编译原理实习报告

王焱 1900013022

一、编译器概述

1、基本功能

将 SysY 语言编译到 Koopa IR，将 Koopa IR 生成到 RISC-V 汇编

2、主要特点

使用c/c++开发

二、编译器设计

1、主要模块组成

- sysy.l:词法分析器
- sysy.y:语法分析器
- ast.h: AST树相关的数据结构，以及astDump.cpp相关的函数和数据的定义
- astDump.cpp:与AST类成员函数Dump相关的辅助函数
- riscv.cpp:将koopaIR转换riscv代码具体实现
- riscv.h:riscv.cpp相关函数的定义和相关库的引用
- main.cpp:调用相应的函数实现sys->koopa IR->risc-v

2、主要数据结构

语法分析树ast

```
class Baseast
{
public:
  virtual ~Baseast() = default;
  static int Count_Order; //IR中间变量编码
  int kind;                 //选择的产生式序号
  virtual string Dump() = 0;
  virtual int Calc() = 0;
};
```

符号表:

```
struct Symbol //每个符号相关的定义
{
  int kind;       // 0为局部const常量，1为局部符号变量， 2为全局const， 3为全局变量
  int val;        // 没有初始化时，默认为0，数组用val记录维数
  string type;    //变量类型 非数组 i32 数组 [[i32,len_max],len_max-1]
  int block_num; //记录符号所属于的block的序号，用于IR的变量的命名
};
map<string, Symbol> symbolmap; //代表当前block符号表,便于操作
map<string, Symbol> glo_symbolmap;//全局变量单独的符号表
struct Fun_sym //每个函数符号表
{
  vector<map<string, Symbol>> vec_symbolmap; //函数内每个块的符号表
  int block_num;                             //记录当前block的序号
  int block_cnt;                             // 记录所有block的块的个数，用于命名防止名称重复
  int type;                                  //函数类型1为void  2为int
  Fun_sym()
  {
    block_num = 0;
    block_cnt = 0;
    type = 0;
  }
  void clear()
  {
    block_num = 0;
    block_cnt = 0;
    type = 0;
    vec_symbolmap.clear();
  }
};
Fun_sym fun_symtab;
```

- `Fun_sym fun_symtab;` 记录目前函数内的对当前语块域有用的语块域的符号表，不包括当前语块域
- `map<string, Symbol> symbolmap;` 记录当前语块域已定义或者声明的符号

3、主要的算法设计考虑

符号表的作用域
数组的初始化

三、编译器实现

1、所涉工具软件

EBNF

EBNF即 Extended Backus–Naur Form, 扩展巴科斯范式, 可以用来描述编程语言的语法。使用 SysY 的 EBNF, 我们可以从开始符号出发, 推导出任何一个符合 SysY 语法定义的 SysY 程序。EBNF记法中，使用大写蛇形命名法的符号或者被双引号引起的字符串是终结符。
EBNF特别的记法：

- A | B 表示可以推导出 A, 或者 B.
- [...] 表示方括号内包含的项可被重复 0 次或 1 次.
- {...} 表示花括号内包含的项可被重复 0 次或多次.

flex/bison

本实践使用C/C++语言实现编译器，使用flex/bsion分别来生成词法分析器和语法分析器。

- flex用来描述EBNF中的终结符部分，即描述token的种类还有形式，可以使用正则表达式描述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的语法分析器。

src中的.l文件会描述词法规则由flex读取，.y文件会描述语法规则由bsion读取。

koopa IR

Koopa IR 为北大编译原理课程实践设计的教学使用的一种中间表示(IR)。Koopa IR据有关对应的框架C/C++框架koopa，通过调用框架的接口，可以实现koopa IR的生成/解析/转换。

Koopa IR 是一种强类型的 IR， IR 中的所有的量 (Value) 和函数 (Function) 都具备类型 (Type)，不易出现混淆。

Koopa IR 中, 基本块 (basic block) 必须是显式定义的。 即, 在描述函数内的指令时, 你必须把指令按照基本块分组, 每个基本块结尾的指令只能是分支/跳转/函数返回指令之一，方便程序的优化。

Koopa IR 还是一种 SSA 形式的 IR，时兼容非 SSA 形式和 SSA 形式，可以开展更多复杂且有效的编译优化。

2、各个阶段的编码细节

(1)、lv1和lv2

该阶段主要是构建一个词法分析和语法分析的框架，并没有特定的功能实现。

kooap IR

sysy.l文件描述EBNF中的终结符，把字节流转化为token流。本阶段实现的行为为:

- 空白符和注释就跳过。
- 关键字返回对应的token
- 标识符把对应的字符串作为token返回
- 整数字面量将字符串转为整数作为token返回
- 单个字符直接把该字符作为token返回

具体实现方式说明文档已经给出示例，只需要对补充多行注释的正则表达式,以 `/*` 开头,中间不能出现 `*/` ,即如果出现 `*` ,下个字符不能为 `/` :

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

sysy.y文件描述语法定义，通过bsion构建语法分析器得到AST。需要更具SysY的EBNF的推导规则构建对应的语义动作，语义动作都放在推导规则的最后，在整个推导规则规约时才开始执行。在此过程构建出类似语法分析树的AST。每个非终结符号必须对应AST树中的一个节点，终结符可以更具实际情况选择是否构建一个相应的节点。

AST的设计，本实习将AST的定义单独存放在ast.h文件中。所有的AST节点都为Baseast基类的派生类，节点本身信息以及想关联的信息都定义在派生类。这样的构建可以使用C++的多态功能，用基类指针统一管理所有AST节点，调用该节点需要实现的函数。增强了程序可拓展性，减少了增加新的类型AST节点的工作量。

各节点的在构建koopa IR时所需要输出的信息都定义在类的成员函数Dump中，由于koopa IR的本质是一些字符串，不同节点之间需要传送的信息以字符串为主要形式，而且字符串能够兼容很多类型的变量，所有返回值我定义为字符串。string是自动管理内存的C++字符串，而且连接转化之类操作比较简洁，所以采用了string。Dump函数难免会出现一些同质化的操作，为了代码的简洁性可读性，设计了astDump.cpp专门来定义一些相关的处理函数。

Risc-v

koopa IR到riscv，对文本形式的koopa IR的解析可以调用koopa的库函数解析IR得到 `koopa_raw_program_t raw` 。

```
void AnalyzeIR(const char *str)
{
  // 解析IR文本str，得到 Koopa IR 程序
  koopa_program_t program;
  koopa_error_code_t ret = koopa_parse_from_string(str, &program);
  assert(ret == KOOPA_EC_SUCCESS); // 确保解析时没有出错
  // 创建一个 raw program builder，用来构建 raw program
  koopa_raw_program_builder_t builder = koopa_new_raw_program_builder();
  // 将 Koopa IR 程序转换为 raw program
  koopa_raw_program_t raw = koopa_build_raw_program(builder, program);
  // 释放 Koopa IR 程序占用的内存
  koopa_delete_program(program);

  // 处理 raw program
  // ...
  reg_init();
  Visit(raw);
  //  处理完成，释放 raw program builder 占用的内存
  //  注意，raw program 中所有的指针指向的内存均为 raw program builder 的内存
  //  所以不要在 raw program 处理完毕之前释放 builder
  koopa_delete_raw_program_builder(builder);
}
```

`raw` 本质为包含所有IR所有信息的树，根节点下面是全局变量、函数，函数的子节点为基本块，基本块的子节点为指令，指令的子节点为参与指令的量。遍历 `raw` 的节点，只需要在基本块、指令、指令的子节点层面对riscv构建。

（2）、lv3表达式

koopa IR

sysy.l文件添加新的EBNF的关键字 `+ - ! * / < > <= >= == != || &&` ,由于lv1已经设计了匹配任意单字符的规则，所以只添加双字符的关键字。

sysy.y首先添加sysy.l新定义的关键字的token，根据新的EBNF对语法规则进行修改。本阶段的语法规则出现了 `|` ，为了分辨对语法分析器选择的推导，对Baseast添加了kind的成员变量，方便Dump函数对不同推导做不同处理。

文档提供的语法规则的计算不存在歧义的，父节点的Dump函数只需要从左到右处理子节点。IR的中间变量不能重复，在Baseast添加了静态成员变量 `Count_Order` 记录中间变量的个数，并用于IR的编码。

Risc-v

在指令的子节点层面即 `FunValueVisit` 函数处理字面量，为其分配个寄存器。在指令层面处理表达式所翻译的IR的二元运算

```
void Visit(const koopa_raw_binary_t &exp)
{
  string lhs_reg = FunValueVisit(exp.lhs);
  string rhs_reg = FunValueVisit(exp.rhs);
  string reg = lhs_reg;
  switch (exp.op)
  {
  case KOOPA_RBO_NOT_EQ:
    // fprintf(ASM," xor %s, %s ,%s")
    fprintf(ASM, "  xor %s, %s, %s\n", reg.c_str(), lhs_reg.c_str(), rhs_reg.c_str());
    fprintf(ASM, "  snez %s, %s\n", reg.c_str(), reg.c_str());
    break;
    /*中间省略其他运算的具体代码*/
  default:
    break;
  }
  Sw_stack(reg, fun_size);
  Release_reg(lhs_reg);
  Release_reg(rhs_reg);
}
```

（3）、lv4常量变量和lv5语块域作用域

koopa IR

sysy.s需要根据新的EBNF对语法规则进行修改。本阶段语法出现了 `[...]`，`{...}` ,bsion的语法规则不能直接支持，对EBNF做出修改。
如 `Stmt::=[Exp] ";";` 改为 `Stmt::=Exp ';'| ";";`
如

```
Block::= "{" {BlockItem} "}";
BlockItem::= Decl;
```

改为

```
Block::= "{" "}" | "{" BlockItem "}";
BlockItem::= Decl | Decl BlockItem;
```

本阶段会用到符号表,AST节点在Dump中对符号表做处理。

- `Fun_sym fun_symtab;` 记录目前函数内的对当前语块域有用的语块域的符号表，不包括当前语块域

- `map<string, Symbol> symbolmap;` 记录当前语块域已定义或者声明的符号
- 每遇到一个新的语块域，将 `symbolmap` 的内容加入栈 `fun_symtab.vec_symbolmap` ,并清空 `symbolmap` 。
- 语块域结束时，将栈 `fun_symtab.vec_symbolmap` 的栈顶元素赋值给 `symbolmap` 并弹出。

常量和变量的处理:

- 为处理const常量，Baseast基类添加了虚函数 `virtual int Calc()=0;` ,用于计算const常量的值
- 在声明或者定义中遇到量 `ident` ,生成一个相应的 `Symbol` ， 加入 `symbolmap` ,如果为变量生成IR的 `alloc` 指令
- 在表达式遇到量 `ident` ,查询符号表 `symbolmap` ,从栈顶依次查询 `fun_symtab` 的符号表，返回第一个查询的到的符号表。常量直接用值替换。如果为赋值号右边的变量生成IR的 `load` 指令，赋值号左边的变量生成IR的 `store` 指令

Risc-v

考虑到寄存器个数的问题，引入栈，为每个IR指令分配对应的
栈 `map<koopa_raw_value_t, int> value_map;`
计算函数所需要分配栈的大小，这里考虑后面会出现的数组:

- 对于存在返回值的IR指令非alloc指令栈加4
- 对于alloc指令若为数组计算其 `size` ， 栈空间增加 `4*size` ,非数组加4

对riscv指令的影响:

- IR指令 = 右边的所有的非字面量都从栈中先用lw指令读取到一个临时寄存器reg
- IR的非 `alloc` 指令 = 左边的量都必须用sw指令写入相应栈的位置
- `store value,dest` 指令，若value非字面值需要使用lw指令从value的栈读取值到临时寄存器，dest需要使用sw指令将value的值写入dest所在的栈

（4）、lv6if语句和lv7while语句

koopa IR

sysy.y文件需要根据EBNF修改。语法规范中的if-else规
则 `Stmt ::= "if" "(" Exp ")" Stmt ["else" Stmt]` 具有二义性，会产生经典的else悬空问题。改写文法把 `Stmt` 分为完全匹配和不完全匹配两类

```
Stmt ::= Exma | UExma;
UExma ::=IF '(' Exp ')' Stmt| IF '(' Exp ')' Exma ELSE UExma;
Exma ::=IF '(' Exp ')' Exma ELSE Exma ;
```

本阶段if-else和while语句生成的IR需要划分基本块。

- if-else语句划分为if为真执行的基本块%then，为假执行的基本块%else(若无else语句省略此部分),if语句执行完执行的基本块%if_end。
- while语句划分为循环入口%while_entry，循环内语句基本块%while_body,循环结束%while_end。
- break语句，直接跳转到%while_end，之后基本块划分为%while_body_break。
- continue语句，直接跳转到%while_entry,之后基本块划分为%while_body_continue。
- return语句，为了方便处理return语句问题，把函数基本块划分为%entry，%begin，%end。如果存在返回值，在entry申请一个函数返回值类型的%ret，用于记录返回值，并且直接跳转到%end。

一些难点处理:

- 避免基本块命名重复引入相应指令的计数量和原名称组合起来

```
int IF_cnt = 0;
int While_cnt = 0;
int Break_cnt = 0;
int Continue_cnt = 0;
```

- 由于while是可以的嵌套，break/continue需要本层的while语句序号，设计一个栈来存储嵌套的while语句的序号 `vector<int> vec_while;` ,栈顶即为需要的序号。
- 短路求值，把逻辑表达式拆分成if语句，在IR中申请一个int型变量%result（%开头避免和变量命重复），用来存储逻辑表达式最终的结果，同时引入result_cnt处理出现多个短路求值的变量的命名

Risc-v

对于IR中的br指令和jump指令，可以使用risc-v的bnez指令和j指令处理，标记的名称可以利用IR基本块的名字。

（5）、lv8函数和全局变量

kooap IR

sysy.y 语法规范中 `CompUnit::=Decl | FuncDef` 会出现规约-规约冲突，改写语法规则

```
CompUnit::=Unit|CompUnit Unit;
Unit ::=FuncDef|VarDecl|ConstDecl;
```

函数的参数要加入函数的符号表。为了方便目标代码的生成，为参数 `@x` 在%entry基本块中再申请一块 `%x` 的内存空间。为了确定函数是否需要中间变量来存储返回值，增加一个函数种类表 `map<string, int> all_fun_symtab;` 。

全局变量不属于函数需要单独设计一个符号表 `map<string, Symbol> glo_symbolmap;` ，查询符号表时，最后查询 `glo_symbolmap` 。

Risc-v

可以用risc-v伪指令call和ret实现IR中函数的调用和返回。
函数的参数的接收，传递，返回值:

- 选用寄存器 `x0~7` 相应的参数
- 如果函数参数大于8，从第九个参数开始依次存放在栈sp+0，sp+4...
- 调用call指令
- 被调用函数，使用到参数时，从寄存器 `x0~7` 和栈中取值

```
if (kind.data.func_arg_ref.index < 8)
    fprintf(ASM, "  mv %s,a%zu\n", reg.c_str(), kind.data.func_arg_ref.index);
else
    fprintf(ASM, "  lw %s,%lu(sp)\n", reg.c_str(), (kind.data.func_arg_ref.index - 8) * 4 + ret_
```

- 返回值储存在寄存器a0中
- 返回地址在寄存器ra中

增加栈空间，如果出现call指令，需要为寄存器ra分配栈空间即加4。找到参数最多的call指令，从第九个参数开始为每个参数分配栈。

（6）、lv9数组

koopa IR

sysy.s同之前一样对语法规范处理，转化为bsion支持的语法规范。

数组的初始化:

```
//局部数组和全局数组类似
vector<int> vec_array_constexp; //储存数组各个维数len
vector<int> vec_block_len;      //各层{}初始化单位
vector<string> vec_initval;     //补全的初始化列表
int exp_cnt = 0;                //当前{}的表达式个数

void Count_block_len()
{
  if (vec_block_len.size() == 0) //初始化
  {
    int temp = 1;
    for (auto i : vec_array_constexp)
    {
      temp *= i;
    }
    vec_block_len.push_back(temp);
    return;
  }
  if (exp_cnt != 0)  //从数组最低维大小为单位
  {
    int temp = 1;
    for (int i = vec_array_constexp.size() - 1; i >= 0; i--)
    {
      exp_cnt /= vec_array_constexp[i];
      if (exp_cnt)
      {
        temp *= vec_array_constexp[i];
      }
      else
        break;
    }
    vec_block_len.push_back(temp);
  }
  else if (exp_cnt == 0) //没有表达式，直接出现{}，从上一层{}单位减小
  {
    int temp = vec_block_len.back();
    int block_len = 1;
    for (int i = vec_array_constexp.size() - 1; i >= 0; i--)
    {
      temp /= vec_array_constexp[i];
      if (temp == 1)
      {
        break;
      }
      else
      {
        block_len *= vec_array_constexp[i];
      }
    }
    vec_block_len.push_back(block_len);
```

```
  }
  exp_cnt = 0;
}
// ConstInitVal :ConstExp|'{' '}' |'{'  ConstArrayVal '}';
class ConstInitValast : public Baseast
{
public:
  unique_ptr<Baseast> constexp;
  unique_ptr<Baseast> constarrayval;
  string Dump() override
  {
    string temp;

    if (kind == 1)
    {
      exp_cnt++;
      vec_initval.push_back(to_string(constexp->Calc()));
    }
    else if (kind == 2)
    {
      Count_block_len();
      int size = vec_block_len.back();
      for (int i = 0; i < size; i++)
      {
        vec_initval.push_back("0");
      }
      vec_block_len.pop_back();
      exp_cnt = 0;
    }
    else if (kind == 3)
    {
      Count_block_len();
      int vec_add = vec_initval.size();
      temp = constarrayval->Dump();
      vec_add = vec_initval.size() - vec_add;

      int size = vec_block_len.back();
      for (; vec_add < size; vec_add++)
      {
        vec_initval.push_back("0");
      }
      vec_block_len.pop_back();
      exp_cnt = 0;
    }
    return temp;
  }
  int Calc() override
  {
    return constexp->Calc();
  }
};
```

```
// ConstArrayVal: ConstInitVal | ConstInitval ',' ConstArrayVal;
class ConstArrayValast : public Baseast //避免最外层括号的判断直接从constarrayval开始递归
{
public:
  unique_ptr<Baseast> constinitval;
  unique_ptr<Baseast> constarrayval;
  // int depth;
  string Dump() override
  {
    string temp;
    if (kind == 1)
    {
      constinitval->Dump();
    }
    else if (kind == 2)
    {
      constinitval->Dump();
      constarrayval->Dump();
    }
    return temp;
  }
  int Calc() override
  {
    return 0;
  }
};
```

数组和指针:

- 数组相关的变量是一类是指向数组的指针，一类是数组，两种变量都要插入符号表，数组的维数信息都记录在type。
- 数组的调用，数组的运算是逐层解析指针最终得到对应地址的过程，getelemptr指令获得下一层的数组的地址，直到得到对应元素的地址，使用相应的load或者store指令来操作。
- 数组作为函数参数传送的时实际类型是指针而非数组，先load得到指向的元素，需要使用getptr得到相邻的元素。
- 由于函数参数的影响，每次访问数组时候，我们需要先判断是数组还是指向原数组下一层数组的指针，如果是指针需要使用getptr指令访问第一层，如果是数组使用getelemptr指令。
- 数组在作为函数参数时，数组转变为指向其第一个元素的指针。

Risc-v

alloc数组的需要分配的内存大小， `Calc_arry_size` 函数返回值*4

```
int Calc_array_size(const koopa_raw_type_kind * base)
{
  auto temp = base->data.array.base;
  int sum=base->data.array.len;
  while (temp->tag == KOOPA_RTT_ARRAY)
  {
    sum *= int(temp->data.array.len);
    temp = temp->data.array.base;
  }
  return sum;
}
```

计算getelemptr指令和getptr指令的偏移量:

- 查看数组的维数
- 一维数组偏移量为 `index*4`
- 多维数组，计算下一层数组的大小size,偏移量为 `size*index*4`
  全局变量初始化:
- zero，直接计算出量的大小*4
- 非数组，直接用变量值初始化
- 数组，逐层递归

```
void GlobalVisit(const koopa_raw_value_t &value)
{
    const auto &kind = value->kind;//数组初始化列表 array
    for (size_t i = 0; i < kind.data.aggregate.elems.len; i++)
    {
      koopa_raw_value_t temp = (koopa_raw_value_t)kind.data.aggregate.elems.buffer[i];
      if (temp->kind.tag == KOOPA_RVT_INTEGER)
        fprintf(ASM, "  .word %d\n", temp->kind.data.integer.value);
      else if (temp->kind.tag == KOOPA_RVT_AGGREGATE)
        GlobalVisit(temp);
    }
}
```

3、自测试情况说明

使用了编译系统设计赛官方测试用例。

- break和continue语句对基本块分的时候命名重复了，if语句短路求值时，对基本块划分的时候命名重复。增加了计数变量来组合命名。
- return语句的处理考虑不完善。重新设计了return语句的逻辑，在函数开始阶段就把基本块划分为%entry,%begin,%end,专门申请一个%ret变量用于记录可能返回值，出现return直接跳转到%end

进行返回处理。
- 在查询符号表时候先查询了全局变量常量的符号表。改为最后查询全局的符号表。
- 部分样例的AST爆bsion内存。多申请了内存, `#define YYMAXDEPTH 100000` 。

四、实习总结

1、收获与体会

1）主要的收获

- 学会了使用gdb对程序debug
- 学会了使用git管理代码版本
- 学会了对一大段代码切分成几个函数，提高代码的可读性
- 了解了lex/bsion生成的词法分析器，语法分析器
- 对编译课程的理论部分的词法分析，语法分析理解更深了
- 明白数组是如何初始化的

2）学习过程的难点是什么

代码量过大，很难掌握全局的逻辑，修改时难以达到期望的效果，debug时修改一部分往往不如重写一大段。

2、对课程的建议

1）实习过程优化的建议

建议把数组的内容提前一些，这部分工作量大，需要对前面大部分产生式进行修改，而中间的if语句，while语句，语块域作用工作量都很小。可以把数组放到到变量和常量之后。

2）实习内容的建议

koopa IR的c++框架koopa，函数，基本块，指令都是一个层次的量，但它们在逻辑上难以同时处理，可以把层次划分的更明显一些。