南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

编译原理期末实验报告

编译原理——SysY 编译器

姓名：罗昕珂

学号：2013622

专业：计算机科学与技术

指导教师：王刚

2023 年 1 月 14 日

摘要

使用了通用工具 Lex 和 Bison 进行词法分析和语法分析，产生相应的 C++ 代码，建立抽象语法树详细地记录了函数定义时变量的类型信息，可以进行一定的类型检查使用了一种扩展性很强的内部数据结构存储指令。实现了满足 SysY 文法，生成目标代码的编译器。词法分析、语法分析、错误处理、中间代码生成和目标代码生成等五个阶段实现了编译器。

关键字：SysY 编译器，词法分析，语法分析，中间代码生成，lex ，Yacc

目录

一、 概述

(一) 语言处理系统

语言处理系统的完整工作过程:

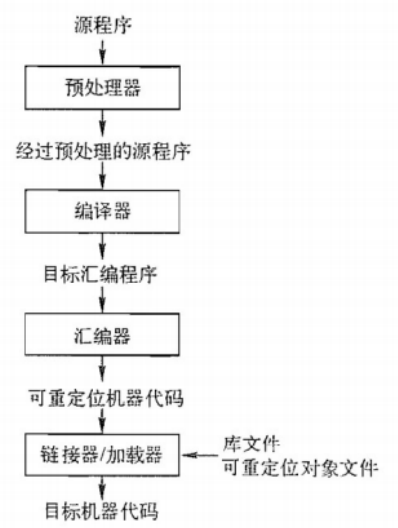

图 1: 处理流程

预处理器: 处理源代码中以 # 开始的预编译指令,例如展开所有宏定义、插入 include 指向的文件等，以获得经过预处理的源程序。

编译器: 将预处理器处理过的源程序文件翻译成为标准的汇编语言以供计算机阅读。

汇编器: 将汇编语言指令翻译成机器语言指令，并将汇编语言程序打包成可重定位目标程序。汇编过程实际上把汇编语言程序代码翻译成目标机器指令的过程。其最终生成的是可重定位的机器代码。

链接器: 将可重定位的机器代码和相应的一些目标文件以及库文件连接在一起，形成真正能在机器上运行的目标机器代码。由汇编程序生成的目标文件不能够直接执行。大型程序经常被分成多个部分进行编译，因此，可重定位的机器代码有必要和其他可重定位的目标文件以及库文件链接到一起，最终形成真正在机器上运行的代码。进而连接器对该机器代码进行执行生成可执行文件。

(二) 编译器

词法分析: lex，这里使用的是 linux 上的 flex

这一过程完成了单词 token 的提取。如果代码中有八进制或十六进制字面常量，这一步将全部转换到十进制

语法分析: yacc，这里使用的是 linux 上的 bison

这一过程先使用 bison 构造 LALR 分析表，然后表驱动翻译 c 代码到抽象语法树。

完成了标识符作用域的分析，将检查标识符重定义和未定义等错误。

语义分析: typecheck() 函数

主要是重新遍历语法树，进行类型检查和检查诸如 break、continue 是否处于循环体内部等语法分析难以检查的语法错误。

中间代码生成：genCode() 函数

用语法树和符号表中信息来检查源程序是否与语言定义语义一致，进行类型检查。

目标代码生成：genMachineCode() 函数

对中间代码进行自顶向下的遍历，从而生成目标代码。

二、 编译器设计

(一) 词法分析

将源程序转换为单词序列。词法分析的任务是对文法进行了解和分析，并对出现的单词进行分析和记录，为后续的语法分析铺垫。整个词法分析思路很清晰：读入文件，逐个字符进行处理和判断并生成对应的 token，为后面的分析提供获取下一个 token 的调用接口。

1. 上下文无关文法

上下文无关文法是一种用于描述程序设计语言语法的表示方式。一般来说，一个上下文无关文法（context-free grammar）由四个元素组成：

（1）一个终结符号集合 VT，它们有时也称为“词法单元”。终结符号是该文法所定义的语言的基本符号的集合。

（2）一个非终结符号集合 VN，它们有时也称为“语法变量”。每个非终结符号表示一个终结符号串的集合。

（3）一个产生式集合 P，其中每个产生式包括一个称为产生式头或左部的非终结符号，一个箭头，和一个称为产生式体或右部的由终结符号及非终结符号组成的序列。产生式主要用来表示某个语法构造的某种书写形式。如果产生式头非终结符号代表一个语法构造，那么该产生式体就代表了该构造的一种书写方式。

（4）指定一个非终结符号为开始符号 S。

因此，上下文无关文法可以通过 (VT ,VN ,P,S) 这个四元式定义。在描述文法时，我们将数位、符号和黑体字符串看作终结符号，将斜体字符串看作非终结符号，以同一个非终结符号为头部的多个产生式的右部可以放在一起表示，不同的右部之间用符号 | 分隔。

2. CFG 描述 SysY 语言特性

$$type \rightarrow basic|constructor$$

$$basic \rightarrow bool|char|int|real|type\_error|void$$

$$constructor \rightarrow array|X|record|pointer|func$$

$$array \to array(I, type)$$

$$I \to \{num(min, max)\}$$

$$X \to type \times type$$

$$record \to record(variables)$$

$$variables \to variables \times variable$$

$$variable \to id \times type$$

$$pointer \to pointer(type)$$

$$func \to (params) \to type$$

$$params \to params \times type | type$$

(二) 语法分析

将词法分析生成的词法单元来构建抽象语法树（Abstract Syntax Tree，即 AST）。

语法分析是对源程序进行递归下降分析，并对语法成分进行判断和输出。

按照文法要求，根据词法分析程序得到的 token 对源程序进行分析和判断。

回溯问题。

在一些分支较多的文法分析中，往往需要进行偷看下一个，或者更多个单词来进行判断需要使用哪个分支进行分析。在经过偷看一个或多个单词并确定分支之后，有两种做法，一种是将当前单词分析的过程回溯到偷看之前的状态并直接进行正确的分支的分析；一种是不进行回溯状态，在当前确定分支之后，直接进行继续分析。两种做法都可以，在有回溯的代码中，由于每次都会回溯到偷看之前的状态，可以直接进行正确的分支运行，并不会多读或漏读，结构更加清晰，但会面临着回溯导致的多余内存和时间的开销。

1. 类型系统

类型系统是编程语言理论的一个重要一部分。类型系统与数理逻辑紧密相关，类型的检查可以视为定理的证明，根据你的目标语言设计好需要的类型系统有关的数据结构。实现了 int、void 和函数类型，const、数组、float、string 等类型。

2. 符号表

栈式符号表。在 SysY 文法中是允许每一个 block 是一个作用域的。

其中内层的 a 作用会覆盖外层的 a。所以需要考虑符号表的栈式维护，也即每出现一个新的作用域（fun，block）便创建一个新的符号表加入栈中，每结束一个作用域便将符号表栈的栈顶弹出。

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，我们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。在语法分析阶段，清楚的知道一个程序的语法结构，如果该标识符用于声明，那么语法分析器将创建相应的符号表条目，并将该条目存入当前作用域对应的符号表中，如果是使用该标识符，将从当前作用域对应的符号表开始沿着符号表链搜索符号表项。

3. 抽象语法树

语法分析的目的是构建出一棵抽象语法树（AST），因此我们需要设计语法树的结点。结点分为许多类，除了一些共用属性外，不同类结点有着各自的属性、各自的子树结构、各自的函数实现。我们可以简单用 struct 去涵盖所有需要的内容，也可以设计复杂的继承结构。结点的类型大体上可以分为表达式和语句，每种类型又可以分为许多子类型。

4. 基本块

block 为语句块，可以是单个 statement
expression 为表达式，可以是任何变量、常量、字面常量、运算语句，不限类型
if (condition) block
if (condition) block else block
while (condition) block
for (expression; condition; expression) block
for (variable declaration; condition; expression) block

(三) 类型检查

使用语法树和符号表中信息来检查源程序是否与语言定义语义一致，进行类型检查等。在语法分析实验的基础上，遍历语法树，进行简单的类型检查，对于语法错误的情况简单打印出提示信息。

错误处理的主要任务是对源程序中出现的语法错误进行处理并对错误进行处理和跳过。由于进行错误处理需要建立符号表结构，错误处理语法分析和代码生成之间的过渡阶段，通过对错误的处理来帮助建立符号表的基础结构和逻辑，为代码生成做铺垫。

(四) 中间代码生成

IRBuilder.h 为中间代码构造辅助类，Unit.h 为编译单元 Function.h 为函数 BasicBlock.h 实现基本块 Instruction.h 生成指令 Operand.h 生成指令操作数。

AsmBuilder.h 中为汇编代码构造辅助类。其主要作用就是在中间代码向目标代码进行自顶向下的转换时，记录当前正在进行转换操作的对象，以进行函数、基本块及指令的插入。

MachineCode.h 中为汇编代码构造相关的框架，大体的结构和中间代码是类似的，只有具体到汇编指令和对应操作数时有不同之处。

LiveVariableAnalysis.h 为活跃变量分析，用于寄存器分配过程。

LinearScan.h 为线性扫描寄存器分配算法相关类，为虚拟寄存器分配物理寄存器。

四元式并生成了中间代码。中间代码的本质是将语法分析和符号表中的信息保存起来，将整个目标代码的生成与语法分析、词法分析分离开来。当拿到了中间代码时，可以根据中间代码直接生成目标代码而不再依赖于词法和语法分析。这样在生成目标代码时逻辑更加的清晰和简单：仅仅需要考虑如何生成目标代码。

中间代码的信息一定要足够生成目标代码。中间代码是为了生成目标代码，所以在设计中间代码的四元式表达式时，要考虑到生成目标代码所需要的全部信息，比如函数定义、变量/常量初始化、各种跳转等，需要生成足够的中间代码才可以保证目标代码的生成。

短路逻辑。短路逻辑的设计的关键在于如何设计好跳转逻辑。短路逻辑的跳转是完全靠中间代码的四元式操作符来进行标记，设计一个比较合理的跳转逻辑。

词法分析和语法分析是编译器的前端，中间代码是编译器的中端，目标代码是编译器的后端，通过将不同源语言翻译成同一中间代码，再基于中间代码生成不同架构的目标代码，我们可以极大的简化编译器的构造。中间代码生成的总体思路是对抽象语法树作一次遍历，遍历的过程中需要根据综合属性和继承属性来生成各结点的中间代码，在生成完根结点的中间代码后便得到了最终结果。

（五） 目标代码的生成

目标代码其中有一系列的指令是编译器指令，作用是告知编译器要如何编译，通常以. 开始，其他指令则为汇编指令。在中间代码生成之后，大家就可以对中间代码进行自顶向下的遍历，从而生成目标代码。

AsmBuilder.h 汇编代码构造辅助类 MachineCode.h 汇编代码构造相关类 LinearScan.h 线性扫描寄存器分配相关类 LiveVariableAnalysis.h 活跃变量分析相关类

函数调用函数调用的本质是在栈空间中开辟新的作用域，同时在跳转到目标函数之前需要完成函数参数的值传递由于需要参数传递，所以 CALL 指令其实不能够立即跳转，只能在 CALL 指令时设置 STR 和 LOAD，目标地址需要等到函数调用的值传递结束才可以确定。在函数的值传递结束之后，直接执行无条件跳转指令即可。

作用域增加。在 SysY 文法中每一个 block 会声明一个新的作用域，但并不是函数跳转，所以并不能使用 CALL 指令，但其行为又很类似 CALL 指令，笔此处新声明了一个 block 作用域，除此之外没有别的用处，以及一个新的指令来标志 block 声明的作用域结束，BLE 指令是作用域结束，只需更新 B 为当前作用域，并相应的更新栈顶。

数组的引用。由于在函数调用时允许数组的引用，所以在函数调用传参时，需要传递的数组的地址，但由于数组在传递过程中可能出现跨多个作用域进行取值，所以数组的引用传递需要传递数组在当前运行栈中的绝对地址，在访问时此引用数组时，将此地址取出并作为基地址进行数组的取值。生成代码时，我们可以拿到数组的头地址在其所在作用域中的地址偏移，以及作用域的头地址的绝对地址，二者相加即可得到绝对地址。

数组的取值。数组取值的过程是通过下标计算出数组元素相对于数组的起始地址的偏移量，然后加上数组的起始地址即可得到，而数组的起始地址需要通过绝对地址来求

得（。由于数组取值时，地址是通过计算出来放到栈顶的，所以需要额外增加指令来允许从栈顶获得目的变量的地址进而从栈中取出地址，同理在向数组元素赋值时，也需要运行时求出地址进行赋值。

跳转指令的回填。在生成跳转指令时，就需要在最后进行跳转指令地址的回填。在目标代码生成的过程中将每一个 label 标签对应的地址进行记录，在全部的目标代码指令生成结束之后，将所有的 label 标签替换为跳转指令的目的地址。

短路逻辑的设计。短路逻辑的本质是将 && 和 || 两个运算符中的表达式分开进行跳转即可。对于 && 当检测到一个表达式为 0 时，直接跳转到下一个 || 即可，对于 || 当检测到一个表达式为 1 时，直接跳转到 cond 结束。

三、 编译器的实现

(一) 词法分析器

将利用 Flex 工具实现词法分析器，识别程序中所有单词，将其转化为单词流。输入是一个 SysY 语言程序，它的输出是每一个文法单元的类别、词素，以及必要的属性。(比如，对于 NUMBER 会有属性它的“数值”属性；对于 ID 会有它在符号表的“序号”，有些标识符会有相同的“序号”。)

1. 定义部分

定义部分包含选项、文字块、开始条件、转换状态、规则等。%option noyywrap 即为一个选项，控制 flex 的一些功能，具体来说，这里的选项功能为去掉默认的 yywrap 函数调用，设计用来对应多文件输入的情况，在每次 yylex 结束后调用。

定义部分

```
%option noyywrap
%option yylineno
```

用%% 包围起来的部分为文字块，可以看到块内可以直接书写 C 代码，Flex 会把文字块内的内容原封不动的复制到编译好的 C 文件中，而%top 块也为文字块，只是 Flex 会将这部分内容放到编译文件的开头，一般用来引用额外的头文件。这部分文件可以直接使用。

定义部分

```
%{
    /*
    * You will need to comment this line in lab5.
    */
    //#define ONLY_FOR_LEX

    #ifdef ONLY_FOR_LEX

```

```
9   #include <math.h>
10  #include <sstream>
11  int cnt=0;
12
13  #else
14  #include "parser.h"
15  #endif
16
17  #define YY_NO_UNPUT
18  #define YY_NO_INPUT
19  #include <string>
20
21  #ifdef ONLY_FOR_LEX
22  #include <ostream>
23  #include <fstream>
24  using namespace std;
25  extern FILE *yyin;
26  extern FILE *yyout;
27
28  struct SymbolTable{
29      int id=0;//作用域编号
30      int num=0;
31      string s[100];
32  }st[100];
33  bool state;
34  int stnum;//结构体复用后的实际序号
35  int stcnt;//为每一个作用域编号
36
37  void DEBUG_FOR_LAB4(std::string s){
38      if(s[0]=='I' && s[1]=='D'){
39          if(state){
40              std::string DEBUG_INFO = s +to_string(yylineno)+"\t"
                    + to_string(cnt) + "\t"+to_string(stcnt)+"\n";
41              fputs(DEBUG_INFO.c_str(), yyout);
42          }else{
43              std::string DEBUG_INFO = s +to_string(yylineno)+"\t"
                    + to_string(cnt) + "\t"+to_string(st[stnum].id)+"\
                    n";
44              fputs(DEBUG_INFO.c_str(), yyout);
45          }
46      }else{
47          std::string DEBUG_INFO = s +to_string(yylineno)+"\t" +
                to_string(cnt) +"\n";
48          fputs(DEBUG_INFO.c_str(), yyout);
```

```
49          }
50
51      }
52      void addChar(int leng){
53          cnt+=leng;
54      }
55      #endif
56  %}
```

在定义部分，我们可以声明一些起始状态，用来限制特定规则的作用范围。用它可以很方便地做一些事情，我们用识别注释段作为一个例子，因为在注释段中，同样会包含数字字母标识符等等元素，但我们不应将其作为正常的元素来识别，这时候通过声明额外的起始状态以及规则会很有帮助。

定义部分

```
1   DIGIT [0-9]
2   DECI ([1-9][0-9]*|0)
3   OCT (0[0-7]+)
4   HEX (0[Xx][[0-9a-fA-F]+)
5   ID [[:alpha:]_][[:alpha:][:digit:]_]*
6   STRING \".*\"
7   EOL (\r\n|\n)
8   WHITE [\t ]
9
10  LPAREN "("
11  RPAREN ")"
12  LBRACK "["
13  RBRACK "]"
14  LBRACE "{"
15  RBRACE "}"
16  SEMI ";"
17  COMMA ","
18
19  EQ "=="
20  GRAEQ ">="
21  LESEQ "<="
22  NEQ "!="
23  ASSIGN "="
24  PLUSASSIGN "+="
25  MINUSASSIGN "-="
26  MULASSIGN "*="
27  DIVASSIGN "/="
28  GRA ">"
29  LESS "<"
```

```
ADD "+"
SUB "-"
MUL "*"
DIV "/"
INCRE "++"
DECRE "--"
MOD "%"
AND "&&"
OR "||"
NOT "!"

CONST "const"
VOID "void"
IF "if"
ELSE "else"
DO "do"
WHILE "while"
FOR "for"
BREAK "break"
CONTINUE "continue"
RETURN "return"
INT "int"
FLOAT "float"


commentbegin "/*"
commentelement .|\n
commentend "*/"
%x COMMENT
commentlinebegin "//"
commentlineelement .
commentlineend \n
%x COMMENTLINE
```

2. 规则部分

规则即为正规定义声明。Flex 除了支持我们学习的正则表达式的元字符包括 [] * + ? | () 以外，还支持像 /^$ 等等元字符。规则部分包含模式行与 C 代码，当存在二义性问题时，Flex 采用两个简单的原则来处理矛盾

1. 匹配输入时匹配尽可能多的字符串——最长前缀原则。
2. 如果两个模式都可以匹配的话，匹配在程序中更早出现的模式。

这里的更早出现，指的就是规则部分对于不同模式的书写先后顺序

规则部分

```
"int" {
    /*
    * Questions:
    *   Q1: Why we need to return INT in further labs?
    *   Q2: What is "INT" actually?
    */
    #ifdef ONLY_FOR_LEX
        DEBUG_FOR_LAB4("INT\t\tint\t\t");
        addChar(yyleng);
    #else
        return INT;
    #endif
}
"float" {
    #ifdef ONLY_FOR_LEX
        DEBUG_FOR_LAB4("FLOAT\t\tfloat\t\t");
        addChar(yyleng);
    #else
        return FLOAT;
    #endif
}

"getarray" {
    char *lexeme;
    lexeme = new char[strlen(yytext) + 1];
    strcpy(lexeme, yytext);
    yylval.strtype = lexeme;
    std::vector<Type*> vec;
    std::vector<SymbolEntry*> vec1;
    ArrayType* arr = new ArrayType(TypeSystem::intType, -1);
    vec.push_back(arr);
    Type* funcType = new FunctionType(TypeSystem::intType, vec,vec1);
    SymbolTable* st = identifiers;
    while(st->getPrev())
        st = st->getPrev();
    SymbolEntry* se = new IdentifierSymbolEntry(funcType, yytext, st
       ->getLevel(), -1, true);
    st->install(yytext, se);
    return ID;
}

{ID} {
    #ifdef ONLY_FOR_LEX
```

```
43          string str=yytext,str1="ID\t\t";
44          str1+=str;
45          if(str.length()%8==0)str1+="\t";
46          else str1+="\t\t";
47
48          bool isIn=false;
49          for(int i=0;i<=st[stnum].num;i++){
50              if(st[stnum].s[i].compare(str)==0){
51                  isIn=true;
52                  break;
53              }
54          }
55          if(!isIn){
56              st[stnum].s[st[stnum].num]=str;
57              st[stnum].num++;
58          }
59          DEBUG_FOR_LAB4(str1);
60          addChar(yyleng);
61      #else
62          char *lexeme;
63          lexeme = new char[strlen(yytext) + 1];
64          strcpy(lexeme, yytext);
65          yylval.strtype = lexeme;
66          return ID;
67      #endif
68  }
69
70  {STRING} {
71      char* lexeme;
72      lexeme = new char[strlen(yytext) + 1];
73      strcpy(lexeme, yytext);
74      yylval.strtype = lexeme;
75      return STRING;
76  }
77
78  {EOL} {
79      #ifdef ONLY_FOR_LEX
80      cnt=0;
81      #endif
82  }
83  {WHITE} {
84      #ifdef ONLY_FOR_LEX
85      addChar(1);
86      #endif
```

```
}

{commentbegin} {BEGIN COMMENT;}
<COMMENT>{commentelement} {}
<COMMENT>{commentend} {BEGIN INITIAL;}
{commentlinebegin} {BEGIN COMMENTLINE;}
<COMMENTLINE>{commentlineelement} {}
<COMMENTLINE>{commentlineend} {BEGIN INITIAL;}
```

3. 用户子例程

用户子例程的内容会被原样拷贝至 C 文件，通常包括规则中需要调用的函数。在主函数中通过调用 yylex 开始词法分析的过程，

用户子例程

```
#ifdef ONLY_FOR_LEX
int main(int argc, char **argv){
    if(argc != 5){
        fprintf(stderr, "Argument Not Enough");
        exit(EXIT_FAILURE);
    }

    if(!(yyin = fopen(argv[1], "r"))){
        fprintf(stderr, "No such file or directory: %s", argv[1]);
        exit(EXIT_FAILURE);
    }

    if(!(yyout = fopen(argv[3], "w"))){
        fprintf(stderr, "No such file or directory: %s", argv[3]);
        exit(EXIT_FAILURE);
    }
    fputs("token\t\tlexeme\t\tlineno\toffset\tpointer_to_scope\n",
        yyout);
    yylex();
    return 0;
}
#endif
```

4. Bison 软件

将所有的词法分析功能均放在 yylex 函数内实现，为 +、-、、(、) 每个运算符及整数分别定义一个单词类别，在 yylex 内实现代码，能识别这些单词，并将单词类别返回

给词法分析程序。实现功能更强的词法分析程序，可识别并忽略空格、制表符、回车等空白符，能识别多位十进制整数。

(二) 语法分析器

1. 语法树的创建

Node 为 AST 结点的抽象基类，ExprNode 为表达式结点的抽象基类，从 ExprNode 中派生出 Id。Node 类中声明了纯虚函数 output，用于输出语法树信息，派生出的具体子类均需要对其进行实现。

语法树的创建

```
class Ast
{
private:
    Node* root;
public:
    Ast() {root = nullptr;}
    void setRoot(Node*n) {root = n;}
    void output();
    void typeCheck(Type* retType = nullptr);
    void genCode(Unit *unit);
};

class Node
{
private:
    static int counter;
    int seq;
    Node* next;
protected:
    std::vector<Instruction*> true_list;
    std::vector<Instruction*> false_list;
    static IRBuilder *builder;
    void backPatch(std::vector<Instruction*> &list, BasicBlock*bb);
    std::vector<Instruction*> merge(std::vector<Instruction*> &list1,
        std::vector<Instruction*> &list2);
public:
    Node();
    int getSeq() const {return seq;};
    static void setIRBuilder(IRBuilder*ib) {builder = ib;};
    virtual void output(int level) = 0;
    void setNext(Node* node);
    virtual void typeCheck(Type* retType = nullptr) = 0;
    virtual void genCode() = 0;
```

```
    Node* getNext() { return next; }
    std::vector<Instruction*>& trueList() {return true_list;}
    std::vector<Instruction*>& falseList() {return false_list;}
};
```

2. Yacc 工具

词法分析得到的，实质是语法树的叶子结点的属性值，语法树所有结点均由语法分析器创建。在自底向上构建语法树时（与预测分析法相对），我们使用孩子结点构造父结点。在 yacc 每次确定一个产生式发生归约时，我们会创建出父结点、根据子结点正确设置父结点的属性、记录继承关系：

语法树的创建

```


Stmt
    : AssignStmt {$$=$1;}
    | ExprStmt {$$=$1;}
    | BlockStmt {$$=$1;}
    | BlankStmt {$$=$1;}
    | IfStmt {$$=$1;}
    | WhileStmt {$$=$1;}
    | ReturnStmt {$$=$1;}
    | BreakStmt {
        if(!whileCnt)
            fprintf(stderr, "\'break\' statement not in while
                statement\n"); //while计数器为0 表示不在while循环中
        $$=$1;
    }
    | ContinueStmt {
        if(!whileCnt)
            fprintf(stderr, "\'break\' statement not in while
                statement\n"); //同上
        $$=$1;
    }
    | DeclStmt {$$=$1;}
    | FuncDef {$$=$1;}
    ;
WhileStmt
    : WHILE LPAREN Cond RPAREN {
        whileCnt++;
        WhileStmt *whileNode = new WhileStmt($3);
        $<stmttype>$ = whileNode;
        whileStk.push(whileNode);
```

```
    }
    Stmt{
        StmtNode *whileNode = $<stmttype>5;
        ((WhileStmt*)whileNode)->setStmt($6);
        $$=whileNode;
        whileStk.pop();
        whileCnt--;
    }
    ;
ReturnStmt
    :
    RETURN SEMI {
        $$ = new ReturnStmt();
    }
    |
    RETURN Exp SEMI {
        $$ = new ReturnStmt($2);
    }
    ;
BreakStmt
    : BREAK SEMI {
       $$ = new BreakStmt(whileStk.top());
    }
    ;
ContinueStmt
    : CONTINUE SEMI {
        $$ = new ContinueStmt(whileStk.top());
    }
    ;
```

借助 Yacc 工具实现语法分析器:

- 语法树数据结构的设计：结点类型的设计，不同类型的节点应保存的信息。
- 扩展上下文无关文法，设计翻译模式。
- 设计 Yacc 程序，实现能构造语法树的分析器。
- 以文本方式输出语法树结构，验证语法分析器实现的正确性。

3. 符号表

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。

在 SymbolTable.cpp 中实现符号表的查找函数:

SymbolTable

```

```

```
SymbolEntry* SymbolTable::lookup(std::string name)
{
    // Todo
    if(symbolTable.count(name) > 0){
        return symbolTable[name];
    }
    else{
        SymbolTable *mprev=getPrev();
        if(mprev!=nullptr){
           return  mprev->lookup(name);
        }
    }
    return nullptr;
}
```

setNext 函数和 install 函数维护每个符号表

SymbolTable

```

// install the entry into current symbol table.
bool SymbolTable::install(std::string name, SymbolEntry* entry)
{
    //检查是否声明以及重定义
    //symbolTable[name] = entry;
    if (this->symbolTable.find(name) != this->symbolTable.end()) {
        SymbolEntry* se = this->symbolTable[name];
        if (se->getType()->isFunc())
            return se->setNext(entry);
        return false;
    } else {
        symbolTable[name] = entry;
        return true;
    }
}

bool SymbolEntry::setNext(SymbolEntry* se) {
    SymbolEntry* s = this;
    long unsigned int cnt =
        ((FunctionType*)(se->getType()))->getParamsType().size();
    if (cnt == ((FunctionType*)(s->getType()))->getParamsType().size
        ())
        return false;
    while (s->getNext()) {
        if (cnt == ((FunctionType*)(s->getType()))->getParamsType().
            size())
```

```
            return false;
        s = s->getNext();
    }
    if (s == this) {
        this->next = se;
    } else {
        s->setNext(se);
    }
    return true;
}
```

(三) 类型检查

1. 变量未声明，及在同一作用域下重复声明
2. 条件判断表达式 int 至 bool 类型隐式转换
3. 数值运算表达式运算数类型是否正确
4. 函数未声明，及不符合重载要求的函数重复声明
5. 函数调用时形参及实参类型或数目的不一致

6.return 语句操作数和函数声明的返回值类型是否匹配

例：非 void 函数需要有 return 语句以及函数语句块为空时必须是 void 函数

typeCheck

```
void FunctionDef::typeCheck(Type* retType)
{

    SymbolEntry* se = this->getSymbolEntry();
    Type* ret = ((FunctionType*)(se->getType()))->getRetType();
    StmtNode* stmt = this->stmt;
    if (stmt == nullptr) {
        if (ret != TypeSystem::voidType)
            fprintf(stderr, "non-void funtion must return a value\n")
                ;
    }
    stmt->typeCheck(ret);
    if(!isret && ret != TypeSystem::voidType){   //不是ret语句
        fprintf(stderr, "function does not have a return statement\n"
            );
    }
    isret=0;
}
```

typeCheck

```
void IfStmt::typeCheck(Type* retType)
```

```
{
    if (thenStmt)
        thenStmt->typeCheck(retType);
}



void IfElseStmt::typeCheck(Type* retType)
{
    if (thenStmt)
        thenStmt->typeCheck(retType);
    if (elseStmt)
        elseStmt->typeCheck(retType);
}

void CompoundStmt::typeCheck(Type* retType)
{
    if (stmt)
        stmt->typeCheck(retType);
}

void SeqNode::typeCheck(Type* retType)
{
    //分别对stmt1和stmt2进行类型检查
    if(stmt1){
        stmt1->typeCheck(retType);
    }
    if(stmt2){
        stmt2->typeCheck(retType);
    }
}
```

判断 return 语句返回类型与函数声明类型是否匹配，函数类型 retType，return 语句 retValue

typeCheck

```
void ReturnStmt::typeCheck(Type* retType)
{
    // Todo

    bool err=false;
    if (!retType) {
        fprintf(stderr, "expected unqualified-id\n");
        err=true;
```

```
    }
    else if (!retValue && !retType->isVoid()) {//函数要求返回，但未return
        fprintf(stderr,"return-statement with no value, in function returning \'%s\'\n",retType->toStr().c_str());
        err=true;
    }
    else if (retValue && retType->isVoid()) {//函数是void，却有返回值
        fprintf(stderr,"return-statement with a value, in function returning \'void\'\n");
        err=true;
    }
    if(retValue){
        Type* type = retValue->getType();
        if (!err && type != retType) { //返回类型不匹配,最后再判定,保证二者有效
            fprintf(stderr,"cannot initialize return object of type \'%s\' with an rvalue of type \'%s\'\n",retType->toStr().c_str(), type->toStr().c_str());
        }
    }
    err = false;
    isret=1;
}
```

(四) 中间代码生成

1. 表达式的翻译

首先通过 builder 得到后续生成的指令要插入的基本块 bb，然后生成子表达式的中间代码，通过 getOperand 函数得到子表达式的目的操作数，设置指令的操作码，最后生成相应的二元运算指令并插入到基本块 bb 中。

Listing:

```
void BinaryInstruction::output() const
{
    std::string s1, s2, s3, op, type;
    s1 = operands[0]->toStr();
    s2 = operands[1]->toStr();
    s3 = operands[2]->toStr();
    type = operands[0]->getType()->toStr();
    switch (opcode)
    {
    case ADD:
```

```
        op = "add";
        break;
    case SUB:
        op = "sub";
        break;
    case MUL:
        op = "mul";
        break;
    case DIV:
        op = "sdiv";
        break;
    case MOD:
        op = "srem";
        break;
    default:
        break;
    }
    fprintf(yyout, "  %s = %s %s %s, %s\n", s1.c_str(), op.c_str(),
        type.c_str(), s2.c_str(), s3.c_str());
}
```

2. 控制流表达式的翻译

我们为每个结点设置两个综合属性 true_list 和 false_list, 它们是跳转目标未确定的跳转指令的列表，回填是指当跳转的目标基本块确定时，设置列表中跳转指令的跳转目标为该基本块。

bool 表达式

```
BasicBlock *bb = builder->getInsertBB();
Function *func = bb->getParent();
BasicBlock *trueBB = new BasicBlock(func);
expr1->genCode();
backPatch(expr1->trueList(), trueBB);
builder->setInsertBB(trueBB);
expr2->genCode();
true_list = expr2->trueList();
false_list = merge(expr1->falseList(), expr2->falseList());
```

3. 控制流的翻译

生成 cond 结点的中间代码，cond 为真时将跳转到基本块 then_bb，cond 为假时将跳转到基本块 end_bb，进行回填。设置插入点为基本块 then_bb，然后生成 thenStmt 结点的中间代码。

控制流

```

void CondBrInstruction::output() const
{
    std::string cond, type;
    cond = operands[0]->toStr();
    type = operands[0]->getType()->toStr();
    int true_label = true_branch->getNo();
    int false_label = false_branch->getNo();
    fprintf(yyout, "  br %s %s, label %%B%d, label %%B%d\n", type.
        c_str(), cond.c_str(), true_label, false_label);
}
Function *func;
BasicBlock *then_bb, *end_bb;
func = builder->getInsertBB()->getParent();
then_bb = new BasicBlock(func);
end_bb = new BasicBlock(func);
cond->genCode();
backPatch(cond->trueList(), then_bb);
backPatch(cond->falseList(), end_bb);
builder->setInsertBB(then_bb);
 thenStmt->genCode();
 then_bb = builder->getInsertBB();
new UncondBrInstruction(end_bb, then_bb);
builder->setInsertBB(end_bb);
```

（五） 目标代码生成

在中间代码生成之后，大家就可以对中间代码进行自顶向下的遍历，从而生成目标代码。

1. 汇编代码

变量、常量的声明和初始化

声明和初始化

```
MachineOperand* Instruction::genMachineOperand(Operand* ope) {
    auto se = ope->getEntry();
    MachineOperand* mope = nullptr;
    if (se->isConstant()) //常数
        mope = new MachineOperand(
            MachineOperand::IMM,
            dynamic_cast<ConstantSymbolEntry*>(se)->getValue());
    else if (se->isTemporary()) //临时变量
```

```
        mope = new MachineOperand(
            MachineOperand::VREG,
            dynamic_cast<TemporarySymbolEntry*>(se)->getLabel());
    else if (se->isVariable()) {  //变量
        auto id_se = dynamic_cast<IdentifierSymbolEntry*>(se);
        if (id_se->isGlobal())  //全局变量
            mope = new MachineOperand(id_se->toStr().c_str());
        else if (id_se->isParam()) { //含参函数, 使用 R0-R3 寄存器传递参数
            if (id_se->getParamCount() < 4)
                mope = new MachineOperand(MachineOperand::REG,id_se->getParamCount());
            else//参数个数大于四个
                mope = new MachineOperand(MachineOperand::REG, 3);
        } else
            exit(0);
    }
    return mope;
}
```

即声明为代码段，将函数名添加到全局符号表中，声明类型为函数。

对每个函数的声明

```
.text
.global functionname
.type functionname, %function
```

访存指令

分为三种情况：存储一个常量，存储一个栈中的临时变量，存储一个全局变量，存储一个数组元素

访存指令

```
StoreMInstruction::StoreMInstruction(MachineBlock* p,MachineOperand* src1, MachineOperand* src2, MachineOperand* src3, int cond)
{
    // TODO
    // str 源寄存器 存储地址
    this->parent = p;
    this->type = MachineInstruction::STORE;
    this->op = -1;
    this->cond = cond;
    this->use_list.push_back(src1);
    this->use_list.push_back(src2);
    if (src3)
        this->use_list.push_back(src3);
```

```
    src1->setParent(this);
    src2->setParent(this);
    if (src3)
        src3->setParent(this);
}

void StoreMInstruction::output()
{
    // TODO
    //str r4, [fp, #-8]
    //str r4, [fp]
    fprintf(yyout, "\tstr ");
    this->use_list[0]->output();
    fprintf(yyout, ", ");
    if (this->use_list[1]->isReg() || this->use_list[1]->isVReg())
        fprintf(yyout, "[");
    this->use_list[1]->output();
    if (this->use_list.size() > 2) {
        fprintf(yyout, ", ");
        this->use_list[2]->output();
    }
    if (this->use_list[1]->isReg() || this->use_list[1]->isVReg())
        fprintf(yyout, "]");
    fprintf(yyout, "\n");
}

void StoreInstruction::genMachineCode(AsmBuilder* builder) {
    auto cur_block = builder->getBlock();
    MachineInstruction* cur_inst = nullptr;
    auto dst = genMachineOperand(operands[0]);
    auto src = genMachineOperand(operands[1]);
    // 存储一个常量
    if (operands[1]->getEntry()->isConstant()) {
        auto dst1 = genMachineVReg();
        cur_inst = new LoadMInstruction(cur_block, dst1, src);
        cur_block->InsertInst(cur_inst);
        // src = dst1;
        src = new MachineOperand(*dst1);
    }
    // 存储一个栈中的临时变量
    if (operands[0]->getEntry()->isTemporary() && operands[0]->getDef
        () &&
        operands[0]->getDef()->isAlloc()) {
        auto src1 = genMachineReg(11);
```

```
56          int off = dynamic_cast<TemporarySymbolEntry*>(operands[0]->
                getEntry())->getOffset();
57          auto src2 = genMachineImm(off);
58          if (off > 255 || off < -255) {
59              auto operand = genMachineVReg();
60              cur_block->InsertInst((new LoadMInstruction(cur_block,
                    operand, src2)));
61              src2 = operand;
62          }
63          cur_inst = new StoreMInstruction(cur_block, src, src1, src2);
64          cur_block->InsertInst(cur_inst);
65      }
66      // 存储一个全局变量
67      else if (operands[0]->getEntry()->isVariable() &&dynamic_cast<
            IdentifierSymbolEntry*>(operands[0]->getEntry())->isGlobal())
            {
68          auto internal_reg1 = genMachineVReg();
69          // example: load r0, addr_a
70          cur_inst = new LoadMInstruction(cur_block, internal_reg1, dst
                );
71          cur_block->InsertInst(cur_inst);
72          // example: store r1, [r0]
73          cur_inst = new StoreMInstruction(cur_block, src,
                internal_reg1);
74          cur_block->InsertInst(cur_inst);
75      }
76      // 存储一个数组元素
77      else if (operands[0]->getType()->isPtr()) {
78          cur_inst = new StoreMInstruction(cur_block, src, dst);
79          cur_block->InsertInst(cur_inst);
80      }
81  }
```

内存分配指令

为指令的目的操作数在栈内分配空间，将其相对于 FP 寄存器的偏移存在符号表中。

内存分配指令

```
1   void AllocaInstruction::genMachineCode(AsmBuilder* builder) {
2       /* HINT:
3        * Allocate stack space for local variabel
4        * Store frame offset in symbol entry */
5       auto cur_func = builder->getFunction();
6       int size = se->getType()->getSize() / 8;
7       if (size < 0)
```

```
        size = 4;
    int offset = cur_func->AllocSpace(size);
    dynamic_cast<TemporarySymbolEntry*>(operands[0]->getEntry())->
        setOffset(-offset);
}
```

二元运算指令

二元运算指令

```
void BinaryInstruction::genMachineCode(AsmBuilder* builder) {
    // complete other instructions
    auto cur_block = builder->getBlock();
    auto dst = genMachineOperand(operands[0]);
    auto src1 = genMachineOperand(operands[1]);
    auto src2 = genMachineOperand(operands[2]);
    MachineInstruction* cur_inst = nullptr;
    if (src1->isImm()) {
        auto internal_reg = genMachineVReg();
        cur_inst = new LoadMInstruction(cur_block, internal_reg, src1
            );
        cur_block->InsertInst(cur_inst);
        src1 = new MachineOperand(*internal_reg);
    }
    if (src2->isImm()) {
        if ((opcode <= BinaryInstruction::OR &&((ConstantSymbolEntry
            *)(operands[2]->getEntry()))->getValue() >255) || opcode
            >= BinaryInstruction::MUL) {
            auto internal_reg = genMachineVReg();
            cur_inst = new LoadMInstruction(cur_block, internal_reg,
                src2);
            cur_block->InsertInst(cur_inst);
            src2 = new MachineOperand(*internal_reg);
        }
    }
    switch (opcode) {
        case ADD:
            cur_inst = new BinaryMInstruction(cur_block,
                BinaryMInstruction::ADD, dst, src1, src2);
            break;
        case SUB:
            cur_inst = new BinaryMInstruction(cur_block,
                BinaryMInstruction::SUB, dst, src1, src2);
            break;
        case AND:
            cur_inst = new BinaryMInstruction(cur_block,
```

```
                BinaryMInstruction::AND, dst, src1, src2);
            break;
        case OR:
            cur_inst = new BinaryMInstruction(cur_block,
                BinaryMInstruction::OR, dst, src1, src2);
            break;
        case MUL:
            cur_inst = new BinaryMInstruction(cur_block,
                BinaryMInstruction::MUL, dst, src1, src2);
            break;
        case DIV:
            cur_inst = new BinaryMInstruction(cur_block,
                BinaryMInstruction::DIV, dst, src1, src2);
            break;
        case MOD: {
            // MOV dst scr1 scr2
            // dst = scr1/ scr2
            cur_inst = new BinaryMInstruction(cur_block,
                BinaryMInstruction::DIV, dst, src1, src2);
            MachineOperand* dst1 = new MachineOperand(*dst);
            src1 = new MachineOperand(*src1);
            src2 = new MachineOperand(*src2);
            cur_block->InsertInst(cur_inst);
            // dst = dst * scr2
            cur_inst = new BinaryMInstruction(cur_block,
                BinaryMInstruction::MUL, dst1, dst, src2);
            cur_block->InsertInst(cur_inst);
            dst = new MachineOperand(*dst1);
            // dst = scr1 - dst
            cur_inst = new BinaryMInstruction(cur_block,
                BinaryMInstruction::SUB, dst, src1, dst1);
            break;
        }
        default:
            break;
    }
    cur_block->InsertInst(cur_inst);
}
```

控制流指令

对 CondBrInstruction，在 Asm-Builder 中添加成员以记录前一条 CmpInstruction 的条件码,从而在遇到 CondBrInstruction 时生成对应的条件跳转指令跳转到 True Branch，之后需要生成一条无条件跳转指令跳转到 False Branch

二元运算指令

```
void CondBrInstruction::genMachineCode(AsmBuilder* builder) {
    auto cur_block = builder->getBlock();
    std::stringstream s;
    s << ".L" << true_branch->getNo();
    MachineOperand* dst = new MachineOperand(s.str());
    auto cur_inst = new BranchMInstruction(cur_block,
        BranchMInstruction::B,dst, cur_block->getCond());
    cur_block->InsertInst(cur_inst);
    s.str("");
    s << ".L" << false_branch->getNo();
    dst = new MachineOperand(s.str());
    cur_inst = new BranchMInstruction(cur_block, BranchMInstruction::
        B, dst);
    cur_block->InsertInst(cur_inst);
}

void UncondBrInstruction::genMachineCode(AsmBuilder* builder) {
    auto cur_block = builder->getBlock();
    std::stringstream s;
    s << ".L" << branch->getNo();
    MachineOperand* dst = new MachineOperand(s.str());
    auto cur_inst =new BranchMInstruction(cur_block,
        BranchMInstruction::B, dst);
    cur_block->InsertInst(cur_inst);
}

void BranchMInstruction::output()
{
    // TODO
    switch (op) {
        //b{条件} .L23 直接调转
        case B:
            fprintf(yyout, "\tb");
            PrintCond();
            fprintf(yyout, " ");
            this->use_list[0]->output();
            fprintf(yyout, "\n");
            break;
        //bx{条件} lr 带状态切换的跳转。
        case BX:
            fprintf(yyout, "\tbx");
            PrintCond();
            fprintf(yyout, " ");
```

```
41              this->use_list[0]->output();
42              fprintf(yyout, "\n");
43              break;
44          //bl{条件} .L21 带链接的跳转。
45          case BL:
46              fprintf(yyout, "\tbl");
47              PrintCond();
48              fprintf(yyout, " ");
49              this->use_list[0]->output();
50              fprintf(yyout, "\n");
51              break;
52      }
53  }
```

函数定义

生成 PUSH 指令保存 FP 寄存器及一些 Callee Saved 寄存器，之后生成 MOV 指令令 FP 寄存器指向新的栈底，之后需要生成 SUB 指令为局部变量分配栈内空间。

函数定义

```
1  void MachineFunction::output()
2  {
3      fprintf(yyout, "\t.global %s\n", this->sym_ptr->toStr().c_str() +
           1);
4      fprintf(yyout, "\t.type %s , %%function\n",this->sym_ptr->toStr()
           .c_str() + 1);
5      fprintf(yyout, "%s:\n", this->sym_ptr->toStr().c_str() + 1);
6      // TODO
7      //push {fp,lr} 保存 FP 寄存器及一些 CalleeSavedRegs
8      // mov fp, sp 令 FP 寄存器指向新的栈底
9      //sub sp, sp, #12 为局部变量分配栈内空间
10     auto fp = new MachineOperand(MachineOperand::REG, 11);
11     auto sp = new MachineOperand(MachineOperand::REG, 13);
12     auto lr = new MachineOperand(MachineOperand::REG, 14);
13     (new StackMInstrcuton(nullptr, StackMInstrcuton::PUSH,
           CalleeSavedRegs(), fp,lr))->output();  //push保存
14     (new MovMInstruction(nullptr, MovMInstruction::MOV, fp, sp))->
           output();
15     int space = AllocSpace(0);
16     auto spaceSize = new MachineOperand(MachineOperand::IMM, space);
17     if (space < -255 || space > 255) {
18         auto r = new MachineOperand(MachineOperand::REG, 4);
19         (new LoadMInstruction(nullptr, r, spaceSize))->output();
20         (new BinaryMInstruction(nullptr, BinaryMInstruction::SUB, sp,
               sp, r))
```

```
            ->output();
    } else {
        (new BinaryMInstruction(nullptr, BinaryMInstruction::SUB, sp,
            sp, spaceSize))
            ->output();
    }


}
```

函数调用指令

在进行函数调用时，一般前四个函数参数使用 r0-r3 号寄存器进行传参，其余参数压入栈中进行传参，往往按照从右至左的顺序逐个压栈；在函数返回时，一般来讲默认将函数的返回值放到 r0 寄存器中。

函数调用指令

```
void CallInstruction::genMachineCode(AsmBuilder* builder) {
    auto cur_block = builder->getBlock();
    MachineOperand* operand;
    MachineInstruction* cur_inst;
    int idx = 0;
    for (auto it = operands.begin(); it != operands.end(); it++, idx
        ++) {
        if (idx == 0)
            continue;
        if (idx == 5)
            break;
        operand = genMachineReg(idx - 1);
        auto src = genMachineOperand(operands[idx]);
        if (src->isImm() && src->getVal() > 255) {
            cur_inst = new LoadMInstruction(cur_block, operand, src);
        } else
            cur_inst = new MovMInstruction(cur_block, MovMInstruction
                ::MOV, operand, src);
        cur_block->InsertInst(cur_inst);
    }
    //参数个数大于四个还生成 PUSH 指令来传递参数
    for (int i = operands.size() - 1; i > 4; i--) {
        operand = genMachineOperand(operands[i]);
        if (operand->isImm()) {
            auto dst = genMachineVReg();
            if (operand->getVal() < 256)
                cur_inst = new MovMInstruction(cur_block,
                    MovMInstruction::MOV, dst, operand);
```

```
            else
                cur_inst = new LoadMInstruction(cur_block, dst,
                    operand);
            cur_block->InsertInst(cur_inst);
            operand = dst;
        }
        std::vector<MachineOperand*> vec;
        cur_inst = new StackMInstrcuton(cur_block, StackMInstrcuton::
            PUSH, vec,operand);
        cur_block->InsertInst(cur_inst);
    }


    //调用函数 链接跳转
    auto label = new MachineOperand(func->toStr().c_str());
    cur_inst = new BranchMInstruction(cur_block, BranchMInstruction::
        BL, label);
    cur_block->InsertInst(cur_inst);
    if (operands.size() > 5) {
        auto off = genMachineImm((operands.size() - 5) * 4);
        auto sp = new MachineOperand(MachineOperand::REG, 13);
        cur_inst = new BinaryMInstruction(cur_block,
            BinaryMInstruction::ADD,sp, sp, off);
        cur_block->InsertInst(cur_inst);
    }
    if (dst) {
        operand = genMachineOperand(dst);
        auto r0 = new MachineOperand(MachineOperand::REG, 0);
        cur_inst =new MovMInstruction(cur_block, MovMInstruction::MOV
            , operand, r0);
        cur_block->InsertInst(cur_inst);
    }
}
```

2. 寄存器分配

活跃区间分析

在前一步的目标代码生成过程中,已经为所有临时变量分配了一个虚拟寄存器 VREG。在这一步需要为每个 VREG 计算活跃区间。

allocateRegisters

```
void LinearScan::allocateRegisters()
{
    for (auto &f : unit->getFuncs())
```

```
    {
        func = f;
        bool success;
        success = false;
        while (!success)        // repeat until all vregs can be
            mapped
        {
            computeLiveIntervals();
            success = linearScanRegisterAllocation();
            if (success)        // all vregs can be mapped to real
                regs
                modifyCode(); //所有的临时变量都被存在了寄存器
            else                // spill vregs that can't be mapped
                to real regs
                genSpillCode();  //寄存器溢出
        }
    }
}
```

寄存器分配

遍历 intervals 列表进行寄存器分配

linearScanRegisterAllocation

```
bool LinearScan::linearScanRegisterAllocation()
{
    // Todo
    bool success = true;
    active.clear();
    regs.clear();
    for (int i = 4; i < 11; i++)
        regs.push_back(i);
    for (auto& i : intervals) {
        expireOldIntervals(i); //active遍历
        if (regs.empty()) {  //物理寄存器是否空闲
            spillAtInterval(i);
            success = false;
        } else {
            i->rreg = regs.front();
            regs.erase(regs.begin());
            active.push_back(i);
            //按照活跃区间结束位置, 将其插入到 active 列表
            sort(active.begin(), active.end(), compareEnd);
        }
    }
```

```
22      return success;
23  }
```

遍历 active 列表，看该列表中是否存在结束时间早于 unhandled interval 的 interval（即与当前 unhandled interval 的活跃区间不冲突），若有，则说明此时为其分配的物理寄存器可以回收，可以用于后续的分配，需要将其在 active 列表删除；

expireOldIntervals

```
1  void LinearScan::expireOldIntervals(Interval *interval)
2  {
3      // Todo
4      auto it = active.begin();
5      while (it != active.end()) {
6          if ((*it)->end >= interval->start)
7              return;
8          regs.push_back((*it)->rreg);
9          it = active.erase(find(active.begin(), active.end(), *it));
               //回收区间
10         sort(regs.begin(), regs.end());
11     }
12 }
```

如果当前所有物理寄存器都被占用，需要进行寄存器溢出操作。具体为在 active 列表中最后一个 interval 和当前 unhandled interval 中选择一个 interval 将其溢出到栈中，选择策略就是看谁的活跃区间结束时间更晚，如果是 unhandled interval 的结束时间更晚，只需要置位其 spill 标志位即可，如果是 active 列表中的 interval 结束时间更晚，需要置位其 spill 标志位，并将其占用的寄存器分配给 unhandled interval，再按照 unhandled interval 活跃区间结束位置，将其插入到 active 列表中。

spillAtInterval 函数实现

如果当前有可用于分配的物理寄存器，为 unhandled interval 分配物理寄存器之后，再按照活跃区间结束位置，将其插入到 active 列表中即可。

active

```
1              i->rreg = regs.front();
2              regs.erase(regs.begin());
3              active.push_back(i);
4              //按照活跃区间结束位置，将其插入到 active 列表
5              sort(active.begin(), active.end(), compareEnd);
```

生成溢出代码

如果有临时变量被溢出到栈内，，就需要在操作该临时变量时插入对应的 LoadMInstruction 和 StoreMInstruction，其起到的实际效果就是将该临时变量的活跃区间进行了切分，以便重新为其进行寄存器分配。

spillAtInterval

```
void LinearScan::spillAtInterval(Interval *interval)
{
    // Todo
    //所有寄存器都占用, 选择一个 interval 将其溢出到栈中
    auto spill = active.back();  //fp偏移分配空间
    if (spill->end > interval->end) {
        spill->spill = true;
        interval->rreg = spill->rreg; //将active占用的寄存器分配给
            unhandled interval
        active.push_back(interval); //插入到 active 列表
        sort(active.begin(), active.end(), compareEnd);
    } else {
        interval->spill = true;//该区间更晚, 标记spill
    }
}
```

参考文献

[1] 高秀武, 黄亮明, 姜军. 基于 GCC 编译器的流式存储优化方法 [J]. 计算机科学,2022,49(11):76-82.

[2]Paul Krill. Wasmer 2.2 revs compiler, supports Apple M1 and Aarch64[J]. InfoWorld.com,2022.

[3] 邓平. 基于 LLVM 架构的 FT-MX DSP 编译器设计与实现 [D]. 国防科技大学,2021.DOI:10.27052/d.cnki.gzjgu.2021.000106.

[4] 姚化吉. 基于 64 位 Linux 系统的 MSVL 编译器设计开发与测试 [D]. 西安电子科技大学,2020.DOI:10.27389/d.cnki.gxadu.2020.001310.

[5] 叶鹏飞. 一种针对大规模 CGRA 的编译器后端设计 [J]. 现代计算机,2020(06):3-6+18.