编译实习报告

丁聪
1600011785

日期：2021 年 1 月 22 日

目录

1 SysY 到 Eeyore 的转换 **3**

1.1 SysY 的语法定义 . . . 3
1.2 词法分析器的设计 . . . 3
1.3 语法树的设计 . . . 4
1.4 符号表的设计 . . . 5
1.5 Eeyore 的语法定义 . . . 6
1.6 语法树的遍历与 Eeyore 的生成 . . . 6

2 Eeyore 到 Tigger 的转换 **8**

2.1 Tigger 的语法定义 . . . 8
2.2 符号表的设计 . . . 8
2.3 栈帧的设计与寄存器分配 . . . 8
2.4 寄存器分配 . . . 8

3 Tigger 到 RISC-V 的转换 **10**

3.1 转换规则 . . . 10
3.2 逻辑与的处理 . . . 10

4 总结 **11**

1 SysY 到 Eeyore 的转换

1.1 SysY 的语法定义

SysY 是 C 语言的一个子集，每个 SysY 程序的源码存储在一个扩展名为 sy 的文件中。该文件中有且仅有一个名为 main 的主函数定义，还可以包含若干全局变量声明、常量声明和其他函数定义。SysY 语言支持 int 类型和元素为 int 类型且按行优先存储的多维数组类型，其中 int 型整数为 32 位有符号数；const 修饰符用于声明常量。

SysY 语言本身没有提供输入/输出 (I/O) 的语言构造，I/O 是以运行时库方式提供，库函数可以在 SysY 程序中的函数内调用。部分 SysY 运行时库函数的参数类型会超出 SysY 支持的数据类型，如可以为字符串。SysY 编译器需要能处理这种情况，将 SysY 程序中这样的参数正确地传递给 SysY 运行时库。有关在 SysY 程序中可以使用哪些库函数，请参见 SysY 运行时库文档。

函数：函数可以带参数也可以不带参数，参数的类型可以是 int 或者数组类型；函数可以返回 int 类型的值，或者不返回值 (即声明为 void 类型)。当参数为 int 时，按值传递；而参数为数组类型时，实际传递的是数组的起始地址，并且形参只有第一维的长度可以空缺。函数体由若干变量声明和语句组成。

变量/常量声明：可以在一个变量/常量声明语句中声明多个变量或常量，声明时可以带初始化表达式。所有变量/常量要求先定义再使用。在函数外声明的为全局变量/常量，在函数内声明的为局部变量/常量。

语句：语句包括赋值语句、表达式语句 (表达式可以为空)、语句块、if 语句、while 语句、break 语句、continue 语句、return 语句。语句块中可以包含若干变量声明和语句。

表达式：支持基本的算术运算（+、-、*、/、%）、关系运算（==、!=、<、>、<=、>=）和逻辑运算（!、&&、||），非 0 表示真、0 表示假，而关系运算或逻辑运算的结果用 1 表示真、0 表示假。算符的优先级和结合性以及计算规则 (含逻辑运算的“短路计算”) 与 C 语言一致。

1.2 词法分析器的设计

对于 SysY 编译器的构造，第一步就是对 SySY 语言进行词法分析。这里我们利用 2.6.4 版本的 Flex 软件进行词法分析。并且在 `sysy.l` 文件中对如下 token 类进行词法分析。

保留字：对于保留字，我们直接返回该保留字对应的 token 即可。

标识符：根据标准中的语法设计正则表达式 `[a-zA-Z_][a-zA-Z0-9_]*` 来捕获文本流中的标识符，并将捕获到的字符串保存到 `yylval.string` 属性中以便后续使用。

整数字面量：SysY 中的整数包含三类：

- 十进制数 `[1-9][0-9]*`
- 十六进制数`"0"[xX][0-9a-fA-F]+`
- 八进制数`"0"[0-7]*`

这三种数都会被识别为整数，并且将字符串保存到到 `yylval.string` 属性中，后续使用 `std::stoi` 并将 `base` 参数指定为 0，即可自动识别进制并转换为整型。

单行注释与空格：值得注意的是，空格一共有五种（`[ \t\v\f\r]`）。单行注释只需要识别从双斜杠直到换行符之间的内容。

多行注释：由于多行注释对应的正则表达式较为复杂，我们在 Flex 程序中用一段代码来提取多行注释。

```
{
    int c;
    while((c = yyinput()) != 0) {
        if(c == '\n')
            yylineno++;
        else if(c == '*') {
            if((c = yyinput()) == '/')
                break;
            else
                unput(c);
            }
    }
}
```

1.3 语法树的设计

我们在 `sysy.y` 文件里进行语法分析，并且在语法分析的过程中给完成语法树的构造。

这里我们先将文件中提供的语法翻译为 BNF，再根据 BNF 文法设计语法树的结点种类，以 Node 为基类，建立如下的继承关系。

- Node 节点基类
  - NRoot 根节点
  - NExpression 表达式节点
    - NStatement 语句节点
      - NBlock 语句块节点
      - NAssignment 赋值语句节点
      - NIfElseStatement If-Else 语句节点
      - NIfStatement If 语句节点
      - NWhileStatement While 语句节点
      - NBreakStatement Break 语句节点
      - NContinueStatement Continue 语句节点
      - NReturnStatement Return 语句节点
      - NVoidStatement 空语句节点
      - NDeclareStatement 变量声明语句节点
    - NIdentifier 标识符节点
      - NArrayIdentifier 数组标识符节点
    - NNumber 整数节点
    - NArrayDeclareInitValue 数组声明维度节点
    - NConditionExpression 条件表达式节点
    - NBinaryExpression 双目表达式节点

        * NUnaryExpression 单目表达式节点
        * NCommaExpression 并列语句节点
        * NFunctionCallArgList 函数调用参数列表节点
        * NFunctionCall 函数调用节点
        * NFunctionDefineArg 函数定义参数节点
        * NFunctionDefineArgList 函数定义参数列表节点
- NDeclare 声明语句节点
    - NVarDeclareWithInit 带初始化变量声明节点
    - NVarDeclare 变量声明节点
    - NArrayDeclareWithInit 带初始值数组声明节点
    - NArrayDeclare 数组声明节点
- NFunctionDefine 函数定义节点

我们在进行 LR 分析时，就会建立节点，并连接形成程序对应的语法树。

1.4 符号表的设计

首先我们设计了 Var 和 Const 两个变量，分别表示变量信息和常量信息。

```
class Var{
public:
  std::vector<int> shape;
  bool is_array;
  std::string name;
  Var(std::string name, bool is_array = false, std::vector<int> shape = {});
};

class Const {
public:
  bool is_array;
  std::vector<int> shape;
  std::vector<int> value;
  Const(std::vector<int> value, bool is_array = false,
            std::vector<int> shape = {});
  Const(int value);
};
```

is_array 用来指示该变量/常量是否为数组，shape 是数组的维度，如果不是数组就为空，name 是该变量在 Eeyore 程序中的名字。value 是这个常量的值。

我们构造 Symtab 类表示符号表，其中有如下几个成员：

```
  using SymbolTable = std::vector<std::unordered_map<std::string, Var>>;
  using ConstTable = std::vector<std::unordered_map<std::string, Const>>;
  using FuncTable = std::unordered_map<std::string, bool>;

  SymbolTable symbol_table = {{}};
```

```
ConstTable const_table = {{}};
FuncTable func_table = {};
```

SymbolTable 表示变量表，ConstTable 表示常量表，FuncTable 表示函数表。

在新进入一个域时，我们就往 SymbolTable 和 ConstTable 中推入一个新 map，用来记录这个域中的变量。退出时直接弹出这个 map 即可。所以，在符号表中寻找符号时，要逐层搜索，直到找到最接近的符号。代码如下：

```
Var&
Symtab::find_symbol(std::string name) {
    int i = symbol_table.size() - 1;
    for (; i >= 0; --i) {
        auto find = symbol_table[i].find(name);
        if (find != symbol_table[i].end()) {
            return find->second;
        }
    }
    throw std::out_of_range("No symbol called "+ name);
}
```

1.5 Eeyore 的语法定义

Eeyore 为一种三地址中间代码，其部分要求如下。

- Eeyore 要求所有语句单独占一行，允许缩进但不要求。
- 变量分作原生变量、临时变量和函数参数。
- 原生变量对应 SysY 中的变量，以“T”开头，其后为数字（T[0-9]+）。
- 临时变量为翻译过程中引入的新变量，以“t”开头，其后为数字（t[0-9]+）。
- 函数参数为该函数的形式参数，以“p”开头，其后为数字（p[0-9]+）。
- 支持所有 SysY 的运算符（五种双目算术运算、六种双目逻辑运算、两种单目算术运算、一种单目逻辑运算）。
- 支持数组赋值（array[index] = value）和数组取值（value = array[index]）。
- 支持条件跳转（if-goto）和直接跳转（goto）控制语句。
- 跳转目的标签以“l”开头，其后为数字（l[0-9]+）。
- 函数名必须以“f_”开头，函数调用时需要设置实际参数（param）。
- 支持 SysY 的四个 IO 库函数。

1.6 语法树的遍历与 Eeyore 的生成

Eeyore 是一种三地址码，用作 SySY 语法分析后的输出格式。Eeyore 的设计同样遵循简洁的原则，使代码易读易调试。

在建立语法树后，我们从根节点出发，调用每个节点的 `generate_Eeyore` 函数，递归下降地完成从 SysY 到 Eeyore 的转变。由于篇幅限制，我不详尽地列举转换的细节，只提及一部分转换的要点。

临时变量的声明语句先输出：由于 Eeyore 语言的设置，我们需要在函数定义的开头声明全部临时变量。这里我们先把待输出语句保存起来，在函数翻译结束时一并输出。先输出变量定义的语句，再遍历以输出其他语句。

全局变量的声明：由于 Eeyore 语法的限制，全局变量的初始化只能再 main 函数中实现。所以对于全局语句，如果是变量声明就直接输出，如果是变量赋值就在 main 函数的开头执行。

变量范围的控制：我们已经在符号表中实现了 `create_scope` 等相关函数。只需要在函数定义、退出以及进入或退出新的代码块时进行 scope 的创建和删除即可。

函数返回值：根据函数名查表，判断函数类型，并决定返回值。值得注意的是 void 函数在 Eeyore 中也需要显示 return。

控制流的转变：调用函数时需要使用 param 语句传参。

表达式的求值：我们使用了一个栈 `EeyoreList` 来进行表达式的求值。将需要求值的运算符压入栈中，运算时弹出，再把运算结果压栈供后续使用。

静态求值：常数相关的定义与表达式将直接进行静态求值。求值的过程实现在 `eval` 方法里。

数组的定义与使用：数组的定义较为复杂。首先我们使用静态求值的方法把数组维度全部计算出来。之后更复杂的是数组的初始化。如果全部是数字（可以有缺省），则 `all_is_number` 为真，直接处理即可。若不是全为数字，则根据对应维度分别处理。数组的使用需要根据符号表中的信息，计算对应数组值相对数组开头的位置，然后在进行赋值或取值。

2 Eeyore 到 Tigger 的转换

2.1 Tigger 的语法定义

Tigger 是一种三地址中间代码，其形式与 RISC-V 相似且寄存器与 RISC-V 设置相同，其部分要求如下。

- 可用寄存器共 28 个，其中 x0 恒为 0，余下的 s0-11、t0-6、a0-7 为通用寄存器。所有 s 寄存器为被调用者保存，所有 t 和 a 寄存器为调用者保存。
- 支持 Eeyore 中所有双目运算，第一个被操作数是寄存器，第二个被操作数可以是寄存器或立即数。支持立即数的二元运算只有 + 和 <。
- 支持 Eeyore 中所有单目运算。操作数是寄存器。
- 跳转语句和跳转目的标号与 Eeyore 相同。
- 函数声明形如 f_somefunc[a][b]，第一个方括号中为形式参数数目，第二个方括号中为函数栈大小。函数实际参数通过 8 个 a 寄存器传递。
- 支持数组访问，数组地址和源/目的值必须为寄存器，数组下表必须是数字。
- 局部变量存于栈中，全局变量声明以 v 开头，其后为数字标号。
- 栈中局部变量可以通过 load/store 访问，可以通过 loadaddr 获得地址。
- 全局变量通过 load 读值，通过 loadaddr 获得地址，不允许 store，通过数组访问进行存值。

2.2 符号表的设计

符号表的设计与上一阶段类似，此时不需要设计常量表。

2.3 栈帧的设计与寄存器分配

函数的栈帧设计如下：

- 参数列表
- 本地变量

函数调用时，先将所有参数压栈，此后检查 Eeyore 代码中定义的本地变量数目，栈帧的大小为本地变量数 + 参数数量。需要访问时可以直接从栈帧对应位置加载到寄存器中。位置（序号）保存在符号表中。这里可以保证寄存器使用的数量永远小于等于 3。

2.4 寄存器分配

我们使用 `find_reg` 和 `release_reg` 函数来寻找空余寄存器和释放寄存器。

```
std::string
find_reg() {
    for (int i = 0; i < 12; ++i) {
        if (!is_used[i]) {
            is_used[i] = true;
            return "s"+std::to_string(i);
        }
```

```
    }
    return "t0";
}

void
release_reg(std::string reg) {
    if (reg[0] == 's')
        is_used[std::stoi(reg.substr(1))] = false;
}
```

由于栈帧的设计和寄存器分配策略，我们可以保证寄存器数量的充足。

3 Tigger 到 RISC-V 的转换

3.1 转换规则

Tigger 作为三地址码，和 RISC-V 已经非常接近，几乎只需要根据文档对每一条语句做转换即可。转换直接在 `tigger.y` 中完成，转换规则可以直接参阅代码。

3.2 逻辑与的处理

这次作业中的一大难点在于逻辑与的处理。由于测试样例中有一个空余的寄存器 s0，所以这里可以使用了 s0 进行辅助，实现逻辑与。（假设需要翻译的语句为 `C = A && B`。

```
snez s0, A
snez C, B
and C, C, s0
```

然而和其他同学讨论的过程中还听说了另一种不需要寄存器的做法，这里写出以供参考。

```
seqz C, A
add C, C, -1
and C, C, B
snez C, C
```

4 总结

本次编译实习利用 Flex 和 Bison 两种工具，实现了从 SysY 语言经过 Eeyore 和 Tigger 的中间表示，最终转换为 RISC-V 的全过程。通过编译实习的实验，有了亲自动手实践词法分析和语法分析、函数调用、寄存器分配等编译流程的机会，也对编译器的构造与实现有了深入的理解。最后作为第一届合并编译理论和实习课的小白鼠，特别感谢助教和老师在平台搭建以及调试工具上的帮助（大家一起踩过了许多坑）！完结撒花！