编译实习报告

一、编译器概述

1.1 基本功能

本编译器的主要功能就是完成编译的工作，将SysY（C语言的子集）编译到RISC-V汇编。选取的中间表示形式是Koopa。

使用如下命令完成KoopaIR和RISC-V的生成：

```
build/compiler -koopa 输入 -o 输出
```

```
build/compiler -riscv 输入 -o 输出
```

1.2 主要特点

这个lab实现的编译器使用了`Lex`和`Yacc`进行词法分析和语法分析，生成对应的C++代码，建立起抽象语法树。在抽象语法树上完成目标语言Koopa IR的输出。如果是输出到RISC-V汇编，编译器还会通过重定向流，将编译器生成的Koopa代码作为输入，进一步得到RISC-V汇编。

二、编译器设计

2.1 主要模块组成

本编译器主要包含五个部分。lexer负责词法分析，parser负责语法分析。这两块借助了工具lex/yacc完成；编译器主体部分包含了main函数，负责读取命令的输入，完成流的重定向，驱动整个编译器工作；ast和riscv分别作为两个头文件，成为生成koopa代码和RISC-V汇编的数据结构和算法载体。

2.2 主要数据结构

1. 抽象语法树的结点基类`BaseAST`以及由此衍生出的一系列派生类，作为抽象语法树上的结点，便于生成中间表示代码。
2. 以`unordered_map`为实现体的符号表，以及存储多个符号表的栈，作为不同代码块当中变量的作用域。
3. 结构体`struct entry`，作为符号表的条目。
4. 两个存储`while`循环当中生成的基本块名称的栈，用于`continue`和`break`的跳转。
5. 用`unordered_map`实现的存储函数信息（返回类型等）的表。

2.3 主要算法设计

建立起抽象语法树后，对抽象语法树进行遍历。从根节点开始，进行深度优先的访问，自顶向下完成中间表示的输出。输出RISC-V时也类似，对koopa的slice进行访问，判断类型，逐步深入。

三、编译器实现

3.1 工具软件的介绍

本次lab采取的工具有lex/yacc，以帮助词法、语法分析。另外，开发环境在由助教搭建的docker镜像当中，部署了所需的工具链，十分给力。接下来着重介绍一下Lex/Yacc。

Lex工具用于词法分析。开发者最需要关注的是正则表达式的书写。构建起一组组由正则表达式构成的词法规则后，Lex会像语法分析工具输送一个个token，作为语法分析的基础。

Yacc工具用于语法分析。根据使用人员给定的一系列语法规则，构建起一个LALR的Parser来进行语法分细的工作。碰到对应的语法，工具就会执行给定的动作，借此我们可以完成抽象语法树的构建。

Lex/Yacc工具相辅相成，不可分割。二者有着诸多协作。具体的实现难点就放在下一部分具体展开。

3.2 编码细节

在这一部分，我将介绍一下编译器重点部分的实现过程，以及过程中碰到的困难和解决方案。

词法分析

不同于以往熟悉的C/C++语言文件，Lex和Yacc的文件有着自己的一套规则。例如由%%分割开的三个部分，以及 `%option noyywrap` 等语句，是对于初学者来说比较陌生的编程语法规则。类似的细节还有非常多，比如 `%parse-param`、`%union` 等等。对于不了解的人来说甚至不知道"还能这样"。不过好在，助教已经构建了对应文件的框架，可以更好地专注于主体部分的实现。

词法分析具体的难点并不是很多。唯一值得提及的就是多行注释的正则表达式，应该是这里面最复杂的正则式。我给出的表达式是：

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

语法分析

Yacc文件（.y文件）的代码量比较多。首先要注意的是yylval的定义。yylval作为词法分析器和语法分析器之间关键的沟通桥梁，重要性不言而喻。由于token可能是整数，可能是字符串，因此选取union为其定义。为了传递字符串，不能直接将类型定为string，因为string是一个有析构函数的类，不能放在union里面。union是共享空间的一种存储结构。

然后主要值得关注的是语法规则的书写。例如对于生成式：

```
Items -> {BlockItem}
```

我实现如下所示：

```
Items
  : BlockItem{
    auto s = new ItemsAST();
    s->itemsList.push_back($1);
    $$ = s;
  }
  | Items BlockItem{
    auto ptr = dynamic_cast<ItemsAST*>($1);
    ptr->itemsList.push_back($2);
    $$ = ptr;
  }
  | {
    auto s = new ItemsAST();
```

```
    $$ = s;
  }
  ;
```

一开始我的实现不包含生成式右边为空的情况，因为没有注意到大括号{}表示的是0次到多次。后面需要Items为空的情况增加了第三部分，完善了语法分析。在这个例子里面，每一个Items有一个vector，用于存储一系列的BlockItem。同时，根据不同的情况完善一个个对象的成员变量的赋值。通过一个个生成式以及后面对应的代码完成了抽象语法树上节点的构建。由此，可以自底向上地完成这一颗树的构建工作。这一个例子可以体现出语法分析的主要内容。构建完抽象语法树之后，语法分析器的主要工作就完成了。

抽象语法树的输出

父类BaseAST当中声明了纯虚函数Dump，子类都需要实现Dump()，用于输出中间表示。同时还设立了一个虚函数valueSpread()，用于处理常量的运算，在生成语句之前就运算出常量的值。

符号表

根据要求，因为语句块可以嵌套，所以符号表也需要可以应对嵌套定义的变量。由于进入语句块和出语句块比较接近栈的后进先出的特点，所以我选择了一个栈来存储所有的符号表。至于符号表本身，我选择了unordered_map，哈希表的方式来实现。符号表里的键值对，键为变量的ident，值为结构体struct entry，专门为了符号表构建的结构体，里面存储了变量所需的信息，包括：

- 是否为常量
- 变量的名字
- 变量的值（若为常量）
- 变量的深度

```
struct entry{
    bool isConst;
    int number;
    string name;
    int depth;
    entry(){
        isConst = false;
        depth = 1;
    }
};
```

每个变量的深度，做如下定义：若为全局变量深度为1，每进入一层语句块的局部变量，深度加一。根据深度这个属性，可以做出如下判断：相同深度的同名变量只能同时存在一个。那么就可以用每个变量的id加上深度来为koopa中的临时变量命名。这样子得到的命名可以解决重名的困扰。想要知道当前代码块中变量的深度可以通过访问栈中的符号表个数来实现，是O(1)的。

另外，在实现的过程中，还遇到了同名同深度不同时变量的重复alloc问题。如果仅仅使用这个栈来解决，旧有符号表出栈后，表里的变量信息就丢失了。那么同名变量会被认为未被分配空间，会出现alloc两次的现象。这是不符合语法规范的。为了解决这个问题，我又设立了一个unordered_set，存储出现过的变量的名字和深度。通过查询这个set，可以避免重复alloc的问题。

符号表的实现过程中还遇到一个问题。由于我将这个栈放在ast.hpp当中，初始化的工作不容易进行。因为要存储全局变量，一开始就需要有符号表在栈中。所以栈需要初始化一个符号表在栈中。一开始我的实现方式是在main函数内部进行一次push，但是由于处于不同文件，使用extern关键字来声明定义。可能是c++语言的extern关键字特性，初始化并未成功，因此我在ast.hpp当中多声明了两个静态全局变量，为初始化符号表栈而服务。

```
static unordered_map<string,entry> tempSymbolTable;
static deque<unordered_map<string,entry> > deq(1,tempSymbolTable);
static stack<unordered_map<string,entry> > symbolTableStack(deq);
```

基本块

根据koopa的规范要求，每个基本块必须由br、jump、ret其中之一结尾。由此带来的多个jump相连、空基本块等现象都是不合规的。为了解决这个问题，我设立了一个布尔类型的全局变量，标识了当前处于一个基本块之中还是已经离开了基本块有效范围。如果已经离开了基本块，要生成语句就要先生成基本块标号。如果还在块中，要生成jump等语句就即将离开当前基本块。通过这样的方法补足足够的块号和jump语句，解决基本块不匹配的问题。

悬空else二义性

为了避免if-else可能带来的二义性问题，我修改了语法，借鉴了课本上解决二义性的语法，引入了两个非终结符，matched_stmt和open_stmt。通过不同的产生式，避免了二义性问题。另外，为了实现短路求值，也修改了语法，对含有&&和||的表达式做特别处理，翻译成类似于if语句。

输入和输出

在main函数当中构建编译器的输入和输出。我采取流重定向的方法，构建koopa时，将输出流重定向到输出文件，那么在输出语句时可以通过cout达到目的。构建riscv时，将输出流先重定向到字符串，然后将字符串转成const char*类型的变量，输入到处理程序中，再重定向到输出文件。

3.3 自测试情况

经常遇到样例代码不通过的问题，采取的应对方法主要是自行构造测试用例进行初步试探，根据样例的名字做推测。另外，还参考了2021春秋的测试样例以及编译id大赛的测试用例，通过相近的名字来做推断。

测试出的最大的问题是基本块不匹配的问题，解决方案在上一部分已经讨论。

四、实习总结

收获和体会