Befehlsliste

Eine vollständige Liste aller Puppet-Befehle

Einleitung

Dieser Text beschreibt in knapper Weise die Beschreibungssprache für CommandPuppets. Es wird vorausgesetzt, dass das Tutorial bereits gelesen wurde.

* Steht im Folgenden ein Wort/Satz in spitzen Klammern so bedeutet dies, dass der Text an dieser Stelle in der Puppetbeschreibung ersetzt werden muss. Beispiel:

LOGIN: <string>

Dies bedeutet, dass man in die Puppetbeschreibung beispielsweise folgendes schreiben kann:

LOGIN: " biegt um die Ecke."

* Eckige Klammern bedeuten, dass der eingeklammerte Teil wegfallen kann. Beispiel:

WHEN CHAT [FROM <liste>] DO <action>

Hier kann man sowohl

WHEN CHAT FROM "berni Yordan BadPritt" DO guckgucks

als auch

WHEN CHAT DO guckgucks

schreiben. Die beiden Befehle haben allerdings unterschiedliche Wirkung.

* Gibt es für eine Stelle mehrere Alternativen, so werden diese durch / getrennt geschrieben. In der Puppetbeschreibung darf hierbei immer nur eine der Alternativen stehen. Beispiel:

NEWOWNER: YES/NO

Definitionen

Strings

Strings werden durch Gänsefüßchen (") umschlossen. Soll innerhalb eines Strings ein Gänsefüßchen stehen, so ist dieses doppelt zu schreiben. Beispiel: 'Dieser Text enth"alt G"ansef"u"schen.' wird so geschrieben: "Dieser Text enth""alt G""ansef""u""schen." Enthält der String keine Leerzeichen, so können die Gänsefüßchen auch entfallen.

Listen

Listen sind nichts anderes als Strings, bei denen die Leerzeichen die Elemente der Liste voneinander trennen.

Kommentare

Kommt in einer Zeile ein Doppelkreuz (#) vor, so wird alles nach diesem Doppelkreuz ignoriert. Dies ist nicht der Fall, wenn das Doppelkreuz in einem String steht, also von Gänsefüßchen umrahmt ist. Weiterhin werden Leer-Zeilen komplett ignoriert.

Kommentare nach ">>"-Befehlen und nach "--"-Befehlen werden nicht ignoriert!

Steuerzeichen

Befinden sich Steuerzeichen in der Eingabe, so werden diese durch einen Punkt (.) ersetzt. Eine Ausnahme bildet das Tab-Steuerzeichen, welches durch ein Leerzeichen ersetzt wird. Leerzeichen und Steuerzeichen am Anfang und am Ende einer Zeile werden entfernt.

Der Aufbau der Beschreibung eines Puppets

Syntax:

PUPPET <name>

<Einstehllungen und @-Befehle>

<actions>

[PUPPETEND]

Vor PUPPET und nach PUPPETEND können noch beliebig viele Zeilen mit unformatierter Information stehen. Dies ist besonders dann hilfreich, wenn der Webspace-Lieferant Werbung auf seiner Seite platziert hat.

Der Name beim Puppet-Befehl wird beim /info-Befehl angezeigt.

Einstellungen

Es gibt eine Reihe von Einstellungen, die schon vor dem Ausführen der ersten Action bekannt sein müssen. Diese können direkt nach dem PUPPET-Befehl angegeben werden:

Syntax:

LOGIN: <string>

LOGOUT: <string>

APPEAR: <string>

DISAPPEAR: <string>

INFO: <string>

OVERFLOW: ERROR/HARAKIRI/IGNORE

NEWOWNER: YES/NO

CITYCHAT: YES/NO

CASESENSITIV: YES/NO

SAVE: <phrase>

DEBUG: <liste>

Die ersten vier Einstellungen legen die entsprechenden Texte fest. Leerzeichen am Anfang nicht vergessen!

INFO legt eine Beschreibung des Puppets fest (wird bei /info angezeigt).

OVERFLOW legt fest, was geschehen soll, wenn das Puppet zu viel Speicher verbraucht. Bei ERROR wird eine ERROR-Action aufgerufen (siehe WHEN ERROR). HARAKIRI (default) bedeuted, dass das Puppet sich einfach beendet. IGNORE macht nix.

NEWOWNER legt fest, ob das Puppet von anderen Leuten geownt werden darf, oder nicht. Wird automatisch auf NO gestellt, wenn CITYCHAT auf YES gestellt wird. Default: YES

CITYCHAT legt fest, ob das Puppet den Citychat des Starters benutzt, oder ob es auf einen Citychat verzichten muss. Default: OFF Wird der CITYCHAT aktiviert, so wird automatisch NEWOWNER auf NO gesetzt.

CASESENSITIV legt fest, ob das Puppet Groß und Kleinschreibung bei den Events CHAT, KEYWORD und MATCH unterscheidet (YES) oder nicht (NO). Default: NO

SAVE macht die Variablen SAVE1 bis SAVE100 resistent. D.h. wenn das Puppet wieder geladen wird, sind die Werte in diesen Variablen wieder da. <phrase> ist dazu da, dass zwei Puppets nicht aus Versehen die gleichen Daten benutzen. Kommt in <phrase> ein * vor, so wird dieser durch den Namen des Puppets ersetzt.

DEBUG schaltet in den DEBUG-Modus. Siehe Debugging.

@-Befehle

Syntax:

@<command>: <action>

Variablen:

PARAM = Die Parameter des Kommandos

WHO = Die Person, die das Kommando ausgelöst hat.

Immer dann, wenn jemand im Raum das @-Kommando <command> eingibt, wird <action> aufgerufen. Ein @-Kommando hat die Syntax:
@<command>[:<puppetname>] <parameter>

Wird <puppetname> weggelassen, so bekommen alle Puppets im Raum das @-Kommando.

In der Variablen PARAM werden die Parameter des Kommandos gespeichert; in WHO die Person, die das Kommando ausgelöst hat. Standardmäßig sollte das Puppet bei Eingabe des Befehls "@list" eine Liste der (@-) Befehle ausspucken, die es kennt. Hierfür kann der "--"-Befehl benutzt werden.

Actions

Syntax:

ACTION <name>

<befehle>

END

Durch diverse Events bzw. Befehle werden Actions aufgerufen. Dies bedeutet, dass die <befehle> der Action der Reihe nach ausgeführt werden.

Das Puppet beginnt direkt nach dem Laden damit, die Action "start" auszuführen, sofern diese existiert.

Die Befehle

In den nachfolgenden WHEN- und IGNORE-Befehlen kann in FROM <liste> immer auch ein "*" angegeben werden. Dies bedeutet, dass der Befehl für alle Leute gilt, für die es keinen anderweitigen Eintrag gibt (mit Ausnahme des Puppets selber). Lässt man FROM <liste> ganz weg, so wird die Action unabhängig davon, wer den Event ausgelöst hat, ausgeführt. Insbesondre kann es passieren, dass ein Event zwei verschiedene Actions aufruft.

Der IGNORE-Befehl macht den entsprechenden WHEN-Befehl rückgängig. Da dies gelegentlich zu Verwechslungen führt, hier ein Beispiel:

WHEN APPEAR * DO gruss

IGNORE APPEAR berni

Manche Leute verstehen das so, dass jetzt alle Leute außer berni gegrüßt wird. Das stimmt nicht. Der erste Befehl sagt, dass wen irgendwer erscheint, soll er gegrüßt werden. Der zweite Befehl besagt, dass ab sofort die Sonderbehandlung von berni (sofern es eine gab) nicht mehr beachtet werden soll und stattdessen auch für berni die Action gruss aufgerufen werden soll.

Die angegebenen Variablen werden vor Ausführung der <action> auf den entsprechenden Wert gesetzt. Dieser Wert wird während der Ausführung nicht verändert (es sei den durch (UN)SET oder EVAL-Befehle).

Es kann bei CHAT, KEYWORD und MATCH passieren, dass der Eintrag in WHO nicht korrekt ist (nur beim Typ CHAT). Dies liegt daran, dass das Puppet versucht, aus der eingegebenen Zeile mit Hilfe eines heuristischen Verfahrens den Sprecher zu ermitteln. Gelegentlich geht dies einfach nicht (etwa //-Befehl).

Syntax:

WHEN CHAT [FROM <liste>] DO <action>

Variablen:

WHO = Der Sprecher

CHAT = Gesamte Chat-Zeile

ROOM = Name des Raumes in dem sich der Sprecher befindet

(bzw. bei CTELL Name der Stadt und bei GTELL Name des Kanals)

TYPE = Art des Chats (CHAT,TELL,CTELL,GTELL)

Wenn jemand aus <liste> Text schreibt, den das Puppet mitbekommt, so wird <action> aufgerufen. Das Puppet reagiert auf sich selbst nur dann, wenn es explizit in <liste> aufgeführt ist.

Syntax:

IGNORE CHAT [FROM <liste>]

Chat von Leuten aus <liste> wird ab sofort ignoriert.

Syntax:

IGNOREALL CHAT

Chat wird komplett ignoriert (Starteinstellung).

Syntax:

WHEN KEYWORD <keywordliste> [FROM <liste>] DO <action>

Variablen:

WHO = Der Sprecher

CHAT = Gesamte Chat-Zeile

ROOM = Name des Raumes in dem sich der Sprecher befindet

(bzw. bei CTELL Name der Stadt und bei GTELL Name des Kanals)

TYPE = Art des Chats (CHAT,TELL,CTELL,GTELL)

KEYWORD = Das Wort, das den Aufruf ausgelöst hat

(Großkleinschreibung wie im Chat).

Wenn jemand aus <liste> Text schreibt, den das Puppet mitbekommt, und der eines der Wörter aus <keywordliste> enthält, so wird <action> aufgerufen. Das Wort muss dabei durch Leerzeichen vom Rest der Zeile abgegrenzt sein. Kommen mehrere (auch gleiche) KEYWORDs vor, so wird für jedes KEYWORD die zugehörige <action> (auch mehrmals) aufgerufen, und zwar von links nach rechts. Das Puppet reagiert auf sich selbst nur, wenn es explizit in <liste> aufgeführt ist.

Syntax:

IGNORE KEYWORD <liste> [FROM <liste>]

Die Schlüsselwörter aus <keywordliste> von Leuten aus <liste> wird ab sofort ignoriert bzw. nicht mehr gesondert behandelt, falls zu diesem Schlüsselwort noch eine "*"-Eintragung existiert.

Syntax:

IGNOREALL KEYWORD [FROM <liste>]

Das Puppet reagiert nicht mehr auf beliebige KEYWORDs von Leuten aus <liste>, bzw. nicht mehr gesondert. Ist <liste> nicht angegeben, reagiert das Puppet auf überhaupt keine KEYWORDS mehr.

Syntax:

WHEN MATCH <match> [FROM <liste>] DO <action>

Variablen:

WHO = Der Sprecher

CHAT = Gesamte Chat-Zeile

ROOM = Name des Raumes in dem sich der Sprecher befindet

(bzw. bei CTELL Name der Stadt und bei GTELL Name des Kanals)

TYPE = Art des Chats (CHAT,TELL,CTELL,GTELL)

SUBST1 = Substitution für erstes * in <match>

SUBST2 = Substitution für zweites * in <match>

...

Zu den Leuten aus <liste> wird abgespeichert, dass <action> ausgeführt werden soll, wenn <match> auf eine Eingabezeile die von der Person stammt, passt. Die Person "*" wird hier etwas unintuitiv verarbeitet, nämlich alle Leute, zu denen es keinen MATCH-Eintrag gibt. Wird "FROM <liste>" ganz weggelassen, so wird eine Eingabezeile unabhängig vom Sprecher abgearbeitet. Das Puppet reagiert auf sich selbst nur, wenn es explizit in <liste> gesetzt ist.

Syntax:

IGNORE MATCH <match> [FROM <liste>]

Der <match> wird ab sofort bei den Leuten aus <liste> nicht mehr überprüft. Wird FROM <liste> weggelassen, so wird <match> allgemein gestrichen.

Syntax:

IGNOREALL MATCH [FROM <liste>]

Das Puppet reagiert nicht mehr auf beliebige MATCHs von Leuten aus <liste>, bzw. nicht mehr gesondert. Ist <liste> nicht angegeben, reagiert das Puppet auf überhaupt keine MATCHs mehr.

Syntax:

WHEN APPEAR <liste> DO <action>

Variablen:

WHO = Die Person, die den Raum betritt

Wenn jemand aus <liste> den Raum betritt wird <action> ausgelöst. Personen, deren Name mit "Geist" beginnt, werden ignoriert (=Leute, die connected haben, aber noch nicht eingeloggt sind).

Syntax:

IGNORE APPEAR <liste>

In Zukunft wird nichts gemacht, wenn jemand aus <liste> den Raum betritt.

Syntax:

IGNOREALL APPEAR

Alle APPEAR-Einträge werden gelöscht (default).

Syntax:

WHEN DISAPPEAR <liste>

Variablen:

WHO = Die Person, die den Raum verlässt

Wenn jemand aus <liste> den Raum verlässt wird <action> ausgelöst. Personen, deren Name mit "Geist" beginnt, werden ignoriert (=Leute, die connected haben, aber noch nicht eingeloggt sind). Vor dem Ausführen von <action> werden folgende Variablen gesetzt:

Syntax:

IGNORE DISAPPEAR <liste>

In Zukunft wird nichts gemacht, wenn jemand aus <liste> den Raum verlässt.

Syntax:

IGNOREALL DISAPPEAR

Alle DISAPPEAR-Einträge werden gelöscht (default).

Syntax:

WHEN NEWROOM DO <action>

Variablen:

ROOM = Die ID des neuen Raumes.

Wenn das Puppet den Raum wechselt, wird <action> ausgeführt.

Syntax:

IGNORE NEWROOM

Raumwechsel werden nicht mehr beachtet.

Syntax:

WHEN TIME <hour>[:<min>] DO <action>

Jedes Mal zu der (Server-)Uhrzeit, die von <hour> und <min> bestimmt wird, wird <action> ausgeführt. Das Puppet kann sich nur eine Uhrzeit merken!

Syntax:

IGNORE TIME

Puppet führt nix mehr zu bestimmten Uhrzeiten aus.

Syntax:

WHEN TIMER <sek> DO <action>

Alle <sek> Sekunden wird die <action> aufgerufen. Das Puppet merkt sich nur einen Timer.

Syntax:

IGNORE TIMER

Der Timer wird abgeschaltet.

Syntax:

WHEN OWNED [FROM <liste>] DO <action>

Variablen:

WHO = Die Person, die das Puppet owned. Siehe:

NEWOWNER-Einstellung

Wenn jemand von <liste> das Puppet owned, wird <action> ausgeführt.

Syntax:

IGNORE OWNED [FROM <liste>]

Die Leute von <liste> werden nicht mehr beim ownen beachtet. Fehlt die Liste ganz, so wird überhaupt nicht mehr auf ownen geachtet.

Syntax:

WHEN UNOWNED DO <action>

Wenn das Puppet freigesetzt wird, wird <action> aufgerufen.

Syntax:

IGNORE UNOWNED

Beim unownen passiert nix.

Syntax:

WHEN ERROR DO <action>

Variablen:

MESSAGE = Die Fehlermeldung (in der Regel eine Java-Fehlermeldung) Siehe

OVERFLOW-Einstellung

Tritt ein Fehler bei der Ausführung eines Puppets auf (Teilen durch 0, etc.) so wird <action> aufgerufen (und zwar sofort und nicht erst nachdem die aktuelle Action beendet wurde). Tritt während der Ausführung von <action> ein Fehler auf, so wird <action> nicht aufgerufen.

Syntax:

IGNORE ERROR

Tritt ein Fehler auf, so wird nix gemacht.

Syntax:

WHEN PING [FROM <liste>] DO <action>

Variablen:

WHO = der Pinger

Wenn jemand aus <liste> das Puppet anpingt, wird <action> ausgelöst.

Syntax:

IGNORE PING [FROM <liste>]

Wenn jemand aus <liste> das Puppet anpingt, wird nichts mehr gemacht. Wird <liste> ganz weggelassen, reagiert das Puppet auf überhaupt keine pings mehr.

Syntax:

WHEN KICKED DO <action>

Variablen:

WHO = der Kicker

Wenn jemand das Puppet kicked, hat das Puppet noch ca. 2 Sekunden Zeit ein paar Befehle auszuführen, bevor es beendet wird. Dafür wird sofort die Action <action> aufgerufen. Stehen noch andere Befehle an, so wird die Abarbeitung dieser zurückgestellt.

Syntax:

IGNORE KICKED

Nix passiert, wenn das Puppet gekickt wird.

Syntax:

SET <var> <string>

Weist der Variablen <var> den String <string> zu.

Syntax:

UNSET <var>

Löscht die Variable <var>

Syntax:

EVAL <var> = <ausdruck>

Berechnet <ausdruck> (siehe Ausdrücke) und weist das Ergebnis der Variablen <var> zu.

Syntax:

SLEEP <sek> [<millisek>]

Verzögert die Ausführung um <sek> Sekunden und <millisek> Millisekunden. Diesen Befehl sollte man von Zeit zu Zeit ausführen, damit das Puppet den Server nicht zu sehr belastet. Während der Verzögerungszeit reagiert das Puppet auf nichts. (Eingetretene Events werden danach abgearbeitet.)

Syntax:

HARAKIRI

Das Puppet beendet sich selbst.

Syntax:

GETDATE

Variablen:

SEC = Sekunden

MIN = Minuten

HOUR = Stunden

DAY = Tag im Monat

MONTH = Monat (als Zahl)

YEAR = Jahr

DAYOFWEEK = Wochentag (Sonntag=1, Montag=2, ..., Samstag=7)

Liefert die aktuelle Uhrzeit und das aktuelle Datum.

Syntax:

GETINFO <name>

Variablen:

CITY = Name der Stadt des Spielers

CITYNR = Nummer der Stadt

LANGUAGE = de/en je nach Einstellung

PUPPET = TRUE, falls es sich um ein Puppet handelt.

PLAYING = TRUE, falls der Spieler ein Spiel spielt.

TITEL = Der Titel des Spielers

RANK = Der Rang des Spielers

SEX = n/m/w je nach Geschlecht des Spielers

TUTOR = TRUE, falls der Spieler ein Tutor ist, sonst FALSE

REPORTER = TRUE, falls der Spieler ein Reporter ist, sonst FALSE

AMT = Amt des Spielers

GILDENAMT = Gildenamt des Spielers

Liefert Informationen zu einem Spieler.

Syntax:

GETWHO

Variablen:

WHO = Liste aller im Raum anwesenden Spieler

Liefert eine Liste der anwesenden Personen.

Syntax:

GETROOMINFO

Variablen:

ROOM = Nummer des Raumes, in dem sich das Puppet befindet.

NAME = Name des Raumes, in dem sich das Puppet befindet.

GAME = Name des Spiels in dem Raum.

Liefert die Nummer, Name und Spiel des Raumes.

Syntax:

WHEREIS PUPPET/OWNER/STARTER

Variablen:

ROOM = Nummer des Raumes, in dem sich die entsprechende Person befindet.

Liefert die Nummer des Raumes, in dem sich das Puppet, der Owner bzw. der Starter des Puppets befindet.

Syntax:

WHOIS PUPPET/OWNER/STARTER

Variablen:

WHO = Name der entsprechende Person.

Liefert den Namen des Puppets, des Owners bzw. des Starters.

Syntax:

MASTERRESET

Führt einen Neustart des Puppets durch. Alle Variablen und alle WHEN-Einstellungen, sowie alle wartenden Events werden gelöscht.

Syntax:

>> <text>

Der <text> wird vom Puppet ausgegeben. Wenn Text mit einem Slash (/) beginnt, so wird der entsprechende Befehl ausgeführt. Folgende Befehle sind erlaubt:

/who

/room

/tell

/mtell

/reset

/shout

/name

/hook

/channel

/gtell

/ctell

/leave

/reset

/game

/blacklist

/inhabitants

/cityinfo

/mutectell

Wird versucht, einen anderen Befehl auszuführen, so wird dieser ignoriert.

Syntax:

-- <text>

Hat nur eine Wirkung, wenn die Action durch einen @-Befehl ausgelöst wurde. In diesem Fall bekommt nur derjenige, der den @-Befehl eingegeben hat den <text> als "--"-Meldung.

Struktur-Befehle

Syntax:

DO <action>

Die Abarbeitung der Befehle der aktuellen Action wird unterbrochen und die Befehle aus <action> ausgeführt.

Syntax:

IF <ausdruck>

<befehl>

[

ELSE

<befehl>

]

oder

IF <ausdruck>

BEGIN

<befehle>

END

[

ELSE

BEGIN

<befehle>

END

]

Liefert <ausdruck> TRUE zurück, so wird <befehl> bzw. <befehle> ausgeführt. Falls der ELSE-Teil angegeben ist, wird der <befehl> dort ausgeführt, falls <ausdruck> nicht TRUE ist. Achtung, bei verschachtelten IF-ELSE-Anweisungen kann es passieren, dass nicht klar ist, zu welchen IF ein ELSE gehört. Hier wird immer das äußerste IF genommen.

Syntax:

WHILE <ausdruck> DO

<befehl>

oder

WHILE <ausdruck> DO

BEGIN

<befehle>

END

Solange <ausdruck> TRUE liefert, wird <befehl> bzw. <befehle> ausgeführt.

Syntax:

FOR <var> IN <liste> DO

<befehl>

oder

FOR <var> IN <liste> DO

BEGIN

<befehle>

END

Für alle Elemente von <liste> wird <befehl> bzw. <befehle> ausgeführt. Dabei enthält die Variable <var> das entsprechende Element.

Syntax:

RETURN

Beendet die Ausführung einer Action.

Variablenersetzung

Kommt in einem der Parameter irgend eines Befehls ein eckiges Klammernpaar vor, so wird vor Ausführung des Befehls der Variablenname zwischen den Klammern durch dessen Wert ersetzt. Eckige Klammern können verschachtelt werden. Möchte man eckige Klammern nicht ersetzen lassen, so muss man sie durch einen Backslash (\) vor der Klammer zu einer normalen Klammer machen. Einem Backslash muss man ebenfalls einen (weiteren) Backslash voranstellen.

Ausdrücke

Wichtig bei Ausdrücken ist, dass alle Symbole/Variablen/Konstanten etc. durch Leerzeichen voneinander abgetrennt werden.

Syntax:

<boolean> OR <boolean>

Liefert TRUE, falls einer der beiden Wahrheitswerte TRUE ist, ansonsten FALSE.

Syntax:

<boolean> AND <boolean>

Liefert TRUE, falls beide Wahrheitswerte TRUE sind, ansonsten FALSE.

Syntax:

NOT <boolean>

Liefert TRUE, wenn der Wahrheitswert FALSE ist und FALSE sonst.

Syntax:

ISEMPTY <string>

Liefert TRUE, falls <string> der leere String ("") ist, FALSE sonst.

Syntax:

EXISTS <var>

Liefert TRUE, falls die Variable mit dem Namen <var> einen Wert enthält, ansonsten FALSE.

Syntax:

ISINTEGER <string>

Liefert TRUE, falls <string> einen Integerwert enthält, sonst FALSE.

Syntax:

<string> == <string>

Liefert TRUE, falls beide Strings gleich sind, ansonsten FALSE.

Syntax

<string> != <string>

Liefert TRUE, falls beide Strings verschieden sind, ansonsten FALSE.

Syntax

<int> < <int>

<int> > <int>

<int> <= <int>

<int> >= <int>

Liefert TRUE, falls die erste Zahl kleiner (größer, kleinergleich, größergleich) als die zweite ist, ansonsten FALSE.

Syntax:

<string> IN <string>

Liefert TRUE, falls der erste Sting im zweiten String enthalten ist, ansonsten FALSE.

Syntax:

<string> INLIST <liste>

Liefert TRUE, falls <string> ein Element von <liste> ist, ansonsten FALSE.

Syntax:

<string> STARTSWITH <string>

<string> ENDSWITH <string>

Liefert TRUE, falls der erste String mit dem zweiten String beginnt (endet) und ansonsten FALSE.

Syntax:

FIRSTOF <liste>

LASTOF <liste>

FIRSTCHAROF <string>

LASTCHAROF <string>

Liefert das erste (letzte) Element/Zeichen von <liste>/<string>.

Syntax:

WITHOUTFIRST <liste>

WITHOUTLAST <liste>

WITHOUTFIRSTCHAR <string>

WITHOUTLASTCHAR <string>

Liefert alle Elemente/Zeichen von <liste>/<string>, ausser dem Ersten (Letzten).

Syntax:

LOWERCASE <string>

UPPERCASE <string>

Liefert einen String, bei dem alle Zeichen in <string> klein/groß geschrieben sind. (ß wird zu SS bei UPPERCASE).

Syntax:

LENGTH <string>

LISTLENGTH <liste>

Liefert die Länge von <string>/<liste>, d. h. die Anzahl der Zeichen/Elemente.

Syntax:

<n> ELEMENTOF <liste>

<n> CHAROF <string>

Liefert das <n>-te Listenelement/Zeichen von <liste>/<string>

Syntax:

<int> + <int>

<int> - <int>

<int> * <int>

<int> / <int>

<int> % <int>

Liefert das entsprechende Ergebnis. Das Puppet beherrscht die Punkt-vor-Strich-Regel.

Syntax:

- <int>

+ <int>

Unäres Minus/Plus.

Syntax:

(<ausdruck>)

```
Liefert <ausdruck> zurück.
```

Debugging

Enthält das Puppet die Einstellung DEBUG, so wird mit dem Start des Puppets ein Debugging-Fenster geöffnet. In diesem Fenster wird unterschiedliche Debugging-Information angezeigt. Welche, steuert die Liste, die man bei der DEBUG-Einstellung angibt. Diese kann zwischen keinem und allen der folgenden Werte enthalten:

ACTION

EVENT

VARIABLES

SINGLESTEP

Die letzten beiden werden nur aktiv, wenn im Programm irgendwo der Befehl DEBUG ON steht. Mit DEBUG OFF kann man die Wirkung dieser beiden Werte wieder deaktivieren.

ACTION

Wenn eine Action aufgerufen wird, so wird dies durch

A <name> [(sofort)]

angezeigt. (sofort) wird angegeben, wenn die Action vorrangig ausgeführt wird (siehe etwa WHEN KICKED, aber auch DO).

EVENT

Jedes Mal, wenn ein Event auftritt (unabhängig davon, ob der Event auch mit einem WHEN-Befehl abgefragt wird) wird ein

E <typ>: <weitere Infos>

ausgegeben. <typ> gibt an, was für ein Event es ist, und abhängig davon, gibt <weitere Infos> weitere Informationen dazu an.

VARIABLES

Gibt bei jeder Änderung einer Variablen

V <name>: <alterwert> => <neuerwert>

aus.

SINGLESTEP

Gibt jeden einzelnen Befehl durch

C <name>: <parameter> (<Zeilennummer>)

aus. Bei den Parametern kann es sein, dass ein #<zahl> angegeben wird. Dabei handelt es sich um den (internen) Namen von BEGIN/END-Blöcken, die intern wie eigene Actions behandelt werden. Weiterhin wird bei Ausdrücken nur eine Zahl angegeben. Diese ist ebenfalls ein (interner) Name für den zugehörigen Ausdruck.

Gelegentlich werden weitere Befehle ausgegeben, die nur intern sind, wie etwa ERROR ON/OFF, die dazu da sind, dass innerhalb einer ERROR-Action diese nicht wieder aufgerufen werden kann, etc.

Weiterhin gibt dieser Befehl auch die Auswertung von Ausdrücken an. Dies geschieht durch

EX <name>: <parameter>

Hierbei wird der Ausdruck in Einzelteile zerlegt. Die Auswertung von

```
EVAL x = ( 3 + 5 ) * ( FIRSTOF WITHOUTFIRST "genau 100 Gramm Mehl" )
```

führt zu folgender Debug-Ausgabe:

```
C EVAL: >x< >6< (7)
EX +: >3< >5<
EX WITHOUTFIRST: >genau 100 Gramm Mehl<
```

EX FIRSTOF: >100 Gramm Mehl<

EX *: >8< >100<

Dies bedeutet: Der EVAL-Befehl hat zwei Parameter, nämlich "x" (die Variable in der das Ergebnis gespeichert werden soll) und "6" (den internen Namen des Ausdrucks, der gleich ausgewertet werden wird).

Dieser Ausdruck berechnet das Produkt von zwei Zahlen (genauer wieder Ausdrücken), nämlich (3 + 5) und (FIRSTOF WITHOUTFIRST "genau 100 Gramm Mehl"). Bevor dieses Produkt ausgewertet werden kann, müssen die beiden Faktoren berechnet werden. Die EX-Zeilen geben genau die Reihenfolge an, in der die Auswertung stattfindet.

ACHTUNG: SINGLESTEP kann sehr viel Ausgabe erzeugen und die Ausführung des Programms drastisch verlangsamen. Es ist deshalb ratsam, nur den interessanten Teil des Codes durch DEBUG ON/OFF-Anweisungen debuggen zu lassen.

Weiterhin gibt es noch ein paar Befehle, die nur im Debugging-Modus benutzt werden können.

VARDUMP

Gibt alle Variablen und deren Belegung aus (VD).

EVENTDUMP

Gibt alle Events auf die das Puppet reagiert (WHEN-Befehle) aus (ED...).

QUEUEDUMP

Gibt alle Befehle aus, die darauf warten, vom Puppet ausgeführt zu werden (die Warteschlange) (QD).

Serverbelastung durch das Puppet

Wenn das Puppet viele Befehle in Folge ausführt, kann dies dazu führen, dass der Server stärker belastet wird. In diesem Fall kann die Anzahl der Befehle, die das Puppet pro Sekunde ausführen kann von standardmäßig 100 auf bis zu 0 gesenkt werden. Ist der Server zu sehr überlastet (nicht unbedingt nur durch das Puppet), kann es passieren, dass der Server das Puppet ganz rausschmeißt.

Weiterhin überwacht der Server, dass das Puppet nicht zuviel Speicherplatz verbraucht. Zu lange Programme, zu viele Events, zu tief verschachtelte Action-Aufrufe oder zu viele Variablen können dazu führen, dass ein "Overflow" entsteht. Je nach Einstellung beendet sich in diesem Fall das Puppet selber, die ERROR-Routine wird aufgerufen, oder es passiert einfach garnix. In den letzten beiden Fällen wird die Variable/der Event etc. einfach nicht gesetzt.

Maximalbelegung:

20000 Befehlszeilen+Variablen 5000 Zeichen pro Variable 1000 Events

500 Befehle in der Warteschlange

Stand: 1.12.2002

Autor: berni