Einführung in Sage - Einheit 9

Strings, interaktive Grafiken, Wärmeleitungsgleichung, Sage-Code

Jochen Schulz

Georg-August Universität Göttingen

Aufbau

1. **Umgang mit Strings**

2. **Programmierung**
   - Schleifen

Klausur

- Zeit: 01.03.2010 von 10:00 --- 12:00
- Ort: HS1 (A bis J) und AudiMax (K bis Z)
- Hilfsmittlel: Papier, Schreibgerät(e) und Unterlagen in Papierform
- Studenten-Ausweis mitbringen

Aufbau

1. **Umgang mit Strings**

2. **Programmierung**
   - Schleifen

Strings

- Zeichenketten (engl. strings) sind eine geordnete Aneinanderreihung von Zeichen. Zeichen sind z.B. Buchstaben, Ziffern, Sonderzeichen,...
- Mit ihnen kann man in MuPAD Texte gestalten. Sie sind wichtig für die Ausgabe der Ergebnisse.
- Sie haben den Datentyp `DOM_STRING`.
- Sie werden innerhalb der Begrenzer `"` angegeben.

Beispiele für Strings

```
>> text1:="Dies ist ein String."

   "Dies ist ein String."

>> text2:="Dies ist noch ein String."

   "Dies ist noch ein String."

>> domtype(text1)

   DOM_STRING
```

Zugriff

Mit dem Indexoperator [ ] können einzelne Zeichen einer Zeichenkette extrahiert werden oder auch geändert werden.

```
>> text1[1], text1[4], text1[5]

   "D", "s", " "
>> text1[4]:="i": text1

   "Diei ist ein String."
```

Die extrahierten Teile sind wieder Strings.
Vorsicht: In älteren Versionen von MuPAD beginnt die Indizierung bei 0.

Operationen für Strings I

- Zusammenhängen von Strings

```
> A:="Letzte ": B:="Vorlesung": A.B

  "Letzte Vorlesung"
```

- Ausgabe von Zeichenketten

```
>> print(Unquoted,A.B)

  Letzte Vorlesung
```

- Strings können nicht addiert oder multipliziert werden.

```
>> A+B

  Error: Illegal operand [_plus]
```

Der .-Operator

`obj1`	`obj2`	`obj1.obj2`
Zeichenkette	Zeichenkette	Zeichenkette
Zeichenkette	Bezeichner	Zeichenkette
Zeichenkette	ganze Zahl	Zeichenkette
Zeichenkette	Ausdruck	Zeichenkette
Bezeichner	Zeichenkette	Bezeichner
Bezeichner	Bezeichner	Bezeichner
Bezeichner	ganze Zahl	Bezeichner
Bezeichner	Ausdruck	Bezeichner
Liste	Liste	Liste

Operationen für Strings II

- `length` gibt die Anzahl der Zeichen in einer Zeichenkette an.

```
> a:=length(A.B);

  16

>> (A.B)[a]

  "g"
```

- Beliebige MuPAD-Objekte können durch `expr2text` in einen String verwandelt werden.

```
>> expr2text(x^2+2), expr2text([1,2,3])

  "x^2 + 2", "[1, 2, 3]"
```

Operationen für Strings III

- Nachfolgend ein Beispiel zur Maskierung von Sonderzeichen.

```
>> a:=expr2text("hallo"), a[1]

  "\"hallo\""  ,  "\""
```

- Weitere Befehle zur Manipulation von Strings findet man in der Bibliothek `stringlib`.

Anwendungen

- Geben Sie alle Bezeichner in alphabetischer Reihenfolge an! Erstellen Sie eine Liste aller Bezeichner!

```
>> A:=map(anames(All),expr2text)
>> L:=[op(A)]
>> sort(L)
```

- Schreiben Sie eine Funktion, die eine Zeichenkette rückwärts berechnet.

```
>> revers:= A->_concat(A[length(A)-i+1]
   $\text{dollar}$ i=1..length(A))

>> revers("Hallo")

  "ollaH"
```

Aufbau

1. Umgang mit Strings

2. **Programmierung**
   - Schleifen

Größter gemeinsamer Teiler (ggT)

Berechnung des ggT von natürlichen Zahlen a und b mit Hilfe des euklidischen Algorithmus.

Idee: Es gilt:

1. $ggT(a, b) = ggT(a, b - a)$ für $a < b$.
2. $ggT(a, b) = ggT(b, a)$.
3. $ggT(a, a) = a$.

Algorithmus:
Wiederhole, bis $a = b$

- Ist $a > b$, so $a = a - b$.
- Ist $a < b$, so $b = b - a$

Implementierung

```
ggT:= proc(a,b)
/* Bestimme den ggT von a und b */
begin
  while (a<>b) do
    if (a>b)
          then a:=a-b;
          else b:=b-a;
    end_if;
  end_while;
  return(a);
end_proc:
```

Logische Ausdrücke

- Es gibt die drei logische (Boolsche) Werte `TRUE`, `FALSE` und `UNKNOWN`.
- Sie haben den Datentyp `DOM_BOOL`.
- Gleichungen $(=)$, Ungleichungen $(<>)$ oder Größenvergleiche (`<`,`>`,`<=`,`>=`) können durch `bool` zu `TRUE` oder `FALSE` ausgewertet werden.

Logische Ausdrücke

```
>> bool(3<=4)

   TRUE

>> bool(TRUE=TRUE)

   TRUE

>> bool("AS"<="B")

   TRUE

>> bool(TRUE<>UNKNOWN)

   TRUE
```

Aufbau

1. Umgang mit Strings

2. **Programmierung**
   - Schleifen

Repeat

Neben `for` ist durch `repeat` eine weitere Schleifenvariante gegeben:

```
x:=1.1:
repeat
  i:=x; x:=i^2; print(i,x)
until x>100 end_repeat:
```

Die Befehle zwischen `repeat` und `until` werden so lange wiederholt, bis die Bedingung (hier $x > 100$) wahr wird.

while

So ähnlich wie die `repeat`-Schleife funktioniert die `while`-Schleife.

```
x:=2:
while x<=100 do
 i:=x; x:=i^2; print(i,x)
end_while:
```

Die Befehle zwischen `while` und `end_while` werden so lange wiederholt, wie die Bedingung (hier $x <= 100$) wahr ist.

Verzweigung

- Ein wichtiges Werkzeug jeder Programmiersprache sind Verzweigungen.
- Je nach Wert oder Bedeutung von Variablen werden unterschiedliche Befehle ausgeführt.
- In MuPAD gibt es das `if`-Konstrukt und das `case`-Konstrukt.

Beispiel

```
for i from 2 to 100 do
 if isprime(i)
   then print(i,"ist Primzahl")
   else print(i,"ist keine Primzahl")
 end_if
end_for:
```

if-Konstrukt

Die Verzweigung `if` hat die folgende Struktur:

```
if Bedingung
    then Befehle1
    else Befehle2
end_if
```

Ist die `Bedingung` wahr, so wird `Befehle1` ausgeführt, ansonsten `Befehle2`. Befehle in den Befehlsfolgen sind durch `:` oder `;` zu trennen. Der `else` Aufruf ist optional.

Berechnung von Primzahlzwillingen

```
T:=[]: anz:=0:
for i from 2 to 10000 do
 if (isprime(i) and isprime(i+2))
      then anz:=anz+1;
      T:=T.[[i,i+2]];
 end_if:
end_for:
print("Anzahl = ",anz);
```

Case

Hat man eine Verzweigung mit mehreren Alternativen, so kann man entweder geschachtelte `if` Konstrukte verwenden, oder das Konstrukt `case` verwenden.

```
case var
 of wert1 do ...
 of wert2 do ...
    ...
 otherwise
    ...
end_case
```

`Case` funktioniert wie die `switch` Anweisung in C.

Erklärung

- Ist keiner der `of`-Zweige richtig, so wird die Befehlssequenz zwischen `otherwise` und `end_case` ausgeführt.
- Durch den Befehl `break` kann ein vorzeitiges Verlassen der `case`-Anweisung bewirkt werden.
- Stimmt der Wert von `var` mit einem der Werte `wert1`, `wert2`, ..., überein, werden von dieser Stelle an alle nachfolgenden Befehle ausgeführt, insbesondere also auch von allen anderen `of` und `otherwise` Sequenzen). Sogenanntes *'fall-through'*.

Beispiel: Betrag

```
betrag:=proc(a)
      begin
        case(domtype(a))
          of DOM_INT do
          of DOM_RAT do
          of DOM_FLOAT do
            if a>0 then y:=a: else y:=-a: end_if:
            break;
          of DOM_COMPLEX do
            y:=sqrt(Re(a)^2+Im(a)^2);
            break;
          otherwise
            print("Falscher Eingabetyp");
          end_case:
          return(y);
      end_proc:
```

Gültigkeit von Variablen I

- Mit der Gültigkeit von Variablen ist die Bestandsdauer von Variablen bzw. der Werten dieser Variablen gemeint.
- Beim interaktiven Gebrauch von MuPAD sind alle Variablen *global*, d.h. die den Variablen zugewiesenen Werte bleiben für die gesamte Laufzeit von MuPAD erhalten bis sie geändert werden. Man kann auf die Variablen jederzeit zugreifen und die Werte der Variablen ändern.
- Daneben gibt es aber auch *lokale Variablen*, die nur innerhalb einer Prozedur gültig sind. Nach Beenden der Prozedur werden diese Variablen wieder gelöscht.

Gültigkeit von Variablen II

- In MuPAD sind in Prozeduren definierte Variablen standardmäßig global. Die interaktiv erstellten Variablen sind sowieso global.
- Mit dem Schlüsselwort `local` können in Prozeduren lokale Variablen erzeugt werden. `local` steht zwischen `proc()` und `begin`.
- Die lokalen Bezeichner sind vom Typ `DOM_VAR` (nicht wie erwartet `DOM_IDENT`).
- Die Input-Argumente von Prozeduren sind lokale Variablen.

Goldene Regel

Die Verwendung von globalen Variablen in Prozeduren ist schlechter Programmierstil und sollte vermieden werden.

Beispiele

Ein Beispiel mit globalen Variablen

```
>> a:=b:
>> f:=proc()
      begin
        a:=1+a^2:
      end_proc:
>> f(); f(); f()
```

Ein Beispiel mit lokalen Variablen

```
>> a:=b:
>> f:=proc()
      local a;
        begin a:=2
      end_proc:
>> f(); a
```

Rekursives Beispiel

Berechnung der Fakultät einer natürlichen Zahl

```
/* Berechnung der Fakultaet */
fakultaet := proc(n) begin
        if domtype(n)=DOM_INT and n>0
            then
               if n=1 then return(1)
               else n*fakultaet(n-1)
               end_if
            else
               print("Falscher Datentyp");
            end_if
end_proc:
```

Symbolische Rückgabe

```
>> fakultaet(9)

  362880

>> x!

  fact(x)

>> fakultaet(x)

  "Falscher Datentyp"
```

- Viele Systemfunktionen wie `fact`, geben den Prozeduraufruf symbolisch zurück, wenn er nicht auszuwerten ist.
- Die Auswertung kann dann später erfolgen.

Fakultät II

```
/* Berechnung der Fakultaet */
fakultaet2 := proc(n)
  begin
       if testtype(n,Type::PosInt)
           then
              if n=1 then return(1)
              else n*fakultaet2(n-1)
              end_if
            else
              return(procname(args()));
            end_if
  end_proc:
```

Erklärungen

- Durch `args()` erhält man die Folge der Argumente.
- `args(0)` ist die Anzahl der Argumente.
- Durch `args(i)` erhält man das i-te Element.
- Mit diesen Befehlen kann man Prozeduren mit einer beliebigen Anzahl von Argumenten programmieren.
- `procname` ist der Name der Prozedur.

Testen des Typs

Durch den Aufruf

```
testtype(Objekt, Typenbezeichner)
```

wird getestet, ob ein `Objekt` dem Typenbezeichner entspricht.
Rückgabewert ist `TRUE` oder `FALSE`.

- Prinzipiell kann man auch `domtype` zum Überprüfen des Typs benutzen.
- Die Typenbezeichner sind aber differenzierter.
- Übersicht der verfügbaren Typenbezeichner erhält man durch `? Type`.

Beispiele

```
>> testtype(sqrt(3),Type::Real)

  FALSE

>> testtype(float(sqrt(3)),Type::Real)

  TRUE

>> testtype(3,Type::Real)

  TRUE

>> select([i $\text{dollar}$ i=100..120],testtype,Type::
    Prime)

  [101, 103, 107, 109, 113]
```

Mandelbrot-Menge

Die Mandelbrot-Menge ist die Menge von Punkten $c \in \mathbb{C}$ bei denen die Folge $(z_n)_n$, die durch

$$z_0 := c, \qquad z_{n+1} = z_n^2 + c, \quad n \in \mathbb{N}$$

definiert ist, beschränkt ist.

Programm - Mandelbrot

```
mandel:=proc(x,y)
  local it,a0,a,MAX_IT;
  begin
    if not (testtype(x,Type::Real) and
            testtype(x,Type::Real))
     then procname(x,y)
    else
      MAX_IT := 150;
      it := 0;
      a0 := x + I*y;
      a := a0;
      while (abs(a)<2 and it<MAX_IT) do
        a := a^2 + a0;
        it := it + 1;
      end_while;
      return(float(it/MAX_IT));
    end_if;
  end_proc:
```

Plot - Mandelbrot

Die Funktion `mandel` gibt zu $x + iy$ die relative Anzahl der Iterationsschritte zurück.
Geplottet wird die Funktion nun wie folgt:

```
PlotteMandel:=proc()
  begin
    mandelPlot:=plot::Function3d(mandel(x,y),
      x=-2.1..1.2, y=-1.1..1.1, Mesh=[100,100]);
    plot(mandelPlot,Width=20*unit::cm,
      Height=15*unit::cm);
  end_proc:
```

Programmierregeln

- Falls möglich, nur lokale Variablen benutzen.
- Programme vollständig kommentieren. Das heißt zum einen das eine Kommentarzeile zu Beginn steht, was das Programm macht und wieviele und welche Eingabeparameter es erhalten darf und was die Prozedur zurückgibt. Zusätzlich sollten auch alle wesentlichen Operationen kommentiert werden.
- Werte explizit mit `return()` zürückgeben.
- Programme übersichtlich gestalten, z.B. Schleifen oder Verzweigungen einrücken.
- Keine Umlaute in Programmkommentaren verwenden.

Letztes Beispiel I

```
Gadisch:=proc(x,basis)
  /*-------------------------------------
   Berechnung der  Darstellung
   einer natuerlichen Zahl x zur Basis b
   Rueckgabe des Ergebnis als Liste!
  -------------------------------------*/
  local T,T_r,i;  /* lokale Variablen*/
  begin /* Beginn lokale Prozedur */
  /* Abfangen der Eingabe */
    if not testtype(x,Type::PosInt)
       then return(procname(args()));
    end_if;
    if (not testtype(basis,Type::PosInt)) or basis=1
       then return(procname(args()));
    end_if;
```

Letztes Beispiel II

```
  T:=[];  /* leere Liste */
  /* Beginn Schleife */
  while x>0 do
    T:=[x mod basis].T;
    print(Unquoted,expr2text(x)." : "
      .expr2text(basis)." = "
      .expr2text(x div basis)." Rest "
      .expr2text(x mod basis));
    x:=(x div basis);
  end_while;
  /* Rueckgabe der Liste */
  return(T);
end_proc:
```

Allerletztes Beispiel: Kochsche Kurven I

- Seien y_1, y_2 zwei Punkte im $\mathbb{R}^2$.
- Betrachte die Strecke mit Endpunkten y_1 und y_2.
- Ersetze diese Strecke durch 4 Strecken $\overline{y_1 z_1}$, $\overline{z_1 z_2}$, $\overline{z_2 z_3}$, $\overline{z_3 y_2}$ mit Endpunkten

$$
\begin{aligned}
z_1 &= \tfrac{2}{3} y_1 + \tfrac{1}{3} y_2 \\
z_2 &= \tfrac{\sqrt{3}}{6} \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} (y_1 - y_2) + \tfrac{1}{2}(y_1 + y_2) \\
z_3 &= \tfrac{1}{3} y_1 + \tfrac{2}{3} y_2
\end{aligned}
$$

- Dieses Prozedere wird nun für jede einzelne Teilstrecke wiederholt.

Allerletztes Beispiel II

```
koch:=proc(y1,y2,lev)
  local z1,z2,z3;
  begin
    if (lev = 0)
      then Listelinien:=append(Listelinien,
        plot::Line2d([y1[1],y1[2]],[y2[1],y2[2]]));
    else
      /* Definieren der neuen Punkte */
      z1 := 2/3 * y1 + 1/3 * y2;
      z3 := 1/3 * y1 + 2/3 * y2;
      z2 := sqrt(3)/6*matrix([[0, 1],[ -1, 0]])*
              (y1-y2) + 1/2 * ( y1 + y2);
      /* Definieren der 4 Strecken */
      koch(y1, z1, lev-1);
      koch(z1, z2, lev-1);
      koch(z2, z3, lev-1);
      koch(z3, y2, lev-1);
    end_if;
  end_proc:
```

Allerletztes Beispiel III

```
/* Einfacher Fall einer Linie */
plotKoch1:=proc(lev)
           begin
             Listelinien := [];
             y1 := matrix([0,0]);
             y2 := matrix([1,0]);
             koch(y1,y2,lev);
             plot(Listelinien, Axes = None);
           end_proc:
```