

# Virtuelle Maschine in der Informatik

- Eine virtuelle Maschine ist ein Rechner, der physisch nicht existiert. Seine Bestandteile werden durch ein Programm realisiert, das alle Bestandteile eines physischen Rechners nachbildet.
  - Register
  - Speicher
  - Steuerschleife
  - Befehlssatz

# Dreiadressmaschine

Eine beispielsweise arithmetische Operation wird durch einen Befehl mit 3 Parametern realisiert, den beiden Operanden und dem Ziel, auf dem das Ergebnis der Operation abgelegt wird. Zwischenergebnisse komplexer Ausdrücke werden in temporären Variablen abgelegt.

Erg  $\leftarrow$  Op1 + Op2

Erg  $\leftarrow$  Op1 - Op2

Erg  $\leftarrow$  Op1 \* Op2

Erg  $\leftarrow$  Op1 / Op2

Erg  $\leftarrow$  Op1 & Op2

...

# Zweiadressmaschine

ähnlich der Dreiadressmaschine, jedoch überschreibt das Ergebnis einen der beiden Operanden, so dass der 3. Parameter entfällt.

$Op1 \leftarrow Op1 + Op2$

$Op1 \leftarrow Op1 - Op2$

$Op1 \leftarrow Op1 * Op2$

$Op1 \leftarrow Op1 / Op2$

$Op1 \leftarrow Op1 \& Op2$

...

Beispiel ASM x86:

```
xor    eax,eax
add    esi,eax ; Summe
shl    al,3
```

# Einadressmaschine

Alle Operationen werden mit einem Akkumulatorregister ausgeführt, das den jeweils 1. Operanden enthält

Load Operand

Akkumulator += Operand

Akkumulator -= Operand

Akkumulator &= Operand

....

Store Destination

# Nulladressmaschine - Stackmaschine

Operanden und Ergebnisse werden auf einem Kellerspeicher abgelegt. Auch die Variablen der Prozeduren werden im Keller angelegt. Sie werden durch eine Relativadresse, die relativ zum Anfang des Variablenbereiches (Adresse der 1. Variablen) gebildet wird, adressiert.

Push op1

Push op2

Push op3

$(op1+op2)*op3$

Add

Mul

....

Die virtuelle Maschine der LV basiert auf einer Stackmaschine

# Befehlssatz

- Befehle zum Datentransport (push, pop, store)
- Arithmetische Befehle (add, sub, cmp,...)
- Sprungbefehle(jmp, call, jnot, ...)
- Befehle zur Programmorganisation (entryproc)
- I/O-befehle

```

typedef enum TCODE
{
    /*--- Kellerbefehle ---*/
    puValVrLocl, /*00 (int Displ) [Kellern Wert lokale Variable] */
    puValVrMain, /*01 (int Displ) [Kellern Wert Main Variable] */
    puValVrGlob, /*02 (int Proc, int Displ) [Kellern Wert globale Variable] */
    puAdrVrLocl, /*03 (int Displ) [Kellern Adresse lokale Variable] */
    puAdrVrMain, /*04 (int Displ, int Proc) [Kellern Adresse Main Variable] */
    puAdrVrGlob, /*05 (int Displ) [Kellern Adresse globale Variable] */
    puConst, /*06 (int Index) [Kellern einer Konstanten] */
    storeVal, /*07 () [Speichern Wert -> Adresse, beides aus Keller] */
    putVal, /*08 () [Ausgabe eines Wertes aus Keller nach stdout] */
    getVal, /*09 () [Eingabe eines Wertes von stdin -> Keller ] */
    /*--- arithmetische Befehle ---*/
    vzMinus, /*0A () [Vorzeichen -] */
    odd, /*0B () [ungerade -> 0/1] */
    /*--- binaere Operatoren kellern 2 Operanden aus und das Ergebnis ein ----*/
    OpAdd, /*0C () [Addition] */
    OpSub, /*0D () [Subtraktion ] */
    OpMult, /*0E () [Multiplikation ] */
    OpDiv, /*0F () [Division ]

```

```

cmpEQ      ,/*10 ()          [Vergleich = -> 0/1]          */
cmpNE      ,/*11 ()          [Vergleich # -> 0/1]          */
cmpLT      ,/*12 ()          [Vergleich < -> 0/1]          */
cmpGT      ,/*13 ()          [Vergleich > -> 0/1]          */
cmpLE      ,/*14 ()          [Vergleich <=> 0/1]           */
cmpGE      ,/*15 ()          [Vergleich >=> 0/1]           */
/*--- Sprungbefehle ---*/
call       ,/*16 (int ProzNr) [Prozeduraufruf]              */
retProc    ,/*17 ()          [Ruecksprung]                  */
jmp        ,/*18 (int RelAdr) [SPZZ innerhalb der Funktion] */
jnot       ,/*19 (int RelAdr) [SPZZ innerhalb der Funkt.,Beding.aus Keller] */
entryProc  ,/*1A (int lenCode, int ProcIdx, lenVAR)         */
putStrg    ,/*1B (char[])                                     */
pop        ,/*1C */
swap       ,/*1D           Austausch Adresse gegen Wert    */
EndOfCode  ,/*1E */
put        ,/*1F Wert und Port vom Stack*/
get        ,/*20 Adr und Port vom Stack*/
OpAddAddr  /*21 add 64bitAddress + 32 bit Offset          */
}tCode;

```



# Arbeitsweise der Stackmaschine

```
PL/0:
const c=3;
var a,b;
begin
  ?b;
  a:=B*10+c;
  !a
end
```

Code :

Procedure

00000000: 1A EntryProc 001F, 0000, 0008

00000007: 04 PushAdrVarMain 0004

0000000A: 09 GetVal

0000000B: 04 PushAdrVarMain 0000

0000000E: 01 PushValVarMain 0004

00000011: 06 PushConst 0001

00000014: 0E Mul

00000015: 06 PushConst 0000

00000018: 0C Add

00000019: 07 StoreVal

0000001A: 01 PushValVarMain 0000

0000001D: 08 PutVal

0000001E: 17 ReturnProc

Const 0000:0003

Const 0001:0010

Adresse Variable B

Adresse Variable A

Wert Variable B

Ablegen des Ergebnisses

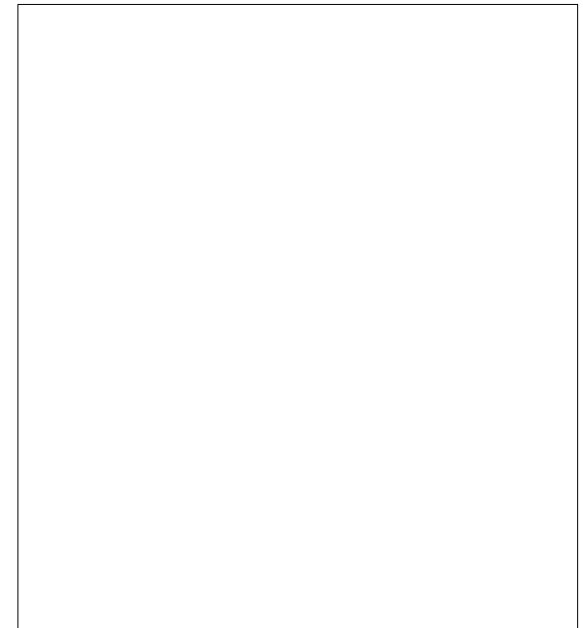
Code:

Procedure

```
00000000: 1A EntryProc      001F,0000,0008
00000007: 04 PushAdrVarMain  0004
0000000A: 09 GetVal
0000000B: 04 PushAdrVarMain  0000
0000000E: 01 PushValVarMain  0004
00000011: 06 PushConst      0001
00000014: 0E Mul
00000015: 06 PushConst      0000
00000018: 0C Add
00000019: 07 StoreVal
0000001A: 01 PushValVarMain  0000

0000001D: 08 PutVal
0000001E: 17 ReturnProc
Const 0000:0003
Const 0001:0010
```

```
const c=3;
var a,b;
begin
  ?b;
  a:=B*10+c;
  !a
end
.
```



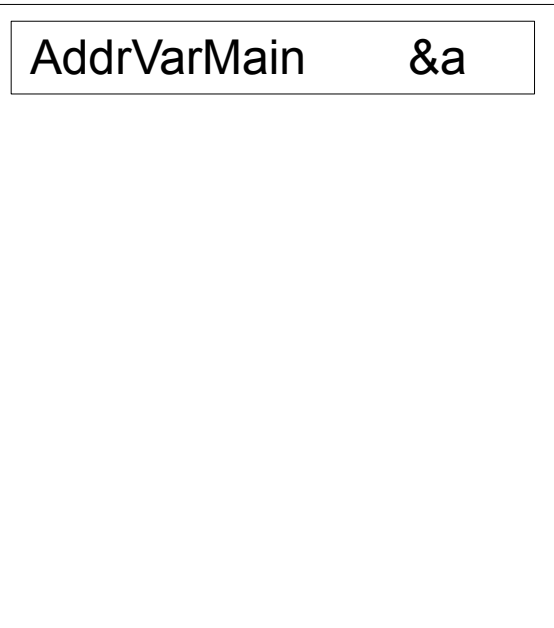
Code:

Procedure

```
00000000: 1A EntryProc      001F,0000,0008
00000007: 04 PushAdrVarMain  0004
0000000A: 09 GetVal
0000000B: 04 PushAdrVarMain  0000
0000000E: 01 PushValVarMain  0004
00000011: 06 PushConst       0001
00000014: 0E Mul
00000015: 06 PushConst       0000
00000018: 0C Add
00000019: 07 StoreVal
0000001A: 01 PushValVarMain  0000

0000001D: 08 PutVal
0000001E: 17 ReturnProc
Const 0000:0003
Const 0001:0010
```

```
const c=3;
var a,b;
begin
  ?b;
  a:=B*10+c;
  !a
end
.
```



Code:

Procedure

```
00000000: 1A EntryProc      001F,0000,0008
00000007: 04 PushAdrVarMain  0004
0000000A: 09 GetVal
0000000B: 04 PushAdrVarMain  0000
0000000E: 01 PushValVarMain  0004
00000011: 06 PushConst      0001
00000014: 0E Mul
00000015: 06 PushConst      0000
00000018: 0C Add
00000019: 07 StoreVal
0000001A: 01 PushValVarMain  0000

0000001D: 08 PutVal
0000001E: 17 ReturnProc
Const 0000:0003
Const 0001:0010
```

```
const c=3;
var a,b;
begin
  ?b;
  a:=B*10+c;
  !a
end
.
```

AddrVarMain	&a
ValVarMain	b

Code:

Procedure

```
00000000: 1A EntryProc      001F,0000,0008
00000007: 04 PushAdrVarMain  0004
0000000A: 09 GetVal
0000000B: 04 PushAdrVarMain  0000
0000000E: 01 PushValVarMain  0004
00000011: 06 PushConst      0001
00000014: 0E Mul
00000015: 06 PushConst      0000
00000018: 0C Add
00000019: 07 StoreVal
0000001A: 01 PushValVarMain  0000

0000001D: 08 PutVal
0000001E: 17 ReturnProc
Const 0000:0003
Const 0001:0010
```

```
const c=3;
var a,b;
begin
  ?b;
  a:=B*10+c;
  !a
end
.
```

AddrVarMain	&a
ValVarMain	b
Const	10

Code:

Procedure

```
00000000: 1A EntryProc      001F,0000,0008
00000007: 04 PushAdrVarMain  0004
0000000A: 09 GetVal
0000000B: 04 PushAdrVarMain  0000
0000000E: 01 PushValVarMain  0004
00000011: 06 PushConst      0001
00000014: 0E Mul
00000015: 06 PushConst      0000
00000018: 0C Add
00000019: 07 StoreVal
0000001A: 01 PushValVarMain  0000

0000001D: 08 PutVal
0000001E: 17 ReturnProc
Const 0000:0003
Const 0001:0010
```

```
const c=3;
var a,b;
begin
  ?b;
  a:=B*10+c;
  !a
end
.
```

AddrVarMain	&a
ValVarMain	b

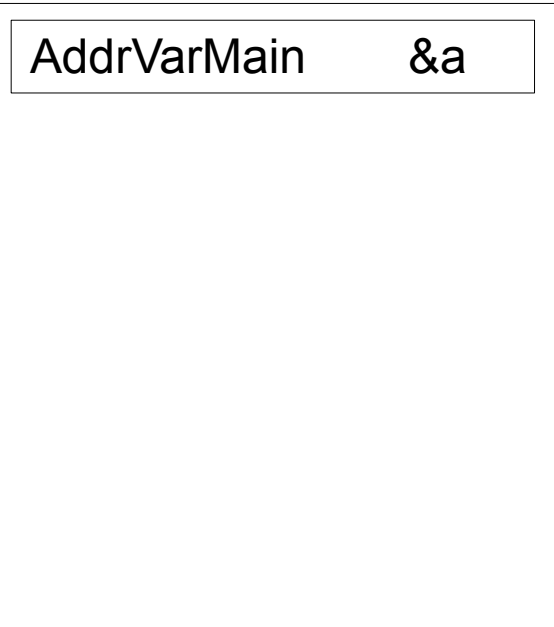
Code:

Procedure

```
00000000: 1A EntryProc      001F,0000,0008
00000007: 04 PushAdrVarMain  0004
0000000A: 09 GetVal
0000000B: 04 PushAdrVarMain  0000
0000000E: 01 PushValVarMain  0004
00000011: 06 PushConst      0001
00000014: 0E Mul
00000015: 06 PushConst      0000
00000018: 0C Add
00000019: 07 StoreVal
0000001A: 01 PushValVarMain  0000

0000001D: 08 PutVal
0000001E: 17 ReturnProc
Const 0000:0003
Const 0001:0010
```

```
const c=3;
var a,b;
begin
  ?b;
  a:=B*10+c;
  !a
end
.
```



Code:

Procedure

```
00000000: 1A EntryProc      001F,0000,0008
00000007: 04 PushAdrVarMain  0004
0000000A: 09 GetVal
0000000B: 04 PushAdrVarMain  0000
0000000E: 01 PushValVarMain  0004
00000011: 06 PushConst       0001
00000014: 0E Mul
00000015: 06 PushConst       0000
00000018: 0C Add
00000019: 07 StoreVal
0000001A: 01 PushValVarMain  0000

0000001D: 08 PutVal
0000001E: 17 ReturnProc
Const 0000:0003
Const 0001:0010
```

```
const c=3;
var a,b;
begin
  ?b;
  a:=B*10+c;
  !a
end
.
```

AddrVarMain	&a
Produkt B*10	



Code:

Procedure

```
00000000: 1A EntryProc      001F,0000,0008
00000007: 04 PushAdrVarMain  0004
0000000A: 09 GetVal
0000000B: 04 PushAdrVarMain  0000
0000000E: 01 PushValVarMain  0004
00000011: 06 PushConst      0001
00000014: 0E Mul
00000015: 06 PushConst      0000
00000018: 0C Add
00000019: 07 StoreVal
0000001A: 01 PushValVarMain  0000

0000001D: 08 PutVal
0000001E: 17 ReturnProc
Const 0000:0003
Const 0001:0010
```

```
const c=3;
var a,b;
begin
  ?b;
  a:=B*10+c;
  !a
end
.
```

AddrVarMain	&a
Produkt B*10	
Const	3

Code:

Procedure

```
00000000: 1A EntryProc      001F,0000,0008
00000007: 04 PushAdrVarMain  0004
0000000A: 09 GetVal
0000000B: 04 PushAdrVarMain  0000
0000000E: 01 PushValVarMain  0004
00000011: 06 PushConst      0001
00000014: 0E Mul
00000015: 06 PushConst      0000
00000018: 0C Add
00000019: 07 StoreVal
0000001A: 01 PushValVarMain  0000

0000001D: 08 PutVal
0000001E: 17 ReturnProc
Const 0000:0003
Const 0001:0010
```

```
const c=3;
var a,b;
begin
  ?b;
  a:=B*10+c;
  !a
end
.
```

AddrVarMain	&a
Produkt B*10	

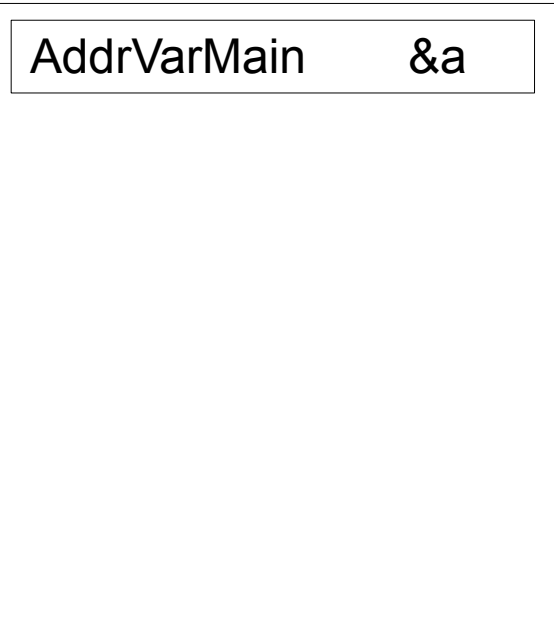
Code:

Procedure

```
00000000: 1A EntryProc      001F,0000,0008
00000007: 04 PushAdrVarMain  0004
0000000A: 09 GetVal
0000000B: 04 PushAdrVarMain  0000
0000000E: 01 PushValVarMain  0004
00000011: 06 PushConst      0001
00000014: 0E Mul
00000015: 06 PushConst      0000
00000018: 0C Add
00000019: 07 StoreVal
0000001A: 01 PushValVarMain  0000

0000001D: 08 PutVal
0000001E: 17 ReturnProc
Const 0000:0003
Const 0001:0010
```

```
const c=3;
var a,b;
begin
  ?b;
  a:=B*10+c;
  !a
end
.
```

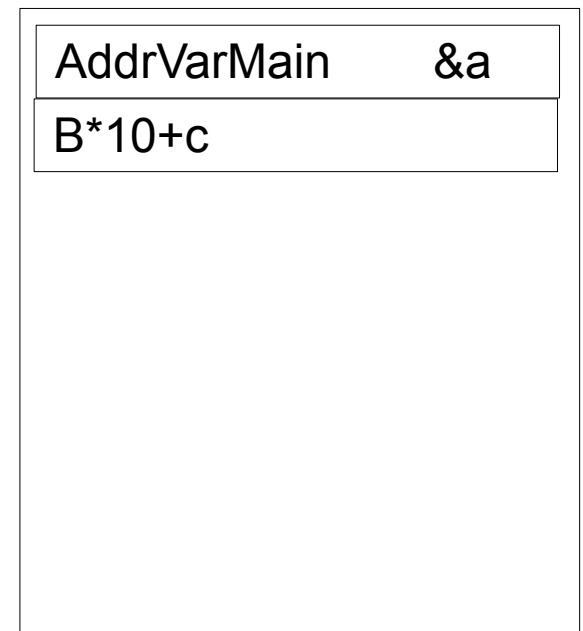


Code:

Procedure

```
00000000: 1A EntryProc      001F,0000,0008
00000007: 04 PushAdrVarMain  0004
0000000A: 09 GetVal
0000000B: 04 PushAdrVarMain  0000
0000000E: 01 PushValVarMain  0004
00000011: 06 PushConst      0001
00000014: 0E Mul
00000015: 06 PushConst      0000
00000018: 0C Add
00000019: 07 StoreVal
0000001A: 01 PushValVarMain  0000
0000001D: 08 PutVal
0000001E: 17 ReturnProc
Const 0000:0003
Const 0001:0010
```

```
const c=3;
var a,b;
begin
  ?b;
  a:=B*10+c;
  !a
end
.
```



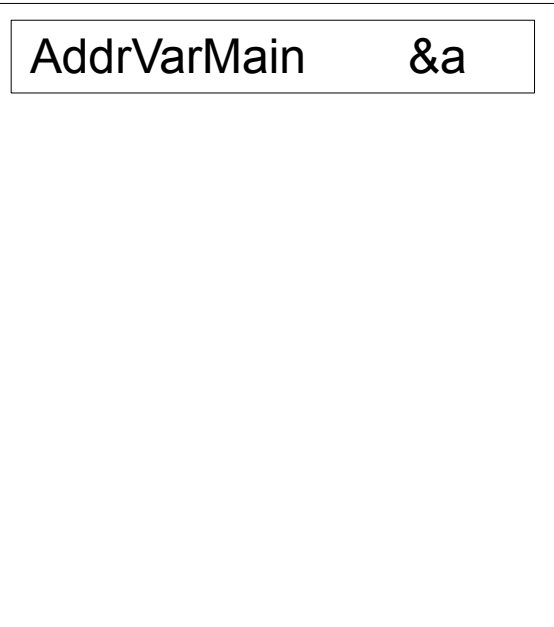
Code:

Procedure

```
00000000: 1A EntryProc      001F,0000,0008
00000007: 04 PushAdrVarMain  0004
0000000A: 09 GetVal
0000000B: 04 PushAdrVarMain  0000
0000000E: 01 PushValVarMain  0004
00000011: 06 PushConst      0001
00000014: 0E Mul
00000015: 06 PushConst      0000
00000018: 0C Add
00000019: 07 StoreVal
0000001A: 01 PushValVarMain  0000

0000001D: 08 PutVal
0000001E: 17 ReturnProc
Const 0000:0003
Const 0001:0010
```

```
const c=3;
var a,b;
begin
  ?b;
  a:=B*10+c;
  !a
end
.
```



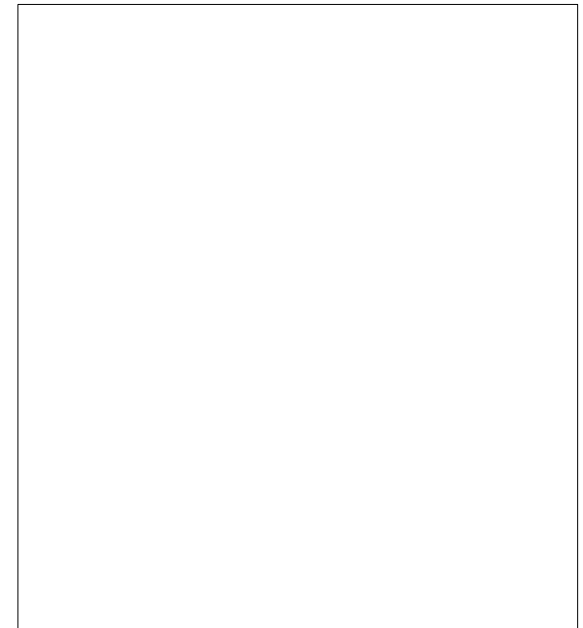
Code:

Procedure

```
00000000: 1A EntryProc      001F,0000,0008
00000007: 04 PushAdrVarMain  0004
0000000A: 09 GetVal
0000000B: 04 PushAdrVarMain  0000
0000000E: 01 PushValVarMain  0004
00000011: 06 PushConst      0001
00000014: 0E Mul
00000015: 06 PushConst      0000
00000018: 0C Add
00000019: 07 StoreVal
0000001A: 01 PushValVarMain  0000

0000001D: 08 PutVal
0000001E: 17 ReturnProc
Const 0000:0003
Const 0001:0010
```

```
const c=3;
var a,b;
begin
  ?b;
  a:=B*10+c;
  !a
end
.
```



# Datenstrukturen der VM

- Speicherbereich für den Stack: Wird beim Start angelegt, während der Programmabarbeitung überwacht und bei Bedarf vergrößert.
- Speicherbereich für den Code und die Konstanten: Wird beim Laden des Zwischencodefiles angelegt.
- Prozedurtabelle: wird nach dem Laden des Zwischencodefiles angelegt. Sie enthält die Anfangsadressen der Prozeduren sowie zunächst leere Felder für die Anfangsadressen der Variablenbereiche (Stackframes der Prozeduren), diese Einträge werden durch `entryProc` beim Aufruf der Prozedur belegt.
- Die Einträge der Prozedurtabelle sind nach Prozedurnummer aufsteigend geordnet, so dass die Prozedurtabelle durch die Prozedurnummer indiziert werden kann.

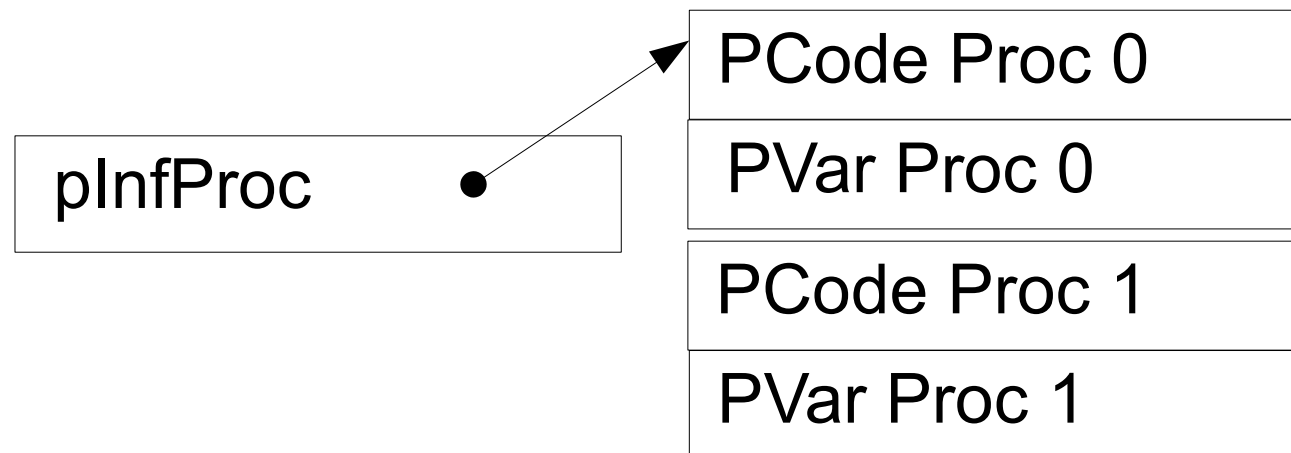
# Register der VM

<i>Register</i>	<i>Variable der VM</i>
ProgrammCounter	pC
StackPointer	pS
KonstantenPointer	pConst
Aktuelle Prozedur	iCProc
Pointer auf Prozedurtable	pInfProc

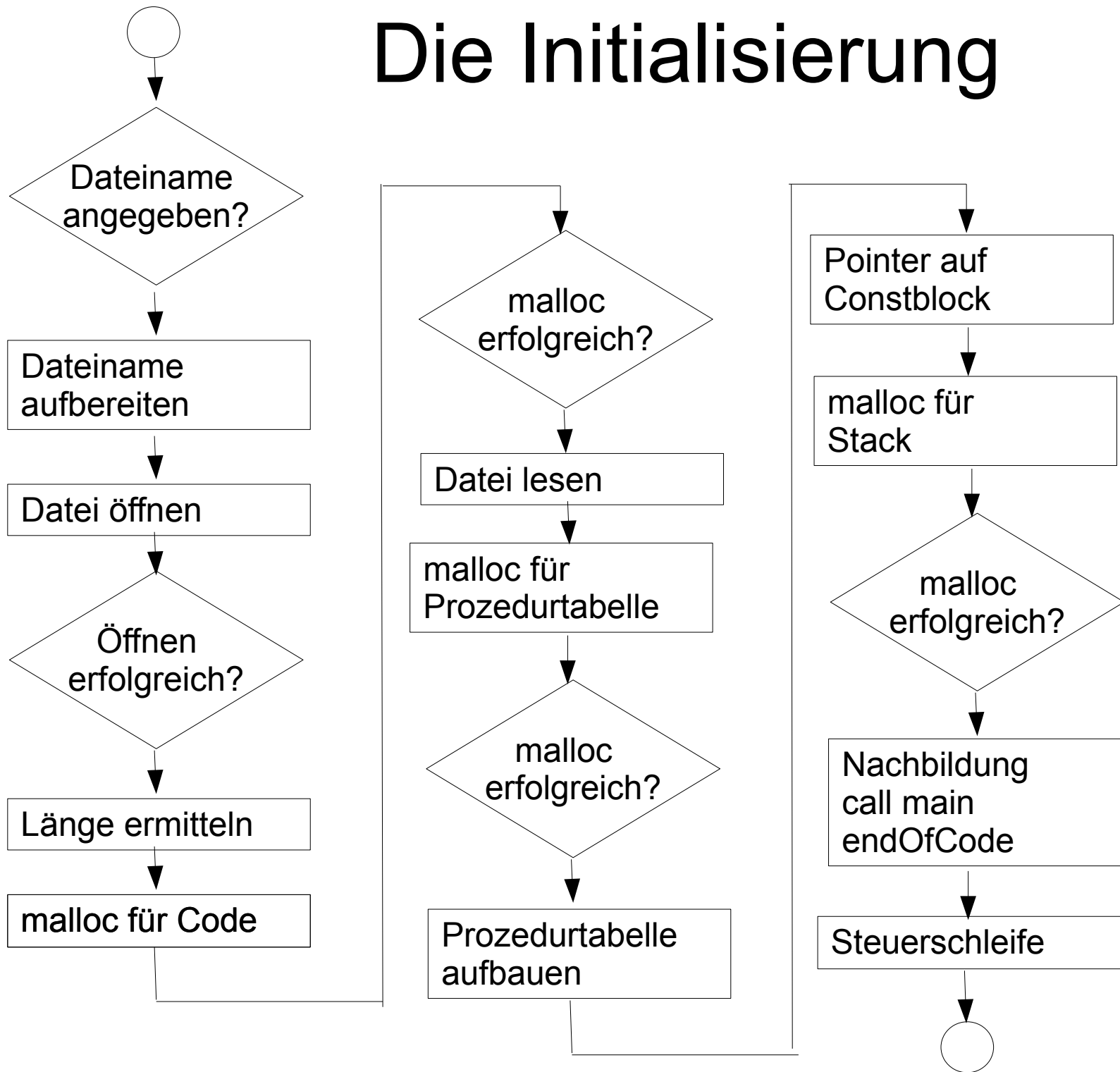


# Prozedurtabelle

- Enthält für jede Prozedur 2 Einträge (Pointer)
  - Startadresse des Codes (Adresse von EntryProc der Prozedur), wird beim Programmladen eingetragen
  - Adresse des Variablenbereiches der Prozedur (Adresse der Variablen mit Offset 0), wird bei Prozeduraufruf zur Laufzeit eingetragen
- Tabelle wird durch die Prozedurnummer indiziert

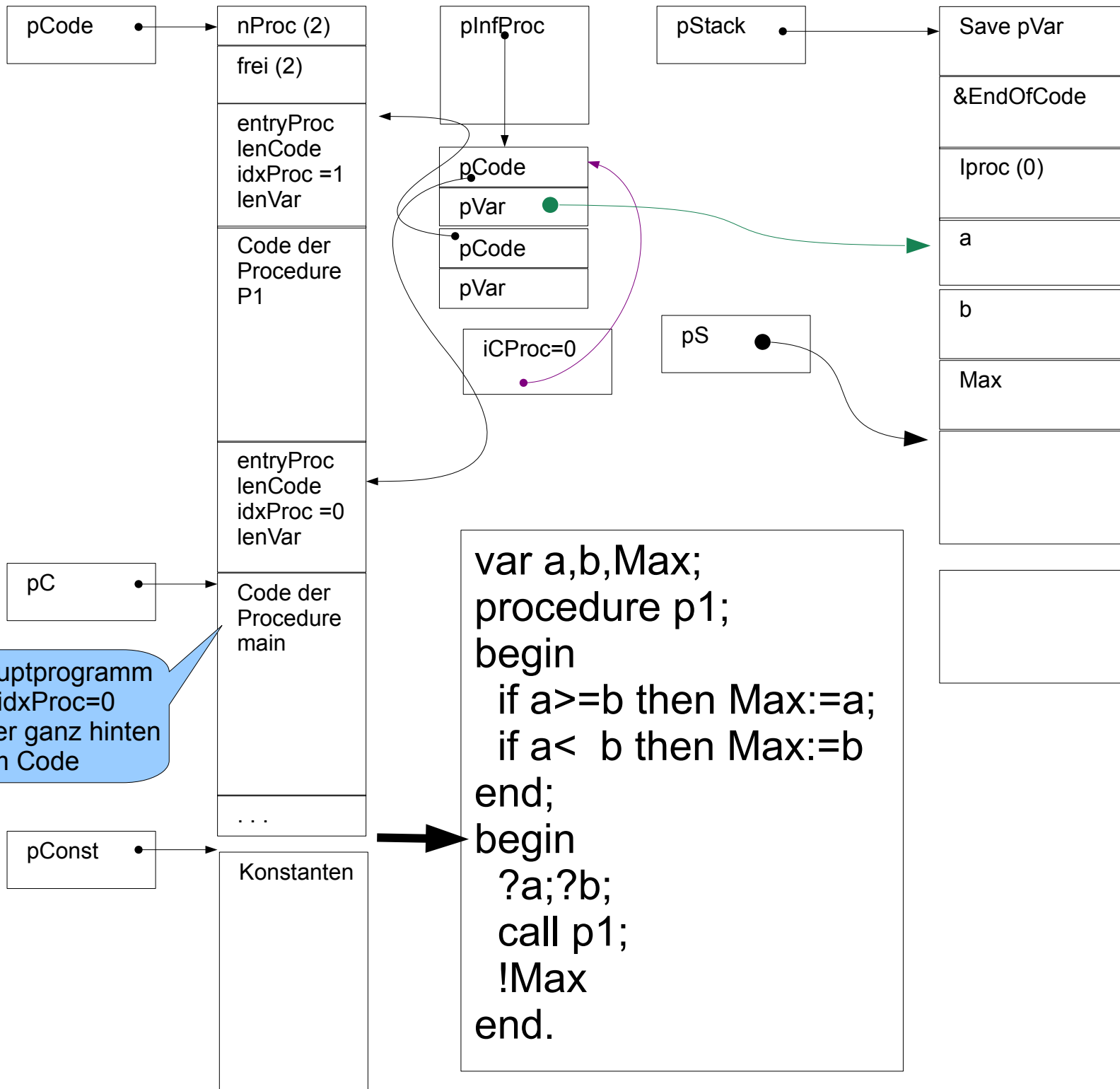


# Die Initialisierung



# Datenstruktur der virtuellen Maschine

- Die nachfolgende Folie zeigt für das angegebene Beispiel zum Zeitpunkt des Pfeils die Datenstruktur während der Programmausführung.
- Für die Variablen a, b, und Max ist Speicherplatz im Stack reserviert.
- Der Stackpointer pS zeigt hinter die Variablen a, b, und Max des Hauptprogramms.



```
typedef int (*fx) (void);
```

```
fx vx[]={  
    FcpuValVrLocl, FcpuValVrMain,  
    FcpuValVrGlob, FcpuAdrVrLocl,  
    FcpuAdrVrMain, FcpuAdrVrGlob,  
    FcpuConst,    FcstoreVal,  
    FcputVal,     FcgetVal,  
    FcvzMinus,    Fcodd,  
    FcOpAdd,      FcOpSub,  
    FcOpMult,     FcOpDiv,  
    FccmpEQ,      FccmpNE,  
    FccmpLT,      FccmpGT,  
    FccmpLE,      FccmpGE,  
    Fccall,       FcRetProc,  
    Fcjmp,        Fcjnot,  
    FcEntryProc, FcputStrg,  
    Fpop,         Fswap,  
    FcEndOfCode, Fcput,  
    Fcget  
};
```

# Die Steuerschleife I

```
while (!Ende)  
{  
    vx[*pC++] ();  
}
```



Setzt Ende auf true

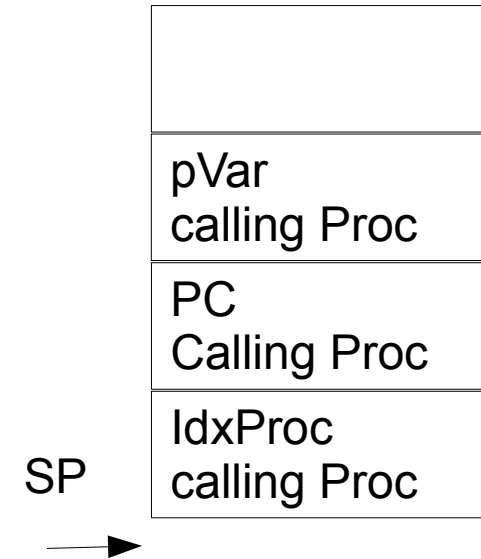
# Steuerschleife II

```
while (!Ende)
{
    switch (*pC++)
    {
        case puValVrLocl:FcpuValVrLocl(); break;
        case puValVrMain:FcpuValVrMain(); break;
        case puValVrGlob:FcpuValVrGlob(); break;
        case puAdrVrLocl:FcpuAdrVrLocl(); break;
        case puAdrVrMain:FcpuAdrVrMain(); break;
        case puAdrVrGlob:FcpuAdrVrGlob(); break;
        case puConst      :FcpuConst();      break;
        case storeVal     :FcstoreVal();     break;
        case putVal       :FcputVal();       break;
        case getVal       :FcgetVal();       break;
        case vzMinus      :FcvzMinus();      break;
        case odd          :Fcodd();          break;
        case OpAdd        :FcOpAdd();        break;
        case OpSub        :FcOpSub();        break;
        case OpMult       :FcOpMult();       break;
        case OpDiv        :FcOpDiv();        break;
        . . .
        case pop          :Fpop();           break;
        case swap         :Fswap();         break;
        case EndOfCode    :FcEndOfCode();    break;
        case put          :Fcput();         break;
        case get          :Fcget();         break;
    }
}
```

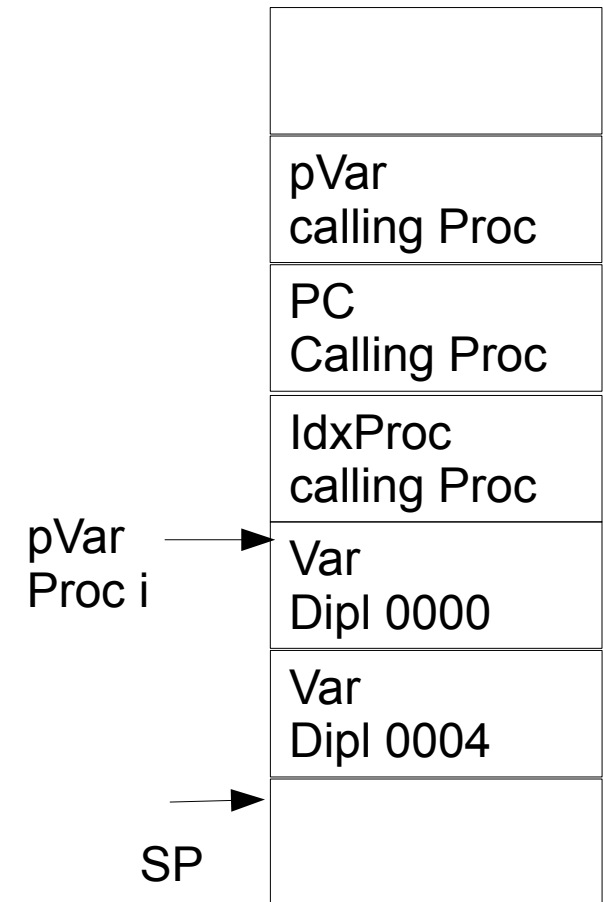
# Prozeduraufruf

- Call PrNr

- Kellern des Zeigers (relativ) auf Variablenbereich der rufenden Prozedur aus der Prozedurtabelle.
- Kellern des aktuellen Befehlszählers (zeigt jetzt hinter call)
- Kellern der Prozedurnr. der rufenden Procedure
- Ausführen des Sprunges durch Eintragen der Adresse des Befehls entyProc der aufzurufenden Prozedur aus der Prozedurtabelle in den Programmcounter.



- EntryProc
  - Setzen der aktuellen Prozedur
  - Einrichten des Variablenbereiches im Stack
  - Setzen des Zeigers auf Variablenbereich in der Prozedurtabelle
- Die Verwaltung der Variablenbereiche über die Prozedurtabelle ermöglicht den rekursiven Prozeduraufruf, wobei immer der richtige Satz von Variablen benutzt wird.





# Beispiel Rekursion

Rekursion mit globaler Variable

```
var x;  
  procedure r;  
  var a;  
  procedure p;  
  !a;  
  begin  
    if x<5 then  
    begin  
      x:=x+1;  
      a:=x;  
      call p;  
      call r;  
      call p  
    end  
  end;  
begin  
x:=0;  
call r  
end.
```

```
$ r64 rec  
1  
2  
3  
4  
5  
5  
4  
3  
2  
1
```

Berechnung der Fakultät

```
var a, fac;  
Procedure p1;  
  var b, c;  
  begin  
    b := a;  
    a := a-1;  
    c := a;  
    !c;  
    if c>1 then call p1;  
    fac := fac*b;  
    !fac  
  end;  
begin  
  ?a;  
  fac := 1;  
  call p1;  
  !fac  
end.
```

# Fakultät mit Parameter

(Parameter sind in PL0 eigentlich nicht vorgesehen)

```
var x,y;
  procedure facult(a);
  begin
    if a-1>1 then call facult(a-1);
    y:=y*a
  end;
begin
  ?x;
  y:=1;
  call facult(x);
  !y;
  !"\n"
end.
```