Project3：中间代码生成

通过前面对 AST 遍历，完成了语义分析后，没有语法、语义错误时，对 AST 进行遍历，计算相关的属性值，已经可以建立并随时访问符号表，那么接下来，可以在遍历 AST 树时，增加中间代码生成所需要的功能，要求生成以三地址代码格式 TAC 作为中间语言的中间语言代码序列。

中间代码结构体定义

```c
typedef struct code_node
{
  struct code_node* prev;    //前一个
  struct code_node* next;    //后一个
  int args_count;         //有多少个词。具体见上面注释内的分类。
  char args[6][32];       //每个词都是什么
}code_node;
```

翻译模式：

translate_Exp(Exp, place) = case Exp of	

INT	value = to_int(INT)
	return [place := #value]

ID	variable = symtab_lookup(ID)
	return [place := variable.name]

Exp$_1$ ASSIGN Exp$_2$	variable = symtab_lookup(Exp$_1$.ID)
	tp = new_place()
	code1 = translate_Exp(Exp$_2$, tp)
	code2 = [variable.name := tp]
	code3 = [place := variable.name]
	return code1 + code2 + code3

Exp$_1$ PLUS Exp$_2$	t1 = new_place()
	t2 = new_place()
	code1 = translate_Exp(Exp$_1$, t1)
	code2 = translate_Exp(Exp$_2$, t2)
	code3 = [place := t1 + t2]
	return code1 + code2 + code3

MINUS Exp	tp = new_place()
	code1 = translate_Exp(Exp, tp)
	code2 = [place := #0 - tp]
	return code1 + code2

cond. Exp	lb1 = new_label()
	lb2 = new_label()
	code0 = [place := #0]
	code1 = translate_cond_Exp(Exp, lb1, lb2)
	code2 = [LABEL lb1] + [place := #1] + [LABEL lb2]
	return code0 + code1 + code2

translate_cond_Exp(Exp, lb_t, lb_f) = case Exp of	

Exp$_1$ EQ Exp$_2$	t1 = new_place()
	t2 = new_place()
	code1 = translate_Exp(Exp$_1$, t1)
	code2 = translate_Exp(Exp$_2$, t2)
	code3 = [IF t1 == t2 GOTO lb_t] + [GOTO lb_f]
	return code1 + code2 + code3

Exp$_1$ AND Exp$_2$	lb1 = new_label()
	code1 = translate_cond_Exp(Exp$_1$, lb1, lb_f) + [LABEL lb1]
	code2 = translate_cond_Exp(Exp$_2$, lb_t, lb_f)
	return code1 + code2

Exp$_1$ OR Exp$_2$	lb1 = new_label()
	code1 = translate_cond_Exp(Exp$_1$, lb_t, lb1) + [LABEL lb1]
	code2 = translate_cond_Exp(Exp$_2$, lb_t, lb_f)
	return code1 + code2

NOT Exp	return translate_cond_Exp(Exp, lb_f, lb_t)

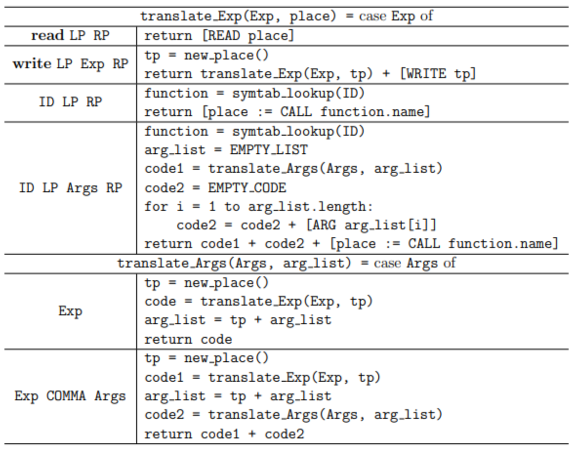
translate_Stmt(Stmt) = case Stmt of	

RETURN Exp SEMI	tp = new_place()
	code = translate_Exp(Exp, tp)
	return code + [RETURN tp]

IF LP Exp RP Stmt	lb1 = new_label()
	lb2 = new_label()
	code1 = translate_cond_Exp(Exp, lb1, lb2) + [LABEL lb1]
	code2 = translate_Stmt(Stmt) + [LABEL lb2]
	return code1 + code2

IF LP Exp RP Stmt$_1$ ELSE Stmt$_2$	lb1 = new_label()
	lb2 = new_label()
	lb3 = new_label()
	code1 = translate_cond_Exp(Exp, lb1, lb2) + [LABEL lb1]
	code2 = translate_Stmt(Stmt$_1$) + [GOTO lb3] + [LABEL lb2]
	code3 = translate_Stmt(Stmt$_2$) + [LABEL lb3]
	return code1 + code2 + code3

WHILE LP Exp RP Stmt	lb1 = new_label()
	lb2 = new_label()
	lb3 = new_label()
	code1 = [LABEL lb1] + translate_cond_Exp(Exp, lb2, lb3)
	code2 = [LABEL lb2] + translate_Stmt(Stmt) + [GOTO lb1]
	return code1 + code2 + [LABEL lb3]



translate_Exp(Exp, place) = case Exp of	
read LP RP	return [READ place]

write LP Exp RP	tp = new_place()
	return translate_Exp(Exp, tp) + [WRITE tp]

ID LP RP	function = symtab_lookup(ID)
	return [place := CALL function.name]

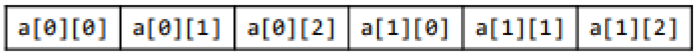
ID LP Args RP	function = symtab_lookup(ID)
	arg_list = EMPTY_LIST
	code1 = translate_Args(Args, arg_list)
	code2 = EMPTY_CODE
	for i = 1 to arg_list.length:
	    code2 = code2 + [ARG arg_list[i]]
	return code1 + code2 + [place := CALL function.name]

translate_Args(Args, arg_list) = case Args of	

Exp	tp = new_place()
	code = translate_Exp(Exp, tp)
	arg_list = tp + arg_list
	return code

Exp COMMA Args	tp = new_place()
	code1 = translate_Exp(Exp, tp)
	arg_list = tp + arg_list
	code2 = translate_Args(Args, arg_list)
	return code1 + code2

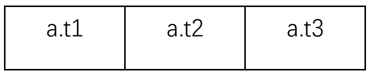
除此以外，该程序还能对多维数组以及结构体进行翻译。数组的实现采取内存中线性排列的方式实现



多维数组的表达策略

结构体的实现按照结构体的定义顺序在内存中线性排列

```
struct a{
    int t1;
    int t2;
    int t3;
}
```



数组在内存中的表达方式

中间代码优化：

优化 1：label 优化，如果多个 label 在生成的中间代码排列在一起会缩减至 1 个

优化 2：表达式优化，去除 a=a, a=a*1, a=a+0 等无意义赋值，去除赋值后没有使用的值

优化 3：去除赋值后没有使用的值，例如$a = \ldots(1); \ldots(\textit{没用到}a); a = \ldots(2);\ ==> \ldots(\textit{没用到}a); a = \ldots(2);$

代码文件说明：

Dictionary.c：定义了词典类型

Ir_buffer.c：中间代码优化

semantics.c：语法分析

symbols.c：符号表

translate.c：中间代码翻译

额外功能实现：

Test_a.spl：实现了数组作为参数传入函数

Test_b.spl：实现了结构体作为参数传入函数以及相关赋值

Test_c.spl：实现了多维数组申明以及相关赋值