电子科技大学信息与软件工程学院

实 验 报 告

学　　号 2023091602014

姓　　名 张烨涛

（实验）课程名称 代码生成

理论教师 陈安龙

实验教师 陈安龙

电 子 科 技 大 学

实　验　报　告

学生姓名：张烨涛　　学号：2023091602014　　指导教师：陈安龙

一、 实验名称：代码生成

二、 实验学时：4h

三、 实验目的：学习中间代码生成的具体实现

四、 实验原理：

LLVM 提供了一套适合编译器系统的中间语言（Intermediate Representation，IR），有大量变换和优化都围绕其实现。经过变换和优化后的中间语言，可以转换为目标平台相关的汇编语言代码。

LLVM 的中间语言与具体的语言、指令集、类型系统无关，其中每条指令都是静态单赋值形式（SSA）， 即每个变量只能被赋值一次。这有助于简化变量之间的依赖分析。LLVM 允许静态编译代码，或者通过实时编译（JIT）机制将中间表示转换为机器码（类似 Java）。

五、 实验内容：

代码生成

完成算术表达式、逻辑表达式、赋值语句、条件语句、循环语句对应的代码生成，函数命名分别为：

genArithmeticExpr, genLogicExpr, genAssignStmt, genIfStmt, genWhileStmt

函数参数都为：

(past node, char* result)

node 为相关类型的结点

result 为用来保存 LLVM 指令，每行只放一条指令；

注：在生成代码时，生成每个函数对应的代码即可，无需生成其它辅助信息。例如，针对下面的源代码：

```
int main(){
    int a;
    a = 10;
    return a % 3;
}
```

用 clang -emit-llvm -S 生成的 llvm 指令如下：

```
; ModuleID = './20_rem.c'
source_filename = "./20_rem.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 10, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = srem i32 %3, 3
  ret i32 %4
}

attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8"
```

```
"target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"tune-cpu"="generic" }

    !llvm.module.flags = !{!0, !1, !2, !3, !4}
    !llvm.ident = !{!5}

    !0 = !{i32 1, !"wchar_size", i32 4}
    !1 = !{i32 7, !"PIC Level", i32 2}
    !2 = !{i32 7, !"PIE Level", i32 2}
    !3 = !{i32 7, !"uwtable", i32 1}
    !4 = !{i32 7, !"frame-pointer", i32 2}
    !5 = !{!"Ubuntu clang version 14.0.6"}
```

而本实验生成的 llvm 指令只需要 main 函数对应的代码即可，且无需生成 llvm 中的函数修饰符 dso_local：

```
define i32 @main() {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 10, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = srem i32 %3, 3
  ret i32 %4
}
```

六、 实验器材（设备、元器件）：无

七、 实验步骤：

查看指令手册 LLVMRef.mht

用 clang 学习 llvm 虚拟指令：

编写简单的 C 语言程序 test.c

用 clang -emit-llvm -S ./test.c

生成该文件对应的 llvm 指令

学习示例程序，理解代码生成过程

在实验 3 及示例程序的基础上完成算术表达式、逻辑表达式、赋值语句、条件语句、循环语句对应的代码生成

八、 实验结果与分析（含重要数据结果分析或核心代码流程分析）

实验结果：

Exp1:

当测试程序为

```
int get_one(int a) {
  return 1;
}

int deepWhileBr(int a, int b) {
  int c;
  c = a + b;
  while (c < 75) {
    int d;
    d = 42;
    if (c < 100) {
      c = c + d;
      if (c > 99) {
        int e;
        e = d * 2;
        if (get_one(0) == 1) {
          c = e * 2;
        }
      }
    }
  }
  return (c);
}

int main() {
  int p;
  p = 2;
  p = deepWhileBr(p, p);
  putint(p);
  return 0;
}
```

时，

程序输出为

```
%2 = alloca i32, align 4
store i32 %0, i32* %2, align 4
ret i32 1
 }
define i32 @deepWhileBr(i32 %0, i32 %1) {
%3 = alloca i32, align 4
%4 = alloca i32, align 4
%5 = alloca i32, align 4
%6 = alloca i32, align 4
%7 = alloca i32, align 4
store i32 %0, i32* %3, align 4
store i32 %1, i32* %4, align 4
%8 = load i32, i32* %3, align 4
%9 = load i32, i32* %4, align 4
%10 = add nsw i32 %8, %9
store i32 %10, i32* %5, align 4
br label %11
11:
%12 = load i32, i32* %5, align 4
%13 = icmp slt i32 %12, 75
br i1 %13, label %14, label %34
14:
store i32 42, i32* %6, align 4
%15 = load i32, i32* %5, align 4
%16 = icmp slt i32 %15, 100
br i1 %16, label %17, label %33
17:
%18 = load i32, i32* %5, align 4
%19 = load i32, i32* %6, align 4
%20 = add nsw i32 %18, %19
```

define i32 @get_one(i32 %0) {
%2 = alloca i32, align 4
store i32 %0, i32* %2, align 4
ret i32 1
 }
define i32 @deepWhileBr(i32 %0, i32 %1) {
%3 = alloca i32, align 4
%4 = alloca i32, align 4
%5 = alloca i32, align 4
%6 = alloca i32, align 4
%7 = alloca i32, align 4
store i32 %0, i32* %3, align 4
store i32 %1, i32* %4, align 4
%8 = load i32, i32* %3, align 4
%9 = load i32, i32* %4, align 4
%10 = add nsw i32 %8, %9
store i32 %10, i32* %5, align 4
br label %11
11:
%12 = load i32, i32* %5, align 4
%13 = icmp slt i32 %12, 75
br i1 %13, label %14, label %34
14:
store i32 42, i32* %6, align 4
%15 = load i32, i32* %5, align 4
%16 = icmp slt i32 %15, 100
br i1 %16, label %17, label %33
17:
%18 = load i32, i32* %5, align 4
%19 = load i32, i32* %6, align 4

```
%20 = add nsw i32 %18, %19
store i32 %20, i32* %5, align 4
%21 = load i32, i32* %5, align 4
%22 = icmp sgt i32 %21, 99
br i1 %22, label %23, label %32
23:
%24 = load i32, i32* %6, align 4
%25 = mul nsw i32 %24, 2
store i32 %25, i32* %7, align 4
%26 = call i32 @get_one(i32 0)
%27 = icmp eq i32 %26, 1
br i1 %27, label %28, label %31
28:
%29 = load i32, i32* %7, align 4
%30 = mul nsw i32 %29, 2
store i32 %30, i32* %5, align 4
  br label %31
31:
br label %32
32:
br label %33
33:
br label %11, !llvm.loop !6
34:
%35 = load i32, i32* %5, align 4
ret i32 %35
}
define i32 @main() {
%1 = alloca i32, align 4
%2 = alloca i32, align 4
store i32 0, i32* %1, align 4
store i32 2, i32* %2, align 4
%3 = load i32, i32* %2, align 4
%4 = load i32, i32* %2, align 4
%5 = call i32 @deepWhileBr(i32 %3, i32 %4)
store i32 %5, i32* %2, align 4
%6 = load i32, i32* %2, align 4
%7 = call @putint(i32 %6)
ret i32 0
}
```

Exp2：

当测试程序为

```
const  int  a[5]={0,1,2,3,4};
```

```
int main(){
    return a[4];
}
```

时，

程序输出为：

```
@a = constant [5 x i32] [i32 0, i32 1, i32 2, i32 3, i32 4], align 16
define i32 @main() {
%1 = alloca i32, align 4
store i32 0, i32* %1, align 4
%2 = load i32, i32* getelementptr inbounds ([5 x i32], [5 x i32]* @a, i64 0, i64 4), align 16
ret i32 %2
}
```

Exp3:

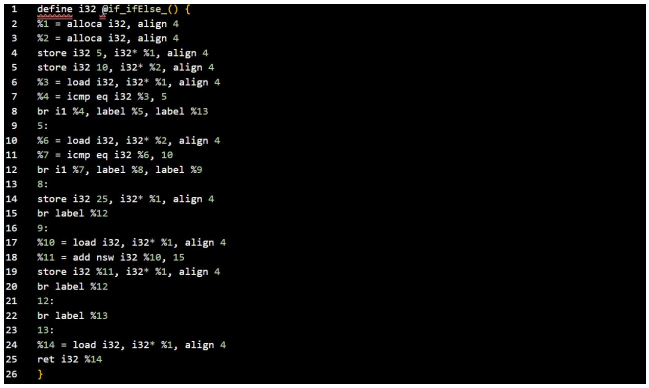
当测试程序为

```
int if_ifElse_() {
  int a;
  a = 5;
  int b;
  b = 10;
  if(a == 5){
    if (b == 10)
      a = 25;
    else
      a = a + 15;
  }
  return (a);
}

int main(){
  return (if_ifElse_());
}
```

时，

程序输出为：



```
define i32 @if_ifElse_() {
%1 = alloca i32, align 4
%2 = alloca i32, align 4
store i32 5, i32* %1, align 4
store i32 10, i32* %2, align 4
%3 = load i32, i32* %1, align 4
%4 = icmp eq i32 %3, 5
br i1 %4, label %5, label %13
5:
%6 = load i32, i32* %2, align 4
%7 = icmp eq i32 %6, 10
br i1 %7, label %8, label %9
8:
store i32 25, i32* %1, align 4
br label %12
9:
%10 = load i32, i32* %1, align 4
%11 = add nsw i32 %10, 15
store i32 %11, i32* %1, align 4
br label %12
12:
br label %13
13:
%14 = load i32, i32* %1, align 4
ret i32 %14
}
define i32 @main() {
%1 = alloca i32, align 4
store i32 0, i32* %1, align 4
%2 = call i32 @if_ifElse_()
ret i32 %2
}
```

Exp4:

当测试程序为

```
int main(){
    int a, b;
    a = 10;
    b = -1;
    return a + b;
}时，
```

程序输出为：

```
define i32 @main() {
%1 = alloca i32, align 4
%2 = alloca i32, align 4
%3 = alloca i32, align 4
store i32 0, i32* %1, align 4
store i32 10, i32* %2, align 4
store i32 -1, i32* %3, align 4
%4 = load i32, i32* %2, align 4
%5 = load i32, i32* %3, align 4
%6 = add nsw i32 %4, %5
ret i32 %6
}
```

```
define i32 @main() {
%1 = alloca i32, align 4
%2 = alloca i32, align 4
%3 = alloca i32, align 4
store i32 0, i32* %1, align 4
store i32 10, i32* %2, align 4
store i32 -1, i32* %3, align 4
%4 = load i32, i32* %2, align 4
%5 = load i32, i32* %3, align 4
%6 = add nsw i32 %4, %5
ret i32 %6
}
```

程序从 main.c 的 main 函数开始执行，主要流程如下：

1. main 函数首先处理命令行参数，打开需要编译的源文件。如果没有指定文件，则默认打开"./test.c"。
2. 调用 yyparse()函数进行语法分析，这个函数会构建抽象语法树(AST)，并将根节点保存在 astRoot 中。
3. 之后程序调用 genExpr(astRoot)开始生成 LLVM IR 代码，并用 showAst 显示语法树结构。

在 ast.c 中定义了创建各种 AST 节点的函数：

1. newAstNode()：基础函数，为节点分配内存并初始化为 0
2. newNum(int value)：创建数值节点，存储整数值
3. newExpr(int oper, past left, past right)：创建表达式节点，用于算术运算
   (1)oper 存储运算符(+,-,*,/)
   (2)left 和 right 指向操作数
4. newDoubleExpr()：创建逻辑表达式节点，处理比较运算(==, != 等)
5. newBasicNode()：创建基本语句节点，如 if、while 等控制结构
6. newNextNode()：处理语句序列，将多个语句连接成链表
7. newTypeNode()和 newIDNode()：分别创建类型节点和标识符节点

genllvm.c 实现了代码生成的核心逻辑：

1. genExpr()是主要入口函数，它遍历 AST 并调用 process()处理各类节点
2. process()根据节点类型调用相应的处理函数：
   (1)genDeclStmt()：处理变量声明，生成 alloca 和 store 指令
   (2)genAssignStmt()：处理赋值语句，生成 store 指令
   (3)genIfStmt()和 genIfElseStmt()：处理 if 语句，生成条件跳转指令
   (4)genWhileStmt()：处理 while 循环，生成循环和跳转指令
   (5)genReturnStmt()：处理 return 语句，生成返回指令
3. genExprStmt()处理表达式：

(1)算术运算：生成 add、sub、mul、div 指令

(2)比较运算：生成 icmp 指令

(3)变量访问：生成 load 指令

4. checkVariable()负责查找变量对应的寄存器编号
5. addLLVMCodes()将生成的 LLVM IR 代码输出

整个程序通过维护以下状态来管理代码生成：

1. regCount 和 varCount：跟踪寄存器和变量编号
2. variables 数组：存储变量名
3. variable_type 数组：记录变量类型
4. whilecheckpoint 和 ifcheckpoint：管理控制流标签

以下是完整的代码：

九、 总结及心得体会：

(1)编写代码之前需要写出递归下降翻译器的伪代码，重点就是要找到对于每个非终结符的属性哪些是继承属性，哪些是综合属性。然后将继承属性作为参数，综合属性作为返回值，进行计算。利用实验二所写的递归下降分析器的伪代码做出改写，加入参数返回值以及一些初始化。

(2)编写代码的时候，需要用到实验一和实验二的代码，写实验一代码的时候没考虑到后面会用到，直接将结果输出并没有保存中间结果，以至于自己在编码的时候需要先将实验一的结果存放在一个自定义的结构体中，里面包含词法分析的两个因素：值和类型。而分析器分析的时候，直接调取这个结构体的内容，四元式的结果也会放在一个特殊的结构体，里面记录了四元式的四个值，方便输出。如果是数字运算式，可以模拟计算器对于这四个值进行计算，并且需要数组和判定运算符函数来判断是数字还是辅助变量，根据对应符号进行运算。

(3)通过这次实验，从词法分析到语法分析到语义分析的知识点有了大致的回顾，并且重点回顾了每个阶段输入什么，输出什么，这些信息怎么存储，用什么算法来计算。还需要进一步优化自己的代码，比如在这次的实验代码过程中，需要改进的是将词法分析和语法分析合并，降低时间复杂度，提高执行效率。

(4)通过这四次的实验过程，让我对于编译原理这门课有了比较清晰的认识，可能理论课当时听懂了，过一会可能就会遗忘。但实验课不一样，花费了很久时间然后又是动手敲代码，又是写实验报告梳理思路更加深了对于这门课的理解。通过学习编译原理，感觉用到了数据结构，算法等思维理解，又需要对于许多概念的理解记忆。这也是这门课的难点所在。通过这次学习，懂得更要注重对于基础科目的掌握，不断加强和拓展自己的计算机思维。

通过完成这个实验，我能够更好地理解和实现编译过程中的关键步骤，尤其是中间代码生成这一重要技术。这项实验对于提升我的编程能力和培养良好的程序设计风格非常有帮助。同时，通过这个实验，我进一步加深了对编译技术中的中间代码生成算法的理解，并学会了使用中间代码生成的方法来根据给定的文法分析程序，并生成相应的中间代码表示。

此外，通过参与这个实验，我也获得了开发和调试编译程序的经验，对中间代码的生成过程有了更深入的理解。这个实验让我更加熟悉整个编译过程，并为我未来在编译器开发领域的学习和实践奠定了坚实的基础。

十、 对本实验过程及方法、手段的改进建议：无

报告评分：

指导教师签字：