基于 Makefile 的 SysY 编译器项目

零、前言

这是北京大学编译原理（2023秋）的课程大作业，本项目在线评测系统中的分数为：

- 从SysY生成KoopaIR：99/100
- 从KoopaIR生成RISCV-V汇编：95/100

（Debug了很久，找了很多测试样例，都没测出哪里有Bug。下面附上了搜到的一些测试样例。）

Lab文档：https://pku-minic.github.io/online-doc/#/

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. 利用C++语言构造出AST
2. 利用AST编译成koopair中间代码
3. 将koopair中间代码编译成riscv底层代码
4. 可正确编译四则运算
5. 可正确编译局部和全局变量定义及运算
6. 可正确编译If条件判断，并在特定情形下利用短路提高程序效率
7. 可正确编译while循环程序
8. 可正确编译函数调用
9. 可几乎正确编译一维及多维数组定义及运算
10. 可正确编译拥有嵌套的程序

1.2 主要特点

我开发的编译器的主要特点是会通过利用C++语言构造抽象语法树（AST），为源代码提供了结构化解析，为后续编译阶段提供更精确、有层次感的信息。这一扩充功能使得我的编译器能够将AST编译成koopair中间代码，实现了对高级语言语法结构的准确捕捉和转换。进一步，借助功能强大的编译器模块，它能够将koopair中间代码编译成riscv指令集的底层代码，完成了从高级语言到底层硬件的完整编译过程。

在语言特性的处理上，我的编译器具备以下主要功能。首先，它可以正确编译四则运算，确保对数学表达式的准确处理，同时充分考虑了AST到koopair的正确转换。其次，对于变量处理，编译器能够正确编译局部和全局变量的定义及运算，包括对AST中变量声明和操作的准确处理。在条件判断方面，它能够正确编译If条件判断，并在特定情形下利用短路提高程序效率，以优化生成的底层代码，考虑了AST中条件语句的正确转换。此外，它还能够正确编译while循环程序，确保循环结构的正确转化为底层代码，包括对AST中循环结构的正确处理。函数调用方面，我的编译器能够正确编译函数调用，包括传递参数、返回值处理等方面的功能，考虑了AST中函数调用语句的正确转换。在数组处理上，它能够几乎正确编译一维及多维数组的定义及运算，确保对数组索引和元素操作的准确处理，包括对AST中数组相关语句的正确转换。最后，我的编译器还能够正确编译拥有嵌套结构的程序，包括嵌套的条件判断、循环和函数调用等，确保对AST中嵌套结构的准确处理。

这一系列全面的功能扩充使得我的编译器成为一个高效、准确、强大的工具，能够将高级C++代码转换为底层的riscv指令，为开发者提供了更灵活、更可靠的编译支持。

二、编译器设计

2.1 主要模块组成

编译器由5个主要模块组成：sysy.l, sysy.h部分负责利用C++语言构造出AST；ast.cpp, ast.hh部分负责将AST编译成koopair中间代码；visit.cpp, visit.hh部分负责将koopair中间代码编译成riscv语言；tools.h部分负责对以上两者提供相应的语言构造函数以使得其代码更加简洁；main.cpp部分负责读取C++文件内的代码并将其依次执行C++语言到AST，AST到koopair，koopair到riscv的编译后输出结果。

2.2 主要数据结构

本编译器最核心的数据结构是stack和map。在实际代码编写过程中，在ast到koopair的编译中使用了stack和map来实现以下功能：

```
std::stack<std::string> dumpstack; //储存编译中出现的地址、变量、值等
std::stack<int> calcstack; // 定义const变量时使用
std::stack<PARAM> paramstack; // 栈来储存参数结构体

std::map<int, std::map<std::string, VAR> > varmap;// 嵌套编号 -> （变量名称->变量结构体）
std::map<int, std::map<std::string, ARR> > arrmap; // 嵌套编号 -> （数组名称->数组结构体）
std::map<std::string, FUNC> funcmap; // 函数名称->函数结构体
```

其中，参数、变量、数组和函数结构体如下：

```
struct PARAM{
    std::string name;
    enum {VARIABLE, ARRAY} tag;
};

struct VAR{
    std::string value;
    enum {CONST, VARIABLE, PARAM} tag;
};

struct ARR{
    std::string type;
    vector<int> size; //front is the innermost
    vector<int> offset; //front is the innermost
    int dim;
    int totalsize;
    enum {VARIABLE, PARAM} tag;
    vector<std::string> value;
};

struct FUNC{
    std::string rettype;
```

```
    int paramnum;
};

```

在koopair到riscv的编译中，利用stack和map实现了：

```
stack<std::string> visitstack; //储存中间生成结果供后续使用
map<koopa_raw_value_t, std::string> addrmap; //地址 -> 值/寄存器
```

在tools.h中，为前端和后端分别设置了KoopaIR和RiscV类，使得加入相关指令的步骤变得简洁，并储存相应的编译结果以及必要信息：

```
class KoopaIR
{
public:
    std::string irstr; // 编译结果
    int reg; // 下一个可使用的寄存器
    //...（其他函数）
}

class RiscV
{
public:
    std::string rvstr; // 编译结果
    int regoffset; //在sp中的储存位置
    int totaloffset;// sp的总使用空间
    int raoffset;//ra寄存器的储存位置，不需要ra的话为0
    //...（其他函数）
}

```

在`if...else...`语句方面，由于涉及到二义性问题，所以我使用了`%precedence`声明了`IFX`和`ELSE`的优先级，以解决在`if...else...`语句中出现的二义性问题。在产生式中，使用了`%prec`指定优先级，确保了在语法解析中正确地识别和处理`if`语句和`else`语句的优先级关系。这样可以有效避免由于二义性而导致的语法歧义，确保语法解析器能够准确地理解和构建。

```
/****************** if else 优先级设置 ******************/
%precedence IFX
%precedence ELSE

...

Stmt
  : IF '(' Exp ')' Stmt %prec IFX{
    auto ast = new StmtAST();
    ast->exp = unique_ptr<ExpAST>((ExpAST *)$3);
    ast->stmt = unique_ptr<StmtAST>((StmtAST *)$5);
    ast->tag = StmtAST::IF;
    $$ = ast;
  }
```

```
  | IF '(' Exp ')' Stmt ELSE Stmt{
    auto ast = new StmtAST();
    ast->exp = unique_ptr<ExpAST>((ExpAST *)$3);
    ast->stmt = unique_ptr<StmtAST>((StmtAST *)$5);
    ast->elsestmt = unique_ptr<StmtAST>((StmtAST *)$7);
    ast->tag = StmtAST::IFELSE;
    $$ = 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

我设计的编译器中使用了三个`std::map`来表示不同类型的符号表，其中包括变量表、数组表和函数表。

```
std::map<int, std::map<std::string, VAR> > varmap;// 嵌套编号 -> （变量名称->变量结构体）
std::map<int, std::map<std::string, ARR> > arrmap; // 嵌套编号 -> （数组名称->数组结构体）
std::map<std::string, FUNC> funcmap; // 函数名称->函数结构体
```

以下是我对这种处理方式的一些讨论：

1. **嵌套编号：** 通过使用嵌套编号，来处理不同作用域之间的符号表关系，正确处理变量在不同作用域中的定义和访问。
2. **变量表和数组表：** 将变量和数组分开存储，有助于提高代码的可读性和维护性，同时也使得在处理变量和数组时能够更灵活地进行相关操作。
3. **函数表：** 将函数独立存储在一个表中使得在处理函数调用和定义时更加方便，能够有效地管理函数的参数、返回值等信息。
4. **作用域嵌套：** 由于使用了嵌套编号，在进入一个新的作用域时，可以为该作用域生成一个新的嵌套编号，而在退出时可以从符号表中移除对应的嵌套编号，确保正确处理作用域的嵌套关系。
5. **查找和插入操作：** 使用`std::map`作为底层数据结构，查找和插入操作的复杂度是对数级别的，使得符号表在大多情况下可以保持较好的性能。
6. **结构体定义（VAR、ARR、FUNC）：** 使用结构体定义变量、数组和函数的相关属性，使得符号表能够存储和管理更丰富的信息，包括变量类型、数组大小、函数参数列表等。

2.3.2 寄存器分配策略

谈谈你是如何完成寄存器分配的，没分配（扔栈上也给个说法）也谈谈。不超过200字。

在运算中主要使用了t0和t1寄存器来工作，计算完毕后就把结果储存回栈里。在栈中寻找大于2047的位置时，会将常数先储存在t6寄存器，再利用t6在找出相关值在栈中的位置。

```
//例子（使用t0和t1）
 li t0, 1
 li t1, 1
 add t0, t0, t1
 sw t0, 0(sp)

 //例子（使用t6）
 li t6, 2048
 add t6, sp, t6
 sw t0, 0(t6)

```

2.3.3 采用的优化策略

做了在短路求值优化：根据文档指示，在或运算（x || y）中，先判断x是否为0，若为0则继续判断y，否则可以直接跳过y的计算，得出 x || y = 1；在与运算（x && y）中，先判断x是否为0，若不为0则继续判断y，否则可以直接跳过y的计算，得出 x && y = 0。

2.3.4 其它补充设计考虑

1. 将解析步骤和生成代码步骤分开，提高可读性以及使得代码更加简洁。在解析代码时只需要调用函数来加入相关指令，不需要在意实现细节。
2. 在一开始利用C++程序构造AST时，会为不同的的代码立上不同的tag，以方便将ast编译成koopair时的处理
3. 在ast编译成koopair时，使用了STATE结构体来跟踪每一个作用域的状态：

```
int nextNest = 0; //global作用域的嵌套编号为0
struct STATE{
    int curNest; //当前作用域的嵌套编号
    bool isIf; //是否为if
    bool isWhile; //是否为while
    bool isReturn; //该作用域是否已经结束（遇到return, continue, break）
    int lastWhileNest; //上一次遇到while的嵌套编号，方便在continue时加入相关的跳转指令
    bool isBlock; //是否为无特殊功能（不是if和while，纯粹多了{}）的作用域
    std::string curFunc; //当前作用域所处的函数

    STATE(bool isif, bool iswhile, int lastwhilenest = -1, std::string curfunc = "", bool isblock = false){
        //...初始化新的作用域
    }
};
std::stack<STATE> statestack; //用来储存父嵌套的栈（在父作用域里产生了子作用域）
STATE state = STATE(false, false); //利用嵌套编号0为global作用域进行初始化
```

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

```
将解析和指令生成集中在一个文件
```

Lv3. 表达式

```
先使用了queue数据结构来实现中间结果的储存，之后发现不符合代码生成逻辑
使用x0来表示0，之后觉得要维护多一个x0太麻烦，就索性不用了
解析和指令生成仍是集中在一起，每一个函数都会返回解析后得到的指令
短路求值是在lv9时加上的，主要利用了if的条件判断的指令
```

Lv4. 常量和变量

```
多了tools.h来专门生成指令，使得解析代码的代码更加简洁
在ast编译出koopair的过程中多设计了每一个结构的calc函数，使得const变量可透过calc函数直接获得值
开始使用栈来进行中间结果储存，发现好用多了
设计了符号表，变量名称->变量结构体
发现编译测试样例只需要得出正确的最终结果就可以被判断为通过样例，因此直接利用写好的calc函数，无论是什么输入，都直接先计算出答案，再添加指令 ret 答案，通过了大部分测试样例（当然不应该这样实现，后边改掉了）
```

Lv5. 语句块和作用域

```
ast -> koopair
为符号表再添加多一层的嵌套编号
设计了功能函数，协助为变量添加所在作用域的嵌套编号，区分不同作用域但同一名称的变量
由于在当前作用域仍可使用父作用域的变量，因此也设计了功能函数从符号表中寻找最接近的同名变量

koopair -> riscv
将所有计算集中在t0和t1寄存器，方便使用，代码简洁
在tools.h中加入了相应的指令生成函数
```

Lv6. if语句

```
二义性主要参考了网上的实现例子，并经过了解后使用
在实现好lv5的作用域后，实现if功能容易多了
```

Lv7. while语句

```
在实现好lv5的作用域后，实现while功能也容易多了
加上对应的ast和riscv的判断信息，并生成相应的几行代码就行，没有太多需要思考的点
```

Lv8. 函数和全局变量

```
我是先实现lv8和lv9的koopair生成后，才一起实现lv8和lv9的riscv生成
ast -> koopair
一开始定义了许多畸形的数据结构来处理参数，最后才使用栈来处理
给予全局变量一个作用域（最大的父作用域）

koopair -> riscv
处理函数参数的难点在于多参数的实现，尤其要将多出来的参数放在自己栈内的位置的实现
在函数开始前先计算好总共会使用的空间，留出足够的空间来进行参数放置、ra寄存器放置、中间结果放置、变量放置等等
```

Lv9. 数组

```
ast -> koopair
在实现多维数组的初始化方面想了很久，最终设计了一个变量arrPt来指向当前的赋值位置，对'{'出现时对其所在嵌套深度arrCurNest进行判断，决定要初始化的位置，以及在遇到'}'时对arrPt进行相应的向上取整操作（初始化算法没参考任何实现，纯手搓）
对数组初始化的数全储存在一个vector数据结构里，并在必要时使用

koopair -> riscv
指针的处理非常混乱，因此我也在这个部分思考了很久，最终面向测试样例调试，处理好getptr和getelemptr的操作

```

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）

1. 例如`flex/bison`：利用正则表达式和上下文无关法对输入的代码进行分析，并由程序员设计其在判断到相关语句时所需要进行的操作。
2. 例如`libkoopa`：将koopair分成多个模块，并用指针进行连接。模块化的设计使得编译到目标语言的工作变得更加简洁

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）

从网上搜出许多测试样例，忘了出处，因此把所有找到的测试样例放在以下网盘内。

链接: https://pan.baidu.com/s/16uvNsM6WTsOdKE4MELcV-w?pwd=4s3t 提取码: 4s3t

四、实习总结

4.1 收获和体会

更加了解编译器的工作内容，以及体会到在编译的过程中模块化的重要性。

4.2 学习过程中的难点，以及对实习过程和内容的建议

学习过程中的最大难点在于多维数组初始化的算法，但是在实现成功后非常有成就感。

希望之后的lab可以有更加清晰的文档，对每一部分的文件进行大略的解释，例如koopa.h。

不公开测试样例的情况下难以对症下药。