$SPAD/src/algebra combfunc.spad

Manuel Bronstein, Martin Rubey

August 6, 2014

Abstract

Contents

1 category COMBOPC CombinatorialOpsCategory

⟨*category COMBOPC CombinatorialOpsCategory*⟩≡

```
)abbrev category COMBOPC CombinatorialOpsCategory
++ Category for summations and products
++ Author: Manuel Bronstein
++ Date Created: ???
++ Date Last Updated: 22 February 1993 (JHD/BMT)
++ Description:
++   CombinatorialOpsCategory is the category obtaining by adjoining
++   summations and products to the usual combinatorial operations;
CombinatorialOpsCategory() : Category ==
  CombinatorialFunctionCategory with
    factorials : % -> %
      ++ factorials(f) rewrites the permutations and binomials in f
      ++ in terms of factorials;
    factorials : (%, Symbol) -> %
      ++ factorials(f, x) rewrites the permutations and binomials in f
      ++ involving x in terms of factorials;
    summation  : (%, Symbol)            -> %
      ++ summation(f(n), n) returns the formal sum S(n) which verifies
      ++ S(n+1) - S(n) = f(n);
    summation  : (%, SegmentBinding %)  -> %
      ++ summation(f(n), n = a..b) returns f(a) + ... + f(b) as a
      ++ formal sum;
    product    : (%, Symbol)            -> %
      ++ product(f(n), n) returns the formal product P(n) which verifies
      ++ P(n+1)/P(n) = f(n);
    product    : (%, SegmentBinding  %) -> %
      ++ product(f(n), n = a..b) returns f(a) * ... * f(b) as a
      ++ formal product;
```

⟨*package COMBF CombinatorialFunction*⟩≡

```
)abbrev package COMBF CombinatorialFunction
++ Provides the usual combinatorial functions
++ Author: Manuel Bronstein, Martin Rubey
++ Date Created: 2 Aug 1988
++ Date Last Updated: 30 October 2005
++ Description:
++   Provides combinatorial functions over an integral domain.
++ Keywords: combinatorial, function, factorial.
++ Examples:  )r COMBF INPUT



CombinatorialFunction(R, F) : Exports == Implementation where
  R : Join(Comparable, IntegralDomain)
  F : FunctionSpace R

  OP  ==> BasicOperator
  K   ==> Kernel F
  SE  ==> Symbol
  O   ==> OutputForm
  SMP ==> SparseMultivariatePolynomial(R, K)
  Z   ==> Integer

  POWER        ==> '%power
  OPEXP        ==> 'exp
  SPECIALDIFF  ==> '%specialDiff
  SPECIALDISP  ==> '%specialDisp
  SPECIALEQUAL ==> '%specialEqual

  Exports ==> with
    belong?    : OP -> Boolean
      ++ belong?(op) is true if op is a combinatorial operator;
    operator   : OP -> OP
      ++ operator(op) returns a copy of op with the domain-dependent
      ++ properties appropriate for F;
      ++ error if op is not a combinatorial operator;
    "^"        : (F, F) -> F
      ++ a ^ b is the formal exponential a^b;
    binomial   : (F, F) -> F
      ++ binomial(n, r) returns the number of subsets of r objects
      ++ taken among n objects, i.e. n!/(r! * (n-r)!);
    permutation : (F, F) -> F
      ++ permutation(n, r) returns the number of permutations of
      ++ n objects taken r at a time, i.e. n!/(n-r)!;
    factorial  : F -> F
```

```
    ++ factorial(n) returns the factorial of n, i.e. n!;
  factorials : F -> F
    ++ factorials(f) rewrites the permutations and binomials in f
    ++ in terms of factorials;
  factorials : (F, SE) -> F
    ++ factorials(f, x) rewrites the permutations and binomials in f
    ++ involving x in terms of factorials;
  summation  : (F, SE) -> F
    ++ summation(f(n), n) returns the formal sum S(n) which verifies
    ++ S(n+1) - S(n) = f(n);
  summation  : (F, SegmentBinding F)  -> F
    ++ summation(f(n), n = a..b) returns f(a) + ... + f(b) as a
    ++ formal sum;
  product    : (F, SE) -> F
    ++ product(f(n), n) returns the formal product P(n) which verifies
    ++ P(n+1)/P(n) = f(n);
  product    : (F, SegmentBinding  F) -> F
    ++ product(f(n), n = a..b) returns f(a) * ... * f(b) as a
    ++ formal product;
  iifact     : F -> F
    ++ iifact(x) should be local but conditional;
  iibinom    : List F -> F
    ++ iibinom(l) should be local but conditional;
  iiperm     : List F -> F
    ++ iiperm(l) should be local but conditional;
  iipow      : List F -> F
    ++ iipow(l) should be local but conditional;
  iidsum     : List F -> F
    ++ iidsum(l) should be local but conditional;
  iidprod    : List F -> F
    ++ iidprod(l) should be local but conditional;
  ipow       : List F -> F
    ++ ipow(l) should be local but conditional;

Implementation ==> add

  COMB := 'comb

  ifact     : F -> F
  iiipow    : List F -> F
  iperm     : List F -> F
  ibinom    : List F -> F
  isum      : List F -> F
  idsum     : List F -> F
  iprod     : List F -> F
  idprod    : List F -> F
```

```
dsum      : List F -> O
ddsum     : List F -> O
dprod     : List F -> O
ddprod    : List F -> O
equalsumprod  : (K, K) -> Boolean
equaldsumprod : (K, K) -> Boolean
fourth    : List F -> F
dvpow1    : List F -> F
dvpow2    : List F -> F
summand   : List F -> F
dvsum     : (List F, SE) -> F
dvdsum    : (List F, SE) -> F
dvprod    : (List F, SE) -> F
dvdprod   : (List F, SE) -> F
facts     : (F, List SE) -> F
K2fact    : (K, List SE) -> F
smpfact   : (SMP, List SE) -> F

-- Must be a macro to produce fresh symbol for each use
dummy ==> new()$SE :: F

opfact  := operator('factorial)$CommonOperators
opperm  := operator('permutation)$CommonOperators
opbinom := operator('binomial)$CommonOperators
opsum   := operator('summation)$CommonOperators
opdsum  := operator('%defsum)$CommonOperators
opprod  := operator('product)$CommonOperators
opdprod := operator('%defprod)$CommonOperators
oppow   := operator(POWER::Symbol)$CommonOperators

factorial x          == opfact x
binomial(x, y)       == opbinom [x, y]
permutation(x, y)    == opperm [x, y]

import from F
import from Kernel F

number?(x : F) : Boolean ==
  if R has RetractableTo(Z) then
    ground?(x) or
     ((retractIfCan(x)@Union(Fraction(Z),"failed")) case Fraction(Z))
  else
    ground?(x)

x ^ y               ==
  -- Do some basic simplifications
```

```
  is?(x, POWER) =>
    args : List F := argument first kernels x
    not(#args = 2) => error "Too many arguments to ^"
    number?(first args) and number?(y) =>
      oppow [first(args)^y, second args]
    oppow [first args, (second args)* y]
  -- Generic case
  exp : Union(Record(val:F,exponent:Z),"failed") := isPower x
  exp case Record(val : F, exponent : Z) =>
    expr := exp::Record(val : F, exponent : Z)
    oppow [expr.val, (expr.exponent)*y]
  oppow [x, y]

belong? op          == has?(op, COMB)
fourth l            == third rest l
dvpow1 l            == second(l) * first(l) ^ (second l - 1)
factorials x        == facts(x, variables x)
factorials(x, v)    == facts(x, [v])
facts(x, l)         == smpfact(numer x, l) / smpfact(denom x, l)
summand l           == eval(first l, retract(second l)@K, third l)

product(x : F, i : SE) ==
  dm := dummy
  opprod [eval(x, k := kernel(i)$K, dm), dm, k::F]

summation(x : F, i : SE) ==
  dm := dummy
  opsum [eval(x, k := kernel(i)$K, dm), dm, k::F]

dvsum(l, x) ==
  opsum [differentiate(first l, x), second l, third l]

dvdsum(l, x) ==
  x = retract(y := third l)@SE => 0
  if member?(x, variables(h := third rest rest l)) or
     member?(x, variables(g := third rest l)) then
    error "a sum cannot be differentiated with respect to a bound"
  else
    opdsum [differentiate(first l, x), second l, y, g, h]

dvprod(l, x) ==
  dm := retract(dummy)@SE
  f := eval(first l, retract(second l)@K, dm::F)
  p := product(f, dm)

  opsum [differentiate(first l, x)/first l * p, second l, third l]
```

```
dvdprod(l, x) ==
  x = retract(y := third l)@SE => 0
  if member?(x, variables(h := third rest rest l)) or
     member?(x, variables(g := third rest l)) then
    error "a product cannot be differentiated with respect to a bound"
  else
    opdsum cons(differentiate(first l, x)/first l, rest l) * opdprod l

dprod l ==
  prod(summand(l)::O, third(l)::O)

ddprod l ==
  prod(summand(l)::O, third(l)::O = fourth(l)::O, fourth(rest l)::O)

dsum l ==
  sum(summand(l)::O, third(l)::O)

ddsum l ==
  sum(summand(l)::O, third(l)::O = fourth(l)::O, fourth(rest l)::O)

equalsumprod(s1, s2) ==
  l1 := argument s1
  l2 := argument s2

  (eval(first l1, retract(second l1)@K, second l2) = first l2)

equaldsumprod(s1, s2) ==
  l1 := argument s1
  l2 := argument s2

  ((third rest l1 = third rest l2) and
   (third rest rest l1 = third rest rest l2) and
   (eval(first l1, retract(second l1)@K, second l2) = first l2))

product(x : F, s : SegmentBinding F) ==
  k := kernel(variable s)$K
  dm := dummy
  opdprod [eval(x, k, dm), dm, k::F, lo segment s, hi segment s]

summation(x : F, s : SegmentBinding F) ==
  k := kernel(variable s)$K
  dm := dummy
  opdsum [eval(x, k, dm), dm, k::F, lo segment s, hi segment s]
```

```
    smpfact(p, l) ==
      map(x +-> K2fact(x, l), x +-> x::F, p)$PolynomialCategoryLifting(
        IndexedExponents K, K, R, SMP, F)

    K2fact(k, l) ==
      empty? [v for v in variables(kf := k::F) | member?(v, l)] => kf
      empty?(args : List F := [facts(a, l) for a in argument k]) => kf
      is?(k, opperm) =>
        factorial(n := first args) / factorial(n - second args)
      is?(k, opbinom) =>
        n := first args
        p := second args
        factorial(n) / (factorial(p) * factorial(n-p))
      (operator k) args

    operator op ==
      is?(op, 'factorial)   => opfact
      is?(op, 'permutation) => opperm
      is?(op, 'binomial)    => opbinom
      is?(op, 'summation)   => opsum
      is?(op, '%defsum)     => opdsum
      is?(op, 'product)     => opprod
      is?(op, '%defprod)    => opdprod
      is?(op, POWER)                 => oppow
      error "Not a combinatorial operator"

    iprod l ==
      zero? first l => 0
--      one? first l => 1
      (first l = 1) => 1
      kernel(opprod, l)

    isum l ==
      zero? first l => 0
      kernel(opsum, l)

    idprod l ==
      member?(retract(second l)@SE, variables first l) =>
        kernel(opdprod, l)
      first(l) ^ (fourth rest l - fourth l + 1)

    idsum l ==
      member?(retract(second l)@SE, variables first l) =>
        kernel(opdsum, l)
      first(l) * (fourth rest l - fourth l + 1)
```

```
    ifact x ==
--      zero? x or one? x => 1
      zero? x or (x = 1) => 1
      kernel(opfact, x)

    ibinom l ==
      n := first l
      ((p := second l) = 0) or (p = n) => 1
--      one? p or (p = n - 1) => n
      (p = 1) or (p = n - 1) => n
      kernel(opbinom, l)

    iperm l ==
      zero? second l => 1
      kernel(opperm, l)

    if R has RetractableTo Z then
      iidsum l ==
        (r1 := retractIfCan(fourth l)@Union(Z,"failed"))
         case "failed" or
          (r2 := retractIfCan(fourth rest l)@Union(Z,"failed"))
            case "failed" or
             (k := retractIfCan(second l)@Union(K,"failed")) case "failed"
               => idsum l
        +/[eval(first l, k::K, i::F) for i in r1::Z .. r2::Z]

      iidprod l ==
        (r1 := retractIfCan(fourth l)@Union(Z,"failed"))
         case "failed" or
          (r2 := retractIfCan(fourth rest l)@Union(Z,"failed"))
            case "failed" or
             (k := retractIfCan(second l)@Union(K,"failed")) case "failed"
               => idprod l
        */[eval(first l, k::K, i::F) for i in r1::Z .. r2::Z]

      iiipow l ==
          (u := isExpt(x := first l, OPEXP)) case "failed" => kernel(oppow, l)
          rec := u::Record(var : K, exponent : Z)
          y := first argument(rec.var)
          (r := retractIfCan(y)@Union(Fraction Z, "failed")) case
              "failed" => kernel(oppow, l)
          (operator(rec.var)) (rec.exponent * y * second l)

      if F has RadicalCategory then
        ipow l ==
          (r := retractIfCan(second l)@Union(Fraction Z,"failed"))
```

```
            case "failed" => iiipow l
          first(l) ^ (r::Fraction(Z))
      else
        ipow l ==
          (r := retractIfCan(second l)@Union(Z, "failed"))
            case "failed" => iiipow l
          first(l) ^ (r::Z)

    else
      ipow l ==
        zero?(x := first l) =>
          zero? second l => error "0 ^ 0"
          0
--        one? x or zero?(n := second l) => 1
        (x = 1) or zero?(n : F := second l) => 1
--        one? n => x
        (n = 1) => x
        (u := isExpt(x, OPEXP)) case "failed" => kernel(oppow, l)
        rec := u::Record(var : K, exponent : Z)
--        one?(y := first argument(rec.var)) or y = -1 =>
        ((y := first argument(rec.var))=1) or y = -1 =>
            (operator(rec.var)) (rec.exponent * y * n)
        kernel(oppow, l)

    if R has CombinatorialFunctionCategory then
      iifact x ==
        (r := retractIfCan(x)@Union(R,"failed")) case "failed" => ifact x
        factorial(r::R)::F

      iiperm l ==
        (r1 := retractIfCan(first l)@Union(R,"failed")) case "failed" or
          (r2 := retractIfCan(second l)@Union(R,"failed")) case "failed"
            => iperm l
        permutation(r1::R, r2::R)::F

      if R has RetractableTo(Z) and F has Algebra(Fraction(Z)) then

         -- Try to explicitly evaluete binomial coefficient
         -- We currently simplify binomial coefficients for non-negative
         -- integral second argument, using the formula
         -- $$ \binom{n}{k}=\frac{1}{k!}\prod_{i=0..k-1} (n-i), $$
         iibinom l ==
           (s := retractIfCan(second l)@Union(R,"failed")) case R and
              (t := retractIfCan(s)@Union(Z,"failed")) case Z and t>0 =>
                ans := 1::F
                for i in 0..t-1 repeat
```

```
                ans := ans*(first l - i::R::F)
            (1/factorial t) * ans
       (s := retractIfCan(first l-second l)@Union(R,"failed")) case R and
         (t := retractIfCan(s)@Union(Z,"failed")) case Z and t>0 =>
            ans := 1::F
            for i in 1..t repeat
                ans := ans*(second l+i::R::F)
            (1/factorial t) * ans
       (r1 := retractIfCan(first l)@Union(R,"failed")) case "failed" or
         (r2 := retractIfCan(second l)@Union(R,"failed")) case "failed"
           => ibinom l
       binomial(r1::R, r2::R)::F
  else
     iibinom l ==
       (r1 := retractIfCan(first l)@Union(R,"failed")) case "failed" or
         (r2 := retractIfCan(second l)@Union(R,"failed")) case "failed"
           => ibinom l
       binomial(r1::R, r2::R)::F

else
  iifact x  == ifact x
  iibinom l == ibinom l
  iiperm l  == iperm l

if R has ElementaryFunctionCategory then
  iipow l ==
    (r1 := retractIfCan(first l)@Union(R,"failed")) case "failed" or
      (r2 := retractIfCan(second l)@Union(R,"failed")) case "failed"
        => ipow l
    (r1::R ^ r2::R)::F
else
  iipow l == ipow l

if F has ElementaryFunctionCategory then

  dvpow2 l == if zero?(first l) then
                0
              else
                log(first l) * first(l) ^ second(l)

evaluate(opfact, iifact)$BasicOperatorFunctions1(F)
evaluate(oppow, iipow)
evaluate(opperm, iiperm)
evaluate(opbinom, iibinom)
evaluate(opsum, isum)
evaluate(opdsum, iidsum)
```

```
evaluate(opprod, iprod)
evaluate(opdprod, iidprod)
derivative(oppow, [dvpow1, dvpow2])
setProperty(opsum,   SPECIALDIFF, dvsum@((List F, SE) -> F) pretend None)
setProperty(opdsum,  SPECIALDIFF, dvdsum@((List F, SE)->F) pretend None)
setProperty(opprod,  SPECIALDIFF, dvprod@((List F, SE)->F) pretend None)
setProperty(opdprod, SPECIALDIFF, dvdprod@((List F, SE)->F) pretend None)

setProperty(opsum,   SPECIALDISP, dsum@(List F -> O) pretend None)
setProperty(opdsum,  SPECIALDISP, ddsum@(List F -> O) pretend None)
setProperty(opprod,  SPECIALDISP, dprod@(List F -> O) pretend None)
setProperty(opdprod, SPECIALDISP, ddprod@(List F -> O) pretend None)
setProperty(opsum,   SPECIALEQUAL, equalsumprod@((K, K) -> Boolean) pretend None)
setProperty(opdsum,  SPECIALEQUAL, equaldsumprod@((K, K) -> Boolean) pretend None)
setProperty(opprod,  SPECIALEQUAL, equalsumprod@((K, K) -> Boolean) pretend None)
setProperty(opdprod, SPECIALEQUAL, equaldsumprod@((K, K) -> Boolean) pretend None)
```

2 package FSPECF FunctionalSpecialFunction

⟨*package FSPECF FunctionalSpecialFunction*⟩≡

```
)abbrev package FSPECF FunctionalSpecialFunction
)boot $tryRecompileArguments := nil
++ Provides the special functions
++ Author: Manuel Bronstein
++ Date Created: 18 Apr 1989
++ Date Last Updated: 4 October 1993
++ Description: Provides some special functions over an integral domain.
++ Keywords: special, function.
FunctionalSpecialFunction(R, F) : Exports == Implementation where
  R : Join(Comparable, IntegralDomain)
  F : FunctionSpace R

  OP  ==> BasicOperator
  K   ==> Kernel F
  SE  ==> Symbol
  SPECIALDIFF  ==> '%specialDiff

  Exports ==> with
    belong? : OP -> Boolean
      ++ belong?(op) is true if op is a special function operator;
    operator : OP -> OP
      ++ operator(op) returns a copy of op with the domain-dependent
      ++ properties appropriate for F;
      ++ error if op is not a special function operator
    abs     : F -> F
      ++ abs(f) returns the absolute value operator applied to f
    Gamma   : F -> F
      ++ Gamma(f) returns the formal Gamma function applied to f
    Gamma   : (F, F) -> F
      ++ Gamma(a, x) returns the incomplete Gamma function applied to a and x
    Beta :      (F, F) -> F
      ++ Beta(x, y) returns the beta function applied to x and y
    digamma :   F->F
      ++ digamma(x) returns the digamma function applied to x
    polygamma : (F, F) ->F
      ++ polygamma(x, y) returns the polygamma function applied to x and y
    besselJ :   (F, F) -> F
      ++ besselJ(x, y) returns the besselj function applied to x and y
    besselY :   (F, F) -> F
      ++ besselY(x, y) returns the bessely function applied to x and y
    besselI :   (F, F) -> F
      ++ besselI(x, y) returns the besseli function applied to x and y
    besselK :   (F, F) -> F
```

```
  ++ besselK(x, y) returns the besselk function applied to x and y
airyAi :    F -> F
  ++ airyAi(x) returns the Airy Ai function applied to x
airyAiPrime :    F -> F
  ++ airyAiPrime(x) returns the derivative of Airy Ai function applied to x
airyBi :    F -> F
  ++ airyBi(x) returns the Airy Bi function applied to x
airyBiPrime :    F -> F
  ++ airyBiPrime(x) returns the derivative of Airy Bi function applied to x
lambertW :  F -> F
  ++ lambertW(x) is the Lambert W function at x
polylog : (F, F) -> F
  ++ polylog(s, x) is the polylogarithm of order s at x
weierstrassP : (F, F, F) -> F
  ++ weierstrassP(g2, g3, x)
weierstrassPPrime : (F, F, F) -> F
  ++ weierstrassPPrime(g2, g3, x)
weierstrassSigma : (F, F, F) -> F
  ++ weierstrassSigma(g2, g3, x)
weierstrassZeta : (F, F, F) -> F
  ++ weierstrassZeta(g2, g3, x)
-- weierstrassPInverse : (F, F, F) -> F
--    ++ weierstrassPInverse(g2, g3, z) is the inverse of Weierstrass
--    ++ P function, defined by the formula
--    ++ \spad{weierstrassP(g2, g3, weierstrassPInverse(g2, g3, z)) = z}
whittakerM : (F, F, F) -> F
    ++ whittakerM(k, m, z) is the Whittaker M function
whittakerW : (F, F, F) -> F
    ++ whittakerW(k, m, z) is the Whittaker W function
angerJ : (F, F) -> F
    ++ angerJ(v, z) is the Anger J function
weberE : (F, F) -> F
    ++ weberE(v, z) is the Weber E function
struveH : (F, F) -> F
    ++ struveH(v, z) is the Struve H function
struveL : (F, F) -> F
    ++ struveL(v, z) is the Struve L function defined by the formula
    ++ \spad{struveL(v, z) = -%i^exp(-v*%pi*%i/2)*struveH(v, %i*z)}
hankelH1 : (F, F) -> F
    ++ hankelH1(v, z) is first Hankel function (Bessel function of
    ++ the third kind)
hankelH2 : (F, F) -> F
    ++ hankelH2(v, z) is the second Hankel function (Bessel function of
    ++ the third kind)
lommelS1 : (F, F, F) -> F
    ++ lommelS1(mu, nu, z) is the Lommel s function
```

```
lommelS2 : (F, F, F) -> F
    ++ lommelS2(mu, nu, z) is the Lommel S function
kummerM : (F, F, F) -> F
    ++ kummerM(a, b, z) is the Kummer M function
kummerU : (F, F, F) -> F
    ++ kummerU(a, b, z) is the Kummer U function
legendreP : (F, F, F) -> F
    ++ legendreP(nu, mu, z) is the Legendre P function
legendreQ : (F, F, F) -> F
    ++ legendreQ(nu, mu, z) is the Legendre Q function
kelvinBei : (F, F) -> F
    ++ kelvinBei(v, z) is the Kelvin bei function defined by equality
    ++ \spad{kelvinBei(v, z) = imag(besselJ(v, exp(3*%pi*%i/4)*z))}
    ++ for z and v real
kelvinBer : (F, F) -> F
    ++ kelvinBer(v, z) is the Kelvin ber function defined by equality
    ++ \spad{kelvinBer(v, z) = real(besselJ(v, exp(3*%pi*%i/4)*z))}
    ++ for z and v real
kelvinKei : (F, F) -> F
    ++ kelvinKei(v, z) is the Kelvin kei function defined by equality
    ++ \spad{kelvinKei(v, z) =
    ++ imag(exp(-v*%pi*%i/2)*besselK(v, exp(%pi*%i/4)*z))}
    ++ for z and v real
kelvinKer : (F, F) -> F
    ++ kelvinKer(v, z) is the Kelvin kei function defined by equality
    ++ \spad{kelvinKer(v, z) =
    ++ real(exp(-v*%pi*%i/2)*besselK(v, exp(%pi*%i/4)*z))}
    ++ for z and v real
ellipticK : F -> F
    ++ ellipticK(m) is the complete elliptic integral of the
    ++ first kind: \spad{ellipticK(m) =
    ++ integrate(1/sqrt((1-t^2)*(1-m*t^2)), t = 0..1)}
ellipticE : F -> F
    ++ ellipticE(m) is the complete elliptic integral of the
    ++ second kind: \spad{ellipticE(m) =
    ++ integrate(sqrt(1-m*t^2)/sqrt(1-t^2), t = 0..1)}
ellipticE : (F, F) -> F
    ++ ellipticE(z, m) is the incomplete elliptic integral of the
    ++ second kind: \spad{ellipticE(z, m) =
    ++ integrate(sqrt(1-m*t^2)/sqrt(1-t^2), t = 0..z)}
ellipticF : (F, F) -> F
    ++ ellipticF(z, m) is the incomplete elliptic integral of the
    ++ first kind : \spad{ellipticF(z, m) =
    ++ integrate(1/sqrt((1-t^2)*(1-m*t^2)), t = 0..z)}
ellipticPi : (F, F, F) -> F
    ++ ellipticPi(z, n, m) is the incomplete elliptic integral of
```

```
    ++ the third kind: \spad{ellipticPi(z, n, m) =
    ++ integrate(1/((1-n*t^2)*sqrt((1-t^2)*(1-m*t^2))), t = 0..z)}
jacobiSn : (F, F) -> F
    ++ jacobiSn(z, m) is the Jacobi elliptic sn function, defined
    ++ by the formula \spad{jacobiSn(ellipticF(z, m), m) = z}
jacobiCn : (F, F) -> F
    ++ jacobiCn(z, m) is the Jacobi elliptic cn function, defined
    ++ by \spad{jacobiCn(z, m)^2 + jacobiSn(z, m)^2 = 1} and
    ++ \spad{jacobiCn(0, m) = 1}
jacobiDn : (F, F) -> F
    ++ jacobiDn(z, m) is the Jacobi elliptic dn function, defined
    ++ by \spad{jacobiDn(z, m)^2 + m*jacobiSn(z, m)^2 = 1} and
    ++ \spad{jacobiDn(0, m) = 1}
jacobiZeta : (F, F) -> F
    ++ jacobiZeta(z, m) is the Jacobi elliptic zeta function, defined
    ++ by \spad{D(jacobiZeta(z, m), z) =
    ++ jacobiDn(z, m)^2 - ellipticE(m)/ellipticK(m)} and
    ++ \spad{jacobiZeta(0, m) = 0}.
jacobiTheta : (F, F) -> F
    ++ jacobiTheta(q, z) is the third Jacobi Theta function
lerchPhi : (F, F, F) -> F
    ++ lerchPhi(z, s, a) is the Lerch Phi function
riemannZeta : F -> F
    ++ riemannZeta(z) is the Riemann Zeta function
charlierC : (F, F, F) -> F
    ++ charlierC(n, a, z) is the Charlier polynomial
hermiteH : (F, F) -> F
    ++ hermiteH(n, z) is the Hermite polynomial
jacobiP : (F, F, F, F) -> F
    ++ jacobiP(n, a, b, z) is the Jacobi polynomial
laguerreL: (F, F, F) -> F
    ++ laguerreL(n, a, z) is the Laguerre polynomial
meixnerM : (F, F, F, F) -> F
    ++ meixnerM(n, b, c, z) is the Meixner polynomial
if F has RetractableTo(Integer) then
    hypergeometricF : (List F, List F, F) -> F
      ++ hypergeometricF(la, lb, z) is the generalized hypergeometric
      ++ function
    meijerG : (List F, List F, List F, List F, F) -> F
      ++ meijerG(la, lb, lc, ld, z) is the meijerG function
-- Functions below should be local but conditional
iiGamma : F -> F
  ++ iiGamma(x) should be local but conditional;
iiabs     : F -> F
  ++ iiabs(x) should be local but conditional;
iiBeta     : List F -> F
```

```
    ++ iiBeta(x) should be local but conditional;
  iidigamma  : F -> F
    ++ iidigamma(x) should be local but conditional;
  iipolygamma : List F -> F
    ++ iipolygamma(x) should be local but conditional;
  iiBesselJ  : List F -> F
    ++ iiBesselJ(x) should be local but conditional;
  iiBesselY  : List F -> F
    ++ iiBesselY(x) should be local but conditional;
  iiBesselI  : List F -> F
    ++ iiBesselI(x) should be local but conditional;
  iiBesselK  : List F -> F
    ++ iiBesselK(x) should be local but conditional;
  iiAiryAi   : F -> F
    ++ iiAiryAi(x) should be local but conditional;
  iiAiryAiPrime : F -> F
    ++ iiAiryAiPrime(x) should be local but conditional;
  iiAiryBi   : F -> F
    ++ iiAiryBi(x) should be local but conditional;
  iiAiryBiPrime : F -> F
    ++ iiAiryBiPrime(x) should be local but conditional;
  iAiryAi   : F -> F
    ++ iAiryAi(x) should be local but conditional;
  iAiryAiPrime : F -> F
    ++ iAiryAiPrime(x) should be local but conditional;
  iAiryBi   : F -> F
    ++ iAiryBi(x) should be local but conditional;
  iAiryBiPrime : F -> F
    ++ iAiryBiPrime(x) should be local but conditional;
  iiHypergeometricF : List F -> F
    ++ iiHypergeometricF(l) should be local but conditional;
  iiPolylog : (F, F) -> F
    ++ iiPolylog(x, s) should be local but conditional;
  iLambertW : F -> F
    ++ iLambertW(x) should be local but conditional;

Implementation ==> add

  SPECIAL := 'special

  INP ==> InputForm

  SPECIALINPUT ==> '%specialInput

  iabs      : F -> F
  iGamma    : F -> F
```

```
iBeta     : (F, F) -> F
idigamma  : F -> F
iiipolygamma : (F, F) -> F
iiiBesselJ  : (F, F) -> F
iiiBesselY  : (F, F) -> F
iiiBesselI  : (F, F) -> F
iiiBesselK  : (F, F) -> F
iPolylog : List F -> F

iWeierstrassP : (F, F, F) -> F
iWeierstrassPPrime : (F, F, F) -> F
iWeierstrassSigma : (F, F, F) -> F
iWeierstrassZeta : (F, F, F) -> F

iiWeierstrassP : List F -> F
iiWeierstrassPPrime : List F -> F
iiWeierstrassSigma : List F -> F
iiWeierstrassZeta : List F -> F
iiMeijerG : List F -> F


opabs       := operator('abs)$CommonOperators
opGamma     := operator('Gamma)$CommonOperators
opGamma2    := operator('Gamma2)$CommonOperators
opBeta      := operator('Beta)$CommonOperators
opdigamma   := operator('digamma)$CommonOperators
oppolygamma := operator('polygamma)$CommonOperators
opBesselJ   := operator('besselJ)$CommonOperators
opBesselY   := operator('besselY)$CommonOperators
opBesselI   := operator('besselI)$CommonOperators
opBesselK   := operator('besselK)$CommonOperators
opAiryAi    := operator('airyAi)$CommonOperators
opAiryAiPrime := operator('airyAiPrime)$CommonOperators
opAiryBi    := operator('airyBi)$CommonOperators
opAiryBiPrime := operator('airyBiPrime)$CommonOperators
opLambertW := operator('lambertW)$CommonOperators
opPolylog := operator('polylog)$CommonOperators
opWeierstrassP := operator('weierstrassP)$CommonOperators
opWeierstrassPPrime := operator('weierstrassPPrime)$CommonOperators
opWeierstrassSigma := operator('weierstrassSigma)$CommonOperators
opWeierstrassZeta := operator('weierstrassZeta)$CommonOperators
opHypergeometricF := operator('hypergeometricF)$CommonOperators
opMeijerG := operator('meijerG)$CommonOperators
opCharlierC := operator('charlierC)$CommonOperators
opHermiteH := operator('hermiteH)$CommonOperators
opJacobiP := operator('jacobiP)$CommonOperators
```

```
opLaguerreL := operator('laguerreL)$CommonOperators
opMeixnerM := operator('meixnerM)$CommonOperators
op_log_gamma := operator('%logGamma)$CommonOperators
op_eis := operator('%eis)$CommonOperators
op_erfs := operator('%erfs)$CommonOperators
op_erfis := operator('%erfis)$CommonOperators

abs x         == opabs x
Gamma(x)       == opGamma(x)
Gamma(a, x)     == opGamma2(a, x)
Beta(x, y)      == opBeta(x, y)
digamma x      == opdigamma(x)
polygamma(k, x)== oppolygamma(k, x)
besselJ(a, x)  == opBesselJ(a, x)
besselY(a, x)  == opBesselY(a, x)
besselI(a, x)  == opBesselI(a, x)
besselK(a, x)  == opBesselK(a, x)
airyAi(x)     == opAiryAi(x)
airyAiPrime(x) == opAiryAiPrime(x)
airyBi(x)     == opAiryBi(x)
airyBiPrime(x) == opAiryBiPrime(x)
lambertW(x) == opLambertW(x)
polylog(s, x) == opPolylog(s, x)
weierstrassP(g2, g3, x) == opWeierstrassP(g2, g3, x)
weierstrassPPrime(g2, g3, x) == opWeierstrassPPrime(g2, g3, x)
weierstrassSigma(g2, g3, x) == opWeierstrassSigma(g2, g3, x)
weierstrassZeta(g2, g3, x) == opWeierstrassZeta(g2, g3, x)
if F has RetractableTo(Integer) then
    hypergeometricF(a, b, z) ==
        nai := #a
        nbi := #b
        z = 0 and nai <= nbi + 1 => 1
        p := (#a)::F
        q := (#b)::F
        opHypergeometricF concat(concat(a, concat(b, [z])), [p, q])

    meijerG(a, b, c, d, z) ==
        n1 := (#a)::F
        n2 := (#b)::F
        m1 := (#c)::F
        m2 := (#d)::F
        opMeijerG concat(concat(a, concat(b,
                     concat(c, concat(d, [z])))), [n1, n2, m1, m2])

import from List Kernel(F)
opdiff := operator(operator('%diff)$CommonOperators)$F
```

```
    dummy ==> new()$SE :: F

    ahalf : F := recip(2::F)::F
    athird : F    := recip(3::F)::F
    afourth : F   := recip(4::F)::F
    asixth : F := recip(6::F)::F
    twothirds : F := 2*athird
    threehalfs : F := 3*ahalf

    -- Helpers for partially defined derivatives

    grad2(l : List F, t : SE, op : OP, d2 : (F, F) -> F ) : F ==
        x1 := l(1)
        x2 := l(2)
        dm := dummy
        differentiate(x1, t)*kernel(opdiff, [op [dm, x2], dm, x1])
             + differentiate(x2, t)*d2(x1, x2)

    grad3(l : List F, t : SE, op : OP, d3 : (F, F, F) -> F ) : F ==
        x1 := l(1)
        x2 := l(2)
        x3 := l(3)
        dm1 := dummy
        dm2 := dummy
        differentiate(x1, t)*kernel(opdiff, [op [dm1, x2, x3], dm1, x1])
           + differentiate(x2, t)*kernel(opdiff, [op [x1, dm2, x3], dm2, x2])
             + differentiate(x3, t)*d3(x1, x2, x3)

    grad4(l : List F, t : SE, op : OP, d4 : (F, F, F, F) -> F ) : F ==
        x1 := l(1)
        x2 := l(2)
        x3 := l(3)
        x4 := l(4)
        dm1 := dummy
        dm2 := dummy
        dm3 := dummy
        kd1 := kernel(opdiff, [op [dm1, x2, x3, x4], dm1, x1])
        kd2 := kernel(opdiff, [op [x1, dm2, x3, x4], dm2, x2])
        kd3 := kernel(opdiff, [op [x1, x2, dm3, x4], dm3, x3])
        differentiate(x1, t)*kd1 + differentiate(x2, t)*kd2 +
          differentiate(x3, t)*kd3 +
            differentiate(x4, t)*d4(x1, x2, x3, x4)

    -- handle WeierstrassPInverse

)if false
```

```
    opWeierstrassPInverse := operator('weierstrassPInverse)$CommonOperators

    weierstrassPInverse(g2, g3, z) == opWeierstrassPInverse(g2, g3, z)

    eWeierstrassPInverse(g2 : F, g3 : F, z : F) : F ==
        kernel(opWeierstrassPInverse, [g2, g3, z])

    elWeierstrassPInverse(l : List F) : F == eWeierstrassPInverse(l(1), l(2), l(3))
    evaluate(opWeierstrassPInverse, elWeierstrassPInverse)$BasicOperatorFunctions1(F)

    if F has RadicalCategory then

        eWeierstrassPInverseGrad_g2(l : List F) : F ==
            g2 := l(1)
            g3 := l(2)
            z := l(3)
            error "unimplemented"

        eWeierstrassPInverseGrad_g3(l : List F) : F ==
            g2 := l(1)
            g3 := l(2)
            z := l(3)
            error "unimplemented"

        eWeierstrassPInverseGrad_z(l : List F) : F ==
            g2 := l(1)
            g3 := l(2)
            z := l(3)
            1/sqrt(4*z^3 - g2*z - g3)

        derivative(opWeierstrassPInverse, [eWeierstrassPInverseGrad_g2,
               eWeierstrassPInverseGrad_g3, eWeierstrassPInverseGrad_z])

)endif

    -- handle WhittakerM

    opWhittakerM := operator('whittakerM)$CommonOperators

    whittakerM(k, m, z) == opWhittakerM(k, m, z)

    eWhittakerM(k : F, m : F, z : F) : F ==
        kernel(opWhittakerM, [k, m, z])

    elWhittakerM(l : List F) : F == eWhittakerM(l(1), l(2), l(3))
```

```
evaluate(opWhittakerM, elWhittakerM)$BasicOperatorFunctions1(F)

eWhittakerMGrad_z(k : F, m : F, z : F) : F ==
    (ahalf - k/z)*whittakerM(k, m, z) +
        (ahalf + k + m)*whittakerM(k + 1, m, z)/z

dWhittakerM(l : List F, t : SE) : F ==
    grad3(l, t, opWhittakerM, eWhittakerMGrad_z)

setProperty(opWhittakerM, SPECIALDIFF, dWhittakerM@((List F, SE)->F)
                                             pretend None)

-- handle WhittakerW

opWhittakerW := operator('whittakerW)$CommonOperators

whittakerW(k, m, z) == opWhittakerW(k, m, z)

eWhittakerW(k : F, m : F, z : F) : F ==
    kernel(opWhittakerW, [k, m, z])

elWhittakerW(l : List F) : F == eWhittakerW(l(1), l(2), l(3))
evaluate(opWhittakerW, elWhittakerW)$BasicOperatorFunctions1(F)

eWhittakerWGrad_z(k : F, m : F, z : F) : F ==
    (ahalf - k/z)*whittakerW(k, m, z) - whittakerW(k + 1, m, z)/z

dWhittakerW(l : List F, t : SE) : F ==
    grad3(l, t, opWhittakerW, eWhittakerWGrad_z)

setProperty(opWhittakerW, SPECIALDIFF, dWhittakerW@((List F, SE)->F)
                                             pretend None)

-- handle AngerJ

opAngerJ := operator('angerJ)$CommonOperators

angerJ(v, z) == opAngerJ(v, z)

if F has TranscendentalFunctionCategory then

    eAngerJ(v : F, z : F) : F ==
        z = 0 => sin(v*pi())/(v*pi())
        kernel(opAngerJ, [v, z])

    elAngerJ(l : List F) : F == eAngerJ(l(1), l(2))
```

```
    evaluate(opAngerJ, elAngerJ)$BasicOperatorFunctions1(F)

    eAngerJGrad_z(v : F, z : F) : F ==
        -angerJ(v + 1, z) + v*angerJ(v, z)/z - sin(v*pi())/(pi()*z)

    dAngerJ(l : List F, t : SE) : F ==
        grad2(l, t, opAngerJ, eAngerJGrad_z)

    setProperty(opAngerJ, SPECIALDIFF, dAngerJ@((List F, SE)->F)
                                         pretend None)

else

    eeAngerJ(l : List F) : F == kernel(opAngerJ, l)
    evaluate(opAngerJ, eeAngerJ)$BasicOperatorFunctions1(F)

-- handle WeberE

opWeberE := operator('weberE)$CommonOperators

weberE(v, z) == opWeberE(v, z)

if F has TranscendentalFunctionCategory then

    eWeberE(v : F, z : F) : F ==
        z = 0 => 2*sin(ahalf*v*pi())^2/(v*pi())
        kernel(opWeberE, [v, z])

    elWeberE(l : List F) : F == eWeberE(l(1), l(2))
    evaluate(opWeberE, elWeberE)$BasicOperatorFunctions1(F)

    eWeberEGrad_z(v : F, z : F) : F ==
        -weberE(v + 1, z) + v*weberE(v, z)/z - (1 - cos(v*pi()))/(pi()*z)

    dWeberE(l : List F, t : SE) : F ==
        grad2(l, t, opWeberE, eWeberEGrad_z)

    setProperty(opWeberE, SPECIALDIFF, dWeberE@((List F, SE)->F)
                                         pretend None)

else

    eeWeberE(l :  List F) : F == kernel(opWeberE, l)
    evaluate(opWeberE, eeWeberE)$BasicOperatorFunctions1(F)

-- handle StruveH
```

```
opStruveH := operator('struveH)$CommonOperators

struveH(v, z) == opStruveH(v, z)

eStruveH(v : F, z : F) : F ==
    kernel(opStruveH, [v, z])

elStruveH(l : List F) : F == eStruveH(l(1), l(2))
evaluate(opStruveH, elStruveH)$BasicOperatorFunctions1(F)

if F has TranscendentalFunctionCategory
   and F has RadicalCategory then

    eStruveHGrad_z(v : F, z : F) : F ==
        -struveH(v + 1, z) + v*struveH(v, z)/z +
           (ahalf*z)^v/(sqrt(pi())*Gamma(v + threehalfs))

    dStruveH(l : List F, t : SE) : F ==
        grad2(l, t, opStruveH, eStruveHGrad_z)

    setProperty(opStruveH, SPECIALDIFF, dStruveH@((List F, SE)->F)
                                         pretend None)

-- handle StruveL

opStruveL := operator('struveL)$CommonOperators

struveL(v, z) == opStruveL(v, z)

eStruveL(v : F, z : F) : F ==
    kernel(opStruveL, [v, z])

elStruveL(l : List F) : F == eStruveL(l(1), l(2))
evaluate(opStruveL, elStruveL)$BasicOperatorFunctions1(F)

if F has TranscendentalFunctionCategory
   and F has RadicalCategory then

    eStruveLGrad_z(v : F, z : F) : F ==
        struveL(v + 1, z) + v*struveL(v, z)/z +
           (ahalf*z)^v/(sqrt(pi())*Gamma(v + threehalfs))

    dStruveL(l : List F, t : SE) : F ==
        grad2(l, t, opStruveL, eStruveLGrad_z)
```

```
    setProperty(opStruveL, SPECIALDIFF, dStruveL@((List F, SE)->F)
                                              pretend None)

-- handle HankelH1

opHankelH1 := operator('hankelH1)$CommonOperators

hankelH1(v, z) == opHankelH1(v, z)

eHankelH1(v : F, z : F) : F ==
    kernel(opHankelH1, [v, z])

elHankelH1(l : List F) : F == eHankelH1(l(1), l(2))
evaluate(opHankelH1, elHankelH1)$BasicOperatorFunctions1(F)

eHankelH1Grad_z(v : F, z : F) : F ==
    -hankelH1(v + 1, z) + v*hankelH1(v, z)/z

dHankelH1(l : List F, t : SE) : F ==
    grad2(l, t, opHankelH1, eHankelH1Grad_z)

setProperty(opHankelH1, SPECIALDIFF, dHankelH1@((List F, SE)->F)
                                              pretend None)

-- handle HankelH2

opHankelH2 := operator('hankelH2)$CommonOperators

hankelH2(v, z) == opHankelH2(v, z)

eHankelH2(v : F, z : F) : F ==
    kernel(opHankelH2, [v, z])

elHankelH2(l : List F) : F == eHankelH2(l(1), l(2))
evaluate(opHankelH2, elHankelH2)$BasicOperatorFunctions1(F)

eHankelH2Grad_z(v : F, z : F) : F ==
    -hankelH2(v + 1, z) + v*hankelH2(v, z)/z

dHankelH2(l : List F, t : SE) : F ==
    grad2(l, t, opHankelH2, eHankelH2Grad_z)

setProperty(opHankelH2, SPECIALDIFF, dHankelH2@((List F, SE)->F)
                                              pretend None)

-- handle LommelS1
```

```
opLommelS1 := operator('lommelS1)$CommonOperators

lommelS1(m, v, z) == opLommelS1(m, v, z)

eLommelS1(m : F, v : F, z : F) : F ==
    kernel(opLommelS1, [m, v, z])

elLommelS1(l : List F) : F == eLommelS1(l(1), l(2), l(3))
evaluate(opLommelS1, elLommelS1)$BasicOperatorFunctions1(F)

eLommelS1Grad_z(m : F, v : F, z : F) : F ==
    -v*lommelS1(m, v, z)/z + (m + v - 1)*lommelS1(m - 1, v - 1, z)

dLommelS1(l : List F, t : SE) : F ==
    grad3(l, t, opLommelS1, eLommelS1Grad_z)

setProperty(opLommelS1, SPECIALDIFF, dLommelS1@((List F, SE)->F)
                                             pretend None)

-- handle LommelS2

opLommelS2 := operator('lommelS2)$CommonOperators

lommelS2(mu, nu, z) == opLommelS2(mu, nu, z)

eLommelS2(mu : F, nu : F, z : F) : F ==
    kernel(opLommelS2, [mu, nu, z])

elLommelS2(l : List F) : F == eLommelS2(l(1), l(2), l(3))
evaluate(opLommelS2, elLommelS2)$BasicOperatorFunctions1(F)

eLommelS2Grad_z(m : F, v : F, z : F) : F ==
    -v*lommelS2(m, v, z)/z + (m + v - 1)*lommelS2(m - 1, v - 1, z)

dLommelS2(l : List F, t : SE) : F ==
    grad3(l, t, opLommelS2, eLommelS2Grad_z)

setProperty(opLommelS2, SPECIALDIFF, dLommelS2@((List F, SE)->F)
                                             pretend None)

-- handle KummerM

opKummerM := operator('kummerM)$CommonOperators

kummerM(mu, nu, z) == opKummerM(mu, nu, z)
```

```
eKummerM(a : F, b : F, z : F) : F ==
    z = 0 => 1
    kernel(opKummerM, [a, b, z])

elKummerM(l : List F) : F == eKummerM(l(1), l(2), l(3))
evaluate(opKummerM, elKummerM)$BasicOperatorFunctions1(F)

eKummerMGrad_z(a : F, b : F, z : F) : F ==
    ((z + a - b)*kummerM(a, b, z)+(b - a)*kummerM(a - 1, b, z))/z

dKummerM(l : List F, t : SE) : F ==
    grad3(l, t, opKummerM, eKummerMGrad_z)

setProperty(opKummerM, SPECIALDIFF, dKummerM@((List F, SE)->F)
                                             pretend None)

-- handle KummerU

opKummerU := operator('kummerU)$CommonOperators

kummerU(a, b, z) == opKummerU(a, b, z)

eKummerU(a : F, b : F, z : F) : F ==
    kernel(opKummerU, [a, b, z])

elKummerU(l : List F) : F == eKummerU(l(1), l(2), l(3))
evaluate(opKummerU, elKummerU)$BasicOperatorFunctions1(F)

eKummerUGrad_z(a : F, b : F, z : F) : F ==
    ((z + a - b)*kummerU(a, b, z) - kummerU(a - 1, b, z))/z

dKummerU(l : List F, t : SE) : F ==
    grad3(l, t, opKummerU, eKummerUGrad_z)

setProperty(opKummerU, SPECIALDIFF, dKummerU@((List F, SE)->F)
                                             pretend None)

-- handle LegendreP

opLegendreP := operator('legendreP)$CommonOperators

legendreP(nu, mu, z) == opLegendreP(nu, mu, z)

eLegendreP(nu : F, mu : F, z : F) : F ==
    kernel(opLegendreP, [nu, mu, z])
```

```
elLegendreP(l : List F) : F == eLegendreP(l(1), l(2), l(3))
evaluate(opLegendreP, elLegendreP)$BasicOperatorFunctions1(F)

eLegendrePGrad_z(nu : F, mu : F, z : F) : F ==
    (nu - mu + 1)*legendreP(nu + 1, mu, z) -
      (nu + 1)*z*legendreP(nu, mu, z)

dLegendreP(l : List F, t : SE) : F ==
    grad3(l, t, opLegendreP, eLegendrePGrad_z)

setProperty(opLegendreP, SPECIALDIFF, dLegendreP@((List F, SE)->F)
                                             pretend None)

-- handle LegendreQ

opLegendreQ := operator('legendreQ)$CommonOperators

legendreQ(nu, mu, z) == opLegendreQ(nu, mu, z)

eLegendreQ(nu : F, mu : F, z : F) : F ==
    kernel(opLegendreQ, [nu, mu, z])

elLegendreQ(l : List F) : F == eLegendreQ(l(1), l(2), l(3))
evaluate(opLegendreQ, elLegendreQ)$BasicOperatorFunctions1(F)

eLegendreQGrad_z(nu : F, mu : F, z : F) : F ==
    (nu - mu + 1)*legendreQ(nu + 1, mu, z) -
      (nu + 1)*z*legendreQ(nu, mu, z)

dLegendreQ(l : List F, t : SE) : F ==
    grad3(l, t, opLegendreQ, eLegendreQGrad_z)

setProperty(opLegendreQ, SPECIALDIFF, dLegendreQ@((List F, SE)->F)
                                             pretend None)

-- handle KelvinBei

opKelvinBei := operator('kelvinBei)$CommonOperators

kelvinBei(v, z) == opKelvinBei(v, z)

eKelvinBei(v : F, z : F) : F ==
    kernel(opKelvinBei, [v, z])

elKelvinBei(l : List F) : F == eKelvinBei(l(1), l(2))
```

```
evaluate(opKelvinBei, elKelvinBei)$BasicOperatorFunctions1(F)

if F has RadicalCategory then

    eKelvinBeiGrad_z(v : F, z : F) : F ==
        ahalf*sqrt(2::F)*(kelvinBei(v + 1, z) - kelvinBer(v + 1, z)) +
          v*kelvinBei(v, z)/z

    dKelvinBei(l : List F, t : SE) : F ==
        grad2(l, t, opKelvinBei, eKelvinBeiGrad_z)

    setProperty(opKelvinBei, SPECIALDIFF, dKelvinBei@((List F, SE)->F)
                                              pretend None)

-- handle KelvinBer

opKelvinBer := operator('kelvinBer)$CommonOperators

kelvinBer(v, z) == opKelvinBer(v, z)

eKelvinBer(v : F, z : F) : F ==
    kernel(opKelvinBer, [v, z])

elKelvinBer(l : List F) : F == eKelvinBer(l(1), l(2))
evaluate(opKelvinBer, elKelvinBer)$BasicOperatorFunctions1(F)

if F has RadicalCategory then

    eKelvinBerGrad_z(v : F, z : F) : F ==
        ahalf*sqrt(2::F)*(kelvinBer(v + 1, z) + kelvinBei(v + 1, z)) +
          v*kelvinBer(v, z)/z

    dKelvinBer(l : List F, t : SE) : F ==
        grad2(l, t, opKelvinBer, eKelvinBerGrad_z)

    setProperty(opKelvinBer, SPECIALDIFF, dKelvinBer@((List F, SE)->F)
                                              pretend None)

-- handle KelvinKei

opKelvinKei := operator('kelvinKei)$CommonOperators

kelvinKei(v, z) == opKelvinKei(v, z)

eKelvinKei(v : F, z : F) : F ==
    kernel(opKelvinKei, [v, z])
```

```
elKelvinKei(l : List F) : F == eKelvinKei(l(1), l(2))
evaluate(opKelvinKei, elKelvinKei)$BasicOperatorFunctions1(F)

if F has RadicalCategory then

    eKelvinKeiGrad_z(v : F, z : F) : F ==
        ahalf*sqrt(2::F)*(kelvinKei(v + 1, z) - kelvinKer(v + 1, z)) +
          v*kelvinKei(v, z)/z

    dKelvinKei(l : List F, t : SE) : F ==
        grad2(l, t, opKelvinKei, eKelvinKeiGrad_z)

    setProperty(opKelvinKei, SPECIALDIFF, dKelvinKei@((List F, SE)->F)
                                             pretend None)

-- handle KelvinKer

opKelvinKer := operator('kelvinKer)$CommonOperators

kelvinKer(v, z) == opKelvinKer(v, z)

eKelvinKer(v : F, z : F) : F ==
    kernel(opKelvinKer, [v, z])

elKelvinKer(l : List F) : F == eKelvinKer(l(1), l(2))
evaluate(opKelvinKer, elKelvinKer)$BasicOperatorFunctions1(F)

if F has RadicalCategory then

    eKelvinKerGrad_z(v : F, z : F) : F ==
        ahalf*sqrt(2::F)*(kelvinKer(v + 1, z) + kelvinKei(v + 1, z)) +
          v*kelvinKer(v, z)/z

    dKelvinKer(l : List F, t : SE) : F ==
        grad2(l, t, opKelvinKer, eKelvinKerGrad_z)

    setProperty(opKelvinKer, SPECIALDIFF, dKelvinKer@((List F, SE)->F)
                                             pretend None)

-- handle EllipticK

opEllipticK := operator('ellipticK)$CommonOperators

ellipticK(m) == opEllipticK(m)
```

```
eEllipticK(m : F) : F ==
    kernel(opEllipticK, [m])

elEllipticK(l : List F) : F == eEllipticK(l(1))
evaluate(opEllipticK, elEllipticK)$BasicOperatorFunctions1(F)

dEllipticK(m : F) : F ==
    ahalf*(ellipticE(m) - (1 - m)*ellipticK(m))/(m*(1 - m))

derivative(opEllipticK, dEllipticK)

-- handle one argument EllipticE

opEllipticE := operator('ellipticE)$CommonOperators

ellipticE(m) == opEllipticE(m)

eEllipticE(m : F) : F ==
    kernel(opEllipticE, [m])

elEllipticE(l : List F) : F == eEllipticE(l(1))
evaluate(opEllipticE, elEllipticE)$BasicOperatorFunctions1(F)

dEllipticE(m : F) : F ==
    ahalf*(ellipticE(m) - ellipticK(m))/m

derivative(opEllipticE, dEllipticE)

-- handle two argument EllipticE

opEllipticE2 := operator('ellipticE2)$CommonOperators

ellipticE(z, m) == opEllipticE2(z, m)

eEllipticE2(z : F, m : F) : F ==
    z = 0 => 0
    z = 1 => eEllipticE(m)
    kernel(opEllipticE2, [z, m])

elEllipticE2(l : List F) : F == eEllipticE2(l(1), l(2))
evaluate(opEllipticE2, elEllipticE2)$BasicOperatorFunctions1(F)

if F has RadicalCategory then

    eEllipticE2Grad_z(l : List F) : F ==
        z := l(1)
```

```
        m := l(2)
        sqrt(1 - m*z^2)/sqrt(1 - z^2)

    eEllipticE2Grad_m(l : List F) : F ==
        z := l(1)
        m := l(2)
        ahalf*(ellipticE(z, m) - ellipticF(z, m))/m

    derivative(opEllipticE2, [eEllipticE2Grad_z, eEllipticE2Grad_m])

inEllipticE2(li : List INP) : INP ==
    convert cons(convert('ellipticE), li)

input(opEllipticE2, inEllipticE2@((List INP) -> INP))

-- handle EllipticF

opEllipticF := operator('ellipticF)$CommonOperators

ellipticF(z, m) == opEllipticF(z, m)

eEllipticF(z : F, m : F) : F ==
    z = 0 => 0
    z = 1 => ellipticK(m)
    kernel(opEllipticF, [z, m])

elEllipticF(l : List F) : F == eEllipticF(l(1), l(2))
evaluate(opEllipticF, elEllipticF)$BasicOperatorFunctions1(F)

if F has RadicalCategory then

    eEllipticFGrad_z(l : List F) : F ==
        z := l(1)
        m := l(2)
        1/(sqrt(1 - m*z^2)*sqrt(1 - z^2))

    eEllipticFGrad_m(l : List F) : F ==
        z := l(1)
        m := l(2)
        ahalf*((ellipticE(z, m) - (1 - m)*ellipticF(z, m))/m -
          z*sqrt(1 - z^2)/sqrt(1 - m*z^2))/(1 - m)

    derivative(opEllipticF, [eEllipticFGrad_z, eEllipticFGrad_m])

-- handle EllipticPi
```

```
opEllipticPi := operator('ellipticPi)$CommonOperators

ellipticPi(z, n, m) == opEllipticPi(z, n, m)

eEllipticPi(z : F, n : F, m : F) : F ==
    z = 0 => 0
    kernel(opEllipticPi, [z, n, m])

elEllipticPi(l : List F) : F == eEllipticPi(l(1), l(2), l(3))
evaluate(opEllipticPi, elEllipticPi)$BasicOperatorFunctions1(F)

if F has RadicalCategory then

    eEllipticPiGrad_z(l : List F) : F ==
        z := l(1)
        n := l(2)
        m := l(3)
        1/((1 - n*z^2)*sqrt(1 - m*z^2)*sqrt(1 - z^2))

    eEllipticPiGrad_n(l : List F) : F ==
        z := l(1)
        n := l(2)
        m := l(3)
        t1 := -(n^2 - m)*ellipticPi(z, n, m)/((n - 1)*(n - m)*n)
        t2 := ellipticF(z, m)/((n - 1)*n)
        t3 := -ellipticE(z, m)/((n - 1)*(n - m))
        t4 := n*z*sqrt(1 - m*z^2)*sqrt(1 - z^2)/
               ((1 - n*z^2)*(n - 1)*(n - m))
        ahalf*(t1 + t2 + t3 + t4)

    eEllipticPiGrad_m(l : List F) : F ==
        z := l(1)
        n := l(2)
        m := l(3)
        t1 := m*z*sqrt(1 - z^2)/sqrt(1 - m*z^2)
        t2 := (-ellipticE(z, m) + t1)/(1 - m)
        ahalf*(ellipticPi(z, n, m) + t2)/(n - m)

    derivative(opEllipticPi, [eEllipticPiGrad_z, eEllipticPiGrad_n,
                              eEllipticPiGrad_m])

-- handle JacobiSn

opJacobiSn := operator('jacobiSn)$CommonOperators

jacobiSn(z, m) == opJacobiSn(z, m)
```

```
eJacobiSn(z : F, m : F) : F ==
    z = 0 => 0
    if is?(z, opEllipticF) then
        args := argument(retract(z)@K)
        m = args(2) => return args(1)
    kernel(opJacobiSn, [z, m])

elJacobiSn : List F -> F
elJacobiSn(l : List F) : F == eJacobiSn(l(1), l(2))
evaluate(opJacobiSn, elJacobiSn)$BasicOperatorFunctions1(F)

jacobiGradHelper(z : F, m : F) : F ==
    (z - ellipticE(jacobiSn(z, m), m)/(1 - m))/m

eJacobiSnGrad_z(l : List F) : F ==
    z := l(1)
    m := l(2)
    jacobiCn(z, m)*jacobiDn(z, m)

eJacobiSnGrad_m(l : List F) : F ==
    z := l(1)
    m := l(2)
    ahalf*(eJacobiSnGrad_z(l)*jacobiGradHelper(z, m) +
       jacobiSn(z, m)*jacobiCn(z, m)^2/(1 - m))

derivative(opJacobiSn, [eJacobiSnGrad_z, eJacobiSnGrad_m])

-- handle JacobiCn

opJacobiCn := operator('jacobiCn)$CommonOperators

jacobiCn(z, m) == opJacobiCn(z, m)

eJacobiCn(z : F, m : F) : F ==
    z = 0 => 1
    kernel(opJacobiCn, [z, m])

elJacobiCn(l : List F) : F == eJacobiCn(l(1), l(2))
evaluate(opJacobiCn, elJacobiCn)$BasicOperatorFunctions1(F)

eJacobiCnGrad_z(l : List F) : F ==
    z := l(1)
    m := l(2)
    -jacobiSn(z, m)*jacobiDn(z, m)
```

```
eJacobiCnGrad_m(l : List F) : F ==
    z := l(1)
    m := l(2)
    ahalf*(eJacobiCnGrad_z(l)*jacobiGradHelper(z, m) -
       jacobiSn(z, m)^2*jacobiCn(z, m)/(1 - m))

derivative(opJacobiCn, [eJacobiCnGrad_z, eJacobiCnGrad_m])

-- handle JacobiDn

opJacobiDn := operator('jacobiDn)$CommonOperators

jacobiDn(z, m) == opJacobiDn(z, m)

eJacobiDn(z : F, m : F) : F ==
    z = 0 => 1
    kernel(opJacobiDn, [z, m])

elJacobiDn(l : List F) : F == eJacobiDn(l(1), l(2))
evaluate(opJacobiDn, elJacobiDn)$BasicOperatorFunctions1(F)

eJacobiDnGrad_z(l : List F) : F ==
    z := l(1)
    m := l(2)
    -m*jacobiSn(z, m)*jacobiCn(z, m)

eJacobiDnGrad_m(l : List F) : F ==
    z := l(1)
    m := l(2)
    ahalf*(eJacobiDnGrad_z(l)*jacobiGradHelper(z, m) -
       jacobiSn(z, m)^2*jacobiDn(z, m)/(1 - m))

derivative(opJacobiDn, [eJacobiDnGrad_z, eJacobiDnGrad_m])

-- handle JacobiZeta

opJacobiZeta := operator('jacobiZeta)$CommonOperators

jacobiZeta(z, m) == opJacobiZeta(z, m)

eJacobiZeta(z : F, m : F) : F ==
    z = 0 => 0
    kernel(opJacobiZeta, [z, m])

elJacobiZeta(l : List F) : F == eJacobiZeta(l(1), l(2))
evaluate(opJacobiZeta, elJacobiZeta)$BasicOperatorFunctions1(F)
```

```
eJacobiZetaGrad_z(l : List F) : F ==
    z := l(1)
    m := l(2)
    dn := jacobiDn(z, m)
    dn*dn - ellipticE(m)/ellipticK(m)

eJacobiZetaGrad_m(l : List F) : F ==
    z := l(1)
    m := l(2)
    ek := ellipticK(m)
    ee := ellipticE(m)
    er := ee/ek
    dn := jacobiDn(z, m)
    res1 := (dn*dn + m - 1)*jacobiZeta(z, m)
    res2 := res1 + (m - 1)*z*dn*dn
    res3 := res2 - m*jacobiCn(z, m)*jacobiDn(z, m)*jacobiSn(z, m)
    res4 := res3 + z*(1 - m + dn*dn)*er
    ahalf*(res4 - z*er*er)/(m*m - m)

derivative(opJacobiZeta, [eJacobiZetaGrad_z, eJacobiZetaGrad_m])

-- handle JacobiTheta

opJacobiTheta := operator('jacobiTheta)$CommonOperators

jacobiTheta(q, z) == opJacobiTheta(q, z)

eJacobiTheta(q : F, z : F) : F ==
    kernel(opJacobiTheta, [q, z])

elJacobiTheta(l : List F) : F == eJacobiTheta(l(1), l(2))
evaluate(opJacobiTheta, elJacobiTheta)$BasicOperatorFunctions1(F)

-- handle LerchPhi

opLerchPhi := operator('lerchPhi)$CommonOperators

lerchPhi(z, s, a) == opLerchPhi(z, s, a)

eLerchPhi(z : F, s : F, a : F) : F ==
    -- z = 0 => 1/a^s
    a = 1 => polylog(s, z)/z
    kernel(opLerchPhi, [z, s, a])

elLerchPhi(l : List F) : F == eLerchPhi(l(1), l(2), l(3))
```

```
evaluate(opLerchPhi, elLerchPhi)$BasicOperatorFunctions1(F)

dLerchPhi(l : List F, t : SE) : F ==
    z := l(1)
    s := l(2)
    a := l(3)
    dz := differentiate(z, t)*(lerchPhi(z, s - 1, a) -
            a*lerchPhi(z, s, a))/z
    da := -differentiate(a, t)*s*lerchPhi(z, s + 1, a)
    dm := dummy
    differentiate(s, t)*kernel(opdiff, [opLerchPhi [z, dm, a], dm, s])
       + dz + da

setProperty(opLerchPhi, SPECIALDIFF, dLerchPhi@((List F, SE)->F)
                                             pretend None)

-- handle RiemannZeta

opRiemannZeta := operator('riemannZeta)$CommonOperators

riemannZeta(z) == opRiemannZeta(z)

eRiemannZeta(z : F) : F ==
    kernel(opRiemannZeta, [z])

elRiemannZeta(l : List F) : F == eRiemannZeta(l(1))
evaluate(opRiemannZeta, elRiemannZeta)$BasicOperatorFunctions1(F)

-- orthogonal polynomials


charlierC(n : F, a : F, z : F) : F == opCharlierC(n, a, z)

eCharlierC(n : F, a : F, z : F) : F ==
    n = 0 => 1
    n = 1 => (z - a)/a
    kernel(opCharlierC, [n, a, z])

elCharlierC(l : List F) : F == eCharlierC(l(1), l(2), l(3))

evaluate(opCharlierC, elCharlierC)$BasicOperatorFunctions1(F)


hermiteH(n : F, z: F) : F == opHermiteH(n, z)

eHermiteH(n : F, z: F) : F ==
```

```
    n = -1 => 0
    n = 0 => 1
    n = 1 => (2::F)*z
    kernel(opHermiteH, [n, z])

elHermiteH(l : List F) : F == eHermiteH(l(1), l(2))

evaluate(opHermiteH, elHermiteH)$BasicOperatorFunctions1(F)

eHermiteHGrad_z(n : F, z : F) : F == (2::F)*n*hermiteH(n - 1, z)

dHermiteH(l : List F, t : SE) : F ==
    grad2(l, t, opHermiteH, eHermiteHGrad_z)

setProperty(opHermiteH, SPECIALDIFF, dHermiteH@((List F, SE)->F)
                                             pretend None)


jacobiP(n : F, a : F, b : F, z : F) : F == opJacobiP(n, a, b, z)

eJacobiP(n : F, a : F, b : F, z : F) : F ==
    n = -1 => 0
    n = 0 => 1
    n = 1 => ahalf*(a - b) + (1 + ahalf*(a + b))*z
    kernel(opJacobiP, [n, a, b, z])

elJacobiP(l : List F) : F == eJacobiP(l(1), l(2), l(3), l(4))

evaluate(opJacobiP, elJacobiP)$BasicOperatorFunctions1(F)

eJacobiPGrad_z(n : F, a : F, b : F, z : F) : F ==
    ahalf*(a + b + n + 1)*jacobiP(n - 1, a + 1, b + 1, z)

dJacobiP(l : List F, t : SE) : F ==
    grad4(l, t, opJacobiP, eJacobiPGrad_z)

setProperty(opJacobiP, SPECIALDIFF, dJacobiP@((List F, SE)->F)
                                             pretend None)


laguerreL(n : F, a : F, z : F) : F == opLaguerreL(n, a, z)

eLaguerreL(n : F, a : F, z : F) : F ==
    n = -1 => 0
    n = 0 => 1
    n = 1 => (1 + a - z)
```

```
    kernel(opLaguerreL, [n, a, z])

elLaguerreL(l : List F) : F == eLaguerreL(l(1), l(2), l(3))

evaluate(opLaguerreL, elLaguerreL)$BasicOperatorFunctions1(F)

eLaguerreLGrad_z(n : F, a : F, z : F) : F ==
    laguerreL(n - 1, a + 1, z)

dLaguerreL(l : List F, t : SE) : F ==
    grad3(l, t, opLaguerreL, eLaguerreLGrad_z)

setProperty(opLaguerreL, SPECIALDIFF, dLaguerreL@((List F, SE)->F)
                                             pretend None)


meixnerM(n : F, b : F, c : F, z : F) : F == opMeixnerM(n, b, c, z)

eMeixnerM(n : F, b : F, c : F, z : F) : F ==
    n = 0 => 1
    n = 1 => (c - 1)*z/(c*b) + 1
    kernel(opMeixnerM, [n, b, c, z])

elMeixnerM(l : List F) : F == eMeixnerM(l(1), l(2), l(3), l(4))

evaluate(opMeixnerM, elMeixnerM)$BasicOperatorFunctions1(F)


--
belong? op == has?(op, SPECIAL)

operator op ==
  is?(op, 'abs)      => opabs
  is?(op, 'Gamma)    => opGamma
  is?(op, 'Gamma2)   => opGamma2
  is?(op, 'Beta)     => opBeta
  is?(op, 'digamma)  => opdigamma
  is?(op, 'polygamma)=> oppolygamma
  is?(op, 'besselJ)  => opBesselJ
  is?(op, 'besselY)  => opBesselY
  is?(op, 'besselI)  => opBesselI
  is?(op, 'besselK)  => opBesselK
  is?(op, 'airyAi)   => opAiryAi
  is?(op, 'airyAiPrime) => opAiryAiPrime
  is?(op, 'airyBi)   => opAiryBi
  is?(op, 'airyBiPrime) => opAiryBiPrime
```

```
is?(op, 'lambertW) => opLambertW
is?(op, 'polylog) => opPolylog
is?(op, 'weierstrassP)   => opWeierstrassP
is?(op, 'weierstrassPPrime)   => opWeierstrassPPrime
is?(op, 'weierstrassSigma)   => opWeierstrassSigma
is?(op, 'weierstrassZeta)   => opWeierstrassZeta
is?(op, 'hypergeometricF)  => opHypergeometricF
is?(op, 'meijerG)  => opMeijerG
-- is?(op, 'weierstrassPInverse) => opWeierstrassPInverse
is?(op, 'whittakerM) => opWhittakerM
is?(op, 'whittakerW) => opWhittakerW
is?(op, 'angerJ) => opAngerJ
is?(op, 'weberE) => opWeberE
is?(op, 'struveH) => opStruveH
is?(op, 'struveL) => opStruveL
is?(op, 'hankelH1) => opHankelH1
is?(op, 'hankelH2) => opHankelH2
is?(op, 'lommelS1) => opLommelS1
is?(op, 'lommelS2) => opLommelS2
is?(op, 'kummerM) => opKummerM
is?(op, 'kummerU) => opKummerU
is?(op, 'legendreP) => opLegendreP
is?(op, 'legendreQ) => opLegendreQ
is?(op, 'kelvinBei) => opKelvinBei
is?(op, 'kelvinBer) => opKelvinBer
is?(op, 'kelvinKei) => opKelvinKei
is?(op, 'kelvinKer) => opKelvinKer
is?(op, 'ellipticK) => opEllipticK
is?(op, 'ellipticE) => opEllipticE
is?(op, 'ellipticE2) => opEllipticE2
is?(op, 'ellipticF) => opEllipticF
is?(op, 'ellipticPi) => opEllipticPi
is?(op, 'jacobiSn) => opJacobiSn
is?(op, 'jacobiCn) => opJacobiCn
is?(op, 'jacobiDn) => opJacobiDn
is?(op, 'jacobiZeta) => opJacobiZeta
is?(op, 'jacobiTheta) => opJacobiTheta
is?(op, 'lerchPhi) => opLerchPhi
is?(op, 'riemannZeta) => opRiemannZeta
is?(op, 'charlierC) => opCharlierC
is?(op, 'hermiteH) => opHermiteH
is?(op, 'jacobiP) => opJacobiP
is?(op, 'laguerreL) => opLaguerreL
is?(op, 'meixnerM) => opMeixnerM
is?(op, '%logGamma) => op_log_gamma
is?(op, '%eis) => op_eis
```

```
      is?(op, '%erfs) => op_erfs
      is?(op, '%erfis) => op_erfis

      error "Not a special operator"

    -- Could put more unconditional special rules for other functions here
    iGamma x ==
--      one? x => x
      (x = 1) => x
      kernel(opGamma, x)

    iabs x ==
      zero? x => 0
      is?(x, opabs) => x
      smaller?(x, 0) => kernel(opabs, -x)
      kernel(opabs, x)

    iBeta(x, y) == kernel(opBeta, [x, y])
    idigamma x == kernel(opdigamma, x)
    iiipolygamma(n, x) == kernel(oppolygamma, [n, x])
    iiiBesselJ(x, y) == kernel(opBesselJ, [x, y])
    iiiBesselY(x, y) == kernel(opBesselY, [x, y])
    iiiBesselI(x, y) == kernel(opBesselI, [x, y])
    iiiBesselK(x, y) == kernel(opBesselK, [x, y])

    import from Fraction(Integer)

    if F has ElementaryFunctionCategory then
        iAiryAi x ==
            zero?(x) => 1::F/((3::F)^twothirds*Gamma(twothirds))
            kernel(opAiryAi, x)
        iAiryAiPrime x ==
            zero?(x) => -1::F/((3::F)^athird*Gamma(athird))
            kernel(opAiryAiPrime, x)
        iAiryBi x ==
            zero?(x) => 1::F/((3::F)^asixth*Gamma(twothirds))
            kernel(opAiryBi, x)
        iAiryBiPrime x ==
            zero?(x) => (3::F)^asixth/Gamma(athird)
            kernel(opAiryBiPrime, x)
    else
        iAiryAi x == kernel(opAiryAi, x)
        iAiryAiPrime x == kernel(opAiryAiPrime, x)
        iAiryBi x == kernel(opAiryBi, x)
        iAiryBiPrime x == kernel(opAiryBiPrime, x)
```

```
if F has ElementaryFunctionCategory then
    iLambertW(x) ==
        zero?(x) => 0
        x = exp(1$F) => 1$F
        x = -exp(-1$F) => -1$F
        kernel(opLambertW, x)
else
    iLambertW(x) ==
        zero?(x) => 0
        kernel(opLambertW, x)

if F has ElementaryFunctionCategory then
    if F has LiouvillianFunctionCategory then
        iiPolylog(s, x) ==
            s = 1 => -log(1 - x)
            s = 2::F => dilog(1 - x)
            kernel(opPolylog, [s, x])
    else
        iiPolylog(s, x) ==
            s = 1 => -log(1 - x)
            kernel(opPolylog, [s, x])
else
    iiPolylog(s, x) == kernel(opPolylog, [s, x])

iPolylog(l) == iiPolylog(first l, second l)

iWeierstrassP(g2, g3, x) == kernel(opWeierstrassP, [g2, g3, x])
iWeierstrassPPrime(g2, g3, x) == kernel(opWeierstrassPPrime, [g2, g3, x])

iWeierstrassSigma(g2, g3, x) ==
    x = 0 => 0
    kernel(opWeierstrassSigma, [g2, g3, x])

iWeierstrassZeta(g2, g3, x) == kernel(opWeierstrassZeta, [g2, g3, x])

-- Could put more conditional special rules for other functions here

if R has abs : R -> R then
  import from Polynomial R
  iiabs x ==
    (r := retractIfCan(x)@Union(Fraction Polynomial R, "failed"))
      case "failed" => iabs x
    f := r::Fraction Polynomial R
    (a := retractIfCan(numer f)@Union(R, "failed")) case "failed" or
      (b := retractIfCan(denom f)@Union(R,"failed")) case "failed" => iabs x
    abs(a::R)::F / abs(b::R)::F
```

```
else iiabs x == iabs x

if R has SpecialFunctionCategory then
  iiGamma x ==
    (r := retractIfCan(x)@Union(R,"failed")) case "failed" => iGamma x
    Gamma(r::R)::F

  iiBeta l ==
    (r := retractIfCan(first l)@Union(R,"failed")) case "failed" or _
    (s := retractIfCan(second l)@Union(R,"failed")) case "failed" _
        => iBeta(first l, second l)
    Beta(r::R, s::R)::F

  iidigamma x ==
    (r := retractIfCan(x)@Union(R,"failed")) case "failed" => idigamma x
    digamma(r::R)::F

  iipolygamma l ==
    (s := retractIfCan(first l)@Union(R,"failed")) case "failed" or _
    (r := retractIfCan(second l)@Union(R,"failed")) case "failed" _
        => iiipolygamma(first l, second l)
    polygamma(s::R, r::R)::F

  iiBesselJ l ==
    (r := retractIfCan(first l)@Union(R,"failed")) case "failed" or _
    (s := retractIfCan(second l)@Union(R,"failed")) case "failed" _
        => iiiBesselJ(first l, second l)
    besselJ(r::R, s::R)::F

  iiBesselY l ==
    (r := retractIfCan(first l)@Union(R,"failed")) case "failed" or _
    (s := retractIfCan(second l)@Union(R,"failed")) case "failed" _
        => iiiBesselY(first l, second l)
    besselY(r::R, s::R)::F

  iiBesselI l ==
    (r := retractIfCan(first l)@Union(R,"failed")) case "failed" or _
    (s := retractIfCan(second l)@Union(R,"failed")) case "failed" _
        => iiiBesselI(first l, second l)
    besselI(r::R, s::R)::F

  iiBesselK l ==
    (r := retractIfCan(first l)@Union(R,"failed")) case "failed" or _
    (s := retractIfCan(second l)@Union(R,"failed")) case "failed" _
        => iiiBesselK(first l, second l)
```

```
      besselK(r::R, s::R)::F

    iiAiryAi x ==
      (r := retractIfCan(x)@Union(R,"failed")) case "failed" => iAiryAi x
      airyAi(r::R)::F

    iiAiryAiPrime x ==
      (r := retractIfCan(x)@Union(R,"failed")) case "failed" => iAiryAiPrime x
      airyAiPrime(r::R)::F

    iiAiryBi x ==
      (r := retractIfCan(x)@Union(R,"failed")) case "failed" => iAiryBi x
      airyBi(r::R)::F

    iiAiryBi x ==
      (r := retractIfCan(x)@Union(R,"failed")) case "failed" => iAiryBiPrime x
      airyBiPrime(r::R)::F

  else
    if R has RetractableTo Integer then
      iiGamma x ==
        (r := retractIfCan(x)@Union(Integer, "failed")) case Integer
          and (r::Integer >= 1) => factorial(r::Integer - 1)::F
        iGamma x
    else
      iiGamma x == iGamma x

    iiBeta l == iBeta(first l, second l)
    iidigamma x == idigamma x
    iipolygamma l == iiipolygamma(first l, second l)
    iiBesselJ l == iiiBesselJ(first l, second l)
    iiBesselY l == iiiBesselY(first l, second l)
    iiBesselI l == iiiBesselI(first l, second l)
    iiBesselK l == iiiBesselK(first l, second l)
    iiAiryAi x == iAiryAi x
    iiAiryAiPrime x == iAiryAiPrime x
    iiAiryBi x == iAiryBi x
    iiAiryBiPrime x == iAiryBiPrime x

  iiWeierstrassP l == iWeierstrassP(first l, second l, third l)
  iiWeierstrassPPrime l == iWeierstrassPPrime(first l, second l, third l)
  iiWeierstrassSigma l == iWeierstrassSigma(first l, second l, third l)
  iiWeierstrassZeta l == iWeierstrassZeta(first l, second l, third l)

  -- Default behaviour is to build a kernel
  evaluate(opGamma, iiGamma)$BasicOperatorFunctions1(F)
```

```
    evaluate(opabs, iiabs)$BasicOperatorFunctions1(F)
--    evaluate(opGamma2    , iiGamma2   )$BasicOperatorFunctions1(F)
    evaluate(opBeta      , iiBeta     )$BasicOperatorFunctions1(F)
    evaluate(opdigamma   , iidigamma  )$BasicOperatorFunctions1(F)
    evaluate(oppolygamma , iipolygamma)$BasicOperatorFunctions1(F)
    evaluate(opBesselJ   , iiBesselJ  )$BasicOperatorFunctions1(F)
    evaluate(opBesselY   , iiBesselY  )$BasicOperatorFunctions1(F)
    evaluate(opBesselI   , iiBesselI  )$BasicOperatorFunctions1(F)
    evaluate(opBesselK   , iiBesselK  )$BasicOperatorFunctions1(F)
    evaluate(opAiryAi    , iiAiryAi   )$BasicOperatorFunctions1(F)
    evaluate(opAiryAiPrime, iiAiryAiPrime)$BasicOperatorFunctions1(F)
    evaluate(opAiryBi    , iiAiryBi   )$BasicOperatorFunctions1(F)
    evaluate(opAiryBiPrime, iiAiryBiPrime)$BasicOperatorFunctions1(F)
    evaluate(opLambertW, iLambertW)$BasicOperatorFunctions1(F)
    evaluate(opPolylog, iPolylog)$BasicOperatorFunctions1(F)
    evaluate(opWeierstrassP, iiWeierstrassP)$BasicOperatorFunctions1(F)
    evaluate(opWeierstrassPPrime,
             iiWeierstrassPPrime)$BasicOperatorFunctions1(F)
    evaluate(opWeierstrassSigma, iiWeierstrassSigma)$BasicOperatorFunctions1(F)
    evaluate(opWeierstrassZeta, iiWeierstrassZeta)$BasicOperatorFunctions1(F)
    evaluate(opHypergeometricF, iiHypergeometricF)$BasicOperatorFunctions1(F)
    evaluate(opMeijerG, iiMeijerG)$BasicOperatorFunctions1(F)

    diff1(op : OP, n : F, x : F) : F ==
        dm := dummy
        kernel(opdiff, [op [dm, x], dm, n])

    iBesselJ(l : List F, t : SE) : F ==
        n := first l; x := second l
        differentiate(n, t)*diff1(opBesselJ, n, x)
          + differentiate(x, t) * ahalf * (besselJ (n-1, x) - besselJ (n+1, x))

    iBesselY(l : List F, t : SE) : F ==
        n := first l; x := second l
        differentiate(n, t)*diff1(opBesselY, n, x)
          + differentiate(x, t) * ahalf * (besselY (n-1, x) - besselY (n+1, x))

    iBesselI(l : List F, t : SE) : F ==
        n := first l; x := second l
        differentiate(n, t)*diff1(opBesselI, n, x)
          + differentiate(x, t)* ahalf * (besselI (n-1, x) + besselI (n+1, x))

    iBesselK(l : List F, t : SE) : F ==
        n := first l; x := second l
        differentiate(n, t)*diff1(opBesselK, n, x)
          - differentiate(x, t)* ahalf * (besselK (n-1, x) + besselK (n+1, x))
```

```
dPolylog(l : List F, t : SE) : F ==
    s := first l; x := second l
    differentiate(s, t)*diff1(opPolylog, s, x)
       + differentiate(x, t)*polylog(s-1, x)/x

ipolygamma(l : List F, x : SE) : F ==
    import from List(Symbol)
    member?(x, variables first l) =>
        error "cannot differentiate polygamma with respect to the first argument"
    n := first l; y := second l
    differentiate(y, x)*polygamma(n+1, y)
iBetaGrad1(l : List F) : F ==
    x := first l; y := second l
    Beta(x, y)*(digamma x - digamma(x+y))
iBetaGrad2(l : List F) : F ==
    x := first l; y := second l
    Beta(x, y)*(digamma y - digamma(x+y))

if F has ElementaryFunctionCategory then
  iGamma2(l : List F, t : SE) : F ==
    a := first l; x := second l
    differentiate(a, t)*diff1(opGamma2, a, x)
      - differentiate(x, t)* x ^ (a - 1) * exp(-x)
  setProperty(opGamma2, SPECIALDIFF, iGamma2@((List F, SE)->F)
                                             pretend None)

inGamma2(li : List INP) : INP ==
    convert cons(convert('Gamma), li)

input(opGamma2, inGamma2@((List INP) -> INP))

dLambertW(x : F) : F ==
    lw := lambertW(x)
    lw/(x*(1+lw))

iWeierstrassPGrad1(l : List F) : F ==
    g2 := first l
    g3 := second l
    x := third l
    delta := g2^3 - 27*g3^2
    wp := weierstrassP(g2, g3, x)
    (weierstrassPPrime(g2, g3, x)*(-9*ahalf*g3
      *weierstrassZeta(g2, g3, x) + afourth*g2^2*x)
        - 9*g3*wp^2 + ahalf*g2^2*wp + 3*ahalf*g2*g3)/delta
```

```
iWeierstrassPGrad2(l : List F) : F ==
    g2 := first l
    g3 := second l
    x := third l
    delta := g2^3 - 27*g3^2
    wp := weierstrassP(g2, g3, x)
    (weierstrassPPrime(g2, g3, x)*(3*g2*weierstrassZeta(g2, g3, x)
      - 9*ahalf*g3*x) + 6*g2*wp^2 - 9*g3*wp-g2^2)/delta

iWeierstrassPGrad3(l : List F) : F ==
    weierstrassPPrime(first l, second l, third l)

iWeierstrassPPrimeGrad1(l : List F) : F ==
    g2 := first l
    g3 := second l
    x := third l
    delta := g2^3 - 27*g3^2
    wp := weierstrassP(g2, g3, x)
    wpp := weierstrassPPrime(g2, g3, x)
    wpp2 := 6*wp^2 - ahalf*g2
    (wpp2*(-9*ahalf*g3*weierstrassZeta(g2, g3, x) + afourth*g2^2*x)
       + wpp*(9*ahalf*g3*wp + afourth*g2^2) - 18*g3*wp*wpp
         + ahalf*g2^2*wpp)/delta

iWeierstrassPPrimeGrad2(l : List F) : F ==
    g2 := first l
    g3 := second l
    x := third l
    delta := g2^3 - 27*g3^2
    wp := weierstrassP(g2, g3, x)
    wpp := weierstrassPPrime(g2, g3, x)
    wpp2 := 6*wp^2 - ahalf*g2
    (wpp2*(3*g2*weierstrassZeta(g2, g3, x) - 9*ahalf*g3*x)
      + wpp*(-3*g2*wp - 9*ahalf*g3) + 12*g2*wp*wpp - 9*g3*wpp)/delta


iWeierstrassPPrimeGrad3(l : List F) : F ==
    g2 := first l
    6*weierstrassP(g2, second l, third l)^2 - ahalf*g2

iWeierstrassSigmaGrad1(l : List F) : F ==
    g2 := first l
    g3 := second l
    x := third l
    delta := g2^3 - 27*g3^2
    ws := weierstrassSigma(g2, g3, x)
```

```
    wz := weierstrassZeta(g2, g3, x)
    wsp := wz*ws
    wsp2 := - weierstrassP(g2, g3, x)*ws + wz^2*ws
    afourth*(-9*g3*wsp2 - g2^2*ws
             - 3*afourth*g2*g3*x^2*ws + g2^2*x*wsp)/delta

iWeierstrassSigmaGrad2(l : List F) : F ==
    g2 := first l
    g3 := second l
    x := third l
    delta := g2^3 - 27*g3^2
    ws := weierstrassSigma(g2, g3, x)
    wz := weierstrassZeta(g2, g3, x)
    wsp := wz*ws
    wsp2 := - weierstrassP(g2, g3, x)*ws + wz^2*ws
    ahalf*(3*g2*wsp2 + 9*g3*ws
            + afourth*g2^2*x^2*ws - 9*g3*x*wsp)/delta

iWeierstrassSigmaGrad3(l : List F) : F ==
    g2 := first l
    g3 := second l
    x := third l
    weierstrassZeta(g2, g3, x)*weierstrassSigma(g2, g3, x)

iWeierstrassZetaGrad1(l : List F) : F ==
    g2 := first l
    g3 := second l
    x := third l
    delta := g2^3 - 27*g3^2
    wp := weierstrassP(g2, g3, x)
    (ahalf*weierstrassZeta(g2, g3, x)*(9*g3*wp + ahalf*g2^2)
         - ahalf*g2*x*(ahalf*g2*wp+3*afourth*g3)
           + 9*afourth*g3*weierstrassPPrime(g2, g3, x))/delta

iWeierstrassZetaGrad2(l : List F) : F ==
    g2 := first l
    g3 := second l
    x := third l
    delta := g2^3 - 27*g3^2
    wp := weierstrassP(g2, g3, x)
    (-3*weierstrassZeta(g2, g3, x)*(g2*wp + 3*ahalf*g3) +
                 ahalf*x*(9*g3*wp + ahalf*g2^2)
                   - 3*ahalf*g2*weierstrassPPrime(g2, g3, x))/delta


iWeierstrassZetaGrad3(l : List F) : F ==
```

```
    -weierstrassP(first l, second l, third l)


OF ==> OutputForm
SEX ==> SExpression
NNI ==> NonNegativeInteger

if F has RetractableTo(Integer) then
    get_int_listf : List F -> List Integer
    get_int_listo : (Integer, List OF) ->  List Integer
    get_int_listi : (Integer, List INP) ->  List Integer

    get_int_listf(lf : List F) : List Integer ==
        map(z +-> retract(z)@Integer, lf)$ListFunctions2(F, Integer)

    replace_i(lp : List F, v : F, i : NNI) : List F ==
        concat(first(lp, (i - 1)::NNI), cons(v, rest(lp, i)))

    iiHypergeometricF(l) ==
        n := #l
        z := l(n-2)
        if z = 0 then
            nn := (n - 2)::NNI
            pq := rest(l, nn)
            pqi := get_int_listf(pq)
            p := first(pqi)
            q := first(rest(pqi))
            p <= q + 1 => return 1
        kernel(opHypergeometricF, l)

    idvsum(op : BasicOperator, n : Integer, l : List F, x : Symbol) : F  ==
        res : F := 0
        for i in 1..n for a in l repeat
            dm := dummy
            nl := replace_i(l, dm, i)
            res := res + differentiate(a, x)*kernel(opdiff, [op nl, dm, a])
        res

    dvhypergeom(l : List F, x : Symbol) : F ==
        n := #l
        nn := (n - 2)::NNI
        pq := rest(l, nn)
        pqi := get_int_listf(pq)
        ol := l
        l := first(l, nn)
        l1 := reverse(l)
```

```
    z := first(l1)
    p := first(pqi)
    q := first(rest(pqi))
    aprod := 1@F
    nl := []@(List F)
    for i in 1..p repeat
        a := first(l)
        nl := cons(a + 1, nl)
        aprod := aprod * a
        l := rest(l)
    bprod := 1@F
    for i in 1..q repeat
        b := first(l)
        nl := cons(b + 1, nl)
        bprod := bprod * b
        l := rest(l)
    nl0 := reverse!(nl)
    nl1 := cons(z, pq)
    nl := concat(nl0, nl1)
    aprod := aprod/bprod
    idvsum(opHypergeometricF, nn - 1, ol, x) +
         differentiate(z, x)*aprod*opHypergeometricF(nl)

add_pairs_to_list(lp : List List F, l : List F) : List F ==
    for p in lp repeat
        #p ~= 2 => error "not a list of pairs"
        l := cons(p(2), cons(p(1), l))
    l

dvmeijer(l : List F, x : Symbol) : F ==
    n := #l
    nn := (n - 4)::NNI
    l0 := l
    nl := rest(l, nn)
    nli := get_int_listf(nl)
    l := first(l, nn)
    l1 := reverse(l)
    z := first(l1)
    n1 := first(nli)
    n2 := nli(2)
    a := first l
    sign : F := 1
    if n1 > 0 or n2 > 0 then
         na := a - 1
         if n1 = 0 then sign := -1
         l2 := cons(na, rest l)
```

```
    else
         na := a
         if nli(3) > 0 then sign := -1
         l2 := cons(a + 1, rest l)
    nm : F := opMeijerG(concat(l2, nl))
    om : F := opMeijerG(l0)
    idvsum(opMeijerG, nn - 1, l0, x) +
        differentiate(z, x)*(sign*nm + na*om)/z

get_if_list(n : Integer, lf : List INP) : List List INP ==
    a := []@(List INP)
    for i in 1..n repeat
        a := cons(first(lf), a)
        lf := rest(lf)
    a := cons(convert('construct), reverse!(a))
    [a, lf]

get_if_lists(ln : List Integer, lf : List INP) : List List INP ==
    rl := []@(List List INP)
    for n in ln repeat
        al := get_if_list(n, lf)
        rl := cons(first(al), rl)
        lf := first(rest(al))
    rl := reverse!(rl)
    cons(lf, rl)

get_int_listi(n : Integer, lo : List INP) : List Integer ==
    n0 := (#lo - n)::NNI
    lo := rest(lo, n0)
    rl := []@(List Integer)
    for i in 1..n repeat
        p := integer(first(lo) pretend SEX)$SEX
        rl := cons(p, rl)
        lo := rest(lo)
    rl := reverse!(rl)
    rl

get_of_list(n : Integer, lo : List OF) : List List OF ==
    a := []@(List OF)
    for i in 1..n repeat
        a := cons(first(lo), a)
        lo := rest(lo)
    a := reverse!(a)
    [a, lo]

get_of_lists(ln : List Integer, lo : List OF) : List List OF ==
```

```
    rl := []@(List List OF)
    for n in ln repeat
        al := get_of_list(n, lo)
        rl := cons(first(al), rl)
        lo := first(rest(al))
    rl := reverse!(rl)
    cons(lo, rl)

get_int_listo(n : Integer, lo : List OF) : List Integer ==
    n0 := (#lo - n)::NNI
    lo := rest(lo, n0)
    rl := []@(List Integer)
    for i in 1..n repeat
        p := integer(first(lo) pretend SEX)$SEX
        rl := cons(p, rl)
        lo := rest(lo)
    rl := reverse!(rl)
    rl

dhyper0(op : OF, lo : List OF) : OF ==
    n0 := (#lo - 2)::NNI
    pql := get_int_listo(2, lo)
    lo := first(lo, n0)
    al := get_of_lists(pql, lo)
    lo := first(al)
    al := rest(al)
    a := first al
    b := first(rest(al))
    z := first(lo)
    prefix(op, [bracket a, bracket b, z])

dhyper(lo : List OF) : OF ==
    dhyper0("hypergeometricF"::Symbol::OF, lo)

ddhyper(lo : List OF) : OF ==
    dhyper0(first lo, rest lo)

dmeijer0(op : OF, lo : List OF) : OF ==
    n0 := (#lo - 4)::NNI
    nl := get_int_listo(4, lo)
    lo := first(lo, n0)
    al := get_of_lists(nl, lo)
    lo := first(al)
    al := rest(al)
    z := first(lo)
    prefix(op, concat(
```

```
                 map(bracket, al)$ListFunctions2(List OF, OF), [z]))

dmeijer(lo : List OF) : OF ==
    dmeijer0('meijerG::OF, lo)

ddmeijer(lo : List OF) : OF ==
    dmeijer0(first lo, rest lo)

setProperty(opHypergeometricF, '%diffDisp,
             ddhyper@(List OF -> OF) pretend None)
setProperty(opMeijerG, '%diffDisp,
             ddmeijer@(List OF -> OF) pretend None)
display(opHypergeometricF, dhyper)
display(opMeijerG, dmeijer)
setProperty(opHypergeometricF, SPECIALDIFF,
         dvhypergeom@((List F, Symbol)->F) pretend None)
setProperty(opMeijerG, SPECIALDIFF, dvmeijer@((List F, Symbol)->F)
                                    pretend None)

inhyper(lf : List INP) : INP ==
    pqi := get_int_listi(2, lf)
    al := get_if_lists(pqi, lf)
    lf := first(al)
    al := rest(al)
    a := first al
    ai : INP := convert(a)
    b := first(rest(al))
    bi : INP := convert(b)
    zi := first(lf)
    li : List INP := [convert('hypergeometricF), ai, bi, zi]
    convert(li)

input(opHypergeometricF, inhyper@((List INP) -> INP))

inmeijer(lf : List INP) : INP ==
    pqi := get_int_listi(4, lf)
    al := get_if_lists(pqi, lf)
    lf := first(al)
    al := rest(al)
    a := first al
    ai : INP := convert(a)
    al := rest(al)
    b := first(al)
    bi : INP := convert(b)
    al := rest(al)
    c := first(al)
```

```
        ci : INP := convert(c)
        al := rest(al)
        d := first(al)
        di : INP := convert(d)
        zi := first(lf)
        li : List INP := [convert('meijerG), ai, bi, ci, di, zi]
        convert(li)

    input(opMeijerG, inmeijer@((List INP) -> INP))

else
    iiHypergeometricF(l) == kernel(opHypergeometricF, l)

iiMeijerG(l) == kernel(opMeijerG, l)

d_eis(x : F) : F == -kernel(op_eis, x) + 1/x

if F has TranscendentalFunctionCategory
   and F has RadicalCategory then

    d_erfs(x : F) : F == 2*x*kernel(op_erfs, x) - 2::F/sqrt(pi())

    d_erfis(x : F) : F == -2*x*kernel(op_erfis, x) + 2::F/sqrt(pi())

    derivative(op_erfs,     d_erfs)
    derivative(op_erfis,    d_erfis)

derivative(opabs,       (x : F) : F +-> abs(x)*inv(x))
derivative(opGamma,     (x : F) : F +-> digamma(x)*Gamma(x))
derivative(op_log_gamma, (x : F) : F +-> digamma(x))
derivative(opBeta,      [iBetaGrad1, iBetaGrad2])
derivative(opdigamma,   (x : F) : F +-> polygamma(1, x))
derivative(op_eis,      d_eis)
derivative(opAiryAi,    (x : F) : F +-> airyAiPrime(x))
derivative(opAiryAiPrime,    (x : F) : F +-> x*airyAi(x))
derivative(opAiryBi,    (x : F) : F +-> airyBiPrime(x))
derivative(opAiryBiPrime,    (x : F) : F +-> x*airyBi(x))
derivative(opLambertW,  dLambertW)
derivative(opWeierstrassP, [iWeierstrassPGrad1, iWeierstrassPGrad2,
                            iWeierstrassPGrad3])
derivative(opWeierstrassPPrime, [iWeierstrassPPrimeGrad1,
               iWeierstrassPPrimeGrad2, iWeierstrassPPrimeGrad3])
derivative(opWeierstrassSigma, [iWeierstrassSigmaGrad1,
               iWeierstrassSigmaGrad2, iWeierstrassSigmaGrad3])
derivative(opWeierstrassZeta, [iWeierstrassZetaGrad1,
               iWeierstrassZetaGrad2, iWeierstrassZetaGrad3])
```

```
setProperty(oppolygamma, SPECIALDIFF, ipolygamma@((List F, SE)->F)
                                                        pretend None)
setProperty(opBesselJ, SPECIALDIFF, iBesselJ@((List F, SE)->F)
                                                    pretend None)
setProperty(opBesselY, SPECIALDIFF, iBesselY@((List F, SE)->F)
                                                    pretend None)
setProperty(opBesselI, SPECIALDIFF, iBesselI@((List F, SE)->F)
                                                    pretend None)
setProperty(opBesselK, SPECIALDIFF, iBesselK@((List F, SE)->F)
                                                    pretend None)
setProperty(opPolylog, SPECIALDIFF, dPolylog@((List F, SE)->F)
                                                    pretend None)
```

3 package SUMFS FunctionSpaceSum

⟨*package SUMFS FunctionSpaceSum*⟩≡

```
)abbrev package SUMFS FunctionSpaceSum
++ Top-level sum function
++ Author: Manuel Bronstein
++ Date Created: ???
++ Date Last Updated: 19 April 1991
++ Description: computes sums of top-level expressions;
FunctionSpaceSum(R, F) : Exports == Implementation where
  R : Join(IntegralDomain, Comparable,
          RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(FunctionSpace R, CombinatorialOpsCategory,
          AlgebraicallyClosedField, TranscendentalFunctionCategory)

  SE  ==> Symbol
  K   ==> Kernel F

  Exports ==> with
    sum : (F, SE) -> F
      ++ sum(a(n), n) returns A(n) such that A(n+1) - A(n) = a(n);
    sum : (F, SegmentBinding F) -> F
      ++ sum(f(n), n = a..b) returns f(a) + f(a+1) + ... + f(b);

  Implementation ==> add
    import from ElementaryFunctionStructurePackage(R, F)
    import from GosperSummationMethod(IndexedExponents K, K, R,
                              SparseMultivariatePolynomial(R, K), F)

    innersum : (F, K) -> Union(F, "failed")
    notRF?  : (F, K) -> Boolean
    newk    : () -> K

    newk() == kernel(new()$SE)

    sum(x : F, s : SegmentBinding F) ==
      k := kernel(variable s)@K
      (u := innersum(x, k)) case "failed" => summation(x, s)
      eval(u::F, k, 1 + hi segment s) - eval(u::F, k, lo segment s)

    sum(x : F, v : SE) ==
      (u := innersum(x, kernel(v)@K)) case "failed" => summation(x,v)
      u::F

    notRF?(f, k) ==
      for kk in tower f repeat
```

```
      member?(k, tower(kk::F)) and (symbolIfCan(kk) case "failed") =>
        return true
    false

  innersum(x, k) ==
    zero? x => 0
    notRF?(f := normalize(x / (x1 := eval(x, k, k::F - 1))), k) =>
      "failed"
    (u := GospersMethod(f, k, newk)) case "failed" => "failed"
    x1 * eval(u::F, k, k::F - 1)
```

4 License

⟨*license*⟩≡

⟨*⟩≡
⟨*license*⟩

```
-- SPAD files for the functional world should be compiled in the
-- following order:
--
--   op  kl  function  funcpkgs  manip  algfunc
--   elemntry  constant  funceval  COMBFUNC  fe
```

⟨*category COMBOPC CombinatorialOpsCategory*⟩
⟨*package COMBF CombinatorialFunction*⟩
⟨*package FSPECF FunctionalSpecialFunction*⟩
⟨*package SUMFS FunctionSpaceSum*⟩

References

[1] nothing