```
(* ============================================================================ *)
(* Patrick L. Nash, Ph.D.     (c) 2022, under GPL ; do not remove this notice   *)
(* Professor, UTSA Physics and Astronomy, Retired (UTSA)                        *)
(* Patrick299Nash  at    gmail   ...                                            *)
(* Enhanced Version 2 - Fixed HTML entity handling and partial derivatives      *)
(* blame: PLN and friends (Claude Opus 4.5 and Manus-Lite)                          *)
(* ============================================================================ *)

BeginPackage["ConvertMapleToMathematicaV2`"];

ConvertMapleToMathematicaV2::usage = "ConvertMapleToMathematicaV2[str] converts a Maple s
ConvertMapleToMathematicaStringV2::usage = "ConvertMapleToMathematicaStringV2[str] conver

Begin["`Private`"];

(* ============================================================================ *)
(* SECTION 0: PREPROCESSING - Handle HTML entities and derivative notation      *)
(* ============================================================================ *)

(* Parse derivative specifications like "t^2z^2" or "t^3" from string *)
ParseDerivativeSpec[spec_String] := Module[
  {result = {}, remaining = spec, varMatch, var, power},

  remaining = StringReplace[remaining, " " → ""];

  While[StringLength[remaining] > 0,
    varMatch = StringCases[remaining,
      RegularExpression["^([a-zA-Z][a-zA-Z0-9]*)\\^(\\d+)"] :> {"$1", "$2"}, 1];
    If[Length[varMatch] > 0,
      {var, power} = varMatch[[1]];
      AppendTo[result, {var, ToExpression[power]}];
      remaining = StringDrop[remaining, StringLength[var] + 1 + StringLength[power]];
      ,
      varMatch = StringCases[remaining,
        RegularExpression["^([a-zA-Z][a-zA-Z0-9]*)"] :> "$1", 1];
      If[Length[varMatch] > 0,
        var = varMatch[[1]];
        AppendTo[result, {var, 1}];
        remaining = StringDrop[remaining, StringLength[var]];
        ,
        remaining = StringDrop[remaining, 1];
      ]
    ]
  ];
  result
];
```

```
(* Format derivative spec for Mathematica D[] *)
FormatDerivSpec[derivSpec_List] :=
  StringRiffle[
    Map[If[#[[2]] == 1, #[[1]], "{" <> #[[1]] <> ", " <> ToString[#[[2]]] <> "}"]&,
        derivSpec],
    ", "
  ];

(* Main preprocessing: Convert HTML entities and derivative notation to Mathematica synta
PreprocessMapleString[str_String] := Module[
  {result, i, n, prev},

  result = str;

  (* Step 1: Replace HTML entities with unique markers *)
  result = StringReplace[result, {
    "&PartialD;" → "PARTIALD",
    "&DifferentialD;" → "DIFFD",
    "&int;" → "INTGRL"
  }];

  (* Step 1b: In derivative specs, PARTIALD is used as separator - replace with comma *)
  (* Pattern: (PARTIALDspec1PARTIALDspec2) → (PARTIALDspec1,spec2) *)
  result = StringReplace[result,
    RegularExpression["(\\(PARTIALD[a-zA-Z0-9\\^]+)PARTIALD"] :> "$1,"
  ];

  (* Step 2: Handle derivative notations iteratively *)
  (* We need to handle patterns like:
     ((PARTIALD)^n)/(PARTIALDspec) func(args) → D[func[args], derivspec]
     (PARTIALD)/(PARTIALDvar) func(args) → D[func[args], var]
     ((DIFFD)^n)/(DIFFDvar^m) func(args) → D[func[args], {var, m}]
     (DIFFD)/(DIFFDvar) func(args) → D[func[args], var]
  *)

  For[i = 1, i ≤ 50, i++,
    prev = result;

    (* Pattern 1: ((PARTIALD)^n)/(PARTIALDspec) func(args) - partial with power *)
    (* Now spec can include commas for multiple variables *)
    result = StringReplace[result,
      RegularExpression["\\(\\(PARTIALD\\)\\^(\\d+)\\)/\\(PARTIALD([a-zA-Z0-9\\^,]+)\\)
        Module[{spec = "$2", func = "$3", args = "$4", derivSpec, argsFixed},
          derivSpec = ParseDerivativeSpec[spec];
          argsFixed = StringReplace[args, "," → ", "];
          "D[" <> func <> "[" <> argsFixed <> "], " <> FormatDerivSpec[derivSpec] <> "]
```

```
      ]
  ];

  (* Pattern 2: (PARTIALD)/(PARTIALDvar) func(args) - simple partial *)
  result = StringReplace[result,
    RegularExpression["\\(PARTIALD\\)/\\(PARTIALD([a-zA-Z][a-zA-Z0-9]*)\\)\\s+([a-zA-
      Module[{var = "$1", func = "$2", args = "$3", argsFixed},
        argsFixed = StringReplace[args, "," → ", "];
        "D[" <> func <> "[" <> argsFixed <> "], " <> var <> "]"
      ]
  ];

  (* Pattern 3: ((DIFFD)^n)/(DIFFDvar^m) func(args) - ordinary deriv with power *)
  result = StringReplace[result,
    RegularExpression["\\(\\(DIFFD\\)\\^(\\d+)\\)/\\(DIFFD([a-zA-Z][a-zA-Z0-9]*)\\^(\
      Module[{var = "$2", power = "$3", func = "$4", args = "$5", argsFixed},
        argsFixed = StringReplace[args, "," → ", "];
        "D[" <> func <> "[" <> argsFixed <> "], {" <> var <> ", " <> power <> "}]"
      ]
  ];

  (* Pattern 4: (DIFFD)/(DIFFDvar) func(args) - simple ordinary deriv *)
  result = StringReplace[result,
    RegularExpression["\\(DIFFD\\)/\\(DIFFD([a-zA-Z][a-zA-Z0-9]*)\\)\\s+([a-zA-Z][a-z
      Module[{var = "$1", func = "$2", args = "$3", argsFixed},
        argsFixed = StringReplace[args, "," → ", "];
        "D[" <> func <> "[" <> argsFixed <> "], " <> var <> "]"
      ]
  ];

  If[result === prev, Break[]];
];

(* Step 3: Handle integrals - Pattern: INTGRL expr DIFFDvar → Integrate[expr, var] *)
(* The integral notation in Maple is: &int; integrand &DifferentialD;var *)
(* After HTML entity replacement: INTGRL integrand DIFFDvar *)
(* We need to handle nested integrals from innermost to outermost *)

For[i = 1, i ≤ 100, i++,
  prev = result;

  (* Use non-greedy matching to find: INTGRL integrand DIFFDvar *)
  (* The integration variable is a single letter following DIFFD *)
  result = StringReplace[result,
    RegularExpression["INTGRL(.*?)DIFFD([a-zA-Z])"] :→
      "Integrate[$1, $2]"
  ];
```

```
    If[result === prev, Break[]];
  ];

  (* Step 4: Clean up any remaining markers *)
  result = StringReplace[result, {
    "PARTIALD" → "pd",
    "DIFFD" → "dd",
    "INTGRL" → "Integrate"
  }];

  result
];

(* ============================================================================ *)
(* SECTION 1: LEXER                                                             *)
(* ============================================================================ *)

IsDigit[char_] := StringMatchQ[char, RegularExpression["[0-9]"]];
IsAlpha[char_] := StringMatchQ[char, RegularExpression["[a-zA-Z_]"]];
IsAlphaNum[char_] := StringMatchQ[char, RegularExpression["[a-zA-Z0-9_]"]];
IsSpace[char_] := StringMatchQ[char, RegularExpression["\\s"]];

mapleReservedWords = {
  "and", "assuming", "break", "by", "do", "done", "elif", "else", "end",
  "for", "from", "if", "in", "local", "mod", "next", "not", "or", "proc",
  "return", "then", "to", "while", "xor"
};

GetTokens[str_String] := Module[
  {chars, len, i, char, tokens = {}, numStr, idStr, strLit},

  chars = Characters[str];
  len = Length[chars];
  i = 1;

  While[i ≤ len,
    char = chars⟦i⟧;

    Which[
      IsSpace[char],
      i++,

      IsDigit[char],
      numStr = char;
      i++;
      While[i ≤ len && IsDigit[chars⟦i⟧],
```

```
    numStr = numStr <> chars[[i]];
    i++;
  ];
  If[i ≤ len && chars[[i]] == "." && (i+1) ≤ len && IsDigit[chars[[i+1]]],
    numStr = numStr <> ".";
    i++;
    While[i ≤ len && IsDigit[chars[[i]]],
      numStr = numStr <> chars[[i]];
      i++;
    ];
  ];
  AppendTo[tokens, {"NUMBER", ToExpression[numStr]}],

  IsAlpha[char],
  idStr = char;
  i++;
  While[i ≤ len && IsAlphaNum[chars[[i]]],
    idStr = idStr <> chars[[i]];
    i++;
  ];
  If[MemberQ[mapleReservedWords, idStr],
    AppendTo[tokens, {"KEYWORD", idStr}],
    AppendTo[tokens, {"IDENTIFIER", idStr}]
  ],

  char == "\"",
  i++;
  strLit = "";
  While[i ≤ len && chars[[i]] ≠ "\"",
    strLit = strLit <> chars[[i]];
    i++;
  ];
  If[i ≤ len, i++];
  AppendTo[tokens, {"STRING", strLit}],

  char == ":" && (i+1) ≤ len && chars[[i+1]] == "=",
  AppendTo[tokens, {"ASSIGN", ":="}];
  i += 2,

  char == "." && (i+1) ≤ len && chars[[i+1]] == ".",
  If[(i+2) ≤ len && chars[[i+2]] == ".",
    AppendTo[tokens, {"ELLIPSIS", "..."}];
    i += 3,
    AppendTo[tokens, {"RANGE", ".."}];
    i += 2
  ],

```

```
      char == "<" && (i+1) ≤ len && chars⟦i+1⟧ == "=",
      AppendTo[tokens, {"LE", "<="}];
      i += 2,

      char == ">" && (i+1) ≤ len && chars⟦i+1⟧ == "=",
      AppendTo[tokens, {"GE", ">="}];
      i += 2,

      char == "!" && (i+1) ≤ len && chars⟦i+1⟧ == "=",
      AppendTo[tokens, {"NE", "!="}];
      i += 2,

      char == "+", AppendTo[tokens, {"PLUS", "+"}]; i++,
      char == "-", AppendTo[tokens, {"MINUS", "-"}]; i++,
      char == "*", AppendTo[tokens, {"STAR", "*"}]; i++,
      char == "/", AppendTo[tokens, {"SLASH", "/"}]; i++,
      char == "^", AppendTo[tokens, {"CARET", "^"}]; i++,
      char == "(", AppendTo[tokens, {"LPAREN", "("}]; i++,
      char == ")", AppendTo[tokens, {"RPAREN", ")"}]; i++,
      char == "{", AppendTo[tokens, {"LBRACE", "{"}]; i++,
      char == "}", AppendTo[tokens, {"RBRACE", "}"}]; i++,
      char == "[", AppendTo[tokens, {"LBRACKET", "["}]; i++,
      char == "]", AppendTo[tokens, {"RBRACKET", "]"}]; i++,
      char == "=", AppendTo[tokens, {"EQUALS", "="}]; i++,
      char == "<", AppendTo[tokens, {"LT", "<"}]; i++,
      char == ">", AppendTo[tokens, {"GT", ">"}]; i++,
      char == ",", AppendTo[tokens, {"COMMA", ","}]; i++,
      char == ";", AppendTo[tokens, {"SEMICOLON", ";"}]; i++,
      char == "$", AppendTo[tokens, {"DOLLAR", "$"}]; i++,

      char == "#",
      While[i ≤ len && chars⟦i⟧ ≠ "\n", i++];
      i++,

      True,
      i++
    ];
  ];

  AppendTo[tokens, {"EOF", "EOF"}];
  tokens
];

(* ============================================================================ *)
(* SECTION 2: FUNCTION MAPPING                                                  *)
(* ============================================================================ *)
```

```
(* List of known Maple function names - identifiers NOT in this list followed by ( will b
knownMapleFunctions = {
  "abs", "ceil", "floor", "round", "trunc", "frac", "signum", "min", "max", "sqrt", "surd
  "exp", "ln", "log", "log10", "lambertw",
  "sin", "cos", "tan", "cot", "sec", "csc",
  "arcsin", "arccos", "arctan", "arccot", "arcsec", "arccsc",
  "sinh", "cosh", "tanh", "coth", "sech", "csch",
  "arcsinh", "arccosh", "arctanh", "arccoth", "arcsech", "arccsch",
  "diff", "int", "limit", "sum", "product", "series", "taylor",
  "dsolve", "desol", "pdsolve",
  "gamma", "lngamma", "psi", "beta", "binomial", "factorial", "doublefactorial", "pochham
  "besselj", "bessely", "besseli", "besselk", "hankelh1", "hankelh2", "sphericalbesselj",
  "airyai", "airybi", "airy",
  "erf", "erfc", "erfi", "dawson", "fresnelc", "fresnels",
  "elliptick", "elliptice", "ellipticf", "ellipticpi",
  "jacobisn", "jacobicn", "jacobidn", "jacobicd", "jacobisd", "jacobind", "jacobidc", "ja
  "weierstrassp", "weierstrasspprime", "weierstrasssigma", "weierstrasszeta",
  "zeta", "dilog", "polylog", "lerchphi",
  "hypergeom", "kummeru", "kummerm", "whittakerm", "whittakerw", "meijerg",
  "chebyshevt", "chebyshevu", "legendrep", "legendreq", "laguerrel", "hermiteh", "gegenba
  "fibonacci", "lucas", "bernoulli", "euler", "stirling1", "stirling2", "bell", "catalan"
  "ithprime", "isprime", "nextprime", "prevprime", "ifactor", "igcd", "ilcm", "irem", "iq
  "re", "im", "argument", "conjugate", "csgn",
  "determinant", "det", "trace", "rank", "transpose", "conjugatetranspose", "norm",
  "eigenvalues", "eigenvectors", "eigenvects", "characteristicpolynomial",
  "ludecomposition", "qrdecomposition", "singularvalues", "svd", "jordanform", "schurdeco
  "matrixexponential", "matrixlog",
  "expand", "factor", "simplify", "combine", "normal", "rationalize", "evalf", "collect",
  "solve", "fsolve", "isolve", "minimize", "maximize",
  "nops", "seq", "type",
  "fourier", "invfourier", "laplace", "invlaplace", "ztrans", "invztrans",
  "integrate", "d"
};

(* Check if a name should be treated as a function call when followed by ( *)
(* Heuristic: Known functions, or identifiers that look like user-defined functions *)
(* NOT a function: single uppercase letter, or uppercase letter followed by digits like Q
IsFunctionLikeName[name_String] := Module[{lowerName = ToLowerCase[name]},
  (* Known Maple functions are always treated as functions *)
  If[MemberQ[knownMapleFunctions, lowerName], Return[True]];

  (* Single uppercase letters like M, Q - treat as symbols, not functions *)
  If[StringMatchQ[name, RegularExpression["^[A-Z]$"]], Return[False]];

  (* Uppercase letter followed by digits like Q1, A4 - treat as symbols *)
  If[StringMatchQ[name, RegularExpression["^[A-Z][0-9]+$"]], Return[False]];
```

```
  (* All uppercase short names - treat as symbols *)
  If[StringLength[name] ≤ 2 && StringMatchQ[name, RegularExpression["^[A-Z]+$"]], Return

  (* Otherwise assume it's a function *)
  True
];

MapleToMathematicaMap[funcName_String] := Switch[ToLowerCase[funcName],
  "abs", "Abs", "ceil", "Ceiling", "floor", "Floor", "round", "Round",
  "trunc", "IntegerPart", "frac", "FractionalPart", "signum", "Sign",
  "min", "Min", "max", "Max", "sqrt", "Sqrt", "surd", "Surd",
  "exp", "Exp", "ln", "Log", "log", "Log", "log10", "Log10", "lambertw", "ProductLog",
  "sin", "Sin", "cos", "Cos", "tan", "Tan", "cot", "Cot", "sec", "Sec", "csc", "Csc",
  "arcsin", "ArcSin", "arccos", "ArcCos", "arctan", "ArcTan", "arccot", "ArcCot", "arcsec
  "sinh", "Sinh", "cosh", "Cosh", "tanh", "Tanh", "coth", "Coth", "sech", "Sech", "csch",
  "arcsinh", "ArcSinh", "arccosh", "ArcCosh", "arctanh", "ArcTanh", "arccoth", "ArcCoth",
  "diff", "D", "int", "Integrate", "Int", "Integrate", "limit", "Limit", "sum", "Sum", "p
  "dsolve", "DSolve", "desol", "DSolve", "pdsolve", "DSolve",
  "gamma", "Gamma", "lngamma", "LogGamma", "psi", "PolyGamma", "beta", "Beta", "binomial"
  "besselj", "BesselJ", "bessely", "BesselY", "besseli", "BesselI", "besselk", "BesselK",
  "airyai", "AiryAi", "airybi", "AiryBi", "airy", "AiryAi",
  "erf", "Erf", "erfc", "Erfc", "erfi", "Erfi", "dawson", "DawsonF", "fresnelc", "Fresnel
  "elliptick", "EllipticK", "elliptice", "EllipticE", "ellipticf", "EllipticF", "elliptic
  "jacobisn", "JacobiSN", "jacobicn", "JacobiCN", "jacobidn", "JacobiDN", "jacobicd", "Ja
  "weierstrassp", "WeierstrassP", "weierstrasspprime", "WeierstrassPPrime", "weierstrasss
  "zeta", "Zeta", "dilog", "PolyLog", "polylog", "PolyLog", "lerchphi", "LerchPhi",
  "hypergeom", "Hypergeometric2F1", "kummeru", "HypergeometricU", "kummerm", "Hypergeomet
  "chebyshevt", "ChebyshevT", "chebyshevu", "ChebyshevU", "legendrep", "LegendreP", "lege
  "fibonacci", "Fibonacci", "lucas", "LucasL", "bernoulli", "BernoulliB", "euler", "Euler
  "ithprime", "Prime", "isprime", "PrimeQ", "nextprime", "NextPrime", "prevprime", "NextP
  "re", "Re", "im", "Im", "argument", "Arg", "conjugate", "Conjugate", "csgn", "Sign",
  "determinant", "Det", "det", "Det", "trace", "Tr", "rank", "MatrixRank", "transpose", "
  "eigenvalues", "Eigenvalues", "eigenvectors", "Eigenvectors", "eigenvects", "Eigensyste
  "ludecomposition", "LUDecomposition", "qrdecomposition", "QRDecomposition", "singularva
  "matrixexponential", "MatrixExp", "matrixlog", "MatrixLog",
  "expand", "Expand", "factor", "Factor", "simplify", "Simplify", "combine", "Simplify",
  "solve", "Solve", "fsolve", "NSolve", "isolve", "Solve", "minimize", "Minimize", "maxim
  "nops", "Length", "seq", "Table", "type", "Head",
  "fourier", "FourierTransform", "invfourier", "InverseFourierTransform", "laplace", "Lap
  "pi", "Pi", "e", "E", "i", "I", "infinity", "Infinity",
  "d", "D",
  _, StringReplace[funcName, StartOfString ~~ x_ :> ToUpperCase[x]]
];

(* ========================================================================== *)
(* SECTION 3: PARSER                                                          *)
(* ========================================================================== *)
```

```
3
4  ParseTokens[tokens_List] := Module[
5    {pos = 1, currentToken, eat, peek, parseExpression, parseEquation,
6     parseAddExp, parseMulExp, parsePowerExp, parseUnaryExp, parsePrimary,
7     parseArgs, parseList},
8
9    currentToken := tokens[[pos]];
10   peek[] := tokens[[pos]];
11
12   eat[type_] := If[currentToken[[1]] == type,
13     pos++;
14     True,
15     False
16   ];
17
18   parseExpression[] := parseEquation[];
19
20   parseEquation[] := Module[{left, right},
21     left = parseAddExp[];
22     If[currentToken[[1]] == "EQUALS",
23       eat["EQUALS"];
24       right = parseAddExp[];
25       {"Equation", left, right},
26       left
27     ]
28   ];
29
30   parseAddExp[] := Module[{node, right, op},
31     node = parseMulExp[];
32     While[MemberQ[{"PLUS", "MINUS"}, currentToken[[1]]],
33       op = currentToken[[2]];
34       eat[currentToken[[1]]];
35       right = parseMulExp[];
36       node = {"BinaryOp", op, node, right};
37     ];
38     node
39   ];
40
41   parseMulExp[] := Module[{node, right, op},
42     node = parsePowerExp[];
43     While[True,
44       If[MemberQ[{"STAR", "SLASH"}, currentToken[[1]]],
45         op = currentToken[[2]];
46         eat[currentToken[[1]]];
47         right = parsePowerExp[];
48         node = {"BinaryOp", op, node, right},
49         If[MemberQ[{"IDENTIFIER", "NUMBER", "LPAREN", "LBRACE"}, currentToken[[1]],
```

```
        right = parsePowerExp[];
        node = {"BinaryOp", "*", node, right},
        Break[]
      ]
    ]
  ];
  node
];

parsePowerExp[] := Module[{node, right},
  node = parseUnaryExp[];
  If[currentToken[[1]] == "CARET",
    eat["CARET"];
    right = parsePowerExp[];
    {"BinaryOp", "^", node, right},
    node
  ]
];

parseUnaryExp[] := Module[{op, node},
  If[MemberQ[{"PLUS", "MINUS"}, currentToken[[1]],
    op = currentToken[[2]];
    eat[currentToken[[1]]];
    node = parseUnaryExp[];
    {"UnaryOp", op, node},
    parsePrimary[]
  ]
];

parsePrimary[] := Module[{token, node, name, args},
  token = currentToken;
  Switch[token[[1]],
    "NUMBER", eat["NUMBER"]; {"Number", token[[2]]},
    "IDENTIFIER",
    eat["IDENTIFIER"];
    name = token[[2]];
    If[currentToken[[1]] == "LPAREN" && IsFunctionLikeName[name],
      (* This is a function call *)
      eat["LPAREN"];
      args = parseArgs[];
      eat["RPAREN"];
      {"Call", name, args},
      If[currentToken[[1]] == "LBRACKET",
        (* Array/subscript access *)
        eat["LBRACKET"];
        args = parseArgs[];
        eat["RBRACKET"];
```

```
            {"Call", name, args},
            (* Just an identifier - if followed by ( it will be handled as multiplication i
            {"Identifier", name}
          ]
        ],
        "LPAREN", eat["LPAREN"]; node = parseExpression[]; eat["RPAREN"]; node,
        "LBRACKET", eat["LBRACKET"]; node = parseExpression[]; eat["RBRACKET"]; node,
        "LBRACE", parseList[],
        "ELLIPSIS", eat["ELLIPSIS"]; {"Identifier", "..."},
        "EOF", {"Error", "EOF"},
        _, eat[token[[1]]]; {"Error", token}
      ]
    ];

    parseList[] := Module[{elements},
      eat["LBRACE"];
      elements = parseArgs[];
      eat["RBRACE"];
      {"List", elements}
    ];

    parseArgs[] := Module[{args = {}, arg},
      If[currentToken[[1]] ≠ "RPAREN" && currentToken[[1]] ≠ "RBRACE" && currentToken[[1]] ≠ "RB
        arg = parseExpression[];
        AppendTo[args, arg];
        While[currentToken[[1]] == "COMMA",
          eat["COMMA"];
          arg = parseExpression[];
          AppendTo[args, arg];
        ];
      ];
      args
    ];

    parseExpression[]
  ];

  (* ============================================================================ *)
  (* SECTION 4: TRANSLATOR                                                        *)
  (* ============================================================================ *)

  ProcessASTNode[nodeType_, nodeData_, childResults_] := Module[
    {funcName, cleanFuncName, mathFunc, argsStr, paramTransform},

    Switch[nodeType,
      "Number", ToString[nodeData],
      "Identifier", If[StringLength[nodeData] > 0 && StringTake[nodeData, 1] == "_", String
```

```
    "BinaryOp", "(" <> childResults[[1]] <> " " <> nodeData <> " " <> childResults[[2]] <> '
    "UnaryOp", nodeData <> "(" <> childResults[[1]] <> ")",
    "Equation", childResults[[1]] <> " == " <> childResults[[2]],
    "List", "{" <> StringRiffle[childResults, ", "] <> "}",
    "Call",
    funcName = nodeData;
    cleanFuncName = If[StringLength[funcName] > 0 && StringTake[funcName, 1] == "_", Stri
    mathFunc = MapleToMathematicaMap[cleanFuncName];
    paramTransform = Which[
      MemberQ[{"EllipticK", "EllipticE", "EllipticF", "EllipticPi"}, mathFunc] && Length[
      ReplacePart[childResults, -1 → "(" <> childResults[[-1]] <> ")^2"],
      StringMatchQ[mathFunc, "Jacobi" ~~ __] && Length[childResults] == 2,
      {childResults[[1]], "(" <> childResults[[2]] <> ")^2"},
      mathFunc == "ArcTan" && Length[childResults] == 2,
      {childResults[[2]], childResults[[1]]},
      ToLowerCase[funcName] == "dilog",
      Prepend[childResults, "2"],
      True,
      childResults
    ];
    argsStr = StringRiffle[paramTransform, ", "];
    mathFunc <> "[" <> argsStr <> "]",
    _, "Error"
  ]
];

ToMathematicaString[ast_] := Module[
  {stack, outputStack, currentItem, node, nodeType, children, childCount,
   childResults, i, result, nodeData},

  stack = {{ast, False}};
  outputStack = {};

  While[Length[stack] > 0,
    currentItem = Last[stack];
    node = currentItem[[1]];
    nodeType = node[[1]];

    If[currentItem[[2]],
      stack = Most[stack];
      {children, nodeData} = Switch[nodeType,
        "Number", {{}, node[[2]]},
        "Identifier", {{}, node[[2]]},
        "BinaryOp", {{node[[3]], node[[4]]}, node[[2]]},
        "UnaryOp", {{node[[3]]}, node[[2]]},
        "Equation", {{node[[2]], node[[3]]}, ""},
        "List", {node[[2]], ""},
```

```
        "Call", {node[[3]], node[[2]]},
        _, {{}, ""}
      ];
      childCount = Length[children];
      childResults = {};
      If[childCount > 0,
        Do[
          PrependTo[childResults, Last[outputStack]];
          outputStack = Most[outputStack];
          , {i, 1, childCount}
        ];
      ];
      result = ProcessASTNode[nodeType, nodeData, childResults];
      AppendTo[outputStack, result];
      ,
      stack[[Length[stack]]] = {node, True};
      children = Switch[nodeType,
        "Number", {},
        "Identifier", {},
        "BinaryOp", {node[[3]], node[[4]]},
        "UnaryOp", {node[[3]]},
        "Equation", {node[[2]], node[[3]]},
        "List", node[[2]],
        "Call", node[[3]],
        _, {}
      ];
      Do[
        AppendTo[stack, {children[[i]], False}];
        , {i, Length[children], 1, -1}
      ];
    ];
  ];

  If[Length[outputStack] > 0, Last[outputStack], "Error"]
];

(* ============================================================================ *)
(* SECTION 5: PUBLIC INTERFACE                                                  *)
(* ============================================================================ *)

ConvertMapleToMathematicaStringV2[inputStr_String] := Module[{preprocessed, tokens, ast},
  preprocessed = PreprocessMapleString[inputStr];
  tokens = GetTokens[preprocessed];
  ast = ParseTokens[tokens];
  ToMathematicaString[ast]
];
```

```
ConvertMapleToMathematicaV2[inputStr_String] := Module[{mathStr},
  mathStr = ConvertMapleToMathematicaStringV2[inputStr];
  If[StringMatchQ[mathStr, "Error" ~~ ___],
    mathStr,
    ToExpression[mathStr]
  ]
];

End[];
EndPackage[];

Print["ConvertMapleToMathematicaV2 loaded successfully!  BUT, WARNING:  DO NOT USE IF YOU
```