ゲームデータ仕様

– ゲームデータのフォーマットと変換 –

2014 年 1 月 10 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 10 日	板垣 衛	（初版）

■ 目次

■ 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、ポインターのビット数（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。

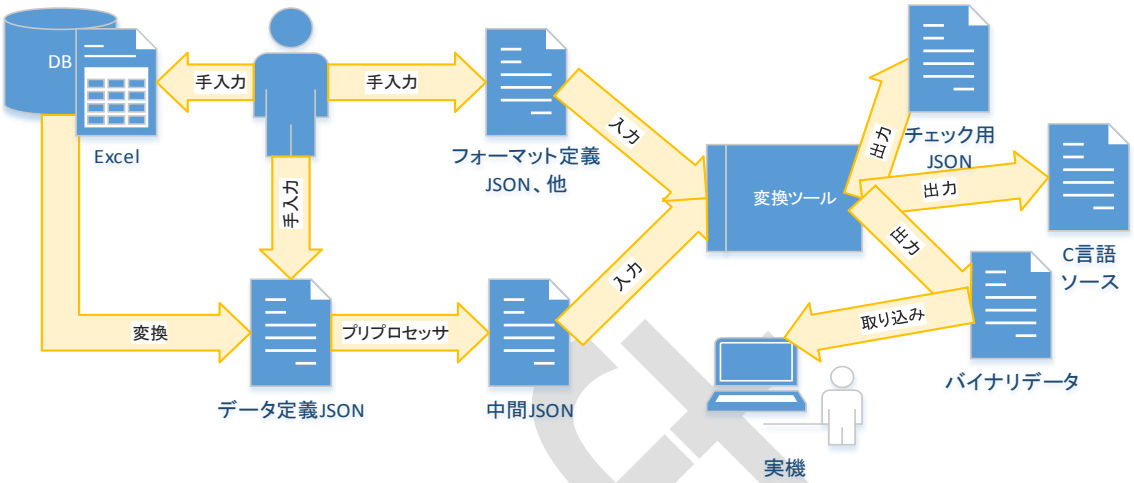
- バイナリデータは、メモリ上のイメージとしてほぼそのまま取り込める。
  - ➢ 文字列のポインター変換などの処理も取り込み時に同時に行われる。
- バイナリデータの構造が変更され、実機上の構造とずれた場合、取り込み処理は自動的にその事を検出し、項目毎のデータ取り込み処理を行う。要は、データ構造が変わっても、プログラムが極力正常に動作するようにする。

■ 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 ..... テキストデータ（JSON 形式＋JavaScript 形式コメント＋C 言語形式#include 文・#define 文＋四則演算式＋特殊関数）
- 中間 JSON ..... コメント除去、#include 文・#define 文展開したもの
- フォーマット定義 JSON ... データ変換・ルール・バージョン定義
- データ型定義リスト ..... フォーマット定義 JSON と共に用いる。データ型とそのデータサイズが定義されたリスト。
- 計算式用拡張関数定義リスト ..... フォーマット定義 JSON と共に用いる。計算式解析処理時に、拡張関数が正しく指定されているか判定するためのリスト。
- 変換ツール ..... JSON データをバイナリデータに変換（四則演算、特殊関数計算も行う）
- 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 ..... 構造体とデータフォーマット定義（オプションで出力、フォーマット定義 JSON のみに基づいて生成）
- チェック用 JSON ..... 変換済みデータと同じ内容の JSON（内容確認用）
- 実機 ..... ゲーム実行時にバイナリデータ取り込み

■ データ仕様

▼ 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用できない。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」と「フォーマット定義 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。

● JSON の採用

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、JaveScript 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
  - 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。
  - 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。
  - 配列を記述可能。
  - オブジェクト（構造体）を記述可能。

- 配列の記述：
  - 「[」～「]」（角括弧）で囲む事で表現する。

例：

```
[ 1, 2, 3 ]
[ “山田”, “田中”, “佐藤” ]
```

多重配列も可能：

```
[
    [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：

```
[
      {"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
```

```
*/
[
      { KEY_ID: CRC( “c0010” ), KEY_NAME: TO_NAME(山田), param: {"str": STR(1), "vit": VIT(1) } },
      { 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。

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 JSON

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。

例：前述の「拡張 JSON」の中間 JSON

```
（コメント除去や#define 消滅後、空行が残る。エラーメッセージ出力時に行番号がずれないように、行位置は変化しない。）


# 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。）
```

```
[
    { "id" : CRC( "c0010" ), "name" : "山田", param: {"str": 10 + 1, "vit": 20 + 1 } },
    { "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 } }
]
```

▼ 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" : "charaDataDecle.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,                   //#pramgma 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。

    "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、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、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
        },
],

 “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」とほどぼ同様の構造だが、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どうか？

                "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
                        "isVariableArray": 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どうか？

                "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
                    }
                ]
            }
    ]

    //【要調査】できれば対応
    //エラー判定用ルール  ※メンバーごとの min, max, isReuired 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 JavaScript 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 obj 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して返すことも可能。
             “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。¥”; ¥
                } ¥
                return obj; ¥
             “
        },
        {
            //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 obj; ¥
             “
        }
    ],
}
```

▼ 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（ポインター）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、
                                        //               構造体にポインター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。ポインターの
                                        //               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, i8, u8, i16, u16, i32, u32, i64, u64, f32, f64, str, crc, crcs, expr

▼ 計算式用拡張関数定義リスト

JSON データ内の「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適用される関数を拡張するものではないことに注意。

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
    }
]
```

▼ チェック用 JSON

バイナリ出力が成功した時にだけ出力される。

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 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” ]
    }
]
```

↓

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構造定義ファイルを出力する。

データ変換ツールのオプションにより出力可能。その際、JSON データは必要なく、フォーマット定義 JSON に基づいて作成される。以下に自動生成されたファイルのサンプルを示す。これは、前述の「フォーマット定義 JSON」のサンプルから出力した結果である。

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 int baseType:3;      //型種別  ※0 = int, 1 = float, 2 = dec, 3 = bool, 4 = str、
                                          //          5 = expr, 6 = ptr, 7 = struct。
            unsigned int isUnsigned:1;    //符号無し型か？  ※型種別 = int の時だけ指定可。
            unsigned int 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 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 char ptrSize;        //ポインターサイズ(4 or 8)
    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
static const int s_structMembersNum = 9 + 4 + 2;
static T_GAME_DATA_DECL::STRUCT::MEMBER s_structMembers[s_structMembersNum] =
{
    //T_CHARA
    { 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
    { 0x0ad84385, 0x00000000, 1, false, 1, false, 10 },    //耐性:" tol" :f32
    { 0xb8388da4, 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
    { 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
    { 0x1b7cbdfb, 0x00000000, 0, true, 4, false, 0 },      //闇:" dark" :u32
    { 0x076291bf, 0x00000000, 0, true, 4, false, 0 },      //光:" shine" :u32
};
static const int s_structsNum = 3;
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
};
static T_GAME_DATA_DECL s_decl =
{
    0xe1e4a645,  //データフォーマット名 CRC：CharaData
    1,           //データフォーマットメジャーバージョン
    0,           //データフォーマットマイナーバージョン
    8,           //ポインターサイズ
    s_structsNum,//構造対数
    &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 charaDataDef
```

「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

■ 組み込み関数仕様

▼ 算術関数

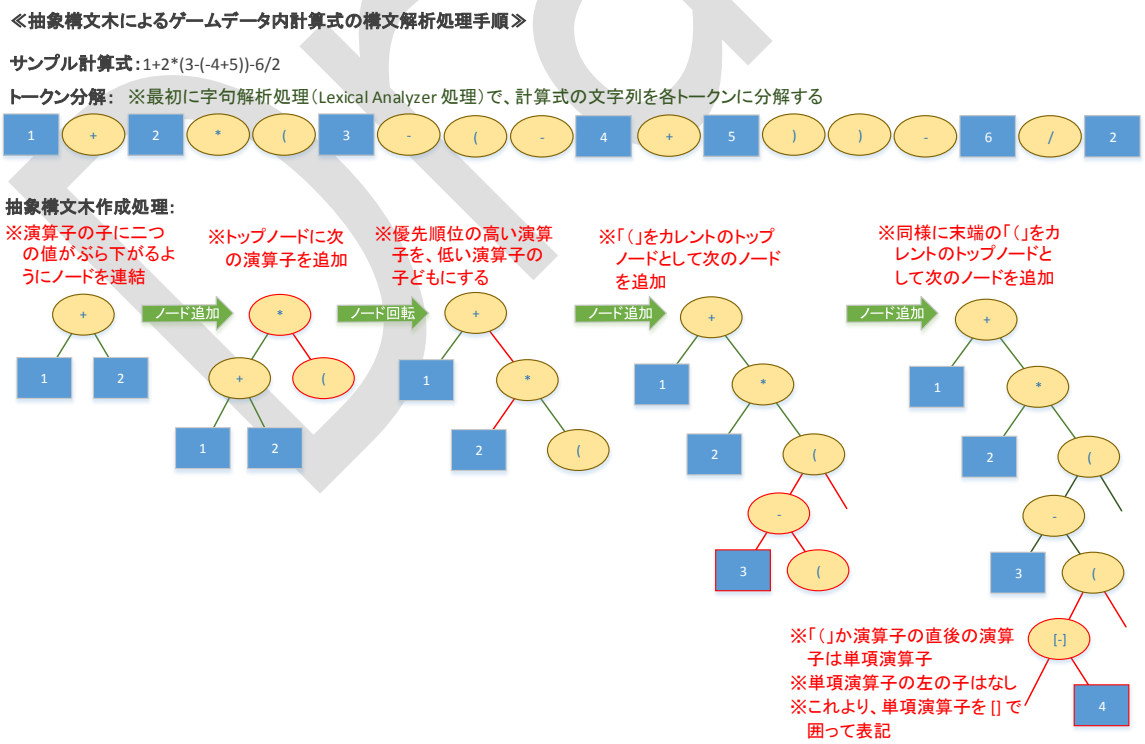
▼ CRC 値変換関数：crc() / crcs()

▼ 計算式解析関数：expr()

■ ゲームデータ内計算式仕様

expr(“expression”)関数で計算される計算式のデータ変換仕様を示す。

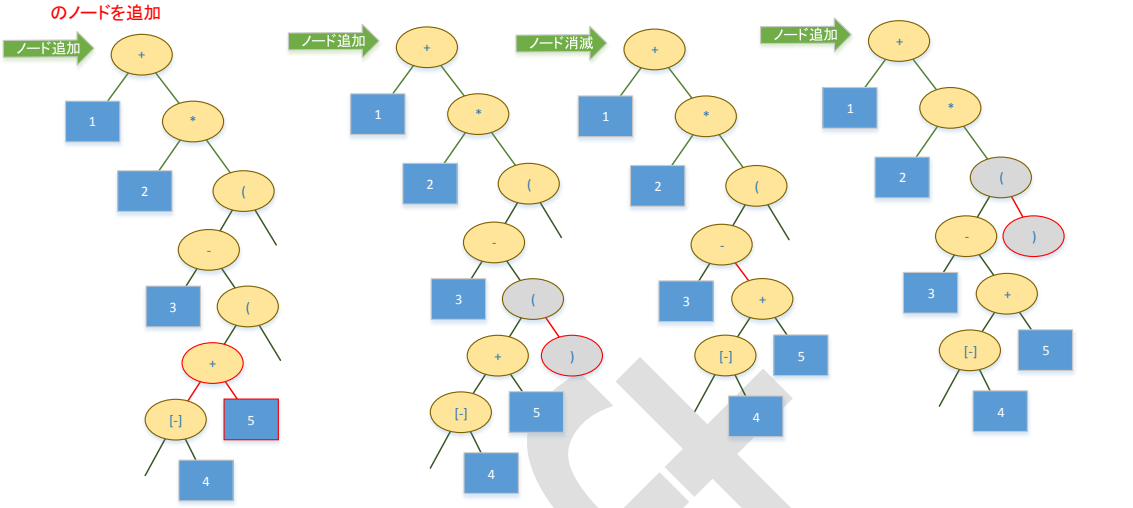
▼ 計算式解析



※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

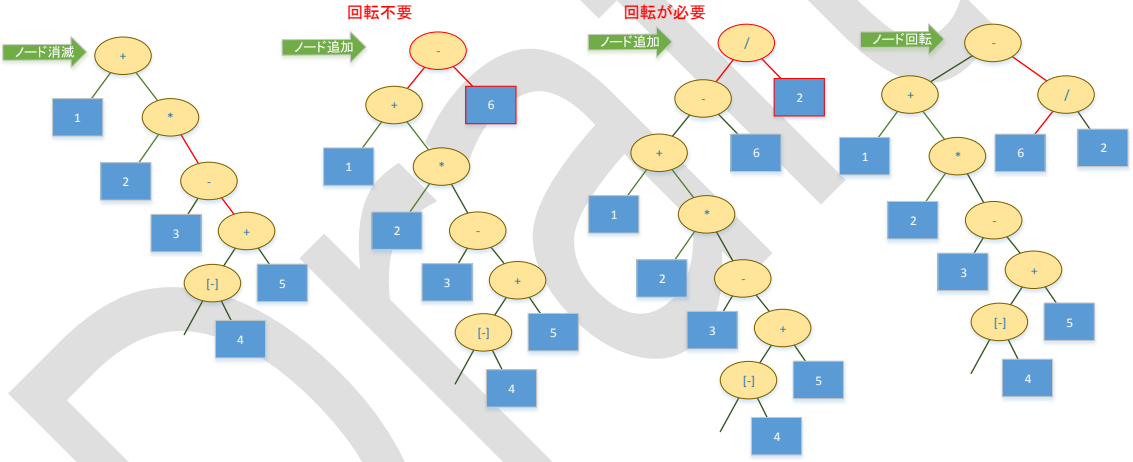
※「)」が来たら、「(」と「)」が対消滅

※同様に、「)」が来たら、「(」と「)」が対消滅



※トップノードに次の演算子を追加
※演算子の優先順位が同じなので回転不要

※トップノードに次の演算子を追加
※演算子の優先順位が異なるので回転が必要



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

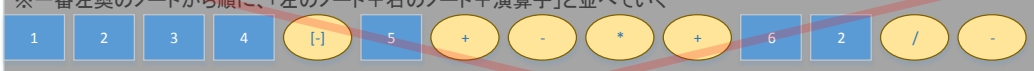
※演算子と値をスタックに積んでいき、値が二つ揃った時点で直前の演算子の計算を実行してスタックに積み直す。
単項演算子の場合は値が一つの時にそれを行う。これを繰り返すと最後に計算結果がスタックに残る。

試算: -,+,1,*,2,-,3,+,[-],4 ⇒ -,+,1,*,2,-,3,+,-4 ⇒ -,+,1,*,2,-,3,+,-4,5 ⇒ -,+,1,*,2,-,3,1 ⇒ -,+,1,*,2,-,3,1 ⇒ -,+,1,*,2,2 ⇒ -,+,1,*,2,2 ⇒ -,+,1,4
⇒ -,+,1,4 ⇒ -,5 ⇒ -,5,/,6,2 ⇒ -,5,3 ⇒ -,5,3 ⇒ 2

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



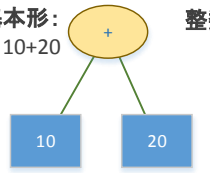
※値をスタックに積んでいき、演算子が来たら二つの値をスタックから取り出して計算し、計算結果をスタックに積み直す。
単項演算子の場合は一つの値に対してそれを行う。これを繰り返すと最後に計算結果がスタックに残る。

試算: 1,2,3,4,[-] ⇒ 1,2,3,-4 ⇒ 1,2,3,-4,5,+ ⇒ 1,2,3,1 ⇒ 1,2,3,1,- ⇒ 1,2,2 ⇒ 1,2,2,* ⇒ 1,4 ⇒ 1,4,+ ⇒ 5 ⇒ 5,6,2,/ ⇒ 5,3 ⇒ 5,3,- ⇒ 2

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

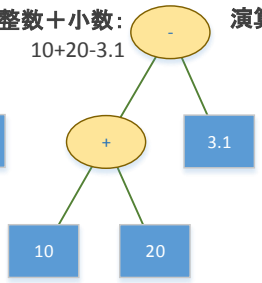
≪ゲームデータ内計算式の抽象構文木による解析とデータ化≫

基本形：
10+20



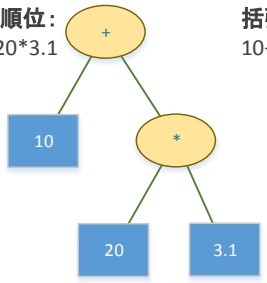
①+,10,20
②[3]+,0,1:i10,i20
③[3]+,0,1:i,i:10,20

整数＋小数：
10+20-3.1



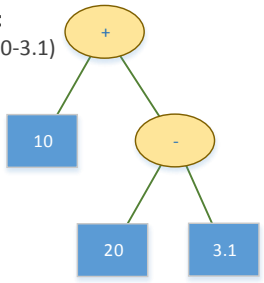
①-,+,10,20,3.1
②[5]-,+,0,1,2:i10,i20,f3.1
③[5]-,+,0,1,2:i,i,f:10,20,3.1

演算子優先順位：
10+20*3.1



①+,10,*,20,3.1
②[5]+,0,*,1,2,+:i10,i20,f3.1
③[5]+,0,*,1,2:i,I,f:10,20,3.1

括弧：
10+(20-3.1)



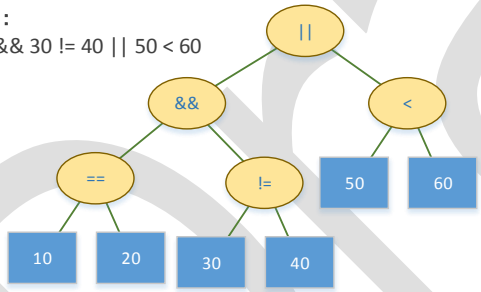
①+,10,-,20,3.1
②[5]+,0,-,1,2:i10,i20,f3.1
③[5]+,0,-,1,2,:i,I,f:10,20,3.1

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

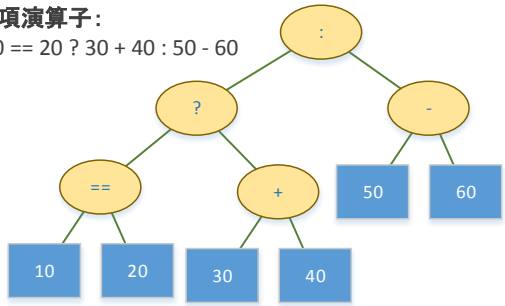
論理演算：
10 == 20 && 30 != 40 || 50 < 60



①||,&&,==,10,20,!=,30,40,<,50,60
②[11]||,&&,==,0,1,!=,2,3,<,4,5:i10,i20,i30,i40,i50
③[11]||,&&,==,0,1,!=,2,3,<,4,5:i,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。

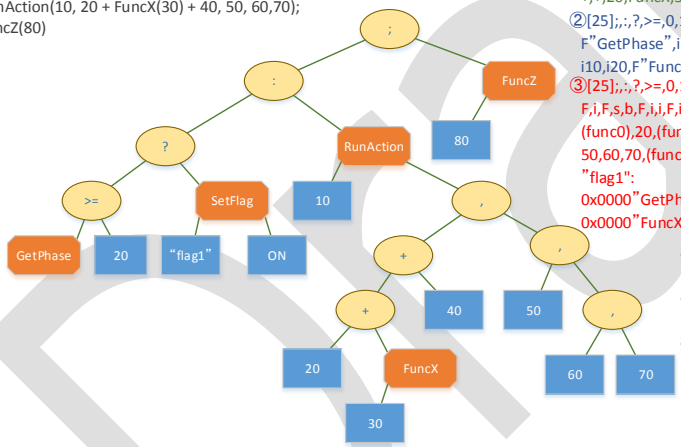
3項演算子：
10 == 20 ? 30 + 40 : 50 - 60



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,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（パラメーター一つ）
- 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）

▼ 対応演算子

≪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
		--	a--	後置減分演算子	非対応
2		++	++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	
		sizeof	sizeof a	sizeof演算子	非対応
3		()	(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
		|=	a |= b	ビット単位のOR代入演算子	非対応
16	25	,	a , b	コンマ演算子	※「, 」と「; 」は、同じ扱いとする 列挙された値全部が返され、関数の引数などになる
	26	;	a ; b	セミコロン（ターミネータ）	
	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。

※↑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。

なお、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。

■ 処理仕様

▼ プリプロセッサ

MinGW(GCC)

http://sourceforge.net/projects/mingw/files/Installer/

▼ データ変換ツール

▼ 実機（取り込み処理）

■ 環境の改善

▼ SCons の利用

SCons

Python

以上

■ 索引

ゲームデータ仕様

以　上