halec
Herrnröther Str. 54
63303 Dreieich

www.halec.de

roloBasic

Dokumentenversion 1.0 vom 2011-06-26

Inhaltsverzeichnis

I Der Einstieg in roloBasic

In Kapitel I wird ein knapper Überblick über die Möglichkeiten von roloBasic gegeben. Kapitel II erklärt die Funktionsweise der Sprache detailliert. Kapitel III ist eine Referenz für Systemfunktionen und die Syntax der Sprache.

1 Was ist roloBasic?

roloBasic ist eine einfache Programmiersprache, die auf vielen Mikrocontrollern autark lauffähig ist. Ein PC wird nicht zwingend benötigt, um einen Mikrocontroller mit roloBasic zu programmieren. Über den Kommandointerpreter kann ein Mikrocontroller direkt gesteuert und programmiert werden. roloBasic-Quelltext kann direkt in den Mikrocontroller geladen und ausgeführt werden.

roloBasic verwendet einen Bytecode-Compiler. Die übersetzten Programme werden in einer virtuellen Maschine (VM) ausgeführt. Diese Architektur ermöglicht, roloBasic je nach Anwendung auf sehr unterschiedliche Art einzusetzen. Bei Anwendungen, für die Autarkie nicht relevant ist, kann im Mikrocontroller auf den Compiler verzichtet werden. Der Compiler kann dann z.B. auf einem PC eingesetzt werden, so daß auf dem Mikrocontroller nur noch die VM benötigt wird.

roloBasic ist sehr genügsam. Ungefähr 32 kB Flash und 1 kB RAM genügen schon, um roloBasic auf einem Mikrocontroller nutzen zu können. Wird nur die VM verwendet, kann der Ressourcenbedarf noch weiter reduziert werden.

roloBasic ist weitgehend konfigurierbar, so daß sehr unterschiedliche Produkte sehr spezifisch mit roloBasic unterstützt werden können. Beispielsweise kann roloBasic lediglich zur flexiblen Konfiguration eines Produktes verwendet werden. Im Gegensatz dazu kann jedoch auch eine Mikrocontroller-Anwendung vollständig in roloBasic implementiert werden. Sehr unterschiedliche Szenarien sind möglich.

roloBasic kann auch in Form eines Kommandointerpreters genutzt werden. Dies ist die roloBasic-Shell. Hier kann man zusammenhängende Anweisungen eingeben, diese werden dann direkt nach der Eingabe übersetzt und sofort ausgeführt.

2 Herausragende Spracheigenschaften

roloBasic verzichtet auf komplizierte Sprachkonstrukte und ist in kurzer Zeit erlernbar.

```
! Hello, world! in roloBasic:
print "Hello, world!"
```

2.1 Speicherverwaltung

Die automatische Speicherverwaltung ermöglicht komfortables Programmieren. Arrays und Zeichenketten können jederzeit angelegt, gelöscht, vergrößert und verkleinert werden, solange genug freier Speicher vorhanden ist. Um große Datenelemente zu löschen, genügt es, sie zu "vergessen" -- z.B. indem einer Variablen eine 0 zugewiesen wird.

```
a = dim(int, 7)   ! Int-Array mit 7 Elementen. Index: 0-6
resize a, 12      ! Vergrößere das Array auf 12 Elemente
print size(a)     ! Gibt die Arraygröße aus
a = 0             ! Array löschen
```

2.2 Ausnahmebehandlung (Exceptions)

Die Ausnahmebehandlung erlaubt, übersichtlich zu programmieren, ohne auf Fehlerbehandlung zu verzichten.

```
print a[index]      ! Ein Array-Element ausgeben
index = index + 1
print a[index]      ! Ein zweites Array-Element ausgeben

catch ex        ! Mögliche Exception fangen und in ex speichern
                ! Wenn ex=0, wurde keine Ausnahme geworfen
if ex=rangeCheckError
  print "Bad Index: ", index, " Array size is: ", size(a)
endif
```

II Beschreibung der Sprache

Um ernsthaft in roloBasic zu programmieren, sollte dieses Kapitel vollständig durchgearbeitet werden.

1 Allgemeines

roloBasic ist eine zeilenorientierte Sprache. Die meisten Anweisungen werden durch ein Zeilenende abgeschlossen. Innerhalb von Kontrollstrukturen werden Zeilenumbrüche fest vorgeschrieben.

Zeilen können durch Verwendung des Unterstriches ("_") auf mehrere Zeilen verlängert werden:

```
print "Dies ist eine besonders lange Zeile, die auf " + _
  "zwei Zeilen aufgeteilt wurde."
```

Um mehrere Anweisungen in eine Zeile packen, kann der Doppelpunkt (":") verwendet werden. Der Compiler faßt diesen als Zeilenende auf:

```
if isArray(a) : print size(a) : endif
```

Ein Ausrufezeichen signalisiert, daß der Rest einer Zeile ein Kommentar ist und somit vom Basic-Compiler ignoriert wird. Innerhalb eines Kommentares werden auch Doppelpunkte ignoriert.

```
print "Hallo!"  !Kommentar: Es ist schlechter Stil, hier das
                 !offensichtliche zu schreiben.
```

roloBasic ist "case insensitive". Das bedeutet: Es ist egal, ob Schlüsselworte und Variablennamen mit Groß- oder Kleinbuchstaben geschrieben werden -- die Bedeutung bleibt dieselbe. In den Quelltextbeispielen verwenden wir der besseren Schreib- und Lesbarkeit halber Kleinschreibung mit CamelCase.

2 Einzeilige Anweisungen

Ein roloBasic-Programm ist eine Folge von Anweisungen. Die einfachsten Anweisungen sind Prozeduraufrufe und Zuweisungen.

2.1 Systemfunktionen

Über Systemfunktionen werden Ein- und Ausgabe sowie spezielle Eigenschaften und Fähigkeiten eines mikrocontrollergesteuerten Geräts aufgerufen. Deshalb sind viele Systemfunktionen nur bei bestimmten Produkten verfügbar.

Die wichtigsten Systemfunktionen sind in jeden System vorhanden. Dies ist die roloBasic-Standardumgebung. In einem eigenen Kapitel wird sie vollständig dokumentiert. Viele Funktionen der Standardumgebung werden zusätzlich im Rahmen ihres Einsatzgebietes beschrieben.

Produktspezifische Systemfunktionen werden nicht in dieser Anleitung beschrieben. Bitte konsultieren Sie hierfür die Bedienungsanleitung des jeweiligen Produkts.

Beispiel für eine Systemfunktion der Standardumgebung:

```
print <wert> [, <wert 2> [, ...]]

Beispiel:
=========
print "Die Antwort auf die ultimative Frage: ", 42
```

Diese Systemfunktion gibt einen oder mehrere Werte über die Standardausgabe aus. Die Standardausgabe kann je nach Produkt etwas sehr Unterschiedliches sein: Z.B. ein Display, eine Logdatei, eine serielle Schnittstelle ... oder etwas völlig anderes.

2.2 Variablen und Zuweisungen

Die wichtigste roloBasic-Anweisung ist die Zuweisung. Hierfür wird das Gleichheitszeichen ("=") verwendet:

```
<ziel> = <ausdruck>
```

Das Zuweisungsziel wird meistens eine Variable sein. roloBasic verwendet ein dynamisches Typsystem. Das bedeutet: Einer Variablen kann jeder beliebige Wert zugewiesen werden.

Hierbei ist wichtig zu verstehen: Bei einer Zuweisung wird einem Wert ein Name zugeordnet. Der Wert wird dabei *nicht* kopiert!

Variablen müssen nicht deklariert werden. Der roloBasic-Compiler merkt automatisch, wenn eine neue Variable auftaucht, und verwaltet diese selbständig. Variablen werden vom Compiler stets mit dem Wert 0 initialisiert:

```
print neuerName ! Hier wird die Variable "neuerName" neu
                ! angelegt und sogleich ausgegeben.
                ! Somit wird der Wert 0 ausgegeben.
```

2.3 Ausdrücke und Operatoren

Ein Ausdruck ist in roloBasic alles, was bei einer Zuweisung rechts vom Gleichheitszeichen stehen darf. roloBasic kennt folgende Arten von Audrücken:

```
a = "Hallo"       ! Literal
b = a             ! Variable
a = b + " Welt"   ! Anwendung eines Operators
b = size(a)       ! Funktionsaufruf
b = (b+1)*2       ! Geklammerter Ausdruck
```

Die wichtigsten Operatoren sind in der Standardumgebung enthalten.

3 Typen und Werte

Normalerweise muß man sich in roloBasic kaum um Typen kümmern, da diese automatisch verwaltet werden. Doch auch wenn es nicht unmittelbar auffällt: Jeder Wert hat stets einen eindeutigen Typ. In roloBasic werden Typen nicht Variablen, sondern Werten zugeordnet.

3.1 Basistypen

roloBasic kennt folgende Basistypen:

- char: Wertebereich 0 ... 255
- int: Wertebereich -32768 ... 32767
- long: Wertebereich -2147483648 ... 2147483647

Automatische Typzuordnung

Diese Typen werden bei normaler Arithmetik automatisch zugeordnet. So gilt z.B:

```
type(7) = char
type(-7) = int
type(40 * 1000) = long
```

Die Auswahl des Typs geschieht automatisch und weitgehend unsichtbar. Für das Ergebnis einer mathematischen Operation wird immer der kleinstmögliche Typ ausgewählt.

Literale

roloBasic kennt grundsätzlich nur positive numerische Literale. Um negative Zahlen darzustellen, muß der "-"-Operator verwendet werden.

Die ganzen Zahlen von 0 bis 2147483647 sind als zusammenhängende Dezimalzahlen zu schreiben. Hexadezimalzahlen werden mit dem "`$`"-Zeichen gekennzeichnet.

```
a = -2147483647 - 1   ! Weist a den Wert -2147483648 zu
b = $1ff              ! Weist b den Wert 511 zu
```

roloBasic verwendet den Typ `char` sowohl für Zeichen als auch für positive Zahlen im Bereich von 0 bis 255. Literale für einzelne Zeichen werden mit einfachen Hochkommata eingeschlossen. Somit gilt:

```
'A' = 65
```

Weiterhin gibt es folgende Escape-Sequenzen für Zeichenliterale:

```
'\\'   : Backslash (\)
'\''   : Einfaches Hochkomma
'\"'   : Anführungszeichen
'\n'   : LF (Zeilenvorschub)
'\r'   : CR (Wagenrücklauf)
'\t'   : Tabulator
'\b'   : Backspace
'\f'   : Seitenvorschub
'\0'   : 0 (Null)
'\x??' : hexadezimaler Zeichencode (? steht für je eine Ziffer)
'\d???': dezimaler Zeichencode
```

Nicht unterstützte Basistypen

Um roloBasic klein und sparsam zu halten, wurde das Typmodell bewußt einfach gestaltet. Es wurden nicht alle "normalen" Basistypen implementiert, die man von anderen Programmiersprachen her kennt.

Die Standardumgebung unterstützt keine Fließkommazahlen. Da viele Mikrocontroller keine native Unterstützung für Fließkommazahlen bieten, müßte auf diesen hierfür eine umfangreiche Software-Implementierung eingebunden werden -- dies ist jedoch in vielen Fällen kontraproduktiv, da somit roloBasic deutlich umfangreicher würde und für Projekte auf sehr kleinen Mikrocontrollern oft nicht mehr genutzt werden könnte.

roloBasic kennt keinen "Boolean"-Typ. Wahrheitswerte werden durch Zahlenwerte codiert. Der Wert 0 wird als "false", alle anderen Werte als "true" interpretiert. Operatoren, die einen Wahrheitswert berechnen, geben ausschließlich entweder 0 (als "false") oder 1 (als "true") zurück.

roloBasic kennt keinen Typ, um Typen darzustellen. Die Systemfunktion `type(<wert>` gibt einen Zahlenwert zurück, der einen Typ codiert. Die Systemkonstanten `char`, `int`, `long` und `vari` spiegeln diese Werte wider.

3.2 Arrays

Die Speicherverwaltung von roloBasic erlaubt einen bequemen Umgang mit Arrays. Ein Array ist für roloBasic ein Wert und kann genau wie Zahlenwerte an beliebige Variablen zugewiesen oder an Funktionen übergeben werden. Im Gegensatz zu Zahlenwerten ist ein Array jedoch veränderbar. Die Systemfunktion `dim` erzeugt Arrays und gibt diese als Rückgabewert zurück.

Veränderbarkeit

Es ist vielleicht auf den ersten Blick nicht offensichtlich, daß ein Zahlenwert nicht veränderbar ist. Doch wenn einer Variablen ein neuer Wert zugewiesen wird, bedeutet dies nicht, daß der alte Wert verändert wird, sondern daß er komplett ersetzt wird. Ein Array hingegen kann verändert werden, ohne vollständig ersetzt zu werden.

```
a = dim(int, 10)   ! Array für 10 int-Elemente erzeugen
```

```
a[0] = 30000        ! Veränderung 1: Wert in Array schreiben
resize a, 20        ! Veränderung 2: Array vergrößern auf 20

! Es gilt: size(a) = 20
!           isArray(a) = 1    ! (1 bedeutet: "true")
!           type(a) = int
!           mutable(a) = 1    ! ("true")
```

Arrays, die nicht mit der Systemfunktion `dim` erzeugt wurden, haben oftmals den Typzusatz "immutable". Diese Eigenschaft kann durch die Systemfunktion `mutable` abgefragt werden ("immutable" bedeutet: nicht mutable). Immutable Arrays können nicht mehr verändert werden. Der Vorteil ist, daß ein solches Array konfliktfrei von verschiedenen Parteien für verschiedene Zwecke verwendet werden kann.

Elementtypen

Arrays haben immer einen festen Elementtyp, der nicht nachträglich verändert werden kann. Dieser bestimmt, welche Art von Werten das Array aufnehmen kann -- und somit auch, wieviel Speicher das Array pro Element benötigt. Wird einem Array-Element ein Wert zugewiesen, den dieses nicht aufnehmen kann, so wird die Exception `typeFault` geworfen. Der Elementtyp eines Arrays kann mit `type(<array>)` abgefragt werden. Ob ein Wert ein Array ist oder nicht, kann mit `isArray(<Wert>)` ermittelt werden.

Größe und Indexbereich

Der Indexbereich eines Arrays fängt stets mit 0 an. Der letzte gültige Index für einen Zugriff auf ein Array `a` ist stets: `size(a)-1`. Ein Zugriff auf einen ungültigen Array-Index führt zur Exception `illegalArrayIndex`.

Bei der Erzeugung eines Arrays mit `dim` muß zwar eine Größe angegeben werden. Diese kann jedoch nachträglich mit `resize` verändert werden. Ein Array kann auch die Größe 0 annehmen und somit kein einziges Element enthalten. Die maximale Größe, die ein Array annehmen kann, hängt nur davon ab, wieviel Speicher frei ist.

Zuweisung ist nicht Kopie

Wenn einer Variablen ein Array zugewiesen wird, wird es nicht kopiert, sondern die Variable als neuer (zusätzlicher) Name für das Array eingesetzt. Um eine echte Kopie eines Arrays zu machen, kann die Systemfunktion `copy` verwendet werden.

```
a = dim(int, 10) ! Ein int-Array der Größe 10 erzeugen
b = a            ! b und a sind nun Namen für dasselbe Array
b[0] = 77
print a[0]       ! gibt 77 aus
b = copy(a)      ! eine Kopie von a in Variable b ablegen
b[0] = 42
print a[0]       ! gibt immer noch 77 aus.
```

Literale und Strings

roloBasic hat keinen eigenen Typ für Strings (Zeichenketten). Stattdessen werden einfach char-Arrays verwendet. Ein String-Literal ist somit eigentlich ein Literal für (immutable) char-Arrays:

```
s = "Hello roloBasic!"
! Es gilt: isArray(s) = 1 !(true)
!          type(s) = char
!          mutable(s) = 0 !(false)
!          s[0] = "H"
```

In String-Literalen können dieselben Escape-Sequenzen wie bei Zeichenliteralen verwendet werden.

Der "+"-Operator kann in roloBasic auch mit Arrays verwendet werden. Er erzeugt dann ein neues Array mit dem aneinandergehängten Inhalt der beiden anderen Arrays. Somit kann man Strings sehr einfach mit dem "+"-Operator aneinanderhängen. Dies ist allerding nicht besonders effizient. Besser ist es, die Systemfunktion `append` zu verwenden, um Strings (bzw. Arrays) aneinanderzuhängen. Um einzelne Zeichen (bzw. Elemente) anzuhängen, kann `push` verwendet werden.

Veränderbare Kopien erzeugen

Hinweis: Die Systemfunktion `copy` erzeugt immer mutable Arrays. Wenn man also eine modifizierte Version eines immutable Arrays benötigt, kann man es erst mit copy kopieren und dann die Kopie verändern.

Schließlich gibt es in roloBasic Array-Konstruktoren. Diese sind einfacher zu demonstrieren als zu erklären. Sie können mit beliebig vielen Argumenten aufgerufen werden:

```
a = int(0,1,2,3) ! erzeugt ein immutable Int-Array der Länge 4
                 ! mit den Einträgen 0,1,2,3
b = copy(long()) ! Ein leeres mutable Long-Array erzeugen
```

Hinweis: Der Compiler versucht, Berechnungen (wie z.B. die Erzeugung eines Arrays oder auch das Zusammenfügen von Arrays mit dem "+"-Operator) schon zur Übersetzungszeit durchzuführen. Wenn ihm das gelingt, wird das erzeugte Array immutable sein und im Bytecode als Konstante gehandhabt. Das passiert z.B., wenn alle Parameter konstant und immutable sind. Der Vorteil hiervon ist Speicherersparnis. Der Nachteil ist, daß man beim Programmieren ein wenig aufpassen muß. Der effizienteste Weg, ein vorinitialisiertes mutable Array zu erzeugen ist, die `copy`-Funktion zu verwenden. Ein weniger effizienter Weg: Falls in einem Ausdruck eine Variable vorkommt, wird dieser grundsätzlich erst zur Laufzeit berechnet, und das Ergebnis wird mutable.

```
a = copy(char(0,1,2,3,4,5,6,7)) ! Effiziente Art, ein mutable
                                ! Char-Array zu initialisieren

c = 0                           ! Ineffiziente Art, ein mutable
b = char(c,1,2,3,4,5,6,7)       ! Char-Array zu initialisieren
```

3.3 Vari-Arrays, Strukturen

In roloBasic gibt es noch einen weiteren, speziellen Array-Typ, dessen Elemente jede Art von Werten aufnehmen können -- auch Arrays.

Dieser Typ heißt `vari`.

Vari-Arrays werden in roloBasic anstelle von Strukturen verwendet. Daher ist immer, wenn von einer Struktur die Rede ist, ein vari-Array gemeint. Mit vari-Arrays können auf einfache Weise komplexe, verschachtelte Datenstrukturen erzeugt werden:

```
a = dim(vari,0)
push a, 4215                ! Das Array a mit dem push-
push a, "Königswasser"      ! Kommando Stück für Stück
push a, int(1,2,3)          ! mit verschiedenen Werten füllen
```

Auch vari-Arrays können per Konstruktor erzeugt werden:

```
a = vari("Ein", "paar", "Strings", dim(char,2))
```

```
! nun gilt: a[2] = "Strings"
!           a[2][0] = 'S'
!           mutable(a) = 1 ! (mutable aufgrund des dim-Aufrufs)
```

Ein vari-Array-Element funktioniert wie eine Variable. Auch hier ist eine Zuweisung keine Kopie. Somit können leicht verkettete oder sogar zyklische Strukturen erzeugt werden:

```
a = copy(vari(1,0))
b = copy(vari(2,a))
a[1] = b
! nun gilt z.B.: a[1][1][0] = 1
!                 a[1][1][1][0] = 2
```

3.4 Äquivalenz und Identität

Die Systemfunktion `same` prüft zwei Werte auf Identität. Dies ist ein sehr strenges Kriterium. Zwar sind zwei gleiche Zahlenwerte immer identisch. Ein Array ist jedoch immer nur mit sich selbst identisch. Zwei individuell angelegte Arrays sind immer unterschiedlich, auch wenn sie dieselben Werte enthalten.

Der Vergleichsoperator "=" überprüft, ob zwei Werte äquivalent sind. Dies ist ein weniger strenges Kriterium. Zwei Arrays gelten als äquivalent, wenn sie dieselbe Länge haben, und die enthaltenen Werte paarweise identisch sind. Dies gilt auch dann, wenn der Typ der Arrays unterschiedlich ist, so kann z.B. ein int-Array mit einem long-Array äquivalent sein.

```
! Für Zahlenwerte gilt immer: Gleiche Zahlen sind identisch.
! z.B. same(10,10) = 1

a = vari(1,2)
b = int(1,2)
c = a

! Es gilt:  same(a,b) = 0   ! a und b sind nicht identisch,
!           a = b           ! aber äquivalent!
!           same(a,c) = 1   ! a und c sind Namen für dasselbe
!                           ! Array, somit gilt same(a,c) = 1

a1 = vari(a,3)  ! Wir erzeugen nun zwei gleich aufgebaute
b1 = vari(b,3)  ! verschachtelte Strukturen.

! Nun gilt:
!   a1 <> b1  ! a1 und b1 sind weder äquivalent noch identisch
```

Dies ist vielleicht auf den ersten Blick überraschend. Doch die Elemente von `a1` und `b1` sind nicht identisch, daher können `a1` und `b1` nicht als äquivalent betrachtet werden -- denn `a1` ist von `b1` nur durch Vergleich der Werte mittels `same` unterscheidbar:

```
! Offensichtlich gilt: same(a1[0],a) = 1
!                       same(b1[0],a) = 0
! also sind a1 und b1 nicht äquivalent!
```

4 Kontrollstrukturen

roloBasic hat nur wenige Kontrollstrukturen. Auf die aus anderen Basic-Dialekten bekannte Goto-Anweisung wurde verzichtet.

4.1 Verzweigung

Verzweigungen werden mit den Schlüsselwörter `if`, `endif`, `else` und `elseif` formuliert. Direkt hinter `if` und `elseif` wird erst ein Ausdruck, dann ein Zeilenumbruch erwartet. Nach `else` und `endif` wird ebenfalls ein Zeilenumbruch erwartet. Beispiel:

```
if isArray(x) or isArray(y)
  print "x oder y ist ein Array."
elseif x>y
  print x, " ist größer als ", y
elseif x=y
  print x, " ist gleich ", y
else
  print x, " ist kleiner als ", y
endif
```

4.2 Iteration

Iteration ist meistens deutlich effizienter als Rekursion.

For-Schleife

Die For-Schleife ist eine bequeme, aber nicht unbedingt die effizienteste Art der Iteration:

```
a = "Hello roloBasic!"
b = dim(char,0)

for i=0 to 4
  push b, a[i]+1
```

```
next

for i=size(a)-1 downto 0 step -2
  push b, a[i]
next

! Es gilt: b = "Ifmmp!iaoo le"
```

Wichtig zu wissen ist: Der End-Index (der Wert hinter `to`) sowie die Schrittweite (der Wert hinter `step`) werden bei jedem Durchlauf der Schleife neu ausgewertet.

Das Schlüsselwort `downto` kann verwendet werden, wenn die Schleifenvariable heruntergezählt werden soll. Bei einer Schrittweite ungleich 1 (bzw. ungleich -1 bei `downto`) muß noch das Schlüsselwort `step` und die Schrittweite folgen.

Do-Schleife

Die Endlosschleife sieht in roloBasic wie folgt aus:

```
do
  print "endlos..."
loop
```

Es kann jedoch auf das `do` ein `while` und ein Eintrittskriterium folgen. Auf das `loop` kann ein `until` und ein Abbruchkriterium folgen. Außerdem kann jede Schleife durch den `break`-Befehl abgebrochen werden.

```
! Eine Schleife mit 3 Ausgängen
do while y < x*x
  y = getnextvalue(y)
  if y = check
    break
  endif
loop until y < max
```

5 Prozeduren und Funktionen

Wird eine Befehlsfolge oder Berechnung häufig benötigt, sollte sie als Prozedur oder Funktion implementiert werden.

5.1 Prozeduren

Der Kopf einer Prozedurdefinition beginnt mit dem Schlüsselwort `procedure`. Danach folgt der Prozedurname, gefolgt von durch Kommata getrennten formalen Parametern. Der Prozedurkopf wird durch einen Zeilenumbruch abgeschlossen.

Die darauf folgenden Anweisungen bilden den Rumpf der Prozedur. Die Prozedur wird durch das Schlüsselwort `end` abgeschlossen.

```
procedure printarray a
  if isarray(a)
    print "("
    do
      printarray a[i] : i=i+1
      if i=size(a) : break : endif
      print ","
    loop
    print ")"
  else
    print a
  endif
end

printarray "Hallo" ! --> (72,97,108,108,112)
```

Alle Parameter sind Wert-Parameter. Sie verhalten sich so, als würde ihnen beim Prozeduraufruf der jeweils übergebene Wert zugewiesen.

Innerhalb einer Prozedur verwendete Variablen sind lokal und werden automatisch mit 0 initialisiert. Die Laufvariable `i` wird automatisch als lokale Variable angelegt, bei jedem Prozeduraufruf wird sie aufs Neue mit dem Wert 0 initialisiert.

Es ist auch möglich, innerhalb von Prozeduren auf globale Variablen zuzugreifen. Hierzu muß das Schlüsselwort `global` dem Variablennamen vorangestellt werden. Beispiel:

```
procedure log message
  append global logstring, message
end

log "Hello"
log ", "
log "roloBasic"
log "!"

! --> logstring = "Hello, roloBasic!"
```

Prozeduren werden selbst auch in Variablen gespeichert. Eine Prozedurdefinition legt eine Variable an, welche die Prozedur selbst in einem vari-Array speichert. Somit können Prozeduren kopiert, in Arrays gespeichert oder als Parameter an andere Prozeduren übergeben werden.

Hierbei sind einige wichtige Besonderheiten zu beachten:

- Für Prozeduraufrufe muß die Prozedur in einer Variablen gespeichert sein. Ein Aufruf direkt aus einem Array heraus ist nicht möglich. Man kann allerdings eine Prozedur aus einem Arrayelement in eine Variable kopieren und dann aufrufen.
- Während normale Variablenzugriffe innerhalb einer Prozedur standardmäßig auf lokale Variablen zugreifen, ist dies bei Prozeduraufrufen innerhalb einer Prozedur genau umgekehrt: Bei einem Prozeduraufruf wird grundsätzlich davon ausgegangen, daß die Prozedur in einer globalen Variablen gespeichert ist. Will man eine Prozedur aufrufen, die in einer lokalen Variablen gespeichert ist, muß das Schlüsselwort `local` dem Variablennamen vorangestellt werden.
- Beim Prozeduraufruf muß sich zwischen Prozedurnamen (bzw. Namen der Variablen, in der die Prozedur gespeichert ist) und dem ersten Parameter mindestens ein Leerzeichen befinden.

```
procedure test1
  print "hallo"
end

procedure test2 p
  proc = p[0]       ! Prozedur aus p[0] in lokale Var. kopieren
  local proc        ! Prozedur in lokaler Var. aufrufen
end

a = dim(vari, 1) ! Vari-Array mit einem Element anlegen
a[0] = test1     ! Prozedur test1 in Array a kopieren
test2 a          ! Aufruf von Prozedur test2 mit a
```

Eine Prozedur wird beendet, sobald der Rumpf vollständig durchlaufen wurde. Es ist jedoch auch möglich, die Prozedur vorzeitig mittels des Befehls `return` zu verlassen.

Eine Prozedur kennt keine direkten Rückgabewerte. Um Werte aus einer Prozedur zurückzugeben, können globale Variablen verwendet werden. Wenn Arrays als Parameter übergeben werden, können diese innerhalb der Prozedur verändert werden. Somit kann ein Array als Möglichkeit, mehrere Werte aus einer Prozedur zurückzugeben, verwendet werden.

5.2 Funktionen

Eine Funktion funktioniert fast genauso wie eine Prozedur. Der wichtigste Unterschied ist, daß eine Funktion immer einen Rückgabewert hat. Weiterhin wird die Parameterliste -- im Gegensatz zu Prozeduren -- immer von runden Klammern umschlossen.

Der Rückgabewert wird mittels des Befehls `return <Ausdruck>` zurückgegeben. Bei Funktionen muß -- im Gegensatz zu Prozeduren -- auf das Schlüsselwort `return` immer der Rückgabewert folgen. Befindet sich kein `return`-Befehl in einer Funktion, wird der Wert 0 zurückgegeben.

Funktionen können genauso wie Prozeduren in Variablen und Arrays kopiert werden. Eine Funktion kann -- im Gegensatz zu Prozeduren -- auch direkt als Array-Element aufgerufen werden.

```
function fib(x)
  if x<=0
    return 0
  elseif x=1
    return 1
  else
    return fib(x-2) + fib(x-1)
  endif
end

procedure test2 p
  print p[0](10) ! Aufruf der in p[0] enthaltenen Funktion
end

print fib(10)    ! Gibt 55 aus

a = dim(vari, 1) ! Vari-Array mit einem Element anlegen
a[0] = fib       ! Funktion fib in Array a kopieren
test2 a          ! Aufruf von Prozedur test2 mit a
                 ! --> Gibt ebenfalls 55 aus
```

6 Exceptions

Alle Laufzeitfehler lösen in roloBasic Exceptions aus, die mit dem `catch`-Befehl gefangen werden können. Wird eine Exception nicht gefangen, führt sie dazu, daß das Programm vorzeitig beendet wird und der Wert der Exception als Fehlermeldung ausgegeben wird.

Mit dem `throw`-Befehl können eigene Exceptions geworfen werden. Diese funktionieren genauso wie Exceptions des Systems. Man kann somit sogar Exceptions, die Systemfehler signalisieren, selbst erzeugen.

```
i = 5
a = 1000
do
  a = a div i
  i = i - 1
loop

catch ex
if ex = divisionByZero
  print a
else
  throw ex  ! Falls es eine andere Exception war, wird sie
            ! einfach wieder geworfen.
endif
```

Der Parameter des `catch`-Befehls muß eine Variable sein, dieser wird die gefangene Exception zugewiesen. Falls der `catch`-Befehl ohne Exception erreicht wird, wird der Variablen eine 0 zugewiesen. Somit wird die Fehlerbehandlung oftmals wie folgt aussehen:

```
catch ex
if ex
  ... Fehlerbehandlung ...
endif
```

Der Parameter des `throw`-Befehls darf ein beliebiger Ausdruck sein. Der von diesem Ausdruck berechnete Wert ist dann der Wert der geworfenen Exception. Prinzipiell dürfen auch Arrays geworfen werden. Werden diese nicht gefangen, wird das System jedoch nicht den gesamten Inhalt des Arrays als Fehlermeldung ausgeben.

III Die Standardumgebung

Die hier beschriebenen Operatoren, Prozeduren, Funktionen und Konstanten sind feste Bestandteile von roloBasic. Sie können wie normale (d.h. in roloBasic programmierte) Prozeduren oder Funktionen aufgerufen werden.

Ihre Namen sind reserviert: Es ist nicht möglich, diesen Namen andere Funktionen oder Prozeduren zuzuweisen. Außerdem können diese Systemfunktionen oder Systemprozeduren nicht in Variablen gespeichert werden.

1 Arithmetische Operatoren

Zwar wird intern nur wenig Speicher für kleine numerische Werte verwendet. Doch diese Automatik ist weitestgehend transparent -- man kann praktisch davon ausgehen, daß mit 32 Bit Genauigkeit gerechnet wird, also im Wertebereich -2147483648 ... 2147483647. Numerische Überläufe über diesen Bereich hinaus werden jedoch nicht erkannt. Somit führt z.B. die Rechnung `2147483647 + 1` zum Ergebnis -2147483648.

1.1 Addition

```
value = a + b
```

Addiert zwei Werte.

- **Parameter:** Zwei numerische Werte oder zwei Arrays desselben Elementtyps. Es muß gelten:
  ```
  (not isarray(a) and not isarray(b)) or _
  isarray(a) and isarray(b) and type(a)=type(b)
  ```
  Eine Verletzung dieser Bedingung führt zur Exception **typeFault**.
- **Rückgabewert:** Resultat der Addition. Waren die Parameter Arrays, so wird ein neues Array desselben Elementtyps erzeugt, welches die Konkatenation der beiden Parameter enthält.
  (`"hi"+"ho" = "hiho"`)

1.2 Subtraktion, Multiplikation

```
value = a - b
value = a * b
```

Subtrahiert bzw. multipliziert zwei Werte.

- **Parameter:** Zwei numerische Werte. Es muß gelten:
  `not isarray(a) and not isarray(b)`
  Eine Verletzung dieser Bedingung führt zur Exception **typeFault**.
- **Rückgabewert:** Resultat der Rechnung.

1.3 Division, Modulo

```
value = a div b
value = a mod b
```

Berechnet die ganzzahlige Division zweier Werte bzw. den Rest der ganzzahligen Division.

- **Parameter:** Zwei numerische Werte. Es muß gelten:
  `not isarray(a) and not isarray(b)`
  Eine Verletzung dieser Bedingung führt zur Exception **typeFault**.
  Falls der zweite Parameter den Wert 0 hat, wird die Exception **divisionByZero** geworfen.
- **Rückgabewert:** Resultat der Rechnung

2 Vergleichsoperatoren

Alle Vergleichsoperationen in roloBasic haben gemeinsam, daß sie nicht rekursiv sind. Bei verschachtelten Strukturen werden enthaltene Arrays nicht weiter betrachtet oder nur bezüglich Identität verglichen.

Alle Vergleiche resultieren in einem Wahrheitswert. roloBasic verwendet als Wahrheitswerte die Zahlenwerte 0 (für "false") und 1 (für "true").

2.1 Identität (same)

```
value = same(a,b)
```

Ermittelt die Identität zweier Werte, d.h. ob es sich bei `a` und `b` um den *selben* Wert handelt. Dies ist bei numerischen Werten immer dann gegeben, wenn sie denselben Zahlenwert darstellen.

Ein Array ist ausschließlich mit sich selbst identisch. Zwei separat angelegte Arrays mit demselben Inhalt sind *nicht* identisch.

Wenn `a` ein Array ist und `same(a,b)` gilt, führt eine Änderung eines Arrayelementes in `a` zu derselben Änderung in `b`, da in diesem Falle `a` und `b` lediglich verschiedene Namen für denselben Wertcontainer darstellen.

Die Zuweisung eines Wertes an `a` führt zu keiner Änderung in `b`, da hierbei lediglich die Bedeutung des Namens `a` geändert wird, ohne daß eine Änderung in einem Wertcontainer stattfindet.

- **Parameter:** Zwei beliebige Werte.
- **Rückgabewert:** 0 oder 1

2.2 Äquivalenz (=, <>)

```
value = a = b
```

Ermittelt, ob zwei Werte äquivalent (wertgleich) sind.

```
value = a <> b
```

Ermittelt, ob zwei Werte *nicht* äquivalent sind.

Es gilt: `same(a=b, not (a <> b))`

Zwei Wertcontainer gelten genau dann als äquivalent, wenn sie dieselbe Anzahl Werte enthalten und die in ihnen enthaltenen Werte jeweils paarweise identisch sind.

Folgende Funktion berechnet die Äquivalenz zweier Werte unter Verwendung der Funktion `same`:

```
function equivalent(a,b)
  if not isarray(a)
    return same(a,b)
  elseif not same(size(a), size(b))
    for i=0 to size(a)-1
      if not same(a[i], b[i])
        return 0
```

```
      endif
    next
    return 1
  endif
end
```

Der Zuweisungsoperator "=" verwendet zwar dasselbe Symbol wie der Äquivalenz-Vergleichsoperator, hat jedoch sonst nicht viel mit diesem gemein. Nach einer Zuweisung

```
a = b
```

gilt stets: `same(a,b)`. Es wird also nicht nur die Wertgleichheit hergestellt. Vielmehr gilt: Nach der Zuweisung sind `a` und `b` Namen für denselben Wertcontainer.

- **Parameter:** Zwei beliebige Werte.
- **Rückgabewert:** 0 oder 1

2.3 Ordnungsrelationen (>,<,>=,<=)

```
value = a > b
value = a < b
value = a >= b
value = a <= b
```

Alle vier Operatoren definieren dieselbe Ordnung. Es gilt daher:

```
(a < b) = (b > a)
(a < b) = (not (a >= b))
(a < b) = (not (b <= a))
```

Genau genommen sind zwei Ordnungen definiert:

- Numerische Ordnung über Zahlenwerte
- Lexikographische Ordnung über Arrays, die Zahlenwerte enthalten

Somit können zwei Zahlen oder zwei Arrays mit diesen Operatoren verglichen werden. Werden Arrays verglichen, dürfen diese ausschließlich Zahlenwerte enthalten. Werden Strukturen verglichen, die weitere Arrays enthalten, wird die Exception **typeFault** geworfen.

Jedenfalls können zwei Zeichenketten mit diesen Operatoren verglichen werden, da sie durch char-Arrays repräsentiert werden. Es gilt also:

```
"Zwerg" < "Zwickel"
```

- **Parameter:** Zwei numerische Werte oder zwei Arrays, die ausschließlich numerische Werte enthalten.
- **Rückgabewert:** 0 oder 1

3 Logik und Bitmanipulation

Alle logischen Operatoren außer `not` interpretieren Zahlen als Bitfelder. Sie werden ebenfalls für boole'sche Terme verwendet, hierbei wird lediglich das erste Bit in der Zahlendarstellung eingesetzt.

Dies hört sich vielleicht kompliziert an, ist jedoch in der Praxis einfach:

- Um boole'sche Terme zu formulieren, können die Operatoren `and`, `or`, `xor` und `not` verwendet werden.
- Um Bitfelder zu manipulieren, können die Operatoren `and`, `or`, `xor`, `<<`, `>>` und `bnot` verwendet werden. Alle Zahlenwerte können hierbei als 32-Bit-Werte mit Zweierkomplement angesehen werden.

3.1 and, or, xor

```
value = a and b
value = a or b
value = a xor b
```

Bitweise logische Operationen im 32-Bit-Zweierkomplement. Können auch als boole'sche Operatoren verwendet werden.

- **Parameter:** Zwei numerische Werte. Dies müssen keine Long-Werte sein. Das numerische Ergebnis ist jedoch dasselbe, als wenn alle Werte in einer 32-Bit-Darstellung wären.
- **Rückgabewert:** Ergebnis der Bitmanipulation.

3.2 Boole'sche Negation (not)

value = not a

Boole'sche Negation.

- **Parameter:** Ein numerischer Wert.

- **Rückgabewert:** Falls `a=0`, wird 1 zurückgegeben. Ansonsten wird 0 zurückgegeben.

3.3 Negation für Bitmanipulation (bnot)

```
value = bnot(a, type)
```

Negation für Manipulation von Bitfeldern. Der Wert `a` wird als ein Bitfeld der Größe des Typs `type` betrachtet. Dies sind 8 Bit für den Typen `char`, 16 Bit für `int` und 32 Bit für `long`. Alle Bits in diesem Feld werden negiert, das Resultat wird wieder als numerischer Wert aufgefaßt.

- **Parameter:** Ein numerischer Wert und ein numerischer Typ-Code.
- **Rückgabewert:** Resultat der Bitfeld-Negation.

3.4 Arithmetische Schiebeoperationen (<<, >>)

```
value = a << b
```

Schiebt alle Bits in `a` um `b` Stellen nach links. Dies entspricht einer Multiplikation mit 2^b.

```
value = a >> b
```

Schiebt alle Bits in `a` um `b` Stellen nach rechts. Das höchstwertige Bit (und somit das Vorzeichen) wird beibehalten.

- **Parameter:** Zwei numerische Werte.
- **Rückgabewert:** Resultat der Schiebeoperation.

4 Verwaltung von Arrays

Arrays sind die einzige Art von Datenstrukturen, die roloBasic kennt, und sie sind sehr effizient. Arrays können jederzeit angelegt werden, solange genug freier Speicher da ist -- und um den Speicher wieder freizugeben, genügt es, ein Array zu "vergessen", indem man dafür sorgt, daß man keinen Zugriff mehr auf das Array hat. Arrays können jederzeit in ihrer Größe verändert werden. Arrays dürfen leer sein, also 0 Elemente haben. Auf der anderen Seite wird die maximale Größe eines Arrays nur durch den

verfügbaren freien Speicher beschränkt. Beim Vergrößern eines Arrays geht der Inhalt nicht verloren.

Arrays müssen einen festen Elementtyp haben. Es stehen zur Auswahl:

```
char, int, long, vari
```

Der Elementtyp bestimmt, welche Arten von Werten als Elemente eines Arrays zulässig sind. `char`, `int`, und `long` erlauben nur numerische Werte, die den jeweiligen Wertebereich nicht überschreiten. vari hingegen erlaubt beliebige Werte, auch Arrays. Daher wird ein Array mit dem Elementtyp `vari` oft auch Struktur genannt.

Die Namen dieser Elementtypen werden von roloBasic doppelt genutzt. Zum einen stellt der Name eines Elementtyps einen numerischen Typ-Code dar. Zum anderen kann er als Array-Konstruktor verwendet werden, wenn er wie eine Funktion mit einer Parameterliste in runden Klammern aufgerufen wird. Auf diese Weise kann ein immutable Array mit den Werten in dieser Parameterliste erzeugt werden.

4.1 Erzeugung von mutable Arrays (dim)

```
value = dim(type, size)
```

Erzeugt ein neues Array vom Typ `type` mit `size` Elementen. Alle Arrayelemente werden mit 0 vorinitialisiert. Mit `dim` erzeugte Arrays sind immer mutable.

- **Parameter:** Ein numerischer Typ-Code und ein numerischer Wert.
- **Rückgabewert:** Das erzeugte Array.
- **Exceptions:** outOfMemory

4.2 Array-Konstruktoren(char, int, long, vari)

```
value = char(...)
value = int(...)
value = long(...)
value = vari(...)
```

Erzeugt ein immutable Array des jeweiligen Elementtyps.

- **Parameter:** Beliebig viele Werte. Diese müssen als Elemente des zu erzeugenden Arrays zulässig sein. Auch eine leere Parameterliste ist statthaft, hiermit kann ein leeres immutable Array erzeugt werden.
- **Rückgabewert:** Das erzeugte immutable Array.
- **Exceptions:** outOfMemory

4.3 Erkennung von Arrays(isArray)

```
value = isArray(a)
```

Erkennt, ob `a` ein Array ist oder nicht.

- **Parameter:** Ein beliebiger Wert `a`
- **Rückgabewert:** 1, falls `a` ein Array ist. Ansonsten 0.

4.4 Größenänderung eines Arrays (resize)

```
resize a, newSize
```

Verändert die Größe (Anzahl der Elemente) des Arrays `a` zu `newSize`. Dabei erzeugte Arrayelemente werden mit 0 initialisiert. Arrays dürfen, sofern sie mutable sind, jederzeit beliebig vergrößert oder verkleinert werden; die minimale erlaubte Größe ist 0, das Maximum ist vom verfügbaren Speicher abhängig.

- **Parameter:** Ein Array und ein numerischer Wert
- **Exceptions:** outOfMemory

4.5 Arraygröße ermitteln (size)

```
value = size(a)
```

Ermittelt die Anzahl der Datenelemente des Werts `a`. Ist `a` ein Array, entspricht `size(a)` der Größe des Arrays in Arrayelementen. Ist `a` ein Zahlenwert, ist `size(a) = 1`.

- **Parameter:** Ein beliebiger Wert. Normalerweise ein Array.
- **Rückgabewert:** Die Größe dieses Werts als Anzahl enthaltener Datenelemente.

4.6 Erzeugung leerer mutable Arrays (reserve)

```
value = reserve(type, size)
```

Erzeugt ein leeres Array vom Typ `type`, das jedoch darauf vorbereitet ist, besonders effizient bis auf die Größe `size` (Anzahl Elemente) vergrößert zu werden. Diese Art, Arrays zu erzeugen, ist besonders vorteilhaft, wenn man Arrays als Stacks (Stapelspeicher) verwenden möchte. (Eine Vergrößerung über `size` hinaus ist trotzdem möglich. Und es können auch mit `dim` erzeugte Arrays als Stacks verwendet werden.)

- **Parameter:** Ein numerischer Typ-Code und ein numerischer Wert.
- **Rückgabewert:** Das erzeugte leere Array.
- **Exceptions:** outOfMemory

4.7 Arrays als Stack verwenden (push, pop, top)

```
push a, x
value = pop(a)
value = top(a)
```

Arrays können in roloBasic sehr einfach als Stack (Stapelspeicher) verwendet werden.

- Die Prozedur `reserve` ist geeignet, einen leeren Stack vorzubereiten.
- Die Prozedur `push` fügt den Wert `x` als neues letztes Element an ein Array `a` an und vergrößert das Array dabei um 1.
- Die Funktion `top` liest das letzte Element des Arrays `a` aus.
- Die Funktion `pop` liest das letzte Element und entfernt es aus dem Array `a`.
- **Parameter:** `a` muß ein Array sein. `x` darf ein beliebiger Wert sein, der als Element für `a` erlaubt ist.
- **Rückgabewert:** Das letzte Element des Arrays `a`.
- **Exceptions:** outOfMemory

5 Weitere Befehle zur Datenverwaltung

Erleichtern den Umgang mit roloBasic-Datenobjekten.

5.1 Kopieren eines Datenobjekts (copy)

```
value = copy(a)
```

Erzeugt eine Kopie von `a`.

Falls `a` ein Array ist, so ist die erzeugte Kopie von `a` mutable.

Es wird keine "tiefe" Kopie angefertigt. Die Kopie enthält also exakt dieselben Elemente wie `a` -- diese Elemente werden jeweils nicht kopiert, sondern zugewiesen!

- **Parameter:** `a` darf ein beliebiger Wert sein, doch ist der Aufruf von `copy` eigentlich nur mit Arrays sinnvoll.
- **Rückgabewert:** Die Kopie von `a`
- **Exceptions:** outOfMemory

5.2 Abfrage des Elementtyps (type)

```
value = type(a)
```

Ermittelt den Elementtyp von `a`. Ist a ein numerischer Wert, so ist sein Elementtyp allein vom Zahlenwert abhängig. Somit gilt:

```
! type(0) = char
! type(256) = int
! type(-1) = int
! type(100000) = long
```

Falls `a` ein Array ist, so wird derselbe Elementtyp zurückgegeben, der beim Erzeugen des Arrays (z.B. mit `dim`) angegeben wurde.

- **Parameter:** ein beliebiger Wert
- **Rückgabewert:** Die numerische Element-Typ-ID, welche dem Elementtyp von `a` entspricht.

5.3 Abfrage der Veränderbarkeit (mutable)

```
value = mutable(a)
```

Ermittelt, ob `a` mutable ist. Werte, die nicht mutable sind, nennt man auch "immutable". Datenobjekte, die "immutable" sind, können nicht verändert werden. Immutable Arrays kann man weder vergrößern noch verkleinern, außerdem kann man ihnen keine neuen Elemente zuweisen.

- **Parameter:** ein beliebiger Wert
- **Rückgabewert:** 1, falls `a` mutable ist. Ansonsten 0.

6 Konstanten

Es gibt in roloBasic einige globale Systemkonstanten. Die Namen dieser Konstanten sind reserviert und können nicht als Variablennamen verwendet werden. Diese Konstanten stehen jeweils für einen numerischen Wert. Sie werden eingesetzt, um Typen und Exceptions zu codieren.

6.1 Typ-IDs

Die 4 Typ-ID-Systemkonstanten lauten:

```
char
int
long
vari
```

Diese Konstanten sind Codes für die Basistypen von roloBasic. Sie können verwendet werden, um Typen zu vergleichen oder Arraytypen festzulegen. Der Typ `vari` hat eine Sonderstellung: Er kann nur als Elementtyp von Arrays verwendet werden.

```
! Beispiel:
if type(a) = char and isArray(a)
  print "a ist eine Zeichenkette"
endif
```

6.2 Fehler-IDs

Immer wenn in roloBasic ein Laufzeitfehler auftritt, wird eine Exception geworfen. Diese Exceptions haben jeweils einen numerischen Wert, der durch die folgenden Systemkonstanten definiert wird:

```
outOfMemory         ! Zu wenig freier Speicher vorhanden

rootstackOverflow   ! Interner Systemfehler
nullpointerAccess   ! Interner Systemfehler

valueRange          ! Wertbereichsüberschreitung, z.B. bei
                    ! Zuweisung von Werten an Arrays oder bei
                    ! Parametern von Systemfunktionen mit ein-
                    ! geschränktem Wertebereich

divisionByZero      ! Division durch 0. Kann bei div oder mod
                    ! auftreten.

argumentFault       ! Ungültige Anzahl Argumente beim Aufruf
                    ! einer Systemfunktion.

illegalFunction     ! Eine Variable wurde wie eine Funktion oder
                    ! Prozedur aufgerufen, enthält jedoch keine
                    ! gültige Funktion oder Prozedur.

indexRange          ! Indexbereichsüberschreitung bei Array-
                    ! zugriff

typeFault           ! Typfehler.
                    ! Z.B. beim Aufruf einer Systemfunktion, die
                    ! als Argument einen numerischen Wert erwar-
                    ! tet, wurde ein Array übergeben.

! Beispiel:

print "Ganzzahliger Quotient: ", a div b
print "Rest: ", a mod b
catch x
if x = divisionByZero
  print "Division durch Null!"
endif
```

IV ANHANG

Tabellarische und formale Informationen zu roloBasic.

1 Syntax (in EBNF)

```
Program   = { ( Statement | Procedure | Function ) NewLine } .
Procedure = "procedure" Ident FormalParams NewLine Block "end" .
Function  = "function" Ident "(" FormalParams ")" NewLine
            Block "end".

Block       = { Statement NewLine } .
Statement   = Assignment | ProcCall | Do | For | If .
Assignment  = Lvalue "=" Expression .

VarAccess   = [ "global" | "local" ] Ident .
Lvalue      = VarAccess | ArrayAccess .
ArrayAccess = Expression "[" Expression "]" .

Expression      = FunCall | UnaryOperation | BinaryOperation |
                  Literal | VarAccess | ( Expression ) .
UnaryOperation  = UnaryOperator Expression .
BinaryOperation = Expression BinaryOperator Expression .

ProcCall     = Ident ActualParams .
FunCall      = Ident "(" ActualParams ")" .
FormalParams = [ { Ident "," } Ident ] .
ActualParams = [ { Expression "," } Expression ] .

Do =  "do" [ "while" Expression ] NewLine
      Block
      "loop" [ "until" Expression ] .

For = "for" Ident "=" Expression "to" Expression
      [ "step" Expression ] NewLine
      Block
      "next" .

If =  "if" Expression NewLine
      Block
      { "elseif" Expression NewLine Block }
      [ "else" NewLine Block ]
      "endif" .
```

1.1 Terminals

- Ident: Bezeichner. Darf nur Zeichen A-Z, Ziffern 0-9, und Unterstrich _ enthalten. Darf nicht mit einer Ziffer beginnen.
- NewLine: Echter Zeilenumbruch oder Doppelpunkt :
- Literal: Konstanten für einzelne Zeichen, Zeichenketten, Integerwerte, oder Fließkommawerte, Beispiele: `'x' "Hallo" -333 0.127`
- UnaryOperator: Unärer Operator, z.B. Negation
- BinaryOperator: Binärer Operator, z.B. Addition

In Zeichenkettenliteralen dient \ zur Darstellung spezieller Zeichen, z.B: `\" \\ \n \t \064`

1.2 Operatorprioritäten

Operatorsymbole	Priorität	Bemerkung
- , not	8	unär
* , div , mod	7	mod, div: erzeugt Integer
+ , -	6	+ : auch für Strings/Arrays anwendbar
<< , >>	5	Schiebeoperationen
< , <= , > , >=	4	vergleicht auch Strings/Arrays
= , <>	3	vergleicht auch Strings/Arrays
and	2	and, xor, or sind auch bitweise!
xor, or	1	auch bitweise

2 Typen, Speicherbedarf von Werten

Typ	Speicherbedarf (Bytes)	als Arrayelement (Bytes)
char	2	1 (in char-Arrays)
int	4	2 (in int-Arrays)
long	6	4 (in long-Arrays)
array of char (String)	2+size(...)	4 + size(...)
array of int	2+size(...)*2	4 + size(...)*2
array of long	2+size(...)*4	4 + size(...)*4
array of vari (Struktur)	2+size(...)*2	4 + size(...)*2