解　説

オブジェクト指向論理型言語 Common ESP†

中澤　修††　近山　隆†††

1. はじめに

オブジェクト指向機能をもった論理型言語ESP[1]が第五世代コンピュータプロジェクトから生まれて6年，さらにESPの誕生に大きな影響を与えたオブジェクト指向言語Smalltalk-80[2]が世に発表されてから10年がたつ．この間にオブジェクト指向の考え方は多くのプログラミング言語に取り入れられ，いろいろな融合言語の設計が行われている．

オブジェクト指向との融合の大きなメリットの一つは，オブジェクト指向概念に基づくモジュール化機能による大規模ソフトウェアの生産性向上にあり，その有益性への理解も年々深まっている．しかし，一言で融合と言い表しても，その形態はさまざまであり，「メッセージ送信の対象となるオブジェクトは実行時に決まる」というオブジェクト指向の代表的な特徴を積極的には利用しない融合形態も現れている．

一方，従来標準的なモジュール化機能ももたない論理型言語にもオブジェクト指向によるモジュール化機構を導入する試みが増えている[3)~6)]．これらの多くはESPと同じように「公理系＝オブジェクト」というアプローチを採用し，Prologプログラムつまりホーン節の集合をオブジェクトとみなすことによって，プログラムのモジュール化・階層化を行う．こうしたモジュール化機構のもたらす良好な記述性は逐次型推論マシンPSIの数十万ステップにわたるオペレーティングシステム[7)]をESPにより短期間に記述できたこと，さらにPSI上の各種の大規模AIプログラムが容易に記述できた経験などから明らかであろう．

このような高水準言語ESPを実用的な効率で処理できるのは，ESP向けに最適化した逐次型推論マシンPSIの性能およびESP処理系の効率の良い実装法に負うところが大きかった．しかし，ESPで採用したオブジェクト指向と論理型言語の融合技術は，PSIのような専用機上のAIプログラムだけでなく，汎用機上の広範囲の応用にも有効である．特に，RISCアーキテクチャの登場以来，汎用ワークステーションなどの実行性能の向上は著しく，また，それにともないプログラム解析や最適化といった一般的なコンパイル技術が進歩し，論理型言語の抽象実行技術との組み合わせにより，論理型言語についても実行効率上の欠点を克服できつつある．そこで，拡張部分であるオブジェクト指向機能の実現いかんでは，十分実用に耐えうる言語処理系が作れる．

その実装の一例がCommon ESP[8)]（以下，CESPと略す）である．CESPはPSI上のESPの仕様を汎用機上での実現に合った言語仕様/言語処理系へと改良拡張した言語である☆．

本稿では，大規模ソフトウェアの開発を可能とするため論理型言語に施した拡張機能を中心にCESPの解説を行う．また，プログラミング環境さらに種々のハードウェア上での実装を目的としたポータビリティのある処理系の構成方法についても簡単に説明を加えることにしたい．

2. オブジェクト指向機能

最も広く使われている論理型言語であるPrologの種々のバリエーションの中で，現在まで標準仕様とみなされてきたものにEdinburgh版Prologがある．しかし，Edinburgh版Prologにはプロ

† Object-Oriented Logic Programming Language Common ESP by Osamu NAKAZAWA (AI Language Research Institute, Ltd, The First Research Lab.) and Takashi CHIKAYAMA (Institute for New Generation Computer Technology, The Second Research Lab.).

†† (株)AI言語研究所第一研究室
††† (財)新世代コンピュータ技術開発機構第二研究室

☆ CESPは第五世代コンピュータプロジェクトの成果を基に，基盤技術研究促進センターと民間会社11社の共同出資を得て設立されたAI言語研究所で研究開発を行っている．

グラムを構造的に分割するためのモジュール化機能がない．そのため，実用的なプログラム作成を目的とした Prolog では種々のモジュール化機能を導入する傾向があり，現在 ISO で進めている Prolog 標準化作業においても，標準化案としてモジュール化機能を含める方向で議論が行われている[9]．

一方，単なるモジュール化機能に留まらず，オブジェクト指向概念との融合を目指す言語もある．そのアプローチは出発点を CESP と同じくし，まず基本に論理型言語を置き，論理型プログラムを構造化する手段としてオブジェクト指向を使うものが多い．また，その融合法はホーン節の集合を一つのオブジェクトと定義するところまではほぼ共通であるが，オブジェクト指向の特徴である，オブジェクト間のメッセージ通信の方法，オブジェクトの状態を保持するための変数 (Smalltalk におけるインスタンス変数) の扱い方，継承の方式などはまったく異なっている．

言語仕様として，どのような融合法がベストであるかは，プログラムの記述対象の違いなどにより異なると考えられるが，CESP では

- 汎用計算機上で実用となる大規模プログラムが簡単に記述できる枠組をもつこと
- 処理系実装時にあまり効率を落としてしまうような仕様は含めないこと

などの点を考慮して論理型との融合を行った．

以下に CESP のオブジェクト指向機能の概要を示す．

2.1 プログラムの構造化・階層化

CESP におけるモジュール化はクラスと呼ばれるプログラムを定義することで行う．CESP ではクラスもオブジェクト（クラスオブジェクト）として扱い，クラスの状態（スロット）の表現やクラスごとに異なる処理（メソッド）を定義できる．また，クラス間の多重継承機能をもち，クラス間に親子関係を結ぶことで，親クラスのスロットおよびメソッドを子クラスへコピーしたのと同様の意味をもたせることもできる．

単なるプログラムモジュール化との違いの一つは，クラスという型から派生するインスタンスを複数生成できることにある．インスタンスはスロットの内容つまり状態だけが異なるようなオブジェクトで，プログラムは同じクラスに属するインスタンスに共通に定義する．

最も一般的なクラス定義の型は

```
class 〈クラス名〉 has
nature〈継承クラス宣言〉
  〈クラススロット/メソッド定義〉
instance
  〈インスタンススロット/メソッド定義〉
local
  〈ローカル述語定義〉
end.
```

である．ローカル述語はメソッドの内部処理を定義する述語で，直接外部のクラスからは呼び出すことのできないような処理を記述する．このような述語の隠蔽はプログラムのモジュール性を高めるために必要な機能であり，メソッドの外部インタフェースの変更さえ行わなければ，内部処理をどのように変更しても，外部のクラスへの影響がまったくない仕組となっている．

2.2 オブジェクトの状態

モジュール化により情報の隠蔽度を高めるためには，プログラム中で使用するデータについても考慮が必要である．CESP のようなオブジェクト指向言語ではオブジェクトの保持するスロットのオブジェクト外への公開/非公開機能に相当する．そこで

- 外部クラスでも参照/更新が可能な attribute
- クラス内だけで参照/更新が可能な component

の二種類のスロットを用意し，目的により使い分けができるような仕様とした．

スロットが公開かどうかも述語の公開性と同じ枠組で解釈できる．公開スロットについてはアクセスするためのメソッドを自動的に作り，非公開スロットについてはローカル述語を作る，と考えれば良い．

ところで，論理型言語との融合で大きな問題となるのが，このようなオブジェクトの状態を論理型言語の枠組にどう整合良く取り入れるかである．Prolog では一般に時間とともに変化する状態を保存するような副作用概念がないため，すべてを Prolog の枠組だけで表現するのであれば，以下に示すような方法を用いることになる．

1) 公理系の変更に相当する述語 assert/retract

を用いる方法.

たとえば，サイズ 100*200 のクラス window のインスタンスに window 1 と名前を付け

state (window, window 1, size (100, 200))

のような unit clause で表し，これを assert することで状態を変化させる.

2) 項にオブジェクトとしての意味をもたせ，項中の変数で状態を表す方法.

1) に対応させると

window (window 1, size (X, Y))

という項の X に 100, Y に 200 を代入し，それをインスタンスとみなす.

しかし，1) は Prolog の節データベースの動的な変更が必要なため，効率の良い実装が難しい，2) は副作用として残さないため，バックトラッキングによってオブジェクトが消滅してしまう，などの問題点がある.

CESP では言語仕様上は 1) に近い，つまり公理の変更という方針を採用したが，そのまま実現すると生じる効率上の問題を，言語仕様上はスロットの値にもてるものを制限する（変数を含みうるような項をもてないようにする，つまりバックトラッキングにより解放される size (X, Y) というような項はスロットに代入できない）ことで解決した.

また，実装上はクラスごとのスロットハッシュ表を用意することで

オブジェクト

⇢ スロット名をキーにスロット番号検出

⇢ 番号に対応する値の参照/更新

という手順による高速なアクセスを可能とした. なお，スロットは CESP 固有の副作用エリア上に置くため，バックトラッキングによってもその値は失われることがない.

2.3 メッセージ交信

オブジェクト指向言語ではオブジェクトに定義されたメソッドの実行は，そのオブジェクトにメッセージを送ることにより行う. CESP ではメッセージ送信も論理型言語のゴール呼び出しの形式と合わせて

：メソッド名（オブジェクト，
引数 1, …, 引数 n)

- 述語名の代わりに「：メソッド名」，
- 第一引数に受信側となる「オブジェクト」

と記述する. 第一引数のオブジェクトは，Smalltalk などのレシーバと同じように実行時までに決まれば良いので，

- 柔軟なモジュール間インタフェースが実現できる
- プログラムの拡張/修正が容易にできる

などの利点がある. その反面，呼び出すプログラムとの結合が実行時になるため，実行効率は通常のゴール呼び出しと比べて悪くなるという欠点もある. そこで，スロットと同様クラスにメソッドハッシュ表を用意し，

オブジェクト

⇢ メソッド名，引数個数をキーとしたメソッド番号の検出

⇢ 番号に対応するコードの実行

という手順により実行効率の改善を図った.

また，メッセージ交信に関しては，並列に動作するオブジェクトが互いに通信しあって計算する，というモデルがオブジェクト指向の計算モデルとして適切なものの一つである. これをそのまま実現した言語に ABCL[10] などがある. しかし，この枠組は Prolog のバックトラッキングによる縦型探索の枠組との整合性は良くない. CESP ではあくまで Prolog の実行機構をそのままにしてオブジェクト指向機能を導入する方針で設計したため，メッセージの送信側は対応するメソッドの呼び出しが終了するまで次の処理へ移ることはできないようになっている. そのため，メソッドの実行も Prolog の述語呼び出しと同じように成功/失敗の対象となり，バックトラッキングによる探索も Prolog と同じ考え方で制御できる.

しかし，CESP では並列処理がまったく記述できないというわけではなく，言語仕様としてではなく，ライブラリとして用意したマルチプロセス機能用のクラスを利用することで

- サーバクライアント型のプロセス間通信を行う
- CESP プロセスの子プロセスを生成し，その子プロセス上でゴールを実行する

などの処理を記述できる.

また，明示的に並列処理を記述しなくても，探索の並列処理を暗黙のうちに行うような処理系[11]を実現することができるのは Prolog と同様である.

2.4 メソッド結合

メソッド呼び出しで実際に実行されるメソッドを決定する規則は，継承をどのように解釈するかによって決まる．多くのオブジェクト指向言語では子クラスがメソッドを定義すると親クラスのもつ同じ名前のメソッドはそれに隠されてしまう．CESP では継承を「クラスが表す公理群の合併集合を作り出すこと」と論理型言語として自然な解釈を与えた．そこで，親クラスと子クラスに同一名称/引数個数をもつメソッドがあるとどちらも実行可能となる．Prolog では公理の適用順が重要になるが，CESP ではそれを親クラスの階層を左から右へ深さ優先でたどった順と決め，それらのメソッドの実行を OR で結合する規則とした．

しかし，すべてが子クラスに OR で結合されると，論理型特有のバックトラッキング機能により，子クラスで実行が失敗した場合には必ず親クラス内の実行に移ってしまう．そこで，失敗した場合には親クラスをみない，つまり決定的なプログラムとするための記法も必要となる．このような子クラスでのオーバライディングに相当する機能の実現には Prolog における公理の選択を制限する機構であるカットオペレータを用い，メソッド定義中のカットオペレータにこの役割をもたせることにした．

また，メソッドに条件を付けるための機能としてはオーバライディングのほかにも

- before demon

実行条件付加，初期値設定などの前処理記述用

- after demon

割当資源の登録/解放，主処理実行後の後始末などの後処理記述用

- wrapper

子クラス内メソッドの実行抑制，条件設定などの制御記述用（before demon より優先度が高い）があり[12]，これらのメソッドを組み合わせたものが実際の呼び出し対象となるメソッドとなる．

なお，このようなメソッドの結合の解析はコンパイル時に行うので，実行時のオーバヘッドは小さい．また，その際にクラス間にわたる決定性の解析も行っている．

2.5 クラスの定義の例

ここまでに述べてきた言語機能を，以下のような簡単なクラス定義例に基づいて説明する．

```
class window has
nature label, standard-io;
 : create (Class, Information, Window) :-
        : new (Class, Window),
        initiate (Window, Information),
        create (Window);
instance
component
        x, y, width, height;
 : show (Window) :-
        show (Window, Window!x, Window!y,
        Window! width, Window! height);
local
initiate (Window, Information) :-
        「生成情報 Information から位置/サイズなどの情報を取り出し，スロットに設定する」
create (Window) :-
        「実際にウィンドウ本体を作る」
show (Window, X, Y, Width, Height) :-
        「ウィンドウを表示する」
end.
```

このクラスはウィンドウを識別するためのクラス label，標準入出力機能をもつクラス standard_io を継承するような window という名前のオブジェクトの定義である．機能としては，

- ウィンドウを生成する：create/3
- ウィンドウを表示する：show/1

をもつ．：create/3 はクラスメソッドとし，クラス window に：create メッセージを送るごとにインスタンス生成用のメソッド ：new/2 が実行され，ウィンドウの実体とインスタンスの対応付けを行う．またウィンドウ生成後には，表示（：show/2）などの命令はインスタンスごとに指定できるインスタンスメソッドとして定義してある．

なお，これらの直接定義されたメソッドに加え，親クラスに定義されたメソッドも継承機能によって取り込まれる．たとえばクラス standard_io に ：getc/2，：putc/2 といった文字入出力メソッドが定義されていれば，このクラスのオブジェクトはそれらのメソッドも合わせもつことになる．

オブジェクトのスロットに関しては，ウィンドウごとに異なった固有の値を保持する必要があるため，インスタンスとして，位置 (x, y)，サイズ

(width, height) の情報をもつ．このときスロットの種類は component と定義することで，他クラスでの誤った更新などを防止できる．（クラス定義中の「Window!x」という記法はスロット値の参照を表す．）

ウィンドウ生成で使用するメソッド結合の一例としては，クラス window の中で継承クラスへの依存関係を記述しないですむ（ラベルを継承している場合の処理/継承していない場合の処理を個別に記述しない）以下のようなクラス label の記述法がある．

```
class label has
before : create (Class, Information, Window) :-
        「ラベル情報を取り出す」
after : create (Class, Information, Window) :-
        「実際にラベルを作る」
end.
```

このクラス定義により ：create/2 の実行は

before ：create/2 ⟶ ：create/2 ⟶ after ：create/2

と進み，クラス window 中にラベル処理を記述する必要がなくなる．なお，window が label を継承しなければ，label 中の demon は実行されない．

3. 論理型言語機能に対する拡張

論理型言語を用いて大規模ソフトウェアを効率良く作成できるためには，**2.** で述べたオブジェクト指向機能を融合するという方向だけでなく，論理型言語機能に対する拡張も必要となる．以下では，CESP で導入した代表的な拡張機能を紹介する．

3.1 データ型の追加

普通の Prolog がもつデータ型には

- 変数，アトム，整数，浮動小数点数
- 構造体データ（リスト，ファンクタ）

がある．CESP ではこれらのデータ型以外に **2.2** で述べたようにオブジェクトを表現する構造体として副作用をもつデータ型がある．副作用をもつデータ型はインピュア構造体と呼ばれ，オブジェクトのほかにも

- インピュアベクタ

破壊代入の可能な一次元配列に相当する構造体

- インピュアストリング

通常は I/O 用の文字列を表現するが，単なるビットデータの代入も可能な構造体がある．一方，破壊代入の概念のない Prolog の構造体データに相当するものはピュア構造体と呼び，インピュア構造体に対応して

ピュアベクタとピュアストリング

がある．ピュア構造体は通常の Prolog のユニフィケーション規則にしたがって扱うが，インピュア構造体は破壊代入を許すため，その構造体の実体つまりメモリ上でのアドレスまで同じでないとユニフィケーションは成功しない．ユニフィケーションの時点では内容は同じだが，実体は異なるようなインピュア構造体についてもユニフィケーションが成功するような仕様では，Prolog の言語仕様との整合性が悪い．たとえば

```
p(S, S, X) :-
        set_vector_element (X, 7, a), q(S);
```

という述語の第 1，3 引数にインピュアベクタ，第 2 引数に別のインピュアベクタを渡したとする．たまたま呼び出し時点で両者の内容が同じであるからといってユニフィケーションを成功させてしまうと，q に渡される S が二つの実体のどちらになるのかがはっきりしない．Prolog の言語仕様はいったんユニフィケーションに成功した二つの値が，後に別のものになってしまうような操作を想定していないのである．

その一方，インピュア構造体はバックトラッキングによっても値を解放しないため，オブジェクトのもつスロットへの代入が可能である．ただし，ピュア構造体に関しても，その内容をスロットに代入したい場合がある．そのため，ピュア構造体内にある変数同士の対応関係だけを残し，要素を直接アクセスできない凍結形に変換したスロットに代入可能なフローズンタームと呼ばれるデータ型も用意した．

なお，上述した基本データ型だけでは複雑なデータ構造を扱うような場合にユーザのプログラミング時の負担が増えるため，基本データ型とは異なった，オブジェクトとして生成/演算/操作が可能な

ハッシュインデックス，リストインデックス

リスト，配列，スタック，BIGNUM，部分項

などのデータ構造を実現するためのクラスをライブラリとして用意している．

3.2 マクロ記法の導入

Prolog のような論理型言語では，関数型とは異なり，数式の値が欲しい場所に直接，式を書くことはできない．また，通常の手続き型言語にみられるマクロ記法ももたないため，プログラムの大規模化により記述性/可読性などが悪くなる．そこで，CESP では数式などの記述が簡単に行えるように処理系として標準装備したシステムマクロと展開制御などを利用者が定義可能なマクロ定義機能を導入した．

マクロは展開対象の種類（クラス全体，述語定義節，その中の項など）に応じて定義できる．展開はマクロ呼び出しを置き換えるだけでなく，呼び出しを含む節，ゴールなどの前後に節，ゴールなどを挿入することもできる．また，マクロ定義にあたっては，ユニフィケーション機能が便利に使える．

このようなマクロ定義はクラス定義とは独立したマクロバンクにより行う．マクロバンクでは，クラスと同様に継承機能（優先順位もクラスと同じ），マクロ定義中で使用するスロットさらに内部処理記述用のローカル述語の定義も可能である．

3.3 実行制御機能の拡張

CESP の論理型言語としての基本機能は Prolog と同じで，実行制御はユニフィケーション，バックトラッキング，カットが基本となる．しかし，これらの制御機能だけでは，効率の良いプログラムを書きにくい，あるいはエラー処理の定義が難しいなどの問題点もある．そのため CESP では実行制御機能の拡張を図った．

以下にいくつかの代表的な拡張機能を示す．

（1） 大域脱出機能

実用となるプログラムでは常に例外的な事象に備えたプログラミングが必要となる．例外処理の代表的な方法は，例外の生じたプログラム部分の実行を中止し，処理の準備のあるレベルまで制御を戻すことである．そこで，多段にネストして呼び出された述語群の中で，一番深いレベルから途中の必要のない複数の述語を飛び越して，呼び出しレベルまで一気に戻るような catch & throw 機構を用意した．機能的には cut & fail と似ているが，選択肢を取り除く範囲がカットより広く，失敗と異なりどの地点まで戻るかを明示的に指定できるなどの特徴がある．

また，エラー発生時には，その地点からの脱出あるいはバックトラッキング時の後処理を定義したい場合がある．そのため，カットによって取り除かれることなく，バックトラッキング時のみに実行する，つまりバックトラッキング時まで実行を遅延させる述語を設定できる機能も用意した．

（2） 例外処理

Prolog のような言語での「失敗」は通常の実行機構の一部であり，エラー処理は別の枠組で行うことが望ましい．そこで，CESP では組込述語などのエラーの発生原因によって規定した

- type mismatch　引数タイプの不一致
- integer overflow　整数演算オーバフロー

などの 14 種類の例外を設けた．例外発生時には例外処理環境にその原因を通知し，そこで規定された処理を起動する．処理系として標準に設定した例外処理は，シチュエーションと呼ばれ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．

しかし，簡単なエラーの場合は引数の値を変えるなどによって処理を継続したい場合もある．そのためにクラスとして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）:-
        「実行したい例外処理」
```

の形式で定義し，そのクラスを例外処理環境として設定できる機能がある．

例外処理環境の設定にはネストを許す．その場合は

システム例外処理環境
⟹ ユーザ例外処理環境 1
⟹ ユーザ例外処理環境 2 ⟹…

というように，システム例外処理環境をルートとする階層構造ができ，例外発生時にはその例外に対応したメソッドの定義してある例外処理環境で処理を行うことになる．したがって，ユーザ定義のない場合はすべてシステムで標準に設定した例外処理を実行する．

4. 他言語インタフェース

プログラミング言語を用いてなんらかの問題を解く場合に，その問題によって最も適したプログラミング言語というものがある．CESP のような言語では論理型の特徴であるバックトラッキング，ユニフィケーションの利用，さらにはオブジェクト指向的な機能/構造分割などの利用により，試行錯誤的な AI 処理の記述に向く．しかし，数値解析などで Fortran のコンパイラのほうが良質なコードを生成できる場合や，OS や他のユーティリティとのインタフェースなどCで書いたほうが楽に書ける場合もある．また，すでに存在する他言語で記述された資産の活用もできる．

そのような場合，CESP から他言語，逆に他言語から CESP を呼び出すことができれば，両者の特徴を生かしたプログラミングが可能となる．

こうした要求から生まれた機能が他言語インタフェース機能であり[13]

- CESP で作成したプログラムの一部を簡単に他言語に置き換えできること
- CESP のオブジェクト指向によるモジュール化機能を生かし，CESP のクラス定義と他言語プログラムとの結合を図ること
- CESP と他言語との間に生じる，データ型のギャップを簡単な枠組で吸収できること
- 他言語側から CESP のオブジェクトをアクセスできること

という方針で設計を行った．ここでは呼び出し方に着目して，インタフェースを説明する．

4.1 他言語インタフェースクラス

他言語によるルーチンとの結合にも CESP のもつオブジェクト指向のモジュール性を生かすには，クラス単位にインタフェースを取るのが望ましい．そのためのクラスは他言語インタフェースクラスと呼ばれ，

- CESP のメソッドとして呼び出す他言語ルーチンの宣言
- 他言語ルーチン側から呼び出す CESP のメソッドの定義
- データ変換などの規則を定義した他言語バンク（**4.2** で示す）の宣言

を記述する．これらの特殊項目の記述以外は普通のクラス定義とまったく同じで，クラスの継承/参照の扱い，クラス/インスタンスメソッドの指定に関しても同様の扱いが可能である．

このような仕様とすることで，メソッドを他言語ルーチンで書き直すような場合にも，メソッド定義本体の変更は必要であるが，呼び出し側のクラスへの変更は一切必要なくなる．

4.2 他言語バンク

CESP と他言語側のデータ型のギャップなどを埋めるための機構として他言語バンクと呼ばれるモジュールがある．他言語バンクでは

- 他言語情報宣言部

他言語の名称（たとえば，C, Fortran），呼び出す手続きの定義されているオブジェクトファイル名，使用するライブラリファイル名などの宣言

- export 宣言部

他言語からの CESP メソッドの呼び出し方の宣言

- import 宣言部

CESP からの他言語ルーチンの呼び出し方の宣言などを記述する．

また，上記宣言以外にも他言語バンクの継承機能や，他言語とのデータのやりとりをスムーズに行えるためにC言語などの typedef 宣言に対応した仮想的なデータ型を定義できる機能もある．

以下，C言語を例にとり，相互呼び出しの記法を簡単に説明する．

（1） CESP 側からの呼び出し：export 宣言部

CESP のメソッドとC関数との対応を

〈述語名〉(〈引数情報〉,…)→

〈関数値受け取り用変数〉=〈関数呼び出し〉

の形式で定義する．ここで，引数情報はC関数引数とのデータ型や入出力モードの対応を宣言する部分で，たとえば X: (−int) Y: (+object) のように記述する．処理系ではこれらの情報を利用し，CESP 側とC側とのデータ型変換を呼び出し時に自動的に行う．

（2） C関数側からの呼び出し：import 宣言部

C関数中から呼び出す CESP メソッドの情報を

〈述語名〉(〈引数情報〉,…)

の形式で定義する．C関数の中からの実際の呼び出し部分には，ライブラリとして用意する関数

cespFLImethod (〈オブジェクト〉, 〈メソッド名〉, 〈引数個数〉, 〈引数〉,…, 〈引数〉)

を使用する．この呼び出しを行うと他言語バンク中に宣言したデータ型変換に従い，実際に他言語

インタフェースクラスの中に定義したメソッドを呼び出す.

また,他言語からは単純にCESPのルーチンを呼び出せるだけではなく,失敗を起こしてバックトラック探索をさせる機能(Prologのfailにあたる),実行中に残した選択肢を取り除く機能(Prologのカットにあたる)も提供している.もちろんこうした論理型言語機能部分は普通CESPで記述したほうが簡明ではあるが,扱うデータ構造などとの兼ね合いから効率上他言語との組み合わせで記述したほうが有利な場合もある.

5. 処理系/プログラミング環境

CESPはPSI上のESPあるいはSmalltalk, LISPなどと同じように総合的プログラミング環境をもつ言語である.このような言語では通常,環境のベースにウィンドウを置き,ビットマップディスプレイやマウスの特性を駆使した処理系を作ることが多い.しかし,CESPの目的の一つとして種々の汎用機上で動作可能な処理系とすることがあり,移植性の重視や普通のキャラクタ端末上での実行を目標とした構成を取ることが重要であった.そのためにCESPの言語処理系としては環境のベースを操作性/移植性の良いGNU Emacs[14](以下Emacsと略す)とし,エディタ固有の優れたユーザインタフェースを利用することにした.

一方,言語の普及には環境の良さ,移植性の高さだけでなく,言語処理系の実行効率が重要となる.実行性能を上げることを考えると,実行する計算機の機械語に翻訳して直接実行するのが有利であるが,その反面移植性に問題が生じる.そこで,CESPでは機械語生成用のコンパイル環境での実行を目指すと同時に,機械語生成のために入り込む機種依存部を極力小さくするコードジェネレータジェネレータ方式[15),16)](以下CGGと略す)を用い,移植性の高さと効率の良い実行の両立を目指した.

以下では,Emacsを利用した環境の構成法およびコード生成方式を中心としたコンパイル環境についての概要を述べる.

5.1 Emacsをベースとした対話環境

Emacsは単にプログラムを編集するための機能を備えるだけでなく,優れたプログラミング環境を構成するための各種の充実した機能,さらにユーザ固有のカスタマイズが可能なツール群が整備されたエディタである.CESPではこのようなEmacsの優れた機能を利用するため,Emacsプロセスから起動する一つのプロセスとしてCESP処理系を組み込み,CESP処理系の実行時はEmacs側と通信できるような構成法を採用した.以下に代表的な機能を示す.

(1) ソースレベルデバッグ機能

オブジェクト間のメッセージ通信あるいはオブジェクト内での内部処理がどのように実行されるかをみるためには,多くのProlog処理系で用いるボックスコントロールフローモデルに基づくトレーサが有効である.しかし,呼び出した述語とその引数内容が表示されるだけでは,ソースプログラム中のどこを実行しているかの判断がつかない場合が多い.特に,複数のファイルにモジュール分割される大規模プログラムにとっては,バグ発見時にプログラム中の問題箇所との対応付けが困難な場合も出てくる.

そこで,CESPではプログラムのコンパイル時に対応するファイル名およびそのファイル上での位置をコンパイルコード中に埋め込み,トレース実行時にはソースプログラム上でどのメソッドまたは述語が実行されているのかを同時に表示できる機能を用意した.この機能はエディタのバッファ上で操作するため,バグ発見時にはそのまま修正が行え,修正範囲を指定して,そのクラスだけをコンパイルすることも可能となる.

(2) Emacs操作機能

移植性が良く,しかも柔軟な入出力手段を提供するため,Emacsのバッファを CESPのインスタンスとして扱うことができるようにした.通常の文字入出力のほかに,画面分割,メニューなどのユーティリティ機能も用意している.

また,処理系自身もEmacsのINFO機能と組合せ,Emacsバッファに対する

- トレース,例外発生時のマニュアル表示,対応する組込述語/ライブラリィの自動検索
- プログラミング時のマニュアル表示,コンプリーション機能を用いた検索

などを瞬時に行える構成とした.

5.2 コンパイル環境

CESP のようなオブジェクト指向言語にとって，実行性能を良くするためには，実行時に決まるメソッドと実コードとのバインディングや頻繁に行われるスロットアクセスの高速化を図る必要がある．また，論理型言語の実行という面でみた場合には述語呼び出しにおける決定性の検出をなるべくコンパイル時に行うなどの方策も有効となる．しかしながら，最も根底にある問題は論理型言語という高水準言語自身の実行性能であり，これをいかに実際のハードウェアに近い記述（仮想機械語）に変換できるかで性能が決まる．Prolog の場合はこのような仮想機械語の代表格に機種独立な命令セットをもつ Warren Abstract Machine (WAM)[17] がある．PSI のような専用計算機では WAM 自身が直接機械語として実行され，ESP の高速な処理が可能であったが，汎用機の場合はこの仮想機械語をエミュレート実行する方式を取ることが多い．

WAM は機種に独立なものであるから，UNIX 系のマシンであればC言語で書かれたエミュレータを作ることで，簡単に移植性の高いものが作れる．しかし，少なくとも実行回数が多い部分は機械語にまで変換する方式が有利である．ところが，機械語の生成には機種に依存する処理が必ず必要となるため，逆に移植性に問題が生じる．

そこで，CESP では移植への影響を最小限にとどめ，対象となる計算機の機械語を生成する方式の有効な手法の一つである，機種に依存する部分を自動生成する CGG 方式の利用を試みた．以下に CESP における機械語生成の過程を示す．

[ソースプログラム]
|← 字句/構文解析
|← マクロ展開
|← WAM コンパイラ
[中間的 WAM コード]
|← 継承解析
[エミュレート実行可能 WAM コード]
|← LIR コンパイラ　　独立部
[仮想機械語 LIR コード]……
|← コードジェネレータ　　依存部
|← スケジューラ
[機械語]

仮想機械語 LIR (Low-level Intermediate Representation) というのは，機種独立部と依存部の橋渡しをする抽象的なアセンブラ言語で，WAM 生成時までにできていない最適化などをこの時点で行う．問題となる機種依存部が LIR から機械語を生成するコードジェネレータ部である．コードジェネレータの自動生成にはマシン記述と呼ぶ，機種固有な情報

- ハードウェアレジスタ情報
- LIR から機械語へのマッピングルール
- 最適化情報

を利用し，これらの情報を機種ごとに記述することによりコードジェネレータが自動生成できる．なお，当 CGG 方式の利用により，MC 68000 系，SPARC への移植が完了し，方式の妥当性も検証されている．

また，コンパイル環境の一部として，コンパイラが生成した機械語ファイルとその基になったクラスから呼び出すシステムクラスおよび実行に必要な最低限の環境をリンクすることで，CESP 処理系とは別の自立的な実行イメージを生成するような機能も用意してある．

6. おわりに

論理型言語による大規模ソフトウェアの開発を可能とするため，論理型への拡張としてオブジェクト指向や他言語とのインタフェース機能をどう導入したか，さらに移植性/実行性能の良さを求めたプログラミング環境をどのように設計したかについて述べた．しかし，本文中で述べた言語仕様/処理系は研究開発途上のものであり，まだ改良の余地が残っている．特に，持続性をもつオブジェクトやデータベースとのインタフェースを言語仕様の一部として取り込む研究については現在仕様検討の段階にある．また，ユーザインタフェースに関しても実際的な使用経験下での評価に基づく改良/拡張が必要となろう．

今後は 90 年 10 月に研究目的に限り無償配布を行った基本仕様版 CESP での評価を参考にしつつ，より優れた言語へと発展させる予定である．なお，91 年 4 月からは基本仕様版に機能拡張を行ったフルセット仕様版と呼ばれる CESP の配布も開始している．

謝辞 本稿の内容は CESP の研究開発に携わった AI 言語研究所内外の数多くの方々の成果に基づいたものである．ここに深謝の意を表したい．

参考文献

1) Chikayama, T.: Unique Features of ESP, Proceedings of FGCS '84, ICOT (1984).
2) Goldberg, A. and Robson, D: Smalltalk-80 The Language and Its Implementation, Addison-Wesley (1983).
3) McCabe, F. G.: Logic and Objects, Imperial College Research Report (1986).
4) LAP USER's MANUAL V 3.1 Elsa Software (1989).
5) Hodas, J. S. and Miller, D.: Representing Objects in a Logic Programming Language with Scoping Constructs, Proceedings of the Seventh International Conference on Logic Programming, MIT Press (1990).
6) Monteiro, L. and Porto, A.: A Transformational View of Inheritance in Logic Programming Proceedings of the Seventh International Conference on Logic Programming, MIT Press (1990).
7) 近山 隆：オブジェクト指向言語による OS の開発例，情報処理，Vol. 29, No. 4 (1988).
8) 近山 隆，中澤 修他：ESPer への道，bit, Vol. 22, No. 1-12, 共立出版 (1990).
9) Scowen, R. S.: PROLOG Draft for Working Draft 4.0, ISO/IEC WG 17, N 64 (1990).
10) 米澤明憲，柴山悦哉他：オブジェクト指向に基づく並列情報処理モデル ABCM/1 とその記述言語 ABCL/1，コンピュータソフトウェア，Vol. 3, No. 3 (1986).
11) Lusk, E., Warren, D. H. D. et al.: The Aurora Or-Parallel Prolog System, Proceedings of FGCS '88, ICOT (1988).
12) 中澤 修，実近憲昭：論理型言語とオブジェクト指向言語の融合，INAP '89 論文集 (1989).
13) 伊藤民哉：Common ESP における他言語インタフェースの実際，INAP '90 論文集 (1990).
14) Stallman, R.: GNU Emacs Manual, Fifth Edition, Emacs V18, Free Software Foundation (1986).
15) 佐藤良治，高橋文男他：Common ESP 機械語ジェネレータの概要，情報処理学会第 40 回全国大会論文集 (1990).
16) Ganapathi, M., Fischer, C. N.: Affix Grammar Driven Code Generation, ACM Transactions on Programming Languages and Systems, Vol. 7, No. 4 (1985).
17) Warren, D. H. D.: An Abstract Prolog Instruction Set, SRI Technical Note 309 (1983).

(平成3年1月8日受付)

中澤 修（正会員）

昭和31年生．昭和54年日本大学理工学部電気工学科卒業．昭和56年同大学院理工学研究科博士前期課程修了．同年沖電気工業(株)総合システム研究所入社，現在(株) AI 言語研究所との兼務．この間，知識ベースシステム，論理型言語ワークステーションのオペレーティングシステム，オブジェクト指向言語の研究開発に従事．電子情報通信学会会員．

近山 隆（正会員）

昭和28年生．昭和52年東京大学工学部計数工学科卒業．57年同工学系大学院情報工学専門課程博士課程修了．工学博士．同年富士通(株)入社．(財)新世代コンピュータ技術開発機構に出向，現在に至る．この間，手続き型言語，関数型言語，論理型言語，オブジェクト指向言語とこれらの逐次および並列処理系，プログラミング環境，論理型言語専用計算機とそのオペレーティングシステムの研究開発に従事．