编译原理课程实践报告：一个朴素的编译器

岳禹彤 2000012952

一、编译器概述

1.1 基本功能

本编译器基本具备如下功能（都是按照官方文档的要求）：

1. 实现了简单的编译器前端，能够将SysY程序（C语言的一个子集）翻译成规定范式的koopa IR（编译原理课程实践设计的一种中间表示）

   运行指令`./compiler -koopa hello.c -o hello.koopa`即可

2. 实现了简单的编译器后端，能够在前端基础上将koopa IR生成到RISCV汇编代码。

   运行指令`./compiler -riscv hello.c -o hello.s`即可

1.2 主要特点

本编译器的主要特点是

1. 没啥大的特点（有些测试点都没通过）
2. 主要使用C++编写，使用安全的`unique_ptr`

二、编译器设计

2.1 主要模块组成

编译器由下面5个主要模块组成：

• `main.cpp`: 编译器的 main 函数, 调用 SysY -> Koopa IR 和Koopa IR -> RISC-V 的部分.

• `sysy.l`: Flex 源文件, 包含 SysY 语言的词法规则.

• `sysy.y`: Bison 源文件, 包含 SysY 语言的语法规则以及语义动作.

•`ast.hpp`: 定义抽象语法树、生成 Koopa IR 的代码 .

• `asm.hpp`: 生成 RISC-V 的代码.

2.2 主要数据结构

•本编译器最核心的数据结构是抽象语法树。在实际代码编写过程中，设计了AST类用来表示抽象语法树上的一个节点。

```
class BaseAST
{
public:
  int number = 5;//常量的nuber
  std::string op;

  virtual ~BaseAST() = default;
  virtual void Dump() const = 0;
};
```

•这里每个类都会有`number`和`op`，但不是每个类都会用到。

•每个类的`Dump()`方法, 是遍历 AST 来生成 Koopa IR.

•实际不同的节点我们用不同的派生类表示：

```
class BaseAST;
class CompUnitAST;
class FuncDefAST;
class StmtAST;
class ExpAST;
class PrimaryExp1AST;
class UnaryOpAST;
class NumberAST;
class UnaryExpAST;
class UnaryExp1AST;
...
```

•在描述语法结构的 AST 类中,一般包含指向其下一层 AST 的指针和自身具有的属性，以及某些类会有一些特殊的对象来标识此节点具体采用 EBNF 中的哪一条规则。

•对于 EBNF 中不能确定长度的结构，我使用`<vector>`结构来处理。

•如果将一个SysY程序视作一棵树，那么一个`struct CompUnit`的实例就是这棵树的根，这棵树可以表示很多种语法结构。比如`FuncDef`和 `Decl`.

•我使用了下列数据结构来表示变量or函数：

```
class ValType
{
public:
  enum class ValTypeEnum
  {
    const_type,//常量
    var_type,//变量
    void_,//无返回值函数
    int_,//有返回值函数
  } type;

  int val;//对应的值
};
```

•在前端，我用`std::unordered_map<std::string, ValType> Map[1024]`管理出现的变量和函数

•在后端：我用`std::unordered_map<koopa_raw_value_t, int> Map_`管理变量映射到栈上的位置

2.3 主要设计考虑及算法选择

2.3.1 符号表的设计考虑

前端，维护了下列数据结构来处理不同块的作用域。

```
static std::unordered_map<std::string, ValType> Map[1024];


static int BlockDepth = 0
```

ValType就是前文提到的可以判断类型，容纳类型的数据结构；

具体来说，一开始是第0层。每进入一个{ }深度+1，（当然进入if{ } 或者 else { } 或者 while{ }深度都+1）

```
    for (int i = BlockDepth; i >= 0; i--)
    {
      if (Map[i].count(ident) > 0)
      {
        const ValType &valType = Map[i][ident];
        if (valType.type == ValType::ValTypeEnum::var_type)
        {
         ...
        }
      }
    }
```

每次利用👆代码的方法来寻找我们能用到的函数or变量。但是推出一个块的时候要消除痕迹。

```
static void clear_block_status()
{
Map[BlockDepth].clear();
if_end[BlockDepth]=0;
}
```

这样子会出现，同一层可能会有很多block，为了区分变量的分配，我们采用如下变量名生成策略：用 NameMap记录名字出现的次数，每出现一次就+1，防止重复。

```
static std::unordered_map<std::string, int> NameMap[1024];//防止每一层的名字出现重复分配

static std::string name(std::string ident, int x)//根据名字和块的深度生成 名字
{
  return ident + "_" + std::to_string(x) +
std::to_string(NameMap[x][ident]);
}
```

2.3.2 寄存器分配策略

没分配（扔栈上）

我维护着一个 std::unordered_map，记录每个会有局部值的 koopa_raw_value_t 结构（直接在栈上分配 4 bytes）对应在栈上的偏移量。

2.3.3 采用的优化策略

好像没优化。

2.3.4 其它补充设计考虑

好像，也没有补充设计?

三、编译器实现

3.1 各阶段编码细节

Lv1. main函数和Lv2. 初试目标代码生成

这两部分感觉对我上手lab的帮助较大

首先是编译器的基本结构我了解了

然后是用来描述编程语言的语法EBNF

最后是可以使用 Flex（描述 EBNF 中的终结符部分，正则表达式这里就有用了） 和 Bison（描述 EBNF 本身） 来分别生成词法分析器和语法分析器。

Lv3. 表达式

我一开始用了一个比较那啥的方法。因为每个baseAST类都有一个number，于是我就

```
MulExp1
:MulExp UnaryOp2 UnaryExp {
   auto ast=new MulExp1AST();
    ast->exp1=unique_ptr<BaseAST>($1);
    ast->exp2=unique_ptr<BaseAST>($2);
    ast->exp3=unique_ptr<BaseAST>($3);
    if(ast->exp2->op=="*")
    {
      ast->number=(ast->exp1->number)*(ast->exp3->number);
    }
    else if(ast->exp2->op=="/")
    {
      if((ast->exp3->number)!=0)
      {ast->number=(ast->exp1->number)/(ast->exp3->number);
      }
    }
    else if(ast->exp2->op=="%")
    {
      ast->number=(ast->exp1->number)%(ast->exp3->number);
    }
    $$=ast;
}
;
```

在这个阶段就算出了每个数的值。后续直接用number就行。我对返回值放到了寄存器里，方便统一处理。

Lv4. 常量和变量

常量，我直接根据上面的numebr记录在符号表中。

变量，需要求值时递归地调用 DumpIR () 函数，将各子节点计算出来，放到寄存器里，然后再求出变量的值即可。

我是这样子得到单句定义很多个常量pr变量的。所有的常量和变量都会被放到我定义的vector结构体中。

```
ConstDecl:
  CONST BType ConstDef MoreConstDefs ';' {
    auto ast = new ConstDeclAST();
    ($4)->insert(($4)->begin(), unique_ptr<BaseAST>($3));
    ast->const_defs = std::move(*($4));
    $$ = ast;
  };

MoreConstDefs:
  MoreConstDefs ',' ConstDef {
    ($1)->push_back(unique_ptr<BaseAST>($3));
    $$ = $1;
  } |
  %empty {
    $$ = new vector<unique_ptr<BaseAST>>();
  };
```

Lv5. 语句块和作用域

我是怎么整作用域嵌套问题的捏？ 数据结构的设计请参考前面【符号表的设计考虑】。

举个例子：比如我们要进入一个block：

```
 else if (type == StmtAST::Stmt_type::block)
    {
```

```
      num++;
      int num_ = num;
      if (!if_end[BlockDepth])
        my_koopa << "jump %starttt" << num_ << "\n";
      my_koopa << "%starttt" << num_ << ":\n";
      BlockDepth++;
      exp->Dump();
      if (!if_end[BlockDepth])
        my_koopa << "jump %enddd" << num_ << "\n";
      my_koopa << "\n";
      my_koopa << "%enddd" << num_ << ":\n";

      clear_block_status();
      BlockDepth--;

    }
```

可以看到`BlockDepth++`后才 `exp->Dump()`，最后 `clear_block_status(); BlockDepth--;`

这里的`if_end[BlockDepth]`来判断当前的块有没有返回。有返回的话按理说不应该生成之后的了。

Lv6. if语句

bison 优先采用了 shift，

```
 Stmt: IF '(' Exp ')' Stmt .  | IF '(' Exp ')' Stmt . ELSE Stmt
```

二义性好像不用解决?

我把所有的Stmt都

```
class StmtAST : public BaseAST
{
```

```
public:
  enum class Stmt_type
  {
   ...//各种指令tpye
  } type;

  std::unique_ptr<BaseAST> exp; // exp or block
  std::unique_ptr<BaseAST> lval;
  std::unique_ptr<BaseAST> exp_stmt;
  std::unique_ptr<BaseAST> else_stmt;
  std::string str_;

  void Dump() const override
  {
    else if (type == StmtAST::Stmt_type::if0)
    {...
    }
    // IF '(' Exp ')' Stmt ELSE Stmt
    else if (type == StmtAST::Stmt_type::if_)
    {
      if_cnt++;
      int now_if = if_cnt;
      exp->Dump();
      my_koopa << "br %" << now - 1 << ", %then" << now_if << ",
%else" << now_if << "\n";
      my_koopa << "\n";

      my_koopa << "%then" << now_if << ":"
               << "\n";
      BlockDepth++;
      exp_stmt->Dump();
      if (!if_end[BlockDepth])
        my_koopa << "jump %end" << now_if << "\n";
      my_koopa << "\n";
      clear_block_status();
      BlockDepth--;

      my_koopa << "%else" << now_if << ":\n";
      BlockDepth++;
```

```
      else_stmt->Dump();
      if (!if_end[BlockDepth])
        my_koopa << "jump %end" << now_if << "\n";
      my_koopa << "\n";
      my_koopa << "%end" << now_if << ":\n";

      clear_block_status();
      BlockDepth--;
    }
  ...//还有很多

};
```

用`if_cnt`记录下当前是第几个if,然后进行循规蹈矩地生成exp，然后生成`br`指令，生成{块里的东西}出了块以后再进行`jump`指令的生成。

值得注意的是，`BlockDepth`的处理以及`if_end[BlockDepth]`的判断。

Lv7. while语句

下面是处理while的部分。

引入了while循环检查（%whilecheck）和结束（%endwhile）的标签。

Continue语句：对于continue语句，它会跳回while检查的标签，有效地继续循环。

Break语句：对于break语句，它会跳到while循环的结束位置，跳出循环。

使用wh_cnt：跟踪程序中嵌套的while循环数量

和now_wh来管理嵌套的while循环。now_wh管理当前while循环的标识符、if_end来跟踪块的结束/

```
    else if (type == StmtAST::Stmt_type::while_)
    {
```

```
      wh_cnt++;
      whf[wh_cnt] = now_wh;
      now_wh = wh_cnt;

      if (!if_end[BlockDepth])
        my_koopa << "jump %whilecheck" << now_wh << "\n";
      my_koopa << "\n";
      my_koopa << "%whilecheck" << now_wh << ":\n";

      exp->Dump();
      if (!if_end[BlockDepth])
        my_koopa << "\tbr %" << now - 1 << ", %whilethen" << now_wh
<< ", %endwhile" << now_wh << "\n";
      my_koopa << "\n";
      my_koopa << "%whilethen" << now_wh << ":"
               << "\n";

      BlockDepth++;
      exp_stmt->Dump();

      if (!if_end[BlockDepth])
        my_koopa << "\tjump %whilecheck" << now_wh << "\n";

      my_koopa << "\n";
      my_koopa << "%endwhile" << now_wh << ":"
               << "\n";

      clear_block_status();
      BlockDepth--;
      now_wh = whf[now_wh];
    }
    else if (type == StmtAST::Stmt_type::continue_)
    {
      my_koopa << "jump %whilecheck" << now_wh << "\n";
      if_end[BlockDepth] = 1;
    }
    else if (type == StmtAST::Stmt_type::break_)
    {
```

```
      my_koopa << "jump %endwhile" << now_wh << "\n";
      if_end[BlockDepth] = 1;
    }
    else if (type == StmtAST::Stmt_type::exp)
    {
      exp->Dump();
    }
```

Lv8. 函数和全局变量

还是前面说的，用一个统一的结构来管理函数和变量。

用std::vector类型的 func_params管理函数参数。

只需要在进入函数的时候把参数挨个Dump一下以便后续使用（这时会加到符号表里）

```
  void Dump() const override
  {
    my_koopa << "fun "
             << "@" << ident << "(";

    for (auto it = this->func_params.begin(); it != this-
>func_params.end(); ++it)
    {
      if (it != this->func_params.begin())
        my_koopa << ", ";
      (*it)->number = 0; // 参数初始化为0
      (*it)->Dump();
    }
    my_koopa << ")";

    if (func_type->number == 1)
    {
      Map[0][ident] = ValType(ValType::ValTypeEnum::int_, 0); // 把
函数加入符号表
```

```
      my_koopa << ": i32 ";
    }
    else
    {
      Map[0][ident] = ValType(ValType::ValTypeEnum::void_, 0);
    }

    my_koopa << "{"
             << "\n";
    my_koopa << "%entry:\n";

    for (auto it = this->func_params.begin(); it != this-
>func_params.end(); ++it)
    {
      (*it)->number = 1;
      (*it)->Dump();
    }

    BlockDepth++;
    block->Dump();


    if (!if_end[BlockDepth])
    {
      if (Map[0][ident].type == ValType::ValTypeEnum::int_)
        my_koopa << "    ret 0\n";
      else
        my_koopa << "   ret\n";
    }

    my_koopa << "}"
             << "\n";
    clear_block_status();
    BlockDepth--;
    if_end[BlockDepth] = 0;
  }
```

Lv9. 数组

> 好像没做这个。。。

3.2 工具软件介绍（若未使用特殊软件或库，则本部分可略过）

3.3 测试情况说明（如果自己进行过额外的测试，可增加此部分内容）

四、实习总结

4.1 收获和体会

做的时候还是一口气做完比较好。不然隔一段时间就忘记了得重温...

也是因为把lab拖到了期末季导致最后的数组懒得整了（？给自己的懒惰找借口

总得来说让我对编译理解更深刻了（以前确实不懂代码是怎么自己变成汇编的）

不过还是发现了自己编程中不好的习惯：没有构建好所有的思路。都是边写边debug。这样其实不太好。。

4.2 学习过程中的难点，以及对实习过程和内容的建议

一开始不熟悉一些代码和工具的时候感觉比较难☹️

后来发现其实比较简单 就是 「构建语法树」「巧妙地设计符号表Dump即可」

4.3 对老师讲解内容与方式的建议

很好捏！希望越来越好！