**数式処理** *J.JSSAC (1999)*
*Vol. 7, No. 2, pp. 2 - 17*

**解説**

# Open asir 入門

小原 功任　　高山 信毅

神戸大学大学院自然科学研究科*　　神戸大学理学部数学教室†

野呂 正行

富士通研究所 HPC 研究センター‡

(Received 1997/12/1)

## 1 Open asir とは

Open xxx は, 同じタイプまたは異なるタイプの数学プロセス間のメッセージのやりとりの規約である. 開発の動機は, 手作り (または研究的な) 数学ソフトの相互乗り入れの実現および分散計算の実装が第一であったが, もちろん数学ソフト間だけでなく, ワープロソフトや, インタラクティブな数学本, さらには数学デジタル博物館用のソフトがこの規約に従い, 数学ソフトを呼び出すことなどにも利用できる. さらに, だれでもこのプロジェクトに参加できるように規約を拡張していくための規約も定めるものとする.

設計の方針として, (1) 単純 (2) 拡張性 (3) 実装の簡便さ (4) 実用性, に重きをおいている. Open xxx はなにも考えずに簡単に接続できるシステムを作ろう, というまで野心的ではない. 数学的なオブジェクトは一筋縄ではいかないし, 完全な統一規格をつくるというのは気が遠くなる仕事である. そのかわり, 今よりすこしだけこのようなデータ交換や分散システム構築の仕事を楽にしたいというのがささやかな第 1 目標である.

数学的なオブジェクトをどのように表現するのか, どう伝えるのかを考えることは決してつまらない問題ではない. このような問題は, 新しい数学記号を創造する問題と似ているかもしれない. 我々は, 数字を 0 を含んだ 10 進数で表記し, 微分を $dx$ と書き, 写像を $\longrightarrow$ であらわす. これらの記号法からどれだけ多くの利益を得ているか, 思いをはせて欲しい. また,

*ohara@math.kobe-u.ac.jp
†taka@math.kobe-u.ac.jp
‡noro@para.flab.fujitsu.co.jp

Mathematica や Maple といった統合ソフトを, 我々自身の手でつくっていくための基礎でもある.

さて, Open asir は Open xxx 規約を実装した asir サーバである. Open xxx 規約にしたがったメッセージを Open asir サーバに送ることにより, asir の全機能を利用することができる. 現在, Open xxx 規約に基づく計算サーバとしては, Linux 上で動作する Open asir と Open sm1 (kan/sm1) がバイナリで提供されている. ユーザは自分でクライアントを C, Java, Lisp (さらに asir, sm1) などの言語で作成して, Open asir を利用することができる. また, MathLink との変換ソフトを書けば, Mathematica のサーバになることも可能である.

Open asir の基本的な計算のモデルはメッセージの交換である. Open asir サーバに計算させたいものをメッセージとして送り, 次いで計算結果をメッセージとして受け取るという形で計算は進行する. 通常, メッセージのやりとりには大きく分けて, 同期的なやりとりと非同期的なやりとりが考えられる. ここで同期的なやりとりとは, 双方のプロセスがあらかじめタイミングを決めてメッセージのやりとりをすることをいう. このとき, それぞれのプロセスは相手からのメッセージが届くまでの間はブロックされる. 非同期的なやりとりとは, それぞれのプロセスがメッセージを送った後, 相手の状態にかかわらず, すぐに復帰するようなやりとりのことである. 現在のサーバは, 非同期的なメッセージのやりとりのみ実装している.

このモデルを現在は, TCP/IP とソケットを用いて実装しているが, Open xxx 規約自身は TCP/IP に縛られているわけではないので, もちろん, MPI や PVM などの上にこのモデルを実装してもいいし, ファイルよりメッセージを読み込み結果をファイルへ書き出してもよい. この Open xxx 規約は文献 [1] で定義されている. [1] の TEX ソースは

`http://www.math.kobe-u.ac.jp/openxxx/`

から入手することが出来る.

なお, Open xxx 規約は現在研究中のプロトコルであり, 規約, 実装の変更が随時おこなわれている. また, 同様の試みとして, OpenMath

`http://www.openmath.org/`

がある. こちらとの関係も考慮して行きたい.

この文書は Open asir クライアントの作成を通じて, Open xxx 規約 (プロトコル) の考え方を説明することをその目的とするため, 必要に応じて C 言語によるプログラムのソースを例示する. この文書で, 引用されているクライアントの完全なソース (このソースは, Redhat Linux 4.2 または FreeBSD 2.2.6 でコンパイル可能である) は, [1] と同じページより入手することが出来る. Open sm1 サーバも同じページより入手できる. また, Linux 版の Open asir サーバ実験版も同じページにおくよう努力したい. 問題点などの指摘は, `openxxx@math.kobe-u.ac.jp` へメールをおくってほしい.

## 2 Open asir で扱われるデータ

Open asir とやりとりするメッセージは, データあるいは命令 (コマンド) である. これについてはあとでまた説明することにして, この節ではこのデータの形式 (フォーマット) を説明しよう. Open asir で扱われるデータは Common Math Object (CMO) と呼ばれる形式のデータである. CMO はデータ型を示すタグとデータ本体より構成されている. 例えば 32bit 整数値 1234 は, (CMO_INT32, 1234) なる CMO として表現される. この表記方法は CMO expression と呼ばれている. CMO expression は, CMO の実体の構造を人間が読みやすいような形式で表現したものであり, マシン語 (バイト列) に対応するアセンブリ言語または高級言語のようなものだと思ってもよい. ここで CMO_INT32 がデータ型を表すタグである. おなじように, 長さが 6 の文字列 `"abcdef"` は (CMO_STRING, 6, "abcdef") という CMO で表現する.

これらのデータ型を表すタグ CMO_INT32, CMO_STRING などは 32bit 整数値であり, Open xxx 規約で

```
#define CMO_NULL        1
#define CMO_INT32       2
#define CMO_DATUM       3
#define CMO_STRING      4

#define CMO_LIST        17
#define CMO_MONOMIAL32  19
#define CMO_ZZ          20
#define CMO_QQ          21
#define CMO_ZERO        22
#define CMO_DMS         23
```

などと定義されている. 現在の TCP/IP での実装では, たとえば (CMO_INT32, 1234) は

```
00 00 00 02 00 00 04 d2
```

という 16 進バイト列に変換される. さっきのマシン語のたとえをつかうとこちらがマシン語 (バイト列) に対応する.

その他のデータ型としては, リスト, 多重精度整数 (bignum), 分散表現多項式などが現在用意されている. 主要なデータ型の非形式的な定義をあげておく.
いま述べた 32bit 整数 $n$ は CMObject としては Integer32 と呼ばれ,

| int32 CMO_INT32 | int32 $n$ |
|---|---|

なる形で表現する. ここで, 表の各項目は, | データ型 | データ | なる形で表現している.
長さ $n$ の 文字列 s は, CMObject としては, Cstring 型とよばれ

| int32 CMO_STRING | int32 $n$ | byte s[0] | byte s[1] | $\cdots$ | byte s[$n-1$] |
|---|---|---|---|---|---|

と表現する. C 言語で普通用いられる, 文字列の終端の 0 は文字列には含めない.

長さ $m$ のリストは

| int32 CMO_LIST | int32 $m$ | CMObject ob[0] | $\cdots$ | CMObject ob[$m-1$] |
|---|---|---|---|---|

で表現する.

各データ型のさらに詳しい定義, および形式的記述法は, [1] を参照してほしい. CMO の定義を正確かつ簡単に行うには形式的記述法が不可欠であることを注意しておく.

# 3 Open asir サーバとの通信

現在のサーバの実装では, サーバ・クライアント間の通信はソケットを用いて, ストリーム型で行う. 現在実装されているサーバは, ふたつのプロセスに別れていて, 実際の計算を行うプロセスと制御命令を受け取るプロセスからなる. 計算の中断, サーバの終了などの制御命令はコントロールプロセスに送ることになる. この文書では, 特に断らない場合には「サーバへの送信」は実際の計算を行うプロセスへの送信のことを指す.

現在実装されているサーバがふたつのプロセスに分れているのは以下の理由による. 我々は, クライアントがサーバに任意のタイミングで計算の中断を指示できる機能が必要であると考える. この機能は, 現在の実装では, コントロールプロセスが計算プロセスに UNIX のシグナルを送ることで実現している. Open xxx 規約はネットワーク透過性を目指しているが, サーバとクライアントが異なるマシンで動作している場合には, クライアントが直接シグナルを送ることは困難である. しかしながら, 計算プロセスと同じマシン上にコントロールプロセスがあってクライアントと別の通信路で結ばれていれば, クライアントからのメッセージをコントロールプロセスが受取り, かわりに計算プロセスへのシグナルを送ることはできる. 我々は, このような構成で信頼性の高いメッセージのやりとりを実現しているのである. 実際にどのような手順で計算の中断を実現しているかについては Section 5 で詳しく説明する.

さて, 最初にクライアントは両方のプロセスとの通信路を確保する必要がある. これは, コントロールプロセス, 計算プロセスの順に通信路を確保しなければならない. 例えば以下のような関数を実行することになる.

```
#define PORT_CONTROL  1200
#define PORT_STREAM   1300

int fdControl, fdStream;

int mysocketOpen(const char* hostname, int portnum)
{
  struct sockaddr_in serv;
  struct hostent *host;
  int s;

  bzero(&serv, sizeof(serv));
```

```
  serv.sin_family = AF_INET;
  serv.sin_port = htons(portnum);
  host = gethostbyname(hostname);
  bcopy(host->h_addr, (char *)&serv.sin_addr, host->h_length);

  s = socket(AF_INET, SOCK_STREAM, 0);
  connect(s, (struct sockaddr *)&serv, sizeof(serv));
  return s;
}

void ox_start(char* servername)
{
  fdControl = mysocketOpen(servername, PORT_CONTROL);
  fdStream  = mysocketOpen(servername, PORT_STREAM);
}

int main()
{
  ...
  ox_start("localhost");
  ...
}
```

通信路が確保されたら, メッセージの交換が開始できるわけであるが, ここで, Open xxx 規約における 32bit 整数の扱い (CMO_INT32 ではない) について注意しておこう. 一般的に注意しなければならない点は, 負数の扱いとバイトオーダであると思われる. 現在実装されているサーバは負数は 2 の補数表現されているものと見做して動作する. ただしこれは旧版の [1] には明記されていない. 次にサーバとの通信において, 用いるべきバイトオーダに関してであるが, Open xxx 規約では通信時にはネットワークバイトオーダを用いることと定められている. よって, C 言語でプログラムを作成する場合には, 適時, 変換する必要がある. 例えば 32bit 整数を送信する場合には以下のような関数を用いればよい.

```
int send_int32(int fd, int integer32)
{
  integer32 = htonl(integer32);
  return write(fd, &integer32, sizeof(int));
}
```

# 4 メッセージ

Open asir はサーバとクライアントの間でメッセージを交換するという形で計算を進行させる. Open xxx 規約では, 数学プロセスはスタックマシンであると定義されている. サーバはクライアントから受け取ったメッセージがデータであればスタックに積み, 命令であればスタックマシンとしてその命令を実行して結果をスタックに積む. Open asir における全てのメッセージにはメッセージの種類を表す情報 (32bit 整数値で表されるタグ) と, シリアル番号が与えられる. Open xxx 規約では, 以下の形式のメッセージが定義されている.

| | |
|---|---|
| (OX_COMMAND, *serial number*, *SMObject*) | 命令 |
| (OX_DATA, *serial number*, *CMObject*) | データ |
| (OX_SYNC_BALL, *serial number*) | 同期用メッセージ |

サーバ・クライアント間で交換されるメッセージは全て上のいずれかの形式をとらなければならない. これらのメッセージの種類を表すタグは次のように定義されている.

```
#define OX_COMMAND                    513
#define OX_DATA                       514
#define OX_SYNC_BALL                  515
```

メッセージのシリアル番号も 32bit 整数で与えられる. これはクライアントまたはサーバが自由につけてよい. シリアル番号はエラー処理に主に利用される. サーバはエラーを発生したメッセージのシリアル番号をエラーメッセージに含めてもどす. 上のメッセージの定義で, SMObject は, スタックマシンへの命令であり, CMObject は, CMO に属するデータである. スタックマシンへの命令は 32bit 整数値で与えられる. [1] では例えば以下のような命令を定義している.

```
#define SM_popCMO                     262
#define SM_popString                  263
#define SM_mathcap                    264
#define SM_pops                       265
#define SM_setName                    266
#define SM_evalName                   267
#define SM_executeStringByLocalParser 268
#define SM_executeFunction            269

#define SM_control_kill               1024
#define SM_control_reset_connection   1030
```

Open xxx のスタックマシンは二種類の異なる機能をもつと言える. これはマシン語のたとえを使うと, いくつかの異なるマシン語のエミュレーションモードをもつ CPU のようなものである. モード (1) では, ローカル言語 (Open asir の場合は asir) の文法にしたがった文字列を受け取り, 計算した結果をローカル言語で記述された文字列でもどす. この計算は

命令 SM_executeStringByLocalParser を用いて行われる. モード (2) では, Open xxx で定められたスタックマシンとして動作する. 基本的に, CMO をスタックに積み, 命令を受け取ると演算を行う.

この節ではモード (1) を用いた計算の例をあげる. ユーザが, Open asir に “diff((x+2*y)^2,x);” を評価させたいときは, サーバにメッセージ

(OX_DATA, *serial number*, (CMO_STRING, 18, ”diff((x+2*y)^2,x);”))

(OX_COMMAND, *serial number*, SM_executeStringByLocalParser)

を順に送る. 最初のメッセージで文字列 “diff((x+2*y)^2,x);” がスタックに積まれ, 次のメッセージでスタックの最上位の “diff((x+2*y)^2,x);” が Open asir サーバによって実行される. 結果は再びスタックに積まれる.

これを実行するクライアントのプログラムは C 言語で表すと次のようになる.

```
/* (OX_tag, serial number) を送信する */
int send_ox_tag(int fd, int tag)
{
  static int serial_number = 0;
  send_int32(fd, tag);
  return send_int32(fd, serial_number++);
}

void send_ox_cmo_string(int fd, const char* str)
{
  int len;
  send_ox_tag(fd, OX_DATA);   /* OX タグとシリアル番号を送信 */
  send_int32(fd, CMO_STRING); /* CMO_STRING 形式の文字列を送る */
  len = strlen(str);
  send_int32(fd, len);
  write(fd, str, len);
}

void send_ox_command(int fd, int sm_command)
{
  send_ox_tag(fd, OX_COMMAND); /* OX タグとシリアル番号を送信 */
  send_int32(fd, sm_command);
}

int executeStringByLocalParser(const char* str)
{
```

```
  send_ox_cmo_string(fdStream, str);
  send_ox_command(fdStream, SM_executeStringByLocalParser);
}

int main()
{
  ...
  executeStringByLocalParser("diff((x+2*y)^2,x);");
  ...
}
```

スタックに積まれた結果を文字列として取り出すときは, SM_popString 命令を用いる. すなわち, サーバへメッセージ

(OX_COMMAND, *serial number*, SM_popString)

を送る. その後, Open asir サーバから計算結果を表すメッセージ

(OX_DATA, *serial number*, (CMO_STRING, 7, ”2*x+4*y”))

が送られてくるので, それを読めばよい.

```
typedef struct cmo_t {
  int tag;
  union {
    int integer;
    char *string;
  } u;
} cmo_t;

char *receive_cmo_string(int fd)
{
  int len = receive_int32(fd);
  char* str = malloc(len+1);
  bzero(str, len+1);
  read(fd, str, len);
  return str;
}

cmo_t receive_cmo(int fd)
{
  cmo_t m;
```

```
  m.tag = receive_int32(fd);
  switch(m.tag) {
  case CMO_STRING:
    m.u.string = receive_cmo_string(fd);
    break;
  case CMO_INT32:
    m.u.integer = receive_int32(fd);
    break;
  default:
  }
  return m;
}

/* (OX_tag, serial number) を受信する */
int receive_ox_tag(int fd)
{
  int tag = receive_int32(fd);
  receive_serial_number(fd);
  return tag;
}

char* popString()
{
  cmo_t m;

  send_ox_command(fdStream, SM_popString);
  receive_ox_tag(fdStream);   /* OX タグとシリアル番号を受信 */
  m = receive_cmo(fdStream);

  return m.u.str;
}

int main()
{
  char *result;
  ...
  result = popString();
  puts(result);
  ...
```

}

# 5 計算の中断

計算の中断は, Open xxx の設計でいちばん苦労した部分である. このアイデアは asir が本来もっていた分散計算の機能に由来する. まず基本として全てのメッセージ (OX_ ではじまるデータまたは命令のひとかたまり) をいわゆる synchronized object として扱っている. したがって, メッセージを転送しているときは転送が終了するまで割り込みは保留される.

計算の中断についてはサーバがコントロールプロセスと計算プロセスに別れて実装されていることが大きな意味をもつ. 以下, このことに留意して説明を行う.

まず, 計算の中断の依頼が行われる前には, クライアントおよびコントロールプロセスは待機状態, 計算プロセスは計算中の状態にある. 計算の中断の必要があると認められたとき, クライアントは, 待機状態にあるコントロールプロセスにメッセージ

(OX_COMMAND, *serial number*, SM_control_reset_connection)

を送信する. このメッセージの送信後, クライアントはコントロールプロセスからの通信路のみを監視し待機状態に入る. クライアントは計算プロセスからのメッセージは読まないようにする.

このメッセージを受け取ったコントロールプロセスは活動を再開し, クライアントに対しメッセージ

(OX_DATA, *serial number*, (CMO_INT32, 0))

を投げ返し, さらに計算プロセスには中断処理を開始せよとのシグナルをおくる. この時点では, クライアントはコントロールプロセスとの通信路を監視している状態, コントロールプロセスはシグナルを送った後, 待機状態に入り, 計算プロセスは計算中にシグナルを受け取った状態である. 計算プロセスはシグナルをうけとったときクリティカル区間にいればこの区間がおわるまで計算の中断をしない. とくに (Open xxx 規約の意味での) メッセージの送受信中は, クリティカル区間であると見做される.

クライアントはコントロールプロセスとの通信路のみを最初監視しているが, 上のコントロールプロセスからのメッセージを受信した後はコントロールプロセスとの通信路を監視するのは止めて, 計算プロセスからの `OX_SYNC_BALL` を待つ状態に遷移する. ある通信路において `OX_SYNC_BALL` を待つ状態とは, その通信路を通してメッセージ

(OX_SYNC_BALL, *serial number*)

を受け取るまで, その通信路からの全てのメッセージを読み飛ばす状態のことである. 計算プロセスは中断処理が終了した後に `OX_SYNC_BALL` をクライアントに送信してクライアントからの `OX_SYNC_BALL` を待つ状態に入る. 計算プロセスは `OX_SYNC_BALL` 以前のメッセージをすべて読みとばす.

クライアントは計算プロセスから OX_SYNC_BALL を受け取ると, すぐさま, メッセージ

(OX_SYNC_BALL, *serial number*)

を計算プロセスに送信し, 通常の状態に復帰する. サーバもこのメッセージを受信後, 通常の状態に復帰する. これで, 手続きは全て終了する.

我々の中断手続きの設計において注意すべきことは, この手順でクライアントが OX_SYNC_BALL を送った時点で, クライアント側からはサーバが待ち状態になっていて, かつ次のストリーム (クライアント, 計算プロセス間の通信路) に対する読み出しは, これから送るコマンドのうちサーバが最初にストリームに書き出した結果であることが保証されていると思ってよい, ということである. これには,

1. サーバの中断処理は, クライアントからの依頼 (シグナル) によって開始し, クライアントからの OX_SYNC_BALL 受領により終了する.
2. サーバの中断処理が開始してから, クライアントからの OX_SYNC_BALL が送られてくる.

という順序がなりたつことが本質的に重要である. 例えば, サーバからの OX_SYNC_BALL を待たずにクライアントが OX_SYNC_BALL を送ってしまった場合, シグナルに先着することもありえる.

サーバでの中断処理の読み出し中に, さらに中断依頼が来た場合などのクライアントからの OX_SYNC_BALL が複数来る場合に対応するために, OX_SYNC_BALL は, 通常状態では無視する必要があるので, 実際, 通常状態では OX_SYNC_BALL はサーバで無視されることには留意しなければならない. サーバはクライアントに OX_SYNC_BALL を送るが, クライアントがもう OX_SYNC_BALL を送った気でいると, 自分が送った OX_SYNC_BALL の返事をずっと待ち続けることになる.

なお, クライアントにおける中断処理のための関数は以下のようになる.

```
void ox_reset()
{
  send_ox_command(fdControl, SM_control_reset_connection);

  receive_ox_tag(fdControl);      /* OX タグとシリアル番号を受信 */
  receive_cmo(fdControl);     /* メッセージの本体を受信 */

  while(receive_ox_tag(fdStream) != OX_SYNC_BALL) {
    receive_cmo(fdStream);    /* メッセージを読み飛ばす. */
  }
  send_ox_tag(fdStream, OX_SYNC_BALL);
}

int main()
```

```
{
  ...
  ox_reset();
  ...
}
```

## 6 クライアントの終了

クライアントを終了するときには, コントロールプロセスにメッセージ

(OX_COMMAND, *serial number*, SM_control_kill)

を送る. このメッセージを受け取るとただちにサーバは終了するので, それを待ってクライアントも終了する. 現在実装されているサーバはクライアントが突然終了するなどのために通信路が閉じられると, サーバも自動的に終了するように設計されてはいるが, 明示的にコントロールプロセスにメッセージを送るべきである.

```
void ox_close()
{
  send_ox_command(fdControl, SM_control_kill);
}
```

## 7 サーバにローカルに定義された関数の実行

先にあげた計算の例では, 計算の対象を文字列として送信し, スタックマシンの命令 SM_executeStringByLocalParser を利用していた. 次にモード (2) を用いた計算の例をあげよう. モード (2) では CMO_STRING 型のオブジェクトだけでなく, 一般の CMO が直接スタックにプッシュされ, 計算処理の対象となる.

Open xxx 規約は以下のような意味でサーバの自由な拡張を許す. Open xxx 規約に従う数学プロセスでは, その数学プロセス独自の関数が存在してもよい. 例えば, asir における “igcd” などである. これを [1] では「サーバのローカル関数」と呼んでいる. Open asir では, asir マニュアルに説明されているすべての関数をモード (2) のサーバのローカル関数として実行できる. もちろん, asir などで用意されている「関数定義」を利用して新しく関数を追加してもよい.

Open xxx 規約にはスタック上の CMO データに対してサーバのローカル関数を実行するための命令 SM_executeFunction が定義されている. この節ではこの命令の使い方について説明しよう.

まず, 実行したい関数の引数となるオブジェクトを「右から」順にサーバに送信すると, サーバは順にそれらのオブジェクトをスタックに積む. 次いで, 引数の個数を CMO_INT32 型のオブジェクトでサーバに送信すると, サーバはスタックに積む. 次に, 実行したいと思っ

ている関数の名前を CMO_STRING 型で送信すると, サーバは再びスタックに積む. 最後に命令 SM_executeFunction を送信すると, サーバはスタックの最上位に積まれている「関数」を実行する. 関数は引数をスタックからポップして実行し, サーバはその実行結果をスタックに積む.

例をあげよう. SM_executeFunction を用いて, asir の文

print(”Hello World.”);

を実行するには次の手順をとる.

まず, 次の順にメッセージを送信する.

(OX_DATA, *serial number*, (CMO_STRING, 12, ”Hello World.”))

(OX_DATA, *serial number*, (CMO_INT32, 1))

(OX_DATA, *serial number*, (CMO_STRING, 5, ”print”))

これらはサーバによって順にスタックに積まれる. 次いで命令

(OX_COMMAND, *serial number*, SM_executeFunction)

を送信すると関数 “print” が実行され, サーバは “Hello World.” を画面に出力する. 実行結果として (CMO_NULL) がスタックに積まれる.

# 8 CMO ZZ

この節では Open xxx 規約における任意の大きさの整数 (bignum) の扱いについて説明する. 多重精度整数についての一般論については Knuth [2] に詳しい. Open xxx 規約における多重精度整数を表すデータ型 CMO_ZZ は GNU MP ライブラリなどを参考にして設計されていて, 符号付き絶対値表現を用いている. CMO_ZZ は次の形式をとると [1] で定義されている.

| int32 CMO_ZZ | int32 $f$ | byte $b_1$ | $\cdots$ | byte $b_n$ |
|---|---|---|---|---|

$f$ は 32bit 整数であるが, 正数とは限らない. $b_1, \ldots, b_n$ は 1 バイト符号なし整数である. このバイト列の長さ $n$ は絶対値 $|f|$ と一致しなければならない. この CMO の符号は $f$ の符号で定める. 前述したように, 32bit 整数の負数は 2 の補数表現で表される.

Open xxx 規約では上の CMO は以下の整数を意味する.

$$\mathrm{sgn}(f) \times (b_1 R^{n-1} + b_2 R^{n-2} + \cdots + b_{n-1} R + b_n).$$

ここで $R = 2^8$ である. 例えば, 整数 14 は CMO_ZZ で表わすと,

(CMO_ZZ, 1, e)

となり, これをバイト列で表すと以下のようになる.

```
00 00 00 14 00 00 00 01 0e
```

あるいはゲタをはかせて CMO_ZZ で

(CMO_ZZ, 4, 0, 0, 0, e),

と表してもよい. これはバイト列では

```
00 00 00 14 00 00 00 04 00 00 00 0e
```

となる. 後者の表現の方が C 言語でクライアントを作成するのには, 簡単であろう.

さて, 前節で説明した命令 SM_executeFunction を利用して, 整数 14 と 22 の最大公約数を求めることを考えよう. これには asir の関数 igcd が利用できる.

まず, CMO_ZZ で表した二つの整数, 引数の数 (ここでは 2) と関数名 “igcd” を順にサーバに送信する.

(OX_DATA, *serial number*, (CMO_ZZ, 4, 0, 0, 0, e))

(OX_DATA, *serial number*, (CMO_ZZ, 4, 0, 0, 0, 16))

(OX_DATA, *serial number*, (CMO_INT32, 1))

(OX_DATA, *serial number*, (CMO_STRING, 4, ”igcd”))

最後に命令 SM_executeFunction をサーバに送信する.

(OX_COMMAND, *serial number*, SM_executeFunction)

スタックの最上位には計算結果が積まれるが, 命令 SM_popString をサーバに送ると, それが文字列に変換されてクライアントに返される. なお CMO 形式でデータを受け取るには, 命令 SM_popCMO をサーバに送る.

一連の操作を C 言語で実装すれば以下のようになるだろう.

```
typedef struct {
  int size;
  unsigned char *mp_d;
} cmo_zz_t;


cmo_zz_t cmo_zz_set_si(int integer)
{
  cmo_zz_t zz;
  zz.size = (integer < 0)? -sizeof(int): sizeof(int);
  integer = htonl(integer);
  zz.mp_d = malloc(sizeof(int));
  bcopy(&integer, zz.mp_d, sizeof(int));
  return zz;
}
```

```
void send_ox_cmo_zz(int fd, cmo_zz_t zz)
{
  int len = (zz.size < 0)? -zz.size: zz.size;
  send_ox_tag(fd, OX_DATA); /* OX タグとシリアル番号を送信 */
  send_int32(fd, CMO_ZZ);
  send_int32(fd, zz.size);
  write(fd, zz.mp_d, len);
}

int main()
{

  ...
  send_ox_cmo_zz(fdStream, cmo_zz_set_si(14));
  send_ox_cmo_zz(fdStream, cmo_zz_set_si(22));
  send_ox_cmo_integer(fdStream, 2);  /* number of arguments */
  send_ox_cmo_string(fdStream, "igcd");

  send_ox_command(fdStream, SM_executeFunction);

  fprintf(stderr, "igcd(%d, %d) == %s.\n", 14, 22, ox_popString(fdStream));
  ...

}
```

# 9 エラー処理

この節ではエラー処理について説明する. Open xxx 規約ではエラーを検出したとき, サーバは CMO_ERROR2 型のオブジェクトをスタックに積むと定めている. CMO_ERROR2 型のオブジェクトはエラーを表すために用意されたオブジェクトであり, どのようなエラーであるか, どのメッセージで発生したのかの情報を含む. CMO_ERROR2 という名称は歴史的な理由によるものである. このオブジェクトは次のように定義されている.

| int32 CMO_ERROR2 | CMO_LIST error |
|---|---|

error は CMO_LIST 型のオブジェクトで, リストの各要素がエラーについての情報になっている. 特にリストの先頭の要素は Integer32 型の CMObject で, その値はエラーが発生したメッセージのシリアル番号を Integer32 で表現したものでなければならない. このリスト error の各要素についての詳細は [1] を参照してほしい.

さてクライアントはエラーが起こったことを知らないわけであるが, それは次のようにして検出できる. まず, 命令 SM_popCMO をサーバに送る. すると, サーバはスタックの最上位をクライアントに送信する. そこでクライアントはその CMO のデータ型を見て, エラーが起こったことを確認できる.

## 10 補足

この解説を執筆中に, Mathematical Sciences Research Institute (Berkeley) で, Parallel Symbolic Computation なるワークショップがあった. 関係する興味深い発表が沢山あったので, 発表に関係ある URL とキーワードをメモとして加えておく.

1. `www.openmath.org` (OpenMath)
2. `www.w3.org` (math web)
3. `posso.lip6.fr/~jcf` (Faugere $F_4$, parallel GB algorithm)
4. `www.cs.berkeley.edu/~yelick` (multipol)
5. `www.mupad.de` (parallel muPAD)
6. `www.math.ncsu.edu/~kaltofen` (open math virtual machine)
7. `norma.nkkhef.nl/~t68/summer` (integral)
8. `www.inf.ethz.ch/personal/mannhart` ($\Pi^{it}$)
9. `www.nag.co.uk` (useful links to computer algebra projects including Frisco project)
10. `www.can.nl` (useful links to computer algebra projects)
11. `www.cecm.sfu.ca` (useful links to computer algebra projects)
12. `www.icot.or.jp` (KLIC, Fujise-Murao's project)
13. Parallel mathematica.

## 参 考 文 献

[1] 野呂正行, 高山信毅. Open xxx の設計と実装, xxx = asir,kan, 1998/06/16

[2] D. E. Knuth. 準数値算法/算術演算, サイエンス社, 1986.