

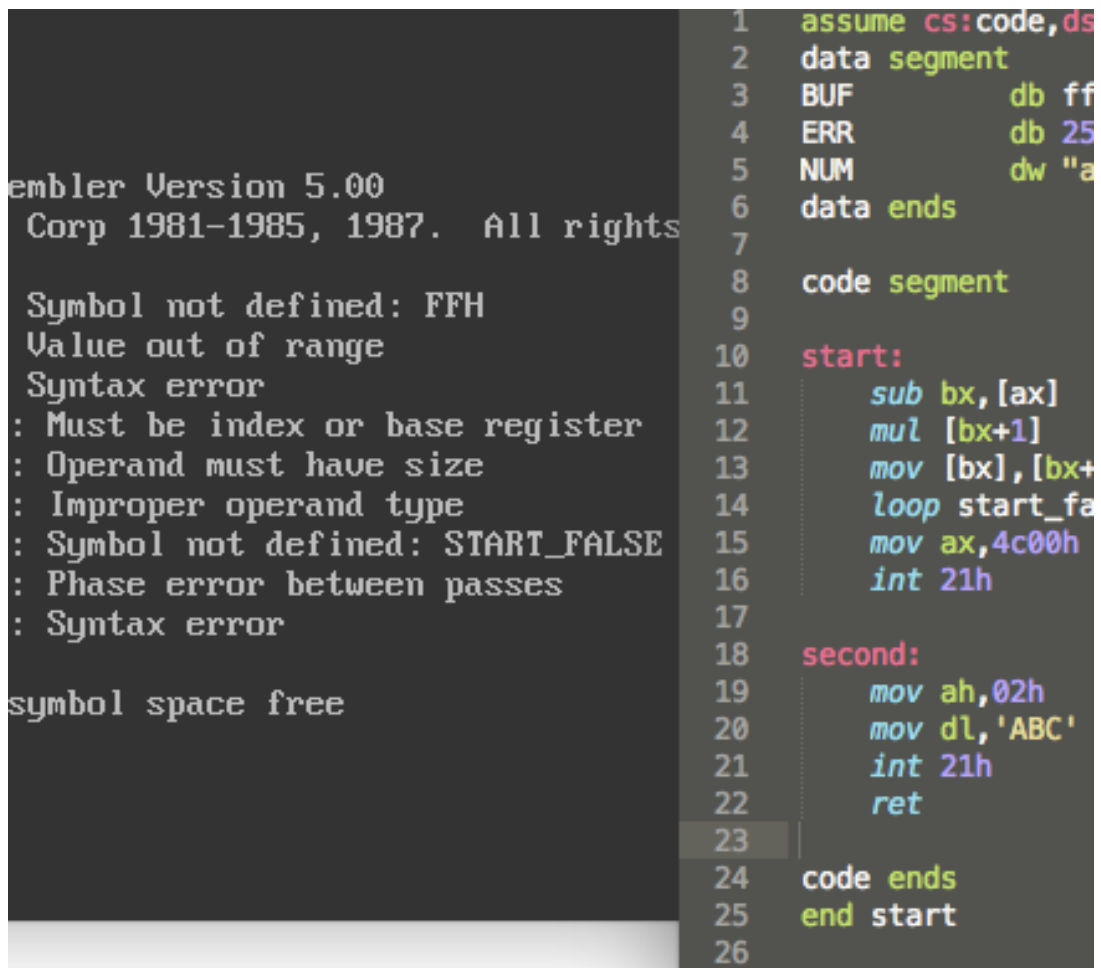
小型MASM编译器

李新星, PB13214040

简介：利用Lex和Bison (Yacc) 工具，实现对MASM语言源文件的读入和分析，目的是实现一款适用于教学的编译器，以便于新手学习汇编语言

一.为什么要做小型MASM编译器（价值&意义）

- gcc和clang
 - 已经有了gcc编译器，为什么还要做clang呢？是因为两款编译器的目的不同：clang是为了更高的效率



```
emblem Version 5.00
  Corp 1981-1985, 1987.  All rights reserved.

Symbol not defined: FFH
Value out of range
Syntax error
: Must be index or base register
: Operand must have size
: Improper operand type
: Symbol not defined: START_FALSE
: Phase error between passes
: Syntax error

symbol space free

1  assume cs:code,ds
2  data segment
3  BUF      db  ff
4  ERR      db  25
5  NUM      dw  "a
6  data ends
7
8  code segment
9
10 start:
11     sub  bx,[ax]
12     mul  [bx+1]
13     mov  [bx],[bx+1]
14     loop start_fa
15     mov  ax,4c00h
16     int  21h
17
18 second:
19     mov  ah,02h
20     mov  dl,'ABC'
21     int  21h
22     ret
23
24 code ends
25 end start
26
```

- 微软的MASM编译器缺点：
 - 如上图，微软的MASM编译器是一款强大的编译器，它很适合软件工程设计，但是它的报错信息非常简略，不适合新手学习汇编语言
 - 词法和语法错误报错信息不完善，只告诉你错，不告诉你为什么错，我希望我设计的编译器可以找出出现错误的行号、列号，输出源文件对应行的源码，以及建议用户如何书写正确的语法，如下图

```
[ERROR]: test/err1.asm: 5.(17) Wrong to store ASCII to [word]! please use [byte]
```

```
num      dw  "abcd"
           ^
```

- 无法检测运行时错误
- 无法预测程序输出什么字符。clang会在用户用错%d、%s、&a时给出提示，而汇编程序可能会蠢蠢地输出一堆不可见字符
- 配合DOS获得更多缺点：
 - 一旦陷入死循环就会死机
 - 有些版本只有大写、单色字母（太丑）
 - 在linux和mac上只能用DOS虚拟机来编译masm源文件，其实不是本地编译

二.我做了哪些工作

1. 自己构建语法结构，实现词法分析、语法分析

```

assume cs:code,ds:data
data segment
BUF      db 0ffh dup(?)          ; 存读到
ERR      dw 256
NUM      db "abcd"
data ends

code segment
printf MACRO x
test:
    mov x,12
    int 21h
    ENDM

start:
    mov ax,offset BUF          ; mov指令 + 偏移量寻址
    add ax,ERR                 ; add指令 + 直接寻址
    sub bx,[bx]                ; sub指令 + 寄存器寻址
    mul byte ptr [bx+1]        ; mul指令 + 寄存器加偏移
    div word ptr [bx+di+1]     ; div指令 + 更灵活的寻址

    inc NUM                    ; inc指令
    dec cx                     ; dec指令
    loop start                 ; loop指令
    cmp cx,ax                  ; cmp指令
    jge start                  ; 比较指令
    call second                ; call指令
    push ax                    ; push指令
    pop ax                     ; pop指令
    mov ax,4c00h
    int 21h                    ; 调用中断驱动，结束程序

second:                          ; 定义函数
    mov ah,02h
    mov dl,'A'
    int 21h
    ret

code ends
end start

```

1.1支持的结构有：assume、数据段代码段分离、宏

支持的指令	支持的寻址方式
mov	offset寻址
add,sub	直接寻址
mul,div	寄存器寻址
inc,dec	寄存器加立即数寻址
loop,jmp,je,jle,jg,jge,jne	基础寄存器加(si/di)加偏移寻址
cmp	
call,ret	
push,pop	
int	

1.2 源码中的词法单元：

- num_tok 十进制数字和十六进制数字
- var_tok 不是关键字key的用户自定义名字，例如print
- String 夹在双引号或单引号中间的字符串
- AnoBegin/End /* */ 注释
- ; 分号，单行注释
- 其它各种保留字key_tok
- 其它各种单个符号 (' " ' (' ' [' 等)

1.3 源码中的文法结构：

```
input : /*empty*/
      | input CompUnit
      ;
CompUnit : Assume
          | Segment
          | end_tok var_tok
          ;

Assume : assume_tok cs_tok ':' code_tok
        | assume_tok ds_tok ':' data_tok
        | assume_tok ss_tok ':' stack_tok
        | Assume ',' cs_tok ':' code_tok
        | Assume ',' ds_tok ':' data_tok
        | Assume ',' ss_tok ':' stack_tok
        ;

Segment : data_tok segment_tok Alloca data_tok ends_tok
        | stack_tok segment_tok Alloca stack_tok ends_tok
        | code_tok segment_tok Content code_tok ends_tok
        ;

Content : Block
        | Macro
        | Content Block
        | Content Macro
        ;

Alloca : AllocaLine
        | Alloca AllocaLine
        ;

AllocaLine : var_tok db_tok AllocaExpH
            | var_tok dw_tok AllocaExpH
            | var_tok dd_tok AllocaExpH
            ;

AllocaExpH : AllocaExp
            | AllocaExpH ',' AllocaExp
            ;
```

```

AllocaExp      : string_tok
                | num_tok
                | num_tok dup_tok '(' num_tok ')'
                ;

RegS           : ax_tok
                | bx_tok
                | cx_tok
                | dx_tok
                | si_tok
                | di_tok
                | ds_tok
                | ss_tok
                | sp_tok
                | offset_tok var_tok
                ;

RegE           : ah_tok
                | al_tok
                | bh_tok
                | bl_tok
                | ch_tok
                | cl_tok
                | dh_tok
                | dl_tok
                ;

MemAddr        : var_tok
                | data_tok
                | stack_tok
                | code_tok
                | MemAddrExp
                ;

MemAddrExp     : '[' num_tok ']'
                | '[' MemAddrExp_a ']'
                | '[' MemAddrExp_a '+' MemAddrExp_a ']'
                | '[' MemAddrExp_a num_tok ']'
                | '[' MemAddrExp_a '+' MemAddrExp_a num_tok ']'
                ;

MemAddrExp_a   : ax_tok
                | bx_tok
                | cx_tok
                | dx_tok
                | si_tok
                | di_tok
                ;

DoubleOE       : RegE ',' RegE
                | RegE ',' num_tok
                | RegE ',' string_tok
                | RegE ',' MemAddr
                | MemAddr ',' RegE
                | MemAddr ',' num_tok

```

```

      | MemAddr ',' MemAddr
      | byte_tok ptr_tok MemAddr ',' MemAddr
      | byte_tok ptr_tok MemAddr ',' num_tok
      ;
Double0S : RegS ',' RegS
      | RegS ',' num_tok
      | RegS ',' MemAddr
      | MemAddr ',' RegS
      | word_tok ptr_tok MemAddr ',' MemAddr
      | word_tok ptr_tok MemAddr ',' num_tok
      ;
Instruction : Int_ins
      | Mov_ins
      | Cmp_ins
      | Jmp_ins
      | Add_ins
      | Sub_ins
      | Mul_ins
      | Div_ins
      | Call_ins
      | Push_ins
      | Pop_ins
      | Ret_ins
      | Block
      ;

Int_ins   : int_tok num_tok
      ;

Mov_ins   : mov_tok Double0E
      | mov_tok Double0S
      ;

Cmp_ins   : cmp_tok Double0E
      | cmp_tok Double0S
      ;

Jmp_ins   : jmp_tok var_tok
      | jle_tok var_tok
      | jl_tok var_tok
      | jge_tok var_tok
      | jg_tok var_tok
      | je_tok var_tok
      | jne_tok var_tok
      | loop_tok var_tok
      ;

Add_ins   : add_tok Double0E
      | add_tok Double0S
      | inc_tok RegE
      | inc_tok RegS
      | inc_tok MemAddr

```

```

Sub_ins      :
              : sub_tok DoubleOE
              | sub_tok DoubleOS
              | dec_tok RegE
              | dec_tok RegS
              | dec_tok MemAddr
              ;

Mul_ins      : mul_tok RegE
              | mul_tok num_tok
              | mul_tok RegS
              | mul_tok MemAddr
              | mul_tok word_tok ptr_tok MemAddr
              | mul_tok byte_tok ptr_tok MemAddr
              ;

Div_ins      : div_tok RegE
              | div_tok RegS
              | div_tok num_tok
              | div_tok MemAddr
              | div_tok word_tok ptr_tok MemAddr
              | div_tok byte_tok ptr_tok MemAddr
              ;

Call_ins     : call_tok var_tok
              ;

Push_ins     : push_tok RegE
              | push_tok RegS
              | push_tok num_tok
              ;

Pop_ins      : pop_tok RegE
              | pop_tok RegS
              ;

Ret_ins      : ret_tok
              ;

Block        : var_tok ':' BlockItem
              | var_tok ':'
              ;

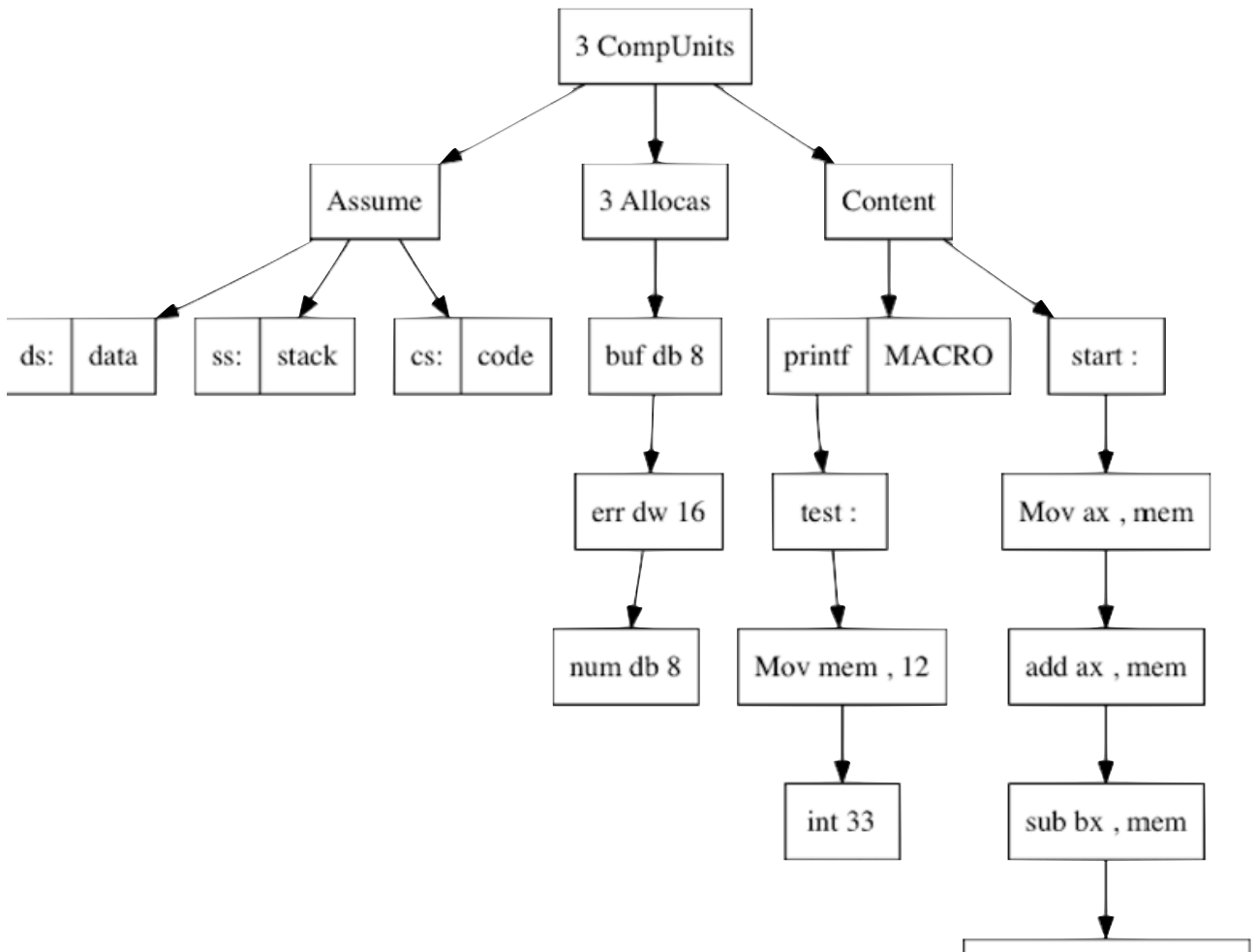
BlockItem    : Instruction
              | BlockItem Instruction
              ;

Macro        : var_tok macro_tok MacroExp BlockItem endm_tok
              ;

MacroExp     : var_tok
              | MacroExp ',' var_tok
              ;

```

2.构建抽象语法树AST，利用dumpdot函数和dot工具，绘制语法树的图（图片放在masm/watch.png）。



3.在语法分析时检查简单词法、语法错误

```
a5 ASTER - bash - 90x30
rishinseitekiMacBook-Air:a5 ASTER Li$ ./bin/parser -d asgn.dot test/err1.asm
[WARNING]:test/err1.asm: 3.(22) Hex Number should begin with '0-9', instead of 'a-f'

buf          db ffh dup(?)          ; 存读到的数字
               ^
[ERROR]: test/err1.asm: 4.(16) Can't store Number over 256 to [byte]!

err          db 257
               ^
[ERROR]: test/err1.asm: 5.(17) Wrong to store ASCII to [word]! please use [byte]

num          dw "abcd"
               ^
[ERROR]: test/err1.asm: 11.(13) There can't be ax! please use bx|si|di|num

sub bx,[ax]
               ^
[WARNING]:test/err1.asm: 12.( 8) Mem length missing,do you mean 'word ptr' or 'byte ptr'?

mul [bx+1]
    ^
[ERROR]: test/err1.asm: 13.(14) [Severe ERROR]:It's wrong to 'mov mem,mem'

mov [bx],[bx+1]
    ^
[ERROR]: test/err1.asm: 20.(12) Too long string!

mov dl,'abc'
    ^
```

3.1 错误类型（如上图）：

- 十六进制未以数字开头的错误
- 分配内存空间时db/dw/dd符号使用错误
- 误在[]中使用ax、cx、dx的错误
- 操作数中未指定内存长度的错误
- 操作数为两个内存地址的错误
- 误将一个大数（或多于1位的ASCII码）存入一个较小空间的错误

4. 建立符号表，检查引用Label和Mem时的错误

```
a5 ASTER — bash — 90x11
mov ax,num_f
^
rishinseitekiMacBook-Air:a5 ASTER Li$ ./bin/parser -d asgn.dot test/err2.asm
[ERROR]: test/err2.asm: 12.(12) The Memory address isn't defined : num_f

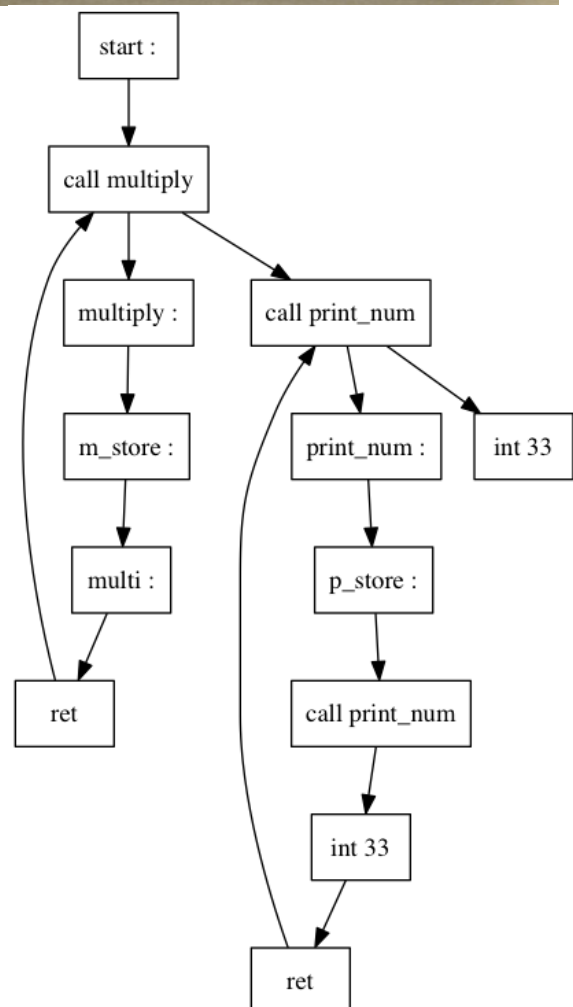
mov ax,num_f
^
rishinseitekiMacBook-Air:a5 ASTER Li$ ./bin/parser -d asgn.dot test/err2.asm
[ERROR]: test/err2.asm: 11.( 5)This label isn't defined : start_false

loop start_false
```

5. 可以画出函数调用图（其实更像是label的流程图）

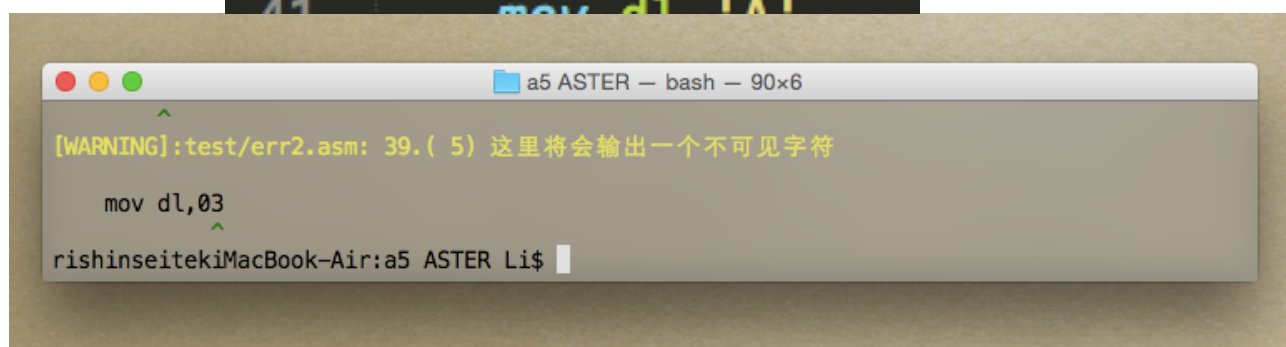
5.1 说明：

在此处，主要识别call和ret这两个函数跳转指令，还有程序中的label标号，以及int中断调用



6. 检测“程序企图输出一个不可见字符”的错误

```
36  second:
37      mov ah,02h
38      mov dl,03
39      int 21h
40      mov ah,02h
41      mov dl,1Ah
```



6.1说明

一个错误的程序输出，往往要么什么都不输出，要么输出一堆乱码停不下来。在此希望在程序运行之前尽量多检查出输出错误。

6.2算法实现

```
int IntInsNode::dumpdot(DumpDOT *dumper) {
    if(val == 33)
    {
        if(global->ax->h->getvalue() == 2)
        {
            int num = global->dx->l->getvalue();
            if((num >= 0 && num <= 31 && num != 10) || num == 127)
            {
                warning("%s:%3d.(%2d) 这里将会输出一个不可见字符\n",infile_name,loc->first_li
                show_line(global->dx->l->getlcl(),infile_name);
                print_blank(loc->first_column+8);
            }
        }
    }
}
```

在调用Int中断时，先检查Int后面的中断号是否为21h,接着检查ah的内容是否为2，如果是则为输出调用。再接着检查dl所存的将要输出的ASCII码，如果是不可见字符（除去常用的回车符），就输出报错信息

7.死循环检测

```
a5 ASTER - bash - 76x20
[WARNING]:test/err2.asm: 16.(14) There is a died circle! Please
    loop loop1
    ^
[WARNING]:test/err2.asm: 21.(12) There is a died circle! Please
    je loop2
    ^
[WARNING]:test/err2.asm: 25.(13) There is a died circle! Please
    jge loop3
    ^
[WARNING]:test/err2.asm: 28.(10) 除数不能为0 !
    div al
    ^
[WARNING]:test/err2.asm: 30.(13) There is a died circle! Please fix it!
    jle loop4
    ^
```

```
13
14 loop1:
15     mov cx,2
16     loop loop1
17 loop2:
18     push 3
19     pop ax
20     cmp ax,3
21     je loop2
22 loop3:
23     add ax,30
24     cmp ax,20
25     jge loop3
26 loop4:
27     mul ah
28     div al
29     cmp cx,10
30     jle loop4
31
```

7.1算法实现

首先，在执行运算指令的时候，就不断维护RegNode节点的三个数性值：

- lcl : last change line，上次被修改的行号
- value : 值
- trend : 执行一遍循环后它的增长趋势，可为负数

接下来，在CMP指令时，比较左右操作数的value和trend，左右比较大小使得cmp为 - 1或0或1，左边trend和右边比较得到cmp_trend为 - 1或0或1。

最后以 jl 为例来看一个死循环，其它同理

```
strs << "jl";
if(cmp == -1)
{
    if(lcl < line)
    {
        diedcircle();
    }
    else {
        if(trend <= 0)
        {
            line("cmp:%d,trend:%d\n",cmp,trend);
            diedcircle();
        }
    }
}
```

在遇到 `jl` 指令时，首先检查 `cmp` 看上一步比较是否得到 `-1`，即左操作数小于右操作数。如果是，那么语义是“条件判断为真，执行跳转”，此时，检查 `lcl` 是否小于要跳转的目的行号，如果是，说明循环执行一遍后对设置条件码所需的操作数无任何影响，是死循环。再其次，检查趋势 `trend` 是否小于 `0`，如果是，说明“执行一遍循环后，左操作数会比右操作数减小得更多”，那么再遇到 `jl` 时“条件判断为真，执行跳转”，就会进入死循环。

三.未完成的工作，希望后面能添加、补充的工作

- 1.添加对内存 `Memory` 的标号进行初始化、赋值、运算的相关代码（很简单，只需建立一个 `std::map<string*,Node*>`）
- 2.添加对无符号比较指令的支持（`ja,jae,jb,jbe`）
- 3.完善 `mul` 和 `div` 指令对于 `16` 位寄存器的计算（现在对 `8` 位寄存器的计算支持的不错）
- 4.添加对浮点数运算的支持（这个超级难）
- 5.将微软 `MASM` 工具封装进来，以实现正确源码的代码生成工作（这个涉及软件工程吗？我现在还不会）

四.参考文献

《Intel微处理器》，微机原理课本
《汇编语言（第二版）》，王爽著
Bison和Flex使用教程

2016.1.20