南開大學
Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

编译原理实验报告

编译器实现

姚鑫

年级：2020 级

专业：信息安全

指导教师：王刚

2023 年 1 月 15 日

摘要

本次编译器大作业持续一整个学期，我们小组共同合作完成了编译器的词法分析、语法分析、类型检查和中间代码生成、目标代码生成四个阶段。实现了变量、常量的声明和初始化，赋值语句，表达式语句，语句块，if 语句，while 语句，return 语句，一元、二元算术运算，关系运算，逻辑运算，清理块注释和行注释，叶函数的定义和调用，输入输出这些基本要求的部分，以及非叶函数的定义和调用，break 语句，continue 语句，常量变量作用域的区分这些进阶要求。本文将从引言，编译器总体设计，本人负责的具体模块设计，以及总结这四部分进行阐述。

关键字：编译器，语法树，符号表，词法分析，语法分析，中间代码生成，类型检查，目标代码生成

目录

一、 引言

在本次的大作业过程中，我们小组共同合作完成了编译器的词法分析、语法分析、类型检查和中间代码生成、目标代码生成四个阶段，实现了从输入 SysY 语言可以输出单词 token，到输入 SysY 语言可以输出语法树并构建符号表，到输入 SysY 语言可以进行类型检查、将正确的程序输出中间代码，到输入 SysY 语言可以输出汇编代码的四个过程。最终编译器可以实现的 SysY 语言特性包括：变量、常量的声明和初始化，赋值语句，表达式语句，语句块，if 语句，while 语句，return 语句，一元、二元算术运算，关系运算，逻辑运算，清理块注释和行注释，输入输出，实现了叶函数和非叶函数的定义和调用，break 语句，continue 语句，常量变量作用域的区分。

二、 编译器总体设计和结果展示

编译器总设计分为四个阶段，分别是词法分析阶段、语法分析阶段、类型检查和中间代码生成阶段、目标代码生成阶段。

在词法分析阶段，我们根据设计的编译器所支持的语言特性，设计了正规定义，实现了 lexel.l 文件。最终利用 Flex 工具实现词法分析器，识别了程序中所有词素，转化成为为单词流，使得当输入一个 SysY 语言程序时，最终输出每一个文法单元的类别、词素、行号，以及变量的作用域。在这个过程中包含实现词法分析的定义部分、规则部分（根据词素返回单词，八进制十六进制的识别以及转换为十进制，int、float、double、void 等类型的返回，行注释和块注释的处理，输入输出函数的处理）的内容，以及实现一个简单的符号表。如下图是测试样例 1 输出的单词流的部分截图展示：

```
INT     int     line:1
ID      func    line:1  0
LPAREN  (       line:1
INT     int     line:1
ID      a       line:1  1
COMMA   ,       line:1
INT     int     line:1
ID      b       line:1  2
RPAREN  )       line:1
LBRACE  {       line:2
RETURN  return  line:3
ID      a       line:3  3
MUL     *       line:3
ID      b       line:3  4
SEMICOLON       ;       line:3
RBRACE  }       line:4
INT     int     line:5
ID      main    line:5  5
LPAREN  (       line:5
VOID    void    line:5
RPAREN  )       line:5
LBRACE  {       line:6
INT     int     line:7
ID      a       line:7  6
COMMA   ,       line:7
ID      b       line:7  7
COMMA   ,       line:7
ID      c       line:7  8
SEMICOLON       ;       line:7
```

图 1: 词法分析部分结果展示

在语法分析阶段，lexer.l 文件不再输出单词、词素、行号等，而是返回单词给 parser.y 文件，帮助实现语法树。我们根据之前预备工作中针对要实现的 SysY 语言特性实现的上下无关文法，

设计了翻译模式，并实现了语法树结点的设计，完善了符号表（主要是符号表的查找函数），完善了类型（此时包括 IntType，VoidType，FunctionType 类型）。在 parser.y 文件中根据翻译模式生成语法树结点，并将需要的信息存入到符号表中，并在语法树结点类中实现了不同的 output 函数，最终利用 yacc 工具实现了语法分析器，并输出了语法树的结构。最终实现样例的语法树输出如下所示：

```
program
    Sequence
        DeclStmt
            Id  name: a scope: 0        type: int
        FunctionDefine function name: main      type: int()
            FuncDefParamsNode NULL
            CompoundStmt
                Sequence
                    Sequence
                        Sequence
                            DeclStmt
                                Id      name: a scope: 2        type: int
                            AssignStmt
                                Id      name: a scope: 2        type: int
                                BinaryExpr      op: add
                                    IntegerLiteral      value: 1        type: int
                                    IntegerLiteral      value: 2        type: int
                        IfStmt
                            BinaryExpr  op: less
                                Id      name: a scope: 2        type: int
                                IntegerLiteral  value: 5        type: int
                            ReturnStmt
                                IntegerLiteral  value: 1        type: int
                    ReturnStmt
                        IntegerLiteral  value: 0        type: int
~
```

图 2: 语法分析部分结果展示

在类型检查和中间代码生成阶段，先遍历语法树，进行类型检查并输出错误信息，之后进行中间代码的生成，最后进行中间代码的输出。首先是类型检查，从语法树的根节点开始遍历语法树，每个语法树结点都设置 typeCheck() 函数，通过查符号表、看语法树的上下结构等，最终实现了变量未定义、变量重定义、函数未定义、函数缺少返回语句或返回值、函数返回类型为 void 但有返回语句、函数返回类型为 void 参与了表达式运算这几个方面的类型检查，其中变量未定义和变量重定义在 parser.y 中进行类型检查，其余都在 typeCheck() 函数中进行检查。

我们构造一个 IRBuilder 类并声明一个静态对象 builder 来辅助中间代码的生成，在中间代码生成时，传入 Unit 顶层模块，从语法树的根节点开始再遍历一遍语法树，在语法树的每个节点都有 genCode() 函数，实现每个结点的 genCode() 函数，构建中间代码的 Function、BasicBlock、Instruction。除此之外，我们还在函数定义的中间代码生成函数中实现了流图的构造，并完善了基本快的 insertBefore() 函数。在中间代码输出时，涉及到的类均有 output() 函数，我们完善了这些类的输出函数，输出时从顶层模块 Unit 开始，Unit 下包含程序里对应的多个函数 Function，因此再进行每个函数进行输出，每个函数包含多个基本快 BasicBlock，再对每个基本快进行输出，每个基本快又包含多个指令 Instruction，因此最后对基本快中的每条指令输出。对 level1-1 和 level1-2 执行 make test，结果如下：

```
PASS: 057_if_complex_expr
PASS: 095_empty_stmt
FAIL: 099_skip_spaces   Compile Error
PASS: 1068_accumulate
PASS: 1070_multi
PASS: 1072_enum
PASS: 1073_exchange_value
Total: 39       Accept: 38      Fail: 1
```

图 3: 中间代码生成结果展示 1

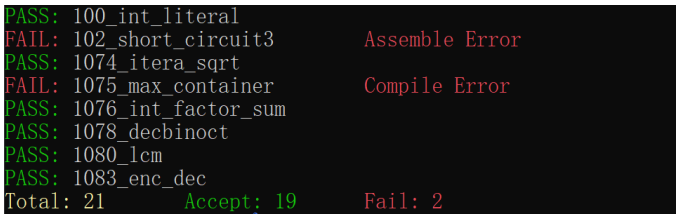

图 4: 中间代码生成结果展示 2

在最终目标代码生成时，我们构造一个 AsmBuilder 类并声明一个静态对象 builder 来辅助中间代码的生成，对原来的中间代码进行遍历，传入 MachineUnit 类对象 mUnit，从原来 Unit 开始生成汇编代码，遍历 Unit 下的所有函数 Function 生成汇编代码，再遍历 Function 下的所有基本快 BasicBlock 生成汇编代码，最后再遍历 BasicBlock 下的所有指令 Instruction 生成汇编代码，其中每个类下都有一个 genMachineCode() 函数用于汇编代码的生成。在遍历的过程中，声明和中间代码结构一样的汇编代码类 MachineFunction、MachineBlock、MachineInstruction。在原来的 Instruction.cpp 文件中完善各个指令的 genMachineCode() 函数，并生成各种汇编代码指令，在各种汇编代码指令类中完善输出函数 output() 以便之后汇编代码的输出。最后还完善了线性扫描寄存器分配算法，主要分为活跃区间分析、寄存器分配、生成溢出代码三个步骤。分配完物理寄存器之后，就可以进行汇编代码的输出了，输出时候的顺序和中间代码相似，从 MachineFunction 到 MachineBlock 再到 MachineInstruction 进行 output。

最终完成的编译器实现的 SysY 语言特性包括：变量、常量的声明和初始化，赋值语句，表达式语句，语句块，if 语句，while 语句，return 语句，一元、二元算术运算，关系运算，逻辑运算，清理块注释和行注释，输入输出等基本要求，并实现了叶函数和非叶函数的定义和调用，break 语句和 continue 语句，常量、变量作用域的区分。最终结果基本要求部分 level1 和中间代码生成相同，level2 主要实现了 break 和 continue 的部分，变量、常量作用域的区分，还有非叶函数和部分复杂样例，希冀平台测试结果如下图所示：

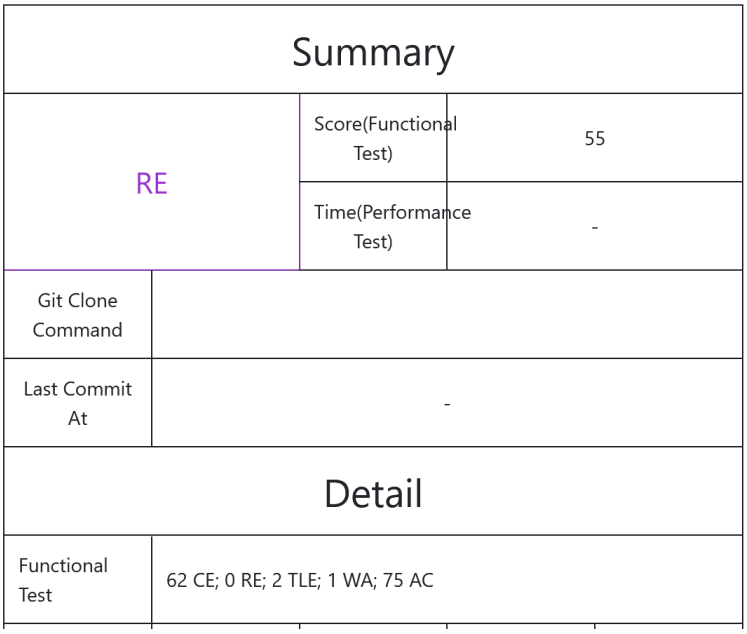

图 5: 完成编译器结果

三、 编译器各模块具体设计

我将从词法分析、语法分析、类型检查和中间代码生成、目标代码生成这四个部分简述我负责部分的编译器的具体设计。

(一) 词法分析部分

在词法分析部分，我们共同合作完成规则段（正则式），之后我的队友负责八进制、十六进制转十进制数以及它们的输出部分，我负责行注释、注释块，以及简单的符号表的实现。在我负责的部分，实现的符号表能够存储变量的作用域，而行号的实现就是简单的累加过程，注释则通过开始和结束的标志将中间的部分忽略即可。因此最后可以输出词素、单词、对应的行号、以及如果是变量则输出符号表中存储的作用域。

(二) 语法分析部分

在语法分析阶段，主要需要完善语法分析 parser.y 文件、语法树结点 Ast 相关文件、完善词法分析 lexer.l 文件并补充输入输出部分，以及完成符号表的查找函数。其中我的队友负责语法分析中的表达式语句部分（包括一元、二元算术运算，逻辑运算，关系运算），if、while、break、continue 以及块语句部分，我主要负责符号表的查找函数，语法分析中的变量、常量定义部分，赋值语句，return 语句、空语句、函数定义和调用部分、以及输入输出函数。

在我负责的部分，首先符号表的查找函数是利用递归来查找作用域中的所有符号表，在每个符号表中利用 SymbolTable 类中的 map，通过查找 string 类型的变量名找到对应的存储值。

符号表的查找函数

```
SymbolEntry* SymbolTable::lookup(std::string name) //在所有作用域查找
{
        if(symbolTable[name]!=nullptr)
        {
                return symbolTable[name];
        }
        else
        {
                if(this->level==0) return nullptr; //没找到
                return this->prev->lookup(name); //递归查找前一个符号表
        }
}
```

变量、常量定义的部分是类似的，在定义语句下分为变量定义语句和常量定义语句，变量定义语句又分为左递归的变量定义链连接逗号再连接变量定义语句，以及终结递归的变量定义语句，变量定义语句又分为初始化和不初始化两部分，最终定义的变量 ID 需要保存到符号表中，并在整个过程中都要生成对应的语法树结点，常量定义部分的区别在于在最终定义语句时没有不初始化的那部分，并且对应的定义语法树结点也不同。

函数定义部分包括函数定义整体框架和函数定义参数部分。函数定义时，函数名 ID 也需要存到符号表中，且类型存储的是 FuncType 类型，同时创建一个新的符号表（作用域 +1），保证参数部分和函数名是两个作用域，将参数类型写入函数名的符号表中，之后创建 FunctionDef 语法树结点，FunctionDef 的构造函数传入函数名 ID 的符号表和参数链结点以及块语句，在块语句中也需要创建新的符号表，保证参数部分和函数体内部又是两个作用域。函数定义参数部分，

主要分为递归参数链连接逗号再连接参数定义，终结递归的参数定义，以及无参数三个部分，参数定义最终也需要生成语法树结点。

输入输出的部分主要在 lexer.l 中实现，捕获到”getint”、”getch”、“putint”、“putch”这几个词素后，查找符号表如果之前没存过，则把这些库函数当成参数定义一样存到全局符号表中，这样之后调用时就能从符号表中找到这些库函数，实现简单的输入和输出。

(三) 类型检查和中间代码生成部分

在类型检查和中间代码生成部分，主要在 Ast.cpp 文件中进行类型检查和中间代码生成，完善每个语法树结点类中的 typeCheck() 函数和 genCode() 函数，在类型检查时遇到错误的类型就报错然后退出程序，在中间代码生成时创建了一些指令，在这些指令类中完善 output() 函数来进行最后中间代码的输出。其中我的队友负责一元、二元关系、逻辑、算术运算、if、while、break、continue 等语句中间代码生成，以及流图的完善和基本快的 insertBefore() 函数，我负责类型检查部分，中间代码生成部分的全局变量常量、局部变量常量的定义，赋值语句，return 语句，块语句，函数定义和调用，输入输出。

在类型检查中，实现了变量未定义、变量重定义、函数未定义、函数缺少返回语句或返回值、函数返回类型为 void 但有返回语句、函数返回类型为 void 参与了表达式运算这几个方面的类型检查。其中变量未定义和变量重定义这两个错误类型在 parser.y 中直接通过查找符号表来判断，在变量定义时查找符号表看是否已经定义了（检查重定义），在变量调用时查找符号表看变量是否存在（检查未定义）。

其余几个错误类型基本在 typeCheck() 函数中进行检查：函数未定义先在 parser.y 检查函数名是否在符号表中存在，不存在则报错退出，在函数调用结点的 typeCheck() 函数中检查函数参数是否一致，从上到下进行三种错误情况的判断，函数定义时无参数实际调用有参数、函数定义和调用参数数量不一致、函数定义和调用参数中有类型不一致的情况。函数的返回不一致主要在 FunctionDef 和 return 语句的语法树结点的 typeCheck() 函数中检查，主要分为函数需要返回但没返回语句，函数返回类型为 void 但 return 语句有返回值，函数需要返回值但实际 return 语句返回值为空这三种错误情况。函数返回为 void 类型但参与表达式计算错误类型主要在一元、二元表达式语法树结点的 typeCheck() 函数进行检查，判断表达式的操作数如果是函数，则进一步判断函数的返回类型是否是 void，如果是则报错退出。最后是隐式转换的部分，在 if、ifelse、while 语句的语法树结点的 typeCheck() 函数中，判断如果条件部分是 int 类型或者条件中存在函数且返回值为 int 类型，则将条件 cond 的符号表中的类型设置成 bool 型。

类型检查函数返回值部分

```
void FunctionDef::typeCheck()
{
    // Todo
    if(params!=NULL)
    {
        params->typeCheck();
    }
    // 获取函数的返回值类型
    returnType = ((FunctionType*)se->getType())->getRetType();
    // 判断函数体内是否有返回语句
    funcReturned = false;
    stmt->typeCheck();
    // 出错：函数需要返回值，但函数体里没返回值
```

```
    if(!funcReturned && !returnType->isVoid()){
        fprintf(stderr, "%s should return, but no returned value found\n",
            returnType->toStr().c_str());
        exit(EXIT_FAILURE);
    }
    returnType = nullptr;
}

void ReturnStmt::typeCheck()
{
    if(returnType == nullptr){//不是个函数
        fprintf(stderr, "This is not function,but return\n");
        exit(EXIT_FAILURE);
    }
    else if(returnType->isVoid() && retValue!=nullptr){//函数返回为void类型,
        但有return值
        fprintf(stderr, "value returned in a void() function\n");
        exit(EXIT_FAILURE);
    }
    else if(!returnType->isVoid() && retValue==nullptr){//需要return, 但没
        return值
        fprintf(stderr, "%s should return, but no returned value found\n",
            returnType->toStr().c_str());
        exit(EXIT_FAILURE);
    }
    if(!returnType->isVoid()){ //return值的检查
        retValue->typeCheck();
    }
    this->retType = returnType;
    funcReturned = true;
}
```

在我负责的中间代码生成部分，重点主要是全局变量、常量的定义，函数部分的处理和输入输出。全局变量常量的处理主要是在 Unit 中加一个 vector 存储定义全局变量常量的指令 globalvar_list，在 Instruction 中新定义一个类 globalAllocaInstruction，并在语法树的定义结点的 genCode() 函数中区分出全局变量常量，全局变量单独定义到 globalAllocaInstruction 指令中，并存到 globalvar_list 中，之后在 Unit 输出时，先单独将 globalva_list 中的指令遍历输出，再遍历存的函数，输出每个函数对应许多基本块下的所有指令。

函数定义部分先声明一个新的中间代码函数 Function，存到 Unit 中，获取函数第一个基本块，之后分成函数参数部分生成中间代码和函数体部分生成中间代码。参数部分主要是声明 AllocaInstruction 指令，以及如果有赋值的部分需要的 StoreInstruction 指令，函数体部分主要是块语句，之后块语句中的其他语句中间代码生成在别的语法树结点完成。函数调用，如果有参数则先进行参数的中间代码生成，之后利用 CallInstruction 指令来完成函数调用中间代码生成。

函数参数结点中间代码生成部分

```
void FuncDefParamsNode::genCode()
{
    Function *func = builder->getInsertBB()->getParent();
```

```
4       BasicBlock *entry = func->getEntry();
5       for(auto id : paramsList){
6           Operand* idop = id->getOperand(); //获取id的操作数
7           idop->set_isParam(true); //设置操作数是参数: 这里主要用于在
                Instruction.cpp中生genMachineOperand函数中使用
8           func->insertParam(idop); //插入到参数链表中
9           IdentifierSymbolEntry* se = dynamic_cast<IdentifierSymbolEntry*>(id->
                getSymbolEntry());
10          Type *type = new PointerType(id->getType()); //建立临时变量, 并存入临
                时变量符号表
11          SymbolEntry *addr_se = new TemporarySymbolEntry(type, SymbolTable::
                getLabel());
12          Operand* addr = new Operand(addr_se);
13          Instruction *allo = new AllocaInstruction(addr, se); //声明这个临时变
                量类型的指令
14          entry->insertFront(allo);
15          se->setAddr(addr);
16
17          new StoreInstruction(addr, idop, entry); //如果有赋值表达式, 那么需要
                store指令
18      }
19  }
```

输入输出由于之前在 lexer.l 中存到全局符号表了，相当于函数定义过了，之后函数调用和普通的函数一样已经实现了，因此只需要在 Ast 生成中间代码时输出库函数的定义语句即可。

（四） 目标代码生成部分

在目标代码生成阶段，主要需要完成基础指令的翻译部分，以及和中间代码类似的结构从 MachineUnit 到 MachineFunction 到 MachineBlock 再到 MachineInstruction 的输出 output() 函数，以及寄存器分配部分。其中我主要负责变量常量定义赋值部分（包括全局变量常量的实现）、函数定义和调用部分、输入输出部分的翻译工作。

全局变量常量的实现在变量定义 Declstmt 语法树结点将变量的符号表值存到了 MachineUnit 的全局变量容器 global_list 中。之后在 MachineUnit 的输出函数中在.text 段之前先进行了全局变量的声明，在 PrintGlobalDecl() 函数中实现，并且在遍历所有 MachineFunction 的输出函数之后，在最后也进行了全局变量的定义，在 printGlobal() 函数中实现。

由于函数定义需要知道栈空间的大小，因此函数的定义部分在 MachineFunction 和 MachineBlock 的输出函数中做了预备工作和收尾工作。在 MachineFunction 的输出函数中，首先 push 指令输出保存的寄存器和 fp、lr 寄存器，之后 sp 赋值给 fp，栈顶变栈底，之后用 sub 指令对 sp 操作拓展栈空间。之后就是遍历 MachineBlock，实现输出函数。在 MachineBlock 的输出函数中进行函数收尾工作，遍历所有存储的指令，当遇到 bx 指令时，代表函数的结束，因此 pop 出保存到寄存器和 fp、lr 寄存器；当遇到 store 指令且参数大于 4 时，需要参数到 r3 寄存器中；当遇到 add 指令且下一条时 bx 指令时说明这是在调整栈空间，需要 sp 加上栈空间偏移量。

函数的调用主要在 CallInstruction 指令中实现。首先处理函数调用的参数，前四个参数分别通过 r0-r3 来传递，之后的参数都先 mov 到虚拟寄存器中，并将虚拟寄存器 push 到栈中，在参数传递时如果是立即数且绝对值超过 255，需要先 load 到寄存器中再进行传递。然后调用分支指令 bl 跳转到要调用的函数，执行完该指令后，需要释放之前参数大于 4 时占用的栈空间，并

在最后将返回值 mov 到 r0 中，传回函数调用的地方。

输入输出由于之前在 lexer.l 中存到全局符号表了，相当于函数定义过了，之后函数调用目标代码能正确实现即可。

四、 总结

编译器最终实现，通过了 76 个样例，基本要求部分完成了，进阶要求部分实现了变量、常量作用域的区别，break、continue 语句，非叶函数，比较遗憾的是没能实现浮点和数组，之后会继续研究完善。

这次编译大作业维持了整整一个学期，自己动手实现一个编译器，让我对编译器有了更加深入的了解，同时也更加熟悉了 linux 系统，收获颇多。在小组合作中，我也学会了合理规划分工，培养了小组之间沟通的能力。