Dieser Text ist eine Ergänzung zur Spachreferenz shell1.wri, er soll die shell Sprache an verschiedenen Beispiele erläutern.
Es soll gezeigt werden, wie die einzelnen Funktionen zusammen arbeiten; shell1.wri beschreibt nur die Funktionen, Operatoren etc., ohne auf irgendwelche Zusammenhänge einzugehen.

Die hier gezeigten Beispiele sind mehr oder weniger auf meinen Rechner und meine Bedürfnisse zugeschnitten. Wahrscheinlich können Sie sie nicht direkt übernehmen. Das ist auch garnicht Sinn der Sache, im Gegenteil, Sie sollen mit sh Ihre eigenen Probleme lösen. Ihre eigene Probleme aber haben mit meinen wahrscheinlich nicht viel gemeinsam. Modifizieren Sie meine Beispiele bis sie Ihren Anforderungen entsprechen.

Dieser Text soll Ideen und Zusammenhänge vermitteln.

Ein Programmassistent

Als ich anfing den shell Interpreter zu programmieren hatte ich folgende Idee: Es wäre eine Tolle Sache, wenn ich eine gewisse Anzahl frei programmierbarer Buttons (einfach und mit schnellem turn-around Verhalten) hätte von denen aus ich z.B. Programme starten könnte. Ja ja, es gibt ja den Progamm Manager, aber der ist nicht immer einnfach griffbereit, sondern ist i.a. bei nur als Icon zu sehen. Bin ich dann im Programm Manager muß ich die Programmgruppe suchen, ggf. aufklappen und kann dann mein Programm starten. Dann muß ich den Programm Manager wieder verkleinern um zu meinem Ausgangspunkt zurück zukehren. Das kann es doch wohl nicht sein.

Da der Programm Assistent der Auslöser war, möchte ich Ihnen dieses Programm auch als erstes zeigen.

```
/F2 "start application"/ {
                x = select(250, 100, "Applications", lastapp,
                                                enum(application),
""")
                if (x != "\z") {
                        x = application[lastapp = x];
                        printf ("%s", x);
                        exec(x, 1);
                        }
                }

BEGIN {
                ctrl("application assistent")
                }
```

Zunächst wird nur die F2 Taste belegt. Wenn Sie F2 auslößen (entweder die Funktionstaste oder mit der Maus auf den Button klicken) können Sie aus den vorhandenen Programmen eins auswählen (oder abbrechen) das dann gestartet wird. Die Auswahl geschieht durch

```
x = select(250, 100, "Applications", lastapp,
                                                enum(application),
""")
```

in einer Listbox 250 Pixel breit und 100 hoch, der Text "Applications" steht im Titel, `lastapp` enthält den von Ihnen zuletzt ausgewählten Eintrag, der beim Aufruf von

`select()` vorselektiert wird.
Die Programme selbst sind im assoziativen Array `application` gespeichert, Sie könnten neue Einträge in das Array z.B. durch Zuweisungen der Form

```
application["Textverarbeitung"] = amipro
```

vornehmen. Der Indexstring "Textverarbeitung" wird dann in der Listbox angezeigt und `amipro` ist das Programm das dann gestartet werden soll.

Die oben gezeigte Methode des Arrayaufbaus ist jedoch nicht die, die ich Ihnen empfehle. Sie können problemlos direkt die Einträge in der Variablendatei vornehmen. Starten Sie hierzu den Assistenten mit

```
sh apa
```

und wählen Sie "Variablen" aus dem Systemmenu. Damit das funktioniert muß der Eintrag editor in der Datei sh.cfg, Sektion config ihren (oder irgendeinen) Editor, z.B. notepad, angeben.
Wenn Sie einen Editor gesetzt haben wird die Variablendatei in den Editor geladen und Sie können die Arrays direkt aufbauen, z.B.:

```
[application]
Amipro=d:\amipro\amipro
Excel=excel
Kermit=c:\win\local\kermit\win100.exe
DOS Shell=command.com
Q+E=qe.exe
```

Sie können auf diese Art und Weise die Variablendatei zur Laufzeit des Assistenten beliebig oft ändern. Sie sollten die Datei jedoch nicht direkt in Ihren Editor laden, da der Interpreter einige Variablen (nur nicht indizierte Variablen, Arrayeinträge sind immer extern) intern hält. Diese Variablen werden automatisch extern gespeichert wenn Sie den Systemmenueintrag wählen.
Hier liegt vielleicht auch ein kleines Problem. Eine Variable die aus dem externen Speicher geladen wird (und das werden sie nach dem Schreiben auf Platte alle) hat immer den Stringdatentyp. Beachten Sie das wenn Sie Variablen vergleichen oder Wertzuweisungen vornehmen.

Arrays werden immer extern (auf Platte) gespeichert. Das mag Ihnen ineffizient vorkommen (Plattenzugriffe sind ja so langsam). Aber ich kann Sie da ein wenig beruhigen. Windows hält die (von irgendeinem Programm) zuletzt benutzte .ini Datei (die Variablendateien sind ganauso aufgebaut, haben nur eine andere Erweiterung) in einem internen Speicher was die Sache beschleunigt. Solange Sie außerdem interaktiv mit dem Assistenten arbeiten (ihn also nicht in allzu lange Programmschleifen schicken) sind Sie das mit Abstand langsamste Glied in der Kette.

Die Sache hat aber auch ihre Vorteile. Zum einen sind Arrayvariablen statisch über mehrere Programmstarts, ohne das dafür irgendein Aufwand betrieben werden muß, zum anderen gestattet Ihnen die Methode Ihre Datenarrays auf eine sehr einfache Art aufzubauen, alles was Sie brauchen ist ein Texteditor (beim Entwurf stand ganz klar die einfache Bedienung und nicht die Effizienz im Vordergrund, da kann ich dann ja gleich C nehmen).

Zurück zum Programmm:

```
                    if (x != "\z") {
```

testet, ob Sie die Auswahl abgebrochen haben (wählen Sie dazu "Schließen" aus dem Systemmenu der Listbox). Übrigens, jede Eingabedialogfunktion liefert `"\z"` wenn Sie den Dialog abbrechen.

```
                        x = application[lastapp = x];
```

setzt die Variable `lastapp` auf Ihre jetzige Auswahl und holt das Programmstart Kommando aus dem Array `application`.
Das nächste, was passiert, ist das der Interpreter das Startkommando in die Statuszeile schreibt und das Programm startet.

```
                        printf ("%s", x);
                        exec(x, 1);
                        }
```

Diese paar Programmzeilen realisieren zusammen mit einem Datenarray einen einfachen und leicht modifizierbaren Programmstarter.
Der F3 Button ist vollkommen analog definiert (die Unterscheidung zwischen Programmen und Utilities ist ziemlich willkürlich):

```
/F3 "start utility"/ {
                if ((x = select(250, 100, "Utilities",
lastutil,
                                enum(util), "")) != "\z") {
                        x = util[lastutil = x];
                        printf ("%s", x);
                        exec(x, 1);
                        }
                }
```

Der F1 Button soll mir (der guten alten Tradition gemäß) immer irgendeine Hilfe Datei aufrufen:

```
/F1 "get help"/ {
                if ((x = select(250, 100, "Helpfiles", "",
                                enum(help),"")) != "\z") {
                        exec("winhelp " help[x], 1);
                        }
                }
```

und shift+F2 wird mit der Funktion belegt beliebige Programm zu starten:

```
/shift F2 "launch command"/ {
                if ((x = input("Launch Command", "Command",
                                "", lastcmd)) != "\z") {
                        printf ("%s = %s", x, exec(lastcmd =
x, 1))
                        }
                else
                        printf ("cancelled");
                }
```

Damit ist der (sicher sehr einfache) Programm Assistent fertig. Um ihn immer und jederzeit erreichbar zu haben können Sie jetzt einen Hotkey installieren, z.B. Alt+F2, und das Fenster in den Vordergrund legen.

Die BEGIN Section sähe dann so aus:

```
BEGIN {
                ctrl("application assistent",
                     2, 3, 113)
                }
```

Sie können beliebig viele Instanzen von sh starten (jedenfalls solange Sie noch ausreichend Speicher haben). Sie können also auch beliebig viele Programm Assistenten starten. Diese verschiedenen Instanzen wären jedoch alle gleich (der einzige Unterschied wäre vielleicht das Startup Verzeichnis), da sie alle dieselbe Variablendatei (apa.var) benutzen.
Wenn Sie die Assistenten jedoch mit

```
sh -d./apa.var apa
```

starten, erhält jeder Assistente eine eigene Variablendatei in seinen Startup Verzeichnis und Sie müssen für jeden dann die Datenarray aufbauen. Das dürfte wiederum am einfachsten gehen, wenn Sie den Assistenten einfach starten und dann (wie bereits oben) "Variablen" aus dem Systemmenu wählen.

Eine andere Methode lokalität zu ist folgende. Auf meiner Platte sind unterschiedliche Projekte ín unterschiedlichen Verzeichnissen. Außerdem habe ich im allgemeinen projektspezifische Anforderungen an den Programm Assistenten. Also lege ich mir in einen Projektverzeichnis einen Datei an, die verschiedene Tasten mit speziellen Funktionen belegt (das folgende Listing ist nicht bei den Demo Dateien).

```
/F2 "testbutton"/ {
                printf ("dies ist nur ein Test.")
                }

/F5 "start shell"/ {
                x = input("shell Parameter", "Parameter", "",
args);
                if (x != "\z") {
                        exec("sh " (args = x), 1)
                        }
                }

/F6 "codeview shell"/ {
                x = input("shell Parameter", "Parameter", "",
args);
                if (x != "\z") {
                        exec("cvw i " (args = x), 1)
                        }
                }
```

F5 startet den shell Interpreter mit Parametern, die vorher noch eingelesen werden, F6 lädt die CodeView Version des Interpreters mit Parametern. Wenn ich jetzt den Interpreter im Projektverzeichnis mit

```
sh ./local apa
```

starte besteht das komplette Programm aus beiden Einzeldateien und der Interpreter

benutzt die Variablendatei `./local.var` (die apa Datei bildet jetzt eine Art Vorlage (Template), das Sie an Ihre jeweiligen Anforderungen anpassen).
In der obigen Form überschreibt die apa Datei die Funktion der Taste F2 mit ihrer eigenen Version, weil sie hinter der Datei ./local kommt.
Umgekehrt ist es bei

```
sh apa ./local
```

In diesem Fall wird F2 mit der lokalen Variante belegt und der Interpreter verwendet apa.var als Variablendatei (die Variablendatei leitet sich immer aus der ersten Programmdatei ab).
Zu guter letzt haben Sie bei

```
sh -d./local.var apa ./local
```

lokale F2 Belegung und lokale Variablen.

Mit dem Eintrag "Variablen" aus dem Systemmenü haben Sie zur Laufzeit des Programms die Möglichkeit die zum Programm gehörenden Variablen zu editieren. Analog dazu "Konfiguration" die Konfigurationsdatei *winpath*`/sh.cfg` in Ihren Editor (*winpath* bezeichnet Ihr Windows Verzeichnis).

Sie können aber auch das Programm selbst "interaktiv" verändern. Mit dem Eintrag "Programmdatei" wird eine der Programmdateien in den Editor geladen. Sie können sie dann modifizieren, abspeichern und die neue Version mit "Neustart" laufen lassen.

Mit der "Neustart" Funktion ist aber ein vielleicht seltsames Variablenverhalten verbunden.
Beim Start der zweiten Instanz löscht diese in der Variablendatei die `var` Section. Das betrifft auch die erste Instanz (beide arbeiten mit derselben Variablendatei), jedoch hat die erste Instanz wahrscheinlich einige Variablen intern gespeichert. Wenn Sie jetzt die erste Instanz beenden werden diese Variablen auf Platte geschrieben. Diese Variablenwerte wird die zweite Instanz in der Folge benutzen, obwohl sie die garnicht gesetzt hat.
Ich habe jetzt einige Zeit mit diesen Funktionen gearbeitet und irgendwie ist dieses Verhalten nie zum Problem geworden.

Außerdem werden, wenn Sie "Neustart" benutzen, die DDE Ports und die DDE I/O Kanäle deaktiviert (Sie können in der ersten Instanz nicht mehr mit DDE arbeiten), sowie ein etwaiger auf die erste Instanz gesetzter Hotkey gelöscht.
Dadurch können Sie sicher sein, daß etwaige installierte globale Eigenschaften der ersten Instanz die zweite stören.
Sollte die zweite Instanz nicht hochlaufen (d.h. der Interpreter meldet einen Compiler Fehler und bricht ab), so können Sie die erste Instanz in jedem Fall noch als 'Entwicklungs Relais' benutzen.

Write Makros mit gkbd()

Die gkbd() Funktion generiert Tastaturevents. Diese werden dann von Windows an die Anwendung weitergeben, die den Eingabefokus hat.

```
gkbd("Hello World!");
```

sendet die Tasten für "Hello World!" an jedes beliebige Programm (und sei es eine DOS Box). Man kann insbesondere mit dieser Funktion Abläufe in anderen Programmen automatisieren, die sowas eigentlich nicht unterstützen, also z.B. einen Write Assistenten programmieren.

Ein Problem besteht allerdings darin die Zielapplikation (zuverlässig !) zu aktivieren bevor die Tasten losgeschickt werden. Mir fallen dazu zwei Lösungen ein (es mag ohne weiteres noch andere geben):

Die eine Methode funktioniert nur dann, wenn aus der Zielapplikation (also z.B. Write) direkt in den Assistenten gewechselt wird. Die Tastenkombination Alt+Tab schaltet nämlich aus einer Anwendung in die zuletzt aktive um. Also könnte man vor jedem Tastenmakro die Sequenz "`%{tab}`" absetzen um die Applikation zum Empfänger des Makros zu machen.

Die andere eröffnet sich einen, wenn der Assistent selbst eine Kopie von Write startet. Er erhält dann zunächst ein Task-Handle das in ein Top-Level Window-Handle umgewandelt werden kann:

```
win = exec("write", 1) |: appwin();
```

In der Folge kann dann mit `focus(win)` diesem Fenster der Fokus übergeben werden.

So ein Assistent sollte natürlich schnell ansprechbar sein. Es wäre ziemlich farblos, wenn man ihn erst durch Strg+Esc über den Task Manager aktivieren müßte. Abhilfe schafft hier erstens ein Hotkey, und zweitens ein Assistenten Window das immer im Vordergrund liegt.

Die BEGIN Section sieht dann ungefähr so aus:

```
BEGIN {
                ctrl("write assistent", 2, 3, 123)

                win = exec("write", 1) |: appwin();
                printf ("window = %d", win);
                }
```

`ctrl(3, 123)` setzt dabei Alt+F12 als Hotkey ein. In der obigen Form hätte ein zweiter Write Assistent (den Sie starten wenn der erste noch läuft) keinen Hotkey, da Alt+F12 ja bereits belegt ist.

Zu den Funktionen die ich so brauche gehört zunächst einmal das Einschalten des Helvetica- und des Courier-Fonts:

```
/F2 "write aktivieren"/ {
                focus(win);
                }

/F3 "Helvetica 10"/ {
                focus(win);
                gkbd("%fahelvetica{tab}{tab}10{down}{enter}");
                }

/F4 "New Courier 10"/ {
```

```
                focus(win);
                gkbd("%facourier new{tab}{tab}10{down}
{enter}");
                }
```

Wichtig ist bei der gkbd() Funktion ist das als Parameter ein String angegeben wird der immer eindeutig funktioniert. Da wo wir als Anwender interaktiv beurteilen, ob wir noch Tasten drücken müssen hat die gkbd() Funktion keinerlei Feedback.

Wenn ich mir diesen ersten Assistenten jetzt so ansehe und mal damit herumprobiere finde ich, daß der Assistent eigentlich ein schlechtes Turnaround Verhalten hat. Um die Tastensequenzen zu ändern muß man den Assistenten und Write beenden, den Assistenten umprogrammieren und neu starten.

Kommen wir also zu einer zweiten, flexibleren Version:

```
BEGIN {
                ctrl("write assistent", 2, 3, 123)

                win = exec("write", 1) |: appwin();
                printf ("window = %d", win);
                }

/F2 "write aktivieren"/ {
                focus(win);
                }

/F3 "Helvetica 10"/ {
                focus(win);
                gkbd(MACRO["helv10"]);
                }

/F4 "New Courier 10"/ {
                focus(win);
                gkbd(MACRO["cour10"]);
                }
```

Bei dieser zweiten Version werden die eigentlichen Steuersequenzen im Array MACRO in der Variablendatei des Assistenten gespeichert. Wenn Sie Makros ändern oder neue eingeben wollen dann editieren Sie die Variablendatei (Variablen im Systemmenu).

```
[macro]
helv10=%fahelvetica{tab}{tab}10{down}{enter}{F5}
cour10=%facourier new{tab}{tab}10{down}{enter}{F5}
```

Die beiden Sequenzen wählen übrigens nun nicht mehr nur den Font aus, sondern stellen auch normalen Text ohne Attribute ein.

Weitere Funktionen wären das Einstellen der richtigen Zeileneinzüge.

```
[macro]
1.5=%ai1.5{tab}0{tab}0{enter}
2.0=%ai2.0{tab}0{tab}0{enter}
0.0=%ai0{tab}0{tab}0{enter}
hängend=%ai5{tab}-3.5{tab}0{enter}
```

Beachten Sie bei diesen Sequenzen, daß ich auf meinem System 1.5 und nicht 1,5 eingeben muß, da ich den Dezimalpunkt als Dezimalzeichen eingestellt habe. Die Demodatei arbeitet hingegen mit dem Kommata.

So wie ich die Einzüge brauche schalte ich den 2.0er Einzug immer zusammen mit dem Courier Font ein, den 1.5er und den hängenden immer mit Helvetica.

```
/F5 "normaler Text"/ {
                focus(win)
                gkbd(MACRO["1.5"])
                gkbd(MACRO["helv10"])
                }

/F6 "Beispiel"/ {
                focus(win)
                gkbd(MACRO["2.5"])
                gkbd(MACRO["cour10"])
                }

/F7 "kein Einzug"/ {
                focus(win)
                gkbd(MACRO["0.0"])
                }

/F8 "hängender Text"/ {
                focus(win)
                gkbd(MACRO["hängend"])
                gkbd(MACRO["helv10"])
                }
```

Damit habe ich jetzt einen kleinen Write Assistenten, der mir ab jetzt beim Schreiben dieser Dokumentation helfen wird.

Anmerkung zu gkbd()

Die gkbd() Funktion hat ein Problem, wenn sie von einem shift-key Event (also z.B. `shift F8`) aus aufgerufen wird und Sie halten die shift-Taste drücken.
Wenn Ihr erstes Zeichen eine Alt-Taste ist dann können Sie `"+%"` statt `"%"` eingeben, das lößt offenbar das Problem.

Ein Trashcan

Ein Trashcan ist ein Programm, das im wesentlichen Dateien löscht. Die Dateien werden dem Trashcan durch Drag & Drop aus dem File Manager heraus übergeben, woraufhin der Trashcan die Dateien in eine andere Datei umkopiert und das Original von der Platte nimmt (BurnIt, ein verwandtes Programm, löscht die Dateien einfach beim drop).
Die Originaldatei ist damit tatsächlich entfernt, aber jederzeit wiederherstellbar. Ich kann dann erst einmal problemlos testen, ob ich die Datei wirklich löschen will und auch kann. Zu einem späteren Zeitpunkt, dann wenn ich meiner dann Sache sicher bin, werden die temporären Dateien dann tatsächlich gelöscht.

Der Trashcan (so wie er mir vorschwebt) arbeitet nur als Icon und immer im Vordergrund, anstatt Tasten werden Systemmenueinträge als Auslößer verwendet.

Zunächst zeige ich Ihnen das Drag & Drop Interface.

```
/drop "trashcan"/ {
```

installiert einen drop-handler für den Trashcan, er muß später noch eingeschaltet werden.

Während eines drop-events gibt DROPC die Anzahl der Dateien, DROPV[] die Dateiennamen an, für jede übergebene Datei wird ein eindeutiger Name für die Sicherungskopie gebraucht. Dieser wird über die tmpfile() Funktion besorgt (diese Funktion legt eine temporäre Datei in dem Verzeichnis an, auf das die DOS Umgebungsvariable TMP zeigt oder in dem, das der erste Parameter spezifiziert wenn TMP nicht gesetzt ist).

```
                for (i=0; i<DROPC; i++) {
                        x = tmpfile(".", "tc");
```

Da die gleich folgende move() Operation teilweise recht lange läuft, ich aber sicher sein will, daß das System überhaupt noch was tut, füge ich eine kleine Statusausgabe ein:

```
                        sprintf ("%d von %d: %s", i+1, DROPC,
                              DROPV[i]) |: ctrl();
```

Als nächstes wird die Datei umkopiert und dabei das automatisch Original entfernt:

```
                        move(DROPV[i], x);
```

Zu guter letzt merkt der Interpreter sich noch den Namen der Originaldatei und den Namen der Sicherungkopie in einem Array namens `trashcan`:

```
                        trashcan[DROPV[i]] = x;
```

Der Name der Originaldatei läßt sich in diesem Fall ganz gut als Index verwenden, da es sich ja um absolute Pfadnamen handelt die durch Drag & Drop übergeben werden (Primärschlüsseleigenschaft). Verwechslungen oder Kollisionen mit anderen Dateiname ist nicht zu befürchten.

```
                        }

                ctrl("trashcan " &
compute_len(enum(trashcan)))
                }
```

Die fraglichen Dateien werden so in einem Array auf Platte gespeichert und ich kann die Dateien da beliebig lange liegen lassen, bevor ich sie lösche.

Beim Löschen der Dateien stehe ich nun vor dem Problem eine for() Schleife über alle `trashcan` Indizes zu erzeugen. Dazu benutze ich ein Hilfsarray (ich glaube eine andere Möglichkeit gibt es nicht):

```
FUNCTION settext() {
                ctrl(abs(enum(trashcan)) " trashcan dateien")
```

```
                }

FUNCTION is_empty() {
                inform("Trashcan",
                       "Der Trashcan enthält keine Dateien");
                }


/menu "Trashcan &leeren"/ {
                delete(ii);
```

Zunächst mal hole ich mir dir komplette Indexmenge der `trashcan` Variablen und teste, ob sie nicht leer ist.

```
                if ((ii = enum(trashcan)) == "") {
                        is_empty()
                        }
                else {
```

Wenn nicht spalte ich die Variable `ii` in ein Array gleichen Namens auf

```
                        n = split(ii, ii, "\t");
```

und lasse dann die Schleife über die Anzahl der aufgespaltenen Indizes laufen:

```
                        for (i=0; i<n; i++) {
                                x = ii[i];
                                y = trashcan[x];
```

`x = ii[i]` kopiert den Namen der `i`-ten Originaldatei in die Variable `x` (die Originaldateinamen werden ja als Indizes verwendet),
`y = trashcan[x]` liefert dann den Namen der Sicherungskopie, diese Datei muß jetzt gelöscht werden.

```
                                ctrl(sprintf ("lösche %s
(%s)", y, x))
                                remove(y);
                                }
```

Noch ein cleanup, dann ist der Interpreter fertig.

```
                        delete(trashcan);
                        settext();
                        inform("Trashcan",
                               "Es wurden " n " dateien
gelöscht");
                        }
                }
```

Um Dateien aus dem Trashcan zu retten programmiere ich mir eine passende Listbox:

```
/menu "&Zurückholen ..."/ {
                while (1) {
                        if ((ii = enum(trashcan)) == "") {
                                is_empty();
```

```
                                break;
                                }

                        x = select(250, 174, "Trashcan
Dateien", "",
                        " [abbrechen]\t" ii, "")
                        if (x == "\z"  ||  x == "
[abbrechen]")
                                break;

                        y = trashcan[x];
                        move(y, x);
                        delete(trashcan, x);
                        }

                ii = "";
                settext();
                }
```

Die Listbox wird solange aufgerufen, bis ich abbreche oder der Trashcan leer ist.

Außerdem füge ich noch eine Funktion hinzu, dir den ganzen Trashcan leert, also alle Dateien wiederherstellt:

```
/menu "&Alle zurückholen"/ {
                delete(ii);
                if ((ii = enum(trashcan)) == "") {
                        is_empty()
                        }
                else {
                        n = split(ii, ii, "\t")
                        for (i=0; i<n; i++) {
                                x = ii[i];
                                y = trashcan[x];

                                ctrl(x);
                                move(y, x);
                                delete(trashcan, x);
                                }
                        }

                ii = ""
                settext();
                }
```

Was jetzt noch bleibt sind ein paar einfache Steuereinrichtungen. Erstens benutzt der Trashcan keine Tasten, sondern das Systemmenü. Das Standardsystemmenü will ich dazu abschalten. Um trotzdem die Variablendatei editieren zu können mache ich mir einen entsprechenden Eintrag:

```
/menu "Variablen ..."/ {
                ctrl(5);
                exec(*("sh.cfg!config")["editor"] " " varfile,
1);
                }
```

Diese Methode ist äquivalent zum Eintrag "Variablen" aus dem normalen Systemmenü, `ctrl(5)` sorgt dafür, das alle Variablen auf Platte geschrieben werden.

Zweitens soll der Trashcan nur als Icon laufen, also muß ich das Öffnen unterbinden:

```
/wm queryopen/ {
                queropen = 0;
                }
```

Zuallerletzt kommt noch die BEGIN Section:

```
BEGIN {
                DROPmode = "trashcan";
                ctrl(1, 2, 4, 20);
                settext();
                ownsysmenu = 1;
                }
```

`ctrl(1, 2, 4, 20)` minimiert das Fenster, stellt es in der Vordergrund und setzt Icon Nummer 20 ein.

Auf meinen Rechner läuft der Datei Manager standardmäßig immer mit. Wenn das auf Ihrem System nicht so ist, dann können Sie ja einen Eintrag in das Systemmenü aufnehmen, der der Datei Manager startet:

```
/menu "Datei Manager"/ {
                exec("winfile", 1);
                }
```

Ein Kommandozeileninterpreter

Nun wie im Text tty.wri erwähnt kann mit dem Programm tty.exe eine (mehr oder weniger) normale Terminal Ein / Ausgabe realisiert werden.
Die Kommandozeilenshell die mir vorschwebt nimmt beliebige Interpreterausdrücke entgegen, wertet sie mit der eval() Funktion aus und schreibt das Ergebnis aus die Ausgabe.

```
string | "tty|0!print"
```

gibt *string* auf das Default Ausgabefenster des TTY-Programms aus.
Von diesem Standardausgabefenster ist jedoch keine Eingabe möglich, so daß zuerst ein privates IO-Fenster angelegt werden muß. HIerfür steht ein DDE Port des TTY-Programms zur Verfügung.

Um überhaupt zuverlässig ein Ausgabefenster anlegen zu können muß ich zunächst testen, ob tty.exe überhauput läuft:

```
if (getmod("tty") == 0) {
                inform(DDEservice,
                       "Starten Sie zunächst das TTY
Programm.");
```

```
                exit();
                }
```

Das obige Programmstück sieht nach, ob das Programm läuft, wenn nicht gibt es eine Meldung aus und stoppt.

getmod() testet, ob sich ein betimmtes Programmmodul im Speicher befindet oder nicht, und liefert das Modulhandle zurück (oder 0, falls es nicht geladen ist). Das hat allerdings nicht zwingend etwas mit den von einem Programm angebotenen DDE Diensten zu tun.
Bei einigen Anwendungsprogrammen (z.B. Excel, QE, Amipro u.a.) ist die Äquivalenz "Modul geladen = DDE Service verfügbar" erfüllt.

Wenn Sie jedoch testen wollen, ob ein bestimmter DDE Service bereits von einem Modul mit anderem Namen, z.B. von einem shell93 Interpreterprogramm angeboten wird so können Sie das nicht getmod() Funktion. Insbesondere liefert `getmod("shell93")` immer einen Wert != 0 sobald der Interpreter läuft.
Hier brauchen Sie eine andere Technik: Sie versuchen den DDE Service anzusprechen, ja nach Statuscode läuft er dann oder nicht.

```
if (("" | "tty|0!print") != 0) {
                close("tty|0!print")
                ...
                }
```

Beachten Sie, daß auch wenn die Verbindung nicht zustande kommt der Ein / Ausgabe Kanal geöffnet bleibt, also explizit geschlossen werden muß, wenn Sie ihn später wieder verwenden wollen.

Im Prinzip könnten Sie auf die Idee kommen das Programm zu starten wenn es nocht nicht läuft und dann den DDE Kanal zu benutzen:

```
if (getmod("tty") == 0) {
                exec("tty", 1)
                }
```

Das wird jedoch i.a. nicht funktionieren, da das gestartete Programm nicht ausreichend Rechenzeit bekommt um die DDE Ports zu installieren.

Zurück zum TTY-Programm. Man kann vom TTY-Programm neue (sog. private) Ein / Ausgabefenster anlegen lassen, der weitere Zugriff erfolgt dann über den Namen. Damit der Name nicht mit anderen Anwendungen kollidiert sollte er eindeutig sein.

Also wie konstruiert man einen garantiert eindeutigen Namen? Für Dateien können Sie die tmpfile() Funktion verwenden, die Ihnen immer einen eindeutigen, noch nicht gebraichten Dateinamen liefert. Aber wir haben es hier eigentlich nicht mit Dateien zu tun. Wir können problemlos den Wert der PROGwindow Variablen nehmen, sie enthält ein systemweites eindeutiges Fensterhandle, kein Fenster hat dieselbe Nummer.

```
tty = "tty|stdio " PROGwindow
```

liefert einen garantiert eindeutigen Namen.

```
BEGIN {
                DDEservice = "ci " PROGwindow
                ctrl(2, DDEservice, 1)
```

```
                if (getmod("tty") == 0) {
                        inform(DDEservice,
                               "Das TTY Programm läuft
nicht.");
                        exit();
                        }

                tty = "tty|stdio " PROGwindow
                DDEservice "|input" | tty "!open"

                stdout = tty "!print"
                interpret("\"system ready\"")
                }
```

Die Zeile

```
DDEservice = "ci " PROGwindow
```

liefert nach derselben Idee einen eindeutigen DDE Service Namen (wir brauchen einen, um die Eingabenachrichten von TTY aufzufangen).

```
DDEservice "|input" | tty "!open"
```

legt erstens ein neues Ein / Ausgabefenster an und zweitens definiert es einen DDE Service (`"ci " PROGwindow`) und eine DDE Topic (`"input"`) an die TTY die Eingabenachrichten schicken kann.

`interpret("\"system ready\"")` gibt die Meldung "`system ready`" und einen Prompt ("`> `") aus. Die interpret() Funktion ist eigentliche Leistungsträger des Programms.

```
FUNCTION interpret(text) {
                status (text)

                k = index(text, " ")
                key = (k != 0)? substr(text, 1, k - 1): text;
                cmd = (k != 0)? substr(text, k+1): "";
                if (key == "!")
                        exec(cmd, 1) | stdout
                else if (text != "")
                        eval(text) | stdout

                printf ("\n> ") | stdout
                return (0);
                }
```

Die Funktion sucht zuerst nach einem abspaltbaren Schüsselwort und interpretiert es gegebenenfalls. Ansonsten wird die Eingabe einfach an die eval() Funktion weitergereicht und das Ergebnis ausgegeben.

Die Annahme von TTY Input ist recht einfach:

```
/| input input/ {
                interpret(DDEinput);
                }
```

ebenso die Reaktion auf click Events im TTY:

```
/| input click/ {
                if (substr(DDEinput, 1, 1) == ">") {
                        DDEinput = substr(DDEinput, 3)
                        printf ("%s\n", DDEinput) | stdout
                        }
                }
```

In der vorliegenden Form gibt es Ihnen das Kommandozeileninterface die Möglichkeit jeden Ausdruck interaktiv an der Tastatur abzusetzen und direkt das Ergebnis zu sehen.

Wünschenswert wäre vielliecht noch eine cleanup Funktion, die das Ein / Ausgabefenster vom Bildschirm entfernt wenn der Interpreter beendet wird:

```
END {
                "" | tty "!close"       # falsch!
                }
```

Wenn Sie das tatsächlich tun bleibt TTY stehen und macht garnichts mehr, d.h. nimmt keine Eingaben an und zeigt auch keine Ausgabe mehr an.

Eine andere Sache, die Sie nicht tun sollten, ist die exit() Funktion benutzen, während Sie auf ein DDE Event reagieren. Zu den Dingen, die exit() erledigt gehört u.a. das Herunterfahren der DDE Ports und als Resultat sehen Sie die große weiße Message-Box die eine allgemeins Schutzverletzung bemängelt und Ihnen sagt das Sie raus sind.

Der oben gezeigt Kommandozeileninterpreter funktioniert, hat aber wohl einige unschöne Eigenschaften (z.B. keine Möglichkeit zum editieren der Eingabe).
Also habe ich hier noch eine zweite Version:

```
FUNCTION interpret(text) {
                status (text)

                k = index(text, " ")
                key = (k != 0)? substr(text, 1, k - 1): text;
                cmd = (k != 0)? substr(text, k+1): "";
                if (key == "!") {
                        status(cmd)
                        exec(cmd, 1)
                        }
                else if (key == "dos") {
                        status(cmd)
                        dos(cmd " >output");
                        f2c("output")
                        }
                else if (text == "bye")
                        exit()
                else if (text != "")
                        eval(text) |: status()

                return (0);
                }
```

```
/shift F9 "kommando ausführen"/ {
                x = input("interactive", "Kommando:",
"",lastcmd)
                if (x != "\z") {
                        interpret(lastcmd = x);
                        }
                }


BEGIN {
                ctrl(2, "interactive")
                }
```

Grundsätzlich sollten Sie, wenn Sie mit der eval() Funktion arbeiten, beachten, daß es im Compilerteil von shell93 offenbar noch einige Lücken bei denen ein Syntax Fehler gemeldet werden sollte, der Compiler aber einfach in eine Endlosschleife übergeht (das kann Ihnen natürlich auch bei der Programmübersetzung passieren).

Wenn Sie Kommandozeilen mögen sollten Sie den obigen Interpreter auf Ihre persönlichen Bedürfnisse zuschneiden (Funktionstasten belegen, eigene Schlüsselworte etc.) und einsetzen.
Sie können im Prinzip beliebig viele Instanzen des Kommandozeileninterpreters starten, sollten dabei aber darauf achten, das jede Instanz ihre eigene Variablendatei bekommt.

Sie können aber auch den Interpreter mit anderen Programmen kombinieren:

```
sh write interact
```

startet einen Write Assistenten der zusätzlich die shift F9 Taste mit dem Interpreter belegt hat. Außerdem benutzt der Interpreterteil die Variabledatei des Assistenten und überschreibt den Fenstertitel mit "interactive".
Hierzu kann man die BEGIN Section von interact berarbeiten:

```
BEGIN {
       if (WINtext == "")
               ctrl(2, "interactive")

       if (WINkey + 0 == 0)
               ctrl(3, 120)
       }
```

Das neue BEGIN Handling setzt den Window Titel nur noch dann, wenn er nicht bereits gesetzt wurde. Außerdem wird, wenn kein Hotkey definiert ist, Alt+F9 dafür eingesetzt.

Sie können jetzt ziemlich problemlos z.B. Ausgaben von DOS Kommandos in Ihr Write Dokument einfügen. Schalten Sie zunächst mit Alt+F12 in den Assistenten und setzen dort mit shift+F9 das Kommando "`dos dir *.exe`" ab. Dann gehen Sie zurück nach Write (Taste F2) und wählen "Einfügen" aus dem "Bearbeiten" Menü, markieren den eingefügten Text und tippen noch Alt+F12 und F6. Fertig.
Bei mir sieht das dann so aus (ja ja, die Umlaute):

```
Datentr„ger in Laufwerk D ist DRIVE2
Datentr„gernummer: 0B59-15DA
```

```
Verzeichnis von D:\LAB3\X

SH       EXE    140800 04.09.93   20:36
I        EXE    222648 04.09.93   20:09
TTY      EXE     70656 04.09.93   20:20
WRAPPER  EXE     25830 04.09.93   19:59
TEST     EXE     33092 27.08.93   11:25
       5 Datei(en)     493026 Byte
                      3475456 Byte frei
```

Sie können diesen und ähnliche Vorgänge natürlich noch weiter automatisieren.

Excel Tabellen

Kommen wir nun zu etwas völlig anderem, Excel Tabellen, und wie man Excel über DDE bedienen kann (wahrscheinlich lassen sich die hier gezeigten Methoden leicht auf die Tabellenkalkulationen andere Hersteller übertragen).

Die Beispieldatei xltab enthält ein Programm, daß in Excel Kalkulationen Funktionstabellen erstellt. Dieses Programm ist ziemlich umfangreich und unhandlich, so daß ich hier auf eine vollständige Wiedergabe verzichte.

Wenn Sie xltab ausprobieren möchten, können Sie dazu die Datei xldemo.xls als Kalkulation verwenden.
Der Tabellenbereich E11:H21 berechnet iterativ (sog. Newton Verfahren) eine Nullstelle einer Parabel (ja, für dieses Beispiel ist keine Iteration nötig).
Im Bereich F11:F21 stehen dabei x-Werte, die in die Formeln in G11:G21 und H11:H21 eingesetzt werden. G11:G21 berechnet einfach den Funktionswert der Parabel in Abhängigkeit von F11:F21, H11:H21 einen Schätzwert für die Nullstelle. Der in Zeile k errechnete Schätzwert wird in Zeile k+1 (für k= 0, 1, ..., 9) als neuer x-Wert eingesetzt. Die Iteration wird insgesamt 10-mal durchgeführt. H21 enthält den zuletzt berechneten neuen Schätzwert.
Als Startwert der Iteration wird in Zelle E11 der Wert der Zelle B4 eingesetzt.

Die Zellen alpha, beta und gamma (B5, B6 und B7) enthalten die Koeffizienten der Parabel, es ist y(x) = alpha*x^2 + beta*x + gamma. Im Bereich A11:C31 wird die Parabel im Bereich xa und xe = xa + 20 (B1 und B2) tabelliert und die Werte graphisch angezeigt.

Die Frage ist nun, wie verändert sich die Nullstelle (bzw. der zuletzt ermittelte Näherungswert), wenn einer der Koeffizienten verändert wird? Setzen Sie hierzu nun in die Koeffizientenzellen verschiedene Werte ein und beobachten Sie das Resultat in H21.

Aber wie kann man jetzt den Verlauf des letzten Näherungswertes für verschiedene Koeffizientenwerte tabellieren?
Nun einmal können Sie die Nullstelleniteration (oder was immer Sie ausrechnen) sooft kopieren wie Sie es tabellieren wollen, und die Ergebniswerte in einer Tabelle zusammenfassen. Das ist eine Methode die Sie immer dann anwenden können (und es vermutlich auch tun), wenn Ihr Verfahren in einer (sog. geschlossenen) Formel darstellbar ist. Für die Berechnung, die ich gewählt habe ist das allerdings ziemlich unhandlich.
Sie können auch Excel Makros programmieren, das geht. Natürlich.

Eine dritte Methode ist ein shell Programm zu schreiben, das die Sache erledigt.

Und damit kommen wird zum xltab Programm. Wenn Sie es noch nicht aufgerufen haben, so tun Sie es jetzt, indem Sie den shell Interpreter mit

```
sh xltab
```

starten.
Mit der Taste shift+F2 starten Sie (immer) eine Excel Instanz. Laden Sie dann (wenn noch nicht geschehen) die Datei xldemo.xls nach Excel.

Die F5 Taste übernimmt die in Excel augenblicklich markierte Zelle als die variable Zelle ('x-Wert' der Tabelle), F6 setzt die abhängige Zelle ('Funktionswert' der Tabelle).
Setzen Sie in die Zellen alpha, beta und gamma z.B. die Wert -1, 0 und 0 ein. Markieren Sie B7 (gamma) und klicken Sie im Excel Tabellierer auf F5. Jetzt haben Sie den gamma-Wert als den für die Funktiontabelle unabhängigen Wert eingesetzt. Wählen Sie in Excel die Zelle H21 (letzter Iterationswert) und klicken Sie im Tabellierer auf F6. Damit wird die Zelle H21 zum Funktionswert der (noch zu erstellenden) Tabelle.

Als nächstes legen Sie einen Spaltenbereich (keine ganze Spalte) mit x-Werten der Tabelle fest, z.B. J11:J21. Markieren Sie dazu den Bereich und klicken auf F7. Dann tragen Sie dort die Werte ein, die nacheinander in die unabhängige Zelle (B7) eingesetzt werden sollen.
Um die Tabellierung zu starten, bewegen Sie den Excel Cursor nun in die Zelle, in der der Tabellierer mit dem Eintragen der entstehenden Funktionswerte (spaltenweise) beginnen soll, also z.B. K11. Wenn Sie das haben und die Tabellierung starten wollen klicken Sie auf F8.

Wenn Sie der Beispielbelegung gefolgt sind, dann haben Sie jetzt im Bereich J11:K21 eine Tabelle die die Nullstellen (Näherungswerte!, gerade der für gamma = 0 liegt ziemlich falsch) in Abhängigkeit der verschiedenen Belegungen des gamma Koeffizienten, daneben die dazu gehörende Graphik.

Nun dem Tabellierer ist es vollkommen egal, was er eigentlich tabelliert. Ich habe dieses Beispiel bloß gewählt, weil mir nichts besseres einfiel.
Es ist ihm aber auch egal, über welche (und wieviele) Dateien sich die von Ihnen markierten Bereiche verteilen. Demenstprechend aufwendig ist das Programm xltab, Sie können es sich ja mal ansehen.

Besonders erwähnenswert scheint mir am xltab Programm die Funktion get_range() zu sein, aus zweierlei Gründen. Erstens zeigt Sie eine bisher noch nicht angesprochene Möglichkeit den Interpreter zu programmieren (eine Möglichkeit die Sie kaum aus anderen Sprachen kennen), und zweitens eignet sie sich gut als Bibliotheksfunktion für Ihre eigenen Programme.

```
FUNCTION get_range(range) {
                local   file, top, bottom, left, right, x

                if (index(range, "!") == 0)
                        return (1);
                else
                        file = substr(range, 1, rstart-1);

                x = substr(range, rstart+1);                #
ZaSb:ZcSd
                top = substr(x, 2) + 0;
```

```
                if (index(x, "S") == 0)
                        return (1);
                else
                        left = substr(x, rstart+1) + 0

                if (index(x, ":") == 0) {
                        bottom = top;
                        right = left;
                        return (0);
                        }

                x = substr(x, rstart+1)                 #
ZcSd
                bottom = substr(x, 2) + 0
                if (index(x, "S") == 0)
                        return (1);
                else
                        right = substr(x, rstart+1) + 0

                return (0)
                }
```

Die get_range() Funktion zerlegt eine Excel Bereichsangabe der Form

```
datei!ZaSb              oder
datei!ZaSb:ZcSd
```

in ihre Bestandteile. Die einzelnen Teile werden dabei in den lokalen Variablen `file`, `top`, `left`, `right` und `bottom` gespeichert. Die Funktion liefert 0 im Falle, daß der Parameter gültig war und zerlegt werden konnte, und 1 falls irgendein Fehler beim Aufspalten auftrat.

Nach einem get_range() Aufruf können nun die lokalen Variablen der Funktion über Arrayadressierung ausgelesen (auch neu geschrieben) werden:

```
get_range(eingabe)
varF = get_range["file"] "!"     # Achtung !
varZ = get_range["top"] + 0;     #    die Werte sind Strings!
varS = get_range["left"];        #
end  = get_range["bottom"] + 0;
```

Wichtig ist dabei, daß Sie darauf achten, das diese Adressierung auf externe Arrayvariablen zugreift und immer Strings liefert, selbst bei Variablen, die innerhalb der Funktion numerisch waren.
Da in der Folge mit `varZ` und `end` Vergleiche durchgeführt werden (hier ist das besonders wichtig), werden die Werte dieser Arrayvariablen explizit in Zahlen umgewandelt.

Eine andere Methode, einen Tabellierer zu programmieren, ist (wie oben schon erwähnt) mit Excel Makros, oder, und darauf gehe ich jetzt ein, ein shell Programm zusammen mit Excel Makros. Die Datei xltab2 ist ein solches Programm.
In der BEGIN Section werden zunächst einige Variablen gesetzt

```
BEGIN {
                ctrl(2, "Excel Tabellierer V2", 1, 4, 113)
                close("") |: status()
```

```
                DDEservice = "xltab"

                remoteDDE  = "excel|"
                remoteSYS  = "excel|system"
                SYS        = remoteSYS "!"
                SEL        = SYS "selection"
                MACROLIB   = "xltab.xlm"
                MACROFILE  = shellpath "\\" MACROLIB
```

und dann sichergestellt, daß midestens eine Kopie von Excel (im Prinzip können Sie Excel öfter starten) mit der xltab.xlm Datei läuft:

```
# excel und makrodatei laden
#
                if (getmod("excel") == 0) {
                        status("excel wird gestartet ...")
                        exec("excel " MACROFILE, 1)
                        }
                else {
                        if (index((SYS "topics" | get) |:
strlwr(),
                                  "\t" MACROFILE "\t") == 0)
{
                                status (file " wird
geöffnet ...")
                                cmd = "[open(\"" & MACROFILE &
"\")]"
                                cmd | remoteSYS
                                }
                        }
                }
```

Die Abfrage des Ports `"excel|system!topics"` liefert eine Tabulator getrennte Liste aller geöffneten Tabellen. Die Tabellennamen werden mit Pfadangabe geliefert.
Excel Makros können Sie über den Kanal `"excel|system"` absetzen. Dazu schließen Sie jedes Makro in eckige Klammer ein und geben den Makronamen in Englisch an (der shell Interpreter verwendet nur CF_TEXT für DDE Operationen, also gibt's da wohl Probleme mit dem 'Ö' in 'Öffnen'). Die Makroparameter müssen entweder Konstanten oder Zellen (ZS-Schreibweise sein), keine Excel Funktionen.

Die F3 Taste startet nun ein Makro, daß eine Dialogmaske anzeigt und die verschiedene Zellbereiche zur Übergabe an den Tabellierer vorbereitet.

```
/F3 "dialog"/ {
                ctrl(1)
                "[App.Activate(,0)]"              | remoteSYS
                "[Run(\"" MACROLIB "!dialog1\")]" | remoteSYS
                }
```

`App.Activate(,0)` ist wichtig, da Excel den Eingabefokus haben muß um zu laufen, die shell ihn aber noch hat. Beachten Sie außerdem, das zum Makrostart nur der Tabellenname, d.h. ohne Dateipfadangabe, angegeben werden muß.
Offenbar arbeitet das Excel Makro asynchron (ich weiß nicht, ob Excel über DDE grundsätzlich asynchron ist), deswegen ist `dialog1` so programmiert, daß Excel über DDE zurückruft:

```
/| xyTAB data/ {
                beep();
                n = split(DDEinput, data, "\r\n");

                variabel = data[0]
                ergebnis = data[1]
                eingabe  = data[2]
                ausgabe  = data[3]

                enter("xyTAB")
                }
```

Die Daten werden vom Porthandler nur entgegen genommen und dann das `user` Event `xyTAB` ausgelöst. Dadurch wird der DDE Transfer beendet und Excel kann sich neuen Dingen zuwenden, nämlich der Bearbeitung der Anweisungen die der Tabellierer gleich über DDE schickt. Ohne enter() geht es nicht.
Zur Datenübergabe: Wenn Excel Daten über DDE oder das Clipboard transportiert werden mehrere Zellen in einer Zeile durch Tabulatoren getrennt. Am Zeilenende wird immer ein `"\r\n"` angehangen. Im obigen Programmstück sendet ein Excel Makro einen Datenbereich mit 4 Zeilen und 1 Spalte. Die split() Funktion spaltet diesen Datenblock entsprechend auf.

Sie werden die beiden Programme vielleicht etwas verworren und kompliziert finden. Ich habe sie in dieser Form aus zwei Gründen geschrieben. Zum einen habe ich jetzt einen vollkommen tabellenunabhängigen Tabelliermechanismus (eine Sache die ich schon immer haben wollte), zum anderen wollte ich herausfinden, ob man mit dem Interpreter auch Dinge erledigen kann, die nicht ganz so einfach sind.
Man kann.

Schrieben Sie den Tabellierer in C, so würden Sie bestimmt, genau wie ich, großen Wert auf die Tabellenunabhängigkeit legen (das Tabellierprogramm macht also keine Annahmen über die von ihm zu berarbeitenden Kalkulationsabellen, bzw. stellt möglichst wenig Anforderungen an die Tabellen). Schließlich ist es ja doch ein ziemlicher Aufwand das Programm zu schreiben, und später ggf. noch zu modifizieren. Nun, für solch umfassende Programme ist der Interpreter nicht konzipiert, nichts desto trotz: Sie können es wenn Sie wollen. Ich habe stattdessen versucht die shell so zu programmieren, daß sie interaktive Programmentwicklung unterstützt, damit Sie bei einem konkreten Tabellierproblem den dazu passenden konkreten Tabellierer schnell und einfach erstellen können.

Ich schätze, daß wenn Sie das Prinzip verstanden haben (ich habe jetzt empirisch festgestellt, daß das eigentliche Problem in der mangelhaften Excel DDE Dokumentation liegt), einen konkreten (d.h. wenig flexiblen, mit einigen oder vielen Annahmen und Voraussetzungen über die Tabellen) Tabellierer in (maximal) 20-30 Minuten schreiben können. Dann haben Sie ihren eigenen Tabellierer fertig und können ihn benutzen, ohne sich lange den Kopf über alle möglichen Inputs, die das Programm bearbeiten müßte, zu zerbrechen. Brauchen Sie morgen jedoch einen anderen Tabellierer, dann nehmen Sie entweder den alten und modifizieren ihn entsprechend Ihren neuen Anforderungen, oder Sie schreiben eben einen neuen.
Wären Sie z.B. darauf gekommen, nach dem Einsetzen des unabhängigen Wertes ein Excel Makro abzusetzen, um z.B. den Solver zu starten? Mit wievielen Unabhängigen Werten soll der Tabellierer arbeiten? Zwei, drei? Meiner arbeitet mit einem. Wenn es mehr als einer ist, wie sind die Werte angeordnet? In Spalten oder als linker und oberer Rand einer Excel Matrix? Und wieviel abhängige Werte sollen tabelliert werden?

Aber das sind auch nur Parametrisierungsmöglichkeiten, die mir jetzt mehr oder weniger spontan einfallen, und bestimmt nicht alle möglichen.

Ich würde mich freuen, Ihre Meinung über, und von Ihren Erfahrungen im Umgang mit dem shell Interpreter zu hören.

Wolfgang Zekoll
wzk@dx3.informatik.uni-koeln.de