Entwicklung eines Kommandointerpreters

command.com im Eigenbau

Oliver Müller • Befehlseingaben in der Kommandozeile gelten im Zeitalter „bunter Bildchen“ als verpönt und veraltet. Aber sie haben auch unbestrittene Vorteile. Grund genug zu zeigen, was hinter einem Kommandozeilen- Interpreter steckt und wie man ihn programmiert.

Ein Kommandozeilen-Interpreter besteht im wesentlichen aus vier Teilen:

- einem Initialisierungsteil
- dem Editor
- dem Parser
- dem Ausführungsteil

Neben gewöhnlichen Initialisierungsarbeiten, die jedes Programm erledigen muß, erzeugt der Initialisierungsteil einige essentielle Datenstrukturen und setzt Variablen auf Ausgangswerte. Außerdem analysiert er die übergebenen Parameter und baut das Environment-Segment auf. Wird nämlich ein gewöhnliches Programm gestartet, bekommt es entweder vom Betriebssystem eine Kopie des Eltern-Environments oder einen explizit vom Elternprozeß angegebenen Umgebungsspeicher zugewiesen. Bei einem Kommandoprozessor verhält es sich etwas anders: Wird dieser nämlich als “Basis-Shell” aus der CONFIG.SYS heraus gestartet, übergibt ihm der DOS-Kernel keinen gültigen Umgebungsspeicher. Diesen muß der Befehlsinterpreter erst selber generieren.

■ Aufbau des Environments

In der hier entwickelten Beispiel-Shell CMD baut die Funktion init() das Environment auf. CMD erkennt In der Kommandozeilenanalyse anhand des Parameters /P, ob es sich um den Basisprozeß handelt. Wenn dem so ist, generiert es eine leere Umgebung von der Größe envSize und weist diese environ zu. Der erste Zeiger in das Environment (eptr[0]) wird dabei zum Null-Pointer und markiert so das Ende der Variablenliste. Läuft CMD aber als Kindprozeß (durch Aufruf in der Kommandozeile), so passiert zweierlei: Erstens wird der allokierte Speicher mit den Strings der Eltern-Task gefüllt und zweitens jeweils ein Zeiger des Arrays eptr auf die Adresse des kopierten Strings innerhalb von environ gesetzt. Der letzte Pointer (eptr[i]) wird hier auf Null gesetzt und fungiert ebenfalls als Endmarke.
Nachdem die Initialisierung mit dem Durchlaufen von init() abgeschlossen ist, tritt das System in die Hauptschleife ein. Dort ruft es die jeweiligen Routinen der Module Editor und Parser ständig nacheinander auf.

■ Verwaltung des Editors

Der Editor von CMD ist als Einzeileneditor konzipiert. Er ist in der Lage Escape, Backspace und Cursorpositionierung mittels der Pfeiltasten rechts und links korrekt zu interpretieren (allerdings gibt es lediglich einen Overwrite-Modus). In edit() existieren vier Variablen, die für die Verwaltung der eingegebenen Zeichen notwendig sind. Dies sind

- len (die Länge der Eingabe)
- cur (die aktuelle Cursorposition relativ zum Eingabe-String)
- sowie x0 und y0, die die Startkoordinaten bezogen auf einen 80x25-Zeichen-Display bezeichnen.

Bei Eingabe eines normalen Zeichens legt die Routine dieses Zeichen Im Pufferspeicher an der Position cur ab und erhöht diese Positionsvariable anschließend um eins. Entspricht dagegen die betätigte Taste der linken oder rechten Cursortaste, so wird cur einfach dekrementiert beziehungsweise inkrementiert. Die anschließende Neupositionierung der Schreibmarke erfolgt mit Hilfe des Makros curpos(): Es errechnet aufgrund von x0, y0 und cur die Bildschirmkoordinaten des Cursors und setzt ihn mittels gotoxy(). Die nächste Eingabe überschreibt anschließend das Zeichen unter dem Cursor (cmd[cur]).
Um einen Einfüge-Modus zu implementieren, müßten bei jeder Eingabe die Zeichen ab cur um eine Position verschoben und das Zeichen anschließend in cmd[cur] gespeichert werden. Anschließend wären noch cur und len zu erhöhen und die Anzeige zu aktualisieren.

■ Das Herzstück – der Parser

Die Leistungsfähigkeit einer Kommandosprache hängt nicht zuletzt vom Parser ab. Er ist unter anderem für die formale Funktionalität zuständig und entscheidet beispielsweise, ob es sich bei der Eingabe um ein Schlüsselwort (interner Befehl) oder um ein Executable (externer Befehl) handelt. Ebenfalls zu seinen Aufgaben gehört es, die Kommandozeile in Argumente aufzuspalten und den ausführenden Teil zu starten. Außerdem muß der Parser eventuell vor der Ausführung die Standardkanäle umleiten.
Die Routine parse() besteht aus zwei Abschnitten: Teil 1 ist dafür zuständig, die Eingabezeile in die einzelnen Argumente aufzuspalten, während Teil 2 untersucht, ob es sich um einen internen oder externen Befehl handelt und diesen gegebenenfalls ausführt.
Der erste Abschnitt besteht aus einer Schleife, die bis zur Null-Marke des übergebenen Kommando-Strings durchlaufen wird. Parse überspringt zunächst alle Whitespaces (wie Leerzeichen, Tabulatoren) und liest danach solange, bis das folgende Argument beendet ist. Dies ist dann der Fall, wenn entweder ein Whitespace, ein Null-Zeichen oder ein Anführungszeichen vorliegt. Ist das Argument auf diese Weise extrahiert, fordert die Routine Speicher an und weist Ihn einem Element von p zu. Anschließend wird das Argument dorthin kopiert und die Prozedur wiederholt. Dem auf den letzten Argumentenzeiger folgenden Pointer wird der Wert Null als Endmarke zugewiesen.

■ Ein/Ausgabeumlenkung

Stößt der Parser innerhalb des Aufspaltens der Argumente auf die Zeichen “>” oder “<”, ruft er die Subroutine redirect() auf. Dieses Unterprogramm extrahiert die Dateinamen aus der Kommandozeile und öffnet anschließend die zugehörigen Files. Gibt redirect() Null zurück, so ist ein Fehler aufgetreten, andernfalls ist die Argumentenaufspaltung abgeschlossen.
Bevor der Parser in den zweiten Abschnitt eintritt, untersucht er die Dateihandles out und in. Weisen diese nicht den Wert -1 auf wird das Handle des entsprechenden Standardkanals (zum Beispiel Handle von CON: für stdin) mittels der Funktion dup() dupliziert, das heißt, eine Sicherheitskopie erstellt (entspricht dem Aufruf der DOS-Funktion 45hex). Anschließend wird der Standardkanal durch den Aufruf von force() umgeleitet. In diesem Unterprogramm ist die DOS-Routine 46hex gekapselt. Diese Systemfunktion beschreibt der Kasten “Schlüssel zur Ein/Ausgabeumlenkung”. Nach einer solchen Umleitung erfolgen alle Ein- respektive Ausgaben in eine Datei – egal ob aus der Shell selbst oder aus einem externen Prozeß.
Nun beginnt der zweite Abschnitt des Parsings. Durch Analyse des Elements p[0] des Argumentenfeldes läßt sich feststellen, ob es sich um ein internes Kommando handelt. In der Beispielimplementierung sind lediglich die Befehle EXIT, CD, SET und TYPE realisiert. Diese stehen jedoch exemplarisch für jeweils eine Gruppe von ähnlichen Befehlen – eine individuelle Erweiterung sollte also kein Problem sein.

■ Einbau der Befehle

EXIT ist der einfachste Befehl, da er, vergleichbar mit CLS, keine Parameter erwartet. Hier ist eine Analyse von p[0] ausreichend. CD ist dagegen ein Kommando, welches sich nicht immer durch die Argumentenaufspaltung von seinem Parameter trennen laßt. Man denke an Angaben wie CD\DEVELOP\CMD. Hier müssen die ersten drei Zeichen von p[0] ausgewertet werden, wobei die ersten beiden CD ergeben müssen. Ist das dritte Zeichen dann ein Null-Byte, ein Punkt oder ein Backslash bei angehängtem Parameter, kann man je nach Art der Verzeichnisangabe durch chdir(&p[0][2]) oder chdir(p[1]) wechseln.
RD, MD und die “langen” Varianten wie CHDIR, RMDIR und MKDIR lassen sich auf die gleiche Weise implementieren. Hier muß im übrigen nur chdir() durch rmdir() beziehungsweise mkdir() ersetzt werden.
Die Programmierung von SET stellt dagegen höhere Anforderungen. Die Analyse innerhalb des Parser beschränkt sich auf das Argument p[0]. Danach erfolgt der Aufruf von set(), einer spezifizierten Funktion des ausführenden Moduls. Übergeben wird der Kommando-String CMD und die Argumentenanzahl, die mittels der Argumentenaufspaltung ermittelt wurde. Hier ist die Übergabe der gesamten Kommandozeichenkette nötig, da im Feld p die Whitespaces de facto eliminiert wurden. Leerzeichen und Tabulatoren sind jedoch für den SET-Parameter keine Trennzeichen, sondern Bestandteil der Wertzuweisung an die Umgebungsvariable.

■ Verwaltung der Umgebungsvariablen

Für gewöhnlich verwendet man die Funktion getenv() und setenv() der Standard-C-Bibliothek zur Verwaltung von Environment-Variablen. Dieses Verfahren greift hier aber nicht, da der Umgebungsspeicher explizit während der Initialisierung erzeugt wurde. Infolgedessen ist eine Speziallösung erforderlich:
Schritt 1 ist die Extrahierung des Parameters aus der Kommandozeile. Ob ein Parameter existiert, zeigt sich daran, welchen Wert die übergebene Argumentenanzahl argc besitzt. Ist argc gleich eins, wurde SET ohne Parameter aufgerufen. In diesem Fall sind nur die Strings der Umgebungsspeicher auszugeben.
Da Environment-Strings immer den Aufbau Variablenname=Wert besitzen, wird nach der Ermittlung des Parameter-Substrings die Position des Gleichheitszeichens ermittelt. Um die Variablenbezeichnung vom Wert zu trennen, schreibt man aber anstelle des Gleichheitszeichens ein Null-Byte und setzt auf das folgende Zeichen

den Zeiger ptr. Auf diese Weise existieren nun zwei Zeichenketten – eine für den Namen und eine für den Wert der Umgebungsvariablen.

■ Environment-Variablen

In dem Ausführungsmodul (EXEC.C) finden sich die Routinen del() und add() zum Löschen respektive zum Hinzufügen von Variablen. Dabei erfolgt, unabhängig davon, ob eine Variable gelöscht oder dieser nur ein neuer Wert zugewiesen werden soll, ein Aufruf von del(), denn auch im letzteren Fall löscht exec zuerst den alten Environment-String und fügt anschließend einen Neuen hinzu.
Die Funktion del() erwartet als Parameter den Namen der Umgebungsvariablen, die gelöscht werden soll. Diese Routine durchsucht zuerst das Environment nach dem übergebenen Namen. Findet es ihn nicht, liefert del den Index des das Ende markierenden Nullzeigers zurück. Im anderen Fall werden alle folgenden Environment-Strings um die Länge der zu löschenden Zeichenkette nach oben verschoben und die Zeiger des Vektors eptr entsprechend korrigiert. Als Rückgabe erfolgt ebenfalls der Index des Null-Pointers.
Soll die Variable aber nicht gelöscht werden, erfolgt der Aufruf von add() mit der Übergabe des von del() zurückgelieferten Wertes und des Namens der Variablen. Durch Ersetzen des abschließenden Null-Bytes des übergebenen String-Zeigers durch ein "=", wird die Trennung zwischen Namen und Wert wieder aufgehoben. Reicht der verbliebene Speicher aus, schreibt die Routine nun die Zeichenkette in das Environment und aktualisiert die Zeiger.

■ Befehle entwickeln

Der Parser verwendet ein Format, das es auch ermöglicht, interne Kommandos wie gesonderte Programme zu entwerfen. Das Array p und die Variable i, welche die Anzahl der Argumente enthält, können durchaus mit den formalen Parametern argv und argc der C-typischen Hauptroutine main() verglichen wenden. Die Umsetzung des Befehls TYPE veranschaulicht dies.
Der Parser startet die Routine type() und übergibt i und p. Innerhalb von type() wird nun so vorgegangen, als ob ein neues Programm geschrieben und der Befehlsinterpreter nicht existieren würde. In einer for-Schleife werden alle in der Kommandozeile übergebenen Dateien nacheinander geöffnet und deren Inhalt ausgegeben.
Selbst der Parameter env von main kann "simuliert" werden. Entweder der Zugriff erfolgt direkt auf eptr oder das Environment-Parameterfeld wird ebenfalls an das ausführende Unterprogramm übergeben.
Aufgrund von diesem Mechanismus ist eine einfache Erweiterbarkeit der Shell gegeben. Befehle wie COPY oder DIR sollten so kein größeres Problem mehr darstellen.
Wenn das erste Element des Argumenten-Arrays keinem Schlüsselwort entspricht, tritt die Subroutine call() in Kraft. Hierbei wird ein externer Prozeß gestartet. Als Parameter erwartet call lediglich das Argumentenfeld.

■ Starten externer Prozesse

In dieser Prozedur wird zunächst der Programmname mittels fnsplit() In seine Bestandteile zerlegt. Anschließend wird getestet, ob die Eingabe mit Erweiterung erfolgte. Ist dies nicht der Fall, werden an den Dateinamen nacheinander die Extensionen .exe, .com und .bat angehängt und im über PATH definierten Pfad gesucht. Wird kein passendes File gefunden erfolgt eine Fehlermeldung.
Die Beispiel-Shell CMD besitzt noch keinen Batch-Modus. Als Konsequenz hieraus folgt, daß call() keine Stapeldateien ausführen kann. Eine Anregung, wie dieses Feature zu realisieren ist, findet sich im Kasten "Auf den Stapel gelegt". Der Aufruf des Programmes erfolgt mittels der Bibliotheksfunktion spawnvpe(), welche als Übergabe ein Argumentenfeld und ein Environment-Array erwartet. Beides wurde vom Kommandoprozessor im geeigneten Format erzeugt. Nachdem der Befehl durch das ausführende Modul abgearbeitet ist, beginnt der Parser mit den Abschlußarbeiten. Die Ein/Ausgabeumlenkungen werden durch Aufruf von force() mit den duplizierten Handles wieder zurückgesetzt und der Speicher des Argumenten-Array wieder freigegeben.

■ Vorschläge zur Erweiterung

Eine sinnvolle Erweiterung für CMD wäre beispielsweise ein Control-C-Handler, damit unter anderem bei der Ausgabe von langen Dateien, diese abgebrochen werden könnte. Ein raffiniertes Verfahren könnte der unter Linux üblichen bash zum Ausbau des Editors von CMD abgeschaut werden. Hier ist es möglich, über die Tabulatortaste teilweise eingegebene Dateinamen zu erweitern. Unter Linux muß nicht immer der gesamte Pfad eingegeben werden, exemplarisch sei hier folgendes betrachtet:
Die Eingabe von /usr/bin/find kann in /u<TAB>b<TAB>find verkürzt werden. Sind mehrere passende Möglichkeiten verfügbar, ertönt ein akustisches Signal und alle möglichen Dateinamen werden aufgelistet. Die Kommandozeile kann nun entsprechend erweitert werden. Jeder, der schon einmal mit der bash gearbeitet hat und dann wieder mit COMMAND.COM hantieren mußte, wird den Komfort der Tab-Erweiterung zu schätzen wissen.

Außerdem gehört in diesen “modernen Zeiten” zu jedem Kommandointerpreter eine History-Funktion. In diesem Zusammenhang bieten sich die Cursor-Tasten hoch und runter an, deren Auswertung im Editor analog zu der links- und rechts-Taste erfolgt. Die Codes für hoch und runter sind 72 respektive 80.
Eine Alias-Verwaltung, ähnlich dem DOSKEY-Makros, würde dem Interpreter auch gut zu Gesichte stehen. Hier wäre innerhalb der Kommandozeile vor der Aufspaltung der Argumente durch den Parser die entsprechende Sequenz in der Befehlszeichenkette zu ersetzen. Komfortabel wäre auch die Einführung des Kommandos “noalias”, welches die Substitution des folgenden Befehls unterdrückt.

Literatur

[1] *Günther Born:* MS-DOS 6.2 Programmierhandbuch.
Unterschleißheim: Microsoft Press Deutschland 1993

Schlüssel zur Ein-/Ausgabeumlenkung

Die DOS-Funktion 46hex bietet die Möglichkeit Ein/Ausgabekanäle umzulenken.
Die Schnittstelle ist wie folgt definiert:

```
AH = 46hex          mov ah,46h      ; die Funktionsnummer
BX = neues Handle   mov bx,[hand]   ; Handle übergeben
CX = Kanal          mov cx,1        ; Umlenkung von stdout
CALL Int 21hex      int 21h
```

Falls nach dem Aufruf das Carry-Flag gesetzt sein sollte, ist ein Fehler aufgetreten.
Der Fehlercode befindet sich dann im AX-Register.

Die definierten Fehlerwerte sind:

4 : Zu viele offene Dateien, kein Handle mehr frei
6 : Handle ist nicht eröffnet oder ungültig

(Hinweis: BX = CX führt in DOS 3.3 zu einem Systemabsturz.)

Die Standardkanäle von DOS:

Kanal-Nr.	Name	Standardverknüpfung
0	Input device (stdin)	CON
1	Output device (stdout)	CON
2	Error device (stderr)	CON
3	Auxiliary device	AUX
4	Printer device	PRN

Checkliste für den Befehlseinbau

Die Argumente einer Befehlszeile liegen im Parser im Array p vor, welches als letztes Element einer Null-Zeiger als Endemarke enthält. Die Anzahl der Argumente ist in der Variablen i gespeichert. Die eingegebene Kommandozeile befindet sich in CMD.
Der Name des eingegebenen Befehls oder Programmnamens befindet sich in p[0]. Es gibt im Prinzip nur zwei Kommandokategorien:

- Befehle, die komplexe Parameter erwarten, die nicht durch Whitespaces getrennt werden können. In diesem Fall muß CMD übergeben werden und ein spezifischer Auswertungsalgorithmus entwickelt werden (siehe Implementierung von SET).
- Kommandos, die wie externe Programme implementiert werden können. Hier entsprechen p und i den formalen Parametern argv und argc von main(). Falls das Feld env von main() bei einer externen Implementierung berücksichtigt werden würde, kann diesen durch eptr “simuliert” werden. (Beispiel: TYPE)

Datenaustausch zwischen Prozessoren – Piping

Die hohe Kunst des Pipings beruht auf der Ein/Ausgabeumlenkung. Die Ausgabe eines Prozesses A wird auf die Eingabe einer Task B transferiert. Durch den Umweg über eine temporäre Datei kann dies einfach realisiert werden.
Ein simpler Weg die Beispiel-Shell CMD um das Piping zu erweitern, wäre den Parser in eine Schleife innerhalb einer neuen Subroutine zu legen. Diese Routine könnte preparse() genannt werden. Dem Unterprogramm preparse() wird eine Kommandozeile ähnlich der folgenden übergeben: “type c:\autoexec.bat | more”.
Eine solche Kommandozeile kann einfach in einzelne Parts aufgespalten werden. Diese Teile wäre im obigen Beispiel “type c:\autoexec.bat” und “more”. Hier kann ein ähnlicher Algorithmus wie bei der Argumentenaufspaltung verwendet werden.
Die einzelnen Kommandos sind immer durch ein „|“ getrennt. Sehr einfach kommt man zu den einzelnen Teilen, wenn jeweils das „|“ durch ein Null-Byte substituiert wird. Ein Zeiger aus einem Array muß nun lediglich auf das auf „|“ folgende Zeichen zeigen.
Nun müssen nur noch die Pointer aus dem Feld nacheinander an die Routine parse() übergeben werden und Ein- und Ausgabe umgelenkt werden.
Das Unterprogramm parse() muß des weiteren einen Fehlerstatus zurückliefern, da im Falle eines Fehlers die Abarbeitung des Piping abgebrochen werden muß.

Auf den Stapel gelegt

Eine Batch-Verarbeitung besteht auf den ersten Blick nur aus der Änderung der Eingabequelle. Im normalen Dialogbetrieb ist für den Eingang der Befehlszeilen der Editor verantwortlich. Während der Stapelverarbeitung muß die Eingabe jedoch von einer Datei aus erfolgen. Hier ist also lediglich ein neues Modul zum Einlesen aus einer Datei erforderlich. Außerdem muß noch ein Flag existieren, das beim Ausführen eines Batch-Files gesetzt wird und die Eingabe auf das Batch-Modul setzt. Beim Erreichen von EOF wird dieses Flag wieder zurückgesetzt und damit die Eingabe wieder auf den Editor transferiert.
Hierauf folgt nun das eigentliche Problem: Die Verwaltung von Labels. Die grundlegende Administration wird mittels einer Tabelle realisiert. Die Datensätze bestehen aus zwei Zellen: Name des Labels und Adresse relativ zum Dateianfang der folgenden Anweisung. Hierzu ein Beispiel:

```
...
:tp
turbo.exe
...
```

Der Datensatz für die Marke “tp” würde als Adresse die Position des “t” von TURBO.EXE enthalten.
Für den Aufbau einer derartigen Tabelle gibt es zwei Strategien. Entweder man verwendet ein 2-Pass-Konzept oder ein 1-Pass-Konzept
Bei der 2-Pass-Version sind, wie der Name schon andeutet, zwei Durchläufe notwendig. Im ersten Pass werden nur die Labels betrachtet und sämtliche anderen Anweisungen ignoriert. Hier werden lediglich die Marken In die Tabelle gefüllt. Im zweiten Durchlauf werden umgekehrt die Labels Ignoriert und Befehle ausgeführt. Dieses Verfahren hat den Vorteil, daß bei jedem GOTO-Statement das übergebene Label mit der Tabelle verglichen werden und so auf Vorhandensein getestet werden kann.
Das 1-Pass-Konzept benötigt nur einen Durchgang. Während der Ausführung werden die angetroffenen Labels in die Tabelle eingetragen. Soll nun ein Sprungkommando ausgeführt werden, das der Tabelle noch nicht existiert, so wird der Rest der Datei nach dem Label durchsucht. Hierbei werden wiederum alle angetroffenen Marken ebenfalls in der Tabelle gespeichert, jedoch nun die Anweisungen bis zur gesuchten Marke ignoriert. Existiert das Label nicht, so kann dies erst bei Erreichen des Dateiendes festgestellt werden. Das wirkt sich bei großen Stapeldateien nicht gerade vorteilhaft auf die Laufzeit aus. Jedoch benötigt dieses Verfahren bei (korrekt programmierten) Batch-Files nur einen Durchlauf.
Beide Varianten haben ihre Vor- und Nachteile, demzufolge muß die Antwort auf die Frage nach dem besseren Verfahren wohl als individuell verschieden betrachtet werden.

Listing 1: cmd.c

```
/* Projekt: Kommandointerpreter
   Datei  : cmd.c
   Zweck  : Hauptmodul
   Sprache: C
   Autor  : Oliver Müller */

#include <string.h>
#include "error.h"
#include "cmd.h"

short    pFlag=0;
unsigned envSize=1024;
char     *environ, **eptr;

void init(char *a[], char *e[])
{
  unsigned n=0, i, len;
  char *curr;
  for (i=1; a[i]; i++)
    if (a[i][0] == '/')
      switch (a[i][1])
      {
        case 'P' :
        case 'p' : pFlag=1; break;
        default  : error (eUNKNOWN, a[i]);
      }
  environ = emalloc (envSize);
  eptr = (char**) emalloc (envSize * sizeof(char*));
  if (pFlag) eptr[0]=0;
  else
  {
    for (i=0; e[i]; i++)
    {
      len=strlen(e[i]);
      curr=&environ[n];
      if ((n+=len+1) > envSize)
        error(eENV);
      strcpy(curr,e[i]);
      eptr[i]=curr;
    }
    eptr[i]=0;
  }
}

#pragma argsused

int main(int argc, char *argv[], char *env[])
{
  char cmd[MAXCMD];
  init (argv, env);
  for (;;)
  {
    prompt();
    edit(cmd);
    parse(cmd);
  }
}
```

Listing 2: cmd.h

```
/* Projekt : Kommandointerpreter
   Datei   : cmd.h
   Sprache : C
   Autor   : Oliver Müller */

#ifndef _CMD_H_OGM
#define _CMD_H_OGM
#define MAXCMD 128

void prompt(void);
void edit(char *cmd);
void parse(char *cmd);
void call(char *args[]);
void set(char *cmd, unsigned argc);
void type(unsigned argc, char *args[]);
char *emalloc(unsigned size);

#endif
```

Listing 3: edit.c

```
/* Projekt : Kommandointerpreter
   Datei   : edit.c
   Sprache : C
   Autor   : Oliver Müller */

#include <stdio.h>
#include <dir.h>
#include <conio.h>
#include "cmd.h"

void prompt()
{
  char p[MAXPATH];
  getcwd (p, MAXPATH);
  printf ("%s> ", p);
}

#define curpos() \
        { \
          if (cur > 80-x0) gotoxy (cur-80+x0, y0+1); \
          else gotoxy (x0+cur, y0); \
        }

void edit(char *cmd)
{
  char c;
  unsigned n, cur=0, len=0, x0=wherex(), y0=wherey();
  for (;;)
    switch (c=getch())
    {
      case '\r' : cmd[len]='\0';
                  putchar('\n'); return;
```

```
      case '\b' : if (cur == 0) putchar('\a');
                  else
                  {
                    cur--; curpos();
                    for (n=cur; n < len-1; n++)
                    {
                      cmd[n]=cmd[n+1];
                      putchar(cmd[n]);
                    }
                    putchar(' '); len--;
                    curpos();
                  }
                  break;
     case 27    : gotoxy (x0,y0); cur=0;
                  for (n=len+1; n > 0; n--)
                    putchar(' ');
                  len=0; gotoxy(x0,y0);
                  break;
     case '\f'  :
     case '\t'  : putchar('\a');
                  break;
     case 0     : switch (getch())
                  {
                    case 75 : /* links */
                              if (cur==0) putchar('\a');
                              else cur--;
                              curpos(); break;
                    case 77 : /* rechts */
                              if (cur==len) putchar('\a');
                              else cur++;
                              curpos(); break;
                    default : putchar('\a');
                  }
                  break;
     default    : if (len >= MAXCMD-1)
                    putchar('\u');
                  else
                  {
                    cmd[cur++]=c; putchar(c);
                    if (len < cur) len=cur;
                    if (len > 80-x0 && y0==25)
                    y0=24;
                  }
    }
}
```

Listing 4: exec.c

```
/* Datei   : exec.c
   Zweck   : Ausführendes Modul
   Sprache : C
   Autor   : Oliver Müller */

#include <stdio.h>
#include <process.h>
#include <dir.h>
#include <string.h>
#include <ctype.h>
#include <mem.h>
#include "error.h"
#include "cmd.h"

extern unsigned envSize;
extern char **eptr, *environ;

#define nobat() { printf ("No batch mode!\n"); return; }

void call (char *args[])
{
  char
dr[MAXDRIVE],di[MAXDIR],fi[MAXFILE],ex[MAXEXT],pa[MAXPATH];
  fnsplit (args[0], dr, di, fi, ex);
  if (ex[0] == '\0')
  {
    strcpy (ex, ".COM");
    fnmerge (pa, dr, di, fi, ex);
    if (!searchpath(pa))
    {
      strcpy (ex, ".EXE");
      fnmerge (pa, dr, di, fi, ex);
      if (!searchpath(pa))
      {
        strcpy (ex, ".BAT");
        fnmerge (pa, dr, di, fi, ex);
        if (!searchpath(pa))
        {
          error (eNOTFOUND); return;
        }
        else nobat();
      }
    }
  }
  else if (!strcmpi(ex,".BAT")) { nobat(); }
       else strcpy(pa, args[0]);
  if (spawnvpe(P_WAIT, args[0], args, eptr) == -1) perror("");
}

unsigned del(char *name)
{
  unsigned n, l=strlen(name);
  for (n=0; eptr[n]; n++)
    if (!strncmp(name, eptr[n], l) && eptr[n][l] == '=') break;
  if (eptr[n] == 0) return n;
  for (n++; eptr[n]; n++)
  {
    memmove (eptr[n-1], eptr[n], strlen(eptr[n])+1);
    eptr[n]=eptr[n-1] + strlen(eptr[n-1])+1;
  }
  eptr[--n]=0;
  return n;
}
```

```
void add(char *s, unsigned last)
{
  unsigned size=0, n;
  for (n=0; eptr[n]; n++)
    size+=strlen(eptr[n])+1;
  s[strlen(s)] = '=';
  if (size+strlen(s)+1 > envSize)
  {
    error(eENV); return;
  }
  eptr[last] = &environ[size];
  strcpy(eptr[last], s);
  eptr[last+1] = 0;
}

void printenv(void)
{
  unsigned n;
  for (n=0; eptr[n]; n++)
    printf("%s\n",eptr[n]);
}

#pragma warn -eff

void set (char *cmd, unsigned argc)
{
  char *ptr, c[MAXCMD];
  unsigned last;
  if (argc == 1) printenv();
  else
  {
    while (isspace(*cmd))  *cmd++;
    while (!isspace(*cmd)) *cmd++;
    while (isspace(*cmd))  *cmd++;
    strcpy (c, cmd);
    if ((ptr=strstr(c, "=")) == NULL)
    {
      error(eSYN); return;
    }
    *ptr='\0'; *ptr++;
    strupr(c); last=del(c);
    if (ptr[0] != '\0') add(c, last);
  }
}

#pragma warn .eff

void type (unsigned argc, char *args[])
{
  unsigned n; int c; FILE *f;
  if (argc == 1)
  {
    error(eMISSARG); return;
  }
  for (n=1; args[n]; n++)
  {
    f=fopen(args[n], "rt");
    if (!f)
    {
      error(eOPEN, args[n]); return;
    }
    while ((c=getc(f)) != EOF) putchar(c);
    fclose(f); putchar('\n');
  }
}
```

Listing 5: parse.c

```
/* Projekt : Kommandointerpreter
   Datei   : parse.c
   Zweck   : Parser-Modul
   Sprache : C
   Autor   : Oliver Müller */

#ifdef __TURBOC__
  #include <alloc.h>
#else
  #include <malloc.h>
#endif
#include <string.h>
#include <ctype.h>
#include <dir.h>
#include <stdlib.h>
#include <fcntl.h>
#include <io.h>
#include <stdio.h>
#include <sys\stat.h>
#include "error.h"
#include "cmd.h"

#define pfree() \
        { \
          for (n=0; p[n]; n++) \
            free(p[n]); \
          return; \
        }

int out, in;
extern short pFlag;

void force (int a, int b)
{
  asm {
    mov ah,0x46
    mov bx,b
    mov cx,a
    int 0x21
  }
}

#pragma warm -eff

int redirect(char *rest)
{
  char ofile[MAXPATH], ifile[MAXPATH];
  unsigned n;
  ofile[0]='\0'; ifile[0]='\0';
  for (; *rest!='\0'; *rest++)
    if (*rest == '<')
    {
      if (ifile[0] != '\0')
      {
        error(eINRED); return 0;
      }
      for (*rest++; isspace(*rest); *rest++);
      for (n=0; !isspace(*rest) && *rest != '\0'; *rest++,n++)
        ofile[n]=*rest;
      ofile[n]='\0';
    }
```

```
    else if (*rest == '>')
    {
      if (ofile[0] != '\0')
      {
        error(eOUTRED); return 0;
      }
      for (*rest++; isspace(*rest); *rest++);
      for(n=0; !isspace(*rest) && *rest != '\0'; *rest++,n++)
        ofile[n]=*rest;
      ofile[n]='\0';
    }
  if (ofile[0])
    if ((out=open(ofile, O_CREAT|O_TRUNC|O_TEXT,
                                          S_IWRITE|S_IREAD)) ==
-1)
    {
      perror(""); return 0;
    }
 if (ifile[0])
   if ((in=open(ifile, O_RDONLY|O_TEXT)) == -1)
   {
     if (out != -1) close(out);
     perror(""); return 0;
   }
 return 1;
}

#pragma warn .eff

void parse (char *cmd)
{
  char *p[MAXCMD], *ptr;
  unsigned n=0, m, i=0;
  int odup, idup;
  in=-1; out=-1;
  for(;;)
  {
    while (isspace(cmd[n]))
    {
      if (cmd[n] == '\0') break;
      n++;
    }
    if (cmd[n] == '\0') break;
    ptr=&cmd[n]; m=n; n++;
    if (ptr[0] == '>' || ptr[0]=='<')
    {
      if (!redirect(ptr)) pfree();
      break;
    }
    else
    {
      if (ptr[0] == '"')
      {
        while (cmd[n] != '"')
        {
       if (cmd[n] == '\0')
          {
         error(eQUOTE); pfree();
          }
       n++;
        }
        n++;
      }
      else
```

```
        while (!isspace(cmd[n]))
        {
          if (cmd[n] == '\0') break;
          n++;
        }
      p[i]=(char*)malloc(n-m+1);
      if (!p[i])
      {
        error(eOUTOFMEM);
        pfree();
      }
      strncpy(p[i], ptr, n-m);
      p[i][n-m]='\0'; i++;
    }
  }
  p[i]=0;
  if (out != -1)
  {
    odup=dup(1); force(1, out);
  }
  if (in != -1)
  {
    idup=dup(0); force(0, in);
  }
  if (p[0] == 0); /* Leerzeile */
  else if (!strcmpi(p[0], "EXIT"))
       {
         if (!pFlag) exit(0);
       }
       else if (!strncmpi(p[0], "CD", 2)
                 && (p[0][2] == '\0' || p[0][2] == '.'
                                     || p[0][2] == '\\'))
            {
              if (p[0][2] == '\0')
              {
                if (i != 2) error(eARGS, "CD", 1);
                if (chdir(p[1]) == -1) perror("");
              }
              else
              {
                if (i != 1) error(eARGS, "CD", 1);
                if (chdir(&p[0][2]) == -1) perror("");
              }
            }
            else if (!strcmpi(p[0], "TYPE"))
                 {
                   type(i,p);
                 }
                 else if (!strcmpi(p[0], "SET"))
                      {
                        set(cmd,i);
                      }
                      else call(p);
  if (out != -1)
  {
    force(1, odup); close(out);
  }
  if (in != -1)
  {
    force(0, idup); close(in);
  }
  putchar('\n'); pfree();
}
```

Listing 6: error.c

```
/* Projekt : Kommandointerpreter
   Datei   : error.c
   Zweck   : Fehlerbehandlungsmodul
   Sprache : C
   Autor   : Oliver Müller */

#ifdef __TURBOC__
  #include <alloc.h>
#else
  #include <malloc.h>
#endif
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include "error.h"

static char *msg[] =
{
  "Environment full !\n",
  "Missing quote!\n",
  "%s expects %d parameter(s)!\n",
  "Input already redirected!\n",
  "Output already redirected!\n",
  "File or path not found!\n",
  "Unknown option %s!\n",
  "Syntax error!\n",
  "Out of memory!\n",
  "Missing arguments!\n",
  "Can't open %s!\n"
};

void error(int no,...)
{
  va_list ap;
  va_start (ap, no);
  vfprintf (stderr, msg[no], ap);
  va_end (ap);
}

char *emalloc(unsigned size)
{
  char *ptr=(char*)malloc(size);
  if (!ptr)
  {
    fprintf (stderr, "Out of memory!\n");
    exit(1);
  }
  return ptr;
}
```

Listing 7: error.h

```
/* Projekt : Kommandointerpreter
   Datei   : error.h
   Zweck   : Spezifikation Fehlerbehandlung
   Sprache : C
   Autor   : Oliver Müller */

#ifndef __ERROR_H_OGM
#define __ERROR_H_OGM

#define eENV 0
#define eQUOTE 1
#define eARGS 2
#define eINRED 3
#define eOUTRED 4
#define eNOTFOUND 5
#define eUNKNOWN 6
#define eSYN 7
#define eOUTOFMEM 8
#define eMISSARG 9
#define eOPEN 10

void error(int no,...);
#endif
```

Listing 8: makefile

```
# Kommandointerpreter für mc extra
# Autor: Oliver Mueller

CC=bcc
LINK=tlink
CFLAGS=-c -v -ms -O -O2 -Ie:\bcc31\include
LFLAGS=/x /v /Le:\bcc31\lib c0s
LIBS=cs.lib
OBJS=cmd.obj error.obj edit.obj parse.obj exec.obj

.c.obj:
     $(CC) $(CFLAGS) $*.c

cmd.exe: $(OBJS)
     $(LINK) $(LFLAGS) $(OBJS), $<,,$(LIBS)

cmd.obj : cmd.c cmd.h error.h

error.obj: error.c error.h

parse.obj: parse.c error.h cmd.h

exec.obj: exec.c error.h cmd.h

edit.obj: edit.c cmd.h

clean:
     del *.obj
```