DOKUMENTATION
IMPORT SERVER

Stand: PROXESS 10

Inhaltsübersicht

Inhaltsübersicht 2
Aufgaben und Datenfluss 3
Aufgaben 3
Colddaten 3
Datenfluss 3
Import Server Setup 5
Programmdateien und Registry 7
Programmdateien 7
Registryeinträge 7
Ablauflogik eines Cold Scripts 8
Prinzip der Sektionen 8
Parametersektion 9
Dokumentsektion/-en 9
Erkennungssektion 9
Verarbeitungssektion 9
‚END' - Sektion 10
Befehle und Zeichensätze 11
Zeichensätze allgemein 11
Befehle und Funktionen der Parametersektion 12
Befehle und Funktionen der Erkennungssektion 16
Muster 19
Befehle und Funktionen der Verarbeitungssektion 21
Funktionen 21
Kontrollstrukturen 25
Arithmetik 26
Datenübergabe 27
Betriebsarten des Coldconverters 30
Reservierte Befehlswörter 32
Bekannte Fehler und Meldungen 33
Bekannte Bugs und Probleme: 33
Fehlermeldungen 34
Anhänge mit Beispielen 36
Anhang A Beispiel eines Cold Scripts 36
Anhang B Sammeln mehrerer Dokumente 39
Anhang C Formale Grammatik von Cold Script 40
Anhang D schrittweise Entwicklung eines Scripts 43
Anhang E Mehrere Dateien in ein Dokument einfügen 49
Anhang F HP - Sequenzen des TVG 52

Aufgaben und Datenfluss

Aufgaben

Der PROXESS Import Server (Import Server) liest eine Textdatei (Spoolfile o. ä.) aus und übersetzt diese in reinen ANSI-Datensatz. Hierbei werden u. a. Zeichensätze konvertiert und „Escape Sequenzen“ des Druckers gefiltert.

Im zweiten Schritt erkennt er Dokumenttypen und zugehörige Indexfelder, wie z. B. einen Kundennamen (=Indexfeld) in einer Ausgangsrechnung (=Dokumenttyp).
Um diese Items erkennen zu können, muss ein Dokumenttyp in einer formalen Sprache, die einer “kleinen” Programmiersprache sehr ähnlich ist, beschrieben werden. Diese Steuerungsdatei wird Coldscript genannt. In diesem wird angegeben:

- wie der Dokumenttyp erkannt wird (z.B.: in Zeile 2 muss ‘RECHNUNG’ stehen).
- was mit den bei Erkennung gesammelten Informationen (Indexdaten o. ä.) geschieht.

Colddaten

Als zu verarbeitende Colddaten kommen prinzipiell alle Textdateien in Frage. Es können entweder die an den Drucker gesendeten Druckjobs verwendet werden („Spoolfiles“) oder die Colddatensätze werden extra zur Archivierung erzeugt. Ein Colddatensatz darf dabei:

- keine NULL-Bytes enthalten
- keine Tabulatoren enthalten
- nur die Schriftart (Courier 10 CPI) verwenden.

Achtung: Druckersteuersequenzen können gefiltert, aber nicht, beziehungsweise nur bedingt, interpretiert werden!

Dies bedeutet, dass der formale Aufbau von PROXESS-Colddaten, also das optische Erscheinungsbild im Spoolfile, dem des ausgedruckten Dokumentes entsprechen sollte.

Prinzipiell gilt folgende Faustregel: Können Sie ein Dokument mit dem Windowstexteditor ‚Notepad’ öffnen und ohne Formatprobleme sehen, dann haben sie schon gewonnen!’

Datenfluss

Der Import Server ist ein sequentiell arbeitendes Programm, d. h. eine Spooldatei wird nach folgendem Muster bearbeitet:

- Beim Start wird das Script in den Cache geladen.
- Das Coldfile wird in einem Arbeitsverzeichnis lokalisiert und gemäß der Konvertierungsbefehle des Coldscripts in Windows ANSI-Zeichensatz gewandelt.
- Die konvertierte Datei wird nach Kriterien der Dokumentzugehörigkeit und nach den Indexinformationen durchsucht. Dies geschieht zeichenweise und sequenziell.
- Nach der Erkennung werden die Indexinformationen formatiert und an PROXESS zur Archivierung übergeben.
- Der Kreislauf beginnt mit einer neuen Datei.

Der Basisalgorithmus für ColdConv:

Führe Parametersektion aus (einmal).

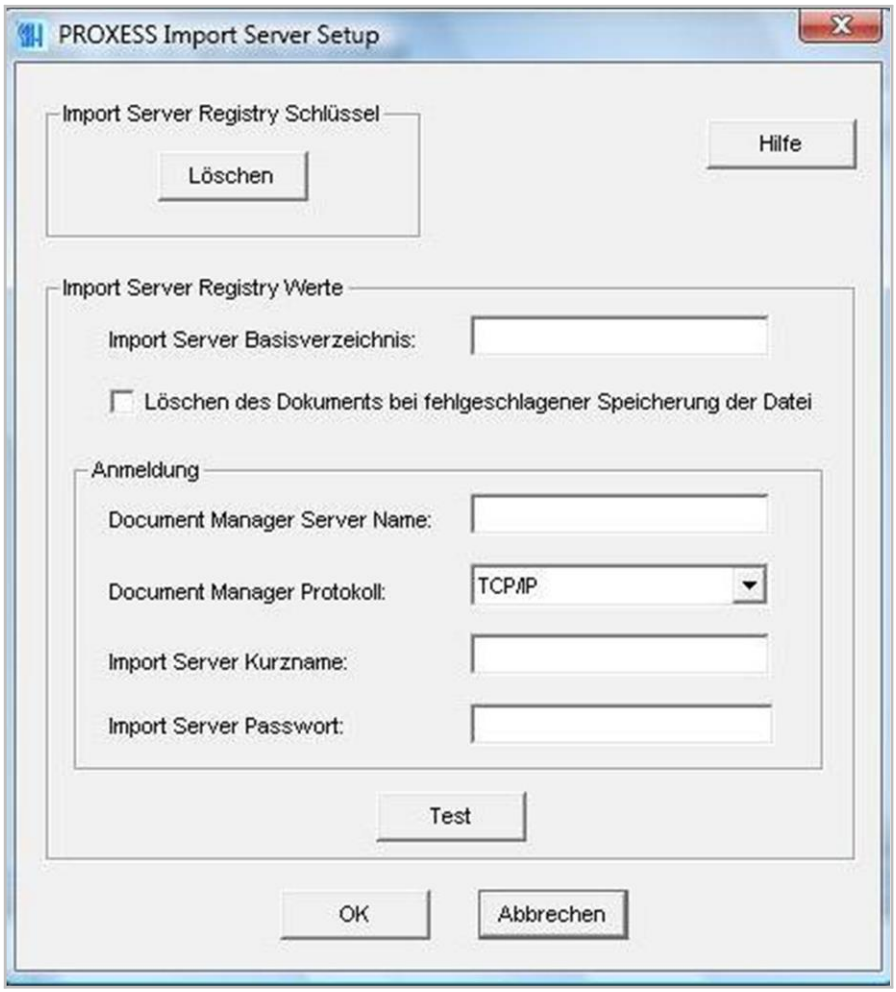
Solange noch Dokumente in der Eingabedatei sind:
Anfang der Schleife: {
Führe, in der Reihenfolge ihres Auftretens im Cold Script, die Erkennungssektionen aus.

Hat eine Erkennungssektion ihren Dokumenttyp erkannt (d.h. alle MUST wahr) führe die Aktionen-Sektion für diesen Dokumenttyp aus. Hole nächstes Dokument
aus der Eingabedatei. Hat keine der Erkennungssektionen das Dokument erkannt, so wird
es ignoriert.
} Ende der Schleife

Gibt es eine END-Sektion, so führe sie aus (einmal).

Import Server Setup

Nach der Installation von PROXESS können Sie das Import Server Setup aus der PROXESS-Programmgruppe starten. Im Import Server Setup werden alle wichtigen Basiseinstellungen für den Betrieb des PROXESS Import Servers gemacht.



Zunächst müssen Sie mit dem Befehl **Aufbauen** die Cold Converter Registry-Schlüssel auf dem lokalen Rechner aufbauen.

Die einzutragenden Registrywerte lauten:

Import Server Basisverzeichnis	Arbeitsverzeichnis in dem die Colddaten zur Verarbeitung abgelegt werden. Dieses Verzeichnis muss existieren. (z. B. C:/ coldbasis)
Löschen des Dokuments bei fehlgeschlagener Speicherung der Datei	Angabe zu Fehlerbehandlung

Anmeldung

Document Manager Server Name	Netzwerkname des Rechners bzw. IP-Adresse auf dem das PROXESS-System installiert ist. Bei verteilten Systemen wird hier Rechnername des Document Managers angegeben.
Document Manager Protokoll	Netzwerk Protokoll mit dem dieser Rechner erreichbar ist. (z. B. TCP/ IP)
Import Server Kurzname	Geben Sie den Benutzernamen an, mit dem der PROXESS Import Server sich am PROXESS System anmeldet um Daten zu importieren.Sinnvollerweise wählen Sie hier einen bezeichnenden Benutzernamen wie z. B. Import Server, mit dem Sie später Systemmeldungen entsprechend auch zuordnen können. Der Benutzername muss in der PROXESS-Benutzerverwaltung zuvor angelegt worden sein.
Import Server Passwort	Mit diesem Passwort meldet sich der PROXESS Import Server am System an. Das Passwort muss mit dem Passwort der Benutzerverwaltung übereinstimmen.

Ihre Angaben zur Anmeldung werden durch den Testbutton überprüft.

Erst mit dem Befehl **OK** werden die gemachten Angaben auch in den lokalen Registryschlüsseln gespeichert.

Programmdateien und Registry

Programmdateien

Durch die Installation von PROXESS professional werden folgende Dateien in das PROXESS-Verzeichnis des Servers kopiert:

- Csetup.cnt
- cSetup.exe
- csetup.GID
- Csetup.hlp
- Convcold.exe

Registryeinträge

Der Basispfad der Coldconvertereinträge ist:

HKEY_LOCAL_MACHINE\
SOFTWARE\
SHD\
Document!\
ColdConv

Folgende Variablen müssen angelegt werden (alle vom Typ REG_SZ):

ColdHomePath : Basisverzeichniss für Serverbetrieb (siehe Anhang B)

Ohne die folgenden Variablen und deren gültige Inhalte werden Cold-Scripts, die BINDOUTFIELDS und/oder WRITEKERNEL enthalten, nicht starten:

HostName : Name des Rechners, auf dem der Proxess-Kernel läuft.
ProtSeq : (RPC-) Protokoll zu Kernel (z.B.: ncacn_nb_ipx)
ShortName : Benutzername, unter dem ColdConv sich beim Kernel anmeldet
PassWord : Benutzerpassword

Ab Version 2.0 ist der Basispfad um den Pfadeintrag ‚COMMAND' erweitert. Dieser enthält den DWORD-Eintrag:

DocDelIfFault : Löschen fehlerhaft angelegter Dokumente, im Fall, dass das Backend gestört ist.

Ablauflogik eines Cold Scripts

Prinzip der Sektionen

Ein Cold Script besteht aus mehreren **Sektionen**. Diese enthalten die Steuerinformationen, nach denen der Import Server die Coldfiles verarbeitet. Die einzelnen Sektionen werden sequentiell durchlaufen.

Jedes Script besteht aus mindestens 2 Sektionen, einer Parametersektion und mindestens einer Dokumenttypsektion. Es können jedoch beliebig viele Dokumenttypsektionen aufeinanderfolgen. Diese werden sequentiell nach dem ‚fall-trough'-Prinzip abgearbeitet, bis eine passende Sektion gefunden wird. Optional kann eine spezielle END-Sektion folgen. Diese wird nur am Ende eines Coldfiles angesprungen.

Die Sektionen werden mit {….} begrenzt.

```
{
        /* Parameter Sektion */
        ENDOFDOCPATTERN = '\x0c';
        ENDOFLINE = '\x0d\x0a';
        {
                variable = 0;
        }
        ……
}

'Dokumentypname#1':
  {
        /* Dokumenttyp#1 */
  }
'Dokumentypname#2':
  {
        /* Dokumenttyp#2 */
  }

……
'Dokumentypname#n':
  {
        /* Dokumenttyp#n */
  }

END
  {
        /* Befehle, die am Ende des Scripts unabhängig von der Erkennung
           der Dokumenttypen ausgeführt werden. (diese Sektion ist optional) */
   }
```

Eine solche, einem **Dokumenttyp** zugeordnete Sektion, besteht wiederum aus zwei Sektionen:

- Erkennungssektion
- Aktionensektion.

Beide werden wiederum sequentiell durchlaufen und haben klar voneinander unterschiedene Funktionen.

```
'Mitglied':
{
        {
        /* Erkennungs-Sektion für Doc Typ 'Mitglied' */
        }

        {
        /* Aktionen-Sektion für Doc Typ 'Mitglied' */
        }
}
```

Parametersektion

Diese Sektionen wird am Anfang der Verarbeitung einer Colddatei aufgerufen. In ihr werden die zu verwendenden Zeichensätze definiert (siehe Kap. Zeichensätze), es wird festgelegt an Hand welcher Begrenzungen Zeilen und Dokumente erkannt werden, welche Steuerzeichen innerhalb der Colddatei ignoriert oder gewandelt werden sollen und es können Initialisierungsbefehle für die zu verwendenden Variablen ausgeführt werden.

Dokumentsektion/-en

Die Dokumentsektionen sind die eigentlichen ‚Arbeitssektionen'. In Ihnen erfolgt die dokumenttyp-bezogene Verarbeitung der Colddatei. Das bedeutet, dass für jeden existierenden Dokumenttyp eine Dokumentsektion vorhanden sein muss, auf dessen spezielle Attribute geachtet und reagiert wird.

Erkennungssektion

In der Erkennung wird zunächst geprüft, ob das vorhandene Colddokument in diese Dokumentsektion passt. Ist dies nicht der Fall, so „fällt das Dokument in die nächste Dokumenttypsektion durch", bis es passt oder das Script zu Ende ist.
Passt das Erkennungsmuster jedoch, so wird anschließend die Indexierung aus dem Textfluss ausgelesen.

Verarbeitungssektion

Wurde ein Dokument erkennungstechnisch erfolgreich bearbeitet, so gelangt es in die Verarbeitungssektion der entsprechenden Dokumentsektion. Hier werden nun Formatierungen, Bildschirmausgaben, Protokollierungen, Fehlerbehandlungen und nicht zuletzt die Archivierung des Dokumentes gesteuert.

‚END' - Sektion

Die ‚END'-Sektion wird wie die Parametersektion nur einmal durchlaufen. Wie der Name jedoch schon sagt nur am Ende einer Colddatei. Sie kann alle Befehle der Aktionen-Sektion enthalten.

```
/* …….restliches Cold-Script */
END {
        PRINT('Die Summe aller Posten war: ', summen_variable);
    }

/* Ende des Coldscripts */
```

Befehle und Zeichensätze

Zeichensätze allgemein

Die Beschreibungssprache (Cold Script) verfügt über das Konzept von “Variablen” wie in den meisten Programmiersprachen.
z.B.

```
my_variable='ich bin eine Zeichenkette';
PRINT('THE VALUE IS: ', my_variable);
```

Ein Variablenname muss mit einem Buchstaben oder Unterstrich beginnen und kann dann beliebig viele Buchstaben, Zahlen oder Unterstriche enthalten. Besonders zu beachten ist, dass er **nicht** mit einer **Zahl** beginnen darf.
Variablennamen dürfen keine reservierten Worte sein (siehe Liste im Anhang A).

An diesem Beispiel werden weitere grundlegende Regeln der Beschreibungssprache sichtbar:

a) Zeichenketten (Strings) werden in ‘ (einzelne Hochkommata, single quotes) eingeschlossen.

b) Soll ein solches einzelnes Hochkomma als Zeichen in eine Zeichenkette, so muss es verdoppelt werden.
z.B.: ‘Otto’’s Imbißbude’ (beachte: zwischen o und s stehen **zwei** einzelne Hochkommata.)
PRINT(‘Otto’’s Imbißbude’); liefert: *Otto’s Imbißbude*

c) Alle Statements (Befehle) werden mit ; (Semikolon) abgeschlossen. Als Lohn für diese Mühe ist die Beschreibungssprache formatfrei, d.h. außerhalb von Zeichenketten und Befehlsnamen sind Leerzeichen und Zeilenschaltungen ohne Bedeutung. Man kann (und sollte) das COLD-Script also lesbar formatieren:
z.B.:

```
PRINT(‘hallo world’, myvar, yourvar);
```

d) Alle Variablen haben den Datentyp “String”. Sie brauchen nicht deklariert zu werden; sie “entstehen”, wenn sie das erste Mal benutzt werden.

e) Zeichenketten können auch Hexadezimale Werte enthalten.
z.B.
‘dies ist eine Zeilenschaltung\x0d\x0a’
An Ende dieser Zeichenkette steht also ein ANSI CR (*x0d)* und ein ANSI LF (*x0a*).

f) Das Zeichen mit dem Wert Hex 00 ist in der Cold Datei **nicht** erlaubt. Kommt es in der Colddatei vor, so wird es vor der Zeichensatzübersetzung (d.h. als aller erstes) durch ein Hex 01 **ersetzt**.

g) Es gibt vorbelegte Variablen:
doctyp enthält den momentanen Dokumenttyp.

h) Kommentare werden in /*….*/ (analog ‘C’) eingeschlossen.
z.B.:

```
/* Dies ist ein Kommentar und wird ignoriert */
```

Kommentare können nicht geschachtelt werden.

```
/* blalalal  /* here am I */ mumpf grumpf */
```

Ist nicht zulässig!

i) Die Schlüsselworte („Befehle“) wie PRINT,IF,WHILE… müssen GROß geschrieben werden.

j) Alle Variablen haben zunächst den Wert EMPTY. Dieser Wert kann auch abgefragt werden: IF(b = EMPTY) { … } .
Sollen Variablen zur numerischen Iteration benutzt werden also z.B. *variable = variable + 1*; so muss man *variable* in der Parameter Sektion initialisieren. Sonst wird die Variable immer den Wert EMPTY
oder, bei der Ausgabe (null) behalten.

k) Es gibt zwei verschiedene Typen von Konstanten:
Ganzzahlen (int_const)
z.B.: x = 4711;
IGNORELINE(12);/* 4711 und 12 sind Ganzzahlen */
ein_fehler = 3,14; /* keine nachkomma oder „nachpunkt“ (3.14) Stellen!! */

Zeichenketten (string_const oder pattern)
z.B.: meine_kette = ‘the quick brown fox jumps over the syntax error 1234567890’;

l) Das Coldscript wird in eine interne Darstellung übersetzt, bevor es ausgeführt wird. D.h. Kommentare und lange Variablennamen kosten keine Strafe in der Ausführungszeit wie bei anderen interpretierten Programmiersprachen.

Befehle und Funktionen der Parametersektion

In der Parametersektion werden zunächst alle wichtigen Konvertierungen und die Dokument-abgrenzungen definiert und festgelegt. Hierfür gibt es ‚Muss’ - und ‚Kann’ - Bedingungen.

Es gibt genau zwei MUSS - Variablen, die in jedem Script verwendet werden müssen! Diese sind:
- ENDOFDOCPATTERN
- ENDOFLINE

Mit Ihrer Hilfe werden die einzelnen Zeilen einer Colddatei und die einzelnen Dokumente / Dokumentseiten innerhalb einer Colddatei unterschieden. Beides sind PFLICHTFELDER!!!!!

Hierzu gilt:

ENDOFDOCPATTERN =’muster’;
gibt an, welches Muster der Import Server finden muss, um im Datenstrom des Cold Files einzelne Dokumente (eine Rechnung; ein Lieferschein…) abgrenzen zu können.

z.B.: ENDOFDOCPATTERN =’\x0c’; Dokument durch FormFeed begrenzt (Hex 0c)
oder: ENDOFDOCPATTERN =’\x0d\x0a\.’ Am Ende eines Dokuments steht immer ein Punkt (\.) am Zeilenanfang (d.h. nach CR und LF)

ENDOFDOCPATTERN = Ganzzahl;
gibt an, dass ein Dokument immer nach Ganzzahl Zeilen zu Ende ist. Hierbei muss ENDOFLINE (s.u.) immer mit angegeben werden.

z.B.: ENDOFDOCPATTERN = 80; Dokument immer 80 Zeilen lang
VORSICHT: ENDOFDOCPATTERN =’80’; Dokument wird durch eine 8 und eine 0 begrenzt!
(legal aber Unsinn)

Das erste Zeichen in ENDOFDOCPATTERN wird nicht als Zeichen eines Regulären Ausdrucks aufgefasst. D.h. gibt man z.B. an:

ENDOFDOCPATTERN ='.';

so meint man einen Punkt und nicht, wie bei den sonstigen regulären Ausdrücken, ein beliebiges Zeichen.

Aber!

ENDOFDOCPATTERN ='..'

Dies meint: ein Punkt gefolgt von einem beliebigen Zeichen! Sorry für dieses komplizierte Vorgehen, ist aber im „einfachen Fall" (z.B. ENDOFDOCPATTERN ='\x0c'; /* FormFeed */) wesentlich schneller.
Das Dateiende (EOF) wird immer als zusätzliches ENDOFDOCPATTERN aufgefasst. D.h. alles zwischen dem letzen ENDOFDOCPATTERN und dem Dateiende gilt als Dokument.

ENDOFLINE='muster';
gibt ein Muster an, das eine Zeile begrenzt.
Beispiel:

ENDOFLINE='\x0d \x0a';
(ANSI CR (*x0d)* und ein ANSI LF (*x0a*)).

Für die beiden Variablen (ENDOFLINE und ENDOFDOCPATTERN) gibt es keinen Default! Sie müssen <u>immer</u> angegeben werden!

Neben diesen beiden entscheidenden Variablen können hier aber auch Standartkonvertierungen vorgenommen werden. Dies kann mit mehreren Methoden geschehen.

A) Zeichensätze

Die Steuerung der Zeichensätze wird in der Parametersektion beschrieben und wird damit unabhängig von der Erkennung und Aufbereitung der Dokumente am Anfang der Ausführung des COLD-Scripts festgelegt.
Der Import Server "versteht" zwei verschiedene Zeichensätze:

ASCII7:	Die Umlaute aus der 7-bit ASCI-Darstellung werden übersetzt ({}… nach äü…)
EBCDIC:	Übersetzung des AS/400 (german) EBCDIC Zeichensatzes nach ANSI.

Die zeichenweise Übersetzung wird in der globalen Sektion eingestellt. Z.B.:

```
/* Anfang der Cold Script Datei */
{
        ...
        /* Parameter Sektion */
        INPUTCONVERSION = EBCDIC;
}
```

B) Zeichenkonvertierung

Sollen außerdem noch einzelne Zeichen eine gesonderte Übersetzung erfahren, so geht dies mit dem Befehl CHARMAP. Wie der Name schon sagt werden einzelne CHARAKTER in andere konvertiert (Syntax siehe im Beispiel).

```
/* Anfang der Cold Script Datei */
{
        ...
        /* Parameter Sektion */
        CHARMAP
        {
                'a' | 'A'
                '\x24' | '\xa3'
        }
}
```

In diesem Beispiel werden alle kleinen a durch große A ersetzt und die Dollarzeichen ($, Hex Code 24) durch Pound-Zeichen (£ , Hex Code A3).

Die allgemeine Syntax für den Befehl gilt wie folgt:

```
        CHARMAP
        {
                'zu übersetzendes Zeichen'  |  'Zeichen, in das übersetzt wird'
                ….. davon beliebig viele …..
        }
```

Wichtig ist, dass nur das **erste** Zeichen ausgewertet wird. D.h., dass das folgende Beispiel **nur** festlegt, dass O in F übersetzt wird.

```
        CHARMAP
        {
                'Otto' | 'Franz'
        }
```

C) Stringkonvertierungen

Sollen ganze Zeichenketten übersetzt werden, muss der Befehl STRINGMAP verwendet werden. Dieser ist jedoch langsamer als der Charmap! Die Syntax ist analog zu Charmap.

```
        STRINGMAP
        {
                'zu übersetzende Zeichenfolge'  |  'Zeichenfolge, in die übersetzt wird'
                ….. davon beliebig viele …..
        }
```

!!!ACHTUNG!!!
Charmap wird vor Stringmap ausgeführt! Das heißt, dass Strings die ursprünglich ein in Charmap konvertiertes Zeichen enthalten nicht mehr mit dem alten Zeichen existieren und somit con Stringmap auch nicht mehr gefunden werden!

D) Externe DLL

Sollten die Möglichkeiten, die der Import Server zur Anpassung des Coldtexts bietet nicht ausreichen, so kann eine DLL eingebunden werden, die diese Anpassungen vornimmt. Die Benutzung dieser DLL wird ebenfalls in der **Parameter Sektion** angemeldet:

```
{
        ...
        /* Parameter Sektion */
        USEDLL = 'meine.dll';
}
… restliches ColdScript
```

Diese DLL muss die folgende Funktion exportieren:

long ProcessFile (const char *const pInputFileName, const char *const pOutputFileName)

Der Import Server ruft ProcessFile auf und übergibt die beiden Dateinamen. ProcessFile muss pInputFileName öffnen, lesen, die gewünschten Anpassungen vornehmen und schließlich diese in das zu öffnende pOutputFileName zurückschrieben.

Der Import Server arbeitet dann mit pOutputFileName weiter. POutputFileName wird aus pInputFileName von coldconv durch Anhängen der Extension ".tmp" gebildet.

Neben den Methoden zur Datenkonvertierung können in der Parametersektion auch, wie bereits erwähnt, die zu verwendenden Variablen deklariert werden. Dies geschieht mit Hilfe der Initsubsektion.

Sie muss in einem weiteren Paar geschweifter Klammern ({}) stehen! Damit wird ‚erzwungen', dass alle Initialisierungen an einem Platz zusammen gehalten werden.
Variablen, deren Inhalt man als Zahl interpretiert sehen will, müssen, wenn sie in Iterationen (wie *i = i + 1;)* vorkommen sollen z.B. mit *variable = 0*; initialisiert werden. Der Import Server initialisiert Variablen mit EMTPY (leer). Aber EMPTY plus irgendwas ergibt wieder EMPTY!

Die allgemeine Syntax für den Befehl gilt wie folgt:

```
{
        /* Parameter Sektion */
        ENDOFDOCPATTERN = '\x0c';          /* neues Dokument nach FormFeed (Hex 0c) */
        /* oder:   ENDOFDOCPATTERN = 80;          alle 80 Zeilen ein neues Dokument */
        ENDOFLINE = '\x0d\x0a';            /* Zeilenende durch CR (0d) und LF (0a) */

        /* Init-Subsektion */
        {
                myvariable='0';
                anothervariable = 'hier ist der Anfang';
        }
}
…. /* Rest vom Cold Script (s.u.) */
```

Befehle und Funktionen der Erkennungssektion

Im Gegensatz zu einer normalen Programmiersprache, wo einzig der Kontrollfluss (sequentielle Abarbeitung der Befehle und Veränderung dieser Sequenzen durch IF, WHILE…) eine Abfolge bestimmt, kommt hier noch der Begriff des **Textzeigers** hinzu. Zu Beginn der Abarbeitung eines Coldscripts steht der Textzeiger auf dem Dokumentanfang. Es gibt Befehle (CANMATCH, MUSTMATCH…), die die Position dieses Zeiger im Dokumenttext verändern und dabei "im Vorbeigehen" Felddaten einsammeln und festlegen, ob der beschriebene Dokumenttyp wirklich vorliegt (MUSTMATCH). Konkrete Beispiele im Folgenden werde dies näher erklären.

!!!Hinweis!!!
Zu 'pattern' siehe Kapitel Muster (folgt im Anschluß).
Im einfachsten Fall gibt 'pattern' eine Zeichenkette an, nach der gesucht wird.

Folgende Befehle liefern einen String zurück und/oder verändern den Textzeiger:

MUSTMATCH(' pattern ');

z.B. my_var = MUSTMATCH('otto');
PRINT(MUSTMATCH('otto'));

MUSTMATCH('otto'); sucht in der **momentanen Zeile** des Dokuments nach dem ersten Auftreten des Strings 'otto'. Wird er gefunden, so steht der Textzeiger **hinter** 'otto'. Die Variable my_var enthält dann die Zeichenkette 'otto'.
Wird 'otto' nicht gefunden, so bleibt der Textzeiger unverändert, aber es wird angenommen, das dies nicht die geeignete Beschreibung für den Dokumenttyp des momentanen Dokuments ist. Z.B. erkennt man eine Rechnung **immer** an dem Text 'R e c h n u n g', so kann man mit:

My_var = MUSTMATCH('R e c h n u n g');

erreichen, dass diese Beschreibung für den Dokumenttyp "Rechnung' benutzt wird.
MUSTMATCH dient also:
zum Finden einer Zeichenkette, die für den Dokumenttyp bestimmend ist innerhalb einer Zeile.

MUSTMATCHMULTI (' pattern ')

Dieser Befehl funktioniert wie MUSTMATCH, jedoch arbeitet er über mehrere Zeilen u.U. bis zum Dokumentende! Er dient also:
zum Finden einer Zeichenkette, die für den Dokumenttyp bestimmend ist innerhalb mehrer Zeilen.

Analog zu den ‚Muss'-Treffern gibt es ‚Kann'-Treffer. Die Syntax ist analog zu der oben beschriebenen.

CANMATCH (' pattern ')

Wie MUSTMATCH, nur hat keinen Einfluss darauf, ob die momentane Beschreibung als für den vorliegenden Dokumenttext akzeptiert wird.

CANMATCHMULTI (' pattern ')

Wie CANMATCH, nur arbeitet über mehrere Zeilen u.U. bis zum Dokumentende!

GETCHARS (int_const)

z.b.: my_var = GETCHARS(25);

Liest so viele Zeichen, wie in int_const angegeben, ab der Position Textzeiger, in my_var ein. Der Textzeiger wird hinter den gelesenen String gesetzt. **GETCHARS überliest auch das Zeilenende**!
Wenn das verbleibende Stück des Dokuments hinter dem Textzeiger kürzer ist, als die in int_const verlangte Anzahl Zeichen, so werden nur die Zeichen zwischen Textzeiger und dem Dokument-Ende zurückgegeben.

GETLINE (int_const)

z.b.: my_var = GETLINE(256);

Liest so viele Zeichen, wie in int_const angegeben, ab der aktuellen Position des Textzeigers in my_var ein. Der Textzeiger wird hinter den gelesenen String gesetzt. **Diese Funktion überliest nicht, wie GETCHARS das Zeilenende!** D.h. Mit GETLINE(maximale_zeilenlänge) kann man eine Zeile einlesen. Mit GETLINE(32) liest man 32 Zeichen ein, wenn die Zeile ab dem Textzeiger noch 32 Zeichen enthielt. Enthielt sie weniger, so liest man auch entsprechend weniger Zeichen ein. GETLINE setzt den Textzeiger **nicht** in eine neue Zeile. Hierzu ist IGNORELINE(1) notwendig.
Wenn das verbleibende Stück des Dokuments hinter dem Textzeiger kürzer ist, als die in int_const verlangte Anzahl Zeichen und keine Zeilenschaltung mehr enthält, so werden nur die Zeichen zwischen Textzeiger und dem Dokument-Ende zurückgegeben.

GOTO(Zeile, Spalte)

Plaziert den Textzeiger innerhalb des momentanen Dokuments auf Zeile, Spalte.
z.B.: GOTO(2,45); /* Textzeiger in die 2. Zeile, 45. Zeichen */

MOVETO(ZeilenInkrement, SpaltenInkrement)

Plaziert den Textzeiger innerhalb des momentanen Dokuments relativ. D.h. MOVETO(2,1); geht relativ zur momentanen Position 2 Zeilen und eine Spalte weiter.

IGNORELINE (int_const)

Plaziert den Textzeiger an den Anfang der nächsten Zeile (int_const=1). (CANMATCH und MUSTMATCH tun dies nicht!). Ist int_const > 1 so werden entsprechend Zeilen übergangen.
z.B.: IGNORELINE(4); /* 4 Zeilen weiter */

IGNOREWHITE ( )

Steht der Textzeiger vor einer Folge von Blanks und/oder Tab’s, so wird er dahinter plaziert.

IGNORECHARS (int_const)

Der Text Zeiger wird int_const Zeichen weiter gesetzt.
z.B.: IGNORECHARS(12); /* Gehe 12 Zeichen weiter; ignoriere sie */

Muster

Alle Such- (…MATCH…)- Befehle benötigen in ihren Argumenten ein pattern (Muster). Im einfachsten Fall ist dies eine normale Zeichenkette, nach der gesucht werden soll. So z.B.:

CANMATCH ('Rechnung');

Dies sucht in der momentanen Zeile nach der Zeichenkette *Rechnung*. Nun könnte es aber vorkommen, dass *Rechnung* sowohl klein als auch groß geschrieben werden kann. Also muss der Suchbegriff erweitert werden:

CANMATCH('[rR]echnung');

Dies passt sowohl auf *rechnung* als auch auf *Rechnung*.

Wollen wir eine (ganze) Zahl finden, können wir wie folgt suchen:

CANMATCH('[0-9]');

Dies findet alle Zeichen, die zwischen den Zeichen *0* und dem Zeichen *9* liegen. Es findet aber nur eine einzige Ziffer unserer Zahl also erweitern wir das Statement:

CANMATCH('[0-9] [0-9] [0-9]');

Dieses Pattern findet schon drei Ziffern. Aber auch nur genau drei Ziffern. Wollen wir eine variable Anzahl von Ziffern finden, so können wir dies mit:

CANMATCH('[0-9]+');

Das Plus-Zeichen sagt: finde beliebig viele, aber mindesten eine Ziffer. D.h. in:

Dies ist die Endsumme: 1234 DM

finden wir: *1234*

Tritt keine Ziffer in unserem Suchtext ‚D*ies ist nur Text'*, so finden wir auch nichts. In dem folgenden String:

Dies ist die Endsumme: 4711,33 DM

finden wir: *4711*

Das letzte Beispiel ist unbefriedigend; das Dezimalkomma und die Nachkommastellen werden ignoriert. Folglich muss das Statement erneut erweitert werden.

CANMATCH('[0-9]+,[0-9]+');

Dies findet beliebig viele, aber mindestens eine Ziffer, dann ein Komma und wieder beliebig viele, aber mindesten eine Ziffer.

So können verschiedene Muster gesucht werden.

CANMATCH('[0-9] [0-9]\. [0-9] [0-9]\.19[0-9] [0-9]')

findet ein Datum mit zweistelligem Monat und Tag, einem Jahr das immer mit 19.. beginnt und wo die Angaben durch einen Punkt getrennt sind. Etwa:

14.05.1952 oder *01.01.1995* aber **nicht** *1.1.96*

Folgenden Operationen stehen in Mustern zur Verfügung:

^ Findet nur Zeichenketten, die ab der momentanen Position im Coldtext passen. z.B.: CANMATCH('^Otto') findet unseren Otto nur, wenn der Zeiger im Coldtext unmittelbar vor ihm steht. CANMATCH('Otto') würde Otto überall, ab der momentan Zeigerposition in der momentanen Zeile finden. (Anm.: Funktioniert nicht in ENDOFDOCPATTERN, hier bitte Zeilentrennzeichen explizit angeben; z.B.: '\0x0d\0x0a')

$ …MATCHMULTI('Otto$') findet die Zeichenkette Otto am Ende des Dokuments. …MATCH('Otto$') findet die Zeichenkette Otto am Ende der momentanen Zeile.

. Punkt. Passt auf jedes beliebige Zeichen. CANMATCH('.tto') findet *Otto* und *Ltto* aber nicht *Lotto* (sondern nur das *otto* darin; *L* wird überlesen*)*.

- (Minus Zeichen) Passt auf eine Zeichen in einem Bereich.
CANMATCH('[0-9]') findet ein Zeichen, das eine Ziffer (von 0 bis 9) ist.
CANMATCH('[A-z]') findet ein Zeichen, das ein Buchstabe ist. (ohne Umlaute)
CANMATCH('[A-zäöüÄÖÜß]') findet ein Zeichen, das ein Buchstabe ist. (mit Umlauten)

[ ] Passt auf eines der Zeichen in den eckigen Klammern.
CANMATCH('[0-9ab]') findet ein Zeichen, das entweder eine *Ziffer* (von 0 bis 9) oder ein *a* oder ein *b* ist. CANMATCH('[a b]') findet ein *a* , ein *Leerzeichen* oder ein *b*.

[^] Passt auf alle Zeichen außer denen in den eckigen Klammern.
CANMATCH('[^0-9]') findet ein Zeichen, das **keine** Ziffer ist.

* Passt auf das Muster, hinter dem es steht **kein Mal oder beliebig oft.**
CANMATCH('Ot*o') findet *Oo* und *Oto* und *Otto* und *Otttttttto* ……
CANMATCH('.*') findet alle Zeichen dieser Zeile von der momentanen Textposition an.
CANMATCH('[0-9]*,[0-9]+'); findet Dezimalzahlen : *,12* oder *0,12* oder *47,11*

+ Passt auf das Muster, hinter dem es steht **ein** Mal oder beliebig oft.
CANMATCH('Ot+o') findet *Oto* und *Otto* und *Otttttttto* aber **nicht** *Oo*.

? Passt auf das Muster, vor dem es steht **kein** Mal oder beliebig oft. Ein *Leerzeichen* muss am Ende
stehen..
CANMATCH('br?') findet alles, was mit einem *b* beginnt, auf das kein oder beliebig viele *r*'s
folgen und das mit einem *Leerzeichen* abgeschlossen ist (nützlich zum Finden von Worten).

( ) Speichere ein gefundenes Muster.
CANMATCH('d(..p)') findet etwas, das mit *p* endet und mit *d* beginnt und zwei beliebige Zeichen dazwischen hat. Dies müssen die gleichen beiden Zeichen sein, die vor dem ersten Auftreten von *p* standen. (Selten gebraucht).

Soll eines der oben beschriebenen "Metazeichen" (^$*.+?[]()) gefunden werden, so müssen **zwei**
\ \(backslashes) vorausgehen. CANMATCH('\\.') findet einen Punkt. Ohne backslashes würde ein beliebiges Zeichen gefunden. CANMATCH('\\$') findet ein Dollar Zeichen.
CANMATCH('\\.+') findet einen oder eine Folge von beliebig vielen Punkten. Der backslash selber wird als vier backslashes dargestellt: '\\\\' meint einen mageren backslash. (Sorry, wegen \x0a Hexdarstellung; u.U. später zu ändern).

Befehle und Funktionen der Verarbeitungssektion

Funktionen

CONCATOP (string_atom ,string_atom) oder
string_atom | string_atom

Concatination (Zusammenhängen) von Zeichenketten.

Beispiel:
eine_variable = CONCATOP (‘Anfang ‘ , ‘Ende’); PRINT(eine_variable);
gibt aus: *Anfang Ende*

Kurzschreibweise: eine_variable = ‘Anfang ‘ | ‘Ende’;

SUBSTR (int_const , int_const , string_atom) oder
string_atom [string_atom, string_atom]

Kopiert eine Teilzeichenkette aus einer Zeichenkette:

Beispiel:
x = ‘the quick brown fox’; teil = x[0,3]; PRINT(teil);
gibt aus: *the*

Die Zählung der Zeichen einer Zeichenkette beginnt bei 0 (analog ‘C’)!
‘the quick brown fox’
012... d.h. das h ist das 1’te Zeichen in dem String!

INDEX (pattern , string_atom)

Liefert die Position eines Musters (pattern) in einer Zeichenkette.

Beispiel:
x = ‘the quick brown fox’; position = INDEX(‘quick’, x); PRINT(position);
gibt aus: *4*

x = ‘the quick brown fox’; teil = x[0, INDEX(‘quick’, x)]; PRINT(teil);
gibt aus: *the*

LINE

Gibt die Zeile ab dem momentanen Textzeiger aus. (Debug Möglichkeit) Aufruf: LINE;

TODAY

Gibt Datum und Uhrzeit zurück.

Beispiel:

```
x = TODAY;      PRINT(x);
```

DELETEFILE (dateiname)

Diese Funktion löscht die Betriebssystemdatei, die mit dateiname bezeichnet wird.
Bitte denken Sie daran, in Konstanten einen backslash durch \\ darzustellen!
Bitte mit Vorsicht benutzen! Löscht wirklich!

Beispiel:

```
DELETEFILE('aaa.txt');
```

MOVEFILE (ausgangsdatei, zieldatei)

Diese Funktion benennt (kopiert) die Ausgangsdatei in die Zieldatei um.
Achtung! Wie bei dem MOVE-Befehl von DOS ist die **Nichtexistenz** einer gleichnamigen Datei im Zielverzeichnis zu beachten!

Beispiel:

```
MOVEFILE('xxx.dat', 'yyy.dat');
```

curfile

Diese Funktion liefert den Namen der aktuell in Bearbeitung befindlichen Colddatei.

SYSTEM (befehl)

Diese Funktion führt den Betriebssystembefehl, der in der Variablen befehl steht, aus.

Beispiel:

```
SYSTEM ( 'DIR *.txt' );
befehl = ‘COPY  a.txt  b.txt’;
SYSTEM ( befehl );
```

LEFTCLIP (string_atom)
RIGHTCLIP (string_atom)

Entfernen Blanks und Tabulatoren am Anfang (LEFTCLIP) und vom Ende (RIGHTCLIP) einer Zeichenkette.

Beispiel:

```
x = ‘       hier     ‘;
a= RIGHTCLIP(x);
b= LEFTCLIP(x);
c= RIGHTCLIP(LEFTCLIP(x));
PRINT(a, b, c);
gibt aus: ‘   hierhier     hier‘
```

EMPTY

Dies ist das Attribut für “leer”.

Beispiel:

```
IF( x = EMPTY)
{ …… }
```

MATCH (string_atom , pattern)

Findet das Muster (pattern) in einer Zeichenkette (string_atom).

Beispiel:

```
string_atom = ‘the quick brown fox’;
pattern = ‘quick’;

x = MATCH(string_atom, pattern);
x = MATCH(‘the quick brown fox’, ‘quick’);

Beide liefern das selbe Ergebnis!
PRINT(x);
gibt aus:    quick
```

LENGTH (string_atom)

Gibt die Länge einer Zeichenkette zurück.

Beispiel:

```
PRINT( LENGTH(‘hallo’) );
gibt aus:  5
```

HEX (string_atom)

Gibt den hexadezimalen Code des ersten Zeichens in string_atom aus mit nachgestelltem Leerzeichen aus.

Beispiel:

```
PRINT(  HEX(‘Abel’)  );
gibt 41 (Hexcode von A) aus.
```

PRINT (string_atom)

Gibt den String auf den Bildschirm aus.

Beispiel:
PRINT(‘Abel’); Æ Gibt ‘Abel’ auf den Bildschirm aus.

Kontrollstrukturen

In der Aktionen-Sektion stehen folgende Kontrollstrukturen (Befehle, die den “Ablauf” des “Programms” beeinflussen) zur Verfügung:

IF (condition) doc_handle_section ELSE doc_handle_section

Bedingte Ausführung:

```
IF( a > 4 )
 {
        ……
        PRINT(‘hallo größer vier’);
}
ELSE
{
      …..
      PRINT(‘hallo kleiner gleich vier’);
}
```

Die geschweiften Klammern {} müssen dem IF und dem ELSE folgen und schließen beliebig viele Befehle ein.
Der ELSE-Zweig kann folgen, muss aber nicht.

WHILE (condition) doc_handle_section

Wiederholte Ausführung.

```
i = 0;       /* Variablen werden mit leeren Zeichenketten initialisiert nicht mit numerisch
0  !!! */
WHILE(  i < 5 )
{
      PRINT(‘Wert von I: ‘, i);
      i = i + 1;
}
```

WHILE ist **abweisend**, d.h. stimmt die Bedingung (condition) bei Eintritt in WHILE schon nicht, so werden die Befehle in der doc_handle_section (zwischen WHILE(){….}) **nie** ausgeführt.

Conditions (Bedingungen)

Es stehen folgende Bedingungen für IF und WHILE zur Verfügung:

a > b ist a größer als b? , a < b ist a kleiner als b?
a = b ist a gleich als b?, , a <> b ist a ungleich b?

Sind a **und** b Zahlen, so wird numerisch verglichen (d.h. 10 ist größer als 9). Ist entweder a **oder** b **nicht** numerisch (enthält ein Zeichen, das keine Ziffer ist), so wird als Zeichenkette verglichen. D.h. ‘10’ ist kleiner als ‘9’ (weil ‘1’ kleiner als ‘9’ ist).

Bedingungen (Conditions) können mit AND und OR verknüpft werden:

```
IF(  (a > b) AND ( b = 55) )
{
   …./* wenn a größer als b ist  und  b gleich der Zahl 55 ist dann… */
}

WHILE(  (a > b) OR ( b = 55) )
{
   …./* führe while-Schleife aus, solange wie a größer als b oder b gleich 55 ist */
}
```

Arithmetik

In der Aktionen-Sektion steht eine einfache, ganzzahlige Arithmetik mit den Grundrechenarten (+ - * /) und Klammerungsregel zur Verfügung.
Beispiel:

```
x = 2;  y = 4; z = 10;
PRINT(x+y);              ergibt  6
PRINT(x+y*z);            ergibt  42
PRINT(  (x+y)*z );       ergibt  60
PRINT( y/x );            ergibt  2
```

Es wird **nicht** zwischen Variablen, die **Zahlen** enthalten und solchen, die **Zeichenketten** enthalten unterschieden und es gibt dementsprechend auch keine Konversionsfunktionen Zahlen <->Zeichenketten.
Aber es ist möglich, direkt mit Zeichenketten, die (ganzzahlige) numerische Werte darstellen, zu rechnen:

```
'Mein_Dok_Typ':
{
        {       /* Beschreibungs Sektion   */
                x = CANMATCH('[0-9]+'); /* in x steht eine Zeichenkette nur aus Ziffern */
                ….
        }
        {       /* Aktionen Sektion */
                y = x + 4711;     /* rechnen mit dieser Zeichenkette */
                ….
        }
}
```

Sie sollten allerdings sicherstellen, dass solche Zeichenketten, mit denen gerechnet werden soll immer nur Ziffern enthalten (auch keine Dezimalkommata oder -punkte!).

Datenübergabe

In der Aktionen-Sektion werden die dort berechneten Werte mit den Funktionen BINDOUTFIELDS und WRITEKERNEL an den Proxess Kernel übergeben.

BINDOUTFIELDS (fieldname , string_atom);

Diese Funktion verbindet den **momentanen** Inhalt von *string_atom* mit dem Datenbankfeld *fieldname*.

```
{  /* Aktionen Sektion */
        x = 'Hallo Kernel';
        BINDOUTFIELDS(AName, x);
        x = 'du nicht';
}
```

Das Beispiel verbindet das Datenbankfeld AName mit dem Inhalt der Variablen x zu der Zeit, an der BINDOUTFIELDS aufgerufen wurde (also mit: 'hallo Kernel').
Beachte: Feldnamen sind Konstanten und stehen **nicht** in Hochkommata!

fieldname muss ein, für die im folgenden WRITEKERNEL angegebene Datenbank gültiger, physikalischer Feldname sein.

Besonderheiten:
A)
Der Feldname **FileData** hat eine besondere Funktion:

```
BINDOUTFIELDS(FileData, my_variable);
```

Dies legt fest, dass **nicht mehr** der Text des momentanen Dokuments aus der Cold-Datei beim nächsten WRITEKERNEL Aufruf als (Proxess-) Datei geschrieben wird, sondern der Inhalt der Variablen *my_variable*.
Diese Festlegung muss vor jedem WRITEKERNEL erneut erfolgen!
FileData muss genau so geschrieben werden! Groß- und Kleinschrift sind signifikant!
FileData steht an der Stelle, an der sonst die Kernelfeldnamen stehen und ist ein Schlüsselwort, das die Semantik der Funktion BINDOUTFIELDS erweitert. FileData darf deshalb nicht als Kernelfeldname in einem ColdScript vorkommen!

B)
Der Feldname **FilePath** hat ebenfalls eine besondere Funktion. Mit seiner Hilfe können eine oder mehrere beliebige Dateien an ein Dokument angehängt werden.

```
BINDOUTFIELDS( FilePath,  'path');
```

Hier werden an das momentane Dokument Dateien angehängt, die mit path beschrieben werden. Path ist ein beliebiger Pfadname. Z.B.:

Beispiel:

```
BINDOUTFIELDS( FilePath, '\\x\\y\\*.tif' );
```
alle Dateien mit der Extension tif im Verzeichnis \x\y

```
BINDOUTFIELDS( FilePath, '\\x\\y');
```
alle Dateien im Verzeichnis \x\y

FilePath steht an der Stelle, an der sonst die Kernelfeldnamen stehen und ist ein Schlüsselwort, das die Semantik der Funktion BINDOUTFIELDS erweitert. FilePath darf deshalb nicht als Kernelfeldname in einem ColdScript vorkommen!

BINDOUTFIELDS(FilePath**, 'path')** kann nur einmal vor einem WRITEKERNEL benutzt werden.
Das Muster in **'path'** ist im Sinne von z.B. DIR zu verstehen (a*tif oder a????.txt ; **nicht** [0-9]*.tif).
Bitte denken Sie daran in Konstanten einen backslash durch \\ darzustellen!
Den **Dateityp** (TIFF-Datei, WinWord-Datei.... wichtig um das richtige Bearbeitungsprogramm im Frontend aufrufen zu können) leitet der Import Server aus der **Extension** (Dateiendung, test**.txt** , mein**.doc**...) ab. Welcher Dateityp welcher Extension zugeordnet ist kann mit dem Administrator- Frontend eingestellt werden.
Damit der Import Server diese Zuordnung eindeutig vornehmen kann, dürfen gleiche Extensions nicht mehreren Dateitypen zuordnet werden! Der Import Server unterscheidet nicht zwischen groß- und kleingeschriebenen Extensionen.

Beispiel für Problemfall:
Extension= TXT, zugeordnet RedLine und NotpadText

Außerdem müssen alle Dateien, die verarbeitet werden sollen, eine Extension besitzen!

Beispiel für Problemfall:
\\x\\y\\DoesNotWork funktioniert nicht
aber:
\\x\\y\\GoodBoy.TXT funktioniert!

Beispiel:
.....

```
BINDOUTFIELDS(AName, 'Klaus');
BINDOUTFIELDS(StringField2, kundennr );
BINDOUTFIELDS(StringField3, rechnungsnr );
BINDOUTFIELDS(StringField4, rechnungsdatum );
BINDOUTFIELDS(FilePath, '\\x\\y\\a*.txt' );      /* alle a*.txt in \\x\\y werden zu Dateien im Dokument */
WRITEKERNEL('red2','eine Datei','Notiz-Zettel', 'Hymer','Ausgangsrechnung');*/
```

path kann auch eine Variable sein:

```
verz  = '\\x\\y\\';
datei = 'myfile';
pfad = verz  |  datei | '.txt' ; /* verbindet verz ,  datei und '.txt'  zu '\\x\\y\\ myfile.txt' */
BINDOUTFIELDS(FilePath, pfad);
```

WRITEKERNEL (databasename, filedes, filetyp , docdes , doctyp);

Beauftragt den Proxess Kernel, die bis hierher mit BINDOUTFIELDS gebundenen Felder und den Inhalt von *Coldtext* wegzuschreiben.
Hierbei sind *databasename, filedes, filetyp, docdes* und *doctyp* Pflichtfelder und müssen **alle** angegeben werden.
Besondere Beachtung ist dem Databasename zu widmen. Dieser ist eine **Stringkonstanten**, d.h. der Datenbankname muss immer explizit als String angegeben werden. **Er kann nicht in einer Variablen an die Funktion übergeben werden!**

Allgemein gilt:
databasename: Name der Datenbank, die den Datensatz aufnehmen soll (muss existieren und dem Kernel "bekannt" sein; richtige "master" Datenbank; richtiger Benutzer)
filetyp: Der Dateityp, der unter dem *Coldtext* geschrieben werden soll. (muss in database existieren)
doctyp: Der Dokumententyp, unter dem dieser Datensatz entstehen soll (muss in database existieren).
filedes : beliebiger Namen für die angehängte Datei (darf aber nicht leer sein!).
docdes: beliebiger Namen für das Dokument (darf aber nicht leer sein!).

Besonderheiten:
Ist der Coldtext kürzer als ein Zeichen, so wird **keine** Datei angelegt.
Steht in *filedes* 'nofile', so wird **keine** Datei angelegt.

Beispiel:

```
{  /* Aktionen Sektion */
        x = 'Hallo Kernel';
      my_var ='noch ein gedicht!';
      BINDOUTFIELDS(AName, x);
      BINDOUTFIELDS(StringField2, my_var);
      BINDOUTFIELDS(StringField3, 'immer das selbe');
      WRITEKERNEL('red2', 'erste Datei', 'Notiz-Zettel', 'erstes Dokument',
'Aktennotiz');
}
```

Anmerkung:
Der Import Server überprüft das Vorhandensein einer Kernelverbindung nur, wenn das Coldscript kernelrelevante Befehle wie etwa den WIRTEKERNEL enthält. Coldscripts, die keine solchen Befehle enthalten, können auch völlig „stand alone", also auf einer isolierten Maschine, ohne Kernelverbindung ablaufen. Dies reicht zum Testen meist völlig aus.

GETKERNELERR()

Diese Funktion liefert eine '1' zurück, wenn der vorangegangene WRITEKERNEL erfolgreich war. Im Fehlerfall gibt sie eine '0' zurück.

Beispiel:

```
WRITEKERNEL('red2','eine Datei','Notiz-Zettel', 'Hymer','Ausgangsrechnung');
IF( GETKERNELERR() = '1' )
{
        /* hier, wenn WRITEKERNEL erfolgreich war */
        MOVEFILE (bearbeitete_datei, zieldatei)
}
```

Betriebsarten des Coldconverters

Aufruf und Serverbetrieb

Es gibt drei Möglichkeiten, coldconv.exe von der Kommandozeile aus aufzurufen:
ad1)

1) *Coldconv -i script_datei daten_datei*
Hier wird die script_datei eingelesen und auf die daten_datei angewandt. Danach wird coldconv beendet. Diese Betriebsart eignet sich besonders zum Entwickeln von Cold Scripts.

2) *coldconv* (ohne parameter)
Serverbetrieb ohne Verzeichnissuche

3) *convcold -d Wurzelverzeichniss Colddaten Coldscript*
Serverbetrieb mit Verzeichnissuche (Bsp. convcold -d \x\y\daten *.dat mein.scr)

ad 2) Beim **Serverbetrieb ohne Verzeichnissuche** erwartet coldconv folgende Verzeichnissstruktur:

ColdHomePath \data
\done
\errs und

das Cold-Script in:

ColdHomePath \coldscript

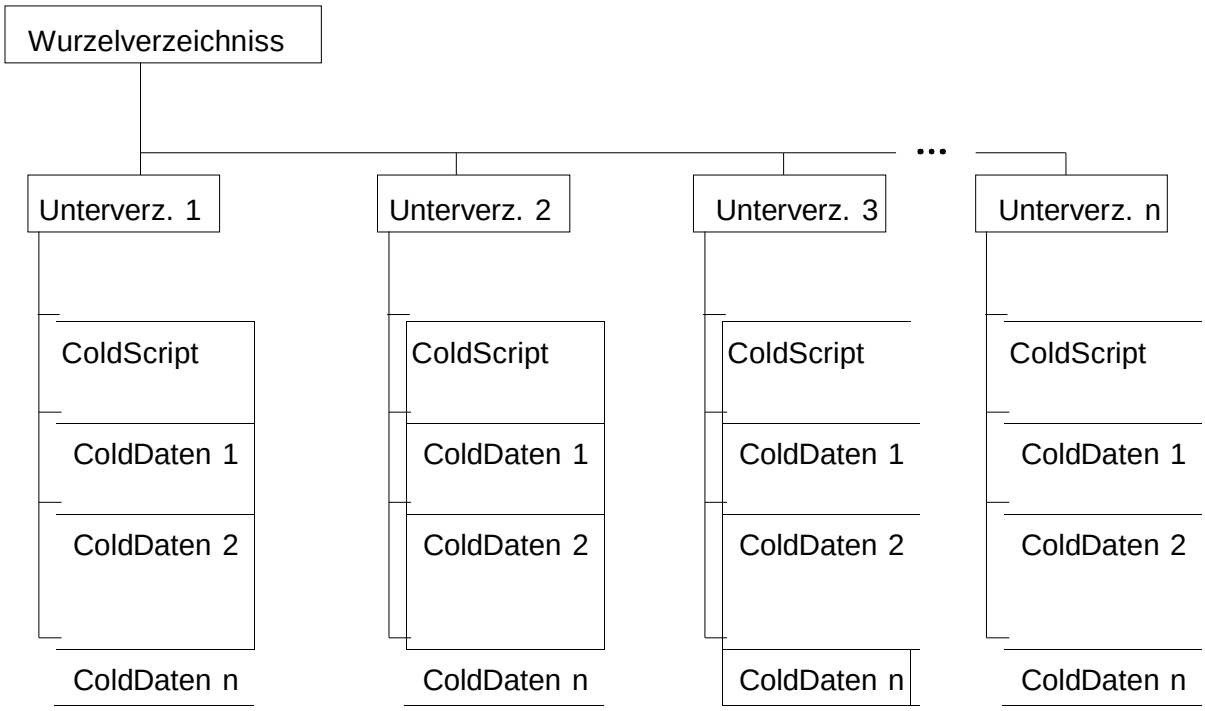
ColdHomePath wird aus dem Registry-Eintrag
HKEY_LOCAL_MACHINE\\software\\shd\\document!\\coldconv
ColdHomePath
gelesen.

Im Serverbetrieb verarbeitet der Import Server alle Dateien, die er im Verzeichnis ColdHomePath\data findet. Gesteuert wird diese Verarbeitung durch das Cold-Script, das er in der Datei ColdHomePath\coldscript erwartet. Es wird auf jede Datei ColdHomePath\data angewandt. Nach Abschluss der Verarbeitung der Dateien aus dem Verzeichnis ColdHomePath\data stehen diese im Verzeichnis ColdHomePath\done. Der Dateiname im Verzeichnis done wird aus einem „C“ und der laufenden Nummer der Cold Datei gebildet (C0,C1,C2……).
Vorsicht: Wenn es beim Start von Import Server schon C* (C0,C1,C2……) -Dateien im Verzeichnis ColdHomePath\done gibt, so werden diese **nicht überschrieben! Die Daten bleiben im Basisverzeichnis stehen und werden immer wieder übergeben!**
Es besteht keine explizite Synchronisation zwischen Import Server und dem (Host-) Programm, das die Dateien in ColdHomePath\data erstellt. Der Import Server erwartet, dass das Betriebssystem (analog SHARE) Probleme wie „Import Server liest und Host hat Datei noch zum schreiben geöffnet“ auflöst.
Hat der Import Server alle Dateien im Verzeichnis ColdHomePath\data abgearbeitet, so wartet er ca. 30 Sekunden, bevor er nachschaut, ob neue Dateien angekommen sind.

ad 3) **Serverbetrieb mit Verzeichnissuche:**

In dieser Betriebsart durchsucht ColdConv periodisch (alle 30 Sek.) alle Unterverzeichnisse vom Wurzelverzeichniss**.** Er sucht dort nach ColdDatenDateien**,** die auf das Muster in ColdDaten passen**.**

Das Muster ist im Sinne von z.B. DIR zu verstehen. ColdConv führt dann das Script Coldscript aus**. Der Name von Coldscript darf kein Muster sein; es darf nur eines pro Unterverzeichnis geben.**
ColdConv wechselt sein Arbeitsverzeichniss in die Unterverzeichnisse. D.h. die Dateien der Unterverzeichnisse können im ColdScript direkt (ohne vorherige Pfadangabe) angesprochen werden.



„Aufräumarbeiten“, die sicherstellen das ColdConv im nächsten „Durchgang“ durch die Unterverzeichnisse nicht noch einmal die gleichen Dateien bearbeitet, muss das ColdScript selbst vornehmen. Damit kann das ColdScript auch dieses Verhalten an die Kundenwünsche anpassen.

Reservierte Befehlswörter

DATABASE
DOCNAME
ENDOFDOCPATTERN
ENDOFLINE
IGNORECHARS
IGNORELINE
IGNOREWHITE
IF GOTO
MOVETO
INTEGER
CONCAT
WRITEKERNEL
PRINT
INDEX LINE
INPUTCONVERSION
OUTPUTCONVERSION
NONE
ASCII7
EBCDIC
MUSTMATCHMULTI
MUSTMATCH
CANMATCH
CANMATCHMULTI
SUBSTR
GETCHARS
TODAY
LEFTCLIP
RIGHTCLIP
FILENAME
EMPTY
MATCH
ELSE
FOR
WHILE
END
STRINGMAP
CHARMAP
BINDOUTFIELDS
WRITEKERNEL
USEDLL
MOVEFILE
DELTEFILE
SYSTEM

Bekannte Fehler und Meldungen

Bekannte Bugs und Probleme:

1) In Regulären Ausdrücken müssen Zeichen, die für den regulären Ausdruck selbst Bedeutung haben (*.[]+$^\) mit **zwei** backslashes escaped werden. Beispiel: MATCH(x, ‘bananas$’) findet bananas am Stringende (Bedeutung vom $ in regulären Ausdrücken). MATCH(x, ‘bananas\\$’) findet bananas$, also wirklich ein Dollar-Zeichen.

2) Hat man sich bei dem Wert von ENDOFDOCPATTERN geirrt (Muster existiert in der Cold-Datei nicht), so zeigt der Import Server sein Logo und bleibt anscheinend hängen. Bei großen Cold- Dateien und klein ausgelegten PCs beginnt u.U. eine heftige Plattenaktivität. Jedoch untersucht der Import Server die gesamte Cold-Datei (die meist recht lang ist) nach dem (nicht existenten) Muster. Das dauert.
   Lösung: mit <cntrl>C abbrechen; ENDOFDOCPATTERN berichtigen.

3) Bis Version 0.0.1 geht IGNORELINE nur eine Zeile weiter, egal wie groß der Parameter gewählt wurde.

Fehlermeldungen

```
C:\data>convcold a.cld data.dat
Cold Converter Ver. 0.0.1  (c) SHD (1996)
No RPC-Connection.
Cannot log in
```

Hier ist der Kernel nicht hoch gefahren oder ist netzwerkmäßig nicht erreichbar. Oder der Registry- Eintrag „HostName“ stimmt nicht.

```
Cold Converter Ver. 0.0.1  (c) SHD (1996)
Cannot log in
```

Hier war der Benutzername in der Registry („ShortName“) oder dessen Passwort („PassWort“) falsch.

```
Cold Converter Ver. 0.0.1  (c) SHD (1996)
Exception code: 0x20008000
Kernel exception!
Execption from kernel for database: red5
```

Hier wurde versucht, mit einer nicht existierenden Datenbank (red5) zu arbeiten.

Im ColdScript:

```
WRITEKERNEL('red5','eine Datei','Notiz-Zettel', 'Hymer','Ausgangsrechnung');
```

```
Cold Converter Ver. 0.0.1  (c) SHD (1996)
WRITEKERNEL: FileTypeName  Notiz-ZettelError does not exist in red2
Valid entries are:

F i e l d s
FieldName:  AName    Length: 50 Type: 2
FieldName:  DateField1    Length: 0 Type: 5
FieldName:  DateField2    Length: 0 Type: 5
FieldName:  DateField3    Length: 0 Type: 5
FieldName:  DoubleField1    Length: 0 Type: 3
FieldName:  DoubleField2    Length: 0 Type: 3
FieldName:  DoubleField3    Length: 0 Type: 3
FieldName:  IntegerField1    Length: 0 Type: 0
FieldName:  IntegerField2    Length: 0 Type: 0
FieldName:  IntegerField3    Length: 0 Type: 0
FieldName:  StringField1    Length: 5 Type: 2
FieldName:  StringField2    Length: 10 Type: 2
FieldName:  StringField3    Length: 15 Type: 2
FieldName:  StringField4    Length: 25 Type: 2
FieldName:  StringField5    Length: 50 Type: 2
FieldName:  StringField6    Length: 75 Type: 2
FieldName:  StringField7    Length: 100 Type: 2
FieldName:  StringField8    Length: 25 Type: 2
FieldName:  StringField9    Length: 10 Type: 2
D o c t y p e s
DocTypeName:  Aktennotiz    DocTypeId: 53799
DocTypeName:  Ausgangsrechnung    DocTypeId: 32338
DocTypeName:  Barcode-Scanhalde    DocTypeId: 40000
DocTypeName:  Eingangsrechnung    DocTypeId: 32337
DocTypeName:  Kuendigung    DocTypeId: 32343
DocTypeName:  Mahnung    DocTypeId: 32340
DocTypeName:  Rechnung    DocTypeId: 32339
DocTypeName:  Reklamation    DocTypeId: 32341
DocTypeName:  Technical notes    DocTypeId: 41152
DocTypeName:  muellhalde    DocTypeId: 54316
DocTypeName:  nichtsnutz    DocTypeId: 54276
F i l e t y p e s
FileTypeName:  DClip-Datei    FileTypeId: 11111
FileTypeName:  Notiz-Zettel    FileTypeId: 43313
FileTypeName:  Redlines    FileTypeId: 53179
FileTypeName:  Screenshot    FileTypeId: 53788
FileTypeName:  Word fnr Windows    FileTypeId: 43316
FileTypeName:  gescanntes Dokument    FileTypeId: 43314
deleting: red2
```

Hier wurde versucht, einen falschen Dateitypnamen anzugeben (Notiz-ZettelError).

Im ColdScript:

```
WRITEKERNEL('red2','eine Datei',''Notiz-ZettelError'', 'Hymer','Ausgangsrechnung');
```

Analoges passiert, wenn der Dokumenttyp (hier: 'Ausgangsrechnung') falsch angegeben wurde.

Anhänge mit Beispielen

Anhang A Beispiel eines Cold Scripts

Beispiel eines Cold Scripts:
(nicht besonders sinnfällig, aber zeigt die wesentlichen Strukturen)

```
{
/* Parameter Sektion */
        USEDLL = 'xlat.dll';
        ENDOFDOCPATTERN = '\x0c';  /* ANSI FormFeed */
        ENDOFLINE = '\x0d\x0a';          /* ANSI CR   und  ANSI LF  */
        STRINGMAP
        {
        }
        CHARMAP
        {
        }
}

'Mitglied':
{
        {
        /* Erkennungs Sektion für Doc Typ Mitglied */
                x1 = CANMATCH('.*');            /* die 1. Zeile, .* paßt auf gesamte Zeile */
                IGNORELINE(1);                  /* eine Zeile weiter */
                x2 = CANMATCH('.*');            /* die 2. Zeile */
                IGNORELINE( 2 );                /* zwei  Zeilen weiter */
                x3 = CANMATCH('.*');            /* die gesamte Zeile */
                x4 = MUSTMATCHMULTI('Mitglied:');  /* von hier an über das gesamte
                                                      Dokument; wenn Treffer, führe
AKTIONEN aus*/
                nummer = CANMATCH('[0-9]+');  /* eine ganze Zahl */
        }
        {
        /* Aktionen für Doc Typ Mitglied */
                PRINT('got doctyp: ',doctyp);
                a = 4;
                b = 10;
                PRINT('Arith: ', a + b, '   ', b * 2,  '   ', a + b * 2 , '   ',  (a + b) * 2 );
                IF(b > 2)
                {
                        PRINT('right, 10>2');
                }
                ELSE
                {
                        PRINT('NOPE!');
                }

                PRINT(x1);
                PRINT(x2);
                PRINT(x3);
                PRINT(x4,'   Length= ', LENGTH(x4));
                PRINT('ein teil ' | 'und das andere');
                xxx = 'I am a substr';
                PRINT(xxx, '  von 2 bis anfang von "sub" ist:  ', xxx[2,INDEX('sub', xxx)] );
                i = 0;
                j = 10;
                WHILE(i < j + 1)
```

```
                {
                        PRINT('i = ', i);
                        i = i + 1;
                }
                PRINT('Mitgliednummer: ', nummer);
                i = LENGTH(xxx) - 1 ;
                PRINT('Länge von ',xxx,' ist ', i);
                WHILE(i > 0)
                {
                        PRINT(i, ' --- ', xxx[i,i+1]);
                        i = i - 1;
                }
        }
}


'Summe': {
        {
                /* Erkennungs Sektion für Doc Typ Summe */
                x1 = CANMATCH('.*');
                IGNORELINE(1);
                x2 = CANMATCH('.*');
                IGNORELINE(1 );
                IGNORELINE(1);
                x3 = CANMATCH('.*');

                x4 = MUSTMATCHMULTI('S u m m e');
                test = CANMATCH('[0-9]+');

        }
        {
                /* Aktionen für Doc Typ Summe */
                PRINT('got doctyp: ',doctyp);
                PRINT(x1);
                PRINT(x2);
                PRINT(x3);
                PRINT(x4);
                PRINT('meine zahl ist:', test);
                PRINT('--------------Summe---------------');
        }
}

'Lieferant' :{
        {
                x1 = CANMATCH('.*');
                IGNORELINE(1);
                x2 = CANMATCH('.*');
                IGNORELINE(1 );
                IGNORELINE(1);
                x3 = CANMATCH('.*');
                x4 = MUSTMATCHMULTI('Lieferant:');
                nummer = CANMATCH('[0-9]+');
        }
        {
                PRINT('got doctyp: ',doctyp);
                PRINT(x1);
                PRINT(x2);
                PRINT(x3);
                PRINT(x4);
                PRINT('Lieferant nummer: ', nummer);
```

```
                PRINT('----------Lieferant----------');
        }
}

'catchall' :{ /* ohne MUST…., fängt alles, was bis hierher noch nicht erkannt wurde */
        {
                x1 = CANMATCH('.*');
                IGNORELINE(1);
                x2 = CANMATCH('.*');
                IGNORELINE(1 );
                IGNORELINE(1);
                x3 = CANMATCH('.*');
                IGNORELINE(1);
                x4 = CANMATCH('.*');
        }
        {
                PRINT('got doctyp: ',doctyp);
                PRINT(x1);
                PRINT(x2);
                PRINT(x3);
                PRINT(x4);
                PRINT('----------Catch all the rest----------');
        }
}
END
{

        PRINT('bin an EOF…');
}
```

Anhang B Sammeln mehrerer Dokumente

„Strickmuster“ zum Sammeln mehrerer Dokumente in einer (Proxess-) Datei.

Problemstellung: In der 2. Zeile in jedem Dokument finden wir …BLATT <nummer>. Diese „Blätter“ sollen zu einer Datei in einem (Proxess-) Dokument zusammengefasst werden. Zusammengehörige Blätter liegen immer in aufsteigender Richtung hintereinander (BLATT 1, BLATT 2,….). Sobald wieder BLATT 1 auftaucht, beginnt eine neue Datei.

```
{
/* Parameter Sektion */
        ENDOFDOCPATTERN = '\x0c';   /* ANSI FormFeed */
        ENDOFLINE = '\x0d\x0a';          /* ANSI CR und LF */
        {  /* Initialisierung; nicht schön aber wirksam */
                oldblatt=0;
                allpages=' ';
        }
}

'Alles':
{
        {
        /* Erkennungs Sektion für Doc Typ ALLES*/
                IGNORELINE(1);
                x1 = CANMATCH('.*');                    /* Zeile, in der ….BLATT 1… vorkommt
*/
                GOTO(0,0);
                all=CANMATCHMULTI('.*');             /* gesamtes doc */
        }

        {
        /* Aktionen für Doc Typ Alles */
                blatt=MATCH(x1 ,'BLATT[ ]*[0-9]*');
                IF(blatt = EMPTY)
                {
                        PRINT(‘BLATT nicht gefunden;  H I L F E!!');
                        PRINT(all);
                }
                blattnummer=MATCH(blatt ,'[0-9]+');
                IF( (oldblatt >  blattnummer) OR  (oldblatt =  blattnummer) )
                {          /* hier wird Dokument + Datei geschrieben */
                        oldblatt = 0;
                        allpages=all;
                }
                ELSE
                {
                        oldblatt = blattnummer;
                        allpages=allpages | all;   /* Hänge momentanes Dokument an */
                }
        }
}


END
        {
                /* hier wird das letzte Dokument + Datei geschrieben */
        }
```

Anhang C Formale Grammatik von Cold Script

```
program:        global_section doc_list

doc_list:    /* empty */
             |        doc_list ASTRING ':'   '{' doc_des_section doc_handle_section '}'

global_section:  /* empty */
             |        '{'  global_ops '}'

global_ops:   /* empty */
             |              global_ops global_op

global_op:           DOCNAME '=' ASTRING ';'
             |       ENDOFDOCPATTERN '=' ASTRING ';'
             |       ENDOFLINE '=' ASTRING ';'
             |       USEDLL '=' ASTRING ';'
             |       DATABASE '=' ASTRING ';'
             |       INPUTCONVERSION '=' conversion ';'
             |       OUTPUTCONVERSION '=' conversion ';'
             |       STRINGMAP '}'
             |       CHARMAP '}'

char_map_list: /* empty */
             |       char_map_list char_map

char_map: ASTRING '|' ASTRING

string_map_list: /* empty */
             |       string_map_list string_map

string_map: ASTRING '|' ASTRING

conversion:          NONE
             |       ASCII7
             |       EBCDIC

doc_des_section:              '{'  doc_des_ops '}'

doc_des_ops: /* empty */
             |       doc_des_ops    doc_des_op

doc_des_op:          string_doc_des  ';'
             |       variable  '='  string_doc_des  ';'
             |       doc_des_void  ';'

doc_des_void:        IGNORELINE '(' int_const ')'
             |       IGNOREWHITE '('     ')'
             |       IGNORECHARS '(' int_const ')'
             |       GOTO  '(' int_const ',' int_const ')'
             |       MOVETO '(' int_const  ','  int_const ')'
             |       LINE

string_doc_des:               MUSTMATCH '(' pattern ')'
             |       CANMATCH '(' pattern ')'
             |       CANMATCHMULTI '(' pattern ')'
             |       MUSTMATCHMULTI '(' pattern ')'
             |       GETCHARS '(' int_const ')'
```

```
pattern:            string_const

variable:                   NAME_CONST

doc_handle_section:     /* empty */
                    |       '{'  doc_handle_ops  '}'

doc_handle_ops:
                    |       doc_handle_op doc_handle_ops

doc_handle_op:          string_ops
                    |       kernel_ops
                    |       ifstmt
                    |       whilestmt

string_ops:     variable '=' string_atom ';'

string_const:   ASTRING

int_const:      INTEGER

variable_ref:   variable

string_atom:                variable_ref
                    |       int_const
                    |       string_const
                    |       CONCATOP '(' string_atom ',' string_atom ')'
                    |       SUBSTR '(' int_const ',' int_const ',' string_atom ')'
                    |       string_atom '[' string_atom ',' string_atom ']'
                    |       INDEX '(' pattern ',' string_atom ')'
                    |       LINE
                    |       TODAY
                    |       LEFTCLIP '(' string_atom ')'
                    |       RIGHTCLIP '(' string_atom ')'
                    |       EMPTY
                    |       MATCH '(' string_atom ',' pattern ')'
                    |       LENGTH '(' string_atom ')'
                    |       HEX ‘(‘string_atom ')'
                    |       '(' string_atom ')'
                    |       string_atom '+' string_atom
                    |       string_atom '-' string_atom
                    |       string_atom '*' string_atom
                    |       string_atom '/' string_atom
                    |       string_atom '|' string_atom


ifstmt:                     IF  '(' cond  ')'   '{' doc_handle_ops  '}'  elseclause

elseclause:  /* leer */
                    |       ELSE '{' doc_handle_ops '}'

whilestmt:                  WHILE '(' cond ')'    '{'  doc_handle_ops  '}'

cond:                       '(' cond ')'
                    |       cond AND cond
                    |       cond OR cond
                    |       string_atom '<' string_atom
                    |       string_atom '>' string_atom
                    |       string_atom '=' string_atom
```

```
                | string_atom NE string_atom


kernel_ops:     WRITEKERNEL '(' ASTRING ',' ASTRING ',' ASTRING ',' ASTRING ','
ASTRING')' ';'
                | BINDOUTFIELDS '(' fieldname ',' string_atom ')' ';'
                | PRINT '(' print_list ')' ';'

print_list: /*leer */
                | string_atom
                | print_list ',' string_atom

fieldname:      NAME_CONST
```

Anhang D schrittweise Entwicklung eines Scripts

Tipps zur schrittweisen Entwicklung eines ColdScripts

Schritt eins: Zeichen und Aufteilung in Einzeldokumente

Zunächst sollte man sich die Cold Datei mit einem einfachen Editor (notepad) ansehen. Ist sie gänzlich unleserlich, so könnte sie im EBCDIC Zeichensatz vorliegen. Dann mit einem Konversionswerkzeug nach ANSI übersetzen.
Danach versuchen, im Textfluss Zeichen(-folgen) zu finden, die einzelne Dokumente abgrenzen. Ein FormFeed (0x0c) ist ein guter Anfang dafür. Um den genauen Hex Code zu ermitteln einen Hex Editor (HEX-Works) benutzen und nach dem Kontext suchen. Z.B.: Im Notepad war zu sehen:

mfg
Ambrosius<komisches Zeichen>

dann im Hex-Editor nach Ambrosius suchen und nachsehen, welches Zeichen folgte. Die Hypothese (mein ENDOFDOCPATTERN ist…) testen, indem man im HEX-.Editor mehrfach nach diesem Zeichen (dieser Zeichenfolge) sucht und nachschaut, ob der Kontext plausibel ist. Im Hex-Editor nachschauen, wie das Zeilenende aussieht CR(0x0d),LF(0x0a) oder LF,CR oder nur LF…..
Dann ein erstes ColdScript schreiben:

```
{
/* Parameter Sektion */
        ENDOFDOCPATTERN = '\x0c';           /* Dokument Trenner ist FormFeed */
        ENDOFLINE = '\x0d\x0a';             /* Zeilentrenner ist CR,LF  */
}

'Alles':
{
        {
                /* Erkennungs-Sektion für Doc Typ ALLES*/
                all=CANMATCHMULTI('.*');    /* das gesamte Dokument in die Variable all*/
        }
        {
                /* Aktionen für Doc Typ Alles */
                /* Wird immer ausgeführt, da kein MUSTMATCH… in Erkennungs-Sektion */

                PRINT('************************************************************');
                PRINT(all);

        }
}
```

Das Script mit
convcold script_name cold_text_datei
ausführen. Zu diesem Zeitpunkt wird noch kein Proxess-Kernel benötigt. Auf dem Bildschirm sollte nun die cold_text_datei, durch Zeilen von „***..“ nach Dokumenten aufgeteilt, vorbei scrollen. Wenn das zu schnell geht, kann man die Ausgabe in eine Datei lenken:
convcold script_name cold_text_datei > tempor.tmp
und dann tempor.tmp „in Ruhe“ mit notepad ansehen.

Nun können z.B. noch unerwünschte Escape Sequenzen (Druckersteuerung) in cold_text_datei vorkommen. Im Beispiel unten werden alle 2-stelligen Escape Sequenzen (Hex 1b gefolgt von zwei beliebigen Zeichen (\.)) durch leer (‘ ‘) ersetzt.

```
{
/* Parameter Sektion */
        ENDOFDOCPATTERN = '\x0c';           /* Dokument Trenner ist FormFeed */
        ENDOFLINE = '\x0d\x0a';             /* Zeilentrenner ist CR,LF  */
        STRINGMAP
        {
                '\x1b\.\.'   |    ' '
        }
}

'Alles':
{
        {
                /* Erkennungs Sektion für Doc Typ ALLES*/
                all=CANMATCHMULTI('.*');       /* das gesamte Dokument in die Variable all*/
        }
        {
                /* Aktionen für Doc Typ Alles */
                /* Wird immer ausgeführt, da kein MUSTMATCH… in Erkennungs Sektion */

                PRINT('************************************************************');
                PRINT(all);

        }
}
```

Zeichensätze und einzelne Zeichen können umdefiniert werden; DLLs dazu geladen werden usw. .
An dieser Stelle sollte cold_text_datei in allen Zeichen lesbar und nach einzelnen Dokumenten aufgeteilt beim Durchlauf durch ColdConv zu sehen sein.

Schritt zwei: Verschiedene Dokumenttypen

Annahme: Wir haben (keep it simple!) zwei Dokumenttypen, die sich dadurch unterscheiden, dass sie entweder in der 1. Zeile des Dokuments mit „RECHNUNG“ oder mit „LIEFERSCHEIN“ beginnen.
Wir ergänzen unser ColdScript um die Dokumenttypen 'TypRechnung' und ‘TypLieferschein’:

```
{
/* Parameter Sektion */
        ENDOFDOCPATTERN = '\x0c';           /* Dokument Trenner ist FormFeed */
        ENDOFLINE = '\x0d\x0a';             /* Zeilentrenner ist CR,LF  */
        STRINGMAP
        {
                ‘\x1b\.\.’   |    ‘ ‘
        }
}

'TypRechnung':
{
        {
                /* Erkennungs-Sektion für Doc Typ TypRechnung */
                MUSTMATCH('RECHNUNG');   /* nur bei RECHNUNG in der 1. Zeile */
        }
        {
                /* Aktionen für Doc Typ TypRechnung */
                PRINT(‘Habe Rechnung gefunden’);
        }
}

‘TypLieferschein’:
{
        {
                /* Erkennungs-Sektion für Doc Typ TypLieferschein */
                MUSTMATCH(‘LIEFERSCHEIN’);     /* nur bei LIEFERSCHEIN in der 1. Zeile */
        }
        {
                /* Aktionen für Doc Typ TypLieferschein */
                PRINT(‘Habe Lieferschein gefunden’);
        }
}
‘FangDenRest’:
{
        {       /* Erkennungs-Sektion für alles, was vorher nicht erkannt wurde */
                all=CANMATCHMULTI('.*');       /* das gesamte Dokument in die Variable all*/
        }

        {
                /* Aktionen für Doc Typ FangDenRest*/
                PRINT(‘********** unbekannt **************’);
                PRINT(all);
        }
}
```

Die Aktionen für den Dokumenttyp FangDenRest werden nur ausgeführt, wenn vorher noch kein anderer Dokumenttyp „passte“. D.h. alles was dort erscheint, muss einen eigenen Dokumenttyp erhalten oder die MUSTMATCH(…) der anderen Dokumettypen müssen entsprechend korrigiert werden.
An dieser Stelle sollten alle Dokumenttypen in cold_text_datei erkannt worden sein.

Schritt drei: Felder (Schlagworte)

Wir sollten vom Kunden Informationen haben, welche Felder (Schlagworte) wir aus den verschiedenen Dokumenttypen extrahieren sollen. In unserem Beispiel setzen wir voraus, daß in der 2. Zeile der Rechnung der Kundenname steht. Den Lieferschein betrachten wir nicht mehr. Außerdem betrachten wir nur noch den für TypRechnung zuständigen Teil des ColdScripts.

```
'TypRechnung':
{
        {
                /* Erkennungs Sektion für Doc Typ TypRechnung */
                MUSTMATCH('RECHNUNG');   /* nur bei RECHNUNG in der 1. Zeile */
                IGNORELINE(1);                  /* nächst Zeile */
                kundenname = CANMATCH('[A-z].*');
        }
        {
                /* Aktionen für Doc Typ TypRechnung */
                PRINT('Habe Rechnung gefunden');
                PRINT('Kundenname:  ', kundenname);
        }
}
```

Der reguläre Ausdruck **'[A-z].*'** findet eine Zeichenkette, die mit einem Buchstaben beginnt und dann mit beliebig vielen beliebigen Zeichen (maximal bis Zeilenende) weitergeht. Dieser Werte (hoffentlich der Kundenname) wird der Variablen kundenname in der Erkennungssektion zugewiesen.
An dieser Stelle sollte zu jeder Rechnung in cold_text_datei der Kundenname ausgegeben worden sein.

Schritt vier: Felder und Dokumenttext in POXESS wegschreiben

```
'TypRechnung':
{
        {
                /* Erkennungs-Sektion für Doc Typ TypRechnung */
                MUSTMATCH('RECHNUNG');   /* nur bei RECHNUNG in der 1. Zeile */
                IGNORELINE(1);           /* nächst Zeile */
                kundenname = CANMATCH('[A-z].*');
        }
        {
                /* Aktionen für Doc Typ TypRechnung */
                PRINT('Habe Rechnung gefunden');
                PRINT('Kundenname:  ', kundenname);
                BINDOUTFIELDS(StringField5,  kundenname);
                WRITEKERNEL('red2',  'MeineDatei',  'Notiz-Zettel',  'Eine Rechnung',
                                'Ausgangsrechnung');
        }
}
```

Erst jetzt müssen wir sicherstellen, dass wir eine Connection zum Proxess-Kernel haben.

BINDOUTFIELDS verbindet das (Datenbank-) Feld **StringField5** mit der (ColdConv-) Variablen **kundenname**. WRITEKERNEL beinhaltet die eigentliche Schreiboperation: auf der Datenbank **'red2'** wird ein Dokument vom Typ **'Ausgangsrechnung'** mit dem Dokumentnamen **'Eine Rechnung'** angelegt. Dazu wird der Cold-Text in der Datei vom Typ **'Notiz-Zettel'** mit Namen **MeineDatei** abgelegt.
An dieser Stelle sollte zu jeder Rechnung in cold_text_datei ein Datensatz in Proxess entstanden sein, der den Cold-Text als Datei und den Kundennamen in StringField5 enthält.

Schritt fünf: zur Verzierung…
Gibt am Ende des Scripts die Anzahl der gefundenen Rechnungen aus.

```
{/* Parameter Sektion */
        ENDOFDOCPATTERN = '\x0c';           /* Dokument Trenner ist FormFeed */
        ENDOFLINE = '\x0d\x0a';             /* Zeilentrenner ist CR,LF  */
        STRINGMAP
        {
                '\x1b\.\.'   |    ' '
        }
        {
                zahl = 0;
        }
}
'TypRechnung':
{
        {
                /* Erkennungs-Sektion für Doc Typ TypRechnung */
                MUSTMATCH('RECHNUNG');   /* nur bei RECHNUNG in der 1. Zeile */
                IGNORELINE(1);           /* nächst Zeile */
                kundenname = CANMATCH('[A-z].*');
        }
        {
                /* Aktionen für Doc Typ TypRechnung */
                PRINT('Habe Rechnung gefunden');
                PRINT('Kundenname:  ', kundenname);
                zahl = zahl + 1;
                BINDOUTFIELDS(StringField5,  kundenname);
                WRITEKERNEL('red2',  'MeineDatei',  'Notiz-Zettel',  'Eine Rechnung',
                                'Ausgangsrechnung');
        }
}
'TypLieferschein':
{
        {
                /* Erkennungs-Sektion für Doc Typ TypLieferschein */
                MUSTMATCH('LIEFERSCHEIN');     /* nur bei LIEFERSCHEIN in der 1. Zeile */
        }
        {
                /* Aktionen für Doc Typ TypLieferschein */
                PRINT('Habe Lieferschein gefunden');
        }
}
'FangDenRest':
{
        {       /* Erkennungs-Sektion für alles, was vorher nicht erkannt wurde */
                all=CANMATCHMULTI('.*');       /* das gesamte Dokument in die Variable all*/
        }

        {               /* Aktionen für Doc Typ FangDenRest*/
                PRINT('********** unbekannt **********');
                PRINT(all);
        }
}
END
{
        PRINT('Habe ',  zahl, ' Rechnungen gefunden');
}
```

Anhang E Mehrere Dateien in ein Dokument einfügen

Beispiel: Mehrere Dateien in ein Dokument einfügen

Folgendes Beispiel zeigt die Anwendung der oben genannten Funktionen in der Praxis. In der zu vercoldenden Datei steht in jedem Dokument vom Typ Rechnung eine Zeile der Form:
PATH: <ein Dateipfad> z.B.: PATH: *.tif
In der Erkennungs-Sektion zum Dokumenttyp 'Rechnung' wird der Dateipfad in die Variable apath eingelesen:

```
CANMATCHMULTI('PATH:');
apath = CANMATCH('.*');
```

Mit

```
BINDOUTFIELDS(FilePath, apath );
```

wird das Schreiben aller, durch den Inhalt von apath beschriebenen Dateien durch das nächste WRITEKERNEL vorbereitet.
In der Initialsierungssektion wird die Variable **erfolg** beim Programmstart auf JA gesetzt.
Durch die Abfrage:

```
retval = GETKERNELERR();
IF( retval = '0' )
{
        erfolg = '0';
}
```

direkt nach dem WRITEKERNEL wird, sobald bei WRITEKERNEL ein Problem aufgekommen ist, **erfolg** auf NEIN gesetzt.
In der END-Sektion, die einmal am Ende des ColdScripts ausgeführt wird, wird **erfolg** ausgewertet:

```
END     {
        IF( retval = '1' )
        {
                dos_befehl = 'del ' | '*.*';
                SYSTEM( dos_befehl );
        }
}
```

Wenn wir immer erfolgreich waren, dann löschen wir alles im momentanen Verzeichnis. Dies geht von der Annahme aus, das ColdConv in der oben beschriebenen „Neuen Serverbetriebsart" gestartet wurde. Hier wird nämlich in jedes Verzeichnis „gewechselt" (cd). In der Praxis sollte etwas differenzierter gelöscht werden; etwa DEL *.tif und DEL *.txt. DEL *.* ist recht gefährlich!

```
_______
{

/* Globale Informationen */
        ENDOFDOCPATTERN = '\x0c';
        ENDOFLINE = '\x0d\x0a';                 /*ANSI CR, ANSII LF, Zeilenende*/
        {
                erfolg = '1';
        }
}

'Rechnung':
{
        {
        /* Erkennungs Sektion für Doc Typ Rechnung*/
```

```
                x = MUSTMATCHMULTI('\\* R E C H N U N G \\*');
                /* erkenne Documenttyp Rechnung */
                IGNORELINE(2);                                  /* 2 Zeilen runter */
                CANMATCH('Kunden-Nr.    :');            /* sync */
                kundennr = CANMATCH('[0-9]+');                  /* Eine oder beliebig viele
                                                                Ziffern */
                IGNORELINE(1);                                  /* 1 Zeile runter */
                CANMATCH('Rechnung-Nr.  :');
                rechnungsnr = CANMATCH('[0-9]+');       /* Eine oder beliebig
                                                                viele Ziffern */
                IGNORELINE(1);
                CANMATCH('Rechnung-Datum:');
                rechnungsdatum = CANMATCH('[0-9]+\\.[0-9]+\\.[0-9]+');
                                                                /* bel. Ziffern, Punkt, bel.
                                                                Ziffern, Punkt, bel. Ziffern */
                IGNORELINE(1);
                CANMATCHMULTI('PATH:');
                apath = CANMATCH('.*');
                CANMATCHMULTI('Rechnungsbetrag');
                                                                /* CANMATCHMULTI überliest
                                                                Positionsdaten */
                rechnungsbetrag = CANMATCH('[0-9]+[\\.]*[0-9]*,*[0-9]*');

        }

        {
        /* Aktionen für Doc Typ Rechnung*/
                PRINT('habe Rechnung erkannt!');
                PRINT( 'Dateipath       : ', apath );
                PRINT( 'Kundennummer            : ', kundennr );
                PRINT( 'Rechnungsnummer       : ', rechnungsnr  );
                PRINT( 'Rechnungs Datum         : ', rechnungsdatum );
                PRINT( 'Rechnungs Betrag: ', rechnungsbetrag );

                BINDOUTFIELDS( author, kundennr  );
                BINDOUTFIELDS( description, rechnungsnr );
                BINDOUTFIELDS( heading, rechnungsdatum );
                BINDOUTFIELDS( issue, rechnungsbetrag );

                BINDOUTFIELDS(FilePath, apath );
                WRITEKERNEL('pressedb','nofile','Word für Windows',
                                'test_it111','andere Zeitschriften');
                retval = GETKERNELERR();
                IF( retval = '0' )
                {
                        erfolg = '0';
                }
                PRINT('kernel sagt: ',  retval, '       path= ', apath);
                PRINT('-------------------------------------');
        }
}

'Alles':
/* catch all Dokumenttyp,greift wenn alle vorigen nicht passten */
{
        {
        /* Erkennungs Sektion für Doc Typ Alles */
                x = CANMATCHMULTI('.*');                        /* catch all, passt auf alles */
        }
        {
```

```
        /* Aktionen für Doc Typ Alles */
                PRINT('Dokument nicht erkannt!  ',  x);
        }
}
END     {
                PRINT('Am Ende...');
                IF( retval = '1' )
                {
                        dos_befehl = 'del ' | '*.*';
                        SYSTEM( dos_befehl );
                        PRINT( 'Dos Befehl= ', dos_befehl );
                }
        }
```

Anhang F HP - Sequenzen des TVG

Eingangs der Dokumentation wird erwähnt, dass der Import Server nur in einer Schriftart (Courier 10 CPI) Dokumente archivieren kann.
Generell ist diese Aussage richtig und sollte immer als grundlegender Maßstab bezüglich Verarbeitbarkeit und Darstellung des Cold's nach und in Proxess gelten. Jedoch ist es, aufgrund der Tatsache, dass der TVG-Viewer einige HP-Sequenzen versteht, möglich, diese Regel teilweise zu brechen.
Es sei jedoch davor gewarnt, da HP-Steuersequenzen im Cold die Standardparameter im TVG-Viewer für das aktuelle Dokument explizit übersteuern. Dies kann zu ungewollten und extrem hässlichen Folgen für das darzustellende Colddokument führen.

Der TVG-Viewer, der zum Diaclip von Cold und Hintergrundtif verwendet werden muss, ist nur massiv eingeschränkt als HP-kompatibel zu betrachten!

Im Folgenden sind die Sequenzen gelistet, die der TVG versteht:

{esc} ist das Escape,
{n} bezeichnet einen numerischen Parameter.

Manche Sequenzen "kennt" das Programm, ignoriert sie aber (== wegfiltern). Diese ignorierten Sequenzen sind mit einem "*" markiert.

{esc}(s{n}H	Schriftauswahl nach cpi
{esc}(s{n}V	Schriftauswahl nach Fonthöhe
{esc}(s{n}T	* Schriftauswahl nach Schriftbild
{esc}(s{n}P	* Proportionalschrift ein/aus
{esc}(s{n}S	Kursiv ein/aus
{esc}(s{n}B	Fett ein/aus
{esc}&d{n}@	Unterstreichen aus
{esc}&d{n}D	Unterstreichen ein
{esc}&k{n}H	* Zeichenabstand einstellen
{esc}&l{n}D	Zeilenabstand einstellen
{esc}&l{n}E	Oberen Rand einstellen
{esc}&l{n}F	* Seitenlänge einstellen

Die im Folgenden stehende Tabelle enthält für „Schriftauswahl nach CPI“ und „Schriftauswahl nach Fonthöhe“ einige Grundwerte.

	Normal	Blite	15 CPI	16.66 CPI	17 CPI	20 CPI
CPI:	10	12	15	16.66	17	20
Höhe:	12	10	8	7.2	7	6

Bitte beachten Sie, dass die gesetzten Steuersequenzen über ein FF (FormFeed) hinaus gültig bleiben! Es gibt keinen globalen Reset auf die Standarteinstellungen! Diese muss manuell hergestellt werden!

Sind in einem Colddokument z.B. Epsondruckersteuerzeichen vorhanden, so können diese per STRINGMAP in die oben beschriebenen Steuerzeichen Konvertiert oder weggefiltert werden. Somit ist die optische Erscheinung des Cold's in gewissen Grenzen veränderbar.

Dieser Eingriff in eine Colddatei sollte jedoch nach Möglichkeit vermieden werden. Er stellt eine extrem hohe Fehlerquelle bei Dokumentänderungen dar!