编译 (H) 期末报告

张鑫 21307130295

2024 年 6 月 20 日

1 Introduction

复旦大学编译 (H) 课程以 Fudan MiniJava(MiniJava 的一个子集，一种类 java 语法的简单语言) 为源语言，详细阐述了一个最基本的、完整的编译器的整个过程与结构，并要求学生利用所学知识，实现一个 Fudan MiniJava 的编译器，最终生成 arm 上的目标汇编代码并用 qemu 模拟运行，检验正确性。

本报告旨在总结本人在课程中的所学所获，主要对整个编译流程及其对应的代码实现进行阐述，以便未学习本课程的计算机学生或感兴趣人士，能够对较为容易地理解我的编译器的代码实现。

本课程和报告所涉及的编译器采用前后端分离的设计，使用 Tiger IR+ 和 LLVMIR 作为中间表示 (intermediate representation)。前端过程主要包括 Parsing、Type Checking、Translation to IR 和 Canonicalization；后端过程主要包括 LLVM Instruction selection、Liveness analysis、Static Single Assignment、RPi Instruction selection 和 Register allocation。

接下来，本报告将详细阐述前端流程和后端流程的每一个阶段在我的编译器中的具体实现方式。在 Section "Frontend" 中我将详细阐述前端，在 Section "Backend" 中我将详细阐述后端，最后简单介绍我实现的编译器的构建和使用方式，并总结我一学期以来的所学所获。

在正式开始之前，我将提供一个样例.fmj 程序，以之为案例详细解释每一个阶段到底做了什么工作。完整程序见/docs/example/example_in_report.fmj

2 Frontend

2.1 Parsing

Parsing 是整个编译的第一阶段，其目的是将源程序的字符串形式的代码去除注释后，转换为具有结构、语义等信息的抽象语法树（Abstract Syntax Tree，简记为 AST）。

在我的实现中主要分为词法分析和语法分析。其中词法分析通过 Lex 快速生成词法分析器，将字符串形式的代码转换为一系列合法的 terminal，在这个过程中，如果遇到不合法的字符串（例如不合法的标识符：0xaw）会产生由词法分析错误导致的 "Syntax Error"；语法分析通过 Yacc 快速生成语法分析器，语法分析器接收由词法分析器输出的一系列 terminal，根据事先用 CFG（Context-Free Grammar）定义好的语法规则进行派生，不断重复进行 "匹配——动作"。在每一条语法规则的动作中，调用对应的 AST 节点的初始化函数，返回这个 AST 节点，并将之与派生出的 non-terminal 符号所关联，以便递归使用。例如样例程序中的 length(d.c)，根据我们的语法规则，d.c 会被派生为一个 EXP，并且将它匹配-动作过程中得到的 AST 节点（记为 node_a）与之绑定。因此 length(d.c) 会得到 length(EXP)，又对应了另一条语法规则，因此又可以返回新的 AST 节点 A_LengthExp(pos, node_a)，如此反复，我们最终会得到 AST 的根节点 A_Prog。

在 Parsing 的语法分析阶段，我实现了如下 3 种错误报告、处理和恢复：

- **缺失的分号**：例如当样例程序中第 9 行的分号去除，则编译器会输出错误提示信息："Line_9: ';' expected"，并且跳过到下一个合法处，继续进行后续的语法分析。具体做法是定义了一个 non-terminal 符号：CHECK_SEMI，用它替换每一条语法中末尾出现的分号，则根据它的 CFG 语法规则，当缺少分号时，会跳到最近的"}"或者";"，但是为了不破坏"}"与之前的匹配，需要自行在输入流中加入一个"}"，使用定义在 src/lib/frontend/lexer.lex 文件末尾的 add_char 函数完成此功能，调用 lex 的 unput 函数实现。将分号替换为 CHECK_SEMI 后，需要进行错误报告和恢复，使用如上图自定义的函数 check_semi，根据 CHECK_SEMI 返回的值来判断是否发生缺失分号，并报告错误行数最后调用 yyerrok 恢复正常 parsing。
- **缺失的成对小括号**：处理逻辑与"缺失的分号"相同，都是通过新定义一个特殊的 non-termin 符号：CHECK_PARENT 和其对应的检查函数 check_parentheses 实现。
- **形参缺失类型定义**：处理方式为，在缺失类型定义的地方设置 error 的检查，忽略后续所有的形参声明，一直跳过到形参定义的结束标志：")"。

完成 Parsing 后我们得到一颗完整的抽象语法树，其根节点即 main.c 中的"A_prog root;"。将其打印输出得到可视化的抽象语法树结构，结果见/docs/example/example_in_report.2.ast 文件。

2.2 Type Checking

Type Checking 是针对 Parsing 阶段得到的 AST 进行类型检查，这是因为一个程序可能符合语法，但是符合语法的语句不一定有实际意义。例如样例程序中，变量 b 是一个 class 类型的变量，如果我们对它赋值为 0，或者定义了一个未定义的类为类型的变量等等，都是不允许出现的情况。我们希望在类型检查阶段对所有的表达式的类型进行严格的检查，只允许 int 和 float 的隐式转换，子类和父类的 upcast，以减少"即使符合语法规范，但是编译出的程序无法正常运行"的情况。并且在检查到错误的类型时输出相应的提示信息和出现的位置信息，例如上述例子，输出："(line:7 col:5) Assignment between incompatible types."。

我的编译器中，该过程实现在文件 src/lib/frontend/semant.c 中，入口为：

```
T_funcDeclList transA_Prog(FILE *out, A_prog p, int arch_size);
```

其中 out 是输出错误信息的文件对象，p 是一个 AST 的根节点；arch_size 是所使用的架构所使用的字大小，为 8 表示 64 位，为 4 表示 32 位，对本阶段而言可暂时忽略。该函数主要进行 Type Checking 的初始化，然后调用对应的函数，分别对所有类定义和主方法进行类型检查。

2.2.1 类的检查过程

类检查的入口函数为 transA_Classes。

首先调用 build_cenv 函数构建类环境。我的类环境是一系列表，但对于类型检查阶段而言，只涉及到表 CENV，记录了"类名->ClassEntry"的映射。每一个 ClassEntry 包含了类定义的 ast 节点、父类名、类继承检查状态、变量环境（venv）和方法环境（menv）。build_cenv 遍历每一个类，首先检查是否重定义，然后调用 transA_VarDecList 函数进行变量的类型检查，构建 venv；调用 build_menv 构建 menv，然后插入一条"类名->ClassEntry"到表 CENV 中。需要注意的是，在检查类变量的类型时，如果碰到变量的类型是一个类，暂时跳过它，因为可能它的类还没有被加入到 CENV，在后续检查。

然后调用 check_inheritance 进行类继承的检查，以每一个类继承检查状态为 E_transInit 的类为入口，递归检查它的父类是否存在，并在继承父类的方法和变量的过程中，检查是否存在变量重定义和方法重写的正确性。处理继承父类的变量的函数为 inherit_vars，处理继承父类方法的变量为 inherit_mtds。

以样例程序为例，构建完类环境 CENV 之后，我们得到如下表和数据结构：其中因为 base 是一个基类，

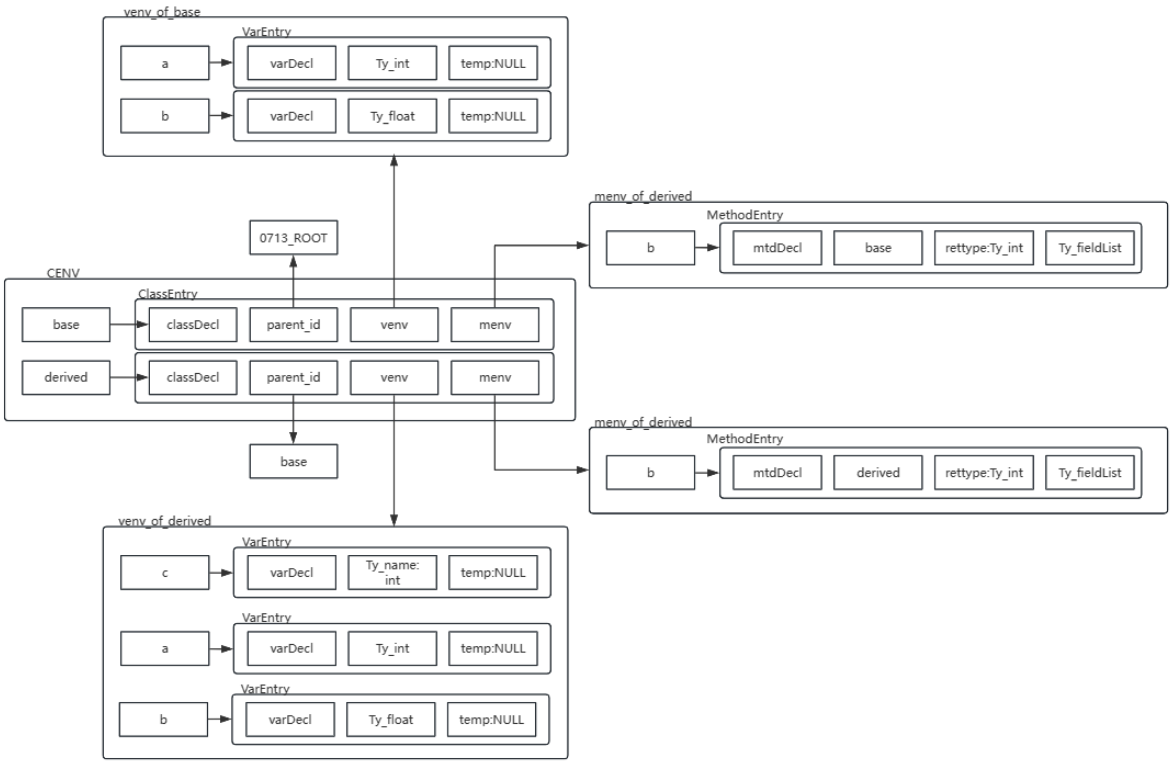


图 1: 样例程序的 CENV 结果

没有父类的类我们给它一个 dummy 父类，名为 0713_ROOT。同时，子类 derived 的 venv 中有继承自父类的变量 a 和 b。

获取类环境后，我们在 transA_Classes 中检查所有的、之前跳过的类型是一个类的类变量，其类型是否是一个被定义的类。然后遍历类的方法列表，调用 transA_ClassMethod 检查类的每一个方法中的类型是否正确。

2.2.2 主方法的检查过程

主方法类型检查的入口为 transA_MainMethod。主方法的类型检查流程较为简单，基本为线性流程。首先调用 transA_VarDecList 构建变量环境 VENV，然后调用 transA_StmList 对每一条 statement 进行类型检查。所用的数据结构与上方 CENV 的样例完全类似。

2.2.3 类型检查的几个重要函数

```
static Tr_exp transA_VarDecList(FILE *out, A_varDeclList vars, S_table venv);
```

接收变量定义列表 vars，输出变量环境到 venv。

```
static expty transA_Exp(FILE *out, A_exp exp, int* has_location);
```

接收一个表达式 exp，返回它的类型，并将其是否有 location 设置到 *has_location 中。

```
static Tr_exp transA_Stm(FILE* out, A_stm stm);
```

接收一个语句 stm，检查它的类型使用是否正确。

```
static void build_menv(FILE* out, A_classDecl cls, S_table menv);
```

接收一个类定义列表 cls，输出方法环境到 menv。

另有一些 helper functions，但不涉及到类型检查的主流程。各方法的详细实现参见 src/lib/frontend/semant.c，其中有一些代码注释可以帮助理解，在此不作赘述。

2.3 Translation to IR

完成类型检查之后，编译器前端的处理基本就结束了，下一步是将 AST 转换为中间表示 IR。我的实现中，将其转换为 Tiger IR+，详细结构定义见文件 src/include/utils/dsa/tigerirp.h。

这一步其实可以与类型检查同步进行，这也是为什么在类型检查部分的函数，都以 “trans” 开头来命名。具体而言，当类型检查完一条语句或者一个表达式后，不仅返回该表达式的类型（如果有），还要返回表示该语句或表达式的 IR 数据结构节点。因为类型检查的过程也是递归遍历整个 AST，所有这个过程是可以同步进行的。

我的具体实现中，在 transA_Exp 和 transA_Stm 的 return 语句调用对应的、实现在 src/lib/frontend/-translate.c 中的构造函数。但是为了减少 semant 模块和 translate 模块的耦合性，我们使用了一个公共的数据结构 Tr_exp 作为两个模块信息传递的媒介。具体而言，semant 模块生成 Tr_exp，并调用 translate 模块的构造函数；translate 模块的构造函数根据其参数（包含 Tr_exp），将之解析转换为它期望的 Tiger IR+ 中的数据结构节点。

translation 结束之后，我们得到一个完整的 Tiger IR+ 表示的程序，以样例程序为例，它的 IR 表示参见 src/docs/example/example_in_report.3.irp 文件。

2.3.1 实现中的一些关键点

- **T_Seq 的正确使用**：由于 T_Seq 默认不是以 NULL 结尾，并且只有一条 stm 时不应该使用 T_Seq，这给之前 semant 时，许多用到递归的地方带来了不小的影响。为了不大幅度影响总体逻辑，我实现了一个简单的 sequence buffer，当翻译线性表结构的时候，首先将元素的指针存于 buffer，再调用函数将这些元素组合成可能的 T_Seq，提供了可变长参数的宏 Sequence() 来增加可读性。这看似没有必要，但是像翻译 VarDecList 中可能存在声明时初始化的情况，可以提供很大的便利。sequence buffer 的另一种使用方式是手动调用 seqBufPush 添加元素，然后使用宏 SequenceFromBuffer() 获得 T_Seq 序列。

- **patchList 和 doPatch**：这是正确处理 if 语句和 while 语句的关键。因为在 translate 这两种语句中的布尔表达式时，暂时不知道其跳转的目的标签。因此我们的处理方式是将其暂时留存，利用 patchList 存放需要跳转到同一个标签的这些留存位置，当我们能够确定这些位置所需要填入的标签时，用 doPatch 进行填充。例如样例程序中，if(b.a == d.a)，在翻译完 b.a == d.a 并解析为一个跳转的节点时，因为要调用 IR 中跳转结构的构造函数 T_Cjump，需要指定参数 true label 和 false label，但此时 label 尚未可知，需要等到调用 Tr_IfStm 翻译这条 if 语句时才能确定，在 Tr_IfStm 函数中使用 patchList 和 doPatch 机制进行标签的填放。

- **检查是否处于 while 语句中**：维护了一个栈 while_stack，存放 while 语句的 test 标签和 end 标签，既可以用于 continue 和 break 的 semant 阶段，也方便 translate 阶段 patch 这些标签。

2.3.2 弃用了 unified object record，用 offset_env 记录类成员的偏移信息

因为 unified object record 会导致开辟多余的空间，在类数量多或者有几个十分庞大的类时，生成的程序对空间的利用率大大降低。为了解决这个问题，我的实现方式是：为每一个类记录一个私有的 offset 表。

新增了一个的全局表 OFFSET_ENV，记录类中每个成员内存位置的偏移量，从 arch_size 开始（为了与 null 区分，表中记录时不从 0 开始，最后生成指令时才从 0 开始）。

新增了一个新的 E_enventry 类型：

```
E_enventry E_OffsetEntry(unsigned long long local_size, S_table local_offtbl);
```

全局表 OFFSET_ENV 存储 class_name -> local_offtbl 的映射。

每一个类的 local_offtbl 中存储 member_name -> private_offset 的映射。

重要性质：父类的所有 offset 全都处于前半部分，子类只在父类的 offset 项后面新增。

处理 offset 的流程如下：

1. 在为每一个类建立 venv 时，同步记录和增加 local_offtbl

2. 在为每一个类建立 menv 时，暂不记录 method 的偏移（因为可能是重写父类的方法，父类中已经记录了相应的偏移）

3. 检查继承关系时，先继承父类的 offset 表，再继承 var 表和 method 表，最后再将新增的子类方法的偏移加到子类 offset 表的末尾（完成第 2 步中未记录的偏移量）

4. 继承父类的 offset 表时，子类的 offset 表中只包含子类新增的 var 的偏移，为了维护上方所说的“重要性质”，将子类的表中各项的 offset 加上父类 offset 表的 size（即偏移）

仍然以样例程序为例，最后得到的 offset_env 结构和内容（以 arch_size=8 为例）如下：

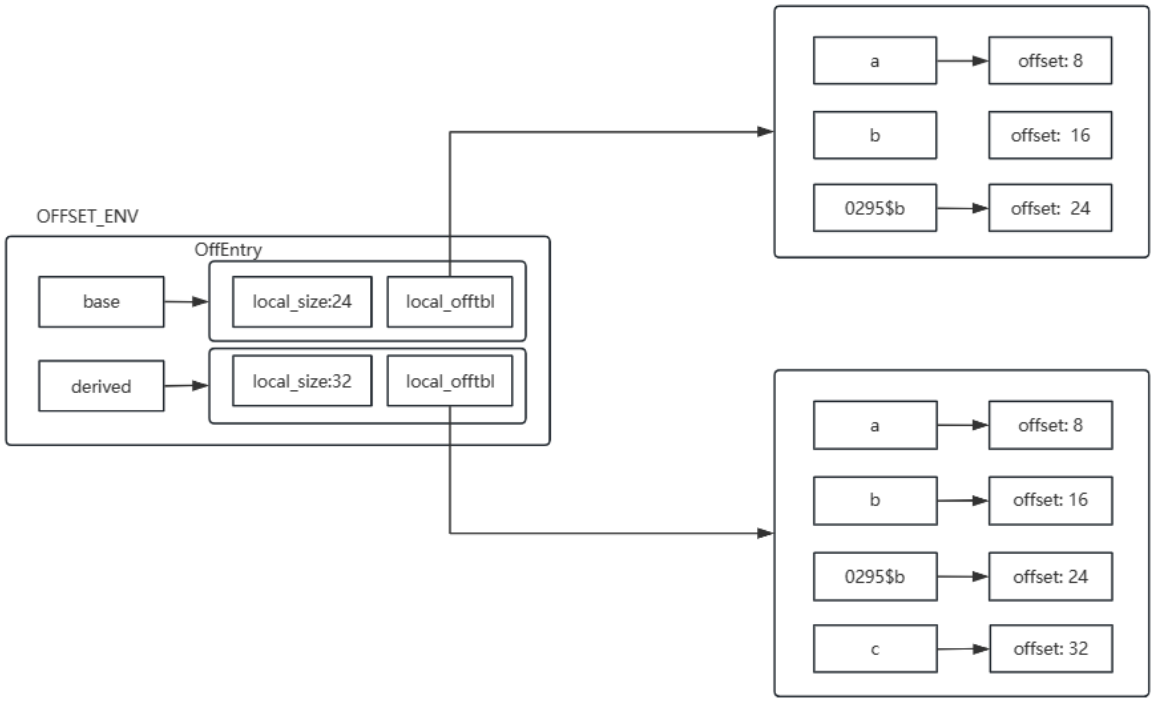


图 2: 样例程序的 OFFSET_ENV 结果

c 是 derived 子类自己的类变量，因此根据“重要性质”，偏移量在所有父类变量之后。在类的方法前面加上一个 0295$，使得重名的变量和方法可以被区分。

2.4 Canonicalization

这是整个前端的最后一个阶段。注意到在我们使用的 Tiger IR+ 中，有 T_Seq 和 T_Eseq 这类只表示结构信息的节点，它们的内容才是真正的一条条需要被执行的语句。为了去除这些不能被真正执行的节点，得到一个线性执行的语句列表，我们需要对 IR 进行 Canonicalization。

Canonicalization 主要分三步进行：第一步是 linearize，得到一个线性语句列表；第二步是根据这个线性语句列表，转换为多个 basic block，每一个 basic block 以一个标签开头，以一条跳转指令结尾；第三步是 trace，将第二步得到的多个 basic block 重新合并为新的线性语句列表，以满足一些性质。我们的实现位于 src/lib/optimizer/canon.c 文件，样例程序经过 Canonicalization 之后的结果见 src/docs/example/example_in_report.4.stm 文件。

2.4.1 linearize

入口函数为：

```
T_stmList C_linearize(T_stm stm);
```

接受一个 IR 节点 stm，调用 linear 函数和 do_stm 函数将它线性化，返回一个链表。

linear 函数递归遍历所有 stm 节点，如果当前节点为 T_Seq，提取其中的左右子节点，重组为线性结构。如此迭代进行，可消除所有的 T_Seq 节点。

do_stm 函数主要是将一个 stm 节点中的 T_Eseq 节点完全消除，具体做法是用一个等价的 T_Seq 节点重新表示每一个 T_Eseq 节点。

2.4.2 basic block

入口函数为：

```
struct C_block C_basicBlocks(T_stmList stmList);
```

接受第一步得到的线性语句列表 stmList，输出多个 basic block。

linearize 的结果中，第一个块一定没有 label 作为开头，因此用一个新的 label 加到开头。使用 mkBlocks 函数递归提取 basic block，如果一个 stmList 不是以 label 开头，加入一个新 label；如果以 label 开头，调用 next 函数寻找当前 basic block 的结束位置。

较难理解的点主要是 next 函数：

```
static C_stmListList next(T_stmList prevstms, T_stmList stms, Temp_label done);
```

如果当前语句节点 stms 的内容为 jump、cjump 或 return，表明达到了一个 basic block 的结尾，调用 mkBlocks 进行后续 basic block 块的提取。

如果直到遇到下一个 basic block 的 label 都没遇到跳转，这说明需要向上一个 basic block 的末尾添加一个“不必要”的跳转语句，跳转到这个 label。之所以称之为不必要，是因为源程序是顺序执行，本来没必要添加这个跳转语句，但是为了满足 basic block 的性质而添加。

此步骤返回一个 C_block 结构的数据，对于样例程序而言，结果如下图：

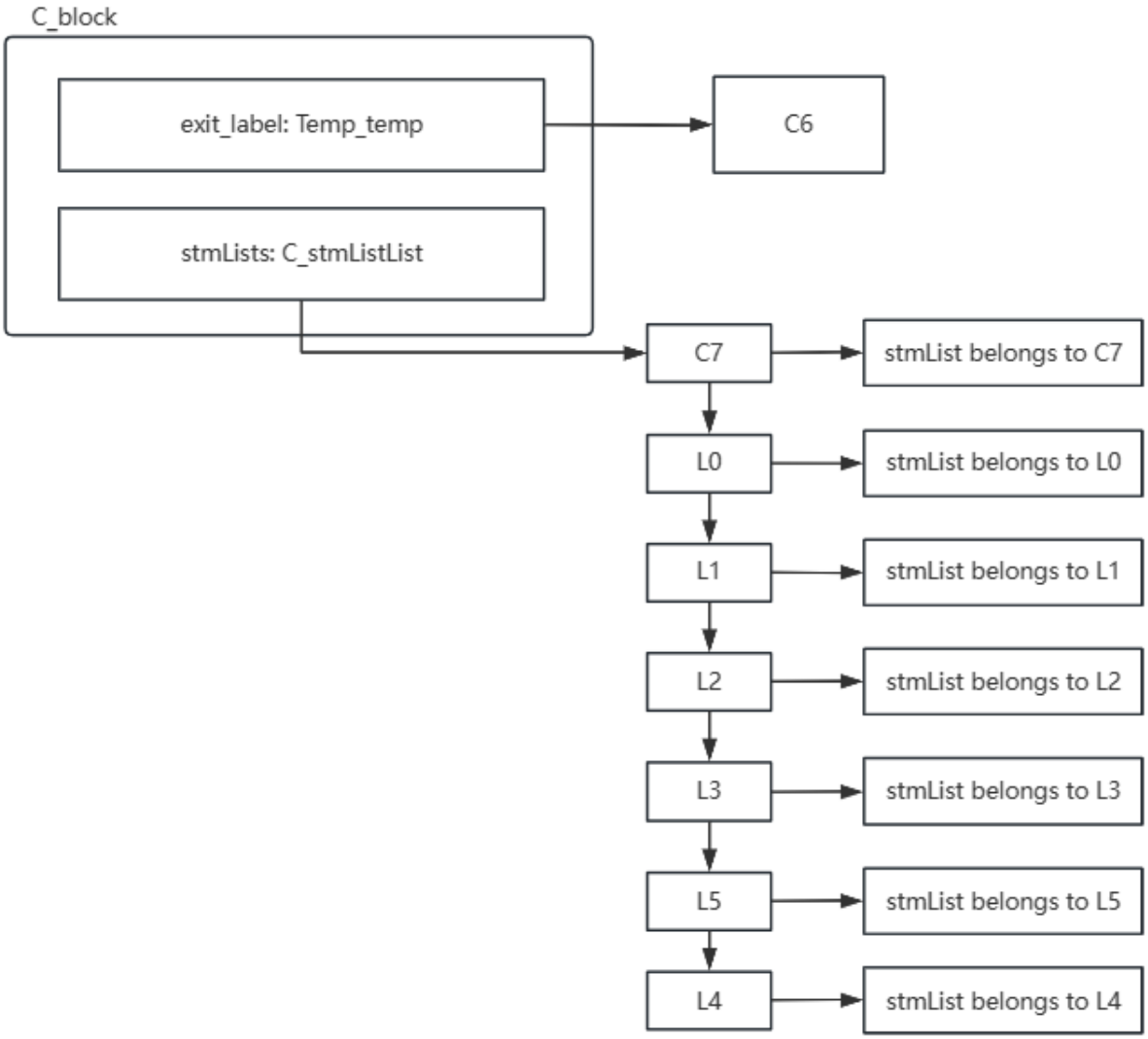


图 3: 样例程序的 C_block 结果

2.4.3 trace

trace 的目的是为了让每一条 cjump 语句，紧跟着的下一个 basic block 都是它的 false label，其入口函数为：

```
T_stmList C_traceSchedule(struct C_block b);
```

该函数首先构建一个表 block_env，每个表项形如 “label->stmList”，记录一个 basic block 的起始标签到它的语句列表首节点的映射。

然后进行 trace 过程。如果没有 basic block，那么添加一个 dummy basic block 作为程序的退出点，它的内容只有一个标签和 ret -1。否则如果是一个存在于 block_env 中的 basic block，调用 trace 函数使其满足性质。

trace 函数中，找到当前 basic block 的 label 和最后一条语句，如果最后一条语句是 jump 语句，并且 label 存在，那么可以进行合并后去除 jump 语句，否则保留 jump 语句；如果最后一条语句是 cjump，根据 true label 和 false label 是否存在，来判断紧跟着的下一个 basic block 到底是哪个 label，如果都不存在，则

修正这条 cjump 语句，强行让它的 false label 为下一个 basic block。

样例程序 trace 之后的结果请参见：src/docs/example/example_in_report.4.stm 中 “——Canonical IR Tree——” 部分

3 Backend

3.1 LLVM Instruction Selection

前端部分向我们提供了 Canonicalization 之后的 Tiger IR+ 形式的中间表示，后端根据这个中间表示，生成目标平台上的汇编代码。本阶段进行 Tiger IR+ 到 LLVMIR 的转换。LLVMIR 不仅可作为一种中间表示，也可以在 LLVM 的环境下运行。

LLVM Instruction Selection 的具体实现在 src/lib/backend/llvm/llvmgen.c 文件。

保存指令选择结果的数据结构为 AS_instr，定义在 src/include/utils/dsa/assem.h 文件中。共有三种 kind，其中 I_OPER 最为常用，包含字符串形式的汇编指令 assem、使用到的源 temps 列表 dst 和目的 temps 列表 src、跳转地点 jumps。assem 只是一个格式字符串，会在 src/lib/utils/dsa/assem.c 文件中定义的 format 函数里完成相应的格式替换，“‘di” 被替换为 dst 中第 i 个 temp 的 name，“‘si” 被替换为 src 中第 i 个 temp 的 name，“‘ji” 被替换为 jumps 中第 i 个 label 的 name（这里 i 为 0 从开始的整数）。

以样例程序 derived$b 函数的 Canonical IR Tree 为例，它将被转化为如图 4所示的数据结构：

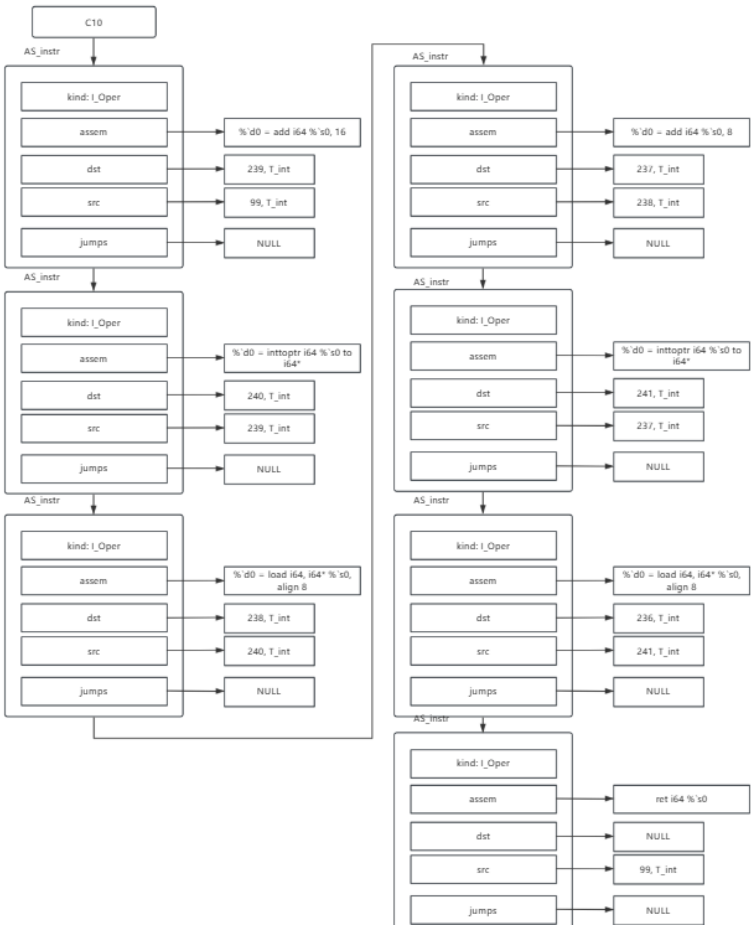


图 4: 样例程序 derived$b 函数的 llvm instruction selection 结果

经过格式化并打印后的结果为：

```
C10:
%r239 = add i64 %r99, 16
%r240 = inttoptr i64 %r239 to i64*
%r238 = load i64, i64* %r240, align 8
%r237 = add i64 %r238, 8
%r241 = inttoptr i64 %r237 to i64*
%r236 = load i64, i64* %r241, align 8
ret i64 %r236
```

LLVM 指令选择的过程遍历 Tiger IR+ 中的 T_funcDeclList，对于每一个 T_funcDecl，调用 llvmprolog 函数获取 llvm 代码中的函数定义，例如样例中 derived 类的 b 函数，返回的数据结构携带的指令为 “define i64 @derived$b(i64 %r99){”；调用 llvmepilog 获取 llvm 代码中的函数结束部分，例如样例的 main 函数，返回的数据结构携带的指令为 “}”。

重点在于函数语句的指令选择，入口函数为：

```
AS_instrList llvmbody(T_stmList stmList);
```

该函数遍历所有 stm，调用 munchStm 函数进行指令选择。其中 munchStm 根据 stm 的类型 kind，输出等价的 llvm 指令来完成一条 T_stm。

另有两个函数 munchExp 和 munchMove。

munchExp 是用于获取表达式的值，例如一个 exp 为 T_CONST，那么将它的值 move 到一个 temp 中，并返回这个 temp。即输入一个 exp，munchExp 返回存有这个 exp 的值的 temp。

munchMove 是为了优化对 T_Move 的指令选择。因为如果在获取需要 move 的值的时候，直接把目的 temp 传递给 munchExp 使用的话，就可以减少一条 “move temp1，temp2” 形式的指令。

对于样例程序指令选择并格式化打印输出后的结果，参见 src/docs/example/example_in_report.5.ins 文件。

3.2 Liveness Analysis

Liveness Analysis 是为了分析数据的流动和活跃范围，以便我们获取丰富的数据流信息和 temp 活跃情况。

活跃分析基于图进行，因此简要介绍一下我们使用的图结构（定义在 src/include/utils/dsa/graph.h 中）。每个图节点为 G_node，其中 mygraph 成员为其所属图、mykey 成员为其在图上的标号、succs 为前驱节点、preds 为后继节点、info 可以是任何与之关联的信息。每个图 G_graph，其中 nodecount 表示节点数、mynodes 表示图上的节点列表，mylast 表示最后一个加入到图中的节点。针对图操作的函数参见代码注释。

3.2.1 flow graph

活跃分析是基于流图进行的，因此我们首先需要根据指令选择之后得到的指令序列，构建它的流图。流图中每一个节点对应一条指令，有向边表示跳转关系。flow graph 相关的代码实现位于 src/lib/optimizer/flowgraph.c 文件。

构建流图的入口函数为：

```
G_graph FG_AssemFlowGraph(AS_instrList il);
```

该函数遍历所有 assem 指令，生成对应的图节点并维护前后跳转关系，同时维护了一个表，用于存放每个节点对应的 def 的 temp 和 use 的 temp 列表，后续通过 FG_def 和 FG_use 获取这些信息。

3.2.2 liveness

有了流图之后，我们根据公式：

$$in[n] = use[n] \cup (out[n] - def[n])$$
$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

EQUATIONS 10.3. Dataflow equations for liveness analysis.

不断迭代，直到达到不动点，即为 liveness 的结果。

该过程的相关实现位于 src/lib/optimizer/liveness.c 文件，入口函数为：

```
G_nodeList Liveness(G_nodeList nodes);
```

其中参数 nodes 为 flow graph 的节点，该函数不断循环调用 LivenessIteration，根据是否产生变化（即是否达到不动点）来决定是否继续循环。LivenessIteration 是对上述公式的一次迭代，并保存或更新信息到表 InOutTable，返回这轮迭代是否导致 liveness 结果产生变化。

liveness 的结果通过 FG_In 和 FG_Out 访问，本质就是查 InOutTable 这个表。

对于样例程序的 liveness 结果，参见 src/docs/example/example_in_report.5.ins 文件。

3.3 Static Single Assignment

静态单赋值（Static Single Assignment，简写为 SSA）是一种指令序列的形式，其满足每个 temp 只被定义一次，可被使用多次。

在这一阶段，我们将可能不满足 SSA 形式的指令序列转换为满足 SSA 形式的指令序列。转换为 SSA 形式的具体实现位于 src/lib/optimizer/ssa.c 文件，入口函数为：

```
AS_instrList AS_instrList_to_SSA(AS_instrList bodyil, G_nodeList lg, G_nodeList bg);
```

其中 bodyil 为函数体指令序列，lg 为具有 liveness 信息的图节点，bg 为 control flow graph。

3.3.1 control flow graph

一个 basic block 内部是线性的，没有跳转，因此没必要直接对它内部的流图节点计算 dominators。我们只需要计算各个 basic block 直接的 dominate 情况，因此我们构建新的图，控制流图（control flow graph，cfg）。控制流图类似于流图，只是图上每个节点代表一个 AS_Block 形式的 basic block，而非一条指令。构建控制流图的入口函数为：

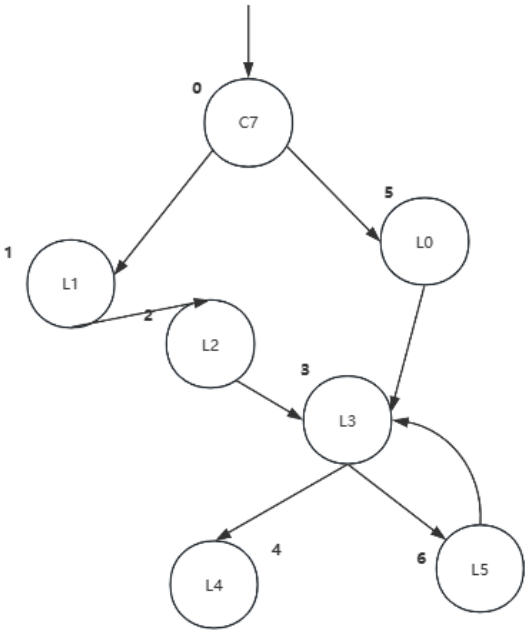
```
G_nodeList Create_bg(AS_blockList bl);
```

该函数也是遍历所有 AS_block，根据每个 basic block 最后一句指令的跳转关系，来添加有向边到图上的节点。

样例程序 main 函数的的 cfg 结果如下（括号中为当前节点，冒号右侧为它的后继）：

C7 (0): 1 5
L1 (1): 2
L2 (2): 3
L3 (3): 4 6
L4 (4):
L0 (5): 2
L5 (6): 3

化成图的形式为：



3.3.2 steps

1. 计算 dominators 和 immediate dominator

计算公式为：

Let $D[n]$ be the set of nodes that dominate n. Then

$$D[s_0] = \{s_0\} \qquad D[n] = \{n\} \cup \left(\bigcap_{p \in \text{pred}[n]} D[p] \right) \qquad \text{for } n \neq s_0$$

此步骤先由

```
static void compute_dominators(G_table dominators);
```

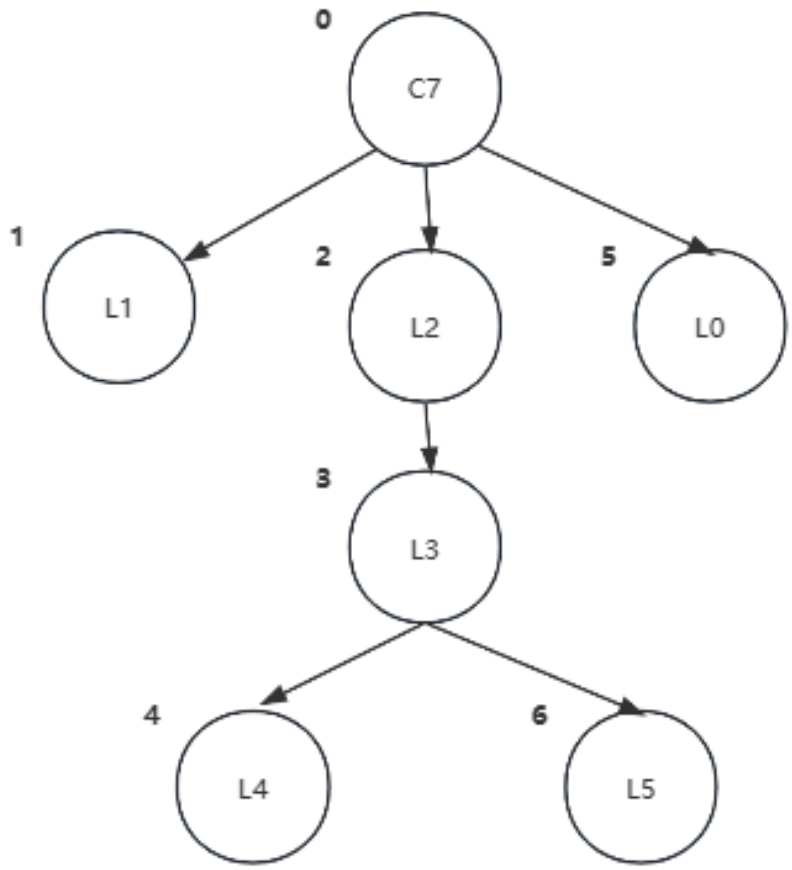
计算出所有节点的 dominators 表。dominators 表，初始化为 “node->all nodes” 的形式（除了 startnode 起始节点外，所有的 node 一开始初始化为 dominate 所有节点，startnode 只 dominate 它自己）。其中 node 是 control flow graph 上的图节点。该函数依据公式不断迭代，直到达到不动点，代码与公式的对应关系参见代码注释。函数结束后，dominators 表记录了每个节点（key）被哪些节点（value）所 dominate。

然后调用 retrive_immediate_dominator 函数构建 dominator tree，以邻接表的形式存放到表格 idom 中，idom 中记录了每个节点（key）的直接 dominator（value）。

对于样例程序，此步的结果如下:

(1): 0
(2): 0
(3): 2
(4): 3
(5): 0
(6): 3

对应的 dominator tree 为:



2. 计算 dominance frontier

算法为：

```
computeDF[n] =
    S ← {}
    for each node y in succ[n]                    This loop computes DF_local[n]
        if idom(y) ≠ n
            S ← S ∪ {y}
    for each child c of n in the dominator tree
        computeDF[c]
        for each element w of DF[c]               This loop computes DF_up[c]
            if n does not dominate w
                S ← S ∪ {w}
    DF[n] ← S
```

该算法有一小错误：”if n does not dominate w” 应为”if n does not strictly dominate w”

我的实现位于 compute_dominance_frontier_impl 函数，compute_dominance_frontier 函数只是一个入口。我的实现与上述伪代码表示的算法的对应关系参见代码注释，在此不作赘述。

对于样例程序的 main 函数，此步结果为：

(0):
(5): 2
(2):
(3): 3
(6): 3
(4):
(1): 2

3. 插入 phi 函数

算法为：

```
Place-φ-Functions =
    for each node n
        for each variable a in A_orig[n]
            defsites[a] ← defsites[a] ∪ {n}
    for each variable a
        W ← defsites[a]
        while W not empty
            remove some node n from W
            for each Y in DF[n]
                if Y ∉ A_φ[n]
                    insert the statement a ← φ(a, a, ..., a) at the top
                        of block Y, where the φ-function has as many
                        arguments as Y has predecessors
                    A_φ[n] ← A_φ[n] ∪ {Y}
                    if Y ∉ A_orig[n]
                        W ← W ∪ {Y}
```

对于最内层的:

```
for each Y in DF[n]:
if Y not in A_phi[n]:
    insert
    A_phi[n] |= Y
    if Y not in A_orig[n]:
        w |= Y
```

应为:

```
for each Y in DF[n]:
if a not in A_phi[Y]:
    insert
    A_phi[Y] |= a
    if a not in A_orig[Y] and a in live-in of Y:
        w |= Y
```

在我的实现中首先初始化表格结构记录信息，包括记录“temp->defsites”的表格、记录算法中 A_ϕ 与 A_{orig} 的表格、记录每个 cfg 节点对应的 live-in 和 live-out 的表格。初始化完毕这些基本信息之后，根据上述伪代码编程实现即可。我的实现与上述算法的对应关系参见代码注释，插入 phi 函数时有多少个前驱结点就需要插入多少个 phi 函数的参数，为后续的重命名预留空间。

4. 重命名 temp

算法为:

在具体实现上，我使用表来实现了算法中的 stack。基于底层的 TAB_table，实现了 Temp_stack，存放“temp->renamed temp”的映射。并支持 beginScope 机制和 endScope 机制，插入一个特殊的 temp（其 num 为-1）来标识 Scope 的开始，控制重命名的有效域。

该步骤的入口函数为 rename_temps，初始化 Temp_stack 为“temp->temp”（即未重命名）的状态，然后调用 rename_temps_impl 函数。

rename_temps_impl 函数从 dominator tree 的根节点开始递归遍历。函数开始前先 beginScope。然后对于当前重命名的 basic block（对应节点为 rename_temps_impl 函数的第一个参数 n）中的所有非 label 指令，如果不是 phi 函数，那么对它 use 中的 temps 重命名；不论是不是 phi 函数，对它 def 中的 temps 重命名为一个新的 temp 并记录到 stack 中。接着按照上述算法，重命名它的后继节点中可能存在的 phi 函数中的变量。最后递归遍历后继节点，退出前 endScope 防止子节点的重命名影响其它子节点。

3.4 RPi Instruction Selection

该阶段也属于指令选择（Instruction Selection），本质与 LLVM Instruction Selection 没有区别。区别在于 LLVM Instruction Selection 的输入为 Tiger IR+ 树，而 RPi Instruction Selection 的输入为 LLVM 指令序列构成的 basic block，因此 RPi Instruction Selection 需要大量识别和处理字符串。具体实现位于 src/lib/backend/arm/armgen.c 文件。

在正式进行指令选择之前，需要去除 phi 函数，因为没有真实的 arm 指令可以实现 phi 函数的功能。我们通过在有 phi 函数的地方，找到这个 basic block 的所有前驱节点，在它的跳转指令之前添加对应的 move 指令，这样就可以用多条指令，实现 phi 函数的效果。

然后类似于 llvmbody，我们根据不同的 llvm 指令，选择与之对应的 arm 指令即可。一些关键点在于:

```
Initialization:
    for each variable a
        Count[a] ← 0
        Stack[a] ← empty
        push 0 onto Stack[a]

Rename(n) =
    for each statement S in block n
        if S is not a φ-function
          for each use of some variable x in S
              i ← top(Stack[x])
              replace the use of x with x_i in S
        for each definition of some variable a in S
            Count[a] ← Count[a] + 1
            i ← Count[a]
            push i onto Stack[a]
            replace definition of a with definition of a_i in S
    for each successor Y of block n,
        Suppose n is the jth predecessor of Y
        for each φ-function in Y
            suppose the jth operand of the φ-function is a
            i ← top(Stack[a])
            replace the jth operand with a_i
    for each child X of n
        Rename(X)
    for each definition of some variable a in the original S
        pop Stack[a]
```

- **立即数处理**：所有立即数都使用两次 ldr 指令加载它的类模式到寄存器，以防止它是不合法的立即数。
- **简单的常量折叠**：有大部分的代码用于识别 LLVM 指令中两个操作数的类型组合方式（register-register, register-immediate, immediate-register, immediate-immediate），并在 immediate-immediate 的情况下进行编译时计算。
- **函数调用的传参**：我们的函数规定最多有三个参数，加上 this 参数，最多需要四个寄存器来传递参数。用 r0-r3 传递 int 型参数，用 s0-s3 传递 float 型参数。为了提取函数调用的参数及其类型，我在 src/lib/backend/arm/armgen.c 中新定义了数据结构 Arg，和 ArgList，作为 extract_args 函数的输出和对 call 指令进行 rpi 指令选择时的输入。每个 Arg 结构的成员为：tp 记录该 Arg 是 T_int 还是 T_float、isConst 记录是立即数还是 temp、联合体 u 用于获取对应的数据。

  需要注意的是，在指令选择这一阶段，我并未对栈帧、caller-save 和 callee-save 寄存器进行处理，而是把这些工作留到了最后的寄存器分配阶段。但是如果非要生成具有一定栈帧结构的指令，只需将入口函数 AS_instrList_to_arm 的参数 save_all_callee 设置为 TRUE 即可，为 FALSE 是不具有栈帧结构的。

3.5 Register Allocation

这一阶段将 rpi 指令转换为可执行的形式。因为对于真实的机器而言，不可能有无限个 temp（即无限个寄存器）供使用，因此我们需要将上一阶段生成的指令中的 temp，映射为真实的寄存器。这一阶段的实现位于 src/lib/backend/arm/regalloc.c 文件。

我实现寄存器分配的算法是 simplify and spill。在我的实现中，r0-r10 为可用整型寄存器，其中 r0-r3 为 caller-save，r4-r7 为 callee-save，r8-r10 专门用于 spill；s0-s31 为可用浮点寄存器，其中 s0-s15 为 caller-save，s16-s28 为 callee-save，s29-s31 专门用于 spill。

3.5.1 Interference Graph

我们的寄存器分配依赖于 Interference Graph，该图是无向图，记录了各个 temp 之间的冲突关系，即如果两个 temp 之间存在一条边，则这两个 temp 不能被分配到同一个寄存器。因此，寄存器分配等价于对 Interference Graph 进行图着色问题。Interference Graph 的图节点关联一个 temp，构建 Interference Graph 的依据为：

1. At any nonmove instruction that *defines* a variable a, where the *live-out* variables are $b_1, \ldots, b_j$, add interference edges $(a, b_1), \ldots, (a, b_j)$.
2. At a move instruction $a \leftarrow c$, where variables $b_1, \ldots, b_j$ are *live-out*, add interference edges $(a, b_1), \ldots, (a, b_j)$ for any b_i that is *not* the same as c.

因此构建 Interference Graph 前需要重新对 rpi 指令进行活跃性分析，利用活跃性分析的结果，调用 Create_ig 函数构建 Interference Graph。过程十分简单，遍历所有的 flowgraph node，满足

```
(!(is_m && tl2->head == use->head)) && (tl2->head != tl1->head)
&& (tl2->head->type == tl1->head->type)
```

条件的就添加一条边 (node(tl1->head), node(tl2->head)) 到图中。第一个条件子句是对 move 指令的特殊处理，第二和第三个条件子句要求没有指向自己的环，并且分离了整型和浮点型。

3.5.2 implementation details

以样例程序中的 main 函数为例，由于 main 函数涉及到的 temp 过多，因此下方展示的结果只挑选了其中具有代表性的几个 temp 来说明我的实现方法。

寄存器分配的入口函数 regalloc 中：

首先调用 simplify_and_spill 函数，该函数接收 Interference Graph，对于所有可用寄存器，设有 k 个，不断删除 Interference Graph 上节点度小于 k 的节点并入栈标记为 color；不能删除之后任选一个非预着色的节点删除并入栈，标记为 spill，直到所有非预着色的节点被删除。删除的过程中记录节点删除时的前驱后继。注意区分 temp 的类型，整型的 temp 和浮点型的 temp 使用不同数量的寄存器。这些标记信息存放于栈 simplify 中。对于样例程序的 main 函数而言，少数结果如下：

```
r252: color
r210: spill
r220: spill
```

然后调用 color_and_actual_spill，该函数根据 simplify_and_spill 产生的栈 simplify，反向弹出栈中的 temp 并根据记录的删除时的前驱后继节点回复图的拓扑结构。如果标记为 color，随便分配一个可用寄存器，并记录到表 color 中。如果标记为 spill，需要先判断它目前的邻居是否用完了所有 k 个寄存器，如果没用完，

也可以分配没用到的一个寄存器给它并记录到表 color 中；否则需要进行 actual spill，在栈上分配一块空间（4 字节）给这个 temp，并记录它的 offset 到表 spill 中。对于样例程序中 main 函数，两个表的结果如下：

color 表的结果（此前被标记为 spill 的 r210 没有发生 actual spill）：

```
r252->r1
r210->r1
```

spill 表的完整结果（包含了 r220）：

```
r226 actual spilled to fake offset 32
r220 actual spilled to fake offset 28
r195 actual spilled to fake offset 24
r193 actual spilled to fake offset 20
r182 actual spilled to fake offset 16
r181 actual spilled to fake offset 12
r180 actual spilled to fake offset 8
r179 actual spilled to fake offset 4
```

构建完 color 和 spill 两个表之后，我们就可以对原始 rpi 指令中用到的 temp 进行替换。对于 color 中的 temp 直接替换即可，对于 spill 中的 temp，使用哪个寄存器，取决于它出现在一条 rpi 指令中从左到右的位置。因为一条 rpi 指令最多用到三个寄存器，因此预留 3 个寄存器专门用于 spill。在每一个发生 actual spill 的 temp 被 def 的指令，加上 str 指令将其保存到栈上对应的 offset 处；在每一个发生 actual spill 的 temp 被 use 的指令之前，加上 ldr 指令，取出栈上对应 offset 处的数据。需要注意的是这里的 offset 相对于 fp 而言向下增长，且暂时没有考虑栈帧大小，所以这是暂时的 offset。

对于上述例子中发生 actual spill 的 r220，def 它的指令为：

```
mov r220, r195
```

actual spill 之后变为（因为 r195 也进行了 actual spill，且位于这条指令从左到右第 2 个寄存器，所以用 r8，r9，r10 中的第 2 个寄存器 r9 来进行 spill）：

```
ldr r9, [fp, #24]
mov r8, r9
str r8, [fp, #28]
```

最后调用函数 build_frame 构建栈帧。在入口保存栈帧和必要的 callee-save 寄存器，在所有 ret 指令之前加入弹出的指令。为什么不需要保存 caller-save 寄存器呢？这是因为我们在进行 interference graph 的计算前，将每一个函数调用指令的 def 列表中，加入了所有 caller-save 寄存器。这样处理之后，在每一条函数调用指令之后的指令中用到的 temp，永远会与 caller-save 寄存器这些预着色节点发生冲突，导致这些函数调用指令之后的指令中用到的 temp 只能被分配到 callee-save 寄存器，因此也就无需要保存 caller-save 寄存器了。处理完栈帧之后需要处理发生 actual spill 的 temp 对应的 offset，将这些 offset 加上 num_used_callee * ARCH_SIZE 并取负（向下增长）即得到真正的相对于 fp 的 offset。如果此时栈没有 8 字节对齐，需要将栈调整到 8 字节对齐，以解决调用 putfloat 时产生的错误。

因为一共保存了 r4-r9，共 ARCH_SIZE * 6 = 24 字节，上述例子现在变为：

```
ldr r9, [fp, #-48]
mov r8, r9
str r8, [fp, #-52]
```

4 Conclusion

以上便是对一个编译器最基本的流程和我对应实现的阐述。关于如何使用我的代码构建出所述编译器并使用，首先需要简单介绍一下代码目录的组织与管理。项目主要使用 CMake 和 Makefile 进行管理，src 是编译器的源代码目录，test 用于存放测试所用的.fmj 源文件。

在代码的根目录下，根据以下指令进行构建和使用：

- **make build**：使用 cmake 与 makefile 对项目进行编译链接，生成编译器的可执行文件 main 到目录 build。
- **make compile**：利用生成的编译器 main，将 test 文件夹中的所有.fmj 文件进行编译，并产生对应的中间输出，其中.6.ssa 文件是可在 llvm 环境下运行的指令文件，.8.s 文件是最终的可在 qemu-arm 上运行的汇编代码。
- **make run-llvm**：重新编译，并使用 llvm 的环境运行 ssa 形式的 llvm 指令文件。
- **make run-rpi**：重新编译，并使用 qemu-arm 环境运行汇编.8.s 汇编文件之后的机器代码。

也可以直接运行 main 函数，其提供.fmj 文件的路径，与 make compile 产生相同的结果。

总结而言，通过本学期的课程，我掌握了基本的编译技术及其实现方式，也有思考将课堂所学的编译知识应用到其他领域的场景，例如利用寄存器分配的思想来优化深度学习训练框架中的显存分配与使用（将显存分块视为一个个寄存器，计算图上的 tensor 节点视为一个个 temp）。这是一门让我受益匪浅的课程，不仅锻炼了我的代码实践能力，还让我掌握了更多样地分析计算机世界中问题的思想与方法。

感谢参阅！