实验环境

推荐使用17.0即以上版本的llvm，如何更新可以参考LLVM Debian/Ubuntu packages，添加对应的源。

本实验使用脚本来自动完成测试，如果使用和助教不同的shell可能会导致脚本出错，需要自己根据对应的shell小小地修改。

增加的库函数

为了方便测试和后续优化实验的评分，增加了如下的库函数：

```
int getint(),getch();

void putint(int a),putch(int a),putarray(int n,int a[]);

void _sysy_starttime(int lineno);
void _sysy_stoptime(int lineno);
```

其中`getint(),getch()`用于输入，`putint(),putch(),putarray()`用于输出，`_sysy_starttime(),_sysy_stoptime()`用于计时，这些函数的实现在文件`syslib.c`中。

实验介绍

本次实验的目标是将ast转换为可以lli执行的llvm ir代码。实验总体分为4个步骤，首先第一次遍历ast生成全局变量，结构体和函数声明的llvm ir代码，这一部分代码已给出，可做参考。然后再遍历一次ast生成函数实现中的llvm ir代码，这是实验的主体部分。第三个步骤是将llvm ir从列表转换为基本块的形式。第四个步骤是将所有的alloca语句移动到第一个基本块中。

本次实验生成的是伪SSA形式的llvm ir代码，即所有的int类型变量都通过alloca语句在栈上分配一块空间，当其作为右值时通过load取出它的值来使用，其作为左值时通过store来为其赋值。

而对于结构体和数组来说情况就稍微有些复杂。在我们的语言中函数的参数为数组或结构体时实际上传的是该数组或结构体的地址，同时我们的语言没有结构体和数组的直接赋值。所以定义在函数中的数组或结构体要通过alloca语句在栈上分配一块空间，然后通过getelementptr得到其子元素的地址，再通过load和store取值和赋值。而对于作为函数参数的数组和结构体不用像int类型那样alloca一块空间，可以直接使用，因为没有数组和结构体的赋值，数组和结构体作为函数参数不会改变，是天然SSA的。

实验框架

temp.h

```
enum class TempType
{
    INT_TEMP,
    INT_PTR,
    STRUCT_TEMP,
    STRUCT_PTR
};

struct Temp_temp
{
    int num;
```

```
    TempType type;
    std::string structname;
    std::string varname;
    int len;
    Temp_temp(int _num,TempType _type = TempType::INT_TEMP,int _len = 0,const
std::string& _sname = std::string());
};
```

Temp_temp是ir中寄存器的抽象，其中num是其寄存器编号，type是该寄存器中元素的类型，INT_TEMP代表该寄存器存的是int类型的数据，INT_PTR代表int类型的指针或数组，STRUCT_TEMP和STRUCT_PTR同理，值得注意的是在我们的语言中STRUCT_TEMP只存在于全局变量中。如果该寄存器和结构体有关则structname是其结构体的名字，而len用于区分指针和数组，当len为0时代表单个元素的指针，当len为-1时代表数组头指针，这种情况只在数组作为函数参数时存在，当len大于0时代表数组的长度。varname用于debug。

```
struct Temp_label
{
    std::string name;
    Temp_label(std::string _name);
};
```

Temp_label是标签的抽象。

```
struct Name_name
{
    Temp_label *name;
    TempType type;
    std::string structname;
    int len;
    Name_name(Temp_label *_name,TempType _type,int _len = 0,const std::string
&_structname = std::string());
};
```

Name_name是全局变量的抽象。

```
enum class OperandKind
{
    TEMP, NAME, ICONST
};

struct AS_operand
{
    OperandKind kind;
    union {
        Temp_temp *TEMP;
        Name_name *NAME;
        int ICONST;
    } u;
};
```

AS_operand代表ir中操作数，其可能为寄存器，全局变量或立即数。

llvm_ir.h

```

```

```
struct L_structdef
{
    std::string name;
    std::vector<TempDef> members;
    L_structdef(const std::string &_name,const std::vector<TempDef> &_members);
};

struct L_funcdecl
{
    std::string name;
    std::vector<TempDef> args;
    FuncType ret;
    L_funcdecl(const std::string &_name,const std::vector<TempDef>
&_args,FuncType _ret);
};

struct L_globaldef
{
    std::string name;
    TempDef def;
    std::vector<int> init;
    L_globaldef(const std::string &_name,TempDef _def, const std::vector<int>
&_init);
};
```

这些代码和llvm ir的全局声明有关。

```
struct L_binop
{
    L_binopKind op;
    AS_operand *left,*right,*dst;
    L_binop(L_binopKind _op,AS_operand* _left,AS_operand *_right,AS_operand
*_dst);
};
```

L_binop代表的ir是四则运算相关的ir，对应ir如下：

```
%dst = op i32 %left,%right
```

```
struct L_load
{
    AS_operand *dst,*ptr;
    L_load(AS_operand *_dst,AS_operand *_ptr);
};
```

L_load代表的是llvm ir中的load，其可以从一个地址中将对应的值取出，在我们的实验中load只能取出int类型的值。对应ir如下：

```
%dst = load i32, i32* %ptr
```

```
struct L_store
{
    AS_operand *src,*ptr;
    L_store(AS_operand *_src,AS_operand *_ptr);
};
```

L_store代表的是llvm ir中的store，其可以对一个地址中将对应的值赋值，在我们的实验中store只能赋int类型的值。对应ir如下：

```
store i32 %src, i32* %ptr
```

```
struct L_label
{
    Temp_label *label;
    L_label(Temp_label *_label);
};
```

L_label对应llvm ir中的标签，对应ir如下：

```
label:
```

```
struct L_jump
{
    Temp_label *jump;
    L_jump(Temp_label *_jump);
};
```

L_jump对应llvm ir中的直接跳转，对应的ir如下：

```
br label %jump
```

```
struct L_cmp
{
    L_relopKind op;
    AS_operand *left,*right;
    AS_operand *dst;
    L_cmp(L_relopKind _op,AS_operand *_left,AS_operand *_right,AS_operand
*_dst);
};
```

L_cmp代表llvm ir中的icmp，用于得到两个数的相对关系，对应ir如下：

```
%dst = icmp op i32 %left, %right
```

```
struct L_cjump
{
    AS_operand *dst;
    Temp_label *true_label,*false_label;
    L_cjump(AS_operand *_dst,Temp_label *_true_label,Temp_label *_false_label);
};
```

L_cjump代表的是条件跳转，对应的ir如下：

```
br i1 %dst, %true_label, %false_label
```

```
struct L_move
{
    AS_operand *src,*dst;
    L_move(AS_operand *_src,AS_operand *_dst);
};
```

L_move代表的是移动，在llvm ir中没有表示移动的ir，所以用add 0来表示：

```
%dst = add i32 %src, 0
```

```
struct L_call
{
    std::string fun;
    AS_operand *res;
    std::vector<AS_operand*> args;
    L_call(const std::string &_fun,AS_operand *_res,const
std::vector<AS_operand*> &_args);
};
```

L_call代表的是有返回的函数调用,由于我们的语言不支持结构体和数组赋值，所以有返回值的函数只能返回int类型，对应ir如下：

```
%res = call i32 @fun(args)
```

```
struct L_voidcall
{
    std::string fun;
    std::vector<AS_operand*> args;
    L_voidcall(const std::string &_fun,const std::vector<AS_operand*> &_args);
};
```

L_voidcall代表的是不使用返回值的函数调用，对应ir如下：

```
call void @fun(args)
```

```
struct L_ret
{
    AS_operand *ret;
    L_ret(AS_operand *_ret);
};
```

L_ret代表着函数的返回，ret为空时对应的ir为`ret void`,否则对应的是：`ret i32 %ret`

```
struct L_phi
{
    AS_operand *dst;
    std::vector<std::pair<AS_operand*,Temp_label*>> phis;
    L_phi(AS_operand *_dst,const std::vector<std::pair<AS_operand*,Temp_label*>>
&_phis);
};
```

L_phi对应的是llvm ir中的phi语句，本次实验不会用到。

```
struct L_alloca
{
    AS_operand *dst;
    L_alloca(AS_operand *_dst);
};
```

L_alloca对应的是llvm ir中的alloca语句，对应的ir如下：

```
%dst = alloca <ty>
```

```
struct L_gep
{
    AS_operand *new_ptr,*base_ptr,*index;
    L_gep(AS_operand *_new_ptr,AS_operand *_base_ptr,AS_operand *_index);
};
```

L_gep对应的是llvm ir中的getelementptr，当base_ptr为int数组时，对应的llvm ir如下：

```
%new_ptr = getelementptr [len x i32], [len x i32]* %base_ptr, i32 0, i32 %index
```

当base_ptr为int指针时，对应的llvm ir如下：

```
%new_ptr = getelementptr i32, i32* %base_ptr, i32 %index
```

当base_ptr为结构体时，对应的llvm ir如下：

```
%new_ptr = getelementptr %structname, %structname* %base_ptr, i32 0, i32 %index
```

当base_ptr为确定长度的结构体数组时，对应的llvm ir如下：

```
%new_ptr = getelementptr [len x %structname], [len x %structname]* %base_ptr,
i32 0, i32 %index
```

当base_ptr为不定长度的结构体数组时，对应的llvm ir如下：

```
%new_ptr = getelementptr %structname, %structname* %base_ptr, i32 %index
```

更多的llvm ir介绍可以参考[LLVM Language Reference Manual — LLVM 18.0.0git documentation](#).

测试考察功能点

测试考察的是是否正确翻译了llvm ir。本质上除了结构体外我们的语言功能上就是c的子集，所以我们考察的内容大体上是

1. 基本的四则运算
2. if，while的翻译
3. break,continue语句的翻译
4. bool表达式的短路
5. 函数调用的翻译
6. 数组下标的翻译
7. 结构体子元素的翻译
8. 结构体和数组传参的翻译
9. 全局变量
10. 变量定义的翻译
11. 编译期计算（已实现）
12. 函数的递归调用（在arm汇编考察）
13. 库函数的调用
14. 类型推导

名为test*的测试用例供大家参考调试，不计分。