AMPL – Eine kurze Einführung

Sebastian Lohse, Maria Pilecka

März 2014

Gliederung

Historisches

- um 1985 entwickelt und implementiert, seither stetig weiterentwickelt
- algebraische, d.h. gewohnte Symbolik anwendbar, Modellierungssprache für Optimierungsprobleme
- bei Entwicklung an keine spezielle Abkürzung gedacht bzw. beabsichtigt
  - $\longrightarrow$ A Mathematical Programming Language
  - $\longrightarrow$ Algebraic Mathematical Programming Language
  - $\longrightarrow$ AT&T Mathematical Programming Language
- Solver aus anderen Programmpaketen importiert
- `www.ampl.com` - u.a. kostenlose Studentenversion von AMPL und einigen Solvern
- Im Computer-Pool: `Y:\f100\winsoftware\AMPL`
- `http://www.ccdss.org/` Download einer Benutzeroberfläche (VISAGE)

Grundlegende Syntaxregeln

- AMPL ist case-sensitive
- jede Anweisung beginnt mit einem Schlüsselwort und endet mit einem Semikolon
- jedes Element des Modells (inkl. Restriktionen und Zielfunktion) ist mit einem Namen zu definieren
- nach Namenszuweisung folgt ein Doppelpunkt
- Multiplikationspunkt muss mitgeschrieben werden
- bezüglich Rechenreihenfolge gelten die gewöhnlichen Regeln: Punkt vor Strich, Potenzen vor Punkt

Ein einfaches Beispiel

$$
\begin{array}{rcrcl}
3x_1 & + & x_2 & \longrightarrow & \max \\
x_1 & + & x_2 & \leq & 4 \\
-2x_1 & + & 3x_2 & \leq & 6 \\
 & & x_1, x_2 & \geq & 0
\end{array}
$$

$\Longrightarrow$

```
var x1;
var x2;
maximize ziel: 3*x1+x2;
subject to nb1: x1+x2 <= 4;
subject to nb2: -2*x1+3*x2 <= 6;
subject to nb3: x1 >= 0;
subject to nb4: x2 >= 0;
solve;
MINOS 5.5: optimal solution found.
1 iterations, objective 12
display x1,x2,ziel;
x1 = 4
x2 = 0
ziel = 12
```

Gliederung

Allgemeines Vorgehen

Eingabe von Problemen wie auf der Folie 5 möglich, allerdings für größere oder häufig zu lösende Problemstellungen mit wechselnden Daten eher ungeeignet. Daher können das Modell und die entsprechenden Daten aus *.txt (oder *.mod und *.dat) Dateien eingelesen werden.

AMPL orientiert sich somit am praktischen Vorgehen:

1. Modellformulierung (Mengen, Variablen, Parameter, Zielfunktion, Restriktionen) $\longrightarrow$ `model modelname.txt;`
2. Datenbeschaffung $\longrightarrow$ `data dataname.txt;`
3. Lösen $\longrightarrow$ `solve;`
4. Anzeige und Resultatsanalyse $\longrightarrow$ `display, let, delete ...;`

Bemerkung

Vor dem Aufruf eines Solvers (Minos,Cplex,...) wird ein Presolve-Verfahren auf das Modell angewandt (Variablenreduktion, redundante Restriktionen, Schranken) um es in eine für die Algorithmen passende Form zu bringen.

Komplexere Modelle

Modell besteht aus

- Variablen $\longrightarrow$ `var`
- Parametern $\longrightarrow$ `param`
- Mengen $\longrightarrow$ `set`
- Restriktionen $\longrightarrow$ `subject to`
- Zielfunktion(en) $\longrightarrow$ `minimize` oder `maximize`

Diese

- müssen sich im Namen voneinander und von den Schlüsselwörtern unterscheiden
- können in beliebiger Reihenfolge deklariert werden
- müssen deklariert sein, bevor sie benutzt werden können.

Syntaxregeln, Funktionen, Schlüsselwörter

Weitere Syntaxregeln

- Subskripts werden durch eckige Klammern ausgedrückt ($a_{ij} \widehat{=}$ `a[i,j]`)
- Indexmengen einer Summe oder Aussagen für alle Elemente einer Menge mittels `{i in N}`
- `#` für Kommentarzeile oder Umgebung `/* ... */`

Einige Funktionen

```
+ - / * ^
abs(x), acos(x), cos(x), cosh(x), exp(x), log(x),
log10(x), max(x,y,...), min(x,y,...), sqrt(x), ceil(x),
floor(x), a div b, a mod b, a less b, string1 & string2,
length(string)
```

Schlüsselwörter

```
if, then, else, repeat, until, Infinity, all, binary, by,
check, complements, default, dimen, exists, forall, in,
integer, objective, option, setof, sum, within,...
```

Transportproblem

$$\sum_{i=1}^{n}\sum_{j=1}^{m} x_{ij}\,c_{ij} \longrightarrow \min$$

$$\sum_{i=1}^{n} x_{ij} = b_j \quad j = 1,\ldots,m$$

$$\sum_{j=1}^{m} x_{ij} = a_i \quad i = 1,\ldots,n$$

$$0 \le x_{ij} \le s_{ij} \quad i = 1,\ldots,n \;\; j = 1,\ldots,m$$

Im Folgenden stellen Freiberg (FG) und Zwickau (Z) die Angebotsorte, sowie Dresden (DD), Leipzig (L) und Chemnitz (C) die Bedarfsorte dar. Für genaue Daten siehe spätere Folie.

Transportproblem - AMPL-Modell

```
# transport.txt
# kapazitiertes Transportproblem

set M; # Menge von Bedarfsorten
set N; # Menge von Produktionsorten

param a {i in N} >= 0; # Angebotsmengen
param b {j in M} >= 0; # Bedarfsmengen
param c {i in N, j in M}; # Kosten
param s {i in N, j in M}; # Schranken

var x {i in N, j in M} >= 0, <= s[i,j];

minimize kosten: sum {i in N, j in M} x[i,j]*c[i,j];
#sum {j in M, i in N} ... oder sum {i in N} sum {j in M}

subject to bedarf {j in M}: sum {i in N} x[i,j] = b[j];
subject to angebot {i in N}: sum {j in M} x[i,j] = a[i];
```

Mengenoperationen

- `A union B` $\widehat{=} A \cup B$
- `A inter B` $\widehat{=} A \cap B$
- `A diff B` $\widehat{=} A \backslash B$
- `A symdiff B` $\widehat{=} (A \backslash B) \cup (B \backslash A)$
- `A cross B`, oder äquivalent `{A,B}` $\widehat{=} A \times B$
- Durchschnitt mit höchster Priorität; alle anderen gleich, wobei Ausführung von links nach rechts erfolgt
- `card(M)` $\widehat{=} |M|$

Bemerkung

Operanden müssen Mengen sein, d.h. anstatt `A union 2000` müsste man `A union {2000}` schreiben.
Die leere Menge wird mit `{}` bezeichnet.

Ungeordnete Mengen

- oft bedeutungsvolle Zeichenketten, z.B. Bedarfsorte aber auch Zahlen, oder gemischt
- Zeichenketten werden beim Zugriff auf spezielle Elemente von ' oder " eingeschlossen, z.B. `display a['FG']` (im Dateneingabemodus nicht zwingend erforderlich)
- Kompaktzuweisung von Zahlen, funktioniert nicht im Dateneingabemodus!

```
set I = 1990 .. 2020 by 5;
# äquivalent: set I = 1990 .. 2023 by 5;

param start integer;
param end integer;
param interval integer;
set I = start .. end by interval;
```

- beispielsweise in Summenausdrücken `{t in 1 .. T}` verwendbar
- Subskripts wie `[t-1]` nur für Zahlenmengen verwendbar

Geordnete Mengen

Schlüsselwörter:
- `ordered`, z.B. `set name ordered;` - Eingabeordnung wird berücksichtigt
- `circular` - das erste Element folgt nach dem letzten Element

Anwendbare Funktionen:

- `first(name)`, `last(name)` sonst `member(t,name)`
- `prev(t,name)` und `next(t,name)` - direkter Vorgänger bzw. Nachfolger des `t`-ten Elements der Menge `name`
- `prev(t,name,n)` und `next(t,name,n)` - `n`-ter Vorgänger bzw. Nachfolger, wobei `n<0` möglich ist, falls `n` zu groß oder zu klein ist, liefert AMPL Fehlermeldung mit dem maximal möglichen Wert
- `prevw` und `nextw` - die Menge wird als `circular` aufgefasst
- `ord(t,name)` und `ord0(t,name)` liefern Index des Elements `t` wobei `ord0` im Falle der Nichtzugehörigkeit eine 0 liefert
- `A diff B` behält die Ordnung der Menge `A` bei, (Ordnung für `union`, `inter`, `symdiff` nicht definiert)

Beispiel

```
set N := 12 Jg ab 3 56 cd 1 4 rth 34 2 Ac;
```

$\longrightarrow$ aus Daten-File `mengenord.txt` einlesen, nach Definition von `N` und vor `display`

```
set N ordered by ASCII; data mengenord.txt; display N;
set N := 1 12 2 3 34 4 56 Ac Jg ab cd rth;
display last(N); last(N) = rth
display prev('Ac',N,3); prev('Ac', N, 3) = 34
display ord('Ac',N); ord('Ac', N) = 8
display member(8,N); member(8, N) = Ac

set N ordered by EBCDIC; data mengenord.txt; display N;
set N := ab cd rth Ac Jg 1 12 2 3 34 4 56;

set N ordered by reversed EBCDIC; data mengenord.txt;
display N;
set N := 56 4 34 3 2 12 1 Jg Ac rth cd ab;
```

within und setof

`within` ⟶ zur Definition von Teilmengen

Transportproblem mit fehlenden Kanten

```
set kanten within {N,M};
# (i,j) in kanten um auf Element zuzugreifen, k in kanten
# liefert Fehler!
subject to angebot {i in N}: sum {j in M: (i,j) in kanten}
    x[i,j] = a[i]; # oder kurz ohne j in M
```

`setof` ⟶ Mengen aus allgemeineren Datenstrukturen gewinnen

Multicommodity-Problem - Daten sind durch Liefermöglichkeiten in Abhängigkeit vom Produkt gegeben, d.h. Tabelle `liefer` mit `(i,j,p)`, Konstruktion der Mengen `produkte` und `kanten`

```
set liefer dimen 3;
set produkte = setof {(i,j,p) in liefer} p;
set kanten = setof {(i,j,p) in liefer} (i,j);
# doppelte Elemente werden gestrichen
```

Intervalle

```
interval [a,b]
interval (a,b)
# oder auch gemischt
# unendlich = Infinity

integer [a,b]
# in Ausdrücken wie in oder within kann das Wort interval
# entfallen
```

Weiteres Beispiel

```
set N:= integer[3.5,5.8];
display N;
Error executing display command:
error processing set N:
attempt to iterate over an interval.
```

Parameter

- Deklaration meistens mit Indizierung unter Verwendung von Mengen, Eingabe von Schranken oder Intervalle möglich

```
param a {N} in (0,Infinity) integer;
```

- auch aus bisher bekannten Parametern berechenbar, dann allerdings keine Wertzuweisung in der Datendatei erlaubt; Ausweg: default-Werte

```
param a {i in N};
param test default sum {i in N} a[i];
```

- nach bestimmten Wahrscheinlichkeitsverteilungen „zufällig“ generierbar

```
param mittel;    param varianz;
param par = Normal(mittel,varianz);

data;
param mittel := 0; param varianz := 1;
display par; par = -0.529922
```

andere mögliche Verteilungen: `Uniform(m,n)`, `Expotential()`, `Beta(a,b)`, `Poisson(`μ`)`

Variablen

- Variablendeklaration analog wie bei Parameter, kann auch Schranken (bspw. die Nichtnegativitätsbedingungen) enthalten z.B.

```
var x {i in M, j in N} >= 0, <= s[i,j] integer;
```

- strikte Schranken wie beispielsweise `> 0` sind nicht zugelassen
- eine Variable kann auch mithilfe anderer Variablen Parameter oder Mengen deklariert werden

```
set prod; # Produkte
param prodcost {prod} >= 0; # Produktionskosten
var amount {prod} >= 0; # Produktionsmenge
var cost = sum {p in prod} prodcost[p]*amount[p];
```

- `var x := Startwert` oder `default` - ermöglicht die Eingabe von einem Startwert ($\Rightarrow$ nichtlineare Aufgaben), der später durch die optimale Lösung ersetzt wird

Zielfunktion und Nebenbedingungen

- Zielfunktion und Nebenbedingungen können unter Verwendung der Funktionen (s. Folie 9) deklariert werden
- doppelte Ungleichungen sind zugelassen, die obere und untere Schranke darf jedoch keine Variablen enthalten
- strikte Ungleichungen können nicht verwendet werden
- `subject to Time {if a > 0}: sum {i in M} x[i] <= a;` wird nur dann berücksichtigt, wenn die Bedingung $a > 0$ erfüllt ist

Gliederung

Indizierung

- einfache Indizierung ohne oder mit der Dummy-Variable möglich:

```
param a {M};
param a {i in M};
```

- Dummy-Variable muss eingeführt werden, falls in der Deklaration ein konkretes Element der Menge benutzt wird z.B.

```
param a {i in M} <= b[i];
```

- mit Dummy-Variable ist es möglich eine ganze Reihe von Nebenbedingungen in einem Ausdruck zusammenzufassen (mathematisch: $\forall j \in M$)

```
subject to bed {j in M}: sum {i in N} x[i,j] = b[j];
```

- zusätzliche Bedingungen an die Indexmenge möglich

```
sum {i in N: a[i] > 0} x[i,j] = b[j];
```

Weiterführende Konzepte

- `>= 0` oder `in (2,5) integer` bei Parameterdeklaration führt beim Einlesen der Daten automatisch zu einer Gültigkeitsprüfung
- Lösbarkeitsbedingungen ebenfalls in Modell integrierbar:

```
check: sum {i in N} a[i] = sum {j in M} b[j];
check {j in M}: b[j] <= sum {i in N} s[i,j];
```

  Fehlermeldungen beim Nichterfülltsein

```
check at line 13 of transport.txt fails:
check : sum{i in N} a[i] == sum{j in M} b[j];

check['DD'] at line 14 of transport.txt fails:
check{j in M} : b[j] <= sum{i in N} s[i,j];
```

- Indizierung auch problemlos mit mehr als zwei Subskripts möglich, z.B. mehrere Produkte

Transportproblem mit mehreren Produkten

```
# prodtransport.txt

set M; # Menge von Bedarfsorten
set N; # Menge von Produktionsorten
set P; # Menge von Produkten

param a {N,P} >= 0; # Angebotsmengen
param b {M,P} >= 0; # Bedarfsmengen
param c {N,M} >= 0; # Kosten
param s {N,M} >= 0; # Schranken

var x {N,M,P} >= 0;

minimize kosten: sum {i in N, j in M, p in P} x[i,j,p]*c[i,j];

subject to bedarf {j in M, p in P}: sum {i in N} x[i,j,p] = b[j,p];
subject to angebot {i in N, p in P}: sum {j in M} x[i,j,p] = a[i,p];
subject to kap {i in N, j in M}: sum {p in P} x[i,j,p] <= s[i,j];
```

Netzwerkprobleme

Für diese sind spezielle Algorithmen vorhanden, allerdings erfordert dies Modellformulierung mit den Schlüsselwörtern `node, arc, from, to, obj, net_in, net_out`.

Transportmodell

```
set N;
set M;
param angebot {N union M};
param bedarf {N union M};
param c {N,M};
param s {N,M};

node knoten {k in (N union M)}: net_in = angebot[k] - bedarf[k];
arc kanten{(i,j) in (N cross M)} >= 0, <= s[i,j], from knoten[i],
to knoten[j], obj kosten c[i,j];

minimize kosten;
```

Stückweise lineare Zielfunktionen

- werden durch Angabe der Anstiege sowie der Punkte an denen die Funktion nicht differenzierbar ist definiert (Knickpunkte)
- zur eindeutigen Festlegung der Funktion wird der Stelle 0 der Funktionswert 0 zugewiesen
- alternativ auch andere Stelle mit Funktionswert 0 angebbar

```
param n;
param Anstiege {i in 1..n};
param Punkte {i in 1..(n-1)};
var x;

minimize kosten: <<{i in 1..(n-1)} Punkte[i]; {i in 1..n}
Anstiege[i]>> x;
# allgemeiner (x,Nullstelle) anstelle von x
```

Nichtlineare Modelle

- mit gegebenen Funktionen leicht formulierbar
- berechnet werden lokale Optima bzw. nur stationäre Punkte
- eventuell mit verschiedenen Startwerten mehrmals rechnen
- Beispiel: `commands nichtlinear.txt`

Ganzzahlige Probleme

Für Integer Programming oder Mixed Integer Programming Probleme einfach Schlüsselwort `integer` oder `binary` bei der Variablendeklaration hinzufügen.

- Schichtplan (siehe Files)
- Rucksackproblem

```
set N;
var x {N} binary;
param c {N} >= 0;
param a {N} >= 0;
param b >= 0;
maximize wert: sum {i in N} c[i]*x[i];
subject to kap: sum {i in N} a[i]*x[i] <= b;
```

Gliederung

Dateneingabe

entweder AMPL mittels

```
data;
```

in Dateneingabemodus versetzen und Daten per Hand eingeben (endet mit einem Befehl welcher nicht zur Dateneingabe passt, wie `display` oder `model`)

oder einlesen aus einer Datei mittels

```
data dateiname.dat;
data dateiname.txt;
```

Beispieldaten: Transportproblem

		FG		Z	
	a_i	30		20	
	b_j	c_{ij}	s_{ij}	c_{ij}	s_{ij}
DD	20	10	15	13	10
L	15	8	10	9	10
C	15	10	10	7	10

Dateneingabe - Listenform

```
set N:= FG Z;
param a:= FG 30 Z 20;
# oder: die Menge und Parameter können gleichzeitig
# spezifiziert werden
param: N: a:= FG 30 Z 20;
# weitere Leerzeichen/-zeilen zwischen eingegebenen Daten
# spielen keine Rolle
set kanten:= (FG,DD) (Z,DD) ... ;
set kanten:= FG DD Z DD ...;
set kanten:= (*,DD) FG Z ...;
param c:= FG DD 10 Z DD 13 ...;
param c:= [*,DD] FG 10 Z 13 ...;
# mehrere Produkte
set liefer:= (FG,DD,P1) (FG,DD,P2) ...;
set liefer:= FG DD P1 FG DD P2 ...;
set liefer:= (*,DD,P1) FG Z (*,DD,P2) FG Z...;
set liefer:= (*,DD,*) FG P1 FG P2 FG P3 Z P1 Z P2 Z P3...;
```

Dateneingabe - Tabellenform

```
param a: P1   P2   P3:=
      FG 30   25   20
      Z  20   30   25;

# Im Falle unterschiedlicher Kosten für die Produkte

param c:=
      [*,*,P1]:     DD   L   C:=
                FG  10   8   10
                Z   13   10  7
      [*,*,P2]:     DD   L   C:=
                FG  15   11  15
                Z   18   14  9;
```

Dateneingabe mit '+', '-' und '.'

Zugehörigkeit einer Kante zur Menge `kanten` für den Fall, dass `kanten` nicht das Kreuzprodukt von `N` und `M` ist:

```
set kanten: DD  L  C:=
        FG  +   -  +
        Z   -   +  +;
```

Definition der Kosten zu dieser Menge

```
param c: DD  L  C:=
      FG 15  .  10
      Z  .   10 7;
```

Dateneingabe - Bemerkungen

- alle deklarierten Parameter müssen einen Wert zugewiesen bekommen
- bei Wiederholung des gleichen Wertes:
  `param cost default 100:= ...;`
  so werden alle in der Liste bzw. Tabelle fehlenden Werte des Parameters auf 100 gesetzt
- lesen aus Dateien ebenfalls möglich, z.B. sei Inhalt der Datei 'name.txt' (unformatiert, ohne Namen, Kommas oder ':='):

```
4 40 40 32 40
```

```
read T, {t in 1..T} a[t] < name.txt;
```

  Schreiben mit `print ... > dateiname.txt;`
- Lesen von Daten aus Datenbanken oder Excel ebenso wie das Zurückschreiben möglich

Gliederung

display

- `display varname;` $\longrightarrow$ Variablenwert
- `display varname.lb, varname.ub;` $\longrightarrow$ untere und obere Schranken
- `display varname.rc;` $\longrightarrow$ Optimalitätsindikatoren (reduzierte Kosten), wobei die Definition der Aufgabe wesentlich ist - vgl. `einfach.txt` und `einfach2.txt` für x_2
- Berechnungen mittels Variablen- bzw. Parameterwerte
- `display unglname;` $\longrightarrow$ Schattenpreis (duale Variable)
- `display unglname.lb, unglname.body` (Summe aller Terme, die Variablen beinhalten), `unglname.ub;`
- `display name.slack;` $\longrightarrow$ Abstand zur näheren Schranke einer Variablen oder Ungleichung
- nicht möglich: Simplextableau
- `display name > dateiname.endung;` legt 'dateiname.endung' an oder überschreibt es und schreibt `name = wert` hinein

display - Einführungsbeispiel

```
display x1, x2, ziel2;
x1 = 4
x2 = 0
ziel = 12
display x1.rc, x2.rc;
x1.rc = 0
x2.rc = 0
display nb1, nb2, nb3, nb4, nb5;
nb1 = 3
nb2 = 0
nb3 = 0
nb4 = 0
nb5 = -2

display x1, x2, ziel2 > losung.txt;
display 3*x1+x2 > losung.txt;
display x1.rc, x2.rc > losung.txt;
display nb1, nb2, nb3, nb4, nb5 > losung.txt;
```

display - losung.txt

x1 = 4
x2 = 0
ziel = 12

3*x1 + x2 = 12

x1.rc = 0
x2.rc = 0

nb1 = 3
nb2 = 0
nb3 = 0
nb4 = 0
nb5 = -2

show und xref

`show` - gibt die Liste aller Komponenten des aktuellen Modells aus.

Einschränkung auf konkrete Komponenten des Modells:

- `show vars;`
- `show obj;`
- `show constr;`
- `show checks;`
- `show nbname;`
- `show objname;`
- `show varname, parname;`
- `show check 2;`

`xref varname, parname, setname;` - gibt die Liste aller Komponenten aus, die von der gegebenen Variable, Parameter etc. abhängen

show - Ergebnisse für Transportmodell

```
show vars,obj,constr,checks,angebot,kosten,x,a,check 2;
variable: x
objective: kosten
constraints: angebot bedarf
checks: 3, called check 1, check 2, etc.
subject to angebot{i in N} : sum{j in M} x[i,j] == a[i];
minimize kosten: sum{i in N, j in M} x[i,j]*c[i,j];
var x{i in N, j in M} >= 0
     <= s[i,j];
param a{i in N} integer in interval(0, Infinity);
check{i in N} : a[i] <= sum{j in M} s[i,j];
```

xref - Ergebnisse für Transportmodell

```
xref x, a, N;
# 4 entities depend on x:
Initial      kosten
bedarf      angebot
# 3 entities depend on a:
check 1      check 2
angebot
# 10 entities depend on N:
a     c
s     x
check 1      check 2
check 3      kosten
bedarf      angebot
```

expand

`expand nbname, obj;` - detaillierte Ausgabe einer Restriktion oder Zielfunktion

```
expand kosten, bedarf['DD'];
minimize kosten:
    10*x['FG','DD'] + 8*x['FG','L'] + 10*x['FG','C'] +
13*x['Z','DD'] + 10*x['Z','L'] + 7*x['Z','C'];

subject to bedarf['DD']:
    x['FG','DD'] + x['Z','DD'] = 20;
```

`expand varname;` - Ausgabe aller Koeffizienten der Variable im Modell

```
expand x['FG','DD'];
Coefficients of x['FG','DD']:
    bedarf['DD'] 1
    angebot['FG'] 1
    kosten 10
```

Gliederung

Datenmodifikation

- `reset data;` - alle Daten werden gelöscht
- `reset data setname, parname;` - Elemente einer Menge oder Parameter werden gelöscht
- `update data setname, parname;` - die neuen Werte ersetzen die alten Werte, nicht veränderte Werte bleiben erhalten
- `let parname := wert;` - ändert den Wert eines Parameters, eignet sich für Änderungen einzelner Werte
- `let {i in M} a[i] := 1.1*a[i];` - neue Werte können aus den bisherigen berechnet werden

Modellmodifikation

- `delete nbname, obj;` - löscht eine Restriktion oder Zielfunktion
- `purge varname, parname, setname;` - löscht dieses Element und alles was mit ihm in Verbindung steht (einfaches Löschen von Parametern, Variablen oder Mengen mithilfe '`delete`' nich möglich)
- `redeclare keyword name redeclaration;` - ersetzt bisherige Deklaration der Menge, Parameter usw., darf jedoch i. A. den Charakter des Modellelements nicht verändern
- `drop nbname, objectivename;` - Restriktion oder Zielfunktion wird ausgeblendet
- `restore nbname, objectivename;` - Wiederherstellung der Restriktion oder Zielfunktion nach '`drop`'
- `fix varname[ := wert];` - die Variable wird auf den aktuellen Wert (oder gegebenen nach ':=') fixiert
- `unfix varname;` - Variable kann wieder überschrieben werden
- `reset;` - Modell und Daten werden gelöscht

Analyse - Beispiele anhand des Transportmodells

```
delete M;
Error executing "delete" command:
    Cannot delete M: it has dependents.
    use xref to see dependents, and
    use purge to delete them.

display x; # Optimallösung
x := FG C 5  FG DD 15  FG L 10
    Z C 10   Z DD 5  Z L 5;
redeclare var x {i in N, j in M} >= 6, <= s[i,j];
solve; display x;
x := FG C 7  FG DD 14  FG L 9
    Z C 8  Z DD 6  Z L 6;
```

Gliederung

Optionen

- sind mehrere Zielfunktionen in einem Modell vorhanden - Auswahl mittels `objective name;`
- `option;` - gibt alle Optionen mit den aktuellen Werten aus
- `reset options;` - die Veränderungen in den Optionen werden zurückgesetzt

Beispiele der Optionen:

- `option solver name;` - Änderung des Solvers
- `option show_stats 1;` - eine Statistik über die Größe des Modells wird ausgegeben
- `option randseed 0;` - die zufälligen Werte (vgl. Folie 19) wiederholen sich nicht

Optionen

Beispiele der Optionen:

- `option display_round 2;` - Runden von Werten
- `option display_width 60;` - Anzeigeoptionen
- `option relax_integrality 1;` - Ganzzahligkeitsforderungen aller Variablen werden ignoriert
- `let varname[index].relax = 1;` - die Variable wird nicht mehr als ganzzahlig betrachtet
- `option cplex_options 'relax';` - erst Solver betrachtet die Variablen als nicht ganzzahlig

Bemerkung

`option relax_integrality 1; solve;` und `option cplex_options 'relax'; solve;` führen im Allgemeinen nicht zum selben Ergebnis!

Gliederung

Script-Files

- `commands name.*;` $\longrightarrow$ führt Befehle der Datei aus
- `include name.*;` $\longrightarrow$ Inhalt der Datei eingefügt
- `if` Bedingung `then` Anweisung `else` Anweisung
- `for {...} { Anweisung;}`
- `repeat while` Bedingung `{...};`
- `repeat {...} while` Bedingung`;`
- `repeat until` Bedingung `{...};`
- `repeat {...} until` Bedingung`;`
- `break` und `continue`

Script-Files - Einfache Beispiele

- ```
  model modname.txt;
  data datenname.txt;
  for {1..4} {
      solve;
      display varname;
      let parname := parname + 2;}
  ```
- ```
  set parschleife := parname .. parname+6 by 2;
  for {t in parschleife}{
      let parname := t;
      solve; ...}
  ```
- ```
  model transport.txt;
  for {j in 1..3}{
      reset data;
      data("daten"& j & ".txt");
      solve; display x;}
  ```
  ```

Literatur

Fourer, R., Gay, D.M. and Kernighan, B.W. (2003): *AMPL: A Modeling Language for Mathematical Programming*, Thomson Brooks/Cole.