

Genetics-IT

Entwurf einer esoterischen Programmiersprache



**Stand:** 2.5.22  
**Autor:** Zoran Vranešević

**Das Projekt im Internet:**<https://github.com/Zoltan-X/Genetics-IT>

[Vorwort: 2](#_Toc102341199)

[Einleitung: Gene Basics 2](#_Toc102341200)

[Definitionen für ein Minimum an Informationsverarbeitung. 3](#_Toc102341201)

[Technischer Ansatz 3](#_Toc102341202)

[Grundlegendes*:* 3](#_Toc102341203)

[Aminosäuren und eine frei gewählte mögliche Zuordnung als Codeelemente 3](#_Toc102341204)

[Adressieren von Registern, Variablen und Direktwerten: 3](#_Toc102341205)

[Informationen müssen verglichen werden: 4](#_Toc102341206)

[Informationen müssen verändert werden können: 4](#_Toc102341207)

[Flussrichtungen über die Bits der Register 4](#_Toc102341208)

[Definition vom Spin: 4](#_Toc102341209)

[Definition vom call self (unechte Rekursion): 4](#_Toc102341210)

[Definition von Defines (Initialisierung) 4](#_Toc102341211)

[Definition von in Beispiele verwendeter Sprachen Elemente: 5](#_Toc102341212)

[Methoden Defintion with defines Section: 5](#_Toc102341213)

[Defines DSL operator: 5](#_Toc102341214)

[Value assignments: 5](#_Toc102341215)

[Defaults: 5](#_Toc102341216)

[if (conditional Code Branching): 5](#_Toc102341217)

[recursive Loop: 5](#_Toc102341218)

[different elements: 5](#_Toc102341219)

[Beispiele: 6](#_Toc102341220)

[Minimum Information processing Examples 6](#_Toc102341221)

[Definition Increment 6](#_Toc102341222)

[Definition Decrement 7](#_Toc102341223)

[Definition CompareLower 8](#_Toc102341224)

[Definitions of Compare Equal 9](#_Toc102341225)

[Definitions of Compares 10](#_Toc102341226)

[Definition a negation 11](#_Toc102341227)

## Vorwort:

Dies ist eine Esoterische Programmiersprache, welche sich ursprünglich an den Möglichkeiten der Genetik (DNS/RNA) orientierte. Hierbei waren Polymorphe Rekursionen in der Natur der Dinge im Vordergrund, ebenso wie diese Sprache vollständig mit einem absurden Minimum an Befehlen tauglich zu bekommen. Ferner wäre die Flip-Flop Logik höchst geeignet für FPGA Programmierung. Am Anfang ging es hierbei um eine Abbildung von einer Programmiersprache zur Genetik im Sinne der D/RNA. Da die Ausbildung von Proteinen, sowie die RNA Strang Verbiegung und das Liquid Bonding (Bindung von Elementen aus einer Flüssigkeit, wie Fruchtwasser und ähnlichem) für digitale Programmierung doch zu untauglich waren, blieb dann nur noch die Idee, das Kodieren mit nur 4 Befehlen entsprechend der Aminosäure Paarungen Vollumfänglich nutzbar zu entwickeln. Damit sollte es theoretisch in D/RNA abbildbar werden. Abgesehen von der Adressierung von Bitweisen Register­werten ist es dann gelungen mit 4 Befehlen auszukommen. Jenseits dessen waren Hochsprachen Bedürfnisse damit nicht zu befriedigen. Deshalb haben dezente Anpassungen Einzug gehalten, welche aber einerseits bei der Übersetzung zu Maschinencode nur einem steuernden Einfluss dienen und andererseits die Esoterik dahinter hochsprachentypisch stark reduzieren.

## Einleitung: Gene Basics

Gene haben folgende Aminosäuren:

* (C) Cytosin
* (G) Guanin
* (A) Adenosin
* (T) Thymin (DNA only)
* (U) Uracil (RNA only)

Diese haben folgende 4 Paarungen:

* CG
* GC
* AT (DNA only)
* AU (RNA only)
* TA (DNA only)
* UA (RNA only)

Daher mit Respekt für die esoterische Programmiersprache „Brainfuck“ ist GeneticsIT noch nieder­schwelliger, aber durch die Hochsprachenvorteile auch viel einfacher. GenteticsIT arbeitet mit Bit Flipping in Registern. Dies ist entlehnt an die Grundlagen der Digitaltechnik - den Flip Flops.   
Flip Flops sind Grundlage aller Logischen Schaltungen wie „logisches und“, „logisches oder“, „einfaches negieren“, „exklusivem und“ und „exlusivem oder“. Damit sind Flip-Flops die niederste Ebene für alle Arten von Prozessoren. Des Weiteren sind Register­verarbeitungs­richtungen von LowEndian zu BigEndian wechselbar und das Adressieren der einzelnen Bits geschieht reihum entsprechend der Big- oder LowEndian Weise und der Spin Richtungsangabe.

## Definitionen für ein Minimum an Informationsverarbeitung.

### Technischer Ansatz

##### **Grundlegendes:**

Informationen oder Ressourcen müssen adressiert werden können.

* 1.) Adressieren von Registern, Variablen und Direktwerten A := 123, A := B
* 2.) Informationen müssen verglichen werden. If A equ B then Branch
* 3.) Informationen müssen verändert werden können. FlipBit(A(BitIndex))
* 4.) Flussrichtungen über die Bits der Register Spin(L oder R)
* ++) Defines Big- & Low- Endian, Bit-Index,  
   Locals, auto\_default Werte, …

Damit haben wir ein Maximum von 4 Befehlen plus Hochsprachen Komfort. Weitere Hochsprachen Möglich­keiten wie Schleifen in der Codeflusssteuerung werden über Bedingte Rekursionen (Selbstaufrufe) erreicht und der Index des Register­bits wird automatisiert bei jeder Rekursion um eine Stelle in Flussrichtung fortgeführt.   
Rekursion ist in diesem Zusammenhang hier ein erneuter Selbstaufruf bei dem der Funktionsstack nicht mit neuen Parametern belastet wird, da die lokalen Variablen global innerhalb der obersten Ebene des Aufrufs einer solchen Methode bleiben. Das bedeutet im Fazit der Rekursive Selbstaufruf ist eigentlich keiner, sondern eine Schleife aus dem Methodenkörper, an den Anfang des Methodenkörpers. Auch ist eine defines Section eingeflossen, bei der es darum geht Werte und Einstellungen vorab lokal zu definieren, um dann Werte aus der Schnittstelle zu reduzieren, ebenso wie Werte zum Adressieren bereitzustellen oder initial vorzubelegen. Die Steuerung des Verhaltens durch beispielsweise Big-Endian oder Low-Endian Angaben Beeinflussen zum Beispiel die Auswirk­ung von Spins. Ferner kann der BitIndex im Register als auto\_default definiert werden womit die Bits im Register gezielt angesteuert werden können. Hierbei werden die Indizes Reihum iteriert. Am Ende endet die Methode bei Selbstaufruf durch Überschreitung der Registerbreite.   
Somit folgt eine Auflistung und mögliche Zuordnung von Befehlen basierend auf dem Konzept von vier Aminosäurepaarungen als Codeelemente.

##### Aminosäuren und eine frei gewählte mögliche Zuordnung als Codeelemente

Amino Amino-Paarung Program Code DNA RNA   
C CG (Flip) Bind Bind  
G GC (compare EQU) Loose Loose  
A AT (DNA) (Spin-R) Sequence Start ----  
T TA (DNA) (Spin-L) Sequence End ----  
A AU(RNA) (Spin-R) ---- Bend inwards   
U UA(RNA) (Spin-L) ---- Bend outwards

##### Adressieren von Registern, Variablen und Direktwerten:

An dieser Stelle müssen wir eine Hochsprachenlogik voraussetzen, da wir mit nur einem Befehl logisch nur ein Register ansprechen können. In diesem Zusammenhang betrachten wir Variablen als eigene Dimension und jede Variable als anzusprechendes Register. Auch Direktwerte wären nach genetischem Vorbild nicht händelbar, werden hierbei aber ebenso wie Variablen als Registerwert behandelt werden.

##### Informationen müssen verglichen werden:

Da wir nur eine universellen Vergleichsart anstreben müssen wir uns mit einem „Compare Equal“ vergleich begnügen und da heraus die anderen Vergleichsformen ableiten. Aus der Negierung, welche wir bequem mit Inversion eines Registers erreichen können, leiten wir schließlich den Not Opperator ab. Da wir im Digitalen Business zwar negieren aber beim Programmieren im Rahmen einer boolschen Algebra die Wahrheitswerte verneinen.

##### Informationen müssen verändert werden können:

Dieses ist hier vom Konzeptionellen in niederster Ebene nur durch Bit Flips zu bewerkstelligen.

##### Flussrichtungen über die Bits der Register

Mit der Spin-R/Spin-L Wird die Flussrichtung über die Bits eines Registers bestimmt bzw. umgekehrt.  
Im genetischen Ansatz würde es sich dabei um ein Sequenz Anfang- und Ende-Konstrukt handeln. Hierbei ist zu unterscheiden, ob es sich um Big- oder Low-Endian Verhaltensweisen des Registers handelt. Über eine vorherbestimmte Flussrichtung kann auf solche Details Rücksicht genommen werden.

##### Definition vom Spin:

Wenn wir multidimensionalen code Bedenken, so haben wir über Spin-L und Spin-R folgende Dimensionen der Anwendungsfälle:

* Programmablaufflusssteuerung
  + if thens
  + if then else
* Register Flussrichtung <BigEndian|LowEndian>
* BitIndex Adressierung

Ebenso müssen wir bei der Adressierung von Variablen bedenken das wir sie als eigenständige Dimension betrachten, wo wir sie wie Register adressieren.

##### Definition vom call self (unechte Rekursion):

Wenn wir diese speziellen Rekursionen durch Selbstaufruf als Schleifenlogik verstehen wollen, so müssen wir definieren, wie dieser passiert:

1. Register Bit Indizes werden entsprechend dem Spin um einen erhöht oder vermindert
2. Der Selbstaufruf macht einen Rücksprung an den Funktionsanfang
3. Variablen werden nicht übergeben und bleiben lokal erhalten
4. Es entsteht kein iterieren der Aufrufs Kette im Sinne einer Rekursion,   
   wenn eine iterierte Instanz in der Rekursion endet, dann enden alle Instanzen.

##### Definition von Defines (Initialisierung)

Mit den Defines werden:

* Parameter der Schnittstelle aus dieser gezogen.
* Initiale Werte definieren
* DSL Operatoren definieren
  + Unäre
  + Binäre
  + Ternäre
* Methoden Aliase

## Definition von in Beispiele verwendeter Sprachen Elemente:

### Methoden Defintion with defines Section:

Define <name>(<Params>)   
 Defines  
 <local variables>   
 <constants>  
 <Bit-Index>:=<Flow-Directions>   
 End-Defines  
 <Code-Body>  
End-Define

### Defines DSL operator:

Unary:   
 Defines <method name>:=<alias>   
Prefix:   
 Defines <ParamA[\_]>,<method name >:=<alias>   
postfix:   
 Defines <method name>,<ParamA[\_]>:=<alias>   
binary:   
 Defines <ParamA|placeholder[\_]>,<method name>,<ParamB[\_]>:=<alias>   
ternary:   
 Defines <ParamA[\_]>,<operatorA>,<ParamB[\_]>,<operatorb>,<ParamC[\_]>:=<alias>

### Value assignments:

<Variable name, Constant name> := <Immediate Value, Variable, defaults, auto\_default>

### Defaults:

default(<value|range|list>)  
auto\_default(<Bitindex Range|list>

### if (conditional Code Branching):

if <True/False> <codeExpression>   
if <True/False>   
 <code block>   
End-If  
if <True/False>   
 <code block>   
else  
 <code block>   
End-If

### recursive Loop:

call self

### different elements:

<a> equ <b>   
return <Value>   
flip(<variable,[Register Bit Index)>)   
flip(<[single Bit Register|Boolean]>))

## Beispiele:

### Minimum Information processing Examples

#### *Definition In*c*rement*

*##########################################################################################  
#  
# Definition of an incrementation process with   
# automatic abort of recursion if the processing flow overflows the Registerbits range.   
#  
# Go through memory register and flip first Zero Bit.   
# Flip the Flow Direction and flip all bits returning Flow Direction   
#  
# Function Params:   
# Information => Limited Set of Bits,   
# FlowDirection => Sequential Information processing Flow Direction   
# L-Spin or R-Spin   
#  
# Defines:   
# BitIndex => 1 -> n /\*x as Limes value\*/ and   
# n -> 1 /\*as opposit Limes value\*/   
# originalSpin => local Constant only to be set on initial function call   
#   
##########################################################################################*

Definition Increment( Information,   
 FlowDirection := default(L<-R),   
 )   
 defines //initial Values Definition   
 originalSpin := FlowDirection   
 BitIndex := auto\_default(1..bit\_length(Information)),   
 End-defines

// As Long as no Register overflow occured and Flow Direction unchanged   
 if BitIndex equ overflow return Information   
 if FlowDirection equ originalSpin   
 if Information(BitIndex) equ 0   
 Flip(Information,BitIndex)   
 Flip(FlowDirection)  ENDIF   
 call self  
 ENDIF

// After change/swap Flow-Direction invert/flip every single Bit   
 Flip(Information, BitIndex)   
 call self

End-Definition

*/\*   
  
Increment(0, default, default, default)*

*BitIndex Range L<-R 4 3 2 1 0   
BitIndex Range L->R 0 1 2 3 4*

*Run\Iterations => \ First Second Third Fourth Fifth  
 Run*

*1 = 0 0 0, 0 0 1, End  
 2 = 0 0 1, 0 1 1, 0 1 0, End   
 3 = 0 1 0, 0 1 1, End  
 4 = 0 1 1, 1 1 1, 1 0 1, 1 0 0, End  
 5 = 1 0 0, 1 0 1, End  
 6 = 1 0 1, 1 1 1, 1 1 0, End  
 7 = 1 1 0, 1 1 1, End  
 8 = 1 1 1, overflow, End  
\*/*

#### *Definition De*c*rement*

*##########################################################################################  
#  
# Definition of an decrementation process with   
# automatic abort of recursion if the processing flow reaches its Limit.   
#  
# Go through memory register and flip first non-Zero Bit.   
# Flip FlowDirection and flip all bits at returning Flow-Direction   
#  
# Function Params:   
# Information => Limited Set of Bits,   
# FlowDirection => Sequential Information processing Flow Direction   
# L-Spin or R-Spin   
#  
# Defines:   
# BitIndex => 1 -> n /\*x as Limes value\*/ and   
# n -> 1 /\*as opposit Limes value\*/   
# originalSpin => local Constant only to be changed on initial function call   
#   
##########################################################################################*

Definition Decrement( Information,   
 FlowDirection := default(L<-R),   
 )   
 defines  
 originalSpin := FlowDirection   
 BitIndex := auto\_default(0..bit\_length(Information)),   
 End-defines

// As Long as no Register overflow occured and Flow Direction unchanged   
 if BitIndex equ overflow return Information  if FlowDirection equ originalSpin  if Information(BitIndex) equ 1   
 Flip(Information,BitIndex)   
 Flip(FlowDirection)  ENDIF   
 call self   
 ENDIF

// After change/swap Flow-Direction invert/flip every single Bit   
 Flip(Information, BitIndex)   
 call self

End-Definition

*/\**Decrement*(0, auto\_default, default, default)*

*BitIndex Range L<-R 4 3 2 1 0   
BitIndex Range L->R 0 1 2 3 4   
   
Run\Iterations => \ First Second Third Fourth Fifth  
 Run  
 1 = 1 1 1, 1 1 0, End  
 2 = 1 1 0, 1 0 0, 1 0 1, End  
 3 = 1 0 1, 1 0 0, End  
 4 = 1 0 0, 0 0 0, 0 1 0, 0 1 1, End  
 5 = 0 1 1, 0 1 0, End  
 6 = 0 1 0, 0 0 0, 0 0 1, End  
 7 = 0 0 1, 0 0 0, End   
 8 = 0 0 0, Overflow, End  
   
\*/*

#### *Definition CompareLower*

*##########################################################################################  
#  
# Definition of Compare Lower Logic  
#  
# Function Params:   
# a => comparand,   
# b => comparator  
#  
# Defines:   
# BitIndex => Index for bits in registers   
#  
##########################################################################################*  
Definition CompareLower(   
 a,   
 b,   
)   
 Definition \_\_CompareLower(a, b)  
 defines  
 BitIndex := auto\_default(bit\_length(a)..0),   
 End-defines

if a(BitIndex) equ b(BitIndex)   
 call self  
 End-If  
   
 // match?  
 if b(BitIndex) equ 1 return True   
  
 //not matched yet?, then its false!  
 False  
  
 End-Definition

if a Equal b return False  
 \_\_CompareLower(a, b)

End-Definition

*/\*   
 Compare if a lower b   
 CompareLower(a, b)   
  
case (100, 100) = 0 (False)   
Run (A) (B)  
1 100 = 100 //End with returning False*

*case (101, 100) = 0 (False)   
Run (A) (B)  
1 101 = 100 //continue   
1 1 = 1 //continue  
3 0 = 0 //continue  
4 1 = 0 // End with return False*

*case (100, 101) = 1 (True)   
Run (A) (B)  
1 100 = 100 //continue   
1 1 = 1 //continue  
3 0 = 0 //continue  
4 1 = 0 // End with return True*

*\*/*

#### *Definitions of Compare Equal*

##########################################################################################  
#  
# Definitions for Bitwise Equal compare Operator   
#  
# *variables:*  
# a => comparand,   
# b => comparator  
#  
# *defines:*  
# ZeroIndex => auto Index for bits in BitIndex Register with (Decrement)   
# BitIndex => auto Index for bits in registers with (Decrement)   
#  
# Spin == LowEndian   
#  
##########################################################################################  
Definition \_\_Equal(a, b)  
 defines   
 Spin := default(LowEndian)   
 BitIndex := auto\_default(bit\_length(a)..0)  
 False := 0  
 True := 1  
 End-Defines  
   
 private   
 //iterates in Lowendian direction through a,b and   
 //finds first different Bits  
 Definition \_\_firstDiffBit  
 if a(BitIndex) Equ b(BitIndex) call self  
 bitIndex  
 End-Defintion  
   
 End-Private  
 if \_\_firstDiffBit equ 0 return True  
 False

End-Defintion  
  
defines: \_, Equal, \_ = \_\_Equal

*/\*   
 Compare Equal   
 a Equal b [True/False]   
  
case 100 Equal 100) = 0 (True)   
Run (A) (B) ZeroIndex Result  
1 100 100 - True  
2 101 100 1 False  
3 100 110 2 False*  
*4 000 111 3 False*

*\*/*

#### *Definitions of Compares*

##########################################################################################  
#  
# Definitions of atomic compares defined on top of Bitwise Equ and Expressionwise Equal  
#  
# *variables:*  
# a => comparand,   
# b => comparator  
# Condition => condition  
#  
# *defines:*  
# BitIndex => auto Index for bits in registers with (Decrement)   
# Spin => LowEndian   
#  
##########################################################################################  
Definition Compare( a, b, Condition=[”LOW”, ”LEQ”, ”EQU”, “GRE”, “GRE”])  
  
 private  
  
 defines   
 BitIndex := auto\_default(bit\_length(a)..1)  
 Spin := default(LowEndian)  
 End-Defines  
  
 Definition \_\_firstDiffBit  
 if a(BitIndex) Equ b(BitIndex)   
 call self  
 bitIndex  
 End-Defintion  
  
 Definition \_\_CompareLowerLowerEqual  
 if Condition Equal “LOW” return True  
 if Condition Equal “LEQ” return True  
 End-Defintion  
   
 Definition \_\_CompareGreaterGreaterEqual  
 if Condition Equal “GRE” return True  
 if Condition Equal “GEQ” return True  
 End-Defintion  
  
 Definition \_\_CompareLowerGreaterEqualEqual  
 if Condition Equal “LEQ” return True  
 if Condition Equal “EQU” return True  
 if Condition Equal “GEQ” return True  
 End-Defintion  
   
 End-Private  
  
 \_\_firstDiffBit(a, b, BitIndex)  
  
 // match?   
 if BitIndex Equal 0 \_\_CompareLowerGreaterEqualEqual   
 if a(BitIndex) Equ 1 \_\_CompareGreaterGreaterEqual  
 if b(BitIndex) Equ 1 \_\_CompareLowerLowerEqual  
   
 //not matched yet? Then it’s false!  
 False  
  
End-Definition   
  
defines: \_, Lower, \_ = Compare( a, b, “LOW”)  
defines: \_, LowerEqual, \_ = Compare( a, b, ”LEQ”)  
//defines: \_, Equal, \_ = Compare( a, b, ”EQU”) //already Defined  
defines: \_, GreaterEqual, \_ = Compare( a, b, “GEQ”)  
defines: \_, Greater, \_ = Compare( a, b, “GRE”)

#### *Definition a negation*

*##########################################################################################  
#  
# Definition of a register bits negation process.   
#  
# Function Params:   
# Information => Limited Set of Bits,   
#  
# Defines:   
# BitIndex => 1 -> n /\*x as Limes value\*/ and   
# n -> 1 /\*as opposit Limes value\*/   
#   
##########################################################################################*

Definition Neg(Information)   
 defines  
 BitIndex := auto\_default(1..bit\_length(Information)), )  
 End-defines

Flip(Information,BitIndex)  
 call self

End-Definition

defines: Neg, \_ = Not(a)   
  
/\*  
 Information 01010100  
 Negation 10101011  
\*/