《编译原理》实验报告

李思哲
1900013061

一、编译器概述

1. Lab 完成情况

第一部分：koopa IR　　第二部分：RISCV

其中前端完成了 lv1-lv8 的全部实现，以及 lv9.1 的内容；

后端完成了 lv1-lv8 的全部实现。

参与测评的前端代码：https://github.com/sizhelee/sysy-make-template/tree/46f616581fc0fb22f4a1b6a8430842641f44249d

参与测评的前后端代码：https://github.com/sizhelee/sysy-make-template

2. 编译器基本功能

总体来说，编译器可以将 SysY 语言编译成为 RISCV 汇编，通过 Koopa IR 作为中间表示，首先将 SysY 语言编译到 Koopa IR，再从 Koopa IR 生成 RISCV 汇编。

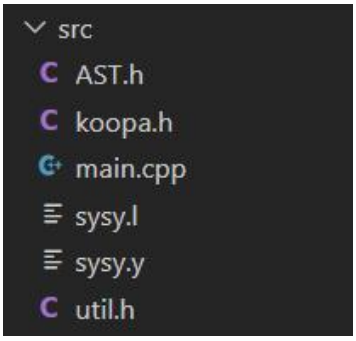
具体讲，从 SysY 到 Koopa IR 的编译过程可处理表达式、常量和变量、语句块、分支和循环、函数和全局变量、以及一维数组；从 Koopa IR 到 RISCV 的生成过程可处理上述除数组外的全部。

3. 编译器主要特点

优：数据结构简明；

缺：无法判别程序合法性，只能按照特定语法规则进行编译。

二、编译器设计



1. 主要模块组成

编译器文件结构如图所示。

其中 AST.h 实现了前端的 AST 架构，通过语法分析得到的 AST，逐步解析语法树并输出 Koopa IR 程序；

Main.cpp 完成后端生成 RISCV 代码的过程；

Sysy.l 和 sysy.y 描述词法和语法规则，用于解析 SysY 程序；

Util.h 中包含后端所需辅助函数及调试所需函数等。

2. 主要数据结构

(1) 语法分析

```
CompUnit    ::= CompUnit Decl | Decl
              | CompUnit FuncDef | FuncDef;
FuncDef     ::= BType IDENT "(" ")" Block
              | BType IDENT "(" FuncFParams ")" Block;
FuncType    ::= "void" | "int";
FuncFParams ::= FuncFParam {"," FuncFParam};
FuncFParam  ::= BType IDENT;
Block       ::= "{" myBlockItem "}";
myBlockItem ::= myBlockItem BlockItem | ;
BlockItem   ::= Decl | Stmt;
Stmt        ::= LVal "=" Exp ";"
              | Exp ";"
              | ";"
              | Block
              | "return" [Exp] ";";
              | "if" "(" Exp ")" Stmt ["else" Stmt];
              | "while" "(" Exp ")" Stmt;
Exp         ::= LOrExp;
ConstExp    ::= Exp;
PrimaryExp  ::= "(" Exp ")" | LVal | Number;
Number      ::= INT_CONST;
UnaryExp    ::= PrimaryExp
              | UnaryOp UnaryExp
              | IDENT "(" ")"
              | IDENT "(" FuncRParams ")"
FuncRParams ::= Exp {"," Exp};
UnaryOp     ::= "+" | "-" | "!";
MulExp      ::= UnaryExp | MulExp ("*" | "/" | "%") UnaryExp;
AddExp      ::= MulExp | AddExp ("+" | "-") MulExp;
RelExp      ::= AddExp | RelExp ("<" | ">" | "<=" | ">=") AddExp;
EqExp       ::= RelExp | EqExp ("==" | "!=") RelExp;
LAndExp     ::= EqExp | LAndExp "&&" EqExp;
LOrExp      ::= LAndExp | LOrExp "||" LAndExp;
Decl          ::= ConstDecl | VarDecl;
ConstDecl     ::= "const" BType myConstDef ";";
myConstDef    ::= myConstDef ',' ConstDef | ConstDef;
BType         ::= "int";
ConstDef      ::= IDENT "=" ConstInitVal
                | IDENT "[" ConstExp "]" "=" ConstInitVal;
ConstInitVal  ::= ConstExp;
                | "{" "}";
```

```
                | "{" ConstExp {"," ConstExp} "}";
VarDecl       ::= BType myVarDef ";";
myVarDef      ::= myVarDef ',' VarDef | VarDef;
VarDef        ::= IDENT
                | IDENT "[" ConstExp "]"
                | IDENT "=" InitVal
                | IDENT "[" ConstExp "]" "=" InitVal;
InitVal       ::= Exp
                | "{" "}"
                | "{" Exp {"," Exp} "}";
LVal          ::= IDENT
                | IDENT "[" Exp "]";
```

(2) AST 结构

```cpp
// 所有 AST 的基类
class BaseAST {
    public:

    std::vector<BaseAST*> son; // 记录下层的结构
    TYPE type;
    int val;
    char op;
    bool isident = false;
    bool isarray = false;
    bool isint = false, ret = false;
    bool isif = false, iswhile = false, isbreak = false, iscontinue = false;
    std::string ident;

    BaseAST() = default;
    BaseAST(TYPE t): type(t){}
    BaseAST(TYPE t, char o): type(t), op(o) {}

    virtual ~BaseAST() = default;
    virtual void Dump(std::string& str0) const = 0; // 向字符串str0输出本层代码

    virtual std::string dump2str(std::string& str0) // 向str输出本层代码并返回结果的寄存器
    {
        return "";
    }
};
```

SysY 程序经过 Parser 解析得到 AST 后，为每一个解析得到的模块设计一个 AST 类。

其中最重要的成员函数包括 dump 和 dump2str，主要功能均为向字符串 str0 中输出 Koopa IR 代码，区别在于是否需要返回结果寄存器。虽然看起来 dump2str 函数可以完全替代 dump 函数，但值得注意的是在真正实现过程中，这两个函数往往用于区别全局/局部变量等。

重要的成员变量包括 son，用于记录推到过程中的规约情况，存储规约得到的非终结符和终结符，ident 用于记录变量名相关信息，val 用于记录和值相关的信息。

(3) 符号表、函数表相关结构

```
int expNumCnt = 0, symTabCnt = 0, ifNumCnt = 0, allsymTabCnt = 0, whileNumCnt = 0;
int brctNumCnt = 0, expIfCnt = 0; // 记录break/continue数，需要短路求值的表达式数
string now_while_end = "", now_while_entry = "";

map<string, pair<int, int>> symbol_table; // 初始全局有一张符号表
map<string, pair<int, int>> *current_table; // 当前block的符号表
map<map<string, pair<int, int>>*,map<string, pair<int, int>>*> total_table; // 记录每个符号表的父亲

map<string, string> func_table; // 函数表,记录函数返回值类型
map<string, pair<int, int>> glob_table; // 全局变量表
```

符号表使用 map 结构，是从符号名的 string 到 pair<int, int>的映射，其中 pair 的第一个元素表示当前符号的值，第二个元素表示是否可修改。

在实现语句块和作用域部分时，对符号表进行扩充，需要层次结构的符号表。考虑到每个符号表最多只能有一个子表，因此采用比较不优雅的 map 结构记录每个符号表的父亲表。

对于符号表的计数有两个全局变量，symTabCnt 记录当前存在的符号表数量，主要用于查找时确认最大查找深度，allsymTabCnt 记录所有符号表的数量，用于为新生成的符号表分配序号，以免 Koopa IR 中出现重复定义。

函数表记录每个函数的类型。

(4) RISCV 相关结构

```
// riscv所需变量
string str1; // riscv str
map<koopa_raw_value_t, int> stackForInsts; // 记录每个变量放在sp多少的地方
map<koopa_raw_function_t, int> func2sp, func2ra;  // 记录函数所需栈的空间
koopa_raw_function_t curFunc;


void Visit(const koopa_raw_program_t &program);
void Visit(const koopa_raw_slice_t &slice);
void Visit(const koopa_raw_function_t &func);
void Visit(const koopa_raw_basic_block_t &bb);
void Visit(const koopa_raw_value_t &value);
void Visit(const koopa_raw_return_t &ret);
void Visit(const koopa_raw_integer_t &integer);
void Visit(const koopa_raw_binary_t &binary);
void Visit(const koopa_raw_store_t &rawStore);
void Visit(const koopa_raw_load_t &load);
void Visit(const koopa_raw_jump_t &jump);
void Visit(const koopa_raw_branch_t &branch);
void Visit(const koopa_raw_call_t &call);
void Visit(const koopa_raw_func_arg_ref_t &funcArgRef);
void Visit(const koopa_raw_global_alloc_t &myGlobalAlloc);
```

其中利用全局变量记录每个函数所需栈空间、每个变量存放在栈中的位置，若有递归调用则记录函数函数应返回的地址，利用 str1 字符串存放生成的 RISCV 程序。

后端实现整体为递归结构，利用 visit 的递归调用进行输出。

3. 主要算法设计

前端完全按照文档实现，在 parse 过程中建立语法树结构，在初始化语法树时维护每个节点的子节点列表、节点类型、名称及初始取值等信息。利用 dump 和 dump2str 成员函数对已构造的语法树进行输出。

后端主要根据 koopa.h 中的定义，利用 dfs，递归调用 visit 函数对函数，基本块和其中的每条指令进行处理。处理过程中跳过了对局部变量的寄存器分配，而是将其全部放入栈中，寄存器方面仅利用 a0-a7 传递参数，从而利用空间换取了时间，简化了寄存器分配的过程。

三、编译器实现

1. 工具及软件

首次使用 docker 和 gitlab，存在诸多不熟悉之处，实现过程中实际使用 VSCode 连接 docker 服务器。对于 git，由于之前 vscode 默认绑定了 GitHub 账户，git 直接上传 gitlab 始终没能成功，因此实验过程中使用 git 同步上传 GitHub，再手动从 GitHub 拉取项目到 gitlab，对于整个过程也带来了一些不便。

2. 编码细节

(1) 符号表

理想的符号表是一个单向链表结构，实现中采用 map 结构，相当于存储链表指针。初始时全局有一张符号表，用于记录全局变量。每进入一个新的基本块，则生成一张新的符号表，该符号表内的符号名称记为 a_1/b_1/c_1...以区别不同的作用域。退出当前块，则删除当前符号表。

```
int main()
{
  int a = 0, b = 1;
  return a && b;
}
```

```
    @tttmp_0 = alloc i32
    store 0, @tttmp_0
    %0 = load @a_3
    %1 = ne 0, %0
    br %1, %expthen_0, %expend_0
%expthen_0:
    %2 = load @b_3
    %3 = ne 0, %2
    store %3, @tttmp_0
    jump %expend_0
%expend_0:
    %4 = load @tttmp_0
    ret %4
```

当从符号表查找符号时，从当前表向前递归查找，直到找到最近的符号定义，返回该符号在当前表中的名称。

Map 结构虽然丑陋，但其合法性在于符号表仅记录符号名称和作用域，编译过程中不会跨层更改符号表，因此不会存在 map 索引失效的问题。

(2) 逻辑表达式的短路求值

逻辑表达式在编译成 Koopa IR 过程中会翻译为 if 语句，以实现短路求值。如左图所示。

(3) 左递归的消除

```
ConstDecl
  : CONST BType myConstDef ';' {
    auto ast = new ConstDecl();
    ast->son.push_back($2);
    ast->son.push_back($3);
    $$ = ast;
  }
  ;

myConstDef
  : myConstDef ',' ConstDef {
    $1->son.push_back($3);
```

```
  $$ = $1;
}
| ConstDef {
  auto ast = new myConstDef();
  ast->son.push_back($1);
  $$ = ast;
}
;
```

以常量定义为例，文档中给出的规约方式会产生左递归，因此改写语法规则消除左递归，并用上述方式维护每个非终结符的推导式。

(4) 移动-规约问题

Level8.3 中遇到了移动-规约问题，导致语法分析报错。分析后得出全局变量声明时，解析得到的首个元素与定义函数时的首个元素相同（即文档中的 BType 和 FuncType），因此产生左递归。为解决左递归，实现中取消了 Btype，将其全部替换为 FuncType，并维护解析得到的元素类型，来判别利用了哪一个产生式。

四、问题及展望

1. 由于时间仓促，本项目前后总共用时约一周，因此诸多细节问题考虑依然不周。

例如符号表数据结构虽然实现，但现有的 map 实现并不便于阅读，也难以进行扩展，更好的实现方式可以采用树结构单独设置结构体，利用指向父亲节点的指针记录符号表的继承信息等。

再例如 AST 中诸多同名函数及变量在实际应用过程中的含义并不一致，比如 val 在表达式 AST 中表示表达式的值，但在数组变量中表示数组长度。虽然现有代码能够实现想要的功能，但对于未来进一步优化编译器无非是提前埋下的雷。

2. 编译器默认传入的程序均正确且合法，并未对程序合法性进行判断，缺少一定的鲁棒性。

3. 本项目在实现时，一开始思路并不清晰，上手过程中按照 https://www.cnblogs.com/zhangleo/p/15963442.html 逐步搭建了环境，并完成了前两个 level 的实现；在第三个 level 的实现中参考了 https://github.com/HowlingNorthWind/Compiler 的做法，借鉴了其中的语法分析结构和符号表结构，并以此为基础沿用至整个实现过程。此外，感谢李畅同学和侯树峭同学在实现思路和过程上对我的帮助。

4. 项目前 8 个 level 的个别数据点仍有不通过的现象，对于没有测试数据的情况 debug 确实是一件比较靠运气的事，也耗费了大量的时间。因此建议在未来的文档中可以提供更多可见的测试数据，或是仅提供每个测试样例的特点，以便 debug。

此外，lv1-2 基础部分给出了较为详细的教程，而 lv3 属实是一个坎，实现过程中大量查阅了资料，也建议未来能够在难度跨越较大的章节给出更多的提示代码。