

数式表現 2 次元数式言語 MatEx-2D の パースと描画処理 詳細設計書

(株) 数理システム
齋藤宗香

2004 年 02 月

目 次

1	概要	2
2	語句の定義	2
2.1	記述ブロックと記述ユニット	2
2.2	ベースライン	3
2.3	数式オブジェクト	3
2.4	親数式・子数式	4
3	テキスト 数式記述の読み込み	4
3.1	入力後のデータ構造	4
3.1.1	ベースラインの調整	5
3.1.2	擬似平面化	5
3.2	パース	6
3.2.1	数式オブジェクトパーサ	6
3.2.2	全体の流れ	7
3.2.3	パーサクラス	7
4	数式	8
4.1	数式クラス	8
4.2	数式の入れ子構造	9
4.3	親となる数式クラスと子数式の保有	9
4.3.1	数式を内包する数式クラス	9
4.3.2	演算子数式を連結する数式クラス	10
4.3.3	単純に数式動詞を連結するクラス	10

4.3.4	ベキと添字クラス	11
4.3.5	その他の親となる数式	12
5	処理の流れ	13
6	インターフェース	13
6.1	パース処理入力	13
6.1.1	記述平面	13
6.1.2	インターフェース	13
6.2	パース処理出力	14
6.3	構造化数式出力	14
7	Window における数式出力	15
7.1	Windows における描画について	15
7.2	描画に用いるクラス	15
7.3	数式の描画	17
7.3.1	フォントを使用して数式オブジェクトを描画するクラス	17
7.3.2	数式オブジェクトを描画するクラス	18
7.3.3	画像を使用して数式オブジェクトを描画するクラス . .	18
7.3.4	子数式の配置を決定して、描画をするクラス	19
7.3.5	まとめ	19

1 概要

この文書では、MatEx-2D Tools における数式記述パース処理と描画処理
についての実装詳細を記述する。

数式表現 2 次元数式言語 MatEx-2D による数式表現は「数式表現 2 次元言
語 MatEx-2D マニュアル」を参照のこと。

2 語句の定義

この章では、この文書内で使用する語句の定義を行う。

2.1 記述ブロックと記述ユニット

i) 記述ブロック

1 つの数式を表している記述単位を表す

ii) 記述ユニット

1 つの数式の一部、あるいは全体を表している記述単位を表す

たとえば

```
1
2  a = b
3
4 .....
5
6  = c
7
8 #####
9
10 x = y
11
```

のようなテキスト数式記述があったとする。各行の先頭の数字は行番号を
表す。この記述の場合、1-7, 9-11 はブロック、1-3, 5-7, 9-11 はそれぞれユニッ
トである。

2つの関係を UML では



のように記述できる。

2.2 ベースライン

数式の‘中心’となる記述行のことをベースラインという。テキスト数式記述は、ベースラインを辿ることによって数式の基本的な構造が解析できる記述である必要がある。

ベースラインは、記述ブロックにおいて最初の可視文字が最も左側にある記述行である。

2.3 数式オブジェクト

1つの数式は複数の数式から構成される。ここでは、1つの数式、数式の最小単位、またはそれに準ずるものを数式オブジェクトと呼ぶ。

たとえば、

$$\sum_{i=0}^n a_i$$

の場合、

$$i = 0, n, a_i$$

はそれぞれ数式であるので、数式オブジェクトである。ここでは便宜上 Σ も数式オブジェクトとして扱っている。さらに細分化すると、

$$\Sigma, i, =, 0, n, a, i$$

となり、これらが最小単位の数式オブジェクトとなる。

2.4 親数式・子数式

親数式と子数式は数式同士の保有関係を意味する。親数式は子数式を保有する。

3 テキスト 数式記述の読み込み

テキスト数式記述は、1 つあるいは複数の記述ユニットによって 1 つの数式が表現される。パーサでは 1 つの数式記述を平面として処理を行うので、それに応じた実装を行う。

3.1 入力後のデータ構造

テキストは入力後、次のような階層構造になっている。

i) テキスト数式記述

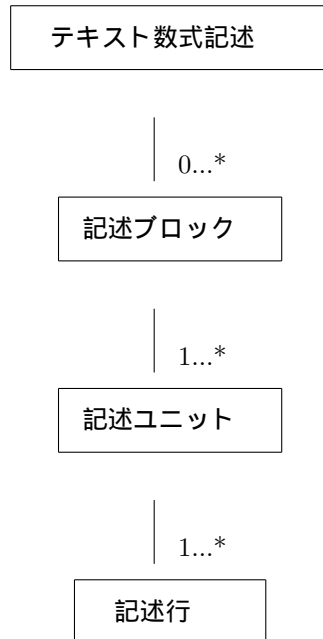
テキスト数式記述は、0 以上の記述ブロックを保持している。

ii) 記述ブロック

記述ブロックは、記述のユニットを双方向リストによって保持する。

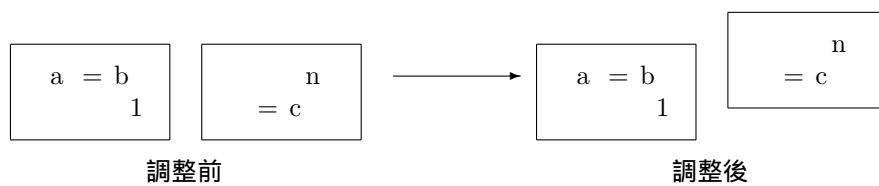
iii) 記述ユニット

記述ユニットは複数の行によって構成される。



3.1.1 ベースラインの調整

擬似平面化の際に記述ユニットを左右に連結し、1つの平面のように見せる必要がある。それぞれのユニットでベースラインが異なる場合があるので、ベースラインが一致するようにユニット間のインデックスを調整する。



3.1.2 擬似平面化

さらに記述ブロックを擬似的に平面化し、パース時に記述ユニットを意識させない。擬似平面化インターフェースは `ScriptBlockWalker` クラスである。

```

class ScriptBlockWalker
{
public:
    virtual ~ScriptBlockWalker() {}

    virtual char    element(int x, int y) = 0;
    virtual int     width() = 0;
    virtual int     height() = 0;
    virtual int     totalLineIndex(int y) = 0;
};

```

記述ブロックの平面化を実装しているのは、StringBlockWalker クラスである。

3.2 パース

記述ブロック全体をパースすることにより、1つの構造化された数式を作成する。

3.2.1 数式オブジェクトパーサ

各数式オブジェクトの記述には対応したパーサがあり、各パーサは記述を解析し記述に応じた数式オブジェクトを作成する。またパーサは、作成した数式オブジェクトと保有関係にある数式の記述領域を切りだす。

パース処理の入出力は以下のとおり。

[入力]

- 数式ブロック
- パースの対象となる記述領域
- パース開始点 (x, y)

[出力]

- 数式オブジェクト
- パースを行った記述領域
- 子数式または親数式のパース領域

3.2.2 全体の流れ

パースの際には、以下のように数式記述を解析し、記述内容に対応した数式オブジェクトを作成する。

1. ベースラインを見つける。
- i. ベースラインを辿り、最初の記述に対応した数式オブジェクトパーサを得る。
- ii. パースを行い、数式オブジェクトを得る。
- iii. 添字記述領域をパースする
- iv. 上記の処理でパースを終えた領域を設定する。

3.2.3 パーサクラス

各数式記述に対応したパーサは、以下の EquationParser クラスを継承する。

```
class EquationParser
{
    virtual Equation *parse(ScriptBlockIterator &itr, ParseRange *target,
                           ParseRange *parsed) = 0;
};
```

parse メソッドにそれぞれの記述のパース処理を実装する。

6.2.4 パース時の記述の優先順位

数式パースの際に、以下のような記述解析の優先順位がある。

- 1) 改行表示記述 2) 数式アライン記述 3) 括弧 4) 比較演算子記述 5) 2 項演算子 6) その他

6.2.5 数式パースの例

パースの過程を例を挙げて説明する。下の図の矩形内部がテキスト数式記述である。矩形の外側の数字は、縦が行番号、横が列番号である。

```
[[R
1234567890123456
-----
1|          1          |
2|  A =  -----      |
3|          2          |
-----
```


上記の数式記述に対してパース処理を行うと、

- 1) ベースラインを見つける。記述の開始位置が一番左にある行をベースラインとし、式の縦方向の中心があるとみなす。この例では、2行目がベースラインとなる。
- 2) (5,2) の '=' が最も優先順位が高いので、これに対応したパーサが生成される。
- 3) パーサにより '=' 演算子オブジェクトが生成される。パーサは '=' の左の (1,1)-(4,3) の領域と、右の (7,1)-(16,3) の領域を切り出す。また、演算子連結オブジェクトが生成され、 '=' 演算子オブジェクト・左右の領域において生成される数式オブジェクトを自分自身と結びつける。
- 4) (1,1)-(4,3) の領域をパースする。変数パーサが生成され、 'A' という名前の変数オブジェクトが生成される。
- 5) (7,1)-(16,3) の領域をパースする。分数式パーサが生成される。
- 6) 分数式パーサは '—' の記述の上下を切り出す。
- 7)

4 数式

4.1 数式クラス

数式クラスは、数式を表す基本クラスである。数式クラスの概略は以下のようになっている。

```
class DrawContext;
class Equation
{
public:
    virtual void    draw(DrawContext *context = 0); // 描画

    virtual Size *  size(DrawContext *context);      // 数式
    の大きさ
    virtual char *  tag()                            // 数式
    の名前
};
```

各数式クラスはこのクラスを継承する。DrawContext は出力の状態を保持するクラスであり、その内容は描画処理の実装に依る。

4.2 数式の入れ子構造

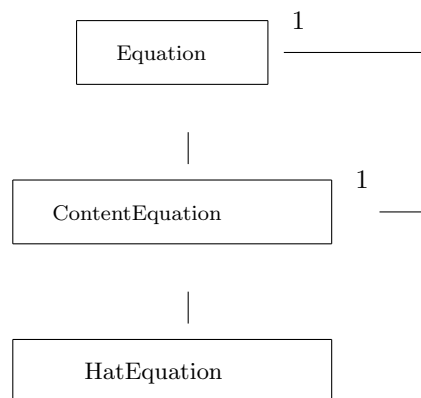
各記述に対応したパーサでパースする範囲がすべて数式となるようにする。よって、ブロック内の記述全体が 1 つの数式であり、それが複数の数式を多階層で保持した、数式の入れ子構造 になっている。

4.3 親となる数式クラスと子数式の保有

数式は入れ子構造になっているが、下の階層の数式をどのように保持するかは各数式により異なる

4.3.1 数式を内包する数式クラス

たとえば、 $A \hat{+} B$ のように括弧によって他の数式を内包している数式の場合、`\hat` を表わす HatEquation は、1 つの数式クラスを保持するクラス ContentEquation を継承している。`\hat` から括弧まで含めた式全体で 1 つの数式とみなし、ContentEquation クラスは Equation クラスを継承している。

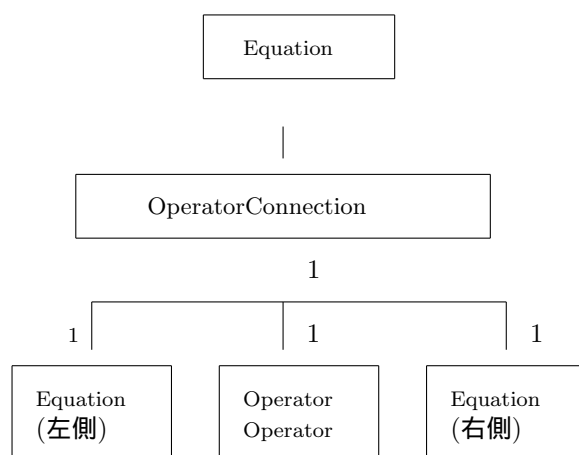


4.3.2 演算子数式を連結する数式クラス

数式と数式を連結し、それを 1 つの数式とするクラスがある。たとえば

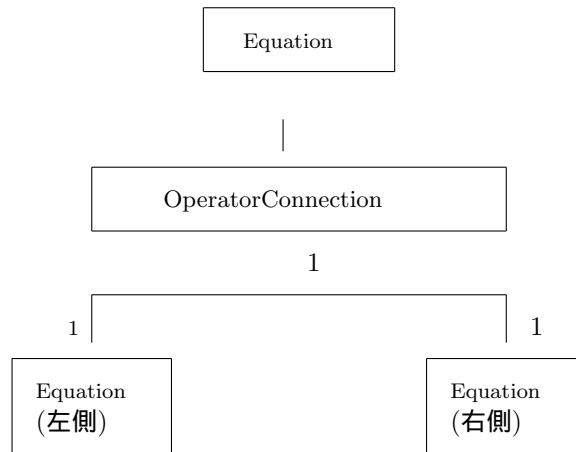
$$a = b$$

は a と b が 演算子 $=$ によって連結されている。この場合、OperatorConnection が作成され、それが左右の数式・演算子を保持する。また、それ全体が数式であるので、OperatorConnection クラスは Equation クラスを継承している。



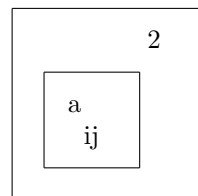
4.3.3 単純に数式動詞を連結するクラス

2π など、間に特別な数式がない場合に数式同士を連結する数式クラスが NoOperatorConnection クラスである。

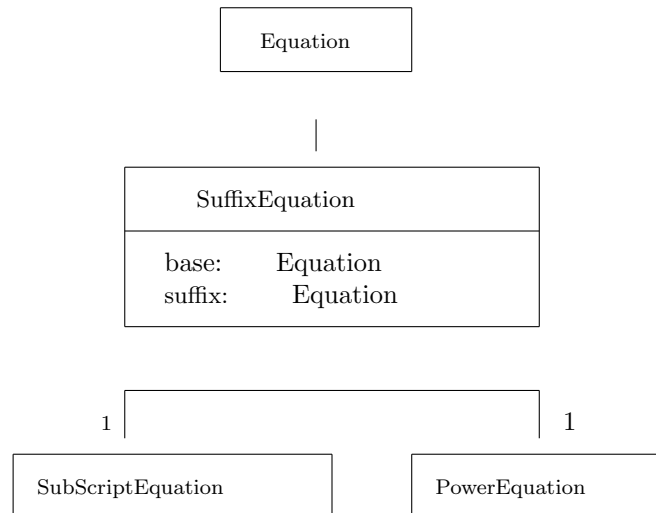


4.3.4 ベキと添字クラス

a_{ij}^2 のような添字とベキが一度に使われている場合、 a に添字が付き、その上にベキが付く構造になっている。



ベキクラス (PowerEquation) ・ 添字クラス (SubScript) とともに SuffixEquation を継承しており、ベースとなる記述と、添字またはベキをそれぞれ数式として保持している。



4.3.5 その他の親となる数式

その他、以下の親となる数式クラスがあり、それぞれの記述に応じたデータ構造を持っている。

[MatrixStructureEquation クラス]

行列構造をもつ数式クラス

[SuperSubScriptEquation クラス]

数式の上下と右側にある数式を内包する数式クラス

[IntegralEquation クラス]

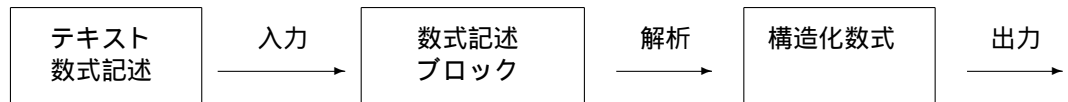
上限・下限を表す式と右側にくる式を内包する数式クラス

[EquationEnumeratorEquation クラス]

1 つ以上の数式を保持し、それらに対して番号付けを行う数式クラス

5 処理の流れ

数式パースの全体の処理の流れは以下のとおりである。



6 インターフェース

6.1 パース処理入力

6.1.1 記述平面

パース処理では記述ブロックが 1 つのテキスト平面であるとして処理を行う。入力データの構造がそのままでは平面的ではない場合、平面をエミュレートする実装を行わなければならない。

6.1.2 インターフェース

記述平面を表すクラスとして `ScriptBlockWalker` クラスが用意されている。パース処理はこのインターフェースクラスを介して数式記述をにアクセスする。

```
1 class ScriptBlockWalker
2 {
3 public:
4     virtual ~ScriptBlockWalker() {}
5     virtual char    element(int x, int y) = 0; // 記述文字へのアクセス
6     virtual int     width() = 0;              // 記述の幅
7     virtual int     height() = 0;             // 記述の高さ
8};
```

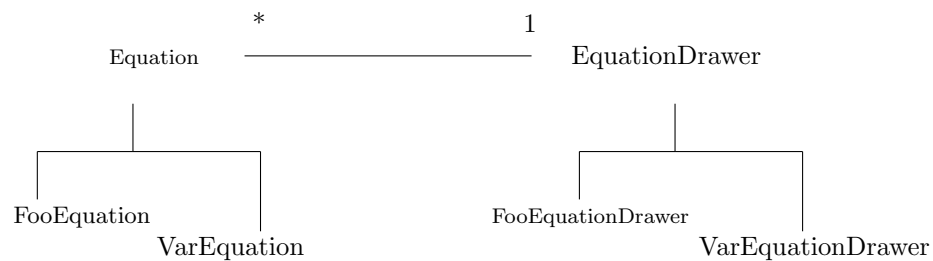
6.2 パース処理出力

パース処理の結果として、1つの構造化された数式を出力する。

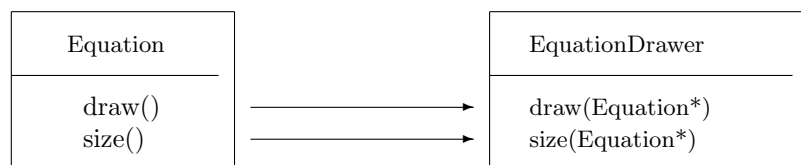
6.3 構造化数式出力

1) 数式描画オブジェクト

数式オブジェクトには対応する描画オブジェクトがある。



数式オブジェクトは描画などの各数式に固有な処理を描画オブジェクトに委譲する。

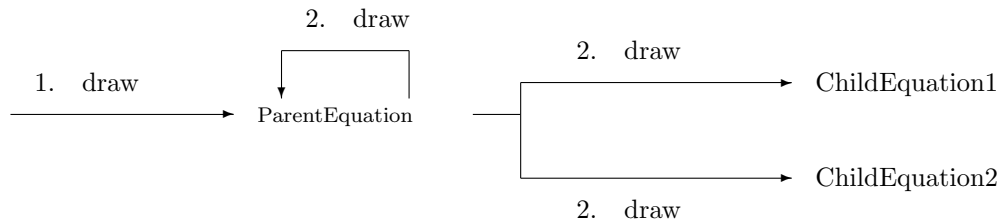


このようにして出力処理をデータ構造から分離・抽象化し、描画オブジェクトを切り替えることによって、様々な形式の出力処理の実装が可能になる。

同じ種類の数式オブジェクトインスタンスは、共通の描画オブジェクトによって描画される。よって、描画処理において副作用があってはならない。

2) 親数式の子数式に対する描画責任

親数式は子数式の描画処理を負う。



親数式が draw メソッドを呼び出されたとき、自身の内容を出力すると共に、子数式に対して適切な設定をした上で描画する。

7 Window における数式出力

7.1 Windows における描画について

できるだけフォントにある文字・記号を使用するが、ないものは画像で代用する。

7.2 描画に用いるクラス

1) DrawContext クラス

Windows での数式描画において、以下のような DrawContext を使用する。

```

class DrawContext
{
public:
    ViewBuffer * buffer; // 描画バッファ
    DrawPoint   point;   // 描画位置
    DrawLevel   level;   // 描画レベル
    bool        margin;  // 数式の周辺にマージンを入れるか？
    bool        tag;     // タグを抽出するか？
    bool        alignSet; // アライン位が置設定されているか？
    Align       align;    // アライン位置

    EquationSizeManager * sizeMgr; // 数式サイズマネージャ
    FontManager *         fontMgr; // フォントマネージャ
  
```



```

        HighlightSwitch *      highlightSwitch; // ハイライトスイッチ
};

```

2) DrawPosition クラス

位置を指定するクラス。

設定した場所の位置 (開始・中心・終了) によらず、オブジェクトの開始・中心・終了それぞれの位置を取得できるクラス。

```

class DrawPosition
{
public:
    void      setStart(int);
    void      setCenter(int);
    void      setEnd(int);

    int       start(int);
    int       center(int);
    int       end(int);
    int       shift(int);
    int       start(Length&);
    int       center(Length&);
    int       end(Length&);
};

```

a) 位置の指定 (setStart, setCenter, setEnd)

位置を引数にとり、その位置が始点・中心・終点のどれにあたるかをそれぞれ指定する。

b) 位置の取得 (start, center, end)

オブジェクトの大きさを引数とし、オブジェクト配置の始点・中心・終点をそれぞれ得る。

オブジェクトの大きさとしては int 値と Length 値を与えることができる。両者の違いは、

[int 値の場合]

始点と終点の真中が中心点となる。

[Length 値の場合]

始点と終点間の任意の点を中心として指定することができる。

3) DrawLevel 型

数式オブジェクトの相対的な大きさを表す型。添字・べきの表示では、値を上げ、`$\frac{\quad}{\quad}$などの表示では値を下げる。DrawLevel 型の値のインクリメント・デクリメントには、DRAWLEVEL_INCR、DRAWLEVEL_DECR マクロを使用する。`

4) EquationDCDrawer

Windows 数式描画クラス。Windows における数式描画クラスのベースクラス。

5) FontManager クラス

フォントの生成・管理を行うクラス。

6) EquationSizeManager クラス

数式の大きさを管理するクラス。

7) HighlightSwitch クラス

Highlight の切替を行うクラス。

8) ImageObject クラス

画像クラス。

9) EquationImageFactory クラス

画像を使用した数式描画の際に使用する ImageObject を管理するクラス。

7.3 数式の描画

数式の描画処理内容は、次のように分類できる。

7.3.1 フォントを使用して数式オブジェクトを描画するクラス

EquationCharacterDCDrawer クラス を継承する。数式オブジェクトを表す文字列と、そのフォントを得るメソッドを実装する。EquationCharacterDCDrawer クラスは以下のようにになっている。

```
class EquationCharacterDCDrawer : public EquationDCDrawer
{
protected:
    virtual HFONT getFont(DrawContext*)=0;
    virtual char *toString(Equation*)=0;
```

```

public:
    void    draw(Equation*, DrawContext*);
    void    size(Equation*, DrawContext*, int&, int &);
};

```

7.3.2 数式オブジェクトを描画するクラス

、' 'など、フォントや画像を使用すると都合が悪い場合、図形を組み合わせで描画する。EquationImageDCDrawer クラスを継承し、drawImage メソッドに描画処理を実装する。

```

class EquationImageDCDrawer : public EquationDCDrawer
{
protected:
    virtual void drawImage(Equation *, DrawContext*, int x0, int y0, int width, int height);
public:
    void    draw(Equation *, DrawContext*);
    virtual void    size(Equation *, DrawContext*, int &width, int &height);
};

```

7.3.3 画像を使用して数式オブジェクトを描画するクラス

EquationImageObjectDCDrawer クラスを継承する。

```

class EquationImageObjectDCDrawer : public EquationImageDCDrawer
{
private:
    class Layout{};
protected:
    virtual ImageObject *image(DrawContext*)=0;
private:
    void drawImage(Equation*,DrawContext*,int,int,int,int);
};

```

通常 EquationImageFactory に画像を登録し、EquationImageObjectDCDrawerTmp テンプレートクラスを使う。

EquationImageObjectDCDrawerTmp テンプレートクラスは

```
template<EquationImageFactory::ImageType TYPE>
class EquationImageObjectDCDrawerTmp : public EquationImageObjectDCDrawer
{
private:
    ImageObject *image(DrawContext *context)
    {
        return EquationImageFactory::get(TYPE,context->highlightSwitch);
    }
};
```

のようになっており、

```
class EpsilonGreekCharacterDCDrawer
    : public EquationImageObjectDCDrawerTmp<EquationImageFactory::IMAGE_EPSILON>
{};
```

のように、テンプレート引数に画像タイプを指定して image メソッドを実装する。

7.3.4 子数式の配置を決定して、描画をするクラス

子数式の描画のみを行い、それ以外の描画処理を行わない。

7.3.5 まとめ

ほとんどの数式描画クラスは上記のカテゴリーに分類され、それぞれのベースクラスを継承し、それに応じた描画処理を実装している。これらのベースクラスの継承関係は以下のようにになっている。

