Библиотека Разработчика
ANALYTICS C#
Версия 6.0

Руководство

Email: lrndlrnd@mail.ru
URL: www.Sergey-L-Gladkiy.narod.ru

Содержание

Введение

ANALYTICS C# – это библиотека для разработчиков. Библиотека ANALYTICS содержит специальные классы для работы с «аналитическими» выражениями в программах .NET - разбор (парсинг) выражений, вычисление значения выражений и т.д.

ПРЕИМУЩЕСТВА библиотеки ANALYTICS:

1. 100% код C#.
2. Строго структурированная иерархия классов.
3. Универсальные алгоритмы (работа с формулами любой сложности).
4. Множество предопределенных функций.
5. Простота добавления новых функций для аргументов любых типов.
6. Простота перегрузки операторов для операндов любых типов.
7. Работа с комплексными числами.
8. Работа с обыкновенными дробями.
9. Работа с 3D векторами и тензорами.
10. Работа с физическими величинами и единицами измерения.
11. Работа с индексированными данными (массивы, матрицы и данные больших размерностей).
12. Работа со специальными функциями.
13. Аналитическое вычисление производных.
14. Набор численных методов, интегрированных с символьными вычислениями.

Зависимости

Библиотека ANALYTICS C# зависит от:

1. Библиотеки NRTE (NET Run-Time Environment).

Расширения

Существуют следующие расширения библиотеки ANALYTICS C#:

1. ANALYTICS Complex (зависит от библиотеки MATHEMATICS C#).
2. ANALYTICS Fractions (зависит от библиотеки MATHEMATICS C#).
3. ANALYTICS Linear Algebra (зависит от библиотеки MATHEMATICS C#).
4. ANALYTICS Special (зависит от библиотеки MATHEMATICS C#).
5. ANALYTICS Mathphysics (зависит от библиотек MATHEMATICS и PHYSICS C#).
6. ANALYTICS Numerics (зависит от библиотеки MATHEMATICS C#).

1. Базовые концепции

Основной целью библиотеки ANALYTICS является обеспечение простого способа вычисления математических выражений. В данном разделе представлены базовые концепции, используемые в библиотеке ANALYTICS для работы с математическими выражениями и формулами.

Выражения

Математическое выражение представляет собой последовательность элементов, для которых может быть вычислено значение (результат). Простой пример выражения 'x + y' - зная значения x и y можно вычислить их сумму. Как правило, выражение содержит такие элементы, как **константы (литералы)**, **переменные**, **операторы**, **функции**. Результат вычисления выражения зависит от элементов, из которых состоит выражение. Например, если "x" и "y" вещественные числа (переменные) – результат данной суммы является вещественным числом, если одно из значений является комплексным – результатом также будет комплексное значение.

Литералы

Литерал - это неименованное постоянное значение или символ, обозначающий некоторую "стандартную" константу. Например, в выражении **'2*(x+y)'** – **'2'** это числовой литерал. Поддерживаются **вещественные** и **комплексные** литералы.

Вещественные литералы могут быть записаны в простой и экспоненциальной форме. Пример простой формы: **'100'**, **'1.23'**, **'-45.67'**[1]. Пример экспоненциальной формы: **'1.23E-3'**, **'-2.45e+5'**.

В комплексных литералах символ **'I'** используется для обозначения **мнимой единицы**. Комплексный литерал состоит из действительной части и мнимой части, разделенных символами '+' или '-'. Действительная часть это просто вещественный литерал, мнимая часть состоит из вещественного литерала и символа мнимой единицы (**Примечание**: между вещественным литералом и символом мнимой единицы нет символа умножения). Примеры комплексных литералов: **'I'**, **'-4I'**, **'2+3.2I'**, **'-2e2-4.45I'**, **'2.45+I'**, **'-1.2e-3+4.5e+2I'**.

Следующие «стандартные» константы распознаются синтаксическим анализатором: 'e' – число Эйлера, 'π', 'Pi' – число Пи, '∞' – положительная бесконечность.

Переменные

Переменная – это поименованное значения (которое может быть изменено во время выполнения программы). Имя переменной может содержать только буквенно-цифровые символы, нижние и верхние цифровые индексы и символ подчеркивания (первым символом может быть только буква)[2]. Значение переменной может быть любого типа. Тип переменной не может быть изменен во время вычисления. Значение переменной доступно через имя переменной. Таким образом, в выражении, имя переменной означает текущее значение переменной. Например, пусть дана вещественная переменная **'A'**, текущее значение которой равно **'1.0'**, тогда результат выражения 'A+1' будет равен **'2.0'**. Таким образом, изменение значений переменных позволяет вычислять одни и те же выражения для разных текущих значений.

Операторы

Синтаксически, **оператор** – это символ, обозначающий некоторую математическую операцию. Например, оператор **'+'** обозначает операцию суммирования. С функциональной точки зрения, оператор выполняет некоторое действие со значениями данных, которые называются **операндами**, и возвращает результирующее значение.

Все операторы могут быть разделены на множество категорий. Наиболее распространенными в использовании являются **унарные** и **бинарные** операторы. **Унарные операторы** имеют один операнд и могут быть **префиксными** (стоят перед операндом) **и постфиксным** (стоят после операнда). Примером унарного префиксного оператора является **оператор отрицания** (**'-x'**, **'-'** это оператор, **'x'** - операнд). Примером унарного постфиксного оператора является **оператор факториала** (**'n!'**, **'!'** это оператор, **'n'** - операнд). **Бинарные операторы** имеют два операнда, и обычно стоят между ними. Примером бинарного оператора является оператор сложения (**'x+y'**, **'+'** это оператор, **'x'** и **'y'** - операнды).

Каждый оператор имеет такие атрибуты, как **приоритет** и **ассоциативность**. Приоритет определяет порядок вычисления выражений. Операторы с более высоким приоритетом применяются к их операндам до операторов с более низким приоритетом. Например, в выражении **'x+y*z'** первой операцией будет выполнено умножение **'y'** и **'z'**, а затем полученный результат будет сложен с **'x'**, поскольку оператор **'*'** имеет более высокий приоритет, чем оператор **'+'**. Порядок вычисления может быть изменен с помощью скобок **"()"**. **Ассоциативность** определяет, как операторы с одинаковым приоритетом группируются в выражении. Рассмотрим выражение **'2^3^4'** (где оператор **'^'** обозначает степень). Результат вычисления зависит от способа трактовки выражения – **'(2^3)^4=8^4'** или **'2^(3^4)=2^81'**. **Левая** ассоциативность означает, что операторы группируются слева направо (первый случай), **правая** ассоциативность соответствует группировке справа налево (второй случай).

В библиотеке ANALYTICS определены следующие операторы:

'+' оператор сложения (бинарный);
'-' оператор разности (бинарный);
'*' оператор умножения (бинарный);
'/' оператор деления (бинарный);
'^' оператор степени (бинарный);

[1] В качестве десятичного разделителя в вещественных литералах используется разделитель для CurrentCulture.

[2] ВНИМАНИЕ: не используйте отдельный символ мнимой единицы в качестве имени переменной, он всегда будет восприниматься как константа из-за приоритета определения литералов.

‘•’ оператор точка (бинарный);
‘-’ оператор отрицания (унарный, префиксный);
‘~’ оператор тильда (унарный, префиксный);
‘!’ оператор факториал (унарный, постфиксный);
‘'’ оператор апостроф (унарный, постфиксный);
‘≡’ оператор равенства (бинарный);
‘≈’ оператор приближенного равенства (бинарный);
‘≠’ оператор неравенства (бинарный);
‘>’ оператор больше (бинарный);
‘<’ оператор меньше (бинарный);
‘≥’ оператор больше или равно (бинарный);
‘≤’ оператор меньше или равно (бинарный);
‘&’ оператор логическое и (бинарный);
‘\’ оператор логическое или (бинарный);
‘¬’ оператор логическое нет (унарный, префиксный);
‘?’ оператор знак вопроса (унарный, префиксный);
‘#’ оператор число (унарный, префиксный);
Также определены следующие ‘Специальные’[3] операторы:
‘←’ оператор стрелка влево (бинарный);
‘→’ оператор стрелка вправо (бинарный);
‘↑’ оператор стрелка вверх (бинарный);
‘↓’ оператор стрелка вниз (бинарный);
‘↔’ оператор стрелка влево-вправо (бинарный);
‘↕’ оператор стрелка вверх-вниз (бинарный);
‘∂’ оператор производная (унарный, префиксный);
‘∫’ оператор интеграл (унарный, префиксный);
‘∆’ оператор дельта (унарный, префиксный);
‘∑’ оператор сумма (унарный, префиксный);
‘∏’ оператор произведение (унарный, префиксный).

Общие правила для всех операторов:
- Бинарные алгебраические операторы (+, -, * и др.) имеют приоритет, соответствующий стандартным математическим правилам.
- Операторы отношения имеют меньший приоритет, чем алгебраические.
- Логические (бинарные) операторы имеют меньший приоритет, чем операторы отношения.
- Операторы стрелка (бинарные) имеют больший приоритет, чем оператор степени.
- Все бинарные операторы являются левоассоциативными.
- Унарные операторы имеют более высокий приоритет, чем бинарные.
- Постфиксные операторы имеют более высокий приоритет, чем префиксные операторы.

Для операндов вещественного типа реализованы все стандартные математические операторы. Операторы для операндов других типов (комплексных чисел, 3D векторов и тензоров и т.д.) реализованы в расширениях библиотеки ANALYTICS (см. выше).

Все предопределенные операторы могут быть перегружены (определены) для любых других типов операндов. При выполнении перегрузки операторов имеются следующие ограничения:
- новые операторы не могут быть определены;
- число операндов и атрибуты операторов (приоритет и ассоциативность) не могут быть изменены.

Функции

С синтаксической точки зрения функция может рассматриваться как имя операции с некоторыми значениями данных - аргументов. Примером является функция синуса **‘sin(x)’**, где ‘sin’ – это имя функции, а ‘x’ – это аргумент. Имя функции определяет, какая операция выполняет с аргументами. Имя функции может содержать только буквенно-цифровые символы, нижние и верхние цифровые индексы и символ подчеркивания (первым символом может быть только буква)[4].

В дополнение к аргументам, функция может иметь параметры. Например, функция **$\log_b$(a)** **(**логарифм **‘a’** по основанию **‘b’**) имеет один аргумент **‘a’** и один параметр **‘b’**. Семантически, параметры имеют тот же смысл, что и аргументы - значения данных, над которыми выполняется операция. Синтаксически (в библиотеке ANALYTICS) параметры заключены в фигурные скобки '{}'.

Общие правила для функций:
- аргументы функции всегда заключаются в скобки "()";
- параметры функции всегда заключаются в фигурные скобки **‘{}’**;
- если функция не имеет параметров, скобки параметров не являются обязательными, скобки аргументов всегда обязательны;
- Функция может иметь множество параметров и/или аргументов;
- аргументы и параметры функции разделяются пробелами ‘ ’;
- может существовать множество функций с одинаковыми именами и различным числом и/или типом аргументов и/или параметров.

Некоторые примеры синтаксически правильных выражений функций:
max(a b) – функция максимум из двух значений (два аргумента);
random() – функция генерации случайного числа (без аргументов);
log{b}(a) – логарифм от **‘a’** по основанию **‘b’** (один параметр **‘b’**, один аргумент **‘a’**);
P{n m}(x) – присоединенный полином Лежандра (два параметра **'n'** и **'m'** и один аргумент **'x'**).

[3] ‘Специальные’ операторы не предопределены для стандартных типов данных (вещественных, комплексных и др.) и могут быть использованы для выполнения специальных операций со специфическим данными программы. Тем не менее, синтаксические правила для этих операторов предопределены и не могут быть изменены.
[4] Отдельный символ мнимой единицы может быть использован в качестве имени функции, поскольку синтаксис функции всегда может быть определен по скобкам ().

Библиотека ANALYTICS содержит множество предопределенных функций вещественных аргументов. Функции для других типов аргументов реализованы в расширениях библиотеки ANALYTICS (см. выше).

Индексация

Библиотека ANALYTICS позволяет использовать **индексацию** в выражениях. Индексация это метод доступа к элементам структурированным данных. Примером структурированных данных является массив. Элемент каждого массива имеет уникальный индекс, по которому осуществляется доступ к значению элемента. По правилам синтаксиса индексы данных записываются в квадратных скобках "[]". Например, i-й элемент массива можно записать как **'A[i]'**. Для нескольких индексов, каждый индекс должны быть записан в отдельных скобках. Например, элемент матрицы может быть записан как **'M[i][j]'**. Такой синтаксис позволяет реализовать получение 'сечений массива' - часть данных, полученных при фиксированном индексе (или нескольких индексах). Получение сечений реализуется путем 'опускания' индексов. Пусть имеется матрица (двумерный массив) **'M'**. Тогда **'M[i][]'** будет обозначать i-ую строку, а **'M[][j]'** - j-ый столбец. Последние индексы могут быть опущены вместе с их скобками. То есть, i-ая строка матрицы может быть записана как **'M[i]'** (что эквивалентно **'M[i][]'** из предыдущего примера).

Общие правила для индексации:

- индексация может быть применена только к переменным, переменные должны реализовывать специальный интерфейс индексации (см. ниже);
- допускается использование множества индексов, каждый индекс должен быть заключен в отдельные квадратные скобки **'[]'**;
- выражения, используемые в качестве индексов должны возвращать значения только вещественного типа, результирующие значения выражений должны быть целыми (приближенно);
- некоторые индексы могут быть опущены для получения 'сечений', данная процедура может быть применена только к переменным, которые реализуют специальный интерфейс (см. ниже);

Библиотека ANALYTICS содержит предопределенные переменные типа массив. Переменная типа массив может содержать данные любого типа. В переменных типа массив реализована индексация и получение сечений для массивов до трех измерений включительно.

Сводка основных правил синтаксиса

- Любое выражение может содержать литералы, переменные, операторы, функции и индексацию.
- Литералами могут быть вещественные числа (в простой и экспоненциальной формах) и комплексные числа.
- Имена переменных могут содержать только буквенно-цифровые символы и символ подчеркивания.
- Могут быть использованы только предопределенные (см. выше) операторы, новые операторы не могут быть определены.
- Операции выполняются в порядке приоритета операторов. Порядок операций можно быть изменен с помощью скобок "()".
- Функции имеют параметры и аргументы. Параметры заключаются в фигурные скобки '{}', аргументы заключены в скобках "()". Параметры и аргументы разделяются пробелом ' '.
- Индексация может быть применена только к переменным (у которых реализован специальный интерфейс). Индексы заключены в квадратные скобки '[]' (каждый индекс в отдельных скобках). Получение сечений индексированных данных может быть реализовано путем 'опускания' некоторых индексов.

Некоторые примеры синтаксически правильных формул:

```
'sin(b)^2+cos(b)^2'                          - формула Пифагора (осн. триг.  тождество).
'sin(a)*cos(b)+cos(a)*sin(b)'                - формула синус суммы углов.
'log{c}(a)+log{c}(b)'                        - формула логарифма произведения.
'A[n]*r^n*sin(n*a)+B[n]*r^-n*cos(n*a)'       - решение уравнения Лапласа в полярных координатах.
'A[n]*sinh(n*Pi/L*(x-a))*sin(n*Pi/L*(y-b))'  - решение уравнения Лапласа в декартовой системе.
```

2.Иерархия классов

Данный раздел содержит описание иерархии базовых классов библиотеки ANALYTICS. Знание внутренней иерархии классов не является необходимым при использовании библиотеки для простых аналитических вычислений. Знание иерархии необходимо только для расширения библиотеки (введения новых функций, перегрузки операторов и т.д.).

Основная концепция библиотеки ANALYTICS - легкая внешняя расширяемость. То есть, библиотека имеет завершенное ядро, которое не изменяется, а новая функциональность может быть реализована подключением внешних библиотек. Для реализации этой концепции, иерархия классов строго структурирована.

Классы переменных

Переменная - это имя и ассоциированное с ним значение (некоторого типа). Для реализации данной концепции в программе предназначен базовый абстрактный класс ***Variable***. Основными свойствами класса являются: ***Name***, ***Value*** и ***ValueType***.

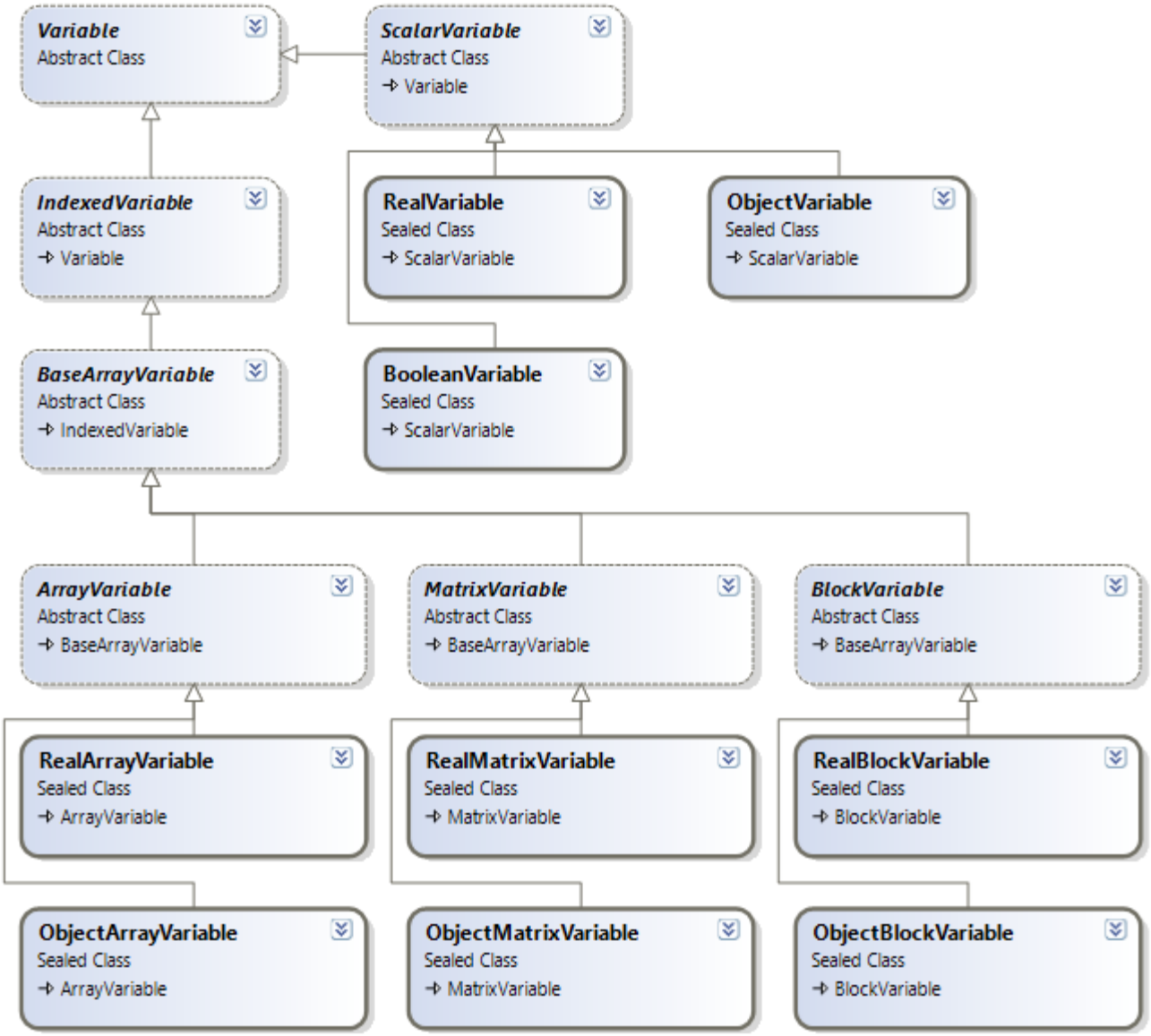


Рис. 2.1. Диаграмма иерархии классов переменных.

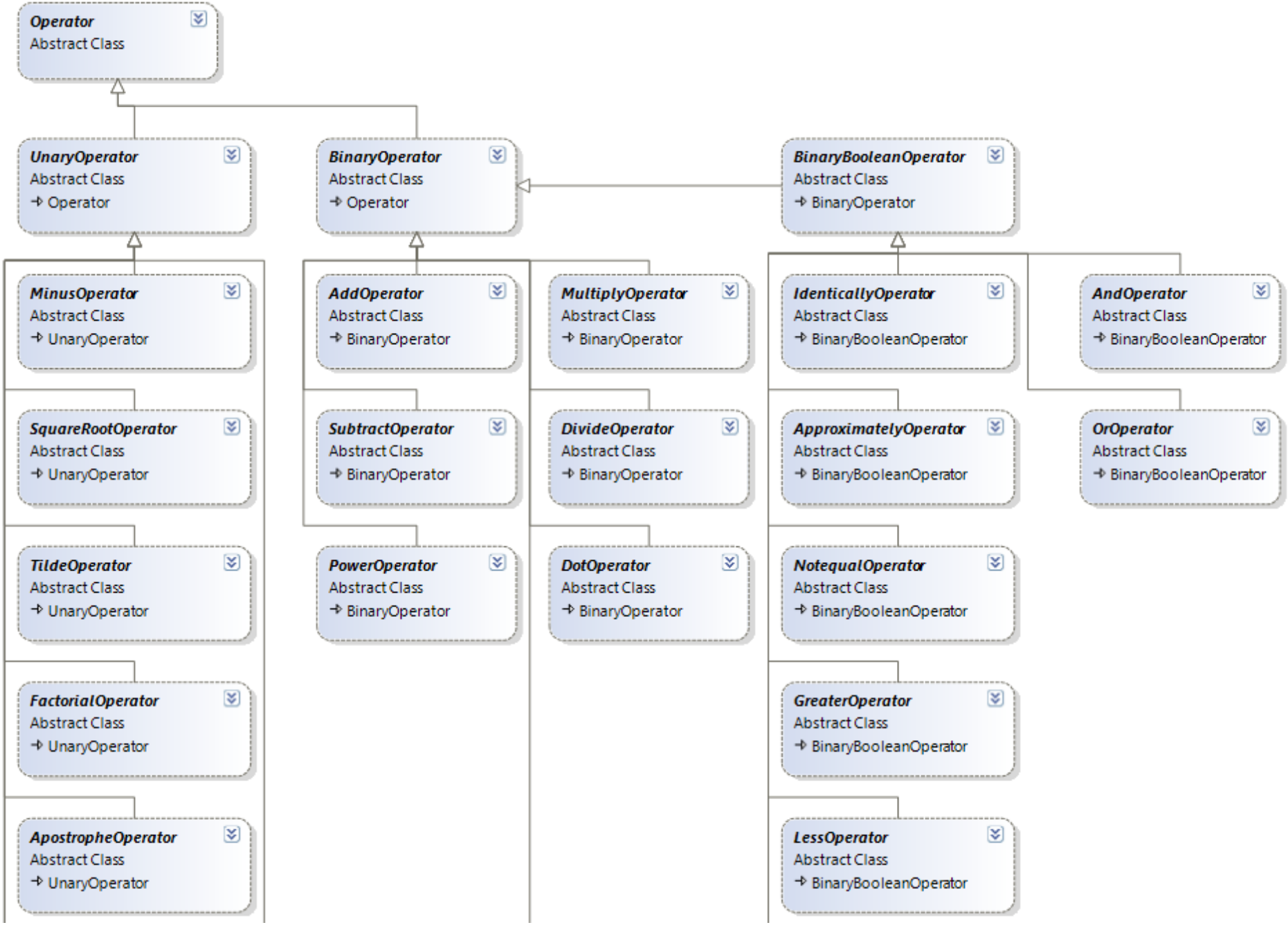
Прямыми наследниками класса ***Variable*** являются два других абстрактных класса. Первый класс - ***ScalarVariable*** содержит одно значение. Второй класс - ***IndexedVariable*** представляет собой интерфейс для реализации индексированных данных, содержащих другие значения, доступ к которым осуществляется по индексам. Следующие классы, унаследованы от ***ScalarVariable***: ***RealVariable*** (содержит одно вещественное значение), ***BooleanVariable*** (содержит одно логическое значение), ***ObjectVariable*** (содержит одно значение любого типа). Класс ***BaseArrayVariable*** наследуется от ***IndexedVariable*** и является абстрактным классом для переменных, содержащих массивы (любой размерности). Три абстрактных класса ***ArrayVariable***, ***MatrixVariable*** и ***BlockVariable*** реализуют интерфейсы для одно-, двух- и трехмерных массивов соответственно. И, наконец, конкретные классы – наследники данных классов, реализуют вещественные и объектные массивы до трех измерений включительно.

Все конкретные классы, представленные в этой иерархии, являются полностью функциональными, то есть реализуют всю функциональность для использования их в вычислениях. Все переменные типа массив реализуют индексацию и получение ‘сечений’.

Классы переменных для данных других типов (комплексные, 3D векторы и тензоры и др.) реализованы в расширениях библиотеки. Они могут быть использованы таким же образом, как и переменные ядра библиотеки ANALYTICS.

Классы операторов

Базовый абстрактный класс **Operator** предназначен для реализации концепции оператора (символ, обозначающий некоторую операцию со значениями операндов) внутри программы. Этот класс имеет свойства и функции для определения символа оператора, типа результата операции и метод вычисления значения. В соответствии с типами операторов, от базового класса унаследованы два абстрактных класса: ***UnaryOperator*** и ***BinaryOperator***. Эти классы вводят интерфейс для определения типа (типов) операнда. Более специфические классы, унаследованные от последних, определяют конкретный тип и символ оператора ('+', '^' и др.).



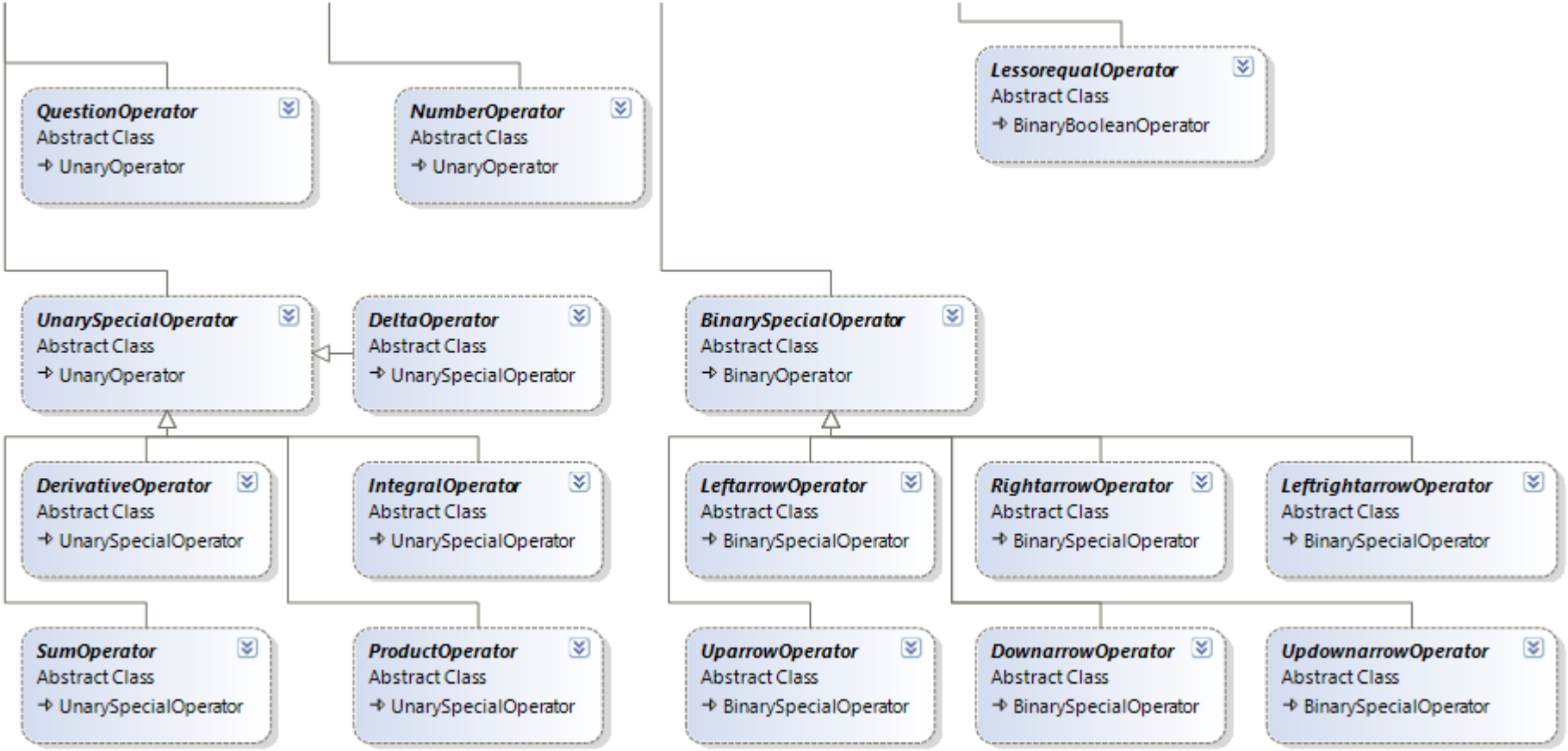


Рис. 2.2. Диаграмма иерархии классов операторов.

Конкретные классы, реализующие всю функциональность (определяющие типы операндов и выполняющие операций со значениями операнда) не представлены на данной диаграмме. Все предопределенные операторы реализованы для вещественных типов операндов. Операторы для других типов операндов (комплексных, 3D векторов и тензоров и др.) реализованы в расширениях библиотеки ANALYTICS.

Существуют также **обобщенные (generic)** аналоги для всех классов операторов. Например, обобщенным аналогом для ***AddOperator*** является класс ***GenericAddOperator***. Обобщенные классы могут быть также использованы для реализации конкретных классов операторов. Обобщенная форма классов упрощает наследование, поскольку уже содержит реализацию некоторых методов (см. ниже перегрузку операторов).

Классы функций

Базовый абстрактный класс ***Function*** реализует концепцию функции. Класс имеет интерфейс для определения имени функции, количества и типа параметров и аргументов, типа возвращаемого значения и метод, выполняющий операцию со значениями данных. Класс ***MonotypeFunction*** является прямым наследником класса ***Function*** и реализует такую функцию, параметры и аргументы которой одного типа, как и тип возвращаемого значения. На следующем уровне иерархии все функции делятся на элементарные и специальные (классы ***BaseElementaryFunction***, ***BaseSpecialFunction*** соответственно). Это деление реализует чисто математическую концепцию и ни как не связано с удобством программирования. Далее, каждый из классов делится на подклассы действительных и комплексных функций (функции с вещественным и комплексным аргументами соответственно). И, наконец, все классы делятся на простые и параметрические функции. Простые функции имеют один аргумент, параметрические - один параметр и один аргумент.

Конкретные классы, реализующие вычисления, не представлены на диаграмме. Ядро библиотеки содержит множество реализованных элементарных функций вещественных аргументов - алгебраические, тригонометрические, обратные тригонометрические, степенные, логарифмические, гиперболические, обратные гиперболические. Функции для других типов аргументов (комплексных, 3D векторов и тензоров и др.) реализованы в расширениях библиотеки ANALYTICS (см. выше).

Рис. 2.3. Диаграмма иерархии классов функций.

Классы выражений

Классы выражений предназначены для представления данных внутри основных алгоритмов библиотеки ANALYTICS. Базовым абстрактным классом для всех выражений является класс ***BaseExpression***.

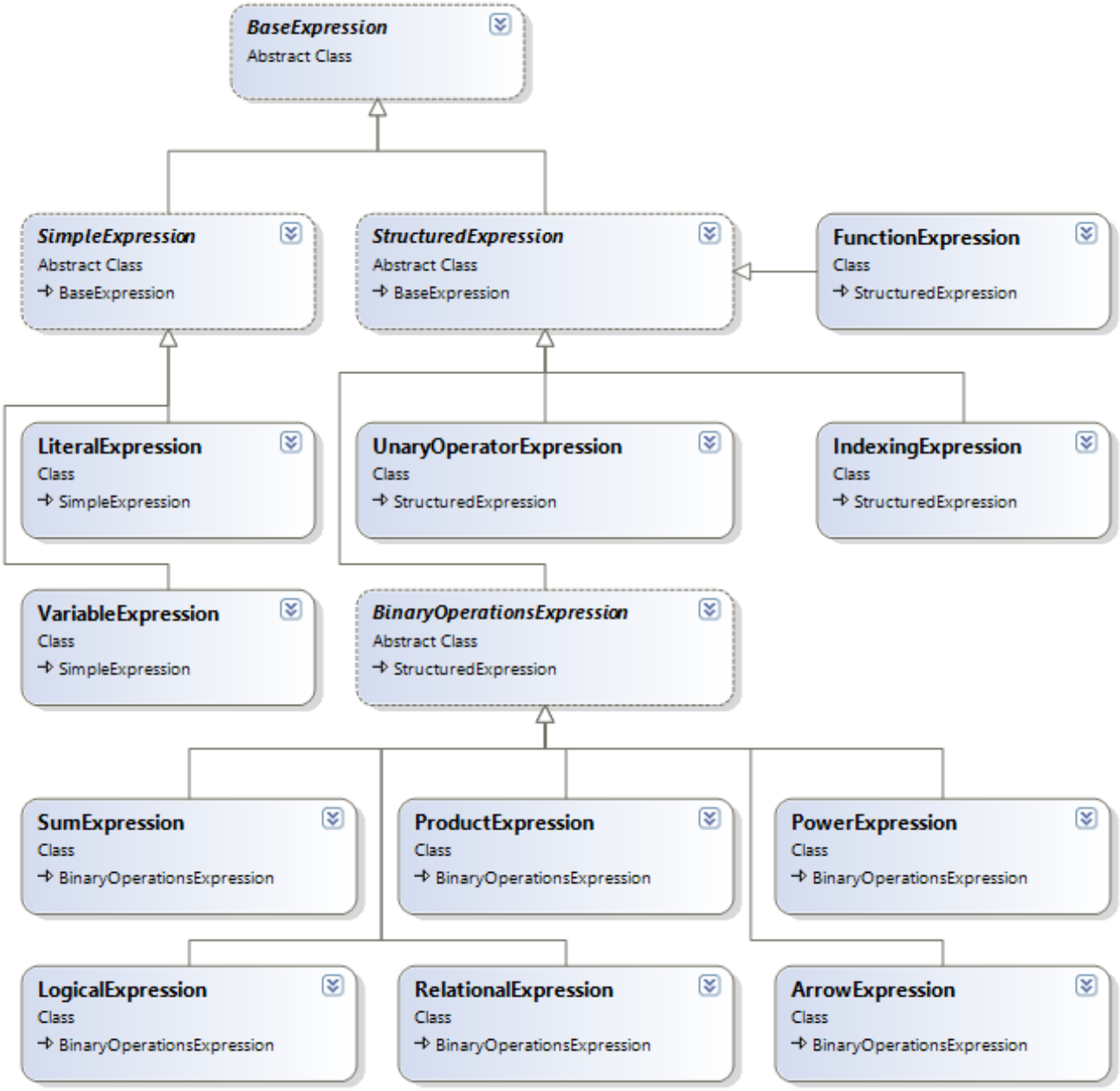


Рис. 2.4. Диаграмма иерархии классов выражений.

Этот класс определяет абстрактный интерфейс для всех выражений и реализует некоторые статические методы для манипуляции выражениями (построение выражений из строки, упрощение выражений и др.). Все остальные выражения делятся на **простые** (не содержат других выражений) и **структурированные** (содержат другие выражения). Простыми выражениями являются ***LiteralExpression*** и ***VariableExpression***. Структурированные выражения включают в себя ***FunctionExpression***, ***IndexingExpression***, ***UnaryOperatorExpression*** и ***BinaryOperationsExpression*** (***LogicalExpression***, ***RelationalExpression***, ***SumExpression***, ***ProductExpression***, ***PowerExpression***, ***ArrowExpression***). Все классы выражений реализуют методы для манипуляции ими: создание выражения из строки, упрощение выражения, реконструкция выражения (конвертация выражения в строку), создание новых выражений из существующих и др.

Понимание функциональности всех классов выражений не является обязательным для использования библиотеки ANALYTICS. Знание иерархии классов необходимо только в случае расширения функциональности библиотеки при вычислении производной выражений в аналитическом виде (см. ниже).

3. Работа с ANALYTICS

Основная функциональность библиотеки ANALYTICS (вычисление математических выражений) реализуется в классе ***Translator***. Класс ***Translator*** инкапсулирует в себе набор переменных и набор операций (зарегистрированных операторов и функций). Таким образом, класс ***Translator*** может осуществлять разбор (парсинг) строковых выражений и вычислять их значения, для текущих значений переменных.

Translator не является статическим классом, поскольку различные трансляторы могут иметь разную функциональность и наборы переменных. Таким образом, для использования функциональных возможностей ***Translator***, должен быть создан экземпляр класса. Далее предполагается, что экземпляр класса ***Translator*** создан:

```
Translator translator = new Translator();
```

Работа с переменными

Класс ***Translator*** полностью реализует интерфейс для добавления и удаления переменных, а так же изменения их значений. Для добавления переменных используется метод ***Add***. Данный метод имеет несколько вариантов для добавления переменных различных типов. Следующие примеры кода демонстрируют процесс добавления переменных:

- добавление вещественной переменной

```
string name = "a";
double v = 1.0;
translator.Add(name, v);
```

- добавление комплексной переменной

```
string name = "x";
Complex z = new Complex(1.0, -2.0);
Variable v = new ComplexVariable(name, z);
translator.Add(v);
```

- добавление переменной ‘вещественный массив’

```
string name = "A";
double[] a = new double[] { 0.0, 1.0, 2.0, 3.0 };
Variable v = new RealArrayVariable(name, a);
translator.Add(v);
```

- добавление переменной любого типа

```
string name = "A";
Complex[] a = new Complex[] { new Complex(0,0), new Complex(1,-1), new Complex(-2,2) };
Variable v = new ComplexArrayVariable(name, a);
translator.Add(v);
```

Существует два способа изменения значения переменной.

1. Если доступна прямая ссылка на переменную, значение может быть изменено с помощью свойства ***Value*** этой переменной.

```
string name = "a";
double a = 1.0;
Variable v = new RealVariable(name, a);
translator.Add(v);
// some code here...
v.Value = (double)2.0;
```

2. Если прямая ссылка на переменную отсутствует, ее можно получить с помощью метода **Get** класса **Translator** (может быть использовано как имя переменной, так и ее индекс).

```
Variable x = translator.Get("x");
x.Value = (double)3.0;
// some code here...
Variable y = translator.Get(1);
y.Value = (double)4.0;
```

ПРИМЕЧАНИЕ (об изменении значений переменных): тип значения переменной не может быть изменен. Несмотря на то, что свойство ***Value*** класса ***Variable*** имеет тип **Object**, при установке значения переменной ее тип должен быть таким же, как текущее значение **ValueType**. Тип значения определяется при создании переменной и не может быть изменен в процессе использования переменной.

Метод ***Delete*** класса **Translator** может быть использован для удаления переменных. Переменная может быть удалена по имени или индексу в текущей таблице переменных объекта ***Translator***. Метод ***DeleteAll*** удаляет все переменные. Следующий код демонстрирует процесс удаления переменных.

```
translator.Delete("x");
// some code here...
translator.Delete(2);
// some code here...
translator.DeleteAll();
```


Вычисление значений формул

В наиболее простом случае, для вычисления значения формулы может использоваться метод ***Calculate*** класса ***Translator***. Следующий код демонстрирует использование данного метода

```
string f1 = "sin(a)^2+cos(a)^2";
double r = (double)translator.Calculate(f1);
// some code here...
string f2 = "2*exp(z)-I*sin(z/3)";
Complex c = (Complex)translator.Calculate(f2);
```

В приведенном выше коде предполагается, что существуют переменные "a" и "z" в списке переменных транслятора, причем "a" - вещественного типа, а "z" является комплексной переменной. Следует заметить, что типом возвращаемого методом **Calculate** значения является ***Object, то есть*** метод может возвращать значение любого типа в зависимости от содержания формулы. Таким образом, возвращаемое значение должно быть непосредственно приведено к действительному типу, причем тип должен быть заранее известен.

Метод ***Calculate*** является довольно медленным, поскольку он анализирует строковое выражение и создает внутреннее представление формулы в виде дерева объектов, чтобы вычислить значение результата. Методы разбора не оптимизированы по скорости, они оптимизированы в смысле строгого объектно-ориентированного структурирования и удобного расширения функциональности. Таким образом, использование метода ***Calculate*** рекомендуется для вычисления отдельных формул.

Другой случай – вычисление одной и той же формулы при различных значениях переменной (-ых). Данное вычисление может быть выполнено следующим образом:

- создания объекта класса ***Formula*** из строкового выражения;
- изменение значений переменных;
- вычисление значения Формулы для текущих значений переменных.

Класс ***Formula*** предназначен для внутреннего представления в программе математических выражений. Следующий код демонстрирует вычисление таблицы значений функции для различных значений аргументов по приведенному выше алгоритму.

```
string s = "2*(sin(x)+cos(x))";
Formula f = translator.BuildFormula(s); // разбор строкового выражения
Variable x = translator.Get("x");
double v = 0.0;
double[] ax = new double[101];
double[] ay = new double[101];
for (int i = 0; i <= 100; i++)
{
        ax[i] = v;
        x.Value = v; // присвоение нового значения переменной
        ay[i] = (double)f.Calculate(); // вычисление формулы для данного значения x
        v += 0.01;
}
```

В приведенном коде, разбор (парсинг) строкового выражения выполнен только один раз при помощи функции **BuildFormula**, которая возвращает объект класса ***Formula***. Когда экземпляр класса ***Formula*** создан, значение формулы может быть вычислено несколько раз для различных значений переменной 'x'.

Проверка синтаксиса выражений

Во всех приведенных выше примерах кода предполагалось, что строковые выражения синтаксически корректны. В приложениях, конечно, возникает необходимость проверять выражения, вводимые пользователем, на синтаксическую корректность. Методы проверки правильности выражений реализованы в классе ***Translator***.

Правильность синтаксиса должна быть проверена в три этапа. Первый - проверка синтаксических правил, которые не требуется разбора. Например, скобки в математическом выражении должна быть парными и это может быть проверено без разложения выражения на составляющие. Такие синтаксические правила проверяет метод ***CheckSyntax*** класса ***Translator*** (он должен использоваться перед любым вычислением). Данный метод возвращает логическое значение ‘истина’, если все правила выполняются, в противном случае - генерирует исключение.

Второй этап состоит в проверке того, что строковое выражение может быть разложено на известные типы выражений. Например, строку ‘sin(x+1)’ можно предствить как функцию с именем ‘sin’ и одним аргументом 'x+1', который так же является известным выражением - сумма постоянной '1' и переменной с именем 'x'. На данном этапе проверки не имеет значения, определена ли функция 'sin' и существует ли переменная "x".

Третий шаг заключается в проверке того, что все элементы в разложении выражения определены. Это означает, что должны существовать все переменные. Кроме того, на данном шаге проверяется соответствие типов выражений и операций, которые должны быть выполнены. Например, в данном выражении, если переменная 'x' является вещественной, должен быть определен оператор '+' для вещественных операндов, а так же функция «sin» с одним вещественным аргументом.

Второй и третий шаги проверки синтаксиса реализованы в методе **BuildFormula** класса ***Translator***. Этот метод возвращает объект типа ***Formula***, если строковое выражение является правильным, и генерирует исключение в противном случае.

В следующем примере приведен код общего алгоритма проверки синтаксиса:

```
string s = "2*(sin(x)+cos(x))";
try
{
```

```
            // первый этап проверки синтаксиса
            if (translator.CheckSyntax(s))
            {
                    // второй и третий этапы проверки синтаксиса
                    Formula f = translator.BuildFormula(s);
                    if (f != null)
                    {
                            // код вычисления формулы,
                            // используя объект f.
                    }
            }
    }
    catch (Exception ex)
    {
            // код обработки исключений.
    }
```

Аналитическое вычисление производных

Библиотека ANALYTICS позволяет вычислять производные выражений в аналитическом виде. Класс ***Translator*** (см. выше) содержит следующий метод:

```
public string Derivative(string formula, string vName)
```

Этот метод вычисляет производную формулы по переменной *vName* в аналитическом виде. Результатом является строка, представляющая производную выражения.

Далее приведены примеры кода вычисления производной:

```
string formula;
string derivative;
// #1
formula = "A*ln(x)*sin(2*x)";
derivative = translator.Derivative(formula, "x");
// derivative = (1/x)*(A*sin(2*x))+(cos(2*x)*2)*(A*ln(x))
// #2
formula = "e^(1-2*x)";
derivative = translator.Derivative(formula, "x");
// derivative = e^(1-2*x)*(-2)
// #3
formula = "(x+1)^(x-1)";
derivative = translator.Derivative(formula, "x");
// derivative = ln(x+1)*(x+1)^(x-1)+(x-1)*(x+1)^((x-1)-1)
```

Примеры демонстрируют простоту вычисления производных.

Замечания (о вычислении производных):

1. Параметр **formula** в методе ***Derivative*** должен быть синтаксически правильным. Правильность можно проверить методом **CheckSyntax** класса ***Translator***.
2. Переменная с именем *vName* (или любая другая) может не существовать. Метод ***Derivative*** манипулирует только именами переменных, а не их значениями.
3. Метод ***Derivative*** выполняет некоторые упрощения в процессе вычисления (удаляет нулевые слагаемые в суммах, единичные множители в произведениях и др.). Тем не менее, некоторые результаты вычисления производных могут быть не "идеальны". Они могут содержать несколько ‘лишних’ скобок и другие «артефакты». Это обусловлено тем, что библиотека ANALYTICS не является системой компьютерной алгебры. Результат вычисления производной предназначен для использования его в дальнейших расчетах (внутри программы), а не для представления его пользователю, как конечный результат.
4. Не все операторы поддерживаются при вычислении производной. Например, оператор '!' не поддерживается, поскольку для него не определены правила дифференцирования. Новые правила дифференцирования для операторов так же не могут быть определены (без изменения основных алгоритмов)
5. Большинство стандартных трансцендентных функций поддерживаются при вычислении производной. Правила дифференцирования могут быть определены для любой функции (см. ниже).

Прямая манипуляция строковыми выражениями

Библиотека ANALYTICS предоставляет функции для непосредственной манипуляции со "строковыми выражениями". Пусть в программе имеются две строки, содержащие математические выражения **‘1-2*x’** и **‘sin(x)+2’**. И пусть необходимо получить выражение, являющееся произведением этих выражений. Данная задача может быть осуществлена манипулированием со строками C#. Для приведенного выше примера необходимо сделать следующие шаги: заключить первое выражение в скобки, заключить второе выражение в скобки и, наконец, объединить результирующие строки с символом оператора '*'. Алгоритм кажется простым, но он становится все более и более сложным с увеличением числа операций.

Структура ***Expression*** (не путать **BaseExpression** в представленной выше иерархии классов) упрощает такие манипуляции со строками. Эта структуры является просто ‘оберткой’ для одного данного - строки (рис. 3.1). Структура содержит явные операторы преобразования - из строки, в строку, и перегруженные операторы, реализующие алгебраические операции **‘+’**, **‘-’**, **‘*’**, **‘/’**, **‘^’** (степень). Такая реализация позволяет манипулировать строковыми выражениями в "естественной" форме.

Рассмотрим пример применения структуры ***Expression*** для решения следующей задачи: реализовать "аналитический" 3D вектор. Этот вектор должен содержать три компоненты - строки, все компоненты должны быть математическими выражениями и все векторные операции (сумма, векторное произведение, длина и др.) должны быть аналитическими.

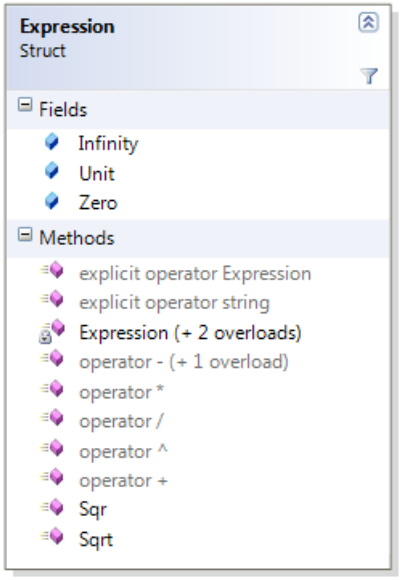


Рисунок 3.1. Структура Expression.

Следующий код является реализацией (частичной) "аналитического" 3D вектора:

```
public struct StringVector
{
    #region Data members
    private Expression e1;
    private Expression e2;
    private Expression e3;
    #endregion Data members

    /// <summary>
    /// Constructor.
    /// </summary>
    /// <param name="x1"></param>
    /// <param name="x2"></param>
    /// <param name="x3"></param>
    public StringVector(string x1, string x2, string x3)
    {
        e1 = (Expression)x1;
        e2 = (Expression)x2;
        e3 = (Expression)x3;
    }

    /// <summary>
    /// Dot Product
    /// </summary>
    /// <param name="v1"></param>
    /// <param name="v2"></param>
    /// <returns></returns>
    public static string Dot(StringVector v1, StringVector v2)
    {
        return (string)(v1.e1 * v2.e1 + v1.e2 * v2.e2 + v1.e3 * v2.e3);
    }

    /// <summary>
    /// Cross Product
    /// </summary>
    /// <param name="v1"></param>
    /// <param name="v2"></param>
    /// <returns></returns>
    public static StringVector operator *(StringVector v1, StringVector v2)
    {
        return new StringVector(
            (v1.e2 * v2.e3 - v1.e3 * v2.e2),
            (v1.e3 * v2.e1 - v1.e1 * v2.e3),
            (v1.e1 * v2.e2 - v1.e2 * v2.e1)
        );
    }
}
```

Структура ***StringVector*** содержит три элемента данных e1, e2, e3 типа ***Expression*** для инкапсуляции трех компонент вектора. Конструктор принимает три строковых аргумента, которые приводятся к типу ***Expression***, используя оператор явного преобразования.

Основные преимущества использования структуры ***Expression*** демонстрируются в методах ***Dot*** и ***Cross***. Код вычисления скалярного и векторного произведения аналогичен коду, если бы векторы содержали вещественные, а не строковые компоненты.

Рассмотрим результат использования структуры StringVector:

```
StringVector x1 = new StringVector("sin(u)", "cos(v)", "0");
StringVector x2 = new StringVector("cos(u)", "-sin(v)", "0");
StringVector r =  x1 * x2;
```

В результате, вектор r будет содержать три компонента (выражения) – '0', '0' и 'sin(u)*(-sin(v))-cos(v)*cos(u))'.

ПРИМЕЧАНИЕ (об использовании структуры **Expression**): все значения строк, при использовании структуры ***Expression***, должны быть синтаксически корректными, иначе, будет сгенерировано исключение. Синтаксис можно проверить методом **CheckSyntax** класса ***Translator*** (см. выше).

4. Расширение ANALYTICS

Библиотека ANALYTICS предоставляет полностью завершенное ядро для разбора и вычисления математических выражений. Также, ядро содержит множество предопределенных "стандартных" функций – тригонометрических, гиперболических и др. и реализованные операторы для вещественных аргументов. Таким образом, библиотека может быть использована без изменений для обеспечения интерфейса ввода данных в виде математических выражений.

Другая цель библиотеки это предоставление возможности выполнения математических вычислений с данными, специфическими для конкретной программы. Например, пусть существует некоторая программа обработки сигналов. Программа должна позволять пользователю выполнять различные операции с сигналами - складывать два сигнала, умножать сигнал на вещественные значения, вычислять логарифмы сигналов, экспоненциировать сигналы и т.д. Данная задача может быть решена с помощью библиотеки ANALYTICS. Основные алгоритмы библиотеки ANALYTICS могут использоваться для вычисления выражений, содержащих сигналы в качестве переменных. Единственное, что требуется, это обеспечить операции, определенные для сигналов (поскольку они являются данными, специфическими для конкретной программы).

Библиотека ANALYTICS построена по такой технологии, которая позволяет легко вводить в программу операции с любыми данными без изменения основных алгоритмов. Все операции с данными (операторы, функции) представлены в библиотеке в виде классов. Соответственно, для добавления в программу операции с некоторыми, специфическими для программы данными, необходимо реализовать класс - наследник некоторого существующего. Данный класс будет автоматически найден и использован для вычисления результатов выражений.

Следующий раздел содержит инструкции по реализации классов, выполняющих операции со специфическими для конкретной программы данными.

Перегрузка операторов

Перегрузка оператора для типов операндов, являющихся специфическими данными конкретной программы, осуществляется путем создания класса – наследника одного из существующих классов операторов. Базовые абстрактные классы реализованы для всех предопределенных операторов (см. иерархию классов): ***AddOperator***, ***SubtractOperator*** и др. Следующий код демонстрирует перегрузку операторов для массивов вещественных чисел:

```
/// <summary>
/// (Double Array) * (Double) = (Double Array)
/// </summary>
public sealed class ArrayRealMultiply : MultiplyOperator
{
    protected override Type GetOperand1Type()
    {
        return typeof(double[]);
    }

    protected override Type GetOperand2Type()
    {
        return typeof(double);
    }

    protected override Type GetReturnType()
    {
        return typeof(double[]);
    }

    protected override object Operation(object operand1, object operand2)
    {
        if (operand1 == null) return null;

        double[] a = (double[])operand1;
        double r = (double)operand2;

        int l = a.Length;
        double[] result = new double[l];
```

```
            for (int i = 0; i < l; i++)
            {
                result[i] = a[i] * r;
            }

            return result;
        }
    }
```

Класс ***ArrayRealMultiply*** наследуется от ***MultiplyOperator***, поскольку осуществляется перегрузка оператора "*". Класс переопределяет методы ***GetOperand1Type***, ***GetOperand2Type*** и ***GetReturnType***, которые определяют, что первый операнд имеет тип массив 'double', второй имеет тип 'double' и результатом операции является так же массив 'double'. Последний переопределенный метод ***Operation*** – реализует собственно операцию, которая выполняется над операндами.

Следующий пример демонстрирует перегрузку оператора "+" для массивов вещественных чисел:

```
    /// <summary>
    /// (Double Array) + (Double Array) = (Double Array)
    /// </summary>
    public sealed class ArrayAdd : AddOperator
    {
        protected override Type GetOperand1Type()
        {
            return typeof(double[]);
        }

        protected override Type GetOperand2Type()
        {
            return typeof(double[]);
        }

        protected override Type GetReturnType()
        {
            return typeof(double[]);
        }

        protected override object Operation(object operand1, object operand2)
        {
            if (operand1 == null || operand2 == null) return null;

            double[] a1 = (double[])operand1;
            double[] a2 = (double[])operand2;

            int l = a1.Length;
            double[] result = new double[l];

            for (int i = 0; i < l; i++)
            {
                result[i] = a1[i] + a2[i];
            }

            return result;
        }
    }
```

Классы операторов будут автоматически найдены библиотекой ANALYTICS и использованы для реализации операций с операндами определенных типов.

Тот же результат, что и в двух предыдущих примерах кода, может быть достигнут с помощью обобщенных (generic) аналогов базовых классов операторов. Следующий код демонстрирует такой подход:

```
    /// <summary>
    /// (Double Array) * (Double) = (Double Array)
    /// </summary>
    public sealed class ArrayRealMultiply : GenericMultiplyOperator<double[], double, double[]>
    {
        protected override double[] TypedOperation(double[] operand1, double operand2)
        {
            if (operand1 == null) return null;

            int l = operand1.Length;
            double[] result = new double[l];

            for (int i = 0; i < l; i++)
            {
                result[i] = operand1[i] * operand2;
            }

            return result;
```

```
        }
    }

    /// <summary>
    /// (Double Array) + (Double Array) = (Double Array)
    /// </summary>
    public sealed class ArrayAdd : GenericAddOperator<double[], double[], double[]>
    {
        protected override double[] TypedOperation(double[] operand1, double[] operand2)
        {
            if (operand1 == null || operand2 == null) return null;

            int l = operand1.Length;
            double[] result = new double[l];

            for (int i = 0; i < l; i++)
            {
                result[i] = operand1[i] + operand2[i];
            }

            return result;
        }
    }
```

Использование обобщенных базовых классов операторов позволяет "сократить" код перегрузки, поскольку только один метод (***TypedOperation***) должен быть переопределен. Типы операндов и возвращаемого значения определены как параметры обобщенного класса, от которого новый класс унаследован. Оба способа перегрузки операторов (с использованием обобщенных классов и без) абсолютно эквивалентны для основных алгоритмов библиотеки ANALYTICS.

Явная перегрузка операторов

Перегрузка операторов в библиотеке ANALYTICS предназначена для реализации операций над данными, специфическими для конкретной программы. Часто, для таких данных, некоторые операторы уже перегружены "внутри" программы (что означает, перегрузку операторов C#). Эти "явно перегруженные операторы" могут быть использованы для реализации операций над операндами соответствующих типов основными алгоритмами библиотеки ANALYTICS. Другими словами, если существует "явно перегруженный" оператор, он будет автоматически найден и использован для вычисления результата соответствующей операции. Например, поскольку тип ***Complex*** реализует перегрузку математических операторов, такие действия, как сложение, вычитание и т.д. могут быть выполнены для комплексных чисел без создания новых классов операторов.

Существуют некоторые ограничения на использование "явной перегрузки" операторов:

1. Не все операторы могут быть "явно перегружены", поскольку не все операторы ANALYTICS имеют аналоги в языке C# или их "значение" может быть неоднозначным (например, оператор "^"). Таким образом, только следующие операторы могут быть "явно перегружены": "**+**", "**-**" (унарный и бинарный), "*****", "**/**", '≡' (==)[5], '≠' (!=), '>', '<', '≥' (>=), '≤' (<=), '&', '\' (|), '¬' (!).
2. "Неявная" перегрузка (создание нового класса оператора) имеет более высокий приоритет. То есть, если есть и "явный" и "неявный" оператор - для вычисления результата операции будет использован последний.

Оба метода перегрузки могут быть использованы совместно. "Неявная" перегрузка (с помощью новых классов операторов) может быть использована как для дополнения "явного" метода, так и для переопределения его поведения.

Определение новых функций

Определение функций, для данных, специфических для конкретной программы, является наиболее общим способом расширения функциональности библиотеки. Функция может иметь любое допустимое имя, любое количество аргументов разных типов и любой тип возвращаемого значения. Для определения новой функции, необходимо реализовать класс - потомок одного из базовых абстрактных классов функций. Выбор базового класса для наследования функции шире, чем для перегрузки оператора (см. иерархию классов). Для распространенных типов аргументов (действительных и комплексных), в качестве класса-предка может быть использован один из предопределенных абстрактных классов: ***RealElementaryFunction***, ***RealParametricElementaryFunction***, ***ComplexElementaryFunction***, ***ComplexParametricElementaryFunction***. В качестве примера, приведен код класса, реализующего функцию синуса комплексного аргумента:

```
    /// <summary>
    /// Sine function of Complex Argument
    /// </summary>
    public sealed class SineComplex : ComplexElementaryFunction
    {
        protected override string GetName()
        {
            return "sin";
        }

        protected override Complex Func(Complex x)
```

[5] Аналоги операторов C# приведены в скобочках.

```
        {
            return Complex.Sin(x);
        }
    }
```

Класс наследуется от ***ComplexElementaryFunction***, поскольку он реализует функцию с одним аргументом комплексного типа. Переопределенными методами являются ***GetName*** и ***Func***. Первый определяет имя функции (используется в выражениях), а последний реализует собственно операцию с аргументом. Другой пример демонстрирует реализацию функции логарифм комплексного аргумента по некоторому основанию:

```
    /// <summary>
    /// Logarithm function of Complex argument
    /// (NOTE: the Base of logarithm is the Parameter of the function)
    /// </summary>
    public sealed class LogarithmComplex : ComplexParametricElementaryFunction
    {
        protected override string GetName()
        {
            return "log";
        }

        protected override Complex Func(Complex parameter, Complex argument)
        {
            return Complex.Log(argument)/Complex.Log(parameter);
        }
    }
```

Класс наследуется от ***ComplexParametricElementaryFunction***, т.к. он реализует функцию с одним параметром и одним аргументом комплексного типа.

В общем случае, для реализации новой функции, класс должен наследоваться от базового абстрактного класса ***Function***. Все абстрактные методы базового класса должны быть переопределены. Методы должны определять количество и тип параметров и аргументов и тип возвращаемого функцией результата. В следующем коде демонстрируется пример реализации функции Min, которая вычисляет минимальное значение в массиве вещественных чисел:

```
    /// <summary>
    /// Min array element
    /// </summary>
    public sealed class MinArray : Function
    {
        protected override string GetName()
        {
            return "Min";
        }

        protected override int GetArgumentCount()
        {
            return 1;
        }

        protected override Type[] GetArgumentTypes()
        {
            return new Type[] { typeof(double[]) };
        }

        protected override int GetParameterCount()
        {
            return 0;
        }

        protected override Type[] GetParameterTypes()
        {
            return null;
        }

        protected override Type GetResultType()
        {
            return typeof(double);
        }

        protected override object DoCalculate(object[] parameters, object[] arguments)
        {
            double[] a = (double[])arguments[0];
            if (a == null || a.Length == 0) return double.NaN;

            double result = a[0];
            int l = a.Length;
            for (int i = 1; i < l; i++)
                if (a[i] < result) result = a[i];
```

```
            return result;
        }
    }
```

Функция имеет один аргумент типа массив вещественных чисел и возвращает в качестве результата одно вещественное значение.

Несколько более быстрый способ определения новой функции для конкретных типов – использование обобщенных базовых классов функций. Данные классы используют параметры обобщений (generic) для определения типов аргументов. Таким образом, только имя функции и алгоритм вычисления должны быть определены в новом классе функции. Следующий код демонстрирует реализацию предыдущего примера с использованием обобщенного базового класса:

```
    /// <summary>
    /// Min array element
    /// </summary>
    public sealed class MinArray : GenericSimpleFunction<double[], double>
    {
        protected override string GetName()
        {
            return "Min";
        }

        protected override double Func(double[] a)
        {
            if (a == null || a.Length == 0) return double.NaN;

            double result = a[0];
            int l = a.Length;
            for (int i = 1; i < l; i++)
                if (a[i] < result) result = a[i];

            return result;
        }
    }
```

Класс наследуется от ***GenericSimpleFunction***, поскольку функция имеет один аргумент и не имеет параметров. Аргумент является массивом вещественных чисел, а результат – одно вещественное число, что определяется первым и вторым параметрами обобщения соответственно.

Библиотека ANALYTICS содержит обобщенные базовые классы для реализации функции до двух параметров и двух аргументов любого типа.

ПРИМЕЧАНИЕ (о функциях): может существовать множество функций с одним и тем же именем, но разным количеством или/и типом аргументов (параметров).

Реализация индексации

Индексация применима только для переменных. Т.о., реализация индексации для специфических данных конкретной программы, заключается в создании нового класса переменной, который должен наследоваться от абстрактного класса **IndexedVariable** (или от одного из его потомков). Унаследованный класс должен переопределить методы, определяющие количество индексов данных, методы доступа к данным и реализации получения ‘сечений’.

В качестве примера реализации индексации, ниже приведен (частично) код стандартного класса ***MatrixVariable***:

```
    /// <summary>
    /// Base abstract class for all matrix variables.
    /// NOTE: Slicing is implemented.
    /// </summary>
    public abstract class MatrixVariable : BaseArrayVariable
    {
        /// <summary>
        /// 2 indexes
        /// </summary>
        /// <returns></returns>
        protected override sealed int GetIndexCount()
        {
            return 2;
        }

        /// <summary>
        /// Slicing is implemented for Matrix variables
        /// </summary>
        /// <returns></returns>
        protected override bool GetSlicingImplemented()
        {
            return true;
        }

        /// <summary>
        /// Sliced item is array of BaseType
        /// </summary>
        /// <param name="indexes"></param>
```

```
        /// <returns></returns>
        public override Type GetItemType(int[] indexes)
        {
            if ( (indexes[0] >= 0 && indexes[1] < 0)
                || (indexes[0] < 0 && indexes[1] >=0) )
            {
                return BaseType.MakeArrayType();
            }

            return BaseType;
        }

        /// <summary>
        /// Implements Array Slicing
        /// </summary>
        /// <param name="indexes"></param>
        /// <returns></returns>
        public override object GetItemValue(int[] indexes)
        {
            // Row
            if (indexes[0] >= 0 && indexes[1] < 0)
            {
                int len = ColumnCount;
                Array result = Array.CreateInstance(BaseType, len);
                int[] row = new int[] { indexes[0], 0 };

                for (int j = 0; j < len; j++)
                {
                    row[1] = j;
                    object x = data.GetValue(row);
                    result.SetValue(x, j);
                }

                return result;
            }
            else
            {
                // Column
                if (indexes[0] < 0 && indexes[1] >= 0)
                {
                    int len = RowCount;
                    Array result = Array.CreateInstance(BaseType, len);
                    int[] column = new int[] { 0, indexes[1] };

                    for (int i = 0; i < len; i++)
                    {
                        column[0] = i;
                        object x = data.GetValue(column);
                        result.SetValue(x, i);
                    }

                    return result;
                }
            }

            // Item
            return base.GetItemValue(indexes);
        }
    }
```

Класс переопределяет метод ***GetIndexCount*** который возвращает 2, поскольку элемент матрицы имеет два индекса. Метод ***GetSlicingImplemented*** возвращает значение 'истина', что означает - данная переменная поддерживает получение 'сечений'. Метод ***GetItemType*** возвращает тип индексированных данных, с учетом возможности получения 'сечений'. Если один из индексов меньше нуля, значит метод должен возвратить сечение по данному индексу. Для матрицы это будет массив (вектор-строка или вектор-столбец). Метод ***GetItemValue*** реализует доступ к данным, с учетом возможности получения 'сечений', аналогично методу ***GetItemType***.
ПРИМЕЧАНИЕ (о реализации индексации): тип индекса всегда является целым, и это не может быть переопределено.

Определение правил дифференцирования функций

Расширение библиотеки ANALYTICS в смысле вычисления производных отличается от других расширений. Например, правила дифференцирования не могут быть переопределены для операторов, так как эти правила являются основой математики.

Правила вычисления производных могут быть определены только для функций. Для определения правила дифференцирования функции необходимо создать новый класс - потомок базового класса **FunctionalDerivative**, и переопределить следующие абстрактные методы:

`protected abstract string GetFunctionName();` - имя функции.

```
protected abstract int GetParameterCount();  - число параметров функции.
protected abstract int GetArgumentCount(); - число аргументов функции.
public abstract BaseExpression Derivative(DerivativeContext context, FunctionExpression
function, string vName);
```

Первые три метода определяют, для какой функции создается правило вычисления производной. Следует обратить внимание, что типы параметров и аргументов не задаются. Это одно из отличий от определения функции, поскольку вычисление производной не зависит от типов аргументов, а вычисление функции зависит.

Последний метод определяет собственно правило дифференцирования. Этот метод принимает в качестве параметра объект класса ***FunctionExpression*** (см. иерархию классов) и параметр *vName* – имя переменной, по которой выполняется дифференцирование. Параметр *context* требуется только для передачи в другие методы вычисления производных. Результатом выполнения метода является объект класса ***BaseExpression***. Таким образом, метод должен получить результат выражения из выражения, заданного параметром *function*.

В общем случае, алгоритм вычисления производной функции реализуется достаточно сложно. Для создания результирующего выражения могут быть использованы методы класса ***BaseExpression*** (см. иерархию классов) или его потомков. Сложность реализации объясняется тем, что правила вычисления производных действительно сложные в общем случае функции с большим количеством аргументов и параметров. В качестве примера можно привести правило дифференцирования функции логарифма:

$$\frac{d}{dx}\left(\log_{a(x)} g(x)\right)=\frac{1}{g(x)\ln(a(x))}\frac{d}{dx}(g(x))-\frac{\ln(g(x))}{a(x)\ln^2(a(x))}\frac{d}{dx}(a(x))$$

Правила дифференцирования не могут быть определены для всех функций однообразно, они должны быть определены для каждой функции индивидуально.

Однако, для стандартных элементарных функций с одним аргументом правило дифференцирования можно обобщить, используя следующую математическую формулу:

$$\frac{d}{dx}(f(g(x)))=\frac{df}{dg}\frac{dg}{dx}$$

Данное правило реализовано в классе ***SimpleFunctionalDerivative***. Класс является абстрактным и реализует правило дифференцирования функции одного аргумента. Он переопределяет базовый метод ***Derivative*** (реализуя изложенное выше правило). Единственный абстрактный метод класса, который должен быть переопределен в потомках это

```
protected abstract BaseExpression BaseDerivative(BaseExpression argument);
```

Параметр метода *argument* соответствует функции g в приведенной выше формуле. Реализация метода довольно проста для большинства стандартных трансцендентных функций. В качестве примера приведен код класса, реализующего производную функции синуса:

```
/// <summary>
/// Sine derivative
/// </summary>
public sealed class SineDerivative : SimpleFunctionalDerivative
{
    protected override string GetFunctionName()
    {
        return "sin";
    }

    protected override BaseExpression BaseDerivative(BaseExpression argument)
    {
        return FunctionExpression.CreateSimple("cos", argument);
    }
}
```

Реализация является достаточно простой. Метод ***GetFunctionName*** определяет, что производная задается для функции с именем ‘sin’. Метод ***BaseDerivative*** создает новое выражение – функцию с именем ‘cos’ и одним заданным аргументом, поскольку $\frac{d}{dx}(\sin(x))=\cos(x)$.

Как видно из примера, для построения результирующих выражений должны быть использованы методы класса ***BaseExpression*** (см. иерархию классов) или его потомков. Класс содержит множество полезных методов для создания суммы, разности, произведения, отношения, степени или других выражений. Кроме того, реализована перегрузка всех алгебраических операций (‘+’. ‘-‘, ‘*’, ‘/’, ‘^’ (степень)[6]) с классами выражений для более удобного и наглядного оперирования выражениями в коде.

Ядро вычисления производных библиотеки ANALYTICS поддерживает простое определение правил дифференцирования и для более сложных функций. Например, в случае функции двух переменных, правило вычисления частной производной функции обобщается следующими формулами:

$$\frac{\partial}{\partial x}(f(g(x,y),h(x,y)))=\frac{\partial f}{\partial g}\frac{\partial g}{\partial x}+\frac{\partial f}{\partial h}\frac{\partial h}{\partial x}$$

[6] Перегруженный оператор ‘^’ должен быть ВСЕГДА заключен в скобки при использовании в выражениях, поскольку C# ‘^’ оператор имеет меньший приоритет, чем алгебраические операторы.

$$\frac{\partial}{\partial y}\left(f\left(g(x,y),h(x,y)\right)\right)=\frac{\partial f}{\partial g}\frac{\partial g}{\partial y}+\frac{\partial f}{\partial h}\frac{\partial h}{\partial y}$$

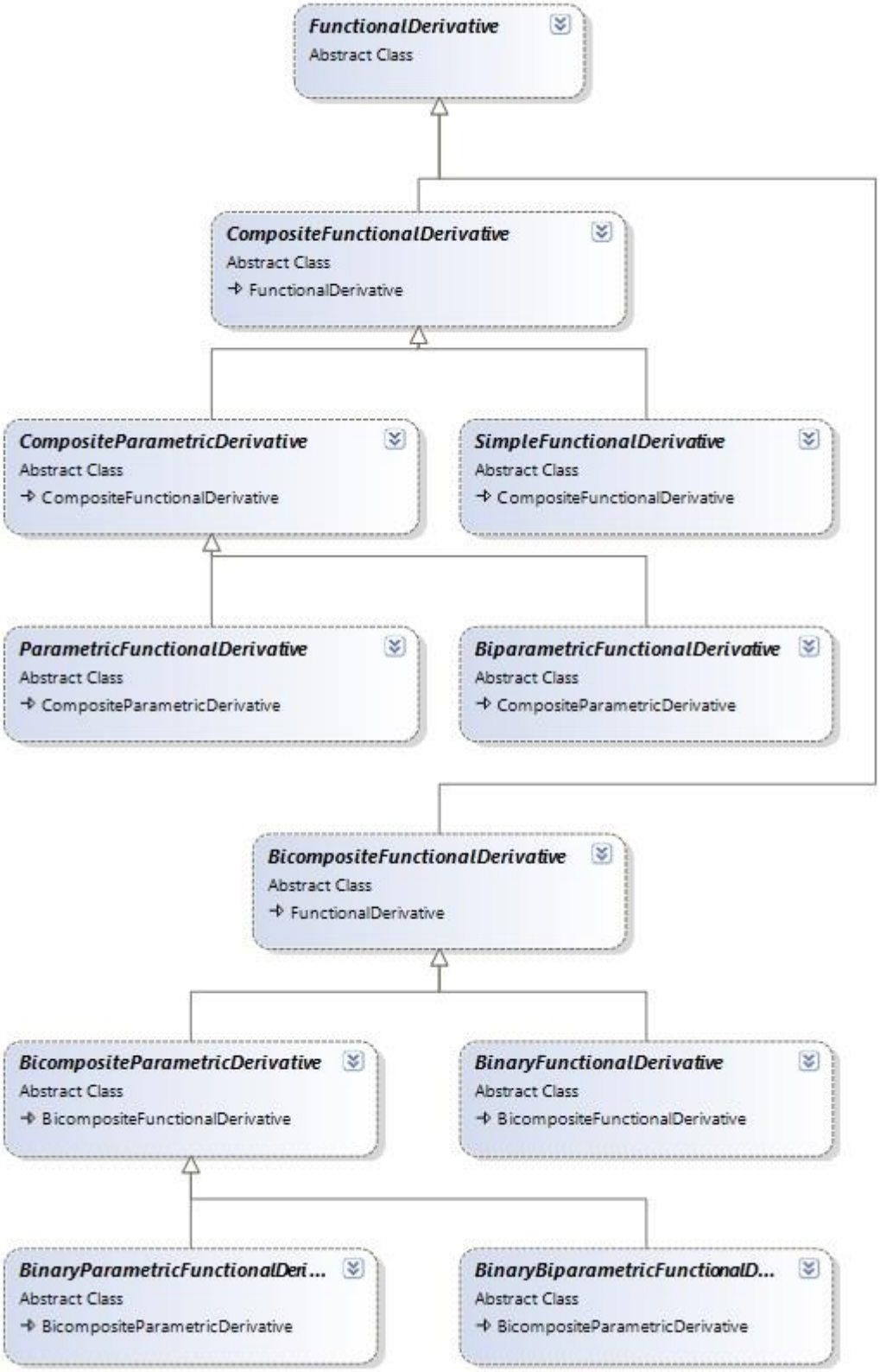


Рис. 4.1. Диаграмма иерархии классов функциональных производных.

Это правило реализовано в абстрактном классе ***BinaryFunctionalDerivative*** и правила вычисления производных всех бинарных функций могут быть реализованы с помощью данной абстракции. Для реализации необходимо переопределить два метода:

```
protected abstract BaseExpression BaseDerivative1(BaseExpression argument1, BaseExpression
argument2);
protected abstract BaseExpression BaseDerivative2(BaseExpression argument1, BaseExpression
argument2);
```

Правило дифференцирования параметрических функций так же может быть обобщено на случай сложной функции, если параметр(-ы) не зависит от переменной(-ых). Реализованы абстракции данного правила для функций до двух аргументов и до двух параметров (не зависящих от переменных). Иерархия абстрактных классов производных функций представлена на рисунке 4.1.

5. Расширения библиотеки ANALYTICS

Библиотека ANALYTICS является завешенным ядром для разбора и вычисления математических выражений. Кроме ядра, существует несколько готовых расширений библиотеки, содержащих функциональность для специфических областей математики. Под **расширением** понимается некоторая библиотека, которая не является обязательной для функционирования основных алгоритмов ядра, но ее наличие вносит новую функциональность. Любое расширение может содержать следующие возможности:

- распознавание новых типов литералов;
- новые типы переменных;
- операторы для специализированных типов операндов;
- функции специализированных типов аргументов.

Следующие разделы содержат описание существующих расширений библиотеки ANALYTICS.

ANALYTICS Complex

Расширение ANALYTICS Complex реализует операции с комплексными числами (тип Complex C#). Как было сказано ранее, ядро библиотеки ANALYTICS поддерживает комплексные литералы. Кроме того, ядро ANALYTICS поддерживает ‘явную’ перегрузку операторов. Таким образом, некоторые операции с комплексными числами могут быть выполнены без наличия расширения. Например, может быть вычислено выражение **‘(2-I)*(3+2I)’**. Однако, данная функциональность ограничена работой с литералами и операторами, допускающими ‘явную’ перегрузку.

Расширение ANALYTICS Complex включает следующую функциональность:

1. Типы переменных: ***ComplexVariable***, ***ComplexArrayVariable***, ***ComplexMatrixVariable***, ***ComplexBlockVariable***.
2. Операторы для комплексных операндов: **‘^’** (степень), **‘~’** (сопряженный), **‘’’** (обратный).
3. Бинарные операторы для вещественных-комплексных операндов: **‘+’**, **‘-’**, **‘*’**, **‘/’**, **‘^’**.
4. Функции для комплексных и вещественных аргументов/параметров: ***Complex*** и ***Polar*** (создает комплексное число по вещественным аргументам), ***Conjugate***, ***Reciprocal***, ***Re*** (вещественная часть), ***Im*** (мнимая часть), ***Arg*** (аргумент), ***abs*** (модуль), ***sqrt*** (корень квадратный), ***root*** (корень n-ой степени), ***pow*** (степень), ***log*** (логарифм), ***ln*** (натуральный логарифм), ***lg*** (десятичный логарифм), ***lb*** (двоичный логарифм), ***exp*** (экспонента), тригонометрические функции (***sin***, ***cos***, ***tan***), обратные тригонометрические функции (***arcsin***, ***arccos***, ***arctan***), гиперболические функции (***sinh***, ***cosh***, ***tanh***), обратные гиперболические функции (***arcsinh***, ***arccosh***, ***arctanh***).

ANALYTICS Fractions

Расширение ANALYTICS Fractions реализует аналитические вычисления с обыкновенными дробями. Реализованы следующие возможности:

1. Тип литералов: библиотека поддерживает распознавание литерала ‘обыкновенная дробь’. После регистрации данного литерала, выражения типа **‘a/b’ распознаются как обыкновенная дробь, если ‘a’** и **‘b’ – целые числа**. ЗАМЕЧАНИЕ: поскольку различные операторы имеют разный приоритет выполнения, дробный литерал должен быть заключен в скобки, если он стоит в выражении с операторами, имеющими больший или такой же приоритет, как оператор ‘/’. Например, **‘2/3+4/2’** – скобки НЕ обязательны, **‘(2/3)*(4/2)’** – скобки ОБЯЗАТЕЛЬНЫ.
2. Тип переменной: ***FractionVariable***.
3. Операторы для дробных операндов: **‘^’** (степень), **‘~’** (преобразование в вещественное значение), **‘’’** (обратная дробь), а так же все стандартные алгебраические операторы и операторы сравнения.
4. Стандартные бинарные алгебраические операторы для операндов дробь-вещественное число и дробь-комплексное число, а так же операторы сравнения для операндов дробь-вещественное число.
5. Функции для аргументов типа обыкновенная дробь: **Fraction** (создает дробь по числителю и знаменателю или преобразует вещественное число в обыкновенную дробь), ***Numerator***, ***Denominator***, ***Real*** (преобразует дробь в вещественное число), ***frac*** (дробная часть), ***int*** (целая часть), ***sgn*** (знак).

ANALYTICS Linear Algebra

Расширение ANALYTICS Linear Algebra реализует возможности использования в аналитических выражениях 3D векторы и тензоры, а так же n-мерные векторы и прямоугольные матрицы. Реализованы следующие возможности[7]:

1. Типы переменных: ***Vector3DVariable***, ***Tensor3DVariable***.
2. Операторы для операндов ***Vector3D***: **‘•’** (скалярное произведение).
3. Операторы для операндов ***Tensor3D***: **‘’’** (транспонирование), **‘^’** (степень – целое число, включая отрицательные значения).
4. Операторы для n-мерных векторов и вещественных операндов: **‘+’**, **‘-’**, **‘*’**, **‘/’**, **‘•’**.

[7] Библиотека ANALYTICS Linear Algebra зависит от библиотеки MATHEMATICS, поскольку использует специальные типы, такие ка Vector3D, Tensor3D и др. Поскольку данные типы имеют перегруженные операторы для выполнения основных математических операций, в расширении реализована только перегрузка остальных операторов (‘явно’ перегруженные операторы используются автоматически).

5. Операторы для прямоугольных матриц, n-мерных векторов и вещественных операндов: **‘+’, ‘-’, ‘*’, ‘/’, ‘^’** (степень – целое число, включая отрицательные значения).
6. Функции для аргументов/параметров ***Vector3D***: ***Vector*** (создает 3D вектор по вещественным компонентам), ***Length***, ***Dot***, ***Cross***, ***Angle***, ***Distance***, ***Direction***, ***Normal***, ***Area***.
7. Функции для аргументов/параметров ***Tensor3D***: ***Tensor*** (создает 3D тензор по вещественным компонентам), ***Invariant1***, ***Invariant2***, ***Invariant3***, ***Transpose***, ***Inverse***, ***Symmetric***, ***Antisymmetric***, ***Invariant1***, ***Solve*** (решает систему линейных уравнений).
8. Функции для аргументов/параметров ***Vector*** (n-мерный): ***Min***, ***Max***, ***Length***.
9. Функции для аргументов/параметров ***Matrix*** (прямоугольная): ***Min***, ***Max***, ***Transpose***, ***Inverse***, ***Minor***, ***Determinant***, ***Adjoint***, ***Solve*** (решает систему линейных уравнений).
10. Дополнительные типы: ***StringVector***, ***StringTensor***. Эти специальные типы реализуют ‘аналитические’ 3D вектор и тензор (их компоненты являются строковыми математическими выражениями, и все операции выполняются в аналитической форме).

ANALYTICS Mathphysics

Расширение ANALYTICS Mathphysics реализует функциональность для использования физических понятий (единицы измерения и физические значения)[8] в аналитических выражениях. Реализованы следующие возможности:

1. Типы литералов: библиотека поддерживает распознавание следующих литералов – ***единица измерения***, ***скалярное значение***, ***векторное значение и тензорное значение***. Литерал единица измерения – это последовательность символов, представляющая корректную запись некоторой единицы измерения. Поскольку существует множество обозначений единиц измерения и префиксов, и запись единиц измерения достаточно ‘близка’ к записи математических выражений (используемой в библиотеке ANALYTICS), литералы единиц измерения ДОЛЖНЫ БЫТЬ ЗАКЛЮЧЕНЫ в треугольные скобки **‘‹›’**[9]. Примеры корректной записи литералов единиц измерения: **‘‹mm^2›’**, **‘‹m kg/s^2›’**, **‘‹m kg s^-2›’**. Литерал скалярное значение это вещественное число и связанная с ним единица измерения. Другими словами – Вещественный литерал + литерал единицы измерения. Вещественный литерал и литерал единицы измерения могут быть разделены символом пробела **‘ ’** или могут быть записаны слитно. Примеры корректной записи литералов скалярных значений: **‘4‹m/s›’**, **‘2.2‹m kg s^-2›’**, **‘-3 ‹m/s^2›’**. ЗАМЕЧАНИЕ: поскольку символ пробела **‘ ’** используется в литералах единиц измерения вместо знака умножения (чтобы запись была более близка к принятой в физике), а в библиотеке ANALYTICS данный символ используется как разделитель аргументов функций, литералы единиц измерения (и скалярных значений) следует заключать в дополнительные скобки, когда они используются как параметры/аргументы функций. Векторное значение это векторный литерал и литерал единицы измерения разделенные (опционально) пробелом. Векторный литерал представляет собой три вещественных литерала (компоненты вектора), разделенные пробелами и заключенные в скобки. Тензорное значение это, аналогично, тензорный литерал и литерал единицы измерения. Тензорный литерал подобен векторному, но содержит девять вещественных литералов (компонент тензора, расположенных по строкам). Примеры корректных литералов: векторное значение **‘(1 0 -1) ‹m/s›’**, тензорное значение **‘(1 0.3 -1 -0.3 2 0.4 1 -0.4 3) ‹N/mm^2›’**.
2. Типы переменныхх: ***UnitVariable***, ***PhysicalScalarVariable***, ***PhysicalVectorVariable***, ***PhysicalTensorVariable***.
3. Операторы для операндов скалярных значений, единиц измерения и вещественных значений: **‘+’, ‘-’, ‘*’, ‘/’, ‘^’** (степень – целое число, включая отрицательные значения).
4. Функции для параметров/аргументов скалярных значений, единиц измерения и вещественных значений: ***Convert*** (конвертирует значение из одной единицы измерения в другую), ***Value*** (вещественное значение физического скаляра), ***Unit*** (единица измерения физического скаляра).
5. Операторы для операндов векторных значений, единиц измерения и вещественных значений: **‘+’, ‘-’, ‘*’, ‘/’, ‘•’** (скалярное произведение векторных значений).
6. Операторы для операндов тензорных и векторных значений, единиц измерения и вещественных значений: **‘+’, ‘-’, ‘*’, ‘/’, ‘^’** (степень – целое число, включая отрицательные значения), **‘'’** (транспонированный тензор).
7. Функции для параметров/аргументов векторных и тензорных значений: ***Vector*** (создает физическое векторное значение), ***Tensor*** (создает физическое тензорное значение).

ANALYTICS Special

Расширение ANALYTICS Special реализует возможность вычисления специальных функций в аналитических выражениях. Реализованы следующие специальные функции: ***P*** – полином Лежандра и присоединенный полином Лежандра первого рода, J_0, J_1, Y_0, Y_1, I_0, I_1, K_0, K_1 – функции Бесселя и модифицированные функции Бесселя первого и второго рода (порядка 0 и 1).

Библиотека Special содержит реализацию правил вычисления производных для всех приведенных выше специальных функций. Дополнительно реализованы правила вычисления производных следующих специальных функций:: J_n, Y_n, I_n, K_n (функции Бесселя n-ого порядка), ***Q*** – полиномы Лежандра и присоединенные функции Лежандра второго рода, B – бета функция и неполная бета функция, Γ – гамма функция, неполная и логарифмическая гамма функция, ψ – дигамма и полигамма функция, erf – функция ошибок, erfc – дополнительная функция ошибок.

[8] Библиотека ANALYTICS Mathphysics зависит от библиотеки PHYSICS. Подробная информация о единицах измерения и физических значениях изложена в руководстве по библиотеке PHYSICS.

[9] Следует различать символы треугольных скобок ‘‹ ›’ и символы ‘<’ – меньше, ‘>’ – больше.

Приложение A. Операторы и функции библиотеки Analytics

Таблица A.1. Список операторов, определенных в библиотеке ANALYTICS.

Оператор	Символ	Тип	Производная[10]
Logical And	&	Binary	False
Logical Or	\	Binary	False
Identically equal	≡	Binary	False
Approximately equal	≈	Binary	False
Not equal	≠	Binary	False
Greater	>	Binary	False
Less	<	Binary	False
Greater or equal	≥	Binary	False
Less or equal	≤	Binary	False
Add	+	Binary	True
Subtract	-	Binary	True
Multiply	*	Binary	True
Divide	/	Binary	True
Dot	•	Binary	False
Power	^	Binary	True
Left arrow	←	Binary	False
Right arrow	→	Binary	False
Up arrow	↑	Binary	False
Down arrow	↓	Binary	False
Left-right arrow	↔	Binary	False
Up-down arrow	↕	Binary	False
Logical Not	¬	Unary, Prefix	False
Question	?	Unary, Prefix	False
Number	#	Unary, Prefix	False
Minus	-	Unary, Prefix	True
Tilde	~	Unary, Prefix	False
Square Root	√	Unary, Prefix	True
Derivative	∂	Unary, Prefix	True
Integral	∫	Unary, Prefix	True
Delta	Δ	Unary, Prefix	False
Sum	∑	Unary, Prefix	True
Product	∏	Unary, Prefix	False
Factorial	!	Unary, Postfix	False
Apostrophe	'	Unary, Postfix	False
Absolute	||	Unary, Outfix	True

[10] Если производная не определена для некоторого оператора, он может использоваться в выражении для получения символьного представления производной, если его операнды не зависят от переменной дифференцирования. Оператор скалярного произведения НЕ может быть использован при символьном вычислении производной в любом случае.

Таблица A.2. Список основных функций, определенных в библиотеке ANALYTICS.

Функция[11]	Имя	Пример	Производная[12]
Absolute value	abs	abs(x)	sgn(x)
Signum R	sgn	sgn(x)	2*delta(x)
Dirac delta function R	delta	delta(x)	not defined
Heaviside step function R	H	H(x)	delta(x)
If (conditional) function	if	if{x>0}(x x^2)	if{x>0}(1 2*x)
Ceiling function R	ceil	ceil(x)	not defined
Floor function R	floor	floor(x)	not defined
Fractional part R	frac	frac(x)	not defined
Sine	sin	sin(x)	cos(x)
Cosine	cos	cos(x)	-sin(x)
Tangent	tan	tan(x)	1/cos(x)^2
Cotangent	cotan	cotan(x)	-1/sin(x)^2
Secant	sec	sec(x)	sin(x)/cos(x)^2
Cosecant	cosec	cosec(x)	-cos(x)/sin(x)^2
Inverse sine	arcsin	arcsin(x)	1/(1-x^2)^(1/2)
Inverse cosine	arccos	arccos(x)	-1/(1-x^2)^(1/2)
Inverse tangent	arctan	arctan(x)	1/(1+x^2)
Inverse cotangent	arccot	arccot(x)	-1/(1+x^2)
Inverse secant	arcsec	arcsec(x)	1/(x^2*(1-1/x^2)^(1/2))
Inverse cosecant	arccsc	arccsc(x)	-1/(x^2*(1-1/x^2)^(1/2))
Hyperbolic sine	sinh	sinh(x)	cosh(x)
Hyperbolic cosine	cosh	cosh(x)	sinh(x)
Hyperbolic tangent	tanh	tanh(x)	1/cosh(x)^2
Hyperbolic cotangent	coth	coth(x)	-1/sinh(x)^2
Hyperbolic secant	sech	sech(x)	-(tanh(x)*sech(x))
Hyperbolic cosecant	cosech	cosech(x)	-(coth(x)*cosech(x))
Inverse hyperbolic sine	arsinh	arsinh(x)	1/(x^2+1)^(1/2)
Inverse hyperbolic cosine	arcosh	arcosh(x)	1/(x^2-1)^(1/2)
Inverse hyperbolic tangent	artanh	artanh(x)	1/(1-x^2)
Inverse hyperbolic cotangent	arcoth	arcoth(x)	1/(1-x^2)
Inverse hyperbolic secant	arsech	arsech(x)	-1/((x^2*(1/x-1)^(1/2))*(1/x+1)^(1/2))
Inverse hyperbolic cosecant	arcsch	arcsch(x)	-1/(x^2*(1+1/x^2)^(1/2))
Logarithm to base	log	log{a}(x)	1/(ln(a)*x)
Natural logarithm	ln	ln(x)	1/x
Decimal logarithm	lg	lg(x)	1/(ln(10)*x)
Binary logarithm	lb	lb(x)	1/(ln(2)*x)
Exponent	exp	exp(x)	exp(x)
Square root	sqrt	sqrt(x)	1/(2*x^(1/2))
Root (with index)	root	root{a}(x)	(1/a)*x^(1/a-1)
Power	pow	pow{a}(x)	a*x^(a-1)

[11] Большинство из основных функций определены для вещественных и комплексных аргументов/параметров. Если функция не определена для комплексных переменных, она отмечена символом 'R'.

[12] Если производная не определена для некоторой функции, она может быть использована в выражении при символьном дифференцировании, если ее аргументы/параметры не зависят от переменной дифференцирования.

Функция[11]	Имя	Пример	Производная[12]
Beta[D] function[13]	B	B(x y)	B(x y)*(ψ(x)-ψ(x+y))
Incomplete Beta[D]	B	B{n m}(x)	x^(n-1)*(1-x)^(m-1)
Gamma[D] function	Γ	Γ(x)	Γ(x)*ψ(x)
Logarithm of Gamma[D]	Γlog	Γlog(x)	ψ(x)
Incomplete gamma[D]	Γ	Γ{n}(x)	-(x^(n-1)*e^-x)
Digamma function[D]	ψ	ψ(x)	ψ{1}(x)
Polygamma[D] function	ψ	ψ{n}(x)	ψ{n+1}(x)
Error[D] function	erf	erf(x)	(2/π^(1/2))*e^-(x^2)
Complementary[D] error	erfc	erfc(x)	(-2/π^(1/2))*e^-(x^2)
Bessel[R] function of order 0	J_0	$J_0(x)$	$-J_1(x)$
Bessel[R] function of order 1	J_1	$J_1(x)$	$J_0(x)-J_1(x)/x$
Bessel[R] function of the second kind, order 0	Y_0	$Y_0(x)$	$-Y_1(x)$
Bessel[R] function of the second kind, order 1	Y_1	$Y_1(x)$	$Y_0(x)-Y_1(x)/x$
Modified Bessel[R] function of order 0	I_0	$I_0(x)$	$I_1(x)$
Modified Bessel[R] function of order 1	I_1	$I_1(x)$	$I_0(x)-I_1(x)/x$
Modified Bessel[R] function, second kind, order 0	K_0	$K_0(x)$	$-K_1(x)$
Modified Bessel[R] function, second kind, order 1	K_1	$K_1(x)$	$-K_0(x)-K_1(x)/x$
Bessel[D] function of order n	J	J{n}(x)	-J{n+1}(x)+n*(J{n}(x)/x)
Bessel[D] function of the second kind, order n	Y	Y{n}(x)	-Y{n+1}(x)+n*(Y{n}(x)/x)
Modified[D] Bessel function of order n	I	I{n}(x)	I{n+1}(x)+n*(I{n}(x)/x)
Modified[D] Bessel function, second kind, order n	K	K{n}(x)	-K{n+1}(x)+n*(K{n}(x)/x)
Legendre polynomial[R]	P	P{n}(x)	((n+1)/(x^2-1))*(P{n+1}(x)-x*P{n}(x))
Legendre polynomial[R] of the second kind	Q	Q{n}(x)	((n+1)/(x^2-1))*(Q{n+1}(x)-x*Q{n}(x))
Associated Legendre polynomial[R]	P	P{n m}(x)	(((n+1)-m)*P{n+1 m}(x)-((n+1)*x)*P{n m}(x))/(x^2-1)
Associated Legendre polynomial[R] of the second kind	Q	Q{n m}(x)	(((n+1)-m)*Q{n+1 m}(x)-((n+1)*x)*Q{n m}(x))/(x^2-1)

[13] Для всех функций, отмеченных символом '[D]', могут быть вычислены только символьные выражения производной, вычисление значения функции не поддерживается.

