从入行开始就对编译器有着一种敬畏之心，虽然 c4 省去了链接及输出可执行文件过程，但是通过对 c4 的学习还是能够大体了解到代码是如何被装载如内存再去执行的。

本文是对开源小项目 c4 进行学习的笔记，也可以视为对其实现的内存追踪，为的是能够更加深刻的了解程序实现的具体步骤。本项目的亮点在于仅仅通过 4 个函数就在内存中模拟出了对程序逐字逐句的解析并转换为伪汇编代码，最后在对伪汇编进行解析，输出程序执行结果；当然简洁性的代价就是该编译器仅支持特定语义的解析，并且刚才提到了编译和执行过程都是在内存中去模拟的，并不会生成特定阶段的目标文件比如：`.i`，`.s`，`.out` 文件，但是便于我们去阅读和学习，从中我们可以看出编译器最复杂的地方可能在于语义分析的逻辑部分了。

基本结构与函数的理解

- 全局变量

```
char *p, *lp,   // 指向 source area 的指针，读入的代码会以文本信息的形式读入在该区域
*data;          // data/bss 段指针
int *e, *le,    // 指向 text area 的指针，解析代码后生成相应的伪汇编代码写入该区域
*id,            // 指向当前正在设置或解析的语义关键字结构
*sym,           // 指向 symble area 的指针，能够解析的关键字、库函数以及函数的参数和局部变量会存储在该区域
tk,             // 当前正在解析的代码的游标
ival,           // 当前解析的数字的值
ty,             // 语义关键字结构的类型
loc,            // 局部变量偏移量标识
line,           // 当前解析的代码行数
src,            // 出书解析的代码和生成的伪汇编代码的标识
debug;          // 输出汇编代码执行结果标识
```

- 枚举类型

```

// 类型、关键字、操作码标识
enum {
    Num = 128, Fun, Sys, Glo, Loc, Id,
    Char, Else, Enum, If, Int, Return, Sizeof, While,
    Assign, Cond, Lor, Lan, Or, Xor, And, Eq, Ne, Lt, Gt, Le, Ge, Shl,
Shr, Add, Sub, Mul, Div, Mod, Inc, Dec, Brak
};

// 汇编操作码
enum
{ LEA ,IMM ,JMP ,JSR ,BZ  ,BNZ ,ENT ,ADJ ,LEV ,LI  ,LC  ,SI  ,SC  ,PSH
,

OR  ,XOR ,AND ,EQ  ,NE  ,LT  ,GT  ,LE  ,GE  ,SHL ,SHR ,ADD ,SUB ,MUL ,D
IV ,MOD ,
    OPEN,READ,CLOS,PRTF,MALC,MSET,MCMP,EXIT };

// 数据类型
enum { CHAR, INT, PTR };

// 语义关键字结构，方便进行偏移读取
enum { Tk, Hash, Name, Class, Type, Val, HClass, HType, HVal, Idsz };


```

- 函数

```
/**
 *  从 p 中读取下一个字符并进行处理
 *  通过 tk 进行标识
 *  遍历关键字结构体并用 id 指向该结构首地址
 */
void next();

/**
 *  操作符语义解析(如:sizof，&，|，++，--，=，==等等)
 *  生成伪汇编代码，写到 text area 中
 *  @param lev 操作符标识
 */
void expr(int lev);

/**
```

```
16 *  逻辑关键字解析(如：if，while，return 等)
17 *  生成伪汇编代码，写到 text area 中
18 */
19void stmt();
20
21/**
22 *  读取源代码文件
23 *  生成伪汇编代码
24 *  执行伪汇编代码
25 */
26int main(int argc, char **argv)
```

准备阶段

为了便于调试我利用 `c4.c` 建立一个 Xcode 工程，并且在 `main` 函数中给 `argc``argv` 进行了赋值：

```
 int main(int argc, char **argv)
{
1    int fd, bt, ty, poolsz, *idmain;
2    int *pc, *sp, *bp, a, cycle; // vm registers
3    int i, *t; // temps
4
5    // 为输入参数赋值
6    argc = 3;
7    char **av = argv;
8    ++av;
9    *av = "-d";
10   ++av;
11   *av = "/Users/ronglei/Documents/Code/c4/add.c";
12
13   --argc; ++argv;
14   if (argc > 0 && **argv == '-' && (*argv)[1] == 's') { src = 1; --
15argc; ++argv; }
16   if (argc > 0 && **argv == '-' && (*argv)[1] == 'd') { debug = 1; -
17-argc; ++argv; }
18   if (argc < 1) { printf("usage: c4 [-s] [-d] file ...\n"); return -
19;1; }
20

   ...
```

在此我们以编译一下代码段为例子进行解析：

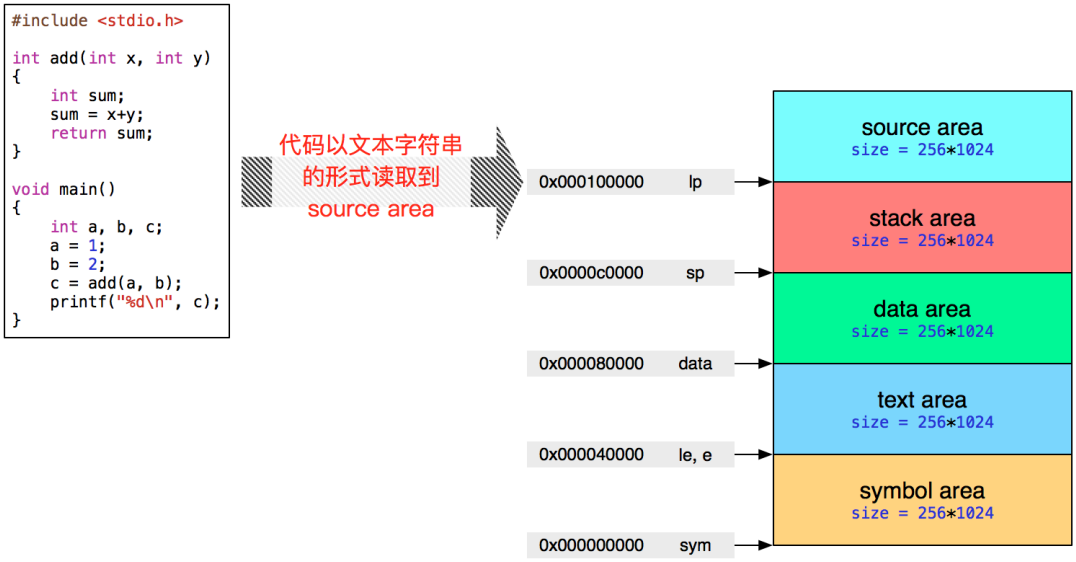
```c
#include <stdio.h>

int add(int x, int y)
{
        int sum;
        sum = x+y;
        return sum;
}

void main()
{
    int a, b, c;
    a = 1;
        b = 2;
    c = add(a, b);
        printf("%d\n", c);
}
```

接下来就是为各个区域申请内存空间：

```c
int main(int argc, char **argv)
{
    ...

    // 申请内存空间
    poolsz = 256*1024; // arbitrary size
    if (!(sym = malloc(poolsz))) { printf("could not malloc(%d) symbol area\n", poolsz); return -1; }
    if (!(le = e = malloc(poolsz))) { printf("could not malloc(%d) text area\n", poolsz); return -1; }
    if (!(data = malloc(poolsz))) { printf("could not malloc(%d) data area\n", poolsz); return -1; }
    if (!(sp = malloc(poolsz))) { printf("could not malloc(%d) stack area\n", poolsz); return -1; }

    memset(sym,  0, poolsz);
    memset(e,    0, poolsz);
    memset(data, 0, poolsz);

    if ((fd = open(*argv, 0)) < 0) { printf("could not open(%s)\n", *argv); return -1; }
    if (!(lp = p = malloc(poolsz))) { printf("could not malloc(%d) source area\n", poolsz); return -1; }
    if ((i = (int)read(fd, p, poolsz-1)) <= 0) { printf("read()
```

```
returned %d\n", i); return -1; }

    ...
```

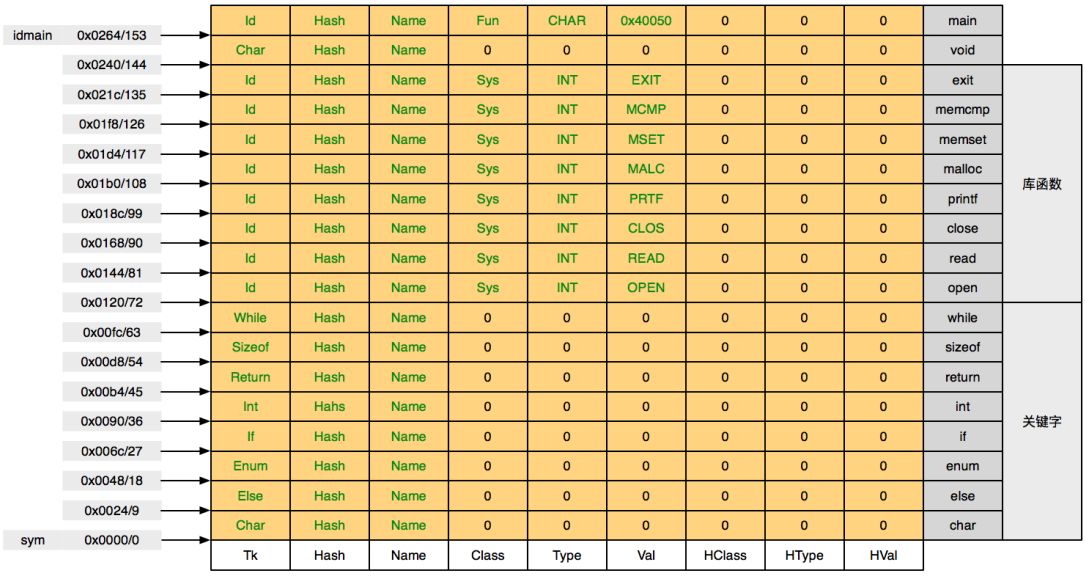
申请完成之后内存布局是这样的：



在申请完 symbol area，text area，data area，stack area 这四片区域之后，程序将支持的关键字和库函数转换为特定结构写入到 symbol 所对应的内存区域中，方便之后解析的时候使用，对应代码：

```
int main(int argc, char **argv)
{
    ...

    p = "char else enum if int return sizeof while "
    "open read close printf malloc memset memcmp exit void main";
    i = Char; while (i <= While) { next(); id[Tk] = i++; } // add keywords to symbol table
    i = OPEN; while (i <= EXIT) { next(); id[Class] = Sys; id[Type] = INT; id[Val] = i++; } // add library to symbol table
    next(); id[Tk] = Char; // handle void type
    next(); idmain = id; // keep track of main

    ...
```

执行完成后 symbol area 内存布局情况如图：



其中 Hash 段和 Name 段是通过关键字或函数名称字符串运算得到到，拿关键字 `char` 举例，源码这里存在一些疏漏，我们会在最后说明：

```
Hash 字段值为：(c+h+a+r)*147*64+strlen(char)
Name 字段值为："*(int *)char..."
```

语义解析阶段

接下来就要进行语义的解析并在 text area 生成响应的伪汇编代码：

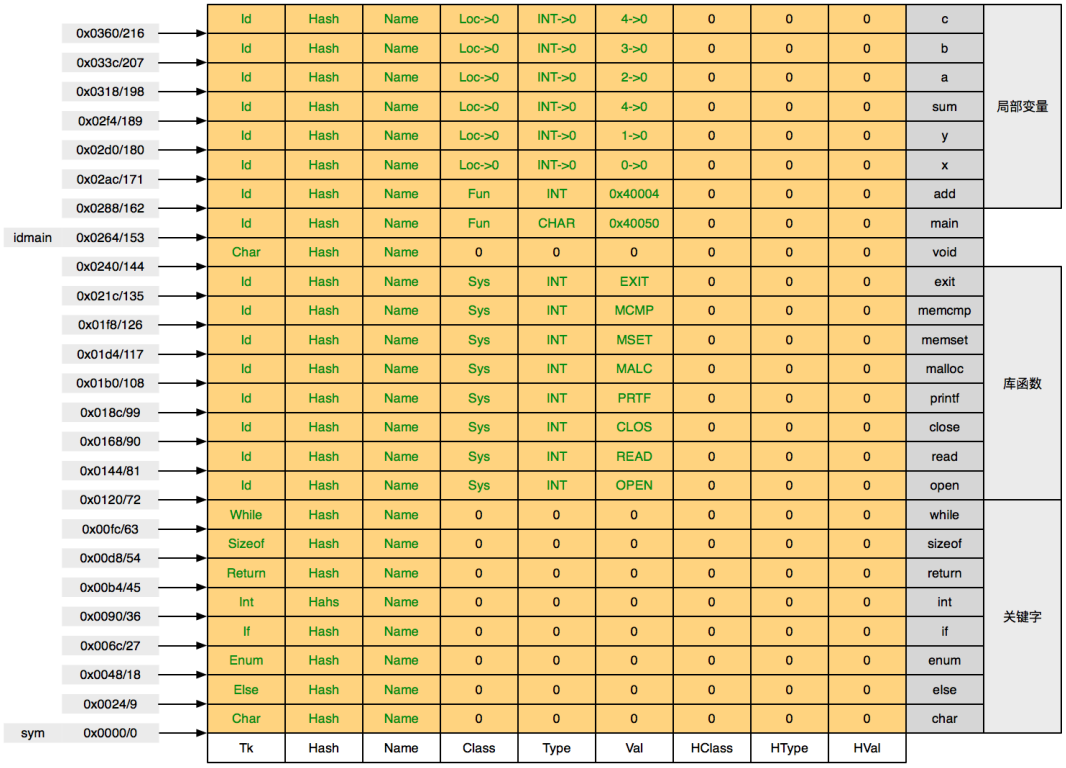
```
int main(int argc, char **argv)
{
        ...

        // parse declarations
    line = 1;
    next();
    // 检测到 int add(...)
    while (tk) {
        bt = INT; // basetype
        if (tk == Int) next(); // 接下来处理 add 函数 id 指向 add 所构建的结构体 tk=Id
        else if (tk == Char) { next(); bt = CHAR; }
        else if (tk == Enum) {
            next();
            if (tk != '{') next();
```

```
            if (tk == '{') {
                next();
                i = 0;
                while (tk != '}') {
                    if (tk != Id) { printf("%d: bad enum
identifier %d\n", line, tk); return -1; }
                    next();
                    if (tk == Assign) {
                        next();
                        if (tk != Num) { printf("%d: bad enum
initializer\n", line); return -1; }
                        i = ival;
                        next();
                    }
                    id[Class] = Num; id[Type] = INT; id[Val] = i++;
                    if (tk == ',') next();
                }
                next();
            }
        }
        while (tk != ';' && tk != '}') {
            ty = bt;
            while (tk == Mul) { next(); ty = ty + PTR; }
            if (tk != Id) { printf("%d: bad global declaration\n",
line); return -1; }
            if (id[Class]) { printf("%d: duplicate global
definition\n", line); return -1; }
            next(); // 过滤掉'('
            id[Type] = ty;
            if (tk == '(') { // function
                id[Class] = Fun;
//                id[Val] = (int)(e + 1);
                id[Val] = (long)(e + 1);
                next(); i = 0;
                while (tk != ')') { // 处理 add 函数的参数列表
                    ty = INT;
                    if (tk == Int) next();
                    else if (tk == Char) { next(); ty = CHAR; }
                    while (tk == Mul) { next(); ty = ty + PTR; }
                    if (tk != Id) { printf("%d: bad parameter
declaration\n", line); return -1; }
                    if (id[Class] == Loc) { printf("%d: duplicate
parameter definition\n", line); return -1; }
                    id[HClass] = id[Class]; id[Class] = Loc;
```

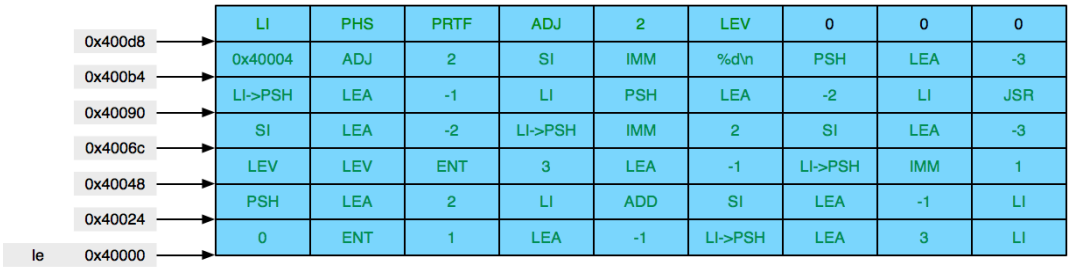
```
                id[HType]  = id[Type];  id[Type] = ty;
                id[HVal]   = id[Val];   id[Val] = i++;
                next();
                if (tk == ',') next();
            }
            next(); // add 函数参数解析完成
            if (tk != '{') { printf("%d: bad function
definition\n", line); return -1; }
            loc = ++i;
            next();
            while (tk == Int || tk == Char) { // 处理函数内的局部变
量
                bt = (tk == Int) ? INT : CHAR;
                next();
                while (tk != ';') {
                    ty = bt;
                    while (tk == Mul) { next(); ty = ty + PTR; }
                    if (tk != Id) { printf("%d: bad local
declaration\n", line); return -1; }
                    if (id[Class] == Loc) { printf("%d: duplicate
local definition\n", line); return -1; }
                    id[HClass] = id[Class]; id[Class] = Loc;
                    id[HType]  = id[Type];  id[Type] = ty;
                    id[HVal]   = id[Val];   id[Val] = ++i;
                    next();
                    if (tk == ',') next();
                }
                next();
            }
            *++e = ENT; *++e = i - loc; // 文本段 ENT 入口 (i-loc)局
部变量个数
            while (tk != '}') stmt(); // 处理函数内部的操作
            *++e = LEV;
            id = sym; // 将函数内部的局部变量的 Class Type Val 清空
            while (id[Tk]) {
                if (id[Class] == Loc) {
                    id[Class] = id[HClass];
                    id[Type] = id[HType];
                    id[Val] = id[HVal];
                }
                id = id + Idsz;
            }
        }
        else {
```

```
                id[Class] = Glo;
                id[Val] = (int)data;
                data = data + sizeof(int);
            }
            if (tk == ',') next();
        }
        next();
    }

    ...
```

再此过程中，会为曾将解析过的函数、函数的参数以及函数作用于内声明的变量，在 symbol area 内生成相应的结构来对他们进行标识和记录，最后在对它们的特定字段进行清理，如图：



在 text area 生成的伪汇编代码后的内存布局情况如图：



汇编执行阶段

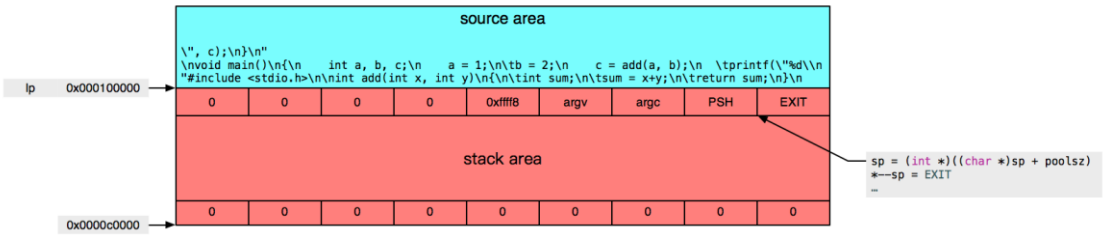
最后就是对 text area 生成的汇编代码进行解析和执行，代码如下：

```
int main(int argc, char **argv)
{
        ...

    // 获取 main 函数入口地址赋值给 pc
    if (!(pc = *(long *)(idmain+Val))) { printf("main() not
defined\n"); return -1; }

    if (src) return 0;
    // 设置栈区域，栈是从高位地址向低位地址扩展的
    sp = (int *)((char *)sp + poolsz);
    *--sp = EXIT; // call exit if main returns
    *--sp = PSH; t = sp;
    *--sp = argc;
    *--sp = (int)argv;
    *--sp = (int)t;

    // run...
    cycle = 0;
    while (1) {
        i = *pc++; ++cycle;
        if (debug) {
            printf("%d> %.4s", cycle,

&"LEA ,IMM ,JMP ,JSR ,BZ  ,BNZ ,ENT ,ADJ ,LEV ,LI  ,LC  ,SI  ,SC  ,PSH
,"

"OR  ,XOR ,AND ,EQ  ,NE  ,LT  ,GT  ,LE  ,GE  ,SHL ,SHR ,ADD ,SUB ,MUL ,
DIV ,MOD ,"
                "OPEN,READ,CLOS,PRTF,MALC,MSET,MCMP,EXIT,"[i * 5]);
```

```
            if (i <= ADJ) printf(" %d\n", *pc); else printf("\n");
        }
        if     (i == LEA) a = (int)(bp + *pc++);
// load local address
        else if (i == IMM) a = *pc++;
// load global address or immediate
        else if (i == JMP) pc = (int *)*pc;
// jump
        else if (i == JSR) { *--sp = (int)(pc + 1); pc = (int *)*pc; }
// jump to subroutine
        else if (i == BZ)  pc = a ? pc + 1 : (int *)*pc;
// branch if zero
        else if (i == BNZ) pc = a ? (int *)*pc : pc + 1;
// branch if not zero
        else if (i == ENT) { *--sp = (int)bp; bp = sp; sp = sp -
*pc++; }     // enter subroutine
        else if (i == ADJ) sp = sp + *pc++;
// stack adjust
        else if (i == LEV) { sp = bp; bp = (int *)*sp++; pc = (int
*)*sp++; } // leave subroutine
        else if (i == LI)  a = *(int *)a;
// load int
        else if (i == LC)  a = *(char *)a;
// load char
        else if (i == SI)  *(int *)*sp++ = a;
// store int
        else if (i == SC)  a = *(char *)*sp++ = a;
// store char
        else if (i == PSH) *--sp = a;
// push

        else if (i == OR)  a = *sp++ |  a;
        else if (i == XOR) a = *sp++ ^  a;
        else if (i == AND) a = *sp++ &  a;
        else if (i == EQ)  a = *sp++ == a;
        else if (i == NE)  a = *sp++ != a;
        else if (i == LT)  a = *sp++ <  a;
        else if (i == GT)  a = *sp++ >  a;
        else if (i == LE)  a = *sp++ <= a;
        else if (i == GE)  a = *sp++ >= a;
        else if (i == SHL) a = *sp++ << a;
        else if (i == SHR) a = *sp++ >> a;
        else if (i == ADD) a = *sp++ +  a;
        else if (i == SUB) a = *sp++ -  a;
```

```
        else if (i == MUL) a = *sp++ *  a;
        else if (i == DIV) a = *sp++ /  a;
        else if (i == MOD) a = *sp++ %  a;

        else if (i == OPEN) a = open((char *)sp[1], *sp);
        else if (i == READ) a = read(sp[2], (char *)sp[1], *sp);
        else if (i == CLOS) a = close(*sp);
        else if (i == PRTF) { t = sp + pc[1]; a = printf((char *)t[-1], t[-2], t[-3], t[-4], t[-5], t[-6]); }
        else if (i == MALC) a = (int)malloc(*sp);
        else if (i == MSET) a = (int)memset((char *)sp[2], sp[1], *sp);
        else if (i == MCMP) a = memcmp((char *)sp[2], (char *)sp[1], *sp);
        else if (i == EXIT) { printf("exit(%d) cycle = %d\n", *sp, cycle); return *sp; }
        else { printf("unknown instruction = %d! cycle = %d\n", i, cycle); return -1; }
    }
}
```

这个阶段利用 pc 读取 text area 生成的伪汇编代码，然后利用指向 stack area 的指针 sp 模拟执行，示意图如下：



纠错

- 查找支持的关键字或库函数过程中，在 `next()`函数以下代码段中：

```
void next()
{
...
 else if ((tk >= 'a' && tk <= 'z') || (tk >= 'A' && tk <= 'Z') || tk == '_') {
    pp = p - 1;
    while ((*p >= 'a' && *p <= 'z') || (*p >= 'A' && *p <= 'Z') || (*p >= '0' && *p <= '9') || *p == '_')
      tk = tk * 147 + *p++;
    tk = (tk << 6) + (p - pp);
    id = sym;
    while (id[Tk]) {
      if (tk == id[Hash] && !memcmp((char *)id[Name], pp, p -
```

```
pp)) { tk = id[Tk]; return; }
     id = id + Idsz;
   }
   id[Name] = (int)pp;
   id[Hash] = tk;
   tk = id[Tk] = Id;
   return;
  }
 ...
 }
```

从代码中我们可以看出，在对之前添加的关键字或函数进行匹配时，是通过对 Hash 段和 Name 段进行比较得出的，条件表达式为即：

```
// 这条语句在编译的时候编译器会报错

tk == id[Hash] && !memcmp((char *)id[Name], pp, p - pp)

// 正确的写法

tk == id[Hash] && !memcmp((char *)(id+Name), pp, p - pp)
```

当我们对 Name 进行存储的时候是将字符串指针强转为 int 型去存储的，即：

```
id[Name] = (int)pp;
```

这里面作者想表达的意图是，将字符串指针强转为整形指针存入到 Name 字段中，所以对 Name 字段进行存储是应该做如下修改：

```
id[Name] = *(int *)pp;
```

注意由于`char`类型占 1 个字节，`int`类型占 4 个字节，所以此时 Name 段记录的字符为 pp 所指向字符的前 4 个字节：
如果`pp = "char else..."`，则 Name 字段内容转换为字符串为`"char"`；
如果`pp = "if else"`，则 Name 字段内容转换为字符串为`"if e"`；
如果`pp = "return else"`，则 Name 字段内容转换为字符串为`"retu"`；
因此在对该字段内容进行比较时，对比较的长度的要求就比较特殊，上面条件判断就应该改为：

```
// 如果比较的字符串长度小于 4 个字节，比较长度就取该长度；

// 如果比较的字符串长度大于4个字节,比较长度就取4个字节长度进行比较，
```

因为在存储的之后我们只记录了字符串的前 4 个字节；

```
int cmp_len = (p - pp) > 4 ? 4 : (p - pp);

if (tk == id[Hash] && !memcmp((char *)(id+Name), pp, cmp_len))

{

tk = id[Tk];

return;

}
```

* 在对生成的汇编代码进行模拟运行的时候，需要获取之前存储的 `main` 函数以及自定义函数的地址进行跳转，而当时存储的时候代码如下：

```
id[Val] = (int)(e + 1);
```

我们看到是将地址值强转成 `int` 型去存储的，对于 32 位机器来说，地址是通过 8 位 16 进制数去表示的，地址的长度正好是 `int` 型的长度 4 个字节；但在 64 位机器上，地址是通过 16 位 16 进制数表示，地址长度为 8 个字节，此处如果想用 `int` 型去存储地址的话，高 4 位地址会被截取，存储的只是低 4 位地址的内容，在跳转的时候就会出问题。