。

```
def accept(self, graph: CFG, info: SubroutineInfo) -> None:
        subEmitter = self.emitter.emitSubroutine(info)
        for bb in graph.iterator():
            if  bb.id > 0 and graph.unreachable(bb.id):
                pass
            else :
                if bb.label is not None:
                    subEmitter.emitLabel(bb.label)
                self.localAlloc(bb, subEmitter)
        subEmitter.emitEnd()
```

> 思考题：
>
> 首先来看看这段代码的三地址码：

```
FUNCTION<main>:
_T1 = 2
_T0 = _T1
_T2 = 3
_T3 = (_T0 < _T2)
if (_T3 == 0) branch _L1
_T5 = 3
_T4 = _T5
return _T4
return _T0
_L1:
return
```

由此可知，控制流程图如下：



step 8

step8中，我们实现了`for/dowhile/continue`

具体说来，首先我们在`tree`中增加了`For/While/Continue`等节点，

```
class For(Statement):
    """
    AST node of for statement.
    """

    def __init__(self, init : Optional[Union[Expression,Declaration]], cond : Optional[Expression], update : Optional[Expression], body : Statement) -> None:
        super().__init__("for")
        self.init = init or NULL
        self.cond = cond or NULL
        self.update = update or NULL
        self.body = body

    def __getitem__(self, key: int) -> Node:
        return (self.init,self.cond,self.update,self.body)[key]

    def __len__(self) -> int:
        return 4

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

    def is_leaf(self):
```

```
        return True
```

其中以`For`为例，我们需要注意for循环的语法

```
statement:
    : ...
    | 'for' '(' expression? ';' expression? ';' expression? ')' statement
    | 'for' '(' declaration expression? ';' expression? ')' statement
    ...
```

因此在初始化时，初始条件`init`既可能为`expression`，也可能为`declaration`。同时还要注意可能会有`for(;;)`这类的语句，因此需要考虑`init`为`NULL`的情形。

我们需要在`visitor`中增加`Visitor`的默认函数

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

需要为这些节点设置相应的词法与语法，在`lex`保留字中增加相应保留字，同时在`ply_parser`中增加相应的扫描文法。

```
def p_for(p):
    """
    statement_matched : For LParen expression Semi expression Semi expression RParen statement_matched
        | For LParen declaration Semi expression Semi expression RParen statement_matched
    statement_unmatched : For LParen expression Semi expression Semi expression RParen statement_unmatched
        | For LParen declaration Semi expression Semi expression RParen statement_unmatched
    """
    p[0] = For(p[3],p[5],p[7],p[9])
```

这里需要注意`for`语句中既有可能第一项不声明，同时也有可能声明新变量，同时还要注意为空的情形，由于测例中语句为空情形有限，我们只需补充：

```
def p_for_empty(p):
    """
    statement_matched : For LParen Semi Semi RParen statement_matched
    statement_unmatched : For LParen Semi Semi RParen statement_unmatched
    """
    p[0] = For(NULL,NULL,NULL,p[6])

def p_continue_empty(p):
    """
    statement_matched : For LParen declaration Semi expression Semi RParen statement_matched
    statement_unmatched : For LParen declaration Semi expression Semi RParen statement_unmatched
    """
    p[0] = For(p[3],p[5],NULL,p[8])
```

在语义分析阶段，我们给`Namer`增加了`visitFor`函数

```
def visitFor(self,stmt:For,ctx:ScopeStack)->None:
        ctx.open(Scope(ScopeKind.LOCAL))
        stmt.init.accept(self,ctx)
        stmt.cond.accept(self,ctx)
        stmt.update.accept(self,ctx)
        ctx.openLoop()
        stmt.body.accept(self,ctx)
        ctx.closeLoop()
        ctx.close()
```

这里需注意for循环自带一个作用域。

在中间代码生成部分，我们补充了函数`visitFor`

```
def visitFor(self,stmt: For,mv:FuncVisitor) -> None:
        if stmt.init is not NULL:
            stmt.init.accept(self,mv)
        beginLabel = mv.freshLabel()
        loopLabel = mv.freshLabel()
        breakLabel = mv.freshLabel()
        mv.openLoop(breakLabel, loopLabel)
        mv.visitLabel(beginLabel)
        if stmt.cond is not NULL:
            stmt.cond.accept(self, mv)
            mv.visitCondBranch(tacop.CondBranchOp.BEQ,
stmt.cond.getattr("val"), breakLabel)
        stmt.body.accept(self, mv)
        mv.visitLabel(loopLabel)
        if stmt.update is not NULL:
            stmt.update.accept(self,mv)
        mv.visitBranch(beginLabel)
        mv.visitLabel(breakLabel)
        mv.closeLoop()
```

这里我们需要考虑`init`等是否为空的情形，同时`update`应该在`loopLabel`之后`branch`之前，而不能将`update`和`body`放到一起。

`dowhile/continue`的实现思路与`for`同理。

> 思考题：
>
> 第二种翻译方式更好。因为在循环开始时，两种翻译方式都要执行`cond`和`beqz BREAK_LABEL`语句，但是当执行过一个`body`或者`continue`以后，第一种方式需要执行
>
> ```
> br BEGINLOOP_LABEL
> cond 的 IR
> beqz BREAK_LABEL
> ```
>
> 才能进入下一个循环或者退出。然而第二种方式只需
>
> ```
> cond 的 IR
> bnez BEGINLOOP_LABEL
> ```
>
> 就可以进入下一个循环或者退出。因此第二种翻译方式执行的指令条数更少。