2021-2022学年 https://www.cc98.org/topic/5355540

12.1.1 大题1

"declaration before use" 和 "most closely nested rule for block structure" 都是编程中的两个重要原则。

首先，"declaration before use" 指的是在程序中必须先声明一个变量或函数，然后才能使用它。换句话说，在使用一个变量或函数之前，必须先在代码中进行相应的声明。这个原则适用于大多数编程语言，包括C、C++、Java等。

例如，在C语言中，我们需要先声明变量，然后再使用它：

```
int x;  // 声明变量x

x = 5;  // 使用变量x，赋值为5
```

如果我们没有先声明变量x，而直接使用它，则会导致编译错误。

而 "most closely nested rule for block structure" 是指在一个程序中，如果多个同名变量被定义在嵌套的作用域或代码块中，那么最内层的作用域或代码块中的变量将覆盖外层作用域或代码块中的同名变量。也就是说，在嵌套的作用域中，最近的那个变量声明将起作用。

例如，考虑以下代码片段：

```
int x = 10;

if (x > 5) {
    int x = 5;
    printf("%d", x);  // 输出5，使用最近的x变量
}

printf("%d", x);  // 输出10，使用外层的x变量
```

在这个例子中，我们在外层作用域中声明了一个变量x，并赋值为10。然后，在if语句的代码块中，我们又声明了一个同名的变量x，并赋值为5。当我们在if语句的代码块中打印变量x时，输出的是最近的内层变量x的值，即5。而在if语句之外的代码中打印变量x时，输出的是外层的变量x的值，即10。

这两个原则都是为了确保程序的变量和函数在使用之前已经被正确声明，并且在作用域之间能够正确地处理同名的变量。

12.1.2 大题2

四元式形式的三地址码

定义：三地址码用来表示算术表达式求值，形如 `x = y op z`。对于三地址码，保存其信息需要 3 个地址以及 1 个操作符。因此我们用 **四元式 quadruple** 的方式来存储三地址码。例如下面的经典

```
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

以及对应的四元式，其中_表示空：

```
(rd,x,_,_)          (asn,t2,fact,_)
(gt,x,0,t1)         (sub,x,1,t3)
(if_f,t1,L1,_)      (asn,t3,x,_)
(asn,1,fact,_)      (eq,x,0,t4)
(lab,L2,_,_)        (if_f,t4,L2,_)
(mul,fact,x,t2)     (wri,fact,_,_)
                    (lab,L1,_,_)
                    (halt,_,_,_)
```

三地址码另一个不同的实现是用自己的指令来代表临时变量，这样地址域从 3 个减少到了 2 个，目标地址总是一个临时变量，用行号表示。这种实现方式称为 **三元式 triple**。前述三地址码用三元式表示就是(if_f是如果错)：

```
(0)    (rd,x,_)
(1)    (gt,x,0)
(2)    (if_f,(1),(11))
(3)    (asn,1,fact)
(4)    (mul,fact,x)        (8)    (eq,x,0)
(5)    (asn,(4),fact)      (9)    (if_f,(8),(4))
(6)    (sub,x,1)           (10)   (wri,fact,_)
(7)    (asn,(6),x)         (11)   (halt,_,_)
```

```
i = 0;
while(i < 5){
    if(i < 3) then{
        x = i*2;
    }else{
        x = i*3;
    }
    i = i+1;
}
```

自拟答案：

```
(0)(asn,0,i)
(1)(lt,5,i)
(2)(if_f,(1),(11))
(3)(lt,i,3)
(4)(if_f,(3),(7))
(5)(mul,i,2)
(6)(asn,x,(5))
```

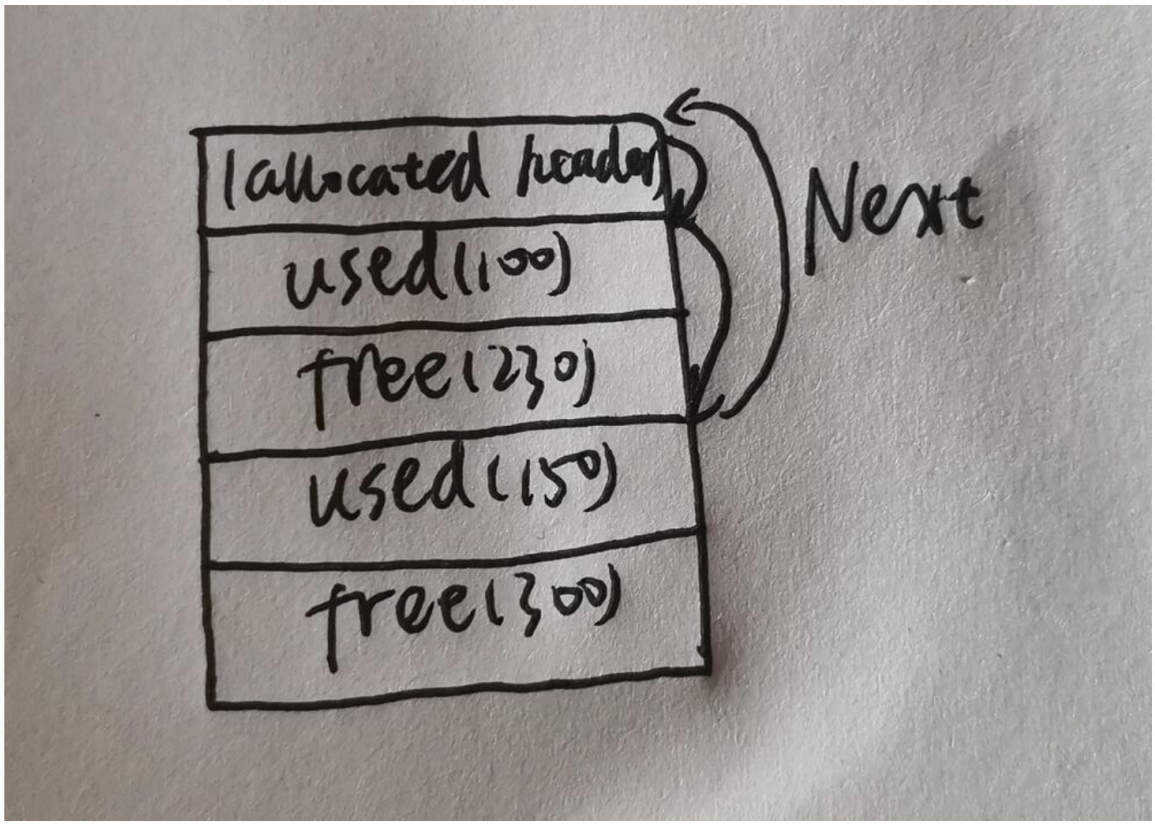
```
(7)(mul,i,3)
(8)(asn,x,(7))
(9)(add,i,1)
(10)(asn,i,(9))
(11)(halt,_,_)
```

12.1.3 大题3

感觉完全没见过==

5.给出如下内存布局，画出相关操作后的内存情况

(1) malloc(240) (2)free(150) after malloc(240)



12.2 2020-2021学年

12.2.1 大题1

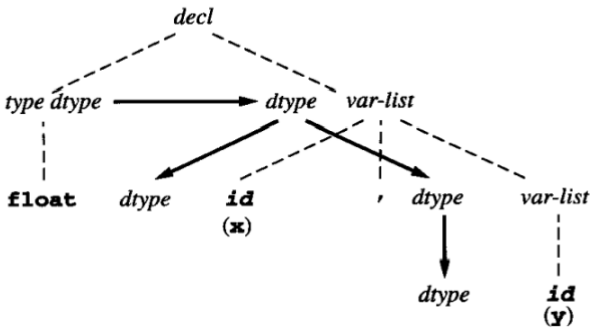
（1）写出计算类型的属性文法；

（2）画出 `x,y,z: int` 的依赖图。

```
decl -> varlist : type
varlist -> varlist, id | id
type -> int
```

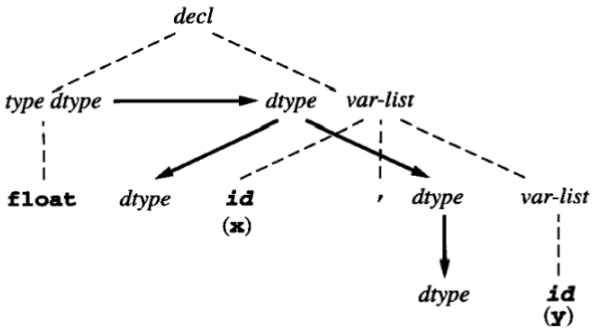
参考复习：

作为一个例子，字符串 `float x, y` 的相关图是：



和

作为一个例子，字符串 `float x, y` 的相关图是：



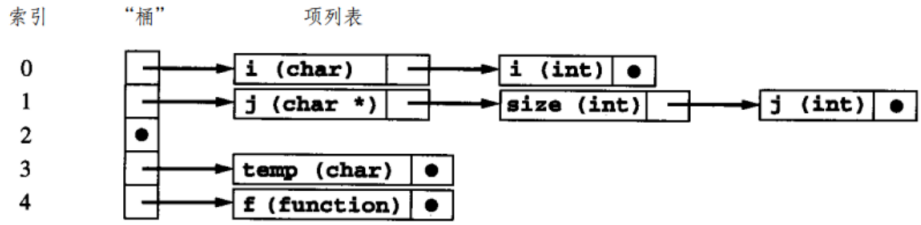
12.2.2 大题2

给出下面代码的分隔符号表，每个作用域在不同的表里（任意挑一个作用域写，保证是三层表即可）。

```
int i, j;
int f(int size) {
    char l, tmp;
    {
        double j;
        ...
    }
        ...
    {
        char* j;
        ...
    }
}
```

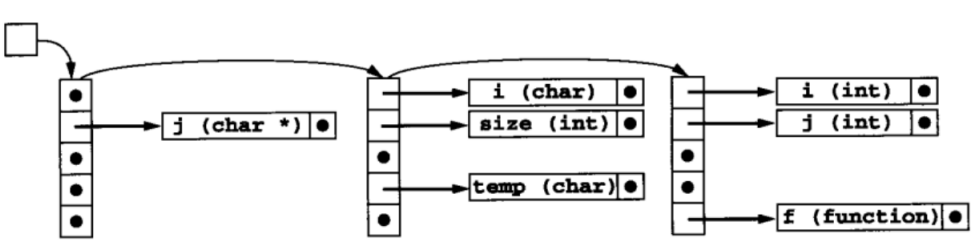
参考知识：

采用哈希：



这里面有多个 `i`，因为作用域屏蔽。
这种有冲突就搞个列表的方式叫 separate chaining 分离链表法；还有开放寻址法 open addressing，就是冲突后找下一个有空的。

多个作用域也可以像这样维护一个符号表栈：



12.2.3 大题3

写出下列代码的 pcode。

```
if (true)
while (true)
{
  if (false) {
        break;
  }
    else {
        other;
    }
}
```

p-code定义：也就是有一个栈，不断弹出和运算，更多的调用看xyx笔记-**感觉基本没提到今年**

P-Code 的运行依赖一个临时的栈。例如，`2 * a + (b - 3)` 的 P-Code 如下：

```
ldc 2          ; load constant 2
lod a          ; load value of variable a
mpi            ; integer multiplication
lod b          ; load value of variable b
ldc 3          ; load constant 3
sbi            ; integer substraction
adi            ; integer addition
```

可以看到，`mpi`，`sbi`，`adi` 这些操作都是从栈上弹出 2 个操作数，进行计算，再压回栈上的。

对于赋值语句 `x := y + 1`，其 P-Code 是：

```
lda x          ; load address of x
lod y          ; load value of y
ldc 1          ; load constant 1
adi            ; add
sto            ; store top to address
               ; below top & pop both
```

12.2.4 大题4

（1）解释 pass-by-value-result 和 pass-by-reference 的区别（2）给出下列代码在 pass-by-value-result 与 pass-by-reference 时的输出。

```
int f(int x, int y) {
    y = 5;
    x++;
}

main() {
    int a=1;
    f(a, a);
    printf("%d", a);
}
```

参考：xyx笔记

12.2.5 大题5

（1）解释 align a; 的作用（2）如果 MEMSIZE 为 8096，求程序的堆内存大小（3）这种内存管理方法堆中是否有碎片?

```
typedef double align;
typedef union header
{
    struct {
        union header* t;
        unsigned aa;
        unsigned bb;
    }
    align a;
} header;
static header mem[MEMSIZE];
static header* memptr = NULL;
```