Lysa
Handbuch

Version 0.4

Geschrieben von
Marcus Daniel Cremer

Laut cloc umfaßte der Quellkode für den Interpreter bei der letzten Zählung 3.047 Zeilen (zuzüglich 821 Zeilen für Kommentare sowie 709 Leerzeilen).

Fassung vom 1. Februar 2021

Neben einer relativ kurzen Einführung in das Konzept von *Lysa* enthält dieses Handbuch vor allem ein Verzeichnis der zur Verfügung stehenden Kommandos. In den Erklärungen finden sich aber auch weitere konzeptionelle Informationen und Beispiele.

Diese Dokumentation wird mit LuaLaTeX gesetzt. Für die Darstellung werden die `kpfonts` verwendet.

Inhaltsverzeichnis

Mona Lysa

Lysa ist eine Programmiersprache, die in Zusammenhang mit dem Projekt pro*ly*sa*rik* entsteht. pro*ly*sa*rik* kombiniert in Videos gesprochenen Text mit kinetischer Typographie. Außerdem ist pro*ly*sa*rik* auch die Software, mit der diese Videos erstellt werden. Dies geschieht vermittels einer im Verlauf einiger Jahre eigens dafür entwickelten Skriptsprache. *Lysa* ergänzt pro*ly*sa*rik* zunächst, um es schließlich als Software zu ersetzen und fortzuführen. Hauptaufgabe für *Lysa* ist also die Erstellung von Videos mit kinetischer Typographie.

Lysa basiert auf Forth, einer »imperativen, stackbasierten Programmiersprache«, deren Erfindung durch Charles H. Moore in die späten 60er-Jahren des 20. Jahrhunderts fällt. Anders als Forth, das interaktiv ausgeführt wird und untrennbar mit seiner Entwicklungsumgebung verbunden ist (es dient gar nicht so selten als Betriebssystem), läßt sich *Lysa* nicht interaktiv ausführen. *Lysa* ist von vornherein als interpretierte Skriptsprache gedacht. Der Interpreter selbst ist in Go[1] geschrieben (die ersten Versionen hingegen in Java). Dementsprechend wird ein Programm in *Lysa* ausgeführt, indem der Interpreter mit dem Skript als Argument aufgerufen wird: `lysa skriptname` (vorzugsweise aus dem Verzeichnis heraus, in dem das Skript zu finden ist). Das Skript kann auch komprimiert vorliegen (Endung: `.lysa.gz`). Insofern eine Skriptdatei nicht darauf oder auch einfach nur auf `.lysa` endet, wird die Dateiendung `.lysa` automatisch ergänzt.

[1] Was in gewisser Weise reizvoll ist, denn *Lysa* ist in vielerlei Hinsicht geradezu das Gegenteil von Go.

Warum Forth? Mit der Sprache in Berührung gekommen bin ich erstmals in der ersten Hälfte der 80er Jahre des letzten Jahrhunderts. Ich hatte einen ZX Spectrum und Forth hatte ich mir zugelegt, da man damit Grafik programmieren können sollte. Insbesondere die umgekehrte polnische Notation (Postfixnotation) fand ich befremdlich, weshalb ich da nur reingeschnuppert habe.

Bei prolysatrik fand ich es erstrebenswert, immer wenn numerische Angaben zu machen sind, dies auch in der Form von Formeln machen zu können, mit der diese Angaben errechnet werden können, wenn sie dann gebraucht werden. In einer solchen Formel können auch Variablen verwendet werden, deren Werte sich je nach Situation ändern können. Folglich brauchte ich einen Formelparser.

Mit dem mXparser von Mariusz Gromada steht auch ein mächtiger Parser als freie Software zur Verfügung, nur leider stellte sich dieser für nebenläufige Berechnungen als nur eingeschränkt geeignet heraus. Wenn verschiedene Threads dieselbe Formel verarbeiten, die darin enthaltenen Variablen aber jeweils etwas anderes bedeuten, funktioniert das nicht mehr richtig. Oder ich habe dabei etwas falsch gemacht. Das würde ich niemals ausschließen.

Jedenfalls habe ich mich dazu entschlossen, selbst einen Formelparser zu schreiben. Der gängige Weg dabei ist, die in der Infixnotation angegebenen Formeln unter Einsatz mehrerer Stapel mit dem Rangierbahnhof-Algorithmus in eine Postfixnotation umzuwandeln und dann mit entsprechenden Funktionen zu berechnen. Das erinnerte mich deutlich an Forth.

Nachdem ich das zu meiner Zufriedenheit umgesetzt hatte, stellte sich zu meiner wachsenden Unzufriedenheit die sich aufdrängende Frage, wenn ich schon zur Berechnung numerischer Angaben eine Art Forth einsetze, warum ich nicht gleich die gesamte Skriptverarbeitung darauf umstellen sollte. Also ein einziger Parser nicht nur für die Formeln, sondern gleich für das gesamte Skript.

In einem Skript gebe ich dem Anwender gewisse Möglichkeiten, und bemühe mich, Fehler möglichst auszuschließen, indem ich

den Eingaben enge Grenzen setze. Jedes Kommando braucht bestimmte Parameter und sowohl dem Anwender als auch dem Entwickler (in diesem Fall ist das dieselbe Person) ist gedient, wenn dabei möglichst nur sinnvolle Eingaben möglich sind.

In einer Programmiersprache hingegen stelle ich dem Programmierer gewisse, eher allgemein gehaltene Möglichkeiten zur Verfügung, ohne dabei genau zu wissen, was er im Einzelfall damit vorhat. Insofern verbietet sich weitgehend, Möglichkeiten einzuschränken. Dadurch nimmt zwar die Fehleranfälligkeit exorbitant zu, andererseits wachsen die kreativen Möglichkeiten des Anwenders entsprechend (immer noch dieselbe Person wie der Entwickler). Ein Skript zur Steuerung einer Anwendung ist also rein konzeptionell etwas ziemlich anderes als eine vollwertige Programmiersprache wie Forth. Aber es gibt wenig reizvollere Dinge für einen Schriftsteller, als sprachschöpferisch tätig zu werden. Und darum wird nun also aus pro*ly*sa*rik* *Lysa*.

Infixnotation:	$(1 + 2) \times 3$
Postfixnotation:	$1\ 2\ +\ 3\ \times$

Was die Postfixnotation (abgesehen von der Reihenfolge) von der Infixnotation unterscheidet ist, daß keine Klammern benötigt werden. Genau aus diesem Grund wandelt ein Formelparser die eine Notation in die andere um. Denn in der Postfixnotation hat er die Angaben und Operatoren genau in der Reihenfolge, in der sie abzuarbeiten sind. In der Infixnotation bringen die Klammern die Reihenfolge oft durcheinander.

Wenn etwas nicht benötigt wird, steht es zur anderweitigen Verwendung zur Verfügung. Die Eingabe des originalen Forth besteht aus durch Leerzeichen getrennten Wörtern (obiges Beispiel für die Postfixnotation ist gültiger Forth-Kode). In *Lysa* hingegen werden Klammern dazu verwendet, um Eingaben sowohl abzugrenzen als auch semantisch zu markieren. Erst einmal wird alles ignoriert, was nicht eingeklammert ist. Kommentare lassen sich also bedenkenlos zwischen die eingeklammerten Wörter schreiben und bedürfen keiner eigenen Kennzeichnung.

Lysa kennt vier verschiedene Klammernpaare für Substantive, Pronomen, Anführungen und Verben. Die Terminologie des originalen Forth wird also beibehalten, daß eine Eingabe aus Wörtern besteht. Durch die Klammern ist für den Interpreter aber erkennbar, um welche Wortart es sich jeweils handelt:

- Als Substantive können entweder Zahlen oder Zeichenketten angegeben werden, sie stehen also unmittelbar für Gegebenheiten.
- Pronomen sind Platzhalter für Gegebenheiten.
- Anführungen sind Eingaben, die nicht direkt abgearbeitet werden. Es wird damit also eine gewisse Uneigentlichkeit markiert, die allerdings aufgelöst werden kann.[2] Anführungen können beliebige Gegebenheiten enthalten (auch weitere Anführungen, welche als Ganzes also auch Gegebenheiten darstellen).
- Verben sind die Wörter, bei denen etwas geschieht. Meist verbrauchen sie Gegebenheiten und/oder schaffen welche.

Gegebenheiten sind das, was auf den Stapel gelegt wird (wenn auch nur mittelbar). Vermutlich erscheint einem der Begriff erst einmal alles andere als naheliegend, doch möglicherweise wird er verständlicher, wenn ich darauf hinweise, daß Datum das lateinische Wort für Gegebenes ist (und Daten ist die Mehrzahl von

[2] »Wenn man in der gewöhnlichen Weise Worte gebraucht, so ist das, wovon man sprechen will, deren Bedeutung. Es kann aber auch vorkommen, dass man von den Worten selbst oder von ihrem Sinne reden will. Jenes geschieht z. B., wenn man die Worte eines Andern in gerader Rede anführt. Die eigenen Worte deuten dann zunächst die Worte des Andern und erst diese haben die gewöhnliche Bedeutung. Wir haben dann Zeichen von Zeichen. In der Schrift schließt man in diesem Falle die Wortbilder in Anführungszeichen ein. Es darf also ein in Anführungszeichen stehendes Wortbild nicht in der gewöhnlichen Bedeutung genommen werden.« Gottlob Frege Über Sinn und Bedeutung. In: Zeitschrift für Philosophie und philosophische Kritik, N. F., Bd. 100/1 (1892), S. 25-50

Datum, erst in neuerer Zeit hat sich die Bedeutung von Datum in der deutschen Sprache auf das Kalenderdatum verengt).

Die Klammern habe ich bisher noch nicht gezeigt, denn es gibt davon mehrere Sätze, Dialekte genannt. Zu Anfang eines Skriptes kann angegeben werden, welcher Klammerdialekt verwendet wird. Innerhalb des Skriptes läßt sich das allerdings dann nicht mehr ändern. Die immer zur Verfügung stehenden Klammerdialekte sind:

Dialekt	*Substantive*		*Pronomen*		*Anführungen*		*Verben*	
Winkel	⟨	⟩	⟨·	·⟩	⟪	⟫	⟦	⟧
	27e8	27e9	2991	2992	27ea	27eb	27e6	27e7
Standard	(	)	{	}	<	>	[	]
Guillemets	‹	›	‚	‘	„	“	«	»
	2039	203a	201a	201b	201e	201c	00ab	00bb
Ecken	⌞	⌟	⌊	⌋	⌜	⌝	⌈	⌉
	231e	231f	230a	230b	231c	231d	2308	2309

Anders als vermutlich angenommen, sind nicht die Standard- sondern die Winkelklammern der normalerweise verwendete Dialekt. In diesem sind auch die Beispiele in diesem Handbuch gehalten. Die Verwendung ansonsten unüblicher Klammern erklärt sich vor allem dadurch, daß diese nicht in den Angaben vorkommen dürfen. Aus diesem Grunde verwendete bereits pro*ly*sa*rik* die Winkelklammern, um Angaben voneinander abzugrenzen. Aber es gibt sicher genug Gelegenheiten, in denen die Standardklammern oder die Guillemets ausreichen, weil in den Angaben keine Klammern oder Anführungszeichen vorkommen. Die Definition eigener Klammerdialekte ist zudem möglich.

Eingestellt werden kann ein Dialekt, indem ab der ersten Spalte der ersten Zeile eines Skriptes (funktioniert auch bei Teilskripten) ein Ausrufezeichen nebst dem Wort Lysa gesetzt wird, dem der Name des zu verwendenden Dialektes in runden Klammern folgt. Also zum Beispiel:

```
!Lysa(Standard)
```

Konvention ist, daß jedes *Lysa*-Skript mit der Bestimmung des verwendeten Dialekts beginnen sollte, selbst wenn es sich um Winkelklammern handelt und die Anweisung insofern überflüssig ist.

Grundsätzlich arbeitet *Lysa* mit Unicode-Eingaben (UTF-8). Ob ein Symbol in der Schrift STIX Two Math vorhanden ist, entscheidet darüber, ob es verwendet werden kann oder nicht. Die Darstellung in diesem Handbuch verwendet LaTeX-Symbole, die nicht immer voll den Unicode-Zeichen entsprechen. Aber der entsprechende Code ist jeweils mit aufgeführt.

Wie das originale Forth arbeitet auch *Lysa* mit Stapelspeichern. Während beim originalen Forth Daten direkt auf dem Stapel abgelegt werden, landen bei *Lysa* dort jedoch grundsätzlich nur Verweise auf die eigentlichen Daten. Das hat den Vorteil, daß alle Gegebenheiten denselben Platz auf dem Stapel beanspruchen (nämlich den des Verweises), gleichgültig, wie umfangreich ihr Inhalt sein mag.

Der Stapel arbeitet nach dem LIFO-Prinzip. Die zuletzt abgelegte Gegebenheit wird als erste wieder abgerufen. Tatsächlich arbeitet *Lysa* sogar mit zwei dieser Speicher, denn neben dem Stapel gibt es noch einen Keller, der aber vor allem als Zwischenspeicher dient. Die Kapazität aller Stapel in *Lysa* wird ausschließlich durch den zur Verfügung stehenden Hauptspeicher begrenzt.

Obiges Beispiel sähe in *Lysa* also folgendermaßen aus (mit Darstellung des Zustands des Stapels):

Wörter	*Stapel*	*Typen*
⟨1⟩	⟨1⟩	Zahl
⟨2⟩	⟨1⟩⟨2⟩	Zahl, Zahl
⟦+⟧	⟨3⟩	Zahl
⟨3⟩	⟨3⟩⟨3⟩	Zahl, Zahl
⟦×⟧	⟨9⟩	Zahl

Die Substantive legen sich sozusagen selbst auf den Stapel, die Verben in diesem Beispiel nehmen jeweils zwei Gegebenheiten vom Stapel (konsumieren diese) und legen stattdessen das Ergebnis dort ab (schaffen eine neue Gegebenheit).

Wenn eines der Verben statt zweier Zahlen zum Beispiel eine Zeichenkette auf dem Stapel vorgefunden hätte, wäre das Programm mit einer Fehlermeldung abgebrochen worden. Es obliegt den Verben, zu überprüfen, ob die richtigen Gegebenheiten für sie bereitliegen. Üblicherweise verhalten sie sich nicht fehlertolerant.

Zahlen sind in *Lysa* übrigens ausschließlich Fließkommazahlen (doppelte Genauigkeit). Sie können negativ sein, dann beginnen sie mit einem Bindestrich. Als Dezimaltrenner dient das Komma. Tausendertrennzeichen sind nicht gestattet. Zeichenkette ist alles, was sich nicht als Zahl interpretieren läßt.

Menschliche Grundsprache für *Lysa* ist Deutsch, wie sich an dem Komma als Dezimaltrenner erkennen läßt. Dies liegt keineswegs daran, daß diese Sprache als geeigneter als andere für das Programmieren in *Lysa* gehalten wird, hat auch nichts mit Deutschtümelei zu tun, sondern es handelt sich hier um eine Programmiersprache, die in Hinblick auf einen einzelnen, deutschsprachigen Benutzer entwickelt wird (von diesem selbst). Es ist schlicht bequemer, in seiner Muttersprache zu arbeiten. Schon beim Programmieren das Komma des separaten Zehnerblocks nutzen zu können und nicht den Punkt suchen zu müssen, ist schon eine Erleichterung.

Neben Zahlen und Zeichenketten gibt es noch zahlreiche weitere Typen von Gegebenheiten. Doch diese lassen sich nicht direkt eingeben, sondern werden durch Verben geschaffen. In der Regel sind Gegebenheiten immutabel. Soll eine Zahl verändert werden, wird nicht die sie repräsentierende Gegebenheit verändert, sondern diese wird durch eine neue Gegebenheit ersetzt. Nur bei Gegebenheiten, die zu komplex und/oder umfangreich sind, um sie jedes Mal zu ersetzen, finden Veränderungen in der Gegebenheit statt. Es ist ineffektiv, bei Änderung eines einzelnen Bildpunktes gleich ein neues Bild zu erschaffen, vor allem wenn dies häufiger vorkommt.

Pronomen und Verben werden durch Zeichenketten identifiziert. Sie können homonym sein, da durch die Klammern zu erkennen ist, was jeweils gemeint ist. Verben können Synonyme haben. Zumeist trifft das auf jene Verben zu, die auch durch Symbole dargestellt werden können. Zum Beispiel wäre oben statt des ⟦+⟧ auch ein ⟦ADD⟧ oder ein ⟦addieren⟧ möglich gewesen. Zusätzliche Verben, die während des Programmlaufs definiert werden können, haben zunächst nur eine einzige Zeichenkette als Bezeichnung; Synonyme sind allerdings leicht zu generieren.

Pronomen werden gebildet, indem eine Zeichenkette mit einer Gegebenheit in Verbindung gebracht wird (durch ein spezielles Verb); zusätzliche Verben werden gebildet, indem eine Zeichenkette in Verbindung mit einer Anführung gebracht wird (durch ein anderes spezielles Verb). Letzteres entspricht mehr oder minder dem Compilermodus des originalen Forth. Anführungen können sowohl Pronomen als auch Verben bilden. Durch andere Verben können Anführungen ausgeführt werden, das heißt, sie verlieren ihre Uneigentlichkeit und der Inhalt wird effektuiert. Insofern können Anführungen sowohl als Verben als auch als Pronomen ausgeführt werden. Der Möglichkeiten sind gar viele, da eine Anführung intern aus einem eigenen Stapel besteht. Nachdem eine Anführung durch die sie schließende Klammer geschlossen wurde, sind allerdings keine weiteren Änderungen mehr möglich.

Sowohl hart kodierte als auch hinzugefügte Verben werden in einem zentral geführten Wörterbuch verwaltet, auf das alle Prozesse Zugriff haben. Es gibt also keine lokalen Verben.

Für Pronomen gibt es auch ein zentral geführtes Wörterbuch. Daneben gibt es aber auch eines für jede Instanz des Interpreters. Einige Verben effektuieren Anführungen mit einem eigenen Interpreter, der dann zwar Zugriff hat auf das zentrale Wörterbuch, nicht jedoch auf das Wörterbuch des ausführenden Verbs. Dafür verfügt er über ein eigenes lokales Wörterbuch.

Anführungen bestehen zwar aus einem eigenen Stapel, aber sie verfügen über keinen eigenen Interpreter, folglich auch nicht generell über ein eigenes lokales Wörterbuch.

Wird ein Pronomen in einem lokalen Wörterbuch gefunden, wird nicht weiter danach im zentralen gesucht. Lokale Wörterbücher haben also Vorrang vor dem zentralen.

Es kann nicht mehrere Verben oder Pronomen unter derselben Zeichenkette geben. Erfolgt eine Verknüpfung mit einer bereits verwendeten Zeichenkette, wird die bestehende ersetzt.

Lysa gibt einige Meldungen auf der Konsole aus. Abgesehen davon, daß sie vermelden, wenn etwas schief gelaufen ist, zeigt diese Ausgabe vor allem an, daß das Programm noch läuft und nicht etwa hängen geblieben ist. Hauptsächlich schreibt *Lysa* seine Ausgaben in eine HTML-Datei, die schließlich mit jedem Browser betrachtet werden kann. Das ermöglicht die Ausgabe auch komplexerer Informationen.

Voreingestellt ist, die HTML-Ausgabedatei als `Lysa.html` ins temporäre Verzeichnis zu schreiben (`/tmp`). In diesem Zusammenhang sei angemerkt, daß *Lysa* in Hinblick auf Linux als Betriebssystem geschrieben wird. Der Name der Ausgabedatei kann allerdings frei gewählt werden (inklusive Pfad). Dazu ist der Interpreter aufzurufen als `lysa ausgabedatei skriptname`. Die Dateiendungen werden automatisch ergänzt.

Bei Programmstart sucht *Lysa* zunächst nach der Datei `/usr/share/lysa/Anpassen.lysa` und lädt sie gegebenenfalls (falls erfolglos, wird auch noch nach `/usr/share/lysa/Anpassen.lysa.gz` gesucht). Es handelt sich dabei um ein Skript, welches abgearbeitet wird wie jedes andere, sein Hauptzweck ist es jedoch, den Wortschatz mit eigenen, in *Lysa* geschriebenen Verben zu erweitern und *Lysa* den eigenen Bedürfnissen gemäß anzupassen. Dafür stehen einige spezielle Verben zur Verfügung.

Ich wurde einmal gefragt, da hatte ich mit der Entwicklung von *Lysa* mehr oder weniger gerade erst begonnen, wie es mit meinem Pseudo-Forth vorangehe. Natürlich habe ich brav geantwortet, auch wenn die Frage falsch gestellt war. Denn es handelt sich nicht um ein falsches Forth, woran ich arbeite, sondern um ein echtes *Lysa*.

Vokabular

Verzeichnis aller Verben

Grundwortschatz

↥ (Strg+Shift+u21a5)

POP
auskellern

⟨*Gegebenheit*⟩ ⟦↥⟧ ⟨*Gegebenheit*⟩

Nimmt eine beliebige Gegebenheit aus dem Keller und legt sie auf den Stapel.

!
MEM
befürworten

⟨*Pronomen*$_{Zk}$⟩⟨*Gegebenheit*⟩[[!]]

Nimmt eine beliebige Gegebenheit (kann sich auch um einen Block handeln) und eine Zeichenkette vom Stapel. Unter der Zeichenkette wird die Gegebenheit ins lokale Lexikon eingetragen und steht damit im aktuellen Interpreter unter der Zeichenkette als Pronomen zur Verfügung.

!!
!! `(Strg+Shift+u203c)`
DEF
Befürwortung

⟨*Pronomen*$_{Zk}$⟩⟨*Gegebenheit*⟩[[!!]]

Nimmt eine beliebige Gegebenheit (kann sich auch um einen Block handeln) und eine Zeichenkette vom Stapel. Unter der Zeichenkette wird die Gegebenheit ins globale Lexikon eingetragen und steht damit überall unter der Zeichenkette als Pronomen zur Verfügung.

Dialekt

⟨*Name*$_{Zk}$⟩⟨*Klammern*$_{Zk}$⟩[[Dialekt]]

Nimmt zwei Zeichenketten vom Stapel und generiert daraus einen neuen Klammerdialekt. Die Zeichenkette mit den Klammern

muß exakt acht Zeichen lang sein (ansonsten Fehlermeldung, Programmabbruch) und vier ineinander geschachtelte Klammerpaare enthalten (von außen nach innen: Anführungen, Pronomen, Substantive, Verben [also: ⟪⟨·⟨[[]]⟩·⟩⟫]).

Da sich ein Klammerdialekt nicht innerhalb eines Skriptes wechseln läßt, wird ein neuer Dialekt üblicherweise im Anpassungsskript (`/usr/share/lysa/Anpassen.lysa`) definiert. Er kann dann in jedem danach geladenen Skript wie üblich in der ersten Zeile gesetzt werden.

Die Klammern können weitgehend frei gewählt werden. Es empfiehlt sich, für jede Klammerart ein eigenes Zeichen zu wählen. Das einzige Zeichen, das nicht für die Definition eines neuen Klammerdialekts zur Verfügung steht, ist die rechte Klammer für Substantive im Klammerdialekt des Skriptes, in dem die Definition stattfindet. Da empfiehlt es sich dann, für dieses Skript einen der vier Standarddialekte zu wählen, in dem an dieser Stelle kein Zeichen verwendet wird, welches man verwenden möchte.

Wird als Name die Bezeichnung eines bereits bestehenden Dialektes gewählt, wird dieser überschrieben. Dies bedeutet dann allerdings, daß dadurch alle Skripte, die in diesem Dialekt geschrieben wurden, unlesbar werden.

↓ (Strg+Shift+u2913)

PUSH
einkellern

⟨*Gegebenheit*⟩[[↓]]	⟨*Gegebenheit*⟩

Nimmt eine beliebige Gegebenheit vom Stapel und legt sie in den Keller.

DROP
entfernen

⟨*Gegebenheit*⟩⟦DROP⟧

Nimmt eine beliebige Gegebenheit vom Stapel.

DUP
duplizieren

⟨*Gegebenheit*⟩⟦DUP⟧ ⟨*Gegebenheit*⟩

Schaut sich eine beliebige Gegebenheit (kann sich auch um einen Block handeln) auf dem Stapel an und legt sie noch einmal darauf.

Es wird dabei keine Kopie der Gegebenheit erzeugt und auf den Stapel gelegt, sondern lediglich die Adresse der Gegebenheit wird noch einmal auf dem Stapel deponiert. Bei immutablen Gegebenheiten ist das unproblematisch, da sie nur konsumiert werden, bei veränderlichen Gegebenheiten ist eventuell ein echtes Kopieren vorzuziehen (siehe S. 25).

↵ (Strg+Shift+u23ce)
EXE
effektuieren

⟪...⟫⟦↵⟧

Nimmt einen Block vom Stapel und effektuiert diesen unbedingt, führt also die sich im Block befindlichen Wörter aus.

```
⟨Test⟩⟪...⟫⟦CMD⟧
⟦Test⟧
```

oder

⟨Test⟩⟪...⟫[[!]]
⟨·Test·⟩[[↵]]

Beide Versionen bewirken, daß die Wörter im Block ausgeführt werden. Nur wird der Block im ersten Fall global als Verb gespeichert, im zweiten lokal als Pronomen (auf diese Weise kann es also doch auch lokale Verben geben, nur eben nicht als Verben). Das Pronomen hätte aber natürlich genausogut auch global gespeichert werden können.

PICK
hervorkramen

⟨*Gegebenheit*⟩$_{Index}$ ... ⟨*Index*$_{ZI}$⟩[[PICK]] ⟨*Gegebenheit*⟩

Nimmt eine Zahl vom Stapel, die zunächst einmal zu einem Integerwert gerundet wird. Danach wird sie als Index verwendet, um sich eine tiefer liegende, beliebige Gegebenheit auf dem Stapel anzuschauen und oben auf den Stapel zu legen.

Wichtig hinsichtlich des Indexes ist, daß er erst hinter dem Index selbst zu zählen beginnt. Die Gegebenheit hinter dem Index (von rechts aus gesehen) ist also die erste usw. Insofern nicht ausreichend Gegebenheiten auf dem Stapel zu finden sind, wird eine Fehlermeldung ausgegeben und das Programm abgebrochen.

hochstapeln

[[hochstapeln]]

Gibt gegebenenfalls eine Liste der auf dem Stapel liegenden Datentypen in die HTML-Datei aus. Das Kommando ist vor allem für die Fehlersuche gedacht.

importieren

⟨$Datei_{Zk}$⟩⟦#⟧

Nimmt eine Zeichenkette vom Stapel. Falls diese nicht bereits auf `.1ysa` oder `.1ysa.gz` endet, wird die Endung `.1ysa` angefügt. Danach wird versucht, die durch die Zeichenkette bestimmte Datei zu öffnen und einzulesen.

Es kann sich bei der importierten Datei auch um ein komprimiertes Skript handeln (Endung: `.1ysa.gz`).

Die verschiedenen in der importierten Datei enthaltenen Wörter werden nicht statistisch erfaßt (fließen also nicht in die Angaben am Ende des Programmlaufs in der HTML-Datei ein). Das importierte Skript kann über einen eigenen Klammerdialekt verfügen.

↕ (Strg+Shift+u21a8) Kellerkopie

⟨*Gegebenheit*⟩⟦↕⟧ ⟨*Gegebenheit*⟩

Schaut sich eine beliebige Gelegenheit aus dem Keller an und legt sie auf den Stapel.

Dieses Kommando könnte auch in *Lysa* formuliert werden, aber die hart kodierte Version ist noch simpler:

⟨↕²⟩⟪⟦↥⟧⟦DUP⟧⟦↓⟧⟫⟦CMD⟧

CPY
kopieren

⟨*Gegebenheit*⟩[[CPY]] ⟨*Gegebenheit*⟩

Schaut sich eine beliebige Gegebenheit (kann sich auch um einen Block handeln) auf dem Stapel an, kopiert diese und legt die Kopie auf den Stapel.

Bei diesem Kommando wird bei der Gegebenheit eine Kopie angefordert. Immutable Gegebenheiten geben einfach ihre Adresse weiter. In solchen Fällen ist das Verhalten von *DUP* (siehe S. 22) und *CPY* identisch. Veränderliche Gegebenheiten erschaffen eine echte Kopie von sich, also ein neues Objekt, dessen Adresse dann auf dem Stapel abgelegt wird. Änderungen an einer Kopie finden sich nicht im Original wieder und umgekehrt.

Laufzeit

⟨$Text_{Zk}$⟩[[Laufzeit]]

Nimmt eine Zeichenkette vom Stapel und verwendet diese als Text für die Ausgabe von Laufzeitinformationen (diese Beziehen sich immer auf die erste, bei Programmstart festgestellte Startzeit). Die Informationen werden sowohl auf der Konsole als auch in die HTML-Datei ausgegeben. In letzterer finden sich dann zudem auch noch Speicherinformationen. Bevor diese erhoben werden, wird die automatische Speicherbereinigung aufgerufen.

DEL
löschen

⟨$Pronomen_{Zk}$⟩[[DEL]]

Nimmt eine Zeichenkette vom Stapel. Diese muß im lokalen Lexikon des aktuellen Interpreters mit einer Gegebenheit verknüpft sein (ansonsten Fehlermeldung und Programmabbruch). Das Pronomen wird aus dem lokalen Lexikon gelöscht und steht danach nicht mehr zur Verfügung.

ERASE
Löschung

⟨*Pronomen*$_{Zk}$⟩[[ERASE]]

Nimmt eine Zeichenkette vom Stapel. Diese muß im globalen Lexikon mit einer Gegebenheit verknüpft sein (ansonsten Fehlermeldung und Programmabbruch). Das Pronomen wird aus dem globalen Lexikon gelöscht und steht danach nicht mehr zur Verfügung.

ROT
rotieren

⟨*erste*⟩⟨*zweite*⟩⟨*dritte*⟩[[ROT]] ⟨*zweite*⟩⟨*dritte*⟩⟨*erste*⟩

Nimmt drei beliebige Gegebenheiten vom Stapel und legt diese in veränderter Reihenfolge wieder darauf, so daß die zuletzt vom Stapel genommene Gegebenheit auch wieder als letzte daraufgelegt wird. Die anderen beiden Gegebenheiten verbleiben in ihrer Reihenfolge.

Synonym

⟨*Kommando*$_{Zk}$⟩⟨*Alias*$_{Zk}$⟩[[Synonym]]

Nimmt zwei Zeichenketten vom Stapel. Das *Kommando* muß bereits ein gültiges Verb sein (ansonsten Fehlermeldung und Programmabbruch). Der *Alias* wird als weitere Bezeichnung des Kommandos gespeichert, kann danach ebenfalls als Verb verwendet werden.

Handelt es sich beim *Alias* bereits um ein Verb, so wird diese Zuordnung überschrieben. Handelte es sich gar um die einzige Bezeichnung eines Kommandos, so ist dieses nach der Zuordnung zu dem anderen Kommando nicht mehr erreichbar.

tiefstapeln

⟦tiefstapeln⟧

Gibt gegebenenfalls eine Liste der im Keller liegenden Datentypen in die HTML-Datei aus. Das Kommando ist vor allem für die Fehlersuche gedacht.

trennen

$\langle Verb_{Zk} \rangle$⟦trennen⟧

Nimmt eine Zeichenkette vom Stapel. Diese muß im Wörterbuch als Verb mit einem Kommando verknüpft sein (ansonsten Fehlermeldung und Programmabbruch). Nach der Überprüfung wird die Zeichenkette aus dem Wörterbuch gelöscht und steht danach nicht mehr als Verb zur Verfügung.

Handelte es sich um das einzige mit einem bestimmten Kommando verknüpfte Verb, so ist das Kommando nach der Trennung nicht mehr erreichbar.

OVER
überspringen

⟨*erste*⟩⟨*zweite*⟩⟦OVER⟧ ⟨*erste*⟩

Schaut sich von mindestens zwei Gegebenheiten auf dem Stapel die zweitoberste an und legt sie auf den Stapel.

CMD
verbalisieren

⟨$Verb_{Zk}$⟩⟪...⟫⟦CMD⟧

Nimmt eine Anführung und eine Zeichenkette vom Stapel und verbindet die beiden zu einem Kommando, welches unter der Zeichenkette als Verb im Wörterbuch gespeichert wird. Danach kann es wie jedes andere Verb verwendet werden.

SWAP
vertauschen

⟨*erstes*⟩⟨*zweites*⟩⟦SWAP⟧ ⟨*zweites*⟩⟨*erstes*⟩

Nimmt zwei beliebige Gegebenheiten (kann sich jeweils auch um einen Block handeln) vom Stapel und legt diese vertauscht wieder zurück auf den Stapel.

.
TYPE
zeigen

⟨*Gegebenheit*⟩[[.]]

Schaut sich eine beliebige Gegebenheit (kann sich auch um einen Block handeln) auf dem Stapel an und gibt deren Inhalt in der HTML-Datei aus. In welcher Form diese Ausgabe erfolgt, hängt vom Typ der Gegebenheit ab.

Rechnen

abrunden

⟨$Zahl_{Zl}$⟩[[abrunden]] ⟨$Ganzzahl_{Zl}$⟩

Nimmt eine Zahl vom Stapel und rundet diese zur nächsten Ganzzahl (immer noch eine Fließkommazahl) ab, welche auf den Stapel gelegt wird.

+
ADD
plus
addieren

⟨$Augend_{Zl}$⟩⟨$Addend_{Zl}$⟩[[+]] ⟨$Summe_{Zl}$⟩

Nimmt zwei Zahlen vom Stapel und addiert den Addend zum Augend. Das Ergebnis wird auf dem Stapel abgelegt (Zahl).

ACOS
Arcuscosinus

⟨*Verhältnis*$_{Zl}$⟩⟦ACOS⟧ ⟨*Winkel*$_{Zl}$⟩

Nimmt eine Zahl vom Stapel, interpretiert diese als Verhältnis von Ankathete zur Hypothenuse und errechnet daraus den Winkel in Grad, der auf den Stapel gelegt wird (Zahl).

ASIN
Arcussinus

⟨*Verhältnis*$_{Zl}$⟩⟦ASIN⟧ ⟨*Winkel*$_{Zl}$⟩

Nimmt eine Zahl vom Stapel, interpretiert diese als Verhältnis von Gegenkathete zur Hypothenuse und errechnet daraus den Winkel in Grad, der auf den Stapel gelegt wird (Zahl).

aufrunden

⟨*Zahl*$_{Zl}$⟩⟦aufrunden⟧ ⟨*Ganzzahl*$_{Zl}$⟩

Nimmt eine Zahl vom Stapel und rundet diese zur nächsten Ganzzahl (immer noch eine Fließkommazahl) auf, welche auf den Stapel gelegt wird.

ABS
Betrag

⟨*Wert*$_{Zl}$⟩⟦ABS⟧ ⟨*Betrag*$_{Zl}$⟩

Nimmt eine Zahl vom Stapel, ermittelt deren Absolutwert und legt diesen auf den Stapel (Zahl).

COS
Cosinus

⟨$Winkel_{Zl}$⟩⟦COS⟧ ⟨$Verhältnis_{Zl}$⟩

Nimmt eine Zahl vom Stapel, interpretiert sie als eine Winkelangabe in Grad und errechnet dazu das Verhältnis von Ankathete zur Hypothenuse, welches auf den Stapel gelegt wird (Zahl).

--
dekrementieren

⟨$Minuend_{Zl}$⟩⟦--⟧ ⟨$Differenz_{Zl}$⟩

Nimmt eine Zahl vom Stapel, zieht Eins von ihr ab und legt das Ergebnis auf den Stapel (Zahl).

/
÷ (Strg+Shift+u00f7)
/ (Strg+Shift+u2215)
DIV
durch
dividieren

⟨$Dividend_{Zl}$⟩⟨$Divisor_{Zl}$⟩⟦÷⟧ ⟨$Quotient_{Zl}$⟩

Nimmt zwei Zahlen vom Stapel, bildet daraus den Quotient und legt diesen auf den Stapel. Ein Versuch, durch Null zu dividieren, führt zu einer Fehlermeldung mit Programmabbruch.

Distanz
Hypothenuse

⟨a_{Zl}⟩⟨b_{Zl}⟩[[Distanz]] ⟨c_{Zl}⟩

Nimmt zwei Zahlen vom Stapel, interpretiert diese als Längen der Katheten in einem rechtwinkligen Dreieck und errechnet daraus gemäß des Satzes des Pythagoras die Länge der Hypothenuse (Zahl).

++
inkrementieren

⟨$Augend_{Zl}$⟩[[++]] ⟨$Summe_{Zl}$⟩

Nimmt eine Zahl vom Stapel, addiert Eins dazu und legt das Ergebnis auf den Stapel (Zahl).

ln
Logarithmus

⟨$Numerus_{Zl}$⟩[[ln]] ⟨$Logarithmus_{Zl}$⟩

Nimmt eine Zahl vom Stapel, bildet dazu den natürlichen Logarithmus und legt diesen auf den Stapel (Zahl).

MAX
Maximum

⟨$erste_{Zl}$⟩⟨$zweite_{Zl}$⟩[[MAX]] ⟨$Maximum_{Zl}$⟩

Nimmt zwei Zahlen vom Stapel, ermittelt die größere von beiden und legt diese auf den Stapel.

MIN
Minimum

⟨$erste_{Zl}$⟩⟨$zweite_{Zl}$⟩[[MIN]] ⟨$Minimum_{Zl}$⟩

Nimmt zwei Zahlen vom Stapel, ermittelt die kleinere von beiden und legt diese auf den Stapel.

MOD
Modulo

⟨$Dividend_{Zl}$⟩⟨$Divisor_{Zl}$⟩[[MOD]] ⟨$Nachkommarest_{Zl}$⟩

Nimmt zwei Zahlen vom Stapel, bildet daraus den Quotienten und legt dessen Nachkommaanteil auf den Stapel. Ein Divisor von Null führt zu einer Fehlermeldung und Programmabbruch.

*
× (Strg+Shift+u00d7)
· (Strg+Shift+u2219)

MUL
mal
multiplizieren

⟨$Multiplikator_{Zl}$⟩⟨$Multiplikand_{Zl}$⟩[[×]] ⟨$Produkt_{Zl}$⟩

Nimmt zwei Zahlen vom Stapel, bildet daraus das Produkt und legt dieses auf den Stapel (Zahl).

GAUSSIAN
Normalverteilung

⟦GAUSSIAN⟧	⟨$Zufallszahl_{Zl}$⟩

Schafft eine normalverteilte Zufallszahl mit dem Median Null und einer Standardabweichung von Eins und legt diese auf den Stapel.

99,73% aller Zufallszahlen werden sich im Bereich von ± 3 bewegen. Auch wenn dies ziemlich unwahrscheinlich ist, sind Ausreißer mit extremen Werten möglich. Insofern diese ausgeschlossen werden müssen, ist das Ergebnis zu kontrollieren und gegebenenfalls zu korrigieren.

Um die Zufallszahl an eine andere Standardabweichung anzupassen, ist diese mit der gewünschten Abweichung zu multiplizieren. Ein anderer Median ist zu erreichen, indem der gewünschte Wert zu der Zufallszahl addiert wird.

**

POW
potenzieren

⟨$Basis_{Zl}$⟩⟨$Exponent_{Zl}$⟩⟦**⟧	⟨$Potenz_{Zl}$⟩

Nimmt zwei Zahlen vom Stapel und bildet daraus die Potenz, die auf dem Stapel abgelegt wird (Zahl).

√ (Strg+Shift+u221a)

SQRT
Wurzel
Quadratwurzel

⟨*Radikand*$_{Zl}$⟩⟦√⟧ ⟨*Wurzel*$_{Zl}$⟩

Nimmt eine Zahl vom Stapel, zieht davon die Quadratwurzel und legt diese auf den Stapel (Zahl).

runden

⟨*Zahl*$_{Zl}$⟩⟦runden⟧ ⟨*Ganzzahl*$_{Zl}$⟩

Nimmt eine Zahl vom Stapel und rundet sie zu einer Ganzzahl (immer noch eine Fließkommazahl), die auf den Stapel gelegt wird.

SIN
Sinus

⟨*Winkel*$_{Zl}$⟩⟦SIN⟧ ⟨*Verhältnis*$_{Zl}$⟩

Nimmt eine Zahl vom Stapel, interpretiert sie als eine Winkelangabe in Grad und errechnet dazu das Verhältnis von Gegenkathete zur Hypothenuse, welches auf den Stapel gelegt wird (Zahl).

-
./. (Strg+Shift+u2052)
− (Strg+Shift+u2212)

SUB
minus
subtrahieren

⟨$Minuend_{Zl}$⟩⟨$Subtrahend_{Zl}$⟩[[./.]]	⟨$Differenz_{Zl}$⟩

Nimmt zwei Zahlen vom Stapel, bildet daraus die Differenz und legt diese auf den Stapel (Zahl).

RANDOM
Zufallszahl

[[RANDOM]]	⟨$Zufallszahl_{Zl}$⟩

Schafft eine Zufallszahl zwischen Null (inklusive) und Eins (exklusive) und legt diese auf den Stapel. Es handelt sich um eine Zufallszahl mit gleichmäßiger Verteilung.

Insofern eine Zufallszahl aus einem bestimmten Zahlenbereich benötigt wird, könnte z. B. solch ein Makro hilfreich sein:

⟨Zfz⟩⟪[[OVER]][[./.]][[RANDOM]][[×]][[+]]⟫[[CMD]]
das folgendermaßen anzuwenden wäre:
⟨von_{Zl}⟩⟨bis_{Zl}⟩[[Zfz]] ⟨$Zufallszahl_{Zl}$⟩

Textverarbeitung

digitize

⟨$Zahl_{Zl}$⟩⟦digitize⟧ ⟨$Zahl_{Zk}$⟩

Nimmt eine Zahl vom Stapel und gibt diese als Zeichenkette aus, welche auf den Stapel gelegt wird. Die Ausgabe ist bis auf sechs Stellen hinter dem hier verwendeten Dezimalpunkt genau.

Dieses Kommando ermöglicht die Ausgabe von Zahlen als Zeichenketten, wie sie namentlich in SVG-Dateien benötigt werden.

verbinden

⟨$Anfang_{Zk}$⟩⟨$Fortsetzung_{Zk}$⟩⟦verbinden⟧ ⟨$Ergebnis_{Zk}$⟩

Nimmt zwei Zeichenketten vom Stapel, verbindet diese zu einer und legt diese auf den Stapel.

Ziffernfolge

⟨$Zahl_{Zl}$⟩⟦Ziffernfolge⟧ ⟨$Zahl_{Zk}$⟩

Nimmt eine Zahl vom Stapel und gibt diese in eine Zeichenkette aus, welche auf den Stapel gelegt wird. Die Ausgabe ist bis auf sechs Stellen nach dem Komma genau.

Bedingungen

Datentypen

Wahrheitswert

Ein Wahrheitswert ist der erste Datentyp in *Lysa*, der nicht direkt eingegeben werden kann, sondern nur durch Verben zu erzeugen ist. Auch Wahrheitswerte sind immutabel. Sie kennen nur die beiden Zustände *richtig* und *falsch*.

Zähler

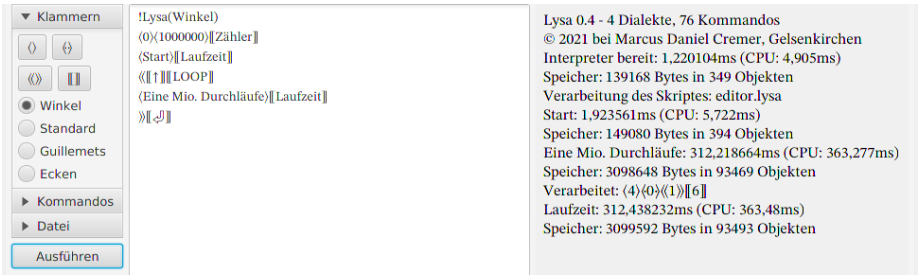
Ein Zähler ist ein mutabler numerischer (ganzzahliger) Datentyp, der insbesondere als Schleifenzähler gedacht ist. Er kann ebenfalls nur durch ein Verb erzeugt werden und wird dabei auch gleich mit einem Startwert und einem Ziel initialisiert. Andere Verben inkrementieren oder dekrementieren den Stand und geben einen Wahrheitswert zurück, ob der Stand kleiner oder größer als das Ziel ist. Ein Zählerstand kann jedoch auch als Zahl abgefragt werden. Vergleichen lassen sich Zähler mit anderen Zählern oder auch mit Zahlen (wobei sich Zahlen nicht mit Zählern vergleichen lassen, also muß bei Vergleichen mit Zahlen der Zähler immer links stehen; die Zahl wird dabei gerundet).

Wird ein Schleifenzähler mit einer Zahl realisiert, wird bei jeder Veränderung eine neue Zahl generiert (der Preis der Immutabilität):

Im Vergleich dazu dasselbe Skript verarbeitet auf demselben Rechner mit der in Java geschriebenen Version 0.3:

Lysa 0.3, 2020.11.14
© 2020 bei Marcus Daniel Cremer, Gelsenkirchen
4 Dialekte geladen!
10 Protokolle geladen!
131 Kommandos geladen!
Verarbeitung des Skriptes: test.lysa
Gestartet: 30.01.2021, 17:41:14 (1612024874)
Start: PT0.742327S (CPU: PT2.02S)
Eine Mio. Durchläufe: PT1.558926S (CPU: PT3.19S)
Verarbeitet: ⟨6⟩⟨2⟩⟪1⟫⟦8⟧
Speicher: (belegt) 86179624 Bytes, (frei) 989659352 Bytes
Laufzeit: PT1.560561S (CPU: PT3.19S)

Unter dem Einsatz eines Zähler verkürzt sich die Laufzeit nochmals spürbar:



??
IFELSE
entscheiden

$\langle Bedingung_{Ww} \rangle \langle\langle positiv \rangle\rangle \langle\langle negativ \rangle\rangle [\![??]\!]$

Nimmt zwei Blöcke und einen Wahrheitswert vom Stapel. Falls der Wahrheitswert positiv ist, wird der eine Block effektuiert, ansonsten der andere.

?
IF
wenn
falls

$\langle Bedingung_{Ww} \rangle \langle\langle \ldots \rangle\rangle [\![?]\!]$

Nimmt einen Block und einen Wahrheitswert vom Stapel. Falls der Wahrheitswert positiv ist, wird der Block effektuiert, ansonsten wird er ignoriert.

⊭ (Strg+Shift+u22ad)

falsch

⟦⊭⟧	⟨$falsch_{Ww}$⟩

Generiert einen negativen Wahrheitswert und legt ihn auf den Stapel.

=
EQ
gleich

⟨$erste$⟩⟨$zweite$⟩⟦=⟧	⟨$Ergebnis_{Ww}$⟩

Nimmt zwei beliebige Gegebenheiten vom Stapel und vergleicht sie miteinander. Insofern sie gleich sind, wird ein positiver Wahrheitswert auf den Stapel gelegt, ansonsten ein negativer

Prinzipiell lassen sich alle Gegebenheiten miteinander vergleichen, aber nicht bei allen macht es Sinn. Handelt es sich um zwei Gegebenheiten unterschiedlichen Typs, wird stets Gleichheit zurückgegeben (sind ja beides Gegebenheiten), gleiches gilt z. B. auch für Blöcke oder Platzhalter jeweils untereinander. Sinnvoll vergleichen lassen sich nur Zahlen, Zähler, Zeichenketten und Wahrheitswerte untereinander. Bei letzteren ist ein *richtig* größer als ein *falsch*. Bei komplexeren Datentypen wird die Vergleichbarkeit jeweils in der Beschreibung aufgeführt.

Leider steht für den Vergleich von Zeichenketten keine Umsetzung der lokalen lexikalischen Reihenfolge zur Verfügung.

>
GT
größer

⟨*erste*⟩⟨*zweite*⟩⟦>⟧ ⟨$Ergebnis_{Ww}$⟩

Nimmt zwei beliebige Gegebenheiten vom Stapel und vergleicht sie miteinander. Insofern das zweite (von rechts aus gesehen) größer ist als das erste, wird ein positiver Wahrheitswert auf den Stapel gelegt, ansonsten ein negativer.

Um zwei Gegebenheiten dahingehend zu vergleichen, ob sie größer oder gleich sind, ist ein Makro zu verwenden:

⟨≥⟩⟪⟦<⟧⟦¬⟧⟫⟦CMD⟧

<
LT
kleiner

⟨*erste*⟩⟨*zweite*⟩⟦<⟧ ⟨$Ergebnis_{Ww}$⟩

Nimmt zwei beliebige Gegebenheiten vom Stapel und vergleicht sie miteinander. Insofern das zweite (von rechts aus gesehen) kleiner ist als das erste, wird ein positiver Wahrheitswert auf den Stapel gelegt, ansonsten ein negativer.

Um zwei Gegebenheiten dahingehend zu vergleichen, ob sie kleiner oder gleich sind, ist ein Makro zu verwenden:

⟨≤⟩⟪⟦>⟧⟦¬⟧⟫⟦CMD⟧

¬ (Strg+Shift+u00ac)

NOT
negieren

$\langle Urteil_{Ww} \rangle [\![\neg]\!] \quad \langle Verkehrung_{Ww} \rangle$

Nimmt einen Wahrheitswert vom Stapel, verkehrt ihn in sein Gegenteil und legt dieses auf den Stapel.

∨ (Strg+Shift+u2228)

OR
oder

$\langle erster_{Ww} \rangle \langle zweiter_{Ww} \rangle [\![\vee]\!] \quad \langle Ergebnis_{Ww} \rangle$

Nimmt zwei Wahrheitswerte vom Stapel und legt einen positiven Wahrheitswert auf den Stapel, falls mindestens einer der beiden positiv ist, ansonsten einen negativen.

↑ (Strg+Shift+u2191)

raufzählen

$\langle Zähler_{Zr} \rangle [\![\uparrow]\!] \quad \langle weiter_{Ww} \rangle$

Schaut sich einen Zähler auf dem Stapel an, inkrementiert diesen und legt auf den Stapel, ob dieser noch unterhalb des vorgegebenen Zieles liegt (Wahrheitswert).

$\models$ (Strg+Shift+u22a8)

richtig

$[\![\models]\!]$	$\langle richtig_{Ww} \rangle$

Generiert einen positiven Wahrheitswert und legt ihn auf den Stapel.

$\downarrow$ (Strg+Shift+u2193)

runterzählen

$\langle Zähler_{Zr} \rangle [\![\downarrow]\!]$	$\langle weiter_{Ww} \rangle$

Schaut sich einen Zähler auf dem Stapel an, dekrementiert diesen und legt auf den Stapel, ob dieser noch oberhalb des vorgegebenen Zieles liegt (Wahrheitswert).

Stand

$\langle Zähler_{Zr} \rangle [\![\text{Stand}]\!]$	$\langle Wert_{Zl} \rangle$

Schaut sich einen Zähler auf dem Stapel an, ermittelt dessen Stand, verwandelt diesen in eine Zahl und legt ihn auf den Stapel.

$\wedge$ (Strg+Shift+u2227)

AND
und

$\langle erster_{Ww} \rangle \langle zweiter_{Ww} \rangle [\![\wedge]\!]$	$\langle Ergebnis_{Ww} \rangle$

Nimmt zwei Wahrheitswerte vom Stapel und legt einen positiven Wahrheitswert auf den Stapel, falls beide positiv sind, ansonsten einen negativen.

⇝ (Strg+Shift+u21b7)
BREAK
verlassen

$$\langle\langle\langle Bedingung_{Ww}\rangle[\![\rightsquigarrow]\!]\ldots\rangle\rangle \qquad (\langle Abbruchsignal_{Zk}\rangle)$$

Nimmt einen Wahrheitswert vom Stapel. Ist dieser positiv, wird ein Abbruchsignal (Zeichenkette) auf den Stapel gelegt, welches allerdings sofort wieder entfernt wird, insofern das Kommando innerhalb eines Blocks steht. Der Block wird dann unmittelbar verlassen. Eventuell noch folgende Kommandos werden nicht mehr ausgeführt.

↺ (Strg+Shift+u27f2)
LOOP
wiederholen

$$\langle\langle\ldots\langle Bedingung_{Ww}\rangle[\![\circlearrowleft]\!]\ldots\rangle\rangle \qquad (\langle Wiederholungssignal_{Zk}\rangle)$$

Nimmt einen Wahrheitswert vom Stapel. Ist dieser positiv, wird ein Wiederholungssignal (Zeichenkette) auf den Stapel gelegt, welches jedoch innerhalb eines Blocks sogleich entfernt wird. Die Abarbeitung des Blocks wird an dieser Stelle unterbrochen und von vorne begonnen. Kommandos, die hinter dem Wiederholungskommando stehen, werden also erst beim letzten Durchlauf des Blocks ausgeführt, wenn kein Signal mehr auf den Stapel gelegt wurde.

Zähler

⟨$Start_{Zl}$⟩⟨$Ziel_{Zl}$⟩[[Zähler]] ⟨$Zähler_{Zr}$⟩

Nimmt zwei Zahlen vom Stapel, rundet sie und initialisiert mit ihnen einen Zähler, der auf den Stapel gelegt wird.

Sammlungen

Datentypen

Sammlung

Eine Sammlung ist ein mutabler Datentyp, der beliebige Gegebenheiten (außer andere Sammlungen) aufnehmen kann. Diese müssen allerdings alle desselben Typs sein. Gegebenheiten welchen Typs eine Sammlung beinhaltet, entscheidet sich erst mit der ersten hinzugefügten Gegebenheit. Danach werden keine Elemente eines anderen Typs mehr akzeptiert (Fehlermeldung, Programmabbruch). Eine geleerte Sammlung kann bei einer Wiederbefüllung allerdings dann wieder Elemente eines anderen Typs aufnehmen.

Sammlungen können untereinander über ihre Mächtigkeit verglichen werden.

Konglomerat

Ein Konglomerat ist ein zusammengesetzter Datentyp. Er enthält Paare von Schlüsseln (Zeichenketten) und beliebigen Gegebenheiten. Ein Konglomerat ist mutabel und erst einmal nicht untereinander vergleichbar. Es ist jedoch möglich, einen Schlüssel für ein vergleichbares Feld zu setzen, wodurch dann auch Konglomerate vergleichbar sind (wenn bei beiden der gleiche Schlüssel für die Vergleichbarkeit angegeben wurde).

ausgeben

⟨*Konglomerat*$_{Kg}$⟩⟨*Schlüssel*$_{Zk}$⟩[[ausgeben]] ⟨*Gegebenheit*⟩

Nimmt eine Zeichenkette vom Stapel und schaut sich von dort ein Konglomerat an, von dem es die zu der Zeichenkette gehörende Gegebenheit ermittelt und auf dem Stapel ablegt. Insofern zu dem Schlüssel keine Gegebenheit in dem Konglomerat gibt, erfolgt nach einer Fehlermeldung ein Programmabbruch.

belegt

⟨*Sammlung*$_{Sl}$⟩[[belegt]] ⟨*Auskunft*$_{Ww}$⟩

Schaut sich eine Sammlung vom Stapel an und legt auf den Stapel, ob diese mindestens eine Gegebenheit enthält (Wahrheitswert).

∋ (Strg+Shift+u220b)

einsammeln

⟨*Sammlung*$_{Sl}$⟩[⟨*Gegebenheit*⟩][[∋]] ⟨*Sammlung*$_{Sl}$⟩

Nimmt beliebig viele Gegebenheiten gleichen Typs vom Stapel (mindestens eine), bis es auf eine Sammlung stößt. In diese werden die Gegebenheit eingetragen (in der Reihenfolge, in der sie geschrieben stehen), wonach die Sammlung wieder auf den Stapel gelegt wird.

Mögliche Fehlerquellen sind, daß zwischen Verb und Sammlung keine Gegebenheiten einheitlichen Typs liegen, daß in der Sammlung bereits Gegebenheiten eines anderen Typs gespeichert sind, oder daß gar keine Sammlung auf dem Stapel liegt. Dabei kommt es dann jeweils zu einer Fehlermeldung und das Programm wird abgebrochen.

enthält

⟨*Konglomerat*$_{Kg}$⟩⟨*Schlüssel*$_{Zk}$⟩[[enthält]] ⟨*Auskunft*$_{Ww}$⟩

Nimmt eine Zeichenkette vom Stapel und schaut sich von dort ein Konglomerat an. Es überprüft, ob dieses zu der Zeichenkette als Schlüssel einen Wert enthält und legt das Ergebnis auf den Stapel (Wahrheitswert).

FIFO

⟨*Sammlung*$_{Sl}$⟩[[FIFO]] ⟨*Gegebenheit*⟩

Schaut sich eine Sammlung vom Stapel an, entfernt daraus deren erste Gegebenheit und legt diese auf den Stapel.

∈ (Strg+Shift+u2208) hinzufügen

⟨*Konglomerat*$_{Kg}$⟩⟨*Schlüssel*$_{Zk}$⟩⟨*Gegebenheit*⟩[[∈]]

Nimmt eine beliebige Gegebenheit und eine Zeichenkette vom Stapel und schaut sich von dort ein Konglomerat an, in welches es die Zeichenkette und die Gegebenheit als Schlüssel-Wertepaar hinzufügt.

Konglomerat

[[Konglomerat]] ⟨*Konglomerat*$_{Kg}$⟩

Generiert ein Konglomerat und legt es auf den Stapel.

LIFO

⟨$Sammlung_{Sl}$⟩[[FIFO]] ⟨*Gegebenheit*⟩

Schaut sich eine Sammlung vom Stapel an, entfernt daraus deren letzte Gegebenheit und legt diese auf den Stapel.

mischen

⟨$Sammlung_{Sl}$⟩[[mischen]]

Schaut sich eine Sammlung vom Stapel an und bringt deren Gegebenheiten in eine zufällige Reihenfolge.

Sammlung

[[Sammlung]] ⟨$Sammlung_{Sl}$⟩

Generiert eine Sammlung und legt diese auf den Stapel.

sortieren

⟨$Sammlung_{Sl}$⟩[[sortieren]]

Schaut sich eine Sammlung vom Stapel an und sortiert deren Gegebenheiten in eine aufsteigende Reihenfolge. Eine Sortierung ist natürlich nur bei vergleichbaren Gelegenheitstypen sinnvoll.

tilgen

⟨*Konglomerat*$_{Kg}$⟩⟨*Schlüssel*$_{Zk}$⟩[[tilgen]]

Nimmt eine Zeichenkette vom Stapel und schaut sich ein Konglomerat davon an. Dann entfernt es das der Zeichenkette entsprechende Schlüssel-Wertepaar aus dem Konglomerat.

Umfang

⟨*Behälter*$_{Sl \vee Kg}$⟩[[Umfang]] ⟨*Anzahl*$_{Zl}$⟩

Schaut sich eine Gegebenheit auf dem Stapel an. Handelt es sich um eine Sammlung oder ein Konglomerat, wird deren Umfang festgestellt und auf den Stapel gelegt (Zahl). Ansonsten gibt es eine Fehlermeldung nebst Programmabbruch.

Vergleichsfeld

⟨*Konglomerat*$_{Kg}$⟩⟨*Schlüssel*$_{Zk}$⟩[[Vergleichsfeld]]

Nimmt eine Zeichenkette vom Stapel und schaut sich dort ein Konglomerat an, dem die Zeichenkette als Schlüssel zur Herstellung einer Vergleichbarkeit übergeben wird. Es wird nicht geprüft, ob auch ein entsprechendes Schlüsselpaar vorhanden ist (dies kann also erst später gesetzt werden).

Pronomenverzeichnis

Φ (Strg+Shift+u03a6)

⟨·Φ·⟩	Goldener Schnitt

Verhältnis der inneren und äußeren Teilung (> 1), besser bekannt als ›Goldener Schnitt‹. Es steht im globalen Lexikon.

π (Strg+Shift+u03c0)

⟨·π·⟩	Kreiszahl

Das Verhältnis von Kreisdurchmesser und -umfang. Es ist im globalen Lexikon zu finden.

e (Strg+Shift+u212f)

⟨·*e*·⟩	Eulersche Zahl

Der ›natürliche‹ Exponent oder auch Eulersche Zahl. Sie steht im globalen Lexikon.