Text Parser Library

Ver. 1.2

Program Description

Center of Research on Innovative Simulation Software
Institute of Industrial Science
the University of Tokyo

http://www.iis.u-tokyo.ac.jp/

November 2012

目次

図目次

1 概要

C/C++,Frotran などで記述されたシミュレーションソフトウェアにおいて, パラメータを記述, 取得する機能を提供するクラスライブラリを構築する. ユーザが指定するアスキーファイルに記述されたパラメータを読み込み, クラスライブラリ内部にそのパラメータを保持し, シミュレータから取り出せる機能を提供する. さらに, マルチプラットホームで動作することにより, 多くのシミュレーションプログラムでの利用が可能になる.

2 プログラムの構造

本プログラムは機能的には主に以下の部分から構成される. 図 1 参照.

1 パラメータを階層的に保持するためのパラメータデータ構造
2 ファイルからパラメータを入力しパラメータデータ構造へ格納
3 パラメータの取得及び型変換
4 パラメータのファイルへの書き出し
5 パラメータデータ構造の破棄

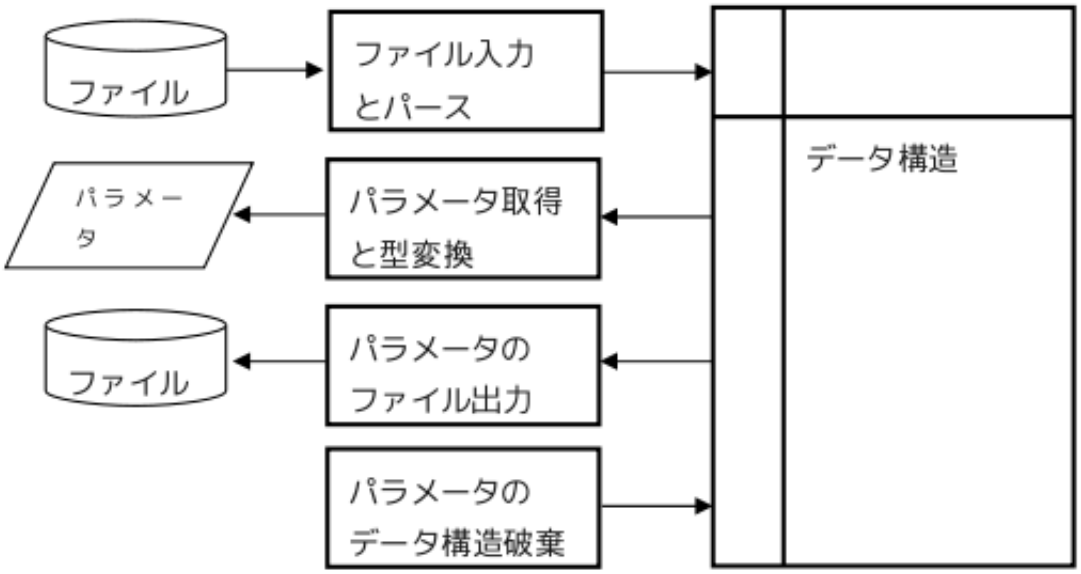

図 1 プログラムの構造

又, 本プログラムは構造的にはデータ構造に直接アクセスするクラスライブラリ部分とユーザーがアクセスするためのインターフェース部分とからなる.

3 パラメータデータ構造

パラメータのデータ構造は主に次の要素 (クラス) から成り立っている.

3.1 データベース本体 (TextParserTree クラス, ソース:TextParserTree.h, TextParserTree.cpp)

パラメータデータベースの基本となるクラスで唯一のスタティックなインスタンスを有する (シングルトン). 本プログラムが有する機能は最終的に殆どこのクラスの関数により行なわれる. これ自体ルートノード "/" の意味合いが有り, 複数の子ノードを持つことが出来る. 又, ルートノードは子ノードの配列形式ラベルの配列添え字の基点となるため, 使用されている配列形式ラベル名とその使用回数が保存される.

3.2 エレメント (TextParserElement クラス, ソース*TextParserElement.h, TextParserElement.cpp)

データベースの要素を表す. TextParserNode,TextParserLeaf,TextParserValue の基本クラスで, これらに共通なラベルや要素のタイプ, 親の要素等のデータを有する.

3.3 ノード (TextParserNode クラス, ソース:TextParserElement.h, TextParserElement.cpp)

パラメータデータ構造を階層的に構築するための要素で, 複数の子ノードと複数のリーフによって構成される. 又, ノードは配列形式ラベルの配列添え字の基点となるため, 使用されている配列形式ラベル名とその使用回数が保存される.

3.4 リーフ (TextParserLeaf クラス, ソース:TextParserElement.h, TextParserElement.cpp)

実際のパラメータを定義するための要素で, ラベル (パラメータの名前) と値によって構成される. リーフが持つ値は一つ又は複数 (ベクトル値) である.

3.5 値 (TextParserValue クラス, ソース:TextParserElement.h, TextParserElement.cpp)

値を保持する要素で値のタイプと値の文字列で構成される. 値のタイプには数値も文字列として保存される. 依存関係付き値も条件式に従って最終的に文字列として保存される.

4 パラメータの入力及びデータ構造への格納

パラメータの入力及びデータ構造への格納は TextParserTree::readParameters 関数で行なう. ユーザーは TextParaser クラスのインターフェース関数を介してこの機能を使用出来る. TextParserTree::readParameters 関数内部では, std::getline 関数によりファイルから解析可能な文字 (解析結果が確定する文字) が出現するまで一行ずつ読み込み,TextParserTree::parseLine 関数で解析してノードとリーフをデータ構造へ格納する (パラメータのパース).
尚, MPI 対応版では, rank 数 0 のプロセスのみがパラメータファイルの読み込みを行い, 他のプロセスへ読み

込んだ内容を MPI_Bcast で各プロセスに通信する．全てのプロセスが, パラメータファイルを読み込んだ内容に基づいてパースを行い, プロセスそれぞれがパース後の内容をデータ構造へ格納する．
リーフの出現順序は任意なので依存関係付き値をパースする時点で定義されていないラベルを参照する可能性がある．その場合未解決の依存関係付き値として,TextParserTree::_unresolved_leaves に保存し, 全てのファイルを読み込んだ後に再帰的に未解決の依存関係付き値を解決する．

4.1 パラメータのパース

パラメータのパースは以下の 4 つのモードが有り, 状況に応じて遷移する．

- ノードパースモード (TextParserParseMode::TP_NODE_PARSE)
- リーフパースモード (TextParserParseMode::TP_LEAF_PARSE)

4.1.1 ノードパースモード

ノード内の要素をパースするモードで, ラベルの検出と, 検出されたラベルに続く要素がノードかリーフかを解析する．図 2 参照.

ノードを閉じると親のノードパースモードに戻る．ノード又はリーフを開く際に親のノード (ルートノードを含む) を処理後の戻り先スタック (TextParserElement::_return_elements) の最後に格納する．ノード又はリーフを閉じる時には戻り先スタックの最後に格納されているノードに移動して, それをスタックから削除する．

この仕組みにより, 以下の様に相対的な処理の途中で絶対パスや相対パスにより直接他のノード内のノードやリーフが定義されても, その処理後に元のノードに戻れるようになっている. 以下はパラメータ記述の例とノードパースモードの動きを示してある．

```
foo {                   // ノード foo を開く
        qux = 1         // リーフ /foo/qux に 1 を設定する
        /bar/baz = "val1"     // ノード /bar を追加
                                // リーフ /bar/baz の戻り先スタック
                                // に /foo を追加して val1 を設定.
                                // 戻り先スタックの最後のノード
                                // /foo に移動する.
                                // リーフ /bar/baz の戻り先スタックの最
                                // 後の要素 /foo を削除する.
        switch = "on"          // リーフ /foo/switch に on を設定する
}                               // ノード foo を閉じる
```

4.1.2 リーフ・パースモード

リーフの右辺 (“=” の右部分) を解析するモードで, パラメータの値の種類に応じてデータ構造に保存する．また, 依存関係付き値に関しては一度文字列として値を保存してから解析する．ベクトル値は空白やコメントを除いて文字列として保存する．リーフを閉じると親のノードパースモードに戻る．また、値範囲指定

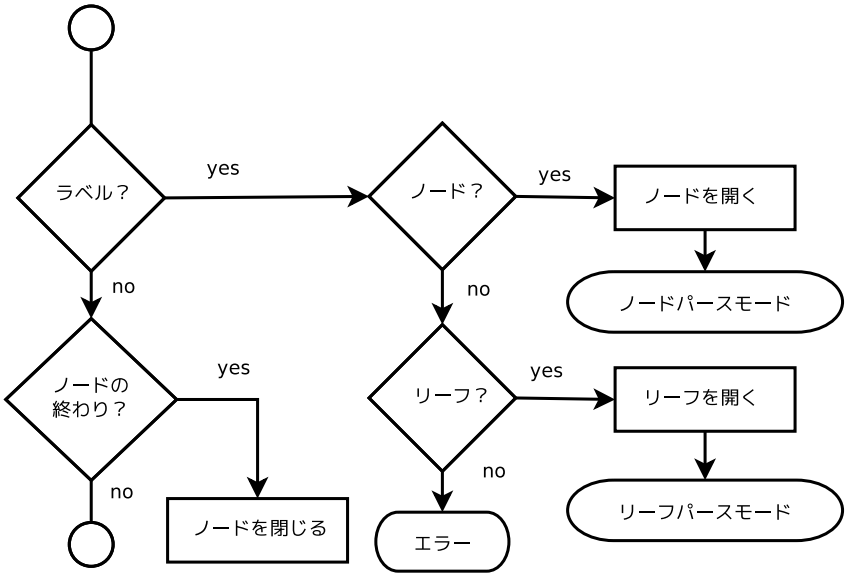

図 2　ノードパースモード状態遷移

(@range,@list) に関しても、リーフ・パースモードで解釈を行う。図 3 参照.

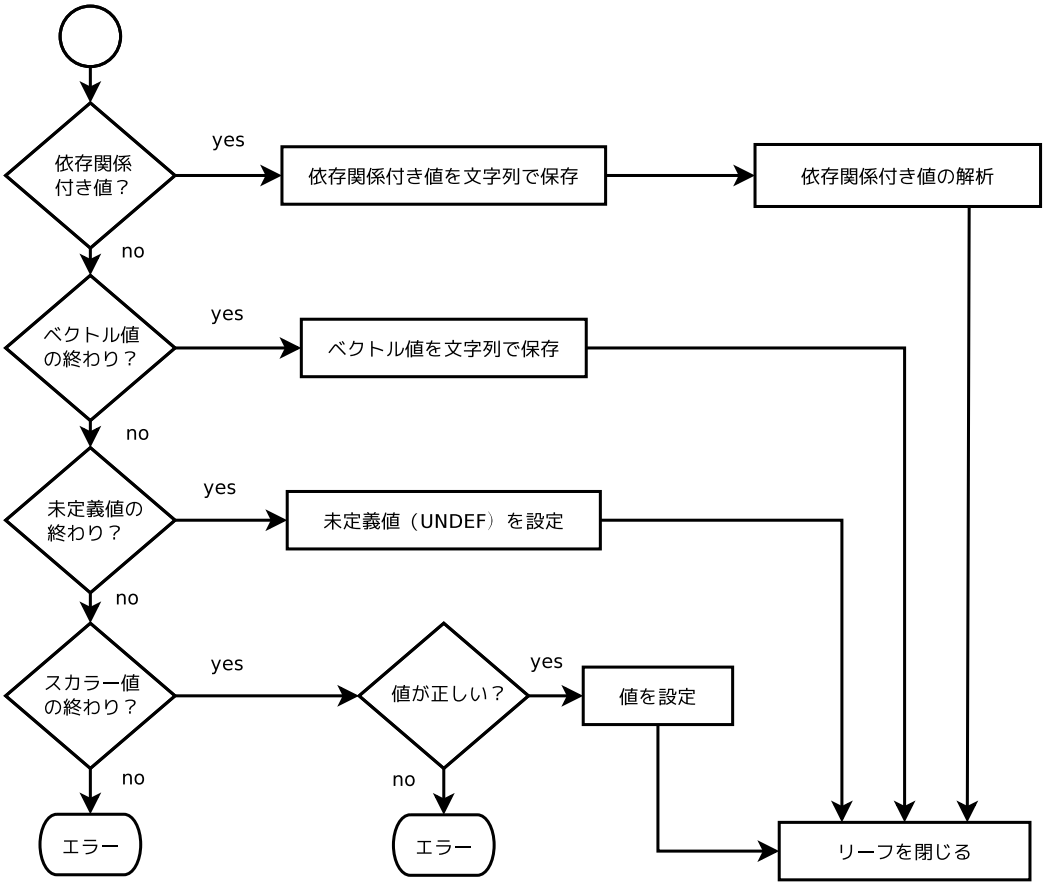

図 3　リーフパースモード状態遷移

4.2 依存関係付き値の解析 (TextParser::parseDependenceExpression 関数)

依存関係付き値は,C 言語の三項演算子と似ており条件式が真なら値 1 を偽なら値 2 を設定する.

ラベル = 条件式 ? 値 1 : 値 2

条件式を解析してその真偽を判定し, 条件式が真なら値 1 , 偽なら値 2 を設定する. 図 4 参照.

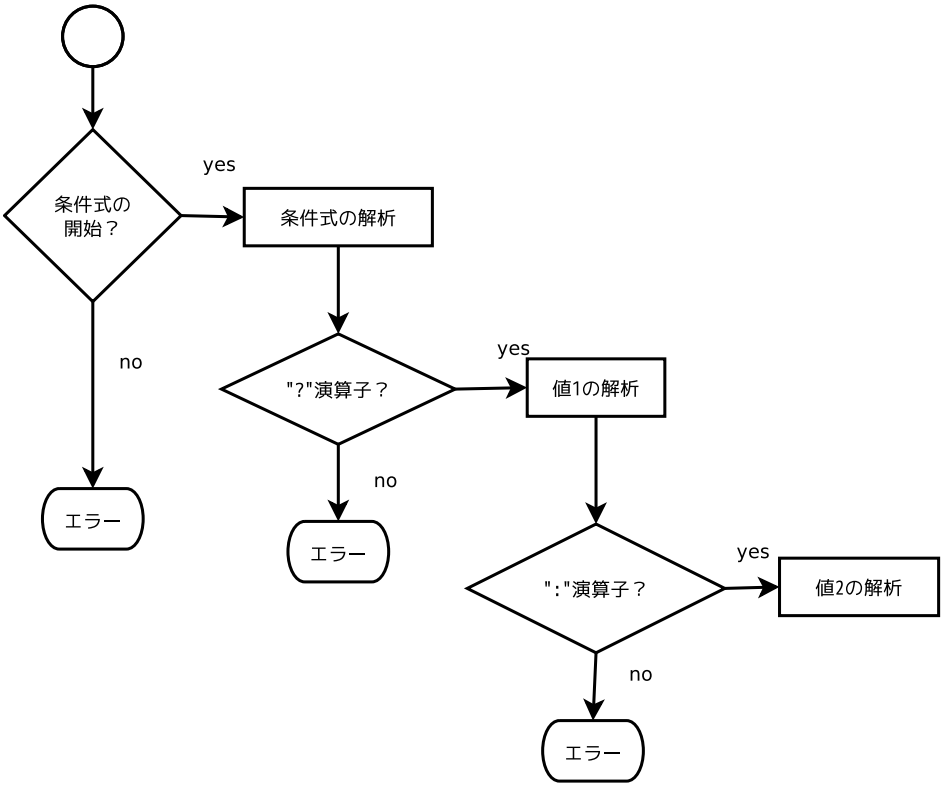

図 4 依存関係付き値の解析遷移

4.3 条件式の解析 (TextParser::parseConditionalExpression 関数)

条件式が真か偽かを (), &&, || 演算子に応じて再帰的に解析する. 図 5 参照.

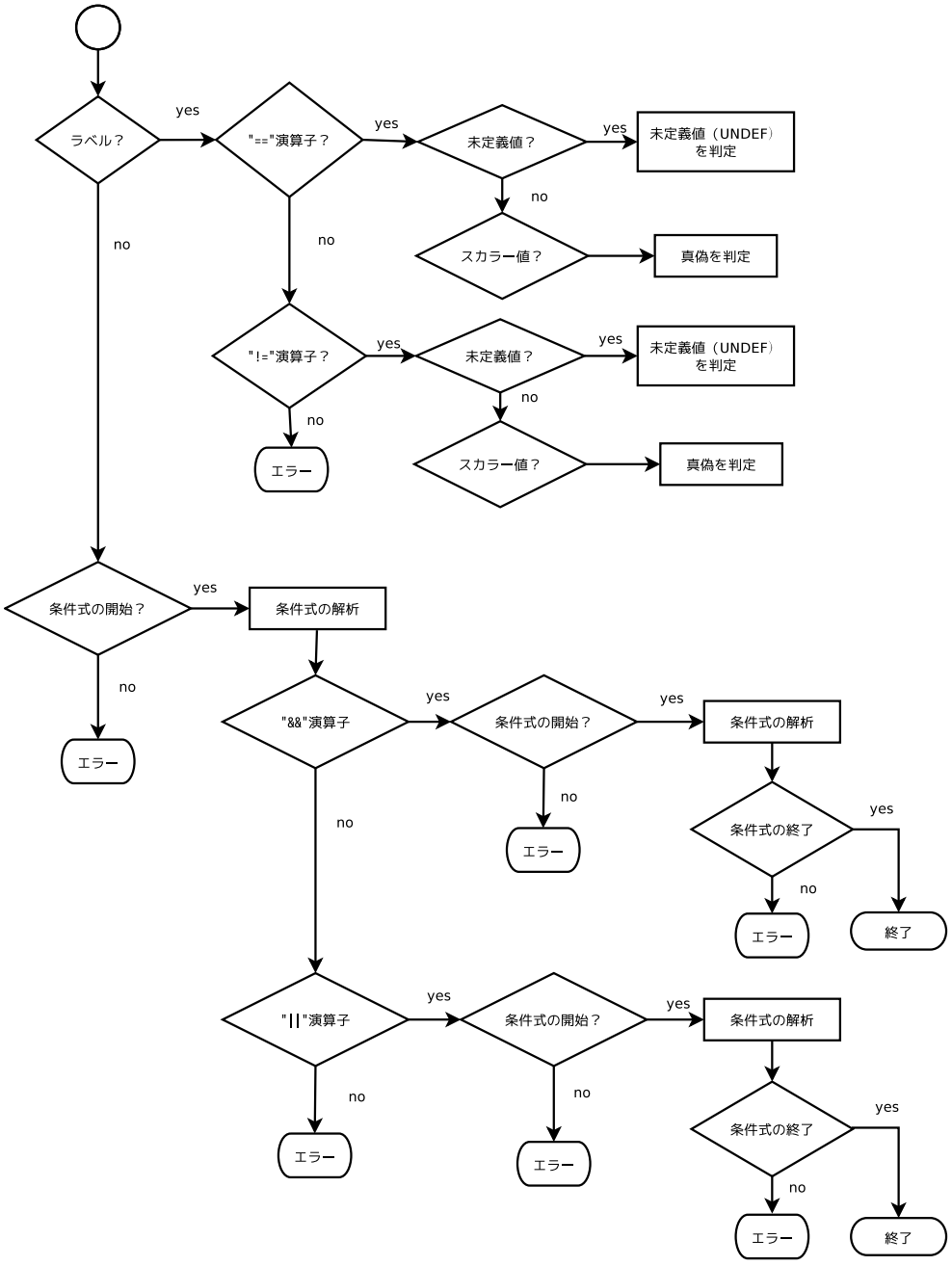

図 5　依存関係付き値の条件式解析遷移

4.4 値１, 値２の解析 (TextParserTree::parseDependenceValue 関数)

依存関係対き値の値１又は値２を解析し設定する．実際に値が設定されるのは条件式が真の場合は値１, 偽の場合は値２だけである．図 6 参照.

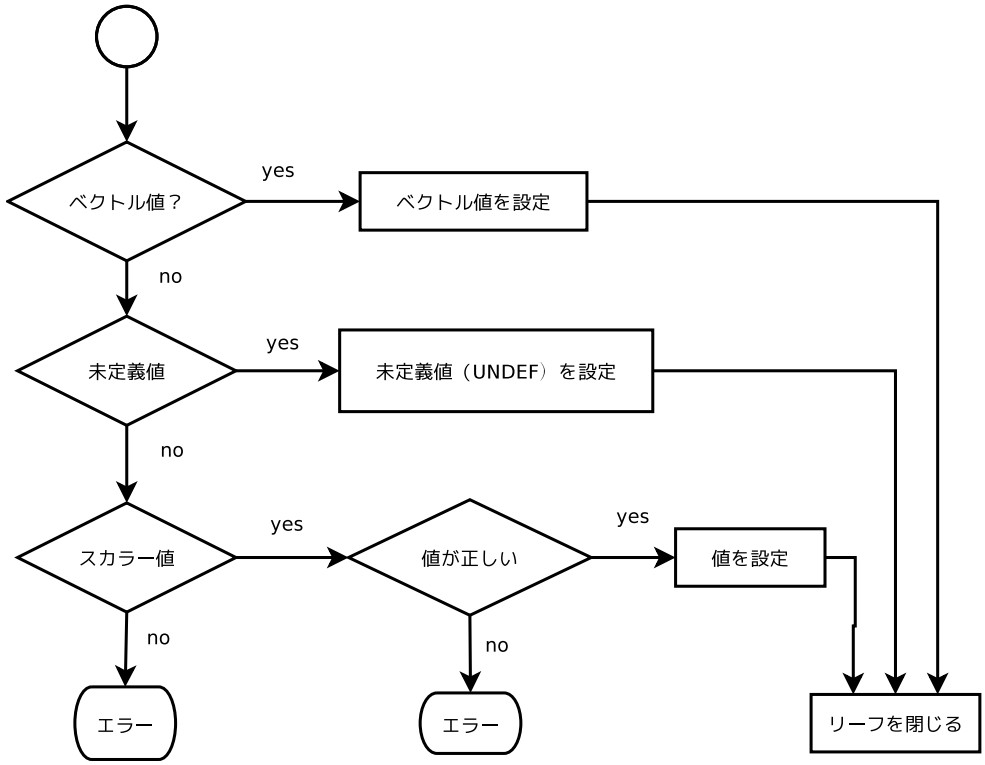

図 6　依存関係付き値の値設定遷移

4.5 真偽の判定 (TextParserTree::resolveConditionalExpression 関数)

条件式　(ラベル == 値) 又は (ラベル != 値) の真偽を判定する．条件式のラベルが既に定義されている場合はその保存されている値と条件式の値が等しいか否かを判定する．値が文字列の場合は大文字小文字を区別しないで比較する．数値の場合は std::stringstream により数値の文字列を double に変換して比較する．引数 is_equal が真の場合は等しければ引数 result に真 (TextParserBool::TP_TRUE) を設定し, 等しくなければ偽 (TextParserBool::TP_FALSE) を設定する．is_equal が偽の場合は逆の設定になる．条件式のラベルが定義されていない場合は未解決の依存関係付き値として, 未解決依存関係付き値リーフ (TextParserTree::_unresolved_leaves) に登録される．

4.6 未解決の依存関係付き値の解決

未解決の依存関係付き値は全てのデータを入力した後に上記の「依存関係付き値の解析」により再帰的に解決される．最終的に未解決の依存関係付き値が残った場合は Warning を表示して未定義値 (UNDEF) を設定

表 1　TextParserValueType で定義されている値

タイプ	値のタイプ
0	未定義値
1	数値
2	文字列
4	未定義値のベクトル
5	数値のベクトル
6	文字列のベクトル
7	値領域指定型 @range
8	値領域指定型 @list

する.

5　パラメータの取得と型変換

以下はパラメータの取得と型変換に関するクラスライブラリの関数である. ユーザーはインターフェース関数群を介してこれらの機能を使用出来る. パラメータの取得はラベルパスを指定して値を文字列の形で取り出す. ラベルパスの指定には相対パスと絶対パスが使用できる. 相対パスでのパラメータ取得を高速に行なうためにノードの移動機能を用意し, カレントノードを移動しながらパラメータの値を取得出来る様にした. 絶対パスで指定する場合はカレントノードに無関係にパラメータの値を取得できる.

(注) マルチスレッド化によりデータ構造に同時にアクセスする可能性がある場合はカレントノードが不定になるため, 絶対パスで指定する必要が有る.

5.1　パラメータの値の取得

5.1.1　パラメータの値の取得 (TextParserTree::getLeafValue 関数)

パラメータの値を個数分,TextParserValue クラスのインスタンスのポインタ配列として取得する. これにより値の数と値のタイプも取得できる. 取得できる値のタイプは文字列値と数値の二つである.

パラメータの値を TextParserValue クラスのインスタンスのポインタとして取得する. これにより値のタイプも取得できる. 値のタイプは enum TextParsarType で定義されている. 取得できる値のタイプは以下の通りである.

タイプ 4〜6 のベクトル値は TextParserTree::splitVectorValue 関数で分離することにより値の文字列を取得出来る.

5.2　ラベルの取得, ノードの移動

5.2.1　全リーフのラベルパス取得 (TextParserTree::getLeafLabels 関数)

データ構造の全てのリーフの絶対パスを std::string の配列として取得する.

5.2.2 ノードの移動 (TextParserTree::changeNode 関数)

データ構造内の任意のノードに移動し,TextParserTree::_current_element に設定する. 移動先のノードには絶対パスと相対パスを指定できる.

5.2.3 カレントノードのラベルパス取得 (TextParserTree::getCurrentNode 関数)

現在のノード (TextParser::_current_node) を GetElementAbsolutePath 関数により絶対パスに変換して返す.

5.2.4 カレントノード内のノード数を取得 (TextParserTree::getCurrentNodeNumber 関数)

現在のノード内のノード数を取得する.

5.2.5 カレントノード内の指定したノードのラベルを取得 (TextParserTree::getCurrentNodeLabel 関数)

現在のノード内の指定した ID のノードのラベルを char*のポインタとして取得する.

5.2.6 カレントノード内の全ノードのラベルを取得 (TextParserTree::getCurrentNodeLabels 関数)

現在のノード内の全てのノードのラベルを std::string の配列として取得する.

5.2.7 カレントノード内のリーフ数を取得 (TextParserTree::getCurrentLeafNumber 関数)

現在のノード内のリーフ数を取得する.

5.2.8 カレントノード内の指定したリーフのラベルを取得 (TextParserTree::getCurrentLeafLabel 関数)

現在のノード内の指定した ID のリーフのラベルを char*のポインタとして取得する.

5.2.9 カレントノード内の全リーフのラベル取得 (TextParserTree::getCurrentLeafLabels 関数)

現在のノード内の全てのリーフのラベルを std::string の配列として取得する.

6 パラメータの書き出し (TextParserTree::writeParameters 関数)

パラメータデータ構造のノードとリーフを階層的にファイルに書き出す.

7 データ構造の破棄 (TextParserTree::removeElement 関数)

読み込んだパラメータのデータ構造を削除し, メモリを開放する.

8 インターフェース

ユーザーのプログラムから呼び出すことが出来る関数群である. インターフェースには C++,C,Fortran90 用のインターフェースがある. C のインターフェースは C++ からも呼び出すことが出来る.

以下に C++ 用のユーザーインターフェイスを説明する.

8.1 C++ 用インターフェース

C++ 用のインターフェースは, クラス TextParser で定義されている. TextParser クラスの定義は, ヘッダファイル TextParser.h 内に記述してある. TextParser.h は, ライブラリをインストールすると, $prefix/include ディレクトリに格納されている. TextParser クラスを利用するには, このヘッダファイルをインクルードすればよい. ユーザーは, このクラスのインスタンスを作成し, インスタンスから定義されているメンバ関数を呼ぶことで, 次のことが可能である.

- パラメータのファイルからの読み込み
- パラメータのファイルへの書き出し
- パラメータデータの破棄
- パラメータへの絶対・相対パスの取得
- パスを使用したパラメータの値へのアクセス
- 文字列として格納している値の整数型や浮動小数点型への変換
- ベクトルして格納している値の文字列の各値への分離
- パラメータデータの編集

以下に各機能, クラス TextParser のメンバ関数を説明する.

8.1.1 インスタンスへのポインタの取得

TextParser クラスのインスタンスは, Singleton パターンによってプログラム内でただ 1 つ生成されるインスタンスと, ユーザーが通常のクラスの様に作成するものと両方が仕様出来る. その singleton インスタンスへのポインタを取得するメソッドは, 次のように定義されている.

インスタンスの生成, インスタンスへのポインタの取得

```
static TextParser* TextParser::get_instance_singleton();
```

戻り値 唯一の TextParser クラスのインスタンスへのポインタ

ユーザーの作成するプログラム内では, このメソッドで得られたインスタンスへのポインタを用いて, 各メンバ関数へアクセスするか, 通常のインスタンス化を用いて使用する.

8.1.2 パラメータの入力及びデータ構造への格納, データ構造のファイルへの出力

パラメータファイルを読み込み, 解析結果をデータ構造へ格納する関数, データ構造に格納したパラメータを, 全てファイルに書き出す関数は次のように定義されている.

パラメータのファイルからの読み込み

```
TextParserError TextParser::read(const std::string& file);
```

file　入力ファイル名
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

パラメータのファイルからの読み込み (MPI local file) 版

```
TextParserError TextParser::read\_local(const std::string& file);
```

file　入力ファイル名
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

ファイルへの書き出し

```
TextParserError TextParser::write(const std::string& file);
```

file　出力ファイル名
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

ファイルに書き出されるデータ構造は, すでにファイルから読み込まれ, 解析, 処理されたデータである. その為, 条件付き定義値は, ファイル読み込み終了時点で確定した条件による定義値になる. MPI 版の場合, read() は, rank 数 0 のプロセスでファイルを読み込んだ後, 内容を分配するが, read_local() は, 各プロセスがそれぞれ指定されたファイルを読み込む. 非 MPI 版で read_local() を呼び出した場合は, ファイルを読み込まず, エラーメッセージを表示し, エラーコードを返す.

8.1.3 全てのパラメータのラベルパス取得, パスを指定しての値及び値の型の取得

データ構造に格納しているパラメータ全てのリーフのパスを取得する関数, パスを指定してパラメータの値を取得する関数, パラメータの値の型を取得する関数はそれぞれ次の様に定義されている.

全てのパラメータへのパスの取得

```
TextParserError TextParser::getAllLabels(std::vector<std::string>& labels);
```

labels　パラメータ全てへのリーフパス
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

パスを指定して値を取得する

```
TextParserError TextParser::getValue(const std::string& label,
                                     std::string& value);
```

label　パラメータへのパス
value　パラメータの値
戻り値　エラーコード (=0: no error). TextParseError で定義される.

値の型を取得する

```
TextParserValueType TextParser::getType(const std::string& label,
                                        int *error);
```

label　パラメータへのパス
value　エラーコード (=0: no error). TextParseError で定義される.
戻り値　パラメータの値の型 TextParserType(表 1) 参照.

8.1.4 カレントノードの取得, 子ノードの取得, ノードの移動

カレントノードの取得, 子ノードの取得, ノードの移動は, 次の様に定義されています.

カレントノードの取得

```
TextParserError TextParser::currentNode(std::string& node);
```

node　カレントノードのパス.
戻り値　エラーコード (=0: no error). TextParseError 参照.

ノード内の子ノードのラベル取得

```
TextParserError TextParser::getNodes(std::vector<std::string> &
labels,int order=0);
```

labels　ノード内のリーフラベルのリスト（相対パス）
order　ラベルの並び順オプション（省略可能）. ユーザーズマニュアル参照のこと.
戻り値　エラーコード (=0: no error). TextParseError 参照.

ノード内のリーフラベル取得

```
TextParserError TextParser::getLabels(std::vector<std::string> &
labels,int order=0);
```

labels　ノード内のリーフラベルのリスト（相対パス）
order　ラベルの並び順オプション（省略可能）. ユーザーズマニュアル参照のこと.
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

ノード移動

```
TextParserError TextParser::changeNode(const std::string& label);
```

label　移動するノードのラベル（相対パス）
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

8.1.5 パラメータの値の特定の型への変換

パラメータの値 (文字列) を特定の型へ変換する関数が次の様に用意されている.

ノード移動

```
char TextParser::convertChar(const std::string value, int *error);
short TextParser::convertShort(const std::string value, int *error);
int TextParser::convertInt(const std::string value, int *error);
long TextParser::convertLong(const std::string value, int *error);
long long TextParser::convertLongLong(const std::string value, int *error);
float TextParser::convertFloat(const std::string value, int *error);
double TextParser::convertDouble(const std::string value, int *error);
bool TextParser::convertBool(const std::string value, int *error);
```

value　パラメータの値 (文字列)
error　エラーコード (=0: no error). `TextParseError` 参照.
戻り値　　パラメータの値をそれぞれの型に変換したもの.

8.1.6 ベクトル型パラメータの値の要素への分割

ベクトル型のパラメータの値を, 要素 (文字列) に分解する関数は, 次の様に用意されている.

ベクトル型パラメータの要素 (リスト) の所得

```
TextParserError TextParser::splitVector(const std::string& vector\_value,
                                        std::vector<std::string>& velem);
```

vector_value　ベクトル型パラメータの値 (文字列)
velem　各要素の値 (文字列)
戻り値　エラーコード (=0: no error). TextParseError 参照.

8.1.7 パラメータデータの破棄

格納しているパラメータのデータを破棄します.

パラメータデータの破棄

```
TextParserError TestParser::remove();
```

戻り値　エラーコード (=0: no error). TextParseError 参照.

8.1.8 パラメータデータの編集

格納しているパラメータのデータを編集します.

パラメータデータの削除

```
TextParserError TestParser::deleteLeaf(std::string& label);
```

label　削除するリーフのラベル
戻り値　エラーコード (=0: no error). TextParseError 参照.

パラメータデータの更新

```
TextParserError TestParser::updateValue(std::string& label,std::string& value);
```

label　更新するリーフのラベル
value　更新するリーフの値. ただし, リーフのタイプ (TP_VALUE_TYPE) を変更しない文字列にすること.
戻り値　エラーコード (=0: no error). TextParseError 参照.

パラメータデータの作成

```
TextParserError TestParser::createLeaf(std::string& label,std::string& value);
```

label　作成するリーフのラベル
value　作成するリーフの値　尚, 文字列の値の場合は, ダブルクオートで囲むこと.
戻り値　エラーコード (=0: no error). TextParseError 参照.

8.1.9 **値範囲指定記述用** API

一様値範囲指定時のパラメータ取得

```
TextParserError splitRange(const std::string & value,
   double *from,double *to,double *step);
```

value 値の文字列
from 開始値
to 終了値
step 増減値
戻り値 エラーコード (=0: no error). `TextParseError` 参照.

一様値範囲指定時のパラメータ展開

```
TextParserError expandRange(const std::string & value,
    std::vector<double>& expanded);
```

value 値の文字列
expanded 展開した数値のベクター
戻り値 エラーコード (=0: no error). `TextParseError` 参照.

任意数列での値範囲指定時のパラメータ展開

```
TextParserError splitList(const std::string & value,std::vector<double>& list,
  TextParserSortOrder order=TP_SORT_NONE);
```

value 値の文字列
list 展開した数値のベクター
order TextParserSortOrder で定義されている。 TP_SORT_NONE:記述順 TP_SORT_ASCENDING:昇順 TP_SORT_DESCENDING:降順
戻り値 エラーコード (=0: no error). `TextParseError` 参照.

表 2 TextParserSortOrder の定義値

TP_SORT_NONE	指定無し、記述順
TP_SORT_ASCENDING	昇順
TP_SORT_DESCENDING	降順

8.2 C **言語用インターフェース**

C 言語用のインターフェースは, API 関数が用意されていて TestParser.h に記述してある.

- パラメータのファイルからの読み込み
- パラメータのファイルへの書き出し
- パラメータデータの破棄
- パラメータへの絶対・相対パスの取得
- パスを使用したパラメータの値へのアクセス
- 文字列として格納している値の整数型や浮動小数点型への変換
- ベクトルして格納している値の文字列の各値への分離

以下に各機能, C 言語用 TextParser ライブラリの API 関数を説明する.

8.2.1 instance の生成, 消滅とシングルトンへのアクセス

C 言語内では TextParser のインスタンスを TP_HANDLE で扱う.

TextParser シングルトンインスタンスへのポインタを取得します.

```
TP_HANDLE tp_getInstanceSingleton();
```

戻り値 TextParser シングルトンインスタンスへのポインタ

TextParser インスタンスを生成してそのポインタを取得します.

```
TP_HANDLE tp_createInstance();
```

戻り値 TextParser インスタンスへのポインタ

TextParser インスタンスを delete します

```
int tp_deleteInstance(TP_HANDLE tp_hand);
```

tp_hand 削除する TextParser インスタンスへのポインタ
戻り値 エラーコード (=0: no error). `TextParseError` で定義される.

8.2.2 パラメータの入力及びデータ構造への格納, データ構造のファイルへの出力

パラメータファイルを読み込み, 解析結果をデータ構造へ格納する関数, データ構造に格納したパラメータを, 全てファイルに書き出す関数は次のように定義されている.

パラメータのファイルからの読み込み

```
int tp_read(TP_HANDLE tp_hand,char* file);
```

tp_hand TextParser インスタンスへのポインタ
file 入力ファイル名
戻り値 エラーコード (=0: no error). `TextParseError` で定義される.

パラメータのファイルからの読み込み (MPI local file 版)

```
int tp_read_local(TP_HANDLE tp_hand,char* file);
```

tp_hand　TextParser インスタンスへのポインタ
file　入力ファイル名
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

ファイルへの書き出し

```
int tp_write(TP_HANDLE tp_hand,char* file);
```

tp_hand　TextParser インスタンスへのポインタ
file　出力ファイル名
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

ファイルに書き出されるデータ構造は, すでにファイルから読み込まれ, 解析, 処理されたデータである. その為, 条件付き定義値は, ファイル読み込み終了時点で確定した条件による定義値になる. MPI 版の場合, tp_read() は, rank 数 0 のプロセスでファイルを読み込んだ後, 内容を分配するが, tp_read_local() は, 各プロセスがそれぞれ指定されたファイルを読み込む. 非 MPI 版で tp_read_local() を呼び出した場合は, ファイルを読み込まず, エラーメッセージを表示し, エラーコードを返す.

8.2.3 全てのパラメータのラベルパス取得, パスを指定しての値及び値の型の取得

データ構造に格納しているパラメータ全てのリーフのパスを取得する関数, パスを指定してパラメータの値を取得する関数, パラメータの値の型を取得する関数はそれぞれ次の様に定義されている.

全てのリーフ数の取得

```
int tp_getNumberOfLeaves(TP_HANDLE tp_hand,int* nleaves);
```

tp_hand　TextParser インスタンスへのポインタ
nleaves　リーフの総数
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

リーフへのラベルパスの取得

```
int tp_getIthLabel(TP_HANDLE tp_hand,int ilabel,char* label);
```

tp_hand　TextParser インスタンスへのポインタ
ilabel　インデックス　i 番目のラベルを指定. 始まりは 0.
label　リーフへのラベルパス
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

パスを指定して値を取得する

```
int tp_getValue(TP_HANDLE tp_hand,char* label,char* value);
```

tp_hand　TextParser インスタンスへのポインタ
label　パラメータへのパス
value　パラメータの値
戻り値　エラーコード (=0: no error). TextParseError で定義される.

値の型を取得する

```
int tp_getType(TP_HANDLE tp_hand,char* label,int *type);
```

tp_hand　TextParser インスタンスへのポインタ
label　パラメータへのパス
type　パラメータの値の型 TextParserType(表 1) 参照.
戻り値　エラーコード (=0: no error). TextParseError で定義される.

8.2.4 カレントノードの取得, 子ノードの取得, ノードの移動

カレントノードの取得, 子ノードの取得, ノードの移動は, 次の様に定義されています.

カレントノードの取得

```
int tp_currentNode(TP_HANDLE tp_hand,char* node);
```

tp_hand　TextParser インスタンスへのポインタ
node　カレントノードのパス.
戻り値　エラーコード (=0: no error). TextParseError 参照.

ノード移動

```
int tp_changeNode(TP_HANDLE tp_hand,char* label);
```

tp_hand　TextParser インスタンスへのポインタ
label　移動するノードのラベル（相対パス）
戻り値　エラーコード (=0: no error). TextParseError 参照.

カレントノードにある子ノードの総数の取得

```
int tp_getNumberOfCNodes(TP_HANDLE tp_hand,int* nnodes);
```

tp_hand　TextParser インスタンスへのポインタ
nnodes　子ノードの数
戻り値　エラーコード (=0: no error). TextParseError 参照.

子ノードのラベル取得

```
int  tp_getIthNode(TP_HANDLE tp_hand,int ilabel,char* label);
int  tp_getIthNodeOrder(TP_HANDLE tp_hand,int ilabel,char* label,int order);
```

tp_hand　TextParser インスタンスへのポインタ
ilabel　インデックス ilabel 番目のノードを指定. 始まりは 0.
label　子ノードへのラベル
order　ラベルの並び順オプション. ユーザーズマニュアル参照.
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

カレントノードにあるリーフの総数の取得

```
int tp_getNumberOfCLeaves(TP_HANDLE tp_hand,int* nleaves);
```

tp_hand　TextParser インスタンスへのポインタ
nleaves　リーフの数.
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

リーフへのラベル取得

```
int  tp_getIthLeaf(TP_HANDLE tp_hand,int ileaf,char* leaf);
int  tp_getIthLeafOrder(TP_HANDLE tp_hand,int ileaf,char* leaf,int order);
```

tp_hand　TextParser インスタンスへのポインタ
ileaf　インデックス ileaf 番目のリーフを指定. 始まりは 0.
leaf　リーフへのラベル
order　ラベルの並び順オプション. ユーザーズマニュアル参照.
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

8.2.5 パラメータの値の特定の型への変換

パラメータの値 (文字列) を特定の型へ変換する関数が次の様に用意されている.

型変換

```
char tp_convertChar(TP_HANDLE tp_hand,char* value, int *error);
short tp_convertShort(TP_HANDLE tp_hand,char* value, int *error);
int tp_convertInt(TP_HANDLE tp_hand,char* value, int *error);
long tp_convertLong(TP_HANDLE tp_hand,char* value, int *error);
long long tp_convertLongLong(TP_HANDLE tp_hand,char* value, int *error);
float tp_convertFloat(TP_HANDLE tp_hand,char* value, int *error);
double tp_convertDouble(TP_HANDLE tp_hand,char* value, int *error);
int tp_convertBool(TP_HANDLE tp_hand,char* value, int *error);
```

tp_hand　TextParser インスタンスへのポインタ
value　パラメータの値 (文字列)
error　エラーコード (=0: no error). `TextParseError` 参照.
戻り値　パラメータの値をそれぞれの型に変換したもの.

8.2.6 ベクトル型パラメータの値の要素への分割

ベクトル型のパラメータの値を, 要素 (文字列) に分解する関数は, 次の様に用意されている.

ベクトル型パラメータの要素数の取得

```
int tp_getNumberOfElements(TP_HANDLE tp_hand,char* vector_value,int* nvelem);
```

tp_hand　TextParser インスタンスへのポインタ
vector_value　ベクトル型パラメータの値 (文字列)
nvelem　要素数
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

ベクトル型パラメータの要素の所得

```
int tp_getIthElement(TP_HANDLE tp_hand,char* vector_value,int ivelem,char* velem);
```

tp_hand　TextParser インスタンスへのポインタ
vector_value　ベクトル型パラメータの値 (文字列)
ivelem　インデックス ivelem 番目の要素を指定. 始まりは 0.
velem　各要素の値 (文字列)
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

8.2.7 パラメータデータの破棄

格納しているパラメータのデータを破棄します.

パラメータデータの破棄

```
int tp_remove(TP_HANDLE tp_hand,);
```

tp_hand　TextParser インスタンスへのポインタ
戻り値　エラーコード (=0: no error). TextParseError 参照.

8.2.8 パラメータデータの編集

格納しているパラメータのデータを編集します.

パラメータデータの削除

```
int tp_deleteLeaf(TP_HANDLE tp_hand,char* label);
```

tp_hand　TextParser インスタンスへのポインタ
label　削除するリーフのラベル
戻り値　エラーコード (=0: no error). TextParseError 参照.

パラメータデータの更新

```
int tp_updateValue(TP_HANDLE tp_hand,char* label,char*value);
```

tp_hand　TextParser インスタンスへのポインタ
label　更新するリーフのラベル
value　更新するリーフの値. ただし, リーフのタイプ (TP_VALUE_TYPE) を変更しない文字列にすること.
戻り値　エラーコード (=0: no error). TextParseError 参照.

パラメータデータの作成

```
int tp_createLeaf(TP_HANDLE tp_hand,char* label,char*value);
```

tp_hand　TextParser インスタンスへのポインタ
label　作成するリーフのラベル
value　作成するリーフの値　尚, 文字列の値の場合は, ダブルクオートで囲むこと.
戻り値　エラーコード (=0: no error). TextParseError 参照.

8.2.9 値範囲指定記述用 API

一様値範囲指定時のパラメータ取得

```
int tp_splitRange(TP_HANDLE tp_hand,char* value,
   double *from,double *to,double *step);
```

tp_hand　TextParser インスタンスへのポインタ
value　値の文字列
from　開始値
to　終了値
step　増減値
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

一様値範囲指定時のパラメータ展開

```
int tp_expandRange(TP_HANDLE tp_hand, char* value,
      double* expanded);
```

tp_hand　TextParser インスタンスへのポインタ
value　値の文字列
expanded　展開した数値のベクター
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

任意数列での値範囲指定時のパラメータ展開

```
int tp_splitList(TP_HANDLE tp_hand,char* value,double* list,
    int order);
```

tp_hand　TextParser インスタンスへのポインタ
value　値の文字列
list　展開した数値のベクター
order　TextParserSortOrder で定義されている。 TP_SORT_NONE:記述順 TP_SORT_ASCENDING:昇順 TP_SORT_DESCENDING:降順
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

8.3 Fortran90 用インターフェース

Fortran90 用のインターフェースは, API 関数が用意されていて TestParser.inc に記述してある.

- パラメータのファイルからの読み込み
- パラメータのファイルへの書き出し
- パラメータデータの破棄

- パラメータへの絶対・相対パスの取得
- パスを使用したパラメータの値へのアクセス
- 文字列として格納している値の整数型や浮動小数点型への変換
- ベクトルして格納している値の文字列の各値への分離

以下に各機能, Fortran90 用 TextParser ライブラリの API 関数を説明する.

8.3.1 instance の生成, 消滅とシングルトンへのアクセス

FORTRAN 内では　TextParser のインスタンスを integer*8　で扱う.

TextParser シングルトンインスタンスへのポインタを取得します.

```
integer TP_GET_INSTANCE_SINGLETON(Integer*8 ptr)
```

ptr　TextParser インスタンスへのポインタ
戻り値　error code

TextParser インスタンスを生成してそのポインタを取得します.

```
integer TP_CREATE_INSTANCE(INTEGER*8 ptr)
```

ptr　TextParser インスタンスへのポインタ
戻り値　error code

TextParser インスタンスを delete します

```
integer TP_DELETE_INSTANCE(INTEGER*8 ptr)
```

ptr　削除する TextParser インスタンスへのポインタ
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

8.3.2 パラメータの入力及びデータ構造への格納, データ構造のファイルへの出力

パラメータファイルを読み込み, 解析結果をデータ構造へ格納する関数, データ構造に格納したパラメータを, 全てファイルに書き出す関数は次のように定義されている.

パラメータのファイルからの読み込み

```
INTEGER TP_READ(Integer*8 ptr,CHARACTER(len=*) file)
```

ptr　TextParser インスタンスへのポインタ
file　入力ファイル名
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

パラメータのファイルからの読み込み (MPI local file) 版

```
INTEGER TP_READ_LOCAL(Integer*8 ptr,CHARACTER(len=*) file)
```

ptr　TextParser インスタンスへのポインタ
file　入力ファイル名
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

ファイルへの書き出し

```
INTEGER TP_WRITE(Integer*8 ptr,CHARACTER(len=*) file)
```

ptr　TextParser インスタンスへのポインタ
file　出力ファイル名
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

ファイルに書き出されるデータ構造は, すでにファイルから読み込まれ, 解析, 処理されたデータである. その為, 条件付き定義値は, ファイル読み込み終了時点で確定した条件による定義値になる. MPI 版の場合, TP_READ() は, rank 数 0 のプロセスでファイルを読み込んだ後, 内容を分配するが, TP_READ_LOCAL() は, 各プロセスがそれぞれ指定されたファイルを読み込む. 非 MPI 版で TP_READ_LOCAL() を呼び出した場合は, ファイルを読み込まず, エラーメッセージを表示し, エラーコードを返す.

8.3.3 全てのパラメータのラベルパス取得, パスを指定しての値及び値の型の取得

データ構造に格納しているパラメータ全てのリーフのパスを取得する関数, パスを指定してパラメータの値を取得する関数, パラメータの値の型を取得する関数はそれぞれ次の様に定義されている.

全てのリーフ数の取得

```
INTEGER TP_GET_NUMBER_OF_LEAVES(Integer*8 ptr,INTEGER nleaves)
```

ptr　TextParser インスタンスへのポインタ
nleaves　リーフの総数
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

リーフへのラベルパスの取得

```
INTEGER TP_GET_ITH_LABEL(Integer*8 ptr,INTEGER ilabel,CHARACTER(len=*) label)
```

ptr　TextParser インスタンスへのポインタ
ilabel　インデックス　i 番目のラベルを指定. 始まりは 1.
label　リーフへのラベルパス
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

パスを指定して値を取得する

```
INTEGER TP_GET_VALUE(Integer*8 ptr,CHARACTER(len=*) label,CHARACTER(len=*) value)
```

ptr　TextParser インスタンスへのポインタ
label　パラメータへのパス
value　パラメータの値
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

値の型を取得する

```
INTEGER TP_GET_TYPE(Integer*8 ptr,CHARACTER(len=*) label,INTEGER type)
```

ptr　TextParser インスタンスへのポインタ
label　パラメータへのパス
type　パラメータの値の型 `TextParserType`(表 1) 参照.
戻り値　エラーコード (=0: no error). `TextParseError` で定義される.

8.3.4 カレントノードの取得, 子ノードの取得, ノードの移動

カレントノードの取得, 子ノードの取得, ノードの移動は, 次の様に定義されています.

カレントノードの取得

```
INTEGER TP_CURRENT_NODE(Integer*8 ptr,CHARACTER(len=*) node)
```

ptr　TextParser インスタンスへのポインタ
node　カレントノードのパス.
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

ノード移動

```
INTEGER TP_CHANGE_NODE(Integer*8 ptr,CHARACTER(len=*) label)
```

ptr　TextParser インスタンスへのポインタ
label　移動するノードのラベル（相対パス）
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

カレントノードにある子ノードの総数の取得

```
INTEGER TP_GET_NUMBER_OF_CNODES(Integer*8 ptr,INTEGER nnodes)
```

ptr　TextParser インスタンスへのポインタ
nnodes　子ノードの数
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

子ノードのラベル取得

```
INTEGER TP_GET_ITH_NODE(Integer*8 ptr,INTEGER ilabel,CHARACTER(len=*) label)
INTEGER TP_GET_ITH_NODE_ORDER(Integer*8 ptr,INTEGER ilabel,
                              CHARACTER(len=*) label,
                              INTEGER order)
```

ptr　TextParser インスタンスへのポインタ
ilabel　インデックス ilabel 番目のノードを指定. 始まりは 1.
label　子ノードへのラベル
order　ラベルの並び順オプション. ユーザーズマニュアル参照.
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

カレントノードにあるリーフの総数の取得

```
INTEGER TP_GET_NUMBER_OF_CLEAVES(Integer*8 ptr,INTEGER nleaves)
```

ptr　TextParser インスタンスへのポインタ
nleaves　リーフの数.
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

リーフへのラベル取得

```
INTEGER TP_GET_ITH_LEAF(Integer*8 ptr,INTEGER ileaf,CHARACTER(len=*) leaf)
INTEGER TP_GET_ITH_LEAF_ORDER(Integer*8 ptr,INTEGER ileaf,
                              CHARACTER(len=*) leaf,
                              INTEGER order)
```

ptr　TextParser インスタンスへのポインタ
ileaf　インデックス ileaf 番目のリーフを指定. 始まりは 1.
leaf　リーフへのラベル
order　ラベルの並び順オプション. ユーザーズマニュアル参照.
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

8.3.5 パラメータの値の特定の型への変換

パラメータの値 (文字列) を特定の型へ変換する関数が次の様に用意されている.

型変換

```
INTEGER*1 TP_CONVERT_CHAR(Integer*8 ptr,CHARACTER(len=*) value, INTEGER error)
INTEGER*1 TP_CONVERT_INT1(Integer*8 ptr,CHARACTER(len=*) value, INTEGER error)
INTEGER*2 TP_CONVERT_SHORT(Integer*8 ptr,CHARACTER(len=*) value, INTEGER error)
INTEGER*2 TP_CONVERT_INT2(Integer*8 ptr,CHARACTER(len=*) value, INTEGER error)
INTEGER*4 TP_CONVERT_INT(Integer*8 ptr,CHARACTER(len=*) value, INTEGER error)
INTEGER*4 TP_CONVERT_INT4(Integer*8 ptr,CHARACTER(len=*) value, INTEGER error)
INTEGER*8 TP_CONVERT_INT8(Integer*8 ptr,CHARACTER(len=*) value, INTEGER error)
REAL TP_CONVERT_FLOAT(Integer*8 ptr,CHARACTER(len=*) value, INTEGER error)
REAL*8 TP_CONVERT_DOUBLE(Integer*8 ptr,CHARACTER(len=*) value, INTEGER error)
LOGICAL TP_CONVERT_LOGICAL(Integer*8 ptr,CHARACTER(len=*) value, INTEGER error)
```

ptr　TextParser インスタンスへのポインタ
value　パラメータの値 (文字列)
error　エラーコード (=0: no error). `TextParseError` 参照.
戻り値　パラメータの値をそれぞれの型に変換したもの.

8.3.6 ベクトル型パラメータの値の要素への分割

ベクトル型のパラメータの値を, 要素 (文字列) に分解する関数は, 次の様に用意されている.

ベクトル型パラメータの要素数の取得

```
INTEGER TP_GET_NUMBER_OF_ELEMENTS(Integer*8 ptr,CHARACTER(len=*) vector_value,
                                  INTEGER nvelem)
```

ptr　TextParser インスタンスへのポインタ
vector_value　ベクトル型パラメータの値 (文字列)
nvelem　要素数
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

ベクトル型パラメータの要素の所得

```
INTEGER TP_GET_ITH_ELEMENT(Integer*8 ptr,CHARACTER(len=*) vector_value,INTEGER ivelem,
                           CHARACTER(len=*) velem)
```

ptr　TextParser インスタンスへのポインタ
vector_value　ベクトル型パラメータの値 (文字列)
ivelem　インデックス ivelem 番目の要素を指定. 始まりは 1.
velem　各要素の値 (文字列)
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

8.3.7 パラメータデータの破棄

格納しているパラメータのデータを破棄します.

パラメータデータの破棄

```
INTEGER TP_REMOVE(Integer*8 ptr);
```

ptr　TextParser インスタンスへのポインタ
戻り値　エラーコード (=0: no error). TextParseError 参照.

8.3.8 パラメータデータの編集

格納しているパラメータのデータを編集します.

パラメータデータの削除

```
integer TP_DELETE_LEAF(INTEGER*8 ptr,CHARACTER(len=*) label)
```

ptr　TextParser インスタンスへのポインタ
label　削除するリーフのラベル
戻り値　エラーコード (=0: no error). TextParseError 参照.

パラメータデータの更新

```
integer TP_UPDATE_VALUE(INTEGER*8 ptr,CHARACTER(len=*) label,\\
                        CHARACTER(len=*) value)
```

tp_hand　TextParser インスタンスへのポインタ
label　更新するリーフのラベル
value　更新するリーフの値. ただし, リーフのタイプ (TP_VALUE_TYPE) を変更しない文字列にすること.
戻り値　エラーコード (=0: no error). TextParseError 参照.

パラメータデータの作成

```
integer TP_CREATE_LEAF(INTEGER*8 ptr,CHARACTER(len=*) label,\\
                       CHARACTER(len=*) value)
```

ptr　TextParser インスタンスへのポインタ
label　作成するリーフのラベル
value　作成するリーフの値　尚, 文字列の値の場合は, ダブルクオートで囲むこと.
戻り値　エラーコード (=0: no error). TextParseError 参照.

8.3.9 値範囲指定記述用 API

一様値範囲指定時のパラメータ取得

```
INTEGER TP_SPLIT_RANGE(INTEGER*8  ptr,CHARACTER value,
   REAL*8 from,REAL*8 to,REAL*8 step);
```

tp_hand　TextParser インスタンスへのポインタ
value　値の文字列
from　開始値
to　終了値
step　増減値
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

一様値範囲指定時のパラメータ展開

```
INTEGER TP_EXPAND_RANGE(INTEGER*8 ptr, CHARACTER value,
      REAL*8 expanded(*));
```

ptr　TextParser インスタンスへのポインタ
value　値の文字列
expanded　展開した数値のベクター
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

任意数列での値範囲指定時のパラメータ展開

```
INTEGER TP_SPLIT_LIST(INTEGER*8  ptr,CHARACTER(len=*) value,REAL*8 list(),
    INTEGER order);
```

ptr　TextParser インスタンスへのポインタ
value　値の文字列
list　展開した数値のベクター
order　TextParserSortOrder で定義されている。 0:記述順　1:昇順 2:降順
戻り値　エラーコード (=0: no error). `TextParseError` 参照.

8.4 C++ での使用例

パッケージの Examples ディレクトリには, サンプルプログラムと, その入力ファイルが置かれている. それぞれのサンプルプログラムでは, 次のような内容が実行されている.

Example1_cpp.cpp　ファイルの入力, 出力とデータの破棄
Example2_cpp.cpp　文法上正しくないファイルの入力（エラーコードの確認用）
Example3_cpp.cpp　ファイルの入力, 出力, 全てのリーフパスを取得して全て値へのアクセスする.

Example4_cpp.cpp　ファイルの入力, 出力, 始めにカレントノード（ルートノード）を取得. その後は子ノードの相対パスを取得しながら, 全ての値へアクセスする.
Example5_cpp.cpp　Leaf の値書き換え, 削除, 新規作成
Example 6 _cpp.cpp　条件付き値のチェック

入力ファイルについては, それぞれのサンプルプログラム内の入力ファイルを参照.

8.5 C での使用例

パッケージの Examples ディレクトリには, サンプルプログラムと, その入力ファイルが置かれている. それぞれのサンプルプログラムでは, 次のような内容が実行されている.

Example1_c.c　ファイルの入力, 出力とデータの破棄
Example2_c.c　文法上正しくないファイルの入力（エラーコードの確認用）
Example3_c.c　ファイルの入力, 出力, 全てのリーフパスを取得して全て値へのアクセスする.
Example4_c.c　ファイルの入力, 出力, 始めにカレントノード（ルートノード）を取得. その後は子ノードの相対パスを取得しながら, 全ての値へアクセスする.
Example5_c.c　Leaf の値書き換え, 削除, 新規作成

入力ファイルについては, それぞれのサンプルプログラム内の入力ファイルを参照.

8.6 Fortran90 での使用例

パッケージの Examples ディレクトリには, サンプルプログラムと, その入力ファイルが置かれている. それぞれのサンプルプログラムでは, 次のような内容が実行されている.

Example1_f90.f90　ファイルの入力, 出力とデータの破棄
Example2_f90.f90　文法上正しくないファイルの入力（エラーコードの確認用）
Example3_f90.f90　ファイルの入力, 出力, 全てのリーフパスを取得して全て値へのアクセスする.
Example4_f90.f90　ファイルの入力, 出力, 始めにカレントノード（ルートノード）を取得. その後は子ノードの相対パスを取得しながら, 全ての値へアクセスする.
Example5_f90.f90　Leaf の値書き換え, 削除, 新規作成

入力ファイルについては, それぞれのサンプルプログラム内の入力ファイルを参照.

9 エラー出力

クラスライブラリ及びインターフェースでエラーや警告が発生すると関数はエラーコードを返す. その際のエラーコードは TextParserCommon.h の enum TextParserError で定義されている. 100 番台はエラー,200 番台は警告である. また, 殆どの場合エラーメッセージが出力されるが, エラーメッセージは TextParserErrorHandler 関数により標準エラー出力 (std::stderr) に出力する. パラメータ入力中は TextParserTree::_current_line を行数として一緒に表示する. TextParserError の定義は表 3,4 に示す.

表 3　TextParserError の定義値 その 1

TextParserError 定義値	値の type
TP_NO_ERROR = 0	エラーなし
TP_ERROR = 100	エラー
TP_DATABASE_NOT_READY_ERROR = 101	データベースにアクセス出来ない
TP_DATABASE_ALREADY_SET_ERROR = 102	データベースが既に読み込まれている
TP_FILEOPEN_ERROR = 103	ファイルオープンエラー
TP_FILEINPpUT_ERROR = 104	ファイル入力エラー
TP_FILEOUTPUT_ERROR = 105	ファイル出力エラー
TP_ENDOF_FILE_ERROR = 106	ファイルの終わりに達しました
TP_ILLEGAL_TOKEN_ERROR = 107	トークンが正しくない
TP_MISSING_LABEL_ERROR = 108	ラベルが見つからない
TP_ILLEGAL_LABEL_ERROR = 109	ラベルが正しくない
TP_ILLEGAL_ARRAY_LABEL_ERROR = 110	配列型ラベルが正しくない

表 4　TextParserError の定義値その 2

TextParserError 定義値	値の type
TP_MISSING_ELEMENT_ERROR = 111	エレメントが見つからない
TP_ILLEGAL_ELEMENT_ERROR = 112	エレメントが正しくない
TP_NODE_END_ERROR = 113	ノードの終了文字が多い
TP_NODE_END_MISSING_ERROR = 114	ノードの終了文字が無い
TP_NODE_NOT_FOUND_ERROR = 115	ノードが見つからない
TP_LABEL_ALREADY_USED_ERROR = 116	ラベルが既に使用されている
TP_LABEL_ALREADY_USED_PATH_ERROR = 117	ラベルがパス内で既に使用されている
TP_ILLEGAL_CURRENT_ELEMENT_ERROR = 118	カレントのエレメントが異常
TP_ILLEGAL_PATH_ELEMENT_ERROR = 119	パスのエレメントが異常
TP_MISSING_PATH_ELEMENT_ERROR = 120	パスのエレメントが見つからない
TP_ILLEGAL_LABEL_PATH_ERROR = 121	パスのラベルが正しくない
TP_UNKNOWN_ELEMENT_ERROR = 122	不明のエレメント
TP_MISSING_EQUAL_NOT_EQUAL_ERROR = 123	==も!=も見つからない
TP_MISSING_AND_OR_ERROR = 124	&&も——も見つからない
TP_MISSING_CONDITION_EXPRESSION_ERROR = 125	条件式が見つからない
TP_MISSING_CLOSED_BRANCKET_ERROR = 126	条件式が見つからない
TP_ILLEGAL_CONDITION_EXPRESSION_ERROR = 127	条件式の記述が正しくない
TP_ILLEGAL_DEPENDENCE_EXPRESSION_ERROR = 128	依存関係の記述が正しくない
TP_MISSING_VALUE_ERROR = 129	値が見つからない
TP_ILLEGAL_VALUE_ERROR = 130	値が正しくない
TP_ILLEGAL_NUMERIC_VALUE_ERROR = 131	数値が正しくない
TP_ILLEGAL_VALUE_TYPE_ERROR = 132	ベクトルの値タイプが一致しない
TP_MISSING_VECTOR_END_ERROR = 133	ベクトルの終了文字が無い
TP_VALUE_CONVERSION_ERROR = 134	値の変換エラー
TP_MEMORY_ALLOCATION_ERROR = 135	メモリが確保できない
TP_REMOVE_ELEMENT_ERROR = 136	エレメントの削除エラー
TP_MISSING_COMMENT_END_ERROR = 137	コメントの終わりが見つからない
TP_ID_OVER_ELEMENT_NUMBER_ERROR = 138	ID が要素数を超えている
TP_GET_PARAMETER_ERROR = 139	パラメータ取得
TP_UNSUPPORTED_ERROR = 199	サポートされていない
TP_WARNING = 200	エラー
TP_UNDEFINED_VALUE_USED_WARNING = 201	未定義のデータが使われている
TP_UNRESOLVED_LABEL_USED_WARNING = 202	未解決のラベルが使われている

10 文字列処理

以下は文字列処理用の関数である.

10.1 数値の文字列を補正する (TextParserCorrectValueString 関数)

数値の無駄なスペースを取り除く. 又, 指数が"d" や"D" で指定された場合は "e" に置き換える.

10.2 文字列の先頭のスペースを削除 (TextParserRemoveHeadSpaces 関数)

文字列の頭のスペースとタブ,<CR>,<LF> を全て削除する.

10.3 文字列の末尾のスペースを削除 (TextParserRemoveTailSpaces 関数)

文字列の末尾のスペースとタブ,<CR>,<LF> を全て削除する.

10.4 文字列を小文字に変換する (TextParserStringToLower 関数)

文字列を全て小文字に変換する.

11 MPI 対応版用一行読み込み関数

MPI 対応版の場合,std::getline 関数に代わって, tp_getline_mpi 関数が呼び出される.

11.1 MPI 対応版用 getline ファイル読み込み (tp_getline_mpi 関数)

MPI 対応版では, プロセス数とランク数によって処理を変える.

プロセス数が 1 の場合 getline をコールする.

プロセス数が 2 以上の場合 ランクが 0 ならば, ファイルをオープンしているインプットストリームから 1 行読み込む. 読み込んだ内容を MPI_Bcast して全てのプロセスに配布する. 全てのプロセスが, 読み込んだ内容をストリングに格納する. エラーがあれば false を戻し, 通常は true を戻す.

12 ビルド方法

以下に Unix ライクなシステムでの標準的なビルド方法を示す. configure スクリプト実行時に使用しているシステムに合わせて, オプション等を設定する. くわしくは, プログラム使用マニュアル（users_manual.pdf）を参照すること.

1. パッケージを展開後, 展開したノードに移動する.

```
$ tar zxf TextParser-X.Y.tar.gz
$ cd TextParser-X.Y
```

2. configure スクリプトを実行する.

```
$ ./configure
```

3. make を実行, ビルドする.

```
$ make
```

4. make　install で,prefix に設定された各ノードにライブラリ, インクルードファイルをインストールする. 適宜インストール先に対する書き込み権限に応じて, sudo を用いての実行, 又は, superuser での実行を行うこと.

```
$ make　install（prefix に書き込み権限がある場合）
$ sudo make　install（prefix に書き込み権限が無い場合 sudo 有り）
# make　install（prefix に書き込み権限が無い場合 sudo 無し）
```

13 アップデート情報

本文書のアップデート情報について記す.

Version 1.2 2012/11/26

- Version 1.2
  @range,@list 用 API の記述.

Version 1.1-dev_d 2012/8/29

- Version 1.1-dev_d
  パッケージの tar ball 名, パッケージディレクトリ名を変更.

Version 1.1-dev_c 2012/8/27

- Version 1.1-dev_c
  バグフィックス.

Version 1.1-dev_b 2012/8/24

- Version 1.1-dev_b
  脱シングルトン化, 及びデータ編集関数の記述追加.

Version 1.0 2012/6/16

- Version 1.0 リリース.

Version 0.9c 2012/5/28

- Version 0.9c リリース. MPI 対応版に関する記述を追加.
  getNodes, getLabels と関連の関数に並び順オプションの記述を追加.

Version 0.9b 2012/5/7

- Version 0.9b リリース. エラーコード表, インターフェイス, および ビルド方法の体裁, 文章を修正.
  C/Fortran90 用 API の説明を追加.

Version 0.9a 2012/4/28

- Version 0.9a リリース.