Dieser Text ist eine Referenz zur shell Sprache. Er beschreibt Operatoren, Funktionen etc., ohne dabei auf irgendwelche Zusammenhänge einzugehen.

Programmaufbau

Das Interpreterprogramm, daß Sie beim Aufruf des shell Interpreters angeben, beschreibt wie der Interpreter auf die verschiedenen, dem Interpreter bekannten Events reagieren soll.
Ihr Programm wird somit Einträge der Form

```
/event/ { statements }
```

haben, wobei `event` ein gültiges, dem Interpreter bekanntes Event bezeichnet (z.B. die Betätigung einer Taste), und `statements` die von Ihnen gewünschte Reaktion (Start eines Programms, Aufbau einer DDE Verbindung etc.).

Ich werde jetzt zunächst einmal auf den Sprachumfang (`statements`) eingehen. Eine Beschreibung der möglichen Events finden Sie gegen Ende dieses Textes.

Funktionen des Fensters

shift-Button

Jedesmal, wenn Sie auch den shift-Button klicken wird der shift-State invertiert (das ist auch die Defaultfunktion der mittleren und rechten Maustaste). Sie haben dadurch mit der Maus Zugriff auf die normalen und geshifteten Tastenfunktionen.
Wenn Sie auf Esc klicken wird der derzeitige shift-State gelöscht.

Anzeige von Hinweistexten

Das Fenster enhält neben den Buttons auch noch eine Statuszeile, in die Sie Ausgaben vornehmen können, aber auch Interpreter benutzt wird.
Zu jeder Buttonfunktion können Sie einen kurzen Hinweistext angeben, der zusammen mit der Taste, über der sich der Mauszeiger gerade befindet, vom Interpreter in der Statuszeile angezeigt wird.
Wenn Sie einen Button betätigen, wird der Buttonname in der Statuszeile angezeigt, und die Statuszeile 'eingefroren', d.h. es werden dann zunächst einmal keine Hinweistexte mehr angezeigt. Sie können die Anzeige jedoch jederzeit reaktivieren, indem Sie mit der linken Maustaste an eine Stelle im Fenster klicken, an der sich kein Button befindet.

Interpreter Limits

Die maximale Programmlänge beträgt 40000 Bytes.
Nach der erfolgreichen Programmübersetzung gibt die Variable PROGheap die erste freie Adresse hinter Ihrem Programm an. PROGheap * 2 ist gleich der Anzahl der vom Programm belegten Bytes.

Sie können pro Programm maximal 100 nichtindizierte verwenden. Diese Einschränkung bezieht sich nicht auf indizierte Arrayvariablen (z.B. `x[y]`, sowohl `x`, als auch `y` zählen hier als nichtindizierte Variablen, jedoch nicht `x[y]`).
Einen Benutzerdefinierte Funktion zählt in diesem Sinne auch als nichtindizierte Variable.
Nach dem Programmstart enthält die Variable PROGvars die Anzahl der von Ihnen benutzen nichtindizierten Variablen.

Der Stack auf dem der Interpreter Ausdrücke auswertet kann maximal 128 Werte mit einem Gesamtvolumen von maximal 40000 Zeichen speichern.

Strings können eine Länge von 8192 Zeichen (8kB) annehmen. Das heißt jedoch nicht, daß der Interpreter jeden String mit einer solchen Länge überall akzeptiert. Wenn der Interpreter den von Ihnen übergebenen String z.B. als Dateinamen interpretiert, so ist die Länge dieses String i.a. auf 200 Zeichen eingeschränkt.

Fließkommazahlen werden mit 8-Byte Genauigkeit (C Datentyp double) gespeichert.

Die maximale Schachteltiefe von Benutzerprogrammierten Interpreterfunktionen (Schlüsselwort FUNCTION) beträgt 6 Funktionen.

Es können maximal 16 I/O Kanäle zur gleichen Zeit geöffnet sein.

Sie können maximal 150 Eventeinträge anlegen.

Achtung:

Es finden keine Sicherheitstests, die ein Überschreiten der Grenzwerte verhindern würden, statt. Keiner der oben genannten Grenzwerte hat sich bis jetzt als ein Problem erwiesen.

Hinweis zum Compiler

Das von Ihnen angegebene Programm wird zunächst vollständig übersetzt. Sollte Ihr Programm Fehler enthalten, so bekommen Sie eine (mehr oder weniger hilfreiche) Meldung und der Interpreter bricht die weiter Berarbeitung komplett ab.

Obwohl ich den Compiler ausführlich getestet habe, wird er noch einige Löcher enthalten. D.h. an Stellen, wo ein Fehler gemeldet werden sollte tritt der Compiler in eine Endlosschleife ein. In diesem Fall sehen Sie (mit ein bißchen Glück) in der Statuszeile die Datei in der der Compiler hängen bleibt. Beenden Sie dann das Programm (Ctrl+Alt+Del funktioniert immer) und suchen Sie den Fehler.

Interpreter Statements

Im folgenden werden die gültigen Statements der shell Sprache beschrieben.

`expression`	irgendein beliebiger Ausdruck (z.B. Wertzuweisung an eine Variable), es können alle unten aufgeführeten Operatoren und Funktionen verwendet werden.

`printf(format, expr-list)` formatiert Ausgabe von *expr-list*, eine genaue Beschreibung des Statements finden Sie weiter unten bei der Ein / Ausgabe.

`if (expr) statement [else statement]` bedingte Ausführung

`for (expr1; expr2; expr3) statement` Zählschleife, *expr1* wird beim Schleifenstart einmal ausgeführt, *expr3* nach jedem Schleifendurchlauf. *statement* wird solange ausgeführt wie die Auswertung von *expr2* logisch 1 liefert.

`while (expr) statement`
Wiederholungsschleife, *statement* wird ausgeführt solange *expr* logisch 1 liefert.

`break` Schleifenabbruch, die derzeitige `for` oder `while` Schleife wird sofort abbgebrochen.

`continue` beginnt sofort eine neue Iteration einer `for` oder `while` Schleife.

`{ statements }` mehrere Statements können zu einem geklammert werden.

Der Testausdruck bei `if`, `for` und `while` ist genau dann logisch 1, wenn er entweder ungleich 0, oder ungleich dem leeren String "" ist.
Vor den Testausdruck von `for`- oder `while`-Schleifen wird automatisch immer ein Aufruf der yield() Funktion zur Weitergabe der Rechenzeit codiert.

Trennzeichen

Statements werden normalerweise durch das Zeilenende oder durch ein Semikolon getrennt (bzw. beendet). Bei `if`, `for` und `while` kann statement jedoch in einer anderen Zeile stehen als das jeweilige Schlüsselwort.
In Programmen sind leere Zeilen möglich. Desweiteren können Leerzeilen hinter jeder geschweiften Klammer eingefügt werden.

sh akzeptiert hinter jedem Komma ein newline und erweitert das Statement entsprechend auf die nächste Zeile.
Ebenso hinter folgenden Operatoren:

```
&   &&   ||   |:   |   ?:
```

`|:` bezieht sich nur auf den binären Funktionspipeoperator, `&` nur auf den Stringadditionsoperator.
Bei dem ternären Bedingungsoperatoren `?:` können newlines hinter dem Fragezeichen `?` und/oder hinter dem Doppelpunkt `:` eingefügt werden.
Außerdem können Sie ein newline hinter den `for()` Semikola einfügen.

Ausdrücke

Numerische- und String- Ausdrücke können mit den unten aufgeführten Operatoren gebildet werden.

Konstanten

Es gibt zwei Arten von Konstanten: Strings und Zahlen. Ein String ist eine beliebige Zeichenkette zwischen doppelte Anführungszeichen, also z.B `"shell93"` oder `"Hello World!"` oder `""`.
Numerische Konstanten sind entweder Integer (`1234` oder `4711`), Dezimalzahlen wie `3.14` (als Kommazeichen verwendet sh den Punkt), oder Zahlen mit Zehnerexponent: `0.314e1`, `31.4e-1`. Numerische Konstanten werden als 8-Byte Fließkomma Zahlen gespeichert.

Escape Sequenzen

Innerhalb von Anfürungszeichen sind folgenden Escape Sequenzen definiert:

`\a`	bell (ASCII 7)
`\b`	backspace
`\f`	formfeed
`\n`	newline
`\r`	carriage return
`\t`	tab
`\z`	ctrl+z
`\`*`ddd`*	bis zu drei Oktalziffern, die das Zeichen angeben

Variablen

Variablen müssen nicht explizit angelegt werden, sie werden automatisch bei der Programmübersetzung generiert.
Ein gültiger Variablenname beginnt mit einem Buchstaben oder einem underscore '_', gefolgt von weiteren Buchstaben, Ziffern oder underscores.

Achtung:

Variablennamen sind nicht case-sensitiv, d.h. `xy`, `Xy`, `xY`, `XY` bezeichnen immer dieselbe Variable.

Systemvariablen:

Einige Variablenname sind reserviert und / oder werden vom Interpreter gesetzt:

`ARGC`	Anzahl der Argumente auf der Kommandozeile.
`ARGV`	Kommandozeileargumente.
`cfgfile`	Name der shell93 Konfigurationsdatei.
`DDEinput`	Daten bei DDE Poke Operationen.
`DDEoutput`	Ausgabedaten bei DDE Peek Operationen.
`DDEservice`	Name des DDE Services.
`DROPC`	Anzahl der Dateien des drop-events.
`DROPmode`	Name des aktuellen drop-handlers.
`DROPV`	Dateien des drop-events.
`PROGheap`	Größe des derzeitigen Programms.
`PROGvars`	Anzahl der benutzten Variablen.
`PROGwindow`	Window Handle des Programmfensters.

`rlength`	Länge des Suchstrings nach index Funktion.
`rstart`	Startposition des Suchstrings.
`shellpath`	shell Verzeichnis
`varfile`	Name der Variablendatei.
`winpath`	Name des Windows-Systemverzeichnises.
`WINiconized`	Statusvariable, == 1 wenn Fenster minimiert.
`WINiconno`	Iconnummer der Fensters.
`WINkey`	Hotkey des Fensters.
`WINontop`	== 1 wenn Fenster immer in Vordergrund.
`WINtext`	Titel des Fensters.

Wie schon gesagt, Groß / Kleinschrift ist bei Variablennamen egal, die obige Liste beruht lediglich auf meinen persönlichen Angewohnheiten.

Operatoren

Folgende Operatoren sind (mit steigender Priorität) definiert:

`= += -= *= /= &=`	Wertzuweisung
`,`	Sequentielle Ausführung
`?:`	Bedingter Ausdruck
`||`	Logisches Oder
`&&`	Logisches Und
`in`	Array Index Test
`< <= > >= != ==`	Vergleichsoperatoren
`|get |: |`	Pipeoperator
`&`	Stringaddition, auch implizit
`+ --`	Addition, Subtraktion
`* / %`	Multipikation, Division, Modulo
`+ - !`	Vorzeichen Plus und Minus, Logisches Nicht
`^`	Exponentiation
`++ --`	Inkrement, Dekrement
`*() [] &`	Indirektion, Indizierung, 'Adressoperator'

Alle Operatoren sind mit Ausnahme der (rechts assoziativen) Wertzuweisung, der bedingten Zuweisung und der Exponentiation links assoziativ.

Arithmetische Operatoren

Die arithmetischen Operatoren + - * / % (Modulo) und ^ (Exponentiation) funktionieren in der gewohnten Art und Weise:

5 + 2 = 7
1 + 2 * 3 ^ 4 = 163
12 / 5 = 2.4
12 % 5 = 2
3.1415 % 1 = .1415

Vergleichsoperatoren

Mit den Vergleichoperatoren < kleiner, <= kleiner gleich, > größer, >= größer gleich, == gleich und != ungleich können Strings und Zahlen verglichen werden. Sie liefern 1 (logich wahr, true) wenn die Relation erfüllt ist, sonst 0.

Logische Operatoren

Es sind Operatoren für logisches Und &&, logisches Oder || und logisches Nicht ! definiert. || liefert 1, wenn mindestens einer der beiden Operanden ungleich 0 oder ungleich dem Leerstring "" ist, && nur dann, wenn beide Operanden ungleich 0 oder ungleich "" sind.

Ein logischer Ausdruck wird nur soweit ausgewertet, bis sein Ergebnis feststeht. D.h. ist der linke Operand von || ungleich 0, so wird der rechte garnicht erst berechnet (also werden auch keine Seiteneffekte ausgeführt). Dementsprechend reagiert der && Operator, wenn der rechte Operand gleich 0 ist.

Bedingungsoperator

Der Bedingungsoperator (rechts assoziativ) hat die Form

expr ? *if-expr* : *else-expr*

Abhängig davon, ob *expr* true (!= 0 oder != "") oder false (== 0 bzw. == "") ist wird *if-expr*, bzw. *else-expr* ausgewertet und das Ergebnis geliefert.

```
(x >= 5)? -1: x;
```

liefert -1 wenn x größer gleich 5 ist, sonst x.
In jedem Fall wird entweder nur *if-expr* oder nur *else-expr* ausgewertet (beachten Sie das, wenn Sie mit Seiteneffekten programmieren).

Datentypen

Der Typ eines Ausdrucks ist entweder numerisch, oder nicht-numerisch (Strings). Die Operatoren nehmen i.a. Operatoren eines bestimmten Typs, hat ein Operand einen anderen Typ, so wird er automatisch in den anderen um gewandelt.
Es gilt z.B.:

```
4 + "7" + 11 = 22
4 & "7" & 11 = "4711"

4 + "x" = 4
4 & "x" = "4x"
```

Operatoren die beide Datentypen verarbeiten sind die Vergleichsoperatoren < <= > >= != und ==, sowie die einfache Wertzuweisung = und der Bedingungsoperator ?:.

Sind die Operanden eines Vergleichsoperators verschiedenen Typs, werden die Operanden in Strings umgewandelt und ein Stringvergleich durchgeführt.
Die Ausdrücke

```
8 > "50"
8 < "50" + 0
```

liefern beide 1.

Explizite Typumwandlung:

Um den Datentyp eines Ausdrucks explizit fest zulegen können Sie entweder `0` (Null) addieren um einen numerischen Ausdruck, oder den Leerstring `""` hinzufügen (&-Operator) um einen Stringausdruck zu erhalten.

Wertzuweisung

Die folgenden Zuweisungs Operatoren sind definiert. Sie sind alle rechts-assoziativ und legen teilweise den Typ der Variablen fest.

`+= -= *= /=`	wandeln der Wert der Variablen zunächst in einen numerischen Wert um, führen die Operation aus und speichern einen numerischen Wert.
`&=`	liefert dementsprechend Stringvariablen.
`=`	der Datentyp der Variablen ist gleich dem Datentyp des Ausdrucks. Ausnahmen hiervon bilden Array- und extern gespeicherte Variablen (siehe unten), ihr Datentyp ist immer der Stringtyp.

Die Operatoren liefern Typ und Wert der Variablen

```
x = 7, x += 2;
x = (y = 7) + 2;
x = 7, x += y = 2;
```

In allen drei Fällen ist x = 9, im zweiten darüber hinaus y = 7 und im dritten y = 2.

```
x = "7", x &= y = 2;
```

hingegen ergibt x = "72" und y = 2.

Inkrement und Dekrement

Die Inkrement / Dekrement Operatoren ++ und -- gibt es in der Prä- und Postfix Version *var*++ und ++*var*.
Der Postfix Operator liefert zuerst den Variablenwert und führt dann die Operation aus, der Präfix Operator funktioniert umgekehrt:

```
x = 7;
y = ++x;
z = x++;
```

ergibt x=9, y=8, z=8.
Geben Sie einen Präfix und einen Postfix Operator oder einen Wertzuweisung Operator an, so wird der Präfix Operator ignoriert; ein Fehler wird nicht gemeldet:

```
++x--;          wird zu    x--;     und
++x = 7;        zu         x = 7;
```

Beide Operatoren wandeln die Variable ggf. erst in eine numerische Variable um.

Sequentieller Operator

Der ,-Operator (Komma) führt verschiedene Ausdrücke hintereinander aus, er liefert den Typ und Wert des letzen Ausdrucks als Resultat.

```
x = 7, 2 * x
```

liefert 14.
Innerhalb von Funktionsklammern werden mit dem Komma Parameter getrennt:

```
sin(x = 7, 2 * x)
```

ergibt sin(7, 14). Wollen Sie in Funktionalklammern mit dem sequentillen Operator arbeiten müssen sie arithmetische Klammern setzen:

```
sin((x = 7, 2 * x))
```

berechnet sin(14).

Stringoperatoren

Der Stringadditions Operator & nimmt Stringausdrücke und liefert einen String:

5 & 2 = "52"

Implizite Stringaddition:

Steht zwischen zwei Operanden kein Operator, so wird automatisch der Stringadditions Operator & codiert.
In diesem Sinne wird ein Variablenname gefolgt von einer öffnenden runden Klammer nicht als Folge von zwei Operanden (Variable und Ausdruck) sondern als Aufruf einer Funktion interpretiert.

```
"x" 12 = "x12"
12 00 = "120"
7 (3 + 4) = "77"
sin (3 + 4) = Funktionsaufruf
```

Arrays

Ebenso wie Variablen müssen Arrays auch weder deklariert, noch dimensioniert werden. Die von einem Array benutzte Indexmenge sind Strings, keine Integerzahlen (man bezeichnet solche Arrays auch als assoziativ).
Die Indizierung eines Arrays geschieht über eckige Klammern:

```
x["1"] = 7;
y = "1", x[y] = 7;
x[1] = 7;
x[1] = 6, x[1]++;
```

Die vier Programmzeilen legen alle zu der Variablen x den Index "1" an und weisen ihm den Wert 7 zu.
Beachten Sie bei der dritten Zeile, daß numerische Ausdrücke (der Index) automatisch in Strings umgewandelt werden.

Beachten Sie außerdem, daß die beiden Ausdrücke "01" und "1" (bzw. eine numerische 1) verschiedene Indizes beschreiben.

Beim lesenden Arrayzugriff können Sie im Index Tabulator getrennte Strings als Index übergeben. Dieser String wird dann in die einzelnen Indizes zerlegt, und die zu den einzelnen Indizes gehörenden Werte werden in einem Tabulator getrennten String (dieselbe Reihenfolge) zurückgeliefert.

Ein entsprechendes Analogon existiert für die Wertzuweisung an ein Array nicht. Die Tabulatoren werden bislang einfach mit in den Index übernommen, und Sie können in der Folge diesen Wert nicht mehr auslesen.
Vermutlich wird hier bei einem späteren Programmrelease ein anderes Verhalten eingesetzt. Bis dahin sollten Sie beim Schreiben von Arraywerten keine Tabulatoren in den Index einbauen.

Arrayindexoperator

Der `in` Operator

expr `in` *var*

testet ob in einem gegebenen Array *var* ein bestimmter Index *expr* vorkommt und liefert 1, wenn ja, sonst 0.
Sie können die Formulierung

```
if (var[expr] != "")
```

nicht benutzen, da durch sie wenn der Index *expr* im Array *var* noch nicht existiert automatisch ein entsprechender Eintrag angelegt und mit dem Leerstring initialisiert wird.

Speicherung von Variablen und Arrays

Variablen werden grundsätzlich in einer externen Datei gespeichert. Diese Datei hat das *.ini Format und i.a. die Endung ".var".
Eine Variable wird dann intern gespeichert, wenn sie nicht indiziert ist (Arrayvariablen sind immer extern), und entweder numerisch ist oder die Länge 90 Zeichen nicht überschreitet.

Achtung:

Eine Variable die aus dem externen Speicher geladen werden muß hat immer den Stringtyp (beachten Sie das bei den Typvariablen Operatoren wie `= ?: < <=` etc.).

Speicherbare Zeichen:

Nun hat jede *.ini Datei Probleme mit einigen Zeichen, so sind z.B. newlines und Tabulatoren nicht speicherbar.
Um dennoch Steuerzeichen zu speichern wird beim Schreiben in die Variablendatei der ASCII Bereich 0x01 bis 0x1F in den Bereich 0xE0 bis 0xFF übersetzt, beim Lesen umgekehrt.
Ist das erste Zeichen des Variablenwertes ein Leerzeichen, dann wird dafür 0xDF eingesetzt.
Sie können also nicht mit dem ASCII Bereich 0xDF bis 0xFF arbeiten (sehen Sie mal in Ihrer ASCII Tabelle nach, was das für Zeichen sind).

Sektionen der Variablendatei:

Nicht indizierte (skalare) Variablen werden in der Datei in der "var" Sektion unter ihrem Namen gespeichert.
Jedes Array wird in einer Sektion mit dem Arrayvariablennamen gespeichert.

Indirekte Variablenaddressierung

Da Variablen letztlich durch Strings identifiziert werden liegt es nahe den Indirektionsoperator *() (die Klammern gehören zum Operator, zwischen * und öffnender Klammer darf kein Leezeichen stehen) zu definieren. Er wird in der Form *(*expr*) verwendet und wandelt *expr* ggf. zunächst in einen String und dann in einen Variablenbezeichner um.

```
y = "x";
*(y "12") = 7
```

weist der Variablen x12 den Wert 7 zu.

Wenn Sie in den Variablennamen ein Ausrufungszeichen ! einbauen, so wird der Text links davon als Dateiname, der Text rechts als Variablenname interpretiert:

```
*(cfgfile "!config")["editor"] = "notepad"
```

greift auf den Index "editor" der 'Arrayvariablen' (eigentlich die config-Sektion) config in der Konfigurationsdatei (Name ist ein der Variablen cfgfile gespeichert) zu, und gibt dieser Variablen den Wert "notepad".
Anders gesagt setzt die obige Zeile notepad als ihren shell Editor ein.

WARNUNG

Indirekte Variablenaddressierung ist eine (der wenigen) Möglichkeiten
Ihr System nachhaltig zu verändern, beschädigen etc.

Erinnern Sie sich daran, das niemand (außer Ihnen vielleicht) für derartige Effekte verantwortlich oder haftbar ist.
Formulierungen wie

```
*("win.ini!" "...")[...] =
```

sollten Sie nur dann gebrauchen wenn Sie genau wissen was Sie tun.

Die Umkehrung des Indirektionsoperators ist der 'Addressoperator', der zu einer skalaren Variablen den Variablennamen als String liefert (eigentlich eine triviale Angelegenheit, wurde nur aus rein formalen Gründen eingeführt). `&x` ergibt den String `"x"`.

Ein / Ausgabe

printf(*format*, ...) formatiert die übergebene Liste von Werten anhand von *format* und gibt den resultierenden Text in die Statuszeile aus.
Die Formatiersequenzen sind im Prinzip die der C-printf() Funktion:

%c	einzelnes Zeichen anhand des ASCII Codes
%d	Dezimalzahl
%e	[-]d.dddE[+-]dd
%f	[-]d.ddd
%g	entweder %e oder %f jenachdem was kürzer ist
%o	vorzeichenlose Oktalzahl
%s	String
%x	vorzeichenlose Hexzahl
%%	gibt ein % aus, nimmt kein Argument

Außerdem sind zwischen % und Formatzeichen folgende Angaben (in dieser Reihenfolge) möglich:

-	linksbündige Ausgabe.
breite	formatiert in ein Ausgabefeld mit einer Breite *breite* Zeichen, wird mit Leerzeichen (Ausnahme, wenn *breite* eine führende Null hat) aufgefüllt.
gen	gibt an wieviele Nachkommaziffern (Genauigkeit) ausgegeben werden sollen.

Umleitung der Ausgabe:

Dem printf() Statement kann der Ausgabeumleitungsoperator > angefügt werden:

```
printf(format, ...) > kanal
```

Dadurch wird die Ausgabe nicht in die Statuszeile ausgegeben, sondern in den durch *kanal* bezeichneten Ausgabekanal umgeleitet.
Der Umleitungsoperator (bzw. die Operatoren, siehe unten) können nur nach dem printf() Statement verwendet werden.

Ein / Ausgabekanäle:

Ein / Ausgabekanäle werden durch Strings identifiziert, die Kanäle werden automatisch bei der erstmaligen Verwendung geöffnet und bleiben bis zum Programmende, oder bis sie explizit mit close() geschlossen werden, offen.
Ein Kanal kann entweder eine Datei, das Clipboard oder ein DDE-Kanal sein, der Kanaltyp wird durch den Kanalbezeichner festgelegt:

service|topic!item	der Kanal bezeichnet einen DDE-Kanal der gelesen und geschrieben werden kann.
service|topic	der DDE-Kanal kann nur geschrieben werden, eine Schreibanweisung wird in eine DDE-Execute Transaktion umgewandelt, eine Leseoperation liefert immer einen leeren String.
"clipbrd"	identifiziert das Clipboard als Ein / Ausgabekanal.
String mit Leerzeichen	der Kanal ist ein DOS Kommando. Wenn Sie vom Kanal lesen, wird zuerst das Kommando ausgeführt, die Standardausgabe in einer Datei gesammelt und dann die I/O Operation auf der Datei ausgeführt. Wollen Sie auf den Kanal schreiben, so wird die Ausgabe zunächst in eine temporären Datei geschrieben. Das Kommando wird nur dann ausgeführt, wenn Sie den Kanal explizit schließen. Dem Kommando wird dabei die Ausgabe als Standardeingabe übergeben

jeder andere Kanalbezeichner wird als Dateiname interpretiert.

Anhängen der Ausgabe:

Neben dem Umleitungsoperator > existiert auch der >> Operator. Wird ein Kanal mit > geöffnet so wird der bisherige Inhalt überschrieben, bei Verwendung von >> bleibt der alte Inhalt erhalten und die Ausgabedaten angehangen.

Diese Unterscheidung wird nur gemacht, wenn der Kanal geschlossen ist, und der Kanal entweder das Clipboard oder eine Datei identifiziert.

Besonderheiten der verschiedenen Kanäle:

Clipboard schreiben	vor jedem newline (\n) wird ein carriage-return (\r) eingefügt. Jedes Zeichen nach einem \r wird unverändert übernommen. Es steht Ihnen also frei ein Zeilenende durch \n oder \r\n zu markieren.
Clipboard lesen	die Daten werden unverändert weitergegeben, es wird immer der komplette Inhalt gelesen. Das Lesen des Clipboard öffnet keinen Kanal.
Dateien	Dateien können nicht gleichzeitig zum Lesen und zum Schreiben

	geöffnet werden, um die Transferrichtung zu ändern muß die Datei zunächst mit close() geschlossen werden. Beim Schreiben werden newlines in \r\n Sequenzen umgewandelt, beim Lesen umgekehrt. Dateien werden zeilenweise gelesen.
DDE Kanäle	Es wird immer die komplette Datenmenge unverändert weitergegeben.

Übertragungsformate:

Das DDE Übertragungsformat ist immer CF_TEXT, beim Clipboard immer CF_OEMTEXT.
Unabhängig davon, was an irgendwelchen anderen Stellen in der Dokumentation (insbesondere shell2.wri) gesagt wird, klappt die Übertragung von Umlauten über DDE. Intern arbeitet der Interpreter mit dem DOS Zeichensatz (OEM). Bei DDE Schreiboperationen werden die Daten an den Windows Zeichensatz (ANSI) angepaßt, und umgekehrt. Insbesondere läuft dann auch die Übertragung von Umlaute nach und von Excel.

Daten einlesen:

Kanäle können auf unterschiedliche Arten abgefragt werden:

```
(kanal | get var)
```

ließt von *kanal* und speichert den Wert in der Variablen *var* zu. Das Ergebnis des Ausdrucks ist der resultierende iostatus (s.u.).
Hingegen liefert

```
(kanal | get)
```

direkt den Wert der Abfrage. Ob diese Leseoperation erfolgreich war hängt davon ab, ob `iostatus == 0` ist oder nicht, Sie müssen das seperat testen.

Sie können die Klammern weglassen, wenn dem Operator ein newline oder Semikolon folgt.
Der get-Operator liefert immer Stringdaten.

Mit dem get-Operator können keine indizierten Variablen (z.B. `line[k]`) gelesen werden.

```
for (delete(line), k=0; (datei | get x) == 0; k++)
       line[k] = x;
```

ließt die durch `datei` spezifizierte Datei zeilenweise in das Array `line` ein.

Pipes:

Datenpipes geben Ihnen die Möglichkeit mit einer Operation Daten auf einen DDE Kanal zu schreiben und zu lesen:
Statt

```
printf ("%d\r\n", x) > "excel|tab1!Z4S2";
"excel|tab1!Z4S3" | get y
```

können Sie auch

```
y = x | "excel|tab1!Z4S2" : "Z4S3"
```

schreiben, das Resultat ist dasselbe.
Der (ternäre) pipe-Operator |: hat die allgemeine Form

```
expr | write : read
```

write identifiziert den DDE Kanal auf den geschrieben werden soll, *read* ist eine Angabe die zusammen mit dem *write* Kanal die komplette Beschreibung des Lese-Kanals ergibt:

""	ist *read* == "" so ist der Lese- mit dem Schreibkanal identisch.
item	gelesen wird von einem anderen *item*, service und topic des Schreibkanals bleiben erhalten.
topic!*item*	es wird zu *topic* und *item* gewechselt.
service|*topic*!*item*	es wird von einem komplett anderem DDE Kanal gelesen (ob Sie dann noch eine pipe vorliegen haben ist ganz alleine Ihre Sache).

Diese Transformationen werden nicht durchgeführt, wenn *write* keinen DDE Kanal identifiziert (grundsätzlich arbeitet der pipe-Operator mit allen Arten von Kanälen)

Der ternäre pipe-Operator liefert das Ergebnis der Leseoperation, `iostatus` gibt den Statuscode dieser Operation an.

Im Gegensatz dazu liefert der binäre pipe-Operator

```
(expr | write)
```

den iostatus der Schreiboperation. Die Klammern können wieder entfallen, wenn ein newline oder ein Semikolon folgt.

Wird durch einen der beiden Operatoren ein Kanal geöffnet, so wird der alte Inhalt überschrieben.

Schließen von Kanälen

Die close-Funktion schließt einen (oder alle) Kanäle:

```
close(s)
```

schließt den durch `s` bezeichneten Kanal oder alle, wenn `s == ""` ist. close() liefert als Ergebnis die Anzahl der Kanäle, die geschlossen wurden.

Sie sollten DDE Kanäle dann schließen und neu öffnen, wenn die Server Applikation neu gestartet wurde, Dateien um die Transferrichtung zu ändern, und um den

Dateiinhalt tatsächlich in die Datei zu schreiben (Pufferung durch das Betriebssystem).

DOS Kommandos (Schreibzugriff) müssen geschloßen werden, um das Kommando zu starten. Das Kommando wird nur dann geschloßen, wenn Sie den Kanal explizit schließen. D.h. wenn Sie einen DOS Kommando Schreibkanal durch

```
close("")
```

schließen, wird der Kanal geschloßen, aber das Kommando nicht ausgeführt.

Funktionen

In der folgenden Übersicht stehen *n*, *m*, *w*, *x*, *y* für numerische, *p*, *q*, *r*, *s*, *t* für Stringausdrücke, und *v* für eine skalare Variable.

Array Funktionen:

abs(*s*)	siehe Eintrag bei den Mathematischen Funktionen.
delete(*v*, ...)	löscht aus dem Array *v* die angegebenen Indizes, bzw. die komplette Indexmenge wenn kein Index angegeben ist. Rückgabewert ist die Anzahl der angebenen Indizes.
enum(*v*)	liefert die komplette Indexmenge der Variablen *v*.
split(*s*, *v*, *t*)	spaltet den String *s* anhand den Trennzeichen *t* auf und speichert sie beginnend mit dem Index 0 im Array *v*.

Datei Funktionen:

noctrl(*s*)	schneidet ctrl-codes (ASCII Code <= 0x20) vom Ende des Strings *s* ab.
chdir(*s*)	macht *s* zum neuen Arbeitsverzeichnis. Liefert das neue Arbeitsverzeichnis zurück.
fullpath(*s*)	wandelt den relativen Dateinnamen *s* in eine absolute Dateiangabe um.
getcwd()	liefert das aktuelle Arbeitsverzeichnis.
getdir(*s*)	liefert das Verzeichnis (mit Laufwerksangabe) der Datei *s.*
getname(*s*)	liefert den Dateinamen (mit Erweiterung) von *s*.
tmpfile(*r*, *s*)	erzeugt einen eindeutigen Dateinamen (und eine leere Datei mit diesem Namen) mit dem Päfix *s* (maximal drei Buchstaben) im Verzeichnis der DOS Ungebungsvariablen TMP, oder im Verzeichnis *r*.

Ein / Ausgabe:

close(*s*) schließt den Ein / Ausgabekanal *s*, oder alle offenen Kanäle wenn *s* ein leerer String ist.

Mathematische Funktionen:

abs(*x*) lefert den Absolutwert von *x*, wenn *x* numerisch ist. Ist der Parameter ein String und komplett als Zahl interpretierbar, so wird ebenfalls der Absolutwert berechnet, sonst die Anzahl der Tabulator getrennten Felder in *x*.

atan2(*y*, *x*) Arcustangens von *y*/*x* in Radiant.

cos(*x*) Cosinus von *x*.

exp(*x*) Exponentialfunktion e^x.

int(*x*) Rundet auf die nächste ganze Zahl ab.

log(*x*) natürlicher Logarithmus.

max(*x*, *y*, ...) berechnet das Maximum der Werte.

min(*x*, *y*, ...) berechnet das Minimum.

rand() liefert eine (Pseudo-) Zufallszahl.

sin(*x*) Sinus von *x*.

sqrt(*x*) Quadratwurzel von *x*.

srand(*x*) Initialisierung des Zufallszahlengenerators mit *x*.

sin() und cos() erwarten ihre Argumente in Radiant. log() und sqrt() liefern für negative Werte 0 zurück.

String Funktionen:

index(*s*, *t*) sucht den String *t* in *s* und liefert bei Erfolg den Index zurück, sonst 0.

length(*s*) liefert die Länge des Strings *s*.

sprintf(*format*, ...) formatiert die angegebenen Parameter anhand von *format*, schreibt sie in den einen String und liefert den daraus resultierenden String zurück. Formatierungsmöglichkeiten siehe printf().

strstr(*r*, *s*) index() Synonym.

strlen(*s*) length() Synonym.

strlwr(*s*) wandelt den Großbuchtaben im String *s* in Kleinbuchstaben um.

strupr(*s*) wandelt den Kleinbuchstaben in *s* in Großbuchstaben um.

substr(*s*, *m*, *n*) liefert den Teilstring von *s*, der an der Stelle *m* beginnt und dessen Länge gleich *n* Zeichen ist. Fehlt *n*, so wird der String ab Position *m* bis zum Stringende zurück geliefert. Ist *m* <= 0 oder > length(*s*) oder n <= 0, so liefert die Funktion einen Leerstring.

System Funktionen:

appwin(*m*) liefert das zur Task *m* gehörende Top-Level Window.

beep() gibt einen Beep aus, liefert als Ergebnis immer 0.

ctrl(*m, ...*) steuert das Programm Fenster in Abhängigkeit von *m*:

m	Bedeutung
string	ist der Parameter ein String, so wird der Text als Window Title eingesetzt.
1	stellt das Fenster als Icon dar, -1 zeigt das Fenster normal an.
2	stellt für das Fenster "immer im Vordergrund" ein, -2 löscht die Einstellung.
3	es folgt als nächster Parameter ein virtueller Keycode, kombiniert mit der Alt-Taste als Hotkey eingesetzt wird.
4	setzt das Window-Icon. Die Iconnummer ist der nächste Parameter.
5	schreibt alle Variablen auf Platte.

Sie können mehrere Operationen mit einem ctrl() Aufruf durchführen. Nach einer Hotkey Operation wird jedoch die Bearbeitung abgebrochen und Sie erhalten den Statuscode der Hotkey Operation:

0	Hotkey ist eingestellt.
1	ungültiger Hotkey
2	der Hotkey ist bereits vergeben.

Ansonsten lefert ctrl() immer 0.
Die Funktion setzt die Variablen

`WINiconno, WINiconized, WINkey,`
`WINontop` und `WINtext`

auf die aktuellen Werte.

dos(*s*) startet das DOS Kommand *s* über sh2dos.pif. Die Ausführung ist synchron, d.h. der Interpreter wartet auf das Ende des Kommandos.

enter(*s*) beendet die derzeitige Programmausführung und startet asynchron die Ausführung des `user` Events *s*, wenn *s* ein definiertes `user` Event ist. Ansonsten wird 0 zurückgeliefert.

eval(*s*) übersetzt den angegebenen Ausdruck *s* und wertet ihn aus, Rückgabewert ist das Ergebnis des Ausdrucks.Tritt während der Übersetzung von *s* ein Fehler auf wird die derzeitige Eventbearbeitung abgebrochen.

exec(*s*, *m*) startet das Programm *s* mit dem Anzeigeparameter *m*. Zurückgegeben wird das Task Handle des Programms, oder 0 wenn das Programm nicht gestartet werden konnte.

exit() beendet das Programm.

focus(*w*) gibt dem Fenster *w* den Eingabefokus. Wenn *w* kein gültiges Fenster identifiziert sehen Sie auf dem Bildschirm eine Fehlermeldung und die weitere Interpretation wird abgebrochen.

getwin(*t*) sucht ein Fenster das im Titel den gegebenen Text *t* enthält und liefert das Window Handle zurück (== 0, wenn kein Fenster gefunden wurde).

getmod(*s*) übergibt das Modul Handle, wenn das Modul mit dem Namen *s* sich im Speicher befindet.

gkbd(*s*) erzeugt Tastaturevents. Für eine komplette Beschreibung der Funktion siehe gkbd.wri.

quit(*s*) lädt die DLL *s* (ohne .dll Endung) aus dem Speicher. Rückgabewert ist der Referenzzähler den die Bibliothek hatte, oder -1 wenn die DLL nicht existiert.
Die quit() Funktion lädt eine beliebige DLL aus dem Speicher, allerdings unabhängig davon, ob diese Bibliothek von irgendeinem Programm noch verwendet wird, also im Speicher verbleiben sollte.
Die quit() Funktion ist also eine Möglichkeit auf Ihrem System Probleme verschiedener Art zu verursachen.

wait(*n*) wartet, bis die Task *n* beendet wurde, liefert 0, wenn *n* kein gültiges Task Handle ist -1.

yield() gibt die Rechenzeit an andere Programme weiter, liefert immer 0. Ein yield() Aufruf wird automatisch in while() und for() Schleifen vor den Testausdruck codiert.

Dialog Funktionen:

inform(*s*, *t*)	zeigt die Meldung *t* in einer Dialogbox mit dem Titel *s* an, die Dialogbox hat nur einen Ok-Button.
input(*r*, *p1*, *p2*, *s*)	zeigt eine Eingabebox mit dem Titel *r* und den beiden Prompts *p1* und *p2* (Strings) an (*p2* erscheint unmittelbar vor dem Eingabefeld, *p1* darüber). *s* ist der Vorgabewert des Eingabefeldes. Die Funktion liefert `"\032"` wenn der Benutzer den Dialog abbricht, sonst die Eingabe.
msg(*r*, *s*, *t*)	Zeigt eine einfache Message Box mit det Titel *r* und dem Text *s* an. *t* enthählt Tabulator getrennte Buttontexte (maximal 3). msg() liefert die Nummer des Buttons zurück, mit dem der Dialog beendet wurde.
select(*n*, *m*, *r*, *s*, *t*, *p*)	zeigt eine Listbox *n* Pixel breit und *m* Pixel hoch mit dem Titel *r* an. Die Daten der Listbox sind Tabulator getrennte Strings in *t*, *s* bezeichnet das vorzuselektierende Element. Als Ausgabe liefert select() `"\032"` wenn der Benutzer den Dialog abgebrochen hat, sonst den selektierten String. Der ausgewählte String wird von der select() Funktion außerdem in die Variable `SLCitem` gesetzt. In *p* können Sie der Listbox Tabulator getrennte Einträge für das Systemmenu mitgeben. Sie haben dann die Möglichkeit, ein Listboxitem zu selektieren, und die Listbox über die Auswahl eines Menüeintrags zu beenden. Der Text des Eintrags wird dann in die Variable `SLCcommand` geschrieben. Ist '#' (Raute) das erste Zeichen in *p*, so werden die Daten von der Listbox nicht sortiert.

Funktionspipes:

Sie können Ausdrücke wie

```
x = getname(strlwr(noctrl("clipbrd" | get)))
```

durch Funktionspipes in eine andere Notation bringen:

```
x = ("clipbrd" | get) |: noctrl() |: strlwr() |: getname()
```

Diese Form des pipe-Operators

```
expr |: function
```

(nicht zu verwechseln mit der Ein / Ausgabepipe, bei Funktionspipes folgt der Doppelpunkt direkt dem pipe-Symbol) macht nichts weiter, als *expr* als erstes Argument von *function* einzusetzen, und das Ergebnis von *function* zu liefern.

Aufbau einer Programmdatei

Eine Programmdatei ist eine Folge von event/action Sequenzen. Eventbeschreibungen werden in Schrägstriche eingefaßt (Ausnamen sind die sog. special-events), der folgende Programmtext i.a. in geschweifte Klammern.

Sollten Sie ein Event mehrfach aufführen, so wird zur Laufzeit nur die letzte Behandlungssequenz ausgeführt (neue Reaktionen überschreiben alte).

Tastenprogrammierung:

(Funktions-) Tasteneventnamen bestehen aus der Tastennamen mit einem optionalen 'shift' davor:

```
/F2/            { printf ("Hello World!") }
/shift F2/      { printf ("Taste shift F2 gedrückt"); }
```

Den Tastaturevents kann ein Hinweis (als String) angefügt werden. Dieser String wird dann in der Statuszeile angezeigt, wenn der Mauscursor sich über der Taste befindet.

```
/F2 "Hello"/    { printf ("Hello World!") }
```

BEGIN und END:

BEGIN und END sind sog. special-events. Was sie von den anderen Events unterscheidet ist, das sie nicht durch externe Ereignisse ausgelößt sondern beim Programmstart, bzw. Programmende (na gut, in gewissem Sinne sind auch das externe Events) ausgeführt werden, und daß jede Reaktion auf das BEGIN (bzw. END) Event ausgeführt wird.
Außerdem werden special-event Namen nicht von Schrägstrichen umgeben.

```
BEGIN   { printf ("Hello World!") }
END     { printf ("Good bye, it was nice to meet you.") }
```

Systemmenu:

Systemmenueinträge werden mit dem Schlüsselwort menu eingeleitet, es folgt der quotierte Text, der im Menu angezeigt werden soll (max 20 Zeichen):

```
/menu "Menu &Funktion ..."/ {
       inform ("Hello World!",
               "This is a test.")
       }
```

DDE Ports:

Mit den 'Schlüsselworten' | und ? gefolgt von einem Topic und einem Item Bezeichner legen Sie einen entsprechenden DDE Port zum Schreiben, bzw. Lesen an.
Um die DDE Ports dann auch zu aktivieren müssen Sie während des BEGIN events die Variable DDEservice setzen.
Wenn zur Laufzeit eine Schreib/Lese Funktion ausgeführt wird, so setzt die shell zunächst die Variablen DDEtopic und DDEitem auf den angesprochenen Kanal (bei Schreiboperationen werden außerdem die Daten in der Variable DDEinput übergeben) und ruft dann die Behandlungsprozedur auf.

```
/| port1 data/ {
       printf ("received some data");
       }
```

```
/? port1 data/ {
        DDEoutput = "Hello DDE World!")
        }

BEGIN {
        DDEservice = "new-service"
        }
```

Bei Leseoperationen übergibt die Prozedur in der Variablen DDEoutput die Daten die ausgegeben werden sollen.

Mit dem Itemnamen * legen Sie einen wildcard-port an:

```
/| port1 */ {
        printf ("received some data, topic: %s", DDEtopic);
        }
```

Um zur Laufzeit die DDE Ports zu aktivieren muß während in der BEGIN Section die Variable `DDEservice` auf den Servicenamen gesetzt werden.

Drag & Drop

Sie können mehrere drop Handler in der Form

```
/drop string/
```

wobei *string* eine quotierte Zeichenkette ist spezifizieren. Zur Laufzeit bestimmt der Interpreter dann anhand der Variablen DROPmode den auszuführenden Handler. Kann der Interpreter keinen Handler finden der den in DROPmode angegebenen Modus behandelt wird ein `wm drop` Event ausgelößt.
Während eines drop Events gibt DROPC die Anzahl der Dateien, das Array DROPV die Dateinamen (beginnend mit dem Index 0) an.

Sonstige Events

Es gibt noch eine Reihe anderer Events die Sie abfangen können:

`wm rclick` wird ausgelößt wenn Sie mit der rechten Maustaste in das Fenster klicken. Wenn Sie keinen rclick Handler so wird automatisch der Zustand der shift-Taste invertiert.

`wm ncrclick` rechter Mausclick in non-client area (z.B. Title, minimiertes Window).

`wm mclick`, `wm ncmclick` werden nur ausgelößt wenn Sie eine Drei-Knpof Maus haben.

`wm queryopen` wird beim Versuch ein minimiertes Fenster wiederherzustellen ausgelößt. Wenn Sie einen queryopen Handler programmieren setzen Sie die Variable queryopen auf 0 um das Fenster nicht zu öffnen, sonst ungleich 0.

`wm drop`	Default drop Handler.

Benutzerdefinierte Events

Mit der Schlüsselwort `user` können Sie eigene Events anlegen. Dieses Event kann asynchron mit der enter() Funktion ausgelößt werden.

```
/F2 "Test"/ {
        status("lößе event 'test' aus")
        enter("test")
        beep();
        }

/user test/ {
        inform("enter", "user test event handler");
        }
```

Die enter() Funktion beendet sofort die weitere Interpretation, d.h. die beep() Funktion bei F2 wird nicht ausgeführt.

Benutzer Funktionen

Mit dem Schlüsselwort FUNCTION legen Sie eine eigene Funktion an:

```
BEGIN {
        x = 7, y = 0;
        printf ("x= %g", x);
        }

FUNCTION test(y) {
        local              x;

        x = y * y;
        x[i++] = x;

        printf ("local y= %g, x= %g, i= %g, *(\"x\")= %g",
                y, x, i, *("x"));

        return ((x % 2)? "ungerade": x);
}
```

Die Variablen in der formalen Parameterliste und hinter dem Schlüsselwort `local` sind lokal. Alle Variablen die weder formale Parameter noch lokale Variablen sind werden automatisch als globale Variablen definiert.
Es werden nur skalare (nicht indizierte) Variablen lokal gespeichert (die Konstruktion `x[i] = x;` weißt dem globalen Arrayeintrag `x[i]` den Wert der lokalen Variablen `x` zu).

Die Funktionen sind nicht reentrant, d.h. Sie können Funktionen insbesondere nicht

rekursiv aufrufen. Überhaupt sind maximal nur 6 verschachtelte Funktionsaufrufe möglich.

Die Parameterübergabeart ist call-by-value, als aktuelle Parameter können also beliebige Ausdrücke, aber keine Variablen übergeben werden:

```
/F2 "x= "/ { printf ("x= %g", x); }

/F3 "test #1"/ {
       y = test_x(i);
       inform("Test 1", "Ergebnis= " y)
       }

/F4 "test #2"/ {
       y = test_x(x, 2*x);
       inform("Test 2", "Ergebnis= " y)
       }

BEGIN {
       i = 0;
       delete (x);
       }
```

Beim Aufruf können Sie beliebig viele Parameter angeben, sowohl mehr als auch weniger als die Anzahl der formalen Parameter:

```
FUNCTION sum(x) {
       local   i;

       if ($# == 0)
               return (0);
       else {
               for (i=1; i<$#; i++)
                       x += $i;
               }

       return (x);
       }

/F4 "test #2"/ {
       p = input("Summations Test",
                 "Geben Sie einige," &
                 " durch Kommata getrennte, Zahlen ein.",
                 "", args)
       if (p != "\z") {
               call = "sum( " (args = p) ")"
               sum  = eval(call);
               printf ("summe von %s = %g", args, sum);
               }
       else
                          printf ("summation abgebrochen");
       }
```

Sämtliche Parameter sind wie oben gezeigt über $-Sequenzen abrufbar. $*expr* erwartet einen numersichen Ausdruck und liefert immer Strings, $# liefert immer die Anzahl der

Parameter als Zahl.

Die Funktion speichert ihre lokalen Variablen in einer eigenen Section in der Variablendatei. Diese Section wird beim Eintritt in die Funktion gelöscht. Die lokalen Variablen der Funktion können jeweils nach einem Funktionsaufruf über einen gewöhnlichen Arrayzugriff weiterverarbeitet werden, z.B.:

```
x = sum["i"];
```

Beachten Sie dabei jedoch, das diese Variablen als Arrayvariablen immer aus dem externen Speicher kommen und somit immer den Stringdatentyp haben (während die Funktion arbeitet ist die Variablentypisierung normal).

Programmstart

Beim Programmstart können Sie auf der Kommandozeile mehrere Programmdateien (Sie müssen minestens eine angeben, sonst macht das Programm einen Beep und hört dann auf) und Programmparameter angeben.
Die komplette Kommandozeile hat folgendes Format:

```
sh [optionen] progfile1 [progfile2 ...] [: args ...]
```

Wenn eine Programmdateiangabe weder mit einer vollen Verzeichnisangabe (also einem backshlash, einem Punkt oder einem Schrägstrich), noch mit einer Laufwerksangabe beginnt, sucht sh die Datei automatisch im Shellverzeichnis.

Programmstartsequenz

Nachdem alle Programmdateien eingelesen und erfolgreich übersetzt wurden wird dem kompletten Programm eine Variablendatei zugeteilt. Der Name dieser Datei ist gleich dem Namen der ersten Programmdatei mit der Erweiterung .var.

Dann wird die var-Sektion der Variablendatei gelöscht, andere Sektionen bleiben unverändert, d.h. Arrays sind über mehrere Programmläufe statisch.
Desweiteren werden dann die Systemvariablen gesetzt und die optionalen Argumente in das ARGV-Array (sämtliche Kommandozeilenparameter hinter einem einzelnen Doppelpunkt, beginnend mit dem Index 0) eingetragen, ARGC gesetzt.

Zum Schluß wird das BEGIN Event ausgelößt und danach die letzten Einstellungen vorgenommen (Systemmenu, DDE Server etc.)

Programmoptionen

Optionen beginnen immer mit einem Minuszeichen (beide Arten von Schrägstrichen kennzeichnen absolute Verzeichnisnamen).

Es sind folgende Optionen möglich:

-i*iconno* — *iconno* gibt die Nummer des Icons an das dem Fenster zugeordnet werden soll. Sie können sich die Icons mit dem Programm Manager

ansehen (wählen Sie Programmeigenschaften, Anderes Symbol). Ist *iconno* gleich 0, so wird das Default Icon IDI_APPLICATION eingesetzt.

-d*varfile* *varfile* spezifiziert den Namen einer Variablendatei wenn Sie vom obigen Defaultverhalten abweichen wollen.

Erweiterung durch DLLs

Der shell Interpreter ist durch Funktionen aus DLLs erweiterbar. Ihre Funktionen müssen dafür folgenden Prototyp haben:

```
int __far __pascal __export __loadds function(HWND win,
                                    char *result, char *extra,
                                    int argc, char * __far
argv[]);
```

`win` gibt dabei das Windowhandle des Programmfensters an, `argc` die Anzahl der in `argv` enthaltenen Argumente. `result` zeigt einen Puffer der das Ergebnis aufnimmt, `extra` wird (z.Zt. ?) nicht benutzt.
Die Funktion selbst muß 0 zurückgeben.

Ihre Funktion muß außerdem einen Eintrag in der `sh.cfg` Datei in der `prototypes` Section haben:

```
name=lib.function proto
```

`name` ist der Name, den Sie im Interpreter für die Funktion *`function`* in der Bibliothek *`lib`*`.dll` verwenden wollen.
`proto` ist eine Buchstabensequenz. Der erste Buchstabe gibt an ob die Funktion einen String, geben Sie dazu `s` an, oder eine Zahl (als String in `result` codiert), Angabe `n`, zurückgibt.
Für jeden Parameter der Funktion können sie `e` (expression), `n` (numerisch) oder `s` (string) einsetzten. Der Funktion wird jedoch in jedem Fall ein String in `argv[]` übergeben.
Wenn Ihre Funktion eine Parameterliste mit variabler Länge erwartet, setzen Sie für den Start dieser variablen Liste (als letztes Zeichen) ein `+` ein.

Verweildauer einer DLL im Speicher

Wenn der Interpreter eine DLL in den Speicher lädt, weil Sie eine Funktion aus dieser DLL in Ihrem Programm verwenden (shlib2.dll ist so eine DLL), dann bleibt Sie bis auf weiteres im Speicher. Der Interpreter selbst sorgt nicht dafür, daß eine von ihm geladene Bibliothek wieder aus dem Speicher entfernt wird.
Verwenden Sie die quit() Funktion um eine DLL definitiv freizugeben.