FriCAS Tutorium

Franz Lehner

Version 0.6

Inhaltsverzeichnis

Date: 10. Mai 2010.

1. FriCAS als Taschenrechner [1, Kap. 1]

Man kann FriCAS als erweiterten Taschenrechner verwenden, der die verschiedensten Zahlensysteme kennt.

1.1. Ein- und Ausschalten. FriCAS ist ein Textmodusprogramm und wird gestartet mit `fricas`. Es existiert ein Emacs-Modus, der die Rechenergebnisse mit Hilfe von LaTeXautomatisch setzen kann. Start mit `efricas`. Beenden einer Sitzung mit `)quit`.

1.2. Eingabe von Rechnungen. Nach Eingabe einer Rechenaufgabe und abschließendem **Enter** werden die eingegebenen Daten interpretiert und das Resultat berechnet.

2a ⟨*tutor.input* 2a⟩≡ 2b▷

```
1+1
```

(1) 2

Type: PositiveInteger

Dabei wird auch der **Typ** des Ergebnisses angezeigt. Jedes Objekt hat einen fest definierten Typ und lebt in einem *Domain*.

2b ⟨*tutor.input* 2a⟩+≡ ◁2a 2c▷

```
1/2
```

(2) $\frac{1}{2}$

Type: Fraction(Integer)

2c ⟨*tutor.input* 2a⟩+≡ ◁2b 2d▷

```
2^9
```

(3) 512

Type: PositiveInteger

1.3. Langzahlarithmetik.

2d ⟨*tutor.input* 2a⟩+≡ ◁2c 3a▷

```
2^1000
```

107150860718626732094842504906000181056140481170553360744375038837035105112493612249319837881

Type: PositiveInteger

1.4. Algebraische Zahlen.

3a ⟨*tutor.input* 2a⟩+≡ ◁2d 3b▷

```
sqrt 2
```

(4) $\sqrt{2}$

Type: AlgebraicNumber

Es müssen nicht unbedingt Klammern um das Argument gesetzt werden, wenn letzteres eindeutig gefunden werden kann, siehe jedoch unten Beispiel (11) und (13).

1.5. Gleitkommazahlen [1, Kap. 9.31].

3b ⟨*tutor.input* 2a⟩+≡ ◁3a 3c▷

```
sqrt(2.0)
```

(5) 1.4142135623 730950488

Type: Float

3c ⟨*tutor.input* 2a⟩+≡ ◁3b 3d▷

```
numeric sqrt 2
```

(6) 1.4142135623 730950488

Type: Float

Rechengenauigkeit wird mit `digits` festgelegt, Ausgabegenauigkeit mit `outputFixed`, `outputFloating` etc.:

3d ⟨*tutor.input* 2a⟩+≡ ◁3c 3e▷

```
digits(10)
outputFixed(50)
numeric sqrt 2
```

(7) 1.41421356238424777984619140625000000000000000000000

Type: Float

1.6. Komplexe Zahlen.

3e ⟨*tutor.input* 2a⟩+≡ ◁3d 3f▷

```
%i^2
```

(8) -1

Type: Complex(Integer)

Real- und Imaginärteil

3f ⟨*tutor.input* 2a⟩+≡ ◁3e 3g▷

```
real((1-%i)/(2+%i))
```

(9) $\frac{1}{5}$

Type: Fraction(Integer)

3g ⟨*tutor.input* 2a⟩+≡ ◁3f 4a▷

```
imag((1-%i)/(2+%i))
```

(10) $$-\frac{3}{5}$$

Type: Fraction(Integer)

AlgebraicNumber

4a ⟨*tutor.input* 2a⟩+≡ ◁3g 4b▷

```
sqrt (-4)
```

(11) $$2\sqrt{-1}$$

Type: AlgebraicNumber

Was passiert, wenn man die Klammern wegläßt?

1.7. Exakte Auswertung mathematischer Funktionen.

4b ⟨*tutor.input* 2a⟩+≡ ◁4a 4c▷

```
sin (%pi/2)
```

(12) 1

Type: Expression(Integer)

Vorsicht!

4c ⟨*tutor.input* 2a⟩+≡ ◁4b 4d▷

```
sin %pi/2
```

(13) 0

Type: Expression(Integer)

1.8. Einige Konstante.

4d ⟨*tutor.input* 2a⟩+≡ ◁4c 5a▷

```
%i,%e, %pi, %infinity, %plusInfinity, %minusInfinity
```

(14) $$[i,\ e,\ \pi,\ \infty,\ +\infty,\ -\infty]$$

Type: Tuple(OnePointCompletion(OrderedCompletion(Expression(Complex(Integer)))))

2. Variable und Symbole [1, Kap. 1.3]

2.1. Typensystem. FriCAS ist ein *streng typisiertes* Computer*algebra*system. Jedes Objekt hat einen Typ und gehört zu einem *Domain*. *Operationen* sind Funktionen, die von einem Domain A in ein anderes Domain B abbilden, und die je nach Ursprungs- und Zieldomain verschiedene Dinge tun. Zum Beispiel berechnet `factor n` die Primzahlzerlegung, wenn `n` eine natürliche Zahl ist, und `factor p` eine Zerlegung in irreduzible Faktoren, wen `p` ein Polynom ist. Operationen können also den gleichen Namen haben und sind erst durch die *Signatur* $f : A \to B$ eindeutig bestimmt. Der *Interpreter* wählt je nach Typ des Argumente jeweils eine passende Signatur für eine Operation aus, indem er z.T. auch implizit Typkonversionen durchführt.

2.2. Abkürzungen. Aus internen technischen Gründen ist jedes Domain und jede Kategorie durch eine maximal 7 Zeichen lange Abkürzung ansprechbar, z.B. `INT`=`Integer`, `UP`=`UnivariatePolynomial`, `SMP`=`SparseMultivariatePolynomial`.

2.3. Variable. Man kann an Variable Objekte zuweisen, wobei streng zwischen Variablen und Symbolen unterschieden werden muss. Jede Eingabezeile wird interpretiert und in ein Objekt umgewandelt. Wenn man eine "Variable" eingibt, der noch kein Objekt zugewiesen wurde, wird sie als spezielle "leere Variable" interpretiert.

5a ⟨*tutor.input* 2a⟩+≡ ◁4d 5b▷

```
x
```

(15) x

```
                                                              Type: Variable(x)
```

Zuweisung eines Werts erfolgt mit `:=`

5b ⟨*tutor.input* 2a⟩+≡ ◁5a 5c▷

```
x:=3
```

(16) 3

```
                                                          Type: PositiveInteger
```

Man kann auch festlegen, von welchem Typ die Variable sein soll.

5c ⟨*tutor.input* 2a⟩+≡ ◁5b 5d▷

```
x:Integer := 3
```

(17) 3

```
                                                                  Type: Integer
```

Man kann auch im Vorhinein Variablen *deklarieren*:

5d ⟨*tutor.input* 2a⟩+≡ ◁5c 5e▷

```
y:PositiveInteger
```

(18)

```
                                                                    Type: Void
```

5e ⟨*tutor.input* 2a⟩+≡ ◁5d 5f▷

```
y
```

```
   y is declared as being in PositiveInteger but has not been given a value.
```

5f ⟨*tutor.input* 2a⟩+≡ ◁5e 5g▷

```
y:=-1
```

```
   Cannot convert right-hand side of assignment
   - 1

   to an object of the type PositiveInteger of the left-hand side.
```

Variable und Symbole sind streng zu unterscheiden. Die "leere Variable" existiert immer noch:

5g ⟨*tutor.input* 2a⟩+≡ ◁5f 5h▷

```
'x
```

(19) x

```
                                                              Type: Variable(x)
```

und ebenso das Symbol.

5h ⟨*tutor.input* 2a⟩+≡ ◁5g 6a▷

```
'x::Symbol
```

(20) x

Type: Symbol

Zunächst wird wird Deklaration und Wert von `x` gelöscht:

6a ⟨*tutor.input* 2a⟩+≡ ◁5h 6b▷

```
)clear prop x
x
```

(21) x

Type: Variable(x)

6b ⟨*tutor.input* 2a⟩+≡ ◁6a 6c▷

```
p:= x+1
```

(22) $x + 1$

Type: Polynomial(Integer)

(es wird automatisch ein Polynom über dem Ring der ganzen Zahlen erzeugt).

6c ⟨*tutor.input* 2a⟩+≡ ◁6b 6d▷

```
x:=2
```

(23) 2

Type: PositiveInteger

Der Wert von `p` wird dadurch nicht beeinflußt.

6d ⟨*tutor.input* 2a⟩+≡ ◁6c 6e▷

```
p
```

(24) $x + 1$

Type: Polynomial(Integer)

Im Gegensatz zu typenlosen CAS wird auf dem Bildschirm wird immer nur eine `OutputForm` angezeigt, die nicht unbedingt alle Informationen über ein Objekt enthüllt. Objekte verschiedenen Typs können die gleiche `OutputForm` haben, z.B. kann `x` meinen: Symbol, Polynom, Expression Integer, ...

2.4. Beispiel: Alternative Darstellungen ganzer Zahlen.

6e ⟨*tutor.input* 2a⟩+≡ ◁6d 6f▷

```
n:=factor 20
```

(25) $2^2\ 5$

Type: Factored(Integer)

6f ⟨*tutor.input* 2a⟩+≡ ◁6e 6g▷

```
n+1
```

(26) 3 7

Type: Factored(Integer)

6g ⟨*tutor.input* 2a⟩+≡ ◁6f 7a▷

```
m:= roman 8
```

(27) $VIII$

```
                                                        Type: RomanNumeral
```

Römische Ziffern werden an anderer Stelle gebraucht (91). Bei Kombination verschiedener Typen wird wenn möglich ein passender Typ gewählt:

7a ⟨*tutor.input* 2a⟩+≡ ◁6g 7b▷

```
m+n
```

(28) $II^2\ VII$

```
                                          Type: Factored(RomanNumeral)
```

2.5. Restklassenringe.

7b ⟨*tutor.input* 2a⟩+≡ ◁7a 7c▷

```
Z12:=IntegerMod 12
```

(29) $IntegerMod(12)$

```
                                                          Type: Domain
```

Für Objekte aus diesem Ring wird automatisch die "richtige" Addition gewählt:

7c ⟨*tutor.input* 2a⟩+≡ ◁7b 7d▷

```
m:Z12:= 6;
m+m
```

(30) 0

```
                                                     Type: IntegerMod(12)
```

Der Unterschied ist deutlich:

7d ⟨*tutor.input* 2a⟩+≡ ◁7c 7e▷

```
(6^1000000)::Z12
```

(31) 0

```
                                                     Type: IntegerMod(12)
                  Time: 0.004 (IN) + 2.55 (EV) + 0.004 (OT) = 2.56 sec
```

7e ⟨*tutor.input* 2a⟩+≡ ◁7d 7f▷

```
m^1000000
```

(32) 0

```
                                                     Type: IntegerMod(12)
                                                           Time: 0 sec
```

Wenn p eine Primzahl ist, dann ist $\mathbf{Z}_p$ ein Körper, allerdings "weiß" der Bereich `IntegerMod` nichts davon:

7f ⟨*tutor.input* 2a⟩+≡ ◁7e 7g▷

```
Z29:=IntegerMod 29
Z29 has Field
```

(33) $false$

```
                                                        Type: Boolean
```

Es gibt für Primzahlen einen eigenen Typ:

7g ⟨*tutor.input* 2a⟩+≡ ◁7f 8a▷

```
Z29:=PrimeField 29
Z29 has Field
```

(34) $true$

```
                                                                Type: Boolean
```

Und darin kann man auch dividieren, wie es sich für einen Körper gehört:

8a ⟨*tutor.input* 2a⟩+≡ ◁7g 8b▷

```
z:Z29:=7;
z/2
```

(35) 18

```
                                                          Type: PrimeField(29)
```

2.6. Zusammengesetzte Typen. Polynome kann man über beliebigen Ringen definieren:

8b ⟨*tutor.input* 2a⟩+≡ ◁8a 8c▷

```
)sh UnivariatePolynomial )attributes
```

```
UnivariatePolynomial(x: Symbol,R: Ring)  is a domain constructor
Abbreviation for UnivariatePolynomial is UP
This constructor is exposed in this frame.
```

Als Beispiel der Ring der Polynome in x über dem Ring der ganzzahligen Polynome über y:

8c ⟨*tutor.input* 2a⟩+≡ ◁8b 8d▷

```
Zxy:=UnivariatePolynomial(x,UnivariatePolynomial(y,Integer));
p:Zxy := x+x*y+1
```

(36) $(y+1)\,x+1$

```
                  Type: UnivariatePolynomial(x,UnivariatePolynomial(y,Integer))
```

8d ⟨*tutor.input* 2a⟩+≡ ◁8c 8e▷

```
coefficients p
```

(37) $[y+1,\ 1]$

```
                                          Type: List(UnivariatePolynomial(y,Integer))
```

Es muß aber wirklich ein Ring sein:

8e ⟨*tutor.input* 2a⟩+≡ ◁8d 8f▷

```
Nx:=UnivariatePolynomial(x,NonNegativeInteger)
```

```
UnivariatePolynomial(x,NonNegativeInteger) is not a valid type.
```

Manchmal will man explizit ein Polynom in einer bestimmten Variablen:

8f ⟨*tutor.input* 2a⟩+≡ ◁8e 8g▷

```
p:UnivariatePolynomial (x,Integer) := x+1
```

(38) $x+1$

```
                                                  Type: UnivariatePolynomial(x,Integer)
```

Man kann sich auch vom Interpreter einen geeigneten Typ "berechnen" lassen:

8g ⟨*tutor.input* 2a⟩+≡ ◁8f 8h▷

```
aa:=[subscript(a,[i]) for i in 0..4]
```

(39) $[a_0,\ a_1,\ a_2,\ a_3,\ a_4]$

```
                                                            Type: List(Symbol)
```

8h ⟨*tutor.input* 2a⟩+≡ ◁8g 9a▷

```
q:UnivariatePolynomial (x,?) :=reduce(+, [a * x^k for a in aa for k in 0..4])
```

(40) $$a_4\ x^4 + a_3\ x^3 + a_2\ x^2 + a_1\ x + a_0$$

```
Type: UnivariatePolynomial(x,Polynomial(Integer))
```

2.7. Lineare Algebra [1, Kap. 9.59]**.** Für lineare Algebra über beliebigen Körpern (und Ringen) gibt es die Domains `Matrix` und `Vector`. Elemente werden mit `matrix` bzw. `vector` gebildet und es wird wie üblich der "kleinste" passende Ring gewählt:

9a ⟨*tutor.input* 2a⟩+≡ ◁8h 9b▷

```
A := matrix [[1,2],[3,4]]
```

(41) $$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
Type: Matrix(Integer)
```

9b ⟨*tutor.input* 2a⟩+≡ ◁9a 9c▷

```
b:= vector [1/2,-1]
```

(42) $$\left[\frac{1}{2}, -1\right]$$

```
Type: Vector(Fraction(Integer))
```

Matrizen und Vektoren können multipliziert werden:

9c ⟨*tutor.input* 2a⟩+≡ ◁9b 9d▷

```
A*b
```

(43) $$\left[-\frac{3}{2}, -\frac{5}{2}\right]$$

```
Type: Vector(Fraction(Integer))
```

Das Skalarprodukt bekommt man mit `dot`

9d ⟨*tutor.input* 2a⟩+≡ ◁9c 9e▷

```
dot(vector [1,2], vector [3,4])
```

(44) 11

```
Type: PositiveInteger
```

Beim invertieren wird der Ring wenn nötig erweitert:

9e ⟨*tutor.input* 2a⟩+≡ ◁9d 9f▷

```
inverse A
```

(45) $$\begin{bmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{bmatrix}$$

```
Type: Union(Matrix(Fraction(Integer)),...)
```

Für Zeilen- und Spaltenoperationen stehen die Funktionen `row(A,k)`, `column(A,k)` und `setRow!(A,k,v)` zur Verfügung. Weiters kann man mit `horizConcat` und `vertConcat` Matrizen zusammenstückeln:

9f ⟨*tutor.input* 2a⟩+≡ ◁9e 10a▷

```
ai:= [transpose matrix [[x,2*x,3*x]] for x in [a,b,c,d,e]]
```

(46)
$$\left[\begin{bmatrix} a \\ 2\,a \\ 3\,a \end{bmatrix}, \begin{bmatrix} b \\ 2\,b \\ 3\,b \end{bmatrix}, \begin{bmatrix} c \\ 2\,c \\ 3\,c \end{bmatrix}, \begin{bmatrix} d \\ 2\,d \\ 3\,d \end{bmatrix}, \begin{bmatrix} e \\ 2\,e \\ 3\,e \end{bmatrix}\right]$$

Type: List(Matrix(Polynomial(Integer)))

10a ⟨*tutor.input* 2a⟩+≡ ◁9f 10b▷
```
reduce(horizConcat,ai)
```

(47)
$$\begin{bmatrix} a & b & c & d & e \\ 2\,a & 2\,b & 2\,c & 2\,d & 2\,e \\ 3\,a & 3\,b & 3\,c & 3\,d & 3\,e \end{bmatrix}$$

Type: Matrix(Polynomial(Integer))

Wenn man Matrizen über speziellen Typen will, muß man das explizit deklarieren:

10b ⟨*tutor.input* 2a⟩+≡ ◁10a 10c▷
```
B:Matrix PrimeField 29 := matrix [[1,1],[-1,1]]
```

(48)
$$\begin{bmatrix} 1 & 1 \\ 28 & 1 \end{bmatrix}$$

Type: Matrix(PrimeField(29))

10c ⟨*tutor.input* 2a⟩+≡ ◁10b 10d▷
```
inverse B
```

(49)
$$\begin{bmatrix} 15 & 14 \\ 15 & 15 \end{bmatrix}$$

Type: Union(Matrix(PrimeField(29)),...)

Lösen von linearen Gleichungssystemen mit `solve`: Die Lösung wird in Form einer speziellen Lösung zusammen mit einer Basis des Kernraums ausgegeben.

10d ⟨*tutor.input* 2a⟩+≡ ◁10c 10e▷
```
v:Vector PrimeField 29 := vector [1,2]
```

(50) $[1,\ 2]$

Type: Vector(PrimeField(29))

10e ⟨*tutor.input* 2a⟩+≡ ◁10d 10f▷
```
solve(B,v)
```

(51) $[particular = [14,\ 16],\ basis = []]$

Type: Record(particular: Union(Vector(PrimeField(29)),"failed"),basis: List(Vector(PrimeField(29))))

Matrizen können auch in Polynome eingesetzt werden:

10f ⟨*tutor.input* 2a⟩+≡ ◁10e 10g▷
```
p:UP(x,Integer):= x^2+1
```

(52) $x^2 + 1$

Type: UnivariatePolynomial(x,Integer)

10g ⟨*tutor.input* 2a⟩+≡ ◁10f 11a▷
```
eval(p,x=B)
```

(53)
$$\begin{bmatrix} 1 & 2 \\ 27 & 1 \end{bmatrix}$$

Type: Polynomial(SquareMatrix(2,PrimeField(29)))

Dimensionen und Basen von Vektorräumen aller Art bestimmt man am besten, indem man Koeffizienten etc. extrahiert und in geeignete Vektoren abspeichert, diese zu einer Matrix zusammenfaßt und dann `rank` bzw. `columnspace` darüberlaufen läßt.
Den Koeffizientenvektor eines Polynoms erzeugt man dabei am besten durch `vectorise`:

11a ⟨*tutor.input* 2a⟩+≡ ◁10g 11b▷
```
vectorise(1+x^2+x^4,6)
```

(54)
$$[1,\ 0,\ 1,\ 0,\ 1,\ 0]$$

Type: Vector(Integer)

Zu beachten ist, daß es ein univariates Polynom sein muß. In vorangegangenen Beispiel hat der Interpreter eine *versteckte Umwandlung* durchgeführt, solche stehen dem Compiler nicht zur Verfügung, siehe 5.4!

3. Formales Rechnen

3.1. Differentialalgebra. Das Domain `Expression R` steht für Rechnungen mit den üblichen Funktionen der Analysis (sin, cos, exp, ...) über dem Ring R zur Verfügung. Meistens genügt der Ring der ganzen Zahlen oder der Gaußsche Ring der komplexen ganzen Zahlen.

11b ⟨*tutor.input* 2a⟩+≡ ◁11a 11c▷
```
e1:=sin(x+y)
```

(55)
$$\sin(y+x)$$

Type: Expression(Integer)

11c ⟨*tutor.input* 2a⟩+≡ ◁11b 11d▷
```
expand e1
```

(56)
$$\cos(x)\sin(y)+\cos(y)\sin(x)$$

Type: Expression(Integer)

11d ⟨*tutor.input* 2a⟩+≡ ◁11c 11e▷
```
e2:=tan(x)^6+3*tan(x)^4+3*tan(x)^2+1
```

(57)
$$\tan(x)^6+3\tan(x)^4+3\tan(x)^2+1$$

Type: Expression(Integer)

11e ⟨*tutor.input* 2a⟩+≡ ◁11d 11f▷
```
simplify e2
```

(58)
$$\frac{1}{\cos(x)^6}$$

Type: Expression(Integer)

11f ⟨*tutor.input* 2a⟩+≡ ◁11e 12a▷
```
D(log x,x)
```

(59) $$\frac{1}{x}$$

Type: Expression(Integer)

12a ⟨*tutor.input* 2a⟩+≡ ◁11f 12b▷
```
integrate(log x,x)
```

(60) $$x \log(x) - x$$

Type: Union(Expression(Integer),...)

Differenzieren kann man nur formale Ausdrücke, keine Funktionen. Wenn man die Ableitung dennoch als Funktion will, dann hilft `function`:

12b ⟨*tutor.input* 2a⟩+≡ ◁12a 12c▷
```
f(x) == x^2
f1expr:=D(f(x),x))
```

```
Compiling function f with type Variable(x) -> Polynomial(Integer)
```

(61) $$2x$$

Type: Polynomial(Integer)

12c ⟨*tutor.input* 2a⟩+≡ ◁12b 12d▷
```
function(f1expr,f1,x)
```

(62) $$f1$$

Type: Symbol

12d ⟨*tutor.input* 2a⟩+≡ ◁12c 12e▷
```
f1(u)
```

```
Compiling function f1 with type Variable(u) -> Polynomial(Integer)
```

(63) $$2\ u$$

Type: Polynomial(Integer)

3.2. Substitutionen. Man kann Symbole in formalen Ausdrücken vom Typ `Expression R` duch andere Ausdrücke ersetzen:

12e ⟨*tutor.input* 2a⟩+≡ ◁12d 12f▷
```
f:= (x+y)/(u+v)
```

(64) $$\frac{y+x}{v+u}$$

Type: Fraction(Polynomial(Integer))

12f ⟨*tutor.input* 2a⟩+≡ ◁12e 13a▷
```
subst(f,x=z)
```

(65) $$\frac{z+y}{v+u}$$

Type: Expression(Integer)

Dabei wird *vor* der Substitution eine stille Umwandlung von `Fraction(Polynomial(Integer))` nach `Expression Integer` durchgeführt.

3.3. Interner Aufbau von `Expression Integer`. Der Typ `Expression Integer` ist *rekursiv* aufgebaut. Der allgemeinste formale Ausdruck ist ein Bruch, dessen Nenner und Zähler vom Typ `SMP(Integer,Kernel Expression Integer)`[1] sind. Das sind Polynome mit ganzzahligen Koeffizienten in "Variablen" vom Typ `Kernel Expression Integer`. Ein `Kernel` ist ein formaler Ausdruck, der in einen Operator (z.B. `sin`, `cos`, `sqrt`, `nthRoot`) und seine Argumente zerlegt werden kann. Die Kerne eines Ausdrucks können mit `kernels` extrahiert werden:

13a ⟨*tutor.input* 2a⟩+≡ ◁12f 13b▷
```
kk:=kernels(sin(x)+cos(x)*exp(x)+sqrt (1+x) + (1+y)^(2/3))
```

(66) $$\left[\sqrt[3]{y+1},\ \sqrt{x+1},\ \sin(x),\ e^{x},\ \cos(x)\right]$$

Type: List(Kernel(Expression(Integer)))

Ein Kern kann mit `name` und `argument`

13b ⟨*tutor.input* 2a⟩+≡ ◁13a 13c▷
```
[name k for k in kk]
```

(67) $$[nthRoot,\ nthRoot,\ \sin,\ exp,\ \cos]$$

Type: List(Symbol)

13c ⟨*tutor.input* 2a⟩+≡ ◁13b 13d▷
```
[argument k for k in kk]
```

(68) $$[[y+1,\ 3],\ [x+1,\ 2],\ [x],\ [x],\ [x]]$$

Type: List(List(Expression(Integer)))

Ein formaler Kern kann mit `operator` erzeugt werden:

13d ⟨*tutor.input* 2a⟩+≡ ◁13c 13e▷
```
f:=operator 'f
```

(69) $$f$$

Type: BasicOperator

13e ⟨*tutor.input* 2a⟩+≡ ◁13d 13f▷
```
kernels f(x)
```

(70) $$[f(x)]$$

Type: List(Kernel(Expression(Integer)))

13f ⟨*tutor.input* 2a⟩+≡ ◁13e 13g▷
```
name first kernels f(x)
```

(71) $$f$$

Type: Symbol

13g ⟨*tutor.input* 2a⟩+≡ ◁13f 14a▷
```
argument first kernels f(x)
```

[1] siehe Abschnitt 2.2

(72) $[x]$

Type: List(Expression(Integer))

Die folgende Funktion erzeugt rekursiv *alle* in einem Ausdruck vorkommenden Kerne:

14a ⟨*tutor.input* 2a⟩+≡ ◁13g 14b▷

```
allkernels:Expression Integer->List Kernel Expression Integer
allkernels ex ==
   ker:=kernels ex
   arg:=concat [argument x for x in ker]
   concat(ker,concat [allkernels a for a in arg])
```

3.4. Potenzreihen. Des weiteren kann man Taylorreihen (und allgemeinere sogenannte Puiseuxreihen) berechnen. Dahinter steckt natürlich der Datentyp `Stream` und es werden immer nur so viele Terme der Reihe berechnet, wie gerade nötig sind.

14b ⟨*tutor.input* 2a⟩+≡ ◁14a 14c▷

```
exps:=series(exp x,x=0)
```

(73)

$$1+x+\frac{1}{2}\,x^2+\frac{1}{6}\,x^3+\frac{1}{24}\,x^4+\frac{1}{120}\,x^5+\frac{1}{720}\,x^6+\frac{1}{5040}\,x^7+\frac{1}{40320}\,x^8+\frac{1}{362880}\,x^9+\frac{1}{3628800}\,x^{10}+O\left(x^{11}\right)$$

Type: UnivariatePuiseuxSeries(Expression(Integer),x,0)

Reihen können auch durch explizite Angabe der Koeffizienten konstruiert werden:

14c ⟨*tutor.input* 2a⟩+≡ ◁14b 14d▷

```
exps:=series( n+-> 1/factorial n,x=0)
```

(74)

$$1+x+\frac{1}{2}\,x^2+\frac{1}{6}\,x^3+\frac{1}{24}\,x^4+\frac{1}{120}\,x^5+\frac{1}{720}\,x^6+\frac{1}{5040}\,x^7+\frac{1}{40320}\,x^8+\frac{1}{362880}\,x^9+\frac{1}{3628800}\,x^{10}+O\left(x^{11}\right)$$

Type: UnivariatePuiseuxSeries(Expression(Integer),x,0)

`series` gibt immer den allgemeinsten Reihentyp (*Puiseuxreihe*, mit rationalen Exponenten) zurück. Man kann versuchen, einen spezielleren Typ zu finden:

14d ⟨*tutor.input* 2a⟩+≡ ◁14c 14e▷

```
retract exps
```

(75)

$$1+x+\frac{1}{2}\,x^2+\frac{1}{6}\,x^3+\frac{1}{24}\,x^4+\frac{1}{120}\,x^5+\frac{1}{720}\,x^6+\frac{1}{5040}\,x^7+\frac{1}{40320}\,x^8+\frac{1}{362880}\,x^9+\frac{1}{3628800}\,x^{10}+O\left(x^{11}\right)$$

Type: UnivariateLaurentSeries(Expression(Integer),x,0)

Laurentreihen erlauben auch negative Exponenten. Im vorliegenden Fall kann man noch weiter spezifizieren

14e ⟨*tutor.input* 2a⟩+≡ ◁14d 14f▷

```
retract retract exps
```

(76)

$$1+x+\frac{1}{2}\,x^2+\frac{1}{6}\,x^3+\frac{1}{24}\,x^4+\frac{1}{120}\,x^5+\frac{1}{720}\,x^6+\frac{1}{5040}\,x^7+\frac{1}{40320}\,x^8+\frac{1}{362880}\,x^9+\frac{1}{3628800}\,x^{10}+O\left(x^{11}\right)$$

Type: UnivariateTaylorSeries(Expression(Integer),x,0)

Wenn man das von vornherein weiß, kann man das auch explizit sagen:

14f ⟨*tutor.input* 2a⟩+≡ ◁14e 15a▷

```
taylor( n+-> 1/factorial n,x=0)
```

(77)

$$1+x+\frac{1}{2}\ x^2+\frac{1}{6}\ x^3+\frac{1}{24}\ x^4+\frac{1}{120}\ x^5+\frac{1}{720}\ x^6+\frac{1}{5040}\ x^7+\frac{1}{40320}\ x^8+\frac{1}{362880}\ x^9+\frac{1}{3628800}\ x^{10}+O\left(x^{11}\right)$$

Type: UnivariateTaylorSeries(Expression(Integer),x,0)

3.5. Lösen von Gleichungen [1, Kap. 1.14, 8.5]**.** Zum Lösen von Gleichungen stehen die Befehle `solve`, `realSolve` [1, Kap. 9.76,9.103] `realRoots`, `complexRoots`, `complexSolve`, `radicalSolve`, `realZeros`, `complexZeros` zur Verfügung.

3.6. Rechnen mit impliziten algebraischen Funktionen. Zum formalen Rechnen mit den Lösungen von Polynomgleichungen gibt es `rootOf`, `rootsOf` [1, Kap. 8.2.3,8.3]. Dabei kann die Lösung von Parametern abhängen und es kann implizit differenziert und integriert werden. Der folgende Befehl definiert eine Funktion y, die implizit von x abhängt.

15a ⟨*tutor.input* 2a⟩+≡ ◁14f 15b▷

```
w:=rootOf(y^2-x,y)
```

(78) y

Type: Expression(Integer)

Zu beachten ist, daß das *Symbol* y davon unbehelligt bleibt:

15b ⟨*tutor.input* 2a⟩+≡ ◁15a 15c▷

```
y
```

(79) y

Type: Variable(y)

Die Abhängigkeit von x ist äußerlich nicht erkennbar, macht sich jedoch beim Rechnen bemerkbar:

15c ⟨*tutor.input* 2a⟩+≡ ◁15b 15d▷

```
w^2
```

(80) x

Type: Expression(Integer)

Die Ausdrücke für Ableitung und Stammfunktion kann man (in diesem Beispiel) leicht händisch verifizieren:

15d ⟨*tutor.input* 2a⟩+≡ ◁15c 15e▷

```
D(w,x)
```

(81) $$\frac{1}{2\ y}$$

Type: Expression(Integer)

15e ⟨*tutor.input* 2a⟩+≡ ◁15d 15f▷

```
integrate(w,x)
```

(82) $$\frac{2\ x\ y}{3}$$

Type: Union(Expression(Integer),...)

Ein etwas komplexeres Beispiel: Die *Bernoullische Lemniskate* hat die Gleichung

15f ⟨*tutor.input* 2a⟩+≡ ◁15e 16a▷

```
p:= (x^2+y^2)^2- 2*a^2*(x^2-y^2)
```

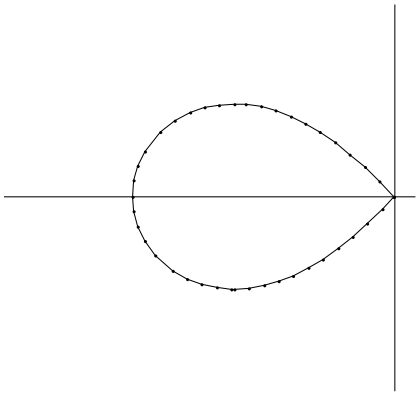
(83) $$y^4 + \left(2\ x^2 + 2\ a^2\right)\ y^2 + x^4 - 2\ a^2\ x^2$$

Type: Polynomial(Integer)

Wegen der Singularität kann nur der halbe Graph gezeichnet werden.

16a ⟨*tutor.input* 2a⟩+≡ ◁15f 16b▷

```
draw(subst(p,a=1)=0,x,y,range == [-2..-1/1000,-1..1])
```



16b ⟨*tutor.input* 2a⟩+≡ ◁16a 16c▷

```
w:=rootOf(p,y)
```

(84) $$y$$

Type: Expression(Integer)

16c ⟨*tutor.input* 2a⟩+≡ ◁16b 16d▷

```
D(w,x)
```

(85) $$\frac{-x\ y^2 - x^3 + a^2\ x}{y^3 + \left(x^2 + a^2\right)\ y}$$

Type: Expression(Integer)

16d ⟨*tutor.input* 2a⟩+≡ ◁16c 17a▷

```
integrate(w,x)
```

(86) $$\frac{-y^3 + \left(x^2 - 2\ a^2\right)\ y}{4\ x}$$

Type: Union(Expression(Integer),...)

Um mit diesen Ausdrücken weiterzuarbeiten, etwa um Extremwerte zu finden, muß man allerdings zunächst die *implizite Funktion* y durch das *Symbol* y ersetzen und die Ausdrücke

in Polynome zurückverwandeln. Als Beispiel bestimmen wir alle Punkte mit horizontaler Tangente, d.h. diejenigen, in denen $y'(x) = 0$ ist.

17a ⟨*tutor.input* 2a⟩+≡ ◁16d 17b▷

```
w1:= subst(D(w,x),w=y)::Fraction Polynomial Integer
```

(87)
$$\frac{-x\ y^2 - x^3 + a^2\ x}{y^3 + (x^2 + a^2)\ y}$$

Type: Fraction(Polynomial(Integer))

Für die Nullstellen genügt es, den Zähler zu betrachten:

17b ⟨*tutor.input* 2a⟩+≡ ◁17a 17c▷

```
w1z:=numer w1
```

(88)
$$-x\ y^2 - x^3 + a^2\ x$$

Type: Polynomial(Integer)

Gesucht sind also alle Punkte, die

1. auf der Kurve liegen
2. in denen die Ableitung $y'(x)$ verschwindet.

Das ergibt zusammen ein System von zwei Gleichungen mit zwei Unbekannten.

17c ⟨*tutor.input* 2a⟩+≡ ◁17b 17d▷

```
radicalSolve([w1z=0,p=0],[x,y])
```

$$\left[\left[x = -\frac{\sqrt{3\ a^2}}{2},\ y = \frac{a}{2}\right],\ \left[x = \frac{\sqrt{3\ a^2}}{2},\ y = \frac{a}{2}\right],\ \left[x = -\frac{\sqrt{3\ a^2}}{2},\ y = -\frac{a}{2}\right],\right.$$
$$\left.\left[x = \frac{\sqrt{3\ a^2}}{2},\ y = -\frac{a}{2}\right],\ [x = 0,\ y = 0],\ \left[x = 0,\ y = \sqrt{-2\ a^2}\right],\ \left[x = 0,\ y = -\sqrt{-2\ a^2}\right]\right]$$

Type: List(List(Equation(Expression(Integer))))

Das funktioniert *nicht*, wenn man versucht, die Gleichungen direkt mit den `rootOf`-Ausdrücken zu lösen:

17d ⟨*tutor.input* 2a⟩+≡ ◁17c 17e▷

```
solve(numer D(w,x)=0,x)
```

(89) []

Type:
List(Equation(Fraction(Polynomial(SparseMultivariatePolynomial(Integer,Kernel(Expression

3.7. Rechnen mit formalen Funktionen. Man kann auch mit unbekannten Funktionen rechnen:

17e ⟨*tutor.input* 2a⟩+≡ ◁17d 17f▷

```
f:= operator 'f
D(f(sin x),x)
```

(90)
$$\cos(x)\ f'(\sin(x))$$

Type: Expression(Integer)

Dabei kommen die römischen Zahlen zum Einsatz:

17f ⟨*tutor.input* 2a⟩+≡ ◁17e 18a▷

```
D(f(x),x,4)
```

(91) $$f^{(iv)}(x)$$

Type: Expression(Integer)

Die Formeln von Faa di Bruno:

18a ⟨*tutor.input* 2a⟩+≡ ◁17f 18b▷

```
f:= operator 'f
g:= operator 'g
h:= operator 'h
faa:=[D(f g x,x,n) for n in 0..]
```

(92) $$\left[f(g(x)),\ f'(g(x))\ g'(x),\ f'(g(x))\ g''(x)+g'(x)^2\ f''(g(x)),\ \ldots\right]$$

Type: Stream(Expression(Integer))

Wir drücken die Ableitungen von $f(x)$ durch die Ableitungen von $h(x) = f(g(x))$ und $g(x)$ aus. Zunächst müssen die Ableitungen durch Symbole ersetzt werden:

18b ⟨*tutor.input* 2a⟩+≡ ◁18a 18c▷

```
faa3:=entries complete [expr for expr in faa for k in 0..3]
ff:= [subscript ('f,[i]) for i in 0..]
```

(93) $$[f_0,\ f_1,\ f_2,\ \ldots]$$

Type: Stream(Symbol)

18c ⟨*tutor.input* 2a⟩+≡ ◁18b 18d▷

```
faa3f:= [subst( expr, [subst(D(f x,x,k),x=g x) = ff (k+1) for k in 0..3]) for expr in
```

```
solve([D(h x,x, i) = hi for  i in 0..3 for hi in faa3f],[ff i for i in 1..4])
```

$$\left[\left[f_0 = h(x),\ f_1 = \frac{h'(x)}{g'(x)},\ f_2 = \frac{g'(x)\ h''(x) - h'(x)\ g''(x)}{g'(x)^3},\right.\right.$$
$$\left.\left. f_3 = \frac{g'(x)^2\ h'''(x) - g'(x)\ h'(x)\ g'''(x) - 3\ g'(x)\ g''(x)\ h''(x) + 3\ h'(x)\ g''(x)^2}{g'(x)^5}\right]\right]$$

Type: List(List(Equation(Expression(Integer))))

3.8. Listen. Listen sind als *Listen von Speicheradressen zu denken.*
Erzeugen einer Liste

18d ⟨*tutor.input* 2a⟩+≡ ◁18c 18e▷

```
aa:=[1,2,3,4]
```

(94) [1, 2, 3, 4]

Type: List(PositiveInteger)

Alternative Eingabe:

18e ⟨*tutor.input* 2a⟩+≡ ◁18d 18f▷

```
bb:= expand [1..4]
```

(95) [1, 2, 3, 4]

Type: List(PositiveInteger)

3.9. Streams. Streams sind (potentiell) "unendliche" Listen:

18f ⟨*tutor.input* 2a⟩+≡ ◁18e 19a▷

```
ss:=expand [1..]
```

(96) $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...]$

Type: Stream(Integer)

Dabei werden immer nur soviele Terme berechnet, wie gerade benötigt: Standardäßig werden

19a ⟨*tutor.input* 2a⟩+≡ ◁18f 19b▷

```
numberOfComputedEntries ss
```

(97) 10

Type: PositiveInteger

Elemente ausgegeben und auf Anfrage berechnet, dieser Wert kann geändert werden mit dem Metabefehl

19b ⟨*tutor.input* 2a⟩+≡ ◁19a 19c▷

```
)set stream calculate 15
ss
```

(98) $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ...]$

Type: Stream(Integer)

Elemente aus Listen oder Streams auswählen

19c ⟨*tutor.input* 2a⟩+≡ ◁19b 19d▷

```
[p for p in ss | prime? p]
```

(99) $[2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ 19,\ 23,\ 29,\ \ldots]$

Type: Stream(Integer)

Aufsummieren etc.

19d ⟨*tutor.input* 2a⟩+≡ ◁19c 19e▷

```
xx:=[subscript(x,[i]) for i in 1..5]
```

(100) $[x_1,\ x_2,\ x_3,\ x_4,\ x_5]$

Type: List(Symbol)

19e ⟨*tutor.input* 2a⟩+≡ ◁19d 19f▷

```
reduce(+,xx)
```

(101) $x_5 + x_4 + x_3 + x_2 + x_1$

Type: Polynomial(Integer)

19f ⟨*tutor.input* 2a⟩+≡ ◁19e 19g▷

```
reduce(*,xx)
```

(102) $x_1\ x_2\ x_3\ x_4\ x_5$

Type: Polynomial(Integer)

Es gibt zwei Möglichkeiten, eine Funktion auf alle Elemente einer Liste anzuwenden: Eine implizite `for`-Schleife und `map`:

19g ⟨*tutor.input* 2a⟩+≡ ◁19f 19h▷

```
[cos x for x in xx]
```

(103) $[\cos(x_1),\ \cos(x_2),\ \cos(x_3),\ \cos(x_4),\ \cos(x_5)]$

Type: List(Expression(Integer))

19h ⟨*tutor.input* 2a⟩+≡ ◁19g 20a▷

```
map(sin,xx)
```

(104) $$[\sin(x_1),\ \sin(x_2),\ \sin(x_3),\ \sin(x_4),\ \sin(x_5)]$$

Type: List(Expression(Integer))

Anstatt `reduce(concat, Liste von Listen)` kann man auch einfach `concat Liste von Listen` schreiben, um z.B. eine geordnete Liste aller rationalen Zahlen im Intervall $[0,1]$, deren Nenner nach Kürzung nicht größer ist als 10.

20a ⟨*tutor.input* 2a⟩+≡ ◁19h 20b▷
```
sort removeDuplicates concat( [[m/n for m in 0..n] for n in 1..10])
```

$$[0,\ \frac{1}{10},\ \frac{1}{9},\ \frac{1}{8},\ \frac{1}{7},\ \frac{1}{6},\ \frac{1}{5},\ \frac{2}{9},\ \frac{1}{4},\ \frac{2}{7},\ \frac{3}{10},\ \frac{1}{3},\ \frac{3}{8},\ \frac{2}{5},\ \frac{3}{7},\ \frac{4}{9},\ \frac{1}{2},\ \frac{5}{9},\ \frac{4}{7},\ \frac{3}{5},\ \frac{5}{8},\ \frac{2}{3},\ \frac{7}{10},\ \frac{5}{7},\ \frac{3}{4},\ \frac{7}{9},\ \frac{4}{5},\ \frac{5}{6},\ \frac{6}{7},\ \frac{7}{8},\ \frac{8}{9},\ \frac{9}{10},\ 1]$$

Type: List(Fraction(Integer))

Die gleiche Liste erhält man, indem man daraus eine Menge macht und sich eine Liste der Elemente dieser Menge erzeugen läßt:

20b ⟨*tutor.input* 2a⟩+≡ ◁20a 20c▷
```
members set concat( [[m/n for m in 0..n] for n in 1..10])
```

3.10. Record [1, Kap. 2.4]**.** Ein Record faßt mehrere Daten verschiedenen Typs zu einem Datensatz (engl. record) zusammen. Dies wird hauptsätzlich benützt, wenn das Ergebnis einer Operation aus mehreren Teilen besteht, z.B. bei Division von ganzen Zahlen.

20c ⟨*tutor.input* 2a⟩+≡ ◁20b 20d▷
```
mn := divide(666,42)
```

(105) $$[quotient = 15,\ remainder = 36]$$

Type: Record(quotient: Integer,remainder: Integer)

Zugriff erfolgt durch `elt` analog zu Listen:

20d ⟨*tutor.input* 2a⟩+≡ ◁20c 20e▷
```
mn.quotient
```

(106) 15

Type: PositiveInteger

4. Funktionen, Macros und Ersetzungsregeln [1, Kap. 6]

4.1. Verzögerte Zuweisung. Mit `==` kann man "verzögerte" Zuweisungen definieren, d.h., es werden die jeweils aktuellen Werte der rechten Seite der Zuweisung eingesetzt. Vgl.

20e ⟨*tutor.input* 2a⟩+≡ ◁20d 20f▷
```
a:=1, b:=a, a:=2, b
```

(107) [1, 1, 2, 1]

Type: Tuple(PositiveInteger)

20f ⟨*tutor.input* 2a⟩+≡ ◁20e 21a▷
```
b==a
a:=1, b, a:=2, b
```

(108) [1, 1, 2, 2]

Type: Tuple(PositiveInteger)

4.2. Funktionen. Funktionen werden definiert wie verzögerte Anweisungen mit Parametern:

21a ⟨*tutor.input* 2a⟩+≡ ◁20f 21b▷
```
f(x) == x^2
```
```
                                                                Type: Void
```

Bei der Auswertung wird eine Signatur errechnet und kompiliert:

21b ⟨*tutor.input* 2a⟩+≡ ◁21a 21c▷
```
f(2)
```
```
Compiling function f with type PositiveInteger -> PositiveInteger
```
```
(109)                                    4
                                                        Type: PositiveInteger
```

21c ⟨*tutor.input* 2a⟩+≡ ◁21b 21d▷
```
f(2.0)
```
```
Compiling function f with type Float -> Float
```
```
(110)                                   4.0
                                                             Type: Float
```

Funktionen können auch rekursiv sein:

21d ⟨*tutor.input* 2a⟩+≡ ◁21c 21e▷
```
fac 1 == 1
fac n == n * fac(n-1)
```
```
                                                              Type: Void
```

21e ⟨*tutor.input* 2a⟩+≡ ◁21d 21f▷
```
fac 4
```
```
Compiling function fac with type Integer -> Integer
Compiling function fac as a recurrence relation.
```
```
(111)                                    24
                                                        Type: PositiveInteger
```

Man kann den Typ von Argument und Rückgabewert auch festlegen:

21f ⟨*tutor.input* 2a⟩+≡ ◁21e 21g▷
```
pow(x:Integer,k:NonNegativeInteger):Integer == x^k
```
```
Function declaration pow : (Integer,NonNegativeInteger) -> Integer
   has been added to workspace.
```

21g ⟨*tutor.input* 2a⟩+≡ ◁21f 21h▷
```
pow(3,2)
```
```
Compiling function pow with type (Integer,NonNegativeInteger) ->
   Integer
```
```
(112)                                    9
                                                        Type: PositiveInteger
```

21h ⟨*tutor.input* 2a⟩+≡ ◁21g 22a▷
```
pow(3,-1)
```

```
Compiling function pow with type (Integer,NonNegativeInteger) ->
   Integer
Conversion failed in the compiled user function pow .

Cannot convert from type Integer to NonNegativeInteger for value
- 1
```

22a ⟨*tutor.input* 2a⟩+≡ ◁21h 22b▷

```
pow1(x:Integer,k:Integer):Integer == x^k
```

```
   Function declaration pow1 : (Integer,Integer) -> Integer has been
      added to workspace.
```

22b ⟨*tutor.input* 2a⟩+≡ ◁22a 22c▷

```
pow1(3,-1)
```

```
Your expression cannot be fully compiled because it contains an
   integer expression (for k ) whose sign cannot be determined (in
   general) and so must be specified by you. Perhaps you can try
   substituting something like
                              (k :: PI)
                                 or
                             (k :: NNI)
   into your expression for k .
FriCAS will attempt to step through and interpret the code.
Compiling function pow1 with type (Integer,Integer) -> Integer
Internal Error
Interpreter code generation failed for expression(^ |#1| |#2|)
```

Man kann auch eine Funktion vorher deklarieren:

22c ⟨*tutor.input* 2a⟩+≡ ◁22b 22d▷

```
pow2:(Integer,Integer) -> Fraction Integer
pow2(x,k) == x^k
pow2(3,-1)
```

```
Compiling function pow2 with type (Integer,Integer) -> Fraction(
   Integer)
```

(113)
$$\frac{1}{3}$$

```
                                                  Type: Fraction(Integer)
```

22d ⟨*tutor.input* 2a⟩+≡ ◁22c 22e▷

```
f:= x +-> x^2
```

(114)
$$x \mapsto x^2$$

```
                                                  Type: AnonymousFunction
```

22e ⟨*tutor.input* 2a⟩+≡ ◁22d 23a▷

```
f(u)
```

(115) u^2

Type: Polynomial(Integer)

Die berühmten Fibonaccizahlen: $F_n = F_{n-1} + F_{n-2}$

23a ⟨*tutor.input* 2a⟩+≡ ◁22e 23b▷

```
nextfib(nn) == [nn.2,nn.1+nn.2]
nextfib [1,1]
```

(116) [1, 2]

Type: List(PositiveInteger)

23b ⟨*tutor.input* 2a⟩+≡ ◁23a 23c▷

```
nextfib %
```

(117) [2, 3]

Type: List(PositiveInteger)

23c ⟨*tutor.input* 2a⟩+≡ ◁23b 23d▷

```
fib2:=[generate(nextfib,[1,1])]
```

(118) [[1, 1], [1, 2], [2, 3], [3, 5], [5, 8], [8, 13], [13, 21], [21, 34], [34, 55], [55, 89], ...]

Type: Stream(List(PositiveInteger))

23d ⟨*tutor.input* 2a⟩+≡ ◁23c 23e▷

```
fib:= [x.1 for x in fib2]
```

(119) [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...]

Type: Stream(PositiveInteger)

23e ⟨*tutor.input* 2a⟩+≡ ◁23d 23f▷

```
fib 42
```

(120) 267914296

Type: PositiveInteger

4.3. Funktionswerte im Cache speichern. Es gibt die Möglichkeit, bereits berechnete Funktionswerte im Speicher auf Abruf liegen zu lassen. Das muß *vor* der Definition der Funktion bekannt gegeben werden, und zwar mit dem Metabefehl

`)set function cache` n `f`

wobei n eine natürliche Zahl (Anzahl der zu speichernden zuletzt berechneten Werte Werte) oder `all` sein kann. In ersterem Fall arbeitet es nach dem Prinzip FIFO.

23f ⟨*tutor.input* 2a⟩+≡ ◁23e 23g▷

```
)set function cache all slowadd
slowadd(x,y) == (for k in 1..y repeat x:=x+1; x)
slowadd(1,100000000)
```

(121) 100000001

Type: PositiveInteger
Time: 1.47 (EV) = 1.47 sec

und beim zweiten Mal

23g ⟨*tutor.input* 2a⟩+≡ ◁23f 24a▷

```
slowadd(1,100000000)
```

(122) 100000001

```
                                                              Type: PositiveInteger
                                                Time: 0.004 (IN) + 0.03 (OT) = 0.03 sec
```

4.4. Macros[1, Kap. 6.2]**.** Ein *Macro* ist eine Abkürzung, die übersetzt wird, bevor der eigentliche Interpreter loslegt, z.B.

24a ⟨*tutor.input* 2a⟩+≡ ◁23g 24b▷

```
macro b1 == subscript(b,[1])
```

4.5. "Pattern Matching". Um komplexere Ausdrücke zu ersetzen, reicht `subst` nicht aus und es muß eine mächtigeres Instrument eingeschaltet werden. Die folgende Ersetzungsregel faßt Logarithmen zusammen

24b ⟨*tutor.input* 2a⟩+≡ ◁24a 24c▷

```
logrule1:= rule log a + log b == log(a*b)
```

(123) $$\log(b) + \log(a) + \%B == \log(a\ b) + \%B$$

```
                                  Type: RewriteRule(Integer,Integer,Expression(Integer))
```

24c ⟨*tutor.input* 2a⟩+≡ ◁24b 24d▷

```
logrule1(log 2 + log 3)
```

(124) $$\log(6)$$

```
                                                              Type: Expression(Integer)
```

Eine Ersetzungsregel kann auch mehrere Teilregeln enthalten:

24d ⟨*tutor.input* 2a⟩+≡ ◁24c 24e▷

```
logrule2:= rule
    log a + log b == log(a*b)
    a*log b == log(b^a)
```

(125) $$\{\log(b) + \log(a) + \%C == \log(a\ b) + \%C,\ a\log(b) == \log(b^a)\}$$

```
                        Type: Ruleset(Integer,Integer,Expression(Integer))
```

24e ⟨*tutor.input* 2a⟩+≡ ◁24d 24f▷

```
logrule2(a*log x + log y)
```

(126) $$\log(y\ x^a)$$

```
                                                              Type: Expression(Integer)
```

Man kann auch zuerst Bedingungen abfragen:

24f ⟨*tutor.input* 2a⟩+≡ ◁24e 24g▷

```
logrule3:=rule
    log a + log b == log(a*b)
    (a|integer? a)*log b == log(b^a)
```

(127) $$\{\log(b) + \log(a) + \%D == \log(a\ b) + \%D,\ a\log(b) == \log(b^a)\}$$

```
                        Type: Ruleset(Integer,Integer,Expression(Integer))
```

24g ⟨*tutor.input* 2a⟩+≡ ◁24f 24h▷

```
logrule3(a*log x + log y)
```

(128) $$\log(y) + a\log(x)$$

```
                                                              Type: Expression(Integer)
```

24h ⟨*tutor.input* 2a⟩+≡ ◁24g 25a▷

```
logrule3(2*log x + log y)
```

(129) $$\log\left(x^2\ y\right)$$

Type: Expression(Integer)

Hier ein Fall wo ohne vorherige Abfrage etwas Sinnloses herauskommt:

25a ⟨*tutor.input* 2a⟩+≡ ◁24h 25b▷
```
sinrule1 := rule sin((n)*x) == cos ((n-1)*x)*sin x + cos x *sin((n-1)*x)
```

(130) $$\sin\left(n\ x\right) == \cos\left(x\right)\ \sin\left(\left(n-1\right)\ x\right) + \cos\left(\left(n-1\right)\ x\right)\ \sin\left(x\right)$$

Type: RewriteRule(Integer,Integer,Expression(Integer))

25b ⟨*tutor.input* 2a⟩+≡ ◁25a 25c▷
```
sinrule1 sin(5*u)
```

(131) $$\cos\left(5\right)\ \sin\left(5\ u-5\right) + \sin\left(5\right)\ \cos\left(5\ u-5\right)$$

Type: Expression(Integer)

Besser

25c ⟨*tutor.input* 2a⟩+≡ ◁25b 25d▷
```
sinrule2 := rule sin((n|integer? n)*x) == cos ((n-1)*x)*sin x + cos x * sin((n-1)*x)
```

(132) $$\sin\left(n\ x\right) == \cos\left(x\right)\ \sin\left(\left(n-1\right)\ x\right) + \cos\left(\left(n-1\right)\ x\right)\ \sin\left(x\right)$$

Type: RewriteRule(Integer,Integer,Expression(Integer))

Im Unterschied zu Funktionen werden Ersetzungsregeln so oft angewandt bis es nicht mehr geht, sodass am Ende kein $\sin kx$ mit $k > 1$ mehr vorkommt.

25d ⟨*tutor.input* 2a⟩+≡ ◁25c 25e▷
```
sinrule2 sin(5*x)
```

(133) $$\left(\cos\left(4\ x\right) + \cos\left(x\right)\ \cos\left(3\ x\right) + \cos\left(x\right)^2\ \cos\left(2\ x\right) + 2\ \cos\left(x\right)^4\right)\ \sin\left(x\right)$$

Type: Expression(Integer)

Damit kann man auch formale lineare Abbildungen bilden:

25e ⟨*tutor.input* 2a⟩+≡ ◁25d 25f▷
```
u:=operator 'u;
myRule:=rule u(x+y) == u(x) + u(y)
```

(134) $$u\left(y+x\right) == {}'v\left(y\right) + {}'u\left(x\right)$$

Type: RewriteRule(Integer,Integer,Expression(Integer))

25f ⟨*tutor.input* 2a⟩+≡ ◁25e 25g▷
```
myRule u(a1+a2+a3+a4)
```

(135) $$u\left(a4\right) + u\left(a3\right) + u\left(a2\right) + u\left(a1\right)$$

Type: Expression(Integer)

5. Programmierkonstrukte, Blöcke und 'Piles'

5.1. Abfragen von Bedingungen [1, Kap. 0.7, 5.3]. `if...then...else`:

25g ⟨*tutor.input* 2a⟩+≡ ◁25f 26a▷
```
k:=1
if k > 0 then 1 else 0
```

5.2. Schleifen [1, Kap. 5.4]**.** Es gibt die üblichen Schleifenkonstrukte mit `for` und `while`:

26a ⟨*tutor.input* 2a⟩+≡ ◁25g 26b▷
```
s:=0
for k in 1..5 repeat s:= s+k^2
s
```

(136) 55

Type: NonNegativeInteger

Dasselbe mit einer `while`-Schleife:

26b ⟨*tutor.input* 2a⟩+≡ ◁26a 26c▷
```
s:=0,k:=0;
while k < 5 repeat (k:=k+1;s:= s+k); s
```

Dasselbe mit konditionalem Abbruch:

26c ⟨*tutor.input* 2a⟩+≡ ◁26b 26d▷
```
s:=0,k:=0;
repeat (if k>5 then break else (s:=s+k;k:=k+1));s
```

Man kann Schleifen über beliebige Listen laufen lassen. Die folgenden Befehle drehen eine Liste um:

26d ⟨*tutor.input* 2a⟩+≡ ◁26c 26e▷
```
l:= [subscript(x,[i]) for i in 1..5]
```

(137) $[x_1,\ x_2,\ x_3,\ x_4,\ x_5]$

Type: List(Symbol)

26e ⟨*tutor.input* 2a⟩+≡ ◁26d 26f▷
```
res:=[];
for x in l repeat res:=cons(x,res)
res
```

(138) $[x_5,\ x_4,\ x_3,\ x_2,\ x_1]$

Type: List(Polynomial(Integer))

Das ist *wesentlich effizienter* als über die Indizes zu iterieren:

26f ⟨*tutor.input* 2a⟩+≡ ◁26e 27b▷
```
for x in 1..5 repeat res:=cons(l.i,res)
```

weil bei der zweiten Methode die Liste l jedesmal neu durchlaufen wird.

5.3. Blöcke [1, Kap. 5.2]**.** Längere Befehlsfolgen werden besser in Dateien ausgelagert und dort übersichtlich in sogenannten *Blöcken* "gestapelt" (engl. *pile*. Solche *piles* können nicht interaktiv eingegeben werden, sondern müssen mit `)read` (siehe8.3) eingelesen werden. Hier ein kurzes Programm, das jeweils die Summe aller positiven und negativen Einträge einer Liste summiert.

27a ⟨*bspprog.input* 27a⟩≡
```
POSNEG ==> Record(pos:Integer,neg:Integer)
f:List Integer -> POSNEG
f(kk) ==
    -- initialisiere Resultat
    respos:Integer := 0
    resneg:Integer := 0
    for k in kk repeat
        if k>0 then
            respos:=respos+k
          else
            resneg:=resneg -k
    [respos,resneg]
```

27b ⟨*tutor.input* 2a⟩+≡ ◁26f 27c▷
```
)read bspprog
f [-1,1,2,-1]
```

(139) $[pos = 3, neg = 2]$

```
                                Type: Record(pos: Integer,neg: Integer)
```

- Lange Zeilen kann man mit _ umbrechen.
- Kommentare werden mit `--` eingeleitet.
- Einrückungen müssen exakt beachtet werden, sonst kommen die wildesten Fehlermeldungen.

27c ⟨*tutor.input* 2a⟩+≡ ◁27b 27d▷
```
sqrt( x +
        a:=2
        b:=1
        a+b
    )
```

(140) $\sqrt{x+3}$

```
                                Type: Expression(Integer)
```

27d ⟨*tutor.input* 2a⟩+≡ ◁27c 28a▷
```
sqrt( x +
        a:=2
        b:=1
        a+b)

>> System error:
Argument X is not a NUMBER: (8)
```

5.4. Der Compiler und was bei Funktionen zu beachten ist. Funktionen werden beim ersten Aufruf compiliert. Dabei wird die Funktion analysiert und "erraten", welchen Typ die einzelnen Variablen des Programms haben. Wenn kein eindeutiger Typ eruiert werden kann, dann gibt der Compiler auf und die Funktion wird nur interpretiert:

28a ⟨*tutor.input* 2a⟩+≡ ◁27d 28b▷
```
dividiere(x,y) ==
   x1:=x
   x1:=x1/y1
```

28b ⟨*tutor.input* 2a⟩+≡ ◁28a 28c▷
```
dividiere(1,2)
```

```
The type of the local variable x1 has changed in the computation.
We will attempt to interpret the code.
```

(141) $$\frac{1}{y1}$$

```
                                   Type: Fraction(Polynomial(Integer))
```

Der Compiler ist nicht so intelligent wie der Interpreter, z.B.:

28c ⟨*tutor.input* 2a⟩+≡ ◁28b 28d▷
```
p:=1+x+x^2
```

(142) $$x^2+x+1$$

```
                                                Type: Polynomial(Integer)
```

28d ⟨*tutor.input* 2a⟩+≡ ◁28c 28e▷
```
coefficient(p,2)
```

(143) $$1$$

```
                                                    Type: PositiveInteger
```

28e ⟨*tutor.input* 2a⟩+≡ ◁28d 29a▷
```
mycoeff(p,k) == coefficient(p,k)
mycoeff(p,2)
```

```
  There are 8 exposed and 4 unexposed library operations named
     coefficient having 2 argument(s) but none was determined to be
     applicable. Use HyperDoc Browse, or issue
                          )display op coefficient
     to learn more about the available operations. Perhaps
     package-calling the operation or using coercions on the arguments
     will allow you to apply the operation.
  Cannot find a definition or applicable library operation named
     coefficient with argument type(s)
                            Polynomial(Integer)
                              PositiveInteger

     Perhaps you should use "@" to indicate the required return type,
     or "$" to specify which version of the function you need.
```

Der Grund ist, daß p den Type `Polynomial Integer` hat, d.h., als Polynom *in mehreren Variablen* angesehen wird.

```
FriCAS will attempt to step through and interpret the code.
```

(144) 1

```
                                                    Type: PositiveInteger
```

Man muss also in diesem Fall mit Hinweisen nachhelfen:

29a ⟨*tutor.input* 2a⟩+≡ ◁28e 29b▷

```
mycoeff(p,k) == coefficient(p::UP(x,?),k)
```

Damit ist es jetzt kompilierbar:

29b ⟨*tutor.input* 2a⟩+≡ ◁29a 29c▷

```
mycoeff(p,2)
```

```
Compiling function mycoeff with type (Polynomial(Integer),
   PositiveInteger) -> Polynomial(Integer)
```

(145) 1

```
                                                    Type: Polynomial(Integer)
```

6. Grafik [1, Kap. 7]

Es gibt ein Anzeigesystem für Grafik, das im Moment so wie das Hilfesystem auf einen X-Window-Server angewiesen ist. Es unterstützt 2- und 3-dimensionale Objekte.

29c ⟨*tutor.input* 2a⟩+≡ ◁29b 29d▷

```
gsin:=draw(sin(x),x=-%pi..%pi)
```

Es öffnet sich ein Fenster wie in Abb. 1 links, ein Mausclick auf das Bild öffnet das Grafikbefehlsfenster (Abb. 1 rechts). Ein Click auf `PS` exportiert die Grafik als Postscript, das

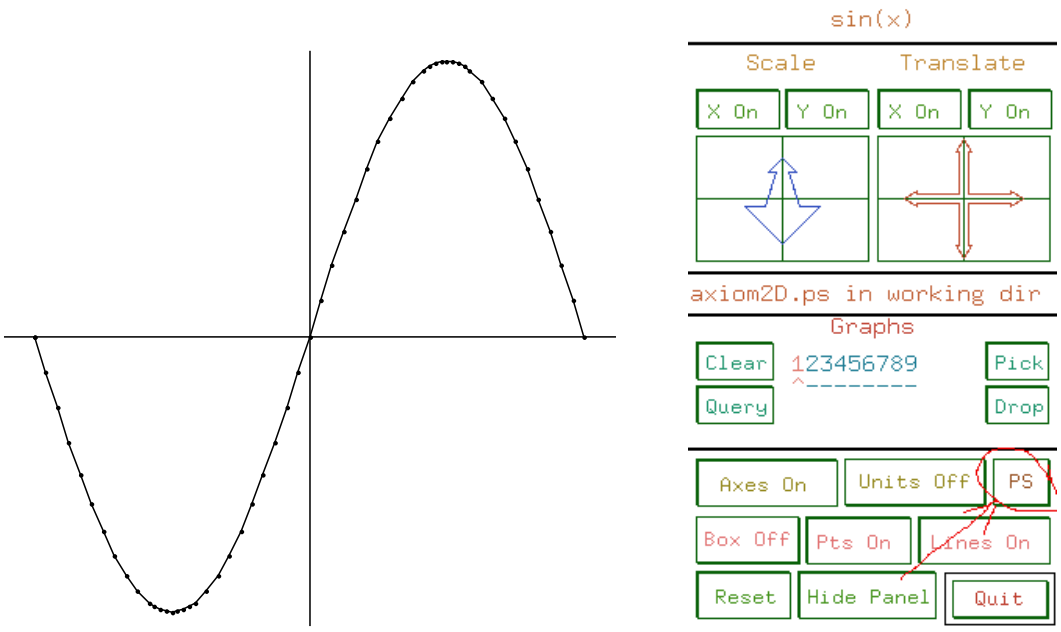


Abbildung 1. Grafikausgabe

dann zu eps, pdf etc. weiterverarbeitet werden kann.
Man kann auch Postscript explizit mit dem Befehl `write` exportieren:

29d ⟨*tutor.input* 2a⟩+≡ ◁29c 30a▷

```
write(gsin,"gsin","postscript");
```

worauf ein Verzeichnis `gsin.VIEW` eröffnet wird, in dem das Postscript-Bild `axiom2D.ps` abgelegt wird.

6.1. Kombinieren von mehreren Graphen. In einem sogenannten *Viewport* haben bis zu 9 Graphen Platz. Dazu ist wie folgt vorzugehen:

30a ⟨*tutor.input* 2a⟩+≡ ◁29d 30b▷
```
v1:=draw(sin x,x=0..%pi)
```

(146) TwoDimensionalViewport: "`sin(x)`"

Type: TwoDimensionalViewport

30b ⟨*tutor.input* 2a⟩+≡ ◁30a 30c▷
```
v2:=draw(cos x,x=0..%pi)
```

(147) TwoDimensionalViewport: "`cos(x)`"

Type: TwoDimensionalViewport

extrahiere den Graphen aus dem zweiten Bild. Es ist der erste (von 9 möglichen):

30c ⟨*tutor.input* 2a⟩+≡ ◁30b 30d▷
```
g2:=getGraph(v2,1)
```

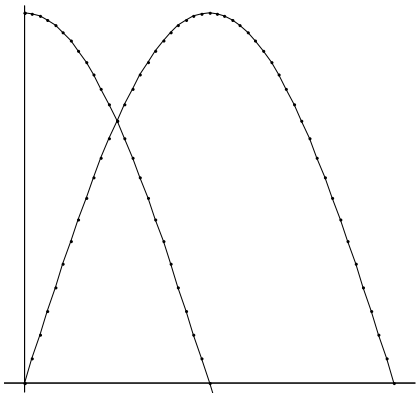
(148) Graph with 1point list

Type: GraphImage

Einfügen dieses Graphen ins erste Bild an zweiter Stelle:

30d ⟨*tutor.input* 2a⟩+≡ ◁30c 31a▷
```
putGraph(v1,g2,2)
```

Type: Void



7. Ein- und Ausgabe [1, Kap. 9.28–29]

Es gibt verschiedene Möglichkeiten, Daten abzuspeichern.

7.1. Sitzung speichern. Mit dem Metabefehl `)history )save foo` kann die aktuelle Sitzung (inklusive Zustand des Arbeitsspeichers) abspeichern. Zu einem späteren Zeitpunkt kann der Zustand mit `)history )restore foo` wiederhergestellt werden. Es wird eine Datei `foo.input` mit sämtlichen eingegebenen Befehlen und ein Verzeichnis `foo.axh` mit den internen Daten angelegt.

7.2. Sitzung mitschreiben. Mit `)spool foo.out` wird alles, was über den Bildschirm geht, in der Datei `foo.out` mitgeschrieben. Es werden auch Kontrollzeichen mitgeschrieben, die man aber mit `grep -v ...` wieder loswerden kann.

7.3. Objekte speichern. Um ein einzelnes Zwischenergebnis abzuspeichern, gibt es zwei Möglichkeiten.

1. Sei `a` ein Objekt vom Typ `T`, dann wird mit den Befehlen

31a ⟨*tutor.input* 2a⟩+≡ ◁30d 31b▷
```
ofile:File T := open("name.lsp"::FileName,"output")
write!(ofile,a)
close! ofile
```
die gesamte Struktur des Objekts abgespeichert. Einlesen desselben in einer späteren Sitzung:

31b ⟨*tutor.input* 2a⟩+≡ ◁31a 31c▷
```
ifile:File T := open("name.lsp"::FileName,"input")
a:=read! ifile
close! ifile
```
wobei natürlich der Typ genau übereinstimmen muß.

2. Die zweite Möglichkeit ist `InputForm`. Diese gibt es nicht für alle Typen, sondern nur solche mit dem Attribut `CoercibleTo InputForm`. Es handelt sich um eine LISP-Darstellung der Objekte, die als solche zunächst unverständlich aussieht:

31c ⟨*tutor.input* 2a⟩+≡ ◁31b 31d▷
```
a:List Polynomial Integer := [1+x,x^2]
```

(149) $$\left[x+1,\ x^2\right]$$

Type: List(Polynomial(Integer))

31d ⟨*tutor.input* 2a⟩+≡ ◁31c 31e▷
```
a::InputForm
```
```
(construct (+ x 1) (^ x 2))
```
Sie kann aber mit `unparse` in eine lesbare Form umgewandelt werden:

31e ⟨*tutor.input* 2a⟩+≡ ◁31d
```
unparse(a::InputForm)
```
```
"[x+1,x^2]"
```

Type: String

7.4. Ein- und Ausgabe in Datei [1, Kap. 9.91]**.** Man kann auch beliebige Daten in Dateien vom Type `TextFile` schreiben und wieder auslesen. Die Daten müssen aber zuerst in Objekte vom Typ `String` umgewandelt werden. Es funktioniert ähnlich wie `File` aus 7.3.

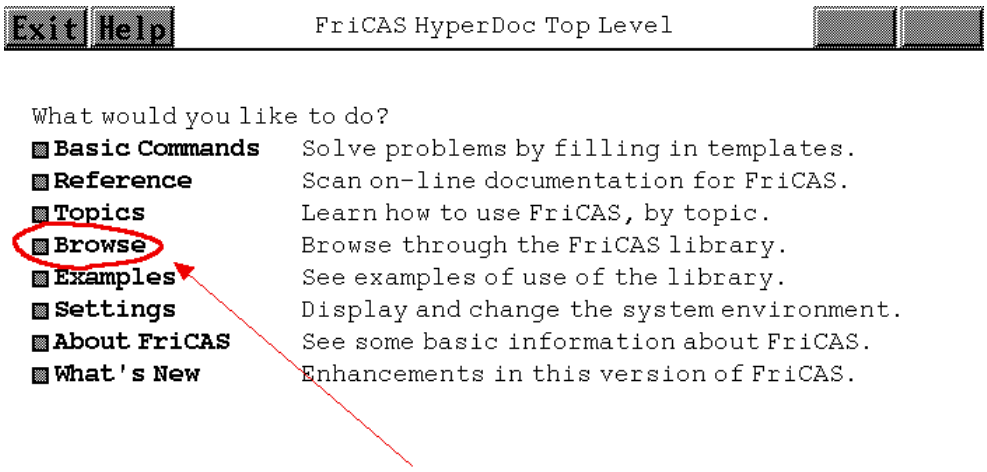
7.5. Ergebnisse drucken. Mit `)set output tex on` können sämtliche Objekte in TEX-Form umgewandelt und mit der Maus weiterkopiert werden (auch in Emacs - es wird nicht das sichtbare Bild kopiert, sondern der "dahinterliegende" TEX-Code).

8. Technische Informationen

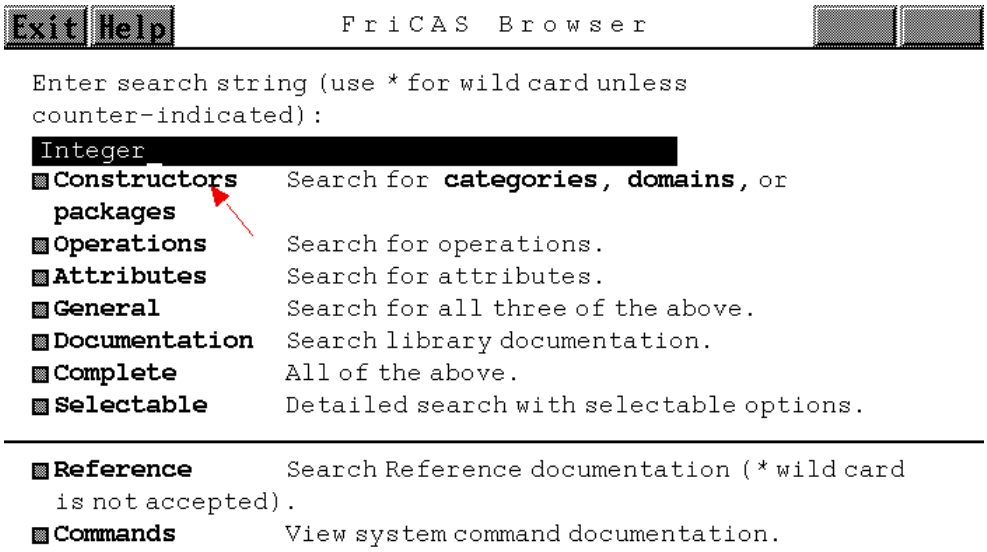
8.1. Emacsmodus. Um den Emacsmodus vernünftig nutzen zu können, ist es zunächst unumgänglich, sich mit Emacs vertraut zu machen und ein entsprechendes Tutorium zu lesen.
Zusätzlich zu den dort gelernten Emacsbefehlen sind speziell im FriCAS-Modus folgende Befehle hilfreich:

`M-p`	fricas-previous-input	Navigation in vorhergegangenen Befehlen
`M-n`	fricas-next-input	Navigation in vorhergegangenen Befehlen
`M-`↑	fricas-up-input	Navigation im Arbeitsblatt
`M-`↓	fricas-down-input	Navigation im Arbeitsblatt
`C-Enter`	fricas-yank	Einlesen und Auswerten einer abgespeicherten Region (unter Berücksichtigung der Einrückungen!)
`TAB`	fricas-dynamic-complete	automatische Ergänzung eines angefangenen Funktionsnamens

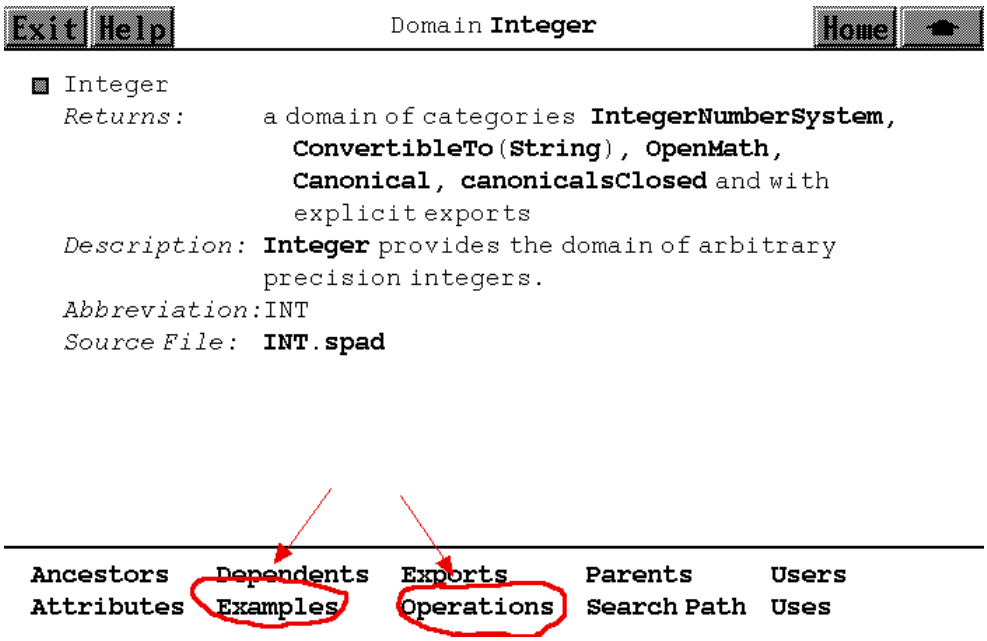
8.2. Hilfesystem [1, Kap. 3]**.** Es gibt ein Hilfesystem im Hypertextformat. Dieses entstand bereits vor dem WWW und HTML und sieht entsprechend antiquiert aus (reine X-Window Schnittstelle), ist aber unentbehrlich bei der Suche nach Befehlen und anderen Typen. Die wichtigste Funktion ist die Suche, die sich hinter *Browse* versteckt:



Nach Eingabe eines Begriffs kann man nach *Domains*, *Operationen* und im Volltext danach suchen lassen:



Neben einer Kurzbeschreibung eines Domains kann man sich eine Liste der verfügbaren Operationen und, soweit vorhanden, Beispiele anzeigen lassen:



Durch Klicken auf `INT.spad` kann man sich auch den gesamten Quellcode für das Domain anzeigen lassen.

8.3. Metabefehle [1, Kap. 1.15, App. A]**.** Die Art der Ausgabe und andere Systemeinstellungen werden durch **Metabefehle** vorgenommen, z.B.

- `)quit`: Beenden
- `)set output tex on`: Einschalten des TEX-Ausgabemodus
- `)set messages time on`: Anzeige der Rechenzeiten
- `)what op foo`: Zeige alle Operationen, die "foo" als Teilstring enthalten
- `)display op foo`: Zeige alle *Signaturen* der Operation "foo" an.
- `)show Integer`: Zeige alle Operationen zum Domain "Integer" an.
- `)read foo`: Lese die Datei `foo.input` ein; einzulesende Dateien *müssen* die Extension `.input` haben.

`)history )save foo:` Die aktuelle Sitzung (inklusive Zustand des Arbeitsspeichers) abspeichern.
`)history )restore foo:` Eine vorher abgespeicherte Sitzung wieder laden.
`)undo :` Den vorhergehenden Befehl rückgängig machen.
`)clear all :` Alle Variablen zurücksetzen.

8.4. Compiler [1, Kap. 11–13]**.** Die Bibliothek, die alle Domains etc. enthält, liegt in übersetzter Form vor. Man kann die Bibliothek mit beliebigen Domains erweitern, die man in der SPAD[2]-Sprache programmieren muß. Der Umgang mit diesem Compiler ist auf Grund von Bugs und irreführenden Fehlermeldungen zur Zeit noch sehr schwierig und wird hier nicht behandelt.

Endbemerkung. Das vorliegende Dokument wurde mit `noweb` [2] geschrieben.

Literatur

[1] Jenks and Sutor. *Axiom*. Oxford, 1992. `http://axiom-developer.org/axiom-website/bookvol0.pdf`.
[2] Norman Ramsey. `noweb`. `http://www.cs.tufts.edu/~nr/noweb/`.

[2]SPAD=scratchpad