stage-4 实验报告

姓名：郑友捷　学号：2021010771　班级：计14

step 9

step9引入了函数部分，对前后端代码均作出了较大程度的修改。

词法语法分析

依据程序要求增添了`parameterlist`与`parameter`节点表示函数声明时的参数列表与单个参数，增添了`expressionlist`节点表示函数调用时传入的参数表达式列表，用`call`节点代表函数调用。

在词法分析方面，做出如下修改：

1. 允许程序包含多个函数，而非只有主函数。
2. 对函数声明与定义做出了词法分析
3. 对函数调用做出了词法分析

语义分析

由于加入了函数，因此全局作用域需要加入函数符号的检查。

1. 当声明或者定义函数时，检查函数符号是否在全局作用域中。若不存在则加入到全局作用域，若存在，检查是否为定义。若为定义则报错。
2. 当进入函数分析时，先新开一个作用域，让函数参数与函数正文共享一个作用域，进而检查参数与正文内容的正确性。
3. 当调用函数时，检查全局作用域是否有对应的函数符号，若没有则报错。若有则检查函数参数的长度是否一致（当前step不需要加入类型检查），若不同则报错。

中间代码生成

1. 由于中间代码生成是按照函数为单位进行生成的，因此我们需要在一开始遍历程序中所有函数节点，为其各自分配一个funcvisitor。若函数已被定义，则对其参数与正文进行中间代码生成。
2. 对函数参数先行分配一个寄存器，方便在函数正文调用到它时可以直接获得其对应的寄存器。
3. 处理函数调用语句时，先检查传入的参数表达式，对每一个参数表达式生成对应的中间代码之后同时生成`PARAM`语句，代表传入参数。之后生成`call`语句代表调用函数。

目标代码生成

对函数生成汇编代码时，需要考虑栈帧大小以及寄存器分配的问题，同时生成目标代码也是以函数为基本单位的。

1. 在进入函数时，先手动将`sp/fp/ra`等寄存器存储到栈帧上，并存储`callee_saved`寄存器，并为其参数分配好对应的寄存器`a0`到`a8`，并将这些寄存器的内容`spill`到栈帧上。若参数过多，剩余参数不需要进行分配，因为已经存在了栈帧上。该操作的目的是为了让所有参数都可以存储在栈帧上。
2. 在分析到`param`语句时，需要为当前的`temp`分配寄存器或者栈帧空间。若是个数小于8，则直接分配参数寄存器。若个数大于8，则用一个向量存储起来。

3. 在进入到`call`语句时，先保存所有的`caller_saved`寄存器，并将多余的参数在向量中倒序从高地址到低地址插入到栈帧中，代表传递参数，之后生成`call`语句。
4. 调用完毕之后，将对应的参数从栈帧中弹出，并恢复`caller_saved`寄存器。
5. 当前函数结束时，从栈帧上恢复`callee_saved`寄存器和`sp/fp/ra`寄存器。

值得注意的是，本程序使用`fp`寄存器存储父函数调用`call`语句时的栈指针位置，即是当前函数的栈基址，用于快速从栈帧上获取当前函数的参数。

step10

step10要求支持全局变量，相比于函数部分简单了许多。

词法语法分析

加入全局变量之后需要支持在全局进行声明，因此添加了`globaldeclaration`节点用于全局变量声明与初始化。

因此`program`节点不止会生成函数，还可能生成变量声明。

语义分析

在语义分析时需要加入对全局变量节点的检查。

1. 若当前声明的变量已经在全局作用域中出现，则进行报错（本程序认为不允许重复声明或者存在声明与定义，因为默认全局变量的声明与定义是绑定的）
2. 为当前变量新建一个符号并声明其为全局符号
3. 若存在初始化语句则进行初始化语句检查，并为该全局符号赋予初值
4. 将其加入到全局作用域与符号表中。

中间代码生成

对全局变量的中间代码生成需要考虑如下：

1. 在全局部分不需要为中间代码生成特定的TAC语句（若想生成也没有问题），可以直接让后端对照符号表进行全局变量定义。
2. 在引用变量时，需要判断当前变量是否为全局变量。若不是全局变量，则直接返回当前变量对应的寄存器，否则需要分配一个新的寄存器之后，载入全局变量当前的值之后再将其赋予给当前节点的属性值。
3. 在修改变量时，需要判断当前变量是否为全局变量。若是，则需要将该修改值写入到全局变量对应的地址，代表修改了全局变量。

目标代码生成

目标代码生成主要集中在全局变量的定义部分。

1. 为了获取前端语义分析时的符号表，需要修改`main.py`将符号表传入给后端。
2. 后端先遍历所有的全局符号，找到所有未初始化的全局变量，划入到`.bss`段。

3. 再次遍历全局符号，找到所有已经初始化的全局变量，划入到`.data`段，并用`.word`为其赋予初值。

思考题

1. 代码如下：

```
int f(int x, int y) {
    return x - y;
}

int main() {
    int x = 1;
    return f(x=x+1, x=x+1);
}
```

2. 为何要引入callee_saved和caller_saved?

   原因：

   1. 若将所有寄存器交由一方报错，每一次将所有的寄存器都压栈出栈，会导致大量的内存访问，导致性能下降与内存占用上升。而将寄存器进行分工保存，调用者和被调用者只需要保存各自需要保存的寄存器，可以大大减少读取内存次数，减少调用前的准备与调用后恢复的耗花费。
   2. 系统中的寄存器有些寄存器是保存声明出来的变量，这类变量的生命周期较长，对应寄存器的使用周期也较长。而有些寄存器保存的是临时变量，生命周期较短，属于用完即废。

      对于被调用者，若想使用临时变量，此时由于其生命周期较短，对外界影响不大，应当追求速度，故应当尽可能减少因需要恢复寄存器带来的读取耗时，故`callee`一般不加保护。此时若是`caller`中想要使用，则需要由`caller`进行保护。

      若被调用者想使用声明的变量（如全局变量）等，如果改变了这些变量，因为其生命周期较长，所以`caller`中的变量也会受到影响，所以需要由`callee`进行保护。

3. 为何`ra`是`caller_saved`寄存器?

   因为`ra`保存的是子函数调用后的返回值。当子函数调用完成之后，`ra`可能发生变化，而`caller`有可能需要利用子函数的返回值，即`ra`变化后的值，因此需要由`caller`来处理变化后的值，而不是让`callee`来恢复。

4. 对于`la v0,a`

   1. 对于非PIC编译参数下的代码，设a代表链接之后的地址数，delta=a-pc

      ```
      auipc v0, delta[31:12]+delta[11]
      addi v0, v0,delta[11:0]
      ```

   2. 对于PIC编译参数下的代码，delta = GOT[a] - pc

      ```
      auipc v0, delta[31:12]+delta[11]
      lw v0,v0,delta[11:0]
      ```