

编译原理实验报告

英才学院

1240009 班

刘秋志

学号：7120310422

一. 实验题目

设计一种类似 Pascal 语法的编程语言。设计并实现该语言的编译器。

二. 实验目的

完成词法分析，语法分析，语义分析与中间代码生成和汇编语言生成等模块并组装成编译程序，实现对设计的高级语言的编译工作。

二. 实验内容和要求

1.词法设计

词法设计中，单词分为四种类别：标识符，关键字，符号和常数。把空格作为单词之间的分隔符。

标识符以一个字母作为开始，后面零或多个数字或字母。标识符可能是变量，过程等。具体的信息要根据语法分析的结果判断。

关键字是预先设定的保留单词，用来表达特殊的语法意义，不能被用作标识符。其词法规则和标识符完全相同。所以可以在识别出标识符后检查该标识符是否是某个关键字。

在实际的编程语言中，有一些关键字并没有在语法规则中被实际地使用，但是仍然不可以作为标识符使用。这类关键字称作保留字。

符号是指加号减号等算数运算符和与或非等逻辑运算符。

运算符有可能是单目的，也有可能是双目的。同时，运算符可以由特殊符号构成，也可以由字母构成。在处理上，可以将运算符自己归为一类，也可以将运算符归入到关键字一类。比如在 python 中，用 and 来表示逻辑与。很难区分 and 究竟是运算符还是关键字。或者说 and 既是一个双目运算符，也是一个关键字。

常数分为整数常数和浮点常数两种。整数常数由若干个数字组成，浮点常数由用“.”分隔开的两串数字字符串组成。

由数字开始，出现非“.”外的非数字字符或者出现第二个“.”会被视为错误字符，分析器报错。出现非 ASCII 字符也会报错。

注释有#表明。会忽略一行中#后面的内容（和 python 类似）。

2.语法设计：

语法设计基本参照标准 Pascal，加入了一些分隔符来表明程序区块的开始和完结。

具体语法见下表

非终结符	终结符	产生式
E	input	PP->P;
P	output	P->prog id (input , output) D seg S;
T	,	D-> beginD D0 D1 endD;
F	+	D0->List : Type proc id D seg S;
Type	-	D1->segD D0 D1 ;
B	*	List -> List , id id;
B1	/	Type-> integer real array C of Type ^
B2	id	Type record D ;
B3	=	C-> [num] C ;

D	(S0->Left := E;
S)	E -> E + T E - T - T T;
List	prog	T-> F F * F F / F;
C	seg	F-> (E) id ;
EList	integer	Left->id;
PList	real	
D0	bool	B->B1 or B1 B1;
D1	array	B1-> B2 and B2 B2;
S0	record	B2-> not B3 B3;
S1	[B3-> true false;
Left]	B3-> E relop E;
List	^	
	num	
	return	
	call	S->S0 S1;
	while	S0 -> beginif if B then S endif;
	if	S0 -> beginif if B then S else S endif;
	else	S0 -> beginwhile while B do S endwhile;
	then	
	beginif	S1 -> segS S0 S1 ;
	endif	
	segD	S0 -> call id (EList);
	segS	S0 -> return E;
	true	EList -> EList , E E;
	false	
	of	S0-> read (PList);
	:=	S0->write (EList);
	or	PList ->PList , id;
	and	
	not	
	do	
	read	
	write	
	proc	
	:	
	beginwhile	
	endwhile	
	beginD	
	endD	
	relop	

3.语义翻译设计:

基本上采用教材中的语法制导翻译模式。在实现过程中利用的 Python 的一些基本

特性，使得编写编译器的工作量有所减少。

具体的翻译过程不再此处赘述。

4. 目标代码生成：

由于 X86 汇编本身功能复杂，通用寄存器数量少而专用寄存器数量多，不利于进行对汇编语言的自定义。所以将源程序编译成 MIPS 汇编。同时不考虑操作系统对程序地址的链接。默认运行环境是一台可编程的 MIPS 机器。

约定在所有程序中，利用 R0 保存 00H 作为计算常数使用。将程序的返回地址保存在 R31，将程序的布尔判定结果保存在 R30 用于条件分支指令。

三. 总体设计

1. 编译过程设计

将整个编译过程分为两遍，第一遍进行词法分析，输出由种别码和属性值组成的二元组。第二遍读入词法分析器产生的单词流，进行自底向上的语法分析和语法制导的语义分析。

在第二遍中，直接输出四元式和相应的汇编代码。

在编译的过程中，可以动态地看到符号表管理器的状态。

2. 软件系统设计

软件分为三个部分。

一个是词法分析器，读入 3 个文件，分别是关键字表，OP 表和源程序代码。

词法分析器的输出是一个文件，内容为源代码生成的单词流。单词流由种别码属性值二元组构成。

第二个部分是语法分析器。语法分析器读入 2 个文件，分别是词法分析器生成的单词流文件和 Action 表 GOTO 表构成的驱动表文件。

语法分析器输出一个文件，是汇编代码。

语法分析器也可以输出四元式序列和符号表。这两个备选功能可以为以后的代码优化提供拓展空间。

第三个部分是驱动表生成器。由于采用编译工作台进行 LALR (1) 分析表的生成，得到的是一个 HTML 格式的驱动表。需要完成手动 parse 并生成语法分析器可以接收的格式的文件。

由于采用了上述的结构，本编译器在词法和语法上和具体的语言无关。如果利用本编译器对一种新的语言进行编译，只需要读入新的词法文件并利用编译工作台和驱动表生成器生成新的驱动表文件。

当然语义翻译的部分是和语言相关的。编译新的语言时要重写翻译部分。如果想进一步提高该编译器的通用性，可以定义一种新的语义文件来对应不同语法分析时的语义动作。这样只需再生成一个翻译文件，即可对新的语言进行编译工作。

由于本实验的主要关注点是实现一个编译器而不是编译器自动生成工具，所以并没有实现这个设计。

四. 详细设计与实现

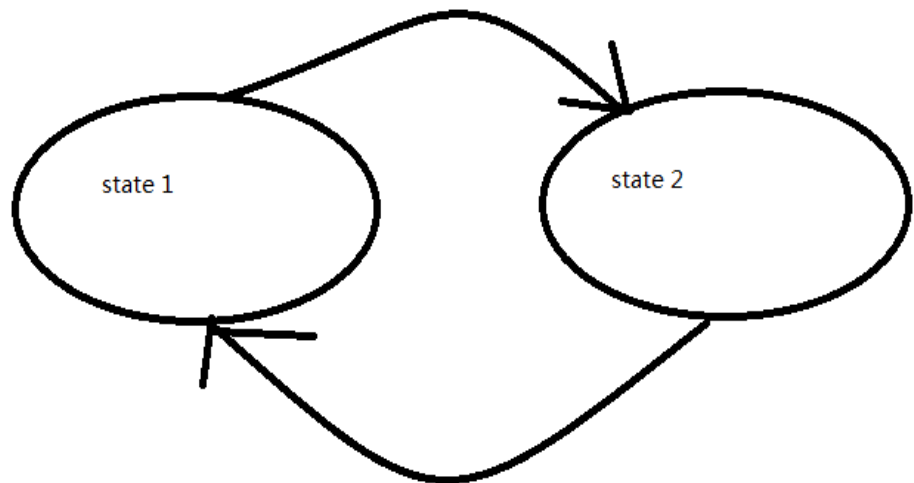
1. 词法分析器

词法分析器的输入是从源代码文件读出来的字符流，输出是由种别码和属性值组成的 token 流。

为了进行词法分析，词法分析器需要一张保留词表和一张运算符表。

在进行词法分析之前，先对读入的字符流进行预处理。预处理的内容包括去除多余的空格和消除被注释的内容。

去掉多余的空格十分简单，在读入一个空格符时，忽略该空格后的所有符号，直到遇到下一个非空格字符为止。这个过程可以被看做一个有两个状态的 DFA。



state 1: 遇到下一个字符，读入字符并形成输入流，继续读入字符

state 2: 遇到下一个字符，忽略字符，继续读入字符

转移函数：

state1->state2：遇到的字符是空格。读入空格并转移到state2

state2->state1：遇到的字符不是空格。读入非空格字符并转移到state1

在处理注释时，可以采用相似的办法。一旦遇到#，则忽略#后面的所有字符，直到遇到一个换行符为止。

这个 DFA 和前面去掉空格所使用的 DFA 很相似。在这里不再赘述。

在词法分析的过程中，也是使用有穷状态自动机来进行单词的识别。

自动机在编程实践中常常被用到。构成自动机的要素是状态集合和状态之间的转移函数。

在实现时，有两种构造自动机的思路。

一种是直接对于所有关键字，常数数字和符号建立自动机。这种自动机可以直接识别出所有的关键字。问题在于，如果关键字数目众多，这种自动机序号很多的状态。同时，一旦关键字发生改变，那么需要重新构造识别自动机。

另一种方法是先忽略所有的关键字，将遇到的单词识别为 OP，标识符和常数三类。如果识别到的单词是标识符，则在由关键字构成的字典中查找。

如果可以找到匹配的关键词，则将这个标识符识别为关键字。否则就识别为普通标识符。

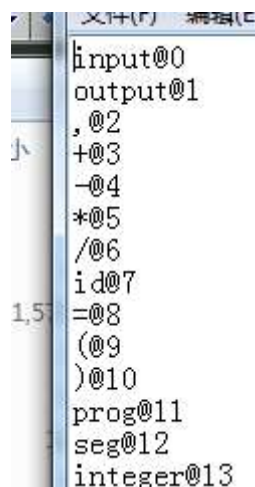
考虑到前文提到的本编译器可以迅速适应不同的关键词集合，第二种实在通用性上显然更胜一筹。同时，由于第一种方法会产生大量的状态，在编码效率上十分底下。和一些快速的字典查找算法（比如红黑树，trie 树）相比，在效率上并没有太大的优势。所以这里选择方法二。

在实现上，采用 python 语言中的 dictionary 来作为关键字字典。Python 中的字典采用散列表实现，可以达到 $O(1)$ 的时间复杂度。和 C++ STL 中的红黑树相比更有优势。几乎不会增加识别单词时的效率。

```
5 class wordList:#This is a word list
6     def __init__(self,a,b):
7         self.dict={}
8         self.reservedDict={}
9         self.OperatorDict={}
10        for i in a:
11            arr=i.split("@")
12            tmp=arr[1].split("\n")
13            self.reservedDict[arr[0]]=int(tmp[0])
14        for i in b:
15            i = i.split("@")
16
17            tmp = i[1].split("\n")[0]
18            self.OperatorDict[i[0]]=int(tmp)
```

图中的 wordlist 维护了两个字典。一个用来保存关键字等待查找，另外一个用来保存合法的运算符集合，来在词法分析阶段发现不合法的运算符。

这两个字典都是从文件中读入的。文件格式为：关键字或运算符@种别码
一个字典文件的例子如图所示：



```
input@0
output@1
,@2
+@3
-@4
*@5
/@6
id@7
=@8
(@9
)@10
prog@11
seg@12
integer@13
```

```

1  class myStack:#This is a stack
2      def __init__(self):
3          self.counter=0
4          self.stack=[]
5          self.state=0
6
7      def push(self,a,b):
8          self.stack.append(a)
9          self.counter+=1
10         self.state=b
11
12     def push(self,a):
13         self.stack.append(a)
14         self.counter+=1
15
16     def getWord(self,c):
17         s=''
18         for i in self.stack:
19             s+=i
20
21         stateValue=c.getValue(self.state)
22         return (s,stateValue)
23
24     def clear(self):
25         self.state=0
26         self.coutner=0
27         self.stack=[]

```

上图中是我实现的一个栈，用来存储没有完成识别的字符串。这个栈的一个扩展功能是可以方便地得到栈中的全部字符。

2. 语法分析器

语法分析器的输入有多个。

首先需要读入有词法分析器在上一遍中生成的单词流。其次，要读入驱动表和帮助驱动表工作的语法变量表。

语法分析器的输出是汇编代码，可选的输出由四元式和符号表。

驱动表的生成在后面的驱动表生成器中进行介绍。

这里采用自底向上的 LALR(1)分析法。

```

class mystack:
    def __init__(self):
        self.l = []
        self.flag=0
    def push(self,a):
        if(self.flag<len(self.l)):
            self.l[self.flag] = a
            self.flag+=1
        else:
            self.l.append(a)
            self.flag+=1
    def pop(self):
        if(self.flag>0):
            self.flag-=1
            return self.l[self.flag]
        else:
            return False
    def top(self):
        return self.l[self.flag-1]
    def show(self):
        s=''
        for i in range(0,self.flag):
            s+=str(self.l[i])
            s+=", "
        print s,"flag :",self.flag
    def is_empty(self):
        if(self.flag == 0):

```

这里我实现了一个通用栈，可以用作语法分析栈和语义栈。和普通栈相比只多了一个 `show` 函数，可以用来查看规约过程中栈内的语法状态或者语义参数。

由于 `python` 是一种动态语言，在这个用 `list` 实现的栈中，我们可以压入任何内容。在语法分析的过程中，我们可以压入整数作为状态的表示。在语义分析中，我们可以压入数字，列表等等数据结构。

```
code = []
```

由于 `python` 良好的语言特性，我们可以用一个 `code_list` 来存储生成的代码。和 `C++`等语言的字符串数组相比，由于待生成的代码长度未知，我们不需要自己管理字符串数组的大小。

同时，通过 `len()`函数，我们还可以在任意时刻得到当前已经生成的代码

的行数而不需要手动记录代码行数。这一点在 patch 代码时节省了一定的工作量。

```
src = []
read_source(objFile,src)
read_table(tableFile)
sta = mystack()
yuyi = mystack()
wl = wordList(wordListFile)
mysigtalbe = sig_table()

dealWithObj(src,action,sta,wl,yuyi,mysigtalbe)

print mysigtalbe.dict
for i in code:
    print i
```

在分析开始时，我们读入所有的单词流。语法分析需要源文件流（src），action 表和 GOTO 表（被合并成 action），语义栈和语法栈（yuyi 和 sta）以及符号表 mysigtalbe。

```
▼ class sig_table:
▼     def __init__(self):
        self.dict = {}
        self.tablelist = {}
        self.pc = 0
```

符号表由两个 python 字典组成。这里利用 python 良好的动态性质，字典中也是可以存入任意的类型。Dict 中存储当前嵌套中声明的所有变量名对应的地址。Self.tablelist 中存储的是外层嵌套的符号表。Pc 指明当前符号表所使用的内存位置。

在进行语法分析时，只需要执行标准的移近归约算法。此处不再赘述。

在执行到声明语句时，执行到在归约出 var list 的过程中，将 list 中的变量逐个在符号表中建立项目。此时还不知道每个变量被分配多少内存，因为决定内存的类型说明还没有被归约出来。

此时维护一个生命 patchlist，在语法分析进行到归约出变量类型之后，按照 patchlist 填写各个变量的地址。

```

def dealWithObj(src,action,stack,wl,yuyis,sigtabe):
    length = len(src)
    stack.push(0)
    flag = 0
    global doflag
    while(True):
        stack.show()
        print src[flag].s,src[flag].num
        tmp = action[int(stack.top())][int(src[flag].num)]
        if(src[flag].s == "while"):
            addToTable("while",0,0,yuyis,sigtabe,0)
            print "(while,"+"R30"+"","+","+"do"+"")
        if(is_num(tmp)):
            if(src[flag].s == "do"):
                addToTable("do",0,0,yuyis,sigtabe,0)
            if(src[flag].s == "then"):
                addToTable("then",0,0,yuyis,sigtabe,0)
            stack.push(int(tmp))
            flag +=1
        else:
            if(tmp=="acc"):
                return 0
            else:
                print "this is tmp :",tmp
                tmp = tmp.split(" ")
                length = len(tmp)
                if(not tmp[length-1]=='-2' and not tmp[length-2]=='-2'):
                    kkk = length - 4

```

在用 C++或其他语言编写的编译器中，由于我们需要在语义栈中保存大量的综合属性或者继承属性，我们需要为不同的文法变量定义不同的结构体或共用体。

在 python 中，我们可以利用 python 的语言特性避免这一个麻烦。在语义栈中，我们只压入不同种类的 list。Python 中的 list 是不定长的，保存的内容也不是同一类型，可以以很大的自由度进行自定义。

我们只需要约定好执行不同的语义动作时需要的属性会出现在占中语义变量的哪个维度上就可以了。省去了定义结构体和共同体的麻烦。

由于生成的是 MIPS 汇编，在条件分支的处理上和 x86 分支不同。在编译器中约定，所有的布尔类型的判断结果都存储在 R30 寄存器中。程序根据 R30 存储器中的内容进行跳转。比如下面的指令

```

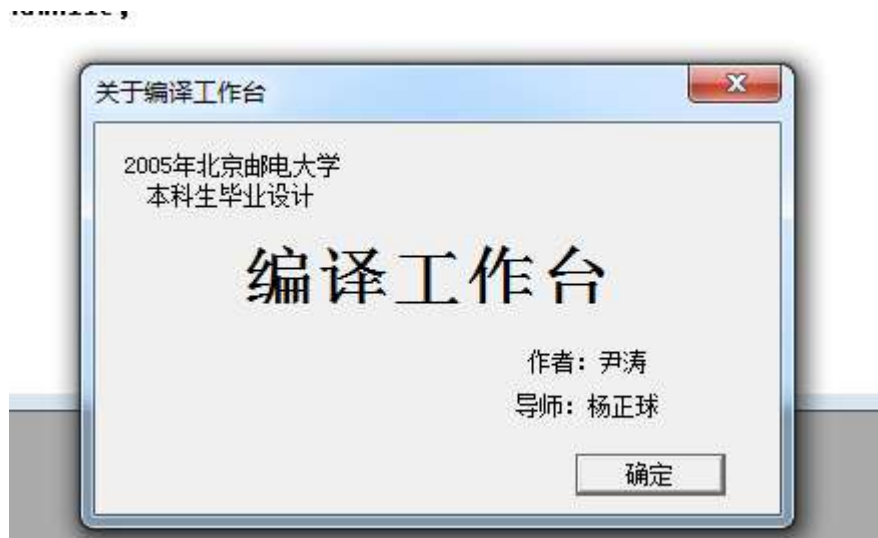
tmp_code += "OR R1,R3\n"
tmp_code += "AND R1,#10000000\n"
tmp_code += "MOV R30,R1\n"

```

3. 驱动表生成器

驱动表生成器并不是整个编译程序的一部分。它是用来辅助生成编译程序的工具。

由于手动构造文法的 LALR（1）分析表工作量很大，所以我使用姜老师推荐的编译工作台进行驱动表的生成。



编译工作台的作者是尹涛。该软件可以根据拓广文法自动生成 SLR, LR 和 LALR 分析表, 并按照分析表和自动机两种方式展示出来。

状态	input	output	,	+	-	*	/
0							
1							
2							
3							
4	shift 5						
5		shift 6					
6		shift 7					
7							
8							
9							
10							
11							
12		shift 26					
13							
14		reduce List -> id					
15							
16							
17							
18							

生成的 LALR 分析表如图所示, 是一个 HTML 文件。

将此表读入语法分析器进行使用的方法有两个。一种是手动写入宏定义。这种方法的优点是编译器执行效率高, 缺点是需要大量的人工劳动。同时可移植性差, 一旦文法改变, 需要手工重写大量的代码。

另一种方法是读入该文件, 对表格内容进行 parse, 得到驱动表。

由于从该 HTML 文件得到驱动表并不是语法分析程序的功能, 所以将该功能分离出来, 有单独的驱动表生成程序执行。

驱动表生成器由两个部分构成。一个部分分析 HTML 的表头, 得到所有文法符号的编号。另外一个程序 parse 表的主体, 得到驱动表内的内容。

该程序用来得到文法符号的编号。

```
1 f = open("./cha.txt")
2 f1 = open("./characterList.txt", 'w')
3 counter = 0
4 for i in f:
5     s = i.split(">")
6     arr = s[1].split("<")
7     print arr[0]
8     tmp=arr[0]
9     tmp+='@'
10    tmp+=str(counter)
11    tmp+='\n'
12    counter+=1
13    f1.write(tmp)
```

该程序用来得到驱动表。

```

37     if(flag == 0):
38         col+=1
39         i = i.split("<td nowrap>")[1]
40         i = i.split("</td>")[0]
41         i = i.split("&nbsp;")
42         if(is_num(i[0]) and isfirst == 0):
43             #print "this is col "+str(col)+" : goto "+i[0]
44             action[row][col] = int(i[0])
45         if(is_num(i[0]) and isfirst == 1):
46             col-=1
47             isfirst-=1
48         if(i[0]=='accept'):
49             action[row][col] = 'acc'
50             print "get acc"
51         if(i[0]=="shift"):
52

```

我定义的驱动表的格式是一个用@作为分隔符的矩阵。其中，如果某个元素是-1，则表明该状态错误。如果某状态为正整数，则表明应当转移到这个状态。如果是规约式，则表明当前应该执行这个规约并且查 GOTO 表得到应该转移到的状态。

部分驱动表如下图所示。

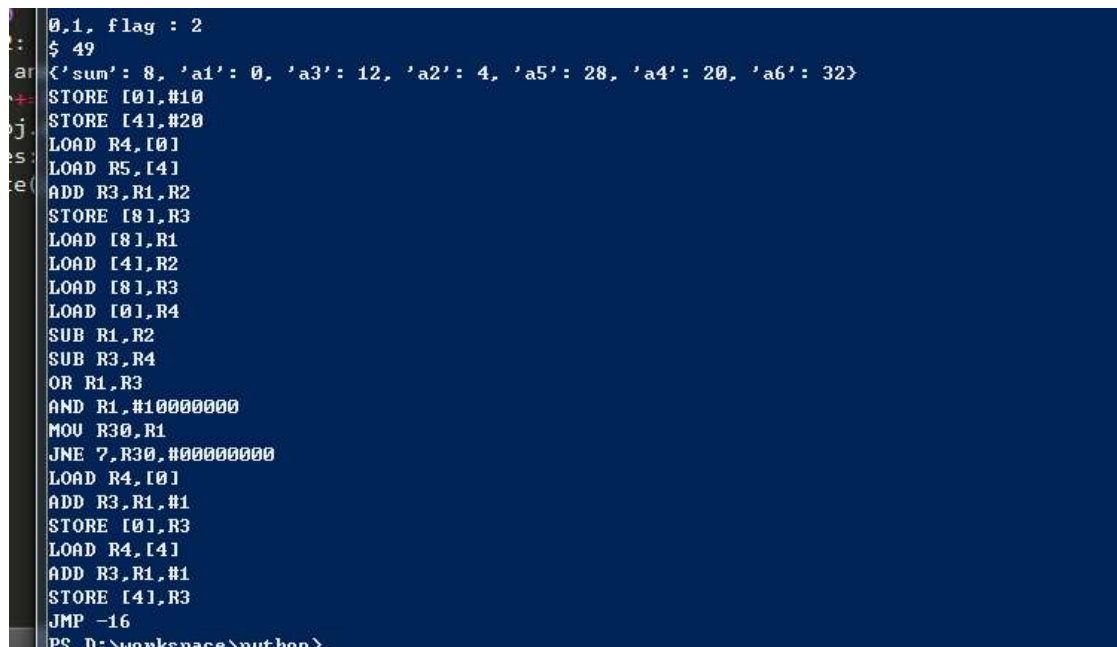
[illegible]

五. 测试

源代码如下

```
prog main (input , output)
beginD
a1,a2,sum:integer segD
a3,a4:real segD
a5,a6:integer
endD
seg
a1:=10 segS
a2:=20 segS
sum:=a1+a2 segS
beginwhile
while sum relop a1 and sum relop a2 do a1 := a1 + 1 segS a2 := a2 + 1
endwhile
```

生成的汇编和符号表如下



```
0,1, flag : 2
$ 49
ar <'sum': 8, 'a1': 0, 'a3': 12, 'a2': 4, 'a5': 28, 'a4': 20, 'a6': 32>
STORE [0],#10
STORE [4],#20
LOAD R4,[0]
LOAD R5,[4]
ADD R3,R1,R2
STORE [8],R3
LOAD [8],R1
LOAD [4],R2
LOAD [8],R3
LOAD [0],R4
SUB R1,R2
SUB R3,R4
OR R1,R3
AND R1,#10000000
MOV R30,R1
JNE 7,R30,#00000000
LOAD R4,[0]
ADD R3,R1,#1
STORE [0],R3
LOAD R4,[4]
ADD R3,R1,#1
STORE [4],R3
JMP -16
PS D:\workspace\python>
```

六. 总结

本次实验是我在大学入学以来完成的内容最多，难度最大的实验。在这个实验中，我不仅练习了用代码实现自己的数据结构和算法设计的能力，还了解了高级语言被编译成机器代码的数学原理。

在设计并实现自己的语言的过程中，对其他常用的高级语言如 C 和 C++ 等的设计和实现也有了更多的了解。给自己提供了一个从编译底层掌握高级语言特性的机会。

感觉收获十分巨大。