simpleC_complier

参考 c4 完成的 C 语言编译器,代码逻辑大部分跟 c4 相同,但是确实是自己理解了并重写,附有大量中文注释。

一个最简单的编译器经过的大致步骤即:词法分析-->语法分析-->语义分析-->代码生成

本项目并没有代码生成,而是直接产生运行结果,所以这其实是一个解释器。解释器和编译器的区别就在于,解释器直接将运行结果回送,而编译器会成目标代码供你运行。

代码中设计了自己的指令集和虚拟机来运行得出结果。

词法分析

词法分析将源代码生成为记号流(会过滤掉空格等不必要的信息),以供下一阶段的语法分析使用,而每个记号由记号本身和记号的值组成,比如一个变量 a,对应的记号可能为(ID,a)

在本项目中,词法分析对应 `next()`函数,它并不是将源代码一次性处理成记号流,这样做的好处是可以根据源代码的上下文来灵活分析。手工构造的词法分析远比书上介绍的 Thompson、子集构造等算法实现的自动扫描程序简单很多。

- 主要的思想就是通过预处理关键字并通过前看符号来分析源代码并将满足条件的标识符存入符号表中。
- 主逻辑:

```
while(token = *src_pos++)
{
   if(...)
```

```
    {
     ...
    }
    else if(...)
    {
     ...
     return; //匹配成功了才 return
    }
}
```

- 处理变量部分采用 hash 的方式,将合法的变量名存入符号表(判断变量是否存在的方法即通过比较 hash 值以及字符)
- 处理数字部分唯一需要注意的就是十六进制,c4 的源码部分为 `ival = ival * 16 + (tk & 15) + (tk >= 'A' ? 9 : 0);`,这样写虽然简洁,但是可能难懂。
- 处理字符部分需要区分字符串和单字符,如果是字符串,则需要记录起始位置,并存入 data 段中;单字符则直接将其类型保存为 Num
- 其他部分不需要太多说明,具体可看注释

词法分析、语义分析

其实本项目中都并未像编译原理课程中那样,严格区分每个阶段。比如词法分析生成记号流给语法分析,语法分析(继续填符号表)生成抽象语法树给语义分析,语义分析生成中间代码,代码生成利用中间代码生成目标代码。而是在词法分析的过程中进行语法分析,并且也涉及到了指令的生成。不过总的来说,逻辑也算清晰。

对应于 `grammar()、statement()、expression()`函数。
逻辑如下：

- 首先通过 next()函数生成的记号来区分声明的变量/函数的类型(int、char、enum)
- 接着开始解析变量/函数的声明,并将其填入符号表中
- 接着处理函数体中的局部变量声明部分(遵循 c89 标准,所以变量必须在函数体的开始处)
- 然后处理语句部分,语句里面有表达式,继续处理表达式部分
- 最后进行将函数栈里面的局部变量等销毁(这里就已经不是词法分析的任务了)

需要额外注意的有：

1. enum 有两种赋值方式,一种是默认从 0 递增,另一种是自己赋值
2. 指令的生成部分可能有点难以理解,需要对应各指令的含义
3. 局部变量保存信息时,需要将对应的全局变量的信息保存在符号表的 Temxxx 中,最后销毁函数栈时,再恢复回来

代码运行

这部分对应 run()函数,其实就是将生成的指令根据含义操作栈还有寄存器等

部分指令及寄存器

- 寄存器:

SP:栈顶指针

BP:栈底指针

PC:程序计数器。和我们在计组/汇编中学到的类似,存放着要执行的下一条指令的地址

AX:规定用来存放一条指令执行后的结果

- 指令:

LEA <offset>:用于获取函数调用中传入的参数

IMM <num>:将 num 存入 ax 中

JMP <addr>:无条件跳转指令,将当前 pc 置为 addr

CALL <addr>:跳转到 addr 处,调用函数

JZ:判断 ax 是否为 0,如果为 0,则跳转

JNZ:判断 ax 是否为 0,如果不为 0,则跳转

ENT <size>:保存当前栈底指针并为函数调用栈安排空间

ADJ:销毁函数调用栈中的数据

LEV:做销毁函数栈的扫尾工作,恢复栈底指针等,对应汇编中 ret 指令

LI <addr>:将地址中的整数放入 ax 中

LC <addr>:将地址中的字符放入 ax 中

SI:将 ax 的值存入栈顶中(要求是地址),针对整型

SC:将 ax 的值存入栈顶中(要求是地址),针对字符

PUSH:将 ax 的值压入栈中