**编译器的代码架构**

原创2022-07-25 19:54·[底层技术栈](https://www.toutiao.com/c/user/token/MS4wLjABAAAAFE8hfWyLu6WnP5zye8AcpgDHMfufkFDkvnuhZG0p-ktFWcB9u1P3yDso3c3fZvlN/?source=tuwen_detail)

**编译器**，是把**高级语言**转化为**机器语言**的工具软件。

高级语言的代码也是个**文本字符串**，所以**编译器的前端**与sed、gawk、grep是差不多的，都是广义上的**字符串匹配**。

编译器把代码转化为**机器码**的**过程**如下：

**1，词法分析**，

这是编译器里**最简单的模块**。

**词法分析**，就是通过**查看下一个字符**来确定怎么把**代码字符串**分割成一个个的**语法词汇**。

起始符、终止符，是词法分析的主要概念。

int day = 24 \* 3600;

这行代码的**第1个词是int**，起始符是i，终止符是**空格**。

在词法分析时它是一个**标志符**，也就是**字母、下划线、数字**组成的一个字符串，必须以字母或下划线开头。

在分析出int这个标志符之后，它后面的**空格**就没用了，直接丢弃它。

**第2个词day**也是一个标志符，起始符是d，终止符也是空格。

习惯把代码写得密集的人可能这么写：int day=24\*3600;

这时day的终止符是**=**，它同时还是下一个词的起始符，在把day加入词汇序列之后需要从=开始接着分析。

第3个词是=，第4个词是24，第5个词是\*，第6个词是3600，第7个词是分号;

在词法分析时要把**数字字符串24和3600**转化为**整数24和3600**，这两个在程序里是不同的。

10进制、16进制、8进制、2进制、浮点数的支持，都是词法分析时的任务。

另外，**转义字符串**也要在这里支持。

'\0' 在源代码里是**字符串文本**，包含着**4个字符**' \ 0 '，要转义成单个字符0。

'\r' '\n' '\t'的处理和'\0'一样。

词法分析还是很好写的。

**2，语法分析**，

这是编译器前端**最难写**的模块，它需要把**源代码**转化成**一棵**描述整个程序结构的**多叉树**。

这个多叉树叫做**抽象语法树**（英文缩写**AST**）。

类型、变量、运算符、函数定义、函数调用、if语句、for/while循环，都是这个这棵树的一部分。

抽象语法树的层次结构，与源代码的结构是一样的。

如果是这样的源代码的话：

int sum = 0;

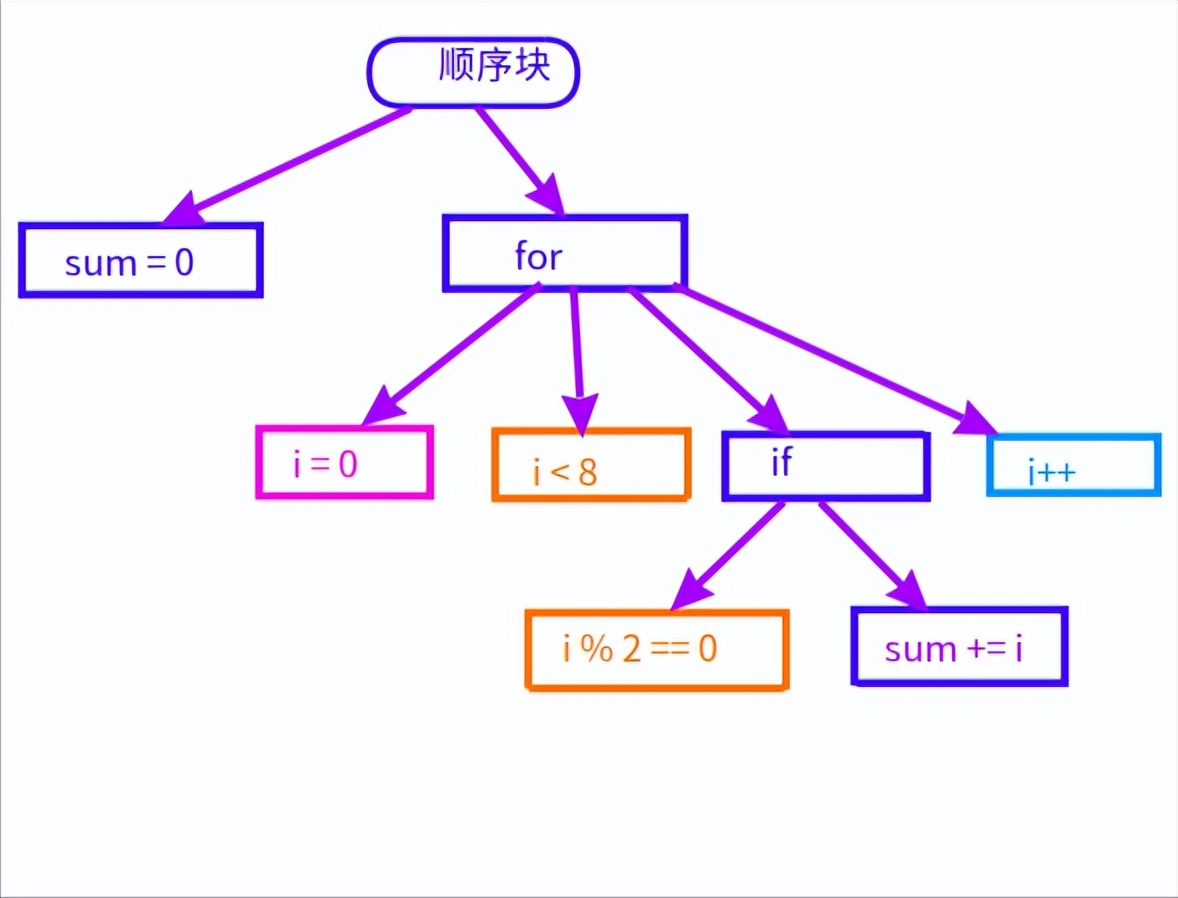
for (int i = 0; i < 8; i++) {

if (i % 2 == 0)

sum += i;

}

那么语法树是这样的：



语法树

初始化语句sum = 0与后续的for循环是顺序执行的，它们属于同一个顺序块，在语法树上有**同一个父节点**。

for循环有**4个子节点**：**初始化**表达式i = 0，**条件**i < 8，**循环体**if语句、**更新**表达式i++。

其中循环体又是个if语句，具有2个子节点：条件表达式i % 2 == 0，主体sum += i。

**while循环**的结构与for类似，只要去掉初始化表达式和更新表达式就行，只有2个节点。

把词法分析之后的**词汇序列**转化成**抽象语法树**时，常用的方法是**有限自动机**。

也可以把代码直接写成**递归函数调用**，但是后续**改起语法来**就比较麻烦。

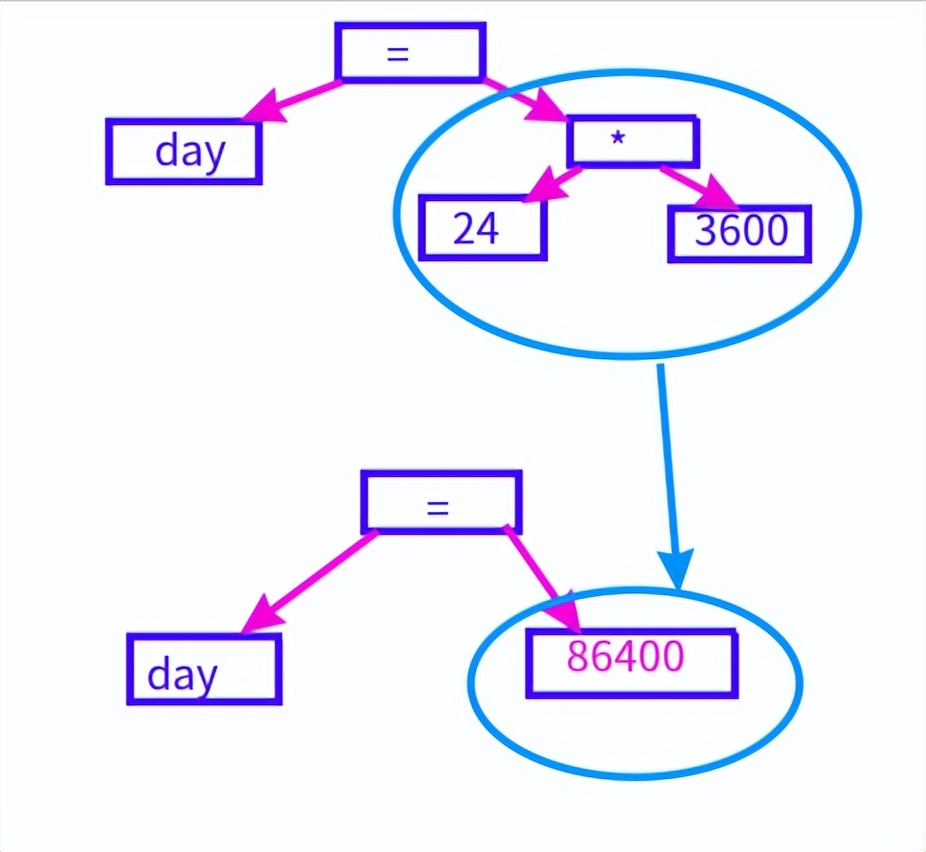
我一开始就是把scf的parse模块写成了递归函数调用，后来为了可以编辑语法，又自己做了个简单的有限自动机。

**3，语义分析**，

把语法树遍历一遍，检查一下**类型是否匹配**，就是语义分析。

如果要支持**面向对象**的话，就可以在这里进行**函数重载**和**运算符重载**。

**常量表达式**也要在这里计算出来，int day = 24 \*3600要转化成day = 86400。



常量表达式的计算

对语法树进行遍历时，不同的**语法节点**使用的**处理函数**是**不同**的，这就是**语义**。

符号=要当作赋值，符号+要当作加法，其他类似。

C语言里常见的函数调用，语法树是这样的：

int printf(const char\* fmt, ...);

int main()

{

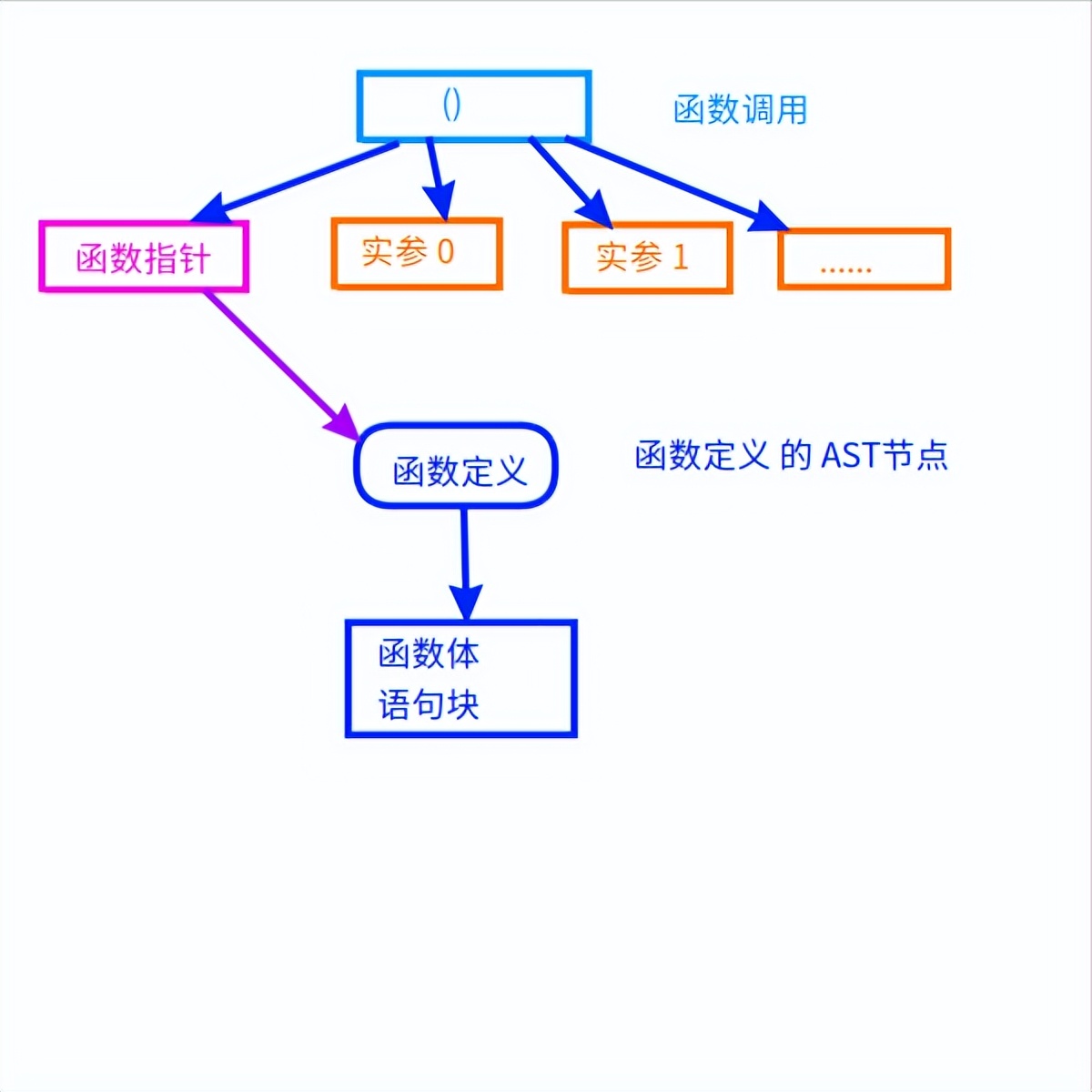
printf("hello world\n");

}

函数调用也是一个运算符，具有一个单独的语法节点，它的子节点都是它的参数：

其中**函数名**也是一个参数，要转化为对应的**函数体的**节点的**指针**。

通过这个指针才可以找到函数的代码，进行**内联优化（inline）**。



函数调用和定义

如果要是调用的**外部函数**，只有声明没有实现，那就没法内联了。

**4，中间代码生成（三地址码）**，从这里开始就是**编译器的后端**了。

这一步也是对语法树进行遍历，把对应的表达式、函数、if语句、for循环都变成类似汇编的三地址码。

上面那段for循环，这时会被变成如下的三地址码序列：

assign sum, 0

assign i, 0

**start: // for循环的开头**

cmp i, 8

**jge end //条件不成立，则结束循环**

assign **t**, i % 2 // t是编译器生成的临时变量

cmp t, 0

jne next

add sum, sum, i // 这行才是三地址码

**next: // 下一轮for循环**

inc i // 循环变量+1

**jmp start //跳转回开头，继续循环**

**end：// for循环结束**

到了这里，那个复杂的**树型结构**已经变成**线形结构**了，可以按顺序写到一个文本文件里，这就是**汇编代码**。

**到这里**，编译器就可以生成**类似gcc -S**的汇编代码了。

**5，中间代码优化**，

这是编译器后端的主要部分，属于**机器无关优化**，这部分的优化是不依赖于**CPU平台**的。

scf框架的这部分包含以下功能：

1）内联函数，

2）有向无环图DAG的生成，

3）带二级指针参数的函数调用分析，

4）指针别名分析，也就是分析指针指向的变量，

5）活跃变量分析，

6）变量的加载保存分析，

7）需要自动内存管理的变量分析，

8）代码流程图的**深度优先排序**，

9）自动内存管理代码的添加，

10）基本块内的优化，

11）循环分析，

会把一些变量**尽量**在循环的**入口加载**，在循环**出口保存**，减少循环内的内存读写。

没有**常量传播**的优化，哪天有空我把它添上

**6，寄存器分配**，

使用图的着色算法，之前的文章写过。

**7，指令选择**，

直接写在代码里的，没做龙书里提到的那个树的覆盖。

**8，机器码生成**，

根据intel x64的手册编写机器码就行。

**9，目标文件生成**，

也就是gcc -c 得到那个.o文件。

Linux上的elf文件是什么样的就怎么写，可以参考**linux的man手册**里对**elf**的讲解。

**10，可执行文件的生成**，

这是**连接器**的功能，它把多个.o .a .so文件连接成一个可执行文件。

这一步的代码在scf/elf目录，有兴趣的可以看看。

连接之后的文件就可以在**shell命令行**里运行了。