编译原理实践 报告

1. 编译器概述

a) 基本功能

在编译原理`lab`中, 我实现了从`SysY`语言到`Koopa-IR`的前端编译器, 以及从`Koopa-IR`到`Risc-V`的后端编译器.

b) 主要特点

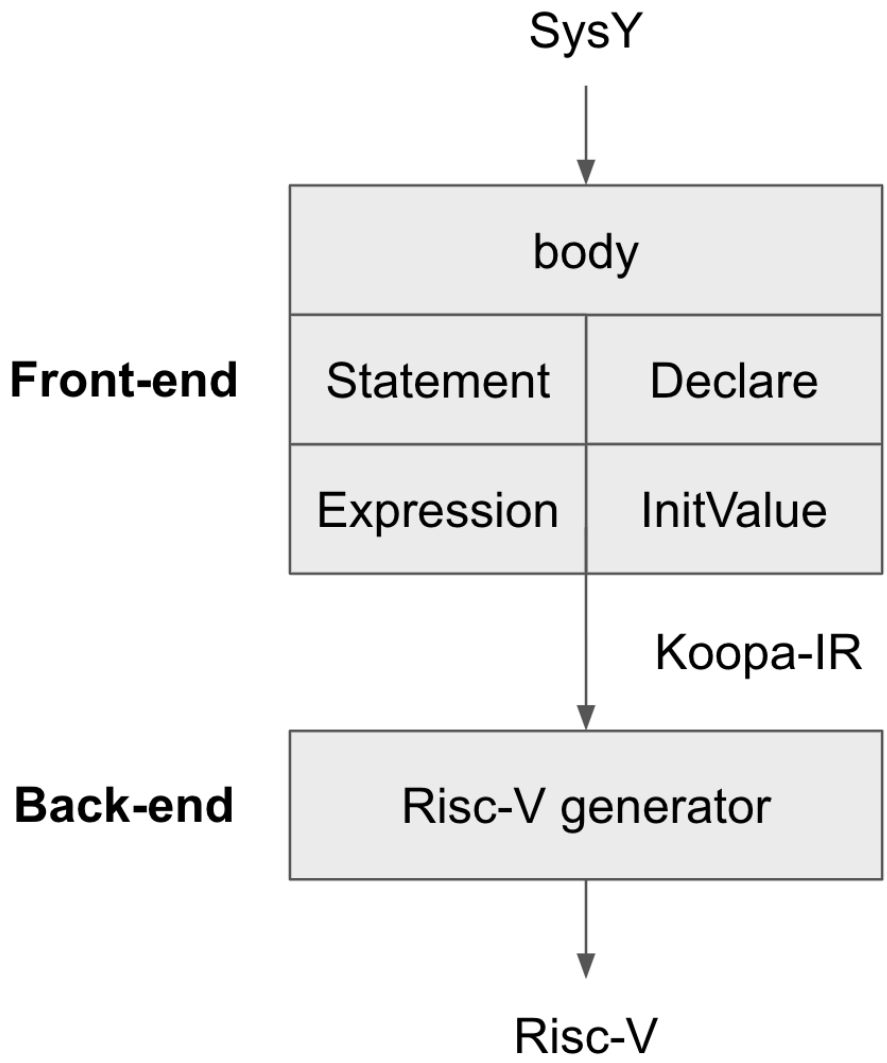
编译器使用`Rust`语言实现, 由于其语言特性保证了内存的安全性.

编译器的工作流程:

1. 词法+语法分析, 使用lalrpop库实现. 我的工作主要是写语法结构.
2. 语法分析得到`AST`抽象语法树.
3. 根据抽象语法树, `front-end`部分生成`Koopa-IR`中间表示(以文本代码的形式).
4. 将`Koopa-IR`文本形式代码转化为结构化`Koopa-IR`, 这里用到助教提供的Koopa标准库.
5. 根据结构化的`Koopa-IR`, 生成`Risc-V`文本代码.

2. 编译器设计

a) 主要模块组成



如图表示, `compiler`总体分为两个部分front-end, back-end. 值得注意的是在front-end, 我设计了三层的分层结构, 分别为`body`组成第一层, `decl`和`statement`组成第二层, `expression`和`InitValue`组成第三层. 高层模块会调用低层模块.

而在`back-end`中, 由于生成`Risc-v`的程序比较简单, 因此没有做过多的抽象从而只有一个模块.

b) 主要数据结构

整个项目只使用了`String`以及`HashMap`这两个`Rust`原生的数据结构.

`String`的使用在生成文本格式的代码中必不可少, 并且通过巧妙使用`String`自带的`replace`函数大大简化了代码复杂度.

`HashMap`的`(key, value)`形式显然完全适合于定义变量, 以及变量的查询操作. 且通过自定义`value`可以附带丰富的信息.

c) 主要算法设计考虑

1) dfs Ast树的算法

这也是在理论课上重点强调的一种实现`compiler`的算法, 通过遍历`parser`生成的`ast`, 我们便可以根据其结构进行每个节点`attribute`的计算, 生成中间代码.

2) HashMap算法

对于经典的`(key, value)`存储问题, 哈希表是一种优越的实现, 可以做到$O(1)$的时间复杂度, $O(n)$的空间复杂度, 达到了理论下界. 因此维护变量的定义以及引用, `HashMap`非常合适.

3) String操作的算法

`String`的使用在生成文本格式的代码中必不可少, 并且通过巧妙使用`String`自带的`replace`函数大大简化了代码复杂度.

3. 编译器实现

a) 对涉及的工具软件的介绍(为何选用).

本节主要解释**为何选用Rust**.

虽然之前没有真正使用过`Rust`语言, 但是已经听说过很多次, 因此编译课既然提供了此机会当然要试一下.

更具体的, `Rust`的内存安全特性是最吸引我的. 在以往写`c`语言的过程中, 我多次感受到了`index out of bound`, `invalid pointer`等问题的困惑, 往往程序行为不正确(例如 程序崩溃)的位置与真正出问题的位置没有关联, 这给我`debug`过程带来了很多困惑. 因此在听说`Rust`保障了内存安全后, 我非常想试一下`Rust`语言.

阅读文章What is memory safety and why does it matter?之后更让我觉得内存安全的语言可能是一种趋势.

b) 各个阶段编码细节.

1) 关键功能实现说明.

front-end

`mod.rs`中主要实现了高层的代码生成, 例如全局`function`的定义, 函数体内的`block`等.

`statement.rs`中实现了函数体内不同语句的生成, 主要处理了`if`, `if-else`, `while`, `return`等语句.

`declare.rs`中实现了函数体内变量定义, 主要处理了数组的生成, 以及定义变量的插入维护.

`initvalue.rs`中实现了定义新变量时初始化的问题, 主要处理多维数组的初始化.

`expression.rs`中实现了表达式的计算, 并通过自然的方式实现了常值表达式的计算.

back-end

`mod.rs`实现了`Koopa-IR`到`Risc-V`的转化, 由于官方`Koopa-IR`给的信息非常全因此比较容易实现.

2) 具体实现方法, (a) 选择该方法的原因, (b) 对必要代码的引用.

在这一小节重点介绍`front-end`, 因为在实现完前端算法后后端算法的思路比较直接.

Parser中加入if-else的解决方案.

这里是理论课上的内容, 在编程过程中参考了助教在文档中提供的dangling else. 实现基本上与wiki中一样.

```
// statement: open_statement
//          | closed_statement
pub enum Statement {
    Open(OpenStatement),
    Closed(ClosedStatement),
}

// open_statement: IF '(' expression ')' statement
//               | IF '(' expression ')' closed_statement ELSE open_statement
pub enum OpenStatement {
    If(Exp, Box<Statement>),
    Ifelse(Exp, ClosedStatement, Box<OpenStatement>),
    While(Exp, Box<Statement>),
}

// closed_statement: non_if_statement
//                 | IF '(' expression ')' closed_statement ELSE closed_statement
pub enum ClosedStatement {
    Stmt(Stmt),
    Ifelse(Exp, Box<ClosedStatement>, Box<ClosedStatement>),
}
```

Parser中加入global decl的解决方案.

相信在加入全局变量定义之后我们都会遇到`LALR`冲突的问题, 在进行分析之后发现主要是`type`冲突, 即输入在观察到`int`之后不知道归约为变量定义的类型还是函数返回值类型.

最终我的解决方案就是去掉`type`, 将函数定义写成一个整体, 这样就不存在规约冲突了.

```
  /home/compiler/src/sysy.lalrpop:161:16: 161:34: Local ambiguity detected

    The problem arises after having observed the following symbols in the input:
      "int"
    At that point, if the next token is a `r#"[_a-zA-Z][_a-zA-Z0-9]*"#`, then the
parser can proceed in two different ways.
```

```
    First, the parser could execute the production at
/home/compiler/src/sysy.lalrpop:161:16: 161:34, which would consume the top 1 token(s)
from the stack and produce a `BType`. This might then yield a parse tree like
      "int"    ┌ VarDef ";"
      ├─BType─┘          │
      └─VarDecl──────────┘

    Alternatively, the parser could shift the `r#"[_a-zA-Z][_a-zA-Z0-9]*"#` token and
later use it to construct a `Ident`. This might then yield a parse tree like
      "int" r#"[_a-zA-Z][_a-zA-Z0-9]*"# "(" FuncFParams ")" Block
      │     └─Ident──────────────────┘                      │
      └─FuncDef──────────────────────────────────────────────┘

    See the LALRPOP manual for advice on making your grammar LR(1).
```

expression.rs 中的return type

我在实现的过程中参考了文档中的建议, 对同一模块中的语句实现一个`trait`达到更好的一致性.

因此需要考虑该`trait`的`eval`函数的返回值的类型.

在`expression.rs`中我做出的抽象如下:

```rust
pub struct ExpRetType {
    pub size: i32,
    pub program: String,
    pub is_constant: bool,
    pub exp_res_id: i32,
}
```

`size`, `program`是根据我实现必须要维护的`attribute`, 在这里省去解释. 主要讨论后两个`field`对应的意义.

为了解决常量求值的问题, 增加`is_constant` `field`代表表达式的求值过程是否为`constant`.

1. `is_constant is true`, `exp_res_id` 就是这个常量值.
2. `is_constant is false`, 则`var{exp_res_id}`存储了这个变量.

因此在实现中非常容易处理常值表达式:

对于二元运算, 只要`l, r`均为常值则返回结果就是常值, 否则为变量. 一元运算同理.

```rust
return ExpRetType {
    size: size + 2,
    program: program,
    exp_res_id: size + 2,
    is_constant: ret_val1.is_constant && ret_val2.is_constant,
};
```

expression.rs 中二元运算的抽象

在实现过程中我们不难发现`mul`, `div`, `add`...这些操作全部类似, 不同的就是运算不同, 因此可以抽象得:

```
fn binary_operation(program: &mut String, size: &mut i32, op: &str, val1: &ExpRetType,
val2: &ExpRetType) -> i32 {
    let mut val = {
        if op == "mul" {
            val1.exp_res_id * val2.exp_res_id
        } else if op == "div" {
            val1.exp_res_id / val2.exp_res_id
        // ...
        } else {
            0
        }
    };

    if !val1.is_constant || !val2.is_constant {
        let name1 = get_name(val1.exp_res_id, val1.is_constant);
        let name2 = get_name(val2.exp_res_id, val2.is_constant);

        // if any value is constant, then there shouldn't be any code.
        if !val1.is_constant {
            program.push_str(&val1.program);
        }
        if !val2.is_constant {
            program.push_str(&val2.program);
        }
        program.push_str(&format!("    %var_{} = {} {}, {}\n", size, op, name1,
name2));
        val = *size; // it is not a constant, so return variable id.
    }

    val
}
```

这样可以大大降低代码复杂度.

而且`Rust`也支持了`match`得合并, 使得调用端实现进一步简化

```
RelExp::Ltexp(relexp, addexp, op) |
RelExp::Gtexp(relexp, addexp, op) |
RelExp::Geexp(relexp, addexp, op) |
RelExp::Leexp(relexp, addexp, op) => {
    let ret_val1 = (*relexp).eval(scope, size, is_pt);
    let ret_val2 = addexp.eval(scope, ret_val1.size, is_pt);
    let mut size = ret_val2.size + 1;
    let mut program = String::from("");
```

```
    let val = binary_operation(&mut program, &mut size, op, &ret_val1, &ret_val2);

    return ExpRetType {
        size: size,
        program: program,
        exp_res_id: val,
        is_constant: ret_val1.is_constant && ret_val2.is_constant,
    };
}
```

HashMap的形式

插入/引用变量的`HashMap`定义为`HashMap<String, (i32, i32)>`.

根据名字(`String`), 得到`(type, param)`. `type`指明了类型, 例如`integer`, `array`, `void`, `consntant`等. 第二维`param`维护额外信息, 例如`type`为`integer`的情况下, `param`指定`var_param`的地址. 在`type`为`function`的情况下, `param`指定参数的类型是否为`array`.

一下为几个例子.

```
scope.insert(format!("{}_function", &func_def.ident), (VARIABLE_INT | (bitset << TYPE_BITS), 0));
scope.insert(format!("{}", ident), (PARAMETER_ARRAY, size));
scope.insert("stoptime_function".to_string(), (VOID, 1)
```

变量作用域

这里的实现非常暴力, 即每一层都将`scope`指定的当前作用域下的定义全部`clone`一遍, 在之后的应用中修改不影响上一层, 并且同名变量会在`HashMap`中自动被替换.

trade-off: pros: 代码实现非常简单, 只需几行修改. cons: 时间复杂度变为平方级别 (根据Rust底层实现的不同可能不同).

```
fn dfs(pt: TreePoint, par: &HashMap<String, (i32, i32)>, size: i32) -> BodyRetType {
    // consider the indent!
    let mut size = size;
    let mut program = String::from("");
    // 变量作用域的主要实现: 每递归一层, clone scope, 之后插入的variable会发生replace.
    let mut scope: HashMap<String, (i32, i32)> = par.clone(); // inherit the variables from parent.

    // ...
}
```

外部函数的插入

由于我们的`HashMap`非常通用, 因此直接手动插入即可.

```
    let mut program = "
decl @getint(): i32
decl @getch(): i32
decl @getarray(*i32): i32
decl @putint(i32)
decl @putch(i32)
decl @putarray(i32, *i32)
decl @starttime()
decl @stoptime()\n\n\n\n".to_string();
    let mut scope: HashMap<String, (i32, i32)> = HashMap::new();

    // add std::functions to scope.
    scope.insert("getint_function".to_string(), (VARIABLE_INT, 0));
    scope.insert("getch_function".to_string(), (VARIABLE_INT, 0));
    scope.insert("getarray_function".to_string(), (VARIABLE_INT + (1 << TYPE_BITS),
0));
    scope.insert("putint_function".to_string(), (VOID, 1));
    scope.insert("putch_function".to_string(), (VOID, 1));
    scope.insert("putarray_function".to_string(), (VOID + (2 << TYPE_BITS), 1));
    scope.insert("starttime_function".to_string(), (VOID, 1));
    scope.insert("stoptime_function".to_string(), (VOID, 1));
```

Koopa-IR规定ret/jump必须是block最后一条语句的处理

这是一个非常棘手的问题, 因为我生成代码的形式是增量生成的, 每次合并两个`program`都需要特殊判断的话非常麻烦.

在和同学的讨论中我们发现了一种简单的方法: 在每一条`ret`/`jump`语句后都加一个`label`, 这显然满足了题目中的要求, 但是`Koopa-IR`中不能存在空的`block`, 因此我们在每个函数的最后加一个`ret 0`即可. 我认为这个实现非常优美.

```
Stmt::RetNone() => {
    program.push_str(&format!("    ret \n",));
    program.push_str(&format!("\n%entry_{}:\n", size + 1)); // insert a label.

    return ExpRetType {
        size: size + 1,
        program: program,
        exp_res_id: 0, // return stmt => -2;
        is_constant: false,
    }
},
```

if-else/while 的处理

由于`while`的处理和`if-else`类似, 即定义`label`以及其跳转即可, 这里只讨论`if-else`.

实现的思路为首先得到各个`submodule`的代码, 例如`if(sub1) sub2 else sub3`, 中的`sub1/2/3`. 这样在本层根据`if-else`的语句加入对应的单条语句, 将各个部分串联起来即可.

```
OpenStatement::Ifelse(exp, cs, os) => {
    let exp_val = exp.eval(scope, size, false);
    let cs_val = cs.eval(scope, exp_val.size);
    let os_val = os.eval(scope, cs_val.size);
    let size = os_val.size + 1;
    let name = get_name(exp_val.exp_res_id, exp_val.is_constant);

    // evaluate the condition.
    program.push_str(&exp_val.program);
    program.push_str(&format!("    br {}, %entry_{}, %entry_{}\n", &name, size, size +
1));

    // first part.
    program.push_str(&format!("\n%entry_{}:\n", size));
    program.push_str(&cs_val.program);
    program.push_str(&format!("    jump %entry_{}\n", size + 2));

    // second part.
    program.push_str(&format!("\n%entry_{}:\n", size + 1));
    program.push_str(&os_val.program);
    program.push_str(&format!("    jump %entry_{}\n", size + 2));

    // after.
    program.push_str(&format!("\n%entry_{}:\n", size + 2));

    return ExpRetType{size: size + 2, program, exp_res_id: 0, is_constant: false,};
},
```

break/continue 的处理

这一部分的难点主要在于处理低层的`break`, `continue`语句时并没有上层`while`产生的信息. 如果要求上层`while`传递参数的话会显著增加代码的复杂度.

本文的解决方法和理论课上讲的回填类似, 在低层首先插入`<replace_me_with_break>`这类特殊语句, 在上层`while`可以确定`label`之后使用`String`的`replace`功能即可完全代码生成.

该方法显著减少了代码复杂度, 并且使得编码的逻辑更加清晰. 代价是增加了一次对`String`的遍历, 我认为可以忽略.

```
Stmt::BreakKeyWord() => { // give `while` a hint, it'll replace it with `jump`.
    program.push_str("    <replace_me_with_break>\n");
    program.push_str(&format!("\n%entry_{}:\n", size + 1));
    ExpRetType {
        size: size + 1,
        program,
        exp_res_id: 0,
        is_constant: false,
    }
},
```

在上层`while`中:

```
// replace `break` and `continue`.
let program = str::replace(program, "<replace_me_with_break>", jump_instr);
let program = str::replace(program, "<replace_me_with_continue>", jump_instr);
```

函数参数

由于在`lv9`中加入了`array`参数, 并且这和`integer`参数的处理是本质不同的, 因此我们需要在函数声明的时候保存信息, 并在之后函数调用中根据该信息对相应的参数进行处理.

这里的实现就是枚举`parameters`, 通过二进制的压位实现.

trade-off: 使得`HashMap`中的`(key, value)`中的`value`仍然是`integer`类型, 不需要隔离函数引用和变量引用. 损失是限制了函数调用的最大参数个数为`32`. 总体上我认为这是一个合理的限制, 传递超过`32`个参数的程序很罕见.

```
fn calc_bitset(node: &FuncDef) -> i32 {
    match &node.params {
        None => {0},
        Some(v) => {
            let mut bitset = 0;
            let mut bin = 1;
            for x in &v.params {
                match x {
                    FuncFParam::Integer(_) => {} ,
                    FuncFParam::Array(_, _) => bitset |= bin,
                }
                bin = bin << 1;
            }
            bitset
        },
    }
}
```

函数调用

上面提到过函数调用需要特殊处理, 这里就体现了枚举参数进行求值, 在evaluate中要穿入`(bitset & 1) != 0`, 即确定我们想要`integer`还是`pointer`.

```
let mut is_first = true;
for exp in &v.params {
    let ret_val = exp.eval(scope, size, (bitset & 1) != 0);
    let name = get_name(ret_val.exp_res_id, ret_val.is_constant);
    size = ret_val.size;

    // we should first evaluate all the value, then use it.
    program.push_str(&ret_val.program);
    if is_first {
        params_str.push_str(&format!("{}", &name));
    } else {
        params_str.push_str(&format!(", {}", &name));
    }
    is_first = false;
    bitset >>= 1;
}
```

数组初始化问题

同样在`initValue.rs`中我定义的`return type`如下:

```
pub struct InitRetType {
    pub size: i32,
    pub program: String,
    pub is_allzero: bool,
    pub val: Vec<(bool, i32)>,
}
```

`is_allzero`对应了`0`初始化, `if is_allzero is true`, 我们就不需要`val: vector`了. 否则我们需要`val: vector`中的每个值顺序对应初始化中的每个域. `val: vector`中包含了我们对于数组初始化转化后的结果. 对应的函数如下:

思路就是确定每个`{}`对应的哪一维, 之后递归生成即可.

```
InitVal::MultiExp(const_vals) => {
    let mut size = size;
    let mut program = "".to_string();
    let mut val = Vec::<(bool, i32)>::new();

    for ele in const_vals {
        let mut ret_val;
        match ele {
```

```
            InitVal::SingleExp(_) => { // if it is a single, then multiple of last
dimension.
                ret_val = ele.eval(scope, size, &dims[dims.len()-1..dims.len()]);
            },
            _ => {
                while val.len() % (dims[dims.len() - 1] as usize) != 0 {
                    val.push((true, 0));
                }
                let pos = {
                    if val.len() == 0 {
                        1// dims.len() - 1
                    } else {
                        let mut pd = 1;
                        let mut pos = dims.len();
                        while val.len() % pd == 0 {
                            pos -= 1;
                            pd = pd * (dims[pos] as usize); // dims must be positive.
                        }
                        pos + 1
                    }
                }; assert!(pos != dims.len()); // must be multiple of last dimension.

                ret_val = ele.eval(scope, size, &dims[pos..dims.len()]);
            }
        }
        val.append(&mut ret_val.val);
        size = ret_val.size;
        program.push_str(&ret_val.program);
    }
    zero_padding(&mut val, dims);

    return InitRetType {
        size: size,
        program: program, // no code is needed.
        is_allzero: false,
        val: val,
    };
},
```

数组定义

上一个问题中我们确定了如何确定初始化的值, 在这个问题中确定如果产生赋值的代码:

这里的难点主要在于讨论所有的情况, 比如`global`和`non-global`的初始化就有所不同. 讨论清晰后根据格式初始化即可.

而数组的引用有了赋值的经验之后就很简单, 因此之后忽略.

```
VarDef::Identinitval(ident, dims, initval) => {
    let dim_pair = evaluate_dimension(&mut size, dims, scope);
    let dims = dim_pair.0;
    let dim_str = dim_pair.1;

    if dims.len() == 0 {
        let ret_val = initval.eval(scope, size, &[1]);
        size = ret_val.size + 1;
        assert!(ret_val.val.len() == 1);
        let name = get_name(ret_val.val[0].1, ret_val.val[0].0);

        if !is_global {
            program.push_str(&ret_val.program);
            // define.
            scope.insert(format!("{}", ident), (VARIABLE_INT, size + 1));
            // @x = alloc i32
            program.push_str(&format!("    @var_{} = alloc i32\n", size + 1));
            // assignment: store %1, @x
            program.push_str(&format!("    store {}, @var_{}\n", name, size + 1));
        } else { // global的初始值必须是constant.
            assert!(ret_val.val[0].0 == true); // must be constant.
            // define.
            scope.insert(format!("{}", ident), (VARIABLE_INT, size + 1));
            // @x = alloc i32
            program.push_str(&format!("global @var_{} = alloc i32, {}\n", size + 1,
ret_val.val[0].1));
        }
        return DeclRetType {size: size + 1, program};
    }

    // array.
    let ret_val = initval.eval(scope, size, &dims);
    let init_value_str = get_const_init_value_str(&ret_val, &dims);

    scope.insert(format!("{}", ident), (VARIABLE_ARRAY, size + 1));
    if is_global {
        program.push_str(&format!("global @var_{} = alloc {}, {}\n", size + 1,
&dim_str, init_value_str));
    } else {
        program.push_str(&format!("    @var_{} = alloc {}\n", size + 1, &dim_str));
        program.push_str(&format!("    store {}, @var_{}\n", init_value_str, size +
1));
    }

    return DeclRetType {size: size + 1, program};
},
```

Risc-V部分指令对imme的限制

有三条指令存在限制: `addi`, `lw`, `sw`. 解决思路就是将`imme`先加载到一个空闲的`register`中, 消除对立即数的引用.

这里主要观察`sw`的例子, 注意保存中间结果的`register`必须规定好, 一开始一个棘手的`bug`是我直接保存在了`src register`中, 在之后的 调用中`src register`的参数可能为`rsp`, 这样直接导致了`segment fault`.

```
fn riscv_sw(dst: &str, src: &str, imme: i32) -> String {
    assert!(src != "t3");
    let mut program = "".to_string();
    if imme < -2048 || imme > 2047 {
        program.push_str(&format!("    li t3, {}\n", imme));
        program.push_str(&format!("    add t3, {}, t3\n", src)); // can't write to rsp!
        program.push_str(&format!("    sw {}, 0(t3)\n", dst));
    } else {
        program.push_str(&format!("    sw {}, {}({})\n", dst, imme, src));
    }
    return program;
}
```

3) 编码的难点

这次实践作业充分让我认识到了大局观设计好的架构的重要性. 以往我们只是给定`input`, `output`, 只需要实现一个非常小的程序. 我们在事先就对整个程序的框架有一个足够的了解. 但是在写编译`lab`的过程中我反复出现了在写`lv_i`时的架构, 到`lv_i+1`必须要重构的现象. 这让我发现了我这方面能力的欠缺, 这也启发我在之后研究过程中注意锻炼自己这方面能力.

对于局部的代码, 我认为困难不大. 因为我已经有很多编程的经验, 加上`Rust`编译器的严格检查, 基本上过了编译不会有很多问题.

c) 自测试情况说明, 如何构造, 结果说明, 测出什么主要错误, 怎么发现的(发生错误的条件), 怎么解决的.

自测试主要通过首先`autotest`, 找到出问题的样例, 单独拿出来进行测试.

在实践的后半部分很多时候生成的代码长度非常长, 我主要通过**二分删除代码**, 定位出问题的语句, 之后重点调试对应部分.

个人`debug`的体验非常好, 基本上没出什么困惑的`bug`(完全得益于Rust), 但是以往开发`c/c++`的`debug`时间有一部分转移到了和`Rust`编译器作斗争上. 找到的`bug`有全局唯一的`id`维护出问题, 维护`InitValue`的`vector`长度不匹配的问题.

在实现`compiler`的过程中我还发现了一个很好的习惯是加入`assert(condition)`. 我们实现的每一部分代码正确工作都是有一定条件的, 往往我们无法周全的考虑所有的状态. 因此将这些条件放入`assert`我可以快速的定位到可能出问题的地方. 我遇到的问题主要是在实现的过程中忘了修改对应 的模块, 这时`assert`出问题直接找到对应的位置补充相应的代码即可.

d) 课程工具中存在的问题

个人在实践过程中的主要困惑在于使用官方`Koopa-IR`形式生成`Risc-V`.

我认为这部分如果更好使用借口的信息不够充分, 在一开始无从下手, 我认为可以通过给简单的`example`解决这个问题.

4. 实习总结

a) 收获与体会

1) 主要的收获是什么

主要收获就是自己第一次从头到位设计了一个比较大规模程序的结构. 最后在front-end得出了一个比较清晰的结构. 对我设计程序结构的能力提升很大. 当然这也让我了解了一个简单的`compiler`的组成.

2) 学习过程的难点是什么

由于助教提供的文档非常详细, 在充分阅读文档以及和同学讨论后我认为思路上难点不多.

难点主要在于调整自己的心态, 积极地去完成这样一个项目.

b) 对课程的建议

1) 实习过程优化的建议

感觉很完善了.

2) 实习内容的建议

我认为可以讲一些权威`compiler`的代码结构.

因为实践课很多都在讲怎么用`yacc/lex`这种技术性的东西, 同学上课也跟不上, 还得下课自己学. 我认为学习他人编译器的结构对我们帮助更大, 这样能让我们在实现自己的编译器时有一个结构性的认识.

3) 对老师讲解内容与方式的建议

个人感觉需要加强lab与实践课的联系.