

Text Parser Library

User's Manual

June 2014

(c) Copyright 2012-2014

Copyright (c) 2012-2014 Institute of Industrial Science, The University of Tokyo.

All rights reserved.

4-6-1 Komaba, Meguro-ku, Tokyo, 153-8505 JAPAN

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



(c) Copyright 2014

Advanced Institute for Computational Science, RIKEN.

All rights reserved.

7-1-26, Minatojima-minami-machi, Chuo-ku, Kobe, 650-0047, JAPAN.

<http://www.aics.riken.jp/>



目次

第 1 章	この文書について	1
1.1	TextParser ライブラリについて	1
1.2	書式について	1
1.3	動作環境	1
第 2 章	パッケージのビルド	3
2.1	パッケージの構造	3
2.2	ライブラリパッケージのビルド	4
2.3	configure スクリプトでのビルド	4
2.4	configure スクリプトのオプション	6
2.5	Fortran 対応版のビルド	7
2.6	MPI 並列対応版のビルド	7
2.7	Windows Cygwin 環境でのビルド, 利用について	8
2.8	TextParser ライブラリの手動設定でのビルド	8
2.8.1	手動設定用 Makefile の編集	8
2.8.2	make の実行 (default)	9
2.8.3	make の実行 (MPI 版)	9
2.8.4	make の実行 (Fortran 用 API)	10
第 3 章	ライブラリの利用法 (ビルドと実行)	11
3.1	C++	11
3.2	C 言語	11
3.3	Fortran 90	11
3.4	実行環境設定 LD_LIBRARY_PATH にインストールしたライブラリのパスを追加	12
3.5	MPI 並列プログラムでの利用	12
第 4 章	ライブラリの利用法 (ユーザープログラムでの利用方法)	13
4.1	Examples ディレクトリのプログラム	13
4.1.1	C++ の例	13
4.1.2	C 言語の例	13
4.1.3	Fortran の例	14
4.1.4	C++ MPI 並列の例	14
4.2	C++ での利用方法	14
4.2.1	include ファイル	14
4.2.2	TextParserValueType	14
4.2.3	TextParserError	14
4.2.4	インスタンスの取得	17

	シングルトンインスタンス取得メソッド	17
	インスタンス取得方法	17
4.2.5	ファイル IO とメモリの開放	17
4.2.6	ラベルの取得と値の取得 (フルパス)	18
4.2.7	既知のデータ型値の取得 (フルパス)	19
4.2.8	ラベル相対パスアクセス用関数	21
4.2.9	型変換用関数	21
4.2.10	ベクトル型の値の分解	22
4.2.11	パラメータデータの編集	22
4.2.12	値範囲指定記述用 API	23
4.2.13	その他の API	24
4.3	C 言語での利用方法	25
4.3.1	include ファイル	25
4.3.2	TextParserValueType	25
4.3.3	TextParserError	25
4.3.4	instance の生成, 消滅とシングルトンへのアクセス	25
4.3.5	ファイル IO とメモリの開放	26
4.3.6	ラベルの取得と値の取得 (フルパス)	27
4.3.7	ラベル相対パスアクセス用関数	27
4.3.8	型変換用関数	29
4.3.9	ベクトル型の値の分解	29
4.3.10	パラメータデータの編集	30
4.3.11	値範囲指定記述用 API	31
4.4	Fortran90 での利用方法	31
4.4.1	instance の生成, 消滅とシングルトンへのアクセス	32
4.4.2	ファイル IO とメモリの開放	32
4.4.3	ラベルの取得と値の取得 (フルパス)	33
4.4.4	ラベル相対パスアクセス用関数	34
4.4.5	型変換用関数	35
4.4.6	ベクトル型の値の分解	36
4.4.7	パラメータデータの編集	37
4.4.8	値範囲指定記述用 API	38
第 5 章	パラメータの記述方式	39
5.1	ラベル	39
5.2	ノード	42
5.3	リーフ	43
5.4	値	44
5.5	改行	46
5.6	コメント	46
5.7	パラメータファイルのサンプル	47
第 6 章	アップデート履歴	48
6.1	アップデート履歴	48

第 1 章

この文書について

この文書は、多用途汎用パラメータパーサライブラリ（以下 TextParser ライブラリ）の使用説明書です。

1.1 TextParser ライブラリについて

このライブラリは、決められた書式でパラメータ定義が書かれたファイルを読み込み、その内容をツリー構造で保持し、文字列で格納します。ユーザーのプログラム中では、その格納されたデータにアクセスし、格納されたパラメータを文字列として取り出すことができます。また、パラメータの値を文字列から任意の型に変換してプログラム中で利用することができます。ユーザーは、C++/C/Fortran90 でこのライブラリが利用可能です。

1.2 書式について

次の書式で表されるものは Shell のコマンドです。

\$ コマンド（コマンド引数）

または、

コマンド（コマンド引数）

“\$” で始まるコマンドは一般ユーザーで実行するコマンドを表し、“#” で始まるコマンドは管理者（主に root）で実行するコマンドを表しています。

1.3 動作環境

TextParser ライブラリは、以下の環境について動作を確認しております。

- GNU/Linux
 - CentOS
 - OS CentOS6.2 i386/x86_64
 - gcc/g++/gfortran(Fortran90) gcc version 4.4.5
 - Debian GNU/Linux
 - OS Debian 6 squeeze i386/amd64
 - gcc/g++/gfortran(Fortran90) gcc version 4.4.5
- MacOS X Snow Leopard

- OS MacOS X Snow Leopard
- gcc/g++/gfortran(Fortran90) gcc4.4 およびその開発パッケージ
- MacOS X Lion
 - OS MacOS X Lion
 - gcc/g++/gfortran(Fortran90) gcc4.4 およびその開発パッケージ
- Microsoft Windows
 - Windows7(64bit)
 - OS Windows7(64bit) Cygwin 1.7.9
 - gcc/g++/gfortran(Fortran90) 4.5
 - WindowsXP(32bit)
 - OS WindowsXP(32bit) Cygwin 1.7.9
 - gcc/g++/gfortran(Fortran90) 4.5

又, MPI 対応版については, 次のような環境で動作を確認しています.

- GNU/Linux
 - CentOS
 - OS CentOS6.2 x86_64
 - gcc/g++/gfortran(Fortran90) gcc version 4.4.5
 - MPICH openMPI 1.4.4

第 2 章

パッケージのビルド

2.1 パッケージの構造

TextParser ライブラリのパッケージは, 次のようなファイル名で保存されています.

TextParser-x.x.tar.gz

(x) は, バージョン番号. このファイルの内部には, 次のようなディレクトリ構造が格納されています.

```
TextParser-x.x
├── doc
├── Examples
├── include
├── lib
├── m4
└── src
```

これらのディレクトリ構造は, 次の様になっています.

doc ドキュメントディレクトリ: この文書を含む TextParser ライブラリの文書が収められている.

Examples テスト用, ライブラリ使用例のプログラムとインプットファイルの例が収められています.

include ヘッダファイルが収められています. ここに収められたファイルは `make install` で `$prefix/include` にインストールされます.

m4 autotools 向けのマクロが収められています.

src ソースが格納されたディレクトリです.

lib ここにライブラリが作成され, `make install` で `$prefix/lib` にインストールされます.

2.2 ライブラリパッケージのビルド

ライブラリは、configure スクリプトによる環境設定を用いたビルドと手動での環境設定を用いたビルドの2種類に対応しています。この章以降では、まず configure スクリプトによるビルドを説明し、最後に手動での環境設定を用いたビルドを説明しています。手動設定によるビルドは、2.8 を見てください。

2.3 configure スクリプトでのビルド

いずれの環境でも shell で作業するものとします。この例では、bash を用いていますが、shell によって環境変数の設定方法が異なるだけで、インストールの他のコマンドは同一です。適宜、環境変数の設定箇所をお使いの環境でのものに読み替えてください。Windows Cygwin 環境の場合は、configure スクリプトで Fortran コンパイラを指定する必要があります。詳しくは 2.7 を参照してください。本ライブラリでは MPI 並列対応版が用意されています。MPI 並列対応版をビルドするには、2.6 を参照してください。

以下の例では、作業ディレクトリを作成し、作業ディレクトリにパッケージを展開し、ビルド、インストールする例を示しています。

1. 作業ディレクトリの構築とパッケージのコピー

まず、作業用のディレクトリを用意し、パッケージをコピーします。ここでは、カレントディレクトリに work というディレクトリを作り、そのディレクトリにパッケージをコピーします。

```
$ mkdir work
$ cp [パッケージのパス] work
```

2. 作業ディレクトリへの移動とパッケージの解凍

先ほど作成した作業ディレクトリに移動し、パッケージを解凍します。

```
$ cd work
$ tar xzf TextParser-x.x.tar.gz
```

3. TextParser-x.x ディレクトリに移動先ほどの解凍で作成された TextParser-X.Y ディレクトリに移動します。

```
$ cd TextParser-X.Y
```

4. configure スクリプトを起動

次のコマンドで configure スクリプトを起動します。

```
$ ./configure
```

configure スクリプトには、オプションを与えて、お使いの環境に合わせ設定が可能です。オプションに関しては、2.4 を参照してください。configure スクリプトで各ディレクトリに指定した環境に合わせた Makefile が作成されます。

5. make コマンドでライブラリ作成、テストプログラムのビルド

make コマンドでライブラリ作成、テストプログラムのビルドを行います。

```
$ make
```

make コマンドでは, 次のファイルが作成されます.

```
src/libTP.a  
src/libTPmpi.a  
src/libTP_fapi.a
```

ただし libTPmpi.a は, configure 実行時に MPI 並列対応版のビルドオプションを有効にした場合に, libTP_fapi.a は, Fortran のビルドオプションを有効にした場合に作成されます.

又, ビルドをやり直す場合に make コマンドで作成されるファイルを削除するには,

```
$ make clean
```

とします. また, configure による設定, Makefile の生成をやり直すには,

```
$ make distclean
```

として, configure スクリプトの実行からやり直してください.

6. make install コマンドでライブラリ, ヘッダファイルのインストール

“make install” コマンドで, configure スクリプトの prefix で指定されたディレクトリに, ライブラリ, ヘッダファイルをインストールします. インストールされる場所とファイルは以下の通りです.

```
{prefix}  
├── lib  
│   ├── libTP.a  
│   ├── libTPmpi.a  
│   └── libTP_fapi.a  
└── include  
    ├── TextParser.h  
    ├── TextParserCommon.h  
    ├── TextParserElement.h  
    ├── TextParserTree.h  
    └── TextParser.inc
```

ここで X,Y,Z は, バージョン番号です.

libTPmpi.a は, configure 実行時に MPI 並列対応版のビルドオプションを有効にした場合に, libTP_fapi.a は, Fortran のビルドオプションを有効にした場合にインストールされます.

prefix でのインストール先の設定等は, 2.4 を参照してください.

インストール先が, ユーザーの権限で書き込み可能である場合は, 次のようにします.

```
$ make install
```

インストール先が, 書き込みの際に管理者権限を必要とする場合で, sudo が使用可能ならば次のようにします.

```
$ sudo make install
```


インストール先が、書き込みの際に管理者権限を必要とする場合で、`sudo` が使用可能で無いなら、`root` としてログインして、`make install` を実行します。

```
$ su
password:
# make install
# exit
```

また、アンインストールするには、書き込み権限によって

```
$ make uninstall または
$ sudo make uninstall または
# make uninstall を実行してください。
```

2.4 configure スクリプトのオプション

インストール場所、コンパイラ等、MPI 対応のオプションは以下のように指定します。

- `--prefix=dir`
`prefix` は、パッケージをどこにインストールするかを指定します。`prefix` で設定した場所が `--prefix=/usr/local` の時、
ライブラリは、`/usr/local/lib`
ヘッダファイルは、`/usr/local/include`
にインストールされます。
デフォルト値は `/usr/local` で、`configure` スクリプトで何も指定しない場合、デフォルト値に設定されます。
- `--enable-fapi`
Fortran 用 API ライブラリと Fortran 用テストプログラムをビルドする為のスイッチです。`--enable-fapi` でスイッチが有効になります。
このオプションはデフォルトで無効になっています。
- コンパイラ等のオプション
コンパイラ、リンカやそれらのオプションは、`configure` スクリプトで半自動的に探索します。ただし、標準ではないコマンドやオプション、ライブラリ、ヘッダファイルの場所は探索出来ないことがあります。また、標準でインストールされたものでないコマンドやライブラリを指定して利用したい場合があります。そのような場合、以下のコンパイル、リンクのコマンド及びオプションを `configure` スクリプトで指定することができます。

`CC C` コンパイラのコマンドパスです。

`CFLAGS C` コンパイラへ渡すフラグです。

`CXX C++` コンパイラのコマンドパスです。

`CXXFLAGS C++` コンパイラへ渡すフラグです。

`LDFLAGS` リンク時にリンカに渡すフラグです。例えば、使用するライブラリが標準でない場所 `<libdir>` にある場合、`-L<libdir>` としてその場所を指定します。

LIBS 利用したいライブラリをリンカに渡すフラグです。例えば、ライブラリ <library> を利用する場合、`-l<library>` として指定します。

CPP プリプロセッサのコマンドパスです。

CPPFLAGS プリプロセッサへ渡すフラグです。例えば、標準ではない場所 <include dir> にあるヘッダファイルを利用する場合、`-I<include dir>` と指定します。

F77 Fortran 77 コンパイラのコマンドパスです。

FFLAGS Fortran 77 コンパイラに渡すフラグです。

FC Fortran コンパイラのコマンドパスです。

FCFLAGS Fortran コンパイラに渡すフラグです。

CXXCPP C++ のプリプロセッサのコマンドパスです。

例えば prefix に /usr/local/TextParser Fortran コンパイラに gfortran を指定する場合は、次のようにします。

```
$ ./configure --prefix=/usr/local/TextParser FC=gfortran
```

その他、\$./configure --help を実行すると、一般的なオプションが表示されますが、有効なものは、インストールディレクトリ指定の `--prefix` `--includedir` `--libdir` `--enable-fapi` と上記のコンパイラ関連の設定です。

2.5 Fortran 対応版のビルド

Fortran 用 API ライブラリとサンプルプログラムをビルドするには、configure スクリプトでオプションを指定する必要があります (2.4 を参照)。お使いの環境によっては、FC に Fortran コンパイラを指定する必要があります。

次の例では、Fortran をオプションで有効にし、Fortran コンパイラに gfortran を指定しています。

```
$ ./configure FC=gfortran --enable-fapi
```

このオプションを有効にすると libTP_fapi.a がインストールされ、`-lTP_fapi` をリンク時に指定出来るようになります。この際、リンクオプションに `-lTPmpi` または `-lTP` を指定してください。

2.6 MPI 並列対応版のビルド

MPI 並列対応版 C++ ライブラリのビルドには、configure スクリプトのオプション CXX において、MPI 対応 C++ コンパイラを指定します。

次の例では、C++ コンパイラに mpicc を指定しています。

```
$ ./configure CXX=mpicc
```

利用するコンパイラによって、コンパイル、リンクオプションを指定する必要がある場合には、CXXFLAGS や LDFLAGS 等で指定してください。

このオプションを有効にすると libTPmpi.a がインストールされ、-lTPmpi をリンク時に指定出来るようになります。この際、リンクオプションに -lTPmpi と -lTP を同時に使用しないでください。

又、MPI 対応が有効な場合、configure スクリプト実行時に生成される config.h 内で、ENABLE_MPI マクロが定義されます。ユーザープログラムでこのマクロを利用したい場合は、config.h をインクルードしてください。ただし、config.h は、ライブラリビルド時の設定の格納が主な目的ですので、prefix 等で指定されたインストール場所 (TextParser.h 等のヘッダファイルのインストール場所) にはインストールされないので注意してください。

2.7 Windows Cygwin 環境でのビルド, 利用について

Cygwin 1.7.9 環境での利用は可能ですが、fortran コンパイラの指定をせずに configure スクリプトを実行した場合に指定される標準の Fortran コンパイラは、古いもの (gcc v3 base) になり、ライブラリの作成に失敗します。これを避ける為、Cygwin 環境での利用には、fortran コンパイラに gfortran(gcc v4 base) を指定してください。fortran コンパイラに gfortran(gcc v4 base) を指定するには、次の様にします。

```
$ ./configure FC=gfortran F77=gfortran
```

2.8 TextParser ライブラリの手動設定でのビルド

TextParser ライブラリのビルド時にコマンド等を手動で設定して行うには、以下の様にします。また、このビルドではスタティックライブラリを生成します。ライブラリ使用に必要なファイルは以下の通りです。ビルド後に適当な場所にコピーしてお使いください。

ライブラリファイル

lib/libTP.a (通常版)

lib/libTPmpi.a (MPI 版)

lib/libTP_fapi.a (FORTRAN 用 API)

インクルードファイル

include/TextParser.h (C/C++ 用)

include/TextParserCommon.h

include/TextParserTree.h

include/TextParserElement.h

include/TextParser.inc (FORTRAN 用)

2.8.1 手動設定用 Makefile の編集

手動設定用に Makefile_hand というファイルをそれぞれ top ディレクトリと src ディレクトリ, Examples ディレクトリに配置してあります。手動設定時のビルドはこれらのファイルを用います。

top ディレクトリの Makefile_hand 内のコマンド等の設定を、お使いのシステム、環境に合わせて適宜変更してください。配布している例では、

- mpi 用 C++ コンパイラ `MPICXX = mpicxx`
- Fortran コンパイラ `FC = gfortran`

の様に変数を設定し、この変数の設定を `lib,Examples` ディレクトリのビルドにも反映させています。

`MPICXX` は、mpi 用ライブラリ、使用例をビルドする場合は、必ず設定してください。それ以外のコマンド変数等は、`make` のデフォルトに準拠しており、`CXX`, `CXXFLAGS`, `AR`, `ARFLAGS`, `RANLIB` 等を用いることができます。また、指定しない場合は、`make` のデフォルトに置き換えられます。配布例では、通常の `c++` コードは、`make` のデフォルトの `c++` コンパイラを用います。それ以外のコンパイラ/コマンドを使用しようとする場合は、必ずその設定をしてください。

尚、この方法で作成するライブラリは、static ライブラリ (`.a`) が `lib` ディレクトリに作成されます。

2.8.2 make の実行 (default)

`make` は、パッケージの `top` ディレクトリで、次の様に実行します。

```
$ make -f Makefile_hand
```

これによって次のものが作成されます。

`lib` ディレクトリ

`lib/libTP.a`

`Examples` ディレクトリ

`Examples/Example1.cpp`

`Examples/Example2.cpp`

`Examples/Example3.cpp`

`Examples/Example4.cpp`

`Examples/Example5.cpp`

`Examples/Example6.cpp`

`Examples/Example7.cpp`

`Examples/Example1.c`

`Examples/Example2.c`

`Examples/Example3.c`

`Examples/Example4.c`

`Examples/Example5.c`

`Examples/Example7.c`

2.8.3 make の実行 (MPI 版)

`make` は、パッケージの `top` ディレクトリで、次の様に実行します。

```
$ make -f Makefile_hand mpi
```

これによって次のものが作成されます。

lib ディレクトリ
lib/libTPmpi.a

Examples ディレクトリ
Examples/Example3.cpp_mpi

2.8.4 make の実行 (Fortran 用 API)

make は、パッケージの top ディレクトリで、次の様に実行します.

```
$ make -f Makefile_hand fapi
```

これによって次のものが作成されます.

lib ディレクトリ
lib/libTP_fapi.a

Examples ディレクトリ
Examples/Example1.f90
Examples/Example2.f90
Examples/Example3.f90
Examples/Example4.f90
Examples/Example5.f90
Examples/Example7.f90

第 3 章

ライブラリの利用法 (ビルドと実行)

TextParser ライブラリは, C++/C 言語及び Fortran90 のプログラム内で利用できます. ユーザーが作成する TextParser を利用するプログラムのビルド方法を示します. 以下の例では, configure スクリプトで "prefix=/usr/local/TextParser" を指定し, ライブラリが /usr/local/TextParser/lib, ヘッダファイルが /usr/local/TextParser/include にインストールされているものとして示します.

3.1 C++

TextParser ライブラリを利用している C++ のプログラム mymain.cpp を g++ でコンパイルする場合は, 次のようにコンパイル, リンクします.

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/TextParser/lib
$ g++ -o mymain mymain.cpp -I/usr/local/TextParser/include \
-L/usr/local/TextParser/lib -lTP
```

この時, リンクライブラリのオプションで MPI 版用オプション -lTPmpi と通常版用オプション -lTP を同時に使用しないでください.

3.2 C 言語

TextParser ライブラリを利用している C 言語のプログラム mymain.c を gcc でコンパイルする場合は, 次のようにコンパイル, リンクします.

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/TextParser/lib
$ gcc -o mymain mymain.c -I/usr/local/TextParser/include \
-lstdc++ -L/usr/local/TextParser/lib -lTP
```

この時, リンクライブラリのオプションで MPI 版用オプション -lTPmpi と通常版用オプション -lTP を同時に使用しないでください.

3.3 Fortran 90

TextParser ライブラリを利用している Fortran90 のプログラム mymain.f90 を gfortran でコンパイルする場合は, 次のようにコンパイル, リンクします.

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/TextParser/lib
$ gfortran -o mymain mymain.c -I/usr/local/TextParser/include \
-lstdc++ -L/usr/local/TextParser/lib -lTP -lTP_fapi
```

この時、リンクライブラリのオプションで MPI 版用オプション `-lTPmpi` と通常版用オプション `-lTP` を同時に使用しないでください。ライブラリのビルド時に `configure` オプションで `'--enable-fapi'` を有効にして、`libTP_fapi.a` をビルドしてください。

3.4 実行環境設定 LD_LIBRARY_PATH にインストールしたライブラリのパスを追加

シェアードライブラリをリンクした実行ファイルを実行する場合には、ライブラリパスの指定が必要になります。その場合は、環境変数 `LD_LIBRARY_PATH` にパスを追加します。例えばライブラリが `/usr/local/TextParser/lib` にインストールされていれば、次のようにします。

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/TextParser/lib
```

3.5 MPI 並列プログラムでの利用

MPI 並列化されたユーザーのプログラムで本ライブラリ (MPI 対応版) を利用する場合は、次のようにします。

1. MPI 対応版のライブラリをビルド、インストールします。2.2,2.4,2.6 を参照してください。
2. `LD_LIBRARY_PATH` を指定します。
3. プログラムを MPI 利用環境でビルドします。

`mymain.cpp` を `mpicxx` でコンパイルする場合は、例えばライブラリが `/usr/local/TextParser/lib` にインストールされていれば、次のようにします。

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/TextParser/lib
$ mpicxx -o mymain mymain.cpp -I/usr/local/TextParser/include \
-L/usr/local/TextParser/lib -lTPmpi
```

この時、リンクライブラリのオプションで MPI 版用オプション `-lTPmpi` と通常版用オプション `-lTP` を同時に使用しないでください。このプログラムを 4 並列で計算させる場合は、MPI 実行コマンドが `mpirun` であるとき以下のようになります。

```
$ mpirun -np 4 mymain
```

第 4 章

ライブラリの利用法 (ユーザープログラムでの利用方法)

以下はライブラリの API の説明 (C++/C/Fortran90) です。これらの関数群は、ライブラリが提供するヘッダファイル、TextParser.h(C++/C) 又はおよび TextParser.inc(Fortran90) で定義されています。ライブラリの関数を使う場合は、このファイルをインクルードします。TextParser.h 及び TextParser.inc は、configure スクリプト実行時の設定 prefix の下 \${prefix}/include に make install 時にインストールされています。

4.1 Examples ディレクトリのプログラム

Examples ディレクトリには、C++/C/Fortran90 での使用例のソースコードが示してあります。参考にしてください。

4.1.1 C++ の例

- Example1_cpp.cpp パラメータファイルの読み込み, 書き出し, 読み込んだデータの破棄
- Example2_cpp.cpp エラーまたは警告となるようなパーサファイルの入力
- Example3_cpp.cpp 全てのパラメータを取得する (フルパス)
- Example4_cpp.cpp 全てのパラメータを取得する (相対パス)
- Example5_cpp.cpp Leaf の VALUE 書き換え, 削除, 新規作成
- Example6_cpp.cpp 依存関係付き値のチェック
- Example7_cpp.cpp @range,@list のテスト
- Example7_cpp.cpp @range,@list のテスト (エラーになるもの)

これらは、make 時にビルドされ、./Example1_cpp, ./Example2_cpp, ./Example3_cpp, ./Example4_cpp, ./Example5_cpp 及び ./Example6_cpp という実行ファイルが作成されます。

4.1.2 C 言語の例

- Example1_c.c パラメータファイルの読み込み, 書き出し, 読み込んだデータの破棄
- Example2_c.c エラーまたは警告となるようなパーサファイルの入力
- Example3_c.c 全てのパラメータを取得する (フルパス)
- Example4_c.c 全てのパラメータを取得する (相対パス)
- Example5_c.c Leaf の VALUE 書き換え, 削除, 新規作成
- Example7_c.c @range,@list のテスト

これらは、make 時にビルドされ、./Example1_c, ./Example2_c, ./Example3_c, ./Example4_c 及び ./Example5_c という実行

ファイルが作成されます。

4.1.3 Fortran の例

- Example1_f90.f90 パラメータファイルの読み込み, 書き出し, 読み込んだデータの破棄
- Example2_f90.f90 エラーまたは警告となるようなパーサファイルの入力
- Example3_f90.f90 全てのパラメータを取得する (フルパス)
- Example4_f90.f90 全てのパラメータを取得する (相対パス)
- Example5_f90.f90 Leaf の VALUE 書き換え, 削除, 新規作成
- Example7_f90.f90 @range, @list のテスト

これらは, make 時にビルドされ, ./Example1_f90, ./Example2_f90, ./Example3_f90 及び ./Example4_f90 という実行ファイルが作成されます。

4.1.4 C++ MPI 並列の例

- Example3_cpp_mpi.cpp 全てのパラメータを取得する (フルパス)
- Example3-1_cpp_mpi.cpp rank によって違うファイルを読み込む例

このプログラムは, make 時にビルドされ, ./Example3_cpp_mpi と ./Example3-1_cpp_mpi という実行ファイルが作成されます。TextParser ライブラリのビルド時に MPI 対応を有効化している場合, rank0 のプロセスがファイルを読み込み, その内容を全てのプロセスに送り, 全てのプロセスが, パースしてデータを格納します。MPI 実行コマンドが mpirun である場合, 4 並列で実行するには次のようにします。

```
$ mpirun -np 4 mymain
```

4.2 C++ での利用方法

C++ で本ライブラリを利用する場合 TextParser.h をインクルードします。TextParser.h には, ユーザーがライブラリを利用する API がまとめられている TextParser クラスが用意されています。プログラム内部からこのライブラリを使用する場合, このクラスのメソッドを用います。

4.2.1 include ファイル

ユーザーが TextParser ライブラリを利用する場合, TextParser.h をインクルードします。他に必要なヘッダファイルは, TextParser.h から読み込まれますので, その他のファイルをインクルードする必要はありません。

プログラム内で使用される型のうち, ユーザーが利用するものについては, TextParserCommon.h に定義されています。

4.2.2 TextParserValueType

TextParserValueType は, TextParserCommon.h で次の様に定義されています。

リーフへの取得後, そのリーフパスが示す値の型を取得することで, その後の変換処理の条件判断に用いることが出来ます。

4.2.3 TextParserError

TextParserError は, TextParserCommon.h で定義されています。定義値は表 4.5, 4.3 の通りです。

表 4.1 TextParserValueType

TextParserValueType 定義値	値の type
TP_UNDEFFINED_VALUE = 0	不定
TP_NUMERIC_VALUE = 1	数値
TP_STRING_VALUE = 2	文字列
TP_DEPENDENCE_VALUE = 3	依存関係付き値
TP_VECTOR_UNDEFFINED = 4	ベクトル型不定
TP_VECTOR_NUMERIC = 5	ベクトル型数値
TP_VECTOR_STRING = 6	ベクトル型文字列

表 4.2 TextParserError 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_FILEINPUT_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
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

表 4.3 TextParserError その 2

TextParserError 定義値	値の type
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_BRACKET_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 が要素数を超過している
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

4.2.4 インスタンスの取得

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自由に作成することが出来るインスタンスの 2 種類があります .

シングルトンインスタンス取得メソッド シングルトンインスタンスへのポインタを取得するメソッドは, TextParser.h 内で次のように定義されています.

——— 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 TextParser クラスのシングルトンインスタンスへのポインタ

インスタンス取得方法 上記のシングルトンインスタンスとは独立に, コンストラクタによってユーザーがインスタンスを作成できます .

例えば, 次のような方法で, インスタンスを生成できます .

```
TextParser tp_instance ;  
TextParser * tp_ptr= new TextParser;
```

ユーザーの作成するプログラム内では, シングルトンインスタンス取得メソッドが生成し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します.

4.2.5 ファイル IO とメモリの開放

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ます.

——— 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). TextParserError で定義されています.

——— 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). TextParserError 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 で定義されています。

MPI 対応版の TextParser::read では, rank=0 のプロセスで, パラメータファイルを読み込み, 内容をすべてのノードに分配します. rank 数 0 のプロセスが実行される計算機からアクセス出来る様にパラメータファイルの場所を指定する必要があります. 並列実行環境や, ファイルシステムに注意してください.

MPI 対応版の TextParser::read_local は, MPI での利用が前提の関数で, 各ノードが独立して指定されたパラメータファイルを読み込みます. MPI 非対応版の TextParser::read_local では, エラーメッセージを表示し, エラーを返します.

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す. したがって, 依存関係付き値は, 確定した条件による定義値になります.

格納しているパラメータのデータを破棄する関数は, 次の様に定義されています.

パラメータデータの破棄

```
TextParserError TextParser::remove();
```

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

4.2.6 ラベルの取得と値の取得 (フルパス)

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ます.

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 で定義されています。

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 で定義されています。

値の型を取得する

```
TextParserValueType 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 で定義されています.

戻り値 パラメータの値の型 TextParserType(表 4.1) を参照してください.

4.2.7 既知のデータ型値の取得 (フルパス)

あらかじめ格納されたデータの型がわかっているパラメータの値を取得する関数はそれぞれ次の様に定義されています.

整数値を取得する

```
bool getInspectedValue(const std::string label, int &ct );
```

label パラメータへのパス

ct 変数返却用 (出力引数)

戻り値 処理成否

備考 ラベルに配列的表現を含む場合, 必ず実 index を指定すること

単精度実数値を取得する

```
bool getInspectedValue(const std::string label, float &ct );
```

label パラメータへのパス

ct 変数返却用 (出力引数)

戻り値 処理成否

備考 ラベルに配列的表現を含む場合, 必ず実 index を指定すること

倍精度実数値を取得する

```
bool getInspectedValue(const std::string label, double &ct );
```

label パラメータへのパス

ct 変数返却用 (出力引数)

戻り値 処理成否

備考 ラベルに配列的表現を含む場合, 必ず実 index を指定すること

—— 文字列値を取得する ——

```
bool getInspectedValue(const std::string label, string &ct );
```

label パラメータへのパス

ct 変数返却用 (出力引数)

戻り値 処理成否

備考 ラベルに配列的表現を含む場合, 必ず実 index を指定すること

—— 整数型ベクトル値を取得する ——

```
bool getInspectedVector(const std::string label, int *vec, const int nvec );
```

label パラメータへのパス

vec ベクトル格納配列ポインタ (出力引数)

nvec ベクトルサイズ

戻り値 処理成否

備考 ラベルに配列的表現を含む場合, 必ず実 index を指定すること

—— 単精度実数型ベクトル値を取得する ——

```
bool getInspectedVector(const std::string label, float *vec, const int nvec );
```

label パラメータへのパス

vec ベクトル格納配列ポインタ (出力引数)

nvec ベクトルサイズ

戻り値 処理成否

備考 ラベルに配列的表現を含む場合, 必ず実 index を指定すること

—— 倍精度実数型ベクトル値を取得する ——

```
bool getInspectedVector(const std::string label, double *vec, const int nvec );
```

label パラメータへのパス

vec ベクトル格納配列ポインタ (出力引数)

nvec ベクトルサイズ

戻り値 処理成否

備考 ラベルに配列的表現を含む場合, 必ず実 index を指定すること

—— 文字列型ベクトル値を取得する ——

```
bool getInspectedVector(const std::string label, std::string *vec, const int nvec );
```

label パラメータへのパス

vec ベクトル格納配列ポインタ (出力引数)

nvec ベクトルサイズ

戻り値 処理成否

備考 ラベルに配列的表現を含む場合, 必ず実 index を指定すること

4.2.8 ラベル相対パスアクセス用関数

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 TestParser::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.

ノード移動

```
TextParserError TestParser::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.

カレントノードの子ノードのラベル取得

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

labels 子ノードへのラベルのリスト (相対パス)

order ラベルの出力順 0:データ格納順 1:配列ラベルのインデックス順 2:パラメータファイル内の出現順

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

カレントノードのリーフラベル取得

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

labels リーフへのラベルのリスト (相対パス)

order ラベルの出力順 0:データ格納順 1:配列ラベルのインデックス順 2:パラメータファイル内の出現順

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

4.2.9 型変換用関数

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ます.

文字列の値の型変換

```

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.

ただし, convertBool については, 表 4.4 のような変換になります.

表 4.4 convertBool の変換表

値の文字列 value	convertBool 戻り値 (bool)
true	true
TRUE	true
1	true
false	false
FALSE	false
0	false

4.2.10 ベクトル型の値の分解

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ます.

ベクトル型パラメータの要素 (リスト) の取得

```

TextParseError TestParser::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.

4.2.11 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

```

TextParseError 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してください.

戻り値 エラーコード (=0: no error). TextParserError 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んでください.

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

4.2.12 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). TextParserError 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). TextParserError 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). TextParserError 参照.

表 4.5 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

4.2.13 その他の API

—— ラベルの有無をチェック ——

```
bool chkLabel(const std::string label);
```

label チェックするラベル（絶対パス）

戻り値 成否.

備考 ラベルに配列的表現を含む場合，必ず実 index を指定すること.

—— ノードの有無をチェック ——

```
bool chkNode(const std::string label);
```

label チェックするノード（絶対パス）

戻り値 成否.

備考 ラベルに配列的表現を含む場合，必ず実 index を指定すること.

—— ノード以下の nnode 番目の文字列を取得する ——

```
bool getNodeStr(const std::string label, const int nnode, std::string &ct);
```

label ノード（絶対パス）

nnode 取得する文字列が現れる順番

ct 取得した文字列（出力引数）

戻り値 成否.

備考 ラベルに配列的表現を含む場合，必ず実 index を指定すること.

—— 指定ノード直下のラベルの数を数える ——

```
int countLabels(const std::string label);
```

label ノード（絶対パス）

戻り値 ラベルの数（エラー，もしくはない場合は-1）

備考 ラベルに配列的表現を含む場合，必ず実 index を指定すること.

指定ノードのラベル文字列リストを作成

```
int getArrayLabels(const std::string label, std::vector<std::string> &labels);
```

label ノード (絶対パス)

labels ラベル文字列のリスト (出力引数)

戻り値 ラベルの数 (エラーの場合は-1)

備考 引数 label に配列的表現 ([@]) を含む場合、一致する全てのラベル文字列を返す。

4.3 C 言語での利用方法

C 言語で本ライブラリを利用する場合、TextParser.h をインクルードします。TextParser.h 内部では、C 言語用の API 関数が用意されており、それら呼び出してライブラリを利用します。

4.3.1 include ファイル

C 言語で本ライブラリを利用する場合、TextParser.h をインクルードします。TextParser.h 内部では、C 言語用の API 関数が用意されています。プログラム内で使用する型のうち、ユーザーの利用するものは C++ 同様 TextParserCommon.h に定義されています。

4.3.2 TextParserValueType

TextParserValueType は、C++ と同様です。TextParserCommon.h 内での定義、表 4.1 で示されています。

4.3.3 TextParserError

TextParserError は、C++ と同様です。TextParserCommon.h 内での定義は表 4.5, 4.3 に示されています。

4.3.4 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 で扱います。

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). TextParserError で定義される。

4.3.5 ファイル IO とメモリの開放

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ます。

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 で定義されています。

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 で定義されています。

MPI 対応版の tp_read では、rank が 0 のプロセスで、パラメータファイルを読み込みます。rank 数 0 のプロセスが実行される計算機からアクセス出来る様にパラメータファイルの場所を指定する必要があります。

MPI 対応版の tp_read_local では、すべてのプロセスで指定されたパラメータファイルを、独立に読み込みます。MPI 非対応版の tp_read_local では、ファイルは読み込まず、エラーメッセージを表示してエラーコードを返します。

並列実行環境や、ファイルシステムに注意してください。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す。したがって、依存関係付き値は、確定した条件による定義値になります。

格納しているパラメータのデータを破棄する関数は、次の様に定義されています。

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。

4.3.6 ラベルの取得と値の取得 (フルパス)

データ構造に格納しているパラメータ全てのリーフの個数を取得する関数, インデックス *i* で指定された *i* 番目のリーフのラベル (フルパス) 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ます.

全てのリーフの個数の取得 —

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

tp_hand TextParser インスタンスへのポインタ

nleaves 全てのリーフの個数

戻り値 エラーコード (=0: no error). TextParseError で定義されています.

i 番目のリーフラベル —

```
int tp_getLabel(TP_HANDLE tp_hand,int i,char* label);
```

tp_hand TextParser インスタンスへのポインタ

i インデックス

label ラベル (フルパス)

戻り値 エラーコード (=0: no error). TextParseError で定義されています.

ラベル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 で定義されています.

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 4.1) を参照してください.

戻り値 エラーコード (=0: no error). TextParseError で定義されています.

4.3.7 ラベル相対パスアクセス用関数

カレントノードの取得, ノードの移動, 子ノードの数の取得, 子ノードの取得, リーフの数の取得, リーフの取得は, 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.

ノードの移動

```
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.

カレントノードの子ノードの数の取得

```
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 子ノードの総数

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

カレントノードの子ノードのラベル取得

```
int tp_getIthNode(TP_HANDLE tp_hand, int i, char* label);
```

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

tp_hand TextParser インスタンスへのポインタ

i インデックス i 番目のノードのラベルを指定

label 子ノードへのラベル (相対パス)

order ラベルの出力順 0: tp_getIthNode 同様 1: 配列ラベルのインデックス順 2: パラメータファイル内の出現順

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

カレントノードのリーフの数の取得

```
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 カレントノードのリーフの総数

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

カレントノードのリーフのラベル取得

```
int tp_getIthLeaf(TP_HANDLE tp_hand,int i,char* label);
int tp_getIthLeafOrder(TP_HANDLE tp_hand,int i,char* label,int order);
```

tp_hand TextParser インスタンスへのポインタ
i インデックス i 番目のリーフのラベルを指定
label リーフのラベル (相対パス)
order ラベルの出力順 0:tp_getIthLeaf 同様 1:配列ラベルのインデックス順 2:パラメータファイル内の出現順
戻り値 エラーコード (=0: no error). TextParseError 参照.

4.3.8 型変換用関数

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ます.

文字列の値の型変換

```
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.

ただし, tp_convertBool については, int 型で返し, 表 4.7 のような変換になります.

表 4.6 tp_convertBool の変換表

値の文字列 value	tp_convertBool 戻り値 (int)
true	1
TRUE	1
1	1
false	0
FALSE	0
0	0

4.3.9 ベクトル型の値の分解

ベクトル型のパラメータの値の要素数を取得する関数, i 番目の要素 (文字列) を取得する関数は, 次の様に用意されています.

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取得

```
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)

ielem インデックス ielem 番目の要素を指定

velem 要素の値 (文字列)

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

4.3.10 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してください .

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んでください .

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

4.3.11 値範囲指定記述用 API

— 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.

— 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 展開した数値の配列

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 展開した数値の配列

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.

4.4 Fortran90 での利用方法

Fortran90 で本ライブラリを利用する場合, TextParser.inc をインクルードしてください. TextParser.inc にはユーザーが用いる API 関数が定義されています. 又, プログラム中で利用するエラーコード TextParseError や 値の型 TextParserValueType は, 表 4.1, 4.5, 4.3 を参照してください. 関数の引数で文字列を取得していますが, その際に用いる文字列は, 文字列の長さ分の空白で初期化してから利用してください. 詳しくは, Examples ディレクトリの例を参照してください.

リンク時に, C 言語でのオプション -lstdc++ -lTP -L\${prefix}/lib に加えて, オプション -lTP.fapi が必要ですので追加してください.

4.4.1 instance の生成 , 消滅とシングルトンへのアクセス

FORTTRAN 内では TextParser のインスタンスを INTEGER*8 で扱います . 適宜利用プログラム内で定義してください .

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 .

4.4.2 ファイル IO とメモリの開放

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ます .

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 で定義されています .

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 で定義されています。

MPI 対応版の TP_READ では, rank が 0 のプロセスで, パラメータファイルを読み込みます. rank 数 0 のプロセスが実行される計算機からアクセス出来る様にパラメータファイルの場所を指定する必要があります. 並列実行環境や, ファイルシステムに注意してください.

MPI 対応版の TP_READ_LOCAL では, 全プロセスが, 指定されたファイルを独立に読み込みます. MPI 非対応版の TP_READ_LOCAL は, ファイルは読み込まず, エラーメッセージを表示してエラーコードを返します. MPI プログラムでない場合は, READ_LOCAL を使用しないでください.

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す. したがって, 依存関係付き値は, 確定した条件による定義値になります.

格納しているパラメータのデータを破棄する関数は, 次の様に定義されています.

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.

4.4.3 ラベルの取得と値の取得 (フルパス)

データ構造に格納しているパラメータ全てのリーフの個数を取得する関数, インデックス i で指定された i 番目のリーフのラベル (フルパス) 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ます.

全てのリーフの個数の取得

```
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 全てのリーフの個数

戻り値 エラーコード (=0: no error). TextParseError で定義されています。

i 番目のリーフラベル

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

ptr TextParser インスタンスへのポインタ

i インデックス インデックスは Fortran の場合, 1 から始まります.

label ラベル (フルパス)

戻り値 エラーコード (=0: no error). TextParseError で定義されています.

ラベル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 (文字列)

value パラメータの値 (文字列)

戻り値 エラーコード (=0: no error). TextParseError で定義されています.

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 4.1) を参照してください.

戻り値 エラーコード (=0: no error). TextParseError で定義されています.

4.4.4 ラベル相対パスアクセス用関数

カレントノードの取得, ノードの移動, 子ノードの数の取得, 子ノードの取得, リーフの数の取得, リーフの取得は, 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.

ノードの移動

```
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.

カレントノードの子ノードの数の取得

```
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 子ノードの総数

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

カレントノードの子ノードのラベル取得

```
INTEGER TP_GET_ITH_NODE(INTEGER*8 ptr, INTEGER i, CHARACTER(len=*) label)
```

```
INTEGER TP_GET_ITH_NODE_ORDER(INTEGER*8 ptr, INTEGER i, CHARACTER(len=*) label,  
                                INTEGER order)
```

ptr TextParser インスタンスへのポインタ

i インデックス i 番目のノードのラベルを指定, (1 から始まる)

label 子ノードへのラベル (相対パス)

order ラベルの出力順 0:TP_GET_ITH_NODE 同様 1:配列ラベルのインデックス順 2:パラメータファイル内の出現順

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

カレントノードのリーフの数の取得

```
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 カレントノードのリーフの総数

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

カレントノードのリーフのラベル取得

```
INTEGER TP_GET_ITH_LEAF(INTEGER*8 ptr, INTEGER i, CHARACTER(len=*) label)
```

```
INTEGER TP_GET_ITH_LEAF_ORDER(INTEGER*8 ptr, INTEGER i, CHARACTER(len=*) label,  
                                integer order)
```

ptr TextParser インスタンスへのポインタ

i インデックス i 番目のリーフのラベルを指定

label リーフのラベル (相対パス)

order ラベルの出力順 0:TP_GET_ITH_LEAF 同様 1:配列ラベルのインデックス順 2:パラメータファイル内の出現順

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

4.4.5 型変換用関数

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ます。

文字列の値の型変換

```

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.

表 4.7 tp_CONVERT_の変換表

値の文字列 value	TP_CONVERT_LOGICAL 戻り値 (LOGICAL)
true	.true.
TRUE	.true.
1	.true.
false	.false.
FALSE	.false.
0	.false.

4.4.6 ベクトル型の値の分解

ベクトル型のパラメータの値の要素数を取得する関数, i 番目の要素 (文字列) を取得する関数は, 次の様に用意されています.

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取得

```
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)

ielem インデックス (1 から始まる) ielem 番目の要素を指定

velem 要素の値 (文字列)

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

4.4.7 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)
```

ptr 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してください .

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んでください .

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

4.4.8 値範囲指定記述用 API

— 様値範囲指定時のパラメータ取得 —

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

ptr 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.

— 様値範囲指定時のパラメータ展開 —

```
INTEGER TP_EXPAND_RANGE(INTEGER*8 ptr, CHARACTER(len=*) 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 展開した数値の配列

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 展開した数値の配列

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.

第 5 章

パラメータの記述方式

本ライブラリが扱うパラメータデータベースはノードの階層構造にリーフ（ラベル/値のペア）を格納したものである。パラメータデータベースの主な構成要素はラベル、ノード、リーフ、値である。

5.1 ラベル

ラベルは、ノードまたはリーフを表す識別子である。以下にラベルの仕様を示す。

- ノードとリーフの識別子である。

```
[ノード]
ラベル {    }
[リーフ]
ラベル = 値
```

- “=” または “{” または @dep (後述) 内の “==” または @dep “!=” の記号の左側にある、空白を途中に含まない文字列とする。記号との間に改行や空白があっても構わない。
 - 二重引用符 [“] で囲まれた文字列は、二重引用符で囲まれていない文字列と等値である。
 - 通常使用できる文字は [a-zA-Z0-9_] である。
 - ラベルはノードパス的な表記（以下、ラベルパスと呼ぶ）も可能であり、絶対パス、相対パスで表すことが出来る。パスの記述用に [/] も使用出来る。
 - 文字列の比較において大文字/小文字の区別はしない。

以下に例を示す。

```
[通常]
foo または "foo" または F00 または "F00"
この 4 つの記法は全て同じラベルになる。
[絶対パス]
/foo/baz または "/foo/baz"
[相対パス]
./foo/baz または "./foo/baz"
foo/baz または "foo/baz"
../bar または "../bar"
```

- 同じノード階層内にあるノードには同一のラベルを付ける事はできない. 又, 同じノード階層内にあるリーフには同一のラベルを付ける事はできない.

可能な例

```
foo {  
    qux { baz = "val1" }  
    qux { x = "val2" }  
}
```

エラーになる例

```
foo {  
    qux = 1  
    qux = 2    // エラー  
}
```

- 同じノード階層内に同一ラベルのノードとリーフが存在してはならない.

```
foo {  
    qux { baz = "val1"}  
    qux = "val2"    // エラー  
}
```

- 絶対パスに同じラベルが複数含まれてはならない.

```
qux {  
    qux {          // エラー  
        baz = "val1"  
    }  
}  
qux {  
    qux = "val1"    // エラー  
}
```

- ラベル文字列末尾に”[@]”と書くことで, 配列形式のラベルを定義できる. 配列形式のラベルはパースされた時点で@がノード内で初期化された一意な配列添え字に変換される.

```
foo {  
    bar[@] { param[@]=1,  
             param[@]=2  
    }  
    bar[@] { param[@]=3,
```

```

        param[@]=4
    }
}

```

パース後，以下の様に変換

```

foo {
    bar[0] { param[0]=1,
            param[1]=2
    }
    bar[1] { param[0]=3,
            param[1]=4
    }
}

```

- 同一ノード内でノードとリーフに同じ配列形式ラベルを定義することは出来ない.

```

foo {
    param[@] { abc = 1 }
    param[@]=1    // エラー
}

```

- 絶対パスに同じ配列ラベルが複数含まれてはならない.

```

param[@] {
    param[@] {
        param[@] = 1
    }
}

```

- ラベルごとにインデントを揃える必要はない. 下記の2例は等価である.

例 1

```

one {
    two {
        three = 1
    }
}

```

例 2

```

one {
    two{
    three = 1
    }
}

```

5.2 ノード

ノードは、ラベルで名前、位置を定義されたもので、リーフまたはノードを保持するものである。以下にノードの仕様を示す。

- ノードは、リーフ又はノードを保持する。ノードはラベルで始まり、その後の左中括弧”{”でノード内要素が始まり、右中括弧”}”で終わる。ノードはネストすることができる。下記 3 例はいずれも等価である。

```
ラベル { }  
(例)  
foo {  
}  
foo {    }  
foo  
{  
}
```

- 相対パス、絶対パスによる定義が可能である。相対ノードパスは、一階層上を表す “../” と 当該のノードを表す “./” が使用できる。

```
絶対パス  
/foo/qux {  
}
```

相対パス 次の 2 例は等価である。

```
例 1  
foo {  
    ./qux/abc { // ./は省略可能  
    ../cde/ {   }  
}  
}
```

```
例 2  
foo {  
    ./qux/abc { }  
    ./qux/cde { }  
}
```

- 一つのノードを分割して定義することが可能

```
foo {
```

```
        baz=1
    }
    foo {
        abc=2
    }
```

- ノードの階層ごとに括弧でくくるが、その位置を揃える必要はない。また、改行の有無に左右されない。下記の2例は等価である。

例 1

```
one { two{ three = 1  }
    }
```

例 2

```
one      {
    two{   three = 1
    }
}
```

5.3 リーフ

本ライブラリではノード内要素のラベル/値のペアをリーフと呼ぶ。以下にリーフの仕様を示してある。

- リーフはラベルと値が代入演算子”=”で接続されたものである。複数のリーフはカンマ”,”もしくはスペース” ”や改行によって区切られる。

```
ラベル = 値
qux="val"
param0=1, param1=2
param0=1 param1=2
```

- 相対パス, 絶対パスによる定義

```
絶対パス
/foo/qux="val1"
相対パス
./bar/baz=1    // ./は省略可能
```

- リーフはパラメータファイル内でどのような順で記述されても良い。

5.4 値

リーフでラベルとペアになっているものを値と呼ぶ。値には5種類あり、それぞれ次のような仕様になっている。

1. 文字列

- 二重引用符””で囲まれた ASCII 文字列
- 使用できる文字は [a-zA-Z0-9_-] である。
- 文字列の比較において大文字/小文字の区別はしない。

文字列の例

"ON" "on"

2. 数値

- 整数
 - “(+/-) 0-9” の “+/-” の後にスペースが有っても可
 - 余分な 0 が有っても可

数値の例

123

0

+ 123

- 0

01

- 浮動小数点
 - “(+/-) 0-9.(0-9)”, “(+/-).0-9” の “+/-” の後にスペースが有っても可

浮動小数点の例

0.10

+ 0.2

- 0.3

3.

.123

- 指数
 - “(+/-) 0-9.(0-9) e/d (+/-) 0-9”, “(+/-).0-9 D/E (+/-) 0-9” の “+/-” の後にスペースが有っても可
 - 同じく e/d の前後にスペースが有っても可

指数の例

1.0 e 10

+ 0.1 E - 5

- 1.2 d 3

.3 e - 4

3. ベクトル

- 文字列もしくは数値の順序付けされたセットである。
- 左括弧“(”で始まり、右括弧”) ”で終わる。
- 値はカンマ”, ”で区切られる。
- 1つのベクトル内の値の要素は全て同じ型でなければならない。
- 未定義値 (UNDEF) のベクトルも可能。
- ベクトルのネストはできない。

ベクトルの例

(1.0 e 10, 1.0 e 10 , 1.0 e 10)

(+ 0.1 E - 5 , - 0.3 E - 5)

(- 1.2 d 3, - 1.2, 3 , .3 e - 4)

("ONE", "TWO", "THREE")

(UNDEF, UNDEF, UNDEF)

4. Numerical Limits

- 次のような記述で limits.h で記述される Numerical Limits の値を利用出来る。尚、記述の際は大文字で指定する。

CHAR_MIN //char 最小値

CHAR_MAX //char 最大値

SHORT_MIN //short 最小値

SHORT_MAX //short 最大値

INT_MIN //int 最小値

INT_MAX //int 最大値

LONG_MIN //long 最小値

LONG_MAX //long 最大値

LONG_LONG_MIN //longlong 最小値

LONG_LONG_MAX //longlong 最大値

FLOAT_MIN //float 最小値

FLOAT_MAX //float 最大値

DOUBLE_MIN //double 最小値

DOUBLE_MAX //double 最大値

5. 依存関係付き値

- "@dep"に続き指定されるC言語ライクな三項演算子による条件式に応じて当該の値を制御可能である。
- "@dep"に続く左括弧"("で始まり、右括弧")"で終わる部分に条件式を記述する。
- 条件式の左辺は依存するパラメータのラベルパス、右辺はその取りうる値である。
- ラベルパスは相対パスも可能。
- ラベルパスとして配列形式ラベル(例: "param[@]")を使うことは出来ない。
- 条件式の右辺の値は一つの文字列又は数値に限る。
- 条件式の右辺の値としてベクトルや未定義値、別の依存関係付き値を指定することは出来ない。
- 条件演算子には"=="と"!="が利用できる。
- 条件式は"&&"と"||"演算子で接続し、複数指定可能で括弧の順に処理する。
- 条件式の右辺の値が数値の場合は double に変換して"=="と"!="を判定する。
- 条件式に続いて"?"の次に条件式が「真」の時の値、次に":"に続いて条件式が「偽」の時の値を記述する。
- 値はベクトルでも可能。":"の左右でベクトルの次元が異なっても良い。
- ":"の左右で値の型が異なっても良い。
- 又、値は未定義値でも可能だが、値として別の依存関係付き値を指定することは出来ない。
- 依存関係付き値で使用するラベルが定義されていない場合は全パラメータのパーズ終了後に再度評価を行なう。
- 最終的に定義されていないラベルを参照する依存関係付き値には未定義値 (UNDEF) をセットし、警告を表示する。

依存関係付き値の書式

@dep (条件式) ? 値 : 値

条件式

```
ラベル == 値
ラベル != 値
条件式 && 条件式
条件式 || 条件式
```

依存関係付き値の例 (例)

```
@dep (swt == "on") ? 1 : 2
@dep ((swt == "on")) ? 1 : 2
@dep ( a == 1) ? (1,2) : (0,1,2)
@dep ( a == 1) ? (1,2) : ("a","b")
@dep ( a == 1) ? 5 : UNDEF
@dep ( (swt1==0) && (swt2==1) ) ? "a" : "b"
@dep ( (swt1==0) || (swt2!=1) ) ? "a" : "b"
```

6. 未定義値

- ラベルの値に対し未定義であることを明示するため未定義値 (UNDEF) を記述することが出来る。
- 未定義値はパラメータのパーズの際に警告を表示する。また、パラメータを取得する際に未定義値だったら警告を表示する。

未定義値の例

```
baz=UNDEF
```

7. 任意の数値による値範囲指定

- @list=(val1,val2,val3,...valN) と記述することで、任意の数値による値範囲指定を利用することが可能である。尚、数値は、TextParser で規程される数値の記述に従う。

数値による値範囲指定の例

```
label = @list(1,3,5,6.0,3.14e-03,-7)
```

8. 一様な増減幅を持つ値範囲指定

- @range(from,to,step) と記述する。from,to,step はそれぞれ開始値、終了値、増減値である。尚、数値は、TextParser で規程される数値の記述に従う。また、開始値、終了値の増減の方向と増減値の符号が不整合の場合にエラーとなる。

一様な増減幅を持つ値範囲指定

```
label = @range(1,10,2) //@list(1,3,5,7,9) と同値
```

```
label = @range(1,3) //stepの省略は1または-1 @list(1,2,3) と同値。
```

5.5 改行

パラメータの記述は要素の単位で改行が可能である。途中で改行できないのはラベル、文字列の値、数値、未定義値 (UNDEF)、依存関係の”@dep”である。

5.6 コメント

コメントは C/C++ ライクな /* ~ */、及び // ~ が利用できる。

5.7 パラメータファイルのサンプル

Examples ディレクトリには, パラメータパーサファイルの例が多数あります. パラメータパーサファイルの例は, Examples/tpp_examples ディレクトリに格納されています. パラメータパーサファイルの例は, 次の様な構成になっています.

- 文法的に正しい例

correct_basic_*.txt 基本的なツリー構造と値 (数値型と数値型ベクトル) のテスト用
correct_string_*.txt 値 (文字列型, 文字列型のベクトル) のテスト用
correct_label_*.txt ラベル (ノード/リーフ) の指定のテスト用
correct_labelarray_*.txt 配列ラベル (ノード/リーフ) の指定のテスト用
correct_cond_*.txt 依存関係付き値 ''@dep'' のテスト用
correct_range_list_*.txt ''@range'', ''@list'' のテスト用

- 文法的に誤った例

incorrect_basic_*.txt 基本的なツリー構造と値 (数値型と数値型ベクトル) のテスト用
incorrect_label_*.txt ラベル (ノード/リーフ) の指定のテスト用
incorrect_labelarray_*.txt 配列ラベル (ノード/リーフ) の指定のテスト用
incorrect_cond_*.txt 依存関係付き値 ''@dep'' のテスト用
correct_range_*.txt ''@range'' のテスト用
correct_list_*.txt ''@range'', ''@list'' のテスト用

また基本的な文法については, プログラム説明書も参照してください.

第 6 章

アップデート履歴

本マニュアルのアップデート情報について記します。

6.1 アップデート履歴

- 2014-06-21
 - 軽微な修正.
- 2014-06-14
 - 最新のコードに対応 (ただし、Python インターフェイス以外).
- 2013-05-07
 - パッケージ変更.
- 2012-11-26
 - @range,@list の Example を追加.
- 2012-05-28
 - MPI 対応版についての記述を追加.
 - パラメータファイルの例の部分の修正.
- 2012-05-07
 - c 言語/Fortran90 用の説明を追加.
- 2012-04-28
 - 初版リリース.