****

C语言课程设计报告

题 目： 简单C编译器制作——C-re

学 院： 数据科学与软件工程学院

专 业： 软件工程·服务外包

姓 名： 秦绍恒

指导教师： 兰淑丽

2018年 6 月 5 日

目 录

[1 问题背景 2](#_Toc517162884)

[2 问题分析 2](#_Toc517162885)

[3 需求分析 5](#_Toc517162886)

[3.1 用户需求分析 5](#_Toc517162887)

[3.2 功能需求分析 5](#_Toc517162888)

[3.3 系统需求分析 6](#_Toc517162892)

[4 逻辑设计 6](#_Toc517162893)

[4.1 数据结构 6](#_Toc517162894)

[4.1 函数说明 7](#_Toc517162895)

[5 详细设计 8](#_Toc517162896)

[5.1 虚拟机 8](#_Toc517162897)

[5.2 词法分析器 12](#_Toc517162898)

[5.3 语法分析器 14](#_Toc517162899)

[5.4 表达式 15](#_Toc517162900)

[6 程序调试与测试 17](#_Toc517162901)

[6.1 合法数据测试结果 17](#_Toc517162902)

[6.2 非法数据测试结果 17](#_Toc517162903)

[7 总结 18](#_Toc517162904)

[7.1 虚拟机与目标代码 18](#_Toc517162907)

[7.2 词法分析 18](#_Toc517162908)

[7.3 语法分析 18](#_Toc517162909)

[7.4 结语 19](#_Toc517162910)

[8 参考文献 20](#_Toc517162911)

[9 附录：程序源代码 20](#_Toc517162912)

# 问题背景

### 一开始学编译原理时，看的《编译原理》这本书，但全是理论和伪代码，很多东西搞不清楚，没多久就放弃了。后来在 Github 上看到了一个比较火的项目，名叫[c4](https://github.com/rswier/c4)，号称用 4 个函数来实现了一个小的 C 语言编译器。它最让我震惊的是能够自举，即能自己编译自己。并且它用很少的代码就完成了一个功能相当完善的 C 语言编译器。

### 一般的编译器相关的教程要么就十分简单（如实现四则运算），要么就是借助了自动生成的工具（如 flex/bison）。而 c4 的代码完全是手工实现的，不用外部工具。可惜的是它的代码初衷是代码最小化，所以写得很乱，很难懂。所以本项目的主要目的：实现一个功能完善的 C 语言编译器

### 首先要说明的是，虽然标题是编译器，但实际上我们构建的是 C 语言的解释器，他包含了需要我们实现自己的虚拟机与指令集，这意味着我们可以像运行脚本一样去运行 C 语言的源代码文件，而不必考虑跨平台时，由于硬件的不同造成相应的机器命令无法运行。

### c4 大致500+行。重写的代码历时一周，总共代码加注释1130行，我将它命名为C-re。

### **声明**：本项目中的代码逻辑绝大多数取自 c4 ，但确为自己重写。

# 问题分析

### 整体编译流程

### 一个简单的类C编译器，忽略开头的预处理（preprocessing）和后续的汇编（assembling）与链接（linking）动作的话，其工作流程从概念上可以如下所述：

### [ 输入的源代码 ]

### -> 词法分析（lexical analysis） -> [ 单词流（token stream） ]

### -> 语法分析（syntactic analysis / parsing） -> [ 语法树（syntax tree） ]

### -> 语义分析（semantic analysis） -> [ 标注过的语法树（annotated syntax tree） ] + [ 符号表（symbol table） ]

### -> 代码生成（code generation） -> [ 目标代码（object code） ]

### 上面每一行都是一“趟”（pass）——遍历一遍整个输入的程序。每趟在遍历时都使用上一趟生成的数据作为输入，而不必每次都在原始输入的源代码文本上遍历。

### 其中，词法分析与语法分析通常在实际实现中是混在同一趟中做的。语法分析器驱动词法分析器，从后者“拉”（pull）出单词（token）来。

### 语法分析过后，生成的树形数据结构有这么些叫法：

### 语法树（syntax tree）：或者叫“具体语法树”（concrete syntax tree）。通常意味着保留了输入源代码的大部分内容的、根据语法组织起来的一种树形结构，可能包括注释、括号、空白等内容。IDE用于支持语法高亮、代码编辑等功能的语法分析器可能会生成这样的树，因为注释之类的内容对用户编辑来说还是有用的。

### 语法分析树（parse tree）：通常跟“语法树”是同一个意思。不过这种叫法的侧重点意味着比“语法树”（syntax tree）更强调语法分析器的执行顺序对树的结构的影响。通常在教科书讲解概念时会用到，但甚少用于实际的编译器。

### 抽象语法树（abstract syntax tree，AST）：实际编译器里最常见的形式。与语法树相比，它会抛弃掉具体语法中一些对已经形成了树的形状后对语义无关紧要的语法细节，例如注释、括号、空白，以及可能会压缩掉一些不影响语义的语法层次结构。

### AST可谓是编译器用于表达程序结构的最重要的一种数据结构了。但是编译器是否一定要生成它才可以走完整个编译流程呢？答案是否定的。

### 如果你对XML处理有一定了解的话可能会听说过，XML有两大种处理方式，一种是“文档对象模型”（DOM），另一种是事件响应模型（SAX）。

### 前者其实就是把XML解析（parse）成一棵AST，并把AST叫做了DOM。整个XML文档的内容都会被转换为树形结构，便于后续代码多次遍历该文档树做各种读写操作。

### 而后者很有趣：与其把XML文档生成为一个实际的树形结构，每当它解析完一个标签的所有内容时就会发出一个事件，而用户代码可以注册事件监听器（event listener）去响应这些事件做具体的动作。这种做法让XML的树形结构隐含在了事件触发的顺序中，而不需要生成显式的树形数据结构，适合只对XML文档遍历一次时使用，无论在内存占用还是执行速度上都会比DOM方式在同样只遍历一次的场景快。

### SAX模型，用编译原理的术语说，就是一种“语法指导翻译”（syntax directed translation，SDT）。

### C-re的编译器部分的核心思想就是通过SDT把上述编译流程一股脑塞到同一趟中，实现所谓的“单趟编译器”（single-pass compiler）。所以C4的编译流程实际是：

### [ 输入的源代码 ]-> 词法分析 / 语法分析 / 语义分析（把声明记录到符号表 / 记录声明类型 / 记录表达式类型等） / 代码生成 -> [ 目标虚拟机代码 ]

### 单趟编译器由于把语义分析和代码生成的各种逻辑都混入到语法分析的过程中，代码通常都是高度浓缩而难懂的。C-re自然也不例外。

### 如果想把这种编译器的代码写得便于理解一些，可以从语法分析器提取出类似SAX模型的API，把嵌入的语义动作以回调函数的方式剥离到外面去，这样就可以让语法分析器的代码尽量保持干净，并且让某些相关的语义动作代码可以组织在一起。Clang的Parser与Sema就是这样在同一趟执行，但代码被分离开的。

### 大家可能留意到了，上面我写的C-re的编译流程，最后生成的是一种虚拟机的代码。那有没有可能让它不生成任何目标代码，而是直接边做语法分析边解释执行呢？

### 答案是肯定的。这其实就是把解释器也通过SDT整合到语法分析中。

### 早期的PHP就是这样实现的。也有比较奇葩的JavaScript引擎也这样实现，例如TinyJS / Espruino。最近大家在讨论的许式伟的qlang的自举版本也是这样实现的。

### 这种做法就比C4更加浓缩而难懂了，而且也意味着如果要重复执行某段代码（例如循环）就得每次执行都重复对那段代码做语法分析，而如果要条件执行某段代码（例如if-else）则无论是需要执行还是不需要执行的分支都得语法分析并解释过去。

### 所以比较正常的、考虑执行效率的编程语言实现是不会采用这么极端的方式的。

# 需求分析

## 用户需求分析

### 用户使用此编译器所要完成的工作主要为：编译并运行相应的以C-re语言(类C语言，因为对于C语言来说是功能不全，这个在后面会介绍到)编写的文件。通过此编译器可以方便的进行上述工作，只需要在命令行下带参数运行相应的C文件即可。每次编译文件所涉及的token信息有：头文件、全局变量、子函数调用、主函数运行、局部变量、运算符、特殊符号、关键字、内置函数、变量类型、变量值、函数参数、函数返回值、注释等。在命令行运行时，还可以通过带参数-d、-s来分别打开debug模式和assembly模式。

### 其中，debug模式可以详细地打印出每一步运算的步骤，而assembly模式是仅打印出每一步的汇编指令，但不经过虚拟机，也就是说，仅编译不运行。

## 功能需求分析

### 功能模块说明：

### 从源文件中读取源码。

### 使用词法分析器将源码解析成token。

### 使用语法分析器解析生成token。

### 在虚拟机中运行生成的汇编代码。

### 编译器流程图

### 

图3.2.2



## 系统需求分析

### 开发环境：PC 机，Windows 7（32bit） 系统

### 运行软件：GCC

# 逻辑设计

## 数据结构

### 结构特点：C-re编译器为了实现自举的目的，只支持int、char及对int或char的一级或多级指针类型，不支持typedef或结构体（struct） / 联合体（union）。但我们用一种更巧妙地方式来模拟出结构体的运用。所以使用C-re一开头就有这么一个声明：

### enum { Tk, Hash, Name, Class, Type, Val, HClass, HType, HVal, Idsz };

### 而在后面的实现中，可以经常看到这样的代码：

### id[Tk] = Id;

### id[Hash] = tk;

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

### id[Class] = Loc;

### id[Type] = INT;

### 而这 'id' 是个全局变量：

### int\* id;

### 其实 id 是总指向 sym 全局变量为开头的一个 int 数组。该数组其实就是通过sym指针开辟的一块连续的内存，通过指针加下标的形式模拟数组，而该数组就是C-re所使用的“符号表”。而这个数组的结构并不只是一个扁平的int数组，而是：

### sym -> [0] [ [0+Tk ] ] //<- sym + 0\*Idsz

### [1] [ [0+Hash ] ]

### [2] [ [0+Name ] ]

### [3] [ [0+Class ] ]

### [4] [ [0+Type ] ]

### [5] [ [0+Val ] ]

### [6] [ [0+HClass] ]

### [7] [ [0+HType ] ]

### [8] [ [0+HVal ] ]

### [9] [ [1+Tk ] ] //<- sym + 1\*Idsz

### [10] [ [1+Hash ] ]

### [11] [ [1+Name ] ]

### [12] [ [1+Class ] ]

### [13] [ [1+Type ] ]

### [14] [ [1+Val ] ]

### [15] [ [1+HClass] ]

### [16] [ [1+HType ] ]

### [17] [ [1+HVal ] ]

### [18] [ [2+Tk ] ] //<- sym + 2\*Idsz

### [..] [ ... ]

### 在原版C4代码的第74行开始的代码就体现了这种结构的用法：

### id = sym;

### while (id[Tk]) {

### if (tk == id[Hash] && !memcmp((char \*)id[Name], pp, p - pp)) { tk = id[Tk]; return; }

### id = id + Idsz;

### }

### 其实就是让id先指向sym所指向的int数组，然后每轮循环去通过 id[固定下标] 去访问这数组里的结构，如果循环还没结束的话就让id步进Idsz这么多个元素——走到下一个结构去。其实这就是模拟了一个“包含9个字段的结构体”的数组而已。

## 函数说明

### next() 用于词法分析，获取下一个标记，它将自动忽略空白字符。

### program() 语法分析的入口，分析整个 C 语言程序。

### expression(level) 用于解析一个表达式。

### eval() 虚拟机的入口，用于解释目标代码

# 详细设计

## 虚拟机

### 计算机的三个基本部件：CPU、寄存器及内存。代码（汇编指令）以二进制的形式保存在内存中，CPU 从中一条条地加载指令执行。程序运行的状态保存在寄存器中。C-re的虚拟机是一个“基于栈的虚拟机”，并且带有1个栈顶缓存（1-TOSCA），然后有可以通过下标访问的变量区。main()里的 'sp' 变量就是那个栈顶缓存。它也叫做“累加器”（accumulator），其变量名就取自此。

### 内存

### 我们从内存开始说起。现代的操作系统都不直接使用内存，而是使用虚拟内存。虚拟内存可以理解为一种映射，在我们的程序眼中，我们可以使用全部的内存地址，而操作系统需要将它映射到实际的内存上。当然，这些并不重要，重要的是一般而言，进程的内存会被分成几个段：

### 代码段（text）用于存放代码（指令）。

### 数据段（data）用于存放初始化了的数据，如int i = 10;，就需要存放到数据段中。

### 未初始化数据段（bss）用于存放未初始化的数据，如 int i[1000];，因为不关心其中的真正数值，所以单独存放可以节省空间，减少程序的体积。

### 栈（stack）用于处理函数调用相关的数据，如调用帧（calling frame）或是函数的局部变量等。

### 堆（heap）用于为程序动态分配内存。

### 它们在内存中的位置如下：

+------------------+

|    stack   |     |      high address

|    ...     v     |

|                  |

|                  |

|                  |

|                  |

|    ...     ^     |

|    heap    |     |

+------------------+

| bss  segment     |

+------------------+

| data segment     |

+------------------+

| text segment     |      low address

+------------------+

### 但我们的虚拟机并不模拟完整的计算机，我们只关心三个内容：代码段、数据段以及栈。其中的数据段我们只存放字符串，因为我们的编译器并不支持初始化变量，因此我们也不需要未初始化数据段。理论上我们的虚拟器需要维护自己的堆用于内存分配，但实际实现上较为复杂且与编译无关，故我们引入一个指令MSET，使我们能直接使用编译器（解释器）中的内存。

### 在main函数中加入初始化代码，真正为其分配内存：

if (!(text = old\_text = malloc(poolsize))) {

printf("could not malloc(%d) for text area\n", poolsize);

return -1;

}

if (!(data = malloc(poolsize))) {

printf("could not malloc(%d) for data area\n", poolsize);

return -1;

}

if (!(stack = malloc(poolsize))) {

printf("could not malloc(%d) for stack area\n", poolsize);

return -1;

}

memset(text, 0, poolsize);

memset(data, 0, poolsize);

memset(stack, 0, poolsize);

### 寄存器

### 计算机中的寄存器用于存放计算机的运行状态，真正的计算机中有许多不同种类的寄存器，但我们的虚拟机中只使用 4 个寄存器，分别如下：

### PC 程序计数器，它存放的是一个内存地址，该地址中存放着 下一条 要执行的计算机指令。

### SP 指针寄存器，永远指向当前的栈顶。注意的是由于栈是位于高地址并向低地址增长的，所以入栈时 SP 的值减小。

### BP 基址指针。也是用于指向栈的某些位置，在调用函数时会使用到它。

### AX 通用寄存器，我们的虚拟机中，它用于存放一条指令执行后的结果。

### 要理解这些寄存器的作用，需要去理解程序运行中会有哪些状态。而这些寄存器只是用于保存这些状态的。

### 全局变量中的如下语句即声明用来储存数据的寄存器：

### int \*pc, \*bp, \*sp, ax, cycle;

### 在main（）中对其进行初始化：

### memset(stack, 0, poolsize);

### bp = sp = (int \*)((int)stack + poolsize);ax = 0;

### 指令集

### 指令集是 CPU 能识别的命令的集合，也可以说是 CPU 能理解的语言。这里我们要为我们的虚拟机构建自己的指令集。它们基于 x86 的指令集，但要更为简单。

### 首先在全局变量中加入一个枚举类型，这是我们要支持的全部指令：

### // instructions

### enum { LEA ,IMM ,JMP ,CALL,JZ ,JNZ ,ENT ,ADJ ,LEV ,LI ,LC ,SI ,SC ,PUSH,

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

### // instructions

### enum { LEA ,IMM ,JMP ,CALL,JZ ,JNZ ,ENT ,ADJ ,LEV ,LI ,LC ,SI ,SC ,PUSH,

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

### 这些指令的顺序安排是有意的，带有参数的指令在前，没有参数的指令在后。这种顺序的唯一作用就是在打印调试信息时更加方便。具体的指令的实现就不在这里一一赘述，但值得一提的是，汇编中最难理解的部分——函数调用。引入的命令CALL,ENT,ADJ及LEV。

### 首先我们介绍CALL <addr>与RET指令，CALL的作用是跳转到地址为<addr>的子函数，RET则用于从子函数中返回。为什么不能直接使用 JMP 指令呢？原因是当我们从子函数中返回时，程序需要回到跳转之前的地方继续运行，这就需要事先将这个位置信息存储起来。反过来，子函数要返回时，就需要获取并恢复这个信息。因此实际中我们将 PC 保存在栈中：

### else if (op == CALL) {\*--sp = (int)(pc+1); pc = (int \*)\*pc;}

### 在实际调用函数时，不仅要考虑函数的地址，还要考虑如何传递参数和如何返回结果。这里我们约定，如果子函数有返回结果，那么就在返回时保存在 ax 中，它可以是一个值，也可以是一个地址。

### 各种编程语言关于如何调用子函数有不同的约定，例如 C 语言的调用标准是：

### 由调用者将参数入栈。

### 调用结束时，由调用者将参数出栈。

### 参数逆序入栈。

### 事先声明一下，我们的编译器参数是顺序入栈的。

### ENT <size> 指的是 enter，用于实现 ‘make new call frame’ 的功能，即保存当前的栈指针，同时在栈上保留一定的空间，用以存放局部变量。对应的汇编代码为：

; make new call frame

push    ebp

mov     ebp, esp

       sub     1, esp       ; save stack for variable: i

### ADJ <size> 用于实现 ‘remove arguments from frame’。在将调用子函数时压入栈中的数据清除，本质上是因为我们的 ADD 指令功能有限。对应的汇编代码为：

; remove arguments from frame

add     esp, 12

### 本质上这个指令并不是必需的，只是我们的指令集中并没有 POP 指令。并且三条指令写来比较麻烦且浪费空间，所以用一个指令代替。对应的汇编指令为：

; restore old call frame

       mov     esp, ebp

pop     ebp

; return

ret

### 上面的一些指令解决了调用帧的问题，但还有一个问题是如何在子函数中获得传入的参数。这里我们首先要了解的是当参数调用时，栈中的调用帧是什么样的。

sub\_function(arg1, arg2, arg3);

 +---------------+

| arg: 1        |    new\_bp + 4

+---------------+

| arg: 2        |    new\_bp + 3

+---------------+

| arg: 3        |    new\_bp + 2

+---------------+

|return address |    new\_bp + 1

+---------------+

| old BP        | &lt;- new BP

+---------------+

| local var 1   |    new\_bp - 1

+---------------+

| local var 2   |    new\_bp - 2

+---------------+

### 所以为了获取第一个参数，我们需要得到 new\_bp + 4，但就如上面的说，我们的 ADD 指令无法操作除 ax 外的寄存器，所以我们提供了一个新的指令：LEA <offset>，实现如下：

else if (op == LEA)  {ax = (int)(bp + \*pc++);}

## 词法分析器

### 由于词法分析的工作很常见，但又枯燥且容易出错，所以人们已经开发出了许多工具来生成词法分析器，如 lex, flex。这些工具允许我们通过正则表达式来识别标记。

### 这里注意的是，我们并不会一次性地将所有源码全部转换成标记流，原因有二：

### 字符串转换成标记流有时是有状态的，即与代码的上下文是有关系的。

### 保存所有的标记流没有意义且浪费空间。

### 所以实际的处理方法是提供一个函数（即前文中提到的 next()），每次调用该函数则返回下一个标记。

### 词法分析

### C-re的词法分析整个就在next()函数里。实现得很直观，没啥特别需要解释的。唯一或许值得一提的是用C/C++写的parser里一种很常见的做法。

### 看C-re的token类别的枚举类型：

### 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};

### 这个枚举类型是从128开始的。再看C4的语法分析器里的一些例子：

### token == '('

### token == Sizeof

### 有时候token跟一个字符相比较，有时候又与上面说的枚举类型的值比较，这是什么意思？其实重点就是：C4所支持的输入源码的字符集限定在7-bit ASCII上，所以每个输入的字符只可能在[0, 127]的闭区间范围内。于是如果有的“单词”（token）就是1个字符的，例如圆括号（'(' ')'）、花括号（'{' '}'）、乘号（'\*'）等，它们的ASCII码就可以直接用作表示它们的token类别；而对于多于一个字符的，例如数字字面量和关键字（"int"、"char"、"sizeof"等），或者需要区分1个或多个字符的，例如加号（'+'）或自增（"++"），则用大于ASCII码范围的数字来表示其token类别。所以就有了上面说的从128开始的枚举类型——其实从0到127，加上这Num（128）到Brak（164）的范围都可以表示token类别，只是ASCII的部分默认映射到对于的字符上而已。

### 标识符和符号表

在词法分析器中最值得一提的应该就是了符号表了，下面我们来着重分析标识符和符号表。

标识符（identifier）可以理解为变量名。对于语法分析而言，我们并不关心一个变量具体叫什么名字，而只关心这个变量名代表的唯一标识。例如 int a; 定义了变量a，而之后的语句 a = 10，我们需要知道这两个 a 指向的是同一个变量。基于这个理由，词法分析器会把扫描到的标识符全都保存到一张表中，遇到新的标识符就去查这张表，如果标识符已经存在，就返回它的唯一标识。

C-re具体使用的符号表结构就是前面提到的那个伪装成int数组的symbol\_t数组。

它里面存的symbol\_t虽然有hash值，但这个数组要叫做“哈希表”还是有点勉强——这真的就是个很简单的、每次从数组开头开始线性遍历的一个表，没有一般哈希表那种计算下标然后解决冲突的动作。这里的hash值主要是为了避免线性查找符号表时每一项都要做完整的memcmp() / strncmp()而做的一层优化。其具体结构如下：

Symbol table:

----+-----+----+----+----+-----+-----+-----+------+------+----

.. |token|hash|name|type|class|value|btype|bclass|bvalue| ..

----+-----+----+----+----+-----+-----+-----+------+------+----

|<---    one single identifier            --->|

这里解释一下具体的含义：

token：该标识符返回的标记，理论上所有的变量返回的标记都应该是 Id，但实际上由于我们还将在符号表中加入关键字如 if, while 等，它们都有对应的标记。

hash：顾名思义，就是这个标识符的哈希值，用于标识符的快速比较。

name：存放标识符本身的字符串。

class：该标识符的类别，如数字，全局变量或局部变量等。

type：标识符的类型，即如果它是个变量，变量是 int 型、char 型还是指针型。

value：存放这个标识符的值，如标识符是函数，刚存放函数的地址。

BXXXX：C 语言中标识符可以是全局的也可以是局部的，当局部标识符的名字与全局标识符相同时，用作保存全局标识符的信息。

C-re向符号表填充信息是分几步做的。

首先，词法分析在遇到标识符的时候就会向符号表插入一个新的项，此时只填入了hash、名字以及token类型（Id）等信息。前文举的那段例子就是词法分析向符号表填入新项或者获取已有项的动作。

然后在对声明做语法分析时，会把该标识符所代表的实体具体的属性信息填到符号表，例如类型是int还是char、存储类别（storage class）是全局还是局部、全局变量或局部变量的下标——这就是变量的存储空间分配，函数定义对应的字节码在代码区里面的偏移量，等等。

每当进入一个函数声明时，其中的局部变量声明会暂时“遮蔽”（hide）掉同名的全局变量声明。此时为了保留原本的全局变量声明的信息，会暂时把这些被遮蔽的声明信息从“正常位置”转移到“备份位置”，例如从 id[Class] 转移到 id[HClass]、id[Val] 转移到 id[HVal]。

同理，每当离开一个函数声明时，需要从符号表中清除该函数中的局部变量声明的信息，同时恢复出可能被遮蔽的全局变量声明的信息。这就是上一步的逆过程，遍历整个符号表中有内容的符号，把信息从“备份位置”恢复到“正常位置”。

值得留意的是，在C-re在main()的开头部分，分配了存储空间之后它就向符号表强行插入了若干符号。这些符号包括C4所支持的关键字，以及它所依赖的外部函数的名字。

强行向符号表插入依赖的外部函数的声明就绕开了要处理#include头文件的需求。注意插入外部函数的符号时 id[Val] = i++; 这句很巧妙的把外部函数名跟C-re的虚拟机字节码对应了起来，例如处理到 i 为 OPEN 字节码时，id 正好对应 "open" 这个符号的内容。这个信息在后面代码生成时会用到。

## 语法分析器

C-re的语法分析器是个典型的手写的自顶向下的实现。

它在（变量和函数）的声明部分以及语句部分采用“递归下降”方式（recursive-descent），在表达式部分采用递归下降与“运算符优先级”方式（operator precedence）混合的做法。对单目运算符、函数调用、单个变量名等的部分可以说是用了递归下降方式对双目运算符采用了运算符优先级方式

手写的语法分析器里，采用递归下降与运算符优先级结合的方式是非常常见的做法。其实两者单独使用都足以分析整个程序的语法，但是前者在分析具有多级优先级的表达式时比较慢，而后者在分析声明、语句等大家一般不看作是运算符的结构时代码比较不直观，所以把两者结合起来用就互补了。

这“运算符优先级”方式的语法分析，需要用到的核心数据结构就是——栈。

还记得我在学数据结构与算法的栈的时候就是用中缀四则混合运算表达式的求值来举例的，其中就用到了两个栈，一个用于处理运算符优先级，另一个用于存放表达式的中间计算结果。

C-re把这俩栈都用上了。只不过，用于处理运算符优先级的那个栈是通过递归的函数调用，隐含在函数调用栈里了；而用于存放表达式中间结果的栈则在C4的虚拟机部分，而不在编译器部分。

为什么编译器部分只用了“两个栈”中的一个？因为C-re的编译器生成的目标代码是C-re的虚拟机的字节码，而这个虚拟机的指令集就是个“基于（表达式）栈”的——所以说另一个栈在虚拟机部分。这个虚拟机的指令集，在表达式部分其实就是一种后缀表达式，也可以叫做逆波兰记法（Reverse Polish Notation，RPN）。

C 语言区分“语句”（statement）和“表达式”（expression）两个概念。简单地说，可以认为语句就是表达式加上末尾的分号。在我们的编译器中共识别 6 种语句：

1. if (...) <statement> [else <statement>]
2. while (...) <statement>
3. { <statement> }
4. return xxx;
5. <empty statement>;
6. expression; (expression end with semicolon)

它们的语法分析都相对容易，重要的是去理解如何将这些语句编译成汇编代码。

## 表达式

运算符的优先级决定了表达式的运算顺序，如在普通的四则运算中，乘法 \* 优先级高于加法 +，这就意味着表达式 2 + 3 \* 4 的实际运行顺序是 2 + (3 \* 4) 而不是 (2 + 3) \* 4。

传统的编程书籍会用“逆波兰式”实现四则运算来讲解优先级问题。实际上，优先级关心的就是哪个运算符先计算，哪个运算符后计算（毕竟叫做“优先级”嘛）。而这就意味着我们需要决定先为哪个运算符生成目标代码（汇编），因为汇编代码是顺序排列的，我们必须先计算优先级高的运算符。

那么如何确定运算符的优先级呢？答曰：栈（递归调用的实质也是栈的处理）。

举一个例子：2 + 3 - 4 \* 5，它的运算顺序是这样的：

将 2 入栈

遇到运算符 +，入栈，此时我们期待的是+的另一个参数

遇到数字 3，原则上我们需要立即计算 2+3的值，但我们不确定数字 3 是否属于优先级更高的运算符，所以先将它入栈。

遇到运算符 -，它的优先级和 + 相同，此时判断参数 3 属于这前的 +。将运算符 + 出栈，并将之前的 2 和 3 出栈，计算 2+3 的结果，得到 5 入栈。同时将运算符 - 入栈。

遇到数字4，同样不能确定是否能立即计算，入栈

遇到运算符 \* 优先级大于 -，入栈

遇到数字5，依旧不能确定是否立即计算，入栈

表达式结束，运算符出栈，为 \*，将参数出栈，计算 4\*5 得到结果 20 入栈。

运算符出栈，为 -，将参数出栈，计算 5-20，得到 -15 入栈。

此时运算符栈为空，因此得到结果 -15。

综上，在计算一个运算符‘x’之前，必须先查看它的右方，找出并计算所有优先级大于‘x’的运算符，之后再计算运算符‘x’。

最后注意的是优先通常只与多元运算符相关，单元运算符往往没有这个问题（因为只有一个参数）。也可以认为“优先级”的实质就是两个运算符在抢参数。

# 程序调试与测试

## 合法数据测试结果

运行： gcc main.c

a.exe hello.c

a.exe -d hello.c

a.exe -s hello.c

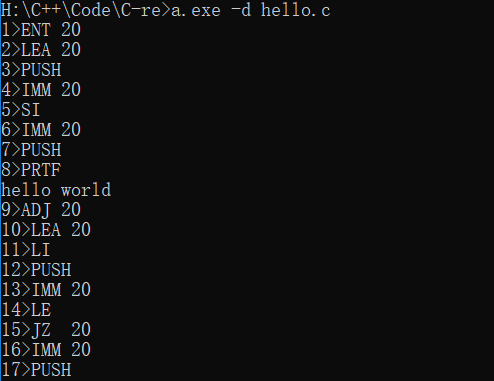
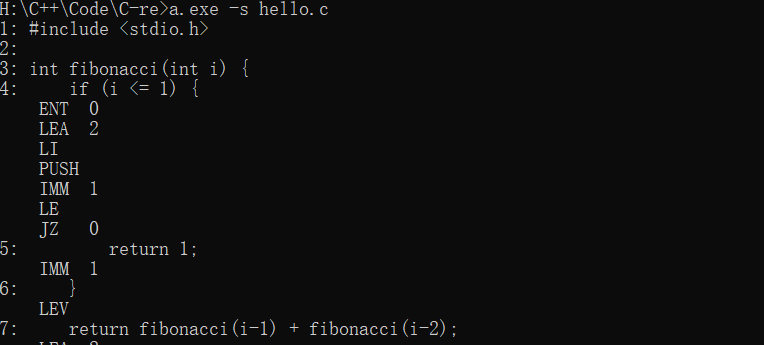
图6.1.1 图6.1.2

图6.1.3 图6.1.4

## 非法数据测试结果

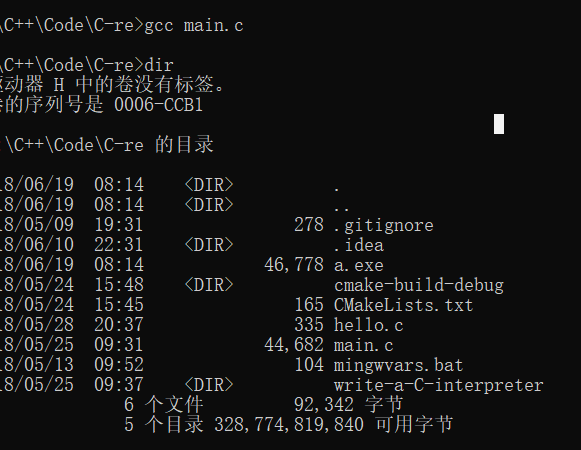
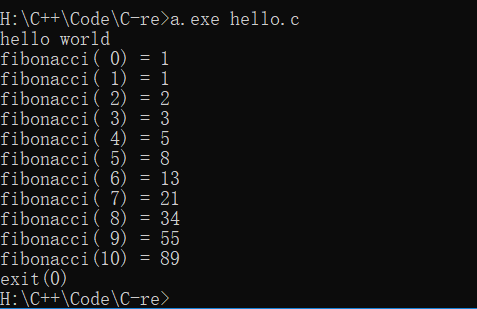
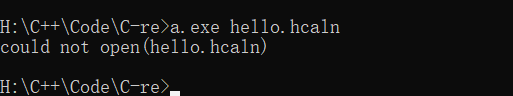
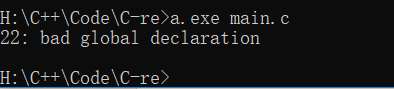
运行：a.exe hello.hcaln

图6.1.5 图6.1.6

# 总结



## 虚拟机与目标代码

整个项目的一开始，我们就着手虚拟机的实现。不知道你是否有同感，这部分对于整个编译器的编写其实是十分重要的。我认为至少占了重要程度的50%。

这里要说明这样一个观点，学习编译原理时常常着眼于词法分析和语法分析，而忽略了同样重要的代码生成。对于学习或考试而言或许可以，但实际编译项目时，最为重要的是能“跑起来”，所以我们需要给予代码生成高度的重视。

同时我们也看到，在后期解析语句和表达式时，难点已经不再是语法分析了，而是如何为运算符生成相应的汇编代码。

## 词法分析

我们用了很暴力的手段编写了我们的词法分析器，我认为这并无不可。

但你依旧可以学习相关的知识，了解自动生成词法分析器的原理，它涉及到了“正则表达式”，“状态机”等等知识。相信这部分的知识能够很大程度上提高你的编程水平。

同时，如果今后你仍然想编写编译器，不妨试试这些自动生成工具。其实我认为如果条件允许，可以尝试写一个自己的正则引擎，并让其作为自己编译器的前端，这是一个令人着迷的念头。

## 语法分析

自学习编译原理以来，语法分析对我而言一直是迷一样的存在，直到真正用递归下降的方式实现了一个。

我们用了专门讲解了“递归下降”与 BNF 文法的关系。希望能减少你对理论的厌恶。至少，实现起来并不是太难。

如果有兴趣，可以学习学习这些文法，因为已经有许多自动生成的工具支持它们。这样你就不需要重复造轮子。可以看看 yacc 等工具，更先进的版本是 bsion。同时其它语言也有许多类似的支持。

题外话，最近知道了一个叫“PEG 文法”的表示方法，无论是读起来，还是实现起来，都比 BNF 要容易，你也可以学习看看。

## 结语

许多人推荐C4给想实现一个简单的类C语言的初学者读，但不知道这些推荐它的人们有多少是真的完全读懂了C4之后才推荐的。（我就被坑的很惨）

Robert Swierczek所写的原始版C4是一个适合有经验的人阅读、把玩、鉴赏的实验作品。它是个精巧的编译器和虚拟机（解释器）。它遵循“极简主义”，只用尽量少的变量和函数，只最小限度的使用C语言的功能，并且只实现自己使用了的功能，为了能“自举”——用自己编译并运行自己。

这导致一些本来用更丰富的C语言功能可以更清晰描述的逻辑，在C4里实现得略微绕弯——知道的人一眼就能看出来它在干什么，但完全没头绪的初学者读它可能像在读密码一样。

所以，其实要完整理解C4到底是如何实现的，与其一开始就一口气钻进代码里然后碰得满头灰，还不如先读读一些入门的编译原理书，了解一下编译的过程中会经历哪些步骤，使用怎样的数据结构，然后再回到C4里去寻找那些步骤和结构。

后续有别人写的一些改进版在不同方面增强了原版C4的功能，例如Dmytro Sirenko版C4把解释器改为针对x86的JIT编译器： GitHub - EarlGray/c4: x86 JIT compiler in 86 lines

Robert Swierczek自己也有在C4的基础上做改进版，例如最近的C5添加了AST：AST + Code Generator · rswier/c4@d8e61a8 · GitHub

在“极简主义”的路上，下面这几个C编译器是从“弱”到“强”——“弱”这里指更普通的、易于理解的设计；而“强”则指把许多步骤/结构糅合在一起，虽然代码简单，但语义高度浓缩、不易于理解的设计：

LCC < TCC < C4 < OTCC

所以如果不想先读啥虎书龙书，而还是想先从一个实际编译器的资料和代码着手学习的话，先从LCC或者TCC开始会更合适一些。

LCC可以算是一个教学用的、完整的C99编译器。代码：drh/lcc: The lcc retargetable ANSI C compiler，资料：《A Retargetable C Compiler》

TCC是Fabrice Bellard大神和伙伴们写的小型C编译器。代码：TCC : Tiny C Compiler，资料：Tiny C Compiler Reference Documentation。

OTCC同样由Farice Bellard大神出品，是比C4还要极简的变态实现。官网：OTCC : Obfuscated Tiny C Compiler。是的，C4显然还不是最“极简”的 >\_<

# 参考文献

### 谭浩强 .C 程序设计（第四版）.北京:清华大学出版社，2006

### 李春葆 .数据结构教程（第二版）.北京:清华大学出版社，2007

### LotAbout 手把手教你做一个C语言编译器 <http://lotabout.me>

### RednaxelaFX 有哪些关于c4 - C in four function 编译器的文章？<https://www.zhihu.com/question/28249756/answer/84307453>

# 附录：程序源代码

# （注：因为源程序中并未支持/\*\*/形式的注释，运行前请删除该种注释）

#include <stdio.h>

#include <stdlib.h>

#include <memory.h>

#include <string.h>

#include <fcntl.h>

int debug;//调试模式

int assembly;//分部模式，列出每步对应的汇编代码

int token,//当前的标记

token\_val,//当前的标记值

line;//行号

//指向目标代码的主函数

int \*idmain;

//用于读取源代码字符串

char \*src, \*old\_src;

int poolsize;//分配内存的大小

//声明类型、表达式类型

int basetype,expr\_type;

//token标记

/\*\*

\* 用枚举类型声明所有常量，声明的常量被枚举成数字，

\* 用指针加下标的形式访问开辟出来的一段内存，模拟结构体类型

\*\*/

//标识符表，每个token的属性

enum { Token, Hash, Name, Type, Class, Value, BType, BClass, BValue, IdSize};

/\*\*

\* token：该标识符返回的标记，理论上所有的变量返回的标记都应该是 Id，但实际上由于我们还将在符号表中加入关键字如 if, while 等，它们都有对应的标记。

\* hash：顾名思义，就是这个标识符的哈希值，用于标识符的快速比较。

\* name：存放标识符本身的字符串。

\* class：该标识符的类别，如数字，全局变量或局部变量等。

\* type：标识符的类型，即如果它是个变量，变量是 int 型、char 型还是指针型。

\* value：存放这个标识符的值，如标识符是函数，刚存放函数的地址。

\* BXXXX：C 语言中标识符可以是全局的也可以是局部的，当局部标识符的名字与全局标识符相同时，用作保存全局标识符的信息。

\* IdSize: 因为枚举本身就是从1开始，所以IdSize就正好是一个token的大小

\* \*/

//标记表，即可识别的token

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 };

/\*\*

\* C-re所支持的输入源码的字符集限定在7-bit ASCII上所以输入的字符只可能在[0,127]的闭区间范围内，

\* 所以单字符的token可以直接用ASCII码表示其token类别，而对于多字符token(如关键字、内置函数)则用

\* 大于ASCII码的数字来表示其token类别，所以以上标记表从128开始

\* \*/

//变量和函数的类型

enum { CHAR, INT, PTR };

//作用域类型

enum { Global, Local };

//当前的标记id，指向标识符表

int \*current\_id, \*symbols;

//栈指针的索引，用于函数调用

int index\_of\_bp;

//虚拟机

/\*\*虚拟机三个基本部件：CPU、寄存器及内存。

\* 代码（汇编指令）以二进制的形式保存在内存中，

\* CPU 从中一条条地加载指令执行。

\* 程序运行的状态保存在寄存器中。

\*/

//虚拟机的内存部分

int \*text, \*old\_text, \*stack;

char \*data;

/\*\*

\* 代码段（text）用于存放代码（指令）。

\* 数据段（data）用于存放初始化了的数据，如int i = 10;，就需要存放到数据段中。

\* 栈（stack）用于处理函数调用相关的数据，如调用帧（calling frame）或是函数的局部变量等。

\* \*/

//虚拟机的寄存器部分

int \*pc, \*bp, \*sp, ax, cycle;

/\*\*

\* PC 程序计数器，它存放的是一个内存地址，该地址中存放着 下一条 要执行的计算机指令。

\* SP 指针寄存器，永远指向当前的栈顶。注意的是由于栈是位于高地址并向低地址增长的，所以入栈时 SP 的值减小。

\* BP 基址指针。也是用于指向栈的某些位置，在调用函数时会使用到它。

\* AX 通用寄存器，我们的虚拟机中，它用于存放一条指令执行后的结果。

\* \*/

//虚拟机中的指令集

enum { LEA, IMM, JMP, CALL, JZ, JNZ, ENT, ADJ, LEV, LI, LC, SI, SC, PUSH,

OR, XOR, AND, EQ, NE, LT, GT, LE, GE, SHL, SHR, ADD, SUB, MUL, DIV, MOD,

OPEN, READ, CLOS, PRTF, MALC, MSET, MCMP, EXIT };

//词法分析

void next(){

char \*last\_pos;

int hash;

while (token=\*src){//此时的token并不是真正的token，而是直接读取的源码

//跳过那些不能识别的字符，同时也用来处理空白字符，

// 也就是说，将那些未识别的字符当成空白字符跳过

++src;

if (token=='\n'){

if (assembly){//分部模式

printf("%d: %.\*s", line, src-old\_src, old\_src);

old\_src = src;

while (old\_text < text) {//在google上没搜到，自己测试了一下，

// 其实就是定义一个字符串，因为C的特性，会把分开的字符串拼接，

// 然后用&定义一个位于首地址的指针，并用下标偏移量来截取字符个数为4的字符串

printf("%8.4s", & "LEA ,IMM ,JMP ,CALL,JZ ,JNZ ,ENT ,ADJ ,LEV ,LI ,LC ,SI ,SC ,PUSH,"

"OR ,XOR ,AND ,EQ ,NE ,LT ,GT ,LE ,GE ,SHL ,SHR ,ADD ,SUB ,MUL ,DIV ,MOD ,"

"OPEN,READ,CLOS,PRTF,MALC,MSET,MCMP,EXIT"[\*++old\_text \* 5]);

if (\*old\_text <= ADJ)

printf(" %d\n", \*++old\_text);

else

printf("\n");

}

}

++line;

} else if (token=='#'){//跳过头文件导入，将内置函数直接插入symbol表

while (\*src!=0 && \*src!='\n'){

src++;

}

} else if ((token>='a' && token<='z')||(token>='A' && token<='Z')||(token=='\_')){//字符串表示

last\_pos=src-1;//因为之前src已经++过，所以这里的last\_pos就是当前的token

hash=token;

//向前看一个字符，若符合要求，则判定与之前的为同一字符串，并计算哈希值

while ((\*src>='a' && \*src<='z')||(\*src>='A' && \*src<='Z')||(\*src>='0' && \*src<='9')||(\*src=='\_')){

hash=hash\*147+\*src;//一个线性相关的哈希求值

src++;

}

//线性查询已存在的标识，使current\_id从symbol table的开头开始遍历

current\_id=symbols;

while (current\_id[Token]){//token存在即不为零

if (current\_id[Hash]==hash && !memcmp((char\*)current\_id[Name],last\_pos,src-last\_pos)){

//若找到则直接返回

token=current\_id[Token];//真实token

return;

}

current\_id=current\_id+IdSize;//查询下一个单位

}

//若没有找到现有的token，则插入表中

current\_id[Name]=(int)last\_pos;

current\_id[Hash]=hash;

token=current\_id[Token]=Id;//真实token

return;

} else if (token>='0' && token<='9'){//数字表示

token\_val=token-'0';//首位字符串转数字

if (token\_val>0){//首位大于零，十进制表示

while (\*src>='0' && \*src<='9'){

token\_val=token\_val\*10 + \*src++ -'0';

}

} else{

if (\*src=='x'||\*src=='X'){//0x...||0X... 十六进制

token=\*++src;

while ((token>='0' && token<='9')||(token>='a' && token<='f')||(token>='A' && token<='F')){

token\_val=token\_val\*16+(token&15)+(token>='A'?9:0);//十六进制计算法

token=\*++src;

}

} else{//0... 八进制

while (\*src>='0' && \*src<='7'){

token\_val=token\_val\*8+\*src++ -'0';

}

}

}

token=Num;

return;

} else if (token=='/'){//注释或除号

if (\*src=='/'){

while (\*src!=0 && \*src!='\n'){

src++;

}

} else{

token=Div;

return;

}

} else if (token=='"'||token=='\''){//解析""和''字符串文字，仅支持'\n'转义字符串,字符串存到数据段data中

last\_pos=data;//指向data

while (\*src!=0 && \*src!=token){//无换行

token\_val=\*src++;//若为单字符，此时存值

if (token\_val=='\\'){//字符串中出现'\'时

token\_val=\*src++;

if (token\_val=='n'){

token\_val='\n';

}

}

if (token=='"'){

\*data++ =token\_val;

}

}

src++;//继续向下进行

if (token=='"'){

token\_val=(int)last\_pos;

} else{

token=Num;//若为单字符则返回Num

}

return;

} else if (token=='='){

if (\*src=='='){//判断是否相等

src++;

token=Eq;

} else{//赋值

token=Assign;

}

return;

} else if (token=='+'){

if (\*src=='+'){//自增

src++;

token=Inc;

} else{

token=Add;

}

return;

} else if (token=='-'){

if (\*src=='-'){//自减

src++;

token=Dec;

} else{

token=Sub;

}

return;

} else if (token=='!'){

if (\*src=='='){// !=

src++;

token=Ne;

}//token='!'

return;

} else if (token=='<'){

if (\*src=='='){//<=

src++;

token=Le;

} else if (\*src=='<'){//<<

src++;

token=Shl;

} else{//<

token=Lt;

}

return;

} else if (token=='>'){

if (\*src=='='){//>=

src++;

token=Ge;

} else if (\*src=='>'){//>>

src++;

token=Shr;

} else{//>

token=Gt;

}

return;

} else if (token=='|'){

if (\*src=='|'){//||

src++;

token=Lor;

} else{//|

token=Or;

}

return;

} else if (token=='&'){

if (\*src=='&'){//&&

src++;

token=Lan;

} else{//&

token=And;

}

return;

} else if (token=='^'){

token=Xor;

return;

} else if (token=='%'){

token=Mod;

return;

} else if (token=='\*'){

token=Mul;

return;

} else if (token=='['){

token=Brak;

return;

} else if (token=='?'){

token==Cond;

return;

} else if (token=='~'||token==';'||token=='{'||token=='}'||token=='('||token==')'

||token==']'||token==','||token==':'){

return;//token就是本身

}

}

}

//匹配当前token，移向下一个token

void match(int tk){

if (token==tk){

next();

} else{

printf("%d: expected token: %d\n",line,tk);

exit(-1);

}

}

//解析表达式

void expression(int level){

/\*\*

\* 表达式主要有两部分，单元和运算符

\* 表达式有以下类型

\* 1. unit\_unary ::= unit | unit unary\_op | unary\_op unit

\* 2. expr ::= unit\_unary (bin\_op unit\_unary ...)

\* \*/

//unit\_unary()

int \*id;

int tmp;

int \*addr;

{

if (!token){

printf("%d: unexpexted token EOF of expression\n",line);

exit(-1);

}

if (token==Num){

//加载数字常量

match(Num);

//记录代码

\*++text=IMM;

\*++text=token\_val;

expr\_type=INT;

} else if (token == '"') {

//加载字符串常量,记录代码

\*++text = IMM;

\*++text = token\_val;

match('"');

//储存剩余的字符串

while (token == '"') {

match('"');

}

//默认在字符串的末尾添加'\n'，对于0，就仅仅移一个相对位置即可

data = (char \*)(((int)data + sizeof(int)) & (-sizeof(int)));

expr\_type = PTR;

} else if (token==Sizeof){

//sizeof是一元运算符，需要知道运算类型，且仅支持int、char、指针

match(Sizeof);

match('(');

expr\_type=INT;//此处expr\_type临时表示参数的类型

if (token==Int){

match(Int);

} else if (token==Char){

match(Char);

expr\_type=CHAR;

}

while (token==Mul){

match(Mul);

expr\_type=expr\_type+PTR;

}

match(')');

//记录代码

\*++text=IMM;

\*++text=(expr\_type==CHAR)? sizeof(char): sizeof(int);

expr\_type=INT;

} else if (token==Id){

//变量与函数调用都以Id开始

match(Id);

id=current\_id;

if (token=='('){

//函数调用

match('(');

tmp=0;//参数数量

while (token!=')'){

//参数顺序入栈

expression(Assign);

\*++text=PUSH;

tmp++;

if (token==','){

match(',');

}

}

match(')');

//内置函数直接调用汇编指令，

// 而普通的函数则编译成 CALL <addr> 的形式。

if (id[Class]==Sys){

//内置函数

\*++text=id[Value];

} else if (id[Class]==Fun){

//自定义函数

\*++text=CALL;

\*++text=id[Value];

} else{

printf("%d: bad function call\n",line);

exit(-1);

}

//清除入栈的参数。因为我们不在乎出栈的值，所以直接修改栈指针的大小即可。

if (tmp>0){

\*++text=ADJ;

\*++text=tmp;

}

expr\_type=id[Type];

} else if (id[Class]==Num){

//当该标识符是全局定义的枚举类型时，直接将对应的值用 IMM 指令存入 AX 即可。

\*++text=IMM;

\*++text=id[Value];

expr\_type=INT;

} else {

//加载变量的值，如果是局部变量则采用与 bp 指针相对位置的形式

// 而如果是全局变量则用 IMM 加载变量的地址。

if (id[Class]==Loc){

//局部变量

\*++text=LEA;

\*++text=index\_of\_bp-id[Value];

} else if (id[Class==Glo]){

//全局变量

\*++text=IMM;

\*++text=id[Value];

} else{

printf("%d: undefined variable\n",line);

exit(-1);

}

//无论是全局还是局部变量，最终都根据它们的类型用 LC 或 LI 指令加载对应的值

expr\_type=id[Type];

\*++text=(expr\_type==Char)?LC:LI;

}

} else if (token=='('){

//撇或括号

match('(');

if (token==Int||token==Char){

//强制转换

tmp=(token==Char)?CHAR:INT;

match(token);

while (token==Mul){

match(Mul);

tmp=tmp+PTR;

}

match(')');

expression(Inc);

expr\_type=tmp;

} else{

//正常括号

expression(Assign);

match(')');

}

} else if (token==Mul){

//指针取值

match(Mul);

expression(Inc);

if (expr\_type>=PTR){

expr\_type=expr\_type-PTR;

} else{

printf("%d: bad dereference\n",line);

exit(-1);

}

\*++text=(expr\_type==CHAR)?LC:LI;

} else if (token==And){

//取值操作

//对于变量先加载它的地址，并根据它们类型使用 LC/LI 指令加载实际内容

//这里只要不执行LC/LI操作即可

match(And);

expression(Inc);//获取地址

if (\*text==LC||\*text==LI){

text--;

} else{

printf("%d: bad address of\n",line);

exit(-1);

}

expr\_type=expr\_type+PTR;

} else if (token=='!'){

//逻辑取反

match('!');

expression(Inc);

\*++text=PUSH;

\*++text=IMM;

\*++text=0;

\*++text=EQ;

expr\_type=INT;

} else if (token=='~'){

//按位取反

match('~');

expression(Inc);

\*++text=PUSH;

\*++text=IMM;

\*++text=-1;

\*++text=XOR;

expr\_type=INT;

} else if (token==Add){

//四则运算中的加减法，不是单个数字的取正取负操作

match(Add);

expression(Inc);

expr\_type=INT;

} else if (token==Sub){

match(Sub);

if (token==Num){

\*++text=IMM;

\*++text=-token\_val;

match(Num);

} else{

\*++text=IMM;

\*++text=-1;

\*++text=PUSH;

expression(Inc);

\*++text=MUL;

}

expr\_type=INT;

} else if (token==Inc||token==Dec){

//自增自减

tmp=token;

match(token);

expression(Inc);

if (\*text==LC){

\*text=PUSH;//因为需要用到两次地址，所以先PUSH一下

\*++text=LC;

} else if (\*text==LI){

\*text=PUSH;

\*++text=LI;

} else{

printf("%d: bad lvalue of pre-increment\n",line);

exit(-1);

}

\*++text=PUSH;

\*++text=IMM;

\*++text=(expr\_type>PTR)? sizeof(int): sizeof(char);//考虑指针的情况

\*++text=(tmp==Inc)?ADD:SUB;

\*++text=(expr\_type==CHAR)?SC:SI;

} else{

printf("%d: bad expression\n",line);

exit(-1);

}

}

//二元运算符和三元运算符

/\*\*

\* 运用了递归下降和运算符优先级混合的方法

\* 这里运用了两个栈，来实现优先级的运算，一个用于储存运算符优先级，一个用于储存中间的计算结果

\* 处理运算符的栈通过函数的递归调用隐含在函数调用栈里，而储存中间计算结果的的栈则位于虚拟机中

\* 因为编译器生成的目标代码是虚拟机中的字节码，而虚拟机中的指令集本就是基于栈的，

\* 所以虚拟机中的指令集在表达式部分其实就是一种后缀表达式

\* \*/

{

while (token>=level){

//在调用expression时，通过参数level定义了当前的优先级

// 只有当前运算符优先级大于level时才会继续递归调用下去

tmp=expr\_type;

if (token==Assign){

//赋值运算是当前优先级最低的运算符

//当解析完=右边的表达式后，相应的值会存放在 ax 中，用PUSH实际将这个值保存起来

match(Assign);

if (\*text==LC||\*text==LI){

\*text=PUSH;//保存运算后的右值

} else{

printf("%d: bad lvalue in assignment\n",line);

exit(-1);

}

expression(Assign);

expr\_type=tmp;

\*++text=(expr\_type==CHAR)?SC:SI;

} else if (token==Cond){

//expr?a:b 唯一的三目运算符，相当于一个小型的if语句

match(Cond);

\*++text=JZ;

addr=++text;

expression(Assign);

if (token==':'){

match(':');

} else{

printf("%d: missing colon in conditional\n",line);

exit(-1);

}

\*addr=(int)(text+3);

\*++text=JMP;

addr=++text;

expression(Cond);

\*addr=(int)(text+1);

}

/\*\*

\* <expr1> || <expr2> <expr1> && <expr2>

\* ...<expr1>... ...<expr1>...

\* JNZ b JZ b

\* ...<expr2>... ...<expr2>...

\* b: b:

\* \*/

else if (token==Lor){

//逻辑或

match(Lor);

\*++text=JNZ;

addr=++text;

expression(Lan);

\*addr=(int)(text+1);

expr\_type=INT;

} else if (token==Lan){

//逻辑与

match(Lan);

\*++text=JZ;

addr=++text;

expression(Or);

\*addr=(int)(text+1);

expr\_type=INT;

}

/\*\*

\* 数学运算符，包括 |, ^, &, ==, != <=, >=, <, >, <<, >>, +, -, \*, /, %

\* \*/

else if (token==Or){

//||

match(Or);

\*++text=PUSH;

expression(Xor);

\*++text=OR;

expr\_type=INT;

} else if (token==Xor){

//^

match(Xor);

\*++text=PUSH;

expression(And);

\*++text=XOR;

expr\_type=INT;

} else if (token==And){

//&&

match(And);

\*++text=PUSH;

expression(Eq);

\*++text=AND;

expr\_type=INT;

} else if (token==Eq){

//==

match(Eq);

\*++text=PUSH;

expression(Ne);

\*++text=EQ;

expr\_type=INT;

} else if (token==Ne){

// !=

match(Ne);

\*++text=PUSH;

expression(Lt);

\*++text=NE;

expr\_type=INT;

} else if (token==Lt){

//<

match(Lt);

\*++text=PUSH;

expression(Shl);

\*++text=LT;

expr\_type=INT;

} else if (token==Gt){

//>

match(Gt);

\*++text=PUSH;

expression(Shl);

\*++text=GT;

expr\_type=INT;

} else if (token==Le){

//<=

match(Le);

\*++text=PUSH;

expression(Shl);

\*++text=LE;

expr\_type=INT;

} else if (token==Ge){

//>=

match(Ge);

\*++text=PUSH;

expression(Shl);

\*++text=GE;

expr\_type=INT;

} else if (token==Shl){

//<<

match(Shl);

\*++text=PUSH;

expression(Add);

\*++text=SHL;

expr\_type=INT;

} else if (token==Shr){

//>>

match(Shr);

\*++text=PUSH;

expression(Add);

\*++text=SHR;

expr\_type=INT;

} else if (token==Add){

//+，注意这里的+操作还有可能是对于指针的

match(Add);

\*++text=PUSH;

expression(Mul);

expr\_type=tmp;

if(expr\_type>PTR){

//指针类型，不包括char\*

\*++text=PUSH;

\*++text=IMM;

\*++text= sizeof(int);

\*++text=MUL;

}

\*++text=ADD;

} else if (token==Sub){

//-,同样有指针操作

match(Sub);

\*++text=PUSH;

expression(Mul);

if (tmp>PTR && tmp==expr\_type){

//指针减法

\*++text=SUB;

\*++text=PUSH;

\*++text=IMM;

\*++text= sizeof(int);

\*++text=DIV;

expr\_type=INT;

} else if (tmp>PTR){

//指针

\*++text=PUSH;

\*++text=IMM;

\*++text= sizeof(int);

\*++text=MUL;

\*++text=SUB;

expr\_type=tmp;

} else{

//普通减法

\*++text=SUB;

expr\_type=tmp;

}

} else if (token==Mul){

//\*

match(Mul);

\*++text=PUSH;

expression(Inc);

\*++text=MUL;

expr\_type=tmp;

} else if(token==Div){

// /

match(Div);

\*++text=PUSH;

expression(Inc);

\*++text=DIV;

expr\_type=tmp;

} else if (token==Mod){

//%

match(Mod);

\*++text=PUSH;

expression(Inc);

\*++text=MOD;

expr\_type=tmp;

} else if (token==Inc||token==Dec){

//后缀形式的自增自减

/\*\*

\* 与前缀形式不同的是，在执行自增自减后， ax上需要保留原来的值。

\* 所以我们首先执行类似前缀自增自减的操作，再将 ax 中的值执行减/增的操作。

\* \*/

if (\*text==LI){

\*text=PUSH;

\*++text=LI;

} else if(\*text==LC){

\*++text=PUSH;

\*++text=LC;

} else{

printf("%d: bad value in increment\n",line);

exit(-1);

}

\*++text=PUSH;

\*++text=IMM;

\*++text=(expr\_type>PTR)? sizeof(int): sizeof(char);

\*++text=(token==Inc)?ADD:SUB;

\*++text=(expr\_type==CHAR)?SC:SI;

\*++text=PUSH;

\*++text=IMM;

\*++text=(expr\_type>PTR)? sizeof(int): sizeof(char);

\*++text=(token==Inc)?SUB:ADD;

match(token);

} else if (token==Brak){

//数组取值操作，a[10]=\*(a+10)

match(Brak);

\*++text=PUSH;

expression(Assign);

match(']');

if (tmp>PTR){

//不含char的指针类型

\*++text=PUSH;

\*++text=IMM;

\*++text= sizeof(int);

\*++text=MUL;

} else if (tmp<PTR){

printf("%d: pointer type expected\n",line);

exit(-1);

}

expr\_type=tmp-PTR;

\*++text=ADD;

\*++text=(expr\_type==CHAR)?LC:LI;

} else{

printf("%d: compiler error, token=%d\n",line,token);

exit(-1);

}

}

}

}

//解析语句块

void statement(){

/\*\*

\* 在我们的编译器中共识别 6 种语句：

\* if (...) <statement> [else <statement>]

\* while (...) <statement>

\* { <statement> }

\* return xxx;

\* <empty statement>;

\* expression; (expression end with semicolon)

\* \*/

int \*a, \*b;//用于控制分支

if (token==If){

/\*\*

\* if (...) <statement> [else <statement>]

\* if (<cond>) <cond>

\* JZ a

\* <true\_statement> ===> <true\_statement>

\* else: JMP b

\* a: a:

\* <false\_statement> <false\_statement>

\* b: b:

\* \*/

match(If);

match('(');

expression(Assign);

match(')');

\*++text=JZ;//跳转

b=++text;//指向当前的text的指针

statement();//解析{<statement>}

if (token==Else){

match(Else);

\*b=(int)(text+3);

\*++text=JMP;

b=++text;

statement();

}

\*b=(int)(text+1);

} else if (token==While){

/\*\*

a: a:

while (<cond>) <cond>

JZ b

<statement> <statement>

JMP a

b: b:

\*/

match(While);

a=text+1;

match('(');

expression(Assign);

match(')');

\*++text=JZ;

b=++text;

statement();

\*++text=JMP;

\*++text=(int)a;

\*b=(int)(text+1);

} else if (token=='{'){

// { <statement> ... }

match('{');

while (token!='}'){

statement();

}

match('}');

} else if (token==Return){

// return [expression];

match(Return);

if (token!=';'){

expression(Assign);

}

match(';');

\*++text=LEV;//跳出函数体

} else if (token==';'){

//空语句

match(';');

} else{

//a=b;或 function\_call

expression(Assign);

match(';');

}

}

//枚举类型声明语句,主要的逻辑用于解析用逗号（,）分隔的变量，将该变量的类别设置为Num，使之成为全局变量

void enum\_declaration(){

int i=0;

while (token!='}'){

if (token!=Id){

printf("%d: bad enum identifier %d\n",line,token);

exit(-1);

}

next();

if (token==Assign){//如果是赋值语句

next();

if (token!=Num){

printf("%d: bad enum initializer\n",line);

exit(-1);

}

i=token\_val;//若有赋值操作，则将i赋予新值

next();

}

current\_id[Class]=Num;

current\_id[Type]=INT;

current\_id[Value]=i++;//无赋值则++

if (token==','){

next();

}

}

}

//解析函数参数

void function\_parameter(){

int type,params=0;//params为参数个数

//匹配类型

while (token!=')'){

type=INT;//默认为int

if (token==Int){

match(Int);

} else if (token==Char){

type=CHAR;

match(Char);

}

//匹配指针

while (token==Mul){

match(Mul);

type+=PTR;

}

//匹配参数名

if (token!=Id){

printf("%d: bad parameter declaration\n",line);

exit(-1);

}

if (current\_id[Class]==Loc){//若此局部变量已被声明

printf("%d: duplicate parameter declaration\n",line);

exit(-1);

}

match(Id);

//储存形参局部变量，将重名的全局变量备份到Bxxx系列，当出函数体时，再还原

current\_id[BClass]=current\_id[Class];current\_id[Class]=Loc;

current\_id[BType]=current\_id[Type];current\_id[Type]=type;

current\_id[BValue]=current\_id[Value];current\_id[Value]=params++;//作为当前参数的索引

if (token==','){

match(',');

}

}

index\_of\_bp=params+1;

}

//解析函数体，函数体内必须遵循先声明在调用原则，即先声明所有变量，再进行调用

void function\_body(){

int pos\_local,type;

pos\_local=index\_of\_bp;

//解析局部变量声明

while (token==Int||token==Char){

basetype=(token==Int)?INT:CHAR;

match(token);

while (token!=';'){

type=basetype;

while (token==Mul){

match(Mul);

type=type+PTR;

}

if (token!=Id){

printf("%d: bad local declaration\n",line);

exit(-1);

}

if (current\_id[Class]==Loc){

printf("%d: duplicate local declaration\n", line);

exit(-1);

}

match(Id);

//储存局部变量

current\_id[BClass] = current\_id[Class]; current\_id[Class] = Loc;

current\_id[BType] = current\_id[Type]; current\_id[Type] = type;

current\_id[BValue] = current\_id[Value]; current\_id[Value] = ++pos\_local;

if (token==','){

match(',');

}

}

match(';');

}//局部变量声明结束

//保存局部变量在栈中的位置，并预留一些空间

\*++text=ENT;

\*++text=pos\_local-index\_of\_bp;

//解析语句块

while (token!='}'){

statement();

}

//释放内存

\*++text=LEV;

}

//函数声明语句，形式为：type func\_name(...){...}

void function\_declaration(){

match('(');

function\_parameter();

match(')');

match('{');

function\_body();

//match('}'); 末尾的'}'用于解析函数体时判断是否结束

//遍历symbol表，恢复全局变量

current\_id=symbols;

while (current\_id[Token]){

if (current\_id[Class]==Loc){

current\_id[Class]=current\_id[BClass];

current\_id[Type]=current\_id[BType];

current\_id[Value]=current\_id[BValue];

}

current\_id=current\_id+IdSize;

}

}

//全局的定义语句，包括变量定义，类型定义（只支持枚举）及函数定义

void global\_declaration(){

int type, i;//临时变量，type用于保存当前类型

basetype=INT;//默认为int类型

//enum枚举类型语法

if (token==Enum){

match(Enum);

// enum [id] { a = 10, b = 20, ... }

if (token!='{'){

match(Id);

}//判断其是否有变量名

if (token=='{'){

match('{');

enum\_declaration();

match('}');

}

match(';');

return;

}

//分析类型

if (token==Int){

match(Int);

} else if (token==Char){

match(Char);

basetype=CHAR;//修改默认类型

}

//分析";"分隔的语句

while (token!=';' && token!='}'){

type=basetype;

while (token==Mul){//指针类型

match(Mul);

type=type+PTR;

}

if (token!=Id){

printf("%d: bad global declaration\n",line);

exit(-1);

}

if (current\_id[Class]){//若标识符已经存在

printf("%d: duplicate global declaration\n",line);

exit(-1);

}

match(Id);

current\_id[Type]=type;

if (token=='('){

current\_id[Class]=Fun;

current\_id[Value]=(int)(text+1);//记录函数的地址

function\_declaration();//解析函数声明

} else{

current\_id[Class]=Glo;

current\_id[Value]=(int)data;

data=data+sizeof(int);

}

if (token==','){

match(',');

}

}

next();

}

//语法分析

void program(){

//获取下一个token

next();

while (token>0){

global\_declaration();

}

}

//虚拟机

int eval(){

int op, \*tmp;

cycle=0;

while (1){

cycle++;

op=\*pc++;//获取下一个操作指令

//打印debug信息

if (debug){

printf("%d>%.4s",cycle,

& "LEA ,IMM ,JMP ,CALL,JZ ,JNZ ,ENT ,ADJ ,LEV ,LI ,LC ,SI ,SC ,PUSH,"

"OR ,XOR ,AND ,EQ ,NE ,LT ,GT ,LE ,GE ,SHL ,SHR ,ADD ,SUB ,MUL ,DIV ,MOD ,"

"OPEN,READ,CLOS,PRTF,MALC,MSET,MCMP,EXIT"[op \* 5]);

if (op<=ADJ)

printf("%d\n",line);

else

printf("\n");

}

if (op == IMM) {ax = \*pc++;}//将当前的pc中值的地址存到寄存器ax中

else if (op == LC) {ax = \*(char \*)ax;}//将对应地址中的字符载入ax

else if (op == LI) {ax = \*(int \*)ax;}//将对应地址中的整数载入ax

else if (op == SC) {ax = \*(char \*)\*sp++ = ax;}//sp中存放的地址，\*sp取出其中的地址，转换成char类型，再次进行取址，作用同下

else if (op == SI) {\*(int \*)\*sp++ = ax;}//将 ax 中的数据作为整数存放入地址中，要求栈顶存放地址

else if (op == PUSH) {\*--sp = ax;}//将ax的值入栈

else if (op == JMP) {pc = (int \*)\*pc;}//跳转到pc中存放的地址

else if (op == JZ) {pc = ax ? pc + 1 : (int \*)\*pc;}//若为1，则进行下一条指令，若为0，则跳转到对应地址

else if (op == JNZ) {pc = ax ? (int \*)\*pc : pc + 1;}//为1跳转，为0继续

else if (op == CALL) {\*--sp = (int)(pc+1); pc = (int \*)\*pc;}//跳转到函数所在地址

//else if (op == RET) {pc = (int \*)\*sp++;}//跳出函数，这里被LEV取代

else if (op == ENT) {\*--sp = (int)bp; bp = sp; sp = sp - \*pc++;}//保存当前的栈指针，同时在栈上保留一定的空间，用以存放局部变量

else if (op == ADJ) {sp = sp + \*pc++;}//将调用子函数时压入栈中的数据清除

else if (op == LEV) {sp = bp; bp = (int \*)\*sp++; pc = (int \*)\*sp++;}//退出函数

else if (op == LEA) {ax = (int)(bp + \*pc++);}//调用函数参数

//运算符指令

/\*\*

\* 每个运算符都是二元的，即有两个参数，第一个参数放在栈顶，第二个参数放在 ax 中。

\* 这个顺序要特别注意。因为像 -，/之类的运算符是与参数顺序有关的。

\* 计算后会将栈顶的参数退栈，结果存放在寄存器 ax中。

\* 因此计算结束后，两个参数都无法取得了

\* \*/

else if (op == OR) ax = \*sp++ | ax;

else if (op == XOR) ax = \*sp++ ^ ax;

else if (op == AND) ax = \*sp++ & ax;

else if (op == EQ) ax = \*sp++ == ax;

else if (op == NE) ax = \*sp++ != ax;

else if (op == LT) ax = \*sp++ < ax;

else if (op == LE) ax = \*sp++ <= ax;

else if (op == GT) ax = \*sp++ > ax;

else if (op == GE) ax = \*sp++ >= ax;

else if (op == SHL) ax = \*sp++ << ax;

else if (op == SHR) ax = \*sp++ >> ax;

else if (op == ADD) ax = \*sp++ + ax;

else if (op == SUB) ax = \*sp++ - ax;

else if (op == MUL) ax = \*sp++ \* ax;

else if (op == DIV) ax = \*sp++ / ax;

else if (op == MOD) ax = \*sp++ % ax;

//内置函数

else if (op == EXIT) { printf("exit(%d)", \*sp); return \*sp;}

else if (op == OPEN) { ax = open((char \*)sp[1], sp[0]); }

else if (op == CLOS) { ax = close(\*sp);}

else if (op == READ) { ax = read(sp[2], (char \*)sp[1], \*sp); }

else if (op == PRTF) { tmp = sp + pc[1]; ax = printf((char \*)tmp[-1], tmp[-2], tmp[-3], tmp[-4], tmp[-5], tmp[-6]); }

else if (op == MALC) { ax = (int)malloc(\*sp);}

else if (op == MSET) { ax = (int)memset((char \*)sp[2], sp[1], \*sp);}

else if (op == MCMP) { ax = memcmp((char \*)sp[2], (char \*)sp[1], \*sp);}

else {

printf("unknown instruction:%d\n", op);

return -1;

}

}

}

int main(int argc, char \*\*argv){

//因为主程序也包含在命令行的参数中，所以减少参数个数，跳过主程序

argc--;argv++;

int i, fd;//fd用于接受打开文件的地址

int \*tmp;//i和\*tmp都是临时变量

//在命令行中加参数-s，开启分部模式,\*\*argv等价于(\*argv)[0],将第一个字符串再次解析

if (argc > 0 && \*\*argv == '-' && (\*argv)[1] == 's') {

assembly = 1;

--argc;

++argv;

}

//在命令行中加参数-d，开启调试模式

if (argc > 0 && \*\*argv == '-' && (\*argv)[1] == 'd') {

debug = 1;

--argc;

++argv;

}

//未输入要运行的文件

if (argc < 1) {

printf("usage: xc [-s] [-d] file ...\n");

return -1;

}

poolsize=256\*1024;//定义内存大小

line=1;//行号为1

//分配内存

if (!(text = malloc(poolsize))) {

printf("could not malloc(%d) for text area\n", poolsize);

return -1;

}

if (!(data = malloc(poolsize))) {

printf("could not malloc(%d) for data area\n", poolsize);

return -1;

}

if (!(stack = malloc(poolsize))) {

printf("could not malloc(%d) for stack area\n", poolsize);

return -1;

}

if (!(symbols=malloc(poolsize))){

printf("could not malloc(%d) for symbol table\n",poolsize);

return -1;

}

//初始化分配的内存

memset(text, 0, poolsize);

memset(data, 0, poolsize);

memset(stack, 0, poolsize);

memset(symbols, 0, poolsize);

old\_text=text;

//向symbol table中插入内置的关键字和函数,这里的顺序与表中的顺序应一致

//用字符串常量的第一个字符的地址赋值给指针变量src

src = "char else enum if int return sizeof while "

"open read close printf malloc memset memcmp exit void main";

i=Char;

while (i<=While){

next();//此时返回的current\_id[Token]=Id，所以下面修改为i

current\_id[Token]=i++;

}

i=OPEN;

while (i<=EXIT){

next();//和上面一样，但作为函数，还需要设置Class、Type、Value

current\_id[Class]=Sys;//区别自定义函数的系统内置函数

current\_id[Type]=INT;

current\_id[Value]=i++;

}

next();current\_id[Token]=Char;//此时Token为void，这里将void设为char

next();idmain=current\_id;//此时指针current\_id指向标识符表的main，跟踪主函数

//以只读方式打开文件，分配src内存并读取源码

if ((fd=open(\*argv,0))<0){

printf("could not open(%s)\n",\*argv);

return -1;

}

if (!(src=old\_src=malloc(poolsize))){

printf("could not malloc(%d) for source area\n",poolsize);

return -1;

}

if ((i=read(fd,src,poolsize-1))<=0){

printf("read() returned %d\n",i);

return -1;

}

src[i]=0;//添加EOF结束符

close(fd);

//printf("hello Jessica!\n");

//语法分析器驱动词法分析器，从后者“拉”（pull）出单词（token）来

program();

/\*\*

\* 词法分析在遇到标识符的时候就会向符号表插入一个新的项，

\* 此时只填入了hash、名字以及token类型（Id）等信息。

\* 然后在对声明做语法分析时，会把该标识符所代表的实体具体的属性信息填到符号表，

\* 例如类型是int还是char、存储类别（storage class）是全局还是局部、

\* 全局变量或局部变量的下标——这就是变量的存储空间分配，函数定义对应的字节码在代码区里面的偏移量，等等

\* \*/

//printf("hello krystal!\n");

//从主函数开始加载，使pc指向主函数

if (!(pc=(int\*)idmain[Value])){

printf("main() not defined\n");

return -1;

}

//分部模式到此结束,只进行编译,不进入虚拟机实际运行汇编程序

if (assembly){

return 0;

}

//设置栈指针

sp=(int\*)((int)stack+poolsize);//从最后开始往前扩大

\*--sp=EXIT ;//当主函数return后，调用exit

\*--sp=PUSH;tmp=sp;

\*--sp=argc;\*--sp=(int)argv;

\*--sp=(int)tmp;//将命令行参数入栈

return eval();

}