2024年编译原理课程实践报告

编译原理课程实践报告：SYSY

信息科学技术学院 2100012821 尹哲晖

一、编译器概述

1.1 基本功能

本编译器基本具备如下功能：

1. 将 `SysY` 语言编译到 `Koopa IR.`
2. 将 `Koopa IR` 生成到 `RISC-V` 汇编.

1.2 主要特点

本编译器的主要特点是实现较为简单，能完成较为基本的表达式编译、常量变量、语句块作用域、if, while语句、函数调用、全局变量等功能。

二、编译器设计

2.1 主要模块组成

编译器由6个主要模块组成：

1. `sysy.l`:描述词法规则并被 `Flex` 读取。
2. `sysy.y` 描述语法规则并被 `Bison` 读取
3. `AST`： 实现 `AST` 结构，进行语义分析生成 `Koopa IR`。
4. `Visit`： 遍历 `Koopa IR`，生成 `RISC-V` 代码。
5. `Symbol_table`： 附属于 `ast`，符号表可以记录作用域内所有被定义过的符号的信息
6. `main` 函数：主函数，照抄lab在线文档即可

2.2 主要数据结构

本编译器最核心的数据结构是 `Ast`。语法分析器生成的 `AST` 即为整个 `SysY` 源程序的抽象语法树.抽象语法树节点均继承自 `BaseAST` 类，参照在线文档，该类定义了两个纯虚函数，`Dump` 的作用是将 `AST` 节点转换为 `Koopa IR` 的输出格式。由于它是纯虚函数，所有继承自 `BaseAST` 的类都必须实现这个函数，具体的实现会根据不同的 `AST` 节点类型而有所不同。 `EVa` 同理，用于计算节点的值（通常为表达式节点）

```
class BaseAST
{
```

```
 public:

  virtual ~BaseAST() = default;

  virtual void Dump() const = 0;

  virtual int EVa() const = 0;//evaluate the value

};
```

在 `if...else...` 语句方面，由于涉及到二义性问题，在线文档提示道：“你只需对 `if/else` 的语法略加修改 (提示: 拆分), 就可以完全规避这个问题.”但是实际实现的时候，感觉拆分还是太麻烦了。这一部分我参考了网上南京大学的项目文档，Project_1.pdf，如图，借助 `bison` 中的符号优先级进行了处理。

显式地解决悬空else问题可以借助于算符的优先级。Bison源代码中每一条产生式后面都可以紧跟一个%prec标记，指明该产生式的优先级等同于一个终结符。下面这段代码通过定义一个比ELSE优先级更低的LOWER_THAN_ELSE算符，降低了归约相对于移入ELSE的优先级：

```
1  …
2  %nonassoc LOWER_THAN_ELSE
3  %nonassoc ELSE
4  …
5  %%
6  …
7  Stmt : IF LP Exp RP Stmt  %prec LOWER_THAN_ELSE
8    |  IF LP Exp RP Stmt ELSE Stmt
```

这里ELSE和LOWER_THAN_ELSE的结合性其实并不重要，重要的是当语法分析程序读到IF LP Exp RP时，如果它面临归约和移入ELSE这两种选择，它会根据优先级自动选择移入ELSE。通过指定优先级的办法，我们可以避免Bison在这里报告冲突。

不过原文件是规定的更低优先级，在我的代码中为了体现出不同，我是设置更高优先级，殊途同归。具体代码如下：

```
%precedence IFX
%precedence ELSE

....

  | IF '(' Exp ')' Stmt %prec IFX
  {
    auto ast = new StmtAST();
    ast->type = 5;
    ast->exp = unique_ptr<BaseAST>($3);
    ast->stmt_if = unique_ptr<BaseAST>($5);
    $$ = ast;
  }
```

```
  | IF '(' Exp ')' Stmt ELSE Stmt
  {
    auto ast = new StmtAST();
    ast->type = 6;
    ast->exp = unique_ptr<BaseAST>($3);
    ast->stmt_if = unique_ptr<BaseAST>($5);
    ast->stmt_else = unique_ptr<BaseAST>($7);
    $$ = ast;
  }
```

2.3 主要设计考虑及算法选择

2.3.1 符号表的设计考虑

代码通过使用栈来管理符号表，从而处理变量的作用域。每当进入一个新的作用域时，调用 `enter_code_block()`，在栈顶添加一个新的符号表；离开作用域时，调用 `exit_code_block()`，从栈顶移除符号表。插入符号时，调用 `insert_symbol()` 将符号添加到当前作用域的符号表中。查询符号时，调用 `query_symbol()` 从栈顶,即先在当前符号表中查询, 如果找不到就去上一层中查询.,逐层查找，直到找到符号或栈为空。

```
    // 进入新的作用域
    inline void enter_code_block()
    {
        symbol_table_stack.emplace(symbol_table_cnt++, symbol_table_t{});
    }

    // 离开当前作用域
    inline void exit_code_block()
    {
        symbol_table_stack.pop();
    }
        // 插入符号定义
    inline void insert_symbol(const string &symbol, symbol_type type, int
value)
    {
```

```
        symbol_table_stack.top().second.emplace(symbol,
make_shared<symbol_value>(symbol_value{ type, value }));

    }
    // 在符号表中寻找符号，返回其所在符号表的 标号 和其本身的 iterator

    inline optional<pair<int, symbol_table_t::iterator>> find_iter(const
string &symbol)

    {

        auto temp_stack = symbol_table_stack;

        while (!temp_stack.empty())

        {

            auto &table = temp_stack.top().second;

            auto it = table.find(symbol);

            if (it != table.end())

            {

                return make_pair(temp_stack.top().first, it);

            }

            temp_stack.pop();

        }

        // 没找到

        return nullopt;

    }
```

2.3.2 寄存器分配策略

所有局部变量都存放在栈上，不进行寄存器分配。在执行单条 Koopa IR 指令时，需要用到局部变量时再从内存加载到寄存器中。指令执行完毕后，如果该指令有返回值，则将返回值写回栈中,原因是这样实现比较简单。

三、编译器实现

3.1 各阶段编码细节

遵循北京大学编译实践课程在线文档

Lv1. main函数和Lv2. 初试目标代码生成

这部分需要构建一个词法分析和语法分析的框架，没有功能的实现，所以只要能够解析代码并且扔掉注释就好了，完全跟着在线文档走就行，文档中给出的代码已经涵盖了大部分内容，一般只需要补充注释下//...的内容即可

注意块注释的正则表达式

```
BlockComment "/*"([^\*]|(\*)*[^\*/])*(\*)*"*/"
```

Lv3. 表达式

要修改设计大量`ast`,文档开头给出的规范告诉了要添加的种类

`sysy`部分，`ast`的设计中，遵照文档中的二选一，选择对`::=`右侧的每个规则都设计一种`AST`,对于比较复杂的则再做一层嵌套, 在`parse`到对应规则时, 构造对应的`AST`.而格式均仿照开始给出的`FuncDef`实现, 即形如

```
//MulExp        ::= UnaryExp | MulExp ("*" | "/" | "%") UnaryExp;
MulExp
  : UnaryExp
  {
    auto ast = new MulExpAST();
    ast->type = 1;
    ast->unaryexp = unique_ptr<BaseAST>($1);
    $$ = ast;
  }
  | MulExp MULOP UnaryExp
  {
    auto ast = new MulExpAST();
    ast->type = 2;
    ast->mulexp = unique_ptr<BaseAST>($1);
    ast->mulop = $2;
    ast->unaryexp = unique_ptr<BaseAST>($3);
    $$ = ast;
  }
  ;
```

关于优先级，文档有言：“`SysY`的语法规范中已经体现了运算符的优先级, 如果你正确建立了`AST`, 那么你在后序遍历`AST`时, 生成的代码自然会是上述形式.”

而关于`&&`和`||`逻辑运算，因为`riscv`指令中没有逻辑与或，只有按位与或，所以要进行如下变换：

```
A&&B => (A!=0)&(B!=0)
A||B => (A!=0)|(B!=0)
```

Lv4. 常量和变量

在 `Lv4` 中引入了符号表，根据文档进行插入符号定义、确认符号定义是否存在和查询符号定义的操作。

遇到了 `Bison` 不能直接支持的形式——一个部分重复出现若干次,如 `VarDecl ::= BType VarDef {"," VarDef} ";";` 通过添加 `ItemList` 进行处理，把后面的部分记为 `List`,用 `VarDefList ::= VarDefList ',' VarDef` 进行递归调用的推导，并使用 `vector` 方便调用

常量在在定义时计算初始值并插入符号表，常量的初始值通过 `EVa` 计算。

变量先 `alloc` 一段内存，插入符号表时，值为0

使用 `RISC-V` 中的 `lw` 和 `sw` 指令来实现 `load` 和 `store`

Lv5. 语句块和作用域

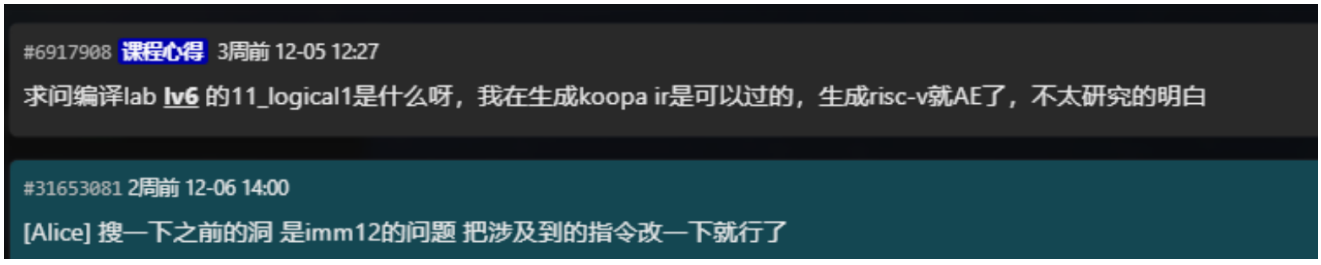
参考在线文档，"本节, 你只需对这个符号表稍加改动, 使其：支持作用域嵌套、在进入和退出代码块时更新符号表的层次结构、只在当前作用域添加符号、能够跨作用域查询符号定义。"

这部分是与 `lv4` 一起完成的，对于语句块和作用域的处理如 `2.3.1` 所述,利用栈结构实现作用域。

Lv6. if语句

二义性问题如 `2.2` 中所讲述

在 `risv` 实现上第11个点不过，树洞说是 `imm12` 的问题,但是修改后依然没有解决，不知道怎么回事，但是之后lv7，lv8的 `riscv` 又全过了，所以就没管了



对于逻辑与 (`&&`) 和逻辑或 (`||`) 的短路求值，通过条件跳转指令实现。根据左侧表达式的值，生成条件跳转指令。如果是逻辑与 (`&&`)，当左侧表达式为假时直接跳转到 `false` 分支；如果是逻辑或 (`||`)，当左侧表达式为真时直接跳转到 `true` 分支。

Lv7. while语句

在实现 `while` 语句循环嵌套时，通过使用全局变量 `whilecnt` 变量计数 `while` 语句的数量，并为每个 `while` 语句生成唯一的标号，同时使用 `whileStack` 栈管理当前嵌套的 `while` 语句的标号， `whileStack.top()` 表示当前 `while` 语句。

在每个 `while` 语句开始时，生成跳转到 `while` 入口的指令和入口标签，计算条件表达式的值并生成条件跳转指令，根据条件跳转到 `while` 主体或结束标签。在 `while` 主体中，生成主体标签和跳转回入口的指令，最后生成 `while` 结束标签。

对于 `break` 语句，跳转到 `whileStack.top()` 对应的当前 `while` 语句的结束标签。

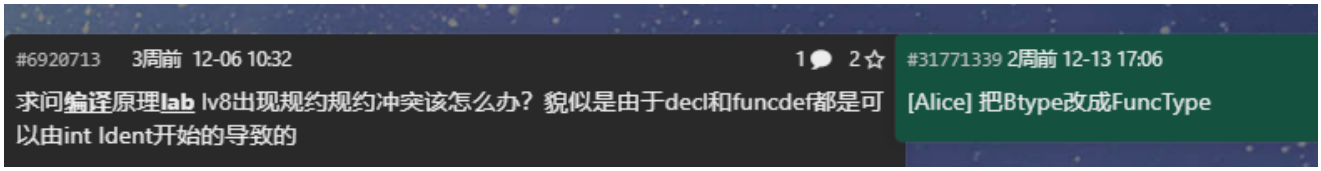
对于 `continue` 语句，跳转到 `whileStack.top()` 对应的当前 `while` 语句的入口标签。

通过这些步骤，确保能够正确处理 `while` 语句的循环嵌套问题。

Lv8. 函数和全局变量

遇到了规约/规约冲突，即无法确定将 `int` 规约为 `FuncType` 还是 `BType`。

这一块参考树洞直接把二者合并为词法分析中 `Token` 的 `string Type` 即可，再在具体调用 `Dump` 的地方进行修改



对于所有全局变量, 你的编译器应该生成全局内存分配指令 (`global alloc`). 这种指令的用法和 `alloc` 类似, 区别是全局分配必须带初始值.

`lv8` 的难点主要在于目标代码生成，即riscv部分

注意

函数的 `prologue` 和 `epilogue`

前8个参数放置在寄存器中，其他参数放置在栈中，函数开始时要把参数复制到自己的栈帧中方便处理

至于"要读寄存器前不能进行函数调用, 否则寄存器的值有一定概率被破坏"这种情况，因为没有进行寄存器分配所以不需要考虑

3.2 工具软件介绍

1. `flex/bison`： `Flex` 用来描述 `EBNF` 中的终结符部分, 也就是描述 `token` 的形式和种类. 你可以使用正则表达式来描述 `token`. `Bison` 用来描述 `EBNF` 本身, 其依赖于 `Flex` 中的终结符描述. 它会生成一个 `LALR parser`.通过编写 `sysy.l` 和 `sysy.y` 完成词法分析和语法分析
2. `libkoopa`： 使用该库的 `Koopa` 各种结构体，构建了 `Koopa` 结构，生成 `Koopa` 代码。

3.3 测试情况说明

结合在线文档给出的基础测试用例，再结合树洞之类的地方给出的一些特殊情况例如没有`return`或者多次`return`来构造，没有`return`就加`return 0`，多次的话取第一次之后的作废

四、实习总结

4.1 收获和体会

实现了一个简单的编译器，体会到了编译器开发中各个部分都需要考虑到的不易

加深理解了这门课程各个部分的内容。

4.2 学习过程中的难点，以及对实习过程和内容的建议

一开始做了一部分lab，后来搁置了，最后捡起来继续做lab还是很赶，且和课程进度结合的程度不高，建议lab和课程内容可以结合下并且逐步跟进。

4.3 对老师讲解内容与方式的建议

以后可以考虑选在早八外的时间开课（）