编译Final实验报告

引言

本报告记录了FDMJ语言编译器的设计与实现，目标读者是还没有修过编译课的计算机专业学生。

报告主要介绍了项目实现的编译器各个阶段的实现细节，包括词法和语法分析、类型检查、翻译成IR、IR的规范化、LLVM指令选择、SSA、RPi指令选择、活跃性分析和寄存器分配。然后给出具体示例对各阶段工作进行更详细的解释。最后介绍编译器使用方式和对应的输出文件。

编译的各阶段

词法和语法分析

本阶段的代码在 `lexer.lex` 和 `parser.yacc` 中。

本阶段的目标是生成AST (抽象语法树)，FDMJ的AST定义在 `fdmjast.h/c` 中。

本阶段的工作主要分成两个步骤：

1. 使用Lex工具进行**词法分析**：在 `lexer.lex` 中编写词法解析规则，工具自动生成 `lexer.c` 用于词法分析，即将源码匹配定义好的token，将token的值填充进 `yylval`，并将token返回给Yacc。
   - **处理注释**：FDMJ中的注释有两种 `// ...` 和 `/* ... */`，分别对应Lex中的 `COMMENT1`，`COMMENT2` 状态，在识别注释后转到对应状态，当遇到换行符或 `*/` 后退出注释状态进入 `INITIAL` 状态
   - **解析token**：直接定义解析规则即可

     > 注意需要将匹配 `ID` 的规则放在所有匹配关键字的规则之后，即 `ID` 不能和关键字重名。

   - 遇到非法token，会打印错误信息 (行号和列号) 并跳过继续解析

     ```
     <INITIAL>. {
       printf("Illegal input \"%c\" at line %d, position %d\n", yytext[0], line, pos);
       pos++;
     }
     ```

2. 使用Yacc工具进行**语法分析**：在 `parser.yacc` 中定义precedence，编写语法解析规则，在代码部分**生成AST**。
   - **定义yylval**：根据AST声明所有的语义值类型，注意到token `ID` 在大多数时候只需要其字符串，但在 `Exp` 的生成式中需要其position信息，故其 `yylval` 设为 `IdExp` 类型。
   - **定义token**：在parser declarations部分定义终结符 (词法分析中的token)、非终结符和起始符。
   - **生成AST**：在规则部分，编写语法对应的产生式，并且将每条规则 (即产生式) 对应的动作部分构造对应的语义值类型对象并返回。
   - **优先级**：需要编写优先级规则来定义各种运算符的优先级，特别是对符号的处理，需要利用 `%prec UMINUS` 来提升优先级。

- **danling else问题**：当yacc的格局为如下所示时

```
Stm -> IF ( Exp ) Stm . ELSE Stm
Stm -> IF ( Exp ) Stm .
```

会产生冲突，此时应该让else和最近的if配对，即应该优先移进而非规约，在Yacc的实现中让else所在的规则拥有更高的优先级：

```
%precedence THEN
%precedence ELSE

STM: IF '(' EXP ')' STM ELSE STM {
  $$ = A_IfStm($1, $3, $5, $7);
} | IF '(' EXP ')' STM %prec THEN {
  $$ = A_IfStm($1, $3, $5, NULL);
} ...
```

此外，在Yacc部分进行了部分**错误处理**：当遇到错误时，直接跳过这一行继续解析；解决的错误处理主要是针对`VarDeclList, StmList, ClassDeclList, MethodDeclList`，若list中某个出错可以跳过继续处理下一个。

错误处理的实现是通过在产生式中添加识别错误模式的产生式完成的，以`VarDecl`为例：

```
// example
VarDecl: ...
  | INT error ';' {
  yyerrok;
  $$ = NULL;
} | FLOAT error ';' {
  yyerrok;
  $$ = NULL;
} | CLASS error ';' {
  yyerrok;
  $$ = NULL;
};

VAR_DECL_LIST: /* empty */ {
  $$ = NULL;
} | VAR_DECL VAR_DECL_LIST {
  if ($1 != NULL) {
    $$ = A_VarDeclList($1, $2);
  } else {
    $$ = $2;
  }
};
```

如果某个变量声明错误就会返回NULL，从而跳过继续处理下一个声明，这里`yyerrok`的作用是避免存在紧邻的error导致错误信息不输出。

错误信息的输出通过`yyerror`函数实现，可以输出错误的行号和列号，并用清晰的方式展示 (^指向出错处)。

```
void yyerror(char *s) {
  extern int pos, line;
  extern char *yytext;
  extern int yyleng;
  extern char linebuf[2000];
  fprintf(stderr, "line %d,%d: %s near %s:\n%s\n", line, pos - yyleng, s, yytext,
linebuf);
  fprintf(stderr, "%*s\n", pos - yyleng, "^");
}
```

类型检查和翻译成IR

本阶段的代码在 `semant.h/c` 和 `translate.h/c` 中。

本阶段的目标是进行类型检查，并同时将AST翻译成IR，其中IR的抽象程度更高，独立于源语言和机器语言的细节。本项目使用的IR定义在 `tigerirp.h/c` 中，分为函数(`T_funcDecl`，后面处理都以函数为单位)、语句(`T_stm`)和表达式(`T_exp`)。模块设计思路如下：

- 在 `translate.h/c` 进行翻译工作；
- 在 `semant.h/c` 模块进行类型检查，充分利用类型检查的爬树过程，即在返回时调用对应的翻译函数，可以做到同时完成类型检查翻译的工作。

其中，使用 `Tr_exp` 连接semant和translate两个模块，分别用 `Tr_ex` 包装有返回值的表达式、`Tr_nx` 包装无返回值的语句、`Cx` 包装条件语句。

数据结构

1. 表达式返回值类型 `expty`：在类型检查阶段，检查表达式后需要返回其类型和是否是左值；同时，翻译时也需要返回表达式结果 `Tr_exp`。

   ```
   typedef struct expty_ *expty;
   struct expty_ {
     Tr_exp exp;       // translate result
     Ty_ty value;      // type
     bool location;    // if it is a left value
   };
   ```

2. 表结构：在类型检查阶段，需要构建变量表、类表和方法表记录信息；在翻译阶段，需要记录类变量和方法的偏移表。这里都用 `symbol.h` 的 `S_table` 实现。
   - 类型检查阶段的表结构存储 `S_symbol` 到 `E_enventry`(见 `env.h/c`)的映射，其中 `E_enventry` 有3种类型：
     - `E_varEntry`：变量声明，主要存储变量类型和对应分配的 `Temp_temp`。
     - `E_classEntry`：类声明，主要记录当前类的变量声明表 `vtbl` 和方法声明表 `mtbl`，同时记录父类名。

        - `E_methodEntry`：类方法声明，主要记录所属的类名、返回值类型和参数列表类型。
    - 翻译阶段的偏移表直接用偏移量作为表项。
3. 栈结构：对于 `continue` 和 `break` 语句，在类型检查阶段需要检测是否合法 (是否被while循环包裹)，在翻译阶段需要确定跳转到哪个label，故需要记录所在while循环的两个label。

```
typedef struct loopstack_ *loopstack;
struct loopstack_ {
  Temp_label whiletest; // while label on top
  Temp_label whileend;  // while end label
  loopstack next;
};
```

类型检查

类型检查阶段将需要用到的表声明为静态全局变量，有以下几种：

1. **cenv**：类环境，存储类名到类信息的映射，其中类信息中包括2个重要的表
    - **vtbl**：类变量表，存储类变量声明
    - **mtbl**：类方法表，存储类方法声明
2. **venv**：变量环境，在对类方法和main method进行类型检查时存储声明的变量 (和参数)
3. **MAIN_CLASS**：一个dummy class的名字，主要作用有2个：
    - 在初始化 `cenv` 时，用作所有无extend的类的父类
    - 在检查主方法时，用作 `curClassId`，从而区分主方法和其他方法
4. **curClassId**：目前正在检查的类名，在预处理阶段和类型检查阶段使用
5. **curMethodId**：目前正在检查的方法名，在预处理阶段和类型检查阶段使用

具体流程如下：

1. **初始阶段**：声明需要用到的静态变量
2. **预处理阶段**：对所有的类声明遍历2次，其中
    - **第一次**遍历：**初始化** `cenv`
        1. 记录父类，即extends关系
        2. 将所有变量插入表 `vtbl` 中，同时检查是否有**类变量重定义**
        3. 将所有方法插入表 `mtbl` 中，同时检查是否有**类方法重定义**
            - 记录返回值类型
            - 记录参数类型：遍历参数列表，在返回参数类型的同时检查**参数重定义**错误
                - 如何检查重定义？先插入 `venv` 中，通过查表检查是否重定义，检查完毕后删除所有插入的参数，这可以通过用 `S_beginScope(venv)` 和 `S_endScope(venv)` 包裹做到。
    - **第二次**遍历：**检测继承环**，并将父类的所有类变量和类方法复制到子类的相应表中
        - 检测算法：将继承关系建模为有向图，利用三色法 (见 `env.h` 的 `E_status`) 检测有向图中是否有环

- 每个节点都有3种状态：
    1. `E_transInit`：未处理，即一开始所有节点都是 `E_transInit`
    2. `E_transFind`：正在处理该节点，但还未处理完它的所有父类
    3. `E_transFill`：该节点及其所有父类都已经处理完
- 若当前节点和其父类都是 `E_transFind`，说明出现继承环，同时可以检测**父类未定义**错误
- 若节点状态是 `E_transFill`，说明已经处理完成，可以复制父类的 `vtbl` 和 `mtbl`
    - 复制过程中可检测**类变量与父类重名**错误
    - 复制过程中可检测继承过程中**同名类方法的函数签名 (返回值类型+参数类型) 不严格相同**错误

3. **类型检查阶段**：检查类和主方法
    1. 遍历类：检查类变量和类方法
        - **检查变量类型未定义**：对类型为class的变量，查询cenv看这个class是否存在；该过程对类变量、返回值、参数和方法内的变量声明都进行一遍。
        - 在检查类方法的Stm和Exp时的类型检查较繁杂，无非是各种类型匹配问题和左值问题，不再赘述。
    2. 检查主方法：类似上面类方法的检测

注意，上面的过程中只要检测到一条语义错误，编译程序就会终止并输出报错信息。

翻译

整个翻译过程只需要将AST中结构转换成对应IR中的结构，下面是需要注意的几点情况：

1. **类的消除**：翻译阶段将类的语义消除，这是通过Unified Object Record策略实现的：即为所有类开辟相同大小的空间，并且确定类变量和方法对应的偏移量。其中确定偏移量的工作是在**类型检查的预处理阶段**确定的，即每遇到一个新的类方法或变量就更新所需空间大小，并在表 `varoff` 或表 `methoff` 中记录变量名或方法名到偏移量的映射。
2. **条件语句**：将条件表达式解包成 `Cx`，然后利用 `doPatch` 对 `Cx` 的真假 `patchList` 进行填充，最后利用 `Tr_Nx` 重新包装成 `Tr_exp`。
3. **逻辑运算表达式**：由于短路逻辑的存在，应该将表达式先转换成对应的if语句，再利用上面的流程处理。
4. **循环的跳出**：维护全局栈用于保存当前while循环的进入和退出label，在遇到 `continue` 和 `break` 语句时只需获取栈顶的label。
5. **申请数组变量**：申请长度为原长度+1的数组，在-1位置处存储数组长度。
6. **申请类对象**：新建一个temp，为其申请 `globaloff` 的空间，分别遍历所属类对应的 `vtbl` 和 `mtbl`，将变量初始值 (如果有) 和方法地址根据偏移表存入对应地址。

IR规范化

本阶段的代码在 `canon.h/c` 中。

Canonicalization的目标是去除IR中的所有SEQ和ESEQ节点，建立基本块并对其重新整理。具体步骤如下：

1. `C_linearize`：利用书上的算法消除ESEQ节点，做等价转换；同时将CALL节点移到顶层，因为汇编语言不允许将其作为子表达式；最后将所有节点线性化为 $SEQ(s_1, SEQ(s_2, \cdots, SEQ(s_{n-1}, s_n)\cdots))$ 的形式。
2. `C_basicBlocks`：遍历IR树，发现一个label就开始一个基本块，发现一个跳转语句就结束一个基本块；如此每个基本块的第一条语句为label，最后一条语句是跳转或返回。
3. `C_traceSchedule`：对基本块排序，使得每个CJUMP后面紧跟着对应的false基本块；同时增加一个exit label，在该基本块中只有 `return -1` 一条语句，以应对空块或源码中无返回语句的情况，这是因为在汇编中必须有返回语句以执行epilog。
4. 在对基本块进行重排后，需要再次调用 `C_basicBlocks` 重建基本块。

LLVM指令选择

本阶段的代码在 `llvmgen.h/c` 中。

LLVM指令选择阶段的工作是将IR翻译成LLVM指令，选择的**tile**如下：

- **T_stm**

1. Label

树型	指令	解释
`T_Label(C0)`	`C0:`	block开头

2. Jump

树型	指令	解释
`T_Jump(L0)`	`br label L0`	直接跳转

3. Cjump

树型	指令	解释
`T_Cjump(op, left, right, t, f)` where type=T_int	`%L = icmp op i64 left, right` `br i1 %L, label t, label f`	`op` for `eq`, `ne`, `slt`, `sgt`, `sle`, `sge`
`T_Cjump(op, left, right, t, f)` where type=T_float	`%L = fcmp op double left, right` `br i1 %L, label t, label f`	`op` for `oeq`, `one`, `olt`, `ogt`, `ole`, `oge`

4. Move：值得注意的是，如果Move的两个exp都是temp，应该用 `AS_Move`

树型	指令	解释
`T_Move(T_Mem(left), right)`	`%L = inttoptr i64 left to i64*` `store i64 (double) src, i64* %L`	这里 `right` 是调用munchExp后的结果
`T_Move(T_Temp(left), T_Temp(right)`	`left = add (fadd) i64 (double) right, 0`	这里要用 `AS_Move`
`src` 为其他类型	根据 `src` 决定对应的指令	目的寄存器指定为 `dst`

5. Exp

树型	指令	解释
`T_Exp(exp)`	根据 `exp` 决定对应的指令	直接调用munchExp，不关心目的寄存器

6. Return

树型	指令
`T_Return(exp)`	`ret i64 (double) exp`

- **T_exp**

1. BinOp

树型	指令	解释
`T_Binop(op, left, right)` where type=T_int	`dst = op i64 left, right`	`op` for `add`, `sub`, `mul`, `sdiv`
`T_Binop(op, left, right)` where type=T_float	`dst = op double left, right`	`op` for `fadd`, `fsub`, `fmul`, `fdiv`

2. Mem：在Move中已经对store的情形做了特判，这里的Mem只可能对应load

树型	指令
`T_Mem(exp)`	`%L = inttoptr i64 exp to i64*` `dst = load i64 (double), i64* %L`

3. Temp：直接返回该temp

4. Name

树型	指令	解释

树型	指令	解释
`T_Name(label)`	`dst = ptrtoint i64* @label to i64`	将method name变成地址

5. Const

树型	指令
`T_Const(num)`	`dst = add (fadd) i64 (double) num, 0 (0.0)`

6. Call

树型	指令	解释
`T_Call(id, obj, args)`	`%L = inttoptr i64 obj to i64*` `dst = call ty %L(args)`	`args` 通过调用munchArgs处理

7. ExtCall

树型	指令	解释
`malloc(size)`	`%L = call i64* malloc(i64 size)` `dst = prttoint i64* %L to i64`	`size` 为常量时直接替换
`getint()`	`dst = call i64 @getint()`	
`getch()`	`dst = call i64 @getch()`	
`getfloat()`	`dst = call double @getfloat()`	
`getarray(arr)`	`%L = inttoptr i64 arr to i64*` `dst = call i64 @getarray(i64* %L)`	`size` 为常量时直接替换
`getfarray(arr)`	`%L = inttoptr i64 arr to i64*` `dst = call i64 @getfarray(i64* %L)`	`size` 为常量时直接替换
`putint(num)`	`call void @putint(i64 num)`	`num` 为常量时直接替换
`putch(ch)`	`call void @putch(i64 ch)`	`ch` 为常量时直接替换
`putfloat(num)`	`call void @putfloat(double num)`	`num` 为常量时直接替换
`putarray(len, arr)`	`%L = inttoptr i64 arr to i64*` `call void @putarray(i64 len, i64* %L)`	`len` 为常量时直接替换
`putfarray(len, arr)`	`%L = inttoptr i64 arr to i64*` `call void @putfarray(i64 len, i64* %L)`	`len` 为常量时直接替换
`starttime()`	`call void @starttime()`	

树型	指令	解释
`stoptime()`	`call void @stoptime()`	

此外，每条指令有对应的 `dst, src, targets`，这些都是和指令的形式一一对应的，无需赘述。

tile过程依然按爬树方式进行，没有专门做tile匹配工作，具体过程不再赘述，只有以下几点需要注意：

1. 处理exp的返回值：用 `expres` 表示，即结果可以是temp或const。

```
typedef struct expres_ *expres;
struct expres_ {
  enum { res_const, res_temp } kind;
  T_type type;
  union {
    int i;
    double f;
    Temp_temp t;
  } u;
};
```

   这样做的目的是

   - 在处理 `ConstExp` 时，可以直接返回常数值
   - 当LLVM指令的操作数为常数时，可以直接写死在指令中，无需再申请一个temp中转
   - 让tiling过程更简单：无需特判tile，直接判断 `expres` 是否是常数即可优化

2. 调用 `munchExp` 时传入参数 `dst`：
   - 直接传入 `dst` 的好处是不用新申请一个temp返回，而是可以直接用 `dst` 作为目的temp输出指令，这里的 `dst` 来自上级 `T_stm` 的信息
   - 如果传入的 `dst` 为空，才会申请一个新的temp作为 `dst` 并返回

另外，每个函数需要生成prolog和epilog，由于没有栈管理，只需要生成如下内容：

```
// prolog
define i64 @main {
    ...
// epilog
}
```

SSA

本阶段的代码在 `ssa.h/c` 中。

SSA可以让代码优化变得十分高效，同时也是LLVM程序可运行的前提。该阶段的算法完全参考虎书，对算法的细节不再赘述。

由于算法中涉及很多集合操作 (大多是关于CFG节点的)，故用bitmap作为集合的表示方式，具体实现见 `utils/dsa/bitmap.h/c`。

同时，算法也用到很多信息表，这些表大多数以block为索引，故将所有信息聚合到一个结构体 `SSA_block_info` 中，并直接用数组存储所有节点的信息 (即AoS):

```
typedef struct SSA_block_info_ *SSA_block_info;
struct SSA_block_info_ {
  G_node mynode;
  bitmap doms;
  int idom;
  blockIdList dom_tree_children;
  blockIdList dom_frontiers;
  Temp_tempList orig_vars;
  Temp_tempList phi_vars;
  instrInfoList instrInfos;
  Temp_tempList blockIn;
  Temp_tempList blockOut;
};

static SSA_block_info *blockInfoEnv;
```

其中:

- `mynode`: `bg` 中对应的节点
- `doms`: 存储必经节点
- `idom`: 直接必经节点
- `dom_tree_children`: 存储必经节点树的子节点
- `dom_frontiers`: 存储必经节点边界
- `orig_vars`: 存储未进行SSA之前的块中所有被def的变量集合
- `phi_vars`: 存储被插入该块的所有 ϕ 函数对应的原始变量集合
- `instrInfos`: 存储该块最终的指令序列，对每条指令还记录了对应的原始的 `orig_def` (在变量重命名时用到)
- `blockIn` 和 `blockOut`: 存储该块对应的Live-In/Live-Out变量集合

在插入 ϕ 函数时，需要为每个变量初始化 $defsites(a)$；变量重命名时，需要为每个变量维护栈存储最新版本。故用 `TAB_table` 存储变量到这些信息的映射关系:

```
typedef struct SSA_var_info_ *SSA_var_info;
struct SSA_var_info_ {
  bitmap defsites;
  var_stack stack;
};

static TAB_table varInfoEnv;
```

实现SSA形式的转换分为以下几个步骤:

1. 得到CFG的**逆后序**结果。

   > 见 `sort_bg_in_RPO` 函数。

2. 计算**必经节点**：根据方程 $D[u] = \{u\} \bigcup (\bigcap_{v \in \text{pred}[n]} D[v]), u \neq s_0$ 不断迭代直到达到不动点，求解所有节点的必经节点集合；其中初始化阶段，应该令 $D[s_0] = \{s_0\}$，而其他所有节点的支配集都初始化为所有节点，并且按照步骤1得到的逆后序进行遍历从而加速算法。

   > 见 `compute_bg_doms` 函数。

3. 求解**必经节点树**：由于 $idom(u) = v$ 等价于存在 $v \in D[u](v \neq u)$，使得 $D[u] \setminus D[v] = \{u\}$，根据该方程求解直接必经节点，从而构建必经节点树 (创建所有 $idom(u) \to u$ 的边)。

   > 见 `compute_bg_idoms` 和 `construct_bg_dom_tree` 函数。

4. 求解**必经节点边界**：根据虎书19.1.2节的伪代码计算，在实现过程中先用bitmap进行运算，最后转成DF的形式。

   > 见 `compute_bg_dom_frontiers` 函数

5. **插入 ϕ 函数**：参考虎书算法19-1，同时做一点修正和改进，即如果变量a在必经节点边界处不是Live-In的，就无需插入对应的 *ϕ 函数。类似地，集合运算采用bitmap的形式。

   > 见 `compute_phi_functions` 函数。

6. **变量重命名**：参考虎书算法19-2实现。

   > 见 `rename_var` 函数。

注意，SSA的构建需要用到活跃分析结果和基于block的控制流图，故在进行SSA之前首先进行**活跃分析**。

RPi指令选择

本阶段的代码在 `armgen.h/c` 中。

RPI指令选择部分的目标是将转换为SSA形式的LLVM指令翻译成ARM指令，主要有两个步骤：消除 ϕ 函数和ARM指令生成。

1. **ϕ 函数的消除**：利用critical edge splitting算法消除 ϕ 函数。

具体来说，对每个 ϕ 函数，设其所在块为 B，前驱为 B_i (即 ϕ 函数形式为 $a_0 = \phi(B_1 : a_1, \cdots, B_n : a_n)$)，若 B 有多个前驱且 B_i 有多个后继，则创建一个新块 B_i'，并用 $B_i \to B_i'$，$B_i' \to B$ 来替换边 $B_i \to B$，在 B_i' 中插入 $a_0 \leftarrow a_i$；否则直接在 B_i 中插入 $a_0 \leftarrow a_i$。

在实际实现时，为每个块 B 维护一个原始前驱块label到splitting完成后的实际前驱块的映射 (`S_table parallelCopyTab`)，由此在消除 ϕ 函数时就可以直接根据原来存储的label找到对应插入Move指令的前驱块。

2. **重排block**：由于critical edge splitting有可能引入新的block，故需要重新trace以确保每个cjump后面紧跟的是false标签。算法完全参考Canonicalization部分，即虎书8.2.3节的步骤。

以上部分也放在 `ssa.h/c` 中。

3. **ARM指令生成**

- prologue
    1. 解析函数名，输出相应的label
    2. 将 `old fp` 压栈，并设置当前栈帧 (`mov fp, sp`)
    3. 将 `lr` 压栈保存
    4. 将callee saved registers压栈保存：暂时输出指令 `push {r4, r5, r6, r7, r8, r9, r10}`，实际需要保存哪些寄存器在寄存器分配阶段决定并替换为新的指令
    5. 根据ARM调用公约，将函数参数移到对应的temp中
        - 首先从寄存器中获取 (整型是 `r0-r3`，浮点数是 `s0-s15`)
        - 如果寄存器不够用，剩余的参数从栈中获取 (`ldr temp, [fp, #i]`, $i = 4, 8, \cdots$)
- epilogue：没有需要做的
- body：主要有以下几点需要注意
    1. call
        - 将参数从temp移到对应的寄存器中，顺序和之前prologue时获取参数的顺序一致
        - 用 `blx` 跳转到寄存器存储的对应位置，注意**将 `lr` 和所有caller saved寄存器写入 `dst` 部分，将上面转移参数用到的所有寄存器写入 `src` 部分**
        - 如果该函数有返回值，将该返回值从 `r0/s0` Move到对应的temp中
    2. ret
        - 将返回值Move到 `r0/s0` 中
        - 将 `sp` 设置为 `fp - 32`，并将栈中保存的callee saved registers弹出

          ```
          emit(AS_Oper("\tsub sp, fp, #32", NULL, NULL, NULL));
          emit(AS_Oper("\tpop {r4, r5, r6, r7, r8, r9, r10}", NULL, NULL, NULL));
          ```

          实际需要弹出哪些寄存器，`sp` 相对 `fp` 的偏移量是多大在寄存器分配阶段决定并替换为新的指令
        - 弹出 `lr`
        - 弹出 `old fp`
        - 用 `bx lr` 跳转
    3. cjump
        - 之前已经确保每个cjump后面紧跟的是false标签，可以直接翻译成 `cmp + br` (跳转到true标签)
        - 注意生成 `br` 指令时，**AS_Target要完整保留** (即true和false的label都保留)，否则后序无法得到正确的冲突图
    4. 立即数的处理
        - 整型
            - 通用：分别将该立即数的低16位和高16位用 `mov` 和 `movt` 赋值即可，利用以下trick，输入对应的整型或浮点型就可获取其低16位和高16位

```c
union uf {
  int i;
  float f;
  unsigned int u;
};
```

    - 对于Operand 2，先判断是否是 `<imm8m>`，如果是就直接编码到 `assem` 字符串中，否则应用上面的通用流程
- 负数 (整型)：可以利用 `mvn` 指令取反得到
    - 负数的反码应该是其补码减1
    - 若 `x` 为负数，先判断 `-x-1` 是否在 `<imm8m>` 的范围内，若是则可以利用 `mvn temp, -x-1` 指令得到立即数
    - 否则直接应用通用流程
- 浮点型：直接应用上面的通用流程

活跃分析

本阶段的代码在 `flowgraph.h/c`，`liveness.h/c`，`ig.h/c` 和 `bg.h/c` 中。

本阶段的工作是对前一阶段生成的RPi指令重新进行活跃分析，包括生成控制流图、活跃分析和构建冲突图。block控制流图在之前生成，但也在此重新陈述。

1. 控制流图：根据指令的 `AS_targets` 确定每条指令能跳转到的指令 (若为空说明跳转到下一条指令)，从而构建出CFG。
2. 活跃分析：根据方程 $in[n] = use[n] \bigcup (out[n] - def[n])$, $out[n] = \bigcup_{s \in succ[n]} in[s]$ 迭代计算直到不动点，其中 $def[n]$ 即指令中的 `dst`，$use[n]$ 即为指令中的 `src`。
3. 冲突图：利用活跃分析的结果，非MOVE指令的 `FG_def` 和 `FG_out` 冲突，MOVE指令的 `FG_def` 和 `FG_out - FG_use` 冲突，直接根据定义构建。

以上的图都是指令级别的。

4. block控制流图：在构建之前需要将程序转换为block链，只需根据block的后继构建CFG，最后删除没有前驱的block。

寄存器分配

本阶段的代码在 `regalloc.h/c` 中。

本阶段的目标是对RPi指令选择阶段生成的指令做寄存器分配。为简化分配过程，仅使用简化和溢出两个步骤。

可供分配的ARM寄存器共10个，为 `r0-r3, lr, r4-r8`，同时为spill预留 `r9` 和 `r10` (callee saved，需要压栈保存)；可供分配的浮点寄存器共30个，为 `s0-s13, s16-s31`，同时为spill预留 `s14` 和 `s15` (caller saved，无需压栈保存)。这里寄存器的顺序即为分配时的优先级。

本阶段主要分成两个模块 (或步骤) 完成：对图着色 (COL模块) 和处理溢出并修改指令 (RA模块)。

着色阶段的工作是确定被简化节点的颜色和所有溢出节点，用到的数据结构如下：

1. 着色的结果：包括着色表、溢出变量集合，还有两个辅助变量用于记录简化过程中用到的寄存器数 (以便后续确认栈大小)。

```
typedef struct COL_result_ *COL_result;
struct COL_result_ {
  Temp_map coloring;
  Temp_tempList spills;
  int maxRegId;
  int maxFloatRegId;
};
```

2. 变量信息表：记录每个节点 (变量) 在冲突图中的度，还有对应的寄存器id。

```
typedef struct COL_tempInfo_ *COL_tempInfo;
struct COL_tempInfo_ {
  G_node node;
  int degree;
  int regId;
};

static TAB_table colEnv = NULL;
```

3. `simplyfyWorklist`：记录所有待简化的节点，在初始化时插入所有度小于 k 的节点。
4. `spillWorklist`：记录所有待溢出的节点，在初始化时插入所有度 $\geq k$ 的节点。
5. `selectStack`：记录从图中删除的简化节点的栈。

其中3、4、5都采用 `G_nodeList` 实现。

着色过程整体流程大致如下：

1. 初始化：将所有预着色节点放入变量着色表 `coloring` 里，并初始化以上工作表。
2. 循环进行简化和溢出过程，直到两个工作表都为空：
   - **简化**：从工作表中取出一个节点 n 压入 `selectStack` 栈中，同时需要减少 n 的所有邻接节点的度数。
   - **溢出**：从工作表中选取度数最大的节点 m 删除，直接将其加入结果的溢出集合 `res->spill` 中 (real spill)，类似地减少所有邻接节点的度数。
3. **着色**：不断从 `selectStack` 中弹出待简化的节点 n
   - 遍历其所有的邻接节点，按优先级找到第一个可用的寄存器，分配给 n
   - 将 n 到该寄存器的映射关系存储在 `res->coloring` 中
   - 维护用到的最大寄存器id值
4. 清空工作表和简化栈

分配阶段的工作是处理溢出变量，并确定栈大小，整体流程如下：

1. 计算栈大小和溢出变量对应的偏移量：
    1. 由于只需要real spill，遍历着色阶段得到的溢出集合，将对应的偏移量 (按顺序) 记录到 `spillMap` 中
    2. 计算用到的预留ARM寄存器数
    3. 计算之前简化过程中，用到的callee saved寄存器
2. 遍历指令序列，修改指令：
    1. 根据上一步的2和3，修改之前的prologue和return语句中的callee saved寄存器压栈/出栈的指令，并插入为溢出变量开辟/恢复栈顶 `sp` 的指令
    2. 对每条指令，如果 `src/dst` 变量有溢出的，根据偏移量表插入对应的 `ldr/str` 指令 (利用前面陈述的预留寄存器)
    3. 对 `AS_Move` 指令，如果 `src` 和 `dst` 分配到同一个寄存器 (或者都溢出了)，删除该指令

最后只需要在输出指令时传入着色表 `coloring`，即可打印出对应的寄存器。

具体例子

下面以 `example.fmj` 为例对编译器的各阶段进行分析和解释。

源码如下：

```
public int main() {
    int[] a={0};
    int i=0;
    int l;
    class c1 o1;
    class c1 o2;

    a=new int[getnum()];
    l=length(a);
    o1=new c1();
    o2=new c2();

    while (i < l) {
        if ( (i/2)*2 == i )
            a[i]=o1.m1(i);
        else
            a[i]=o2.m1(i);
        i=i+1;
    }
    putarray(l, a);
    return l;
}

public class c1 {
    int i1=2;
    public int m1(int x) {
```

```
      return this.i1;
    }
}

public class c2 extends c1 {
    // int i1=3;
    public int m1(int x) {
        return this.i1+x;
    }
}
```

1. **词法和语法分析**

这里对所有语句的解析都是类似的，仅以 `int [] a={0};` 为例：

Lex生成的词法分析器将语句解析为多个token `INT`、`[`、`]`、`ID`、`=`、`{`、`NUM`、`}`、`;`，Yacc生成的语法分析器接收到这些token后根据规则进行规约，覆盖规则如下：

```
VAR_DECL: INT '[' ']' ID '=' '{' CONST_LIST '}' ';' {
    $$ = A_VarDecl($1, A_Type($1, A_intArrType, NULL), $4->u.v, $7);
}

CONST_LIST: CONST CONST_REST {
  $$ = A_ExpList($1, $2);
}

CONST: NUM {
  $$ = $1;
}

CONST_REST: /* empty */ {
  $$ = NULL;
}
```

首先根据产生式1，确定构造的AST节点类型为 `A_varDecl`；碰到 `CONST_LIST` 需要递归应用产生式2进行解析，再依次递归将 `NUM` 规约为 `CONST`，由于只有一个0，用产生式4将空规约为 `CONST_REST`；然后依次返回，在产生式2的动作部分返回 `A_ExpList(1, NULL)`，在产生式1的动作部分返回 `A_VarDecl(A_Pos(...), A_Type(...), A_Type(...), A_ExpList(1, NULL))`，从而构造出 `int[] a={0};` 对应的AST节点。

2. **类型检查和翻译成IR**

着重解释两点：(为方便阅读，这里展示的IR是规范化后的)

一是数组的定义。以 `int [] a={0};` 为例，首先调用 `malloc` 分配16字节的内存 (64位)，将+8的地址赋给 `t103`，然后在 `t103-8` 地址存储数组长度1，在 `t103` 地址存储初始化值0。

```
Move t103:int, Binop(T_plus, t118:int, 8:int):int
Move Mem(Binop(T_plus, t103:int, -8:int):int):int, 1:int
Move Mem(t103:int):int, 0:int
Move t102:int, t103:int
```

二是类的处理。在预处理阶段，两次爬树后建立类环境表 `cenv` 如下：

```
{
    "c1": {
        "cd": A_classDecl,
        "fa": NULL,
        "vtbl": {
            "v1": {"vd": A_varDecl, "ty": Ty_int}
        },
        "mtbl": {
            "m1": {"md": A_methDecl, "from": c1, "ret": Ty_int, "fl": (Ty_int, Ty_int)}
        }
    },
    "c2": {
        "cd": A_classDecl,
        "fa": c1,
        "vtbl": {
            "v1": {"vd": A_varDecl, "ty": Ty_int} // copy from c1
        },
        "mtbl": {
            "m1": {"md": A_methDecl, "from": c2, "ret": Ty_int, "fl": (Ty_int, Ty_int)}
        }
    }
}
```

其中 `c2` 类的变量 `v1` 来自父类 `c1`，`c2` 和 `c1` 类的方法 `m1` 签名相同但 `from` 不同。在翻译时，根据Unified Object Record策略，`c1` 和 `c2` 类的对象申请空间都为16个字节，同时通过在相同位置存储不同方法地址 `c1$m1` 和 `c2$m1` 实现多态。

```
// o1=new c1()
Move t110:int, malloc 16:int:int
Move Mem(t110:int):int, 2:int
Move Mem(Binop(T_plus, t110:int, 8:int):int):int, c1$m1:int
Move t106:int, t110:int
// o2=new c2()
Move t111:int, malloc 16:int:int
Move Mem(t111:int):int, 2:int
Move Mem(Binop(T_plus, t111:int, 8:int):int):int, c2$m1:int
Move t107:int, t111:int
```

3. **IR规范化**

以 `a=new int[getnum()];` 为例，在规范化前IR树如下：

```
T_Move(
  T_Temp(Temp_namedtemp(102,T_int))/*T_int*/,
  T_Eseq(
      T_Seq(
        T_Move(
```

```
          T_Temp(Temp_namedtemp(108,T_int))/*T_int*/,
          T_Cast(
            T_ExtCall(String("getfloat"),
              NULL
            ,T_float)/*T_float*/
          ,T_int)/*T_int*/
        ),
        T_Seq(
          T_Move(
            T_Temp(Temp_namedtemp(109,T_int))/*T_int*/,
            T_Binop(T_plus,
              T_ExtCall(String("malloc"),
                T_ExpList(
                  T_Binop(T_mul,
                    T_Binop(T_plus,
                      T_Temp(Temp_namedtemp(108,T_int))/*T_int*/,
                      T_IntConst(1)/*T_int*/
                    )/*T_int*/,
                    T_IntConst(8)/*T_int*/
                  )/*T_int*/,
                  NULL)
              ,T_int)/*T_int*/,
              T_IntConst(8)/*T_int*/
            )/*T_int*/
          ),
          T_Move(
            T_Mem(
              T_Binop(T_plus,
                T_Temp(Temp_namedtemp(109,T_int))/*T_int*/,
                T_IntConst(-8)/*T_int*/
              )/*T_int*/
            , T_int)/*T_int*/,
            T_Temp(Temp_namedtemp(108,T_int))/*T_int*/
          )
        )
      ),
      T_Temp(Temp_namedtemp(109,T_int))/*T_int*/
  )/*T_int*/
),
```

线性化将CALL语句移到最外层，并且消除了ESEQ节点和SEQ节点。

```
Move t117:float, getfloat :float
Move t108:int,   Cast(t117:float):int
Move t116:int, malloc Binop(T_mul, Binop(T_plus, t108:int, 1:int):int, 8:int):int:int
Move t109:int, Binop(T_plus, t116:int, 8:int):int
Move Mem(Binop(T_plus, t109:int, -8:int):int):int, t108:int
Move t102:int, t109:int
```

至于分块和重排序，太过冗长不做展示。

4. **LLVM指令选择**：规范化后的线性IR和LLVM指令基本上一一对应，不需要解释。只注意一点，在IR中不区分int和ptr，但在LLVM中需要区分，如在 `int [] a={0};` 中调用 `malloc` 后需要增加 `ptrtoint` 指令将返回值转换成i64类型。

```
// stm
Move t118:int, malloc 16:int:int

// ins
%r119 = call i64* @malloc(i64 16)
%r118 = ptrtoint i64* %r119 to i64
```

5. **SSA**

算法完全参考虎书，无需解释，仅以 `int i=0; ... while (i<l) { ... i=i+1; }` 为例说明插入 ϕ 函数。在未转换成SSA之前的LLVM指令如下：

```
C7:
    ...
    %r104 = add i64 0, 0
    ...
L0:
    %r140 = icmp slt i64 %r104, %r105
    ...
L4:
    %r104 = add i64 %r104, 1
    br label %L0
    ...
```

可以看到block `L0` 用到的变量104可能来自block `C7` 或者block `L4` 的定义，显然前面陈述的满足需要插入 ϕ 函数的条件。

在转换成SSA后，指令如下：

```
C7:
    ...
    %r164 = add i64 0, 0
    ...
L0:
    %r192 = phi i64 [%r206, %L4], [%r164, %C7]
    %r193 = icmp slt i64 %r192, %r177
    ...
L4:
    %r206 = add i64 %r192, 1
    br label %L0
    ...
```

可以看到按预期插入了 ϕ 函数，并且变量都被重命名过。

6. **RPi指令选择**：这里的处理逻辑程式化且不难理解，不做解释。

7. **活跃分析**：也都是程式化的，只解释一处，对MOVE指令，如果它的Live-Out和Use变量一样，就不会和Def变量冲突，例子如下。

```
------Ins------
mov r162, r159 //
...
mov r163, r159 // means r159 is live-out for instruction above

------Interference Graph------
159,  (21): // empty
162,  (26): // empty
```

8. **寄存器分配**：逻辑程式化，但过程很繁琐，不做解释。

总结

本学期针对FDMJ语言，学习了如何实现从源码到汇编码的编译过程。

本报告概述了FDMJ编译器的设计与实现，涵盖了从解析到寄存器分配的各个阶段。

下面介绍**使用方法**：

运行 `make build` 编译项目，生成可执行文件用于编译FDMJ语言。

需要编译的源文件放在 `test` 目录下。

运行 `make compile` 会将 `test` 目录下的所有源文件编译为 LLVM IR 和 RPi 两种目标指令集的汇编代码，同时输出所有中间文件。

运行 `make clean` 清理编译生成的文件。

运行 `make run-llvm` 会使用 `lli` 运行 `test` 目录下的所有 LLVM IR 可执行的文件，并echo返回值。

`make run-rpi` 会使用 `qemu-arm` 运行 `test` 目录下的所有 RPi 可执行的文件，并echo返回值。

下面介绍**输出文件**：

1. `*.1.src`：经过词法和语法分析后得到AST，由AST重新构建的源码。
2. `*.2.ast`：xml形式的AST。
3. `*.3.irp`：便于阅读的IR+树。
4. `*.4.stm`：规范化后的IR，包括线性化IR树、基本块和规范化完成的最终IR结果。
5. `*.5.ins`：LLVM指令选择后生成的LLVM指令。
6. `*.6.cfg`：在SSA之前所需要的活跃分析结果，包括CFG、活跃分析结果和block CFG。
7. `*.7.ssa`：转换为SSA形式的LLVM代码，可以链接外部库 `libsysy64` 后运行。
8. `*.8.arm`：RPi指令选择后生成的RPi指令。
9. `*.9.ig`：活跃分析结果，包括CFG、活跃分析结果和冲突图。

10. `*.10.s`：寄存器分配后的最终结果，可以链接外部库 `libsysy32` 后运行。
11. `*.ll`：在链接外部库后，lli可执行形式的LLVM代码。
12. `*.s`：在链接外部库后，可执行形式的RPi程序。

其中从3到7的输出文件中，`ARCH_SIZE` 都为8；从8到10的输出文件中，`ARCH_SIZE` 都为4。

这是由于64位和32位的程序仅有关于 `ARCH_SIZE` 的几个数字不同，所以没有再为32位专门输出IR+树、规范化后的IR、转换为SSA形式前后的LLVM代码。