编译原理实验报告三 & 四

陈永恒 151220012

前言

这次主要使用了LLVM来完成实验，主要使用的库包括:

```
#include <llvm/IR/Module.h>
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
```

实现的功能

1. 生成中间代码(3地址码)
2. 支持中间代码转成汇编代码(`x86,arm,mips` 均可，目前为了方便测试是转成 `x86` 的)
3. 支持汇编代码编译成可执行文件(同样为了方便助教验证是转化成 `x86` 的)

实验说明

语言的支持

由于使用了 `LLVM` 来实现，实现的方式和实验要求略有不同。

1. 支持实验要求的所有功能，但所有变量均为全局变量而不是局部变量(包括函数的传参,因此不支持函数递归)，因此函数的调用变成一下形式

```
//计算一个数的平方
int i;

```

```
int squre(){
    return i*i;
}

int mian(){
    i=4;
    return square();
}
```

2. 额外支持结构体的使用
3. 支持输入输出，由于希望能拓展输入输出的方式，目前用 `readint32()` 和 `writeint32()` 来命名。输入很简单，直接获取 `readint32()` 的返回值即可，而 `writeint32` ，需要将输出的结果提前赋值给一个预定义变量 `output` ,然后调用 `writeint32` .例子如下

```
int guess;
int res;
int main(){
    res=0;
    guess=0x233;
    while(guess!=res){
        res=readint32();
    }
    output=guess;
    writeint32();
    return 0;
}
```

中间代码的实现

`LLVM` 中提供了 `Builder` 模块，里面定义了许多生成指令的函数，如 `CreateRet` ,于是难点就在于如何生成操作数的类型，例如生成 `ret 1` 语句，实现可能就是如下所示:

```
Value* val=ConstantInt::get(*TheContext,APInt(32,1));
Builder->CreateRet(val);
```

控制流转移语句

在 LLVM 的代码生成里有了基本块的概念，有跳转的情况下，就要通过基本块的分割来实现。下面是我的实现代码，解释见注释

```
    /*
    Condition:
        some code
    then:
        some code
    else:
        some code
    ifcont:
        some code
    */
Value* NIfStmt::codegen(){
    auto CondV=this->condition->codegen();
    Function *TheFunction = Builder->GetInsertBlock()->
getParent();
    BasicBlock *ThenBB = BasicBlock::Create(*TheContex
t, "then", TheFunction);
    BasicBlock *ElseBB = BasicBlock::Create(*TheContex
t, "else");
    BasicBlock *MergeBB = BasicBlock::Create(*TheContex
t, "ifcont");

    if(this->elstmt!=NULL){
        Builder->CreateCondBr(CondV, ThenBB, ElseBB);
    }else{
        Builder->CreateCondBr(CondV, ThenBB, MergeBB);
    }


    Builder->SetInsertPoint(ThenBB);
    this->ifstmt->codegen();

    Builder->CreateBr(MergeBB);
    ThenBB = Builder->GetInsertBlock();

    if(this->elstmt!=NULL){
        TheFunction->getBasicBlockList().push_back(Else
BB);
```

```
33.         Builder->SetInsertPoint(ElseBB);
34.
35.         this->elstmt->codegen();
36.         Builder->CreateBr(MergeBB);
37.         ElseBB = Builder->GetInsertBlock();
38.     }
39.     TheFunction->getBasicBlockList().push_back(MergeB
    B);
40.     Builder->SetInsertPoint(MergeBB);
41.     return NULL;
42. }
```

目录列表

目录列表解释如下

```
1. Project 2  #实验二主目录
2.     main.cpp    //主程序
3.     Makefile    //编译文件
4.     tokens.l    //Flex部分源代码
5.     parser.y    //Bison部分源代码
6.     node.h      //语法树中每一种node的定义
7.     node.cpp    //对node的打印函数的定义,语义检查的函数，还有一
   些辅助函数，如createVar(type,name)，新增代码生成函数codegen
8.     compile.py //将源代码生成中间代码，再转中间代码为汇编，再生
   成可执行文件
9.     report.pdf //实验报告，即本文件
10.
11.    test_case/
12.        *       #各种测试文件
13.    util/
14.        io.*    #输入输出函数
```

编译与运行

编译命令: `make`
编译命令: `./compile.py file.c` ,这样会生成下列文件

1. file.ll， LLVM 中间代码文件
2. file.s，汇编文件
3. file , 可执行文件

如 guess.c 的源代码如下

```
➜  Project 3 git:(master) ✗ cat testcase/guess.c
int guess;
int res;
int main(){
    res=0;
    guess=0x233;
    while(guess!=res){
        res=readint32();

    }
    output=guess;
    writeint32();
    return 0;

}
```

编译(由于还没有编译 ./parser ,因此会执行 make)

```
➜  Project 3 git:(master) ✗ ./compile.py testcase/gues
s.c
rm -f -rf tokens.hpp parser.cpp parser.hpp parser token
s.cpp parser.o main.o tokens.o node.o
bison -d -o parser.cpp parser.y
g++ -c `llvm-config --cppflags` -std=c++11 -g -o parse
r.o parser.cpp
g++ -c `llvm-config --cppflags` -std=c++11 -g -o main.o
 main.cpp
flex -o tokens.cpp tokens.l parser.hpp
g++ -c `llvm-config --cppflags` -std=c++11 -g -o token
s.o tokens.cpp
g++ -c `llvm-config --cppflags` -std=c++11 -g -o node.o
 node.cpp
g++ -o parser parser.o main.o tokens.o node.o `llvm-con
fig --libs` `llvm-config --ldflags` -lpthread -ldl -lz
 -lncurses -rdynamic
```

这时候生成了3个文件

```
➜  Project 3 git:(master) ✗ ls testcase
guess  guess.c  guess.ll  guess.s
```

LLVM 中间代码文件

```
➜  Project 3 git:(master) ✗ cat testcase/guess.ll
; ModuleID = 'Program'
source_filename = "Program"

output = external global i32
@guess = common global i32 0
@res = common global i32 0

declare i32 @readint32()

declare i32 @writeint32()

define i32 @main() {
main:
  store i32 0, i32* @res
  store i32 563, i32* @guess
  br label %con

con:                                              ; pre
ds = %body, %main
  %tmp = load i32, i32* @guess
  %tmp1 = load i32, i32* @res
  %tmp_neq = icmp ne i32 %tmp, %tmp1
  br i1 %tmp_neq, label %body, label %next

body:                                             ; pre
ds = %con
  %calltmp = call i32 @readint32()
  store i32 %calltmp, i32* @res
  br label %con

```

```
30. next:                                             ; pre
    ds = %con
31.   %tmp2 = load i32, i32* @guess
32.   store i32 %tmp2, i32* @output
33.   %calltmp3 = call i32 @writeint32()
34.   ret i32 0
35. }
```

汇编文件

```
1.  ➜  Project 3 git:(master) ✗ cat testcase/guess.s
2.      .text
3.      .file   "Program"
4.      .globl  main                    # -- Begin function
     main
5.      .p2align    4, 0x90
6.      .type   main,@function
7.  main:                                   # @main
8.      .cfi_startproc
9.  # %bb.0:                                # %main
10.     pushq   %rax
11.     .cfi_def_cfa_offset 16
12.     movl    $0, res(%rip)
13.     movl    $563, guess(%rip)       # imm = 0x233
14.     jmp .LBB0_1
15.     .p2align    4, 0x90
16. .LBB0_2:                                # %body
17.                                         #   in Loop: He
    ader=BB0_1 Depth=1
18.     callq   readint32
19.     movl    %eax, res(%rip)
20. .LBB0_1:                                # %con
21.                                         # =>This Inner
    Loop Header: Depth=1
22.     movl    guess(%rip), %eax
23.     cmpl    res(%rip), %eax
24.     jne .LBB0_2
25. # %bb.3:                                # %next
26.     movl    guess(%rip), %eax
27.     movl    %eax, output(%rip)
28.     callq   writeint32
```

```
    xorl    %eax, %eax
    popq    %rcx
    retq
.Lfunc_end0:
    .size   main, .Lfunc_end0-main
    .cfi_endproc
                                        # -- End functi
on
    .type   guess,@object           # @guess
    .comm   guess,4,4
    .type   res,@object             # @res
    .comm   res,4,4

    .section    ".note.GNU-stack","",@progbits
```

运行结果:

```
➜  Project 3 git:(master) ✗ ./testcase/guess
123 #输入，不正确
12344 #输入，不正确
222 #输入，不正确
112 #输入，不正确
563 #输入 ，0x233=563
563 #正确，输出，程序结束
➜  Project 3 git:(master) ✗
```

总结

这次的实验我其实把实验四的部分也实现完成。由于使用了 LLVM ，使得我的生成代码可以直接转变成 x86 平台上可以直接运行的二进制文件，因此不需要借助模拟器的使用。目前我的工作是根据需求和时间进一步完善实验。