

LLP: Łukasiewicz List Processor

LLP soll eine minimalistische formale Sprache zur semantischen Auszeichnung von Texten, zur funktionalen Formulierung von Algorithmen und zur generatorischen Beschreibung von Diagrammen, Bildern und Fräskopfbahnen werden.

Grundlagen

Runen als Präfix

Alle für den Parser unterscheidbaren Strukturen erhalten ein Präfix in Form einer nordischen **Rune**.

Die *Runen* werden in keiner heute mehr existierenden Sprache genutzt. Damit sind die Präfixe, durch die Sprachstrukturen kenntlich werden, eindeutig von Textdaten unterscheidbar.

Kommentare +


+ schließt den Rest vom Parsen aus. Damit können nach + beliebige Kommentare notiert werden.

Literale elementarer Datentypen

Präfixe für die Notation von Zahlenwerten

Eine Gleitpunktzahl wie **3.14** ist eine kulturspezifische Notation (**en-US**).

Um die Notation von Zahlenwert von einer textuellen und kulturspezifischen Präsentation in einer Sprache zu unterscheiden, werden diese in **LLP** stets durch ein spezielles *Präfix* explizit gekennzeichnet.

 Zahlen können wie z.B. $\mathbb{R} \text{ _Zähler_ _Nenner_}$ eine listenartige Struktur darstellen, sind aber keine Listen. Die einzelnen Partikel wie im Beispiel _Zähler_ und _Nenner_ dürfen nur Konstanten sein, wie $\mathbb{R} \text{ 1 2}$, jedoch keine Ausdrücke!

Nummerische Datentypen

Die Notationsformen für Zahlenwerte haben Beschränkungen bezüglich der Genauigkeit. Deshalb korrespondieren die Notationsformen auch mit Teilmengen von \mathbb{Q} . Diese Teilmengen Werden

Nummerische Datentypen genannt.

Die numerischen Datentypen werden durch Kombination des speziellen Präfixes für eine Notation (z.B. **K**) mit dem allgemeinen Datentyp- Schalter **Y** verbunden zum Datentyp Symbol **KY**.

Y schaltet allgemein die Evaluierung einer Liste in die Evaluierung einer Typdeklaration um.

Kardinalzahlen K

K ist das Präfix für ganze Zahlen:

K 1	⇔ 1
K -123	⇔ -123
K 16 AFD	⇔ nat. Zahl zur Basis 16 (hex)
K 2 L00LLL	⇔ nat. Zahl zur Basis 2 (dual)
K M	⇔ + Unendlich
K -M	⇔ - Unendlich

KY ist der Datentyp für Kardinalzahlen.

Gebrochen Rationale Zahlen R

R ist das Präfix für gebrochen rationale Zahlen. Diese bestehen aus einem *Nenner* und einem *Zähler*, getrennt durch ein Leerzeichen:

1. R _Zähler_ hier ist der Nenner stets 1
2. R _Zähler_ _Nenner_
3. R _Ganzzahlig_ _Zähler_ _Nenner_

Beispiele:

R 2	⇔ 2/1 = 2.0
R 1 2	⇔ 1/2 = 0.5
R 1 2 3	⇔ 1 2/3 = 1.666
R -4 16	⇔ -4/16 = -0.25

Die rationalen Zahlen können z.B. als Zoll- Maße genutzt werden

RY ist der Datentyp für gebrochen rationale Zahlen.

Gleitpunktzahlen ℱ

ℱ ist das Präfix für rationale Zahlen in der Gleitpunkt- Darstellung. Vor- und Nachkomma- Stellen bilden die beiden Elemente einer Liste. Kulturspezifische Separatoren wie , oder . sind damit überwunden.

ℱ 3 ⇔ 3.0

ℱ 3 14 ⇔ 3.14

ℱ -2 72 ⇔ -2.72

ℱ -2 72 3 ⇔ -2.72e3 = -2720

ℱΥ ist der Datentyp für Gleitpunkt- Zahlen.

Boolsche Werte ℬ

ℬ ist das Präfix für boolsche Werte. Die beiden möglichen boolschen Werte werden durch die Namen **true** und **false** ausgedrückt:

ℬ true ⇔ True

ℬ false ⇔ False

ℬΥ ist der Datentyp für boolsche Werte.

Namensreferenzen ℵ

ℵ ist das Präfix für eine *NamingID*. Eine *NamingID* ist ein eindeutiger Schlüssel zu Identifizierung eines Namenscontainers.

Beispiele:

ℵ milProgramm ⇔ Referenz auf den Namenscontainer, der für Fräsenprogramme steht.

ℵΥ ist der Datentyp für Namensreferenzen.

Hierarchieen ℶ

ℶ ist das Präfix eines Pfades in einer Hierarchie. Der Pfad muß durch ein Listenendsymbol ↵ abgeschlossen werden.

ℶ K23 K10 K15 ↵ ⇔ Kann z.B. eine Versionsnummer mit den drei Hierarchieebnen *Hauptversion*, *Nebenversion*, *Buildnummer* darstellen. Oder die Uhrzeit **23:10:15**. Oder das Datum **15.10.2023**.

`ℙ ℘ millingMachine ℘ circelMilling ℘ millDisc 1 ⇔ Pfad in einem Namensraum`

`ℙ℘` ist der Datentyp für Hierarchien.

Strings `℔`

Strings sind Listen aus beliebigen Zeichen. Sie können auch Leerzeichen enthalten.

Strings, die keine Leerzeichen enthalten, können direkt notiert werden.

+ geschlossener String, enthält keine Leerzeichen

`Hallo`

+ geschlossener Strings, die einzelne Hierarchieebenen benennen

`ℙ All Galaxieen Andromeda 1`

Enthalten *Strings* Leerzeichen, dann müssen sie in eine **B-Liste**: `℔ ... 1` gesetzt werden.

+ String aus mehreren Wörtern

`℔ Hallo Welt1`

+ Komplexe Texte als String

`℔`

Mit **B- Liste** Strings können auch ****Markdown**** formatierte Texte geschrieben werden.

So wird **Text** und **Logik** vollständig vermischt.

`1`

`℔℘` ist der Datentyp für Strings.

Stringinterpolation

Werden in einem String Namensreferenzen eingesetzt, die beim Abruf des Strings evaluiert werden, dann liegt eine Stringinterpolation vor.

Sei `⋈ attrib schöne` eine Namensbindung. Dann kann eine Stringinterpolation wie folgt definiert werden:

`℔ Hallo ⋈* attrib Welt 1`

Diese wird dann evaluiert zu:

Arrays 11

Arrays sind Listen von Werten gleichen elementaren Typs. Sie stellen komplexe, zusammengesetzte Werte dar wie z.B. Real- und Imaginärteil einer komplexen Zahl, oder die Komponenten eines Vektors.

Arrays werden stets mittels 11 eingeleitet, und mittels 1 beendet werden. Das erste Element von links legt dabei den Datentyp für alle anderen Elemente des Array verbindlich fest. Diese Regel unterscheidet das *Array* im wesentlichen vom *String* (neben den unterschiedlichen Präfixen).

+ Array mit den ersten fünf Primzahlen

11 K2 K3 K5 K7 K11 1

+ Fehlerhaft aufgebautes Array: Alle Elemente müssen vom gleichen Typ sein

11 K2 F3 K5 K7 K11 1 ⇒ ERROR!

11 11 1 steht für ein Array aus beliebig vielen ganzen Zahlen.

11 11 K3 1 steht für ein Array aus drei ganzen Zahlen.

11 11 1 steht für ein Array aus beliebig vielen Namensreferenzen.

11 Hred Hgreen Hblue 1 steht für einen Aufzählungstyp/Set: Eingesetzt werden dürfen nur die im Array aufgelistete Werte.

Zugriff auf Array Elemente

Auf einzelne Elemente eines Arrays kann mittels Operator 11 _array_ _index_ zugegriffen werden.

Dieser hat als Parameter den 0 basierte Index und das *Array*, aus dem der Wert zu entnehmen ist.

Soll im Falle eines Zugriffs auf ein nicht vorhandenes Element durch einen zu kleinen, oder zu großen Index keine Ausnahme, sondern eine benutzerdefinierte Fehlerbehandlung starten, dann ist der 11 Operator einzusetzen: 11 _array_ _index_ _errIndexOutOfRangeHandler_ .

Abstraktion durch benennen von Werten mittels 11 Operator

Werte können an einen *Namen* mittels dem **Bind** Operator 11 gebunden werden. Über diesen Namen wird der Wert dann referenziert und abgerufen.

⌘ `_NameAsString_ _Wert_` bindet den Wert an einen Namen, der nur im Kontext der aktuellen *LLP* Datei eindeutig ist.

⌘ `_MonikerForNamingIdAsString_ ⌘ _NamingID_` bindet lokal in der *LLP* Datei einen Namen (Moniker) an eine *NamingId*. Die *Naming ID* ist dabei ein 64bit Wert, der für einen global gültigen Namen steht (Namenskontainer).

+ Konstante PI definieren

⌘ `PI ⌘ 3 14`

+ Den lokal gültigen Namen PI an eine global gültige Naming ID binden.

⌘ `PI ⌘ K 16 7ABC123`

+ Liste der ersten fünf Primzahlen an einen Namen binden

⌘ `ersteFünfPrimzahlen ⌘ K2 K3 K5 K7 K11 1`

Die Bindung eines Namens an einen Wert kann auch als **Attribut Wertepaar** betrachtet werden!

Zugriff Auf den Werte, die an Namen gebunden sind mittels ⌘*

Wurde an einen Namen ein Wert gebunden, dann kann überall, wo normalerweise der Wert eingesetzt wird, der Name eingesetzt werden, dem aber der **Replace Name by Value** Operator ⌘* vorangesetzt werden muss:

+ Konstante PI definieren

⌘ `PI ⌘ 3 14`

+ Den Wert von ****PI**** an den synonymen Namen ****pie**** binden

⌘ `pie ⌘* PI`

+ Den Wert der globalen mit Naming ID definierten Konstante ****PI**** an den synonymen Namen ****pie****

⌘ `pie ⌘* ⌘ K 16 7ABC123I`

Namensraum- Listen ⌘ ... ⌘ ... 1

Eine Menge von *Bind* Operationen können in Listen zusammengefasst werden. Innerhalb einer solchen Liste darf ein bestimmter Name stets nur einmal an einen Wert gebunden werden.

Die Liste selber wird dann ebenfalls mittels Bind an einen Namen gebunden. So entsteht ein *Namensraum*, der eine Untermenge benannter Werte darstellt.

+ Namensraum mathematischer Konstanten

⌘MathConst

└

⌘PI F3 14

⌘e F2 72

└

Für den Zugriff auf die Werte in der benannten Liste kann wieder mittels **Replace by** Operator ⌘* benutzt werden. In diesem Fall sind die Namen jedoch als Hierarchie anzugeben:

⌘* ⌘ _NameListe_ _NameAttribut_ └

+ Organisation einer Mathematischen Bibliothek

⌘Math

└

⌘Const

└

⌘PI F3 14

⌘e F2 72

└

⌘BasicFunctions

└

+ Naming- IDs der math. Grundrechenarten werden an lokale Namen gebunden

⌘add ⌘* H K 16 ADDADD

⌘sub ⌘* H K 16 DE2323

└

└

+ Zugriff auf PI

⌘* ⌘ Math Const PI └

Semantische Referenzen zwischen Namensraum- Listen

Sei **milDiscCircularCenterOfDiskX** ein Namenscontainer, der die X- Koordinaten des Mittelpunktes einer auszufräsenden Kreisscheibe beschreibt. Dieser stehe mit anderen Namenscontainern in folgenden semantischen Beziehungen:



Die semantischen Beziehungen werden durch den ternären Operator **⌘** dargestellt:

⌘ _NID_Referring_ _NID_SemRefName_ _NID_Related_

⌘ milDiscCircular isInstanceOf milProgramm

referring	sem Rel	referred

Abfragen der semantisch referenzierten Instanz

⌘↑ _NID_Referring_ _NID_SemRefName_

Beispiel: Bestimmen, mit wem alles **milDiscCircular** in der semantischen Beziehung **isInstanceOf** steht:

⌘↑ **⌘** milDiscCircular **⌘** isInstanceOf

Abfragen der semantisch referenzierenden Instanzen

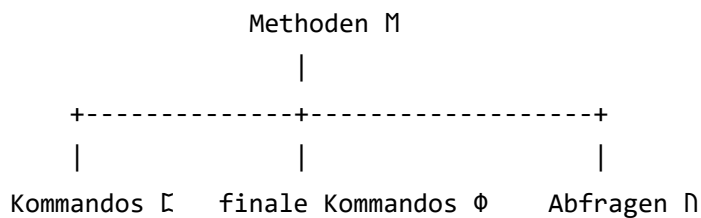
⌘↑ _NID_SemRefName_ _NID_Related_

Beispiel: Bestimmen der Instanzen von **milProgram**:

⌘↑ instanceof milProgram

Methoden M: Kommandos und Abfragen

Methoden sind ein Oberbegriff für den Zustand des Systems verändernde *Kommandos*, und *Abfragen* auf dem Systemzustand selbst:

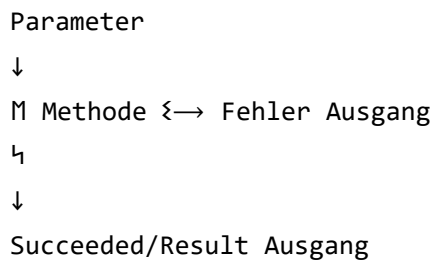


Datenflussgraphen

Kommandos und Abfragen werden mittels *Parameter* vor der Ausführung parametrisiert. Nach der Ausführung gibt es zwei mögliche Zustände:

1. Die Methode konnte erfolgreich ausgeführt werden.
2. Beim Ausführen der Methode kam es zu einem Problem

Dies führt zu folgendem allgemeinen Datenfluss- Graphen:



ξ und ℓ werden *Ausgänge* genannt.

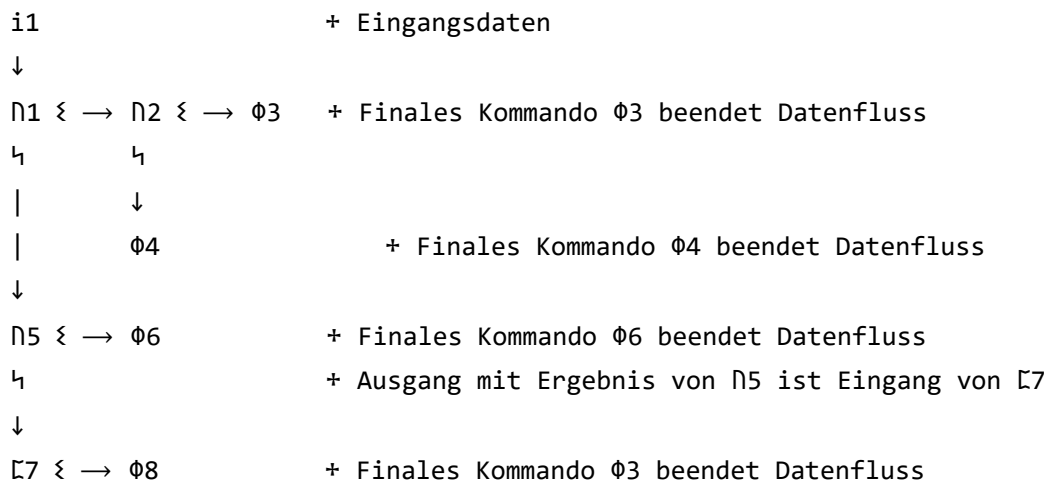
Methoden müssen nicht alle Ausgänge implementieren.

Implementiert eine Methode den Ausgang ℓ, dann ist sie eine **Abfrage** N.

Implementiert eine Methode den Ausgang ℓ nicht, dann ist sie eine **Kommandos** L.

Wird weder ξ noch ℓ implementiert, dann ist es ein finales **Kommando** Φ.

Die beiden Ausgänge einer Methode können auf die Eingänge (Parameterlisten) nachfolgender Methoden geschaltet werden, so daß ein Netz entsteht, durch das die Daten fließen:



Präfixe für Methodenoperatoren

Operator	Bedeutung
M <i>name</i>	Präfix, Definition einer benannten Methode
M•	Präfix, Definition einer anonymen Methode
MΥ	Präfix einer Methodentypen- Signatur
M* <i>name</i>	Präfix einer Methodenreferenz
M↑ <i>name</i>	Präfix eines Aufrufes einer benannten Methode
M↑•	Präfix eines Aufrufes einer anonymen Methode

Parameterlisten von Methoden

Die Mengen der möglichen Eingangsdaten/Parameter einer Methode werden durch die Parameterlisten definiert.

Methodentypen M Υ

Wie bei den elementaren Datentypen können auch Methoden klassifiziert werden. Dabei ist der Aufbau der Parameterliste entscheidend.

M Υ ist das Präfix für einen Methodentyp. Diesem folgt eine Liste von Methodenparameter-Typdeklarationen:

M ⋈ *paramName1* *Typ*₁ ... ⋈ *paramNameN* *Typ*_N ¶

Definition von Methoden

Eine Methodendefinition startet mit dem Präfix **M**, dem folgende Strukturen folgen:

1. Methodenname
2. Parameterliste
3. Ausgänge mit Methodentypen der einsetzbaren Folge- Methoden
4. Implementierende Sequenz von Operationen

```
M _name_ ⋈ _paramName1_ _Typ1_ ... ⋈ _paramNameN_ _TypN_ ¶
{ T ⋈ _paramName1_ _Typ1_ ... ⋈ _paramNameM_ _TypM_ ¶
  h T ⋈ _paramName1_ _Typ1_ ... ⋈ _paramNameM_ _TypP_ ¶
  ♦ _Methodenaufruf_etc_ + 1. Schritt in der Sequenz
  ...
  ♦ _Methodenaufruf_etc_ + N. Schritt in der Sequenz
  ♦♦ + Sequenzende
```

Finales Kommando ⌀

Ein **finales Kommando** ist eine parametrierbare Methode, die weder eine Fehlermeldung, noch ein Ergebnis zurückliefert. Es findet lediglich eine Änderung des Systemzustandes auf Basis der übergebenen Parameter statt.

Finale Kommandos haben das Präfix **⌀**

Beispiele für *finale Kommandos* sind z.B. das reguläre Programmende und der vorzeitige Programmabbruch.

Parameter

↓

⌀ Finales Kommando

+ Finales Kommando in LLP aufrufen

↑⌀ namensReferenz ⋈param1 wert1 ... ⋈paramN wertN ¶

+ Konkretes Beispiel: Text auf der Log- Konsole ausgeben

⌀↑logConsole ⋈txt ⋈ Es wurden ⋈*count Datensätze gelesen¶ ¶

Kommandos ⚡

Kommandos haben das Präfix ⚡, und verändern den Systemzustand (z.B. Insert- Operation in einer DB- Tabelle). Ein Ergebnis liefern sie nicht, können aber scheitern, und haben folglich einen Fehler-Handler ⚡.

Parameter

↓

⚡ Command ⚡→ Error Output

+ Kommando in LLP aufrufen

↑⚡ namensReferenz ⚡param1 wert1 ... ⚡paramN wertN

⚡ _Referenz_auf_Funktion_mit_Fehlerbehandlung_

¶

Abfragen ⌚

Abfragen haben das Präfix ⌚. Sie liefern Informationen über den aktuellen Systemzustand. Verändert wird der Systemzustand durch eine Abfrage explizit nicht.

Das Ergebnis einer Abfrage wird im Result- Output ausgegeben.

Parameter

↓

⌚ Query ⚡→ Error Output

↳

↓

Succeeded/Query Result Output

Von der Laufzeitumgebung bereitgestellte Methoden

Die Laufzeitumgebung hat bereits eine Reihe von Methoden vordefiniert und Implementiert. Diese stammen aus folgenden Bereichen:

Datenstrom- orientierte Ausgabe

Wie in jeder Programmiersprache gibt es auch in LLP eine elementare Funktion zur Ausgabe von Daten in Datenströme:

Φout ⚡ostream _name output_Stream_ ⚡txt ⚡ hier den auszugebenden Text¶ ¶

Logs, Fehlerlogs

1. Fehlerlog `logErr txt` hier die logmeldung
2. Allgemeiner Log `log txt` hier die logmeldung
3. Grundrechenarten wie
`nadd a K1 b K2` hier Methode_referenzieren_die_das_Ergebnis_weiterverarbeitet
4. Basisfunktionen wie Potenzen, Wurzeln,
5. grundlegende wissenschaftliche Funktionen wie Trigonometrische Fkt.
6. Zeichenketten- Funktionen wie Concatentation, String- Interpolation, Split, Trim, SubString

Parameter

♦ ist das Präfix für einen *Parameter*. *Parameter* bestehen im allgemeinen immer aus einem Parameternamen und einem Wert, der an den Parameter gebunden ist: ♦ `_paramName_ _paramValue_`

Wird der *paramName* durch eine *NamingID* definiert, dann kann mittels semantischer Referenzen im Namenscontainer der Datentyp eines Funktionsparameters implizit festgelegt werden.

Der Parameterwert kann direkt gesetzt, durch einen Funktionsaufruf errechnet, aus einer Eigenschaft einer Instanz referenziert oder durch einen Platzhalter offen gehalten werden. Letzteres erfolgt, wenn die Funktion eine *Implementierung* für die Berechnung des Funktionswertes in dem *Return* Abschnitt enthält:

```

M Hadd
  ◊ A F3 14
  ◊ B F2 72
¶

++ Eine Instanz, die eine Punktkoordinate darstellt
X P1
  ++ die folgende semantische Beziehung hat den Charakter einer Typdeklaration
  X HinstanceOf HGeometricPoint

  ++ Koordinaten des Punktes
  ◊ Hpx F3.14
  ◊ Hpy F2.72
¶

++ Addiert die Werte der Koordinaten von P1
M Hadd
  ◊ A ↑ X P1 ◊ Hpx
  ◊ B ↑ X P1 ◊ Hpy
¶

++ Funktion mit einer Implementierung λ

M radiusOfP
  ◊ Hpx X
  ◊ Hpy X
  λ ↑M• HSQRT ↑M•• Hadd
    ↑M•SQU ↑◊ Hpx
    ↑M•SQU ↑◊ Hpy
¶

```

Alternativ könnte man den Datentyp eines Parameters durch eine Default- Wert eines elementaren Datentypen festlegen: ◊ CX F_ ⇔ CX ist vom Typ Gleitkommazahl

Vereinfachte Methoden/Funktionsdefinitionen

M• definiert explizit eine einstellige Funktion. Diese hat genau einen Parameter:

M• _funktionsName_ _parameter1_

M• definiert explizit eine zweistellige Funktion. Diese hat genau zwei Parameter: M• *funktionenName*
parameter1 parameter2

usw..

Platzhalter x für Parameterwerte

x ist ein Platzhalter, der anstelle eines Parameterwertes notiert werden kann.

Aufruf von Funktionen

Funktionen werden mit ↑ (Return) aufgerufen, und liefert den Funktionswert. Nach dem CQR Pattern verändern Funktionen den inneren Zustand nicht.

Beispiele:

```
++ Methode, die keinen Parameter hat (0 Stellig): Stoppt die Fräse  
M HmilStop 1
```

```
++ Funktion, die keinen Parameter hat (0 Stellig): liefert die aktuelle Position X  
↑ M HmilCurrentPosX 1
```

```
++ explizit zweistellige Funktion mit zwei Parametern: liefert die Summe der beiden Gleitpunktz  
↑ M• Hadd F0.1 F1.3
```

Instanzen

x ist das Prefix für eine *Instanz*. Eine Instanz beschreibt ein Objekt aus dem Weltausschnitt, der durch das **LLP** Programm modelliert wird.

Instanzen beginnen stets mit einem Namen. Diesem schließt sich eine Auflistung von *Eigenschaften*, *Methoden* und *Funktionen*.

Die *Eigenschaften* definieren den *inneren Zustand* eines Objektes. Sie werden als Attribut- Wertpaare aufgelistet.

Methoden ermöglichen das Ändern des *inneren Zustandes*.

Beispiel:

Einen Namingcontainer referenzieren

Sei **milDiscCircular** ein Namenscontainer, der eine Familie von Fräsprogrammen benennt, die Kresischeiben aus einer flachen Platte fräsen. Dann kann dieser Namenscontainer wie folgt referenziert werden:

```
⌘ milDiscCircular
```

Diese Referenz kann selber zur Benennung von LLP Strukturen wie Instanzen etc dienen.

Einen Namenscontainer definieren

```
M ⌘defNC
```

```
++ Benennung und Einordnung in Taxionomien
```

```
◇ ⌘ nid 0x1234567890
```

```
◇ ⌘ cnt milDiscCircular
```

```
◇ ⌘ basicNamespace TechTerms.Milling.MilProgs
```

```
++ Beschreibung in verschiedenen Sprachen
```

```
◇ ⌘ lngDE 'Ein Fräsprogramm zum erstellen einer Kreisscheibe auf einer Platte'
```

```
◇ ⌘ lngEN 'A milling Program for milling a circular disc.'
```

```
++ Semantische Beziehungen definieren
```

```
⌘ ⌘ instanceof ⌘ namingContainers
```

```
⌘ ⌘ instanceof ⌘ milProgram
```

```
⌘
```

Die Methode erzeugt in der Laufzeitumgebung einen Instanz mit der **NID** 0x1234567890.

Zugriff auf die Eigenschaften einer Instanz, z.B. Namenscontainer

Auf die Eigenschaften von Instanzen oder Parameter von Methodenblöcken kann mittels *getter* zugegriffen werden. Achtung: Eigenschaftsnamen innerhalb von Methoden oder Funktionen sind stets eindeutig.

++ Getter innerhalb eines Instanz oder Methodenblockes

< ♦ _PropertyName_ M _ErrorHandlerIfAccessToPropFails_

++ Getter, der eine Property einer Instanz oder Methode adressiert

< _InstanceOrMethodName_ ♦ _PropertyName_ M _ErrorHandlerIfAccessToPropFails_

M _ErrorHandlerIfAccessToPropFails_ verweist auf eine Methode, die aufgerufen wird, wenn der Zugriff auf die Eigenschaft zur Laufzeit fehlschlägt. Z.B. weil die referenzierte Eigenschaft nicht existiert. In diesem Fall ist stets ein Default- Wert zurückzugeben, so dass der Ausdruck, in dem der Getter steht evaluiert werden kann.

Beispiele:

++ Liefert den Wert der Eigenschaft ♦ N lngDE der Instanz milDiscCircular

< ✕ milDiscCircular ♦ N lngDE M N errHndLngDoesNotExists

✕ 'Berechnung 1'

♦ 'liste Primzahlen' ⌘ K2 K3 K5 K7 K11 ↵

♦ 'erste Primzahl' ⌘ K1 < ♦ 'liste Primzahlen' M N errPopDoesNotExistsHndl M N errOutOf

Ein weiteres Beispiel ist die Beschreibung eines Namenscontainers in einer Wunsch- Sprache abrufen:

M N ConsoleWriteLine ↑ < ✕ milDiscCircular ♦ N lngDE M N errHndLngDoesNotExists

Semantische Referenzen ausdrücken

Sei **milDiscCircularCenterOfDiskX** ein Namenscontainer, der die X- Koordinaten des Mittelpunktes einer auszufräsenden Kreisscheibe beschreibt. Dieser stehe mit anderen Namenscontainern in folgenden semantischen Beziehungen:



Die semantischen Beziehungen können z.B. durch Funktionsausdrücke dargestellt werden:

```

f H isInstanceOf
  ♦ HsemRefReferring H milDiscCircular
  ↑ H milProgram
  
```

Eine Kurzform für diese Definition semantischer Referenzen ist sinnvoll. Sei \mathbb{X} ein neues Präfix für semantische Referenzen. Dann kann eine semantische Referenz definiert werden durch:

```

 $\mathbb{X}$  _NID_Referring_ _NID_SemRefName_ _NID_Related_
  
```

Das ist die Kurzform für

```

f _NID _SemRefName_
  ♦ HsemRefReferring _NID_Referring_
  ↑ _NID_Related_
  
```

Damit kann das obige Beispiel vereinfacht werden zu:

```

 $\mathbb{X}$  H milDiscCircular H isInstanceOf H milProgram
  
```

Abfragen der semantisch referenzierten Instanz

```
⌘ ↑ ℵ _NID_Referring_ ℵ _NID_SemRefName_
```

Beispiel: Bestimmen, mit wem alles **milDiscCircular** in der semantischen Beziehung **isInstanceOf** steht:

```
⌘ ↑ ℵ milDiscCircular ℵ isInstanceOf
```

Abfragen der semantisch referenzierenden Instanzen

```
⌘ ↑ ℵ _NID_SemRefName_ ℵ _NID_Related_
```

Beispiel: Bestimmen der Instanzen von **milProgram**:

```
⌘ ↑ ℵ isInstanceOf ℵ milProgram
```

Fräsprogramm für einen Kreis

```
⌘ ℵ milCirc
  ◇ ℵ Cx ↑ M• ℵ measureDistance F 0.0 ℵ mm
  ◇ ℵ Cy ↑ M• ℵ measureDistance F 0.0 ℵ mm
  ◇ ℵ Cr ↑ M• ℵ measureDistance F 100.0 ℵ mm
M ℵ Next
  ◇ r _
  ↑ ⌘ ℵ milCircNext
    ◇ ℵ milCircCx < ⌘ ℵ milCirc ◇ Cx
    ◇ ℵ milCircCy < ⌘ ℵ milCirc ◇ Cy
    ◇ ℵ milCircRadius ↑ M• ℵ ADD F 0.5 < ◇ r _
```

Interaktives Parsen von LLP

Es ist ein Editor für LLP zu implementieren, der den Benutzer aktiv bei der Eingabe unterstützt.

Nach jedem vollständig eingegeben Wort kann z.B. der Parser gestartet werden.

Z.B. folgende Sitzung:

⌘ _

Der Parser erkennt das Prefix für semantische Beziehungen. Nun kann die Definition oder die Abfrage einer semantischen Beziehung folgen.

⌘ ↑ _

> [#1] ↑ - semantische Beziehung abfragen

> [#2] ⌘ - semantische Beziehung definieren: _NID_Referring

Nachdem [#1] gewählt wurde, ist nun eine der möglichen semantischen Beziehungen auszuwählen

⌘ ↑ ⌘ isInstanceOf

> [#1] ⌘ isPartOfSemContext

> [#2] ⌘ isInstanceOf

> [#3] ⌘ isPartOf

> [#4] ⌘ isSubTermOf

> [#5] ⌘ isSubNamespace

Nachdem [#2] gewählt wurde, gibt es eine große Auswahl von Namenscontainern, die Klassennamen von Klassen darstellen, mit denen andere Namenscontainer in der Beziehung **isInstanceOf** stehen können. Hier gibt es verschiedene Strategien, um den gesuchten Namenscontainer des Klassennamens zu finden:

1. Über den Namensraum- Pfad zur Naming ID des gesuchten Namenscontainers navigieren.
 - i. Es werden nur Namensraum- Pfade unterstützt, die Klassennamen enthalten
 - ii. Es werden alle Namensraumpfade unterstützt. In einem Namensraum werden nur Namenscontainer von Klassennamen angezeigt.
2. Über ein Autocomplete- Textcontrol, das nur die CNT's von Klassennamen unterstützt, den CNT auswählen lassen.

Sei Variante 2 der Standard- Modus. In Variante 1 kann bei Bedarf umgeschaltet werden:

⌘ ↑ ⌘ isInstanceOf ⌘ milProgram

> [#1] Select Naming- Container with class name via Namespace