Compilers Principals - Lab4

Zhixin Zhang, 3210106357

1 实验内容

本次实验旨在将中间代码（IR）转换为汇编代码. 我们首先根据 function 将中间代码划分为不同的块，每个块中都包含若干个 IR 节点；然后，我们对于每个中间代码块，首先处理其中可能包含的函数参数，变量等信息，然后为每其指定其在内存中的位置，该部分为 IR 块的预处理部分；最后，我们根据 IR 节点的不同类型，将其转化为对应的汇编代码（同样由相应的 ASM 节点保存），将汇编节点依次输出为汇编代码.

我们通过

```
make compiler
./compiler <input file> [output file]
```

可以对输入的 sy 文件进行语法和语义的检查， 如果可以正确解析出语法树并且通过类型检查和数组检查，程序将正常退出并返回 0，并且将生成汇编代码保存到 output file（如果没有定义，则默认为 asm.out）， 同时在错误流中显示：

```
Parse success!
```

否则，程序将汇报错误.

2 代码实现

2.1 主接口

main.cc 在 lab3 的基础上，增加了汇编代码生成的部分：

```
  string ASM_OUT = "asm.out";
  if (argc >= 3) ASM_OUT = string(argv[2]);
  ofstream asm_out(ASM_OUT);
  ASM _asm(ir);
  _asm.print(asm_out);
```

2.2 汇编代码存储格式

2.2.1 class _ASM

_ASM 类是汇编指令节点信息的存储基类，定义如下：

```
class _ASM {
public:
    string type;
    virtual ~_ASM() = default;
    virtual void print(ostream& out) = 0;
};
```

2.2.2 相关派生

在此基础上，我们定义了不同类型的汇编指令类，例如二院操作指令类：`ASM_binop`：

```
class ASM_binop : public _ASM {
public:
    string lv, rv1, rv2, op;
    ASM_binop(string _op, int lv, int rv1, int rv2) : lv(REG(lv)), rv1(REG(rv1)), rv2(REG(rv2)) {
        _ASM::type = "binop"s;
        op = upd(_op);
    }
    void print(ostream& out) { out << op << " " << lv << ", " << rv1 << ", " << rv2 << "\n"; }
};
```

2.3 IR 到汇编代码转换

2.3.1 class IRs

IRs 类用于存储中间代码块及其解析方法，解析方法包括，IR 块中的函数名，变量，参数提取，IR 块中内存分配.

```
class IRs : public IR {
public:
    IRs() {}
    IRs(const IR &ir) : IR(ir) {}
    set<string> var, par;
    string fname;
    map<string, int> pos;
    void get_func_info();
    void place_var();
};
```

2.3.2 class ASMs

ASMs 类用于存储汇编指令块，一个中间代码块将对应地转化为一个汇编指令块.

```
class ASMs {
public:
    vector<unique_ptr<_ASM>> child;
    string fname;
    void print(ostream& out) {
        for (auto &e : child) e->print(out);
    }

    template<typename T>
    void push_back(T o) { child.push_back(unique_ptr<_ASM>(o)); }

    template<typename... Args> void PUSH(Args... args) {
        (child.push_back(unique_ptr<_ASM>(args)), ...);
    }
};
```

2.3.3 汇编指令生成

我们使用 `parse_ir` 函数来生成相应的汇编指令. 为了方便生成，我们通过如下方式生成一个 `_ASM` 节点：

```
#define A(type, ...) (new ASM_##type(__VA_ARGS__))
```

例如对于二元运算 (op, lv, v1, v2)，我们构造汇编指令：

```
o.PUSH(A(lw, Reg["t0"], Reg["fp"], bl.pos[x->v1]),
       A(lw, Reg["t1"], Reg["fp"], bl.pos[x->v2]),
       A(binop, x->op, Reg["t2"], Reg["t0"], Reg["t1"]),
       A(sw, Reg["t2"], Reg["fp"], bl.pos[x->lv]));
```

这对应指令：

```
lw t0, fp, offset(v1)
lw t1, fp, offset(v2)
op t2, t0, t1
sw t2, fp, offset(lv)
```

即分别从内存中取出 v_1, v_2，执行运算后保存回 lv.

对于自带的的 `read/write` 语句，我们使用 `ecall` 指令：

```
if (x->name == "read")
  o.PUSH(A(li, Reg["a0"], 6), A(ecall));
elif (x->name == "write")
  o.PUSH(A(li, Reg["a0"], 1), A(lw, Reg["a1"], Reg["sp"], 0), A(ecall));
```

其余实现细节详见代码 `src/asm.hpp`.

3 测试结果

```
python3 test.py ./compiler lab4 -l
```

tests 下的测试样例全部通过：

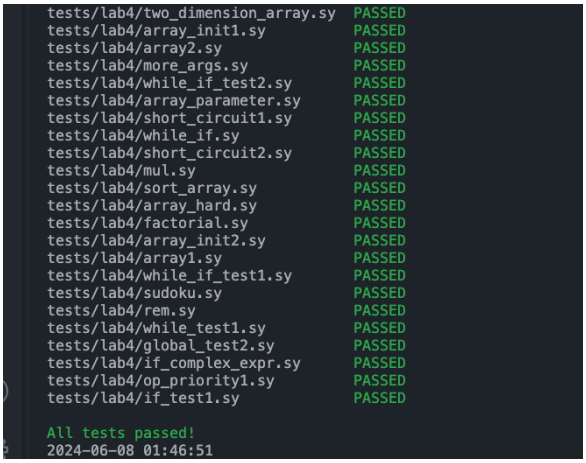

图 1 All tests passed!