Risa/Asir Internals

Risa/Asir 内部仕様
Version 20000530
Jan 2013

by Masayuki Noro

1 Risa/Asir の構成

1.1 engine

engine は, Risa の内部形式を入力, 出力とする関数の集まりである. 主な関数として, 一つの型に対する四則演算のセットがある. また, 例えば, 多項式に対する既約因子分解など, ある型に対して固有の演算が用意されている場合がある. 他に, 整数に対する基数変換 (10 進数と 16 進数の変換など) などが含まれる.

1.2 parser

ユーザからの入力は一般に可読な文字列として与えられる. 一方で engine の受け取れる入力は, 後で述べるような C 言語の構造体で表現された内部形式である. よって, ユーザと対話を行うアプリケーションとしては, ユーザからの文字列入力を内部形式に変換する必要がある. Asir においては, これを

1. 文字列を中間言語に変換 (parser)
2. 中間言語で書かれた命令を実行して内部表現を得る (interpreter)

の 2 段階で行っている. parser においては, まず文字列を一まとまりの部分文字列 (token) の列として切り分ける. この切り分けを字句解析と呼ぶ. 各 token は属性を付与される. 例えば 123+456 は 123, +, 456 の 3 つの token に分解され, 123, 456 は式, + は演算子の属性を付与される. parser は token を順に読んで, 自らが持つ文法定義と参照しながら tree を作っていく. この tree の仕様が中間言語である. これらの操作および中間言語については後に詳しく述べる.

1.3 interpreter

parser で得られた tree は, 入力文字列を中間言語で書き表したものである. interpreter は tree の根の部分から順に解釈し, 必要があれば engine の関数を呼び出しながら, 最終的に内部形式であるデータを生成する. tree の各 node は

1. 識別子
2. 引数配列

なる形をしている. interpreter は, 識別子をみて実行すべき仕事および引数の意味を知る. 必要があれば, それ自身中間言語 tree で表現されている引数を interpreter で内部形式に変換しながら, 識別子に対応する関数を呼び出す. interpreter についても後で詳述する.

2 データ型

この章では, 様々な object の定義, 生成, 演算について解説する. 各々の object のデータ構造は, C の構造体として表現される. しかし, 実際に函数に引数として渡されるのは, それらへのポインタである. 一般に, 構造体そのものは, 'o' で始まる名前をもち, それへのポインタは, 構造体の名前から 'o' をとったものとして typedef されている. すべてのオブジェクトに対し, 0 には NULL ポインタを対応させることとする. すなわち, 0 でないポインタは, 0 でない object を表す.

2.1 Risa object

```
struct oObj {           object の共通部分
    short id;           識別子
    short pad;
};

typedef struct oObj *Obj;
```

Risa object とは構造体 oObj を先頭部分に持つ object の総称である. Risa object は Asir において独立した object として変数の値となり得る. Risa object は id により識別される. 現在, 次のような種類の Risa object が定義されている.

0 **0**

O_N = 1 **number; 数**

O_P = 2 **polynomial; 多項式**（数でない）

O_R = 3 **rational expression; 有理式**（多項式でない）

O_LIST = 4 **list**

O_VECT = 5 **vector**

O_MAT = 6 **matrix**

O_STR = 7 **character string; 文字列**

O_COMP = 8 **composite object**

O_DP = 9 **distributed polynomial; 分散多項式**

O_USINT = 10 **32bit unsigned integer**

O_ERR = 11 **error object**

O_GF2MAT = 12 **matrix over GF(2)**

O_MATHCAP = 13 **MATHCAP object**

O_F = 14 **first order formula**

O_GFMMAT = 15 **matrix over GF(p)**

O_VOID = -1 **VOID object**

2.2 数

```
struct oNum {       数の共通部分
    short id;       識別子 (= O_N)
    char nid;       数識別子
```

```
    char pad;
};

typedef struct oQ *Q;
```

数は, 多項式と同レベルの object として扱われ, 多項式に対する演算ルーチンの入力として同様に用いることができる. メンバ id はそのための識別子であり, 数では常に O_N である. 数は数識別子 nid をもつ. 現在, 数には次のような型がある.

N_Q = 0 **有理数**

N_R = 1 **倍精度浮動小数**

N_A = 2 **代数的数**

N_B = 3 **PARI bigfloat**

N_C = 4 **複素数**

N_M = 5 **小標数素体**

N_LM = 6 **大標数素体**

N_GF2N = 7 **標数 2 有限体**

N_GFPN = 8 **小標数素体の拡大体**

2.2.1 四則

以下の四則は数に対するトップレベルの函数である. 同一の数識別子を持つ数の演算では, 結果はその識別子をもつ数になる. 倍精度浮動小数と有理数の演算結果は倍精度浮動小数, 倍精度浮動小数と PARI bigfloat の演算結果は PARI bigfloat, 有理数と有限体の元の演算結果は有限体の元となる.

```
#include "ca.h"

addnum(Num a,Num b,Num *rp)
*rp = a + b

subnum(Num a,Num b,Num *rp)
*rp = a - b

mulnum(Num a,Num b,Num *rp)
*rp = a * b

divnum(Num a,Num b,Num *qp,Num *rp)
*qp = a / b

pwrnum(Num a,Num e,Num *rp)
*rp = a ^ e

int compnum(Num a,Num b)
```

ある規則による比較. 有理数の場合は a=b のとき 1, a>b のとき 1, a<b のとき -1.

2.3 有理数

```
struct oN {        自然数
    int p;         桁数
    int b[1];      BASE-進表示の各桁
};

typedef struct oN *N;

struct oQ {          有理数
    short id;        識別子 (= O_N)
    char nid;        数識別子 (= N_Q)
    char sgn;        符号   (= 1 or -1)
    N nm;            分子
    N dn;            分母   (整数を表す時 0)
};

typedef struct oQ *Q;
```

有理数は自然数を用いて定義されるが, 自然数自体は Risa object ではない. 自然数は, 'include/base.h'で定義される数 BASE により BASE-進表示され, その桁数が p, 各桁は, 最も下の桁から b[0], b[1], . . . に入る. 現在 BASE は 2^32 である. 定義では, b[] が一桁分しか宣言されていないが, 実際には桁数分の連続領域を用意することになる. 実際の演算においては, oN ではなく N(即ちポインタ) の方を変数あるいは引数として用いる. 現在、自然数は parser において一旦 10^8 進数として oN型に変換され, 次の bnton() 関数により 2^32 進数に変換されている.

有理数は自然数を分母分子とし, 符号を持つ. 有理数は常に既約分数である. 分母を 0 とすることにより整数を表現する.

2.3.1 自然数の生成

```
#include "ca.h"

bnton(int Base,N a,N *rp)
Base-進表示された a を BASE-進表示に変換する.
```

文字列で表された (例えば 10 進の) 多倍長数を内部形式に変換する場合, Base として 10 のべきをとれば容易に Base 表示ができる. これから ntobn() により BASE 表示の自然数が得られる.

2.3.2 自然数の四則

```
#include "ca.h"

addn(N a,N b,N *rp)
*rp = a + b

int subn(N a,N b,N *rp)
*rp = |a - b| return sgn(a-b)

muln(N a,N b,N *rp)
*rp = a * b
```

```
divn(N a,N b,N *qp,N *rp)
*qp = a / b (商) *rp = a mod b (剰余)

pwrn(N a,int e,N *rp)
*rp = a ^ e

gcdn(N a,N b,N *rp)
*rp = gcd(a,b)

int cmpn(N a,N b)
sgn(a-b)
```

sgn(a) は, a の正, 0, 負に応じて 1, 0, -1 を表す. 呼び出し方は次のようになる.

```
...
N n1,n2,n3;
...
addn(n1,n2,&n3);
...
```

2.3.3 有理数の生成

```
#include "ca.h"

NTOQ(n,s,q)    (macro; N n; int s; Q q)
sgn = s, nm = n, dn = 0 なる有理数(整数) q を生成する.

NDTOQ(n,d,s,q) (macro; N n,d; int s; Q q)
sgn = s, nm = n, dn = d なる有理数 q を生成する.

STOQ(n,q)      (macro; int n; Q q)
単整数 n を 有理数 (整数) に変換する.
```

いずれのマクロも必要な領域はマクロ内で確保される.

2.3.4 有理数の四則

```
#include "ca.h"

addq(Q a,Q b,Q *rp)
*rp = a + b

subq(Q a,Q b,Q *rp)
*rp = a - b

mulq(Q a,Q b,Q *rp)
*rp = a * b

divq(Q a,Q b,Q *qp)
*qp = a / b
```

```
invq(Q a,Q *rp)
*qp = 1 / a

pwrq(Q a,Q e,Q *rp)
*rp = a ^ e

int cmpq(Q a,Q b)
sgn(a-b)
```

2.4 倍精度浮動小数

```
struct oReal {      倍精度浮動小数
    short id;       識別子 (= O_N)
    char nid;       数識別子 (= N_R)
    char pad;
    double body;    倍精度浮動小数
};

typedef struct oReal *Real;
```

2.4.1 倍精度浮動小数の生成, 変換

```
#include "ca.h"

MKReal(a,b)    (macro; double a; Real b)
```
body が a である倍精度浮動小数 b を生成する.

```
double RatnToReal(Q a)
```
有理数 a を倍精度浮動小数に変換する.

マクロで必要な領域はマクロ内で確保される.

2.4.2 四則

いずれも, 浮動小数, 有理数両方を入力にとれる. 結果はすべて浮動小数となる.

```
#include "ca.h"

addreal(Num a,Num b,Real *rp)
*rp = a + b

subreal(Num a,Num b,Real *rp)
*rp = a - b

mulreal(Num a,Num b,Real *rp)
*rp = a * b

divreal(Num a,Num b,Real *rp)
*rp = a / b

pwrreal(Num a,Num e,Real *rp)
```

```
*rp = a ^ e

int cmpreal(Num a,Num b)
sgn(a-b)
```

2.5 代数的数

```
struct oAlg {       代数的数
    short id;        識別子 (= O_N)
    char nid;        数識別子 (= N_A)
    char pad;
    struct oObj *body;     代数的数を表す多項式または有理式
};

typedef struct oAlg *Alg;

struct oV {              root
    char *name;          名前
    pointer attr;        定義多項式 (主変数は #n )
    pointer priv;        対応する不定元 (t#n) へのポインタ
};

typedef struct oV *V;

struct oVL {             root リスト
    V v;                 root
    struct oVL *next;    次へのポインタ
};

typedef struct oVL *VL;

extern VL ALG;           定義済み root のリスト
```

代数的数は, root と呼ばれる特殊な代数的数の有理数係数多項式または有理式として表現される. root とは, それまで定義された root で表される代数的数を係数とする 1 変数多項式の根として定義される. root は, 定義が新しい順に, 多項式に対する変数リストと同じ形式で, root リストとして大域変数 ALG に保持される. root は定義順に #n (n は 0 から始まる) という名前で登録される. 代数的数は, それまでに定義された不定元の有理式または多項式として表現されるが, これを実際に多項式として扱うために, 各 root に t#n という名前の特別な不定元を対応させている. これらの操作は, 次に示す mkalg() により自動的に行われる.

2.5.1 代数的数の生成

```
#include "ca.h"

MKAlg(a,b) (macro; Obj a; Real b)
生成元の多項式または有理式から代数的数を生成する.

void mkalg(P p,Alg *r)
p を定義多項式とする root を生成する.
```

`void algtorat(Num n,Obj *r)`
代数的数 `n` の各 `root` を対応する不定元に置き換えた有理式または多項式を生成する.

`void rattoalg(Obj obj,Alg *n)`
`root` に対応する不定元を `root` に置き換えた代数的数を生成する.

マクロで必要な領域はマクロ内で確保される.

2.5.2 四則

いずれも, 代数的数, 有理数両方を入力にとれる. 結果はすべて代数的数となる.

```
#include "ca.h"

addalg(Num a,Num b,Alg *rp)
*rp = a + b

subalg(Num a,Num b,Alg *rp)
*rp = a - b

mulalg(Num a,Num b,Alg *rp)
*rp = a * b

divalg(Num a,Num b,Alg *rp)
*rp = a / b

pwralg(Num a,Num e,Alg *rp)
*rp = a ^ e

int cmpalg(Num a,Num b)
```

`root` の多項式あるいは有理式として比較する.

`cmpalg()` の結果は, 代数的数を, root の多項式あるいは有理式とみて, root 間の順序を元にした順序比較により決まる. root 間の順序は, 後で定義されたもの程順序が高くなるように設定されている.

2.6 小標数素体

```
struct oMQ {        小標数素体の元
    short id;         識別子 (= O_N)
    char nid;         数識別子 (= N_M)
    char pad;
    int cont;         小標数素体の元
};

typedef struct oMQ *MQ;

extern int current_mod;
```

標数が 2^29 未満の素体を効率よく扱うための型である. 有限体の各元自体は属する体に関する情報をもっておらず, 大域変数 current_mod の値により演算が行われる.

2.6.1 小標数素体の元の生成, 変換

```
#include "ca.h"
ptomp(int m,P p,P *pr)
```
有理数係数多項式（有理数を含む）の各係数を, 標数 m の素体の元に変換したものを生成する.

```
mptop(P p,P *pr)
```
小標数素体係数多項式（小標数素体の元を含む）の各係数を有理数型に変換したものを生成する.

2.6.2 四則

引数は 0 または小標数素体の元に限る. 生成された時点での標数にかかわらず, current_mod を標数として計算する.

```
#include "ca.h"

addmi(MQ a,MQ b,MQ *rp)
*rp = a + b

submi(MQ a,MQ b,MQ *rp)
*rp = a - b

mulmi(MQ a,MQ b,MQ *rp)
*rp = a * b

divmi(MQ a,MQ b,MQ *rp)
*rp = a / b

pwrmi(MQ a,Q e,MQ *rp)
*rp = a ^ e

int cmpmi(MQ a,MQ b)
```
cont の値を自然数として比較

2.7 大標数素体

```
struct oLM {            大標数素体の元
    short id;           識別子 (= O_N)
    char nid;           数識別子 (= N_LM)
    char pad;
    struct oN *body;    大標数素体の元
};

typedef struct oLM *LM;
```

任意標数の素体を扱うための型である. 標数は oN 型の自然数としてある静的変数に格納される. 各元自体は属する体に関する情報をもっておらず, 設定された標数の値により演算が行われる.

2.7.1 大標数素体の元の生成, 変換

```
#include "ca.h"
```
void setmod_lm(N p)
自然数 p を大標数素体の標数として設定する. p の素数チェックは行わない.

void getmod_lm(N *p)
その時点で設定されている大標数素体の標数を返す. 設定されていない場合 0 を返す.

void qtolm(Q q,LM *l)
有理数 q を標準的写像によりその時点で設定されている
素体の元に写像した値を生成する. 分母の行き先が 0 の場合 error() 関数が呼ばれる.

void simplm(LM n,LM *r)
n の body の, その時点で設定されている標数による剰余を body とする元を生成する.

2.7.2 四則

引数は 0 または大標数素体の元または有理数. 有理数はその時点で設定されている素体の元に変換される.

```
#include "ca.h"

addlm(LM a,LM b,LM *rp)
*rp = a + b

sublm(LM a,LM b,LM *rp)
*rp = a - b

mullm(LM a,LM b,LM *rp)
*rp = a * b

divlm(LM a,LM b,LM *rp)
*rp = a / b

pwrlm(LM a,Q e,LM *rp)
*rp = a ^ e

int cmplm(LM a,LM b)
```
body の値を自然数として比較

2.8 標数 2 の有限体の元

```
struct oGF2N {          GF(2^n) の元
    short id;           識別子 (= O_N)
    char nid;           数識別子 (= N_GF2N)
    char pad;
    struct oUP2 *body;  GF(2^n) の元
};

typedef struct oGF2N *GF2N;

typedef struct oUP2 {   GF(2) 上の 1 変数多項式
    int w;              配列 b のサイズ
    unsigned int b[1];  係数配列 (dense 表現)
} *UP2;

typedef struct oUP2 *UP2;
```

標数 2 の有限体を扱うための型である. 標数は UP2 型の多項式としてある静的変数に格納される. 各元自体は属する体に関する情報をもっておらず, 設定された標数の値により演算が行われる.

2.8.1 標数 2 の有限体の元の生成, 変換

```
#include "ca.h"
void setmod_gf2n(P p)
```

整数係数多項式 p の各係数を GF(2) の元とみた多項式を, GF(2) の拡大体の定義多項式として設定する. 多項式の GF(2) 上の既約性はチェックしない.

```
void getmod_gf2n(UP2 *p)
```

その時点で設定されている定義多項式を UP2 形式で返す.

```
void ptogf2n(Obj q,GF2N *l)
```

整数係数多項式 p の各係数を GF(2) の元とみた多項式を, GF2N 型に変換する. 定義多項式が設定されている, いないにかかわらず, 簡単化は行わない.

```
void gf2ntop(GF2N q,P *l)
```

ptogf2n() の逆操作である.

```
void simpgf2n(GF2N n,GF2N *r)
```

body が, n の body のその時点で設定されている定義多項式による剰余であるような GF2N 型の元を生成する.

2.8.2 四則

引数は F(2^n) の元または有理数. 有理数はその時点で設定されている GF(2^n) の元に変換される.

```
#include "ca.h"

addgf2n(GF2N a,GF2N b,GF2N *rp)
```

```
*rp = a + b

subgf2n(GF2N a,GF2N b,GF2N *rp)
*rp = a - b

mulgf2n(GF2N a,GF2N b,GF2N *rp)
*rp = a * b

divgf2n(GF2N a,GF2N b,GF2N *rp)
*rp = a / b

pwrgf2n(GF2N a,Q e,GF2N *rp)
*rp = a ^ e

int cmpgf2n(GF2N a,GF2N b)
body の値を GF(2) 上の多項式として比較
```

2.9 多項式

```
struct oV {             変数（不定元）
    char *name;         名前
    pointer attr;       属性（通常の不定元では 0）
    pointer priv;       属性に応じたプライベートデータ
};

typedef struct oV *V;

struct oVL {            変数リスト
    V v;                変数
    struct oVL *next;   次へのポインタ
};

typedef struct oVL *VL;

struct oP {             多項式
    short id;           識別子 (= O_P)
    short pad;
    V v;                主変数
    struct oDCP *dc;    次数係数リストへのポインタ
};

typedef struct oP *P;

struct oDCP {           次数係数リスト
    Q d;                次数
    P c;                係数 ( Q でもよい )
    struct oDCP *next;  次へのポインタ
};
```

```
typedef struct oDCP *DCP;

extern VL CO;              定義済み不定元リスト
```

多項式は, ある変数 (主変数) に関する多項式として再帰的に表現される. 次数係数リストは

```
<係数> * <主変数>^<次数>
```

という多項式の各項を, 降べきの順にリストで表現したものである. 多項式も数と同様に識別子を持つ. 係数が真に多項式であるか, 数であるかは, その識別子により判定する. 変数たちはある順序が定められ, 多項式はその変数順序により構成され, 演算される. 変数順序は VL により表現される.

2.9.1 多項式の生成

```
#include "ca.h"

MKP(v,dc,p)  (macro; V v; DCP dc; P p)
主変数 v, 次数係数リスト dc から多項式 p を生成する.
dc の次数が 0 の場合 p = <dc の係数> となる.

MKV(v,p)     (macro; V v; P p)   変数 v を多項式に変換する.

NEXTDC(r,c)  (macro; MP r,c)
r を次数係数リストの先頭とし, c を末尾とするとき, 末尾に次数係数リストを
追加し, c をその次数係数リストを指すようにする. r が 0 のとき,
次数係数リストが生成され, r および c はその次数係数リストを指す.
```

NEXTDC()は, 次数係数リストの末尾に次数係数ペアを追加していく場合に用いる.

2.9.2 四則

```
#include "ca.h"

addp(VL vl,P a,P b,P *rp)
*rp = a + b

subp(VL vl,P a,P b,P *rp)
*rp = a - b

mulp(VL vl,P a,P b,P *rp)
*rp = a * b

pwrp(VL vl,P a,Q e,P *rp)
*rp = a ^ e

compp(VL vl,P a,P b)
ある規則による比較.
```

これらの函数の引数として数も同様にとれる. 割り算はその他の演算の項で述べる.

compp(vl,a,b) は次のような順序づけによる. 以下で, a の順序が上の場合 1, b が上の場合 -1 を返す.

1. 一方が数, 一方が数でない多項式の場合, 後者が上.
2. 双方とも数の場合, compnum(a,b) の値による.
3. 双方とも多項式の場合, 主変数の順序が vl の方が上.
4. 主変数が等しい場合, 次数が高い方が上.
5. 主変数も次数も等しい場合, 係数を上から比較する.

2.10 有理式

```
struct oR {             有理式
    short id;           識別子 (= O_R)
    short reduced;      既約分数のとき 1
    P nm;               分子
    P dn;               分母
};

typedef struct oR *R;
```

有理式は, 単に分母, 分子という二つの多項式の組合せである. 有理数と異なり, 必ずしも既約とは限らない. 既約にするためには, reductr() により明示的に約分を行なう必要がある. 一度約分された有理式は, メンバ reduced が 1 になるため既約性が保証される. Obj は risa において独立して存在する (識別子を持つ) object に共通するメンバである.

2.10.1 有理式の生成

```
#include "ca.h"

PTOR(p,r)    (macro; P p; R r)
```
多項式 p を, 分子 p, 分母 1 の有理式に変換する.

2.10.2 四則

以下の各函数は入力として数, 多項式, 有理式 (id が O_R 以下) の object がとれる.

```
#include "ca.h"

addr(VL vl,Obj a,Obj b,Obj *rp)
*rp = a + b

subr(VL vl,Obj a,Obj b,Obj *rp)
*rp = a - b

mulr(VL vl,Obj a,Obj b,Obj *rp)
*rp = a * b

divr(VL vl,Obj a,Obj b,Obj *rp)
*rp = a * b

pwrr(VL vl,Obj a,Q e,Obj *rp)
*rp = a ^ e
```

```
reductr(VL vl,Obj a,Obj *rp)
*rp = a を約分したもの.
```

2.11 List

```
typedef struct oNODE {    list を表すための node
    pointer body;         node の内容
    struct oNODE *next;   次への pointer
} *NODE;

typedef struct oLIST {     Risa list object
    short id;              識別子 (=O_LIST)
    short pad;
    struct oNODE *body;    list 本体
} *LIST;
```

list 構造は engine 内部において多用される. 前述の DCP, VL などは用途が特殊化されたものであるが, 一般には NODE が用いられる. LIST は Risa object としての list である. この場合, list 本体の各 node の body は Risa object を指している必要がある.

2.11.1 リストの生成

```
#include "ca.h"

MKNODE(n,b,nn)  (macro; NODE n,nn; pointer b)
body = b, next = nn なる NODE n を生成する.

MKLIST(a,b) (macro; NODE b; LIST a)
body=b なる LIST a を生成する.

NEXTNODE(r,c) (macro; NODE r,c)
r を node の先頭とし, c を node の末尾とするとき, 末尾に空の node
を追加し, c をその空 node を指すようにする. r が 0 のとき,
空 node が生成され, r および c はその node を指す.
```

MKNODE は node の先頭にデータを追加していくときに用いる. 逆に, node の末尾にデータを追加していくには NEXTNODE を用いる.

```
NODE r, c;

for ( r = 0; ...; ... ) {
    NEXTNODE(r,c);
    c->body = ...;
}
if ( r ) c->next = 0;
```

これにより, list の末尾にデータを追加しながら list を伸ばすことができる.

2.12 ベクトル

```
struct oVECT {          ベクトル
    short id;           識別子 (= O_VECT)
    short pad;
    pointer *body;      成分配列へのポインタ
} *VECT;
```

ベクトルは, 数学的な対象としてのベクトルとして用いられるほか, さまざまな object を成分に持つ一次元配列としても用いられる. 前者の場合, ベクトルどうしの和, 差, スカラー倍, 行列との積が基本演算として提供される. 行列との積を計算する場合, 行列を左からかける場合には, ベクトルは列ベクトル, 右からかける場合には行ベクトルと見なされる.

2.12.1 ベクトルの生成

```
#include "ca.h"

MKVECT(m,l)  (macro; VECT m; int l)
長さ l のベクトル m を生成する.
```

2.12.2 四則

```
#include "ca.h"

void addvect(VL vl,VECT a,VECT b,VECT *rp)
*rp = a + b

void subvect(VL vl,VECT a,VECT b,VECT *rp)
*rp = a - b

void mulvect(VL vl,Obj a,Obj b,Obj *rp)
*rp = a * b (一方がスカラーすなわち数, 多項式または有理数の場合のスカラー倍)

void divvect(VL vl,Obj a,Obj b,Obj *rp)
*rp = a / b (スカラー b による 1/b 倍)

int compvect(VL vl,P a,Q e)
以下で述べる順序で比較.
```

ベクトルどうしの順序比較は以下による.

1. 長さが大きい方が順序が上.
2. 長さが等しい場合, 成分どうしを先頭から比較.

2.13 行列

```
struct oMAT {          行列
    short id;           識別子 (= O_MAT)
    short pad;
    pointer **body;      行成分ポインタ配列へのポインタ
} *VECT;
```

行列は, 数学的な対象としての行列として用いられるほか, さまざまな object を成分に持つ二次元配列としても用いられる. 前者の場合, 行列どうしの和, 差, スカラー倍, 行列, ベクト

ルとの積が基本演算として提供される. ベクトルとの積を計算する場合, 行列を左からかける場合には, ベクトルは列ベクトル, 右からかける場合には行ベクトルと見なされる.

2.13.1 行列の生成

```
#include "ca.h"

MKMAT(m,row,col)  (macro; MAT m; int row, col)
row 行 col 列の行列 m を生成する.
```

2.13.2 四則

```
#include "ca.h"

void addmat(VL vl,MAT a,MAT b,MAT *rp)
*rp = a + b

void submat(VL vl,MAT a,MAT b,MAT *rp)
*rp = a - b

void mulmat(VL vl,Obj a,Obj b,Obj *rp)
*rp = a * b (スカラー倍, 行列-ベクトルの積, 行列-行列の積)

void divmat(VL vl,Obj a,Obj b,Obj *rp)
*rp = a / b (スカラー b による除算)

void pwrmat(VL vl,MAT a,Obj e,Obj *rp)
*rp = a^e (正方行列 a の e 乗)

int compmat(VL vl,MAT a,MAT e)
以下で述べる順序で比較.
```

行列どうしの順序比較は以下による.

1. 行数が大きい方が順序が上.
2. 行数が等しい場合, 列数が大きい方が順序が上.
3. サイズが等しい場合, 成分どうしを最初の行から, 行の先頭から比較.

2.14 文字列

```
struct oSTRING {          文字列
    short id;             識別子 (= O_STR)
    short pad;
    char *body;       0 終端 C 文字列へのポインタ
} *VECT;
```

C の 0 終端文字列の wrapper である. 加算 (文字列の接続), 比較演算のみが用意されている.

2.14.1 文字列の生成

```
#include "ca.h"

MKSTR(m,b)  (macro; STRING m; char *b)

0 終端 C の文字列 b を Risa 文字列 object m に変換する.
```

2.14.2 四則

```
#include "ca.h"

void addstr(VL vl,STRING a, STRING b, STRING *rp)
*rp = a->body と b->body を繋げた文字列

int compstr(VL vl,STRING a,STRING b)
標準ライブラリ関数 strcmp() による比較結果を返す.
```

2.15 分散表現多項式

```
typedef struct oDL {    指数ベクトル
    int td;             全次数
    int d[1];           指数ベクトル本体
} *DL;

typedef struct oMP {    単項式リスト
    struct oDL *dl;     指数ベクトルへのポインタ
    P c;                係数
    struct oMP *next;   次の単項式へのポインタ
} *MP;

typedef struct oDP {    分散表現多項式
    short id;           識別子 (=O_DP)
    short nv;           変数の個数
    int sugar;          sugar の値
    struct oMP *body;   単項式リストへのポインタ
} *DP;

struct order_pair {     ブロックに対する項順序型指定
    int order;          項順序型
    int length;         ブロックの長さ
};

struct order_spec {    項順序型指定構造体
    int id;            項順序型指定識別子
                    (0: primitive 項順序; 1:ブロック項順序; 2:行列による項
順序)
    Obj obj;           Risa object として与えられた項順序型
    int nv;            変数の個数
    union {                                    項順序型指定
        int simple;                            primitive 項順序識別子
```

```
        struct {                                ブロック項順序型指定
            int length;                         ブロック数
            struct order_pair *order_pair;      各ブロックの項順序型配列へのポ
インタ
        } block;
        struct {                                行列による項順序型指定
            int row;                            列数
            int **matrix;                       行列本体
        } matrix;
    } ord;
};
```

分散表現多項式は, 多項式を, ある代数系を係数とする単項式の和として表現する. 単項式は, 係数および項 (term) の積である. 項は変数の冪積である. 変数を固定するとき, 項は, 指数のみを取り出した指数ベクトルで表せる. 上記構造体定義において, DP は分散表現多項式の wrapper, MP は各単項式の係数および指数ベクトルを表す. また DL は指数ベクトルを表す.

MPにより単項式リストを構成する際, 項の順序を指定する必要がある. これは, 変数が固定された場合に, 項の集合の間にある条件を満たす全順序を設定することにより行う. Risa においては, 項順序の指定を,

1. 各変数と, 指数ベクトルの成分の位置の対応 (変数順序)
2. 指数ベクトルに対する順序の指定 (項順序型)

により行う. 1. は単に変数をリストとして指定することにより行う. 2. は 3 つのカテゴリに分かれる.

1. primitive 項順序型

   これは, 次のような識別子により指定される型である.

   全次数つき順序は, まず全次数による比較を行う.

   辞書式順序は, 指数ベクトルの先頭から比較を行い, 値が大きい方の順序を上とする.

   逆辞書式順序は, 指数ベクトルの末尾から比較を行い, 値が小さい方の順序を上とする.

   ```
   ORD_REVGRADLEX = 0    全次数逆辞書式順序
   ORD_GRADLEX = 1       全次数辞書式順序
   ORD_DLEX = 2          辞書式順序
   ```

2. ブロック項順序型

   これは, 変数リストを幾つかのブロックにわけ, 各ブロックに primitive 項順序型を指定して, 先頭のブロックから比較を行っていく型である. この型は,

   [[N1,O1],[N2,O2],...]

   なるリストにより指定できる. ここで, Ni はブロック内の変数の個数, Oi はそのブロックにおける primitive 項順序型である.

3. 行列項順序型

   指数ベクトル a, b に対し, ある条件を満たす行列 M により, M(a-b) の最初の 0 でない成分が正の場合に a の順序が上, として全順序が定義できる. この順序型を, M により定義される行列項順序型とよぶ

DP は sugar なるメンバを持つ. これは, 以下の規則により計算される.

以下で説明する ptod() により生成された分散表現多項式の場合には全次数に一致する.

分散多項式に単項式をかけた場合には, sugar の値は多項式の sugar に単項式の全次数を加えたものになる.

分散多項式の加減算においては, 結果の sugar の値は大きい方の値となる.

この値は通常は呼び出した演算関数によりメンテナンスされるが, 独自に分散多項式を生成した場合には, 上の規則により sugar の値を計算し設定する必要がある.

2.15.1 分散表現多項式の生成

```
#include "ca.h"
```

`MKDP(n,m,d)` (macro; int n; MP m; DP d)
変数の個数が `n` である単項式リスト `m` から分散表現多項式を生成する.
`sugar` の値は設定されず, この `macro` を実行したあとに設定する必要がある.

`NEWDL(d,n)` (macro; int n; DL d)
長さ `n` の指数ベクトル `d` を生成する. `d` の全次数は `0`, 各成分は `0` に初期化される.

`NEWMP(m)` (macro; MP m)
単項式リスト構造体 `m` を生成する.

`NEXTMP(r,c)` (macro; MP r,c)
`r` を単項式リストの先頭とし, `c` を末尾とするとき, 末尾に単項式リストを追加し, `c` をその単項式リストを指すようにする. `r` が `0` のとき, 単項式リストが生成され, `r` および `c` はその単項式リストを指す.

`void initd(struct order_spec *spec)`
項順序構造体 `spec` を現在の項順序として指定する.

`void ptod(VL vl, VL dvl, P p, DP *pr)`
変数順序 `vl` の元で構成された (再帰表現) 多項式 `p` を, 変数順序 `dvl` のもとで, 現在設定されている項順序型に従って分散表現多項式に変換し `*pr` とする.

`NEXTMP()` は単項式リストの末尾に単項式を追加していく場合に用いる.

2.15.2 四則

```
#include "ca.h"

void addd(VL vl,DP a,DP b,DP *rp)
*rp = a + b

void subd(VL vl,DP a,DP b,DP *rp)
*rp = a - b

void muld(VL vl,DP a,DP b,DP *rp)
*rp = a * b

compd(VL vl,DP a,DP b)
```

以下の順序で比較

`compd(vl,a,b)` は次のような順序づけによる.

1. 先頭の単項式の指数ベクトルを, 現在設定される順序で比較.
2. 指数ベクトルが等しい場合, 係数を多項式の順序で比較.
3. 先頭の単項式が等しい場合, 1., 2. の比較を次の項以降に対して続ける.

3 その他の演算

以下の諸演算において, 最後の引数は, 呼び出し側によって確保された, 結果のポインタを書く場所を示すポインタである.

3.1 除算

```
#include "ca.h"

divsrp(vl,a,b,qp,rp)   *qp = a / b; *rp = a % b
VL vl;
P a,b,*qp,*rp;

premp(vl,a,b,rp)       *rp = lc(b)^(deg(a)-deg(b)+1)*a % b
VL vl;
P a,b,*rp;
```

一般に多変数多項式に対しては, 必ずしも除算が実行できるとは限らない. `divsrp()` は, 商, 剰余が存在することが分かっている場合にそれらを求める函数である. これは, 例えば除数の主係数が有理数である場合などに用いられる. 一般に多項式剰余は擬剰余を計算することにより求める.

3.2 GCD

```
#include "ca.h"

ezgcdp(vl,a,b,rp)     *rp = gcd(pp(a),pp(b))
VL vl;
P a,b,*rp;

ezgcdpz(vl,a,b,rp)    *rp = gcd(a,b)
VL vl;
P a,b,*rp;

pcp(vl,a,pp,cp)       *pp = pp(a); *cp = cont(cp);
VL vl;
P a,*pp,*cp;
```

`pp(a)` は `a` の原始的部分, `cont(a)` は容量を表す. `ezgcdp()` は整数因子を除いた gcd, `ezgcdpz()` は整数因子をこめた gcd を計算する.

3.3 代入

```
#include "ca.h"

substp(vl,a,v,b,rp)
VL vl;
P a,b,*rp;
V v;
```

```
substr(vl,a,v,b,rp)
VL vl;
R a,b,*rp;
V v;
```

3.4 微分

```
#include "ca.h"

diffp(vl,a,v,rp)
VL vl;
P a,*rp;
V v;

pderivr(vl,a,v,rp)
VL vl;
R a,*rp;
V v;
```

diffp() は多項式 a を v で偏微分する. pderivr() は有理式 a を v で偏微分する.

3.5 終結式

```
#include "ca.h"

resultp(vl,v,a,b,rp)
VL v;
P a,b,*rp;
```

resultp() は多項式 a,b の, v に関する終結式を計算する.

3.6 因数分解

```
#include "ca.h"

fctrp(vl,a,dcp)
VL vl;
P a;
DCP *dcp;

sqfrp(vl,a,dcp)
VL vl;
P a;
DCP *dcp;
```

fctrp(), sqfrp() は多項式 a の有理数体上での既約因子分解, 無平方因子分解をそれぞれ行う. 因数分解の結果は [因子, 重複度] のリストとして表現できる. これを次数係数リスト用のデータ構造 DCP を流用して表現する. すなわち, メンバ d に重複度, メンバ c に因子を代入する. 分解は, まず入力多項式 a を

```
a = c * b   (c は有理数, b は整数上原始的な多項式)
```

と分解した後, b を実際に分解する. 結果は, リストの先頭に, [c, 1] なる定数項, 以下 b の因子が続く.

4 マクロ, 大域変数

4.1 macro

'ca.h' で定義される主なマクロは次の通りである.

4.1.1 一般マクロ

```
MAX(a,b)  ((a) > (b) ? (a) : (b) )
MIN(a,b)  ((a) > (b) ? (b) : (a) )
ABS(a)    ((a)>0?(a):-(a))
ID(p)     ((p)->id)
BDY(p)    ((p)->body)
NEXT(p)   ((p)->next)
VR(p)     ((p)->v)
NM(q)     ((q)->nm)
DN(q)     ((q)->dn)
SGN(q)    ((q)->sgn)
PL(n)     ((n)->p)
BD(n)     ((n)->b)
```

4.1.2 述語

```
NUM(a)    ID(a)==0_Q
INT(a)    (!DN(a))
UNIQ(a)   a が有理数の 1 に等しい
MUNIQ(a)  a が有理数の -1 に等しい
UNIN(a)   a が自然数の 1 に等しい
```

4.1.3 メモリ割り当て器

`(char *) MALLOC(d)`
: d bytes の領域を割り当てる.

`(char *) CALLOC(d,e)`
: d * e bytes の領域を割り当てて, 0 で初期化する. region to 0.

`(N) NALLOC(d)`
: d 桁の自然数用の領域を割り当てる.

これらはすべて領域の先頭ポインタを返す.

4.1.4 cell allocators

`NEWQ(q)` q に \codeQ 用の領域を割り当てる.

`NEWP(p)` p に \codeP 用の領域を割り当てる.

`NEWR(r)` r に \codeR 用の領域を割り当てる.

`NEWNODE(a)`
a に \codeNODE 用の領域を割り当てる.

`NEWDC(dc)`
dc に \codeDCP 用の領域を割り当てる.

`NEWV(v)` v に \codeV 用の領域を割り当てる.

`NEWVL(vl)`
vl に \codeVL 用の領域を割り当てる.

`NEWP()`, `NEWQ()`, `NEWR()` においては, メンバ `id` もしかるべき値に初期化される.

4.2 主な大域変数

`VL CO;` 現在の変数順序.

`Q ONE;` 有理数の 1.

`N ONEN;` 自然数の 1.

`int prime[];`
4 桁までの素数 (小->大).

`int lprime[];`
8 桁程度の素数 1000 個 (大->小).

`CO` は, ユーザが初期化, および新たに変数が出現した場合に更新する必要がある. また, `ONER`, `ONE`, `ONEN`は, 起動時に函数 `nglob_init()` により初期化する必要がある.

5 Parser

5.1 Parser の構成

parser は Asir 言語で書かれた文字列を中間言語に変換する. parser は次のものから構成される.

文法定義

Asir 言語の文法は `yacc` のソースファイルとして定義されている. このファイルは `yacc` により parser プログラムに変換される.

字句解析

yacc で生成される parser プログラムは, 入力文字列の先頭から順に, 属性毎に文字列を切り分ける関数 `yylex()` を呼び出しながら, 文法定義に従って中間言語 tree を構成していく. 字句解析定義ファイルから `yylex()` を生成するプログラム `lex` もあるが, Asir では歴史的な理由から C 言語により直接記述している.

名前管理

`yylex()` によって切り分けられる文字列として, あらかじめ定義された keyword (‘`if`’, ‘`for`’ など) の他に, 変数名, 関数名など動的に生成される名前がある. これらは属性毎にリスト, 配列などの形で管理され, parser の実行中に参照, 追加などが行われる.

5.2 中間言語

```
typedef struct oFNODE {     式を表すノード
    fid id;                 識別子
    pointer arg[1];         引数配列（長さ可変）
} oFNODE *FNODE;

typedef struct oSNODE {     文を表すノード
    sid id;                 識別子
    int ln;                 行番号
    pointer arg[1];         引数配列（長さ可変）
} oSNODE *SNODE;
```

interpreter の入力となる中間言語の構成は極めて単純である. 中間言語の構成要素は **文** および **式** である. Asir 言語による入力文字列は, 文の並びとして parse され, `SNODE` 構造体のリストに変換される. 文の種類は識別子 `id` により示される. 引数配列は文の種類に応じて長さ, およびその各要素の意味 (役割) が決まる. 以下で `S_` で始まるのは識別子の名前で, 実際には相異なる整数が割り当てられている. ‘`|`’ は, 識別子および引数の仕切りを示す.

`S_SINGLE | expr`
: ‘`expr`’を実行する.

`S_BREAK`
: 最も内側のループ (`S_FOR`, `S_DO`) を抜ける.

`S_CONTINUE`
: 最も内側のループの先頭に飛ぶ.

`S_RETURN | expr`
: 関数から抜けて, ‘expr’ の値を返す.

`S_IFELSE | dummy | expr | stat1 | stat2`
: sampexpr の値が 0 でないなら ‘stat 1’ を実行, 0 なら ‘stat 2’ を実行.

`S_FOR | dummy | expr1 | expr | stat | expr2`
: まず ‘expr1’ を実行したあと, ‘expr’ の値が 0 でない間, ‘stat’ と ‘expr2’ の計算を繰り返す.

`S_DO | dummy | stat | expr`
: ‘stat’ を実行したあと ‘expr’ を計算し, その値が 0 でない間繰り返す.

SNODE は行番号を示すメンバ ln を持つ. 各行がファイルから読まれた場合に, そのファイルにおける行番号を格納しておく. これはデバッグ時に行番号から文を特定するためなどに用いられる.

文に対する中間言語定義で分かるように, 文は最終的には式から構成されている. 文中に現れた式は, parser によって解析され, FNODE 構造体のリストに変換される. 式の種類は識別子 idにより示される. 引数配列は式の種類に応じて長さ, およびその各要素の意味 (役割) が決まる. 以下で I_ で始まるのは識別子の名前で, 実際には相異なる整数が割り当てられている. ‘|’ は, 識別子および引数の仕切りを示す.

`I_STR | string`
: 文字列

`I_FORMULA | object`
: 既に Risa object に変換されている \codeobject

`I_ANS | index`
: ‘index’ 番目の計算結果

`I_GF2NGEN`
: 標数 2 有限体の生成元

`I_EV | node`
: ‘node’ を指数ベクトルとみて, 係数 1 の単項式を生成する.

`I_FUNC | function | argument list`
: 関数呼び出し

`I_CAR | list`
: list の先頭要素

`I_CDR | list`
: list から先頭要素を除いたリスト

`I_PVAR | index`
: index 番目のプログラム変数の値

`I_ASSPVAR | expr1 | expr2`
: ‘expr1’ で示されるプログラム変数に ‘expr2’ を代入

`I_INDEX | expr | index`
: ‘expr’ を配列またはリストと見て, ‘index’で指定される要素を取り出す.

`I_POSTSELF | function | expr`
: ‘expr’ の値を取り出したあと, ‘function’ が加算なら ‘expr’ を 1 増やし, ‘function’ が減算なら ‘expr’ を 1 減らす.

`I_PRESELF | function | expr`
: ‘function’ が加算なら ‘expr’ を 1 増やし, ‘function’ が減算なら ‘式’ を 1 減らす. その後 ‘expr’ の値を取り出す.

`I_LIST | node`
: ‘node’からリストを生成する.

`I_NOP | expr`
: ‘expr’ が 0 でないなら 0, 0 なら 1.

`I_OR | expr1 | expr2`
: ‘expr1’, ‘expr2’ がともに 0 なら 0, それ以外は 1.

`I_AND | expr1 | expr2`
: ‘expr1’, ‘expr2’ がともに 0 でないなら 1, それ以外は 0.

`I_CE | expr | expr1 | expr2`
: ‘expr’ が 0 でないとき ‘expr1’ の値, 0 のとき ‘expr2’ の値.

`I_BOP | bop_id | expr1 | expr2`
: ‘bop_id’ で指定される二項演算子 (加減乗除など) に従って, ‘expr1’, ‘expr2’ を引数として演算.

`I_COP | cop_id | expr1 | expr2`
: ‘cop_id’ で指定される比較演算子に従って, ‘expr1’, ‘expr2’ を比較. 結果は 0, 1, -1.

`I_LOP | lop_id | expr1 | expr2`
: ‘lop_id’ で指定される論理演算子により, ‘expr1’, ‘expr2’ を引数として論理式を生成.

5.3 字句解析

字句解析部では, 空白, タブ, 改行をスキップしたあとの最初の文字によって最初の分類を行う.

- 0

  続く 0 をスキップして, 数字が来たら 10 進数, b が来たら 2 進数, x が来たら 16 進数として, あとに続く valid な文字をバッファに読み込み, 2^32 進数に変換する.
- 0 以外の数字

  以下に続く数字をバッファに読み込み, 10 進数として 2^32 進数に変換する.
- 英小文字

  以下に続く, アルファベット, 数字, ‘_’ をバッファに読み込み, keyword の場合にはその識別子, そうでない場合には ‘小文字で始まる文字列’ を意味する識別子を返す.
- 英大文字

  以下に続く, アルファベット, 数字, ‘_’ をバッファに読み込み, ‘大文字で始まる文字列’ を意味する識別子を返す.

- `@`
  `@` はその後に来る文字列によりさまざまな対象を表す.
  - `@@`
    直前の計算結果
  - `@pi`
    円周率を表す不定元
  - `@e`
    自然対数の底を表す不定元
  - `@i`
    虚数単位
  - `@p`
    奇標数有限体の拡大体の生成元
  - `@true`, `@false`, `@impl`, `@repl`, `@equiv`
    論理演算子

  これ以外の場合 `@` は標数 2 の有限体の生成元を表す.
- ‘`"`’
  次の ‘`"`’ の直前の文字までを文字列 Risa object とみなす.
- ‘`'`’
  次の ‘`'`’ の直前の文字までを ‘小文字で始まる文字列’ とみなす. これは, 任意文字列を名前とする不定元を生成する必要がある場合などに用いる.
- その他の記号
  記号に応じてさまざまに扱われる. 多くは演算子として扱われるが, ‘`{`’, ‘`}`’, ‘`[`’, ‘`]`’, ‘`(`’, ‘`)`’ など, 一対で間にある対象に作用するものもある.

5.4 名前管理

Asir においては, 不定元, 関数, プログラム変数という 3 つのカテゴリ別に名前が管理されている.

5.4.1 不定元

不定元は変数リスト `CO` (Current variable Order) で管理される. 不定元と認識される文字列が字句解析部から与えられた場合, `CO` に登録されている不定元の名前とその文字列を順に比較し, 一致した場合にはその不定元構造体ポインタを対応する変数として用いる. 一致する名前がない場合には新たに不定元構造体を生成し, 与えられた文字列を名前として登録し, `CO` の末尾に追加する.

5.4.2 関数

```
typedef struct oFUNC {     Asir 関数
  char *name;              関数名
  int argc;                引数の個数
  int type;                PARI 関数型
```

```
  aid id;                   型（未定義，組み込み，ユーザ，純，PARI）
  union {
    void (*binf)();         組み込み関数
    struct oUSRF *usrf;     ユーザ定義関数構造体
    struct oPF *puref;      純関数
  } f;
} *FUNC;

typedef struct oUSRF {      ユーザ定義関数
  char *fname;              関数名
  short vol;                未使用
  int startl,endl;          ファイル内での開始，終了位置
  NODE args;                仮引数リスト
  VS pvs;                   局所プログラム変数配列
  char *desc;               関数の説明
  struct oSNODE *body;      文リスト（関数本体）
} *USRF;

typedef struct oPF {        純関数
  char *name;               関数名
  int argc;                 引数の個数
  Obj body;                 ユーザ定義純関数の本体
  V *args;                  引数配列
  Obj *deriv;               偏導関数配列
  NODE ins;                 関数インスタンスリスト
  int (*pari)();            PARI 呼び出し関数
  double (*libm)();         C 数学関数
  int (*simplify)();        simplifier
} *PF;

struct oV {                 不定元（再掲）
    char *name;
    pointer attr;           属性
    pointer priv;
};

extern NODE sysf,noargsysf;  組み込み関数リスト
extern NODE usrf;            ユーザ定義関数リスト
extern NODE parif;           PARI 関数リスト
extern NODE pflist;          純関数リスト
```

関数には, 組み込み関数, ユーザ定義関数, PARI 関数および純関数がある. いずれも 関数構造体 FUNC として登録されリストとして保持される. 組み込み関数のうち, 引数を持つものは, sysf に, 持たないものは noargsysf に登録される.

ユーザ定義関数は, 呼び出し時あるいは関数定義時に, usrf を検索し, リスト中にその名の関数がない場合に FUNC 構造体が生成され, リストに追加される. 関数定義が行われる前に呼び出しが行われた場合, FUNC 構造体のメンバ id には, 未定義を意味する識別子がセットされる. 後に実際に関数定義が行われた際に, このメンバはユーザ定義関数を意味する識別子で

上書きされ, その他のメンバも然るべくセットされる. 既に定義されている場合には, これらのメンバは上書きされる. 即ち, 同一関数名では最新の定義が用いられる.

PARI 関数は, 実際の計算に PARI ライブラリを用いるという点を除けば, 組み込み関数と同等である. ただし, PARI ライブラリは PARI bigfloat を結果として返すので, double float の結果を得たい場合のために, C の `libm` ライブラリの同等の関数ポインタを保持している.

組み込み, ユーザ定義, PARI 関数は, 引数を用いて計算を行い, Risa object を結果として返す, 通常のプログラミング言語の意味での関数だが, 純関数は, 引数のみが評価されて, その引数を持つ関数呼び出しそのものを返す. Asir の実行例を示す.

```
[0] A=sin(x);
sin(x);
[1] type(A);
2                       <--- 純関数は多項式
[2] vtype(A);
2                       <--- 属性は純関数
[3] args(A);
[x]                     <--- 引数リスト
[4] B=functor(A);
sin                     <--- 関数子
[5] type(B);
2                       <--- 関数子も多項式
[4] vtype(B);
3                       <--- 属性は関数子
```

この例では, `sin(x)` なる純関数が生成されているが, Risa object としては, `sin(x)` 自身が不定元, 即ち多項式となる. しかし `V` 構造体としての属性 (メンバ `attr`) の値が異なる. この値が純関数を意味するとき, メンバ `priv` にはこの純関数に関するさまざまな情報が格納され, `args`, `functor` により取得される. 関数子自身も, Risa object としては不定元として存在する. この場合も属性により関数子であることが示される.

5.4.3 プログラム変数

```
typedef struct oVS {    プログラム変数配列
  unsigned int n;       現在含まれる変数の個数
  unsigned int asize;   割り当てられた配列の長さ
  unsigned int at;
  unsigned int level;
  struct oFUNC *usrf;
  struct oPV *va;       配列本体
  NODE opt;             未使用
} *VS;

typedef struct oPV {    プログラム変数
  char *name;           変数名
  sort attr,type;
  pointer priv;
} *PV;

extern VS GPVS;   大域変数配列
extern VS CPVS;   現在の変数配列
```

```
extern VS EPVS;  extern 宣言された変数配列
extern VS APVS;  計算結果を格納する配列
```

Asir においては, プログラム変数のスコープは, 大域変数と, プログラム内での局所変数の 2 レベルに単純化されている. 変数は, 現れた時点でいずれかのプログラム変数として登録される. 関数の外で現れたプログラム変数は, 大域変数配列に登録される. 関数定義内で現れたプログラム変数は, 関数定義固有の局所変数配列に登録され, USRF 構造体のメンバ pvsとして登録される. 関数が実行される場合に, 登録された局所変数配列をテンプレートとして, スタックに相当する変数配列が生成され, 実行時の変数の値はこの配列に格納される.

中間言語において, プログラム変数参照はインデックスにより行われる. 関数内での変数参照は, 通常は局所変数配列内の変数に対するインデックスが用いられるが, extern 宣言されている変数に関しては, 同名の大域変数配列の変数に対するインデックスが用いられる. この場合, 実際にはこの区別はインデックスの最上位ビットを 1 にすることで行っている.

6 Interpreter

6.1 Interpreter の構成

interpreter は, 中間言語の構成要素である FNODE (式), SNODE (文) それぞれの解釈を行う関数群からなる.

6.1.1 式の解釈実行

```
pointer eval(FNODE f)
式 f の解釈実行

pointer evalf(FUNC f,FNODE a, FNODE opt)
関数 f の実行

struct oAFUNC {     算術演算テーブル
  void (*add)();    和
  void (*sub)();    差
  void (*mul)();    積
  void (*div)();    商
  void (*pwr)();    冪
  void (*chsgn)(); 符号反転
  void (*comp)();   比較
};

struct oAFUNC afunc[] = {
/* ??? */     {0,0,0,0,0,0,0},
/* O_N */     {addnum,subnum,mulnum,divnum,pwrnum,chsgnnum,compnum},
/* O_P */     {addp,subp,mulp,divr,pwrp,chsgnp,compp},
/* O_R */     {addr,subr,mulr,divr,pwrr,chsgnr,compr},
/* O_LIST */ {notdef,notdef,notdef,notdef,notdef,notdef,complist},
/* O_VECT */ {addvect,subvect,mulvect,divvect,notdef,chsgnvect,compvect},
/* O_MAT */  {addmat,submat,mulmat,divmat,pwrmat,chsgnmat,compmat},
/* O_STR */  {addstr,notdef,notdef,notdef,notdef,notdef,compstr},
/* O_COMP */ {addcomp,subcomp,mulcomp,divcomp,pwrcomp,chsgncomp,compcomp},
/* O_DP */   {addd,subd,muld,divsdc,notdef,chsgnd,compd},
};

void arf_add(VL vl,Obj a,Obj b,Obj *r)
*r = a+b

void arf_sub(VL vl,Obj a,Obj b,Obj *r)
*r = a-b

void arf_mul(VL vl,Obj a,Obj b,Obj *r)
*r = a*b

void arf_div(VL vl,Obj a,Obj b,Obj *r)
```

```
*r = a/b

void arf_remain(VL vl,Obj a,Obj b,Obj *r)
*r = a%b

void arf_pwr(VL vl,Obj a,Obj b,Obj *r)
*r = a^b

void arf_chsgn(Obj a,Obj *r)
*r = -a

int arf_comp(VL vl,Obj a,Obj b)
return 1 if a>b; -1 if a<b; 0 if a=b;
```

eval は, 必要に応じて自分自身を呼び出しながら式を評価する. 四則演算については算術演算テーブルに登録されている関数を, Risa object の識別子に従って呼び出す. 関数呼び出しがあった場合, evalf が呼び出される. 引数を式のリストとして評価して Risa object に変換したあと, それを引数として実際に関数の値の評価を行う. 関数が組み込みの場合, 組み込み関数を呼び出せばよい. 関数がユーザ定義関数の場合, 局所変数配列の生成, 仮引数への実引数の代入の後, 関数定義本体の文リストを, 次に述べる文の解釈実行関数で評価していく.

6.1.2 文の解釈実行

```
pointer evalstat(SNODE f)
```
文の解釈実行

文を解釈実行する場合, 文中の式は eval で評価する. また, 文が複文の場合には evalstat を再帰的に呼び出して評価する. evalstat の重要な仕事として, 'if', 'for' などの制御文の解釈実行がある. また, 次章で述べる debugger と協調して, ブレークポイント情報, ステップ実行情報の管理を行う. 以下で, 主なものについて, 実際にどのように文が解釈実行されるかを示す.

6.1.2.1 単文

```
case S_SINGLE:
  val = eval((FNODE)FA0(f)); break;
```

この場合, f の引数は一つの式であり, eval() を呼び出すことで式の値が計算される.

6.1.2.2 複文

```
case S_CPLX:
  for ( tn = (NODE)FA0(f); tn; tn = NEXT(tn) ) {
    if ( BDY(tn) )
       val = evalstat((SNODE)BDY(tn));
       if ( f_break || f_return || f_continue )
         break;
  }
  break;
```

この場合, f の引数は文リストを表す NODE である. よって, 各 NODE から順に文を取り出して, evalstat() を呼び出す. 各文を実行したあと, break, return, continue フラグをチェックし, どれかが on の場合には残りの文を実行しない.

6.1.2.3 If 文

```
case S_IFELSE:
  if ( evalnode((NODE)FA1(f)) )
    val = evalstat((SNODE)FA2(f));
  else if ( FA3(f) )
    val = evalstat((SNODE)FA3(f));
    break;
```

第 0 引数はダミー, 条件式リストである第 1 引数を evalnode() で評価する. evalnode() はリストの最後尾の式の値を返すので, その値が非零の場合第 2 引数を evalstat() で実行, 零の場合, もし第 3 引数が 0 でなければそれを実行する. else なしの場合, 第 3 引数は 0 である.

6.1.2.4 For 文

```
case S_FOR:
  evalnode((NODE)FA1(f));
  while ( 1 ) {
    if ( !evalnode((NODE)FA2(f)) )
      break;
    val = evalstat((SNODE)FA4(f));
    if ( f_break || f_return )
      break;
    f_continue = 0;
    evalnode((NODE)FA3(f));
  }
  f_break = 0; break;
```

第 0 引数はダミー, 第 1 引数は初期化のための式リストである. まずこれを evalnode() で実行したあと, ループを実行する. まず, 条件式リストである第 2 引数を評価し, その最後尾の式の値が零の場合, ループを抜ける. 非零の場合, ループ本体である第 4 引数を evalstat() で実行する. この実行において, break, continue, return 文が現れた場合, これらに対応するフラグが on になっている. C 言語における規定に従い, break, return の場合にはループを抜ける. さらに break の場合には, ループを一段抜けたことで役目を果たしているため, フラグを off にする. continue の場合には, 単に continue フラグを off にする. 最後にループの最後に実行する式リストである第 3 引数を実行してループの先頭に戻る.

6.2 ユーザ関数の実行

```
pointer evalf(FUNC f, FNODE a, FNODE opt)
関数 f を引数 a, option opt で実行
```

evalf() は組み込み関数, ユーザ関数共に実行できる. evalf() は式の解釈実行のサブルーチンであるが, ユーザ関数の実行は, 実際には文の解釈実行であり, また, プログラム変数の操作を含むやや複雑な手続きである.

```
  /* ユーザ関数の解釈実行 */
case A_USR:
  /* 引数リストを評価して LIST オブジェクトとする */
  args = (LIST)eval(a);
  /* ローカル変数 template */
```

```
pvs = f->f.usrf->pvs;
if ( PVSS ) {
/* 既に関数内の場合，その関数内での現関数呼び出しの位置の記録 */
  ((VS)BDY(PVSS))->at = evalstatline;
  level = ((VS)BDY(PVSS))->level+1;
} else
  level = 1;
/* スタックフレームを生成，リストに追加，現在の変数配列とする */
MKNODE(tn,pvs,PVSS); PVSS = tn;
CPVS = (VS)ALLOCA(sizeof(struct oVS)); BDY(PVSS) = (pointer)CPVS;
CPVS->usrf = f; CPVS->n = CPVS->asize = pvs->n;
CPVS->level = level; CPVS->opt = opts;
if ( CPVS->n ) {
  CPVS->va = (struct oPV *)ALLOCA(CPVS->n*sizeof(struct oPV));
  bcopy((char *)pvs->va,(char *)CPVS->va,
  (int)(pvs->n*sizeof(struct oPV)));
}
/* ステップ実行のためのレベルをアップデート */
if ( nextbp )
  nextbplevel++;
/* 仮引数に実引数を代入 */
for ( tn = f->f.usrf->args, sn = BDY(args);
  sn; tn = NEXT(tn), sn = NEXT(sn) )
  ASSPV((int)FA0((FNODE)BDY(tn)),BDY(sn));
/* 関数本体を実行 */
val = evalstat((SNODE)BDY(f->f.usrf));
/* フラグ，スタックフレームをリセット */
f_return = f_break = f_continue = 0; poppvs();
break;
```

6.3 デバッガ

ユーザ関数の実行は文を単位として行われるが, 文の実行前にデバッグモードに入る場合がある. デバッグモードでは, 以下のような機能が提供される.

- ブレークポイントの設定
- ステップ実行
- 変数の値の参照, 変更
- 関数の実行
- 関数呼び出し列の表示
- ソースのリスティング

6.3.1 デバッグモードへの移行

デバッグモードへの移行は次のような状況で起こる.

- debug 関数が実行された場合.
- キーボード割り込みに対するメニューで 'd' を選択した場合
- 実行中にエラーを起こした場合

- ブレークポイントに達した場合
- デバッグモードからステップ実行が指定された場合
  文の実行前にステップ実行フラグが調べられ, その値によりデバッグモードに移行する.
- 組み込み関数 error() が実行された場合

6.3.2 ステップ実行

デバッガにおけるステップ実行コマンドには step と next がある. これらはいずれも次の文を実行しようとするが, 次のような違いがある.

next　B 次の文がユーザ関数を含んでいても, 文ごと実行し, デバッグモードに戻る.

step　次の文がユーザ関数を含んでいた場合, 最初に評価されたユーザ関数の先頭でデバッグモードに戻る.

この機能を実現するために, 二つの変数が用意してある.

nextbp
: この変数が 0 でないとき, デバッガからステップ実行コマンドのもとで, インタプリタが実行中であることを示す.

nextbplevel
: ステップ実行中にユーザ関数実行に入るとき 1 増え, ユーザ関数から抜けるとき 1 減る.

また evalstat() の先頭では次のようなコードが実行されている.

```
/* ステップ実行中で nextbplevel が非正 */
 if ( nextbp && nextbplevel <= 0 && f->id != S_CPLX ) {
   nextbp = 0;
/* デバッグモードに入る */
   bp(f);
 }
```

これらによりステップ実行は次のように実現される.

next　nextbp = 1, nextbplevel = 0 として実行を継続する. 文の実行中に関数に入ると, nextbplevel が正となるので, 文中の関数実行中はデバッグモードに入らない. 文の実行が終って次の文の先頭に達したとき, nextbp = 0, nextbplevel = 0 となっているためデバッグモードに入る.

step　nextbp = 1, nextbplevel = -1 として実行を継続する. 文の実行中に関数に入っても, nextbplevel = 0 なのでデバッグモードに入る.

6.3.3 ブレークポイントの設定

ブレークポイントの設定は, 対象となる文の前に, ブレークポイント文 (文識別子 S_BP を挿入することで行う. ブレークポイント文は引数として対象となる文, ブレークポイントに入るための条件式を持つ. ブレークポイント文が実行された場合, 条件がないか, または条件式が真 (0 でない) 場合に nextbp = 1, nextbplevel = 1 として, 引数である文を実行する. 既に述べたことより, evalstat() の先頭でブレークポイントに入ることになる.

6.3.4 デバッグモードにおける変数の参照

現在実行中の関数は, その関数構造体へのポインタ (型 FUNC) が tagetf なる変数に登録されている. 現在実行中の関数がユーザ関数の場合, 対応するプログラム変数スタックは CPVS に登録されている.

トップレベルからその関数に至る呼び出し列に対応するプログラム変数スタックリストは, node 変数 PVSS に登録されている. 各プログラム変数スタックは型 oVS であり, ユーザ関数を示すメンバ usrf を持つ.

以上により, 現在実行中の関数に至る呼び出し列中の関数名, 各関数における変数の値などにアクセスすることができる.

7 メモリ管理

7.1 メモリ管理機構

risa におけるメモリ管理は, [Boehm,Weiser] によるものを用いている. このメモリ管理機構の特徴は, タグ付けを必要とせず, 自動的にガーベッジコレクション (GC) を行なうことである. 従ってユーザは必要な領域を取りっぱなしにしてよい. 欠点としては, 一回の GC ですべてのガーベッジを回収できるとはかぎらないことと, コンパクションを行なわないことであるが, 実用上十分な機能を持つ. メモリ割り当て器に現れるマクロはすべてこのメモリ管理のもとでメモリの割り当てを行なっている. GC は, その時点における, スタック, レジスタ, 静的領域からマーキングを始め, これにより到達できない領域をすべて回収する. コンパイラの最適化により, 最初領域の先頭を指していたポインタが, 領域の内部を指している場合にも, GC 正しくその領域が使用中と判断する.

注意すべきことは, 通常の malloc() により割り当てられた領域内はスキャンされないことである. よって, malloc() は, その他の C のライブラリの中から呼ばれる場合を除いて使用は避けなければならない. また, 一つの領域は, 複数の領域から参照されている可能性があるため, ユーザが開放することは危険である. ただし, 作業用のバッファなど, 明らかに他からの参照がないものに関しては開放して構わない. メモリ管理関係の主な函数は次の通り.

```
void GC_init()
初期化ルーチン. 起動時に実行する.

void *GC_malloc(int n)
n bytes の領域を割り当てる. 領域は 0 で初期化される.

void *GC_malloc_atomic(int n)
ポインタを含むことがないと保証される n bytes の領域を割り当てる.

GC_free(void *p)
p の指す領域を開放する. Risa では, ある領域がどこからどの位指されている
か一般には判断できないので, 通常はこの関数が使用されることはない.
関数内で割り当てたバッファの開放などに用いることはできる.
```

通常は GC_malloc() を使用するが, 多倍長数用の領域など, 内部にポインタを含まないことが分かっている領域用に GC_malloc_atomic() が用意されている. GC ルーチンは, GC_malloc_atomic() により割り当てられた領域の内部はスキャンしないので, GC の効率が良くなる.

なお, version 7 以降の GC を用いている場合, これらの関数を直接呼び出してはいけない. 必ず前後処理つきの関数 (Risa_をつけたもの) を呼び出すこと.

7.2 Risa におけるメモリの使用

Risa における各演算関数について共通の振舞いとして, 結果として生成される object の内部で, 入力である object の各部への参照が行われている可能性がある, ということがある.

次の例は, 多項式の加算関数である. この中で, 例えば先頭の項の次数が異なる場合には, 係数 (へのポインタ) がそのまま結果にコピーされている. また, 引数の一方の次数係数リストの末尾までたどった時に, 一方の次数係数リストが残っている場合には, その残りがそのまま結果の次数係数リストにつながれる.

```
#include "ca.h"

void addp(vl,p1,p2,pr)
VL vl;
P p1,p2,*pr;
{
  DCP dc1,dc2,dcr0,dcr;
  V v1,v2;
  P t;

  if ( !p1 )
    *pr = p2;
  else if ( !p2 )
    *pr = p1;
  else if ( NUM(p1) )
    if ( NUM(p2) )
      addnum(vl,p1,p2,pr);
    else
      addpq(p2,p1,pr);
  else if ( NUM(p2) )
    addpq(p1,p2,pr);
  else if ( ( v1 = VR(p1) ) ==  ( v2 = VR(p2) ) ) {
    for ( dc1 = DC(p1), dc2 = DC(p2), dcr0 = 0; dc1 && dc2; )
      switch ( cmpq(DEG(dc1),DEG(dc2)) ) {
        case 0:
          addp(vl,COEF(dc1),COEF(dc2),&t);
          if ( t )  {
            NEXTDC(dcr0,dcr); DEG(dcr) = DEG(dc1); COEF(dcr) = t;
          }
          dc1 = NEXT(dc1); dc2 = NEXT(dc2); break;
        case 1:
          NEXTDC(dcr0,dcr); DEG(dcr) = DEG(dc1); COEF(dcr) = COEF(dc1);
          dc1 = NEXT(dc1); break;
        case -1:
          NEXTDC(dcr0,dcr); DEG(dcr) = DEG(dc2); COEF(dcr) = COEF(dc2);
          dc2 = NEXT(dc2); break;
      }
    if ( !dcr0 )
      if ( dc1 )
        dcr0 = dc1;
      else if ( dc2 )
        dcr0 = dc2;
      else {
        *pr = 0;
        return;
      }
    else
      if ( dc1 )
        NEXT(dcr) = dc1;
```

```
      else if ( dc2 )
        NEXT(dcr) = dc2;
      else
        NEXT(dcr) = 0;
    MKP(v1,dcr0,*pr);
  } else {
    while ( v1 != VR(vl) && v2 != VR(vl) )
      vl = NEXT(vl);
    if ( v1 == VR(vl) )
      addptoc(vl,p1,p2,pr);
    else
      addptoc(vl,p2,p1,pr);
  }
}
```

このように, Risa の演算関数では, 一見不要になった中間的な結果でも, その部分式が最終結果に用いられていることがあるため, 注意が必要である. 特に, 配列を書き換える必要がある場合などには, 配列そのものを新しく生成して, 成分をコピーしてから用いる必要がある.

7.3 GC version 7 に関する注意

version 6 までの Boehm GC においては, GC に入る前にすべての signal をマスクし, 出たあとマスクをクリアする, という操作を自動的に行っていた. 結果として, GC 中に受けた signal は保留され, GC 中に signal により大域脱出するという危険はなく, また, GC 中の signal は GC 後に確実に処理することができた.

しかし, version 7 においてはこの仕組みが廃止された. この結果, 例えば, GC 中に SIGINT が処理されてしまうと, メモリ管理のテーブルが破壊され, それ以降予期せぬ状態になる可能性がある. 実際, 大域変数に保持されているデータが参照できなくなったり, 書き換わったりということが起きた.

これに対する対処法として, `GC_malloc()` などの割り当て関数を直接呼び出さず, これらに対する前後処理を行う関数を呼び出すようにした. 例えば `GC_malloc()` に対する関数は次のようになる.

```
int in_gc, caught_intr;

void *Risa_GC_malloc(size_t d)
{
  void *ret;

  in_gc = 1;
  ret = (void *)GC_malloc(d);
  in_gc = 0;
  if ( caught_intr ) { caught_intr = 0; int_handler(); }
  if ( !ret )
    error("GC_malloc : failed to allocate memory");
  return ret;
}
```

in_gc が 1 のとき, GC 中であること示す. また, caught_intr が 1 のとき, in_gcが 1 の間に SIGINT を受けたことを表す. この場合, SIGINT handler である int_handler() では, 単に caught_intr を 1 にして何もせずに return する.

```
extern int ox_int_received, critical_when_signal;
extern int in_gc, caught_intr;

void int_handler(int sig)
{
  extern NODE PVSS;
  NODE t;


  if ( do_file ) {
    ExitAsir();
  }
  if ( critical_when_signal ) {
    ox_int_received = 1;
    return;
  }
  if ( in_gc ) {
    caught_intr = 1;
    return;
  }
  ...
```

GC 中に SIGINT があった場合, in_gcが 0 となったあとに, caught_intr を 0 にして SIGINT 処理関数 int_handler() を呼び出す. このようにすることで, version 6 と同様に, GC 内で受けた SIGINT の処理の保留および GC 後の処理を行うことができる.

なお, OX server として動作中には SIGINT の他に SIGUSR1 を受け取る可能性がある. この場合, 中断できない OX 通信関数の前後で begin_critical(), end_critical()が実行され, ciritical_when_signal, ox_usr1_sent により SIGUSR1 処理の保留が実現されている.

```
int ox_usr1_sent, ox_int_received, critical_when_signal;
void ox_usr1_handler(int sig)
{
  NODE t;

  signal(SIGUSR1,ox_usr1_handler);
  if ( critical_when_signal ) {
    fprintf(stderr,"usr1 : critical\n");
    ox_usr1_sent = 1;
  } else {
    ox_flushing = 1;
    ...
    ox_resetenv("usr1 : return to toplevel by SIGUSR1");
  }
}

void begin_critical() {
```

```
  critical_when_signal = 1;
}

void end_critical() {
  critical_when_signal = 0;
  if ( ox_usr1_sent ) {
    ox_usr1_sent = 0;
  ox_usr1_handler(SIGUSR1);
  }
  if ( ox_int_received ) {
    ox_int_received = 0; int_handler(SIGINT);
  }
}
```

この場合, OX server 内での GC 中に受け取った SIGINT をどうするかという問題が生ずる. これについては, OX server 内での GC 中に受け取った SIGINT は `ox_int_received` に記録するだけとする. (`int_handler()` 内で, `in_gc` より先に `critical_when_signal` を見ていることに注意.) 結果として, 対応する `int_handler()` の処理は, 次に `end_critical()` が呼ばれたときについでに実行されることになり, 即応性がなくなるが, SIGINT は 手動でのみ送られるので, この点は気にする必要はない.

前後処理つきのメモリ割り当て関数 '`parse/gc_risa.c`' で定義されている.

```
void *Risa_GC_malloc(size_t d)
void *Risa_GC_malloc_atomic(size_t d)
void *GC_malloc_atomic(int n)
void *Risa_GC_malloc_atomic_ignore_off_page(size_t d)
void *Risa_GC_realloc(void *p,size_t d)
void Risa_GC_free(void *p)
```

それぞれ, `Risa_` がない関数に対応している. また, '`include/ca.h`'内のメモリ割り当て関連マクロも, すべてこれらの前後処理つき関数を呼ぶように変更した.

ChangeLog: 以上の変更については OpenXM_contrib2/asir2000 の以下のファイルとその一つ前の版を比較せよ. parse/gc_risa.c 1.11 (処理の本体), parse/asir_sm.c 1.8, parse/glob.c 1.83, parse/load.c 1.22, parse/ytab.c 1.11, builtin/array.c 1.61, builtin/gc.c 1.68, engine/nd.c 1.200, io/sio.c 1.25, io/tcpf.c 1.62.

Appendix A 文献

[Boehm,Weiser]
"Garbage Collection in an Uncooperative Environment", Software Practice & Experience, September 1988, pp. 807-820.

Table of Contents