

STAGE4：函数和全局变量 实验报告

邢竞择 2020012890

1 Step 9

1.1 实验内容

1.1.1 前端

为了实现函数参数列表和表达式列表的读取，在 `parser.y` 中添加了如下规则（以参数列表为例）。

```
1 Formallist : /* EMPTY */
2             { $$ = new ast::VarList(); }
3             | FormallistRecurse Type IDENTIFIER
4             { $1->append(new ast::VarDecl($3, $2, POS(@3))); $$ = $1; }
5             ;
6 ExprList   : /* EMPTY */
7             { $$ = new ast::ExprList(); }
8             | ExprListRecurse Expr
9             { $1->append($2); $$ = $1; }
10            ;
```

此外还添加了 `CallExpr` 的解析。

1.1.2 中端

为了支持函数和声明分离，需要在 `SemPass1` 中进行检查，在访问 `ast::FuncDefn` 节点时，若发现符号表中已经有同名的定义，则需检查符号表中的符号是函数类型，且参数与当前函数相同，并且不能包含函数定义。

为了支持函数调用，需在 `SemPass2` 中检查 `Call` 的表达式列表的求值结果与函数参数列表的类型相同。

添加了三地址码 `PARAM`，`PUSH` 和 `CALL`，具体设计与实验指导书一致。

在 `translation` 中翻译 `CallExpr` 节点时，需要先将无法放在寄存器中的参数 `push` 到栈上，然后生成前八个参数的 `PARAM` 三地址码。

```
1 void Translation::visit(ast::CallExpr *e) {
2     util::List<ast::Expr *> *arguments = e->params;
3     util::Vector<Temp> *param_temp_list = new util::Vector<Temp>();
4     std::vector<Temp> param_temp_list;
5     for (auto ait = arguments->begin(); ait != arguments->end(); ait++) {
6         (*ait)->accept(this);
7         param_temp_list->push_back(tr->genParam((*ait)->ATTR(val)));
8         param_temp_list.push_back((*ait)->ATTR(val));
9     }
10    e->ATTR(val) = tr->genCall(e->ATTR(sym)->getEntryLabel(), param_temp_list);
11    for (int i = param_temp_list.size() - 1; i >= 8; i--)
```

```

12     tr->genPush(param_temp_list[i]);
13     int order = 0;
14     for (auto ait = arguments->begin(); ait != arguments->end() && order < 8;
15         ait++)
16         tr->genParam((*ait)->ATTR(val), order++);
17     e->ATTR(val) = tr->genCall(e->ATTR(sym)->getEntryLabel());
18 }

```

为了在函数调用开头将物理寄存器与虚拟寄存器绑定起来，我额外增加了三地址码 `BindRegToTemp`，它不生成汇编代码，仅将物理寄存器附上对应的虚拟寄存器。

对于新增加的 TAC 指令，也需要相应的修改 `dataflow` 中 `Live` 集合的计算，这部分较为简单。

1.1.3 后端

后端主要是为新增的三地址码生成汇编代码，参考框架可以较为容易地实现。

1.2 思考题

1. MiniDecaf 的函数调用时参数求值的顺序是未定义行为。试写出一段 MiniDecaf 代码，使得不同的参数求值顺序会导致不同的返回结果。

答：

```

1  int func(int x, int y) {
2      return x * y;
3  }
4  int main() {
5      int x = 2;
6      return func(x = x + 1, x = x * 2);
7  }

```

若先计算第一个参数，则结果为 $3*6=18$ ；若先计算第二个参数，则结果为 $5*4=20$ 。

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？

答：标准中关于两类寄存器的设定受到了多种因素的影响。

- 一个函数可能会多次调用，但它事实上可能只要使用少量寄存器，若所有的寄存器都是 `caller-saved`，那么在调用函数前就会花费大量的时间将寄存器的值存到栈上，导致代码庞大，且运行时会引起巨大的访存开销
- 如果所有寄存器都是 `callee-saved`，那么函数如果要使用寄存器，就必须进行栈操作，这也是我们希望减少的
- 在 RISC-V 的规范中，一部分寄存器是 `caller-saved`，这使得 `callee` 可以直接使用这些寄存器，减少栈上的操作；同时这些寄存器只是全体寄存器中的一部分，这使得 `caller` 有一定的余地来保存重要的临时变量。这两个目的权衡之下，形成了现在的标准

由于 `ra` 在 `call` 发生时就被修改，此时 `callee` 还没运行，所以必须由 `caller` 保存。

2 Step 10

2.1 实验内容

2.1.1 前端

修改 `parser`，允许 FOD 中存在 `DeclStmt`

2.1.2 中端

在 `SemPass1` 中，检查变量的初始化值是 `INT_CONST` 类型。

添加了三地址码 `LOAD_SYMBOL`，`LOAD` 和 `STORE`，具体设计与实验指导书一致。

在 `translation` 阶段访问 `AssignExpr` 时，若左侧值是全局变量，则生成 `LOAD_SYMBOL` 和 `STORE` 三地址码，将结果迅速保存到内存中；在访问 `LvalueExpr` 时，若符号是全局变量，则生成三地址码将数据从内存 `LOAD` 到符号绑定的临时变量中。

对于新增加的 TAC 指令，也需要相应的修改 `dataflow` 中 `Live` 集合的计算，这部分较为简单。

2.1.3 后端

在程序的 `preamble` 之前，遍历 `GlobalScope`，对有初始化和无初始化的全局变量，分别按照规定格式存储在 `.data` 段和 `.bss` 段。

2.2 思考题

1. 写出 `la v0, a` 这一 RiscV 伪指令可能会被转换成哪些 RiscV 指令的组合（说出两种可能即可）。

答：第一种可能是 `auipc` 与 `addi` 的组合，即通过全局变量相对于当前的 `pc` 的位置来计算它在内存中的地址；第二种可能是只使用 `addi`，即通过 `gp` 以及变量相对于 `gp` 的位置来计算。