

# ゲームデータ仕様

－ ゲームデータの変換とフォーマット －

2014 年 1 月 20 日 初版

板垣 衛

## ■ 改訂履歴

版	リリース	担当	改訂内容
初版	2014 年 1 月 20 日	板垣 衛	(初版)

**■ 目次**

■ 概略 .....	1
■ 目的 .....	1
■ 基本用語 .....	1
▼ 「ゲームデータ」 .....	1
■ 要件定義 .....	2
▼ 基本要件 .....	2
▼ 要求仕様／要件定義 .....	2
■ 仕様概要 .....	3
▼ 環境3 .....	
▼ ワークフロー .....	4
■ データ仕様 .....	4
▼ DB/Excel .....	5
▼ 拡張 JSON .....	5
● JSON 形式の採用について .....	5
● 拡張 JSON 仕様 .....	6
▼ データ定義 JSON .....	8
▼ 中間 JSON① .....	8
▼ 中間 JSON② .....	9
▼ フォーマット定義 JSON .....	9

---

▼ データ型定義リスト.....	14
▼ 計算式用拡張関数定義リスト.....	16
▼ チェック用 JSON.....	17
▼ C 言語ソース .....	19
▼ バイナリデータ.....	22
■ 値の演算と組み込み関数 .....	24
▼ 計算式解析関数：expr().....	25
■ 計算式 .....	25
▼ 計算式の構文 .....	26
▼ 応演算子 .....	27
▼ 組み込み関数 .....	28
▼ 拡張関数 .....	28
■ 処理仕様.....	29
▼ プリプロセッサ.....	29
▼ データ変換ツール .....	30
▼ データ変換実行バッチ .....	33
▼ データ変換処理（C# Tips） .....	35
● JSON.Net の使用 .....	35
● JSON 解析済みオブジェクトを C#スクリプト内で使用 .....	36
▼ データ取り込み処理（ランタイム） .....	38

---

---

▼ データアクセス処理（ランタイム） .....	38
<hr/>	
■ 計算式処理仕様 .....	38
▼ 計算式解析処理.....	38
▼ 計算式データの構造.....	42
▼ 計算式実行処理（ランタイム） .....	43
<hr/>	
■ データ変換作業環境の改善.....	43
▼ バッチファイル の利用 .....	43
▼ SCons の利用.....	43

## ■ 概略

本書における「ゲームデータ」とは、大まかには、グラフィック関係データとサウンド関係データ以外のデータ全般を指す。

その多くはプランナーが扱うデータで、ゲームを制御するための設定やパラメータなどのことである。多彩なデータを扱い、ゲーム固有のデータ構造となるものが多い。

本書は、「ゲームデータ」の入力フォーマットと実機上のデータフォーマット、および、その変換・取り込み処理に関する基本仕様を規定する。

## ■ 目的

汎用化したゲームデータ処理により、バージョン整合処理や数式解析処理などの高度な処理も標準化し、作業の効率化と安全性の向上を目的とする。

## ■ 基本用語

### ▼ 「ゲームデータ」

本書においては、「ゲームデータ」という用語を下記の意味で扱う。

- ・ グラフィック関係データとサウンド関係データを除くデータ全般。一般的には、これらのデータを含むリソース全般をまとめて「ゲームデータ」と呼ぶが、本書では区別して扱う。
- ・ グラフィック関係データとサウンド関係データであっても、それを制御するためのゲーム固有のデータは「ゲームデータ」に類する。
- ・ 何らかの処理設定やデバッグ用データなど、プログラマーが扱うデータもまた「ゲームデータ」である。
- ・ 基本的には、予め定義された静的なデータを指し、ファイル（リソース）として扱われる。
- ・ メモリ上で内容が変動する動的なデータやセーブデータなどは、「ゲームデータ」の範疇に含まない。しかし、例えば「ゲームデータを読み込んでセーブデータを復元する」といった、動的なデータを再現するためのゲームデータの活用はありえる。
- ・ PlayStation 系では「インストールデータ」を指して「ゲームデータ」と呼ぶが、本書

における「ゲームデータ」はそれとは別物である。

## ■ 要件定義

### ▼ 基本要件

本書が扱うシステムの基本要件は下記の通り。

- ・ ゲームデータの入力データをテキストファイル形式で扱う。
- ・ Excel や DB で管理するデータの場合、直接実機向けのデータに変換せず、いったんテキストファイルに変換して扱う。これにより、全てのゲームデータに対して、一貫したデータ変換とフォーマットを使用するものとする。なお、Excel/DB に関する仕様は本書の範疇外とし、別途仕様を策定する。
- ・ テキストファイルをバイナリデータに変換する汎用ツールを作成する。
- ・ バイナリデータを実機（ゲーム）に取り込むための汎用処理を作成する。
- ・ テキストファイルの取り込みは、実機上では行わないものとする。（パーサーを実装しない。）

### ▼ 要求仕様／要件定義

以下、本書が扱うシステムの要件を定義する。なお、要件として不確定の要求仕様も併記する。

- ・ テキストファイルは、JSON を基本フォーマットとして統一する。
- ・ テキストファイルの文字コードは UTF-8 とし、日本語や欧州文字に対応する。
- ・ テキストファイルは、下記の拡張仕様（JSON が非対応の仕様）に対応する。
  - JavaScript 形式のコメント文を使用できる。（例：`// comment`、`/* comment */`）
  - C 言語形式の`#include` 文と`#define` 文を使用できる。
  - データ部に四則演算を用いることができる。（例：`{ "age": 30 + 3, ... }`）
  - データ部に CRC 変換やゲーム用計算式変換などの特殊な関数を使用できる。（例：`{ "id": CRC("c0010"), ... }, { "condition": Expr("IsFlag(¥"AlreadyMetOldMan¥") == True && GetChapter() >= 2"), ... }`）

注：JSON データを MongoDB などのドキュメント指向データベース（BSON 形式で保存される）で扱う場合、これらの拡張仕様が使えないので注意。同様の情報を扱うための別の仕様も合わせて策定する。

- ・ テキストファイルからバイナリデータに変換するための変換設定もまた JSON 形式のテキストデータとして定義する。期待される値の範囲など、エラー判定用の設定も可能。
- ・ 不定長の配列をメンバーに持つ構造体にも対応。
- ・ 専用のデータ変換ツールを通して、バイナリデータを出力する。
- ・ データ変換ツールは、下記の仕様に対応する。
  - CUI ツールとして構成し、単純に一つのテキストファイルを一つのバイナリファイルに変換する。これは、任意のバッチ処理や他のツールからの呼び出しなどに対応しやすい形式である。
  - エンディアンの指定、ポインタのビット数（32 or 64）指定に対応。
  - 【できれば】シフト JIS などのエンコーディングの指定に対応。
  - データ構造／内容のエラーを検出した場合、出力ファイルは作成されず、エラーが通知される。
- ・ バイナリデータは、メモリ上のイメージとしてほぼそのまま取り込める。
  - 文字列のポインタ変換などの処理も取り込み時に同時に行われる。
- ・ バイナリデータの構造が変更され、実機上の構造とずれた場合、取り込み処理は自動的にその事を検出し、項目毎のデータ取り込み処理を行う。要は、データ構造が変わっても、プログラムが極力正常に動作するようにする。

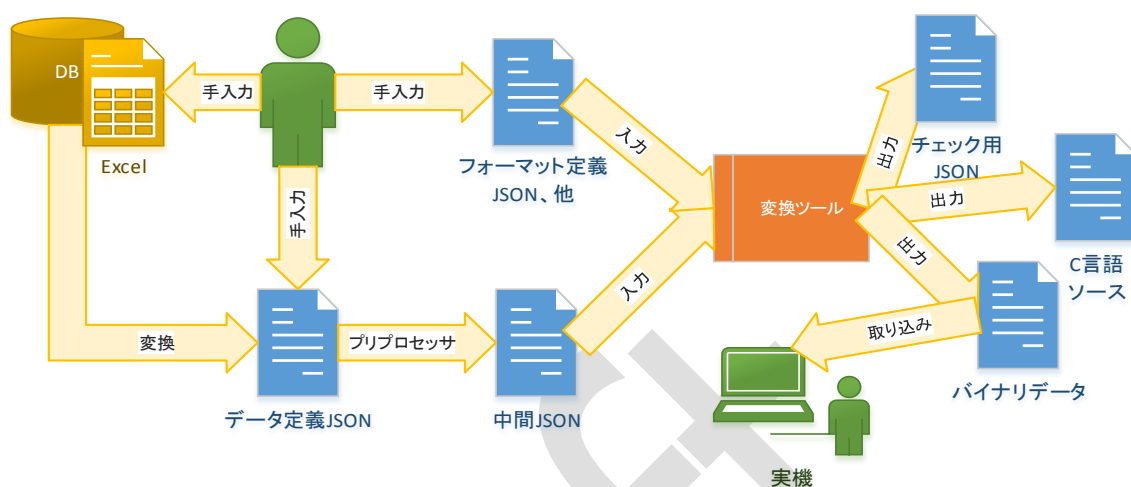
## ■ 仕様概要

### ▼ 環境

- ・ OS : Windows 系 PC (XP 以上) 32bit/64bit
- ・ 必須ツール① : テキストエディタ ※なんでもよい
- ・ 必須ツール② : MinGW(GCC) ※プリプロセッサ
- ・ 必須ツール③ : 変換ツール ※独自開発
- ・ 使用ツール (オプション) : Python + SCons



## ▼ ワークフロー



- ・ DB／Excel ..... 元データ ※この仕様書では扱わない。
- ・ データ定義 JSON ..... テキストで記述されたデータ
- ・ 中間 JSON ..... プリプロセッサを通した後の「データ定義 JSON」
- ・ フォーマット定義 JSON ... データ変換のためのフォーマットとルールを定義
- ・ データ型定義リスト ..... 「フォーマット定義 JSON」に付随するリスト ※「データ型」の仕様を定義
- ・ 計算式用拡張関数定義リスト ..... 「フォーマット定義 JSON」に付随するリスト ※計算式で使用する「拡張関数」を定義
- ・ 変換ツール ..... 「データ定義 JSON」を「バイナリデータ」や「C 言語ソース」に変換するツール
- ・ バイナリデータ ..... バイナリデータ（C 言語構造体と一致）
- ・ C 言語ソース ..... 構造体とバージョン整合用の構造定義情報
- ・ チェック用 JSON ..... バイナリデータと同じ内容を表す JSON ※内容確認／二次利用向け
- ・ 実機 ..... ゲームプログラム ※バイナリデータのみを扱える

## ■ データ仕様

以下、前述の「ワークフロー」に基づいて、各データの内容と構造について説明。

## ▼ DB/Excel

DB/Excel に関する仕様は本書では扱わない。別途仕様を策定。

なお、想定としては、ドキュメント指向 DB (MongoDB など) + RDB (PostgreSQL など) で管理し、Excel へのエクスポート/インポートでデータ編集する。DB には独自のバージョン管理とロック機構を備え、チーム開発を支援する。

DB に記録されるデータも JSON 形式とするが、Excel 化を考慮したデータ構造とするため、本書で示すデータ構造とは必ずしも一致しない。DB 上のデータを変換してデータ定義 JSON を出力する。

DB に記録される JSON は、一切の拡張仕様が使えない。しかし、この場合、直接テキストを編集することはないので、ほぼ問題はない。

## ▼ 拡張 JSON

「データ定義 JSON」に適用する記述仕様。

JSON の基本仕様に独自の拡張仕様を加えたもの。

「フォーマット定義 JSON」と、それに付随する「データ型定義リスト」「計算式用拡張関数定義リスト」にも、この「拡張 JSON」の記述仕様を適用する。

### ● JSON 形式の採用について

データフォーマットには JSON 形式を採用する。採用理由は下記の通り。

- テキストファイルによる柔軟で素早いデータ策定が見込める。
  - 「Excel⇒バイナリデータ出力」よりも、「Excel⇒テキスト⇒バイナリデータ」とすることで、ツール (Excel) の用意に先行してデータ策定と処理を作成できる。
  - 「テキスト⇒バイナリデータ」の変換ツールを汎用化できる。
- XML のように、メンバー名を持った柔軟な構造体 (オブジェクト) を扱える。
- 配列が扱える。
- XML よりも軽量で視覚的にも分かり易い。
  - XML の「<タグ>~</タグ>」のような記述の冗長性がなく、開始タグ (「タグ:」) のみを記述する。
  - 反面、XML の「属性」のような複雑なデータ記述はできない。その方がシンプルで良い。

- XML のような一般的なデータ記述仕様であり、多くのライブラリ（Java や .Net）やデータベース（MongoDB や CouchDB）などが対応している。
  - Ajax で利用が広がり注目を集めた。テキスト形式かつ軽量な点は http 通信にも向いている。XML の代替として注目。
- JavaScript 準拠のフォーマットであり、JavaScript やそれをサポートする多数の言語（ライブラリ）でそのままデータを取り込めるため、データの二次利用がし易い。

## ● 拡張 JSON 仕様

- 基本仕様：
  - JSON 仕様：[http://ja.wikipedia.org/wiki/JavaScript\\_Object\\_Notation](http://ja.wikipedia.org/wiki/JavaScript_Object_Notation)
  - ファイルの拡張子は .json。
  - エンコーディングは Unicode。基本的に UTF-8。
  - キーと値のペアでデータを記述するのが基本。
  - 値は、整数、小数、真偽値（true / false）、文字列、null を記述可能。文字列はダブルクォーテーションで囲む。
  - 整数の記法は 10 進法に限られる。「0127」のような 8 進記法、「0x12af」のような 16 進記法には対応しない。
  - 小数の記法には「1.0e-10」のような指数表記が可能。
  - 文字列には「¥n」「¥r」「¥¥」などのエスケープシーケンスが記述可能。
    - 【注意】 エスケープ文字を含む一部のシフト JIS コードの文字（「ソ」「能」「表」など）は正常に扱えない事に注意。エンコーディングはあくまでも Unicode。
  - 複数のデータを並べる場合は、「,」（コンマ）で区切って列挙。
    - 【注意】 Microsoft の Jscript など、最後のデータには「,」が付いているとエラーになるパーサーもあるので注意。
  - 配列を記述可能。
  - オブジェクト（構造体）を記述可能。
- キーと値の記述：
  - キーと値のペアは、「”キー”+ : + 値」の書式で記述。
  - キーには文字列のみが指定可能で、必ずダブルクォーテーションで囲む。

例：

```
"id" : 123,
"name" : "名前",
"value" : 1.23
```

- 配列の記述：
  - 「[ ] ~ [ ]」（角括弧）で囲む事で表現する。

例：

```
"array" : [ 1, 2, 3 ]
"data" : [ "山田", "田中", "佐藤" ]
```

多重配列も可能：

```
"array" :
[
    [1, 2, 3],
    [4, 5, 6]
]
```

➤ オブジェクトの記述：

- 「{」～「}」(波括弧)で囲む事で表現する。
- 一つのデータ項目は、「キー」+「:」+「値」のペアを記述。「キー」は構造体のメンバー名と同じ意味。重複禁止。ダブルクォーテーションで囲んで記述。

例：

```
{
    "id" : 123 ,
    "name" : "山田",
    "param" :
    {
        "str" : 10,
        "vit" : 20
    }
}
```

➤ データテーブルの記述：

- オブジェクトを配列で列挙する。

例：

```
"table" :
[
    { "id":1, "name":"山田", "param": { "str": 10, "vit": 2.0, "ext": [1, 2, 3], "is_cleared": true } },
    { "id":2, "name":"田中", "param": { "str": 11, "vit": 2.1, "ext": [4, 5, 6], "is_cleared": false } },
    { "id":3, "name":"佐藤", "param": { "str": 12, "vit": 2.2, "ext": [7, 8, 9], "is_cleared": true } }
]
```

➤ 独自拡張仕様：

- C 言語のコンパイラに依存し、多数のプリプロセッサ用の記述をサポート。
  - ・ JavaScript 形式 (C++形式) のコメント (「//」もしくは「/\*」～「\*/」) を使用可能。
  - ・ #include 文を使用可能。
  - ・ #define マクロ、#undef を使用可能。
  - ・ #pragma once を使用可能。
  - ・ #if, #ifdef, #ifndef ~ #elif, #else ~ #endif を使用可能。
  - ・ 文字列化演算子「#」、トークン連結演算子「##」を使用可能。
- 値に 8 進表記 (0 + 0～7 の数字)、16 進表記 (0x + 0～f の数字)、2 進表記 (0b + 0～1 の数字) を指定可能。書式は C 言語準拠。なお、2 進表記の書式は C++14 仕様準拠。
- 値に四則演算などの演算子を記述可能。

- 組み込み関数 (pow(), sqrt(), crc(), expr() など) を使用可能。
- キーのダブルクォーテーション表記は省略可能。

例：

```
//データ定義 JSON サンプル

//共通定義のインクルード
#include "common.jsonh"
//BASE_**マクロが定義されている

//キー定義
#define KEY_ID "id"
#define KEY_NAME "name"

//パラメータ計算マクロ
#define STR(x) BASE_STR + x
#define VIT(x) BASR_VIT + x

//名前用文字列作成マクロ
#define TO_NAME(s) #s

/**
 * キャラデータリスト
 */
"table":
[
    { KEY_ID: CRC("c0010"), KEY_NAME: TO_NAME(山田), param: {"str": STR(1), "vit": VIT(1)},
      "condition": expr("getChapter() >= 20") },
    { KEY_ID: CRC("c0020"), KEY_NAME: TO_NAME(田中), param: {"str": STR(2), "vit": VIT(2)} },
    { KEY_ID: CRC("c0030"), KEY_NAME: TO_NAME(佐藤), param: {"str": STR(3), "vit": VIT(3)} }
]
```

## ▼ データ定義 JSON

データを定義するための JSON。文章中では、短縮して「JSON データ」と表記する場合もあるが、「データ定義 JSON」と同義。

記述仕様は「拡張 JSON」形式。

オブジェクトの配列（いわゆる「テーブル」）で定義するのが基本形。「[」で始まり「」で終わる。その内部には、「{」～「}」で定義されたオブジェクト（構造体）を列挙する。

一つのファイルでは一つのデータ（テーブル）だけを扱う。ネストしたデータ構造も使用可能。

サンプルは、前述の「拡張 JSON」の仕様を参照。

## ▼ 中間 JSON①

データ定義 JSON にプリプロセッサを通した状態の中間データ。プリプロセッサには、フリーの C 言語コンパイラ MinGW(GCC) を使用。

プリプロセッサの機能に頼り、[#line](#) 文が埋め込まれた状態で出力する。これにより、変

換ツールがエラーを検出した際に、プリプロセス前の行位置や、インクルードファイル内のエラーであることを表示できる。

この状態では JSON パーサーに通せないので注意。

例：前述の「拡張 JSON」からの変換結果

```
※空行が多いのは、コメントや#define が除去された結果。
エラーメッセージ出力時に行番号がずれないように、残っている行の位置は変化しないため、空行が多くなる。

# 1 "common.jsonh" 1
(インクルードファイルが展開される。)

(インクルードファイルから元のファイルに戻る。最初の # 5 は、この位置から元のファイルの 5 行目が再開することを示す。)
# 5 "data.json" 2
(コンパイラによっては、「#line」という書式のものもある。この出書式は GCC 準拠。)
```

```
"table" :
[
  { "id" : CRC( "c0010" ), "name" : "山田", param: { "str": 10 + 1, "vit": 20 + 1 },
    "condition" : expr( "getChapter() >= 20" ) },
  { "id" : CRC( "c0020" ), "name" : "田中", param: { "str": 10 + 2, "vit": 20 + 2 } },
  { "id" : CRC( "c0030" ), "name" : "佐藤", param: { "str": 10 + 3, "vit": 20 + 3 } }
]
```

## ▼ 中間 JSON②

中間 JSON①に対して、値の演算と組み込み関数を処理した結果の JSON。`#line` 文も除去する。なお、`expr()` 関数は処理しない。

中間 JSON①への変換＋中間 JSON②への変換により、JSON パーサーに通せる状態になる。

データ変換ツールの実行中にメモリ内で処理されるが、確認用にファイル出力も可能。

例：前述の「中間 JSON①」からの変換結果

```
[
  { "id" : 745853103, "name" : "山田", param: { "str": 11, "vit": 21 },
    "condition" : expr( "getChapter() >= 20" ) },
  { "id" : 123306860, "name" : "田中", param: { "str": 12, "vit": 22 } },
  { "id" : 507687469, "name" : "佐藤", param: { "str": 13, "vit": 23 } }
]
```

## ▼ フォーマット定義 JSON

データフォーマットを定義するための JSON。データ変換ツールに変換方法を指定するために用いる。データ定義 JSON ファイルを入力して、バイナリデータファイルと C 言語

ソースファイルを出力するためのフォーマットとルールが定義される。

基本的な記述仕様は「拡張 JSON」形式に従う。記述内容は、フォーマット定義のための設定項目が定められている。下記のサンプルでその仕様を示す。赤い字で書かれた項目が、サンプル中で初出の定義項目であり、詳しい説明を併記する。

例：

```
//データフォーマット定義 JSON サンプル
{
    "name": "CharaData", //データフォーマット名 ※構造一致照合用。
    "majorVer": 1, //データフォーマットメジャーバージョン ※構造一致照合用。
    "minorVer": 0, //データフォーマットマイナーバージョン ※構造一致照合用。
    "comment": "キャラパラメータ構造定義", //コメント ※C 言語ソース用。

    "headerFileName": "charaData.h", //ソースファイル名 ※C 言語ヘッダーファイル出力用。
    "declFileName": "charaDataDecl.cpp", //ソースファイル名 ※C 言語ソースファイル（バージョン整合用
    // 構造定義）出力用。
    "isUsePragmaOnce": true, //pragmga once 使用指定 ※C 言語ヘッダーファイル出力用。
    "headerIncludeFiles": [ "types.h" ], //インクルードファイル ※C 言語ヘッダーファイルに適用。
    // 複数指定可。
    "declIncludeFiles": [ "gameDataDecl.h" ], //インクルードファイル ※C 言語ソースファイル（バージョン整合用
    // 構造定義）に適用。
    // 複数指定可。
    "headerNamespace": [ "charaDataDef" ], //ネームスペース ※C 言語ヘッダーファイルに適用。
    // ネストする場合は配列で複数指定。
    "declNamespace": [ "charaDataDecl" ], //ネームスペース ※C 言語ソースファイル（バージョン整合用
    // 構造定義）に適用。
    // ネストする場合は配列で複数指定。
    "funcNameForGetDecl": "getGameDataDecl", //バージョン整合用構造定義情報の取得関数名
    // ※C 言語ソースファイル（バージョン整合用
    // 構造定義）に適用。

    "struct": //構造体
    {
        "name": "T_CHARA", //構造体名
        "comment": "キャラ構造体", //コメント ※C 言語ソース用。

        "isMakeSource": true, //構造体の定義を C 言語ヘッダーに出力するか？
        // ※規定値は true。汎用構造体などの定義済みの構造体を用いる場合は
        // false を指定する。

        "primaryKey": "id", //主キー項目 ※メンバー名で指定。主キーの昇順にデータが並べ替えされる。
        // 重複検出でエラー。ネストした構造体のメンバーは指定不可。
        "secondaryKey": "name", //副キー項目 ※メンバー名で指定。検索用のインデックステーブルが作成される。
        // 重複検出でエラー。ネストした構造体のメンバーは指定不可。

        "indexes": [ "kana" ], //インデックス項目 ※キーで指定。並べ替え用のインデックステーブル作成用。
        // 複数キー指定可。複合キー指定不可。重複検出なし。
        // ネストした構造体のメンバーは指定不可。

        "members": //メンバー ※デフォルトでは、定義順がデータの並び順になるので注意。
        [
            //メンバー定義：基本形
            {
                "name": "id", //メンバー名
                "comment": "識別 ID", //コメント ※C 言語ソース用。
                "key": "id", //対象キー ※JSON データ上のキー。
                // 省略時は "name" と同じとみなす。
                "type": "crc", //データ型 ※s8, u8, s16, u16, s32, u32, s64, u64, f31, f64, ptr, struct,
                // str, expr, crc, crcs を指定可。
                // crc, crcs は文字列が 32bit 整数に変換される。
                // expr は文字列が T_EXPR 型の計算式データに変換
```

```

//          される。
//          str と expr はポインターに変換され、バイナリ
//          データにはオフセット値として記録される。
"typeName" : "CRC32" //データ型名 ※C 言語ソース用
//          特別に指定したい場合だけ指定。
//          通常は "type" に対応した型から自動判定される。
},
//メンバー定義：文字列型の場合
{
    "name" : "name", //メンバー名
    "comment" : "名前", //コメント
    "type" : "str" //データ型 ※文字列のデータ型は、C 言語ソース上では
//          const char* などのポインター型に置き換わる。
//          文字列データはバイナリデータの後にまとめられ、
//          ポインターはその位置を指す。
//          バイナリデータ上では、文字列データのオフセット値が
//          記録される。
},
//メンバー定義：計算式型の場合
{
    "name" : "condition", //メンバー名
    "comment" : "有効化条件式", //コメント
    "type" : "expr" //データ型 ※計算式のデータ型は、C 言語ソース上では
//          const T_EXPR* 型に置き換わる。
//          実際のデータはバイナリデータの後にまとめられ、
//          ポインターはその位置を指す。
//          バイナリデータ上では、計算式データのオフセット値が
//          記録される。
//          ※JSON データに記述された計算式（文字列）を解析して、
//          計算式データ（バイナリ）に変換して記録する。
//          この解析の際にエラー判定も行う。
//          ※計算式内で使用される関数は、予め用意された組み込み
//          関数（crc や pow など）のほか、実機側の処理で
//          計算式用に用意された拡張関数（getChapter などの
//          ゲーム依存の関数）を指定できる。
//          正しい名前とパラメータで拡張関数を使用しているか
//          どうかは、変換ツール実行時に渡される
//          「拡張関数定義リスト」に基づいて判定する。
},
//メンバー定義：値の場合 ※エラー判定のサンプル
{
    "name" : "power", //メンバー名
    "comment" : "力", //コメント
    "key" : "param.power", //対象キー ※JSON データ上のネストしたデータは
//          「.」で区切って指定。
    "type" : "i8", //データ型
    "default" : 1, //省略時の規定値 ※JSON データ上で記述されなかった場合の規定値。
    "isRequired" : false, //入力必須項目？ ※JSON データ上で記述が必須か？
//          ※エラー判定用、省略時はエラー判定なし
//          （false 指定と同じ）。ただし、
//          primayKey, secondaryKey, indexes に
//          指定された項目は必然的に入力必須となる。
    "min" : 0, //最小値 ※エラー判定用、省略時はエラー判定なし。
    "max" : 100 //最大値 ※エラー判定用、省略時はエラー判定なし。
//          ※データ型に応じた最小値～最大値の範囲チェックは
//          デフォルトで行われる。例えば、i8 なら -128～127 の
//          範囲外の値が指定されたらエラー。
},
//メンバー定義：固定長配列の場合
{
    "name" : "tol", //メンバー名
    "comment" : "耐性", //コメント
    "key" : "param.tol", //対象キー
    "type" : "f32", //データ型
    "isArray" : true, //配列か？ ※規定値は false。

```



```

// true 指定されたデータが、データ定義 JSON 上で
// 配列として定義されていなければエラー。その逆も同様。
"arraySize" : [ 10 ] //配列の要素数 ※配列の次元数分の要素数を指定。
// 二次元配列なら [ 5, 10 ] のように記述する。
// JSON データ上と要素数が一致しない場合はエラー。
},
//メンバー定義：不定長配列の場合
{
    "name" : "abilities", //メンバー名
    "comment" : "アビリティ", //コメント
    "key" : "abilities", //対象キー
    "type" : "u32", //データ型
    "isVariableArray" : true, //不定長配列の指定 ※isArray の指定を省略可。
    "arraySizeName" : "abilitiesNum",
    //配列の要素数を記録するメンバー名
    "arraySizeType" : "i8" //配列の要素数を記録するメンバーのデータ型
    // ※省略時は i32 とみなす
    //※不定長配列の場合、構造体にはポインターと要素数の二つのメンバーが定義される。
    // 通常、要素数のメンバーが先に、続いてポインターが並ぶ。
    // (例) i8 abilityNum;
    // u32* abilities;
    //※実際のデータはバイナリデータの後部にまとめられ、ポインターはその位置を指す。
    // バイナリデータ上では、データのオフセット値が記録される。
},
//メンバー：ネストした構造体の場合
{
    "name" : "param", //メンバー名
    "comment" : "パラメータ", //コメント
    "key" : "param", //対象キー
    "type" : "struct", //データ型 ※C 言語の構造体をネストする場合は、
    // "struct" を指定して、データ型名に構造体名を指定。
    "typeName" : "T_PARAM" //データ型名
    //※実際の構造体は、" substructs" で定義する。
},
//メンバー定義：固定値の場合
{
    "name" : "fixed", //メンバー名
    "comment" : "固定値", //コメント
    "key" : null, //対象キー ※対象キーに null を指定することで、JSON データ側に
    // 存在しないメンバーを定義することが可能。
    "type" : "i16", //データ型
    "default" : 1, //省略時の規定値 ※固定値の指定に使用
},
//メンバー定義：無視項目の場合
{
    "key" : "dummy", //対象キー
    "type" : "ignore", //データ型 ※データ型に "ignore" を指定すると、対象キーを無視する。
}
],
"membersOrder" : //メンバーの並び順 ※" members" の定義順と変えたい時だけ記述する。
// これを指定する場合、" members" の全項目を指定しなければエラー。
// "name" および "arraySizeName" を全て列挙する。
// アラインメントを考慮した配置にしたい場合などに使用する。
[
    "id", //ID : crc
    "power", //力 : i8
    "abilitiesNum", //アビリティ (Num) : i8
    "fixed", //固定値 : i16
    "tol", //耐性 : f32[10]
    "name", //名前 : str*
    "condition", //有効化条件 : T_EXPR*
    "abilities", //アビリティ : u32*
    "param" //パラメータ : T_PARAM
]

```

```

    ],
    "substructs": //ネストした構造体 ※「struct」とほぼ同様の構造だが、配列で複数の構造体を定義する。
    [
        {
            "name": "T_PARAM", //構造体名
            "comment": "パラメータ構造体", //コメント

            "isMakeSource": true, //構造体の定義をC言語ヘッダーに出力するか？
            "isInternalStructure": true, //親の構造体の中にこの構造体を定義するか？

            "members": //メンバー
            [
                {
                    "name": "atk", //メンバー名
                    "comment": "攻撃力", //コメント
                    "key": "atk", //対象キー ※JSON データ上のネストしたデータのキーだが、親キーは指定しない。
                    // ※メンバー名と同じなら省略可能。
                    "type": "i16" //データ型
                },
                {
                    "name": "def", //メンバー名
                    "comment": "守備力", //コメント
                    "type": "i16" //データ型
                },
                {
                    "name": "specials", //メンバー名
                    "comment": "特殊能力", //コメント
                    "type": "struct", //データ型 ※さらにネストした構造体も指定可能
                    "typeName": "T_SPECIAL_PARAM", //データ型名
                    "isArray": true, //不定長配列の指定
                    "arraySizeName": "specialsNum" //配列の要素数
                }
            ],
            "membersOrder": //メンバーの並び順
            [
                "atk", //攻撃力:i16
                "def", //防御力:i16
                "specialNum", //特殊能力(Num):s32
                "special" //特殊能力:T_SPECIAL_PARAM*
            ],
        },
        {
            "name": "T_SPECIAL_PARAM",
            "comment": "特殊パラメータ構造体", //コメント

            "isMakeSource": true, //構造体の定義をC言語ヘッダーに出力するか？
            "isInternalStructure": false, //親の構造体の中にこの構造体を定義するか？

            "members": //メンバー
            [
                {
                    "name": "dark", //メンバー名
                    "comment": "闇", //コメント
                    "type": "u32" //データ型
                },
                {
                    "name": "shine", //メンバー名
                    "comment": "光", //コメント
                    "type": "u32" //データ型
                }
            ]
        }
    ]
}

```

```

],
//エラー判定用ルール ※メンバーごとの min, max, isRequired 以外のルールを設定したい場合に用いる。
// ※複数のルールを指定可。
// ※メンバーの値を計算結果などで書き換えたい場合にも利用可能。
"rules" :
[
{
//ルール ※エラーメッセージ判定用の C#処理を記述。一塊の文字列として定義する。
// エラーがある場合はエラーメッセージを return し、問題が無い場合は null を return。
// ※一つのオブジェクト（構造体）が取り込まれる毎に実行され、
// 取り込んだオブジェクトは変数 obj として渡される。
// なお、この時の obj は、str や可変長配列などの情報はポインター化（オフセット化）
// されていないため、そのままメンバーにアクセスできる。crc などの組み込み関数、
// expr による計算式解析もまだ行われていない状態。ルールを一通りパスした後に
// それらの処理を行う。
// ※エラーメッセージ出力時は、データ定義位置の行番号と、主キーの情報もいっしょに出力される。
// ※obj のメンバーに値を代入すれば、そのまま実際の値として反映される。
// ※行が連結されてしまうので、「//」によるコメントの記述は禁止
"rule" :
" ¥
var atk = obj.param.atk; ¥
var def = obj.param.def; ¥
if (atk < def) ¥
{ ¥
return ¥ "ATK(¥) + atk + ¥" )は、DEF(¥) + def + ¥ " ) 以上の値にして下さい。¥" ; ¥
} ¥
obj.tol[2] = obj.tol[0] + obj.tol[1]; /*値を書き換え可能*/ ¥
return null; ¥
"
},
{
//ルール
"rule" :
" ¥
if (obj.tol[0] > 0 && obj.tol[1] > 0) ¥
{ ¥
return ¥ "「耐性」は、どれか一つだけ入力して下さい。¥" ; ¥
} ¥
return null; ¥
"
}
],
}

```

## ▼ データ型定義リスト

データフォーマットで使用するデータ型を定義するための JSON。

基本的な記述仕様は「拡張 JSON」形式に従う。記述内容は、データ型定義のための設定項目が定められている。下記のサンプルでその仕様を示す。赤い字で書かれた項目が、サンプル中で初出の定義項目であり、詳しい説明を併記する。

例：

```

//データ型定義リスト
//※「データ型」の内容を定義する。
//※プリミティブな型のみに対応し、構造体や配列は定義できない。
[
  //u16_ex 型

```

```

{
    "type": "u16_ex", //データ型
    "typeName": "unsigned int", //データ型名 ※C 言語ソース用
    "baseType": "int", //基本データ型 ※int (整数) / float (浮動小数点) / dec (固定小数点)
    // bool (真偽値) / str (文字列) / expr (計算式)
    // ptr (ポインタ) のいずれかで指定する。
    // ※ptr は便宜上存在。任意のデータを扱うことができないが、
    // 構造体にポインタ型のメンバーを含めたい場合に
    // 使用する。
    "isUnsigned": true, //符号無し指定 ※基本データ型が int の場合のみ指定可。
    "size": 2, //データサイズ ※基本データ型が int の場合は 1, 2, 4, 8 のいずれか。
    // float の場合は 2, 4, 8 のいずれか。
    // dec の場合は 2, 4, 8 のいずれか。
    // str/scpr/ptr の場合はサイズ指定不要。ポインタの
    // サイズ (32/64bit) になる。
    "decBits": 8, //固定小数の小数部のビット数 ※基本データ型が dec の場合のみ指定可。
    // 省略時はデータサイズの半分 -1 の
    // ビット数。2 バイトなら 7bit、4 バイト
    // なら 15bit、8 バイトなら 31bit。
    "min": -10000, //最小値 ※オプションで指定可。省略時は基本データ型、符号無し指定、
    // データサイズから自動判定。
    "max": 10000, //最大値 ※(同上)
    "default": 1 //規定値 ※オプションで指定可。省略時は 0。
},
//crc 型
{
    "type": "crc", //データ型
    "typeName": "unsinged int", //データ型名
    "baseType": "int", //基本データ型
    "isUnsigned": true, //符号無し指定
    "size": 4, //データサイズ
    "func": "crc" //組み込み関数 ※バイナリデータに変換する際に適用する
    // 組み込み関数を指定する。
},
//文字列型
{
    "type": "str", //データ型
    "typeName": "const char*", //データ型名
    "baseType": "str", //基本データ型 ※str が指定されたデータは、JSON データをバイナリ
    // データに変換した際、データ後部に実際のデータ
    // (文字列) をまとめ、その参照を扱うようになる。
},
//計算式型
{
    "type": "expr", //データ型
    "typeName": "T_EXPR*", //データ型名
    "baseType": "expr", //基本データ型 ※expr が指定されたデータは、JSON データをバイナリ
    // データに変換した際、データ後部に実際のデータ
    // (T_EXPR 型の計算式データ) をまとめ、その参照を
    // 扱うようになる。
    "func": "expr" //組み込み関数 ※expr () 組み込み関数は、計算式が記述された
    // 文字列を、T_EXPR 型 (不定長のデータ部を含む) の
    // バイナリデータに変換する。
}
]

```

データ型定義リストは、複数の定義ファイルを変換ツール実行時に指定することが可能。  
 もっとも基本的な定義リストには、下記のデータ型が定義される。

- bool ..... 真偽値 : 1 バイト : true/false
- i8 ..... 符号付き 8bit 整数 : 1 バイト : -128 ~ 127
- u8 ..... 符号無し 8bit 整数 : 1 バイト : 0 ~ 255

- i16 ..... 符号付き 16bit 整数 : 2 バイト : -32,768 ~ 32,767
- u16 ..... 符号なし 16bit 整数 : 2 バイト : 0 ~ 65,535
- i32 ..... 符号付き 32bit 整数 : 4 バイト : -2,147,483,648 ~ 2,147,483,647
- u32 ..... 符号なし 32bit 整数 : 4 バイト : 0 ~ 4,294,967,295
- i64 ..... 符号付き 64bit 整数 : 8 バイト : -9,223,372,036,854,775,808  
~ 9,223,372,036,854,775,807
- u64 ..... 符号なし 64bit 整数 : 8 バイト : 0 ~ 18,446,744,073,709,551,615
- f32 ..... 32bit 浮動小数点 : 4 バイト : 1 + 8 + 23 bits (符号+指数+仮数)
- f64 ..... 64bit 浮動小数点 : 8 バイト : 1 + 11 + 52 bits (符号+指数+仮数)
- str ..... 文字列 : ポインター (オフセット値) + 実データ (char[]型)
- expr .... 計算式 : ポインター (オフセット値) + 実データ (T\_EXPR 型+データ長)
- crc ..... 符号なし 32bit 整数 : 4 バイト : 32bit CRC 値
- crcs ..... 符号なし 32bit 整数 : 4 バイト : 32bit CRC 値 (文字列を小文字化して算出)

#### ▼ 計算式用拡張関数定義リスト

「データ定義 JSON」内の「計算式」で使用する拡張関数を定義するための JSON。

なお、これはあくまでも「計算式」の中で、ランタイム時に実行される関数を指定するためのものであり、バイナリデータ変換時には処理されないことに注意。(組み込み関数の場合は、パラメータに動的な要素がない限りは、データ変換時にも処理される。)

基本的な記述仕様は「拡張 JSON」形式に従う。記述内容は、拡張関数定義のための設定項目が定められている。下記のサンプルでその仕様を示す。赤い字で書かれた項目が、サンプル中で初出の定義項目であり、詳しい説明を併記する。

例 :

```
//拡張関数定義リスト
//※「拡張関数」の名前とパラメータを定義する。
[
  //フラグを更新 : bool setFlag( "フラグ名", bool)
  {
    "func" : "getChapter",           //関数名
    "args" : [ "str", "bool" ],      //パラメータ ※データ型を列挙
    "return" : "bool",               //戻り値 ※データ型を一つ指定
  },
  //現在の章を取得 : u32 getChapter()
  {
    "func" : "getChapter",           //関数名
    "args" : [ ],                   //パラメータ ※パラメータがない場合は空の配列を指定するか、
                                     // args 自体を指定しない。
    "return" : "u32",               //戻り値
  },
  //デバッグプリント : bool debugPrint()
  {
    "func" : "debugPrint",           //関数名
    "args" : [ "str", "..."],        //パラメータ ※不定長パラメータには "..." (ピリオド三つ) を指定する。
  }
]
```

```

        "return" : "bool" ,           //戻り値
    }
]
```

## ▼ チェック用 JSON

バイナリデータ出力が成功した時にだけ一緒に出力される。

チェック用 JSON は拡張仕様を排除した JSON 仕様のフォーマットのため、データの二次利用にも活用できる。

バイナリデータの構造に合わせた構造だが、文字列や計算式、不定長配列などのポインター（オフセット）要素は展開されず、そのまま本来の位置に記述される。crc などの組み込み関数は計算結果が出力され、計算式のようなバイナリデータは BASE64 エンコードされた文字列が出力される。出力されるデータの並び順は、指定された「主キー」に基づいて並べ替えされた状態となる。「副キー」と「インデックス」に指定されたインデックステーブルも別ファイルに出力され、内容を確認することができる。以下にそれらのサンプルを示す。

例：データ定義 JSON ⇒ チェック用 JSON&インデックスリスト（主キー = "id"、副キー = "name"、インデックス = "kana" を設定し、かつ、"kana" は実機用バイナリデータに出力されないものとする）

```

//キャラ定義
#include "header.jsonh"

[
    //キャラ：山田
    {
        "id" : "c0010" ,           //ID：主キー
        "name" : "山田" ,         //名前：副キー
        "kana" : "やまだ" ,       //読み：インデックス
        "condition" : "getChapter() >= 20" ,
        "param" :
        {
            "atk" : 10,
            "def" : 20
        },
        "abilities" : [ "aaa" , "bbb" ]
    },
    //キャラ：田中
    {
        "id" : "c0020" ,           //ID：主キー
        "name" : "田中" ,         //名前：副キー
        "kana" : "たなか" ,       //読み：インデックス
        "param" :
        {
            "atk" : 11,
            "def" : 21
        },
    },
    //キャラ：佐藤
    {
        "id" : "c0030" ,           //ID：主キー
        "name" : "佐藤" ,         //名前：副キー
```

```

    "kana": "さとう", //読み:インデックス
    "param":
    {
        "atk": 12,
        "def": 22
    },
    "abilities": [ "xxx" ]
}
]

```

↓ (変換後)

チェック用 JSON：※ "id" が CRC 値に変換され、"condition" のバイナリコードが BASE64 に変換され、リストの並びが主キー ("id") の順に変わり、kana が消滅し、abilities などの入力が省略された項目が網羅されている。

```

[
  {
    "id": 123306860,
    "name": "田中",
    "condition": null,
    "param":
    {
        "atk": 11,
        "def": 21
    },
    "abilities": [ ]
  },
  {
    "id": 507687469,
    "name": "佐藤",
    "condition": null,
    "param":
    {
        "atk": 12,
        "def": 22
    },
    "abilities": [ "xxx" ]
  },
  {
    "id": 745853103,
    "sortOrder": 1,
    "condition": "Z2V0Q2hhcHRlcigpID49IDIweA==",
    "param":
    {
        "atk": 10,
        "def": 20
    },
    "abilities": [ "aaa", "bbb" ]
  }
]

```

副キー検索用インデックステーブル：※ "name" の CRC 順。"key" は "name" の CRC 値を、"index" は出力データ (チェック用 JSON) の位置を示す。

```

[
  { "key": 230889854, "index": 2 },
  { "key": 558509826, "index": 1 },
  { "key": 2850539082, "index": 0 }
]

```

```
]
```

並べ替え用インデックステーブル： ※ "kana" 順（さとう→たなか→やまだ）。数値は出力データ（チェック用 JSON）の位置を示す。

```
[ 1, 0, 2 ]
```

## ▼ C 言語ソース

構造体が定義されたヘッダーファイルと、バージョン整合用の構造定義情報が定義されたソースファイル。

データ変換ツールのオプションにより出力可能。その際、「データ定義 JSON」は必要なく、「フォーマット定義 JSON」に基づいて作成される。

「バージョン整合用構造定義」は、T\_GAME\_DATA\_DECL 型のデータで定義される static なデータ。この情報を取得するための関数 `getGameDataDecl()` も合わせて自動生成される。

以下に自動生成されたファイルのサンプルを示す。T\_GAME\_DATA\_DECL 型の扱いについてもサンプルで示す。なお、この内容は、前述の「フォーマット定義 JSON」のサンプルに基づく。

例：ヘッダーファイル：charaData.h

```
#pragma once
#ifndef __CHARA_DATA_H__
#define __CHARA_DATA_H__

//ATTENTION!
//Do not modify this file manually.

#include "types.h"

namespace charaDataDef
{
    //キャラパラメータ構造定義
    //Name: CharaData
    //Version: 1.0
    //Update: 204.1.10 12:34:56

    //特殊パラメータ構造体
    struct T_SPECIAL_PARAM
    {
        unsigned int dark; //闇
        unsigned int shine; //光
    };

    //キャラ構造体
    struct T_CHARA
    {
        //パラメータ構造体
        struct T_PARAM
        {
            short atk; //攻撃力
            short def; //守備力
```



```

        int    specialNum; //特殊能力 (Num)
        T_SPECIAL_PARAM* special; //特殊能力
    };

    CRC    id; //識別 ID
    char    power; //力
    char    abilitiesNum; //アビリティ (Num)
    short    fixed; //固定値
    float    tol[10]; //耐性
    const char* name; //名前
    const T_EXPR* condition; //有効化条件
    unsigned int* abilities; //アビリティ
    T_PARAM    param; //パラメータ
};

//namespace charaDataDef

#endif//__CHARA_DATA_H__

```

例：バージョン整合用構造定義ファイル：charaDataDecl.cpp

```

//ATTENTION!
//Do not modify this file manually.

#include "gameDataDecl.h"
/*
※このインクルード内で下記の構造体が定義されている。
本来このコメントの部分はファイル出力されないが、
このサンプルでは内容を分かり易くするために記述している。

//ゲームデータ定義
struct T_GAME_DATA_DECL
{
    //ゲームデータ構造体定義
    struct STRUCT
    {
        //ゲームデータ構造体メンバー定義
        struct MEMBER
        {
            unsigned int memberNameCrc; //メンバー名 CRC
            unsigned int dataNameCrc; //構造体名 CRC ※型種別 = struct の場合に指定。
            unsigned char baseType:3; //型種別 ※0 = int, 1 = float, 2 = dec, 3 = bool, 4 = str,
            // 5 = expr, 6 = ptr, 7 = struct。
            unsigned char isUnsigned:1; //符号無し型か？ ※型種別 = int の場合のみ指定可。
            unsigned char size:4; //型のサイズ ※int の場合は、1,2,4,8 のいずれか。
            // float の場合は、2,4,8 のいずれか。
            // dec の場合は 2,4,8 のいずれか。
            // bool の場合は 1。
            // str / expr / ptr /struct の場合は 0。
            unsigned char padding[3]; //パディング ※0xdb で埋める
            unsigned int isVariableArray:1; //不定長配列か？ ※true で不定長配列。この時、メンバーの実際の
            // データ型はポインターとなる。
            unsigned int arraySize:16; //配列要素数 ※配列でない時や不定長の時は 0。
        };

        unsigned int structNameCrc; //構造体名 CRC
        unsigned short membersNum; //構造体メンバー数
        MEMBER* members; //構造体メンバー定義の参照
    };

    unsigned int formatNameCrc; //データフォーマット名 CRC
    unsigned short majorVer; //データフォーマットメジャーバージョン
    unsigned short minorVer; //データフォーマットマイナーバージョン
    unsigned short structsNum; //構造体数
    STRUCT* structs; //構造体定義の参照
}

```

```

};
*/

namespace charaDataDecl
{

//キャラパラメータ構造定義
//Name: CharaData
//Version: 1.0
//Update: 204.1.10 12:34:56

//データ構造定義

//構造体のメンバー定義
static const int s_structMembersNum = 9 + 4 + 2; //全構造体の全メンバー数
static T_GAME_DATA_DECL::STRUCT::MEMBER s_structMembers[s_structMembersNum] =
{
    //T_CHARA 構造体
    { 0xe66c3671, 0x00000000, 0, true, 4, false, 0 }, //識別 ID:" id" :crc
    { 0xab8a01a0, 0x00000000, 0, false, 1, false, 0 }, //力:" power" :i8
    { 0x9c504212, 0x00000000, 0, false, 1, false, 0 }, //アビリティ(Num):" abiriliesNum" :i8
    { 0x9ec9ce32, 0x00000000, 0, false, 2, false, 0 }, //固定値:" fixed" :i16
    { 0x5e237e06, 0x00000000, 4, false, 0, false, 0 }, //名前:" name" :str
    { 0xbdd68843, 0x00000000, 5, false, 0, false, 0 }, //有効化条件:" condition" :expr
    { 0xad84385, 0x00000000, 1, false, 1, false, 10 }, //耐性:" tol" :f32
    { 0xb838da4, 0x00000000, 0, true, 4, true, 0 }, //アビリティ:" abilities" :u32*
    { 0xa4fa7c89, 0x22a2e1dc, 7, false, 0, false, 0 }, //パラメータ:" param" :T_PARAM
    //T_PARAM 構造体
    { 0x27677c27, 0x00000000, 0, false, 2, false, 0 }, //攻撃力:" atk" :i16
    { 0x0cc4e161, 0x00000000, 0, false, 2, false, 0 }, //防御力:" def" :i16
    { 0xcd1415d7, 0x00000000, 0, false, 4, false, 0 }, //特殊能力(Num):" specialsNum" :i32
    { 0x4c6b3fe3, 0x84f96e44, 7, false, 0, true, 0 }, //特殊能力:" specials" :T_SPECIAL_PARAM*
    //T_SPECIAL_PARAM 構造体
    { 0x1b7cbdfb, 0x00000000, 0, true, 4, false, 0 }, //闇:" dark" :u32
    { 0x076291bf, 0x00000000, 0, true, 4, false, 0 }, //光:" shine" :u32
};

//構造体の定義
static const int s_structsNum = 3; //全構造体数
static T_GAME_DATA_DECL::STRUCT s_structs[s_structsNum] =
{
    { 0x0ed35394, 9, &s_structMembers[0] }, //T_CHARA
    { 0x22a2e1dc, 4, &s_structMembers[9] }, //T_PARAM
    { 0x84f96e44, 2, &s_structMembers[13] }, //T_SPECIAL_PARAM
};

//フォーマット情報定義 (バージョンを含む)
static T_GAME_DATA_DECL s_decl =
{
    {
        0xe1e4a645, //フォーマット名 CRC : CharaData
        1, //フォーマットメジャーバージョン
        0, //フォーマットマイナーバージョン
        s_structsNum, //構造体数
        &s_structs[0] //構造体定義の参照
    }
};

//データ構造定義取得関数
const T_GAME_DATA_DECL* getGameDataDecl() {return &s_decl;}

} //namespace charaDataDecl

```

「バージョン整合用構造定義」データは、以下の二つの目的で使用される。

① バージョン整合判定：

バイナリデータ取り込み時に、バイナリデータのデータ構造と実機側のデータ構造が一致しているかどうかの判定に用いる。バイナリデータのヘッダー部にも同様の「バージョン整合用構造定義」データが記録されており、両者を比較する。両者が一致する場合はバイナリデータをまるごとコピーできるが、不一致だった場合は、両者の情報を比較しながら、名前が一致する項目のデータを一つずつコピーする。

② ポインター補正：

バイナリデータ取り込み後、文字列、計算式、不定長配列のデータは、データのオフセットが記録されているため、ポインターに修正する。

## ▼ バイナリデータ

「データ定義 JSON」が変換された実機向けのデータ。

基本構造は下記の通り。

項目	型	サイズ	内容
<b>01: ヘッダー部</b>			
マジックナンバー	i8[4]	4	'd', 'B', 0x00, 0xdb
エンディアン	u8	1	0 ... リトルエンディアン、1 ... ビッグエンディアン
ポインターサイズ	u8	1	0 ... 32bit、1 ... 64bit
エンコーディング	u8	1	0 ... UTF-8、1 ... シフトJIS
アライメント(構造体パッキング)	u8	1	0 ... 1、1 ... 2、2 ... 4、3 ... 8、4 ... 16(バイト)
総サイズ	u32	4	ヘッダー部も含む全体サイズ(バイト) デバッグデータ部は含まない
総サイズ(デバッグデータ含む)	u32	4	ヘッダー部とデバッグデータ部も含む全体サイズ(バイト)
構造定義部オフセット	u32	4	ヘッダー部の先頭からのオフセット値(バイト)
データ部オフセット	u32	4	(同上)
検索インデックス部オフセット	u32	4	(同上)
並べ替えインデックス部オフセット	u32	4	(同上)
計算式データ部オフセット	u32	4	(同上)
文字列部オフセット	u32	4	(同上)
終端部オフセット	u32	4	(同上)
デバッグデータ部オフセット (64バイトパディング)	u32	4	(同上)
	u8[16]	16	0xdb で埋める
		(64)	
<b>02: 構造定義部</b>			
フォーマット名CRC	u32	4	
フォーマットメジャーバージョン	u16	2	
フォーマットマイナーバージョン	u16	2	
構造体数	u16	2	
全構造体の全メンバー数	u16	2	
構造体定義情報	T_GAME_DATA_DECL::STRUCT[構造体数]	?	
構造体メンバー定義情報	T_GAME_DATA_DECL::STRUCT::MEMBER[構造体数]	?	
(16バイトアライメントパディング)	u8[]	?	0xdb で埋める
		(12 + ?)	

03: データ部			
基本データ情報		T_GAME_DATA_TABLE_INFO	
データ件数	u32	4	
データ位置オフセット	u32	4	
レコードサイズ	u32	4	
可変長データ種数	u32	4	
可変長データ情報		T_GAME_DATA_TABLE_INFO[可変長データ種数]	
データ型識別情報			※データ型を識別するための情報(構造体以外の可変長データも扱える)
構造体名CRC	u32	4	※型種別 = struct 以外の場合は 0x00000000
型種別	u8	(3 bit)	※0 = int, 1 = float, 2 = dec, 3 = bool, 4 = str, 5 = expr, 6 = ptr, 7 = struct
符号無し型か?	u8	(1 bit)	※型種別 = int の場合のみ指定可
型のサイズ	u8	(4 bit)	※型種別 = int の場合 ... 1, 2, 4, 8, float の場合 ... 2, 4, 8, dec の場合 ... 2, 4, 6, bool の場合 ... 1, str / expr / struct の場合 ... 0
(パディング)	u8[3]	3	0xdb で埋める
データ件数	u32	4	
データ位置オフセット	u32	4	
レコードサイズ	u32	4	
※可変長データ種数分続く...			
(16バイトアラインメントパディング)	u8[]	?	0xdb で埋める
基本データ		任意の構造体[基本データ情報::データ件数]	
?	?	?	
(16バイトアラインメントパディング)	u8[]	?	0xdb で埋める
可変長データ[n]		任意の型/構造体[可変長データ情報[n]::データ件数]	
?	?	?	
※可変長データ種数分続く...			
(16バイトアラインメントパディング)	u8[]	?	0xdb で埋める
		(16 + ?)	

04: 検索インデックス部			
検索インデックス種数		u32	4
検索インデックス情報[n]		T_GAME_DATA_SEARCHING_INDEX[n][基本データ情報::データ件数]	
検索キー	u32	4	CRC値
基本データインデックス	u32	4	
※検索インデックス種数分続く...			
(16バイトアラインメントパディング)	u8[]	?	0xdb で埋める
		(4 + ?)	

05: 並べ替えインデックス部			
並べ替えインデックス種数		u32	4
並べ替えインデックス情報[n]		T_GAME_DATA_SORTING_INDEX[n][基本データ情報::データ件数]	
基本データインデックス	u32	4	
※並べ替えインデックス種数分続く...			
(16バイトアラインメントパディング)	u8[]	?	0xdb で埋める
		(4 + ?)	

06: 計算式データ部			
計算式データ数		u32	4
計算式参照データ[n]		T_STRING_REF[n]	
文字列CRC値	u32	4	計算式文字列のCRC値 ※検索/重複定義検出用
実データオフセット	u32	4	計算式の実データ位置のオフセット ※計算式データ部のトップからのオフセット
※計算式データ数分続く...			データはCRC値順に並び、バイナリサーチ可能な状態
(16バイトアラインメントパディング)	u8[]	?	0xdb で埋める
計算式データ[n]		T_EXPR[n]	
(T_EXPR型の構造については別途記述)		?	※T_EXPRは可変長データ
※計算式データ数分続く...			
(16バイトアラインメントパディング)	u8[]	?	0xdb で埋める
		(4 + ?)	

07: 文字列データ部			
文字列データ数		u32	4
文字列参照データ[n]		T_STRING_REF[n]	
文字列CRC値	u32	4	文字列のCRC値 ※検索/重複定義検出用
実データオフセット	u32	4	文字列の実データ位置のオフセット ※文字列データ部のトップからのオフセット
※文字列データ数分続く...			データはCRC値順に並び、バイナリサーチ可能な状態
(16バイトアラインメントパディング)	u8[]	?	0xdb で埋める
文字列データ[n]		char[n][?]	
※文字列データ数分続く...		?	※文字列は可変長データ
(16バイトアラインメントパディング)	u8[]	?	0xdb で埋める
		(4 + ?)	

<b>08: 終端部</b>			
マジックナンバー	i8[4]	4	0xdb, 'D', 'b', 0x00
(16バイトアラインメントパディング)	u8[12]	12	0xdb で埋める
		(16)	

<b>09: デバッグデータ部</b>		※「識別ID」や「計算式」のような変換して扱うデータの元の文字列を記録しておき、デバッグ時に表示可能にする。	
文字列データ数	u32	4	
文字列参照データ[n]	T_STRING_REF[n]		
文字列CRC値	u32	4	文字列のCRC値 ※検索／重複定義検出用
実データオフセット	u32	4	文字列の実データ位置のオフセット ※デバッグデータ部のトップからのオフセット
※文字列データ数分続く…			データはCRC値順に並び、バイナリサーチ可能な状態
(16バイトアラインメントパディング)	u8[]	?	0xdb で埋める
文字列データ[n]	char[n][]	?	※文字列は可変長データ
※文字列データ数分続く…			
(16バイトアラインメントパディング)	u8[]	?	0xdb で埋める
		(4 + ?)	

<b>10: 最終端部</b>			
(16バイトパディング)	u8[16]	16	0xdb で埋める
		(16)	

「データ部」内のデータには、「可変長データ」「計算式データ」「文字列データ」が含まれる。それらは、それぞれ「可変長データ部」「計算式データ部」「文字列データ部」を指すポインター型で定義されているが、「バイナリデータ」内では、オフセット値が記録されているため、それをポインターに変換する必要がある。

それぞれのオフセット値は、いずれもバイナリデータの先頭からのオフセット値になっているため、単純に先頭ポインターを加算するだけで変換できる。オフセット値が 0 のデータは nullptr。

どのデータ項目をポインターに変換する必要があるかは、ソースコードに組み込まれている「バージョン整合用構造定義」に基づく。

また、データ取り込み時には、この「バージョン整合用構造定義」とバイナリデータ内の「構造定義部」の内容を比較し、そのままコピー可能かどうか判定する。構造が不一致の場合、全体サイズを計算してメモリを確保し、必要な情報を一つずつコピーする。この時、ヘッダー部の各種オフセット値は書き換えられるが、「構造定義部」はそのままとなる。

データ取り込み時に、「デバッグデータ部」を取り込むかどうか指定することが可能。取り込まなかった場合は、ヘッダー部内の「デバッグデータ部のオフセット」を 0 がになり、デバッグテキストの参照ができなくなる。

## ■ 値の演算と組み込み関数

拡張 JSON の仕様により、「データ定義 JSON」の「値」には、演算と組み込み関数の使用が可能。データ変換時に演算が行われ、算出された値がバイナリデータに出力される。

使用可能な演算子と組み込み関数は、後述する「計算式」とほぼ同じ。仕様も「計算式」

の説明に記述。

演算と組み込み関数の扱いについて、「計算式」との違いは下記の通り。

- ・ 拡張 JSON 上では、「+」演算子による文字列の連結が可能。計算式では不可。
- ・ 計算式では、`expr()` 関数が使用不可。`expr()` 関数はデータ定義 JSON 専用。

#### ▼ 計算式解析関数：`expr()`

- ・ 関数名：`expr`
- ・ パラメータ：計算式文字列
- ・ 戻り値：`T_EXPR` 型（計算式型）
- ・ 用例：`expr( "getChapter() >= 20 && getFlag(¥" flag1¥" ) == On" )`

計算式の文字列を、ランタイム時に実行可能なバイナリの計算式データに変換して返す。  
計算式に関する仕様は後述。

## ■ 計算式

「データ定義 JSON」内で、ランタイム時に実行するための計算式を使用できる。

例えば、会話テキストの選択条件として「○章以降なら」「XXX フラグが立っていたなら」「△章時にフラグ A=OFF でフラグ B=ON なら」といった条件を与えたい場合などに使用する。汎用的なデータであるため、Excel シートの会話テキストに並べて条件判定式として記述するようなことが可能。

例：

キャラ	条件判定式	会話テキスト	会話後処理
太郎	<code>getChapter() == 30 &amp;&amp; getFlag("flagA") == OFF &amp;&amp; getFlag("flagB") == ON</code>	これは… 本当に大変なことになった	
	<code>getFlag("XXX") == ON &amp;&amp; getFlag("YYY") == OFF</code>	なんと、あなたは！	<code>runEvent("ev0020"); setFlag("YYY", ON);</code>
	<code>getChapter() &gt;= 20</code>	今日もいそがしい	
		こんにちは	

複雑な判定条件が必要になっても、データシートを拡張せずに対応できる点も利点。拡張関数さえ登録すれば、どのようなタイトルでも、タイトル依存の情報を扱うことができる。

章判定やフラグ判定などの基本的な要素は専用の入力欄を設けて、実機用のデータにする際に計算式化するような方法も有効。この場合、判定要素の追加が必要になって Excel シートを拡張することがあっても、実機用のデータはそのまま（一つの計算式があるだけ）にして、少しでも手間を軽減することができる。

以降、計算式データに関する仕様を示す。

## ▼ 計算式の構文

計算式の構文は、下記の要素で成り立つ。

- ・ 値（整数、小数、真偽値）
  - 整数には、10 進表記、8 進表記（0 + 0~7）、16 進表記（0x + 0~f）、2 進表記（0b + 0~1）が使用可能。
  - 真偽値は、真値に true, yes, on のいずれか、偽値に false, no, off のいずれかを記述可能。大文字／小文字は問わない。
- ・ 文字列
  - 文字列はダブルクォーテーションもしくはシングルクォーテーションのどちらかで囲む。
  - 文字列内にはエスケープ文字を使用可能。使用可能なエスケープ文字は下記のとおり。
    - ¥"（ダブルクォーテーション）
    - ¥'（シングルクォーテーション）
    - ¥n（改行）／ ¥r（改行）／ ¥r¥n（改行） ※どれも単一の改行として扱われる
    - ¥¥（¥マーク）
- ・ 単項演算子（+, -, ~, !）※C 言語準拠
- ・ 四則演算子（+, -, \*, /, %）※C 言語準拠
- ・ ビット演算子（&, |, ^, ~, <<, >>, >>>）※C 言語／Java 準拠
- ・ 判定式（==, !=, >, <, >=, <=）※C 言語準拠
- ・ 論理演算子（&&, ||）※C 言語準拠
- ・ 三項演算子（x ? y : z）※C 言語準拠
- ・ かっこ（丸かっこのみ）※C 言語準拠
- ・ 組み込み関数 ※あらかじめ用意されている関数。
- ・ 拡張関数 ※タイトル固有の関数。
- ・ 「,」（コンマ） ※関数のパラメータの区切り用。
- ・ 「;」（セミコロン） ※演算と無関係に複数の関数を呼び出したい時用。

なお、下記の要素には対応しない。

- ・ 変数
- ・ 代入および代入を伴う演算（=, +=, -=, \*=, /=, ++, --）

## ▼ 応演算子

《ゲームデータ内計算式対応演算子》 ※C言語の仕様がベース

優先順位	ID	演算子	用法	名称	備考
1		[]	a[b]	添字演算子	非対応
		()	a(b)	関数呼出し演算子	非対応
		.	a.b	ドット演算子	非対応
		->	a->b	ポインタ演算子	非対応
		++	a++	後置増分演算子	非対応
2		--	a--	後置減分演算子	非対応
		++	++a	前置増分演算子	非対応
		--	--a	前置減分演算子	非対応
		&	&a	単項&演算子、アドレス演算子	非対応
		*	*a	単項*演算子、間接演算子	非対応
	1	+	+a	単項+演算子	※単項演算子の扱いに注意 左値が演算子で右値が値なら単項演算子
	2	-	-a	単項-演算子	
	3	~	~a	補数演算子	
	4	!	!a	論理否定演算子	
3		sizeof	sizeof a	sizeof演算子	非対応
		()	(a)b	キャスト演算子	非対応
4	5	*	a * b	2項*演算子、乗算演算子	※左値が0なら右値を評価しない(普通に評価する)
	6	/	a / b	除算演算子	※左値が0なら右値を評価しない(普通に評価する)
	7	%	a % b	剰余演算子	右値が0でもNanにせず0とする
5	8	+	a + b	2項+演算子、加算演算子	
	9	-	a - b	2項-演算子、減算演算子	
6	10	<<	a << b	左シフト演算子	※左値が0なら右値を評価しない(普通に評価する)
	11	>>	a >> b	右シフト演算子(符号拡張あり)	※左値が0もしくは0xffffffffなら右値を評価しない(普通に評価する)
	12	>>>	a >>> b	右シフト演算子(符号拡張なし)	※Java仕様、左値が0なら右値を評価しない
7	13	<	a < b	<演算子	※bool値を返す
	14	<=	a <= b	<=演算子	
	15	>	a > b	>演算子	
	16	>=	a >= b	>=演算子	
8	17	==	a == b	等価演算子	※bool値を返す
	18	!=	a != b	非等価演算子	
9	19	&	a & b	ビット単位のAND演算子	※左値が0なら右値を評価しない(普通に評価する)
10	20	^	a ^ b	ビット単位の排他OR演算子	
11	21		a   b	ビット単位のOR演算子	※左値が0xffffffffなら右値を評価しない(普通に評価する)
12	22	&&	a && b	論理AND演算子	※bool値を返す/左値が0なら右値を評価しない
13	23		a    b	論理OR演算子	※bool値を返す/左値が0以外なら右値を評価しない
14	23,24	?:	a ? b : c	条件演算子	※条件に当てはまらなかった方の値は評価しない
15		=	a = b	単純代入演算子	非対応
		+=	a += b	加算代入演算子	非対応
		-=	a -= b	減算代入演算子	非対応
		*=	a *= b	乗算代入演算子	非対応
		/=	a /= b	除算代入演算子	非対応
		%=	a %= b	剰余代入演算子	非対応
		<<=	a <<= b	左シフト代入演算子	非対応
		>>=	a >>= b	右シフト代入演算子	非対応
		&=	a &= b	ビット単位のAND代入演算子	非対応
		^=	a ^= b	ビット単位の排他OR代入演算子	非対応
16		=	a  = b	ビット単位のOR代入演算子	非対応
	25	,	a , b	コンマ演算子	※「,」と「;」は、同じ扱いとする
	26	;	a ; b	セミコロン(ターミネータ)	列挙された値全部が返され、関数の引数などになる
	0			ダミー	

※べき乗や平方根の演算子はない。  
pow(x, y) 関数やsqrt(x) 関数を使用する。

※1の「評価しない」とは、実機上の処理で、実際の演算や関数呼び出しを実行しないことを意味する。

文字列の演算には対応しない。例えば、「+」演算子で文字列連結といったことはできない。  
基本的に、文字列操作のような、メモリ操作を要する処理は計算式内では実行できない。

リスト中の一部の説明「...なら右値を評価しない」とあるのは、例えば、「getChapter0>=



30 && getFlag("flag") == ON」という計算式があった場合、論理演算子「&&」の左値「getChapter() >= 30」の結果が偽だった時点で、「&&」の結果が偽と確定するため、右値の「getFlag("flag") == ON」が評価されず、関数「getFlag()」も呼び出されないことを意味する。これは C 言語と同じ挙動である。評価が省略されることにより、処理が少しでも効率化される。

### ▼ 組み込み関数

下記の組み込み関数に対応する。どれもありきたりの関数なので、細かい仕様は省略し、関数を列挙する。

- ・ `crc( "str" )` ..... 文字列"str"の CRC 値を算出
- ・ `crcls( "str" )` ..... 文字列"str"を小文字化した文字列の CRC 値を算出
- ・ `pow(x, y)` ..... x の y 乗を算出
- ・ `sqrt(x)` ..... x の平方根を算出
- ・ `abs(x)` ..... x の絶対値を返す
- ・ `sign(x)` ..... x の符号を返す (x=負...-1、x=0...0、x=正...1)
- ・ `sin(x)` ..... x の sin 値を算出 ※x はラジアン
- ・ `cos(x)` ..... x の cos 値を算出 ※x はラジアン
- ・ `tan(x)` ..... x の tan 値を算出 ※x はラジアン
- ・ `asin(x)` ..... x の asin 値を算出 ※x は sin 値
- ・ `acos(x)` ..... x の acos 値を算出 ※x は cos 値
- ・ `atan(x)` ..... x の atan 値を算出 ※x は tan 値
- ・ `atan2(x, y)` ..... x, y の atan2 値を算出
- ・ `toRad(x)` ..... 度をラジアンに変換
- ・ `toDeg(x)` ..... ラジアンを度に変換
- ・ `pi()` .....  $\pi$  (3.141592653589) を返す

なお、組み込み関数は、できる限りデータ変換時に計算結果を算出して値に置き換えるようにする。できない場合はランタイム時に計算される。基本的に、拡張関数がパラメータに含まれない限りは算出可能である。(例：`crc( "c0010" )`⇒可、`pow(4, 3)`⇒可、`sqrt(2.0 * 2.0)`⇒可、`toRad(45)`⇒可、`cos(toRad(45))`⇒可、`pow(getUserLevel(), 2)`⇒不可)

### ▼ 拡張関数

拡張関数も使い方は組み込み関数と同様。

タイトル固有の関数を使用することができる。

データ変換時には、「拡張関数定義リスト」に基づいて、関数名とパラメータが正しいかどうかチェックされる。

## ■ 処理仕様

データ変換および実機でのデータ取り込みの処理仕様を示す。

### ▼ プリプロセッサ

プリプロセッサには MinGW(GCC) を使用。

ダウンロードサイト：<http://sourceforge.net/projects/mingw/files/Installer/>

(インストール方法は省略)

実行手順①： MinGW のパスを通す。バッチファイルなどで、実行時にだけパスに追加する方法でも可。

(例)

```
SET PATH=C:¥MinGW¥bin;¥PATH%
```

実行手順②： コマンドラインコンパイラ `g++.exe` を使用し、プリプロセッサを実行。

(例)

```
$ g++.exe -x c++ -E -c data.json -o data.i.json
```

#### 【使用オプション】

```
-x c++ ..... C++ 言語として処理
-E ..... C プリプロセッサだけを実行
-P ..... #line 文を出力しない
-c ..... 入力ファイルを指定
-o ..... 出力ファイルを指定
-Idir ..... インクルードファイルの検索パスを指定（複数指定可）
-include file
..... 強制インクルードファイルを指定
-Dmacro
-Dmacro=defn
..... マクロを定義 (=defn が無いときは 1 になる)
-M ..... 依存ファイル（インクルードファイル）のリストアップ
```

-Mf file

..... 依存ファイルのリストをファイルに出力

## ▼ データ変換ツール

- ・ 開発言語 : C#コンソールアプリケーション (.Net Framework4.0 以上)  
選定理由 : 開発のし易さ、JSON パーサーの利用、dynamic 型 (遅延バインド) の利用。
- ・ 使用ライブラリ : JSON. Net  
<http://json.codeplex.com/downloads/get/744406>
- ・ ツール名 : `gdconv.exe` ※Game Data CONVertor の意。
- ・ 使用方法 : (例)

```
$ gdconv.exe --be --p64 --sp8 --dbgcrcl --dbgexpr --utf8 --dbgutf8 -t
types.json -t ext_types.json -f funcs.json -f ext_funcs.json -d
char_format.json -i chara_data.json -o chara_data.bin -c
chara_data_chk.json
```

【対応オプション】 ※競合オプションは後に指定されたものが有効

```
--le ..... リトルエンディアン ※デフォルト
--be ..... ビッグエンディアン
--p32 ..... ポインターサイズ : 32bit (4 バイト) ※デフォルト
--p64 ..... ポインターサイズ : 64bit (8 バイト)
--sp1 ..... 構造体パッキング (アラインメント) : 1 バイト
--sp2 ..... 構造体パッキング (アラインメント) : 2 バイト
--sp4 ..... 構造体パッキング (アラインメント) : 4 バイト
--sp8 ..... 構造体パッキング (アラインメント) : 8 バイト ※デフォルト
--sp16 ..... 構造体パッキング (アラインメント) : 16 バイト
--dbgcrcl ... crc/crcs 変換前テキストをデバッグデータ化 ※デフォルト
--dbgexpr .. 計算式変換前テキストをデバッグデータ化 ※デフォルト
--nodbg .... デバッグデータなし
--utf8 ..... エンコーディング : UTF-8 ※デフォルト
--sjis ..... エンコーディング : シフト JIS
--dbgutf8 .. デバッグデータエンコーディング : UTF-8 ※デフォルト
--dbgsjis .. デバッグデータエンコーディング : シフト JIS
--makesrc .. ソースファイル生成 ※-c, -o, -c, -m オプションは無視
--srcutf8 .. ソースファイルエンコーディング : UTF-8 ※デフォルト
--srcsjis .. ソースファイルエンコーディング : シフト JIS
```

`-t file` .... データ型定義リストファイル指定 ※複数指定可  
`-f file` .... 計算式用拡張関数定義リストファイル指定 ※複数指定可  
`-d file` .... フォーマット定義 JSON ファイル指定 ※必須  
`-i file` .... データ定義 JSON ファイル（入力ファイル）指定  
`-o file` .... バイナリファイル（出力ファイル）指定  
`-c file` .... チェック用 JSON ファイル（出力ファイル）指定  
`-m file` .... 中間 JSON②ファイル（出力ファイル）指定  
 ※C 言語ヘッダーファイル（出力ファイル）と C 言語ソース（構造定義情報）  
 ファイル（出力ファイル）はデータ定義 JSON 内で指定。  
 ※`-c` オプションがないとチェック用 JSON は出力されない。  
 ※`-d` オプションがないとエラー。  
 ※`-makesrc` オプション非指定時は、`-c`, `-o` オプションがないとエラー。

各データ仕様および上記オプションに基づいてツールを作成。ツール作成時の注意点は下記のとおり。

- ・ 主キー、副キーの重複エラー判定は、文字列と CRC 値の両方で行う。
- ・ 計算式データ、文字列データ、デバッグデータは、バイナリデータ化の際、同じ内容なら同じ実データを参照するようにする。
- ・ 途中エラーが一つでも検出されたら、一切のファイル出力を行わず、その時点で処理を終了する。
- ・ 途中エラーが一つでも検出されたら、コマンドは 1 を返す。（正常時は 0 を返す。）
- ・ 内部での処理順序は下記のとおり。

データ型定義リスト解析（中間 JSON②化→JSON 解析→内部データ化）

⇒ フォーマット定義 JSON 解析（中間 JSON②化→JSON 解析→内部データ化）

⇒ フォーマット定義の整合性チェック

※ 存在しない型が使用されていたらエラー、ネストした構造体が定義されていなければエラー、などをチェック。

【以下、C 言語ソース出力時は行わない処理】

⇒ 計算式拡張関数定義リスト解析（中間 JSON②化→JSON 解析→内部データ化）

⇒ データ定義 JSON 解析（中間 JSON②化→JSON 解析→内部データ化）

※ 中間 JSON②作成と同時に、CRC 値化された文字列をデバッグ用データとして蓄積。

※ 同 CRC 値のデータが蓄積済みなら、文字列が一致するかチェックして不一致ならエラー。

※ `crcs()` 関数使用箇所は、小文字化した文字列で扱う（元の文字列は失われる）。

- ⇒ データのフォーマット照合
    - ※ フォーマット定義に存在しないキーが使用されていたらエラー、主キー／副キー／インデックス対象項目が定義されていなければエラー、など。
  - ⇒ 不定長データを分割
    - ※ 基本のデータと、不定長データ（の型）の数だけリストが作られる。
  - ⇒ 計算式（expr 関数）解析処理（計算式データに変換して蓄積、元の計算式文字列をデバッグ用データとして蓄積）
    - ※ 計算式文字列の CRC 値で計算式データ蓄積済みかどうかをチェックする。デバッグ用データも同様。
    - ※ 同 CRC 値のデバッグ用データが蓄積済みなら、計算式の文字列が一致するかチェックして不一致ならエラー。
  - ⇒ 文字列解析処理（文字列データとして蓄積）
    - ※ 文字列の CRC 値で文字列データ蓄積済みかどうかをチェックする。
    - ※ 同 CRC 値のデータが蓄積済みなら、文字列が一致するかチェックして不一致ならエラー。
  - ⇒ 主キーによるデータソート
    - ※ 重複があったらエラー。
    - ※ 処理対象は基本のデータリストのみ。不定長データのリストはなにもしない。
  - ⇒ 検索インデックス作成（副キー）
    - ※ 重複があったらエラー。
  - ⇒ 並べ替えインデックス作成
  - ⇒ バイナリデータのメモリサイズ計算
  - ⇒ 不定長データ、計算式データ、文字列データのオフセット値化
    - ※ 処理対象は基本のデータリストと不定長データのリスト全て。
  - ⇒ バイナリデータ出力
  - ⇒ チェック用 JSON 出力
  - ⇒ 中間 JSON②出力
    - ※ 対象は、データ定義 JSON の中間 JSON②のみ。
- 【以下、C 言語ソース出力時のみ行う処理】
- ⇒ C 言語ヘッダーファイル出力
  - ⇒ C 言語ソース（構造定義情報）ファイル出力

## ▼ データ変換実行バッチ

プリプロセッサからデータ変換を実行するバッチのサンプル。  
ビルドツールを使用して効率化する手法については後述。

サンプル：

```
@ECHO OFF

rem --- バッチファイルの存在するパス（ドライブ&ディレクトリ）にカレントディレクトリを変更 ---
CD /D %~dp0

rem --- MinGW(GCC) のパスを通す ---
SET MINGW_BIN_DIR=D:\MinGW\bin
SET PATH=%MINGW_BIN_DIR%;%PATH%

rem --- プリプロセッサオプション ---
SET PP_FLAGS=-x c++ -E -P
SET PP_INC_DIRS=-I . -I "lib/common"
rem SET PP_INCLUDES=-include "common.hjson"
SET PP_INCLUDES=

rem --- データ変換オプション ---
SET GD_FLAGS=-l -p32 -sp8 -dbgcr -dbgexr -utf8 -dbgutf8
SET GD_MKSRC_FLAGS=-makesrc -srcutf8

SET GD_TYPE_FILE01=settings/common_type
SET GD_TYPE_FILE02=settings/ext_type

SET GD_FUNC_FILE01=settings/ext_func01
SET GD_FUNC_FILE02=settings/ext_func02

SET GD_FORMAT_FILE=settings/chara_fomat
SET GD_FORMAT_INC_FILE=include/chara_format.h
SET GD_FORMAT_SRC_FILE=src/chara_format.cpp

rem --- 開始 ---
ECHO ***** データ変換開始 *****
ECHO.
SET RESULT_TOTAL=0

rem --- 型定義リストのプリプロセッサ実行 ---
CALL :PREPROCESSOR %GD_TYPE_FILE01%
CALL :PREPROCESSOR %GD_TYPE_FILE02%

rem --- 拡張関数定義リストのプリプロセッサ実行 ---
CALL :PREPROCESSOR %GD_FUNC_FILE01%
CALL :PREPROCESSOR %GD_FUNC_FILE02%

rem --- フォーマット定義 JSON のプリプロセッサ実行 ---
CALL :PREPROCESSOR %GD_FORMAT_FILE%

rem --- ソースファイル作成 ---
CALL :GDSRC CONV %GD_FORMAT_FILE%

rem --- データ変換実行 ---
CALL :GD CONV %GD_FORMAT_FILE% data/data01 bin/data01
CALL :GD CONV %GD_FORMAT_FILE% data/data02 bin/data02
CALL :GD CONV %GD_FORMAT_FILE% data/data03 bin/data03

rem --- 終了 ---
ECHO.
ECHO エラー数: %RESULT_TOTAL%
```

```

CALL :PAUSE_MSG ***** データ変換終了 *****
EXIT /B %RESULT_TOTAL%

rem ===== プリプロセッサ実行 =====
:PREPROCESSOR
SET JSON_FILE=%1.json
SET JSON_PP_FILE=%2
IF "%JSON_PP_FILE%" == "" SET JSON_PP_FILE=%1
SET JSON_PP_FILE=%JSON_PP_FILE%.i.json
ECHO.
ECHO -----
ECHO プリプロセッサ実行: "%JSON_FILE%" to "%JSON_PP_FILE%"
CALL :RESET_ERRORLEVEL
CALL :RUN_CMD g++.exe %PP_FLAGS% %PP_INC_DIRS% %PP_INCLUDES% -c "%JSON_FILE%" -o "%JSON_PP_FILE%"
SET RESULT=%ERRORLEVEL%
IF NOT "%RESULT%" == "0" ECHO エラー！
IF NOT "%RESULT%" == "0" SET /A RESULT_TOTAL=RESULT_TOTAL+1
EXIT /B %RESULT%

rem ===== ソースファイル生成実行 =====
:GDSRCCONV
SET FORMAT_FILE=%1.i.json
ECHO.
ECHO -----
ECHO ソースファイル生成: "%FORMAT_FILE%"
CALL :RESET_ERRORLEVEL
CALL :RUN_CMD gdconv.exe %GD_MKSRC_FLAGS% -t "%GD_TYPE_FILE01%.i.json" -t "%GD_TYPE_FILE02%.i.json" -f
"%GD_FUNC_FILE01%.i.json" -f "%GD_FUNC_FILE02%.i.json" -d "%FORMAT_FILE%"
SET RESULT=%ERRORLEVEL%
IF NOT "%RESULT%" == "0" ECHO エラー！
IF NOT "%RESULT%" == "0" SET /A RESULT_TOTAL=RESULT_TOTAL+1
EXIT /B %RESULT%

rem ===== データ変換実行 =====
:GDCONV
SET FORMAT_FILE=%1.i.json
SET JSON_FILE=%2.json
SET JSON_PP_FILE=%2.i.json
SET JSON_MID_FILE=%2.m.json
SET JSON_CHECK_FILE=%2.c.json
SET BIN_FILE=%3.bin
ECHO.
ECHO -----
ECHO データ変換: "%JSON_FILE%" to "%BIN_FILE%"
CALL :RESET_ERRORLEVEL
CALL :RUN_CMD g++.exe %PP_FLAGS% %PP_INC_DIRS% %PP_INCLUDES% -c "%JSON_FILE%" -o "%JSON_PP_FILE%"
SET RESULT=%ERRORLEVEL%
IF NOT "%RESULT%" == "0" ECHO エラー！
IF NOT "%RESULT%" == "0" SET /A RESULT_TOTAL=RESULT_TOTAL+1
CALL :RUN_CMD gdconv.exe %GD_FLAGS% -t "%GD_TYPE_FILE01%.i.json" -t "%GD_TYPE_FILE02%.i.json" -f
"%GD_FUNC_FILE01%.i.json" -f "%GD_FUNC_FILE02%.i.json" -d "%FORMAT_FILE%" -i "%JSON_PP_FILE%" -o "%BIN_FILE%" -c
"%JSON_CHECK_FILE%" -m "%JSON_MID_FILE%"
SET RESULT=%ERRORLEVEL%
IF NOT "%RESULT%" == "0" ECHO エラー！
IF NOT "%RESULT%" == "0" SET /A RESULT_TOTAL=RESULT_TOTAL+1
EXIT /B %RESULT%

rem ===== 標準サブルーチン =====

rem --- 標準サブルーチン: コマンドラインを表示して実行 ---
:RUN_CMD
ECHO ^> %*
%*
EXIT /B %ERRORLEVEL%

```

```

rem --- 標準サブルーチン：メッセージ付きポーズ ---
:PAUSE_MSG
SET MSG=%*
IF "%MSG%" == "" SET MSG=続行するには何かキーを押してください...
ECHO. %MSG%
PAUSE >NUL
EXIT /B

rem --- ERRORLEVEL リセット ---
:RESET_ERRORLEVEL
DATE /t >NUL
EXIT /B

```

## ▼ データ変換処理（C# Tips）

C#でJSONを活用する方法を解説。

### ● JSON.Net の使用

準備：まずC#のプロジェクトにて、「参照設定」に、JSON.Net の Newtonsoft.Json.dll を追加しておく。

処理例：C#プログラムソース

```

using System;

//JSON.Net のネームスペース使用指定
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;

//JSON パーサーテスト
void test()
{
    //JSON テキスト
    string json_text = @"
    {
        ""data1"":
        [
            { ""id"": ""c0010"", ""name"": ""田中"", },
            { ""id"": ""c0020"", ""name"": ""山田"", },
            { ""id"": ""c0030"", ""name"": ""佐藤"", },
        ],
        ""data2"":
        [
            [1, 2, 3],
            [4, 5, 6],
            [7, 8, 9, 10],
        ]
    }
    ";

    //JSON テキストを解析
    //※本来は JObject 型を返すが、dynamic 型（動的型付け）でないと、
    // 遅延バインディングが行われないため、コンパイル時にエラーになってしまう。
    dynamic json_obj = JObject.Parse(json_text);

    //dynamic 型を使用することにより、型宣言せずに、
    //JSON 内で定義されたメンバーに直接アクセスが可能
    Console.WriteLine();

```



```

Console.WriteLine("data1[0].id={0}¥", json_obj.data1[0].id);
Console.WriteLine("data1[2].name={0}¥", json_obj.data1[2].name);
Console.WriteLine("data2[2][3]={0}", json_obj.data2[2][3]);

//foreach も使用可
Console.WriteLine();
Console.WriteLine("data1=[");
foreach(dynamic rec in json_obj.data1)
{
    Console.WriteLine(" [{ id={0}¥, name={1}¥ }],", rec.id, rec.name);
}
Console.WriteLine("]");

//foreach のネストも使用可
Console.WriteLine();
Console.WriteLine("data2=[");
foreach (dynamic array1 in json_obj.data2)
{
    Console.Write(" [");
    foreach (dynamic array2 in array1)
    {
        Console.Write(array2 + ", ");
    }
    Console.WriteLine("]");
}
Console.WriteLine("]");
}

```

↓ (処理結果)

```

data1[0].id="c0010"
data1[2].name="佐藤"
data2[2][3]=10

data1=[
 { id="c0010", name="田中" },
 { id="c0020", name="山田" },
 { id="c0030", name="佐藤" },
]

data2=[
 [1, 2, 3, ]
 [4, 5, 6, ]
 [7, 8, 9, 10, ]
]

```

### ● JSON 解析済みオブジェクトを C# スクリプト内で使用

準備：まず C# のプロジェクトにて、「参照設定」に「Microsoft.CSharp」を追加しておく。

処理例：C# プログラムソース

```

using System;

//JSON.Net のネームスペース使用指定
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;

//C#コンパイラ使用指定
using Microsoft.CSharp;

```

```
using System.CodeDom.Compiler;
using System.Reflection;

//C#スクリプト+JSON+dynamic 型テスト
void test()
{
    //JSON テキスト
    string json_text = @"
    {
        ""name"": ""田中"",
        ""data"": 10
    }
    ";

    //JSON オブジェクト生成
    dynamic json_obj = JObject.Parse(json_text);

    //C#スクリプト
    string cs_source = @"
using System;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
public class CScriptTest
{
    public string checkJsonData(dynamic json_obj)
    {
        //スクリプト外部から受け渡された型情報不明な JSON 解析済みオブジェクトに直接アクセス
        string msg = ""スクリプト内のメッセージテキスト("" + json_obj.data + ""))"";
        json_obj.name += ""さん"";
        json_obj.data += 1;
        return msg;
    }
}
";

    //コンパイラパラメータ生成
    CompilerParameters param = new CompilerParameters();

    //コンパイラパラメータ設定
    param.GenerateExecutable = false; // 実行ファイル (.exe) を作らない
    param.GenerateInMemory = true; // メモリ上にプログラム (スクリプト) を生成
    param.IncludeDebugInformation = false; // デバッグ情報を付加しない

    //スクリプト内で使用する .Net アセンブリをコンパイラパラメータに追加
    //※以下の設定により、JSON. Net と dynamic 型が使用可能になる
    param.ReferencedAssemblies.Add("System.dll");
    param.ReferencedAssemblies.Add("System.Core.dll");
    param.ReferencedAssemblies.Add("Microsoft.CSharp.dll");
    param.ReferencedAssemblies.Add("Newtonsoft.Json.dll");

    //C#コンパイラ生成
    CSharpCodeProvider provider = new CSharpCodeProvider();

    //C#スクリプトをコンパイル
    CompilerResults script_prog = provider.CompileAssemblyFromSource(param, cs_source);
    if (script_prog.Errors.Count > 0)
    {
        //コンパイルエラーあり
        Console.WriteLine("コンパイルエラー数 = " + script_prog.Errors.Count);
        foreach (var err in script_prog.Errors)
        {
            Console.WriteLine("コンパイルエラー:" + err.ToString());
        }
    }
    else
```

```

{
    //コンパイルエラーなし

    //スクリプト内の型（クラス）情報を取得
    Type script_class = script_prog.CompiledAssembly.GetType("CScriptTest");

    //インスタンスを生成（コンストラクタ呼び出し：パラメータなしコンストラクタ呼び出しのケース）
    dynamic script_obj = script_class.GetConstructors()[0].Invoke(new Object[0]);

    //実行前状態表示
    Console.WriteLine();
    Console.WriteLine("実行前：json_objd.name={0}", json_obj.name, json_obj.data);

    //スクリプト内のメソッド呼び出し ※JSON テキスト解析済みオブジェクトを受け渡し
    string ret_msg = script_obj.checkJsonData(json_obj);

    //実行後結果表示
    Console.WriteLine();
    Console.WriteLine("実行後：json_objd.name={0}", json_obj.name, json_obj.data);
    if(ret_msg != null)
    {
        Console.WriteLine("  戻り値={0}", ret_msg);
    }
}

```

↓ （処理結果）

```

実行前：json_objd.name="田中", .data=10

実行後：json_objd.name="田中さん", .data=11
  戻り値="スクリプト内のメッセージテキスト(10)"

```

#### ▼ データ取り込み処理（ランタイム）

#### ▼ データアクセス処理（ランタイム）

### ■ 計算式処理仕様

#### ▼ 計算式解析処理

データ変換の処理中に行われる「計算式解析処理」について、詳しく説明する。

基本的な処理の流れは、「字句解析（トークン分解）」⇒「抽象構文木作成」⇒「ポーランド記法によるバイナリデータ化」。

《抽象構文木によるゲームデータ内計算式の構文解析処理手順》

サンプル計算式:  $1+2*(3-(-4+5))-6/2$

トークン分解: ※最初に字句解析処理(Lexical Analyzer 処理)で、計算式の文字列を各トークンに分解する



抽象構文木作成処理:

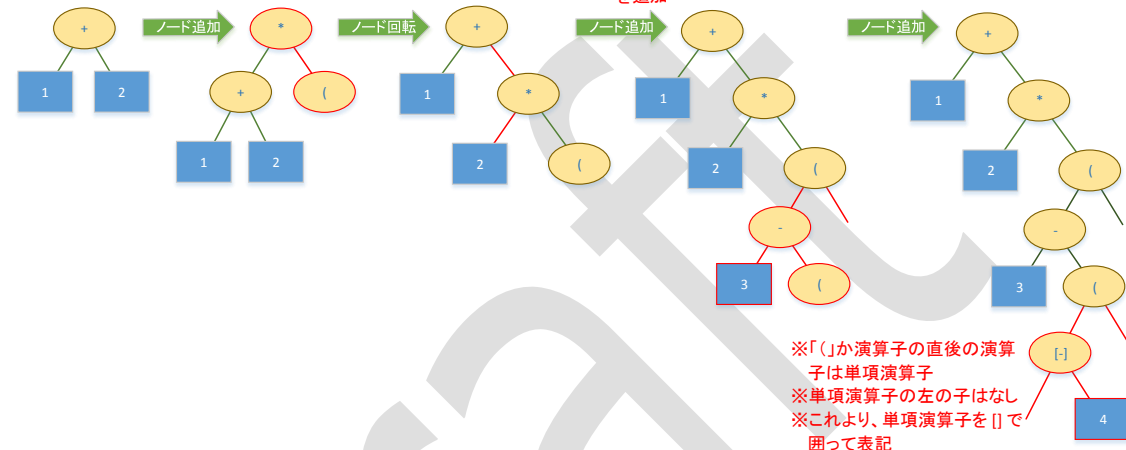
※演算子の子に二つの値がぶら下がるようにノードを連結

※トップノードに次の演算子を追加

※優先順位の高い演算子を、低い演算子の子どもにする

※「(」をカレントのトップノードとして次のノードを追加

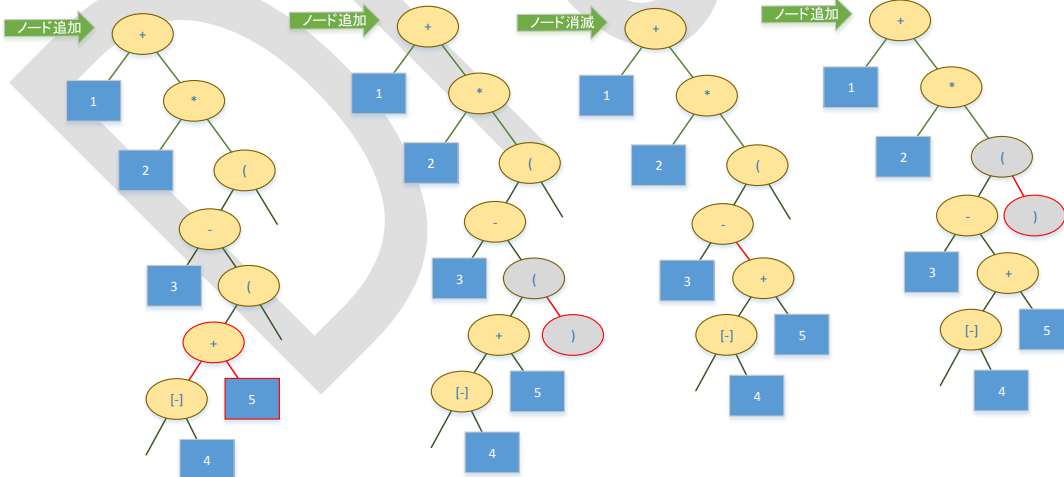
※同様に末端の「)」をカレントのトップノードとして次のノードを追加

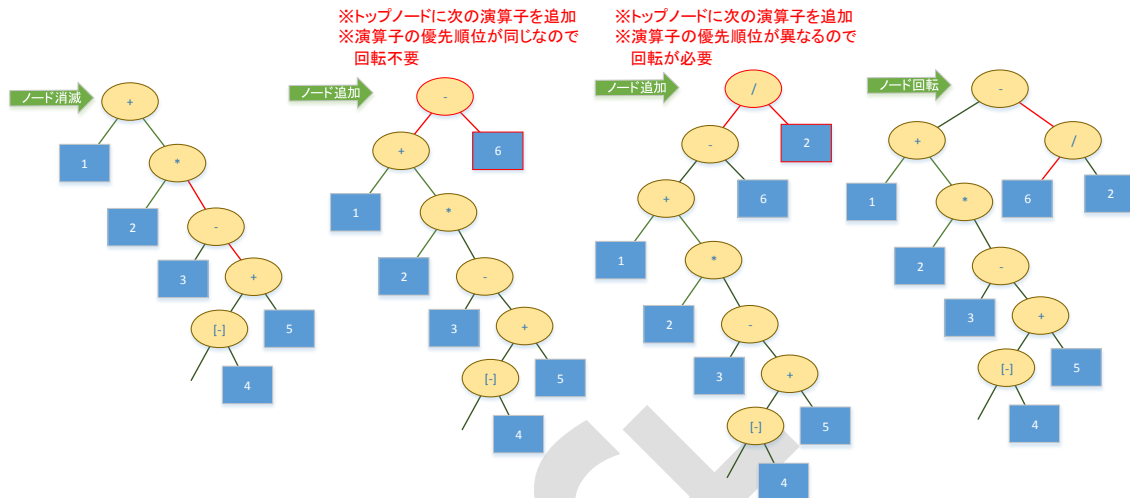


※末端の「)」をカレントのトップノードとして次のノードを追加

※「)」が来たら、「(」と「)」が対消滅

※同様に、「)」が来たら、「(」と「)」が対消滅





頭から順にトークンを再連結してデータ化(ポーランド記法):

※トップから左側のノードに降りて行きながら、「演算子+左のノード+右のノード」と並べていく



※演算子と値をスタックに積んでいき、値が二つ揃った時点で直前の演算子の計算を実行してスタックに積み直す。

単項演算子の場合は値が一つの時にそれを行う。これを繰り返すと最後に計算結果がスタックに残る。

試算:  $\neg, +, 1, *, 2, -, 3, +, [, 4 \Rightarrow \neg, +, 1, *, 2, -, 3, +, -4 \Rightarrow \neg, +, 1, *, 2, -, 3, +, -, 4, 5 \Rightarrow \neg, +, 1, *, 2, -, 3, 1 \Rightarrow \neg, +, 1, *, 2, -, 3, 1 \Rightarrow \neg, +, 1, *, 2, 2 \Rightarrow \neg, +, 1, *, 2, 2 \Rightarrow \neg, +, 1, 4 \Rightarrow \neg, +, 1, 4 \Rightarrow \neg, 5 \Rightarrow \neg, 5, /, 6, 2 \Rightarrow \neg, 5, 3 \Rightarrow \neg, 5, 3 \Rightarrow 2$

元の計算式の試算:  $1+2*(3-(4+5))-6/2 \Rightarrow 1+2*(3-1)-6/2 \Rightarrow 1+2*2-6/2 \Rightarrow 1+4-6/2 \Rightarrow 1+4-3 \Rightarrow 5-3 \Rightarrow 2$

※「逆ポーランド記法」の構造は不採用。こちらの方が消費スタックサイズを抑えて単純な処理にできるが、右値の評価を省略して処理を効率化することが難しいので、その対応が取り易い「ポーランド記法」の構造を採用する。

末端から順にトークンを再連結してデータ化(逆ポーランド記法):

※一番左奥のノードから順に、「左のノード+右のノード+演算子」と並べていく



※値をスタックに積んでいき、演算子が来たら二つの値をスタックから取り出して計算し、計算結果をスタックに積み直す。

単項演算子の場合は一つの値に対してそれを行う。これを繰り返すと最後に計算結果がスタックに残る。

試算:  $1, 2, 3, 4, [] \Rightarrow 1, 2, 3, -4 \Rightarrow 1, 2, 3, -4, 5, + \Rightarrow 1, 2, 3, 1 \Rightarrow 1, 2, 3, 1, - \Rightarrow 1, 2, 2 \Rightarrow 1, 2, 2, * \Rightarrow 1, 4 \Rightarrow 1, 4, + \Rightarrow 5 \Rightarrow 5, 6, 2, / \Rightarrow 5, 3 \Rightarrow 5, 3, - \Rightarrow 2$

元の計算式の試算:  $1+2*(3-(4+5))-6/2 \Rightarrow 1+2*(3-1)-6/2 \Rightarrow 1+2*2-6/2 \Rightarrow 1+4-6/2 \Rightarrow 1+4-3 \Rightarrow 5-3 \Rightarrow 2$

# 「ゲームデータ内計算式の抽象構文木による解析とデータ化」

基本形:

10+20



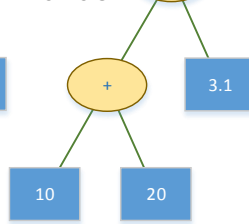
①+,10,20

②[3]+,0,1:i:10,i20

③[3]+,0,1:i,i:10,20

整数+小数:

10+20-3.1



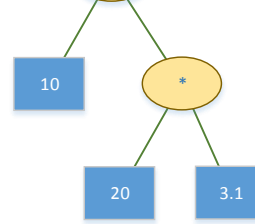
①-,+,10,20,3.1

②[5]-,+,0,1,2:i:10,i20,f3.1

③[5]-,+,0,1,2:i,i,f:10,20,3.1

演算子優先順位:

10+20\*3.1



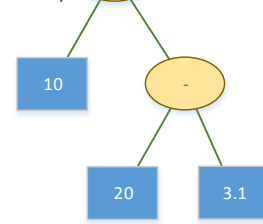
①+,10,\*,20,3.1

②[5]+,0,\*,1,2,+:i:10,i20,f3.1

③[5]+,0,\*,1,2:i,i,f:10,20,3.1

括弧:

10+(20-3.1)



①+,10,-,20,3.1

②[5]+,0,-,1,2:i:10,i20,f3.1

③[5]+,0,-,1,2,-:i,i,f:10,20,3.1

①ポーランド記法に変換(括弧の除去と演算子の優先順位を反映)

②数値部をインデックス化(先頭にデータ個数)

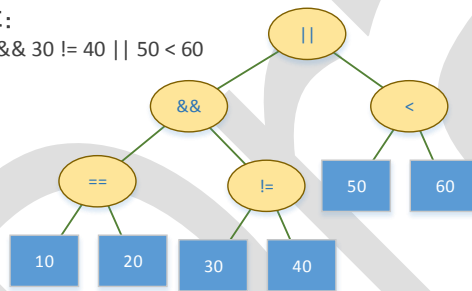
③数値部を型と数値に分離(データサイズ縮小)

※本来は、①のデータ化の過程で、演算子の両側が値だった場合、その時点で計算を済ませてしまい、「値」のノードに変換して扱う。これにより、少しでもデータが小さくなり、ランタイム時の処理も速くなる。つまり、ランタイム時にしか値が分からない要素(=関数)が使用されない限りは、計算式は極力縮小される。

※一部の関数も、パラメータが全て値の場合は、①のデータ化の過程で計算結果(値)に置き換える。⇒対象関数: pow(x, y), sqrt(x, y), crc("str"), crcs("str"), abs(x), sign(x), sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), atan2(x), log(x), log10(x), min(x, y), max(x, y), pi()など

論理演算:

10 == 20 && 30 != 40 || 50 < 60



①||,&&==,10,20,! =,30,40,<,50,60

②[11]||,&&==,0,1,! =,2,3,<,4,5:i:10,i20,i30,i40,i50

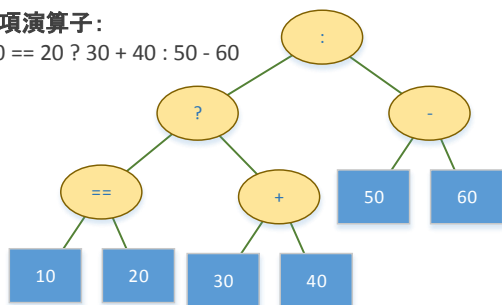
③[11]||,&&==,0,1,! =,2,3,<,4,5:i,i,i,i:10,20,30,40,50

※実機の処理にて、「||,&&==,10,20」までを処理した時点のスタックは「||,&&,false」となる。「&&」の左値がfalseに確定したので、右値は評価する必要がない。後続の「!=,30,40」の部分は計算せずにスタックの消化のみを行う。具体的には、後続の演算子と値の関係にだけ注目を続け、「演算子,左値,右値⇒false(値)」と変換しながら、一つの値が残った時点(&&の右値が確定した時点)で走査を終了する。その結果「||,&&,false,false」となり「||,false」となる。その後、残りの「<,50,60」を処理する。

このようにして、少しでも処理を効率化する。特に関数が使われている場合は、関数の呼び出しを行わなくなるので、処理効率が向上する。実際にC言語なども、このように評価を省略する挙動になっている。なお、「||」の場合は、左値がtrueに確定したなら右値を評価しない。

## 3項演算子:

10 == 20 ? 30 + 40 : 50 - 60



①:,:,?,==,10,20,+,30,40,-,50,60

②[11]:,:,?,==,0,1,+,2,3,-,50,60:i10,i20,i30,i40,i50

③[11]:,:,?,==,0,1,+,2,3,-,50,60:i,i,i,i:10,20,30,40,50

※三項演算子の場合、まず、「?」の親ノードが「:」という関係を前提とする。

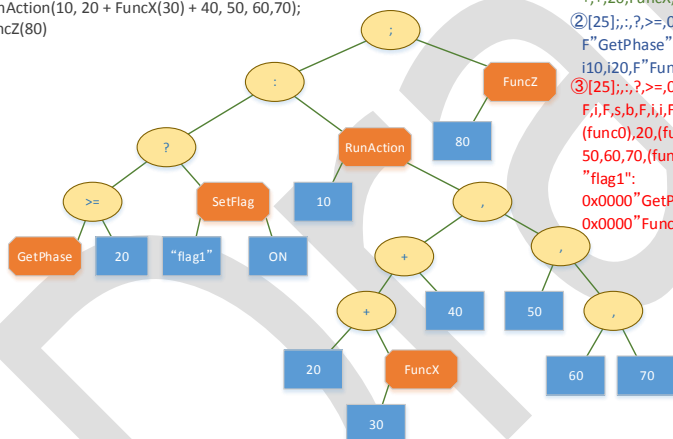
※「:,:,?,==,10,20」までを処理した時点のスタックは「:,:,?,false」となる。「?」の左値がfalseに確定したので、右値は評価する必要がない。後続の「+,30,40」の部分は計算せずにスタックの消化のみを行う。その結果「:,:,?,false,false」となり「:,:,false」となる。「:」の左値がfalseに確定したので、残りの「-,50,60」を処理する。

「?」演算子は、左値がtrueなら右値を評価して値を返し、falseなら右値を評価せずにfalseを返す。

「:」演算子は、左値がfalseなら右値を評価して値を返し、それ以外(0もあり得る)なら右値を評価せずに左値をそのまま返す。

## 関数呼び出し／文字列「:」、「:」の使用:

GetPhase() >= 20 ? SetFlag("flag1", ON) :  
RunAction(10, 20 + FuncX(30) + 40, 50, 60, 70);  
FuncZ(80)



①:,:,;,>=,GetPhase,20,SetFlag,"flag1",ON,RunAction,10,,

+,+,20,FuncX,30,40,,,50,,,60,70,FuncZ,80

②[25]:,:,;,>=,0,1,2,3,4,5,6,,+,+,7,8,9,10,,11,12,,13,14:

F"GetPhase",i20,F"SetFlag",s"flag1",true,F"RunAction",

i10,i20,F"FuncX",i30,i40,i50,i60,i70,F"FuncZ",i80

③[25]:,:,;,>=,0,1,2,3,4,5,6,,+,+,7,8,9,10,,11,12,,13,14,15:

F,i,F,s,b,F,i,i,F,i,i,i,i,i,F,i:

(func0),20,(func16),(str0),1,(func28),10,20,(func44),30,40,

50,60,70,(func56),80:

"flag1":

0x0000"GetPhase",0x0000"SetFlag",0x0000"RunAction",

0x0000"FuncX",0x0000"FuncZ"

※③のデータ構造では、値のリストに続けて、文字列のリストと関数(ID+文字列)のリストが続く。

※値リスト上での文字列と関数は、それぞれリストのオフセット値を格納する。

※関数リストは4バイトアラインメントで、ID+文字列で構成。ID部は4バイトの整数で、関数名のCRC値。通常、実機内の処理では、ID(CRC値)で関数がマッピングされ、ランタイムエラーメッセージ表示時のみ関数名が使用される。

※関数のパラメータが二つの場合は、「:」のノードを削除し、関数の左右の子にパラメータを付ける。三つ以上の場合、右の子に「:」を付け、一つずつパラメータを追加していく。

※実機の処理にて、「:」や「:」は他の演算子のように計算結果をスタックし直すようなことはせず、左右の子の値をそのままスタックする。

※実機の処理にて、関数は下記のタイミングで実行する。

- 関数に子ノードが一つも無い場合(パラメータ0)
- 関数の左の子ノードだけがある場合(パラメータ一つ)
- 関数の左右の子ノードが値の場合(パラメータ二つ)
- 「:」の右の子ノードが値の場合(パラメータ三つ以上)

## ▼ 計算式データの構造

▼ 計算式実行処理（ランタイム）

■ データ変換作業環境の改善

▼ バッチファイル の利用

▼ SCons の利用

SCons  
Python

以上



## ■ 索引

**G**

GCC..... 29

**J**

JSON..... 6

**M**

MinGW..... 29

**P**

Python..... 43

**S**

SCons..... 43

**け**

計算式

逆ポーランド記法..... 40

ポーランド記法..... 40

ゲームデータ..... 1

ゲームデータ仕様

---

以 上