

オブジェクト指向論理型言語 Common ESP の言語機能

佐藤 泰典 伊藤 民哉 中澤 修
(株)AI 言語研究所

概要

オブジェクト指向言語 Smalltalk-80 が発表されて以来、オブジェクト指向の考え方は、C, Lisp, Prolog などの多くのプログラミング言語に組み入れられ、様々な融合言語の設計が行われている。オブジェクト指向機能導入の動機の一つには、オブジェクトの概念に基づく高度なモジュール化機能の活用があり、実際にプログラムの部品化／再利用を考慮することにより大規模なプログラムの作成が可能となってきた。本稿ではオブジェクト指向論理型言語 Common ESP の言語機能の中から、大規模プログラムの作成に有効な種々のモジュール化機能、大域脱出などの実行順序制御機能および C などの他言語ルーチンとの融合を目指した他言語インタフェース機能について述べる。

Some Functions of the Object-Oriented Logic Programming Language Common ESP

Yasunori Satoh, Tamiya Itoh, Osamu Nakazawa

AI Language Research Institute, Ltd.
Sakurai-Building/3F, Shiba 3-15-15,
Minato-Ku, Tokyo, 105 Japan

Abstract

Since the Smalltalk-80 was developed, the object-oriented paradigm is introduced into C, Lisp, Prolog and other languages. As a result, many multi-paradigm languages are designed. One of the motivation to include object-oriented functions is to provide a function for modularizing program on the conceptual basis of "object." This enables to compose a large scale program with re-utilizing shared programs made as objects for building parts. In this report, we describe some of the language functions of our object-oriented logic programming language Common ESP, various modularization functions effective to build a large program, execution ordering functions e.g. global jumping back, and foreign language interface function intended to enable links with those programs written in C or other languages.

1 はじめに

Common ESP(以下 CESP)は, ESP(Extended Self-contained Prolog)[1][2]を改良拡張したプログラミング言語である。言語のベースには ESP と同様, オブジェクト指向と論理型の融合があり, それらの 2 大パラダイムに種々の機能を組み入れている。また, 環境込みの言語として, 開発 / 実行環境の性能, 使い易さ, そして, 普及性を追究している。

ESP は AI 手法を利用したシステムの記述に向いた言語として優れた機能を持っているにもかかわらず, PSI(Personal Sequential Inference machine)と呼ばれる専用機上でしか稼働しない。これは言語の普及ということを考えた場合, 大きな障害となってくる。そこで CESP は実行性能と共に移植性も重視し, 専用の機械語ジェネレータジェネレータを開発することにより, ポータブルな処理系の作成を目指した [3]。実際, 91 年 3 月時点で, アーキテクチャの異なる 8 社のマシン上での動作が確認されている。

また, オブジェクト指向機能を持った論理型言語としては, 実用的な応用プログラムの実行にも遜色のない実行効率を実現でき [4], 柔軟な他言語インタフェース機能と合わせて活用することで大規模なプログラムの開発も可能となっている [5][6][7][8][9]。

もちろん, 実行効率の考慮だけではなく, CESP を実用的なプログラミング言語とするためには, CESP の論理型言語としての面を支えている Prolog の機能に加えて, 大域実行制御, 例外ハンドリング, 高度なモジュール化, 他言語との動的リンク, グローバルデータなどの表現に使える効率的な副作用機能などをサポートする必要があった。これらの機能のいくつかは Prolog の世界においても, ISO の SC22 Prolog 標準化委員会で, block & exit_block による大域脱出, on_exception による例外処理, 述語ベースにアトムの可視性制御を組み込むようなモジュール化機能などとして検討中 [10]であり, CESP での拡張方向が多くの Prolog ユーザ / 開発者の思想とも一致していると考えられる。

本稿では, 大規模なプログラムの作成という観点から CESP に採り入れた機能のうち, まず各種モジュール化機能の概要を述べ, 次いでそれらのモジュール定義の中から,

- 論理型に関する実行制御機能
- 他言語インタフェース機能

に焦点を当てて説明する。また, 最後に現在検討中のオブジェクトの永続機能に関する紹介も行なう。

2 モジュール化

CESP のプログラムモジュールには, 「オブジェクト」と「バンク」と呼ばれるものがある。「オブジェクト」はデータとそれに対する処理をひとまとめにしたもので, プログラム上ではクラスがその単位になる。「バンク」は, クラスに対するさまざまな付随情報を記述したもので, 目標とするクラスと対をなして使用する。

2.1 クラスとは

クラスは CESP におけるプログラムの最小単位で

- 通常の処理を定義するクラス
- 他言語との双方向呼び出しを定義するクラス
- 例外発生時の処理を定義するクラス

に分けることができる。以下にその概要を示す。

2.1.1 通常のクラス

一般的な CESP のクラス定義は

```
class <クラスの名前> has
    <スロットの定義>
    <メソッドの定義>
end.
```

の形式で記述する。<メソッドの定義>は一般にインタフェース定義と内部処理定義に分けることができる。インタフェース定義は Prolog のクローズで言うヘッド部に, 内部処理定義がボディ部に対応する。また, <スロットの定義>は当該クラス中で使用する副作用データなどを保存するための変数宣言と考えられる。

2.1.2 他言語インタフェースクラス

他言語バンクと呼ばれる他言語の呼び出し情報を用いて他言語プログラムとの相互呼び出しを行なうためのクラスである。“with_foreign”というキーワードにより適用する他言語バンクを指定し,

```
class <他言語インタフェースクラスの名前>
with_foreign <他言語バンク名> has
    <スロットの定義>
    <他言語メソッドの定義>
end.
```

と記述する。<他言語メソッドの定義>には, Call out interface method と Call in interface method という 2 つのメソッドが定義できる。前者は CESP から他言語プログラムを呼び出す場合に用い, 後者

はその逆の場合に用いる（他言語バンクについては後述）。

2.1.3 例外処理クラス

プログラムの実行中に生じた例外の後処理を記述するためのクラスである。クラス定義は

```
class <例外処理クラスの名前> has
  <スロットの定義>
  <例外処理メソッドの定義>
end.
```

と書き、例外が発生したときの後処理を例外事象の名称に対応したメソッドとして定義しておく。例外発生時には、その例外発生のゴールが当該メソッドに置き換えられ、処理が継続することになる。なお、このような例外処理環境（例外処理クラス）の設定には、

```
set_exception_object(例外処理クラス,
                     ゴール列)
```

という組込述語を用いる。第二引数のゴール列の実行中に発生した例外は、ここで設定した例外処理環境の定義に従って処理が行われる。また、ゴール列の実行の終了に伴い、登録した例外処理環境は消滅する。

上記の3タイプのクラスは相互に継承/参照が可能であり、プログラム時の難しい制約などは特にない。また、<メソッド>と<スロット>は、クラス定義により自動的にできるオブジェクトであるクラスオブジェクト、および、クラスオブジェクトに「:new/2 メッセージ」を送ることで作られるインスタンスオブジェクトの両者に対して個別に定義することができる。つまり、CESPの場合、クラス自身もオブジェクトであり、インスタンスとの間に利用上のギャップはない。

2.2 バンクとは

CESP のもうひとつのプログラムモジュールとしては、クラスに対する補足情報を記述するバンクモジュールがある。バンクには、

- プログラム中で使用するマクロの定義
- 他言語ルーチンとの呼び出し情報の定義
- メソッド単位の例外処理定義

に対応する3つの異なったモジュールがある。以下に概要を示す。

2.2.1 マクロバンク

クラス定義の中で使うマクロを記述しておくモジュールである。マクロの定義は、

```
macro_bank <マクロバンク名> has
  <マクロの定義>
end.
```

と書く。マクロを使うクラス側では、“with_macro”というキーワードを用い、利用するマクロバンクの宣言を行なう。なお、マクロバンク中で使用するマクロバンクの宣言、親マクロバンクの宣言も可能である。

マクロの機能としては、プログラムの単なる置き換えに留まらず、述語列や節の挿入までも可能になっている。また、ユニフィケーション機能が使用できるので、簡潔で可読性の高いマクロ定義が記述できる[11]。

2.2.2 他言語バンク

CESP から呼び出す他言語プログラムの情報や、他言語側から呼び出す CESP メソッドに関する情報を記述するモジュールである。“foreign_bank”というキーワードを用い

```
foreign_bank <他言語バンク名> has
  <他言語情報の宣言>
  <call_in/call_out 情報の定義>
end.
```

と書く。<call_in/call_out 情報の定義>には、CESP と他言語プログラムとでやり取りをするデータの変換規則を記述する。ここに書かれたデータ変換規則により、データ内部表現をプログラム実行時に必要に応じて自動変換する。また、<他言語情報の宣言>には、他言語の名称、実行に必要なオブジェクトファイルの名称およびライブラリファイル名などの宣言を記述する。

2.2.3 エラーバンク

エラー処理とプログラムのメイン処理部分を分離し、エラー処理のみを一括して記述するためのモジュールである。エラー処理は「このメソッド/述語内で例外が発生したら、この処理を実行せよ」という形式の<エラー処理情報>を

```
error_bank <エラーバンク名> has
  <エラー処理情報>
end.
```

と定義する。前述の例外処理クラスでは例外の種類に応じて処理を定義したが、エラーバンクでは述語に対

して例外発生時の処理を定義できるようになっている。なお、メイン処理とエラー処理の合体、対話的な例外処理環境であるシチュエーションと呼ばれる機構への登録は、プログラムのコンパイル時に自動的に行われるため、必要なくなった時点でエラー処理を解除することも簡単に行える。

以上、CESP のプログラムモジュールに関して簡単な紹介を行なった。これ以外にもプログラムの説明機能などをサポートするためのドキュメントバンクモジュールも用意している。

このような高度なモジュール化機能を導入することで、大規模ソフトウェアの効率的な開発、さらには複数人でのサブシステム分割による開発が可能となったが、クラス名などのモジュールの名称は一意に決定される必要があるため、クラス名の重複が問題となる。そこで、クラスのモジュール空間の上位にパッケージというクラス名アトムモジュール空間の機構を導入して、多重クラス名空間を実装している。

パッケージはクラスの集合であり、クラス名はパッケージ内で一意のものとしている。このような多重クラス名空間の導入により、CESP 世界においてクラスを一意に決定するためにはパッケージ名とクラス名のペアが必要になるが、パッケージの集合であるパッケージ環境の機構と、デフォルトパッケージというものを用意し、ユーザがパッケージをそれほど意識することなくプログラミングを行えるようにしている。

3 実行制御機能

通常の論理型言語の枠のなかだけではプログラムの実行順序が単純なため、実用的なプログラムを実行するには機能上不十分な場合がある。CESP では大域実行制御などの実行順序制御機能を導入し、大規模な応用プログラムの作成を支援している。ここでは、CESP の代表的な実行順序制御機構である大域脱出と例外処理 [12] について述べる。

実用的な応用プログラムでは、常に例外処理に備えたプログラミングが必要である。例外の多くは実行時に発生するため、プログラミング時に処理の各段階ごとの例外処理を書いておく必要がある。また、例外の発生にもかかわらず強制的に処理を継続させたい場合や、例外発生時までの処理の結果を生かしておきたい場合などもある。これらの要求に応えるために、CESP では、

- catch & throw
- on_backtrack

● 例外処理機構

などを提供している。

3.1 catch & throw

多段にネストした述語群の中で、深いレベルから途中の複数の述語を飛び越えて浅いレベルへ戻る機能である。

catch(識別子, ゴール列, 脱出後ゴール列)

という組込述語で戻る場所を指定しておけば、

throw(識別子)

という組込述語を用いて、対応する識別子のところへ戻ることができる。

実行時の動作としては、まず catch/3 の第二引数のゴール列を実行し、その中で throw/1 の呼び出しがあれば実行を止め catch/3 の地点まで戻り、脱出後ゴール列を実行することになる。

3.2 on_backtrack

バックトラック時にのみ実行させたいプログラムを指定できる。一般的には、

..., (true; ゴール列, !, fail), ...

と記述しておけば選択肢を残すことが可能であるが、この記述以降にカットを書くことができない。この記述を

..., on_backtrack(ゴール列, 識別子), ...

としておくことにより、カットの有無にかかわらずバックトラック時に実行するゴール列を指定できる。二引数目の「識別子」は、on_backtrack 処理を解除したい場合に、

reset_on_backtrack(識別子)

で指定するためのもので、実行時に決定されるものである。

前述の throw/1 では on_backtrack 指定も無視してしまうが、rollback/1 を使用すれば catch の地点まで戻る途中の on_backtrack 処理をすべて実行することが可能である。

3.3 例外処理機構

例外処理クラスやエラーバンクにより例外処理環境という環境を設定し、発生した例外をつかまえて処理を行う機構である。

例外処理環境の設定はネスト構造になっている。一番中心にシステム定義の例外処理環境があり、それを外から包むようにユーザ定義の例外処理環境が設定される。

例外処理クラスのメソッド名は例外事象の名前となっており、例外事象が発生すると、まず、発生したゴールの置き換えを行い、次に、例外処理オブジェクトで定義された処理に従って例外処理を行う。

たとえば、引数のタイプミスマッチの例外が発生しても引数を置き換えることによって処理を継続したいような場合には、前述した例外処理クラスの中に例外の名称に対応したメソッドを

```
:type_mismatch(_オブジェクト,  
    _違反引数, _例外情報) :-  
    「引数の値を置き換えて継続する」
```

と定義し、その例外処理クラスを例外処理環境として設定しておけばよい。

もちろん、このような定義を行わなければ、処理系として標準に設定した例外処理環境で処理を行なうことになる。標準の環境は、シチュエーションによる対話的な管理機構で

- 例外の発生した述語を失敗として継続する
- 例外発生の詳細情報を表示する
- 処理系のトップレベルへ戻る

などの処理がメニュー中から選択できる。なお、このシチュエーション管理に対してエラーバンク定義を行なうことで、新規のメニュー項目を追加することも可能である。

4 他言語インタフェース機能

CESP の他言語インタフェース機能 [13] の設計は、以下のような思想に基づいて行なった。

- CESP に向かない数値解析などの処理をそれに適した言語に簡単に置き換えができること
- これまでに蓄えられた資産が利用できること
- 既存システムへの組み込みが容易にできること

これらの目標を実現するため、我々は CESP の他言語インタフェースに必要な機能として

- CESP と他言語とのデータ変換の自動化
- CESP 側あるいは他言語側からの双方向の呼び出し機能

- 他言語に関する情報バンク定義とクラス定義との分離

を挙げた。

CESP 側では他言語クラスというものを用意し、CESP と他言語との結合部をクラスオブジェクトの枠の中に閉じてこめている。インタフェース部分にオブジェクトの皮を被せることにより他言語側の変更の影響が CESP 側まで及ばない仕組みになっている。

他言語クラスは以下の定義から構成される。

- CESP メソッドとして呼び出す他言語ルーチンの定義
Call out interface method と呼ばれるメソッドで他言語ルーチンとの対応を定義する
- 他言語ルーチンから呼び出す CESP メソッドの定義
Call in interface method と呼ばれるメソッドで他言語ルーチンから呼び出されるメソッドの定義を通常のメソッドと同形式で定義する

- 他言語バンクの宣言

また、他言語バンクは以下の定義から構成される。

- Call out 記述
データ型変換の規則、引数の入出力モード、関数値受け取り型 / 関数値不必要型の指定（もちろん、データを裸で見せる機能も必要）
- Call in 記述
引数の入出力モード、他言語側のプログラミングの方法、cespFLImethod、cespFLIredo など他言語側から CESP を呼び出すための記述
- 型宣言
複雑な構造を持つデータののための仮想的な型宣言機能で、他言語とのデータのやりとりをスムーズに行なえるための記述（C 言語の typedef 宣言に対応）

なお、バンクもクラスと同様に継承機能を持つので、バンク自身の部品化 / 再利用も可能である。以下に、Call out と Call in の例をあげる。

【コールアウト例】

2 × 3 行列と 3 × 4 行列の乗算を行うクラスを定義してみる。CESP ではベクタで表した行列を他言語側へ渡すのみで、演算は他言語側で行う。

```
class matrix has  
:multiply2334( _, X, Y, Z ) :-
```

```

multiply( #matrix2334, X, Y, Z );
end.

```

```

class matrix2334
with_foreign matrix has
:multiply( Class, X, Y, Z ) :-
    &multiply_mat( Class, X, Y, Z );
end.

```

```

/* "foreign_bank"に
** 他言語へ渡す引数の型を定義しておく */
foreign_bank typedef2 has
type_of
    vct3 = array_of( float, 3 ),
    vct4 = array_of( float, 4 ),
    mat23 = array_of( vct3, 2 ),
    mat24 = array_of( vct4, 2 ),
    mat34 = array_of( vct4, 3 );
end.

```

```

/* "typedef2"というバンクを継承して
** call out interface methodを定義する.
** 結果は関数値として返る */
foreign_bank matrix has
nature typedef2;
language c;
link_file "matrix.o";
export
multiply_mat( _, X:(+mat23),
              Y:(+mat34), Z:(=mat24) ) ->
    Z = multiply_mat( X, Y );
end.

```

```

/* 他言語側で配列の要素を取り出し演算を行う */
static double z[2][4];
double **multiply_mat( x, y )
double x[][3];
double y[][4];
{
    int i, j, k;
    double s;

    for ( i = 0; i < 2; ++i )
        for ( j = 0; j < 4; ++j ) {
            s = 0.0;
            for ( k = 0; k < 3; ++k ) {
                s += x[i][k] * y[k][j];
            }
        }
    }

```

```

        z[i][j] = s;
    }
    return( z );
}

```

【コールイン例】

簡単なローマ字漢字変換を行ってみる。CESP 側にローマ字と漢字の対応ルールを記述しておき、選択肢を残しながら候補を検索する構造にしている。

```

class ローマ字漢字変換クラス
with_foreign ローマ字漢字変換バンク
has
: 他言語プログラムコール( _ クラス ) :-
    &mainLoop( _ クラス );
instance
:romaji_to_kanji( _ オブジェクト,
    _ ローマ字, _ 漢字 ) :-
    ローマ字漢字変換述語( _ ローマ字,
        _ 漢字 );

```

```

local
ローマ字漢字変換述語( a, gs#"亜 ");
ローマ字漢字変換述語( a, gs#"阿 ");
ローマ字漢字変換述語( a, gs#"啞 ");
ローマ字漢字変換述語( a, gs#"あ ");

ローマ字漢字変換述語( i, gs#"意 ");
ローマ字漢字変換述語( i, gs#"胃 ");
ローマ字漢字変換述語( i, gs#"射 ");
ローマ字漢字変換述語( i, gs#"い ");
end.

```

```

/* 他言語からCESPを呼び出す口
** である"import"と、CESPから
** 他言語を呼び出す口である"export"
** を定義する */
foreign_bank ローマ字漢字変換バンク
has
language c;
link_file "romajiKanji.o";
export
mainLoop(Class:(+class_object)) ->
    mainLoop(Class) : logical;
import
new((+class_object),
    (-instance_object));
romaji_to_kanji((+instance_object),
    (+atom), (-kanji));
end.

```

```

#include "cespFLI.h"
#define YES 1
#define NO 0
LOGICAL mainLoop(classObject)
OBJECT classObject;
{
    OBJECT instanceObject;
    LOGICAL sts;
    char romaji[10], kanji[10];

/* :new/2を呼び、インスタンス
** オブジェクトを生成する */
    sts = cespFLImethod(
        classObject,
        "new",
        2,
        &instanceObject);
    if (!CESPFLIisSuccess(sts))
        return(sts);

    while (1) {
        printf("romaji>");
        *romaji = getchar();
/* ! が入力されたら終了 */
        if (*romaji == '!')
            CESPFLIisSuccess;
        sts = cespFLImethod(
            instanceObject,
            "romaji_to_kanji",
            3,
            romaji,
            kanji);
/* 次候補 (オルタナティブ)
** がある間ループする */
        while (CESPFLIisSuccess(sts)) {
/* 1つの漢字を選択したら
** ループをぬける */
            if (selectKanji(kanji)
                == YES)
            {
                printf("kanji = %s\n",
                    kanji);
                break;
            }
        }
/* さもないければ
** オルタナティブを求める */
        sts = cespFLIredo();
    }
}

```

```

    }
}

int selectKanji(kanji)
char *kanji;
{
    int ch;

    printf("%s ?", kanji);
    ch = getchar();
    return((ch == 'y' || ch == 'Y')
        ? YES : NO);
}

```

5 オブジェクトの永続性

インスタンスは、クラスという型から動的に生成される一時オブジェクトである。プロセス生存中のみインスタンスも生存しており、また、複数プロセス間での共有もできないものである。

しかし、知識表現などの分野では、せっかくオブジェクトとして作った (表現した) 知識を保存しておきたい場合や、複数の人 (プロセス) の間で共用したい場合もある。そこで、CESP としては以下のような2つのアプローチを考えた。

5.1 永続オブジェクト

オブジェクトに永続性を持たせ、一度生成したクラスオブジェクトおよびインスタンスオブジェクトを存続させておく機能である。なお、当機能は現在インプリメント中のものである。

1. 永続オブジェクトの機能

永続性を実現するための特別なシステムクラス “as_persistent_object” を継承するオブジェクトを永続オブジェクトとする。永続オブジェクトは二次記憶上への save と load という機能を持つ。save 対象はオブジェクトの属性 (スロット) であり、メソッドつまりプログラムコードに関しては CESP 処理系の持つプログラムベース管理 [14] へ任せる。

2. データベースインタフェース

save した永続オブジェクトを複数のプロセスで共有するため、二次記憶上のファイルと CESP の間に、サーバとなる仮想データベース管理システムを設定する。データへのアクセスは、CESP 上の永続オブジェクト管理からの要請により、仮

想データベース管理システムが永続オブジェクトデータの save & load などの処理を行うことでなされる。基本的な永続オブジェクトデータの操作はクライアントとなる CESP システム側で管理できるようにし、種々のデータベースとの結合が可能な構成とする。機能面からみると、検索 / 修正など言語側が得意とする機能は言語側に、データの整合性などデータ管理を行う部分のみをデータベース側に受け持たせようという構成である。

5.2 インスタンス記述

永続性ということではなくても、プログラミング時すでにオブジェクトの状態が判明しているインスタンスもある。そこで、名前付きインスタンスのプログラム定義が可能なように、インスタンス記述 (instance_of) というものを導入した。プログラミング時に名前を付けたインスタンスを定義しておくことにより、実行時に任意のインスタンスを特定できる。

インスタンス記述は、

```
instance_of <インスタンス名> :-  
    <インスタンススロットの定義>
```

という形式で書く。具体的には以下のようになる。

```
class nissan has  
instance_of  
( skyline :-  
    車の名前 := skyline_GTR,  
    ドア := 2,  
    エンジン := '2600cc',  
    色 := black ),  
( silvia :-  
    車の名前 := シルビア,  
    ドア := 2ドア,  
    色 := 銀色 ),  
( サニー :- 車の名前 := sunny );  
instance  
component 車の名前, ドア, エンジン, 色;  
end.
```

“instance_of” 以降の記述がインスタンス定義となり、この場合、3つの名前付きインスタンス skyline, silvia, サニーを定義したことになる。

なお、この記法により定義したインスタンスは「クラスオブジェクト@インスタンス名」としてクラス定義中で参照が可能である。たとえば、“silvia” という名前のインスタンスを取り出したいときは、#nissan@silvia と書けばよい。

6 おわりに

本稿では大規模ソフトウェアの開発に必要な CESP の言語機能のいくつかを紹介してきた。

現在、CESP 言語 / 言語処理系は AIR と利用契約を結ぶことで、研究目的に限り、無償配布を行なっており [15]、91 年 3 月末時点で約 240 サイトからの依頼がある。配布可能な処理系としては sun3, sun4(SPARC) の機械語対応版とエミュレータ版があり、実行環境としては GNU Emacs と X-Window R4 に対応している [16]。将来は自然言語処理パッケージ (Language Tool Box) などと合わせてリリースする予定もある。

今後の研究課題としては Lightweight Process 環境 (マルチスレッドプログラミング) などを検討している。CESP ではすでに UNIX のプロセスに対応した、fork 機能によるマルチプロセス環境と socket 通信機能を用いた server/client 型の処理を行う機能は提供している [17] が、軽いプロセスを作って、手軽なルーチンが書きたいような場合には向かない。そこで、プログラミング環境自身も LWP 対応にし、これから一般的になるであろうマルチプロセッサ構成のマシンとの親和性も高めていきたいと考えている。

また、永続オブジェクトの機能とも共通するものがあるオブジェクトの再利用 / 部品化の機能を取り入れることも検討している。

参考文献

- [1] Chikayama, T.: Unique Features of ESP, Proceedings of FGCS'84, ICOT (1984).
- [2] 近山 隆: ESPer への道, bit, Vol.22, No.1, 共立出版 (1990).
- [3] 佐藤 良治他: Common ESP 機械語生成系の自動生成, IPSJ 第 41 回全国大会, 6E-1 (1990).
- [4] 佐藤 良治他: Portable Mapping of Common ESP to Conventional Architectures, 第 61 回記号処理研究会, 91-SYM-59-3 (1991).
- [5] 藤瀬 哲朗他: Common ESP 上の LTB について, IPSJ 第 42 回全国大会, 5M-2 (1991).
- [6] 平島 保彦他: CESP 言語実証評価のための回路部品検索支援システムの開発, IPSJ 第 42 回全国大会, 5M-3 (1991).
- [7] 藤井 良直他: Common ESP によるエキスパートシステム構築 - 漢方薬処方支援システムの試作, IPSJ 第 42 回全国大会, 5M-4 (1991).

- [8] 川越 恭二他: 論理型オブジェクト指向プログラミング言語 CESP による UNIX コンサルテーションシステムの開発 - 基本方式 -, IPSJ 第 42 回全国大会, 5M-5 (1991).
- [9] 川越 恭二他: 論理型オブジェクト指向プログラミング言語 CESP による UNIX コンサルテーションシステムの開発 - 実現方式と評価 -, IPSJ 第 42 回全国大会, 5M-6 (1991).
- [10] Scowen, R.S.: PROLOG Draft for Working Draft 4.0, ISO/IEC WG17, N64 (1990).
- [11] Kondoh, S. and Chikayama, T.: Macro Processing in prolog, Fifth International Conference Symposium on Logic Programming (1988).
- [12] 佐藤 泰典他: ESPer への道, bit, Vol.22, No.7, 共立出版 (1990).
- [13] 伊藤 民哉: Common ESP における他言語インタフェース機能の実際, INAP'90, 6-3 (1990).
- [14] 佐藤 泰典他: Common ESP におけるプログラム管理システム, IPSJ 第 39 回全国大会, 6N-8 (1990).
- [15] AI センターだより別冊, AI センターだより, No.17, ICOT-JIPDEC AI センター (1990).
- [16] CESP マニュアル, CESP ライブラリ 2.0 版 (1990).
- [17] 萩原 つね子他: Common ESP のマルチプロセス環境, IPSJ 第 42 回全国大会, 5M-7 (1991).