Integration Utility Functions

In addition to the utility functions defined here, implementing a general purpose integrator based on the rules provided on this website requires defining system dependent functions for simplifying and expanding mathematical expressions.

Miscellaneous Functions

```
(* Note: Clear[func] also eliminates 2-D display of functions like Integrate. *)
ClearDownValues[func_Symbol] := (
  Unprotect[func];
  DownValues[func]={};
  Protect[func])

SetDownValues[func_Symbol,lst_List] := (
  Unprotect[func];
  DownValues[func]=Take[lst,Min[529,Length[lst]]];
  Scan[Function[ReplacePart[ReplacePart[#,#[[1,1]],1],SetDelayed,0]],Drop[lst,Min[529,Length[lst]]]];
  Protect[func])

(* MoveDownValues[func1,func2] moves func1's DownValues to func2, and deletes them from func1. *)
MoveDownValues[func1_Symbol,func2_Symbol] := Module[{lst},
  SetDownValues[func2,ReplaceAll[DownValues[func1],{func1->func2}]];
  ClearDownValues[func1]]

Map2[func_,lst1_,lst2_] :=
  ReapList[Do[Sow[func[lst1[[i]],lst2[[i]]]],{i,Length[lst1]}]]

ReapList[u_] :=
  Module[{lst=Reap[u][[2]]},
  If[lst==={}, lst, lst[[1]]]]

SetAttributes[ReapList,HoldFirst]

(* MapAnd[f,l] applies f to the elements of list l until False is returned; else returns True *)
MapAnd[f_,lst_] :=
  Catch[Scan[Function[If[f[#],Null,Throw[False]]],lst];True]

MapAnd[f_,lst_,x_] :=
  Catch[Scan[Function[If[f[#,x],Null,Throw[False]]],lst];True]

(* MapOr[f,l] applies f to the elements of list l until True is return; else returns False *)
MapOr[f_,lst_] :=
  Catch[Scan[Function[If[f[#],Throw[True],Null]],lst];False]

(* If u is a sum, MapSum[f,u,x] applies f to the terms of u; else it applies f to u. *)
(* MapSum[f_,u_,x_Symbol] :=
  If[SumQ[u],
    Map[Function[f[#,x]],u],
  f[u,x]] *)
```

Recognizer Functions

```
(* NotIntegrableQ[u,x] returns True if u is definitely not integrable wrt x; else it returns
    False if u is, or might be, integrable wrt x. *)
NotIntegrableQ[u_,x_Symbol] :=
  MatchQ[u,x^m_*Log[a_+b_.*x]^n_ /; FreeQ[{a,b},x] && IntegersQ[m,n] && m<0 && n<0] ||
  MatchQ[u,f_[x^m_.*Log[a_.+b_.*x]] /; FreeQ[{a,b},x] && IntegerQ[m] && (TrigQ[f] || HyperbolicQ[f])]
```

Number Domains

```
(* ZeroQ[u1,u2,...] returns True if u1, u2, ... are all 0; else returns False *)
ZeroQ[u_] := PossibleZeroQ[u]
NonzeroQ[u_] := Not[PossibleZeroQ[u]]

ZeroQ[u__] := Catch[Scan[Function[If[ZeroQ[#],Null,Throw[False]]],{u}];True]

(* OneQ[u1,u2,...] returns True if u1, u2, ... are all 1; else returns False *)
OneQ[u_] := PossibleZeroQ[u-1]

OneQ[u__] := Catch[Scan[Function[If[OneQ[#],Null,Throw[False]]],{u}];True]

(* RealNumericQ[u] returns True if u is a real numeric quantity, else returns False. *)
RealNumericQ[u_] := NumericQ[u] && PossibleZeroQ[Im[N[u]]]

(* ImaginaryNumericQ[u] returns True if u is an imaginary numeric quantity, else returns False. *)
ImaginaryNumericQ[u_] :=
  NumericQ[u] && PossibleZeroQ[Re[N[u]]] && Not[PossibleZeroQ[Im[N[u]]]]

(* PositiveQ[u] returns True if u is a positive numeric quantity, else returns False. *)
PositiveQ[u_] :=
  Module[{v=Simplify[u]},
  RealNumericQ[v] && Re[N[v]]>0]

(* PositiveOrZeroQ[u] returns True if u is a nonpositive numeric quantity, else returns False. *)
PositiveOrZeroQ[u_] :=
  Module[{v=Simplify[u]},
  RealNumericQ[v] && Re[N[v]]>=0]

(* NegativeQ[u] returns True if u is a negative numeric quantity, else returns False. *)
NegativeQ[u_] :=
  Module[{v=Simplify[u]},
  RealNumericQ[v] && Re[N[v]]<0]

(* NegativeQ[u] returns True if u is a negative numeric quantity, else returns False. *)
NegativeOrZeroQ[u_] :=
  Module[{v=Simplify[u]},
  RealNumericQ[v] && Re[N[v]]<=0]
```

Number Types

```
(* IntegerQ[u] returns True if u is an explicit integer; else returns False. *)

(* IntegersQ[u_+m_*(n_+v_)] := RationalQ[m] && RationalQ[n] && IntegerQ[m*n] && IntegerQ[u+m*v]; *)
IntegersQ[u__] := Catch[Scan[Function[If[IntegerQ[#],Null,Throw[False]]],{u}];True];

(* FractionQ[u] returns True if u is an explicit fraction; else returns False. *)
FractionQ[u_] :=
  If[ListQ[u],
    MapAnd[FractionQ,u],
  Head[u]===Rational]

FractionQ[u__] := Catch[Scan[Function[If[FractionQ[#],Null,Throw[False]]],{u}];True]
```

```
(* RationalQ[u] returns True if u is an explicit integers or fractions; else returns False. *)
RationalQ[u_+m_*(n_+v_)] :=
  RationalQ[m] && RationalQ[n] && RationalQ[u+m*v]

RationalQ[u_] :=
  If[ListQ[u],
    u==={} || (RationalQ[First[u]] && RationalQ[Rest[u]]),
  IntegerQ[u] || Head[u]===Rational]

RationalQ[u__] := Catch[Scan[Function[If[RationalQ[#],Null,Throw[False]]],{u}];True]

(* Delete this!!! *)

(* HalfIntegerQ[u] returns True if u is a fraction with a denominator of 2; else returns False *)
HalfIntegerQ[u_] :=
  If[ListQ[u],
    MapAnd[HalfIntegerQ,u],
  FractionQ[u] && Denominator[u]==2]

(* FractionOrNegativeQ[u] returns True if u is a fraction or negative number; else returns False *)
FractionOrNegativeQ[u_] :=
  If[ListQ[u],
    MapAnd[FractionOrNegativeQ,u],
  FractionQ[u] || IntegerQ[u] && u<0]

FractionOrNegativeQ[u__] := Catch[Scan[Function[If[FractionOrNegativeQ[#],Null,Throw[False]]],{u}];Tru

(* SqrtNumberQ[u] returns True if u^2 is a rational number; else it returns False. *)
SqrtNumberQ[m_^n_] :=
  IntegerQ[n] && SqrtNumberQ[m] || HalfIntegerQ[n] && RationalQ[m]

SqrtNumberQ[u_*v_] :=
  SqrtNumberQ[u] && SqrtNumberQ[v]

SqrtNumberQ[u_] :=
  RationalQ[u] || u===I

SqrtNumberSumQ[u_] :=
  SumQ[u] && SqrtNumberQ[First[u]] && SqrtNumberQ[Rest[u]]

(* AbsurdNumberQ[u] returns True if u is a real-valued absurd number (a rational number, a
   positive rational number raised to a fractional power, or a product of absurd numbers);
   else returns False. *)
(* AbsurdNumberQ[u_^v_] :=
  RationalQ[u] && u>0 && FractionQ[v]

AbsurdNumberQ[u_*v_] :=
  AbsurdNumberQ[u] && AbsurdNumberQ[v]

AbsurdNumberQ[u_] :=
  RationalQ[u] *)
```

```
(* AlgebraicNumberQ[u] returns True if u is a real-valued algebraic number (a rational number,
   an algebraic number raised to an integer power, a positive algebraic number raised to a
   fractional power, or a product or sum of algebraic numbers); else returns False. *)
(* AlgebraicNumberQ[u_] :=
  MapAnd[AlgebraicNumberQ,u] /;
ListQ[u]

AlgebraicNumberQ[u_^v_] :=
  AlgebraicNumberQ[u] && (IntegerQ[v] || PositiveQ[u] && FractionQ[v])

AlgebraicNumberQ[u_*v_] :=
  AlgebraicNumberQ[u] && AlgebraicNumberQ[v]

AlgebraicNumberQ[u_+v_] :=
  AlgebraicNumberQ[u] && AlgebraicNumberQ[v]

AlgebraicNumberQ[u_] :=
  RationalQ[u] *)
```

Expression Types

```
FalseQ[u_] :=
  u===False

NotFalseQ[u_] :=
  u=!=False

SumQ[u_] :=
  Head[u]===Plus

NonsumQ[u_] :=
  Head[u]=!=Plus

ProductQ[u_] :=
  Head[u]===Times

PowerQ[u_] :=
  Head[u]===Power

IntegerPowerQ[u_] :=
  PowerQ[u] && IntegerQ[u[[2]]]

PositiveIntegerPowerQ[u_] :=
  PowerQ[u] && IntegerQ[u[[2]]] && u[[2]]>0

FractionalPowerQ[u_] :=
  PowerQ[u] && FractionQ[u[[2]]]

RationalPowerQ[u_] :=
  PowerQ[u] && RationalQ[u[[2]]]

SqrtQ[u_] :=
  PowerQ[u] && u[[2]]===1/2

ExpQ[u_] :=
  PowerQ[u] && u[[1]]===E

ImaginaryQ[u_] :=\
  Head[u]===Complex && Re[u]===0
```

```
FractionalPowerFreeQ[u_] :=
  If[AtomQ[u],
    True,
  If[FractionalPowerQ[u] && Not[AtomQ[u[[1]]]],
    False,
  Catch[Scan[Function[If[FractionalPowerFreeQ[#],Null,Throw[False]]],u];True]]]

ComplexFreeQ[u_] :=
  If[AtomQ[u],
    Head[u]=!=Complex,
  Catch[Scan[Function[If[ComplexFreeQ[#],Null,Throw[False]]],u];True]]

LogQ[u_] :=
  Head[u]===Log

SinQ[u_] :=
  Head[u]===Sin

CosQ[u_] :=
  Head[u]===Cos

TanQ[u_] :=
  Head[u]===Tan

CotQ[u_] :=
  Head[u]===Cot

SecQ[u_] :=
  Head[u]===Sec

CscQ[u_] :=
  Head[u]===Csc

SinhQ[u_] :=
  Head[u]===Sinh

CoshQ[u_] :=
  Head[u]===Cosh

TanhQ[u_] :=
  Head[u]===Tanh

CothQ[u_] :=
  Head[u]===Coth

SechQ[u_] :=
  Head[u]===Sech

CschQ[u_] :=
  Head[u]===Csch
```

```
(* TrigQ[u] returns True if u or the head of u is a trig function; else returns False *)
TrigQ[u_] :=
  MemberQ[{Sin,Cos,Tan,Cot,Sec,Csc},If[AtomQ[u],u,Head[u]]]

(* InverseTrigQ[u] returns True if u or the head of u is an inverse trig function; else returns False
InverseTrigQ[u_] :=
  MemberQ[{ArcSin,ArcCos,ArcTan,ArcCot,ArcSec,ArcCsc},If[AtomQ[u],u,Head[u]]]

(* HyperbolicQ[u] returns True if u or the head of u is a trig function; else returns False *)
HyperbolicQ[u_] :=
  MemberQ[{Sinh,Cosh,Tanh,Coth,Sech,Csch},If[AtomQ[u],u,Head[u]]]

(* InverseHyperbolicQ[u] returns True if u or the head of u is an inverse trig function; else returns
InverseHyperbolicQ[u_] :=
  MemberQ[{ArcSinh,ArcCosh,ArcTanh,ArcCoth,ArcSech,ArcCsch},If[AtomQ[u],u,Head[u]]]

SinCosQ[f_] :=
  MemberQ[{Sin,Cos,Sec,Csc},f]

SinhCoshQ[f_] :=
  MemberQ[{Sinh,Cosh,Sech,Csch},f]

CalculusFunctions={D,Integrate,Sum,Product,Int,Dif,Subst};

(* CalculusQ[u] returns True if the head of u is a calculus function; else returns False *)
CalculusQ[u_] :=
  MemberQ[CalculusFunctions,Head[u]]

CalculusFreeQ[u_,x_] :=
  If[AtomQ[u],
    True,
  If[CalculusQ[u] && u[[2]]===x || Head[u]===Pattern || Head[u]===Defer,
    False,
  Catch[Scan[Function[If[CalculusFreeQ[#,x],Null,Throw[False]]],u];True]]]

SubstQ[u_] :=
  Head[u]===Subst

(* InverseFunctionQ[u] returns True if u is a call on an inverse function; else returns False. *)
InverseFunctionQ[u_] :=
  LogQ[u] || InverseTrigQ[u] && Length[u]==1 || InverseHyperbolicQ[u] || Head[u]===Mods

(* If u is free of inverse or calculus functions involving x,
    InverseFunctionFreeQ[u,x] returns true; else it returns False. *)
TrigHyperbolicFreeQ[u_,x_Symbol] :=
  If[AtomQ[u],
    True,
  If[TrigQ[u] || HyperbolicQ[u] || CalculusQ[u],
    FreeQ[u,x],
  Catch[Scan[Function[If[TrigHyperbolicFreeQ[#,x],Null,Throw[False]]],u];True]]]

(* If u is free of inverse or calculus functions involving x,
    InverseFunctionFreeQ[u,x] returns true; else it returns False. *)
InverseFunctionFreeQ[u_,x_Symbol] :=
  If[AtomQ[u],
    True,
  If[InverseFunctionQ[u] || CalculusQ[u],
(*  If[Head[u]===ArcTan && TanQ[u[[1]]] || Head[u]===ArcCot && CotQ[u[[1]]] ||
       Head[u]===ArcTanh && TanhQ[u[[1]]] || Head[u]===ArcCoth && CothQ[u[[1]]],
      InverseFunctionFreeQ[u[[1,1]],x], *)
    FreeQ[u,x],
  Catch[Scan[Function[If[InverseFunctionFreeQ[#,x],Null,Throw[False]]],u];True]]]
```

```
(* ElementaryExpressionQ[u] returns True if u is a sum, product, or power and all the operands
    are elementary expressions; or if u is a call on a trig, hyperbolic, or inverse function
    and all the arguments are elementary expressions; else it returns False. *)
(* ElementaryFunctionQ[u_] :=
  If[AtomQ[u],
    True,
  If[SumQ[u] || ProductQ[u] || PowerQ[u] || TrigQ[u] || HyperbolicQ[u] || InverseFunctionQ[u],
    Catch[Scan[Function[If[ElementaryFunctionQ[#],Null,Throw[False]]],u];True],
  False]] *)

(* If u is an expression of the form -v, NegativeCoefficientQ[u] returns True; else False. *)
NegativeCoefficientQ[u_] :=
  If[SumQ[u],
(*  MapAnd[NegativeCoefficientQ,u], *)
    NegativeCoefficientQ[First[u]],
  MatchQ[u, m_*v_. /; RationalQ[m] && m<0]]
```

■ Expression Domains

```
(* Real[u] returns True if u is a real-valued quantity, else returns False. *)
RealQ[u_] :=
  MapAnd[RealQ,u] /;
ListQ[u]

RealQ[u_] :=
  PossibleZeroQ[Im[N[u]]] /;
NumericQ[u]

RealQ[u_^v_] :=
  RealQ[u] && RealQ[v] && (IntegerQ[v] || PositiveOrZeroQ[u])

RealQ[u_*v_] :=
  RealQ[u] && RealQ[v]

RealQ[u_+v_] :=
  RealQ[u] && RealQ[v]

RealQ[f_[u_]] :=
  If[MemberQ[{Sin,Cos,Tan,Cot,Sec,Csc,ArcTan,ArcCot,Erf},f],
    RealQ[u],
  If[MemberQ[{ArcSin,ArcCos},f],
    LE[-1,u,1],
  If[f===Log,
    PositiveOrZeroQ[u],
  False]]]

RealQ[u_] :=
  False
```

```
(* If u is not 0 and has a positive form, PosQ[u] returns True, else it returns False. *)
PosQ[u_] :=
  If[RationalQ[u],
    u>0,
  If[NumberQ[u],
    If[PossibleZeroQ[Re[u]],
      Im[u]>0,
    Re[u]>0],
  If[NumericQ[u],
    Module[{v=N[u]},
    If[PossibleZeroQ[Re[v]],
      Im[v]>0,
    Re[v]>0]],
  If[ProductQ[u],
    If[PosQ[First[u]],
      PosQ[Rest[u]],
    NegQ[Rest[u]]],
  If[SumQ[u],
    Module[{v=Together[u]},
    If[SumQ[v],
      PosQ[First[v]],
    PosQ[v]]],
  True]]]]]

NegQ[u_] :=
  If[PossibleZeroQ[u],
    False,
  Not[PosQ[u]]]
```

Product Selector Functions

```
LeadTerm[u_] :=
  If[SumQ[u],
    First[u],
  u]

RemainingTerms[u_] :=
  If[SumQ[u],
    Rest[u],
  0]

(* LeadFactor[u] returns the leading factor of u. *)
LeadFactor[u_] :=
  If[ProductQ[u],
    LeadFactor[First[u]],
  If[ImaginaryQ[u],
    If[Im[u]===1,
      u,
    LeadFactor[Im[u]]],
  u]]

(* RemainingFactors[u] returns the remaining factors of u. *)
RemainingFactors[u_] :=
  If[ProductQ[u],
    RemainingFactors[First[u]]*Rest[u],
  If[ImaginaryQ[u],
    If[Im[u]===1,
      1,
    I*RemainingFactors[Im[u]]],
  1]]
```

```
(* LeadBase[u] returns the base of the leading factor of u. *)
LeadBase[u_] :=
  Module[{v=LeadFactor[u]},
  If[PowerQ[v],
    v[[1]],
  v]]

(* LeadDegree[u] returns the degree of the leading factor of u. *)
LeadDegree[u_] :=
  Module[{v=LeadFactor[u]},
  If[PowerQ[v],
    v[[2]],
  1]]

(* If v^n is a factor of u, FindFactor[u,v] returns the list {n,u/v^n}; else it returns False. *)
(* FindFactor[u_,v_] :=
  If[u===1,
    False,
  If[LeadBase[u]===v,
    {LeadDegree[u], RemainingFactors[u]},
  Module[{lst=FindFactor[RemainingFactors[u],v]},
  If[FalseQ[lst],
    False,
  {lst[[1]], LeadFactor[u]*lst[[2]]}]]]]] *)
```

Symbolic Relational Operators

```
(* LT[u,v] returns True if u and v are real-valued numeric quantities and u<v, else returns False *)
LT[u_,v_] :=
  RealNumericQ[u] && RealNumericQ[v] && Re[N[u]]<Re[N[v]]


LT[u_,v_,w_] :=
  LT[u,v] && LT[v,w]

(* LE[u,v] returns True if u and v are real-valued numeric quantities and u<=v, else returns False *)
LE[u_,v_] :=
  RealNumericQ[u] && RealNumericQ[v] && Re[N[u]]<=Re[N[v]]


LE[u_,v_,w_] :=
  LE[u,v] && LE[v,w]

(* GT[u,v] returns True if u and v are real-valued numeric quantities and u>v, else returns False *)
GT[u_,v_] :=
  RealNumericQ[u] && RealNumericQ[v] && Re[N[u]]>Re[N[v]]


GT[u_,v_,w_] :=
  GT[u,v] && GT[v,w]

(* GE[u,v] returns True if u and v are real-valued numeric quantities and u>=v, else returns False *)
GE[u_,v_] :=
  RealNumericQ[u] && RealNumericQ[v] && Re[N[u]]>=Re[N[v]]


GE[u_,v_,w_] :=
  GE[u,v] && GE[v,w]
```

Variable Dependence Functions

```
IndependentQ[u_,x_Symbol] :=
  FreeQ[u,x]
```

```
(* SplitFreeFactors[u,x] returns the list {v,w} where v is the product of the factors of u free of x
    and w is the product of the other factors. *)
(* Compare with the more active function ConstantFactor. *)
SplitFreeFactors[u_,x_Symbol] :=
  If[ProductQ[u],
    Map[Function[If[FreeQ[#,x],{#,1},{1,#}]],u],
  If[FreeQ[u,x],
    {u,1},
  {1,u}]]

(* SplitFreeTerms[u,x] returns the list {v,w} where v is the sum of the terms of u free of x
    and w is the sum of the other terms. *)
SplitFreeTerms[u_,x_Symbol] :=
  If[SumQ[u],
    Map[Function[SplitFreeTerms[#,x]],u],
  If[FreeQ[u,x],
    {u,0},
  {0,u}]]

(* If u (x) is a sum of the form a+b*v+c*w+..., SplitFactorsOfTerms[u,x] returns the list
    {{1,a},{b,v},{c,w},...}, where v, w, ... are regularized wrt x. *)
SplitFactorsOfTerms[u_,x_Symbol] :=
  Module[{lst=SplitFreeTerms[u,x],v,w},
  v=lst[[1]];
  w=lst[[2]];
  ( If[ZeroQ[w],
      lst={},
    If[SumQ[w],
      lst=Map[Function[SplitFreeFactors[#,x]],Apply[List,w]];
      lst=Map[Function[Prepend[SplitFreeFactors[Regularize[#[[2]],x],x],#[[1]]]],lst];
      lst=Map[Function[{#[[1]]*#[[2]],#[[3]]}],lst],
    lst=SplitFreeFactors[w,x];
    lst=Prepend[SplitFreeFactors[Regularize[lst[[2]],x],x],lst[[1]]];
    lst={{lst[[1]]*lst[[2]],lst[[3]]}}]] );
  If[ZeroQ[v],
    lst,
  Prepend[lst,{1,v}]]]

(* u is a sum.  SplitMonomialTerms[u,x] returns the list {v,w} where v is the sum of the
    monomial terms of u and w is the sum of the other terms. *)
SplitMonomialTerms[u_,x_Symbol] :=
  Map[Function[If[FreeQ[#,x] || MatchQ[#,a_.*x^n_. /; FreeQ[{a,n},x]], {#,0}, {0,#}]], u]
```

Polynomial Functions

■ Polynomial Recognizer Functions

```
(* If u (x) is equivalent to an expression of the form a+b*x where b is not 0, LinearQ[u,x]
    returns True; else it returns False. *)
LinearQ[u_,x_Symbol] :=
  PolynomialQ[u,x] && Exponent[u,x]===1

(* If u is polynomial in x of degree 2, QuadraticQ[u,x] returns True; else it returns False. *)
QuadraticQ[u_,x_Symbol] :=
  PolynomialQ[u,x] && Exponent[u,x]===2

(* If u (x) is equivalent to an expression of the form x^n, MonomialQ[u,x] returns True;
    else it returns False.  Note that not all monomials are polynomials. *)
MonomialQ[u_,x_Symbol] :=
  MatchQ[u,x^n_. /; FreeQ[n,x]]
```

```
(* If u (x) is equivalent to an expression of the form a+b*x^n, BinomialQ[u,x] returns True;
    else it returns False.  Note that not all binomials are polynomials. *)
BinomialQ[u_,x_Symbol] :=
  NotFalseQ[BinomialTest[u,x]]
```

```
(* If u (x) is a sum and each term is free of x or an expression of the form a*x^n,
    MonomialSumQ[u,x] returns True; else it returns False. *)
MonomialSumQ[u_,x_Symbol] :=
  SumQ[u] && Catch[
    Scan[Function[If[FreeQ[#,x] || MonomialQ[SplitFreeFactors[#,x][[2]],x], Null, Throw[False]]],u];
    True]
```

Polynomial Terms Functions

```
(* If u (x) is an expression of the form a*x^n where n is zero or a positive integer,
    PolynomialTermQ[u,x] returns True; else it returns False. *)
PolynomialTermQ[u_,x_Symbol] :=
  FreeQ[u,x] || MatchQ[u,a_.*x^n_. /; FreeQ[a,x] && IntegerQ[n] && n>0]
```

```
(* u (x) is a sum.  PolynomialTerms[u,x] returns the sum of the polynomial terms of u (x). *)
PolynomialTerms[u_,x_Symbol] :=
  Map[Function[If[PolynomialTermQ[#,x],#,0]],u]
```

```
(* u (x) is a sum.  NonpolynomialTerms[u,x] returns the sum of the nonpolynomial terms of u (x). *)
NonpolynomialTerms[u_,x_Symbol] :=
  Map[Function[If[PolynomialTermQ[#,x],0,#]],u]
```

■ Binomial Test Functions

```
(* If u (x) is equivalent to an expression of the form a+b*x^n where a,b and n are free of x,
    BinomialTest[u,x] returns the list {a,b,n}; else it returns False.
    Note that not all binomials are polynomials. *)
BinomialTest[u_,x_Symbol] :=
  If[u===x,
    {0,1,1},
  If[FreeQ[u,x],
    {0,u,0},
  If[PowerQ[u],
    If[u[[1]]===x && FreeQ[u[[2]],x],
      {0,1,u[[2]]},
    False],
  Module[{lst1,lst2},
  If[ProductQ[u],
    lst1=BinomialTest[First[u],x];
    If[FalseQ[lst1],
      False,
    lst2=BinomialTest[Rest[u],x];
    If[FalseQ[lst2],
      False,
    Module[{a,b,c,d,m,n},
    {a,b,m}=lst1;
    {c,d,n}=lst2;
    If[m===0,
      {b*c,b*d,n},
    If[n===0,
      {a*d,b*d,m},
    If[a===0,
      If[c===0,
        {0,b*d,m+n},
      If[m+n===0,
        {b*d,b*c,m},
      False]],
    If[c===0,
      If[m+n===0,
        {b*d,a*d,n},
      False],
    False]]]]]]],
  If[SumQ[u],
    lst1=BinomialTest[First[u],x];
    If[FalseQ[lst1],
      False,
    lst2=BinomialTest[Rest[u],x];
    If[FalseQ[lst2],
      False,
    Module[{a,b,c,d,m,n},
    {a,b,m}=lst1;
    {c,d,n}=lst2;
    If[m===0,
      {b+c,d,n},
    If[n===0,
      {a+d,b,m},
    If[m===n,
      {a+c,b+d,m},
    False]]]]]],
  False]]]]]]
```

Perfect Power Test Function

```
(* If u (x) is equivalent to a polynomial raised to an integer power greater than 1,
   PerfectPowerTest[u,x] returns u (x) as an expanded polynomial raised to the power;
   else it returns False. *)
PerfectPowerTest[u_,x_Symbol] :=
  If[PolynomialQ[u,x],
    Module[{lst=FactorSquareFreeList[u],gcd=0,v=1},
    If[lst[[1]]==={1,1},
      lst=Rest[lst]];
    Scan[Function[gcd=GCD[gcd,#[[2]]]],lst];
    If[gcd>1,
      Scan[Function[v=v*#[[1]]^(#[[2]]/gcd)],lst];
      Expand[v]^gcd,
    False]],
  False]
```

■ Square Free Factor Test Function

```
(* If u (x) can be square free factored, SquareFreeFactorTest[u,x] returns u (x) in
   factored form; else it returns False. *)
(* SquareFreeFactorTest[u_,x_Symbol] :=
  If[PolynomialQ[u,x],
    Module[{v=FactorSquareFree[u]},
    If[PowerQ[v] || ProductQ[v],
      v,
    False]],
  False] *)
```

Function Recognizers

■ Rational Function Recognizer

```
(* If u (x) is a polynomial or rational function of x, RationalFunctionQ[u,x] returns True;
   else it returns False. *)
RationalFunctionQ[u_,x_Symbol] :=
  If[AtomQ[u],
    True,
  If[IntegerPowerQ[u],
    RationalFunctionQ[u[[1]],x],
  If[ProductQ[u] || SumQ[u],
    Catch[Scan[Function[If[RationalFunctionQ[#,x],Null,Throw[False]]],u];True],
  If[FreeQ[u,x],
    True,
  False]]]]
```

```
(* Assuming u is a rational function of x, RationalFunctionExponents[u,x] returns a list of the
   exponent of the numerator of u and the exponent of the denominator of u. *)
RationalFunctionExponents[u_,x_Symbol] :=
  If[PolynomialQ[u,x],
    {Exponent[u,x],0},
  If[IntegerPowerQ[u],
    If[u[[2]]>0,
      u[[2]]*RationalFunctionExponents[u[[1]],x],
    (-u[[2]])*Reverse[RationalFunctionExponents[u[[1]],x]]],
  If[ProductQ[u],
    RationalFunctionExponents[First[u],x]+RationalFunctionExponents[Rest[u],x],
  If[SumQ[u],
    Module[{v=Together[u]},
    If[SumQ[v],
      Module[{lst1,lst2},
      lst1=RationalFunctionExponents[First[u],x];
      lst2=RationalFunctionExponents[Rest[u],x];
      {Max[lst1[[1]]+lst2[[2]],lst2[[1]]+lst1[[2]]],lst1[[2]]+lst2[[2]]}],
    RationalFunctionExponents[v,x]]],
  {0,0}]]]]
```

Algebraic Function Recognizer

```
(* If u (x) is an algebraic function of x, AlgebraicFunctionQ[u,x] returns True; else False. *)
AlgebraicFunctionQ[u_,x_Symbol] :=
  If[AtomQ[u] || FreeQ[u,x],
    True,
  If[RationalPowerQ[u],
    AlgebraicFunctionQ[u[[1]],x],
  If[ProductQ[u] || SumQ[u],
    Catch[Scan[Function[If[AlgebraicFunctionQ[#,x],Null,Throw[False]]],u];True],
  False]]]
```

Quotient of Linears Recognizer and Accessor

```
(* If u is equivalent to an expression of the form (a+b*x)/(c+d*x), QuotientOfLinearsQ[u,x]
   returns True; else it returns False. *)
QuotientOfLinearsQ[a_*u_,x_] :=
  QuotientOfLinearsQ[u,x] /;
FreeQ[a,x]

QuotientOfLinearsQ[a_+u_,x_] :=
  QuotientOfLinearsQ[u,x] /;
FreeQ[a,x]

QuotientOfLinearsQ[1/u_,x_] :=
  QuotientOfLinearsQ[u,x]

QuotientOfLinearsQ[u_,x_] :=
  True /;
LinearQ[u,x]

QuotientOfLinearsQ[u_/v_,x_] :=
  True /;
LinearQ[u,x] && LinearQ[v,x]

QuotientOfLinearsQ[u_,x_] :=
  u===x || FreeQ[u,x]
```

```
(* If u is equivalent to an expression of the form (a+b*x)/(c+d*x), QuotientOfLinearsParts[u,x]
   returns the list {a, b, c, d}. *)
QuotientOfLinearsParts[a_*u_,x_] :=
  Apply[Function[{a*#1, a*#2, #3, #4}], QuotientOfLinearsParts[u,x]] /;
FreeQ[a,x]

QuotientOfLinearsParts[a_+u_,x_] :=
  Apply[Function[{#1+a*#3, #2+a*#4, #3, #4}], QuotientOfLinearsParts[u,x]] /;
FreeQ[a,x]

QuotientOfLinearsParts[1/u_,x_] :=
  Apply[Function[{#3, #4, #1, #2}], QuotientOfLinearsParts[u,x]]

QuotientOfLinearsParts[u_,x_] :=
  {Coefficient[u,x,0], Coefficient[u,x,1], 1, 0} /;
LinearQ[u,x]

QuotientOfLinearsParts[u_/v_,x_] :=
  {Coefficient[u,x,0], Coefficient[u,x,1], Coefficient[v,x,0], Coefficient[v,x,1]} /;
LinearQ[u,x] && LinearQ[v,x]

QuotientOfLinearsParts[u_,x_] :=
  If[u===x,
    {0, 1, 1, 0},
  If[FreeQ[u,x],
    {u, 0, 1, 0},
  Print["QuotientOfLinearParts error!"];
  {u, 0, 1, 0}]]
```

Improper Rational Functions

```
(* u (x) is an improper fraction if it is an expression of the form w (v (x))/t(v (x)) where w (x)
   and t (x) are polynomials in x and the degree of w (x) is greater than or equal the degree
   of t (x). *)
```

■ Improper Rational Function Recognizer

```
(* If u/v is an improper fraction, ImproperFractionQ[u,v,x] returns True; else it returns False. *)
(* ImproperFractionQ[u_,v_,x_Symbol] :=
  Module[{lst1=PolynomialFunctionOf[u,x],lst2=PolynomialFunctionOf[v,x]},
  lst1[[1]]===lst2[[1]] && Exponent[lst1[[2]],x]>=Exponent[lst2[[2]],x]] *)

(* If u/v is an improper rational function where v is of the form fraction a+b*x+c*x^2 or a+b*x^n,
   ImproperRationalFunctionQ[u,v,x] returns True; else it returns False. *)
ImproperRationalFunctionQ[u_,v_,x_Symbol] :=
  PolynomialQ[u,x] &&
  PolynomialQ[v,x] &&
  Not[MatchQ[u,(a_.+b_.*x)^n_. /; FreeQ[{a,b},x] && IntegerQ[n]] &&
      MatchQ[v,(a_.+b_.*x)^n_. /; FreeQ[{a,b},x] && IntegerQ[n]]] &&
  (QuadraticQ[v,x] && Exponent[u,x]>=2 ||
    MatchQ[v,a_+b_.*x^n_. /; FreeQ[{a,b},x] && IntegerQ[n] && 0<n<=Exponent[u,x]])
```

Improper Fraction Expander

```
(* If u is an improper fraction, ExpandImproperFraction[u,x] returns the list {q,a,r}
    where q is the integral part of u and a*r is the proper fractional part of u;
    else it returns False. *)
ExpandImproperFraction[u_,x_Symbol] :=
  Module[{tmp},
  If[NotFalseQ[tmp=ExpandImproperFraction[Numerator[u],Denominator[u],x]],
    tmp,
  If[NotFalseQ[tmp=ExpandImproperFraction[SmartNumerator[u],SmartDenominator[u],x]],
    tmp,
  If[FunctionOfQ[Sin[x],u,x],
    tmp=Regularize[SubstFor[Sin[x],u,x],x];
    If[NotFalseQ[tmp=ExpandImproperFraction[Numerator[tmp],Denominator[tmp],x]],
      Subst[tmp,x,Sin[x]],
    False],
  If[FunctionOfQ[Cos[x],u,x],
    tmp=Regularize[SubstFor[Cos[x],u,x],x];
    If[NotFalseQ[tmp=ExpandImproperFraction[Numerator[tmp],Denominator[tmp],x]],
      Subst[tmp,x,Cos[x]],
    False],
  If[FunctionOfQ[Sinh[x],u,x],
    tmp=Regularize[SubstFor[Sinh[x],u,x],x];
    If[NotFalseQ[tmp=ExpandImproperFraction[Numerator[tmp],Denominator[tmp],x]],
      Subst[tmp,x,Sinh[x]],
    False],
  If[FunctionOfQ[Cosh[x],u,x],
    tmp=Regularize[SubstFor[Cosh[x],u,x],x];
    If[NotFalseQ[tmp=ExpandImproperFraction[Numerator[tmp],Denominator[tmp],x]],
      Subst[tmp,x,Cosh[x]],
    False],
  False]]]]]]]

ExpandImproperFraction[u_,v_,x_Symbol] :=
  Module[{lst1,lst2},
  lst1=PolynomialFunctionOf[u,x];
  lst2=PolynomialFunctionOf[v,x];
  If[lst1[[1]]===lst2[[1]] && Exponent[lst1[[2]],x]>=Exponent[lst2[[2]],x],
    ReplaceAll[PolynomialDivide[lst1[[2]],lst2[[2]],x],x->lst1[[1]]],
  False]]

(* PolynomialDivide[u,v,x] returns the list {q,a,r} where q is the integral part of u/v and
    a*r is the proper fractional part of u/v; else it returns False. *)
PolynomialDivide[u_,v_,x_Symbol] :=
  Prepend[SplitFreeFactors[Regularize[PolynomialRemainder[u,v,x]/v,x],x],
        PolynomialQuotient[u,v,x]]

SmartNumerator[u_] :=
  If[MemberQ[{Cot,Sec,Csc,Coth,Sech,Csch},Head[u]],
    1,
  If[PowerQ[u] && IntegerQ[u[[2]]] && MemberQ[{Cot,Sec,Csc,Coth,Sech,Csch},Head[u[[1]]]]],
    1,
  If[PowerQ[u] && RationalQ[u[[2]]] && u[[2]]<0,
    1,
  If[ProductQ[u],
    Map[SmartNumerator,u],
  u]]]]
```

```
SmartDenominator[u_] :=
  If[MemberQ[{Cot,Sec,Csc,Coth,Sech,Csch},Head[u]],
    1/u,
  If[PowerQ[u] && IntegerQ[u[[2]]] && MemberQ[{Cot,Sec,Csc,Coth,Sech,Csch},Head[u[[1]]]],
    1/u,
  If[PowerQ[u] && RationalQ[u[[2]]] && u[[2]]<0,
    1/u,
  If[ProductQ[u],
    Map[SmartDenominator,u],
  1]]]]
```

- **Polynomial Function Composer**

```
(* PolynomialFunctionOf[u,x] returns the list {v (x),w (x)} where w (v (x)) equals u (x), w (x) is
    a polynomial in x, and v (x) is minimal *)
PolynomialFunctionOf[u_,x_Symbol] :=
  If[AtomQ[u],
    If[u===x,
      {x,x},
    {1,u}],
  If[PositiveIntegerPowerQ[u],
    Module[{lst=PolynomialFunctionOf[u[[1]],x]},
    {lst[[1]],lst[[2]]^u[[2]]}],
  If[ProductQ[u],
    Module[{lst1=PolynomialFunctionOf[First[u],x],lst2=PolynomialFunctionOf[Rest[u],x]},
    If[lst1[[1]]===1,
      {lst2[[1]],lst1[[2]]*lst2[[2]]},
    If[lst2[[1]]===1,
      {lst1[[1]],lst1[[2]]*lst2[[2]]},
    If[lst1[[1]]===lst2[[1]],
      {lst1[[1]],lst1[[2]]*lst2[[2]]},
    {u,x}]]]],
  If[SumQ[u],
    Module[{lst1=PolynomialFunctionOf[First[u],x],lst2=PolynomialFunctionOf[Rest[u],x]},
    If[lst1[[1]]===1,
      {lst2[[1]],lst1[[2]]+lst2[[2]]},
    If[lst2[[1]]===1,
      {lst1[[1]],lst1[[2]]+lst2[[2]]},
    If[lst1[[1]]===lst2[[1]],
      {lst1[[1]],lst1[[2]]+lst2[[2]]},
    {u,x}]]]],
  If[FreeQ[u,x],
    {1,u},
  {u,x}]]]]]
```

Distribution Function

```
(* Dist[u,v] returns the sum of u times each term of v, provided v is free of Int. *)
Dist[u_,v_] :=
  If[SumQ[v],
    Map[Function[u*#],v],
  u*v]
```

Numeric Factors

```
(* If lst is a list of n terms, CommonNumericFactors[lst] returns a n+1-element list whose first
   element is the product of the numeric factors common to all terms of lst, and whose remaining
   elements are quotients of each term divided by the numeric common factor. *)
CommonNumericFactors [lst_] :=
  Module[{num=Apply[GCD,Map[NumericFactor,lst]]},
  Prepend[Map[Function[#/num],lst],num]]

NumericFactor[u_] :=
  If[NumberQ[u],
    If[ZeroQ[Im[u]],
      u,
    If[ZeroQ[Re[u]],
      Im[u],
    1]],
  If[PowerQ[u],
    If[RationalQ[u[[1]]] && FractionQ[u[[2]]],
      If[u[[2]]>0,
        1/Denominator[u[[1]]],
      1/Denominator[1/u[[1]]]],
    1],
  If[ProductQ[u],
    NumericFactor[First[u]]*NumericFactor[Rest[u]],
  1]]]

NonnumericFactors[u_] :=
  If[NumberQ[u],
    If[ZeroQ[Im[u]],
      1,
    If[ZeroQ[Re[u]],
      I,
    u]],
  If[PowerQ[u],
    If[RationalQ[u[[1]]] && FractionQ[u[[2]]],
      u/NumericFactor[u],
    u],
  If[ProductQ[u],
    NonnumericFactors[First[u]]*NonnumericFactors[Rest[u]],
  u]]]
```

Content Factoring

```
(* ContentFactor[expn] returns expn with the content of sum factors factored out. *)
(* Basis: a*b+a*c == a*(b+c) *)
ContentFactor[expn_] :=
  If[AtomQ[expn],
    expn,
  If[ProductQ[expn],
    Map[ContentFactor,expn],
  If[SumQ[expn],
    Module[{lst=CommonFactors[Apply[List,expn]]},
    If[lst[[1]]===1 || lst[[1]]===-1,
      expn,
    lst[[1]]*Apply[Plus,Rest[lst]]]],
  expn]]]
```

```
(* If lst is a list of n terms, CommonFactors[lst] returns a n+1-element list whose first
   element is the product of the factors common to all terms of lst, and whose remaining
   elements are quotients of each term divided by the common factor. *)
CommonFactors [lst_] :=
  Module[{lst1,lst2,lst3,lst4,common,base,num},
  lst1=Map[NonnumericFactors,lst];
  lst2=Map[NumericFactor,lst];
  num=Apply[GCD,lst2];
  ( If[MapAnd[Function[#<0],lst2],
      num=-num] );
  common=num;
  lst2=Map[Function[#/num],lst2];
  While[True,
    lst3=Map[LeadFactor,lst1];
    ( If[Apply[SameQ,lst3],
        common=common*lst3[[1]];
        lst1=Map[RemainingFactors,lst1],
      If[MapAnd[Function[LogQ[#] && IntegerQ[First[#]] && First[#]>0],lst3] &&
          MapAnd[RationalQ,lst4=Map[Function[FullSimplify[#/First[lst3]]],lst3]],
        num=Apply[GCD,lst4];
        common=common*Log[(First[lst3][[1]])^num];
        lst2=Map2[Function[#1*#2/num],lst2,lst4];
        lst1=Map[RemainingFactors,lst1],
      lst4=Map[LeadDegree,lst1];
      If[Apply[SameQ,Map[LeadBase,lst1]] && MapAnd[RationalQ,lst4],
        num=Smallest[lst4];
        base=LeadBase[lst1[[1]]];
        ( If[num!=0,
            common=common*base^num] );
        lst2=Map2[Function[#1*base^(#2-num)],lst2,lst4];
        lst1=Map[RemainingFactors,lst1],
      If[Length[lst1]==2 && ZeroQ[LeadBase[lst1[[1]]]+LeadBase[lst1[[2]]]] &&
         NonzeroQ[lst1[[1]]-1] && IntegerQ[lst4[[1]]] && FractionQ[lst4[[2]]],
        num=Min[lst4];
        base=LeadBase[lst1[[2]]];
        ( If[num!=0,
            common=common*base^num] );
        lst2={lst2[[1]]*(-1)^lst4[[1]],lst2[[2]]};
        lst2=Map2[Function[#1*base^(#2-num)],lst2,lst4];
        lst1=Map[RemainingFactors,lst1],
      If[Length[lst1]==2 && ZeroQ[LeadBase[lst1[[1]]]+LeadBase[lst1[[2]]]] &&
         NonzeroQ[lst1[[2]]-1] && IntegerQ[lst4[[2]]] && FractionQ[lst4[[1]]],
        num=Min[lst4];
        base=LeadBase[lst1[[1]]];
        ( If[num!=0,
            common=common*base^num] );
        lst2={lst2[[1]],lst2[[2]]*(-1)^lst4[[2]]};
        lst2=Map2[Function[#1*base^(#2-num)],lst2,lst4];
        lst1=Map[RemainingFactors,lst1],
      num=MostMainFactorPosition[lst3];
      lst2=ReplacePart[lst2,lst3[[num]]*lst2[[num]],num];
      lst1=ReplacePart[lst1,RemainingFactors[lst1[[num]]],num]]]]]] );
    If[MapAnd[Function[#===1],lst1],
      Return[Prepend[lst2,common]]]]]

MostMainFactorPosition[lst_List] :=
  Module[{factor=1,num=1},
  Do[If[FactorOrder[lst[[i]],factor]>0,factor=lst[[i]];num=i],{i,Length[lst]}];
  num]
```

```
FactorOrder[u_,v_] :=
  If[u===1,
    If[v===1,
      0,
    -1],
  If[v===1,
    1,
  Order[u,v]]]

Smallest[num1_,num2_] :=
  If[num1>0,
    If[num2>0,
      Min[num1,num2],
    0],
  If[num2>0,
    0,
  Max[num1,num2]]]


Smallest[lst_List] :=
  Module[{num=lst[[1]]},
  Scan[Function[num=Smallest[num,#]],Rest[lst]];
  num]
```

■ Monomial factoring

```
(* MonomialFactor[u,x] returns the list {n,v} where x^n*v==u and n is free of x. *)
MonomialFactor[u_,x_Symbol] :=
  If[AtomQ[u],
    If[u===x,
      {1,1},
    {0,u}],
  If[PowerQ[u],
    If[IntegerQ[u[[2]]],
      Module[{lst=MonomialFactor[u[[1]],x]},
      {lst[[1]]*u[[2]],lst[[2]]^u[[2]]}],
    If[u[[1]]===x && FreeQ[u[[2]],x],
      {u[[2]],1},
    {0,u}]],
  If[ProductQ[u],
    Module[{lst1=MonomialFactor[First[u],x],lst2=MonomialFactor[Rest[u],x]},
    {lst1[[1]]+lst2[[1]],lst1[[2]]*lst2[[2]]}],
  If[SumQ[u],
    Module[{lst,deg},
    lst=Map[Function[MonomialFactor[#,x]],Apply[List,u]];
    deg=lst[[1,1]];
    Scan[Function[deg=MinimumDegree[deg,#[[1]]]],Rest[lst]];
    If[ZeroQ[deg] || RationalQ[deg] && deg<0,
      {0,u},
    {deg,Apply[Plus,Map[Function[x^(#[[1]]-deg)*#[[2]]],lst]]}]],
  {0,u}]]]]
```

```
MinimumDegree[deg1_,deg2_] :=
  If[RationalQ[deg1],
    If[RationalQ[deg2],
      Min[deg1,deg2],
    deg1],
  If[RationalQ[deg2],
    deg2,
  Module[{deg=Simplify[deg1-deg2]},
  If[RationalQ[deg],
    If[deg>0,
      deg2,
    deg1],
  If[OrderedQ[{deg1,deg2}],
    deg1,
  deg2]]]]]
```

■ Constant factoring

```
(* ConstantFactor[u,x] returns a 2-element list of the factors of u[x] free of x and the
    factors not free of u[x].  Common constant factors of the terms of sums are also collected. *)
(* Compare with the more passive function SplitFreeFactors. *)
ConstantFactor[u_,x_Symbol] :=
  If[FreeQ[u,x],
    {u,1},
  If[AtomQ[u],
    {1,u},
  If[PowerQ[u] && FreeQ[u[[2]],x],
    Module[{lst=ConstantFactor[u[[1]],x],tmp},
    If[IntegerQ[u[[2]]],
      {lst[[1]]^u[[2]],lst[[2]]^u[[2]]},
    tmp=PositiveFactors[lst[[1]]];
    If[tmp===1,
      {1,u},
    {tmp^u[[2]],(NonpositiveFactors[lst[[1]]]*lst[[2]])^u[[2]]}]]],
  If[ProductQ[u],
    Module[{lst=Map[Function[ConstantFactor[#,x]],Apply[List,u]]},
    {Apply[Times,Map[First,lst]],Apply[Times,Map[Function[#[[2]]],lst]]}],
  If[SumQ[u],
    Module[{lst1=Map[Function[ConstantFactor[#,x]],Apply[List,u]]},
    If[Apply[SameQ,Map[Function[#[[2]]],lst1]],
      {Apply[Plus,Map[First,lst1]],lst1[[1,2]]},
    Module[{lst2=CommonFactors[Map[First,lst1]]},
    {First[lst2],Apply[Plus,Map2[Times,Rest[lst2],Map[Function[#[[2]]],lst1]]]}]]],
  {1,u}]]]]]

(* PositiveFactors[u] returns the positive factors of u *)
PositiveFactors[u_] :=
  If[ZeroQ[u],
    1,
  If[RationalQ[u],
    Abs[u],
  If[PositiveQ[u],
    u,
  If[ProductQ[u],
    Map[PositiveFactors,u],
  1]]]]
```

```
(* NonpositiveFactors[u] returns the nonpositive factors of u *)
NonpositiveFactors[u_] :=
  If[ZeroQ[u],
    u,
  If[RationalQ[u],
    Sign[u],
  If[PositiveQ[u],
    1,
  If[ProductQ[u],
    Map[NonpositiveFactors,u],
  u]]]]
```

Function Of Functions

■ Function of a linear binomial

```
(* If u (x) is equivalent to an expression of the form f (a+b*x) and not the case that a==0 and
    b==1, FunctionOfLinear[u,x] returns the list {f (x),a,b}; else it returns False. *)
FunctionOfLinear[u_,x_Symbol] :=
  Module[{lst=FunctionOfLinear[u,False,False,x,False]},
  If[FalseQ[lst] || FalseQ[lst[[1]]] || lst[[1]]===0 && lst[[2]]===1,
    False,
  {FunctionOfLinearSubst[u,lst[[1]],lst[[2]],x],lst[[1]],lst[[2]]}]]

FunctionOfLinear[u_,a_,b_,x_,flag_] :=
  If[FreeQ[u,x],
    {a,b},
  If[CalculusQ[u],
    False,
  If[LinearQ[u,x],
    If[FalseQ[a],
      {Coefficient[u,x,0],Coefficient[u,x,1]},
    Module[{lst=CommonFactors[{b,Coefficient[u,x,1]}]},
    If[ZeroQ[Coefficient[u,x,0]] && Not[flag],
      {0,lst[[1]]},
    If[ZeroQ[b*Coefficient[u,x,0]-a*Coefficient[u,x,1]],
      {a/lst[[2]],lst[[1]]},
    {0,1}]]]]],
  If[PowerQ[u] && FreeQ[u[[1]],x],
    FunctionOfLinear[Log[u[[1]]]*u[[2]],a,b,x,False],
  Module[{lst},
  If[ProductQ[u] && NonzeroQ[(lst=MonomialFactor[u,x])[[1]]],
    If[False && IntegerQ[lst[[1]]] && lst[[1]]!=-1 && FreeQ[lst[[2]],x],
      If[RationalQ[LeadFactor[lst[[2]]]] && LeadFactor[lst[[2]]]<0,
        FunctionOfLinear[DivideDegreesOfFactors[-lst[[2]],lst[[1]]]*x,a,b,x,False],
      FunctionOfLinear[DivideDegreesOfFactors[lst[[2]],lst[[1]]]*x,a,b,x,False]],
    False],
  lst={a,b};
  Catch[
  Scan[Function[lst=FunctionOfLinear[#,lst[[1]],lst[[2]],x,SumQ[u]];
            If[lst===False,Throw[False]]],u];
  lst]]]]]]]
```

```
FunctionOfLinearSubst[u_,a_,b_,x_] :=
  If[FreeQ[u,x],
    u,
  If[LinearQ[u,x],
    Module[{tmp=Coefficient[u,x,1]},
    tmp=If[tmp===b, 1, tmp/b];
    Coefficient[u,x,0]-a*tmp+tmp*x],
  If[PowerQ[u] && FreeQ[u[[1]],x],
    E^FullSimplify[FunctionOfLinearSubst[Log[u[[1]]]*u[[2]],a,b,x]],
  Module[{lst},
  If[ProductQ[u] && NonzeroQ[(lst=MonomialFactor[u,x])[[1]]],
    If[RationalQ[LeadFactor[lst[[2]]]] && LeadFactor[lst[[2]]]<0,
      -FunctionOfLinearSubst[DivideDegreesOfFactors[-lst[[2]],lst[[1]]]*x,a,b,x]^lst[[1]],
    FunctionOfLinearSubst[DivideDegreesOfFactors[lst[[2]],lst[[1]]]*x,a,b,x]^lst[[1]]],
  Map[Function[FunctionOfLinearSubst[#,a,b,x]],u]]]]]]

(* DivideDegreesOfFactors[u,n] returns the product of the base of the factors of u raised to
    the degree of the factors divided by n. *)
DivideDegreesOfFactors[u_,n_] :=
  If[ProductQ[u],
    Map[Function[LeadBase[#]^(LeadDegree[#]/n)],u],
  LeadBase[u]^(LeadDegree[u]/n)]
```

■ Function of an inverse linear binomial

```
(* If u is a function of an inverse linear binomial of the form 1/(a+b*x),
    FunctionOfInverseLinear[u,x] returns the list {a,b}; else it returns False. *)
FunctionOfInverseLinear[u_,x_Symbol] :=
  FunctionOfInverseLinear[u,Null,x]

FunctionOfInverseLinear[u_,lst_,x_] :=
  If[FreeQ[u,x],
    lst,
  If[u===x,
    False,
  If[QuotientOfLinearsQ[u,x],
    Module[{tmp=Drop[QuotientOfLinearsParts[u,x],2]},
    If[tmp[[2]]===0,
      False,
    If[lst===Null,
      tmp,
    If[ZeroQ[lst[[1]]*tmp[[2]]-lst[[2]]*tmp[[1]]],
      lst,
    False]]]],
  If[CalculusQ[u],
    False,
  Module[{tmp=lst},Catch[
  Scan[Function[If[FalseQ[tmp=FunctionOfInverseLinear[#,tmp,x]],Throw[False]]],u];
  tmp]]]]]]
```

Function of exponential functions of a linear binomial

```
(* If u is a function of f^(a+b*x), FunctionOfExponentialOfLinear[u,x] returns the list {v,a,b,f}
    where v of f^(a+b*x) equals u; else it returns False. *)
FunctionOfExponentialOfLinear[u_,x_Symbol] :=
  Module[{lst=FunctionOfExponentialOfLinear[u,x,False,False,False],a,b,f},
  If[FalseQ[lst] || FalseQ[lst[[1]]],
    False,
  a=lst[[1]];
  b=lst[[2]];
  f=lst[[3]];
  ( If[MatchQ[u,v_*g_^(c_.+d_*x) /; FreeQ[{c,d,g},x] && NumericFactor[d]<0] && NumericFactor[b]>0,
      a=-a;
      b=-b] );
  {FunctionOfExponentialOfLinearSubst[u,a,b,f,x],a,b,f}]]

(* If u is a function of f^(a+b*x), FunctionOfExponentialOfLinear[u,x,False,False,False]
    returns the list {a, b, f}; else it returns False. *)
FunctionOfExponentialOfLinear[u_,x_,a_,b_,f_] :=
  If[FreeQ[u,x],
    {a,b,f},
  If[u===x || CalculusQ[u],
    False,
  If[PowerQ[u] && FreeQ[u[[1]],x] && LinearQ[u[[2]],x],
    FunctionOfExponentialOfLinearAux[a,b,f,Coefficient[u[[2]],x,0],Coefficient[u[[2]],x,1],u[[1]]],
  If[HyperbolicQ[u] && LinearQ[u[[1]],x],
    FunctionOfExponentialOfLinearAux[a,b,f,Coefficient[u[[1]],x,0],Coefficient[u[[1]],x,1],E],
  Module[{lst},
  If[PowerQ[u] && FreeQ[u[[1]],x] && SumQ[u[[2]]],
    lst=FunctionOfExponentialOfLinear[u[[1]]^First[u[[2]]],x,a,b,f];
    If[lst===False,
      False,
    FunctionOfExponentialOfLinear[u[[1]]^Rest[u[[2]]],x,lst[[1]],lst[[2]],lst[[3]]]],
  lst={a,b,f};
  Catch[Scan[Function[
    lst=FunctionOfExponentialOfLinear[#,x,lst[[1]],lst[[2]],lst[[3]]];
    If[lst===False,Throw[False]]],u];
  lst]]]]]]]

FunctionOfExponentialOfLinearAux[a_,b_,f_,c_,d_,g_] :=
  If[FalseQ[a],
    {c,d,g},
  If[ZeroQ[Log[f]*NonnumericFactors[b]-Log[g]*NonnumericFactors[d]],
    Module[{gcd=GCD[NumericFactor[b],NumericFactor[d]]},
    ( If[NumericFactor[b]<0 && NumericFactor[d]<0,
        gcd=-gcd] );
    If[gcd==NumericFactor[b],
      {a,b,f},
    If[gcd==NumericFactor[d],
      {c,d,g},
    {0,gcd*NonnumericFactors[b],f}]]],
  False]]
```

```
(* u is a function of f^(a+b*x).  FunctionOfExponentialOfLinearSubst [u,a,b,f,x] returns u
    with f^(a+b*x) replaced by x. *)
FunctionOfExponentialOfLinearSubst [u_,a_,b_,f_,x_] :=
  If[FreeQ[u,x],
    u,
  If[PowerQ[u] && FreeQ[u[[1]],x] && LinearQ[u[[2]],x],
    Module[{c,d,g},
    c=Coefficient[u[[2]],x,0];
    d=Coefficient[u[[2]],x,1];
    g=u[[1]];
    g^(c-a*d/b)*x^(d*Log[g]/(b*Log[f]))],
  If[HyperbolicQ[u] && LinearQ[u[[1]],x],
    Module[{c,d,tmp},
    c=Coefficient[u[[1]],x,0];
    d=Coefficient[u[[1]],x,1];
    tmp=E^(c-a*d/b)*x^(d/(b*Log[f]));
    If[SinhQ[u],
      tmp/2-1/(2*tmp),
    If[CoshQ[u],
      tmp/2+1/(2*tmp),
    If[TanhQ[u],
      (tmp-1/tmp)/(tmp+1/tmp),
    If[CothQ[u],
      (tmp+1/tmp)/(tmp-1/tmp),
    If[SechQ[u],
      2/(tmp+1/tmp),
    2/(tmp-1/tmp)]]]]]]],
  If[PowerQ[u] && FreeQ[u[[1]],x] && SumQ[u[[2]]],
    FunctionOfExponentialOfLinearSubst [u[[1]]^First[u[[2]]],a,b,f,x]*
    FunctionOfExponentialOfLinearSubst [u[[1]]^Rest[u[[2]]],a,b,f,x],
  Map[Function[FunctionOfExponentialOfLinearSubst [#,a,b,f,x]],u]]]]]
```

■ Function of trig functions of a linear binomial

```
(* If u is a function of trig functions of a linear function of x,
    FunctionOfTrigOfLinearQ[u,x] returns True; else it returns False. *)
FunctionOfTrigOfLinearQ[u_,x_Symbol] :=
  Not[MemberQ[{Null, False}, FunctionOfTrig[u,Null,x]]] &&
    RecognizedFunctionOfTrigQ[SubstInertSineForTrigOfLinear[u,x],x]

(* If u is a function of trig functions of v where v is a linear function of x,
    FunctionOfTrig[u,x] returns v; else it returns False. *)
FunctionOfTrig[u_,x_Symbol] :=
  Module[{v=FunctionOfTrig[u,Null,x]},
  If[v===Null, False, v]]

FunctionOfTrig[u_,v_,x_] :=
  If[AtomQ[u],
    If[u===x,
      False,
    v],
  If[TrigQ[u] && LinearQ[u[[1]],x],
    If[v===Null,
      u[[1]],
    Module[{a=Coefficient[v,x,0],b=Coefficient[v,x,1],
            c=Coefficient[u[[1]],x,0],d=Coefficient[u[[1]],x,1]},
    If[ZeroQ[a*d-b*c] && RationalQ[b/d],
      a/Numerator[b/d]+b*x/Numerator[b/d],
    False]]],
  If[CalculusQ[u],
    False,
  Module[{w=v},Catch[
  Scan[Function[If[FalseQ[w=FunctionOfTrig[#,w,x]],Throw[False]]],u];
  w]]]]]
```

```
(* u is a function of the insert sine function of x.  If u can be put in the
form (sin (c+d*x)^j)^m*(A+B*sin (c+d*x)^k+C*sin (c+d*x)^(2*k))*(a+b*sin (c+d*x)^k)^n
where 2*m and 2*n are integers and j^2=k^2=1, RecognizedFunctionOfTrigQ[u,x]
returns True; else it returns False. *)
RecognizedFunctionOfTrigQ[u_,x_Symbol] :=
  MatchQ[u,(a_.+b_.*sin[c_.+d_.*x]^k_.)^n_. /;
    FreeQ[{a,b,c,d,n},x] && k^2===1] ||
  MatchQ[u,(A_.+B_.*sin[c_.+d_.*x]^i_.)*(a_.+b_.*sin[c_.+d_.*x]^k_.)^n_. /;
    FreeQ[{a,b,c,d,A,B,n},x] && i^2===1 && k^2===1] ||
  MatchQ[u,(A_.+C_.*sin[c_.+d_.*x]^i2_)*(a_.+b_.*sin[c_.+d_.*x]^k_.)^n_. /;
    FreeQ[{a,b,c,d,A,C,n},x] && i2^2===4 && k^2===1] ||
  MatchQ[u,(A_.+B_.*sin[c_.+d_.*x]^i_.+C_.*sin[c_.+d_.*x]^i2_)*(a_.+b_.*sin[c_.+d_.*x]^k_.)^n_. /;
    FreeQ[{a,b,c,d,A,B,C,n},x] && i^2===1 && k^2===1 && i2===2*i] ||

  MatchQ[u,(sin[c_.+d_.*x]^j_.)^m_.*(A_.+B_.*sin[c_.+d_.*x]^k_.) /;
    FreeQ[{c,d,A,B,m},x] && j^2===1 && k^2===1] ||
  MatchQ[u,(sin[c_.+d_.*x]^j_.)^m_.*(A_.+C_.*sin[c_.+d_.*x]^k2_) /;
    FreeQ[{c,d,A,C,m},x] && j^2===1 && k2^2===4] ||
  MatchQ[u,(sin[c_.+d_.*x]^j_.)^m_.*(A_.+B_.*sin[c_.+d_.*x]^k_.+C_.*sin[c_.+d_.*x]^k2_) /;
    FreeQ[{c,d,A,B,C,m},x] && j^2===1 && k^2===1 && k2===2*k] ||

  MatchQ[u,(sin[c_.+d_.*x]^j_.)^m_.*(a_.+b_.*sin[c_.+d_.*x]^k_.)^n_. /;
    FreeQ[{a,b,c,d,m,n},x] && j^2===1 && k^2===1] ||
  MatchQ[u,(sin[c_.+d_.*x]^j_.)^m_.*(A_.+B_.*sin[c_.+d_.*x]^i_.)*(a_.+b_.*sin[c_.+d_.*x]^k_.)^n_. /;
    FreeQ[{a,b,c,d,A,B,m,n},x] && i^2===1 && j^2===1 && k^2===1] ||
  MatchQ[u,(sin[c_.+d_.*x]^j_.)^m_.*(A_.+C_.*sin[c_.+d_.*x]^i2_)*(a_.+b_.*sin[c_.+d_.*x]^k_.)^n_. /;
    FreeQ[{a,b,c,d,A,C,m,n},x] && i2^2===4 && j^2===1 && k^2===1] ||
  MatchQ[u,(sin[c_.+d_.*x]^j_.)^m_.*(A_.+B_.*sin[c_.+d_.*x]^i_.+C_.*sin[c_.+d_.*x]^i2_)*(a_.+b_.*sin[
    FreeQ[{a,b,c,d,A,B,C,m,n},x] && i^2===1 && j^2===1 && k^2===1 && i2===2*i]
```

Function of hyperbolic functions of a linear binomial

```
(* If u is a function of hyperbolic trig functions of v where v is linear in x,
    FunctionOfHyperbolic[u,x] returns v; else it returns False. *)
FunctionOfHyperbolic[u_,x_Symbol] :=
  Module[{v=FunctionOfHyperbolic[u,Null,x]},
  If[v===Null, False, v]]

FunctionOfHyperbolic[u_,v_,x_] :=
  If[AtomQ[u],
    If[u===x,
      False,
    v],
  If[HyperbolicQ[u] && LinearQ[u[[1]],x],
    If[v===Null,
      u[[1]],
    Module[{a=Coefficient[v,x,0],b=Coefficient[v,x,1],
            c=Coefficient[u[[1]],x,0],d=Coefficient[u[[1]],x,1]},
    If[ZeroQ[a*d-b*c] && RationalQ[b/d],
      a/Numerator[b/d]+b*x/Numerator[b/d],
    False]]],
  If[CalculusQ[u],
    False,
  Module[{w=v},Catch[
  Scan[Function[If[FalseQ[w=FunctionOfHyperbolic[#,w,x]],Throw[False]]],u];
  w]]]]]
```

Function of expression predicate

```
(* v is a function of x.
    If u is a function of v, FunctionOfQ[v,u,x] returns True; else it returns False. *)
FunctionOfQ[v_,u_,x_Symbol,PureFlag_:False] :=
  If[FreeQ[u,x],
    False,
  If[AtomQ[v],
    True,
  If[PowerQ[v] && FreeQ[v[[2]],x] (* && NonzeroQ[v[[2]]+1] *),
    FunctionOfPowerQ[u,v[[1]],v[[2]],x],
  If[PureFlag,
    If[SinQ[v] || CscQ[v],
      PureFunctionOfSinQ[u,v[[1]],x],
    If[CosQ[v] || SecQ[v],
      PureFunctionOfCosQ[u,v[[1]],x],
    If[TanQ[v],
      PureFunctionOfTanQ[u,v[[1]],x],
    If[CotQ[v],
      PureFunctionOfCotQ[u,v[[1]],x],
    If[SinhQ[v] || CschQ[v],
      PureFunctionOfSinhQ[u,v[[1]],x],
    If[CoshQ[v] || SechQ[v],
      PureFunctionOfCoshQ[u,v[[1]],x],
    If[TanhQ[v],
      PureFunctionOfTanhQ[u,v[[1]],x],
    If[CothQ[v],
      PureFunctionOfCothQ[u,v[[1]],x],
    FunctionOfExpnQ[u,v,x]]]]]]]]],
  If[SinQ[v] || CscQ[v],
    FunctionOfSinQ[u,v[[1]],x],
  If[CosQ[v] || SecQ[v],
    FunctionOfCosQ[u,v[[1]],x],
  If[TanQ[v] || CotQ[v],
    FunctionOfTanQ[u,v[[1]],x],
  If[SinhQ[v] || CschQ[v],
    FunctionOfSinhQ[u,v[[1]],x],
  If[CoshQ[v] || SechQ[v],
    FunctionOfCoshQ[u,v[[1]],x],
  If[TanhQ[v] || CothQ[v],
    FunctionOfTanhQ[u,v[[1]],x],
  FunctionOfExpnQ[u,v,x]]]]]]]]]]]

FunctionOfExpnQ[u_,v_,x_] :=
  If[u===v,
    True,
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  Catch[Scan[Function[If[FunctionOfExpnQ[#,v,x],Null,Throw[False]]],u];True]]]]
```

```
FunctionOfPowerQ[u_,bas_,deg_,x_] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[PowerQ[u] && ZeroQ[u[[1]]-bas] && FreeQ[u[[2]],x],
    If[RationalQ[deg],
      If[RationalQ[u[[2]]],
        IntegerQ[u[[2]]/deg] && (deg>0 || u[[2]]<0),
      False],
    IntegerQ[Simplify[u[[2]]/deg]]],
  Catch[Scan[Function[If[FunctionOfPowerQ[#,bas,deg,x],Null,Throw[False]]],u];True]]]]

(* If func[w]^m is a factor of u where m is odd and w is an integer multiple of v,
    FindTrigFactor[func1,func2,u,v,True] returns the list {w,u/func[w]^n}; else it returns False. *)
(* If func[w]^m is a factor of u where m is odd and w is an integer multiple of v not equal to v,
    FindTrigFactor[func1,func2,u,v,False] returns the list {w,u/func[w]^n}; else it returns False. *)
FindTrigFactor[func1_,func2_,u_,v_,flag_] :=
  If[u===1,
    False,
  If[(Head[LeadBase[u]]===func1 || Head[LeadBase[u]]===func2) &&
        OddQ[LeadDegree[u]] &&
        IntegerQuotientQ[LeadBase[u][[1]],v] &&
        (flag || NonzeroQ[LeadBase[u][[1]]-v]),
    {LeadBase[u][[1]], RemainingFactors[u]},
  Module[{lst=FindTrigFactor[func1,func2,RemainingFactors[u],v,flag]},
  If[FalseQ[lst],
    False,
  {lst[[1]], LeadFactor[u]*lst[[2]]}]]]]
```

- **Pure function of trig functions predicates**

```
(* If u is a pure function of Sin[v] and/or Csc[v], PureFunctionOfSinQ[u,v,x] returns True;
    else it returns False. *)
PureFunctionOfSinQ[u_,v_,x_] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[TrigQ[u] && ZeroQ[u[[1]]-v],
    SinQ[u] || CscQ[u],
  Catch[Scan[Function[If[Not[PureFunctionOfSinQ[#,v,x]],Throw[False]]],u];True]]]]

(* If u is a pure function of Cos[v] and/or Sec[v], PureFunctionOfCosQ[u,v,x] returns True;
    else it returns False. *)
PureFunctionOfCosQ[u_,v_,x_] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[TrigQ[u] && ZeroQ[u[[1]]-v],
    CosQ[u] || SecQ[u],
  Catch[Scan[Function[If[Not[PureFunctionOfCosQ[#,v,x]],Throw[False]]],u];True]]]]
```

```
(* If u is a pure function of Tan[v] and/or Cot[v], PureFunctionOfTanQ[u,v,x] returns True;
    else it returns False. *)
PureFunctionOfTanQ[u_,v_,x_] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[TrigQ[u] && ZeroQ[u[[1]]-v],
    TanQ[u] || CotQ[u],
  Catch[Scan[Function[If[Not[PureFunctionOfTanQ[#,v,x]],Throw[False]]],u];True]]]]

(* If u is a pure function of Cot[v], PureFunctionOfCotQ[u,v,x] returns True;
    else it returns False. *)
PureFunctionOfCotQ[u_,v_,x_] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[TrigQ[u] && ZeroQ[u[[1]]-v],
    CotQ[u],
  Catch[Scan[Function[If[Not[PureFunctionOfCotQ[#,v,x]],Throw[False]]],u];True]]]]
```

- **Function of trig functions predicates**

```
(* If u is a function of Sin[v], FunctionOfSinQ[u,v,x] returns True; else it returns False. *)
FunctionOfSinQ[u_,v_,x_] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[TrigQ[u] && IntegerQuotientQ[u[[1]],v],
    If[OddQuotientQ[u[[1]],v],
(* Basis: If m odd, Sin[m*v]^n is a function of Sin[v]. *)
      SinQ[u] || CscQ[u],
(* Basis: If m even, Cos[m*v]^n is a function of Sin[v]. *)
    CosQ[u] || SecQ[u]],
  If[IntegerPowerQ[u] && TrigQ[u[[1]]] && IntegerQuotientQ[u[[1,1]],v],
    If[EvenQ[u[[2]]],
(* Basis: If m integer and n even, Trig[m*v]^n is a function of Sin[v]. *)
      True,
    FunctionOfSinQ[u[[1]],v,x]],
  If[ProductQ[u],
    If[CosQ[u[[1]]] && SinQ[u[[2]]] && ZeroQ[u[[1,1]]-v/2] && ZeroQ[u[[2,1]]-v/2],
      FunctionOfSinQ[Drop[u,2],v,x],
    Module[{lst},
    lst=FindTrigFactor[Sin,Csc,u,v,False];
    If[NotFalseQ[lst] && EvenQuotientQ[lst[[1]],v],
(* Basis: If m even and n odd, Sin[m*v]^n == Cos[v]*u where u is a function of Sin[v]. *)
      FunctionOfSinQ[Cos[v]*lst[[2]],v,x],
    lst=FindTrigFactor[Cos,Sec,u,v,False];
    If[NotFalseQ[lst] && OddQuotientQ[lst[[1]],v],
(* Basis: If m odd and n odd, Cos[m*v]^n == Cos[v]*u where u is a function of Sin[v]. *)
      FunctionOfSinQ[Cos[v]*lst[[2]],v,x],
    lst=FindTrigFactor[Tan,Cot,u,v,True];
    If[NotFalseQ[lst],
(* Basis: If m integer and n odd, Tan[m*v]^n == Cos[v]*u where u is a function of Sin[v]. *)
      FunctionOfSinQ[Cos[v]*lst[[2]],v,x],
    Catch[Scan[Function[If[Not[FunctionOfSinQ[#,v,x]],Throw[False]]],u];True]]]]]],
  Catch[Scan[Function[If[Not[FunctionOfSinQ[#,v,x]],Throw[False]]],u];True]]]]]]
```

```
(* If u is a function of Cos[v], FunctionOfCosQ[u,v,x] returns True; else it returns False. *)
FunctionOfCosQ[u_,v_,x_] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[TrigQ[u] && IntegerQuotientQ[u[[1]],v],
(* Basis: If m integer, Cos[m*v]^n is a function of Cos[v]. *)
    CosQ[u] || SecQ[u],
  If[IntegerPowerQ[u] && TrigQ[u[[1]]] && IntegerQuotientQ[u[[1,1]],v],
    If[EvenQ[u[[2]]],
(* Basis: If m integer and n even, Trig[m*v]^n is a function of Cos[v]. *)
      True,
    FunctionOfCosQ[u[[1]],v,x]],
  If[ProductQ[u],
    Module[{lst},
    lst=FindTrigFactor[Sin,Csc,u,v,False];
    If[NotFalseQ[lst],
(* Basis: If m integer and n odd, Sin[m*v]^n == Sin[v]*u where u is a function of Cos[v]. *)
      FunctionOfCosQ[Sin[v]*lst[[2]],v,x],
    lst=FindTrigFactor[Tan,Cot,u,v,True];
    If[NotFalseQ[lst],
(* Basis: If m integer and n odd, Tan[m*v]^n == Sin[v]*u where u is a function of Cos[v]. *)
      FunctionOfCosQ[Sin[v]*lst[[2]],v,x],
    Catch[Scan[Function[If[Not[FunctionOfCosQ[#,v,x]],Throw[False]]],u];True]]]],
  Catch[Scan[Function[If[Not[FunctionOfCosQ[#,v,x]],Throw[False]]],u];True]]]]]]

(* If u is a function of the form f[Tan[v],Cot[v]] where f is independent of x,
    FunctionOfTanQ[u,v,x] returns True; else it returns False. *)
FunctionOfTanQ[u_,v_,x_] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[TrigQ[u] && IntegerQuotientQ[u[[1]],v],
    TanQ[u] || CotQ[u] || EvenQuotientQ[u[[1]],v],
  If[PowerQ[u] && EvenQ[u[[2]]] && TrigQ[u[[1]]] && IntegerQuotientQ[u[[1,1]],v],
    True,
  If[ProductQ[u],
    Module[{lst=ReapList[Scan[Function[If[Not[FunctionOfTanQ[#,v,x]],Sow[#]]],u]]},
    If[lst==={},
      True,
    Length[lst]==2 && OddTrigPowerQ[lst[[1]],v,x] && OddTrigPowerQ[lst[[2]],v,x]]],
  Catch[Scan[Function[If[Not[FunctionOfTanQ[#,v,x]],Throw[False]]],u];True]]]]]]


OddTrigPowerQ[u_,v_,x_] :=
  If[SinQ[u] || CosQ[u] || SecQ[u] || CscQ[u],
    OddQuotientQ[u[[1]],v],
  If[PowerQ[u],
    OddQ[u[[2]]] && OddTrigPowerQ[u[[1]],v,x],
  If[ProductQ[u],
    Module[{lst=ReapList[Scan[Function[If[Not[FunctionOfTanQ[#,v,x]],Sow[#]]],u]]},
    If[lst==={},
      True,
    Length[lst]==1 && OddTrigPowerQ[lst[[1]],v,x]]],
(*If[SumQ[u],
    Catch[Scan[Function[If[Not[OddTrigPowerQ[#,v,x]],Throw[False]]],u];True], *)
  False]]]
```

```
(* u is a function of the form f[Tan[v],Cot[v]] where f is independent of x.
FunctionOfTanWeight[u,v,x] returns a nonnegative number if u is best considered a function
of Tan[v]; else it returns a negative number. *)
FunctionOfTanWeight[u_,v_,x_] :=
  If[AtomQ[u],
    0,
  If[CalculusQ[u],
    0,
  If[TrigQ[u] && IntegerQuotientQ[u[[1]],v],
    If[TanQ[u] && ZeroQ[u[[1]]-v],
      1,
    If[CotQ[u] && ZeroQ[u[[1]]-v],
      -1,
    0]],
  If[PowerQ[u] && EvenQ[u[[2]]] && TrigQ[u[[1]]] && IntegerQuotientQ[u[[1,1]],v],
    If[TanQ[u[[1]]] || CosQ[u[[1]]] || SecQ[u[[1]]],
      1,
    -1],
  If[ProductQ[u],
    If[Catch[Scan[Function[If[Not[FunctionOfTanQ[#,v,x]],Throw[False]]],u];True],
      Apply[Plus,Map[Function[FunctionOfTanWeight[#,v,x]],Apply[List,u]]],
    0],
  Apply[Plus,Map[Function[FunctionOfTanWeight[#,v,x]],Apply[List,u]]]]]]]]

(* If u (x) is equivalent to an expression of the form f (Sin[v],Cos[v],Tan[v],Cot[v],Sec[v],Csc[v])
    where f is independent of x, FunctionOfTrigQ[u,v,x] returns True; else it returns False. *)
FunctionOfTrigQ[u_,v_,x_Symbol] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[TrigQ[u] && IntegerQuotientQ[u[[1]],v],
    True,
  Catch[
    Scan[Function[If[Not[FunctionOfTrigQ[#,v,x]],Throw[False]]],u];
    True]]]]
```

- **Pure function of hyperbolic functions predicates**

```
(* If u is a pure function of Sinh[v] and/or Csch[v], PureFunctionOfSinhQ[u,v,x] returns True;
    else it returns False. *)
PureFunctionOfSinhQ[u_,v_,x_] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[HyperbolicQ[u] && ZeroQ[u[[1]]-v],
    SinhQ[u] || CschQ[u],
  Catch[Scan[Function[If[Not[PureFunctionOfSinhQ[#,v,x]],Throw[False]]],u];True]]]]

(* If u is a pure function of Cosh[v] and/or Sech[v], PureFunctionOfCoshQ[u,v,x] returns True;
    else it returns False. *)
PureFunctionOfCoshQ[u_,v_,x_] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[HyperbolicQ[u] && ZeroQ[u[[1]]-v],
    CoshQ[u] || SechQ[u],
  Catch[Scan[Function[If[Not[PureFunctionOfCoshQ[#,v,x]],Throw[False]]],u];True]]]]
```

```
(* If u is a pure function of Tanh[v] and/or Coth[v], PureFunctionOfTanhQ[u,v,x] returns True;
    else it returns False. *)
PureFunctionOfTanhQ[u_,v_,x_] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[HyperbolicQ[u] && ZeroQ[u[[1]]-v],
    TanhQ[u] || CothQ[u],
  Catch[Scan[Function[If[Not[PureFunctionOfTanhQ[#,v,x]],Throw[False]]],u];True]]]]

(* If u is a pure function of Coth[v], PureFunctionOfCothQ[u,v,x] returns True;
    else it returns False. *)
PureFunctionOfCothQ[u_,v_,x_] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[HyperbolicQ[u] && ZeroQ[u[[1]]-v],
    CothQ[u],
  Catch[Scan[Function[If[Not[PureFunctionOfCothQ[#,v,x]],Throw[False]]],u];True]]]]
```

- **Function of hyperbolic functions predicates**

```
(* If u is a function of Sinh[v], FunctionOfSinhQ[u,v,x] returns True; else it returns False. *)
FunctionOfSinhQ[u_,v_,x_] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[HyperbolicQ[u] && IntegerQuotientQ[u[[1]],v],
    If[OddQuotientQ[u[[1]],v],
(* Basis: If m odd, Sinh[m*v]^n is a function of Sinh[v]. *)
      SinhQ[u] || CschQ[u],
(* Basis: If m even, Cos[m*v]^n is a function of Sinh[v]. *)
    CoshQ[u] || SechQ[u]],
  If[IntegerPowerQ[u] && HyperbolicQ[u[[1]]] && IntegerQuotientQ[u[[1,1]],v],
    If[EvenQ[u[[2]]],
(* Basis: If m integer and n even, Hyper[m*v]^n is a function of Sinh[v]. *)
      True,
    FunctionOfSinhQ[u[[1]],v,x]],
  If[ProductQ[u],
    If[CoshQ[u[[1]]] && SinhQ[u[[2]]] && ZeroQ[u[[1,1]]-v/2] && ZeroQ[u[[2,1]]-v/2],
      FunctionOfSinhQ[Drop[u,2],v,x],
    Module[{lst},
    lst=FindTrigFactor[Sinh,Csch,u,v,False];
    If[NotFalseQ[lst] && EvenQuotientQ[lst[[1]],v],
(* Basis: If m even and n odd, Sinh[m*v]^n == Cosh[v]*u where u is a function of Sinh[v]. *)
      FunctionOfSinhQ[Cosh[v]*lst[[2]],v,x],
    lst=FindTrigFactor[Cosh,Sech,u,v,False];
    If[NotFalseQ[lst] && OddQuotientQ[lst[[1]],v],
(* Basis: If m odd and n odd, Cosh[m*v]^n == Cosh[v]*u where u is a function of Sinh[v]. *)
      FunctionOfSinhQ[Cosh[v]*lst[[2]],v,x],
    lst=FindTrigFactor[Tanh,Coth,u,v,True];
    If[NotFalseQ[lst],
(* Basis: If m integer and n odd, Tanh[m*v]^n == Cosh[v]*u where u is a function of Sinh[v]. *)
      FunctionOfSinhQ[Cosh[v]*lst[[2]],v,x],
    Catch[Scan[Function[If[Not[FunctionOfSinhQ[#,v,x]],Throw[False]]],u];True]]]]]],
  Catch[Scan[Function[If[Not[FunctionOfSinhQ[#,v,x]],Throw[False]]],u];True]]]]]]
```

```
(* If u is a function of Cosh[v], FunctionOfCoshQ[u,v,x] returns True; else it returns False. *)
FunctionOfCoshQ[u_,v_,x_] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[HyperbolicQ[u] && IntegerQuotientQ[u[[1]],v],
(* Basis: If m integer, Cosh[m*v]^n is a function of Cosh[v]. *)
    CoshQ[u] || SechQ[u],
  If[IntegerPowerQ[u] && HyperbolicQ[u[[1]]] && IntegerQuotientQ[u[[1,1]],v],
    If[EvenQ[u[[2]]],
(* Basis: If m integer and n even, Hyper[m*v]^n is a function of Cosh[v]. *)
      True,
    FunctionOfCoshQ[u[[1]],v,x]],
  If[ProductQ[u],
    Module[{lst},
    lst=FindTrigFactor[Sinh,Csch,u,v,False];
    If[NotFalseQ[lst],
(* Basis: If m integer and n odd, Sinh[m*v]^n == Sinh[v]*u where u is a function of Cosh[v]. *)
      FunctionOfCoshQ[Sinh[v]*lst[[2]],v,x],
    lst=FindTrigFactor[Tanh,Coth,u,v,True];
    If[NotFalseQ[lst],
(* Basis: If m integer and n odd, Tanh[m*v]^n == Sinh[v]*u where u is a function of Cosh[v]. *)
      FunctionOfCoshQ[Sinh[v]*lst[[2]],v,x],
    Catch[Scan[Function[If[Not[FunctionOfCoshQ[#,v,x]],Throw[False]]],u];True]]]],
  Catch[Scan[Function[If[Not[FunctionOfCoshQ[#,v,x]],Throw[False]]],u];True]]]]]]

(* If u is a function of the form f[Tanh[v],Coth[v]] where f is independent of x,
    FunctionOfTanhQ[u,v,x] returns True; else it returns False. *)
FunctionOfTanhQ[u_,v_,x_] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[HyperbolicQ[u] && IntegerQuotientQ[u[[1]],v],
    TanhQ[u] || CothQ[u] || EvenQuotientQ[u[[1]],v],
  If[PowerQ[u] && EvenQ[u[[2]]] && HyperbolicQ[u[[1]]] && IntegerQuotientQ[u[[1,1]],v],
    True,
  If[ProductQ[u],
    Module[{lst=ReapList[Scan[Function[If[Not[FunctionOfTanhQ[#,v,x]],Sow[#]]],u]]},
    If[lst==={},
      True,
    Length[lst]==2 && OddHyperbolicPowerQ[lst[[1]],v,x] && OddHyperbolicPowerQ[lst[[2]],v,x]]],
  Catch[Scan[Function[If[Not[FunctionOfTanhQ[#,v,x]],Throw[False]]],u];True]]]]]]


OddHyperbolicPowerQ[u_,v_,x_] :=
  If[SinhQ[u] || CoshQ[u] || SechQ[u] || CschQ[u],
    OddQuotientQ[u[[1]],v],
  If[PowerQ[u],
    OddQ[u[[2]]] && OddHyperbolicPowerQ[u[[1]],v,x],
  If[ProductQ[u],
    Module[{lst=ReapList[Scan[Function[If[Not[FunctionOfTanhQ[#,v,x]],Sow[#]]],u]]},
    If[lst==={},
      True,
    Length[lst]==1 && OddHyperbolicPowerQ[lst[[1]],v,x]]],
(*If[SumQ[u],
    Catch[Scan[Function[If[Not[OddHyperbolicPowerQ[#,v,x]],Throw[False]]],u];True], *)
  False]]]
```

```
(* u is a function of the form f[Tanh[v],Coth[v]] where f is independent of x.
FunctionOfTanhWeight[u,v,x] returns a nonnegative number if u is best considered a function
of Tanh[v]; else it returns a negative number. *)
FunctionOfTanhWeight[u_,v_,x_] :=
  If[AtomQ[u],
    0,
  If[CalculusQ[u],
    0,
  If[HyperbolicQ[u] && IntegerQuotientQ[u[[1]],v],
    If[TanhQ[u] && ZeroQ[u[[1]]-v],
      1,
    If[CothQ[u] && ZeroQ[u[[1]]-v],
      -1,
    0]],
  If[PowerQ[u] && EvenQ[u[[2]]] && HyperbolicQ[u[[1]]] && IntegerQuotientQ[u[[1,1]],v],
    If[TanhQ[u[[1]]] || CoshQ[u[[1]]] || SechQ[u[[1]]],
      1,
    -1],
  If[ProductQ[u],
    If[Catch[Scan[Function[If[Not[FunctionOfTanhQ[#,v,x]],Throw[False]]],u];True],
      Apply[Plus,Map[Function[FunctionOfTanhWeight[#,v,x]],Apply[List,u]]],
    0],
  Apply[Plus,Map[Function[FunctionOfTanhWeight[#,v,x]],Apply[List,u]]]]]]]]

(* If u (x) is equivalent to a function of the form f (Sinh[v],Cosh[v],Tanh[v],Coth[v],Sech[v],Csch[v]
    where f is independent of x, FunctionOfHyperbolicQ[u,v,x] returns True; else it returns False. *)
FunctionOfHyperbolicQ[u_,v_,x_Symbol] :=
  If[AtomQ[u],
    u=!=x,
  If[CalculusQ[u],
    False,
  If[HyperbolicQ[u] && IntegerQuotientQ[u[[1]],v],
    True,
  Catch[Scan[Function[If[FunctionOfHyperbolicQ[#,v,x],Null,Throw[False]]],u];True]]]]

(* If u/v is an integer, IntegerQuotientQ[u,v] returns True; else it returns False. *)
IntegerQuotientQ[u_,v_] :=
  u===v || ZeroQ[u-v] || IntegerQ[u/v]

(* If u/v is odd, OddQuotientQ[u,v] returns True; else it returns False. *)
OddQuotientQ[u_,v_] :=
  u===v || ZeroQ[u-v] || OddQ[u/v]

(* If u/v is even, EvenQuotientQ[u,v] returns True; else it returns False. *)
EvenQuotientQ[u_,v_] :=
  EvenQ[u/v]
```

■ Function of dense polynomials

```
(* If all occurrences of x in u (x) are in dense polynomials, FunctionOfDensePolynomialsQ[u,x]
    returns True; else it returns False. *)
FunctionOfDensePolynomialsQ[u_,x_Symbol] :=
  If[FreeQ[u,x],
    True,
  If[PolynomialQ[u,x],
    Length[Exponent[u,x,List]]>1,
  Catch[
  Scan[Function[If[FunctionOfDensePolynomialsQ[#,x],Null,Throw[False]]],u];
  True]]]
```

Function of logarithm

```
(* If u (x) is equivalent to an expression of the form f (Log[a*x^n]), FunctionOfLog[u,x] returns
    the list {f (x),a*x^n,n}; else it returns False. *)
FunctionOfLog[u_,x_Symbol] :=
  Module[{lst=FunctionOfLog[u,False,False,x]},
  If[FalseQ[lst] || FalseQ[lst[[2]]],
    False,
  lst]]

FunctionOfLog[u_,v_,n_,x_] :=
  If[AtomQ[u],
    If[u===x,
      False,
    {u,v,n}],
  If[CalculusQ[u],
    False,
  Module[{lst},
  If[LogQ[u] && NotFalseQ[lst=BinomialTest[u[[1]],x]] && ZeroQ[lst[[1]]],
    If[FalseQ[v] || u[[1]]===v,
      {x,u[[1]],lst[[3]]},
    False],
  lst={0,v,n};
  Catch[
    {Map[Function[lst=FunctionOfLog[#,lst[[2]],lst[[3]],x];
                 If[lst===False,Throw[False],lst[[1]]]],
            u],lst[[2]],lst[[3]]}]]]]]
```

Power of variable

```
(* If m is an integer, u is an expression of the form f[(c*x)^n] and g=GCD[m,n]>1,
   PowerVariableExpn[u,m,x] returns the list {x^(m/g)*f[(c*x)^(n/g)],g,c}; else it returns False. *)
PowerVariableExpn[u_,m_,x_Symbol] :=
  If[IntegerQ[m],
    Module[{lst=PowerVariableDegree[u,m,1,x]},
    If[FalseQ[lst],
      False,
    {x^(m/lst[[1]])*PowerVariableSubst[u,lst[[1]],x], lst[[1]], lst[[2]]}]],
  False]

PowerVariableDegree[u_,m_,c_,x_Symbol] :=
  If[FreeQ[u,x],
    {m, c},
  If[AtomQ[u] || CalculusQ[u],
    False,
  If[PowerQ[u] && FreeQ[u[[1]]/x,x],
    If[ZeroQ[m] || m===u[[2]] && c===u[[1]]/x,
      {u[[2]], u[[1]]/x},
    If[IntegerQ[u[[2]]] && IntegerQ[m] && GCD[m,u[[2]]]>1 && c===u[[1]]/x,
      {GCD[m,u[[2]]], c},
    False]],
  Catch[Module[{lst={m, c}},
  Scan[Function[lst=PowerVariableDegree[#,lst[[1]],lst[[2]],x];If[lst===False,Throw[False]]],u];
  lst]]]]]
```

```
PowerVariableSubst[u_,m_,x_Symbol] :=
  If[FreeQ[u,x] || AtomQ[u] ||CalculusQ[u],
    u,
  If[PowerQ[u] && FreeQ[u[[1]]/x,x],
    x^(u[[2]]/m),
  Map[Function[PowerVariableSubst[#,m,x]],u]]]
```

Squareroot of quadratic expression

```
(*
Euler substitution #2:
  If u is an expression of the form f (Sqrt[a+b*x+c*x^2],x), f (x,x) is a rational function, and
    PosQ[c], FunctionOfSquareRootOfQuadratic [u,x] returns the 3-element list {
        f ((a*Sqrt[c]+b*x+Sqrt[c]*x^2)/(b+2*Sqrt[c]*x),(-a+x^2)/(b+2*Sqrt[c]*x))*
          (a*Sqrt[c]+b*x+Sqrt[c]*x^2)/(b+2*Sqrt[c]*x)^2,
        Sqrt[c]*x+Sqrt[a+b*x+c*x^2], 2 };

Euler substitution #1:
  If u is an expression of the form f (Sqrt[a+b*x+c*x^2],x), f (x,x) is a rational function, and
    PosQ[a], FunctionOfSquareRootOfQuadratic [u,x] returns the two element list {
        f ((c*Sqrt[a]-b*x+Sqrt[a]*x^2)/(c-x^2),(-b+2*Sqrt[a]*x)/(c-x^2))*
          (c*Sqrt[a]-b*x+Sqrt[a]*x^2)/(c-x^2)^2,
        (-Sqrt[a]+Sqrt[a+b*x+c*x^2])/x, 1 };

Euler substitution #3:
  If u is an expression of the form f (Sqrt[a+b*x+c*x^2],x), f (x,x) is a rational function, and
    NegQ[a] and NegQ[c], FunctionOfSquareRootOfQuadratic [u,x] returns the two element list {
        -Sqrt[b^2-4*a*c]*
        f (-Sqrt[b^2-4*a*c]*x/(c-x^2),-(b*c+c*Sqrt[b^2-4*a*c]+(-b+Sqrt[b^2-4*a*c])*x^2)/(2*c*(c-x^2)))
          x/(c-x^2)^2,
        2*c*Sqrt[a+b*x+c*x^2]/(b-Sqrt[b^2-4*a*c]+2*c*x), 3 };

  else it returns False. *)

FunctionOfSquareRootOfQuadratic [u_,x_Symbol] :=
  If[MatchQ[u,x^m_.*(a_+b_.*x^n_.)^p_ /; FreeQ[{a,b,m,n,p},x]],
    False,
  Module[{tmp=FunctionOfSquareRootOfQuadratic [u,False,x]},
  If[FalseQ[tmp] || FalseQ[tmp[[1]]],
    False,
  tmp=tmp[[1]];
  Module[{a=Coefficient[tmp,x,0],b=Coefficient[tmp,x,1],c=Coefficient[tmp,x,2],sqrt,q,r},
  If[ZeroQ[a] && ZeroQ[b] || ZeroQ[b^2-4*a*c],
    False,
  If[PosQ[c],
    sqrt=Rt[c,2];
    q=a*sqrt+b*x+sqrt*x^2;
    r=b+2*sqrt*x;
    {Simplify[SquareRootOfQuadraticSubst [u,q/r,(-a+x^2)/r,x]*q/r^2],
     Simplify[sqrt*x+Sqrt[tmp]],
     2},
  If[PosQ[a],
    sqrt=Rt[a,2];
    q=c*sqrt-b*x+sqrt*x^2;
    r=c-x^2;
    {Simplify[SquareRootOfQuadraticSubst [u,q/r,(-b+2*sqrt*x)/r,x]*q/r^2],
     Simplify[(-sqrt+Sqrt[tmp])/x],
     1},
  sqrt=Rt[b^2-4*a*c,2];
  r=c-x^2;
  {Simplify[-sqrt*SquareRootOfQuadraticSubst [u,-sqrt*x/r,-(b*c+c*sqrt+(-b+sqrt)*x^2)/(2*c*r),x]*x/r^2]
   FullSimplify[2*c*Sqrt[tmp]/(b-sqrt+2*c*x)],
   3}]]]]]]]
```

```
FunctionOfSquareRootOfQuadratic[u_,v_,x_Symbol] :=
  If[AtomQ[u] || FreeQ[u,x],
    {v},
  If[PowerQ[u] && FreeQ[u[[2]],x],
    If[FractionQ[u[[2]]] && Denominator[u[[2]]]==2 && PolynomialQ[u[[1]],x] && Exponent[u[[1]],x]==2,
      If[(FalseQ[v] || u[[1]]===v),
        {u[[1]]},
      False],
    FunctionOfSquareRootOfQuadratic[u[[1]],v,x]],
  If[ProductQ[u] || SumQ[u],
    Catch[Module[{lst={v}},
    Scan[Function[lst=FunctionOfSquareRootOfQuadratic[#,lst[[1]],x];If[lst===False,Throw[False]]],u];
    lst]],
  False]]]

(* SquareRootOfQuadraticSubst[u,vv,xx,x] returns u with fractional powers replaced by vv raised
    to the power and x replaced by xx. *)
SquareRootOfQuadraticSubst[u_,vv_,xx_,x_Symbol] :=
  If[AtomQ[u] || FreeQ[u,x],
    If[u===x,
      xx,
    u],
  If[PowerQ[u] && FreeQ[u[[2]],x],
    If[FractionQ[u[[2]]] && Denominator[u[[2]]]==2 && PolynomialQ[u[[1]],x] && Exponent[u[[1]],x]==2,
      vv^Numerator[u[[2]]],
    SquareRootOfQuadraticSubst[u[[1]],vv,xx,x]^u[[2]]],
  Map[Function[SquareRootOfQuadraticSubst[#,vv,xx,x]],u]]]
```

Substitution Functions

■ Substitute for variable expression

```
RegularizeSubst[u_,x_,w_] :=
  Module[{lst=ConstantFactor[Regularize[Subst[u,x,w],x],x]},
  lst[[1]]*lst[[2]]]

(* Subst[u,v,w] returns u with all nondummy occurences of v replaced by w *)
Subst[u_,v_,w_] :=
  If[u===v,
    w,
  If[AtomQ[u],
    u,
  If[PowerQ[u],
    If[PowerQ[v] && u[[1]]===v[[1]] && SumQ[u[[2]]],
      Subst[u[[1]]^First[u[[2]]],v,w]*Subst[u[[1]]^Rest[u[[2]]],v,w],
    Subst[u[[1]],v,w]^Subst[u[[2]],v,w]],
  If[SubstQ[u] && (u[[2]]===v || FreeQ[u[[1]],v]),
    Subst[u[[1]],u[[2]],Subst[u[[3]],v,w]],
  Map[Function[Subst[#,v,w]],u]]]]] /;
AtomQ[u] || SubstQ[u] && (u[[2]]===v || FreeQ[u[[1]],v]) ||
    Not[CalculusQ[u] && Not[FreeQ[v,u[[2]]]] || MemberQ[{Pattern,Defer,Hold,HoldForm},Head[u]]]
```

Substitute for subexpressions

```
(* u is a function v.  SubstFor[v,u,x] returns f (x). *)
SubstFor[v_,u_,x_] :=
  If[AtomQ[v],
    Subst[u,v,x],
  If[PowerQ[v] && FreeQ[v[[2]],x] (* && NonzeroQ[v[[2]]+1] *),
    SubstForPower[u,v[[1]],v[[2]],x],

  If[SinQ[v],
    SubstForTrig[u,x,Sqrt[1-x^2],v[[1]],x],
  If[CosQ[v],
    SubstForTrig[u,Sqrt[1-x^2],x,v[[1]],x],
  If[TanQ[v],
    SubstForTrig[u,x/Sqrt[1+x^2],1/Sqrt[1+x^2],v[[1]],x],
  If[CotQ[v],
    SubstForTrig[u,1/Sqrt[1+x^2],x/Sqrt[1+x^2],v[[1]],x],
  If[SecQ[v],
    SubstForTrig[u,1/Sqrt[1-x^2],1/x,v[[1]],x],
  If[CscQ[v],
    SubstForTrig[u,1/x,1/Sqrt[1-x^2],v[[1]],x],

  If[SinhQ[v],
    SubstForHyperbolic[u,x,Sqrt[1+x^2],v[[1]],x],
  If[CoshQ[v],
    SubstForHyperbolic[u,Sqrt[-1+x^2],x,v[[1]],x],
  If[TanhQ[v],
    SubstForHyperbolic[u,x/Sqrt[1-x^2],1/Sqrt[1-x^2],v[[1]],x],
  If[CothQ[v],
    SubstForHyperbolic[u,1/Sqrt[-1+x^2],x/Sqrt[-1+x^2],v[[1]],x],
  If[SechQ[v],
    SubstForHyperbolic[u,1/Sqrt[-1+x^2],1/x,v[[1]],x],
  If[CschQ[v],
    SubstForHyperbolic[u,1/x,1/Sqrt[1+x^2],v[[1]],x],

  SubstForExpn[u,v,x]]]]]]]]]]]]]]]

SubstForExpn[u_,v_,x_] :=
  If[u===v,
    x,
  If[AtomQ[u],
    u,
  Map[Function[SubstForExpn[#,v,x]],u]]]

SubstForPower[u_,bas_,deg_,x_] :=
  If[AtomQ[u],
    u,
  If[PowerQ[u] && ZeroQ[u[[1]]-bas] && FreeQ[u[[2]],x] && IntegerQ[Simplify[u[[2]]/deg]]
       (* && (u[[2]]/deg>0 || FractionQ[deg]) *),
    x^(u[[2]]/deg),
  Map[Function[SubstForPower[#,bas,deg,x]],u]]]
```

```
(* u (v) is an expression of the form f (Sin[v],Cos[v],Tan[v],Cot[v],Sec[v],Csc[v]). *)
(* SubstForTrig[u,sin,cos,v,x] returns the expression f (sin,cos,sin/cos,cos/sin,1/cos,1/sin). *)
SubstForTrig[u_,sin_,cos_,v_,x_] :=
  If[AtomQ[u],
    u,
  If[TrigQ[u] && IntegerQuotientQ[u[[1]],v],
    If[u[[1]]===v || ZeroQ[u[[1]]-v],
      If[SinQ[u],
        sin,
      If[CosQ[u],
        cos,
      If[TanQ[u],
        sin/cos,
      If[CotQ[u],
        cos/sin,
      If[SecQ[u],
        1/cos,
      1/sin]]]]],
    Map[Function[SubstForTrig[#,sin,cos,v,x]],
            ReplaceAll[TrigExpand[Head[u][u[[1]]/v*x]],x->v]]],
  If[ProductQ[u] && CosQ[u[[1]]] && SinQ[u[[2]]] && ZeroQ[u[[1,1]]-v/2] && ZeroQ[u[[2,1]]-v/2],
    sin/2*SubstForTrig[Drop[u,2],sin,cos,v,x],
  Map[Function[SubstForTrig[#,sin,cos,v,x]],u]]]]

(* u (v) is an expression of the form f (Sinh[v],Cosh[v],Tanh[v],Coth[v],Sech[v],Csch[v]). *)
(* SubstForHyperbolic[u,sinh,cosh,v,x] returns the expression
        f (sinh,cosh,sinh/cosh,cosh/sinh,1/cosh,1/sinh). *)
SubstForHyperbolic[u_,sinh_,cosh_,v_,x_] :=
  If[AtomQ[u],
    u,
  If[HyperbolicQ[u] && IntegerQuotientQ[u[[1]],v],
    If[u[[1]]===v || ZeroQ[u[[1]]-v],
      If[SinhQ[u],
        sinh,
      If[CoshQ[u],
        cosh,
      If[TanhQ[u],
        sinh/cosh,
      If[CothQ[u],
        cosh/sinh,
      If[SechQ[u],
        1/cosh,
      1/sinh]]]]],
    Map[Function[SubstForHyperbolic[#,sinh,cosh,v,x]],
            ReplaceAll[TrigExpand[Head[u][u[[1]]/v*x]],x->v]]],
  If[ProductQ[u] && CoshQ[u[[1]]] && SinhQ[u[[2]]] && ZeroQ[u[[1,1]]-v/2] && ZeroQ[u[[2,1]]-v/2],
    sinh/2*SubstForHyperbolic[Drop[u,2],sinh,cosh,v,x],
  Map[Function[SubstForHyperbolic[#,sinh,cosh,v,x]],u]]]]
```

Substitute for fractional power of a linear

```
(* If u has a subexpression of the form (a+b*x)^(m/n) where m and n>1 are integers,
    SubstForFractionalPowerOfLinear[u,x] returns the list {v,n,a+b*x,1/b} where v is u
    with subexpressions of the form (a+b*x)^(m/n) replaced by x^m and x replaced
    by -a/b+x^n/b, and all times x^(n-1); else it returns False. *)
SubstForFractionalPowerOfLinear[u_,x_Symbol] :=
  Module[{lst=FractionalPowerOfLinear[u,1,False,x],n,a,b,tmp},
  If[FalseQ[lst] || FalseQ[lst[[2]]],
    False,
  n=lst[[1]];
  a=Coefficient[lst[[2]],x,0];
  b=Coefficient[lst[[2]],x,1];
  tmp=x^(n-1)*SubstForFractionalPower[u,lst[[2]],n,-a/b+x^n/b,x];
  tmp=SplitFreeFactors[Regularize[tmp,x],x];
  {tmp[[2]],n,lst[[2]],tmp[[1]]/b}]]
```

```
(* If u has a subexpression of the form (a+b*x)^(m/n),
    FractionalPowerOfLinear[u,1,False,x] returns {n,a+b*x}; else it returns False. *)
FractionalPowerOfLinear[u_,n_,v_,x_] :=
  If[AtomQ[u] || FreeQ[u,x],
    {n,v},
  If[CalculusQ[u],
    False,
  If[FractionalPowerQ[u] && LinearQ[u[[1]],x] && (FalseQ[v] || ZeroQ[u[[1]]-v]),
    {LCM[Denominator[u[[2]]],n],u[[1]]},
  Catch[Module[{lst={n,v}},
    Scan[Function[If[FalseQ[lst=FractionalPowerOfLinear[#,lst[[1]],lst[[2]],x]],Throw[False]]],u];
    lst]]]]]
```

Substitute for fractional power of quotient of linears

```
(* If u has a subexpression of the form ((a+b*x)/(c+d*x))^(m/n) where m and n>1 are integers,
    RootOfQuotientOfLinears[u,x] returns the list {v,n,(a+b*x)/(c+d*x),b*c-a*d} where v is u
    with subexpressions of the form ((a+b*x)/(c+d*x))^(m/n) replaced by x^m and x replaced
    by (-a+c*x^n)/(b-d*x^n), and all times x^(n-1)/(b-d*x^n)^2; else it returns False. *)
SubstForFractionalPowerOfQuotientOfLinears[u_,x_Symbol] :=
  Module[{lst=FractionalPowerOfQuotientOfLinears[u,1,False,x],n,a,b,c,d,tmp},
  If[FalseQ[lst] || FalseQ[lst[[2]]],
    False,
  n=lst[[1]];
  tmp=lst[[2]];
  lst=QuotientOfLinearsParts[tmp,x];
  a=lst[[1]];
  b=lst[[2]];
  c=lst[[3]];
  d=lst[[4]];
  If[ZeroQ[d],
    False,
  lst=x^(n-1)*SubstForFractionalPower[u,tmp,n,(-a+c*x^n)/(b-d*x^n),x]/(b-d*x^n)^2;
  lst=SplitFreeFactors[Regularize[lst,x],x];
  {lst[[2]],n,tmp,lst[[1]]*(b*c-a*d)}]]]
```

```
(* If the substitution x=v^(1/n) will not complicate algebraic subexpressions of u,
    SubstForFractionalPowerQ[u,v,x] returns True; else it returns False. *)
SubstForFractionalPowerQ[u_,v_,x_Symbol] :=
  If[AtomQ[u] || FreeQ[u,x],
    True,
  If[FractionalPowerQ[u],
    SubstForFractionalPowerAuxQ[u,v,x],
  Catch[Scan[Function[If[Not[SubstForFractionalPowerQ[#,v,x]],Throw[False]]],u];True]]]


SubstForFractionalPowerAuxQ[u_,v_,x_] :=
  If[AtomQ[u],
    False,
  If[FractionalPowerQ[u] && ZeroQ[u[[1]]-v],
    True,
  Catch[Scan[Function[If[SubstForFractionalPowerAuxQ[#,v,x],Throw[True]]],u];False]]]

(* If u has a subexpression of the form ((a+b*x)/(c+d*x))^(m/n),
    FractionalPowerOfQuotientOfLinears[u,1,False,x] returns {n,(a+b*x)/(c+d*x)}; else it returns False
FractionalPowerOfQuotientOfLinears[u_,n_,v_,x_] :=
  If[AtomQ[u] || FreeQ[u,x],
    {n,v},
  If[CalculusQ[u],
    False,
  If[FractionalPowerQ[u] && QuotientOfLinearsQ[u[[1]],x] && (FalseQ[v] || ZeroQ[u[[1]]-v]),
    {LCM[Denominator[u[[2]]],n],u[[1]]},
  Catch[Module[{lst={n,v}},
    Scan[Function[If[FalseQ[lst=FractionalPowerOfQuotientOfLinears[#,lst[[1]],lst[[2]],x]],Throw[False
    lst]]]]]
```

Substitute for inverse function of a linear

```
(* If u has a subexpression of the form g[a+b*x] where g is the inverse of the function h
    (i.e. h[g[x]] == x) and f[x,g[a+b*x]] equals u, SubstForInverseFunctionOfLinear[u,x] returns
    the list {f[-a/b+h[x]/b,x]*h'[x], g[a+b*x], b} *)
SubstForInverseFunctionOfLinear[u_,x_Symbol] :=
  Module[{tmp=InverseFunctionOfLinear[u,x],h,a,b},
  If[FalseQ[tmp],
    False,
  h=InverseFunction[Head[tmp]];
  a=Coefficient[tmp[[1]],x,0];
  b=Coefficient[tmp[[1]],x,1];
  {SubstForInverseFunction[u,tmp,-a/b+h[x]/b,x]*D[h[x],x], tmp, b}]]

(* If u has a subexpression of the form g[a+b*x] where g is an inverse function,
    InverseFunctionOfLinear[u,x] returns g[a+b*x]; else it returns False. *)
InverseFunctionOfLinear[u_,x_Symbol] :=
  If[AtomQ[u] || CalculusQ[u] || FreeQ[u,x],
    False,
  If[InverseFunctionQ[u] && LinearQ[u[[1]],x],
    u,
  Module[{tmp},
  Catch[
    Scan[Function[If[NotFalseQ[tmp=InverseFunctionOfLinear[#,x]],Throw[tmp]]],u];
    False]]]]
```

Substitute for inverse function of quotient of linears

```
(* If u has a subexpression of the form g[(a+b*x)/(c+d*x)] where g is the inverse of function h
   and f[x,g[(a+b*x)/(c+d*x)]] equals u, SubstForInverseFunctionOfQuotientOfLinears[u,x] returns
   the list {f[(-a+c*h[x])/(b-d*h[x]),x]*h'[x]/(b-d*h[x])^2, g[(a+b*x)/(c+d*x)], b*c-a*d} *)
SubstForInverseFunctionOfQuotientOfLinears[u_,x_Symbol] :=
  Module[{tmp=InverseFunctionOfQuotientOfLinears[u,x],h,a,b,c,d,lst},
  If[FalseQ[tmp],
    False,
  h=InverseFunction[Head[tmp]];
  lst=QuotientOfLinearsParts[tmp[[1]],x];
  a=lst[[1]];
  b=lst[[2]];
  c=lst[[3]];
  d=lst[[4]];
  {SubstForInverseFunction[u,tmp,(-a+c*h[x])/(b-d*h[x]),x]*D[h[x],x]/(b-d*h[x])^2, tmp, b*c-a*d}]]
```

```
(* If u has a subexpression of the form g[(a+b*x)/(c+d*x)] where g is an inverse function,
   InverseFunctionOfQuotientOfLinears[u,x] returns g[(a+b*x)/(c+d*x)]; else it returns False. *)
InverseFunctionOfQuotientOfLinears[u_,x_Symbol] :=
  If[AtomQ[u] || CalculusQ[u] || FreeQ[u,x],
    False,
  If[InverseFunctionQ[u] && QuotientOfLinearsQ[u[[1]],x],
    u,
  Module[{tmp},
  Catch[
    Scan[Function[If[NotFalseQ[tmp=InverseFunctionOfQuotientOfLinears[#,x]],Throw[tmp]]],u];
    False]]]]
```

Substitution for inverse functions

```
(* SubstForFractionalPower[u,v,n,w,x] returns u with subexpressions equal to v^(m/n) replaced
   by x^m and x replaced by w. *)
SubstForFractionalPower[u_,v_,n_,w_,x_Symbol] :=
  If[AtomQ[u],
    If[u===x,
      w,
    u],
  If[FractionalPowerQ[u] && ZeroQ[u[[1]]-v],
    x^(n*u[[2]]),
  Map[Function[SubstForFractionalPower[#,v,n,w,x]],u]]]
```

```
(* SubstForInverseFunction[u,v,w,x] returns u with subexpressions equal to v replaced by x
   and x replaced by w. *)
SubstForInverseFunction[u_,v_,x_Symbol] :=
(*  Module[{a=Coefficient[v[[1]],0],b=Coefficient[v[[1]],1]},
  SubstForInverseFunction[u,v,-a/b+InverseFunction[Head[v]]/b,x]] *)
  SubstForInverseFunction[u,v,
        (-Coefficient[v[[1]],x,0]+InverseFunction[Head[v]][x])/Coefficient[v[[1]],x,1],x]

SubstForInverseFunction[u_,v_,w_,x_Symbol] :=
  If[AtomQ[u],
    If[u===x,
      w,
    u],
  If[Head[u]===Head[v] && ZeroQ[u[[1]]-v[[1]]],
    x,
  Map[Function[SubstForInverseFunction[#,v,w,x]],u]]]
```

Substitute for function of an inverse linear binomial

```
(* If u is a function of an inverse linear binomial of the form f[1/(a+b*x)],
    SubstForInverseLinear[u,x] returns the list {f[x],a+b*x,b}; else it returns False. *)
SubstForInverseLinear[u_,x_Symbol] :=
  Module[{lst=FunctionOfInverseLinear[u,x],a,b},
  If[FalseQ[lst],
    False,
  a=lst[[1]];
  b=lst[[2]];
  {RegularizeSubst[u,x,-a/b+1/(b*x)],a+b*x,b}]]
```

Substitute for trig function of a linear binomial

```
(* u is a function of trig functions of a linear function of x. *)
(* SubstInertSineForTrigOfLinear[u] returns u with the trig functions replaced with the inert sin func
SubstInertSineForTrigOfLinear[u_,x_] :=
  If[AtomQ[u],
    u,
  If[TrigQ[u] && LinearQ[u[[1]],x],
    If[SinQ[u],
      sin[u[[1]]],
    If[CosQ[u],
      sin[u[[1]]+Pi/2],
    If[TanQ[u],
      sin[u[[1]]]/sin[u[[1]]+Pi/2],
    If[CotQ[u],
      sin[u[[1]]+Pi/2]/sin[u[[1]]],
    If[SecQ[u],
      sin[u[[1]]+Pi/2]^(-1),
    sin[u[[1]]]^(-1)]]]]],
  Map[Function[SubstInertSineForTrigOfLinear[#,x]],u]]]
```

Try tangent substitution predicates

```
TryTanSubst[u_,x_Symbol] :=
  FalseQ[FunctionOfLinear[u,x]] &&
  Not[MatchQ[u,r_.*(s_+t_)^n_. /; IntegerQ[n] && n>0]] &&
(*Not[MatchQ[u,Log[f_[x]^2] /; SinCosQ[f]]]  && *)
  Not[MatchQ[u,Log[v_]]]  &&
  Not[MatchQ[u,1/(a_+b_.*f_[x]^n_) /; SinCosQ[f] && IntegerQ[n] && n>2]] &&
  Not[MatchQ[u,f_[m_.*x]*g_[n_.*x] /; IntegersQ[m,n] && SinCosQ[f] && SinCosQ[g]]] &&
  Not[MatchQ[u,r_.*(a_.*s_^m_)^p_ /; FreeQ[{a,m,p},x] && Not[m===2 && (s===Sec[x] || s===Csc[x])]] &
  u===ExpnExpand[u,x]

TryPureTanSubst[u_,x_Symbol] :=
  Not[MatchQ[u,Log[v_]]] &&
  Not[MatchQ[u,f_[v_]^2 /; LinearQ[v,x]]] &&
  Not[MatchQ[u,ArcTan[a_.*Tan[v_]] /; FreeQ[a,x]]] &&
  Not[MatchQ[u,ArcTan[a_.*Cot[v_]] /; FreeQ[a,x]]] &&
  Not[MatchQ[u,ArcCot[a_.*Tan[v_]] /; FreeQ[a,x]]] &&
  Not[MatchQ[u,ArcCot[a_.*Cot[v_]] /; FreeQ[a,x]]] &&
  u===ExpnExpand[u,x]
```

- **Try hyperbolic tangent substitution predicates**

```
TryTanhSubst[u_,x_Symbol] :=
  FalseQ[FunctionOfLinear[u,x]] &&
  Not[MatchQ[u,r_.*(s_+t_)^n_. /; IntegerQ[n] && n>0]] &&
(*Not[MatchQ[u,Log[f_[x]^2] /; SinhCoshQ[f]]]  && *)
  Not[MatchQ[u,Log[v_]]]  &&
  Not[MatchQ[u,1/(a_+b_.*f_[x]^n_) /; SinhCoshQ[f] && IntegerQ[n] && n>2]] &&
  Not[MatchQ[u,f_[m_.*x]*g_[n_.*x] /; IntegersQ[m,n] && SinhCoshQ[f] && SinhCoshQ[g]]] &&
  Not[MatchQ[u,r_.*(a_.*s_^m_)^p_ /; FreeQ[{a,m,p},x] && Not[m===2 && (s===Sech[x] || s===Csch[x])]]]
  u===ExpnExpand[u,x]

TryPureTanhSubst[u_,x_Symbol] :=
  Not[MatchQ[u,Log[v_]]]  &&
  Not[MatchQ[u,ArcTanh[a_.*Tanh[v_]] /; FreeQ[a,x]]] &&
  Not[MatchQ[u,ArcTanh[a_.*Coth[v_]] /; FreeQ[a,x]]] &&
  Not[MatchQ[u,ArcCoth[a_.*Tanh[v_]] /; FreeQ[a,x]]] &&
  Not[MatchQ[u,ArcCoth[a_.*Coth[v_]] /; FreeQ[a,x]]] &&
  u===ExpnExpand[u,x]
```

Derivative Divides Function

```
(* If u is easy to differentiate wrt x and the derivative divides v wrt x, returns the quotient;
   else it returns False. *)
DerivativeDivides[u_,v_,x_Symbol] :=
  If[If[PolynomialQ[u,x], PolynomialQ[v,x] && Exponent[v,x]==Exponent[u,x]-1, EasyDQ[u,x]],
    Module[{w=Block[{ShowSteps=False}, D[u,x]]},
    If[ZeroQ[w],
      False,
    w=Simplify[v/w];
    If[FreeQ[w,x],
      w,
    False]]],
  False]
```

```
(* If u is easy to differentiate wrt x, returns True; else it returns False. *)
EasyDQ[u_,x_Symbol] :=
  If[AtomQ[u] || FreeQ[u,x] || Length[u]==0,
    True,
  If[CalculusQ[u],
    False,
  If[Length[u]==1,
    EasyDQ[u[[1]],x],
  If[RationalFunctionQ[u,x] && RationalFunctionExponents[u,x]==={1,1},
    True,
  If[ProductQ[u],
    If[FreeQ[First[u],x],
      EasyDQ[Rest[u],x],
    If[FreeQ[Rest[u],x],
      EasyDQ[First[u],x],
    False]],
  If[SumQ[u],
    EasyDQ[First[u],x] && EasyDQ[Rest[u],x],
  If[Length[u]==2,
    If[FreeQ[u[[1]],x],
      EasyDQ[u[[2]],x],
    If[FreeQ[u[[2]],x],
      EasyDQ[u[[1]],x],
    False]],
  False]]]]]]]
```

Nth Root Function

```
DownValues[Rt]={};

Rt[u_^m_,n_Integer] :=
  1/Rt[u^-m,n] /;
RationalQ[m] && m<0

Rt[v_.*u_^w_,n_Integer] :=
  Module[{m=Numerator[NumericFactor[w]]},
  Rt[v,n]*Rt[u^(w/m),n/GCD[m,n]]^(m/GCD[m,n]) /;
 m>1] /;
Not[NegativeOrZeroQ[v]]

(* Rt[u_*v_^m_,n_Integer] :=
  Rt[-u,n]/Rt[-v^-m,n] /;
RationalQ[m] && m<0 && NegativeQ[u] *)
```

```
Rt[u_,n_Integer] :=
  Map[Function[Rt[#,n]],u] /;
ProductQ[u] && OddQ[n]

Rt[u_,n_Integer] :=
  Catch[
  Do[If[PositiveQ[u[[i]]],
       Throw[Rt[u[[i]],n]*Rt[Delete[u,i],n]]],
    {i,1,Length[u]}];
  Do[If[NegativeQ[u[[i]]] && NonzeroQ[u[[i]]+1],
       Throw[Rt[-u[[i]],n]*Rt[-Delete[u,i],n]]],
    {i,1,Length[u]}];
  If[u[[1]]===-1,
    Do[If[SumQ[u[[i]]] && (NegQ[u[[i,1]]] || NegQ[u[[i,2]]]),
         Throw[Rt[Dist[-1,u[[i]]],n]*Rt[-Delete[u,i],n]]],
      {i,2,Length[u]}];
    Do[If[AtomQ[u[[i]]],
         Throw[Rt[-u[[i]],n]*Rt[-Delete[u,i],n]]],
      {i,2,Length[u]}];
    Rt[-u[[2]],n]*Rt[Drop[u,2],n],
  Do[If[Not[FreeQ[Delete[u,i],Rt[-u[[i]],n]]],
       Throw[Rt[-u[[i]],n]*Rt[-Delete[u,i],n]]],
    {i,1,Length[u]}];
  Map[Function[Rt[#,n]],u]]] /;
ProductQ[u] && EvenQ[n] && Not[u[[1]]===-1 && Length[u]==2]

(* Note: These simplification rules required because not always done by Simplify! See Warts.m
    for examples of the problem. *)

(* Basis: 1-Sin[z]^2 == Cos[z]^2 *)
Rt[u_.*(a_+b_.*Sin[v_]^2)^m_.,n_Integer] :=
  Rt[u*(a*Cos[v]^2)^m,n] /;
ZeroQ[a+b]

(* Basis: 1-Cos[z]^2 == Sin[z]^2 *)
Rt[u_.*(a_+b_.*Cos[v_]^2)^m_.,n_Integer] :=
  Rt[u*(a*Sin[v]^2)^m,n] /;
ZeroQ[a+b]

(* Basis: 1+Sinh[z]^2 == Cosh[z]^2 *)
Rt[u_.*(a_+b_.*Sinh[v_]^2)^m_.,n_Integer] :=
  Rt[u*(a*Cosh[v]^2)^m,n] /;
ZeroQ[a-b]

(* Basis: 1-Cosh[z]^2 == -Sinh[z]^2 *)
Rt[u_.*(a_+b_.*Cosh[v_]^2)^m_.,n_Integer] :=
  Rt[u*(b*Sinh[v]^2)^m,n] /;
ZeroQ[a+b]

Rt[u_,n_] :=
  -Rt[-u,n] /;
OddQ[n] && NegativeQ[u]

Rt[u_,n_Integer] :=
  Module[{v=Simplify[u]},
  If[LeafCount[Together[v]]<LeafCount[v], v=Together[v]];
  If[v=!=u,
    Rt[v,n],
  u^(1/n)]]
```

```
(* Rt[u_,n_Integer] :=
  If[AtomQ[u],
    u^(1/n),
  If[PowerQ[u],
    If[RationalQ[u[[2]]],
      If[u[[2]]<0,
        1/Rt[u[[1]]^-u[[2]],n],
      If[Numerator[u[[2]]]>1,
        Module[{gcd=GCD[Numerator[u[[2]]],n]},
        Rt[u[[1]]^(1/Denominator[u[[2]]]),n/gcd]^(Numerator[u[[2]]]/gcd)],
      u^(1/n)]],
    u^(1/n)],
  If[ProductQ[u],
    If[OddQ[n],
      Map[Function[Rt[#,n]],u],
    If[NegativeQ[First[u]],
      If[First[u]===-1,
        If[PowerQ[Rest[u]] && OddQ[Rest[u][[2]]],
          If[Rest[u][[2]]<0,
            1/Rt[(-Rest[u][[1]])^-Rest[u][[2]],n],
          Module[{gcd=GCD[Rest[u][[2]],n]},
          Rt[Rest[u][[1]],n/gcd]^(Rest[u][[2]]/gcd)]],
        u^(1/n)],
      Rt[-First[u],n]*Rt[-Rest[u],n]],
    u^(1/n)]], *)
```

IntegrateMonomialSum

```
 (* u is a monomial sum in x.  IntegrateMonomialSum[u,x] returns the antiderivative of u wrt x
    with the antiderivative of the constants terms of u collected into a single term times x. *)
IntegrateMonomialSum[u_,x_Symbol] :=
  Module[{lst=Map[Function[If[FreeQ[#,x],{#,0},{0,#*x*If[Exponent[#,x]===-1,Log[x],1/(Exponent[#,x]+1)
  lst[[1]]*x + lst[[2]]]
```