Compiler Priciple - Project Report

课程名称：编译原理 学院：计算机学院 专业： 计算机科学与技术

指导老师：鲁东明

小组成员：

1. 需求概述

在说明编译器实现的细节之前，我们先对我们编译器的功能需求进行阐述。在本项目中，我们实现了一个小型的类C语言的编译器，但由于并没有实现C的所有功能，因此我们称之为`C--`，文件后缀名为`.cmm`。我们的编译器能够将`C--`源代码转换成MIPS汇编代码，所得到的汇编代码能够在SPIM Simulator上运行。下面我们对以下的几个方面进行详细的阐述。

1.1 语法定义与假设

数据类型

- 整型`int`：32位，可以是八进制、十进制或十六进制
  - 为了生成立即数指令（如addi等）的方便，整型常数限制在16位整数范围内
- 数组：支持1维或高维数组，数组元素类型可以是整型、结构体
- 结构体：成员变量可以是整型、数组、结构体
- 由于在目标代码生成时没有加入浮点指令，因此最后程序的运行是不支持浮点数`float`类型，尽管在语法分析、语义分析时`float`都是可以正常接收的

函数

- 函数名不可与mips指令名重名
- 函数可以有声明、定义
- 函数的参数可以是整型、结构体类型、数组类型
- 函数的返回值是且只能是整型，函数一定有`return`语句

控制语句

- 实现了`if`和`while`语句
- `if`和`while`语句的条件仅能是整型变量
- 在一个块（即`{}`内）中，必须先声明所有要用的变量，再开始操作

大括号块

- 大括号块的前半部分可以进行变量声明与定义，后半部分可以进行运算、函数调用、控制结构等，两部分不可交叉

作用域

- 我们没有对作用域进行处理，因此要求所有的变量包括函数参数均不重名
- 没有全局变量的使用，变量的存储空间都放到该变量所在的函数的活动记录中

错误处理

- 详见语义分析章节

其他

- 可以出现形如`//`和`/*...*/`的注释

1.2 生成代码

- 生成MIPS目标代码，代码在SPIM Simulator模拟器上运行

1.3 运行环境

- 工程构建环境
  - Windows 10（实际上也可以在Linux下构建）
  - bison (GNU Bison) 2.4.1
  - flex version 2.5.4
  - gcc (tdm64-1) 4.9.2
- 中间代码测试环境
  - IR Simulator即中间代码的解释器
  - 文件和使用说明见`irsim`文件夹
  - 得到中间代码后，可以用该程序运行中间代码
- 目标代码运行环境
  - SPIM Simulator（下载地址：http://pages.cs.wisc.edu/~larus/spim.html）
  - 这是一个功能强大的MIPS32汇编语言的汇编器和模拟器，我们能够将目标代码放在上面运行

1.4 输入和输出

- 输入：`C--`源代码
- 中间输出
  - 语法树：由非终结符和终结符构成的树结构
  - 符号表：包括变量符号表和函数符号表
  - 中间代码：线性结构的中间代码
- 输出：可以在SPIM Simulator运行的MIPS汇编代码

2. 词法分析

整数常数：可识别0开头的八进制，0x开头的16进制，以及1-9开头的十进制。识别规则如下：

```
O    [0-7]
D    [0-9]
D_NOZERO [1-9]
H    [a-fA-F0-9]
HP   (0(x|X))
HEX_CONSTANT {HP}{H}+
OCT_CONSTANT 0{O}+
DEC_CONSTANT 0|{D_NOZERO}{D}*
%%
{dec_integer}    {yylval.node=NewNode("INT",yytext);return INT;}
{other_integer} {
    int tmp = String2Int(yytext);
    char *s = (char*)malloc(32);
    sprintf(s, "%d", tmp);
    yylval.node=NewNode("INT",s);
    return INT;
}
```

浮点常数：可识别十进制浮点数以及科学计数法表示的浮点数(因为浮点数计算需要用到浮点mips指令，所以只保留在到语法分析阶段，中间代码阶段开始就不再实现)

```
E   ([Ee][+-]?{D}+)
float {D}*\.{D}+{E}?
{float}     {yylval.node=NewNode("FLOAT",yytext);return FLOAT;}
```

变量：以下划线或字母开头，之后可以包括下划线、字母、数字。

```
ID_character [_0-9a-zA-Z]
Pre_ID [_a-zA-Z]
id {Pre_ID}{ID_character}*
```

注释：

单行注释：以双斜杠开头，\n结尾

```
"//".*\n    {}
```

多行注释：遇到/*时调用comment函数。comment函数一直读取字符，直到遇到注释结尾，若没有注释结尾，会调用yyerror报错

```
"/*"        {comment(yyin);}

static void comment(FILE* f)//读取注释，直到注释结束
{
    int c;
    // 使用getchar()会一直收到-1，-1代表EOF，说明重定向没有将文件流传入stdin
    while ((c = input()) != 0 && c!=EOF)
    {
        if (c == '*')
        {
            while ((c = input()) == '*');
            if (c == '/')return;//找到注释结尾
            if (c == 0)break;//字符串结尾
        }
    } //字符串结尾
    yyerror("unterminated comment");
}
```

其他c语言终结符，省去了一些终结符，具体保留的终结符如下：

```
SEMI → ;
COMMA → ,
ASSIGNOP → =
RELOP → > | < | >= | <= | == | !=
PLUS → +
MINUS → -
STAR → *
DIV → /
AND → &&
OR → ||
DOT → .
NOT → !
TYPE → int | float
LP → (
RP → )
LB → [
RB → ]
LC → {
RC → }
STRUCT → struct
RETURN → return
IF → if
ELSE → else
WHILE → while
```

1.控制语句：不支持break语句和for循环，不支持do{}while()循环。

2.逻辑运算：支持&&，||，！,不支持其他逻辑运算。

3.支持加减乘除、赋值，不支持+=,*=,/=,-=,++,--

4.支持struct不支持union。

5.不支持static

3. 语法分析

语法分析阶段定义了对C语言增加了一些额外限制的简化文法，构建出语法分析树供语义分析调用。

语法分析树节点：

```
typedef struct TreeNode {
    int row;    //节点所在行
    char name[MAX_NAME];  //文法中的名字
    char value
        [MAX_VALUE];  //终结符的value是其yytext，用于区分变量名以及确定常数值
    struct TreeNode* children;
    struct TreeNode* next;
} Node;
```

所有的终结符与非终结符都对应一个语法分析树的结点：

```
/*types*/
%union {
    PtrToNode node;
};

/*tokens*/
%token <node> INT FLOAT ID TYPE
%token <node> SEMI COMMA
%token <node> LC RC
%token <node> IF
```

```
11
12 %right <node> ASSIGNOP
13 %left <node> OR
14 %left <node> AND
15 %left <node> RELOP
16 %left <node> PLUS MINUS
17 %left <node> MUL DIV
18 %right <node> NOT
19 %left <node> LB RB LP RP
20 %left <node> DOT
21 %nonassoc <node> LOWER_THAN_ELSE
22 %nonassoc <node> ELSE
23 %nonassoc <node> STRUCT RETURN WHILE
24
25 /*non-terminal*/
26 %type <node> Program ExtDefList ExtDef ExtDecList Specifier
27 %type <node> StructSpecifier OptTag Tag VarDec FunDec VarList
28 %type <node> ParamDec CompSt StmtList Stmt DefList Def DecList
29 %type <node> Dec Exp Args
```

3.1文法解释

高层定义：

```
Program → ExtDefList
ExtDefList → ExtDef ExtDefList
  | ε
ExtDef → Specifier ExtDecList SEMI
  | Specifier SEMI
  | Specifier FunDec CompSt
ExtDecList → VarDec
  | VarDec COMMA ExtDecList
```

Program代表整个程序，Program可产生外部定义列表ExtDefList

ExtDefList是外部定义ExtDef的集合

ExtDef可以产生全局变量(Specifier ExtDecList SEMI)，结构体声明(Specifier SEMI),函数定义(Specifier FunDec CompSt)

ExtDecList是变量声明列表

类型限定符：

```
Specifier → TYPE
  | StructSpecifier
StructSpecifier → STRUCT OptTag LC DefList RC
  | STRUCT Tag
OptTag → ID
  | ε
Tag → ID
```

Specifier是类型限定符，可以产生基本类型限定TYPE(整型和浮点型)，以及结构体类型StructSpecifer。

结构体类型包括包含结构体定义的(STRUCT OptTag LC DefList RC)和只是结构体声明的(STRUCT Tag)。

结构体声明定义时可以不给结构体类型设置名字OptTag->ID | ε

声明体：

```
VarDec → ID
  | VarDec LB INT RB
FunDec → ID LP VarList RP
  | ID LP RP
VarList → ParamDec COMMA VarList
  | ParamDec
ParamDec → Specifier VarDec
```

VarDec(变量声明)可产生普通变量(ID)和数组变量(VarDec LB INT RB)

FuncDec(函数声明)可产生含有参数列表的函数和不含参数列表的函数

VarList产生的参数列表用逗号分隔

ParamDec每一个参数包括类型限定符和变量声明

语句：

```
CompSt → LC DefList StmtList RC
StmtList → Stmt StmtList
  | ε
Stmt → Exp SEMI
  | CompSt
  | RETURN Exp SEMI
  | IF LP Exp RP Stmt
  | IF LP Exp RP Stmt ELSE Stmt
  | WHILE LP Exp RP Stmt
```

语句有大括号语句CompSt，语句列表StmtList和普通语句Stmt

大括号语句产生 { DefList StmtList },此处是C--文法的额外要求，即前半部分可以进行变量声明与定义，后半部分可以进行运算、函数调用、控制结构等，两部分不可交叉

StmtList可产生0个或多个语句

Stmt包括：运算语句(Exp SEMI)、大括号语句(CompSt)、返回语句、if语句和while语句

局部定义：

```
DefList → Def DefList
  | ε
Def → Specifier DecList SEMI
DecList → Dec
  | Dec COMMA DecList
Dec → VarDec
  | VarDec ASSIGNOP Exp
```

定义列表可以产生0个或多个定义

每个Def就是一条变量定义，它包括一个类型描述符Specifier以及一个DecList，例如 int a, b, c;

由于DecList中的每个Dec又可以变成VarDec ASSIGNOP Exp，这允许我们对局部 变量在定义时进行初始化，例如int a = 5；

表达式：

```
Exp → Exp ASSIGNOP Exp
   | Exp AND Exp
   | Exp OR Exp
   | Exp RELOP Exp
   | Exp PLUS Exp
   | Exp MINUS Exp
   | Exp STAR Exp
   | Exp DIV Exp
   | LP Exp RP
   | MINUS Exp
   | NOT Exp
   | ID LP Args RP
   | ID LP RP
   | Exp LB Exp RB
   | Exp DOT ID
   | ID
   | INT
   | FLOAT
Args → Exp COMMA Args
   | Exp
```

支持加减乘除、与或、赋值、非、负数、函数调用(ID LP Args RP和ID LP RP)、数组调用(Exp LB Exp RB)、结构体调用(Exp DOT ID)。直接使用的变量(ID),整数常数和浮点数(INT、FLOAT)

实参列表是一个或多个表达式的值。

运算符优先级：

运算优先级在声明标志符时已经限定：

```
%right <node> ASSIGNOP
%left <node> OR
%left <node> AND
%left <node> RELOP
%left <node> PLUS MINUS
%left <node> MUL DIV
%right <node> NOT
%left <node> LB RB LP RP
%left <node> DOT
%nonassoc <node> LOWER_THAN_ELSE
%nonassoc <node> ELSE
%nonassoc <node> STRUCT RETURN WHILE
```

4. 语义分析

在这一节，我们对语法分析步骤生成的语法树所有结点进行遍历，在遍历的过程中完成语义分析，包括构建符号表、类型检查等，错误检查的很大一部分就在这一步里完成。在遍历的过程中，编译器可以发现源代码的错误，并打印出错误结果。

这一节涉及到的逻辑代码主要在`table.c`中。

4.1 语义假设

这里我们对`C--`语法作出语义的假设，以便于接下来的实现说明：

- 整型和浮点型不能相互赋值或运算
- `if`和`while`语句的条件仅能是整型变量，且仅有整型变量才能进行逻辑运算
- 所有变量的作用域都是全局的，包括了函数的形参、结构体的域
    - 更具体来说，结构体中的域不与变量重名，并且不同结构体中的域互不重名

    - 说明：如果要实现变量作用域，则有符号表栈和`Imperative Style`两种方式，但是由于较为复杂，就没有实现
- 赋值语句的左值只能是：单个变量访问、数组元素访问或结构体特定域的访问
- 结构体之间采用名等价的方式
    - 与gcc保持一致，匿名结构体和其它结构体不等价，结构体之间类型名字不同即不等价

4.2 错误类型

下面列举出对源代码进行语义分析时需要检测出来的所有错误：

1. 变量未经定义即使用
2. 函数未经定义或定义即使用
3. 变量出现重复定义，或变量与前面定义过的结构体本身的名字重复
4. 函数出现重复定义，也即同样的函数名出现了多次定义
5. 赋值号两边的表达式类型不匹配
6. 赋值语句的左值只能是：单个变量访问、数组元素访问或结构体特定域的访问
7. 操作数类型不匹配或操作数类型与操作符不匹配
8. `return`语句的返回值与函数的返回值不匹配
9. 函数调用时实参与形参的数目或类型不匹配
10. 对非数组变量进行数组方式的访问（`[...]`）
11. 对非函数变量使用函数调用（`(...)`）
12. 数组访问的下标出现非整数，如`a[0.5]`
13. 对非结构体变量使用`.`操作符
14. 访问结构体中未定义的域
15. 结构体在定义时对域进行初始化
16. 结构体的名字与其他变量或已经定义过的结构体的名字重复
17. 直接使用未定义过的结构体来定义变量
18. 函数声明了但没有定义
19. 函数声明和定义之间冲突，或函数的多次声明互相冲突

4.3 符号表

在语义分析遍历语法树结点时，我们需要把结点的信息完整地记录下来，从而可以为后面的分析提供前面部分的代码信息。这个过程就涉及到符号表的填充和查询。

我们维护了两个表，分别是变量表和函数表。首先我们说明这两个表的创建和维护操作。我们采用的数据结构是哈希表，这种数据结构可以让我们的插入、查找的时间复杂度都只有$O(1)$，并且代码写起来也方便。这两个表作为全局数组存在，如下：

```
VarType varTable[TABLE_SIZE] = {0};
FuncType funcTable[TABLE_SIZE] = {0};
```

我们的哈希函数读入一个字符串，代表变量名或函数名，哈希函数返回一个非负整型作为哈希表的下标。函数如下：

```c
// hash function by P.J.Weinberger
unsigned int hash_pjw(char *name) {
    unsigned int val = 0, i;
    for (; *name; ++name) {
        val = (val << 2) + *name;
        if ((i = val) & ~TABLE_SIZE) val = (val ^ (i >> 12)) & TABLE_SIZE;
    }
    return val;
}
```

对于符号表，我们需要对两个表分别实现实现插入表项的函数。这里首先说明我们是如何表达一个变量或者一个函数的。

对于一个变量，我们采用如下的结构体`VarType`进行描述，包括：

- 一个变量需要维护的属性包括名字、类型
- 另外在哈希表中，因为由哈希函数得到的index有可能存在多个变量，因此需要记录在哈希表同一表项中的下一个结点
- 这里为了写代码方便，还加入了`next_field`的属性，实际上是复用了`VarType_`这个结构，而不只是表达哈希表中的表项

```c
typedef struct VarType_ *VarType;
struct VarType_ {
    char *name;
    Type type;
    // open hashing
    VarType next;         // 哈希表同一表项中所构成的链表
    VarType next_field;   // 结构体中连接所有成员变量的链表
};
```

注意到上面有用`Type`这个类型，这个结构表达了整型、浮点型、数组、结构体、const型这几个类型，代码如下。首先用一个枚举类型指明这个类型是哪种类型。其次，因为我们希望能完整地表达类型信息，所以需要用一个`union`类型来存储这个类型更详细的信息。

这里采用`union`使得`Type_`成为一个能够只表达其中一种类型的结构，比如说如果一个变量的类型是数组，那么它的`type`成员就会是`ARRAY`。因为我们已经知道了`type`是`ARRAY`，我们就会去查询它的`type_info.array`结构。`union`的写法能够节省内存，提供一种统一的表达形式。

```c
typedef struct Type_ *Type;
typedef struct Structure_ *Structure;
struct Type_ {
    enum {
        BASIC,
        ARRAY,
        STRUCTURE,
        CONSTANT
    } type;  // BASIC可以做左值，CONSTANT不能做左值
    union {
        // 基本类型信息
        enum { INT_TYPE, FLOAT_TYPE } basic;
        // 数组类型信息: 元素类型与数组大小
        struct {
            Type element;
            int size;
        } array;
```

```
        // 结构体类型信息
        Structure structure;
    } type_info;
};

struct Structure_ {
    char *name;
    VarType varList;
};
```

对于一个函数，我们需要记录它的函数名、是否定义、对应行数、返回值类型、参数列表，另外也需要记录哈希表同一表项中所构成的链表的下个结点，对应`table.h`中的`FuncType_`。

```
struct FuncType_ {
    char *name;       //函数名
    bool isDefined;  //函数可只声明不定义
    int row;  //函数最初被识别时在编辑器中的行，可能是声明也可能是定义
    Type returnType;
    VarType param;  // 参数列表
    FuncType next;  // 哈希表同一表项中所构成的链表
};
```

现在，我们说明符号表的查表和插入表操作。这一部分的代码就不列了，否则篇幅过长，这里就用文字说明一下。

查表是简单的，根据变量名字符串得到哈希index，再看看哈希表中该index对应表项的链表中有没有结点的变量名与查询的变量名一样。

插入相对来说会比较复杂。对于普通变量的哈希表，先用哈希函数求出index，再查找哈希表中该表项的链表，如果没有重名，那么就插入到链表尾部。对于函数的哈希表，同样的，找到需要插入的哈希表表项的链表，然后分情况讨论。这里就要根据`isDefined`区分声明和定义了。按照情况以下分类：

- 如果现在要插入一个函数定义
  - 如果在哈希表中该函数已经定义，返回`REDEFINE_ERROR`
  - 如果没有定义，但由于在哈希表中存在这个函数，说明已经声明，但发现参数或返回值不同，则返回`DEF_MISMATCH_DEC`
  - 否则，将该函数设置为已经定义，返回`INSERT_SUCCESS`
- 如果现在要插入一个声明（注意重复的声明不会报错）
  - 如果插入的声明和之前已经存在的声明或定义的参数或返回值不同
    - 该声明和之前的定义不符，返回`DEC_MISMATCH_DEF`
    - 该声明和之前的声明不符，返回`DEC_MISMATCH_DEC`
  - 否则，声明不矛盾，返回`INSERT_SUCCESS`

最后，我们还需要实现打印两个哈希表的函数，便于调试和查看效果。

4.4 根据语法树构建语义分析

现在，我们需要对语法树的每一种非终结符构建其分析过程。这有点类似树的前序遍历。举例来说，对于非终结符`ExtDefList`，有CFG Rule：`ExtDefList -> ExtDef ExtDefList | empty`。那么就可以如下构建分析：

```c
// ExtDefList -> ExtDef ExtDefList | ε
void ExtDefList(Node *n) {
    if (n) {
        Node *child = n->children;
        if (child) {
            ExtDef(child);
            if (child->next) ExtDefList(child->next);
        }
    }
}
```

这里的`Node`是语法树的结点，上面的函数对该结点进行分析，遍历这个结点以及这个结点所有的后代。

分析函数可以有更多的参数，这个参数可以看作是继承属性；分析函数也可以有返回值，提供给父结点做类型检查等错误处理。

在分析的过程中，当遇到4.2节中所遇到的错误时，就需要打印出来。由于我们希望能够检查整个程序的错误，因此当发现一个错误后，结点遍历的分析是不会停下的，这就需要我们处理好有返回值的分析函数，例如返回`NULL`。

这部分的代码量大且杂，涉及很多具体实现细节，因为我们需要仔细地检查上面的若干种错误。如果助教您有任何疑问，可以查看代码的注释，或来问我们小组的成员。

下面以一个比较简单的例子为例：

```c
// Specifier -> TYPE | StructSpecifier
Type Specifier(Node *n) {
    Node *child = n->children;
    if (!child) return NULL;
    if (!strcmp(child->name, "TYPE")) {
        Type t = (Type)malloc(sizeof(struct Type_));
        if (!t) return NULL;
        t->type = BASIC;
        if (!strcmp(child->value, "int")) {
            t->type_info.basic = INT_TYPE;
        } else if (!strcmp(child->value, "float")) {
            t->type_info.basic = FLOAT_TYPE;
        }
        return t;
    } else if (!strcmp(child->name, "StructSpecifier")) {
        Type t = StructSpecifier(child);
        return t;
    }
    return NULL;
}
```

`Specifier`终结符用来表达类型，比如变量定义时的类型、 函数返回值的类型等。因此该分析函数的返回值是`Type`。根据CFG推导式，它的孩子有两种情况，分别是`TYPE`和`StructSpecifier`。如果是`TYPE`，则可以直接得到它的类型，因为`TYPE`只会推得int或float类型。 如果是`StructSpecifier`，则说明是结构体类型，交给结构体函数去解析这个结构体的类型， 并返回这个类型。

语义分析这一节会有针对4.1节列举的错误类型的测试，详见7.1节。

5. 中间代码生成

这一节负责在语义分析的遍历过程中，添加中间代码的生成。我们的中间代码的测试环境是一个IR Simulator程序，即中间代码的解释器，文件和使用说明见`irsim`文件夹。得到中间代码后，可以用该程序运行中间代码。

5.1 中间代码的形式

我们采用线性结构的中间代码。中间代码的形式和操作规范如下表：

语法	描述
LABEL x :	定义标号x。
FUNCTION f :	定义函数f。
x := y	赋值操作。
x := y + z	加法操作。
x := y - z	减法操作。
x := y * z	乘法操作。
x := y / z	除法操作。
x := &y	取y的地址赋给x。
x := *y	取以y值为地址的内存单元的内容赋给x。
*x := y	取y值赋给以x值为地址的内存单元。
GOTO x	无条件跳转至标号x。
IF x [relop] y GOTO z	如果x与y满足[relop]关系则跳转至标号z。
RETURN x	退出当前函数并返回x值。
DEC x [size]	内存空间申请，大小为4的倍数。
ARG x	传实参x。
x := CALL f	调用函数，并将其返回值赋给x。
PARAM x	函数参数声明。
READ x	从控制台读取x的值。
WRITE x	向控制台打印x的值。

这里说明一下`DEC`语句。`DEC x [size]`会分配`size`字节的空间，但是返回给`x`的并不是地址，而是这块空间第一个32位的地址对应的值，就好像是`a[0]`而不是`a`。如果想要获得`x`的地址，还需要添加`a = &x`，这里`x`往往是一个临时变量。

另外说明一下函数调用的语句。`PARAM`语句对函数中的形参进行说明。`CALL`和`ARG`在函数调用时被使用。在调用一个函数之前，我们先使用`ARG`语句传入所有实参，随后使用`CALL`语句调用该函数并存储返回值。注意`ARG`传入参数的顺序和`PARAM`声明参数的顺序正好相反。

最后说明一下`READ`和`WRITE`语句，这两个语句与控制台进行交互，读取或打印的`x`值只能是整型。

5.2 中间代码数据结构

为了表达所有的中间代码，用一个双向链表把一条条中间代码串起来，我们需要记录这个双向链表的头和尾：

```
extern InterCode head;
extern InterCode tail;
```

对于一条中间代码，我们需要记录它的类型，并存储类型对应的操作数是什么。这里的操作数简单地就是理解成上表中那些小写的字母。上表中每一种中间代码对应一个类型。

```
struct InterCode_ {
    enum {
```

```
        LABEL,
        FUNCTION,
        ASSIGN,
        ADD_KIND,
        SUB_KIND,
        MUL_KIND,
        DIV_KIND,
        RIGHTAT,
        GOTO,
        IFGOTO,
        RETURN_KIND,
        DEC,
        ARG,
        CALL,
        PARAM,
        READ,
        WRITE
    } kind;

    union {
        // LABEL, FUNCTION, GOTO, RETURN, ARG, PARAM, READ, WRITE
        struct {
            Operand op;
        } unary;
        // ASSIGN, RIGHTAT, CALL
        struct {
            Operand left, right;
        } assign;
        // ADD, SUB, MUL, DIV
        struct {
            Operand result, op1, op2;
        } binop;
        // IFGOTO
        struct {
            Operand t1;
            char* op;
            Operand t2, label;
        } ifgoto;
        // DEC
        struct {
            Operand op;
            int size;
        } dec;
    } u;
    InterCode prev, next;
};
```

每个操作数也有不同类型，可以是有名变量、无名的临时变量、标签变量、函数、地址等，操作数的表达如下：

```
struct Operand_ {
    enum {
        VAR,
        CONSTANT_OP,
        VAR_ADDRESS,
        LABEL_OP,
        FUNCTION_OP,
```

```
        TMP_VAR,
        TMP_VAR_ADDRESS
    } kind;

    union {
        int var_no;   // TMP_VAR, LABEL_OPs,
        char* value;  // VAR, CONSTANT, FUNCTION_OP
        Operand var;  // VAR_ADDRESS, TMP_VAR_ADDRESS 地址所对应的变量
    } u;
    Operand next;
};
```

这里说明一下临时变量的表达，在中间代码中会出现很多临时变量，负责存储翻译过程中因为编译器而产生的临时的变量。在中间代码中统一用`tx`表达，`x`是非负整数。我们维护了一个全局变量`varNo`，每次用到一个临时变量，该全局变量自增即可。这样处理可能会遇到用户定义变量也叫`t0, t1`等的情况，考虑到目前以实现功能为主，就暂时没有处理。（但其实处理很方便，比如把前缀`t`换成一个乱七八糟的字符串，这样就不可能和用户定义变量重名）

维护这个双向链表是一件很简单的事情，因为我们只需要将中间代码插入到链表中就可以了。这里没有涉及到代码优化的过程，那样的话就需要实现删除中间代码的函数了。

中间代码的数据结构和操作在`InterCode.c` / `InterCode.h`中。

5.3 翻译模式

上面我们说明了中间代码的形式和怎么存储中间代码（存储后即可打印出这些中间代码），这一小节我们需要说明如何在分析语法树的过程中逐条地添加中间代码，即语法制导翻译的过程。

翻译的过程是和语义分析同步进行的，因为它们俩都是遍历语法树的过程，且不需要回头，遍历到什么地方就输出/添加中间代码即可。这样做虽然会导致代码耦合度高，不能模块化，但是效率会高一些。

简单来说，在分析某个语法树结点时，当发现该结点中产生某些特定组合时，就需要构造相应的中间代码，并输出到链表中。

这里首先以语句`Stmt`结点的翻译为例：

translate_Stmt(Stmt, sym_table) = case Stmt of	
Exp SEMI	return translate_Exp(Exp, sym_table, NULL)
CompSt	return translate_CompSt(CompSt, sym_table)
RETURN Exp SEMI	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) code2 = [RETURN t1] return code1 + code2
IF LP Exp RP $Stmt_1$	label1 = new_label() label2 = new_label() code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = translate_Stmt($Stmt_1$, sym_table) return code1 + [LABEL label1] + code2 + [LABEL label2]
IF LP Exp RP $Stmt_1$ ELSE $Stmt_2$	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = translate_Stmt($Stmt_1$, sym_table) code3 = translate_Stmt($Stmt_2$, sym_table) return code1 + [LABEL label1] + code2 + [GOTO label3] + [LABEL label2] + code3 + [LABEL label3]
WHILE LP Exp RP $Stmt_1$	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(Exp, label2, label3, sym_table) code2 = translate_Stmt($Stmt_1$, sym_table) return [LABEL label1] + code1 + [LABEL label2] + code2 + [GOTO label1] + [LABEL label3]

如果发现是表达式`Exp`，则进入翻译`Exp`的过程。`CompSt`同理。

如果发现是`IF`语句（没有`ELSE`部分），则创建两个label作为跳转的两个分支。而后进入到`translate_Cond`部分，这部分是负责处理条件表达式的，其中存在短路运算，并且实现了GOTO的跳转。随后进入`Stmt`的分析，负责生成语句的中间代码。在生成代码时，会一条一条生成并直接插入中间代码的链表，因此这里的返回值只是形式化的表达。剩下两个的道理是类似的。

translate_Cond(Exp, label_true, label_false, sym_table) = case Exp of	
Exp_1 RELOP Exp_2	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp_1, sym_table, t1) code2 = translate_Exp(Exp_2, sym_table, t2) op = get_relop(RELOP); code3 = [IF t1 op t2 GOTO label_true] return code1 + code2 + code3 + [GOTO label_false]
NOT Exp_1	return translate_Cond(Exp_1, label_false, label_true, sym_table)
Exp_1 AND Exp_2	label1 = new_label() code1 = translate_Cond(Exp_1, label1, label_false, sym_table) code2 = translate_Cond(Exp_2, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2
Exp_1 OR Exp_2	label1 = new_label() code1 = translate_Cond(Exp_1, label_true, label1, sym_table) code2 = translate_Cond(Exp_2, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2
(other cases)	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) code2 = [IF t1 != #0 GOTO label_true] return code1 + code2 + [GOTO label_false]

下面说明一下`translate_Cond`是如何工作的。如上图，该函数获得了一个`Exp`和两个分别表示`true`和`false`的label。该函数计算`Exp`的值，并产生`true`和`false`的跳转。

以`Exp1 RELOP Exp2`部分为例的代码进行说明，代码如下：

- 首先创建两个临时操作数
- 其次进入两个`Exp`的翻译过程
- 最后构造跳转的`IFGOTO`语句

```
Type translate_Cond(Node *n, Operand label_true, Operand label_false) {
    Node *child = n->children;
    if (!child) return NULL;
    if (!strcmp(child->name, "Exp")) {
        Node *op_node = child->next;
        if (op_node && !strcmp(op_node->name, "RELOP")) {
            Operand op1 = (Operand)malloc(sizeof(struct Operand_));
            Operand op2 = (Operand)malloc(sizeof(struct Operand_));
            op1->kind = TMP_VAR;
            op1->u.var_no = varNo++;
            op2->kind = TMP_VAR;
            op2->u.var_no = varNo++;

            Node *exp1_node = child;
            Node *exp2_node = op_node->next;
            Type t1 = Exp(exp1_node, op1, NULL);
            Type t2 = Exp(exp2_node, op2, NULL);
            if (!t1 || !t2) return NULL;

            if ((t1->type == BASIC || t1->type == CONSTANT) &&
                (t2->type == BASIC || t2->type == CONSTANT) &&
                t1->type_info.basic == t2->type_info.basic) {
                // IF t1 op t2 GOTO label_true
                InterCode tmp_code =
                    (InterCode)malloc(sizeof(struct InterCode_));
                tmp_code->kind = IFGOTO;
                tmp_code->u.ifgoto.op = op_node->value;
                tmp_code->u.ifgoto.t1 = op1;
                tmp_code->u.ifgoto.t2 = op2;
                tmp_code->u.ifgoto.label = label_true;
                insertInterCode(tmp_code);

                // GOTO label_false
                tmp_code = (InterCode)malloc(sizeof(struct InterCode_));
                tmp_code->kind = GOTO;
                tmp_code->u.unary.op = label_false;
                insertInterCode(tmp_code);

                return t1;
            } else {
                printf("Error type 7 at line %d: Operand type
mismatched\n",
                       child->row);
                return NULL;
            }
            ...
```

说明一下表达式的翻译，如下图：

translate_Exp(Exp, sym_table, place) = case Exp of	
INT	value = get_value(INT) return [place := #value][2]
ID	variable = lookup(sym_table, ID) return [place := variable.name]
Exp_1 ASSIGNOP Exp_2[3] (Exp_1 → ID)	variable = lookup(sym_table, Exp_1.ID) t1 = new_temp() code1 = translate_Exp(Exp_2, sym_table, t1) code2 = [variable.name := t1] +[4] [place := variable.name] return code1 + code2
Exp_1 PLUS Exp_2	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp_1, sym_table, t1) code2 = translate_Exp(Exp_2, sym_table, t2) code3 = [place := t1 + t2] return code1 + code2 + code3
MINUS Exp_1	t1 = new_temp() code1 = translate_Exp(Exp_1, sym_table, t1) code2 = [place := #0 - t1] return code1 + code2
Exp_1 RELOP Exp_2	label1 = new_label() label2 = new_label() code0 = [place := #0] code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = [LABEL label1] + [place := #1] return code0 + code1 + code2 + [LABEL label2]
NOT Exp_1	
Exp_1 AND Exp_2	
Exp_1 OR Exp_2	

上图的过程即对应了代码中的`Exp(...)`函数，参数`place`的含义是，该表达式本身所代表的变量。下面分情况说明：

- 如果`Exp`产生了整数`INT`，那么只需要给`place`操作数加上立即数的表示即可
- 如果产生了`ID`，那么给`place`操作数赋上这个ID对应的变量即可
- 如果产生了`Exp1 ASSIGN Exp2`，这里就需要对左值类型进行判定，同时产生赋值的代码
- 其他基本上也是类似的，可以查看代码理解

最后再说明一下高维数组和结构体的处理。

对于结构体，由于结构体类型是存放在变量哈希表中的，因此对于其中的某个域，我们是可以通过编译器的分析代码直接得到其偏移量的，这就不需要我们去产生中间代码来计算这个偏移量，这使得生成表达结构体域的表达式的中间代码比较好写。我们只需要根据变量表求出偏移量，然后基地址加上偏移量再取值即可。

对于高维数组，会比结构体复杂一些，因为数组的下标有可能是未知的，即编译器在运行的时候并不能够轻易地知道用户取了什么下标（比如`a[i+j][j-i]`），需要经过中间代码计算才知道。求得高维数组某个项的偏移量也是麻烦的事情，因为要一层一层迭代（不过结构体内有结构体也是需要迭代的）。处理高维数组，基本的思路就是从内存最大的开始，一层一层迭代。举例来说会比较简单，假设现在有数组`int a[2][2]`，访问`a[0][1]=2`产生的中间代码可以是：

```
t18 := #8 * #0
t19 := a + t18
t21 := #4 * #1
t22 := t19 + t21
*t22 := #2
```

6. 目标代码生成

目标代码形式：标准MIPS汇编代码。

6.1 目标代码翻译

因为中间代码采用线性结构，所以目标代码的生成相当于对中间代码的逐句翻译。

大致翻译模式如下：

中间代码	MIPS32指令
LABEL x:	x:
x := #k	li reg(x)[1], k
x := y	move reg(x), reg(y)
x := y + #k	addi reg(x), reg(y), k
x := y + z	add reg(x), reg(y), reg(z)
x := y - #k	addi reg(x), reg(y), -k
x := y - z	sub reg(x), reg(y), reg(z)
x := y * z[2]	mul reg(x), reg(y), reg(z)
x := y / z	div reg(y), reg(z) mflo reg(x)
x := *y	lw reg(x), 0(reg(y))
*x = y	sw reg(y), 0(reg(x))
GOTO x	j x
x := CALL f	jal f move reg(x), $v0
RETURN x	move $v0, reg(x) jr $ra
IF x == y GOTO z	beq reg(x), reg(y), z
IF x != y GOTO z	bne reg(x), reg(y), z
IF x > y GOTO z	bgt reg(x), reg(y), z
IF x < y GOTO z	blt reg(x), reg(y), z
IF x >= y GOTO z	bge reg(x), reg(y), z
IF x <= y GOTO z	ble reg(x), reg(y), z

然而并非这么简单，因为中间代码给的是变量、常数、地址，而目标代码所有的运算都是基于寄存器的，所以需要涉及寄存器分配问题。

另外，函数的调用是基于栈的，需要维护栈指针和帧指针。每次调用函数都需要推入栈，存放返回值、当前帧指针，将当前使用的寄存器存储到内存中，退出函数时则弹出栈，取出返回值和当前帧指针。

6.2 寄存器分配

建立寄存器与变量的映射，每个寄存器对应一个变量。当把变量存储到内存中时，释放该寄存器，并利用帧偏移(offset)记录该变量在内存中的位置。

寄存器采用近似LRU算法进行分配。寄存器记录当前使用轮数，每次需要分配寄存器时，将已经分配出去的寄存器轮数(LRU_count)加一。

若存在空闲寄存器，则使用该空闲寄存器。若所有寄存器已经被分配，则返回使用轮数最多的寄存器。将该寄存器中存的值存储到内存中，并记录其在内存中存储的位置。

当调用函数或跳转时，将当前使用了的寄存器全部存入内存。

寄存器分配与变量列表数据结构：

```
typedef struct reg_struct
{
    char name[3];//寄存器名称
    char *vname;//绑定的变量名
    int LRU_count;//当寄存器满的时候，溢出LRU_count值最大的寄存器
}reg;
```

```
7
8  typedef struct var_node
9  {
10     char *vname;//变量名
11     int reg;//对应的寄存器
12     int offset;//变量的帧偏移
13     struct var_node * next;
14 }vnode;
15
16 reg regs[REG_NUM];
```

6.3 函数定义与调用：

函数定义时，删除上一变量列表，重置寄存器。每个函数都是新的变量列表，所以不能使用其他函数中的变量。

然后将寄存器ra和寄存器fp存入栈，读取参数，将参数添加到变量列表。并且给参数分配寄存器

函数调用时，把当前变量列表存储到内存中。便于函数返回时继续使用。然后给函数传参，传参时尽量使用a系列寄存器，如果参数超过四个，a系列寄存器不够用，则将参数存入内存。然后使用`jal`跳转到函数位置，并在接下来写入读取返回值的语句。

6.4 数组与结构体处理：

数组与结构体分配对应中间代码的DEC指令和`x:=&y`指令，对于DEC申请的内存空间，将其放在栈上。并使用`subu \$sp, $sp, decsize;`并将decsize设置为帧偏移的增加量，以后的变量分配内存时建立在当前帧偏移的基础上，达到为其分配内存空间的目的。

然后分配一个寄存器给x，使用`move reg(x),\$sp;`记录其地址。

6.5 地址和常数处理

有些指令不能直接使用常数运算。如`*x=k`，如果k是变量，那么可以将这句话转为`sw reg(k) 0(reg(x));`

但是如果k是常数，就需要先创建一个临时变量，然后将k存放到该变量中，给变量分配寄存器后再调用`sw`指令。

同样的，地址也是不直接分配寄存器的，需要先将地址转化成变量，然后分配寄存器，才能对地址进行操作。

6.6 运算优化

加减法运算时，如果是变量加常数，可以使用`addi`代替指令`li`和`add/sub`。这样可以将两个指令优化成一个指令。

7. 测试

7.1 语义分析测试

样例1：Test/SemanticTest/1.cmm

```c
int main() {
    int i = 0;
    j = i + 1; // j未定义
    return 0;
}
```

```
PS D:\Code\toycompiler\Code> .\parser.exe ..\SemanticTest\1.cmm
Error type 1 at line 3: Undefined variable 'j'
```

样例2：Test/SemanticTest/2.cmm

```c
int main() {
    int i = 0;
    inc(i); // inc()未定义
}
```

```
PS D:\Code\toycompiler\Code> .\parser.exe ..\SemanticTest\2.cmm
Error type 2 at line 3: Undefined function 'inc'
```

样例3：Test/SemanticTest/3.cmm

```c
int main() {
    int i, j;
    int i; // i重复定义
}
```

```
PS D:\Code\toycompiler\Code> .\parser.exe ..\SemanticTest\3.cmm
Error type 3 at line 3: Redefined global variable'i'
```

样例4：Test/SemanticTest/4.cmm

```c
int func(int i) {
    return i;
}
// 函数重复定义
int func() {
    return 0;
}

int main() {}
```

```
PS D:\Code\toycompiler\Code> .\parser.exe ..\SemanticTest\4.cmm
Error type 4 at line 5: Redefinition of function 'func'
```

样例5：Test/SemanticTest/5.cmm

```c
int main() {
    int i;
    i = 3.7; // 浮点数不能赋给整型
}
```

```
PS D:\Code\toycompiler\Code> .\parser.exe ..\SemanticTest\5.cmm
Error type 5 at line 3: Type mismatched
```

样例6：Test/SemanticTest/6.cmm

```
int main() {
    int i;
    10 = i; // 10在赋值号左边
}
```

```
PS D:\Code\toycompiler\Code> .\parser.exe ..\SemanticTest\6.cmm
Error type 6 at line 3: The left-hand side of an assignment must be a variable
```

样例7：Test/SemanticTest/7.cmm

```
int main() {
    float j = 1.7;
    return j; // 返回值类型错误
}
```

```
PS D:\Code\toycompiler\Code> .\parser.exe ..\SemanticTest\7.cmm
Error type 8 at line 3: Return type mismatched
```

样例8：Test/SemanticTest/8.cmm

```
int func(int i) {
    return i;
}

int main() {
    func(1, 2); // 参数不符
}
```

```
PS D:\Code\toycompiler\Code> .\parser.exe ..\SemanticTest\8.cmm
Error type 9 at line : The method 'func(int)'is not applicable for the arguments '(intint)'
```

样例9：Test/SemanticTest/9.cmm

```
int main() {
    int i;
    return i[0]; // i不是数组类型
}
```

```
PS D:\Code\toycompiler\Code> .\parser.exe ..\SemanticTest\9.cmm
Error type 10 at line 3: 'i' must be an array
```

样例10：Test/SemanticTest/10.cmm

```
int main() {
    int i;
    i(10); // i不是函数
}
```

```
PS D:\Code\toycompiler\Code> .\parser.exe ..\SemanticTest\10.cmm
Error type 11 at line 3: 'i' must be a function
```

样例11：Test/SemanticTest/11.cmm

```c
int main() {
    int i[10];
    i[2.5] = 10; // 数组访问错误
}
```

```
PS D:\Code\toycompiler\Code> .\parser.exe ..\SemanticTest\11.cmm
Error type 12 at line 3: Operands type mistaken
```

样例12：Test/SemanticTest/12.cmm

```c
struct Position {
    float x, y;
};

int main() {
    int i;
    i.x; // i不是结构体
}
```

```
PS D:\Code\toycompiler\Code> .\parser.exe ..\SemanticTest\12.cmm
Error type 13 at line 7: Illegal use of '.'
```

样例13：Test/SemanticTest/13.cmm

```c
struct Position {
    float x, y;
};

int main() {
    struct Position p;
    if(p.n == 3.7) //未定义的域
        return 0;
}
```

```
PS D:\Code\toycompiler\Code> .\parser.exe ..\SemanticTest\13.cmm
Error type 14 at line 7: Un-existed field 'n'
```

样例14：Test/SemanticTest/14.cmm

```c
struct Position {
    int x;
};

struct Position { // 结构体重复定义
    int u;
};
```

```
PS D:\Code\toycompiler\Code> .\parser.exe ..\SemanticTest\14.cmm
Error type 16 at line 6: The name of struct 'Position' duplicates name of another variable or st
ruct
```

样例15：Test/SemanticTest/15.cmm

```c
struct Position {
    float x, y;
};

int func(int a);

int func(struct Position p); // 声明冲突

int main() {
    func(1); // 函数未定义
}
```

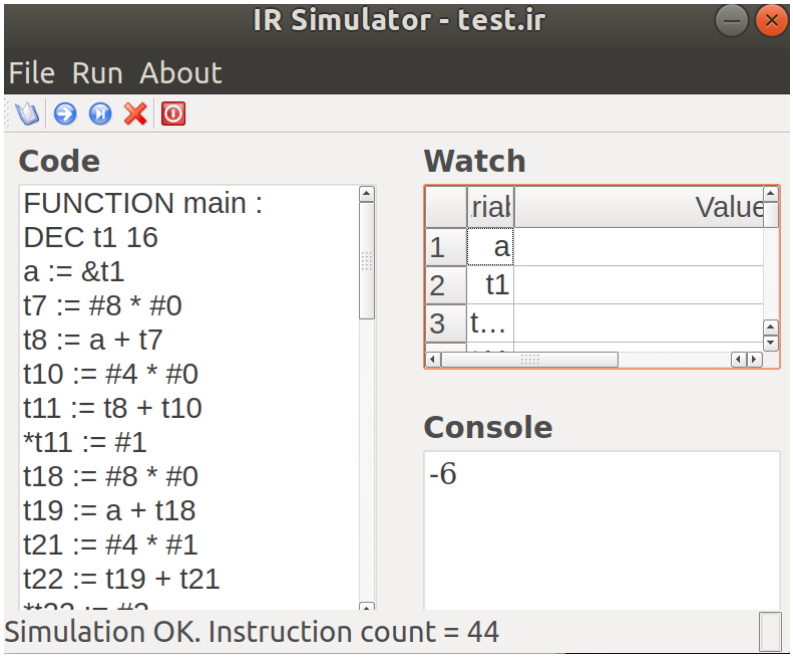
```
PS D:\Code\toycompiler\Code> .\parser.exe ..\SemanticTest\15.cmm
Error type 19 at line 7: Declaration of function 'func' is different from the previous declarati
on
Error type 18 at line 5: Undefined function 'func'
```

7.2 中间代码测试

生成的中间代码在`Test/InterCodeTest/*.ir`中。

样例1：Test/InterCodeTest/1.cmm

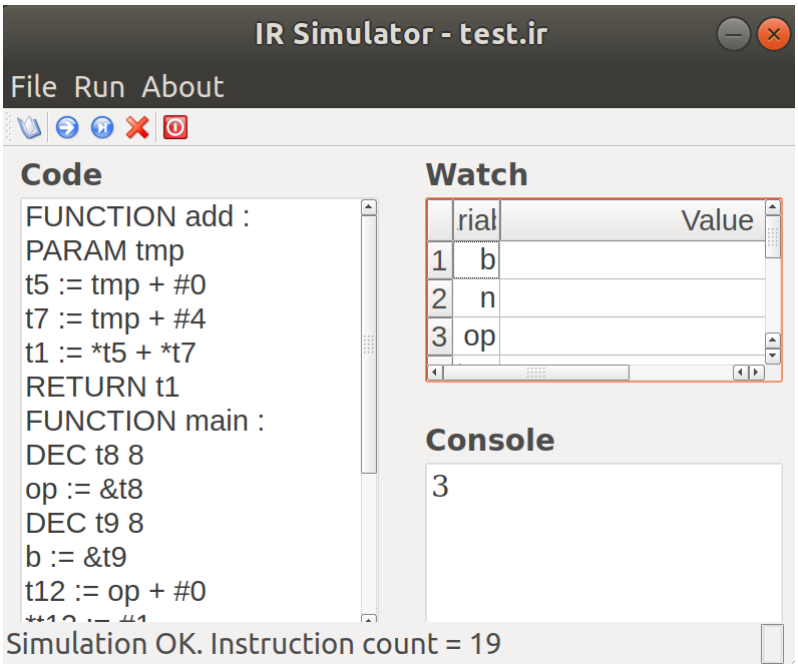
```c
// 高维数组
int main()
{
    int a[2][2];
    a[0][0] = 1;
    a[0][1] = 2;
    a[1][0] = 5;
    a[1][1] = 4;
    write(a[0][0] + a[0][1] - a[1][0] - a[1][1]);
    return 0;
}

```



样例2：Test/InterCodeTest/2.cmm

```c

```

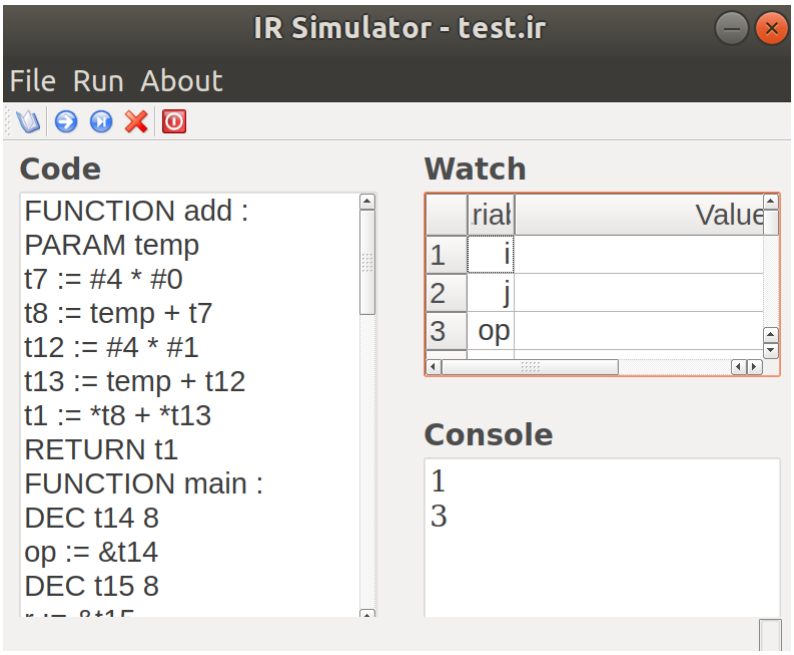
```c
/* 测试结构体 */
struct Operands {
    int o1;
    int o2;
};

/* 结构体作为函数参数 */
int add(struct Operands tmp) {
    return (tmp.o1 + tmp.o2);
}

int main() {
    int n;
    struct Operands op;
    struct Operands b;
    op.o1 = 1;
    op.o2 = 2;
    n = add(op);
    write(n);
    return 0;
}
```



样例3：Test/InterCodeTest/3.cmm

```c
/* 数组类型的变量作为函数参数 */
int add(int temp[2]) {
    return (temp[0] + temp[1]);
}

int main() {
    int op[2];
    int r[1][2]; // 测试高维数组
    int i = 0, j = 0;
    while(i < 2) {
        while(j < 2) {
            op[j] = i + j;
            j = j + 1;
        }
```

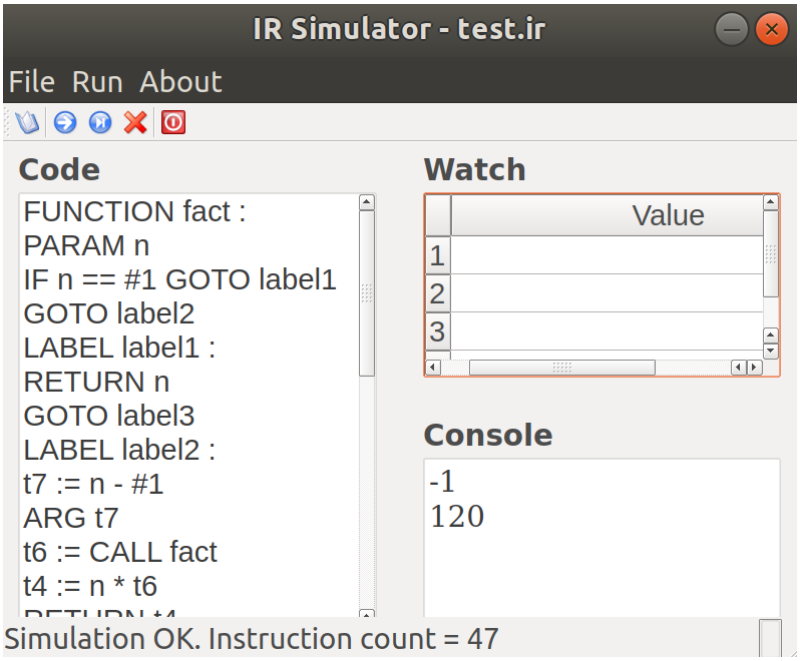
```
        r[0][i] = add(op);
        write(r[0][i]);
        i = i + 1;
        j = 0;
    }
    return 0;
}
```



样例4：Test/InterCodeTest/4.cmm

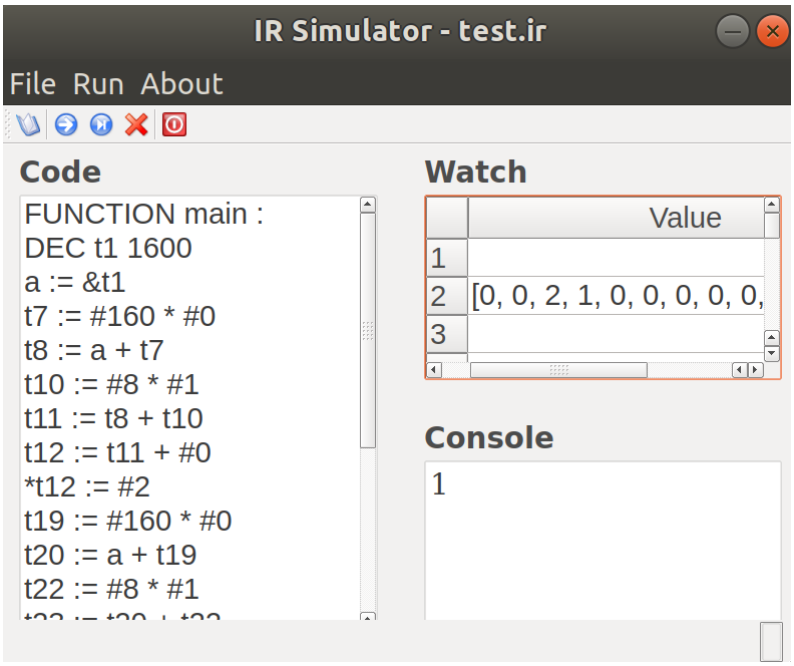
```
/* 求阶乘 */
int fact(int n) {
    if(n == 1)
        return n;
    else
        return (n * fact(n - 1));
}

int main() {
    int m, result = -1;
    write(result); /* 测试变量声明时的赋值 */
    m = read();
    if(m > 1)
        result = fact(m);
    else
        result = 1;
    write(result);
    return 0;
}
```

输入了5，得到了5的阶乘：



样例5：Test/InterCodeTest/5.cmm

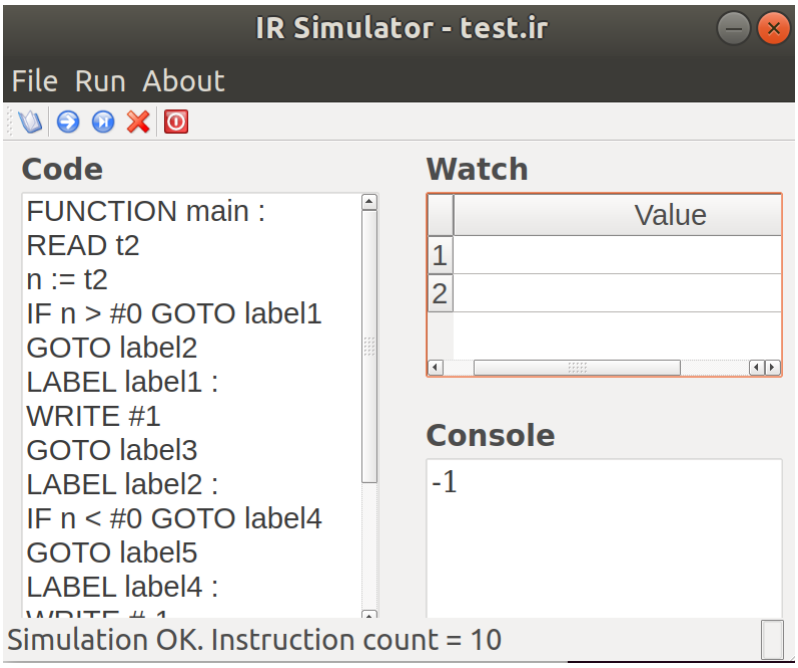
```c
/* 高维结构体数组 */
struct Test {
    int tx, ty;
};

int main() {
    struct Test a[10][20];
    a[0][1].tx = 2;
    a[0][1].ty = 1;
    write(a[0][1].ty);
    return 0;
}
```



样例6：Test/InterCodeTest/6.cmm

```c
/* sgn() */
int main() {
    int n;
    n = read();
    if (n > 0) write(1);
    else if (n < 0) write(-1);
    else write(0);
    return 0;
}
```

输入了-100：

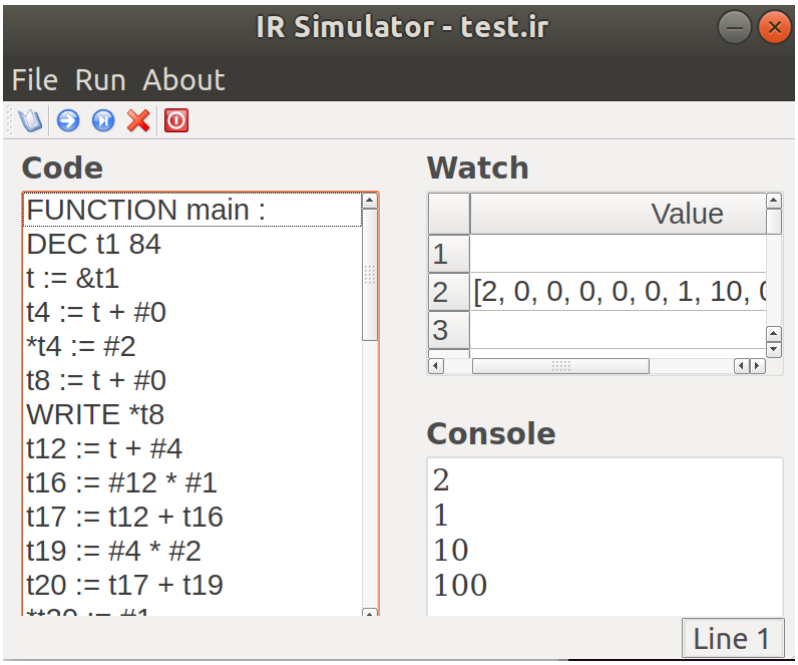


样例7：Test/InterCodeTest/7.cmm

- 这个样例有：结构体中的结构体，结构体中的高维数组，结构体中的高维结构体数组

```c
struct Op {
    int ox, oy;
};

struct Test {
    int tx;
    int ty[2][3];
    struct Op top;
    struct Op top_array[2][3];
};

int main() {
    struct Test t;

    t.tx = 2;
    write(t.tx);

    t.ty[1][2] = 1;
    write(t.ty[1][2]);

    t.top.ox = 10;
    write(t.top.ox);
```

```
23
24      t.top_array[1][2].oy = 100;
25      write(t.top_array[1][2].oy);
26
27  }
```

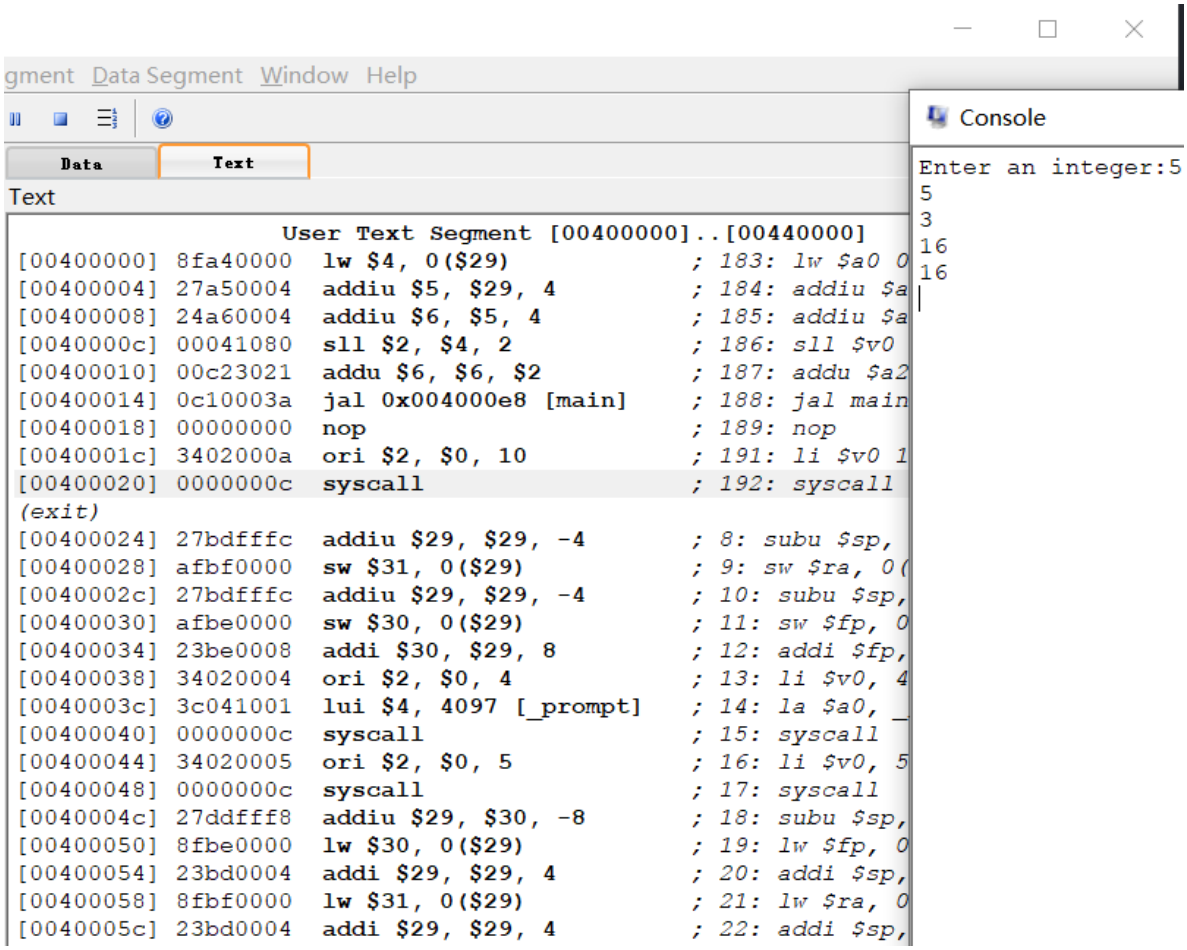


7.3 目标代码测试

样例0：Test/ObjCodeTest/0.cmm

测试函数、read、write、八进制和十六进制

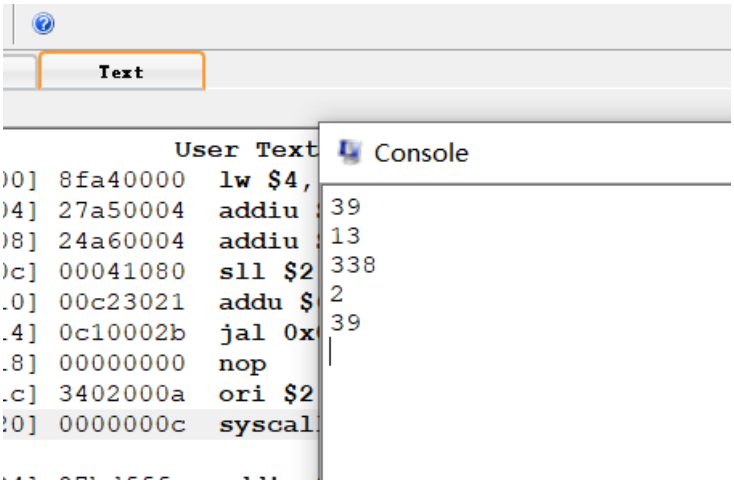
```
1  //测试函数,八进制和十六进制,read和write
2  int f(int a){
3      int x;
4      x=a;
5      return x;
6  }
7  int main(){
8      int i=read();
9      write(i);
10     write(f(3));//打印3
11     write(020);//打印16
12     write(f(0x10));//打印16
13     return 0;
14 }
```



样例1：Test/ObjCodeTest/0-1.cmm

测试十六进制加减乘除

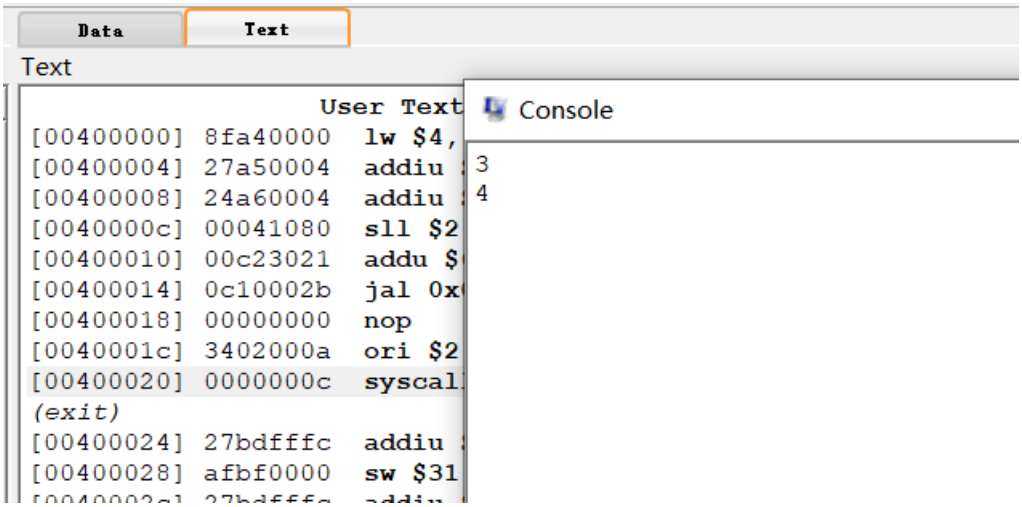
```c
//测试十六进制、八进制加减乘除
int main(){
    int i=0x1A;//26
    int j=0xD;//13
    write(i+j);//39
    write(i-j);//13
    write(i*j);//338
    write(i/j);//2
    write(0x1A+0xD);//39
    return 0;
}
```



样例2：Test/ObjCodeTest/1.cmm

测试结构体

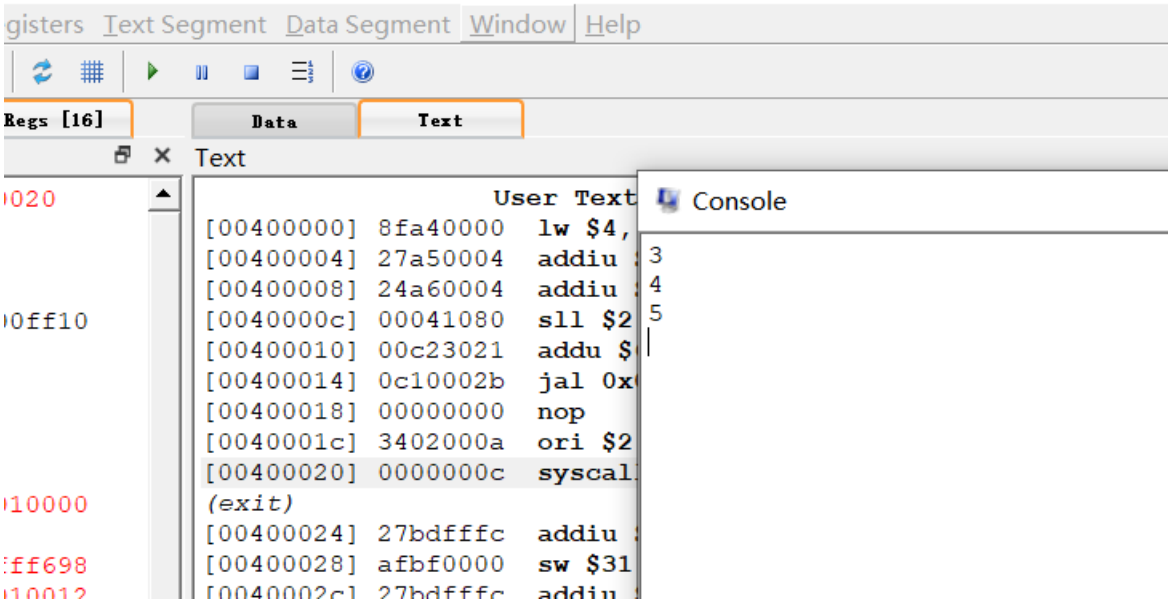
```
//测试结构体
struct Vector
{
    int x,y;
};
int main(){
    struct Vector a;
    a.x=3;
    a.y=4;
    write(a.x);
    write(a.y);
    return 1;
}
```



样例3：Test/ObjCodeTest/1-1.cmm

测试嵌套结构体

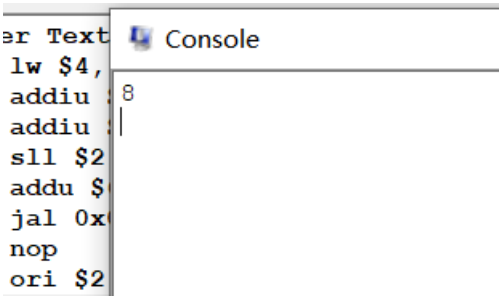
```
//测试嵌套结构体
struct Vector{
    int x;
    int y;
};

int main(){
    int i=3;

    struct Vector2{
        struct Vector v;
        int z;
    }m;
    m.v.x=i;
    m.v.y=4;
    m.z=5;
    write(m.v.x);//3
    write(m.v.y);//4
    write(m.z);//5
    return 0;
}
```



样例4：Test/ObjCodeTest/1-2.cmm

测试结构体传参

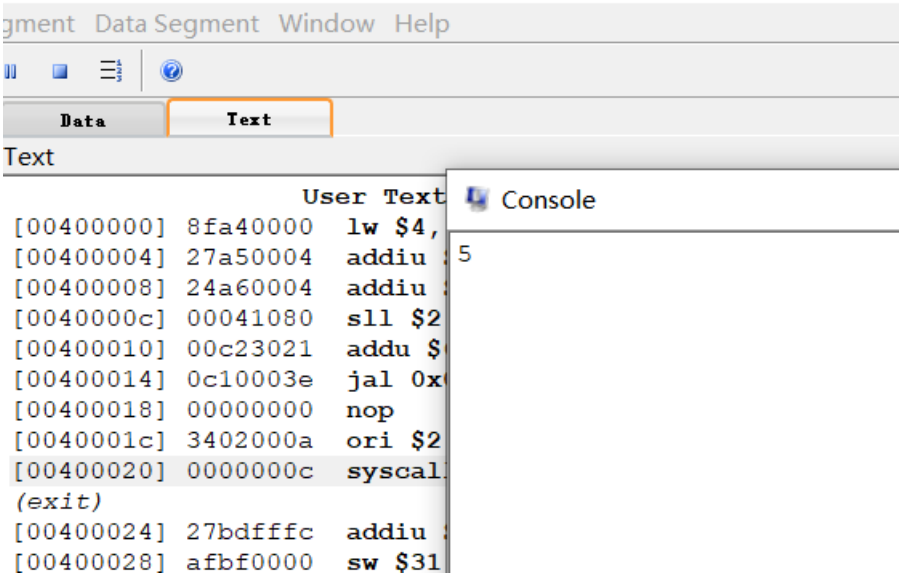
```
//测试结构体传参(可以传参，但是不能做返回值)
struct Vector
{
    int x,y;
};

int f(struct Vector v0){
    return v0.x+v0.y;
}

int main(){
    struct Vector v;
    v.x=3;
    v.y=5;
    write(f(v));
    return 0;
}
```



样例5：Test/ObjCodeTest/2.cmm

测试多个参数

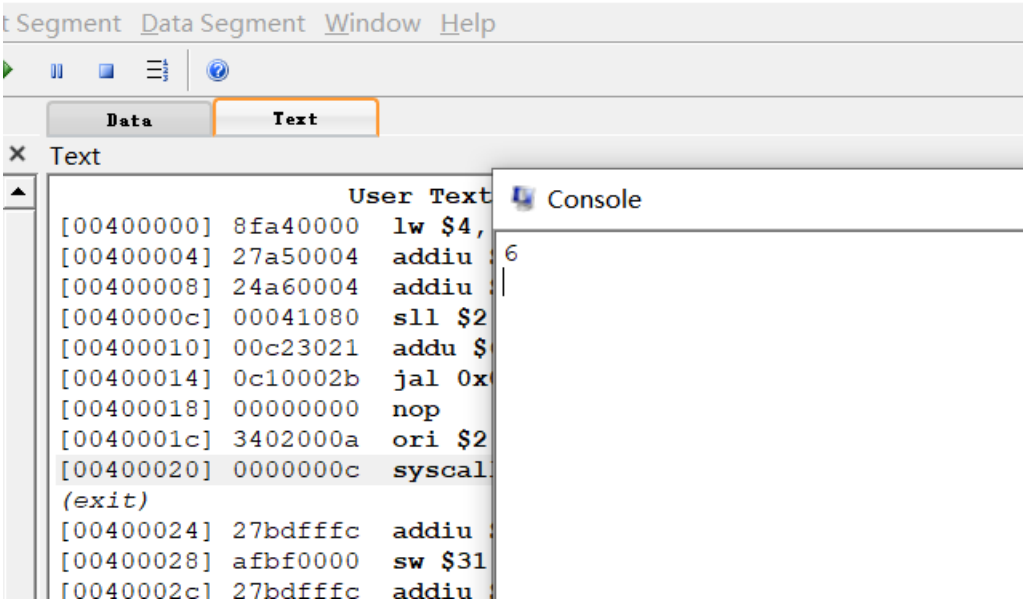
```c
//测试多个参数
int f(int a1,int a2,int a3,int a4,int a5,int a6){
    return a5;
}

int main(){
    int i=f(1,2,3,4,5,6);
    write(i);
    return 0;
}
```



样例6：Test/ObjCodeTest/3.cmm

测试一维数组

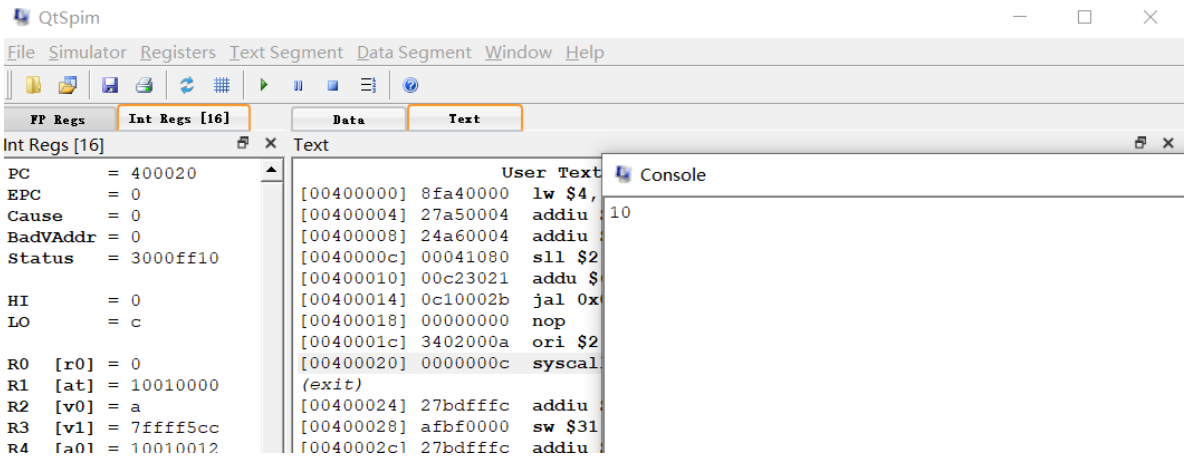
```c
//测试一维数组
int main(){
    int a[5];
    a[0]=3;
    a[2]=3;
    write(a[0]+a[2]);//6
    return 0;
}
```



样例7：Test/ObjCodeTest/3-1.cmm

测试结构体内含一维数组

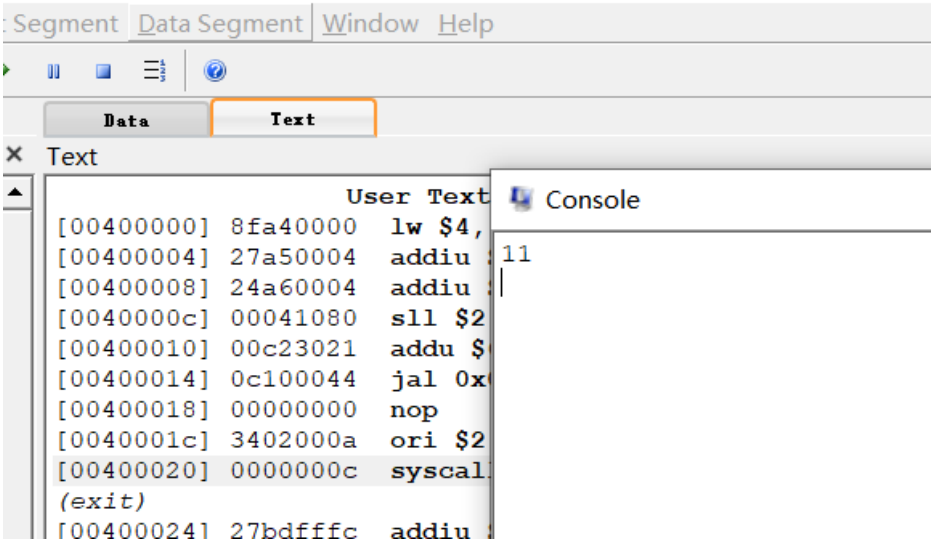
```
//测试结构体内含1维数组
struct Vector{
    int a[10];
    int b;
};

int main(){
    struct Vector x;
    x.a[0]=0;
    x.a[1]=1;
    x.a[2]=2;
    x.a[3]=3;
    x.b=4;
    write(x.a[0]+x.a[1]+x.a[2]+x.a[3]+x.b);
    return 0;
}
```



样例8：Test/ObjCodeTest/3-2.cmm

测试一维数组传参

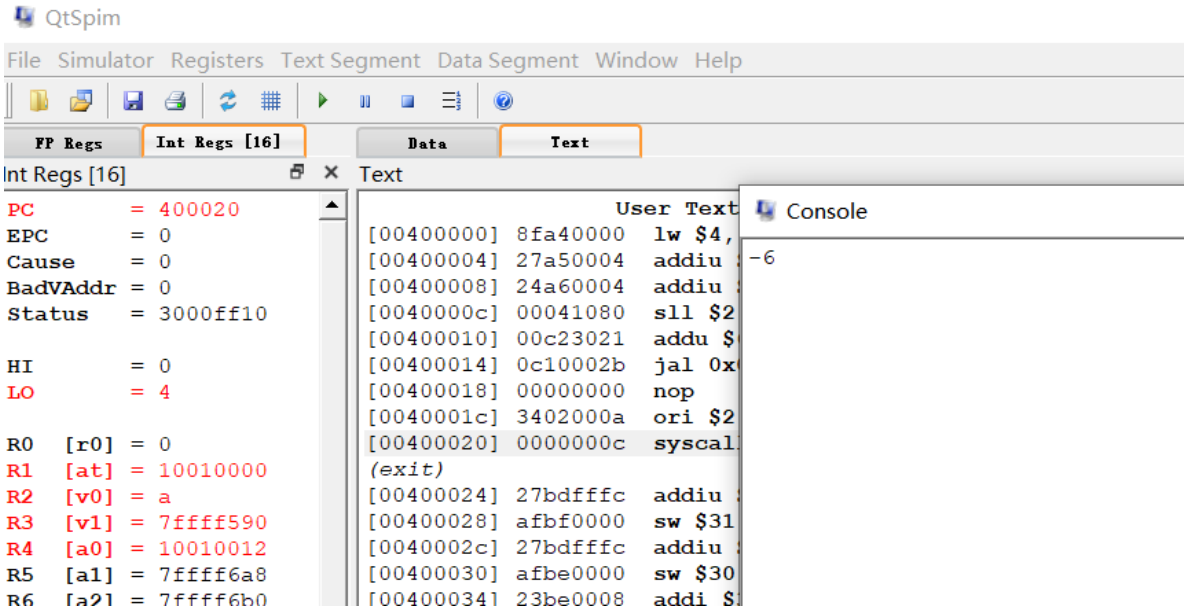
```c
//测试一维数组传参
int f(int a[10]){
    return a[0]+a[1];
}

int main(){
    int x[10];
    x[0]=5;
    x[1]=6;
    x[2]=7;
    write(f(x));
    return 0;
}
```



样例9：Test/ObjCodeTest/4.cmm

测试多维数组

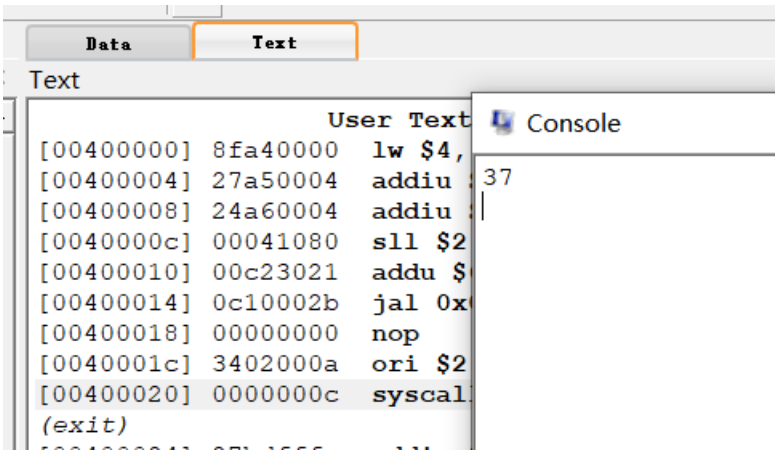
```c
//测试多维数组

int main()
{
    int a[2][2];
    a[0][0] = 1;
    a[0][1] = 2;
    a[1][0] = 5;
    a[1][1] = 4;
    write(a[0][0] + a[0][1] - a[1][0] - a[1][1]);//1+2-5-4=-6
    //write(a[0][0] + a[0][1]);
    return 0;
}
```



样例10：Test/ObjCodeTest/4-1.cmm

测试结构体数组

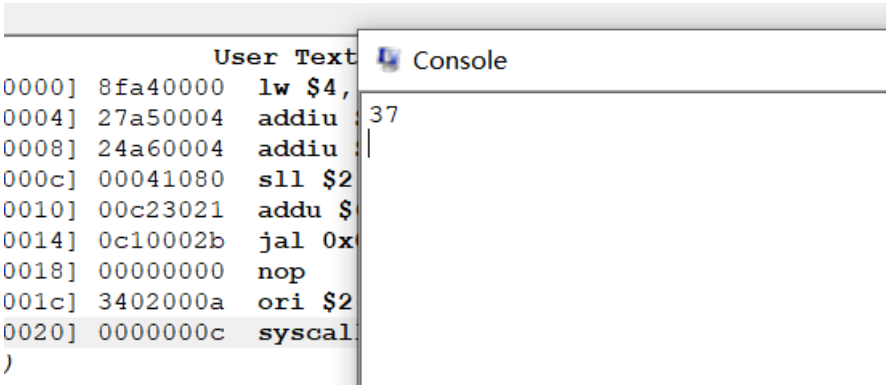
```
//测试结构体数组
int main(){
    struct Vector{
        int x;
        int y;
    }u[10];
    u[2].x=3;
    u[2].y=4;
    u[5].x=5;
    u[6].y=6;
    write(u[2].x+u[2].y+u[5].x*u[6].y);//37
    return 0;
}
```



样例11：Test/ObjCodeTest/4-2.cmm

测试结构体多维数组

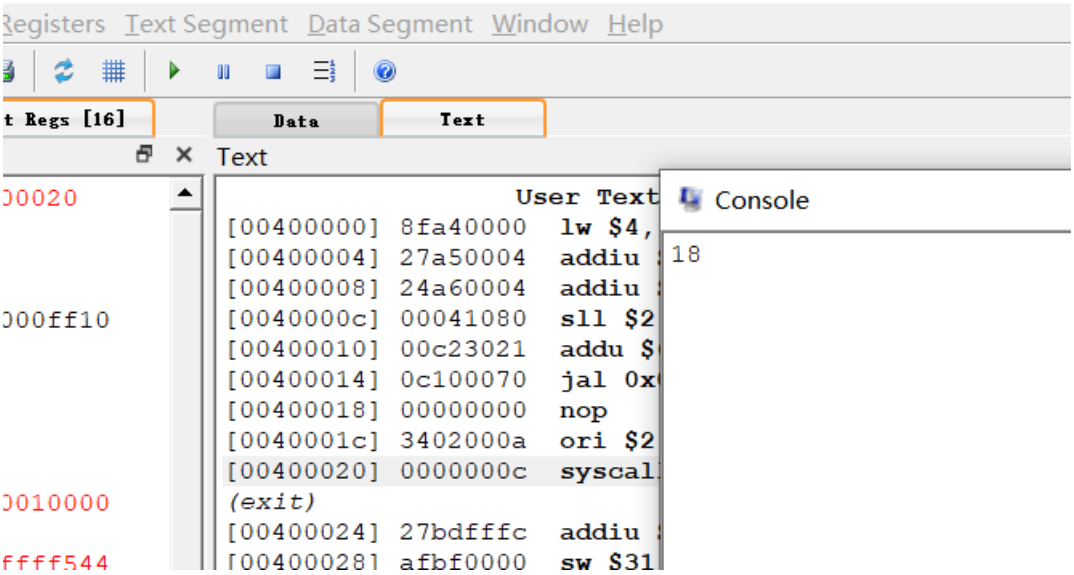
```
//测试结构体多维数组
int main(){
    struct Vector{
        int x;
        int y;
    }u[3][2];
    u[0][0].x=3;
    u[0][1].y=4;
    u[1][0].x=5;
    u[2][0].y=6;
    write(u[0][0].x+u[0][1].y+u[1][0].x*u[2][0].y);//37
    return 0;
}
```



样例12：Test/ObjCodeTest/4-3.cmm

测试多维数组传参

```
//测试多维结数组传参
int f(int a[4][2]){
    return a[0][1]+a[1][0]+a[2][1];
}

int main(){
    int x[4][2];
    x[0][1]=5;
    x[1][0]=6;
    x[2][1]=7;
    write(f(x));
    return 0;
}
```



样例13：Test/ObjCodeTest/5.cmm

测试递归

```
//测试递归
int fact(int n) {
    if(n == 1)
        return n;
    else
        return (n * fact(n - 1));
}

int main() {
    int m=4,result;
    if(m > 1)
        result = fact(m);
    else
        result = 1;
    write(result);
    return 0;
}
```