Makroeditor für Parametrieteile

Einführung

Parametrische Teile (PPM-Dateien) werden unter Verwendung einer Textbeschreibung (Skript) definiert. Das Skript definiert Struktur, bearbeitbare Eigenschaften und die Ausgabe, woraus sich ein parametrisch bearbeitbares Teil ergibt.
Das Skript muss mit der Dateierweiterung *.PPM gespeichert werden. Der Name der Datei bestimmt den Teilenamen.

Prüfen eines Skripts

Ein einfaches Beispiel eines parametrischen Teils ist ein Rechteck, bei dem Breite, Höhe und Drehwinkel über Parameter definierbar sind. Das Skript eines solchen Teils könnte wie folgt aussehen:

```
// Beschreibung eines einfachen Rechtecks.
        H = Parameter("Höhe", 5, LINEAR, Interval(0, 100));
        L = Parameter("Länge", 10, LINEAR, Interval(0, 200));
        Winkel = Parameter("Winkel", 0, ANGULAR, Interval(0, 360));
        Recht1 = Rectangle(H, L);
        Recht = RotateZ(Recht1, Winkel);
        Output(Recht);
```

Lassen Sie uns nun jede Zeile dieses Beispiels prüfen

ZEILE 1

// Beschreibung eines einfachen Rechtecks.
'//' zeigt an, dass es sich um einen Kommentar handelt. Kommentare haben keinen Einfluss auf das Verhalten eines Teils. Der nach '//' folgende Text wird bis zum Ende der Zeile in den Kommentar eingeschlossen.

ZEILE 2

H = Parameter("Höhe", 5, LINEAR, Interval(0, 100));
Die zweite Zeile gibt die Definition des Parameters 'H' an. Es folgt eine Beschreibung jedes Elements dieser Zeile zur Definition der damit verbundenen Funktion:

H	Dies ist die Kennung (Name) des Parameters in der Teilebeschreibung
=	Hiermit wird die Kennung mit dessen Definition assoziiert
Parameter	Dies ist eine Funktion. 'Parameter' bestimmt, dass H ein Parameter ist
(	Zeigt den Beginn der Funktionseigenschaften des Parameters an
"Höhe"	Der Name des Parameters, der im Eigenschaftendialog erscheint
,	Zeigt das Ende einer Eigenschaft und den Beginn der nächsten Eigenschaft an
5	Weist den Standardwert für H zu
,	Trennt Eigenschaften
LINEAR	Gibt an, dass H ein linearer Wert ist
,	Trennt Eigenschaften
Interval(0, 100)	Gibt die erlaubten Werte für H als Intervall zwischen 0 und 100 an
)	Zeigt das Ende der Funktionseigenschaften des Parameters an
;	Ende der Definition für H

ZEILEN 3 – 4

```
        L = Parameter("Länge", 10, LINEAR, Interval(0, 200));
        Winkel = Parameter("Winkel", 0, ANGULAR, Interval(0, 360));
```

Die nächsten beiden Zeilen in diesem Beispiel sind ähnlich der vorhergehenden Zeile. Sie definieren die Charakteristiken der Parameter L und Angle in einem ähnlichen Layout. Bitte beachten Sie, dass der Parameter 'Winkel' das Intervall 'ANGULAR' (WINKEL) statt LINEAR verwendet.

ZEILE 5

Recht1 = Rectangle(H, L);

Diese Zeile verwendet die Rechteckfunktion, um ein Rechteck mit der Bezeichnung 'Recht1' zu erstellen. Dabei werden die zuvor definierten Parameter H und L verwendet, um Eigenschaften, Höhe und Länge des Rechtecks zu definieren.
Die Mitte dieses Rechtecks befindet sich in der Zeichnung im Modellursprung (x=0,y=0,z=0).
Nachfolgend finden Sie weitere Informationen über die Rechteckfunktion.

ZEILE 6

Recht = RotateZ(Recht1, Winkel);

Diese Zeile definiert ein neues Rechteck mit der Bezeichnung 'Recht'. Dies ist die gedrehte Version von 'Recht1'. Dabei wird der Winkelparameter verwendet, um den Drehungswinkel zu definieren.

ZEILE 7

Output(Recht);

Die letzte Zeile gibt an, dass die Ausgabe des Skripts ein gedrehtes Rechteck mit der Bezeichnung 'Recht' ist. Dieses Rechteck wird als Teil gezeichnet.

Skriptsyntax

Die Beschreibung eines parametrischen Teils besteht aus dem Gesamtinhalt einer Textdatei, außer Kommentaren, Tabs und anderen Steuerungszeichen, die ignoriert werden. Kommentare werden entweder mithilfe von "//" angegeben, was bedeutet, dass alle nachfolgenden Zeichnen bis zum Ende der Zeile als Kommentare angesehen werden oder mithilfe der Zeichenpaarung "/*" und "*/", welche den Anfang und das Ende des Kommentars anzeigt
Eine Textbeschreibung ist ein Satz aus zwei Operatorentypen, einem Bezeichner (Identifier) und einem Ausdruck (Expression):
<Bezeichner>
und
<Ausdruck>;

Bezeichner

Der Bezeichner <*Bezeichner*> definiert den symbolischen Namen eines Objekts. Er besteht aus römischen Buchstaben und arabischen Zahlen und muss mit einem Buchstaben beginnen.
Gültige Namen sind beispielsweise:
TEIL2a
MeinTeil
A134
Objektbezeichner dürfen nicht den gleichen Namen wie Funktionen oder Namen wie PI oder LINEAR haben. Es gibt reservierte Wörter, die verwendet werden, um die Konstanten der Skriptsprache zu bestimmen. Eine Liste aller reservierten Namen finden Sie unter Liste der für parametrische Teile reservierten Wörter.

Ausdrücke

Ausdrücke definieren den assoziierten Bezeichner. Die Ausdrucksyntax entspricht der der Mehrzahl aller Programmiersprachen. Sie definiert numerische Werte, arithmetische Operationen , die Abhängigkeit des definierten Objekts zu anderen Objekten und Funktionsaufrufe.

So sieht die Struktur eines Funktionsaufrufs aus:

<Funktionsname> (<Parameterliste>),

Beispiele für korrekte Ausdruckssyntaxen:

D=4;
k=3;
e=(D - 1/4)*k;

```
L=2;
C=3;
m=6;
MyLine=Polyline(Point(0, 0.25 -1/8), Point(0,D),Arc1(L-C,-m,m), Point(0,0));
A = B + 0,5;
B = 7;
Output(MyLine);
```

Arithmetische Operationen

Arithmetische Operationen können die standardmäßigen arithmetischen Operatoren '+' (Addition), '-'(Subtraktion), '*' (Multiplikation), '/' (Division) und Klammern '(' und ')' verwenden, um die Sequenz der Durchführung arithmetischer Operationen zu bestimmen. Objektbezeichner und Zahlen dienen dabei als Operanden.

Skriptsemantik

Ein Skript enthält die volle Beschreibung eines parametrischen Teils. Die Sammlung der Skriptoperatoren bestimmt, welche Aktionen durchgeführt werden müssen, um das oder die sich ergebende(n) Objekt(e) zu erstellen. Um ein Skript korrekt erstellen zu können, muss man ein klares Verständnis darüber haben, wie Operatoren interpretiert werden.

Bezeichner, die in einem <*Ausdruck*> verwendet werden, müssen definiert werden. Mit anderen Worten muss ein Bezeichner wie folgt verwendet werden:

<Bezeichner> = <Ausdruck>;

Die Liste der sich ergebenden Objekte wird im Parameter *Output(..)* definiert. Der Parameter *Output(..)* enthält eine Liste darüber, welche Objekte als sich ergebende Teile angezeigt werden. Dieser Operator muss im Skript vorhanden sein. Jedes Objekt in der Argumenteliste für *Output(..)* muss definiert sein. Mit anderen Worten muss ein Bezeichner wie folgt verwendet werden:

<Bezeichner> = <Ausdruck>;

Dieser Operator muss im Skript vorhanden sein. Mindestens ein Objekt muss in dem Output-Operator aufgeführt sein, jedoch muss nicht jedes im Skript verwendete Objekt ausgegeben werden.
Der Output-Operator bestimmt die Methode, die zur Erstellung eines Objekts mit diesem Namen verwendet wird.

Ein korrektes Skript, das ein parametrisches Teil beschreibt, sollte folgenden Regeln entsprechen:

1. Ein Skript kann mehr als einen *Output(..)*-Operator haben, aber jeder <*Bezeichner*> sollte in nur einem *Output(..)-Operator vorhanden sein.*
2. Für jedes im *Output(..)*-Operator verwendete Objekt sollte es einen (und zwar nur einen) <*Bezeichner*> geben.
3. Für jedes in einem <*Ausdruck*> tverwendete Objekt sollte es einen (und zwar nur einen) <*Bezeichner*> geben.
4. Jeder Bezeichner sollte nur ein Mal als <*Bezeichner*> verwendet werden.
5. Jeder Bezeichner sollte mindestens ein Mal in einem <*Ausdruck*>-Operator oder in einem *Output(..)*-Operator vorkommen.
6. Zirkuläre Berechnungen und ineinandergreifende Bezüge sind nicht erlaubt. Das Skript darf keine Interdependenz enthalten, bei der "Element 1" durch "Element 2" und "Element 2" durch "Element 1" definiert wird. Folgende Bedingungen sind nicht erlaubt:

```
A = B + 0,5;
B = sin(A);
```

oder

```
A = C+5;
B = D+42;
C = (3*(2+A));
D = A/2;
```

Im ersten Fall definieren A und B sich direkt gegenseitig. Im zweiten Fall wird A über C durch B definiert und B ist über D durch A definiert. Dies bedeutet ebenfalls, dass es nicht erlaubt ist, dass ein Bezeichner eine Interdependenz zu sich selbst hat. Sie können beispielsweise keinen Operator in dieser Form verwenden:

```
H = H*1,05;
```

7. Die Sequenz der Skriptoperatoren ist nicht wichtig (außer in bestimmten Fällen, die näher beschrieben werden), da Operatoren sortiert werden, bevor das Skript ausgeführt wird.

Grundfunktionen

Möglicherweise sind die wichtigsten Vorteile für das Erstellen parametrischer Teile durch Skripte die kompakte Dateigröße und die Übersichtlichkeit der Beschreibung in Form eines Textes. Der Satz an Grundfunktionen, der in solch einer Beschreibung verwendet wird, bestimmt die Übersichtlichkeits- und Einfachheitsstufe für eine bestimmte Klasse an parametrischen Teilen.

HINWEIS: *Es ist beabsichtigt, dass der Satz an Hauptfunktionen von Version zu Version erweitert wird*

Parameterbeschreibung

Es ist wichtig, die innerhalb einer Parameterfunktion verwendete Struktur zu verstehen.

Format:

<id> = Parameter(<Name>, <Standardwert>,
<Typ>[, <Bedingung1>][, <Bedingung2>]..);

HINWEIS: *Die Zeichen '<>' werden verwendet, um Elemente im Ausdruck zu definieren. Die Zeichen '[ ]' werden verwendet, um optionale Elemente anzuzeigen*

<Name>	Der in der Bedieneroberfläche angezeigte Name;
<Standardwert>	Der Standardwert des Parameters. Sind z. B. innerhalb einer Klammer mehrere Werte durch ein Komma getrennt, sollten darin vorkommende Dezimalwerte grundsätzlich durch einen Punkt getrennt werden.
<Typ>	Definiert den Parametertyp. Die folgenden Beispielwerte sind möglich: **LINEAR** bedeutet, dass der Parameter ein linearer Wert in den ausgewählten linearen Maßeinheiten ist. **ANGULAR** bedeutet, dass der Parameter ein Winkelwert in den ausgewählten Winkeleinheiten ist. (aktuell sind nur Gradwerte verfügbar) **TEXT** ist eine Textfolge; **FONT** ist der Namen einer Schriftart; **COLOR** ist ein RGB-Farbwert; **MATERIAL** ist ein Materialname; **CHECKBOX** ist ein logischer Wert, entweder ON (AN) oder OFF (AUS)
<Bedingung>	Bedingungen sind optional. Sie definieren mögliche Einschränkungen, die auf Parameter einwirken. Einschränkungen können in willkürlicher Reihenfolge aufgeführt werden und können folgende Form haben: **Set(<Wert>,...)** - eine Liste von erlaubten Parameterwerten **Interval(<Minimalwert>, <Maximalwert>)** - stellt Minimal- und Maximalwerte für den Parameter ein;

LessThan(<Wert>) - gibt an, dass der Parameter kleiner als der angegebene Wert sein sollte
LessOrEqual(<Wert>) - gibt an, dass der Parameter nicht größer als der angegebene Wert sein sollte
GreaterThan(<Wert>) - gibt an, dass der Parameter größe als der angegebene Wert sein sollte
GreaterOrEqual(<Wert>) - gibt an, dass der Parameter nicht kleiner als der angegebene Wert sein sollte
Set(FolderList) - ein bestimmter Einstellungsoperator, der in Erscheinung tritt, wenn eine Liste von erlaubten Werten durch den Operator FolderList definiert wird.
Einschränkungen sollten einander nicht widersprechen.
Beispielsweise können Sie **GreaterThan(5)** und **LessThan(2)** nicht miteinander kombinieren.
Wenn Sie Parametereinschränkungen angeben, ist es nicht erlaubt, Bezeichner oder Ausdrücke zu verwenden, die direkt oder indirekt von anderen Parametern abhängen, wie z. B. Argumente der obengenannten Funktionen. Nur Konstanten oder Konstantenausdrücke dürfen verwendet werden, wie zum Beispiel: **LessOrEqual(PI/2)**.

Beispiel für eine Parameterbeschreibung:

```
Alpha = Parameter("Drehwinkel", 45, ANGULAR, Interval(-90, 90));
// Hier wird ein Parameter erstellt, der einen Drehwinkel definiert.  Der Name ist 'Drehwinkel', der Standardwert ist 45, der Werttyp ist Winkel und das Intervall liegt zwischen '-90' und  '90'
```

Funktionen für das Erstellen von 2D-Objekten

Circle (Kreis)

Die Funktion *Circle* wird verwendet, um Kreise zu erstellen.

Format:

Circle(<Radius>[, <cx>, <cy>]);

<Radius>	Definiert den Kreisradius
<cx>, <cy>	Definiert die optionalen Argumente, die die (x, y)-Koordinaten des Kreismittelpunkts bestimmen. Standardeinstellung: cx = 0, cy = 0

Beispiel:

```
K = Circle(D/2, 0, y0);
```

Ausführlicheres Beispiel:

```
//circle.ppm -- zwei Kreise
r1 = Parameter("Radius1", 2.5, LINEAR, Interval(0.0,10,0));
r2 = Parameter("Radius2", 1.25, LINEAR, Interval(0.0,10,0));
xc = Parameter("MitteX", 3, LINEAR, Interval(-100, 100));
yc = Parameter("MitteY", 3, LINEAR, Interval(-100, 100));
K1 = Circle(r1); // Kreis liegt mittig auf dem Ursprung
K2 = Circle(r2, xc, yc); // Kreis mit Versatz vom Ursprung
Output(K1, K2);
```

Rectangle (Rechteck)

Die Funktion *Rectancgle* wird verwendet, um Rechtecke zu erstellen.

Format:

Rectangle(<Breite>, <Höhe>[, <cx>, <cy>]);

<Breite> Definiert die Rechteckbreite
<Höhe> Definiert die Rechteckhöhe
<cx>, <cy> Definiert die optionalen Argumente, die die (x, y)-Koordinaten des Rechteckmittelpunkts bestimmen. Standardeinstellung: cx = 0, cy = 0

Beispiel:
recht = Rectangle(B, H, B/2, H/2); // Linke untere Ecke befindet sich im Punkt (0, 0)

Polyline (Polylinie)
Die Funktion *Polylinie* wird verwendet, um Polylinien zu erstellen, die aus geraden Liniensegmenten und Bogensegmenten bestehen.

Format:
Polyline(<Argumenteliste>);

<Argumenteliste> Definiert die Argumenteliste, durch Kommata getrennt. Argumente definieren die individuellen Segmente einer Polylinie.

Ein Liniensegment wird durch 2 Punkte definiert.
Ein Bogensegment wird durch eine Abrundungsfunktion oder durch eine Funktion *Arc0* oder *Arc1* und zwei Punkten an den Enden des Bogens definiert.
Für Polylinien, die nur aus geraden Liniensegmenten bestehen, enthält die <*Argumenteliste*> nur 2D-Punkte, die über die Funktion *Point(x,y)* definiert werden.

Format:
Point(<cx>,<cy>)
<cx> Definiert die X-Koordinate des Punkts
<cy> Definiert die Y-Koordinate des Punkts

Ein Rechteck kann beispielsweise wie folgt definiert werden:
recht = Polyline(// Kein Zeilenende
Point(0,0), // diese Funktion befindet sich auf mehreren Zeilen
Point(B, 0),
Point(B, H),
Point(0, H),
Point(0, 0));

Es sollte noch angemerkt werden, dass es sich um eine geschlossene Polylinie handelt, falls die Anfangs- und Endpunkte deckungsgleich sind. Dieser Polylinientyp ist auf einen bestimmten Bereich begrenzt und kann für die Erstellung von 3D-Objekten verwendet werden.

Polylinien mit Bogensegmenten werden durch das Hinzufügen der Hilfsfunktionen *Arc0* und *Arc1* in der Argumenteliste definiert. *Arc0* baut den kreisförmigen Bogen im Uhrzeigersinn auf, während *Arc1* den kreisförmigen Bogen entgegen den Uhrzeigersinn aufbaut.

Format:
Arc0(<cx>,<cy>),
Arc1(<cx>,<cy>),^

<cx> Definiert die X-Koordinate des Bogenmittelpunkts
<cy> Definiert die Y-Koordinate des Bogenmittelpunkts

Anfangs- und Endpunkte eines Bogens werden durch die vorhergehend und nachfolgend genannten Argumente definiert.
Arc0 und *Arc1* können das erste oder letzte Argument in der Argumenteliste darstellen. Für eine Polylinie, die nu rein Bogensegment enthält, besteht die <*Argumenteliste*> aus zwei durch die

Funktion *Point(x,y)* definierten Punkten und aus einem Bogen, der entweder durch die Funktion *Arc0* oder durch *Arc1*definiert wird.

Beispiel für Bögen in einer Polylinie:

```
//Polyarc.ppm -- Polylinie mit Bögen
YGroesse=5;
XGroesse=6;
R = 1;
Pfad = Polyline(Point(0, R),          // Beginn an Oberseite der abgerundeten unteren linken Ecke
Point(0, YGroesse-R),                 //  gehe zur Unterseite der abgerundeten oberen linken Ecke
Arc1(0, YGroesse, R),                 //  aus dieser Ecke einen "Ausschnitt" machen
Point(R, YGroesse),                   //   linke Seite der oberen Kanten
Point(XGroesse-R, YGroesse),
Arc0(XGroesse-R, YGroesse-R, R), // aus dieser Ecke eine  "Abrundung" machen
Point(XGroesse, YGroesse-R),
Point(XGroesse, R),
Arc0(XGroesse-R, R, R),               // weitere Abrundung
Point(XGroesse-R, 0),
Point(R, 0),
Arc1(0, 0, R),                        // weiterer Ausschnitt
Point(0, R));
Output(Pfad);
```

Eine weitere Methode für das Erstellen eines Bogens innerhalb einer Polylinie ist es, die Hilfsfunktion *Fillet* zu verwenden. Diese Funktion "glättet" zwei lineare Segmente, die am vorhergehenden Punkt beginnen und enden, indem an einer Ecke ein Bogen mit einem bestimmten Radius eingefügt wird. Dies gewährleistet glatte Übergangspunkte.

Format:

```
Fillet(<Radius>);
```

<Radius> Definiert den Radius der Abrundung.

Beispiel für Abrundungen in einer Polylinie:

```
// polyfillet.ppm -- Polylinie mit Abrundungen
H = 5;
L = 10;
FR = 1;
p2 = Polyline(   // Rechteck mit abgerundeten Ecken
Point(0,0),       // untere linke Ecke
Point(L,0),       // untere rechte Ecke
Fillet(FR),       //  platziert Abrundung unten rechts
Point(L,H),       // obere rechte Ecke
Fillet(FR),       // platziert Abrundung oben rechts
Point(0,H),       // obere linke Ecke
Fillet(FR),       // platziert Abrundung oben links
Point(0,0),       // schließt das Rechteck
Fillet(FR)        // Anfangs-/Endecke der Abrundung Da es sich um eine geschlossene Form handelt
                  // wird nachfolgend keine Point-Funktion benötigt
);
Output(p2);
```

Abrundungen und Bögen können innerhalb der Funktion *Polyline* auch zusammen verwendet werden.

Beispiel für Bögen und einer Abrundung in einer Polylinie:

```
Poly1 = Polyline(                    // Rechteck mit abgerundeten Ecken
Point(0, 0),
Point(B - r, 0), Arc1(B - r, r), Point(B, r),
Point(B, H - r), Arc1(B - r, H - r), Point(B – r ,H),
Point(0, H), Fillet(r),
Point(0, 0), Fillet(r) );
```

Funktionen für das Erstellen von 3D-Objekten aus 2D-Objekten

Sie können 2D-Objekte als Basis für die Erstellung von 3D-Objekten verwenden.

Thickness (Stärke)

Die Funktion *Thickness* erstellt ein 3D-Objekt, das auf einem 2D-Objekt basiert, indem eine Stärke zugewiesen wird. Sie erlaubt Ihnen ebenfalls, die Stärkeeigenschaft des 3D-Objekts zu ändern.

Format:

```
Thickness(<Objekt>, <Wert>);
```

<Objekt> Definiert das ursprüngliche Grafikobjekt.
<Wert> Definiert den neuen Stärkewert.

Beispiel für das Zuordnen einer Stärke:

```
RechtA = Rectangle(2, 5);
RechtSt = Thickness(RechtA, 3);
```

Beispiel für das Zuordnen einer Stärke zum Erstellen eines Quaders:

```
Input(x0,y0,z0,x1,y1,z1)
R = Rectangle(x1-x0, y1-y0, (x0+x1)/2, (y0+y1)/2);
S = Thickness(R, z1-z0);
Output(Move(S, 0, 0, z0));
```

Weiteres Beispiel für das Zuordnen einer Stärke:

```
//thickrect.ppm -- zeichnet ein 2D-Rechteck und fügt eine Stärke hinzu
L = Parameter("Länge", 4, LINEAR, Interval(0.1, 20));
B = Parameter("Breite", 3, LINEAR, Interval(0.1, 20));
H = Parameter("Höhe", 1.5, LINEAR, Interval(0.1, 20));
Recht = Rectangle(L, B);
Quader = Thickness(Recht, H);
Output(Quader);
```

Beispiel für das Zuordnen einer Stärke zu einem Kreis:

```
//thickcircle.ppm -- zeichnet einen Kreis und fügt eine Stärke hinzu
Zylind=Thickness(Circle(1,2,2),2);
Output(Zylind);
```

Beispiel für das Ändern einer Stärke:

```
//thickcircle2.ppm -- zeichnet einen Zylinder und ändert die Stärke
Zylind=Thickness(Circle(1,2,2),2);
Zyl2 = Thickness(Zylind, 4); // ändert die Stärke des ersten
Zylinders
Output(Zyl2);
```

Sweep (Extrusion)

Die Funktion *Sweep* erstellt ein 3D-Objekt durch Extrusion eines angegebenen Profils entlang eines Pfads, der durch eine 2D-Polylinie oder durch einen Kreis definiert wird. Das Profil wird durch eine geschlossene 2D-Polylinie oder durch einen Kreis definiert.

Format:
Sweep(<Profil>, <Pfad>[,<Drehwinkel>]);

<Profil> Definiert das Profil mithilfe einer 2D-Polylinie.
<Pfad> Definiert den Pfad, entlang dessen das Profil "gezogen" wird. Der Pfad wird durch eine 2D-Polylinie definiert.
Hinweis: Pfadebene und Profilebene müssen nichtparallel verlaufen.
<Drehwinkel> Dieses optionale Argument definiert den Drehwinkel des Profils relative zur Z-Achse. Standardmäßig entspricht dieses Argument dem Wert Null.

Beispiel für eine Extrusion:

```
Poly1 = Polyline(
Point(0,0),
Point(1,0),
Point(1,2),
Point(0,2),
Point(0,0));
PolyProfil = RotateX(Poly1, 90); // die Funktion Rotate wird später erklärt
PolyPfad = Polyline(
Point(0,0),
Point(10,0),
Point(10,10),
Point(0,10),
Point(0,0));
PolySweep = Sweep(PolyProfil, PolyPfad);
Output(PolySweep);
```

Weiteres Beispiel für eine Extrusion:

```
//sweep1.ppm
R = 2;
D = 5;
C1 = RotateX(Circle(R, D/2+R, 0),90);  // Profil
C2 = Circle(D/2, 0, 0);                 // Pfad
Torus = Sweep(C1,C2);
Output(C1, C2, Torus);                  //C1 und C2 werden als Referenz angezeigt
```

Erweitertes Beispiel für eine Extrusion:

```
//sweep2.ppm -- weiteres Extrusionsbeispiel
L = Parameter("Länge", 5, LINEAR, Interval(0.005, 1000));
B = Parameter("Breite", 3, LINEAR, Interval(0.005, 1000));
H = Parameter("Höhe", 1, LINEAR, Interval(0.1, 3));
FR = Parameter("Abrundungsradius", 1, LINEAR, Interval(0.001, 100));
p = Polyline(Point(0,0), Point(0,H), Point(-FR,H), Point(-FR,0), Point(0,0));
p1a = RotateX(p,90,0,0);
p1 = Move(p1a, 0, B/2, 0);
p2 = Polyline(Point(0,0), Point(0,B), Fillet(FR), Point(L,B),Fillet(FR), Point(L,0), Fillet(FR),
Point(0,0), Fillet(FR));
s = Sweep(p1, p2); Output(s);
```

HINWEIS: *Bitte beachten Sie, dass Dezimalwerte, wie oben angegeben, mit einem*

Dezimalpunkt anstelle eines Kommas eingegeben werden müssen (also z.B. „Interval(0.001, 100)“ anstelle von „Interval(0,001, 100)“), da die Werte ansonsten nicht korrekt interpretiert und Skripte evtl. nicht ausgeführt werden können.

Funktionen zur direkten Erstellung von 3D-Objekten

3D-Objekte lassen sich auch direkt und ohne Referenz auf ein 2D-Objekt erstellen.

Sphere (Kugel)

Die Funktion *Sphere* wird verwendet, um eine 3D-Kugel zu erstellen.

Format:

Sphere(<Radius>[,<cx1>,<cy1>,<cz1>]);

<Radius>	Dieser Wert bestimmt den Kugelradius.
<cx1>,<cy1>,<cz1>	Dies sind optionale Argumente, die verwendet werden, um die x-, y-, z-Position des Kugelmittelpunkts zu bestimmen. Standardmäßig haben diese Argumente den Wert Null

Beispiel für eine Kugel:

SR1 = Sphere(10,1,3,5.5);

Weiteres Beispiel für eine Kugel:

```
//sphere.ppm -- einfaches Kugelbeispiel
R = Parameter("Radius", 2, LINEAR, Interval(0.01, 20));
cx = Parameter("MitteX", 0, LINEAR, Interval(-100, 100));
cy = Parameter("MitteY", 0, LINEAR, Interval(-100, 100));
cz = Parameter("MitteZ", 0, LINEAR, Interval(-100, 100));
K = Sphere(R, cx, cy, cz);
Output(K);
```

Cone (Kegel)

Die Funktion *Cone* wird verwendet, um einen 3D-Kegel zu erstellen.

Format:

Cone(<Höhe>,<Grundflächenradius>[,<ObererRadius>]);

<Höhe>	Dieser Wert bestimmt die Kegelhöhe.
<Grundflächenradius>	Dieser Wert bestimmt den Radius der Kegelgrundfläche.
<ObererRadius>	Dieses optionale Argument bestimmt den oberen Radius des Kegels zur Erstellung eines Kegelstumpfs. Standardmäßig hat dieses Argument den Wert Null.

Beispiel für einen Kegel:

CN1 = Cone(10,5,2);

Weiteres Beispiel für einen Kegel:

```
//cone1.ppm -- ein einfacher Kegel
R = Parameter("Grundflächenradius", 1, LINEAR, Interval(0.01, 10));
H = Parameter("Höhe", 3, LINEAR, Interval(0.05, 20));
Kegel1 = Cone(H, R, 0);
Output(Kegel1);
```

Beispiel für einen Kegelstumpf:

```
//cone2.ppm -- ein Kegelstumpf
R1 = Parameter("Grundflächenradius", 3, LINEAR,
```

```
Interval(0.01, 10));
R2 = Parameter("ObererRadius", 1, LINEAR, Interval(0, 10));
H = Parameter("Höhe", 3, LINEAR, Interval(0.05, 20));
Kegel2 = Cone(H, R1, R2);
Output(Kegel2);
```

Funktionen für das Umwandeln geometrischer Objekte

Diese Funktionsklasse wird für das Verschieben und Drehen von geometrischen Objekten verwendet. Diese Vorgänge beziehen sich auf die Umwandlung des Koordinatensystems. Dabei werden umgewandelte Objekte erstellt, während sich die Originalobjekte nicht verändern.

Move (Verschieben)

Die Funktion *Move* wird verwendet, um Grafikobjekte zu verschieben.

Format:

Move(<Objekt>, <dx>, <dy> , <dz>[,Zähler]);

<Objekt>	Definiert das ursprüngliche Grafikobjekt.
<dx>, <dy>, <dz>	Definiert den Verschiebewert entlang den Achsen x,y und z.
<Zähler>	Definiert die Anzahl der erstellten Objekte, wobei jedes nachfolgende Objekt durch Verschieben des vorhergehenden Objekts erstellt wird. Dieses Argument ist optional und hat einen Standardwert von 1.

Beispiel für eine Verschiebung:

PolyProfil = Move(Poly1, 1, 3);

Weiteres Beispiel:

```
//move.ppm -- illustriert die Funktion Move
RB = Parameter("Grundflächenradius", 3, LINEAR,Interval(0,1, 10));
RT = Parameter("ObererRadius", 1, LINEAR, Interval(0, 10));
H = Parameter("Höhe", 4, LINEAR, Interval(0.1, 20));
keg1 = Cone(H, RB, RT);
cx = Parameter("CenterX", 5, LINEAR, Interval(-10, 10));
cy = Parameter("CenterY", 0, LINEAR, Interval(-10, 10));
cz = Parameter("CenterZ", 0, LINEAR, Interval(-10, 10));
Anzahl = Parameter("Kopien", 2, LINEAR, Interval(1, 10));
keg2 = Move(keg1, cx, cy, cz, Anzahl)  ;  // erstellt Anzahl an Kopien, bei der jede um cx, cy, cz
                                          // versetzt wird
Output(keg1, keg2);
```

Rotate (Drehen)

Die Funktionen *RotateX*, *RotateY*, *RotateZ* werden verwendet, um Grafikobjekte um die Achsen X, Y und Z zu drehen.

Format:

RotateX(<Objekt>, <Drehwinkel>[, <cy>, <cz>[, <Zähler>]]);
RotateY(<Objekt>, <Drehwinkel>[, <cx>, <cz>[, <Zähler>]]);
RotateZ(<Objekt>, <Drehwinkel>[, <cx>, <cy>[, <Zähler>]]);

<Objekt>	Definiert das ursprüngliche Grafikobjekt.
<Drehwinkel>	Definiert den Drehwinkel.
<cx>, <cy>, <cz>	Stellt einen Versatzwert für die Drehachse relativ zur X-, Y- und Z-Achse ein (entsprechend den Funktionsnamen). Diese Argumente sind optional, wobei nur alle drei Argumente gleichzeitig weggelassen werden können. Die Standardwerte für <cx>, <cy>, <cz> sind jeweils Null.

<Zähler> Definiert die Anzahl der erstellten Objekte, wobei jedes nachfolgende Objekt durch Umwandlung des vorhergehenden Objekts erstellt wird. Dieses Argument ist optional und hat einen Standardwert von 1.

Beispiel für eine Drehung:

```
PolyProfil = RotateX(Poly1, 90);
```

Weiteres Beispiel für eine Drehung:

```
//rotate.ppm -- demonstriert die Funktion Rotate
c1 = Circle(2, 10, 0);              // erstellt einen Kreis
c2 = RotateX(c1, -90, 0, 0);        // dreht den Kreis zur XZ-Ebene
c3 = Move(c2, 0, -0,05, 0);         // verschiebt ihn zurück, halbe
Stärke
c4 = Thickness(c3, 0,1);
c5 = RotateZ(c4, 30, 0, 0, 11);   //dupliziert den Kreis durch
Drehen um die Z-Achse
c6 = Circle(2, 0, 10);
c7 = Move(c6, 0, 0, -0,05);
c8 = Thickness(c7, 0,1);
c9 = RotateX(c8, -30, 0, 0, 11);
c10 = Circle(2, 0, 0);
c11 = RotateZ(c10, -90, 0, 0);
c12 = Move(c11, 10, 0, -0,05);
c13 = Thickness(c12, 0,1);
c14 = RotateY(c13, 30, 0, 0, 11);
Output(c4, c5, c8, c9, c13, c14);
```

Funktionen für das Laden externer Symbole als Elemente

Externe, nicht parametrische Symbole lassen sich aus externen Dateien laden, um Bestandteil eines parametrischen Teils zu werden. Die Dateien müssen importierbar sein und durch das CAD-System unterstützt werden (zum Beispiel die Formate *.TCW, *.DWG, *.SKP).

StaticSymbol (Statisches Symbol)

Die Funktion *StaticSymbol* lädt nicht parametrische Symbole aus externen Dateien. Wenn der Dateiname des externen Symbols ohne Pfadinformationen angegeben wird, wird automatisch angenommen, dass sich das Symbol in einem Unterordner mit der Bezeichnung **Macro** befindet, das sich im Standardverzeichnis der parametrischen Teiledatei befindet.

Format:

StaticSymbol(<Dateiname>[,Blockname]);

<Dateiname> Definiert den Dateinamen mit Erweiterung. Falls die Erweiterung nicht angegeben wird, wird das native Dateiformat verwendet.

<Blockname> Dies ist ein optionales Argument. Es zeigt an, dass nur der Block mit dem angegebenen Namen als Symbol geladen und der restliche Inhalt ignoriert werden soll. Wenn dieses Argument nicht definiert wird, wird die aktuelle Zeichnung als Symbol geladen.

Beispiel für ein statisches Symbol:

```
//staticsym1.ppm -- lädt eine externe Datei aus dem Macro-Unterordner
S = StaticSymbol("ExternesSymbol.tcw");
Output(S); //statisches Symbol aus Datei ExternesSymbol.tcw wird in die Zeichnung eingefügt
```

Set(FolderList(...)) / Einrichten(Ordnerliste(...))

Um eine Dateiliste in einem Ordner zu erstellen, wird die Funktion *Set(Folder- List(...))* üblicherweise als Parametereinschränkung verwendet.

Format:

<id> = FolderList(<Pfad> <Maske> = ".ppm");*

<Pfad> Definiert den Pfad zum Ordner, aus dem die Dateiliste erstellt wird.
<Maske> Definiert die Maske der Dateinamen und -erweiterungen.

Beispiel für das Einrichten einer Ordnerliste:

```
// staticsym2.ppm - lädt ein externes Symbol aus einem Ordner, der anders als Macro heißt
Zeichnungsname = Parameter("Zeichnung", "Zeichnung1",
Set(FolderList("..\..\..\Zeichnungen", "*.tcw")));
// Anzahl von "..\..\" (vor dem Ordner Zeichnungen) entspricht der Anzahl
// der Schritte zurück von der Ordnerstruktur beginnend beim  Macro-Unterordner.
S0 = StaticSymbol("..\..\..\Zeichnungen\"+Zeichnungsname+".tcw");
// hier wird ein statisches Symbol aus einer Datei mit einer TCW-Erweiterung geladen und
// ein Dateiname wird aus der über den Parameter Zeichnungsname bezogenen Ordnerliste ausgewählt.
Output(S0);
```

Wird ein relativer Pfad angegeben, müssen Sie daran denken, dass der Pfad niemals auf den Ordner mit der PPM-Datei weist, sondern in den Unterordner mit der Bezeichnung **Macro**. In dem unten angezeigten Beispiel gehen wir davon aus, dass sich **staticsym2.ppm** in folgendem Ordner befindet:

C:\Benutzer\Ich\Dokumente\MeinCAD\PPMDateien
Der in der Funktion *FolderList* verwendete Pfad und der Pfad für das Statische Symbol **staticsym2.ppm** muss dann unbedingt hier beginnen:
C:\Benutzer\Ich\Dokumente\MeinCAD\PPMDateien\Macro

Das externe Symbol wird von diesem Pfad geladen:
C:\Benutzer\Ich\Dokumente\MeinCAD\Zeichnungen
Das bedeutet, dass das Skript drei Verzeichnisse vor zum MeinCAD-Ordner navigieren muss und dann eine Ebene zurück zum Ordner **Zeichnungen**. Der kokrrekte Relative Pfad ist also:
..\..\..\Zeichnungen
Ein weiteres Beispiel, das eine bestimmte TCW-Datei aus dem **Zeichnungen**-Ordner lädt:

```
//staticsym3.ppm -- lädt eine bestimmte Datei aus einem  anderen Ordner
S = StaticSymbol("..\..\..\Zeichnungen\3DQuerschnittTest.tcw");
// lädt nur die angegebene Datei 3DQuerschnittTest.tcw.
// Es muss beachtet werden, dass sich der relative Pfad immer vom Macro-Unterordner ausgeht.
Output(S);
```

Ein parametrisches Teil (eine Datei mit einer PPM-Erweiterung) kann durch Aufruf des Dateinamens wie eine Funktion aufgerufen werden, deren Argumente die Parameter des zu ladenden Teils in der innerhalb der Datei beschriebenen Reihenfolge sind. Weitere Details zu diesem Vorgang finden Sie unter Benutzerdefinierte Funktionen.

Funktionen für Boolesche 3D-Operationen

Funktionen dieser Klasse werden zur Ausführung von Booleschen Operationen bei geometrischen 3D-Objekten verwendet.

BooleanUnion (Boolesche Vereinigung)

Die Funktion *BooleanUnion* erstellt ein Objekt durch Vereinigung der angegebenen Objekte miteinander.

Format:
BooleanUnion(<Objekt>, <Objekt>, ...);

<Objekt> Definiert ein Objekt, das in der Booleschen Operation verwendet werden soll. Es müssen mindestens zwei Objekte definiert werden.

Beispiel für Boolesche Vereinigung:

```
S1 = Sphere(5);
S2 = Sphere(5,5,5);
S3 = Sphere(5,5,-5);
S4 = Sphere(5,-5,5);
S5 = Sphere(5,-5,-5);
S6 = BooleanUnion(S1,S2,S3,S4,S5);
Output(S6);
```

Weiteres Beispiel:

```
R = Parameter("Radius", 8, LINEAR, Interval(0.001, 1000));
s = Sphere(R);
c = Circle(R/3);
c1 = Thickness(c, R*2);
c2 = Move(c1, 0, 0, R); // Zylinder
s1 = BooleanUnion(s, c2); // Kugel mit Zylinder
Output(s1);
```

BooleanSubtraction (Boolesche Differenz)
Die Funktion *BooleanSubtract* erstellt ein Objekt durch Subtraktion des Sekundärobjekts vom Primärobjekt.

Format:
BooleanSubtract(<Primärobjekt>, <Sekundärobjekt>, ...);

<Primärobjekt> Definiert ein Objekt, das in der Booleschen Operation verwendet werden soll. Es gibt nur ein Primärobjekt.

<Sekundärobjekt> Definiert ein Sekundärobjekt, das vom Primärobjekt subtrahiert werden soll. Es muss mindestens ein Sekundärobjekt vorhanden sein.

Beispiel für Boolesche Differenz:

```
S1 = Sphere(5);
S2 = Sphere(5,5,5);
S3 = Sphere(5,5,-5);
S4 = Sphere(5,-5,5);
S5 = Sphere(5,-5,-5);
S6 = BooleanSubtract(S1,S2,S3,S4,S5);
Output(S6);
```

Weiteres Beispiel:

```
R = Parameter("Radius", 8, LINEAR, Interval(0.001, 1000));
s = Sphere(R);
c = Circle(R/3);
c1 = Thickness(c, R*2);
c2 = Move(c1, 0, 0, -R);          // Zylinder
s1 = BooleanSubtract(s, c2);      // Kugel mit Loch
Output(s1);
```

BooleanIntersect (Boolesche Schnittmenge)

Die Funktion *BooleanIntersect* erstellt ein Objekt, das von der Schnittmenge des Primär- und Sekundärobjekts abgeleitet wird.

Format:

BooleanIntersect(<Objekt>, <Objekt>)

<Objekt> Definiert ein Objekt, das in der Booleschen Operation verwendet werden soll. Es dürfen nur zwei Objekte definiert werden.

Beispiel für Boolesche Schnittmenge:

```
S1 = Sphere(5);
S2 = Sphere(5,5,5);
S3 = Sphere(5,5,-5);
S4 = Sphere(5,-5,5);
S5 = Sphere(5,-5,-5);
S6 = BooleanIntersect(S1,S2);
Output(S6);
```

Funktionen für das Ändern von 3D-Objekten

Es sind verschiedene Funktionen für das Verändern der Geometrie eines 3D-Objekts verfügbar.

G3Fillet (Kanten abrunden)

Die Funktion *3DFillet* erlaubt das Abrunden von einer oder von mehreren Kanten eines 3D-Objekts.

Format:

G3Fillet(<Objekt>, <Kanten>, <Radien>);

<Objekt> Definiert das 3D-Objekt, dessen Kanten abgerundet werden sollen.

<Kanten> Definiert die Kante oder mehrere Kanten, die abgerundet werden sollen. Jede Kante wird durch *Point(xc,yc,zc)* oder durch eine Punktmatrix definiert.

<Radien> Definiert den Abrundungsradius. Abrundungsradien werden durch die Funktion *Array* eingerichtet. Für eine einzelne Kante enthält die Funktion *Array* ein Wertepaar. Für mehrere Kanten sind mehrere Wertepaare angegeben.

Beispiel für das Abrunden von Kanten:

```
Array(Point(x1,y1,z1), Point(x2,y2,z2), Point(x3,y3,z3));   //  definiert 3 Kanten für die Abrundung
                //Point(x1,y1,z1), Point(x2,y2,z2), Point(x3,y3,z3); - 3 Mittelpunkte  auf 3 Kanten
                werden abgerundet
```

Point(xc,yc,zc) ist der Mittelpunkt einer abzurundenden Kante (dieser Punkt wird in der TurboCAD-Kantenabrundungsfunktion mit einem blauen Quadrat gekennzeichnet). Eine Punktematrix definiert einen Satz an abzurundenden Kanten.

Weiteres Beispiel:

```
Array(r1, r2)          // Matrix von Radiuswerten für die Abrundung der ausgewählten Kante
                       // Sie definiert die Abrundungsradien für 2 Enden der ausgewählten Kante
                       //r1 – Anfangsradius der Abrundung
                       //r2 – Endradius der Abrundung
```

Beispiel für das Abrunden einer Kante:

```
3Fillet(TeilA,Point(xc,yc,zc), Array(r1, r2));      // Point(xc,yc,zc) stellt den Mittelpunkt der Kante dar
```

Weiteres Beispiel:

```
Tür= G3Fillet(Tür0, Point(0, -1, (Height-FHeight-4-3/4)/2), Array(1, 1));
```

Zum Beispiel (Abrundung einer Kante des Quaders):

```
x = Parameter("Groesse", 5, LINEAR, GreaterThan(0));
```

```
r1 = Parameter("r1", 1, LINEAR, GreaterThan(0));
b0 = Box(0, 0, 0, x, x, x);
b1 = G3Fillet(b0, Point(x/2, 0, 0), Array(r1, r1*2));
Output(b1);
```

Beisiel für das Abrunden von vier Kanten eines Quaders:

```
L = Parameter("Länge", 5, LINEAR);
B = Parameter("Breite", 3, LINEAR);
H = Parameter("Höhe", 1, LINEAR);
R = Parameter("Radius", 0,5);
g0 = Box(0,0,0,L,B,H);
g1 = G3Fillet(g0, Array(Point(L/2, 0, 0), Point(0, B/2, 0),
Point(L/2, W, 0), Point(L, B/2, 0)), Array(R, R, R, R, R, R, R, R));
Output(g1);
```

3DChamfer (Kanten fasen)

Die Funktion *3DChamfer* erlaubt das Fasen von einer oder von mehreren Kanten eines 3D-Objekts.

Format:

G3Chamfer(<Objekt>, <Kanten>, <Versatz>);

<Objekt> Definiert das 3D-Objekt, dessen Kanten gefast werden sollen

<Kanten> Definiert die Kante oder mehrere Kanten, die abgerundet werden sollen. Jede Kante wird durch *Point(xc,yc,zc)* oder durch eine Punktmatrix definiert. *Point(xc,yc,zc)* ist der Mittelpunkt einer zu fasenden Kante (dieser Punkt wird in der TurboCAD-Kantenfasenfunktion mit einem blauen Quadrat gekennzeichnet). Eine Punktematrix definiert einen Satz an zu fasenden Kanten.

<Radien> Definiert den Fasenabstand. Diese werden durch die Funktion *Array* eingerichtet. Für eine einzelne Kante enthält die Funktion *Array* ein Wertepaar für den Abstand. Für mehrere Kanten sind mehrere Wertepaare für den Abstand angegeben.

Beispiel für eine Fase:

```
Array(d1, d2)- // Matrix von 2 Versatzwerten an den Enden einer Kante
```

Weiteres Beispiel:

```
Tür= G3Chamfer(Tür0, Point(0, -1, (Height-FHeight-4-3/4)/2), Array(1, 1));
// Hierbei ist Tür0 das Objekt, dessen Kante gefast werden soll
// Point(0, -1, (Height-FHeight-4-3/4)/2) zeigt die Kante an
// Array(1, 1) stellt 2 Fasenabstände ein
```

Weiteres Beispiel:

```
x = Parameter("Größe", 5, LINEAR, GreaterThan(0));
r1 = Parameter("r1", 1, LINEAR, GreaterThan(0));
b0 = Box(0, 0, 0, x, x, x);
b2 = G3Chamfer(b0, Point(x/2, x, x), Array(r1, r1+r1));
Output(b2);
```

G3Offset (Volumenkörper erweitern)

Die Funktion *G3Offset* erweitert eine Volumenkörperfläche nach innen oder außen.

Format:

G3Offset(<Objekt>, <Fläche>, <Versatz>);

<Objekt> Definiert das 3D-Objekt, dessen Kanten erweitert werden soll.

<Fläche> Definiert die zu erweiternde Fläche. Die Fläche wird durch den zur Fläche zugehörigen Punkt *Point(x,y,z)* definiert.

<Versatz> Definiert den Versatzabstand. Bei einem positiven Wert wird die Fläche nach außen, bei einem negativen Wert nach innen versetzt.

Beispiel für Volumenkörpererweiterung:

G3Offset(TeilA, Point(xf, yf, zf), Abst);
Hierbei gilt:
TeilA - ist das 3D-Objekt, dessen Fläche erweitert werden soll
Point(xf, yf, zf) - ist ein Punkt für die Auswahl der zu erweiternden Fläche
Abst - ist der Flächenversatzwert

Weiteres Beispiel:

```
x = Parameter("Größe", 5, LINEAR, GreaterThan(0));
r1 = Parameter("r1", 1, LINEAR, GreaterThan(0));
b0 = Box(0, 0, 0, x, x, x);
b3 = G3Offset(b0, Point(x,x/2,x/2), r1/2);
Output(b3);
```

G3Shell (Volumenkörper umrahmen)

Die Funktion G3Shell erlaubt das Umrahmen von Volumenkörperformen. Dabei bleibt die ausgewählte Fläche offen. Die Funktion erstellt eine Umrahmung eines einzelnen Volumenkörper-Objekts in einer angegebenen Stärke. Die neuen Flächen werden erstellt, indem bereits vorhandene Flächen nach innen oder außen versetzt werden.

Format:

G3Shell(<Objekt>, <Fläche>, <Stärke>);

Beispiel für Volumenkörperumrahmung:

G3Shell(TeilA, Point(xf, yf, zf), Stärke);
Hierbei gilt:
Teil3 - wählt das zu umrahmende Objekt aus
Point(xf, yf, zf) - ist der Punkt auf der Fläche, die offen bleiben soll
Stärke - ist die Umrahmungsstärke

Weiteres Beispiel:

```
L = Parameter("Länge", 5, LINEAR);
B = Parameter("Breite", 3, LINEAR);
H = Parameter("Höhe", 1, LINEAR);
T = Parameter("Stärke", 0.2, LINEAR);
g0 = Box(0,0,0,L,B,H);
g1 = G3Shell(g0, Point(L/2, B/2, H), T);
Output(g1);
// Nach dem Einfügen eines umrahmten Objekts in die Zeichnung
kann die Oberflächenstärke in der Palette Auswahlinformationen
bearbeitet werden (das gleiche gilt für die Parameter
Länge, Breite und Höhe)
```

<Objekt> Definiert das 3D-Objekt, dessen Kanten umrahmt werden sollen.
<Fläche> Definiert die Fläche, die offen bleiben soll. Sie wird durch die Funktion Point(xc,yc,zc) definiert, die einen Punkt auf dieser Fläche angibt.
<Stärke> Definiert die Umrahmungsstärke. Bei einem positiven Wert wird die Umrahmung nach außen, bei

einem negativen Wert nach innen erstellt.

G3Bend (Biegen)

Die Funktion *G3Bend* wird für das Biegen von 3D-Objekten verwendet.

Format:

G3Bend(<Objekt>, <Linie>, <Winkel>, <Radius>, <Tiefe>);

Beispiel für Biegung:

G3Bend(Teil3, Point(x1, y1, z1), Point(x2, y2, z2), Angle, R, 0);

Weiteres Beispiel:

P1=Thickness(Rectangle(10,20),3);
B0 = G3Bend(P1, Point(3, 3, 0),
Point(3,8,0), 90, 2, 0);
Output(B0);

<Objekt> Definiert das zu biegende 3D-Objekt.
<Linie> Definiert eine Linie, um die sich das Volumenkörperobjekt biegen soll. Die Linie wird durch 2 Punkte definiert:
Point(x1, y1, z1), Point(x2, y2, z2).
Die Linie muss auf der für das Biegen ausgewählten Volumenkörperfläche liegen.
<Winkel> Definiert den Biegewinkel. Der Winkel wird von der Ebene der Biegefläche aus gemessen.
<Radius> Definiert den Biegeradius.
<Tiefe> Definiert die Neutrale Tiefe. Sie stellt den Tiefenabstand in das Material, in dem keine Spannungen oder Verdichtungen auftreten, dar.

SetProperties (Einstellen und Ändern von Objekteigenschaften)

Die Funktion *SetProperties* wird zum Einstellen von Objekteigenschaften verwendet.

Format:

SetProperties(<Objekt>, <Eigenschaftenname> = Eigenschaftenwert,
<Eigenschaftenname> = Eigenschaftenwert,
...);

Beispiel für das Einstellen von Eigenschaften:

BlauesRecht=Rectangle(10,5);
RotesRecht = SetProperties(BlauesRecht, "PenColor" = 0xff,
"PenWidth" = 0,2);
Output(RotesRecht);

Weiteres Beispiel:

Seite2M = SetProperties(Seite2, "Material" = "Holz\Fichte",
"PenColor" = 0xff);

Weiteres Beispiel:

PL1 = SetProperties(PL0, "Brush" = "Einfarbig");

Weiteres Beispiel:

EinstPlastik = ("Material" = "Plastik\Weiß");
Boxmaterial = SetProperties(MeineBox,EinstPlastik);

Im Makroeditor für Parametrieteile gibt es ein spezielles Werkzeug zur Auswahl des benötigten Werts für Eigenschaften wie Materialien, Stiftfarben und Füllungsstile. Um es zu aktivieren, klicken Sie mit der rechten Maustaste in die

Palette und wählen Sie die Eigenschaft aus. Dabei wird das Kontextmenü geöff-
<Objekt> Definiert das zu verwendende Objekt als Basis für das neue Objekt mit eingestellten Eigenschaften.
<Eigenschaftenname> Definiert den Namen der einzustellenden Eigenschaft. Der Name sollte in Anführungszeichen eingeschlossen sein.
<Eigenschaftenwert> Definiert den der Eigenschaft zuzuweisenden Wert.

net, und Sie können über den entsprechenden Befehl Tabellen für Materialien, Stiftfarben oder Füllungsstile aufrufen. Die entsprechende Tabelle erscheint an der Stelle, an der der gewünschte Wert ausgewählt werden kann.

Einbetten von Funktionen

Funktionen können innerhalb eines einzelnen Ausdrucks eingebettet werden, um die Effizienz des Skripts zu erhöhen.

Zum Beispiel:

```
BF = BooleanSubtract(B1,Move(RotateZ(RotateY(Box(-5,-5,-5,5,5,5),45)
,45),-1,-1,-1));
```

In kleinem Skript verwendetes Beispiel:

```
B1 = Box(0,0,0,10,10,10);
BF = BooleanSubtract(B1,Move(RotateZ(RotateY(Box(-5,-5,-5,5,5,5),45)
,45),-1,-1,-1));
Output(BF);
```

Funktionen zum Erstellen von Text

Text

Die Funktion *Text* definiert die Zeichenfolge selbst und dessen Charakteristiken, inklusive Schriftarten, Stile, Effekte etc. Akzeptierbare Werte hängen von den auf Ihrem Rechner installierten Schriftarten ab.

Format:

Text(<Textobjekt>, <Textschriftart>, <Textstil>);

Beispiel:

```
bsb = Text("BS(b)", Tfont, Tstyle);
```

<Textobjekt> Definiert die Zeichenfolge. Die Zeichenfolge kann hier entweder direkt (eingeschlossen in Anführungszeichen) eingegeben oder über einen Textobjekt-Bezeichner bestimmt werden.
<Textschriftart> Definiert die Schriftart.
<Textstil> Definiert den Schriftstil.

TextFont (Schriftart)

Die Funktion *TextFont* stellt Schriftart, Schriftgröße und den Winkel der Textzeile ein.

Format:

TextFont(<Modus>, <Höhe>, <Winkel>, <Schriftart>);

Beispiel:

```
Tfont = TextFont(0,2,45, "Arial");
```

Dabei gilt:
0 — bedeutet, dass es sich um Standardtext handelt
2 — Texthöhe
45 — Textzeile liegt in einem Winkel von 45 Grad
Arial — Schriftart

TextStyle (Textstil)

Die Funktion *TextStyle* stellt den Textstil inklusive Ausrichtung, Texteffekten

und Stilen ein.

Format:

TextStyle(<Charakteristikenliste>);

<Modus> Definiert den Textmodus: Standard (bei Modus=0) oder Skalierbar (bei Modus=1 oder ein beliebiger anderer Wert, der von 0 abweicht).

<Höhe> Definiert die Schriftgröße.

<Winkel> Definiert den Winkel der Textlinie.

<Schriftart> Definiert die Schriftart.

Beispiel:

```
Tstyle = TextStyle(LEFT, TOP, UNDERLINE);
```

Weiteres Beispiel:

```
//Standardtext Times New Roman mit einer Schriftgröße von 5,
//mit Ausrichtung Links, Oben und Textbox-Effekt, Fett, Kursiv
im 45-Grad-Winkel
ht=5;
font_name = "Times New Roman";
Tfont = TextFont(0, ht, 45, font_name);
Tstyle = TextStyle(LEFT, TOP, BOX, BOLD, ITALIC);
bsb = Text("BS(b)", Tfont, Tstyle);
Output(bsb);
```

Hilfsfunktionen

Extents (Ausmaße)

Die Funktionen *ExtentsX1*, *ExtentsX2*, *ExtentsY1*, *ExtentsY2*, *ExtentsZ1* und *ExtentsZ2* werden verwendet, um die Ausmaße von Grafikobjekten zu berechnen.

Format:

ExtentsX1(<Objekt>);
ExtentsX2(<Objekt>);
ExtentsY1(<Objekt>);
ExtentsY2(<Objekt>);
ExtentsZ1(<Objekt>);
ExtentsZ2(<Objekt>);

<Charakteristikenliste> Definiert die Textcharakteristika durch Kommata getrennt.
Die folgenden Werte sind dabei erlaubt:
Für die Ausrichtung:
LEFT, CENTER, RIGHT, TOP, MIDDLE,
BASELINE, BOTTOM
Für Texteffekte:
BOX, UNDERLINE, STRIKETHROUGH,
AllCAPS
Für Stile:
BOLD, ITALIC

Beispiel für Parameterpunkt:

```
P0 = ParameterPoint(0, l, -l, 0);
P1 = ParameterPoint(1, 0, 0, 0);
```

PointX, PointY, PointZ (PunktX, PunktY, PunktZ)

Die Funktionen PointX, PointY, PointZ werden verwendet, um die Koordinaten eines parametrischen Punkts zu berechnen. Die Funktion PointX berechnet die X-Koordinate des parametrischen Punkts. Die Funktion PointY berechnet die Y-Koordinate des parametrischen Punkts. Die Funktion PointZ berechnet die Z-Koordinate des parametrischen Punkts.

Format:
PointX (<Punkt>);
PointY(<Punkt>);
PointZ(<Punkt>);
Beispiele für Punkt:
x0 = PointX(P0); // x0=1 für P0 = ParameterPoint(0, l, -l, 0);
y1 = PointY(P1); //y1=0 für P1 = ParameterPoint(1, 0, 0, 0);
z1 = PointZ(P1); //z1=0 für P1 = ParameterPoint(1, 0, 0);

Sonderfunktionen und -operatoren

IF

Die Funktion *IF* erlaubt das Ausführen verschiedener Aktionen, die davon abhängen, ob eine bestimmte Bedingung erfüllt ist oder nicht. Die Funktion spielt die Rolle eines konditionalen Operators und kann dazu verwendet werden, logische Verzweigungen für den Aufbau parametrischer Teile zu erstellen.

Format:
IF(<Bedingung>, <AusdrBeiWAHR>, <AusdrBeiFALSCH>);
<Punkt> Definiert den parametrischen Punkt, von dem die X-, Y- oder Z-Koordinate extrahiert wird.

IF-Beispiel:
A = IF(L >= H, Rectangle(L, H), Rectangle(H, L));
//Unabhängig von der angegebenen Größe von L und H wird das erstellte Rechteck A
//horizontal positioniert (die längere Seite erscheint entlang der X-Achse).
/* In diesem Beispiel liefert "Rectangle(L, H)" das Ergebnis TRUE (wahr) und "Rectangle(H, L" das Ergebnis FALSE (falsch).
*/
Weiteres Beispiel:
Tstyle = IF(richtung > 0, TextStyle(MIDDLE, RIGHT), TextStyle(MIDDLE, LEFT));
//Unabhängig von der angegebenen Größe von "richtung" wird der Textstil mit rechter oder linker Ausrichtung angegeben.

UNITS (Einheiten)

Die Funktion *UNITS* definiert die Einheiten, die im Skript verwendet werden. Sie definiert das System, die Bereichseinheiten und den Bemaßungsmaßstab beim Erstellen Diese Funktion erlaubt das korrekte Laden von Teilen in Zeichnungen mit abweichenden angegebenen Einheiten.
<Bedingung> Definiert die Bedingung, die getestet werden soll. Dabei werden die folgenden Vergleichsoperatoren verwendet:
== (gleich)
< (kleiner als)
> (größer als)
<= (nicht größer als)
>= (nicht kleiner als)
<AusdrBeiWAHR> Definiert den Wert der Funktion *IF*, wenn der Wert von <*Bedingung*> TRUE (wahr) ist;
<AusdrBeiFALSCH> Definiert den Wert der Funktion *IF*, wenn der Wert von <*Bedingung*> FALSE (falsch) ist;

Format:
Units(<N>[<Bemaßungseinheiten>]);
Zum Beispiel:
Units(1[mm]); // bedeutet, dass die Standardeinheiten der
Zeichnung in Millimeter angegeben sind
Units(1[in]); // bedeutet, dass die Standardeinheiten der
Zeichnung in Zoll angegeben sind
Units (1[in]) - bedeutet, dass die Hauptmaßeinheiten in Zoll angegeben sind.
Dies ist die Standardmaßeinheit des Skripts. Alle geometrischen Werte werden
in "Zoll" bemaßt, wenn keine Einheiten angegeben sind.
Es ist möglich, für spezielle Werte andere Einheiten zu verwenden, auch wenn
die gesamte Zeichnung mit der Standardeinheit gezeichnet wurde. Um bei einer
Standardeinheit von Zoll die Einheit Millimeter für bestimmte Werte zu verwenden,
können Sie die gewünschte Einheit für diese Werte explizit angeben.
Sie können beispielsweise den *Wert M=5[mm]* und *Units(1[in])* im gleichen
Skript verwenden. Dies bedeutet, dass nur der Wert M in Millimeter gemessen
wird, während alle anderen Werte in Zoll angegeben sind.
Darüber hinaus erlaubt diese Funktion die Vergrößerung (bei N<1) oder Verkleinerung
(bei N>1) von Objekten.
Zum Beispiel:
Units(2[mm]); // das erstellte Objekt wird um den Faktor 2 im
Vergleich zu Units(1[mm]) skaliert;
Units(0,5[mm]); // das erstellte Objekt wird im Vergleich zu
Units(1[mm]) auf die halbe Größe skaliert;
RefPoint (Bezugspunkt)
Die Funktion *RefPoint* stellt die Position des Bezugspunkts für das parametrische
Teil ein. Wenn der Bezugspunkt einer der Ausgabewerte eines Skripts
darstellt, wird er zusammen mit dem Teil in die Zeichnung eingefügt. Dies
ermöglicht die präzise Einfügung des parametrischen Objekts in die Zeichnung.
Format:
RefPoint(<Punkt>);
<N> Definiert den Objektmaßstab.
<Bemaßungseinheiten> Definiert, ob die Einheiten im englischen oder metrischen
System angegeben werden.

Zum Beispiel:
P0 = ParameterPoint(0, 1, -1, 0);
xPfeil = PointX(P0);
yPfeil = PointY(P0);
rf = RefPoint(xPfeil, yPfeil, 0); //-> Der Bezugspunkt wird
auf dem Punkt (xPfeil,yPfeil, 0) eingefügt
b = Rectangle(xPfeil, yPfeil);
Output(b);
Output(rf);
Input und Output
The Input and Output functions are used for inputting initial values or
objects into the script and outputting result objects from the script.
Format:
Input(<Liste der Variablenbezeichner, durch Komma getrennt);
Output(<Liste der Variablenbezeichner, durch Komma getrennt>);
Zum Beispiel:
Input(H, B, T, A, Abs);
Output(SeiteA_L,Unten_B,Hinten_I, Seite1, FalschD1,
E1,E2,E3,E4, N1, T1, Tür, FF, SeiteA_R);

Beispiel für die bedingte Ausgabe:

```
Sw = Parameter("Schalter", 1, CHECKBOX);
P1 = Thickness(Rectangle(5,5), 3);
S1= Thickness(Circle(2.5),4);
Output(IF(Sw,P1,S1));
// Hier wird entweder ein Zylinder oder eine Box in die
// Zeichnung eingefügt abhängig vom Kontrollkästchen im Wert Sw
```

min und max

Die Funktionen min und max werden für die Auswahl der Minimal- und Maximalwerte innerhalb eines Wertesatzes verwendet.

Format:

min(<Wertesatz>);
max(<Wertesatz>);

<Punkt> Definiert die (x,y,z)-Koordinaten für die Position des Bezugspunkts.

<Liste der Variablenbezeichner, durch Komma getrennt> Definiert die Liste der Variablen oder Objekte für die Eingabe oder eine Liste von Ergebnissen für die Ausgabe.

Zum Beispiel:

```
r=min(2,5,1,7,9);//r=1
R=max(2,5,1,7,9);//R=9
```

Zum Beispiel:

```
A=2; B=5; C=1; D=7; E=9;
A1=2; B1=5; C1=1; D1=7; E1=9;
r=min(A,B,C,D,E);//r=1
R=max(A1,B1,C1,D1,E1);//R=9
```

Beispiel für die Verwendung einer Wertematrix:

```
A=2; B=5; C=1; D=7; E=9;
r=min(Array(A,B,C,D,E));//r=1
```

1 ***HINWEIS:*** *Eine Objektgruppe kann nicht als Argument für diese Funktionen verwendet werden, da eine Gruppe eine Sammlung von Grafikobjekten und keine Reihe von Zahlen darstellt.*

Mod (Divisionsrestwert)

Die Funktion *Mod* wird verwendet, um den Divisionsrest der Ganzzahlteilung zu ermitteln. Beispielsweise liefert *Mod(5,4)* den Wert 1, da 5/4 = 1 mit einem Divisionsrest von 1 ist. *Mod(7,4)* liefert den Wert 3, da 7/4 = 1 mit einem Divisionsrest von 3 ist. *Mod(7,3)* liefert den Wert 1, da 7/3 = 2 mit einem Divisionsrest von 1 ist.

1 ***HINWEIS:*** *Die Mod-Funktion wird häufig verwenden, um zu ermitteln, ob es sich um eine gerade oder ungerade Zahl handelt. Mod(UngeradeZahl, 2) = 1, während (Mod(GeradeZahl), 2) = 0 ist.*

Format:

Mod(<Wert1, Wert2>);

<Wertesatz> Definiert den Satz an numerischen Werten, Bezeichnern von Variablen oder eine Variablenmatrix.

<Wert1> Definiert den Dividenden.

<Wert2> Definiert den Divisor.

Zum Beispiel:

```
r=min(2,5,1,7,9);//r=1
R=max(2,5,1,7,9);//R=9
```

Zum Beispiel:
A=2; B=5; C=1; D=7; E=9;
A1=2; B1=5; C1=1; D1=7; E1=9;
r=min(A,B,C,D,E);//r=1
R=max(A1,B1,C1,D1,E1);//R=9
Beispiel für die Verwendung einer Wertematrix:
A=2; B=5; C=1; D=7; E=9;
r=min(Array(A,B,C,D,E));//r=1
1 ***HINWEIS:*** *Eine Objektgruppe kann nicht als Argument für diese Funktionen verwendet werden, da eine Gruppe eine Sammlung von Grafikobjekten und keine Reihe von Zahlen darstellt.*

Mod (Divisionsrestwert)

Die Funktion *Mod* wird verwendet, um den Divisionsrest der Ganzzahlteilung zu ermitteln. Beispielsweise liefert *Mod(5,4)* den Wert 1, da 5/4 = 1 mit einem Divisionsrest von 1 ist. *Mod(7,4)* liefert den Wert 3, da 7/4 = 1 mit einem Divisionsrest von 3 ist. *Mod(7,3)* liefert den Wert 1, da 7/3 = 2 mit einem Divisionsrest von 1 ist.

1 ***HINWEIS:*** *Die Mod-Funktion wird häufig verwenden, um zu ermitteln, ob es sich um eine gerade oder ungerade Zahl handelt. Mod(UngeradeZahl, 2) = 1, während (Mod(GeradeZahl), 2) = 0 ist.*

Format:
Mod(<Wert1, Wert2>);
<Wertesatz> Definiert den Satz an numerischen Werten, Bezeichnern von Variablen oder eine Variablenmatrix.
<Wert1> Definiert den Dividenden.
<Wert2> Definiert den Divisor.

Weiteres Beispiel:
txt = Parameter("text", "Einfaches Textbeispiel", TEXT);
a = Array(TextFont(0,10,"Arial"),
TextStyle(CENTER, MIDDLE, ITALIC));
//Matrix von 2 Elementen: TextFont und TextStyle)
s0 = Text(txt, a);
Output(s0);

Group (Gruppe)

Die Funktion Gruppe sammelt verschiedene Grafikobjekte in eine Gruppe und weist dem Ergebnis Bezeichnernamen zu. Sie erlaubt es, dass das Skript mit mehreren Objekten funktioniert, die wie ein einzelnes Objekt gehandhabt werden. Eine Gruppe kann der Ausgabewert eines Skripts sein. Objektgruppen können in verschiedenen Operationen eine Rolle spielen. Dazu zählen z. B. das Verschieben (Move), Drehen (Rotate) und weitere.

Format:
Group (<Objektliste>);
Zum Beispiel:
bse = Group(bse_unten, bse_oben); // Gruppe von 2 Grafikobjekten
Br2 = Group(Br0, Br1);
Zum Beispiel:
Bx = Group(Move(BxL, -Dis*1,5), Move(BxR, Dis*1,5));
RegalFBx = BooleanSubtract(RegalF, Bx);
Output(RegalFBx, Bx);

Sonderfunktionen ohne Parameter

PI

Die Funktion PI berechnet den Wert von Pi = 3,14159...

Benutzerdefinierte Funktionen

Wenn Skripte vom gleichen Typ erstellt werden, die eine bestimmte Klasse an parametrischen Teilen beschreiben, kann es nützlich sein, die Sequenz von sich wiederholenden Aktionen als separate Sonderfunktion zur Verfügung zu

<Objektliste> Definiert die Liste der Grafikobjekte, durch Kommata getrennt. Das *<Objekt>* kann ein beliebiges Grafikobjekt sein.

haben. Um dies zu erreichen, können die sich wiederholenden Aktionen in eine separate PPM-Datei gespeichert werden.
In diesem Fall sollten alle Eingabevariablen im Eingabeoperator aufgelistet sein:

Format:

Input(<Liste der Variablenbezeichner, getrennt durch Komma>);

Zum Beispiel:

Input(x0,y0,z0,x1,y1,z1);

Der Ausgabeoperator sollte ebenfalls definiert werden.

Format:

Output(<Liste der Variablenbezeichner, getrennt durch Kommata>);

Eine benutzerdefinierte Funktion, die auf diese Weise erstellt wurde, muss in einem Macro-Ordner liegen, der sich immer im Ordner des aufgerufenen Skripts befinden muss. Wenn die benutzerdefinierte Funktion verwendet wird, wird der Dateiname des Skripts (ohne die Erweiterung .ppm) so verwendet, als handele es sich um eine integrierte Funktion.

Format:

<Dateiname>(<Liste der Eingabeparameter>)

Hier finden Sie ein Beispiel einer benutzerdefinierten Funktion:

```
// box.ppm -- Definiert eine benutzerdefinierte Quader-Funktion.
// Die benutzerdefinierte Funktion wird wie folgt aufgerufen:
// B = Box(Xmin, Ymin, Zmin, Xmax, Ymax, Zmax);
// Die Funktion erstellt einen 3D-Quader mit den angegebenen
Minimal-/Maximalwerten
Input(x0,y0,z0,x1,y1,z1);
R = Rectangle(x1-x0, y1-y0, // Rechteck mit Xmin = x0,
Xmax= x1
(x0+x1)/2, (y0+y1)/2); // Ymin = y0, Ymax = y1
T = Thickness(R, z1-z0); // Tiefe = Zmax - Zmin
Output(Move(T, 0, 0, z0)); // Ergebnis entlang z zu Zmin verschieben
```

Das Skript unten nennt sich quader_einblenden.ppm. Es ruft die benutzerdefinierte Funktion quader.ppm auf

```
// quader_einblenden.ppm verwendet die benutzerdefinierte
Funktion quader.ppm im Macro-Ordner.
x = Parameter("Größe", 5, LINEAR, GreaterThan(0));
r1 = Parameter("r1", 0.5, LINEAR, GreaterThan(0));
b0 = Box(0, 0, 0, x, x, x);
b1 = G3Fillet(b0, Point(x/2, 0, 0), Array(r1, r1*2));
Output(b1);
```

Der Speicherort ist bei der Verwendung von parametrischen Skripten als benutzerdefinierte Funktionen extrem wichtig. Wenn sich das im obigen Beispiel verwendete Skript quader_einblenden.ppm im Ordner D:\Symbole

befindet, kann das Skript quader.ppm nur gefunden werden, wenn es sich im Ordner D:\Symbole\Macro befindet.

Liste der für parametrische Teile reservierten Wörter

| Solid Extrude
UNIQUE GraphicId VertexId
Vertex Face
Edge Source Bound
Intersect OperationList BlendArg
BlendParam BlendType BlendRadiusMode
BlendSetback BlendRadiusBlendSmooth BlendRadiusParam
BlendOffsetParam BlendFaceEntity BlendFaceEdge
BlendFaceVertex BlendEdgeEdge BlendEdgeVertex
BlendEdgeVertexMain BlendEdgeVertexAux ShellArg
ShellThickness ShellFace ShellEdge
FaceEditArg Transform ScaleX
ScaleY ScaleZ ShearXY
ShearXZ ShearYZ RotateX
RotateY RotateZ TranslateX
TranslateY TranslateZ Path
Profile LateralFace LateralEdge
CapFace CapEdge JointEdge
Profiles HighLight FaceMaterialArg
FaceMaterial FaceOffsetArg FaceHoleArg
FaceHole BendId BendRadius
BendAngle BendNeutral BendFlag
BendPosition BendFlangeHeight BendAxialDistance
BendAzimuthAngle BendEdgeStartPosition BendEdgeEndPosition
Face2FaceLoftArg Face2FaceLoft
AssemblyAxis Input Output
Include Units StaticSymbol
FolderList Macro Parameters
Parameter ParameterPoint PointX
PointY PointZ Set
Interval LessThan GreaterThan
LessOrEquail GreaterOrEqual Circle
Rectangle Polyline Point
Arc0 Arc1 Fillet
IF Move Thickness
Sweep Cone BooleanUnion
BooleanSubtract BooleanIntersect G3Fillet
G3Chamfer G3Shell G3Offset
G3Slice G3Bend ExtentsX1
ExtentsX2 ExtentsY1 ExtentsY2
ExtentsZ1 ExtentsZ2 Text
TextFont TextStyle Group
SetProperties PatternCopy

Das Skript unten nennt sich quader_einblenden.ppm. Es ruft die benutzerdefinierte Funktion quader.ppm auf

```
// quader_einblenden.ppm verwendet die benutzerdefinierte
Funktion quader.ppm im Macro-Ordner.
x = Parameter("Größe", 5, LINEAR, GreaterThan(0));
r1 = Parameter("r1", 0.5, LINEAR, GreaterThan(0));
```

```
b0 = Box(0, 0, 0, x, x, x);
b1 = G3Fillet(b0, Point(x/2, 0, 0), Array(r1, r1*2));
Output(b1);
```

Der Speicherort ist bei der Verwendung von parametrischen Skripten als benutzerdefinierte Funktionen extrem wichtig. Wenn sich das im obigen Beispiel verwendete Skript quader_einblenden.ppm im Ordner D:\Symbole befindet, kann das Skript quader.ppm nur gefunden werden, wenn es sich im Ordner D:\Symbole\Macro befindet.

Liste der für parametrische Teile reservierten Wörter

PI LINEAR TEXT
ANGULAR MATERIAL FONT
COLOR CHECKBOX ITALIC
BOLD UNDERLINE BOX
ALLCAPS STRICKETHROUGH TOP
MIDDLE BOTTOM BASELINE
LEFT CENTER RIGHT
Call Array +
- * Div
Mod / -
sin cos tan
atan min max
** = ==
!= < >
<= >= &