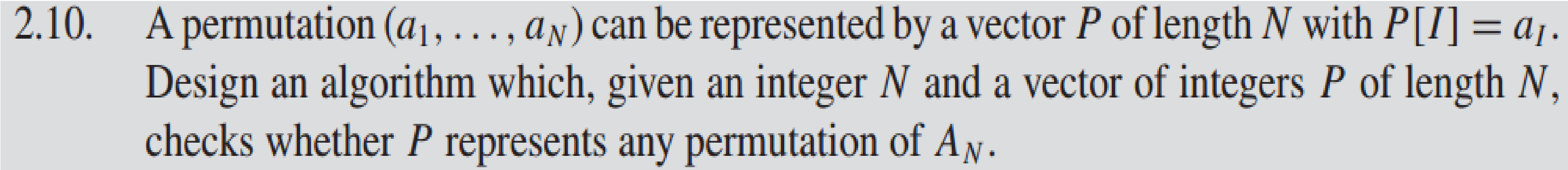
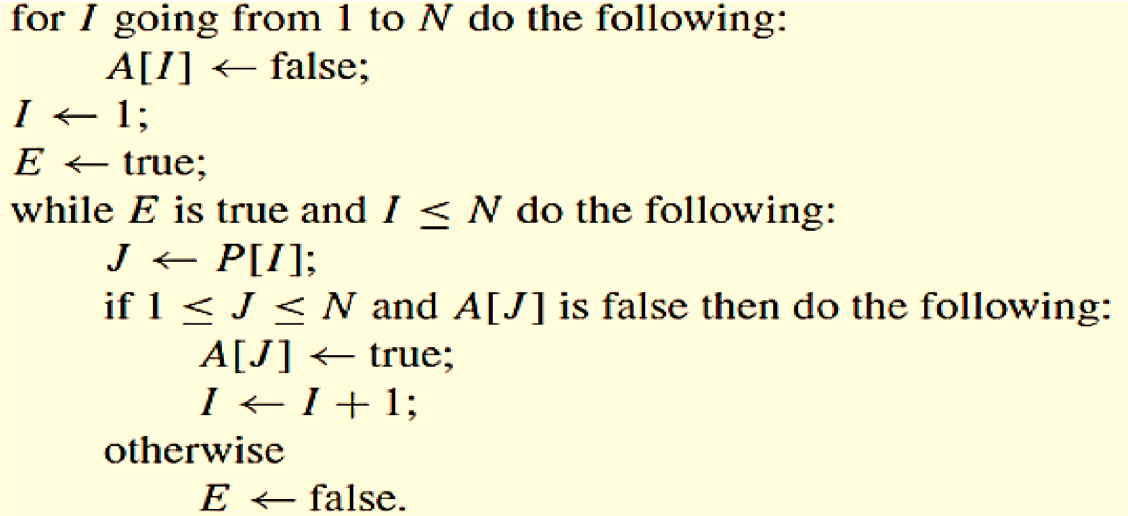
作业1-5

DH第2章练习10、11、12、13、14、15、16

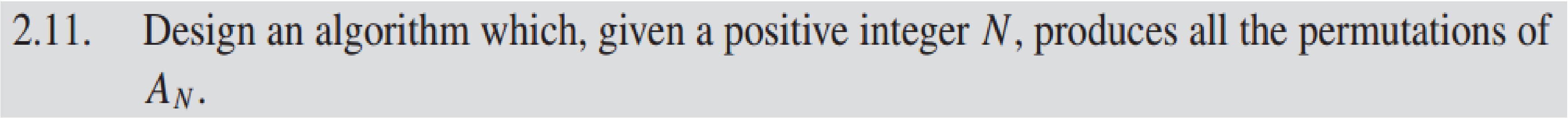


2.10. A permutation $(a_1, \ldots, a_N)$ can be represented by a vector P of length N with $P[I] = a_I$. Design an algorithm which, given an integer N and a vector of integers P of length N, checks whether P represents any permutation of A_N.

- Algorithm isPermutation(N,P)
- Input:
  - N: a integer
  - P:a vector of integers of length N
- Output:
  - true: if P represents a permutation of A_N
  - false: if P does not represent any permutation of A_N



```
for $I$ going from 1 to $N$ do the following:
        $A[I] \leftarrow$ false;
$I \leftarrow 1$;
$E \leftarrow$ true;
while $E$ is true and $I \leq N$ do the following:
        $J \leftarrow P[I]$;
        if $1 \leq J \leq N$ and $A[J]$ is false then do the following:
                $A[J] \leftarrow$ true;
                $I \leftarrow I + 1$;
        otherwise
                $E \leftarrow$ false.
```



2.11. Design an algorithm which, given a positive integer N, produces all the permutations of A_N.

Algorithm getAllPermutation(N)

- Input:
  - N: a positive integer
- Output:
  - All the permutation of A_N

```
int [N+1]used={0,0,…0};
Int [N]perm;
permutation(N,0,used)
```

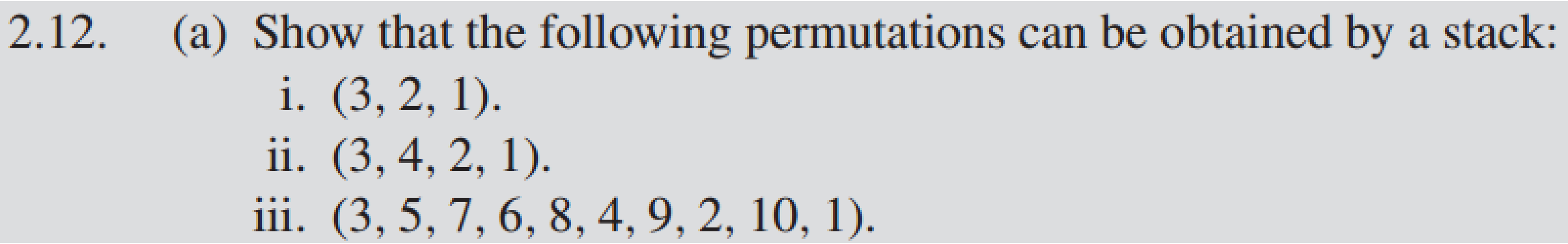
Subprocess permutation(int N, int k, int*used)

```
If(k==N){
    for(int i=0;i<N;i++){
        cout<<perm[i]<<" ";
    }
    cout<<endl;
}
for(int i=1;i<=N;i++){
    if(used[i]==0){
        perm[k]=i;
        used[i]=1;
        permutation(N,k+1,used);
        used[i]=0;
    }
}
```

```
for $I$ going from 1 to $N$ do the following:
    $A[I] \leftarrow$ true;
call perms-from 1.
```

where the subroutine **perms**, with local variable J, is defined by

```
subroutine perms-from $K$:
    if $K > N$ then do the following:
        print("New permutation: (");
        for $J$ going from 1 to $N$ do print($P[J]$);
        print(")");
    otherwise (i.e., $K \leq N$) do the following:
        for $J$ going from 1 to $N$ do the following:
            if $A[J]$ is true then do the following:
                $P[K] \leftarrow J$;
                $A[J] \leftarrow$ false;
                call perms-from $K + 1$;
                $A[J] \leftarrow$ true;
    return.
```



2.12. (a) Show that the following permutations can be obtained by a stack:

i. (3, 2, 1).

ii. (3, 4, 2, 1).

iii. (3, 5, 7, 6, 8, 4, 9, 2, 10, 1).

I.

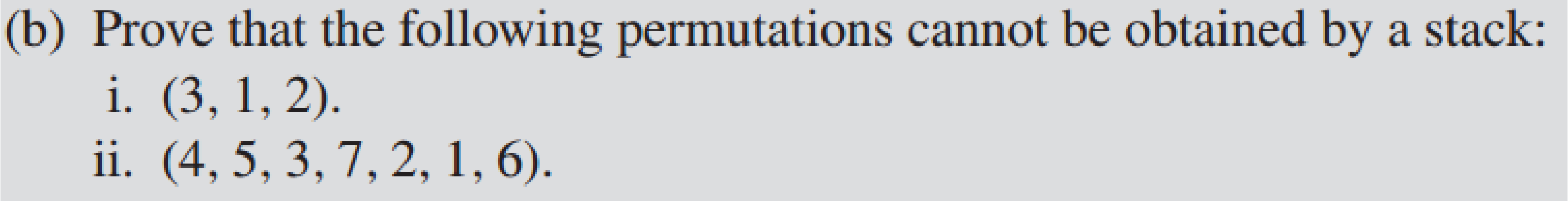
- read(x), push(x,s),
- read(x), push(x,s),
- read(x), print(x),
- pop(x), print(x)
- pop(x), print(x)

II.

- read(x), push(x,s),
- read(x), push(x,s),
- read(x), print(x),
- read(x), print(x),
- pop(x), print(x)
- pop(x), print(x)

III.

- read(x), push(x,s),
- read(x), push(x,s),
- read(x), print(x),//3
- read(x), push(x,s),
- read(x), print(x),//5
- read(x), push(x,s),
- read(x), print(x),//7
- pop(x), print(x),//6
- read(x), print(x),//8
- pop(x), print(x),//4
- read(x), print(x),//9
- pop(x), print(x),//2
- read(x), print(x),//10
- pop(x), print(x),//1



(b) Prove that the following permutations cannot be obtained by a stack:

i. (3, 1, 2).

ii. (4, 5, 3, 7, 2, 1, 6).

I. (3,1,2)

- 3输出时，2和1必然在栈中，2必须在1之上，所以2必然比1先出栈；

II. (4,5,3,7,2,1,6)

- 7输出时，6、2和1必然在栈中，6必须在2、1之上，所以6必然比2、1先出栈；

(c) How many permutations of *A4* *cannot* be obtained by a stack?

- 10：枚举
  - (1,4,2,3)
  - (2,4,1,3)
  - (3,1,2,4) (3,1,4,2) (3,4,1,2)
  - (4,1,2,3) (4,1,3,2) (4,2,1,3) (4,2,3,1) (4,3,1,2)

(c) How many permutations of A_N *cannot* be obtained by a stack?

(c) How many permutations of A_N *cannot* be obtained by a stack?

- C(N)=# permutations of A_N can be obtained by a stack
- $C(N) = \frac{(2*N)!}{N!*(N+1)!}$ (Catalan Numbers)

$$C(n) = C(0) * C(n-1) + C(1) * C(n-2) + C(2) * C(n-3) + \cdots + C(n-1) * C(0)$$

$$= \sum_{i=0\sim n-1} C(i) * C(n-1-i)$$

C(0)=1

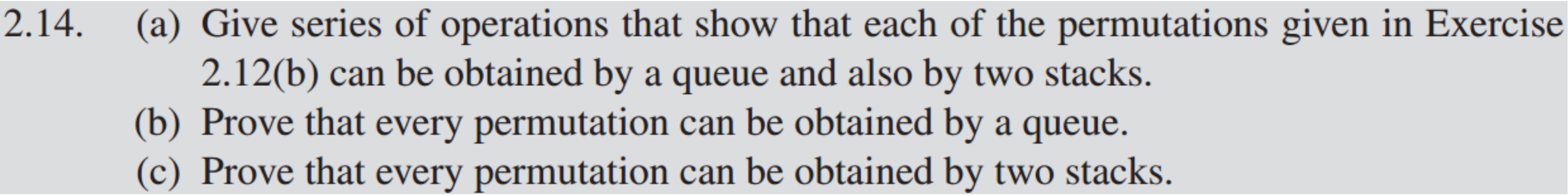
https://en.wikipedia.org/wiki/Catalan_number

2.13. Design an algorithm that checks whether a given permutation can be obtained by a stack. In case the answer is yes, the algorithm should also print the appropriate series of operations. In your algorithm, in addition to **read**, **print**, **push**, and **pop**, you may use the test **is-empty**(*S*) for testing the emptiness of the stack *S*.

Algorithm genOperations(P)

- Input
  - P: a permutation of A_N
- Output
  - "NO", if P can not be obtained by a stack
  - "YES" and S_O : a sequence of operations generate P, if P can be obtained by a stack

```
E ←true;
I ← 1;
ops ← EmptyList;
while input is not empty and E=true do the following:
   read(Y);
   while Y>I do the following:
      push(I,S);
      ops.add(1);
      I ← I+1;
   if Y=I then do the following:
      ops.add(2); I ← I+1;
   else (i.e., Y<I) do the following:
      pop(Z,S);
      ops.add(3);
      if(Z!=Y) do the following:
         E ←false;
         break;
if E=true do the following:
   print("YES");
   for each integer o in ops do:
      if o=1 do:
         print("read(X); push(X,S);");
      else if o=2 do:
         print("read(X); print(X);");
      else if o=3 do:
         print("pop(X,S);");
else do:
   print ("NO");
```

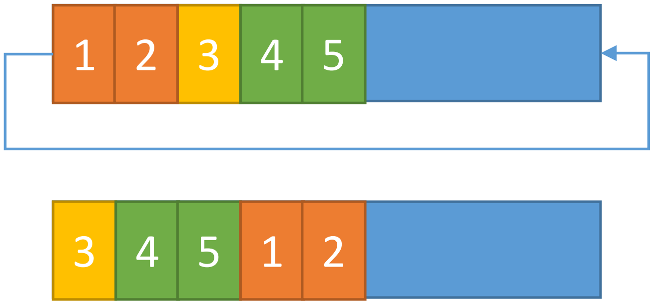


2.14. (a) Give series of operations that show that each of the permutations given in Exercise 2.12(b) can be obtained by a queue and also by two stacks.

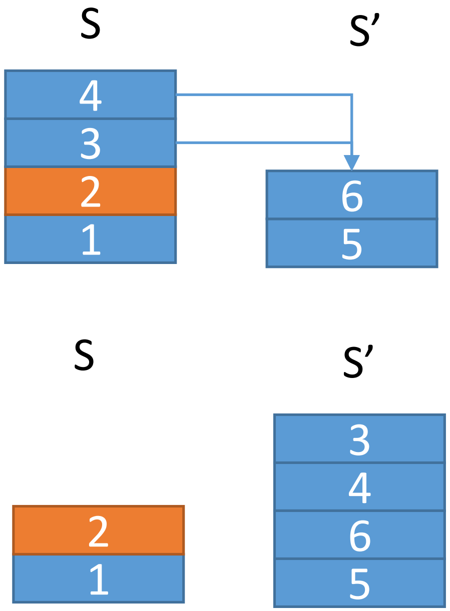
(b) Prove that every permutation can be obtained by a queue.

(c) Prove that every permutation can be obtained by two stacks.

- (a)(3,1,2)
  - read(x); push(x,s); read(x); push(x,s); read(x); print(x); pop(x,s); push(x,s’); pop(x,s); print(x); pop(x,s’); print(x);
- (b)

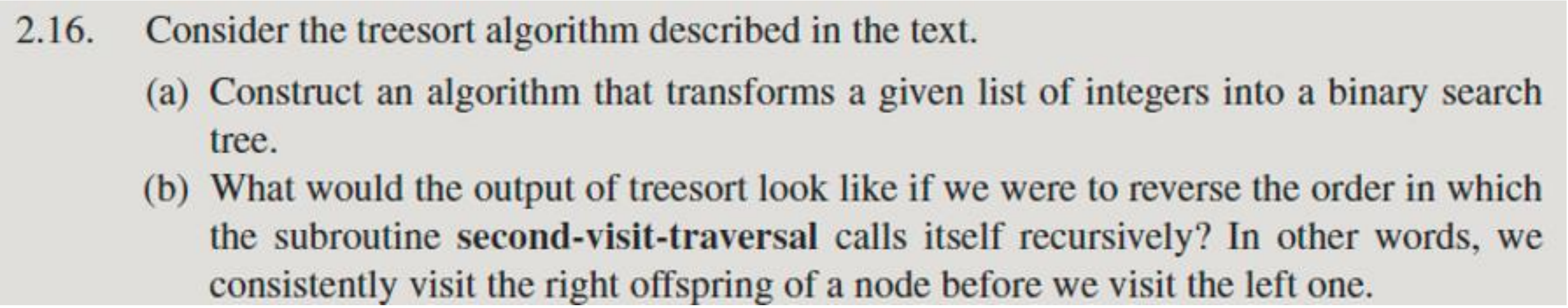


- (C)



2.15. Extend the algorithm you were asked to design in Exercise 2.13, so that if the given permutation cannot be obtained by a stack, the algorithm will print the series of operations on two stacks that will generate it.

```
$E \leftarrow$ true;
$I \leftarrow 1$;
while input is not empty do the following:
    read($Y$);
    while $Y > I$ do the following:
        push($I$, $S$);
        print(“read($X$)”);
        print(“push($X$, $S$)”);
        $I \leftarrow I + 1$;
    if $Y = I$ then do the following:
        print(“read($X$)”);
        print(“print($X$)”);
        $I \leftarrow I + 1$;
    otherwise (i.e., $Y < I$) do the following:
        pop($Z$, $S$);
        print(“pop($X$, $S$)”);
        while $Z \neq Y$ do the following:
            $E \leftarrow$ false;
            push($Z$, $S'$);
            print(“push($X$, $S'$)”);
            pop($Z$, $S$);
            print(“pop($X$, $S$)”);
        print(“print($X$)”);
        while is-empty($S'$) is false do the following:
            pop($Z$, $S'$);
            print(“pop($X$, $S'$)”);
            push($Z$, $S$);
            print(“push($X$, $S$)”).
```



2.16. Consider the treesort algorithm described in the text.

(a) Construct an algorithm that transforms a given list of integers into a binary search tree.

(b) What would the output of treesort look like if we were to reverse the order in which the subroutine **second-visit-traversal** calls itself recursively? In other words, we consistently visit the right offspring of a node before we visit the left one.

- Input:
    - A[1,…,N]: an array of distinct integers
- Output:
    - T: the root node of an binary search tree exactly containing all integers from A;

```
T← new node(A[1]);
for i ← 2 to N do:
    Tc ←T;
    while true do:
        if T.v>A[i] do:
            if T.left=null do:
                T.left ←new node(A[i]);
                break;
            else do:
                T ←T.left;
        else if T.v<A[i] do:
            if T.right=null do:
                T.right ←new node(A[i]);
                break;
            else do:
                T ←T.right;
retrun T;
```

Backus Naur Form (BNF)

- In computer science, **BNF** (**Backus Normal Form** or **Backus–Naur Form**) is one of the two[1] main notation techniques for context-free grammars;
- often used to describe the syntax of languages used in computing, such as computer programming languages, document formats, instruction sets and communication protocols;
- the other main technique for writing context-free grammars is the van Wijngaarden form. They are applied wherever exact descriptions of languages are needed:
    - for instance, in official language specifications, in manuals, and in textbooks on programming language theory.

https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form

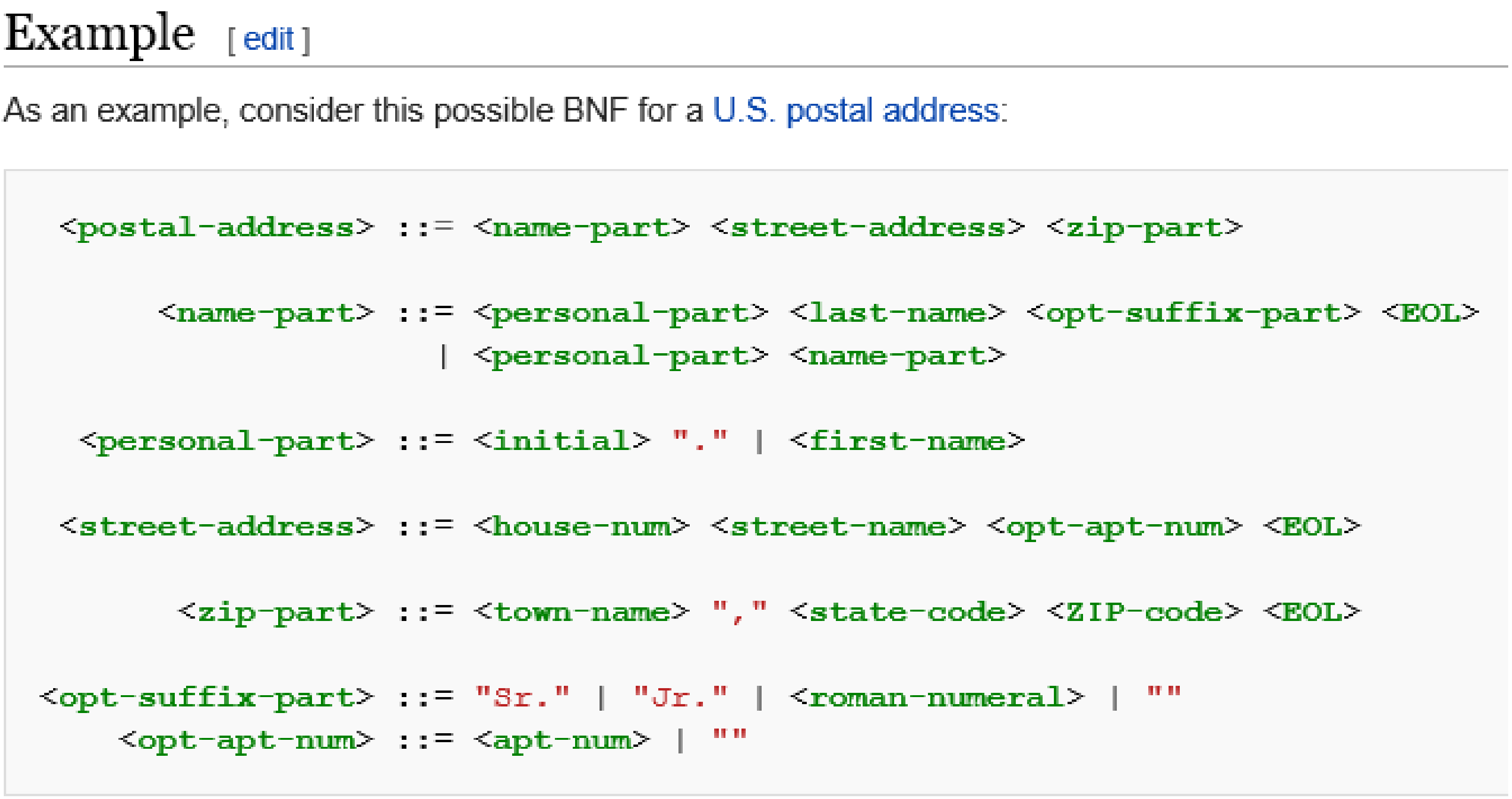
Backus Naur Form (BNF)

- John Backus和Peter Naur首次引入一种 形式化符号来描述给定语言的语法（最早用于描述 ALGOL 60 编程语言）。
- 早在UNESCO（联合国教科文组织）关于ALGOL 58的会议上提出的一篇报告中，Backus就引入了大部分BNF符号。虽然没有 什么人读过这篇报告，但是在 Peter Naur读这篇报告时， 他发现Backus对 ALGOL 58 的解释方式和他的解释方式有一些不同之处，这使他感到很惊奇。首次设计 ALGOL的所有参与者都开始发现了他的解释方式的一些弱点， 所以他决定对于以后版本的 ALGOL应该以一种类似的形式进行描述，以让所 有参与者明白他们在对什么达成一致意见。他做了少量修改，使其几乎可以通用，在设计 ALGOL 60的会议上他为ALGOL 60草拟了自己的BNF。
- 关于那个时期编程语言历史的更多细节，参见1978 年8月，《Communications of the ACM（美国计算机学会通讯）》，第21 卷，第8期中介绍Backus获图灵奖的文章。这个注释是由来自Los Alamos Natl.实验室的William B. Clodius建议的。
- 自从那以后，几乎每一个新编程语言书的作者都使用BNF来描述语言的语法 规则

BNF

- BNF的元符号：
  - ::= 表示“定义为”
  - | 表示“或者”
  - < > 尖括号用于括起类别名字。
    - 尖括号将语法规则名字（也称为非终结符）同终结符区分开来，终结符想表达什么意思就怎么书写
    - 可选项被括在元符号“[”和“]”中
    - 重复项（零个或者多个）被括在元符号“{” 和“}”中
    - 终结符用引号（"）引起来， 以和元符号区别开来

BNF



Example [edit]

As an example, consider this possible BNF for a U.S. postal address:

```
 <postal-address> ::= <name-part> <street-address> <zip-part>

      <name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL>
                    | <personal-part> <name-part>

  <personal-part> ::= <initial> "." | <first-name>

 <street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

       <zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""
    <opt-apt-num> ::= <apt-num> | ""
```

https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form

BNF

- Pascal:
  - http://condor.depaul.edu/ichu/csc447/notes/wk2/pascal.html