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・目的

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。

■ 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

- 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 で管理するデータは、テキストファイルに変換して扱う。なお、この要件については本書の範疇外とし、専用の仕様として別途策定する。
- 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（パーサー）を作成する。
- バイナリデータを実機に取り込む処理を作成する。
- テキストのファイルのパーサーは、実機上では扱わない。

▼ 要件定義

- 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。

- 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。

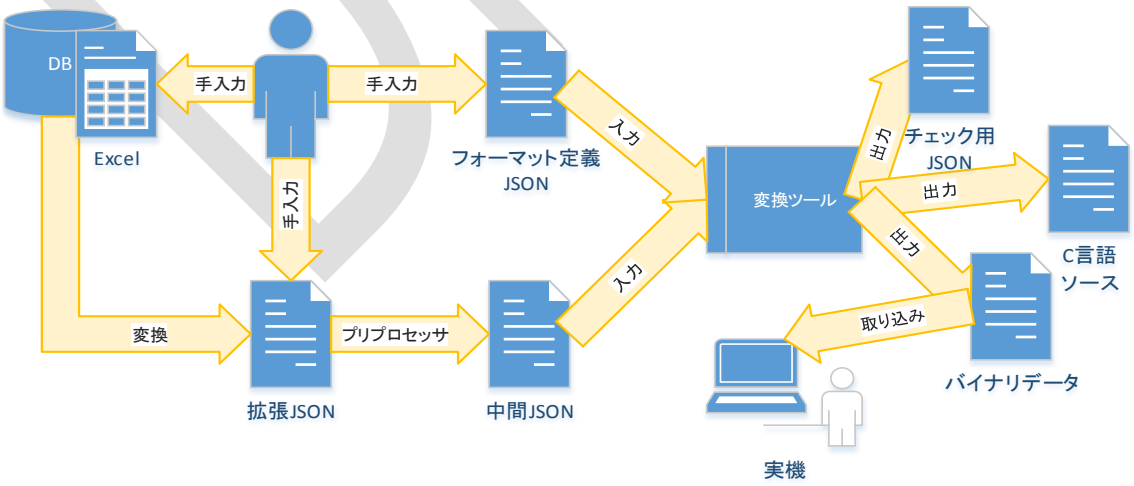
・ バイナリデータの構造が変更され、実機上の構造とずれた場合、取り込み処理は自動的にその事を検出し、項目毎のデータ取り込み処理を行う。要は、データ構造が変わっても、プログラムが極力正常に動作するようにする。

■ 仕様概略

▼ 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
・ 拡張 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
- フォーマット定義 JSON ... データ変換・ルール・バージョン定義
- 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

この仕様書では扱わない。別途仕様を策定。

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。Excel のインポート／エクスポートでデータ編集。独自のバージョン管理とロック機構を備えた仕組みを構築。ツールのインターフェースは Web 形式。

なお、DB に記録される JSON は、一切の拡張仕様が使用できないが、直接テキストを編集しないのでほぼ問題ない。特殊関数の使用が必要な箇所は、フォーマット定義 JSON 側に関数の使用を定義する。

DB 内に記録されるデータ構造は、Excel 形式への変換に特化しているため、実機向けのデータ構造と異なる。一つの DB データから、複数の実機向けデータを出力する場合もある。

▼ 拡張 JSON

基本 JSON 仕様：

▼ 中間 JSON

フォーマットは拡張 JSON とほぼ同じ。コメントが除去され、#include 文と#define 文が展開されているだけの違い。エラー出力用に、#line 文が挿入されている。MinGW(GCC)

による C++言語プリプロセッサの使用を想定。

▼ フォーマット定義 JSON

▼ バイナリデータ

▼ C 言語ソース　※オプションで生成

▼ 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。バイナリデータの構造に合わせた構造。完全な JSON 仕様を踏襲したフォーマットのため、データの二次利用も可能。

■ 関数仕様

▼ crc(“str”)

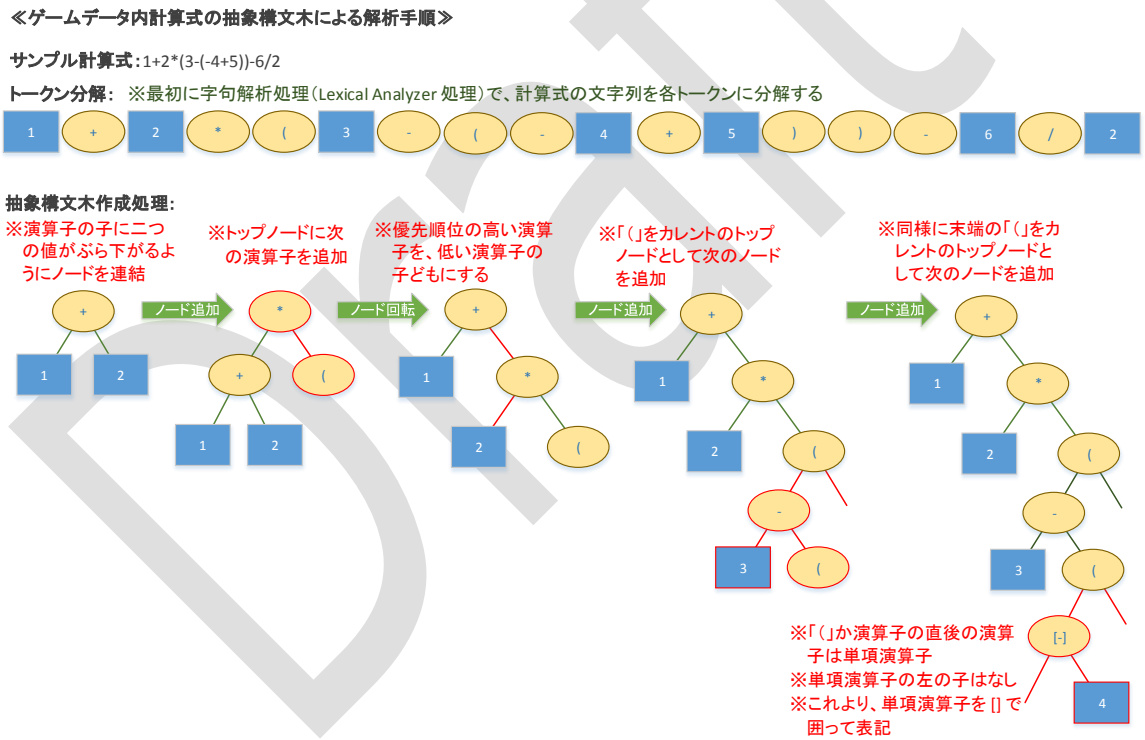
▼ crcs(“str”)

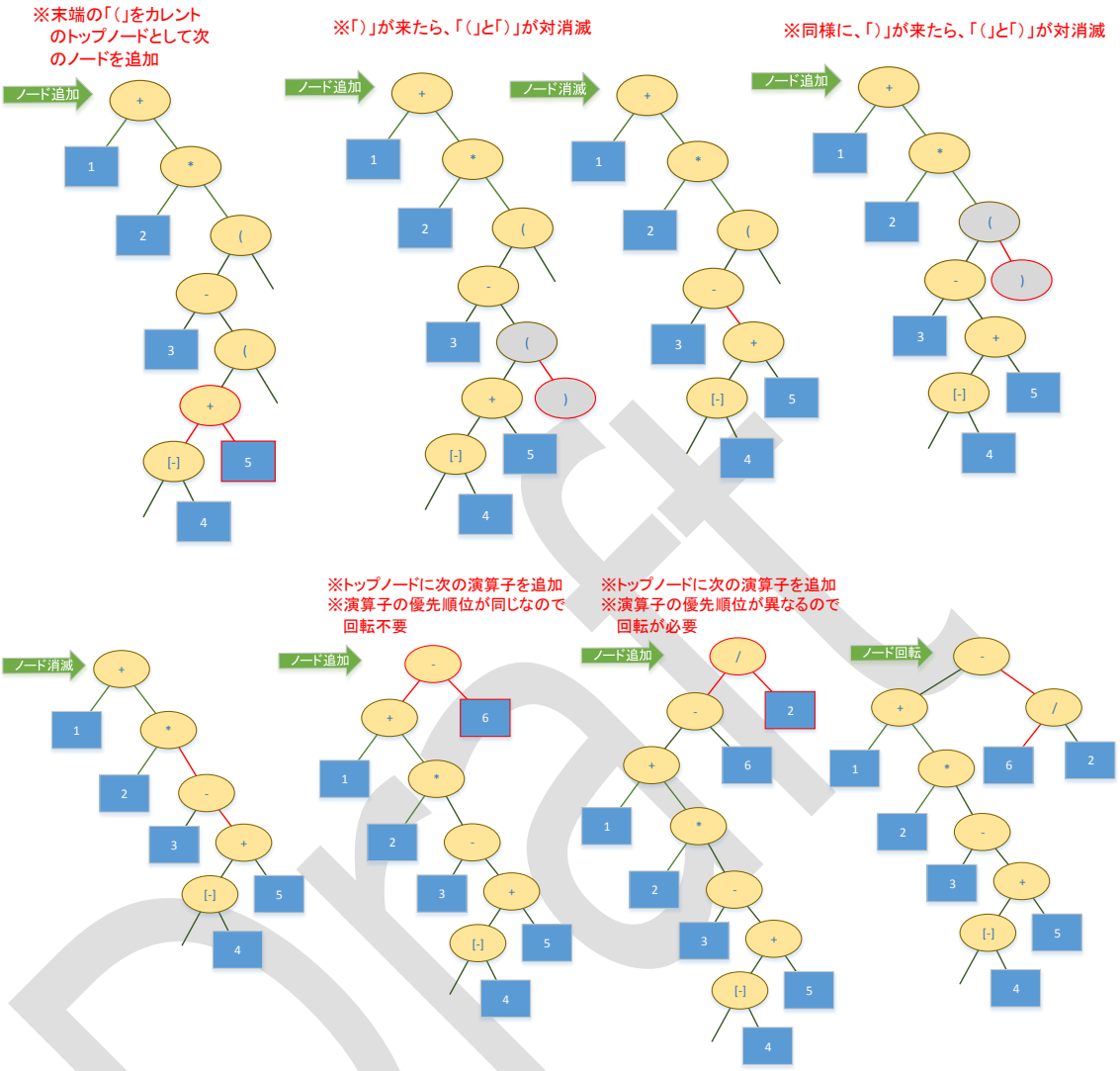
▼ expr(“expression”)

■ ゲームデータ内計算式仕様

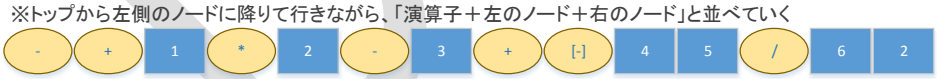
expr(“expression”)で計算される計算式のデータ変換仕様を示す。

▼ 計算式解析





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



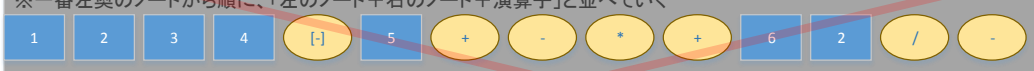
※演算子と値をスタックに積んでいき、値が二つ揃った時点で演算子の計算を実行してスタックに積み直す。
単項演算子の場合は値が一つの時にそれを行う。これを繰り返すと最後に計算結果がスタックに残る。

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。こちらの方が少ないスタックサイズで処理できるが、例えば「&&の左値が0なら右値（の関数）を評価しない」といった、評価を省略して処理を効率化することが難しいので、「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



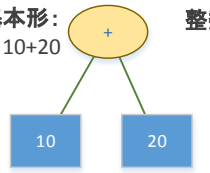
※値をスタックに積んでいき、演算子が来たら二つの値をスタックから取り出して計算し、計算結果をスタックに積み直す。
単項演算子の場合は一つの値に対してそれを行う。これを繰り返すと最後に計算結果がスタックに残る。

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

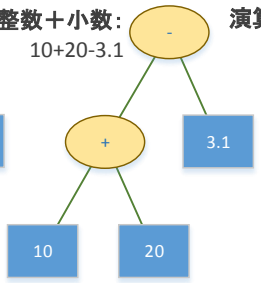
≪ゲームデータ内計算式の抽象構文木による解析とデータ化≫

基本形：
10+20



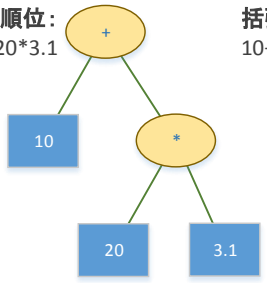
①+,10,20
②[3]+,0,1:i10,i20
③[3]+,0,1:i,i:10,20

整数＋小数：
10+20-3.1



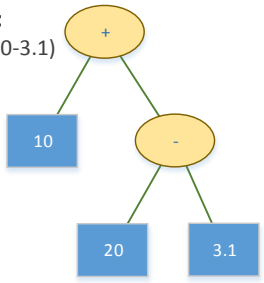
①-,+,10,20,3.1
②[5]-,+,0,1,2:i10,i20,f3.1
③[5]-,+,0,1,2:i,i,f:10,20,3.1

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)



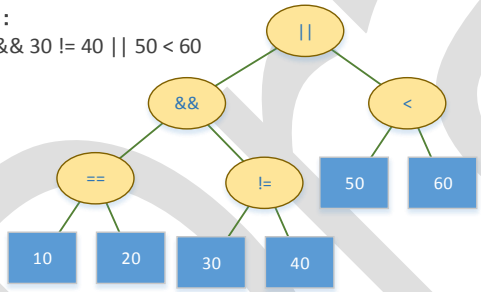
①+,10,-,20,3.1
②[5]+,0,-,1,2:i10,i20,f3.1
③[5]+,0,-,1,2,:i,I,f:10,20,3.1

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

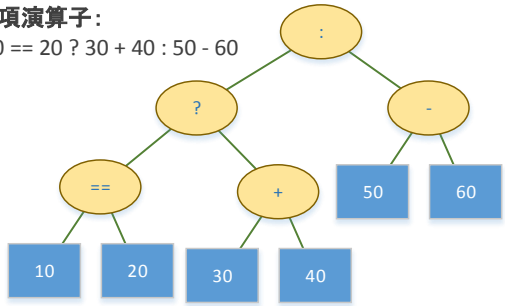
論理演算：
10 == 20 && 30 != 40 || 50 < 60



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。

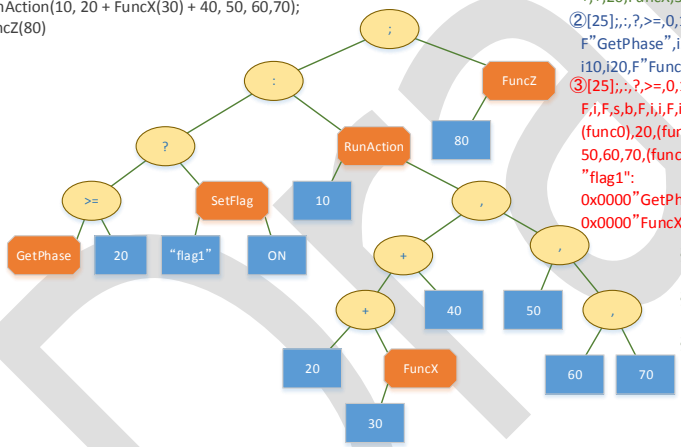
3項演算子：
10 == 20 ? 30 + 40 : 50 - 60



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。

■ 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