|  |
| --- |
| 编译原理课程设计  实验报告  **指导教师：**  **年 级：22级**  **班 级：32班**  **小组编号：**  **组长学号姓名：21220122 张艺卓**  **组员学号姓名：21221020 陆学优**  **组员学号姓名：21220321 李一峰**  **2025 年 4 月 3 日**  **计算机科学与技术学院** |

|  |  |  |
| --- | --- | --- |
| **完成实验内容** | | |
| 一、词法分析模块：  以SNL源程序为输入，通过词法分析得到单词内部表示的Token序列并输出。  二、语法分析模块（递归下降法）：  以词法分析得到的token序列为输入进行递归下降语法分析，建立语法树，若检测到语法错误则输出错误信息。语法分析结束后输出语法树。  三、语义分析模块：  以语法分析得到的语法树为输入生成标识符的符号表以及其他信息表，并进行相关语义检查，若检测到语义错误则输出错误信息。语义分析结束后输出符号表。  四、中间代码生成模块：  以语法分析得到的语法树和语义分析得到的符号表为输入生成四元式形式的中间代码并输出。  五、中间代码优化模块：  以中间代码生成模块生成的四元式为输入，进行常量表达式节省优化处理，优化处理后输出新的四元式中间代码。  六、目标代码生成模块：  以中间代码生成模块生成的四元式为输入，生成Mrs虚拟机可正确运行的Mips目标代码，并输出到文件。 | | |
| **小组成员任务完成情况** | | |
| 姓名 | 具体完成任务 | 工作量  百分比 |
| 张艺卓 | 编程实现中间代码生成模块和常量表达式优化模块，以及输出中间代码四元式；编程实现目标代码生成模块，以及输出Mars虚拟机可正确执行的Mips代码。 | 40% |
| 陆学优 | 编程实现语法分析模块，以及输出语法树结构和语法错误信息。编程实现语义分析模块，以及输出符号表和语义错误信息； | 40% |
| 李一峰 | 编程实现词法分析模块，以及输出Token序列和词法错误信息；完成小组实验报告的撰写；对小组编写的程序进行测试；编写相应的SNL测试文件；编程实现main函数。 | 20% |
| **小组成员协作情况** | | |
| **协作方式：**使用Git进行版本控制，定期进行代码审查和模块联调。  **关键协作点：**  词法分析器与语法分析器的设计（生成以Token为节点的语法树）。  语义分析器与语法树的交互（符号表作用域管理）。  四元式到MIPS指令的映射策略设计（寄存器的分配）。 | | |
| **实验平台与编程语言** | | |
| **编程语言：**Python 3.10  **运行环境：**VS Code  **模拟器：**Mars For Compile（用于运行生成的MIPS代码） | | |
| **实验方案设计** | | |
| **词法分析：**  我们设计了一个 SNLLexer 类，该类利用了 ply 库提供的词法分析工具。每个词法单元都由一个 Token 类实例表示，包含其类型（例如，标识符、关键字、运算符等）、值（例如，标识符的具体名称、运算符符号等）以及在源代码中的行号。我们在 self.tokens 中定义了SNL语言的所有词法单元类型，包括关键字（如 PROGRAM, IF, WHILE 等）、类型（如 INTEGER, CHAR）、标识符和常量（ID, INTC, CHARC）以及各种运算符和符号（如 ASSIGN, EQ, PLUS, LPAREN 等）。  词法分析的核心在于定义识别不同类型词法单元的规则。在我们的实现中，我们使用了正则表达式来描述这些规则。具体地，赋值运算符 := 的规则被定义为 t\_ASSIGN = r':='。对于标识符，我们定义了规则 t\_ID(self, t)，使用正则表达式 r'[a-zA-Z\_][a-zA-Z0-9\_]\*' 来匹配以字母或下划线开头，后跟字母、数字或下划线的字符串。在 t\_ID 方法中，我们还实现了对保留字的处理：通过将匹配到的字符串转换为小写，并在 self.reserved 字典中查找，如果找到匹配项，则将该词法单元的类型设置为对应的关键字类型。对于整型常量和字符常量，我们分别定义了 t\_INTC 和 t\_CHARC 方法，使用相应的正则表达式进行匹配，并将匹配到的字符串转换为整数或去除引号后的字符。我们还定义了 t\_COMMENT 方法来识别并忽略花括号 {} 包裹的注释，以及 t\_newline 方法来识别换行符并更新词法分析器当前的行号。t\_ignore = ' \t' 的设置使得词法分析器能够自动忽略源代码中的空格和制表符。  为了提高编译器的健壮性，我们实现了基本的词法错误处理。当词法分析器遇到无法匹配任何已定义规则的字符时，会调用 t\_error(self, t) 方法，该方法会打印包含错误信息（非法字符及其所在行号）的提示，并跳过该非法字符，继续进行后续的分析。为了方便查看词法分析的结果，我们实现了 analyze\_file 方法。该方法接收输入文件名和输出文件名作为参数，读取输入文件的内容，使用 lexer.input() 方法将数据提供给词法分析器。然后，它循环调用 lexer.token() 方法获取每一个词法单元，并将这些词法单元的信息（行号、值、类型）存储在一个 PrettyTable 对象中，最终将格式化后的词法分析结果写入指定的输出文件。例如，对于 demo.txt 中的输入代码，运行词法分析器后将在 token.txt 文件中生成包含每个词法单元详细信息的表格。  **语法分析：**  语法分析阶段的任务是接收词法分析器生成的词法单元序列，并根据SNL语言的语法规则构建抽象语法树（AST）。我们选择使用自底向上的LR分析方法，并通过 ply 库的 yacc 工具来实现语法分析器。在 SNLParser 类的初始化方法中，我们首先创建了一个 SNLLexer 实例来获取词法单元的定义，并使用 yacc.yacc(module=self) 创建了语法分析器。核心部分在于定义了一系列以 p\_ 开头的函数，每个函数对应SNL语言文法中的一条产生式规则。这些函数通过文档字符串的形式描述了对应的文法规则，并在函数体中定义了如何根据该规则构建AST节点。  我们的语法分析器通过一系列的 p\_ 函数实现了SNL语言的完整文法。比如，程序的基本结构 Program ::= ProgramHead DeclarePart ProgramBody DOT 对应于 p\_Program 函数，该函数将分析得到的 ProgramHead、DeclarePart 和 ProgramBody 作为子节点，构建一个 'Program' 类型的AST节点。对于包含多个选择的文法规则，例如 TypeName ::= BaseType | StructureType | ID，我们定义了多个 p\_TypeName\_ 开头的函数来分别处理不同的选择。对于表示可选部分或重复结构的文法规则，例如 TypeDec ::= epsilon | TypeDeclaration 和 TypeDecList ::= TypeId EQ TypeName SEMI TypeDecMore，我们通过定义对应的 p\_...\_empty 和 p\_...\_list 函数以及递归调用来实现。在每个 p\_ 函数中，我们使用元组来表示AST节点，元组的第一个元素是节点的类型，后续元素是其子节点，这清晰地反映了SNL代码的层次结构。  在语法分析过程中，如果输入的词法单元序列不符合SNL语言的文法规则，ply.yacc 会调用我们定义的 p\_error(self, p) 函数。该函数负责报告语法错误，包括错误发生的 token 值、类型以及所在的行号，从而帮助用户定位代码中的语法问题。为了对输入的SNL源代码进行语法分析，我们实现了 parse\_file 方法。该方法首先读取输入文件的内容，并利用词法分析器得到词法单元序列。然后，它创建了一个词法单元的迭代器，并通过定义 token\_func 函数来逐个获取词法单元，最终将这个 token\_func 传递给 yacc.parse() 方法进行语法分析，生成抽象语法树。我们还提供了一个 print\_ast 函数，用于以易于阅读的缩进格式打印生成的抽象语法树，这对于理解语法分析的结果和调试语法规则非常有帮助。通过运行主程序中的测试代码，我们可以对 demo.txt 文件进行语法分析，并在控制台输出分析结果，包括成功信息和生成的抽象语法树。如果代码中存在语法错误，则会输出相应的错误信息。  **语义分析及中间代码生成：**  语义分析阶段在语法分析的基础上，对程序的结构进行更深层次的检查，以确保程序的逻辑正确性。我们首先定义了 SymbolTable 类，用于管理程序中声明的各种符号（如变量、类型、过程等）及其属性。每个符号表实例代表一个作用域，可以拥有父作用域和子作用域，从而支持嵌套的作用域结构。符号表中存储了符号的类型、在内存中的偏移量以及类别（如变量、过程、形参等）。add\_symbol 方法用于向符号表中添加新的符号，并进行重复定义检查。lookup 方法则用于在当前作用域及其父作用域中查找符号的信息。为了表示不同的数据类型，我们定义了 ProcType、RecType 和 ArrayType 类，分别用于存储过程的参数信息、记录的字段信息和数组的上下界以及元素类型。  SemanticAnalyzer 类是语义分析的核心。在初始化方法中，我们创建了全局作用域的符号表，并使用栈来维护当前作用域和类型表，同时定义了用于存储错误信息和生成的四元式的列表。analyze 方法接收抽象语法树（AST）作为输入，并调用 visit 方法对AST进行遍历。visit 方法根据AST节点的类型，调用相应的 visit\_ 开头的方法进行处理。例如，visit\_VarDeclaration 方法负责处理变量声明，它会解析变量的类型，并将变量名及其类型、偏移量等信息添加到当前作用域的符号表中，同时生成 DECLARE 四元式。对于表达式的处理（如 visit\_Exp、visit\_Term、visit\_Factor），我们会进行类型检查，并生成相应的算术运算四元式。对于控制流语句（如 visit\_ConditionalStm、visit\_LoopStm），我们会生成相应的条件判断和跳转四元式。过程声明和调用也会进行相应的处理，包括创建新的作用域、处理参数列表以及生成 PROCEDURE、CALL 等四元式。  语义分析与符号表管理  语义分析阶段在语法分析的基础上，对程序的结构进行更深层次的检查，以确保程序的逻辑正确性。我们首先定义了 SymbolTable 类，用于管理程序中声明的各种符号（如变量、类型、过程等）及其属性。每个符号表实例代表一个作用域，可以拥有父作用域和子作用域，从而支持嵌套的作用域结构。符号表中存储了符号的类型、在内存中的偏移量以及类别（如变量、过程、形参等）。add\_symbol 方法用于向符号表中添加新的符号，并进行重复定义检查。lookup 方法则用于在当前作用域及其父作用域中查找符号的信息。为了表示不同的数据类型，我们定义了 ProcType、RecType 和 ArrayType 类，分别用于存储过程的参数信息、记录的字段信息和数组的上下界以及元素类型。  类型检查与中间代码生成  SemanticAnalyzer 类是语义分析的核心。在初始化方法中，我们创建了全局作用域的符号表，并使用栈来维护当前作用域和类型表，同时定义了用于存储错误信息和生成的四元式的列表。analyze 方法接收抽象语法树（AST）作为输入，并调用 visit 方法对AST进行遍历。visit 方法根据AST节点的类型，调用相应的 visit\_ 开头的方法进行处理。例如，visit\_VarDeclaration 方法负责处理变量声明，它会解析变量的类型，并将变量名及其类型、偏移量等信息添加到当前作用域的符号表中，同时生成 DECLARE 四元式。对于表达式的处理（如 visit\_Exp、visit\_Term、visit\_Factor），我们会进行类型检查，并生成相应的算术运算四元式。对于控制流语句（如 visit\_ConditionalStm、visit\_LoopStm），我们会生成相应的条件判断和跳转四元式。过程声明和调用也会进行相应的处理，包括创建新的作用域、处理参数列表以及生成 PROCEDURE、CALL 等四元式。  在语义分析过程中，我们使用 Quadruple 类来表示中间代码，每个 Quadruple 对象包含操作符、两个操作数和一个结果。emit\_quad 方法用于生成新的四元式并将其添加到 quadruples 列表中。为了方便调试和查看生成的中间代码，我们重写了 Quadruple 类的 \_\_str\_\_ 方法。在语义分析过程中，如果发现任何语义错误（如未声明的变量、类型不匹配、重复定义等），我们会调用 error 方法生成错误信息，并将其添加到 errors 列表中。在完成对整个AST的遍历后，我们会打印所有的错误信息。如果语义分析成功，则会将生成的四元式以表格的形式写入到文件 ./data/exps.txt 中。  **目标代码生成（MIPS）：**  目标代码生成阶段的任务是将中间代码（四元式）翻译成特定目标机器的机器指令。我们实现了 MIPSGenerator 类来完成这个任务，它以四元式列表作为输入。在初始化方法中，我们设置了用于存储生成的MIPS汇编代码的列表 code，以及用于管理临时标号、寄存器分配、栈帧布局等的数据结构。reg\_map 用于跟踪变量到寄存器的映射，reg\_pool 维护了可用的临时寄存器列表，param\_regs 预留了参数传递的寄存器。我们还使用了 stack\_offset1 和 stack\_list 来管理每个过程的栈帧中变量的偏移量。target\_stack 和 label\_stack 用于辅助生成控制流相关的跳转指令。  get\_reg 方法负责分配寄存器给变量，它会优先复用已经分配的寄存器。如果所有临时寄存器都被占用，则会调用 \_handle\_register\_overflow 方法来处理寄存器溢出（在本实现中是一个简单的警告）。free\_reg 方法用于释放不再使用的寄存器。emit 方法用于将生成的MIPS指令添加到 code 列表中。get\_offset 方法用于获取变量在栈帧中的偏移量，而 is\_var 方法用于判断一个操作数是否是变量。\_resolve\_sp 方法在每个过程的开始计算其栈帧的大小，并调整栈指针 $sp。get\_regs 和 free\_regs 方法是 get\_reg 和 free\_reg 的包装器，它们在分配或释放寄存器的同时，还会根据变量是否在栈上，生成从内存加载或将寄存器值存储回内存的指令。  generate 方法是目标代码生成的入口点。它首先添加MIPS汇编的数据段和文本段的声明，以及 main 函数的标签。然后，它遍历四元式列表，并根据四元式的操作符调用相应的 \_gen\_ 开头的方法来生成MIPS指令。\_gen\_instruction 方法是一个分发器，它根据操作符的类型将控制权交给相应的指令生成函数。例如，\_gen\_assign 生成赋值指令，\_gen\_add 生成加法指令，\_gen\_lt 生成小于比较指令等。对于控制流语句（如 THEN、ELSE、ENDIF、WHILE、DO、ENDWHILE），我们会生成相应的条件跳转和无条件跳转指令，并使用 target\_stack 和 label\_stack 来处理跳转目标的回填。过程的声明和调用（PROCEDURE、ENDPROCEDURE、call）会生成过程的标签、栈帧的建立和恢复、以及跳转指令。\_gen\_input 和 \_gen\_output 分别生成输入和输出相关的系统调用指令。  **编译器：**  我们将各个部分封装成不同的模块，并在MIPSGenerator.py模块中调用所有模块，实现最终的集成。 | | |
| **程序界面及运行截图** | | |
| **正确代码分析情况：**  **冒泡排序**  {输入m,表示要排序的数的个数; 接着输入m个整数; 输出从小到大排序后的结果}  program bubble  var integer i,j,num;  array [1..20] of integer a;  procedure q(integer num);  var integer i,j,k;  integer t;  begin  i:=1;  while i < num do  j:=num-i+1;  k:=1;  while k<j do  if a[k+1] < a[k]  then  t:=a[k];  a[k]:=a[k+1];  a[k+1]:=t  else t:=0  fi;  k:=k+1  endwh;  i:=i+1  endwh  end  begin  read(num);  i:=1;  while i<(num+1) do  read(j);  a[i]:=j;  i:=i+1  endwh;  q(num);  i:=1;  while i<(num+1) do  write(a[i]);  i:=i+1  endwh  end.  **词法分析：**    **语法分析：**    **语义分析及中间代码生成：**  **符号表：**      **中间代码优化：**    **目标代码生成：**    **运行结果：**    **不正确代码分析情况：**   1. **语义错误：**   {  语义错误种类：  （1） 标识符的重复定义；  （2） 无声明的标识符；  （3） 标识符为非期望的标识符类别（类型标识符，变量标识符，过程名  标识符）；  （4） 数组类型下标越界错误；  声明时的下标越界  对于使用时的下标越界情况这里不做检查，VS编译器不做检查  （5） 数组成员变量和域变量的引用不合法；  （6） 赋值语句的左右两边类型不相容；  （7） 赋值语句左端不是变量标识符；  （8） 过程调用中 ，形实参类型不匹配；  （9） 过程调用中， 形实参个数不相同；  （10）过程调用语句中，标识符不是过程标识符；  （11）if 和 while 语句的条件部分不是 bool 类型；  （12）表达式中运算符的分量的类型不相容。  }  program p  {类型声明}  type int = integer;  sarray = array[1..10] of integer;  tarray = array[10..1] of char; {lineno:24 数组上下界错误}  stu = record  integer a, b;  char c, d;  end;  {变量声明}  var integer i, j, k;  sarray sa;  stu astu;  integer ax, bx;  char cx, dx;  integer ax; {lineno:36 标识符的重复定义}  {函数声明}  procedure f(char a);  type ch=char;  var ch qaq;  qaq xa; {lineno:42 非类型标识符}  begin  write(qaq)  end  procedure g(char a; var integer b);  type ch=char;  var ch qaq;  begin  write(qaq)  end  begin  o := 0; {lineno:55 无声明的标识符}  sa[12] := 15;  sa[6+8] := 99;  sa[6-8] := 999;  astu.c := 30; {lineno:59 赋值号两端不相容，左char右int}  astu.ut := 25; {lineno:60 无此域名}  int := 99; {lineno:61 int非变量标识符}  i := 1;  j := 2;  k := 3;  g(cx, 10); {lineno 65 形参为变参，实参为值参，不匹配}  f(i); {lineno:66 形参实参类型不匹配}  f(i, j, k); {lineno:67 参数个数不匹配、类型不匹配}  ax(i); {lineno:68 过程调用语句中，标识符不是过程标识符}  if i < 10  then i := i + int {lineno:70 int非变量类型、表达式中运算符的分量的类型不相容}  else f(cx)  fi;  while ax < 10 do  bx:=sa[ax];  ax := ax + 1  endwh;  if i < 10  then i := i + 1  else f(dx)  fi;  i := c {lineno:83 未声明、赋值号两端不相容}  end.  **错误结果：**     1. **语法错误：**   program pp  type t1=integer;  t2=t1;  var integer v1,v2,c;  t2 v3;  array [2..100] of integer a;  record  array [1..10] of integer b;  integer v4;  end r1;  char m1,m2;  procedure f(var t1 y);  var integer v11,v22;  array [2..100] of char a1;  begin  r1.v4:=1+3\*(7-5);  y:= r1.v4;  end  begin  a[(3+4)\*5]:=5;  v1:=a[35];  if v1 < 10 then write(v1+10)  else write(v1-10)  fi;  f(v1);  write(v1)  end.  **错误结果：**     1. 词法错误   program p {yoriko}  type t1 = integer;  var integer v1,v2,12x;  array [1.20] of integer a;  procedure  q(integer i);  var integer a;  begin  a:=i;  write(a)  end  begin  read(v1);  if v1<10  then v1:=v1+10  else v1:=v1-10  fi;  q(v1)  end. | | |
| **源程序核心代码** | | |
| 1. Quad获取变量类型中结构体和数组的处理。   def \_get\_variable\_value(self, variable\_node):  \_, var\_id, var\_more = variable\_node  # 查找变量基础类型  var\_info = self.current\_scope.lookup(var\_id)  if not var\_info:  self.error(f"未定义的变量: {var\_id}")  return None, None  if isinstance(var\_info[0], tuple):  base\_type = var\_info[0][0]  else:  base\_type = var\_info[0]  # 处理数组下标或结构体访问  current\_type = base\_type  current\_more = var\_more  flag = False    while current\_more and current\_more[1] is not None:  flag = True  #print(var\_id, current\_more)  if current\_more[1][0] == 'Exp': # 数组访问  if not isinstance(current\_type, ArrayType):  self.error(f"{var\_id} 不是数组类型")  return None, None  index\_type, value = self.\_get\_expression\_value(current\_more[1])  if index\_type != 'integer':  self.error("数组下标必须为整数")  cons\_pos = self.generate\_temp\_var()  off\_set = self.generate\_temp\_var()  value\_pos = self.generate\_temp\_var()  pos = self.generate\_temp\_var()  self.emit\_quad(":=", 4, None, cons\_pos)  self.emit\_quad("-", value, current\_type.lower\_bound, pos)  self.emit\_quad("\*", pos, cons\_pos, off\_set)  self.emit\_quad("[]", var\_id, off\_set, value\_pos)  var\_id = value\_pos  current\_type = current\_type.element\_type  current\_more = None    else:  if not isinstance(current\_type, RecType):  self.error(f"{var\_id} 不是记录类型")  return None, None  field\_name = current\_more[1][1] # FieldVar的ID  #print(current\_type.fields)  if field\_name not in current\_type.fields:  self.error(f"字段 {field\_name} 不存在于记录中")  return None, None  cons\_pos = self.generate\_temp\_var()  off\_set = self.generate\_temp\_var()  value\_pos = self.generate\_temp\_var()  self.emit\_quad(":=", 4, None, cons\_pos)  self.emit\_quad(":=", current\_type.offset[field\_name], None, off\_set)  self.emit\_quad("\*", off\_set, cons\_pos, off\_set)  self.emit\_quad("[]", var\_id, off\_set, value\_pos)  current\_type = current\_type.fields[field\_name]  var\_id = value\_pos  current\_more = current\_more[1][2]  print(current\_more)  if flag:  self.emit\_quad("load", value\_pos, None, value\_pos)  return current\_type, var\_id   1. 类型解析   def \_resolve\_type(self, type\_name\_node):  if type\_name\_node and len(type\_name\_node) == 2 and type\_name\_node[0] == 'TypeName':  type\_name\_content = type\_name\_node[1]  if isinstance(type\_name\_content, str):  # Rule 14: TypeName -> ID  if not self.lookup\_type\_table(type\_name\_content):  self.error(f"类型 '{type\_name\_content}' 未定义")  return self.lookup\_type\_table(type\_name\_content)    elif isinstance(type\_name\_content, tuple) and len(type\_name\_content) == 2:  tag = type\_name\_content[0]  if tag == 'BaseType':  # Rule 12: TypeName -> BaseType  return self.analyze\_base\_type(type\_name\_content)  elif tag == 'StructureType':  # Rule 13: TypeName -> StructureType  return self.analyze\_structure\_type(type\_name\_content)  return None  def analyze\_base\_type(self, base\_type\_node):  """  解析 BaseType 节点并返回其具体的类型字符串。  """  if base\_type\_node and len(base\_type\_node) == 2 and base\_type\_node[0] == 'BaseType':  return base\_type\_node[1]  return None  def analyze\_structure\_type(self, structure\_type\_node):  """  解析 StructureType 节点并返回其具体的结构类型信息。  """  if structure\_type\_node and len(structure\_type\_node) == 2 and structure\_type\_node[0] == 'StructureType':  structure\_content = structure\_type\_node[1]  if isinstance(structure\_content, tuple) and len(structure\_content) > 0 and structure\_content[0] == 'ArrayType':  lower\_bound = structure\_content[1]  upper\_bound = structure\_content[2]  element\_type = self.analyze\_base\_type(structure\_content[3])  if element\_type:  try:  lower\_bound = int(lower\_bound)  upper\_bound = int(upper\_bound)  if lower\_bound > upper\_bound:  self.error('lower bound must be less than upper bound')  return ArrayType(lower\_bound, upper\_bound, element\_type)  except ValueError:  print(f"警告: 数组边界不是整数: {lower\_bound}, {upper\_bound}")  return None  elif isinstance(structure\_content, tuple) and len(structure\_content) == 2 and structure\_content[0] == 'RecType' :  fields = self.parse\_dec\_list(structure\_content[1])  #print(fields, "dwaidhiwa")  if fields is not None:  return RecType(fields)  return None   1. 目标代码中标号定位栈和目标指令地址栈的使用。（地址回填）   def \_gen\_then(self, op, cond, \_, \_\_):  cond\_reg = self.get\_reg(cond)  jump\_index = self.emit(f"beqz {cond\_reg}, else\_label")  self.emit("nop")  self.target\_stack.append((jump\_index, "else\_label"))  def \_gen\_else(self, op, \_, \_\_, \_\_\_):  jump\_index = self.emit(f"j endif\_label")  self.emit("nop")  # 回填 THEN 语句的跳转目标  if self.target\_stack:  then\_jump\_index, then\_label = self.target\_stack.pop()  self.code[then\_jump\_index] = self.code[then\_jump\_index].replace(then\_label, f"label{self.label\_count}")  #self.label\_definitions[f"label{self.label\_count}"] = len(self.code)  self.emit(f"label{self.label\_count}:")  self.label\_count += 1  else:  raise RuntimeError(f"找不到与 ELSE 对应的 THEN 标签:")  # 将需要回填的跳转指令索引和目标标号压入栈  self.target\_stack.append((jump\_index, 'endif\_label'))  def \_gen\_endif(self, op, \_\_\_, \_, \_\_):  if self.target\_stack:  print(self.target\_stack)  else\_jump\_index, else\_label = self.target\_stack.pop()  self.code[else\_jump\_index] = self.code[else\_jump\_index].replace(else\_label, f"label{self.label\_count}")  else:  raise RuntimeError(f"找不到与 ENDIF 对应的 THEN 或 ELSE 标签:")  # 定义 ENDIF 标号的位置  #self.label\_definitions[f"label{self.label\_count}"] = len(self.code)  self.emit(f"label{self.label\_count}:")  self.label\_count += 1  def \_gen\_while(self, op, \_, \_\_, \_\_\_):  # 定义循环开始的标号  #self.label\_definitions[f"label{self.label\_count}"] = len(self.code)  self.emit(f"label{self.label\_count}:")  self.label\_count += 1  self.label\_stack.append(f"label{self.label\_count - 1}")  def \_gen\_do(self, op, cond, \_, \_\_):  cond\_reg = self.get\_reg(cond)  jump\_index = self.emit(f"beqz {cond\_reg}, endwh")  self.emit("nop")  self.target\_stack.append((jump\_index, "endwh"))  def \_gen\_endwhile(self, op, \_\_, \_, \_\_\_):  start\_label = self.label\_stack.pop()  self.emit(f"j {start\_label}")  self.emit("nop")  if self.target\_stack:  do\_jump\_index, do\_label = self.target\_stack.pop()  self.code[do\_jump\_index] = self.code[do\_jump\_index].replace(do\_label, f"label{self.label\_count}")  else:  raise RuntimeError(f"找不到与 ENDWHILE 对应的 DO 标签:")  self.emit(f"label{self.label\_count}:")  self.label\_count += 1 | | |
|  | | |