SysY 编译器

2010381　姜凡希
计算机科学与技术

摘要

SysY 语言是 C 语言的一个子集，其基本语法和终结符特征与 C 语言一致。本学期，我和张晏睿同学小组合作，基于助教给出的代码框架，实现了支持 SysY 语言基本语法的编译器。经过希冀平台测试，通过样例数为 129 个。

关键字： SysY; **编译器**

1. 引言

SysY 语言是 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等等。

SysY 语言的文法采用扩展的 Backus 范式[1]（EBNF，Extended Backus-Naur Form）表示，另外，SysY 基本终结符及语法的特征与 C 语言一致。

本学期实验中，我们需要实现一个支持 SysY 基本语法的编译器。编译器就是一个程序，它可以阅读源语言（本实验即 SysY）编写的程序，并把程序翻译为一个等价的、用目标语言（本实验即 ARM 汇编）编写的程序。编译器把源语言书写的程序映射到语义上等价的目标程序。整个映射过程可以分为两个部分：分析部分和综合部分。

分析部分把源程序分解为多个组成要素（词法分析），并在这些要素上添加语法结构（语法分析生成语法树），然后使用这个结构来创建一个中间表示（中间代码），如果在分析过程中检查出源程序没有正确的语法构成，或语义上错误，就要提供报错信息，供用户修正程序所用。除此之外，在分析部分，编译器还需要收集有关源程序的信息，将其存放在符号表中，符号表这一数据结构将和中间表示形式一起传送给综合部分。

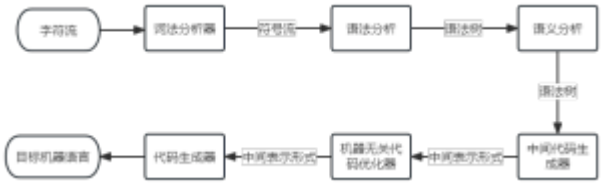

图 1. 一个编译器的各个步骤

综合部分根据中间表示和符号表的信息一起构造目标程序。一个典型的编译过程分解为多个步骤，见图1。

在本学期实验中，本人基本上由自己独立完成了 SysY 编译器。经过希冀平台测试，通过样例数为 129 个。

2. 工作说明

由于主要工作都由一人完成，因此只介绍工作内容详情。

词法分析. 莫各个进制整数、标识符等的正则定义，以及识别它们和 SysY 保留字的语义动作。负责注释、制表符、换行符等的正则定义。二进制转十进制函数、整数转字符串等辅助函 数的编写，以及 SysY 的运行时库函数识别。

语法分析.表达式节点的构造，自底向上搭建从基本表达式（primaryexp）到表达式（exp） 的完整路径，简单初值表达式的构造，函数实参的构 造，变量声明（不包括数组）的构造，break、continue 语句的构造。语法分析过程（parser.y）中相关的非终结符包括：

- Exp AddExp Cond LOrExp PrimaryExp
- LVal RelExp EqExp LAndExp InitVal（不包括数组）
- MulExp UnaryExp FuncRParams Vardef（无数组）
- vardeflist（无数组）、BreakStmt ContinueStmt

语句节点的构造，函数形参的构造、声明语句、控制流语句、赋值语句、数组相关的构 造。语法分析过程（parser.y）中相关的非终结符包 括：

- Stmts Stmt AssignStmt BlockStmt IfStmt WhileStmt
- ReturnStmt DeclStmt FuncDef VarDef VarDefList
- FuncFParams FuncFParam ExprStmt BlankStmt
- InitValList InitVal ArrayIndices FuncArrayIndices

类型检查.变量、函数未声明、变量重 复声明；int2bool 隐式转换函数调用 与函数定义不匹配、函数返回值类型检查、break、 continue 是否在 while 里的检查、数值运算表示运 算数类型检查。

中间代码生成.变量、常量声明和初始化，关系表达式、布尔表达式、条件表达式的隐式 转换，控制流的部分内容。张晏睿负责 if、if-else、 while、break、continue 等语句，id 的数组部分，函数定义部分的中间代码生成。

目标代码生成. laod、store、cmp、mov、 zext、Not、call 指令的翻译和寄存器分配算法;store、alloca、uncond、cond、ret、gep 指令的 翻译。MachineUnit,MachineFunc,MachineBlock 的翻译。

3. 词法分析

词法分析器读入组成源程序的字符流，并将它们组织成为有意义的词素序列。对于每个词素，词法分析器都产生包括这个词素属性的词法单元，并将词法单元传递给后续的语法分析步骤。在实验中，我借助 flex，完成了各个进制整数、标识符等的正则定义，以及识别它们和 SysY 保留字的语义动作。使用正则定义表达式可以方便地识别整数、标识符等。在 SysY 语言中：

- 十进制整数的正则表达式：第一个符号是 1-9 中的一个数字，接下来连接若干个 0-9 的数字，或者连接符号 0。
- 八进制整数的正则表达式：第一个符号是数字 0，第二个符号是 1-7 中的一个数字，接下来连接若干个 0-7 中的数字。
- 十六进制整数的正则表达式：第一个符号是 0，第二个符号是 x，第三个符号是 1-f 中的符号，接下来连接若干个 0-f 中的符号。
- 标识符的正则表达式：第一个符号是字母或下划线，之后连接若干个下划线或 0-9 数字或字母。

正则式设计如下：

```
1 INT_BINARY ( 0 [ b |B] [ 0 | 1 ] + )
2 INT_DEX [1 − 9 ][ 0 −9 ] ∗ | [0 ]
3 INT_HEX [ 0 ] [ Xx](([1 −9 a−fA−F
      ][0 −9a−fA−F] ∗ ) | [ 0 ] )
4 ID [ [ : alpha : ] _ ] [ [ : alpha : ] [ :
      d i g i t : ] _] ∗
```

在语义动作部分，我们需要从 yytext 接收一个字符串，并将其根据前面定义的正则式识别为词素，返回一个语法分析接受的终结符，用十六进制整数

和标识符来举例：

```
{INT_HEX} {
        offset+=strlen(yytext);
        int n;
        sscanf(yytext, "%x",&n);
        yylval.itype = n;
        return INTEGER;
}
```

识别到的词素内容保存在 yytext 中，使用格式化写入方式，将 yytext 中的十六进制整数字符串的值保存在整数 n 中，此时的 n 已经转为十进制数，然后还需要将 n 的值传递给语法分析器，返回终结符 INTEGER，供语法分析使用产生式归约非终结符。除此之外，还可以记录下此词素首次出现的行列信息。

对于标识符，需要将 yytext 里的字符串拷贝到一个新的字符数组中，并通过为 yylval 的 strtype 属性赋值的方式传递：

```
{ID} {
        offset += strlen(yytext);
        if (dump_tokens) {
                char c[1000] = {0};
                sprintf(c, "ID\t%s\t", yytext);
                DEBUG_FOR_LAB4((string)c);
        }
        char * lexeme;
        lexeme = new char[strlen(yytext) + 1];
        strcpy(lexeme, yytext);
        yylval.strtype = lexeme;
        return ID;
}
```

除此之外，在词法分析需要识别各种 SysY 语言的保留字，例如 int、void、while、if、return、break、continue 等，还需要识别运算符，例如：+、-、*、/、%、=、==、！等，以及界符 [、]、{、}、(、) 等。识别这些保留字或运算符的语义动作格式较为单一，以保留字 while 和运算符 + 为例：

```
"while" {
        offset += 5;
        if (dump_tokens)
                DEBUG_FOR_LAB4("WHILE\twhile");
        return WHILE;
}

"+"     {
        offset += 1;
        if ( dump_tokens )
                DEBUG_FOR_LAB4("ADD\t+");
        return ADD;
}
```

只需要识别它并传递给语法分析器即可，并记录此词素长度（可选）。

另外，由于 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。因此选择直接识别这些运行时库函数[2]。

4. 语法分析

编译器的第二个阶段是语法分析，它接收词法分析传递来的各个词法单元来创建抽象语法树。在实验中，我们借助 Yacc 实现相关语法分析代码。

语法分析需要搭建完整的类型系统、完善符号表、设计语法树节点并搭建语法树、扩展文法设计翻译模式。

搭建类型系统. 框架中已给出了 int 型、pointer 类型、void 类型、function 类型，为了支持 const 类型，我们引入SysY定义如下：

- BoolType：布尔类型，供 if、while 的条件判断语句使用。
- IntType ，ConstIntType：供整数立即数、整数变量使用。
- VoidType：供 void 型函数使用。
- FunctionType：供函数使用，记录返回类型、参数个数及类型。
- PointerType：供数组类型查找下标使用。
- ArrayType：供数组类型使用，为应对多维数组的情况，采取嵌套声明的方式。成员 elementtype 记录这一维数组的类型，成员 len 记录这一维的长度。

在 functiontype 上，需要为其添加两个变量成员：

- std::vector<Type *> params;
- std::vector<SymbolEntry *> paramsSe;

以及对应的设置方法 setparams 和访问方法 getparams。

完善符号表. 符号表是编译器用于保存源程序符号信息的数据结构，这些信息在词法分析、语法分析阶段被存入符号表中，最终用于生成中间代码和目标代码。符号表条目可以包含标识符的词素、类型、作用域、行号等信息。符号表主要用于作用域的管理，我为每个语句块创建一个符号表，块中声明的每一个变量都在该符号表中对应着一个符号表条目。

框架代码中，已经定义了三种类型的符号表项：用于保存字面值常量属性值的符号表项、用于保存编译器生成的中间变量信息的符号表项以及保存源程序中标识符相关信息的符号表项。我们需要完善符号表的查找函数（lookup），为了方便后续类型检查时检查变量重复声明，实现了本作用域内的符号表查找函数（localLookup），

完善产生式. 在语法分析源文件 parser.y 中，很多工作着重在一些并非记录在语法树中但起到承上启下重大作用的非终结符，工作范围如下：

- Exp AddExp MulExp UnaryExp PrimaryExp LVal
- Cond LOrExp LAndExp EqExp RelExp
- FuncRParams InitVal（不包括数组）
- Vardef（无数组）、vardeflist（无数组）
- BreakStmt ContinueStmt

其中较为关键的是自底向上搭建从基本表达式（primaryexp）到表达式（exp）的路径。

```
Exp : AddExp
AddExp : MulExp
       |AddExp ADD MulExp
       |AddExp SUB MulExp
MulExp : UnaryExp
       |MulExp MUL UnaryExp
       |MulExp DIV UnaryExp
       |MulExp MOD UnaryExp
UnaryExp : PrimaryExp
         |ID LPAREN FuncRParams RPAREN
         |ADD UnaryExp
         |SUB UnaryExp
         |NOT UnaryExp
PrimaryExp : LV al
           |INTEGER
           |LPAREN Exp RPAREN
LVal : ID
```

在移进、归约的过程中，需要同时建立树的节点，例如，对 LVal，由 ID 或数组归约而成，需要为其创建符号表项和 Id 节点，对于整数立即数，要创建符号表项和 constant 节点；向上归约到 UnaryExp，会创建 UnaryExpr 节点或 FuncCallExpr 节点；再向上规约到 Add、MulExp 需要创建对应的 BinaryExpr 节点。这样逐层搭建，就创建好了语法树。类似地，对于逻辑判断表达式，会走如下一条道路：

```
Cond : LorExp
LorExp : LandExp
       |LOrExp OR LAndExp
LAndExp : EqExp
        |LAndExp AND EqExp
EqExp : RelExp
      |EqExp EQUAL RelExp
RelExp : AddExp
       |RelExp LT AddExp
       |RelExp LTE AddExp
       |......
```

从由 AddExp 规约到 RelExp 一刻起，这个表达式就被认为是逻辑判断表达式的一份子，在这里要注意区分优先级，即 == 优先级应该比 <=、> 等关系表达式高，否则在后续验证 IR 正确性时会发现此处导致的计算结果错误。当表达式最终被规约到 Cond 时，需要添加一个属性 IsCond，这在后续控制流语句生成 IR 时起到了关键作用，使用这个属性可以更方便地实现 Int2Bool 的隐式转换而无需添加树节点。

其余部分的产生式的用途则较为单一，容易理解。对于函数实参，本质上是由 Exp 和逗号递归而成，注意对于产生式的第二个式子，需要设计 next 指针链接器作为函数实参的 Exp，便于后续函数调用时的类型检查。

```
FuncRParams : Exp
            |Exp COMMA FuncRParams
```

类似于函数实参的 next 指针，变量声明时也需要使用 next 指针建立起在同一个 vardeflist 中的变量声明，这是为了方便后续生成 IR 时通过 next 遍历声明列表即可为所有的变量声明生成中间代码。

```
VarDefList : VarDef COMMA VarDefList
           |VarDef
VarDef : ID
       |ID ASSIGN InitVal
```

在不考虑数组的情况下，initval 只由 Exp 归约而成，break、continue statement 的归约也很简单，识别并生成对应的语法树即可，不再赘述。

完善树节点. 框架给出的语法树节点类均直接或间接继承自 node 类，node 类以下共分两类，分别为表达式节点或语句节点，例如，表达式节点包括了二元表达式、一元表达式等、语句节点包括了声明语句、if、ifelse、while 语句节点等。

语法树节点的生成和其信息的储存非常重要，因为在编译的后续过程中，无论是类型检查还是中间代码生成，都需要用到它。为了涵盖更多的语法结构，我们需要扩展了语法树节点。对于 BinaryExpr，BinaryExpr 涵盖了所有操作数为 2 的表达式，包括算数表达式、关系表达式、布尔表达式，我们需要补齐操作数类型。UnaryExpr 也需要补齐操作类型。ConstantExpr、Id、FuncCallExpr 较为简单，无需做太多更改。语法树节点部分的代码主要注重在各类节点的区分，需要在顶层的 exprnode 添加很多成员来实现。继承自 StmtNode 的节点更多，但作为语句节点，其用途也更为简单，这些节点包括：ExprStmt, 使得类似直接由函数调用构成的语句可以被翻译；WhileStmt，翻译 while 语句；BlankStmt，翻译只由一个分号构成的语句；EmptyStmt，翻译空语句，处理函数体中为空的情况；还有 if、ifelse、break 等 statement，不再赘述。

5. 类型检查

类型检查使用语法树和符号表中的信息来检查源程序是否和语言定义的语义一致。具体而言，在本实验中，我们设计的类型检查完成了以下工作：

变量、函数未声明. 变量、函数未声明就作为表达式中的左值或右值出现。这一部分的类型检查可以再语法分析时完成。在赋值语句归约时，使用左值 的 name 在符号表中查询，若得到空指针则说明此 变量未定义，类似地，赋值语句右值也要做相同的 检查。在 ExprStmt 归约时，需要检查函数调用的函 数名是否被定义过。

变量重复声明. 在 vardef 归约时，需要在一开始检查符号表是否已有相同 name 的符号表条目，若有则说明此前以声明过此变量，此处为重复声明。

Int2Bool **隐式转换**. 条件判断表达式中 int2bool 的隐式转换。当 Exp 规约到 Cond 时，即意味着此 Exp 之后会出现在条件表达式中，我们将其成员 IsCond 置为 True。之后在中间代码生成时，我们会将 Exp 的 IsCond 属性逐层向子表达式传递，并生成 bool 类型的操作数和相应的比较指令（CmpInstruction）、条件跳转指令（CondInstruction）、无条件跳转指令（UncondInstruction）。

6. 中间代码生成

在编译器的分析-综合模型中，前端对源程序进行分析并产生中间表示，后端在此基础上生成目标代码。中间代码生成的总体思路是对抽象语法树作一次遍历，使用语法树生成 SSA 形式的中间代码。

在代码框架中，使用一个 unit 单例类指导中间代码生成。每遍历到 functiondef 节点，都新建一个 function，function 下设多个基本块，基本块内部由一个或多个 instruction（指令）构成，基本块之间通过条件跳转指令或无条件跳转指令迁移。unit 的多个 function、function 的多个 basicblock、basicblock 的多个 instruction 由均由链表管理。

变量、常量的声明和初始化. 这一部分的中间代码生成放在 DeclStmt 中进行。首先需要判断此声明为全局声明还是函数作用域内的局部变量声明，或函数实参。如果为全局声明，我们选择将其链入 globlist 中，在 unit 单例类输出中间代码时的一开始，就将全局声明的中间代码输出，形式如下：

```
1  extern FILE *yyout ;
2  for ( auto &se : glob_list )
3  {
4          fprintf (yyout ,
5          "%s␣=␣ global ␣%s␣%d , ␣
               align ␣ 4\n" ,
6          se->toStr () . c_str () ,
7          se->getType ()->toStr () .
               c_str () ,
8          se->getIntValue () ) ;
9  }
```

若为局部变量或函数实参，我们需要在 entry-block 中生成为其分配函数栈空间，并将 allocain-struction 插入在 entryblock 的开始部分。如果是函数参数或局部变量在声明时进行了初始化，还需要再创建 store 指令。

关系表达式的翻译. 关系表达式包括 >,<,>=,<=,==,!=。对于这些关系表达式，我们默认他们只会出现在条件判断语句中。因此要

1. 创建 cmpinstruction,opcode 与当前关系表达式一致。
2. 创建 truebb、falsebb、tempbb 三个基本块，用于控制流的翻译。
3. 创建从 truebb 到 tempbb 的条件跳转语句并加入到此表达式的 true_list 中。
4. 创建从 tempbb 到 falsebb 的无条件跳转并加入到此表达式的 false_list 中。

布尔表达式的翻译. 布尔表达式包括 AND、OR、NOT。

对于 AND，见图2，其中间代码生成需要：

1. 为子表达式 1 生成 IR。
2. 将 truebb 回填至子表达式 1 的 true_list，并将当前块修改为 truebb。
3. 为子表达式 2 生成 IR。

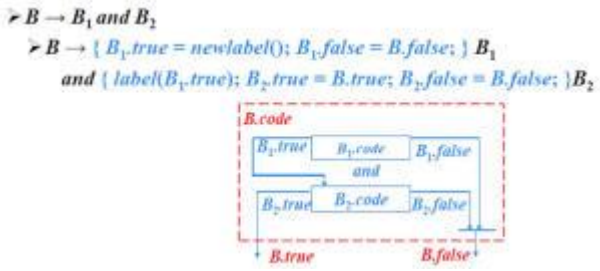

图 2. AND 表达式举例：B1 and B2 的 SDT

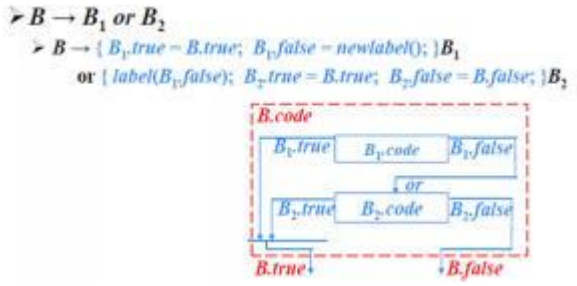

图 3. OR 表达式举例：B1 or B2 的 SDT

4. 将此表达式的 true_list 赋值为子表达式 2 的 true_list。
5. 将此表达式的 false_list 赋值为子表达式 1 和子表达式 2 的 false_list。

对于 OR，见图3，其中间代码生成需要：

1. 为子表达式 1 生成 IR。
2. 将 truebb 回填至子表达式 1 的 false_list，并将当前块修改为 truebb。
3. 为子表达式 2 生成 IR。
4. 将此表达式的 false_list 赋值为子表达式 2 的 false_list。
5. 将此表达式的 true_list 赋值为子表达式 1 和子表达式 2 的 true_list。

对于 not，由于 not 表达式实际上是一个一元表达式，由 op（!）和子表达式构成，我们在生成 IR 时，将!x 转化为 x != 0 处理，生成一个 cmp 指令，再生成一个用异或运算实现的取非指令：

当普通表达式、关系表达式等作为条件判断表达式（隐式转换）. 在语法分析时，当一个（复杂的）由基本表达式/一元表达式/二元表达式组合成的表达式规约到条件判断表达式时，将其成员 IsCond 置为 true。在生成 IR 时，对于二元表达式中的 and 和 or，要将 iscond 传递给子表达式；对于二元表达式中的关系表达式，则需要判断子表达式是否为一元表达式/基本表达式/函数调用表达式，若满足以上情形之一则不能将 iscond 表达式传递给子表达式，举例说明：如果形如（a && b）！=0，那么此时子表达式分别为（a && b）和 0，需要把 iscond 传递下去给（a&&b）做进一步的条件判断，形如 a！=0 则不需要，因为此前我们对一元表达式 a 做条件判断表达式时的处理理解为 a！=0，那么在此处若将 iscond 传递给子表达式 a，则实际上此条件表达式的中间代码生成的结果为（a！=0）！=0 的结果。

通过按照如上形式的 iscond 传递，可在一元表达式/二元表达式/基本表达式 (Id) 生成其内容本身的中间代码之后，生成 cmp 指令判断是否与 0 相等，并添加由 truebb 到 tempbb 的条件跳转指令，加入 true_list；添加由 tempbb 到 falsebb 的无条件跳转指令，加入 false_list。

```
1 if ( isCond ) {
2 SymbolEntry * se = new
      ConstantSymbolEntry (
      TypeSystem : : intType , 0) ;
3 Constant * d i g i t = new Constant
      ( se ) ; // 0
4 Operand *op = this ->getOperand
      ( ) ;
5 Operand * t = new Operand (new
      TemporarySymbolEntry (
      TypeSystem : : boolType ,
      SymbolTable : : getLabel ( ) ) ) ;
6 new CmpInstruction (
      CmpInstruction : : NE, t , op ,
      d i g i t ->getOperand ( ) , bb) ;
7 this ->t r u e L i s t ( ) . push_back (
8       new CondBrInstruction (
            trueBB , tempbb , t , bb) )
            ;
```

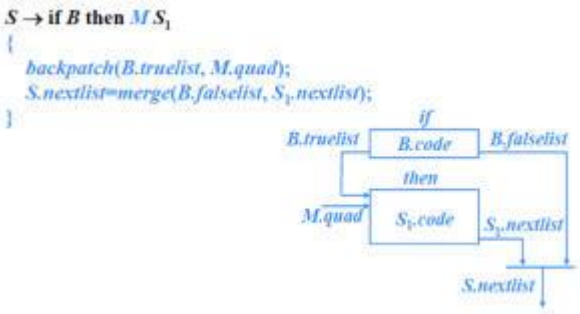

图 4. S → if B then S1 的 SDT

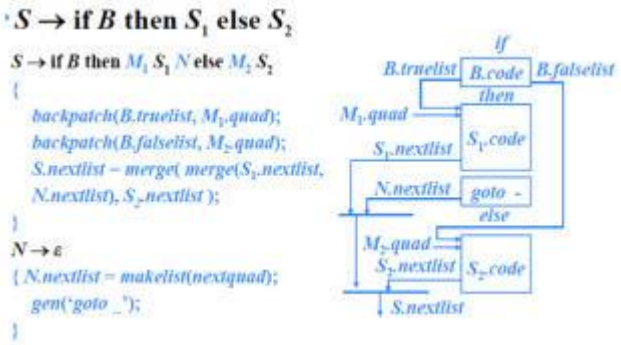

图 5. S → if B then S1 else S2 的 SDT

```
9   this ->f a l s e L i s t ( ) . push_back (
        new   UncondBrInstruction   (
        falseBB , tempbb ) ) ;
10  }
```

控制流语句的翻译. if 语句 (图4)、if-else 语句 (图5)、while 语句 (图6)。

对于 if 语句的翻译，共涉及到四个 basicblock，分别是当前基本块 insert_bb, 条件表达式基本块 cond_bb，条件表达式为真时的基本块 then_bb，if 语句结束后的基本块 end_bb。

首先生成一条从 insert_bb 到 cond_bb 的无条件跳转指令，然后为 cond_bb 中的表达式生成 IR，将 then_bb 回填至 cond_bb 的 truelist，将 end_bb 回填至 cond_bb 的 false 分支，然后为 then-stmt 中的语句生成 IR，最后生成一条从 then_bb 到 end_bb 的无条件跳转指令。

对于 if-else 语句的翻译，其涉及的基本块比 if 语句多一个 elsebb，首先处理好各个块之间的前驱后继关系。分别创建由 insertbb 到 condbb，由 thenbb 到 endbb，由 elsebb 到 endbb 的无条件跳转，并进行如下回填：

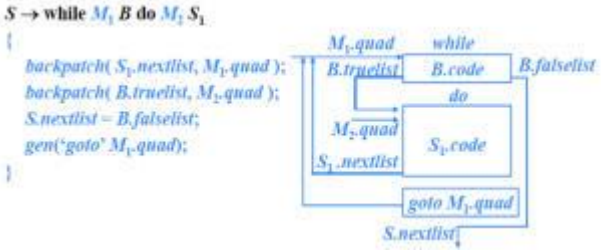

图 6. S → while B do S1 的 SDT

```
1   backPatch ( cond->t r u e L i s t ( ) ,
        then_bb ) ;
2   backPatch ( cond->f a l s e L i s t ( ) ,
        else_bb ) ;
```

while 语句的控制流则更为简洁，涉及 condbb,whilebb(包括 block 里的所有指令),endbb 和当前的基本块共四个。其关键代码如下：

```
1   new   UncondBrInstruction   (
        cond_bb , bb) ;
2   backPatch ( cond->t r u e L i s t ( ) ,
        while_bb ) ;
3   backPatch ( cond->f a l s e L i s t ( ) ,
        end_bb) ;
4   new   UncondBrInstruction   (
        cond_bb , while_bb ) ;
```

构造流图. functiondef 节点，我们需要为此函数的基本块添加其前驱后继，其中一部分我们已在控制流语句的翻译中完成。在这里需要为每一个条件跳转语句和无条件跳转语句分配要去的基本块，其去向我们已经在控制流语句的翻译中回填完毕，在这里直接取出设置即可：

对于条件跳转指令，取出真分支基本块和假分支基本块，二者的前驱都是当前基本块，当前基本块的后继设为这两个基本块即可。

```
BasicBlock *truebb , * f a l s e b b ;
truebb = dynamic_cast<
    CondBrInstruction *>
( i n s t )->getTrueBranch ( ) ;
f a l s e b b = dynamic_cast<
    CondBrInstruction *>
( i n s t )->getFalseBranch ( ) ;
(* block )->addSucc ( truebb ) ;
(* block )->addSucc ( f a l s e b b ) ;
truebb->addPred (* block ) ;
falsebb ->addPred (* block ) ;
```

类似地，对于无条件跳转指令，设置它的分支基本块和当前基本块的前驱后继关系即可。

```
BasicBlock * dst = dynamic_cast
    <UncondBrInstruction *>
i n s t )->getBranch ( ) ;
(* block )->addSucc ( dst ) ;
dst->addPred (* block ) ;
```

7. 目标代码生成

7.1. 生成汇编代码

在中间代码生成之后，对中间代码进行自顶向下的遍历，从而生成目标代码。

访存指令. 对于 load 指令，其一共要处理三种类型：全局变量，局部变量，临时变量。

- 全局变量（eg. load r0, addr_a, load r1, [r0]）：先生成一条 load 指令将其加载到临时寄存器中，再从临时寄存器中 load。
- 局部变量（eg. load r1, [r0, #4]）：由于在 AllocaInstruction 指令中，已经为所有的局部变量申请了栈内空间，并将其相对 FP 寄存器的栈内偏移存在了符号表中，只需要以 FP 为基址寄存器，根据其栈内偏移生成一条 load 指令即可。
- 临时变量（eg. load r1, [r0]）：一条简单的 load 指令即可。

二元运算指令. 二元运算指令生成较为简单，主要需要注意两个操作数可能为立即数，需要先生成 load 指令加载到寄存器中，另外需要注意 arm 不支持 mod 指令，需要手动实现：a mod b = a−(a/b)×b，共由减法、除法、乘法三条指令构成。

比较指令. 同二元运算指令一样，需要注意两个操作数可能为立即数，需要先生成 load 指令加载到寄存器中，另外比较指令生成后，会将标记保存在 aspr 寄存器中，后面跟两个 mov 指令：cmp 为真时存 1、cmp 为假时存 0。

```
if ( opcode >= CmpInstruction : :
&& L
    opcode <= CmpInstruction : :
    GE)
{
    auto trueope =
        genMachineImm (1) ;
    auto f a l s e o p e =
        genMachineImm (0) ;
    i n s t = new MovMInstruction
        (mbb,
    MovMInstruction : :MOV, dst ,
    trueope , opcode ) ;
    mbb->I n s e r t I n s t ( i n s t ) ;
    i n s t = new MovMInstruction
        (mbb,
    MovMInstruction : :MOV, dst ,
    falseope , 7 − opcode ) ;
    mbb->I n s e r t I n s t ( i n s t ) ;
}
```

call **指令**. 函数调用指令。首先为前四个参数分配物理寄存器 r0 到 r3。

```
f o r ( i = 1 ; i < operands . s i z e
    ( ) && i < 5 ; i++)
{
```

```
        operand = genMachineReg (i - 1); // r0~r3
        auto src = genMachineOperand (operands[i]);
        inst = new MovMInstruction (mbb, MovMInstruction::MOV, operand, src);
        mbb->InsertInst(inst);
    }
```

若超过四个寄存器则需要生成 stack push 命令，这样就完成了参数的工作。

```
for (; i > 4; i --)
{
        operand = genMachineOperand (operands[i]);
        std::vector<MachineOperand*> vec;
        inst = new StackMInstruction (mbb, StackMInstruction::PUSH, vec, operand);
        mbb->InsertInst(inst);
}
```

之后生成跳转指令来进入 Callee 函数。

```
new BranchMInstruction (mbb, BranchMInstruction::BL, label);
```

在此之后，需要进行现场恢复的工作，如果之前通过压栈的方式传递了参数，需要恢复 SP 寄存器.

```
if (operands.size() > 5)
{
        auto off = genMachineImm ((operands.size() - 5) * 4);
        auto sp = new MachineOperand (MachineOperand::REG, 13);
        inst = new BinaryMInstruction (mbb, BinaryMInstruction::ADD, sp, sp, off);
        mbb->InsertInst(inst);
}
```

最后，如果函数执行结果被用到，还需要保存 R0 寄存器中的返回值。

```
// return value, mov r0, r4
if (dst)
{
        operand = genMachineOperand (dst);
        auto r0 = new MachineOperand (MachineOperand::REG, 0);
        inst =new MovMInstruction (mbb, MovMInstruction::MOV, operand, r0);
        mbb->InsertInst(inst);
}
```

zext、Not **指令**. zext 指令的汇编代码本质上是一个 mov 指令，其操作数已经在中间代码生成是准备好，此处直接包装为 machineoperand 并生成 mov 指令即可。not 指令的汇编代码可由两个 mov 指令构成：

```
auto dst = genMachineOperand (operands[0]);
auto trueOperand = genMachineImm (1);
```

```
auto falseOperand =
    genMachineImm(0);
auto inst = new
    MovMInstruction(mbb,
MovMInstruction::MOV, dst,
trueOperand ,
    MachineInstruction::EQ);
mbb->InsertInst(inst);
inst = new MovMInstruction(mbb
    ,
MovMInstruction::MOV, dst,
falseOperand ,
    MachineInstruction::NE);
```

machineunit 的 output. 首先要打印出 arm 指令版本等信息，通过遍历 globlist 打印出全局变量信息，然后遍历 function 打印每个函数的 machinecode，最后打印桥信息：

```
// 1. Glob/Decl
PrintGlobalDecl();
fprintf(yyout, "\t.text\n");
// 2. Traverse
for(auto iter:func_list)
    iter->output();
// 3. bridge label
fprintf(yyout, "\n");
for(auto s:global_list)
{
    IdentifierSymbolEntry *se
        =
    (IdentifierSymbolEntry *)s
        ;
    fprintf(yyout, "addr_%s%d
        :\n",
    se->toStr().c_str(), zone)
        ;
    fprintf(yyout, "\t.word␣%s
        \n",
    se->toStr().c_str());
}
```

machinefunction 的 output. 在目标代码中，在函数开头需要进行一些准备工作。

首先需要生成 PUSH 指令保存 FP 寄存器及一些 Callee Saved 寄存器:

StackMInstruction::PUSH, getSavedRegs(), fp, lr)

之后生成 MOV 指令令 FP 寄存器指向新的栈底，

MovMInstruction::MOV, fp, sp)

之后需要生成 SUB 指令为局部变量分配栈内空间。

BinaryMInstruction::SUB, sp, sp, size)

然后遍历 block 生成 machinecode 即可。

7.2. 寄存器分配

代码框架已完成了汇编代码中每一个虚拟寄存器的活跃区间，并将这些区间按开始时间递增排序。我们首先需要新增一个成员变量 actives 记录占据物理寄存器的区间，并在之后始终维护一个按结束时间递增的顺序。

我们需要实现三个功能：

- 以函数为单位，为函数内使用的每一个虚拟寄存器（区间）分配物理寄存器直到函数的所有虚拟寄存器都被替换为物理寄存器。
- 分配失败，要为其生成溢出代码。
- 分配成功，修改汇编代码，将虚拟寄存器替换为其映射的物理寄存器。

在 allocateRegisters 函数中，遍历函数列表，为每一个函数内部的虚拟寄存器计算活跃区间，并调用 linearScanRegisterAllocation 函数尝试分配物理

图 7. 希冀平台测试结果

level	1-1	1-2	2-1	2-2	2-3	2-4	2-5	2-6	sum
通过	39	19	4	5	5	33	19	5	129
总数	39	19	4	5	5	40	26	11	149

表 1. 本地测试样例通过情况

寄存器，若分配失败则溢出代码，成功则去为下一个函数分配寄存器直到所有函数分配完毕。

在 linearScanRegisterAllocation 函数中，遍历此函数的每一个区间，回收结束时间早于此区间开始时间的区间的物理寄存器，若没有成功回收的物理寄存器，则调用 spillAtInterval 依橱一个区间，成功回收则将此区间加入 active 列表。

在 spillAtInterval 函数中，查询 active 列表的最后一个元素（结束时间最晚）并将其与当前区间比较，更晚者的 spill 标志位记为 true，始终维护 active 按结束时间递增的顺序。

回到 allocateRegisters 函数中，若成功为此函数分配好物理寄存器，则调用 modifycode 函数去修改代码，将虚拟寄存器替换为物理寄存器。若失败则调用 genspillcode 函数，遍历每一个区间，若此区间标记为 spill，则为其在栈内分配空间，获取当前在栈内相对 FP 的偏移，遍历其 USE 指令的列表，在 USE 指令前插入 LoadMInstruction，将其从栈内加载到目前的虚拟寄存器中；遍历其 DEF 指令的列表，在 DEF 指令后插入 StoreMInstruction，将其从目前的虚拟寄存器中存到栈内。

8. 实验结果及结论

希冀平台测试实验结果如图7所示。

本地测试样例，得到具体结果见表1。

样例中未通过的部分大部分为极为复杂的样例（控制流多且长）和浮点数的相关样例。经过本学期的编译实验，我对从 C 语言到汇编代码的过程有了清晰深刻的理解。这门课是我成长的一门课。做完全部实验有一种前后打通的感觉。最后感谢课程组的辛苦付出。