TM-0444

小型化PSIにおける
論理型言語KLOの処理系実装方式

中島克人

January, 1988

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

小型化ＰＳＩにおける論理型言語ＫＬ０の処理系実装方式

概要

第５世代コンピュータ・プロジェクトの中期における並列ソフトウェア開発用ツールとして，ＩＣＯＴはＰＳＩのＣＰＵを小型化し，それを多数結合した「マルチＰＳＩ（Ｖ２）」の開発を計画した。この小型化したＣＰＵは並列論理型言語ＫＬ１を実行するためのものであるが，入出力装置を接続したワークステーションの形態とし，マルチＰＳＩのフロントエンド・マシンとしても使用する事とした。
このフロントエンド・マシンはＫＬ０マシンである事から，ＰＳＩを小型化したスタンドアロン・マシンとしての開発も計画された。このマシンを小型化ＰＳＩと称する。
本メモでは，小型化ＰＳＩにおけるＫＬ０処理系のうち，Ｐｒｏｌｏｇと共通な基本部分の処理機構とその実装方式を解説している。

目次

はじめに

論理型言語は第五世代コンピュータ・プロジェクトにおいて採用されたこともあり，世界的に注目を集めてきた。特に，１９７３年にフランスのマルセーユ大学で発明されたＰｒｏｌｏｇは，本プロジェクトの開始以後，日本および欧州を中心に広く使われるようになった。

一階述語論理をもとにしたＰｒｏｌｏｇは，言語の基本機能としてユニフィケーションとバックトラックの機能を備えた高機能言語で，人工知能研究における有力なツールとして発明当初から着目されていたが，実行効率の面で大きな問題があった。

最初に実用的なＰｒｏｌｏｇ処理系を実現したのは英国のエジンバラ大学で，１９７７年頃のことである。これがいわゆる”ＤＥＣ－１０ Ｐｒｏｌｏｇ”である。基本部分はアセンブラで書かれており，ＤＥＣ－１０／２０マシン上で今なお広く使われている効率の良い処理系である。この処理系では，スタックを３本使用している事，構造体データの内部表現に「Structure Sharing 」という手法を用いている事，インタプリタ／デバッガとコンパイラの両方を備えている事，クローズ・インデックスによるクローズの決定的選択手法を用いている事などの特長があり，その後の多くの処理系の手本となった。

ＩＣＯＴではプロジェクトの前期の最初の目標の一つとして，論理型言語ＫＬ０の設計とその言語を効率良く実行するパーソナルマシンＰＳＩの開発を行った[1]。

ＫＬ０（Ｋｅｒｎｅｌ Ｌａｎｇｕａｅ ｖｅｒｓｉｏｎ ０）はＰｒｏｌｏｇをベースに，遮蔽カットやフリーズ（Ｂｉｎｄ＿ｈｏｏｋ）機能などの幾つかの実用的な実行制御機構を導入した言語である。
ＫＬ０を実装する立場から見ると，これらの諸機能はＰｒｏｌｏｇだけの場合に比べてより重いものであったため，ＰＳＩの基本アーキテクチャとして，柔軟性が高く，複雑な処理も効率良く実行出来る水平型マイクロプログラム制御方式を採用することとした。処理方式の基本部分はＤＥＣ－１０ Ｐｒｏｌｏｇ処理系を参考に，新たに「引数コピー方式」を導入した。引数コピー方式とは，２つのフレーム・バッファを交互に使用して述語呼出しの引数の受け渡しを行う方式であり，ＰＳＩではこのためのハードウェアとして，レジスタ・ファイルとそれをアクセスする機構に幾つかの工夫を凝らした[2]。
この引数コピー方式により，述語呼出し引数の受け渡しと，バックトラック時の環境の回復処理を高速化する事ができた[3]。

ＰＳＩの開発が完了した１９８３年に，当時ＳＲＩにいたＤ．Ｈ．Ｄ．Ｗａｒｒｅｎは Ｐｒｏｌｏｇの高速実行に向いた機械命令セットを発明した。これを現在ではＷＡＭ（Warren Abstract Machine），またはＷＡＭの命令セットと称している[4]。
ＷＡＭの大きな特長は，Ｐｒｏｌｏｇのソース・コードからコンパイラが判る範囲の静的な最適化を出来る限り行う事と，多くの汎用機でもそうであるようなレジスタ・マシンを想定している事である。具体的に述べると，述語呼出し用の引数はレジスタに用意する事，メモリに待避すべきものとしなくとも良いものの区別を変数ごとに行い，できる限りレジスタ上の処理で済ますようにしている事，徹底的なクローズ・インデックシングを行う事により，バックトラックの回数を減少させている事などの特長を有している。またリストやベクタなどの構造体の生成は，量的にも小さなサイズのものが多いという予想のもとに，「Structure Copying 」手法を用いている。

ＷＡＭは特別なハードウェアを前提としていない方式であることなどから，汎用機上のＰｒｏｌｏｇ処理系の中間コードとして，また，ＷＡＭを直接解釈実行できるようなマイクロプログラマブル計算機において，急速に採用されるようになったが，当プロジェクトにおいてＰＳＩの次に開発されたＣＨＩ（Co-operable HighPerformance Inference Machine ）においてもＷＡＭが採用され，その実行効率の良さは立証されている。

当プロジェクト中期における並列ソフトウェア開発のツールとして，ＩＣＯＴはＰＳＩのＣＰＵを小型化し，それを多数結合した「マルチＰＳＩ（Ｖ２）」の開発を計画した。この小型化したＣＰＵは並列論理型言語ＫＬ１を実行するためのものであるが，入出力装置を接続したワークステーションの形態とし，マルチＰＳＩのフロントエンド・マシンとしても使用する事とした[5]。

このフロントエンド・マシンはＫＬ０マシンである事から，ＰＳＩを小型化したスタンドアロン・マシンとしての開発も計画された。このマシンを小型化ＰＳＩ（愛称ＰＳＩ－Ⅱ）と称する。

この小型化ＰＳＩには，以下のような開発目標が設定された。

① 速度性能をＰＳＩの３倍程度に向上する。

② メモリ容量はＰＳＩの２～４倍程度に増大する。

③ 消費電力や設置面積を削減する。

これらの目標の内，② と ③ の達成のためには，ＬＳＩ技術の活用を考え，ＣＰＵ部には全面的にＣＭＯＳゲートアレイＬＳＩを使用する事，メモリはダイナミック・メモリの進歩の恩恵をそのまま享受し，アーキテクチャ的な拡張性だけを確保する事とした。一方，① の目標達成のためには，ＰＳＩでの処理方式を全面的に見直す事とした[5]。

ＰＳＩでは，ＫＬ０処理系もしくはＰｒｏｌｏｇ処理系の技術が確立していなかった事もあり，前述のように柔軟性の高い水平型マイクロプログラム制御ＣＰＵ上にマイクロ・インタプリタを実装していた。しかし，ＫＬ０特有の機能も含め，処理系の実装技術が蓄積された結果，動的な判定を頻繁に行うマイクロ・インタプリタ方式ではこれ以上の速度向上が困難である事が判って来た。

そのころ丁度登場したのがＷＡＭである。ＷＡＭはプログラムの静的な解析を十分行う事により効率のよいコードを生成出来るため，ＰＳＩ－Ⅱの基本処理方式としての採用を検討した。採用にあたり最も重要な問題は，バインド・フックや例外などによるＫＬ０の動的な述語呼出し機能をいかに効率よくＷＡＭに取り込むかにあった。これらの機能は，静的解析のもとに生成されたコードの実行を，特殊事象の発生時には異なる機能の実行に切り換えるようなメカニズムを導入する事で実現した。

ＰＳＩ－ⅡのＣＰＵ自体はＰＳＩと同様，水平型マイクロプログラム制御方式とした。それは，ＷＡＭと同等レベルの機械語命令を直接実行する方式を採用したからである。ＷＡＭ命令はそもそもコンパイラによる静的な解析を終えた結果のコードであり，それぞれのＷＡＭ命令では動的な判定を必要とする処理が多い。また，ＫＬ０の動的な述語呼出しのための処理の切り換えや，そのような特殊事象での複雑な処理にも水平型マイクロプログラム制御は適している。即ち，比較的簡単なハードウェアでも十分な性能を達成できるのである。また，生成するコードがＷＡＭレベルであるという事はコンパイラの負担を軽減するし，コード効率（コード量）上も望ましい。

ＫＬ０もしくはＳＩＭＰＯＳの実現のためにＷＡＭを拡張した，もしくはＷＡＭとの共存を図った他の機能としては，カットや組込み述語の実装，効率の良いトレーサの実装，ＧＣとその起動メカニズムなどである。また，ＷＡＭで提案されている以上の色々な最適化も同時に施している。

ＫＬ０特有機能の実装方式や，ＳＩＭＰＯＳ／ＥＳＰのサポート機能に関しては，また別の機会に報告する事とし，本メモでは，ＰＳＩ－ＩＩにおけるＫＬ０処理系のうち，ＷＡＭと共通な基本部分の処理機構とそのインプリメントを中心に分り易く解説する[7]。従って，ＷＡＭの解説書ともなっている部分も多い。

説明はＰＳＩ　ＩＩの処理系や機械命令についてであるが，ＷＡＭとの違いはできるだけ付記した。

１章ではＰＳＩ－ＩＩの機械命令の簡単な説明を，２，３，４章ではＰＳＩ－ＩＩの処理系の入門書的な解説を，５章では機械命令を実際に解釈実行するファームウェアを平易に解説している。

なお本メモでは，読者はＰｒｏｌｏｇ（ＤＥＣ－１０　Ｐｒｏｌｏｇ）のシンタックスおよびセマンティックスは一応理解しているものと仮定している。

１．機械命令概説

ＰＳＩ－Ⅱの機械命令のうち，基本的なものについて，例を交えて概説する。
なお命令のシンタックス（ニーモニック）などは，ＰＳＩ－Ⅱのものを採用するが，ＷＡＭとの主な違いはこの章の最後の節でまとめておく。

機械命令は，Ｐｒｏｌｏｇのクローズを構成するためのクローズ命令と，複数のクローズを繋ぎ合わせる述語命令とに分けられる。カット用命令は本来クローズ命令であるが，説明の都合上，別に説明する。

１．１　クローズ命令

(1) ｐｕｔ，ｕｎｉｆｙ命令

呼出し述語の引数は引数レジスタに用意される。これはｐｕｔ命令で行われる。原則として，１引数に１命令が対応する。
リストなどの構造体引数に対しては，その構造体に対し１つのｐｕｔ命令，そしてその個々の要素に対しそれぞれｕｎｉｆｙ命令が用いられる。
なお，例や説明に用いる Ax や Xx はレジスタを意味する。

（例１）　p(Z) :- q(1,Z,cat,[3|Z]), r(Z,dog).

```
            -------------
        put_integer   A0, 1     % (1,
        put_value_t   A1, X     %    Z,
        put_atom      A2, "cat" %      cat,
        put_list      A3        %         [
        unify_integer 3         %          3|
        unify_value_t X         %            Z]
```

引数はソース上の順番に従って，Ａ０からＡ３までの４つにセットされる。
変数Ｚはこのクローズの引数としてもともとＡ０に与えられたもので，ここではヘッド部でどこかのレジスタ（Ｘ）に移動されているものと仮定している。
Ａ３には実際にはリストへのポインタが置かれ，リスト自身は後続のｕｎｉｆｙ命令で Global Stack （後述）上に生成される。

(2) ｅｘｅｃｕｔｅ，ｃａｌｌ命令

ｐｕｔ命令で引数をセットし終わると，呼出し先述語へ分岐する。クローズの最後の述語とそれ以外の述語のそれぞれに対し ｅｘｅｃｕｔｅ命令と ｃａｌｌ命令の２種類がある。

（例２）　p(Z) :- q(1,Z.cat), r(Z.dog).

```
                          --        --
        put instructions for q  %   (1,Z.cat)
        call          q         %  q          .
        put instructions for r  %                (Z.dog)
        deallocate              %   (後述)
        execute       r         %              r       .
```

execute 命令は単なる分岐命令である。call 命令は分岐の前に復帰後の継続アドレス（上例では"put instructions for r"の先頭）を制御レジスタの１つ（Continuation Point Register : CPR ）にセットしておく事が異なる。

(3) ｇｅｔ，ｕｎｉｆｙ命令

呼出し先述語のヘッド部では，呼出し元の引数とのヘッド・ユニフィケーションを行う。これはｇｅｔ命令で行われる。原則として，１引数に１命令が対応する。

ヘッド側がリストなどの構造体の場合には，その構造体に対し１つのｇｅｔ命令，そしてその要素１つに対し１つのｕｎｉｆｙ命令が用いられる。この場合，呼出し側引数がヘッド引数と同じ形の構造体（リスト同志もしくは同じ要素数のベクタ）であれば，後続のｕｎｉｆｙ命令で各要素のユニフィケーションを行う（Ｒｅａｄ Ｍｏｄｅ）。
呼出し側引数が未定義変数であれば，その変数にヘッドで指定した型の構造体へのポインタを束縛し，構造体自身は後続のｕｎｉｆｙ命令で Global Stack 上に生成する（Ｗｒｉｔｅ Ｍｏｄｅ）。

ｕｎｉｆｙ命令はｐｕｔ命令の後では必ず Ｗｒｉｔｅ Ｍｏｄｅであるのに対し，ｇｅｔ命令の後では，Ｒｅａｄ ＭｏｄｅまたはＷｒｉｔｅ Ｍｏｄｅの何れかが動的に決定される。

（例３）　q(1,Z,cat,[3|Z]) :- r(Z,dog).

```
            -------------
            get_integer       A0, 1      % (1,
            get_variable_t    A1, X      %    Z,
            get_atom          A2, "cat"  %      cat,
            get_list          A3         %         [
            unify_integer      3         %          3|
            unify_variable_t   X         %            Z])
```

ＡＯからＡ３までの４つにセットされている引数とヘッド引数とのユニフィケーションを行う。第２引数の変数Ｚはこのクローズで最初に現れる変数であるので基本的には ｎｏｐ である。しかし，第４引数とのユニフィケーションで使用するため，ここではＡ１以外のレジスタ（Ｘ）に移動している。
呼出し側第４引数がリストの場合は get_list命令で呼出し側のリストの第１要素に Structure Pointer (SP) を指させて，Read Mode に入る。次の unify integer命令では，SP の指している先と "3" のユニフィケーションを行い，成功したならば SP を１つ進める。unify_variable命令では SP の指している先とレジスタＸにセイブしてあった内容との汎用ユニフィケーションを行う。
呼出し側第４引数が未定義変数の場合は get_list命令でその変数に Global Stack の先頭アドレスをリストへのポインタとして束縛し，Write Mode に入る。後続の unify_integer, unify_variable では Global Stack の先頭に順番に "3"，レジスタＸの内容を書込み，Global Stack Top は２つ進む。

(4) ａｌｌｏｃａｔｅ，ｄｅａｌｌｏｃａｔｅ，ｐｒｏｃｃｅｄ命令

ＰＳＩ　ＩＩではクローズを以下の様に分類する。

a. ボディ・ゴールがない（ユニット・クローズ）

```
a-1.    p(1) :- !.
a-2.    p(2).
```

b. ボディ・ゴールがただ１つある（トランジティブ・クローズ）

```
b-1.    p(1,X) :- !, q(3,X).
b-2.    p(2,X) :- q(4,X).
```

c. ボディ・ゴールが２つ以上ある

```
c-1.    p(1,X) :- !, q1(X,Y), r(Y).
c-2.    p(2,X) :- q2(X,Y), r(Y).
```

ユニット・クローズの場合は ｐｒｏｃｅｅｄ命令により呼出し元への復帰が行われる。
トランジティブ・クローズの場合は，ボディ・ゴールの呼出し後にはそのクローズに制御を戻す必要がない（広義のTail Recursion）ために特別な事は何もしない。
ボディ・ゴールが２つ以上ある場合は，１つ目のボディ・ゴールからの復帰後にこのクローズに制御を戻すために，ａｌｌｏｃａｔｅ命令によってこのクローズの環境を生成する。この環境は最後のボディ・ゴールの呼出し前に ｄｅａｌｌｏｃａｔｅ命令によって廃棄される。

従って，上例はそれぞれ以下の様な命令列になる。
なお，execute with_deallocate は execute と deallocate をマージした最適化命令である（１．４（４）参照）。

```
a-1.    get_instructions for p
        cut_me
        proceed

a-2.    get_instructions for p
        proceed

b-1.    get_instructions for p
        cut_me
        put_instructions for q
        execute q

b-2.    get_instructions for p
        put_instructions for q
        execute q

c-1.    allocate
        get_instructions for p
        cut me
        put_instructions for q1
        call    q1
        put_instructions for r
        execute_with_deallocate r

c-2.    allocate
        get_instructions for p
        put_instructions for q2
        call    q2
        put_instructions for r
        execute_with_deallocate r
```

１．２　述語命令

(1) ｔｒｙ，ｒｅｔｒｙ，ｔｒｕｓｔ命令

複数のクローズからなる述語においては，ｔｒｙ系命令によりクローズ間のリンクがとられ，バックトラックに備える。ｔｒｙ系命令には，述語の最初のクローズの前に置かれ，バックトラック環境を生成するｔｒｙ命令，述語の最後のクローズの前に置かれ，バックトラック環境を廃棄するｔｒｕｓｔ命令，最初でも最後でもないクローズに置かれ，バックトラック環境の中のオールタナティブ・クローズ・アドレスのみを書き替えるｒｅｔｒｙ命令の３種がある。

（例４）
```
p(1,X) :- !, q1(X,Y), r(Y).
p(2,X) :- !, q2(X,Y), r(Y).
p(_,X) :- s(X).

    try_me_else L2
    instructions for clause "p(1,X):- ..."
L2: retry_me_else L3
    instructions for clause "p(2,X):- ..."
L3: trust_me_else_fail
    instructions for clause "p(_,X):- ..."
```

または，

```
    try    L1
    retry  L2
    trust  L3
L1: instructions for clause "p(1,X):- ..."
L2: instructions for clause "p(2,X):- ..."
L3: instructions for clause "p(_,X):- ..."
```

try_me_else L2 はこの述語呼出しの環境（バックトラック環境）を後述の Local Stack にセーブし，更にこの命令に引き続くクローズ，即ち p(1,X) の実行が失敗した場合に，バックトラック処理の後に制御を移す飛び先番地 L2 を指定する。try L1 はまず最初の飛び先を L1 とし，失敗時の戻り先をその try 命令の次の番地とする所がtry_me_else と異なる。
retry_me_else L3 はバックトラック環境がセーブ済みの場合の try_me_else 命令と考えれば良い。即ち，失敗時の飛び先番地のみを L3 切り換える。retry L2 も同様である。trust_me_else_fail 命令は最後のオールタナティブ・クローズの実行前に用いられ，バックトラック環境の廃棄を行う。この廃棄により，最後のクローズの失敗時には，この述語が呼ばれる以前のバックトラック環境に制御が移るようになる。

(2) ｓｗｉｔｃｈ＿ｏｎ＿ｔｅｒｍ，ｈａｓｈ＿ｏｎ＿ｖａｌｕｅ命令など

クローズを１つずつ試してみなくても，ヘッド引数の１つを予めテストする事により，ユニフィケーション可能なクローズを１つまたは少数に絞り込む事ができる場合が多い。１つに絞り込めた場合にはバックトラック環境を生成する事も省略できる。

ＰＳＩ－Ⅱではこのクローズ・インデクシングを徹底的に行う方針である。
ＷＡＭではインデクシングの対象には第１引数のみとしているが，ＰＳＩ－Ⅱの機械命令では一応任意引数が指定できるようにしている。しかし，従来からのプログラマの慣習もあるので，コンパイラでは当面第１引数のみをサポートする予定である。

クローズ・インデクシングは以下のように行われる。
まず呼出し側第1引数のデータ・タイプをｓｗｉｔｃｈ＿ｏｎ＿ｔｅｒｍ命令でテストし，リスト，複合項，アトミック，その他または未定義変数の4通りに分類してそれぞれのための専用のコードに制御を移す。

（例５）
```
        p(1,X) :- !, .....
        p([L|Z],X) :- !, .....
        p(3,X) :- !, .....
        p(f(4),X) :- !, .....

        switch_on_term Ai,La,Ll,Lv,Le
   La:  try_me_else Lal
   La0: instructions for clause "p(1,X):- ..."
   Lal: trust_me_else_fail
        instructions for clause "p(3,X):- ..."
   Ll:  instructions for clause "p([L|Z],X):- ..."
   Lv:  instructions for clause "p(f(4),X):- ..."
   Lc:  try    La0      % caller argument =  undefined/else
        retry  Lal
        retry  Ll
        trust  Lv
```

アトミックなヘッド引数を持つクローズの数が多ければ，そこでまたｈａｓｈ＿ｏｎ＿ｖａｌｕｅ命令（ＷＡＭではｓｗｉｔｃｈ＿ｏｎ＿ｃｏｎｓｔａｎｔと称する）でインデクシングをしなおす事ができ，その場合はハッシュを用いる。

```
（例６）   p(1,X) :- !, .....
          p([L|Z],X) :- !, .....
          p(3,X) :- !, .....
          p(4,X) :- !, .....

          switch_on_term Ai,La,L1,Lv,Le
     La:  hash_on_value  Ai,Mask=B'11',Lo,
            1,K0          % collision=1, label for key=0
            1,K1          % collision=1, label for key=1
            0,--          % no match clause (fail) for key=2
            1,K3          % collision=1, label for key=3
     K0:   "4",L4         % if Ai=4 then L4 else fail
     K1:   "1",L1         % if Ai=1 then L1 else fail
     K3:   "3",L2         % if Ai=3 then L3 else fail
     L4:  instructions for clause "p(4,X):- ..."
     L1:  instructions for clause "p(1,X):- ..."
     Lv:  fail
     L3:  instructions for clause "p(3,X):- ..."
     L1:  instructions for clause "p([L|Z],X):- ..."
     Le:  try    L0       % caller argument =  undefined/else
          retry  L1
          retry  L3
          trust  L1
```

ヘッド引数がベクタであるようなクローズ群では，まずｅｎｃｏｄｅ＿ｖｅｃｔｏｒ命令でファンクタとアリティなどを用いてハッシュ値を計算した後，ｈａｓｈ＿ｏｎ＿ｖａｌｕｅでインデクシングする。
各グループのクローズの数が４以下程度の少数の場合は，ｈａｓｈ＿ｏｎ＿ｖａｌｕｅ命令の代わりにｊｕｍｐ＿ｏｎ＿ｖａｌｕｅ命令により逐次探索を行う事もできる。

```
（例７）   p(1,X) :- !, .....
          p([L|Z],X) :- !, .....
          p(3,X) :- !, .....
          p(4,X) :- !, .....

          switch_on_term Ai,La,Ll,Lv,Le
     La:  jump_on_value Ai,Ll,1    % jump to Ll if Ai=1
          jump_on_value Ai,L3,3    % jump to L3 if Ai=3
          jump_on_value Ai,L4,4    % jump to L4 if Ai=4
     Lv:  fail
     Ll:  instructions for clause "p(1,X):- ..."
     L3:  instructions for clause "p(3,X):- ..."
     L4:  instructions for clause "p(4,X):- ..."
     Ll:  instructions for clause "p([L|Z],X):- ..."
     Lc:  try    L0       % caller argument =  undefined/else
          retry  Ll
          retry  L3
          trust  Ll
```

(3) ｊｕｍｐ＿ｏｎ＿ｎｏｎ＿ｌｉｓｔ，ｅｘｅｃｕｔｅ＿ａｎｄ＿ｓｗｉｔｃｈ命令など

ヘッドの第１引数がリストまたはＮＩＬだけの述語や，定数と変数だけからなる述語は結構多い。そのような場合はｓｗｉｔｃｈ＿ｏｎ＿ｔｅｒｍ命令の最適化命令であるｊｕｍｐ＿ｏｎ＿ｎｏｎ＿ｌｉｓｔ命令やｊｕｍｐ＿ｏｎ＿ｎｏｎ＿ｃｏｎｓｔａｎｔ命令が用いられ，処理の高速化とコードの削減が図られる。
また，ｅｘｅｃｕｔｅする先の述語の最初の命令がｓｗｉｔｃｈ＿ｏｎ＿ｔｅｒｍの場合は連続して２回の分岐が行われる事になるので，それを１回の分岐で済ませるためにｅｘｅｃｕｔｅ＿ａｎｄ＿ｓｗｉｔｃｈ命令も用意されている。

```
（例８）   p([],X) :- !, .....
          p([L|Z],X) :- !, .....

          jump_on_non_list  Ai,Lnil,Lels,Lvar
    Ll:   instructions for clause "p([L|Z],X):- ..."
    Lnil: instructions for clause "p(1,X):- ..."
    Lvar: try    Lnil     % caller argument =  undefined
          trust  Ll
    Lels: fail

  jump_on_non_list で呼出し引数がリストと判定されると，Ll のクローズは決定的に（ｔｒｙ命令なしで）実行される。
```

１．３　カット命令

ソース上はオールタネイト・クローズがあっても，インデックス命令により，実行時には決定的に呼び出される場合が多い。その場合のネック・カット（ボディ部の最初に現れるカット）はｎｏｐとなるが，それをできるだけ早く判定するために専用のカット命令が用意されている。また，カットと意図的な失敗（ｆａｉｌ）との組み合わせも結構頻繁である事から，それらを組み合わせた命令を用意して，高速化を図っている。
以下に，クローズの形とカットの位置による命令の違いを示す。

a. 環境の必要のないクローズのネック・カット．．．ｃｕｔ＿ｍｅ命令

```
a-1.    p(1,2) :- !.
a-2.    p(X,Y) :- !, add(X,Y,3).
a-3.    p(X,Y) :- !, q(Y,X).
a-4.    p(X,Y) :- !, add(X,Y,Z), q(Z,X).
a-5.    p(X,Y) :- add(X,Y,3), !.    （この場合組込み述語はゴールと見なさない）
```

b. 環境が生成されるクローズのネック・カット
．．．ｃｕｔ＿ａｌｌｏｃａｔｅｄ＿ｍｅ命令

```
b-1.    p(X,Y) :- !, q(X), r(Z).
b-2.    p(X,Y) :- !, q(X,Z), add(X,Z,Y).
```

c. それ以外のクローズのカット

```
c-1.    p(X,Y) :- q(X), !.
c-2.    p(X,Y) :- q(X), !, r(Z).
```

d. 環境の必要のないクローズのネック・カット＆フェイル
．．．ｃｕｔ＿ｍｅ＿ａｎｄ　ｆａｉｌ命令

```
d-1.    p(1,2) :- !,fail.
d-2.    p(X,Y) :- add(X,Y,3), !, fail.
```

e. それ以外のクローズのカット＆フェイル．．．ｃｕｔ＿ａｎｄ＿ｆａｉｌ命令

```
c-1.    p(X,Y) :- q(X), !, fail.
```

１．４　ＰＳＩ－ⅡとＷＡＭの違い

ＰＳＩ－ⅡとＷＡＭの違いを以下に列挙する。

(1) 機械命令シンタックス

ＰＳＩ－Ⅱでは引数レジスタはＡ０からＡ３１としているのに対し，ＷＡＭではＡ１からとしている（引数レジスタ個数は規定していない）。

また，ｇｅｔ／ｐｕｔ／ｕｎｉｆｙ命令において，オペランドがテンポラリ変数（後述）かパーマネント変数（後述）かの区別をＷＡＭでは以下のようにオペランドのニーモニック（ＸあるいはＹ）だけでおこなっているのに対し，ＰＳＩ－Ⅱでは命令ニーモニックの末尾に”　ｔ”または”＿ｐ”を付けて区別している。
さらに，述語呼出し引数のための引数レジスタとテンポラリ変数やパーマネント変数などのオペランドの順番がＷＡＭとＰＳＩ－Ⅱでは反対となっている。

ＷＡＭ：

```
ｇｅｔ＿ｖａｌｕｅ　　Ｘｊ，Ａｉ
ｐｕｔ＿ｖａｌｕｅ　　Ｙｎ，Ａｉ
```

ＰＳＩ－Ⅱ：

```
ｇｅｔ＿ｖａｌｕｅ＿ｔ　Ａｉ，Ｘｊ
ｐｕｔ＿ｖａｌｕｅ＿ｐ　Ａｉ，Ｙｎ
```

(2) 複合項とベクタ

ＰＳＩ－Ⅱでは「ベクタ」というデータ・タイプがあり，複合項（ｃｏｍｐｏｕｎｄ　ｔｅｒｍ）はベクタの第１要素が定数であるもの（即ちファンクタ）として表現される。
シンタックスは {X,Y,3} や {f,X,Y} または f(X,Y) などのように表記される。
ＷＡＭでは　ＤＥＣ－１０　Ｐｒｏｌｏｇをベースにしているため，ベクタというデータ・タイプは無い。{f,X,Y} は複合項 '{}'(f,X,Y) であり，f(X,Y) とは別のものになる。

(3) インデクシング命令

ＰＳＩ－Ⅱでは前述のように，述語中のクローズの種類に応じてかなり細かい最適化を図っているのに対し，ＷＡＭでは

ｓｗｉｔｃｈ＿ｏｎ＿ｔｅｒｍ

ｓｗｉｔｃｈ＿ｏｎ＿ｃｏｎｓｔａｎｔ
（ＰＳＩ－Ⅱのｈａｓｈ＿ｏｎ＿ｖａｌｕｅに相当）

ｓｗｉｔｃｈ＿ｏｎ＿ｓｔｒｕｃｔｕｒｅ
（ＰＳＩ－Ⅱのｅｎｃｏｄｅ＿ｖｅｃｔｏｒ＋ｈａｓｈ＿ｏｎ＿ｖａｌｕｅに相当）

の３つのみが提案されている。

(4) マージ命令

ＰＳＩ－ⅡではＷＡＭで提案された基本的な命令の内，以下のような頻繁に連続する命令の幾つかを組み合わせたマージ命令を設け，速度およびコード量の点での最適化を図っている[8]。

ＷＡＭ		ＰＳＩ－Ⅱ
deallocate execute	--->	execute_with_deallocate
deallocate proceed	--->	proceed_with_deallocate
get_list unify_variable unify_variable	--->	get_var_var_list
unify_variable get_list	--->	unify_list

など

(5) その他

ＷＡＭではカットや組込み述語のインプリメントは何も提案されていない。
ＰＳＩ－Ⅱではクローズの形に応じた幾つかのｃｕｔ命令を設けている。
組込み述語も，ＫＬ０に特有なものを含みすべて，それぞれに専用の命令により実行する。

２．スタック・メカニズム

２．１　３つのスタック

Ｐｒｏｌｏｇの述語の呼出しおよび復帰の制御にスタックを使用すれば良いと言うことは想像に難くない。処理系が制御情報やデータの格納に使用したメモリ領域は，不要になった時点で即座に解放し再使用できるからである。

ＰＳＩ－Ⅱでは伸長／縮小のタイミングが異なる３つのスタック，

Ｌｏｃａｌ　Ｓｔａｃｋ
Ｇｌｏｂａｌ　Ｓｔａｃｋ
Ｔｒａｉｌ　Ｓｔａｃｋ

を用いている。

以下に，各スタックの役割，伸長・縮小のタイミングについて解説する。

２．２　Ｌｏｃａｌ　Ｓｔａｃｋ

述語呼出しの環境（制御情報・変数値），および，バックトラック時の環境（制御情報・呼出し時の引数値）を保持するために用いられる。

述語呼出し環境として具体的には以下のような内容がローカル・フレーム（環境フレーム）という箱の中に入れられ，フレーム単位で Local Stack に積まれる。
ローカル・フレーム内に格納される変数をパーマネント変数と呼ぶ（後述）。
ＬｖｌｃとＢはＫＬ０の遠隔カットなどを実現するためのもので，ＷＡＭにはない。

	Ｅ　（親の呼出し環境）
	ＣＰ　（継続アドレス）
	Ｂ　（バックトラック環境）
	Ｌｖｌｃ　（呼出しレベル）
ＥＲ==>	Ｙ０　（パーマネント変数０）
	Ｙ１　（パーマネント変数１）
	：　　：
	Ｙｎ　（パーマネント変数ｎ）

バックトラック環境としては具体的には以下のような内容が積まれる。

BR==>	B （前のバックトラック環境）
	T （トレール・トップ）
	G （グローバル・トップ）
	AP（代替クローズアドレス）
	Lvlc （呼出しレベル）
	E （親の呼出し環境）
	CP （継続アドレス）
	A0 （呼出し引数0）
	A1 （呼出し引数1）
	: :
	An （呼出し引数n）

２．２．１ Ｌｏｃａｌ Ｓｔａｃｋの伸長

Local Stack が伸長するのは，基本的には以下の２つの場合である。
例外の発生などによる述語の動的呼出し時やネストした構造体のユニフィケーションなどにも Local Stack を使用するが，ここでは触れない。

(1) ２つ以上ゴールがあるクローズでの最初のゴール（述語）呼出し時

（例１） p(X,Y) :- q(X,Z), r(X,Y,Z), s(Z).
---

q の呼出しからの復帰時に r の実行を続けるために，q の呼出し時点の環境を Local Stack にセーブ(allocate)。

(2) オールタナティブ・クローズがあるような述語の呼出し時

（例２） q(1,X) :- t(X).
---
q(2,X).

q(1,X) の呼出しが（ヘッド・ユニフィケーションまたは s(Z) の実行途中で）失敗した場合に，q(1,X) 呼出し時に環境を完全に戻して，q(2,X) の呼出しを行うためにバックトラック環境をセーブしておく(try)。

２．２．２　Ｌｏｃａｌ　Ｓｔａｃｋの縮小

Local Stack が畳まれるのは以下の３つの場合である。

(1) ２つ以上ゴールがあるクローズ内の最後のゴールの呼出し時

（例３）　p(X,Y) :- q(X,Z), r(X,Y,Z), s(Z).

もはや s の呼出しからの復帰時に続行すべきゴールが無い場合は，p の呼出し元へ直接制御を戻す。そのために，s の呼出し時点で s の環境を Local Stack から取り除いておく(deallocate)。

(2) バックトラック時

（例４）　q(1,X) :- t(X).

　　q(2,X).
　　---*1
　　q(3,X).
　　---*2

*1 :
バックトラック時には，バックトラック環境をリストアした後，その環境情報が積まれた直後まで，Local Stack を畳む(fail)。

*2 :
最後のオルタナティブ・クローズへバックトラックした場合には，そのバックトラック環境はリストア後は不要となるため，Local tack はバックトラック環境のセーブされた前の位置まで畳まれる(fail)。

(3) カット（！）時

カットは，それを含むクローズのためのバックトラック環境を無効化する。そのバックトラック・フレームが Local Stack の先頭にあった場合に限り，その部分までスタックを畳む(cut)。

２．３　Ｇｌｏｂａｌ　Ｓｔａｃｋ

不要になったメモリ領域の解放はスタックを畳む事，即ち，スタック・トップをある範囲まで戻す事によって行われるが，その範囲に１語でも解放出来ないものが含まれているとそのスタック全体が畳めなくなってしまう。

いわゆる制御スタックである Local Stack は決定的な呼出し，つまり，戻り先を記憶する必要がないような呼出し時に畳まれる事を意図して設けられている。
ところが，一見決定的な状況であっても解放できないようなものがある。その１つが構造体である。

Local Stack 上には述語呼出しの環境の一つとして（パーマネント）変数領域が確保される。ＰＳＩ－Ⅱの場合には未定義状態の変数にはそれぞれ１語の領域（変数セル）が確保される。この未定義変数が環境が生成されたずっと後に，構造体にユニファイされる場合には，その構造体を変数セルに入れる事はできない。 従って，構造体は別のスタック，即ち Global Stack に置き，変数セルにはその構造体へのポインタが置かれる。

Global Stack 上に生成される構造体の生成順序は一般に述語呼出しの順序とは一致しないため，その解放は Local Stack と同時には行えない。逆に言うと，Local Stack の解放時に解放できないようなデータ類は Global Stack に置かれる。

このような意味から，Global Stack 上には構造体以外に，バインド・フックやオン・バックトラックなどの述語のフック情報なども格納されるが，ここでは触れない事にする。

２．３．１　Ｇｌｏｂａｌ　Ｓｔａｃｋの伸長

Global Stack が伸長するのは，基本的には以下の場合だけである。

(1) 構造体データの生成時

```
（例５）  p(X,Y) :- q( f(X), [1|Y], Z ), r(Z).
                    -- -  - - -
          q( f(A),B,B ).
```

q の呼出し前に f(X) や [1|Y] が Global Stack 上に生成され，それへのポインタを q への引数とする。
なお，クローズ p において，ボディ q の引数として初めて現れる変数 Z のために，p の呼出し環境内に１語の変数セルが確保されているが，q(f(A),B,B) とのユニフィケーションにより [1|Y] が Z の値として返ってくるため，実際にはZ にはポインタのみが入る。

２．３．２　Ｇｌｏｂａｌ　Ｓｔａｃｋの縮小

Global Stack が畳まれるのは以下の場合のみである。

(1) バックトラック時

> バックトラック時には，復帰すべきバックトラック環境が積まれた以後のすべての内容が不要となるため，Global Stack Top を縮めるだけでよい。

２．４　Ｔｒａｉｌ　Ｓｔａｃｋ

バックトラック時にそれまでに行われたユニフィケーションをすべて無効化（ｕｎｄｏと称す）するための情報が積まれる。

バックトラック時には，バックトラック環境が生成された以後の情報は，原則としてLocal / Global Stack の縮小と同時に解放されるが，バックトラック環境より以前に確保されていた変数セルへのユニフィケーションによる値の代入はもとに戻らない。そこで，このようなユニフィケーション時には，値の代入を行った変数セルのアドレスをTrail Stack に記憶しておく。

２．４．１　Ｔｒａｉｌ　Ｓｔａｃｋの伸長

(1) バックトラック環境よりも古い変数セルへの値の代入時

（例６）
```
p :- q(X),r(Y).
q(X) :- s1(X,Z).
q(X) :- s2(X,Z).
s1(1,a) :- fail.
s2(2,a).
```

s1(1,a) によってユニファイされた変数Ｘおよびｚのうち，Ｘのみがトレイルされる。何故なら，変数Ｚの領域は q(X) の第２クローズを実行するためのバックトラック環境が生成された後に作られたもので，バックトラックにより自動的に回収されるのに対し，Ｘは p の環境の中にあるため，スタックの畳み込みだけでは q(X) の呼出し時点の状態に戻せないからである。

２．４．２　Ｔｒａｉｌ　Ｓｔａｃｋの縮小

(1) バックトラック時

Trail Stack の先頭から，バックトラック環境が生成された時点のTrail Stack の先頭（バックトラック環境の中に保持されている）までに記憶している変数セルを ｕｎｄｏ し，Stack Top もその時点まで戻す。

(2) ｃｕｔ（！）時

（例７） p :- q(X).
q(X) :- r1(X,Y), s(Y).
q(X) :- r2(X,Y), s(Y).
r1(1,2) :- !, ...
r1(3,4) :- ...

上記 r1(1,2) のユニフィケーションで，呼出し側変数 X,Y にそれぞれ 1,2 がユニファイされる。その時点では r1(3,4) なるオールタネィト・クローズがあるため，２つのユニフィケーションはトレールされる。ところが ！ により，r1(3,4) はなくなる事になり，Y に対する 2 のユニファイは決定的となる。ところが X へのユニファイは q(X) の存在により依然として非決定的であり，トレールはしておかなければならない。
Y に対するトレールをキャンセルするかどうかは処理系の都合による。つまり，トレールされたデータをすべて必要最小限のものにしておく方法と，ｕｎｄｏ 処理においてトレールされたデータを取捨選択する方法のどちらでもよい。いずれにしろ，無効なトレール・データまで ｕｎｄｏ する事はできない。何故なら，トレールから指されている Local Stack 上の変数セルの場所は，Local Stack の伸縮に伴い別目的に再利用されている可能性があるからである。ＰＳＩ－Ⅱでは速度上の得失と Garbage Collection の容易さからトレール・データの有効性を保証する方法を採用している。この保証のため，カットされるバックトラック環境が生成された時点から Trail Stack の先頭までのデータの内，バックトラック・ポイントが変わる事によりトレールする必要が無くなったものを取り除き，隙間の詰め合わせを行うが，これを Tidy Trail と称している。

３．制御レジスタ

ＨＲ　（Top of Heap）

３本のスタックのほかに，プログラム（コード）などの副作用的書き替えのできるエリアが用いられ，これをＨｅａｐ　Ａｒｅａ（ヒープ・エリア）と称する。ＨＲはこのＨｅａｐ　Ａｒｅａの先頭を示すレジスタである。
Ｈｅａｐ　Ａｒｅａにはヒープ・ベクタやＥＳＰオブジェクトなども置かれる。これらが動的に生成される時にこのレジスタは更新（増加）される。ＨＲはＧＣ（GarbageCollection）以外では減少する事はない。

ＬＲ　（Top of local stack）

Ｌｏｃａｌ　Ｓｔａｃｋの先頭を示すレジスタである。

ＧＲ　（Top of Global Stack）

Ｇｌｏｂａｌ　Ｓｔａｃｋの先頭を示すレジスタである。

ＴＲ　（Top of trail stack）

Ｔｒａｉｌ　Ｓｔａｃｋの先頭を示すレジスタである。

ＳＡＲ　（Top of System Area）

主にＯＳ（ＳＩＭＰＯＳ）とファームウェアとの交信に用いるため，ＰＳＩ－ⅡではＳｙｓｔｅｍ　Ａｒｅａと呼ぶエリアも備えている。
Ｓｙｓｔｅｍ　Ａｒｅａはファームウェアによる ＧＣ時の作業領域にも使用される。また，ＳＩＭＰＯＳのプロセス管理ブロック（ＰＣＢ）もここに置かれるため，プロセスの新規生成に伴って，Ｓｙｓｔｅｍ　Ａｒｅａは拡張される。
ＳＡＲはこのエリアの先頭を示すレジスタである。

ＳＲ　（Structure Pointer）

構造体要素のユニフィケーションに用いられるレジスタで，ｇｅｔ命令でセットされ，引き続くｕｎｉｆｙ命令で要素の参照に使用された後，＋１される。

ＣＰＲ　（Continuation Program Pointer）

１つのゴールからの復帰先のコード・アドレスと保持するためのレジスタである。ｃａｌｌ命令でセットされ，ｐｒｏｃｅｅｄ命令でその内容が使用される。

ＥＲ　（Last Environment）

Ｌｏｃａｌ　Ｓｔａｃｋ上の最新の環境フレームを示す。実際には，処理系の都合により，フレームの先頭ではなく，パーマネント変数領域の先頭を指し示す。

ＢＲ　（Last Choice Point）

Ｌｏｃａｌ　Ｓｔａｃｋ上の最新のバックトラック環境フレームを示す。

ＧＢＲ　（Backtrack Point of Global Stack）

最新のバックトラック環境がＬｏｃａｌ　Ｓｔａｃｋに積まれた時点のＧｌｏｂａｌ　Ｓｔａｃｋ　Ｔｏｐの位置を保持する。ＢＲの指すフレームの中のＧＲのエントリと同じ内容がキャッシュされている事になり，Ｔｒａｉｌ　Ｃｈｅｃｋ（未定義変数への値の束縛時にトレールすべきかどうかの判定）に使用される。

ＬＶＬＣ（Level Counter）

述語呼出しのレベルを保持する。ＫＬ０特有の，遠隔カットやｓｕｃｃｅｅｄ述語の実行に使用される。

その他にも，以下のようなＫＬ０特有の実行制御に用いられるレジスタがあるが，ここではそれらの説明は省略する。

ＮＥＲ　（Number of Remaining Elements）
ＯＮＴＲ（On_backtrack Top Pointer）
ＢＨＰＲ（Bind_hook Pointer）
ＥＸＴＲ（Exception_hook Top Pointer）
ＳＦＲ　（Return Point of Local Stack in If_then_else Mode Structure Compare）

４．レジスタ・メカニズムと変数の分類

４．１　引数レジスタ

前述のように述語呼出し時の引数値はレジスタにおかれる。これは，述語呼出し命令に先駆けて，ｐｕｔ命令によって行われる。

```
（例１）   p(X) :- q(1,X), r(X).
                 - -
                put_value_t A1, A0      % A1 <-- A0
                put_integer A0, "1"     % A0 <-- 1
                call        q           % q の呼出し
```

ｑの呼出し時に引数レジスタ（Ａ０～Ａ３１）の先頭Ａ０から順番に引数値（整数１と変数Ｘのその時の値）を用意する。
先にＡ０に１をｐｕｔするとＸの内容が破壊されるので，この例では先にＡ１へＸの値をｐｕｔしている。

４．２　パーマネント変数

レジスタ上で渡された引数値は，最初のユーザ・ゴールの呼出しからの復帰後は保証されない。従って，２つ目以降のゴール（組込み述語も含む）に使用する引数は，最初のユーザ・ゴールが呼び出されるまでの間に環境フレーム内の変数領域にｇｅｔ命令によってセーブされる。この変数の事をパーマネント変数と呼ぶ。ただし，ヘッド・ユニフィケーションで具体的なマッチングがとられた引数に関しては，セーブする必要はない。
前述のように２つ以上のゴール（正確には，ユーザ・ゴールの後ろに１つ以上のユーザまたは組込み述語のゴール）があるクローズでは，最初のユーザ・ゴール呼出しからの復帰後に後続ゴールの実行をする必要から，必ずローカル・スタック上に環境フレームが積まれる。このフレームの中に（もしあれば）パーマネント変数の領域も取られる。この操作は ａｌｌｏｃａｔｅ命令で行われる。

```
（例２）   p(X,1) :- q(1,X), r(X).
            - -
        allocate          1          % one permanent var for Y0
        get_variable_p    A0, Y0     % Y0 <-- A0
        get_integer       A1, "1"    % A1 == 1 ?

  q の呼出しからの復帰後は，Ａ１上のＸの値は破壊されている。
  従って，Ｘの値はｇｅｔ＿ｖａｒｉａｂｌｅ命令で環境フレーム内
  にセーブされる。
```

２つ目以降のゴール呼出しのための引数の準備において，パーマネント変数を取り出すのもｐｕｔ命令で行われる。

```
（例３）   p(X,1) :- q(1,X), r(2,X), s(X).
                              - -     -
        put_integer A0, "2"          % A0 <-- 2
        put_value_p A1, Y0           % A1 <-- Y0
        call          r              % r の呼出し
        put_value_p A0, Y0           % A0 <-- Y0
        execute_with_deallocate s    % 環境フレームの廃棄と
                                     % s の呼出し
```

４．３　テンポラリ変数

パーマネント変数にする必要はないが，破壊する事のできない引数レジスタを避けたりするために一時的に使用するレジスタをテンポラリ・レジスタ，そのレジスタに乗せられる変数をテンポラリ変数という。テンポラリ・レジスタといっても，ハードウェア的には引数レジスタと同じものを使用するため，場合によっては，後続のゴール呼出し時の引数としてそのまま使える（即ち，ｐｕｔ命令の省略，または先取り）事もある。

なお，一時的というのは，ヘッド・ユニフィケーションが開始してから最初のゴールが呼び出されるまで，もしくは，１つのゴールから復帰後，次のゴールが呼び出されるまでの間である。

（例４）　p([L | R],L) :- q(R,2,L).

```
                get_list          A0          % A0 == List
                unify_variable_t  X2          % X2(=A2) <-- L
                unify_variable_t  X0          % X0(=A0) <-- R
                get_value_t       A1, X2      % unify L on X2 and L on A1
                put_integer_t     A1, "2"     % A1 <-- 2
```

第１引数のリストのｃａｒ部の変数 L はひとまずＸ２レジスタ（Ａ２レジスタと同一）に置いておき，第２引数とのユニフィケーションに備える。
第１引数のリストのｃｄｒ部の変数 R は，後続の q の呼出しまで使用しないが，Ａ０は解放できるので，q の呼出し時のｐｕｔを省略するために，Ｘ０（＝Ａ０）に置いてしまう。
ユニフィケーション(get_value) が成功すると，q の呼出し準備にかかるが，R および L はそれぞれすでに Ａ０，Ａ２レジスタに乗っているので，ｐｕｔするのはＡ１への”２”だけとなる。

ネストした構造体のユニフィケーションや生成に，テンポラリ・レジスタが良く使われる。

（例５）　p(L,R) :- q([L,R]).

```
                get variable_t    A0, X2      % X2 <-- L
                put_list          X3          % X3 <-- New List for [R]
                unify_value_t     A1          % car of X3 <-- R
                unify nil                     % cdr of X3 <-- nil
                put_list          A0          % A0 <-- New List (outmost)
                unify_value_t     X2          % car of A0 <-- L on X2
                unify_value_t     X3          % cdr of A0 <-- [R] on X3
                execute           q           % q の呼出し
```

q の呼出し引数 [L,R] = [L | [R | nil]] を inner first 即ち，[R | nil] を先にＸ３上に生成し，後から [L | (X3)] を生成する方法である。

（例６）　p(L,R) :- q([L,R]).

```
                    -----
        get_variable_t    A0, X2    % X2 <-- L
        put_list          A0        % A0 <-- New List (outmost)
        unify_value_t     X2        % car of A0 <-- L
        unify_variable_t  X3        % X3<--REF!cdr, cdr<--undf
        get_list          X3        % cdr,X3<--New List for [R]
        unify_value_t     A1        % car of (cdr of A0)<-- R
        unify_nil                   % cdr of (cdr of A0)<-- nil
        execute           q         % q の呼出し
```

q の呼出し引数 [L,R] = [L | [R | nil]] を outer first 即ち，[L | undf] を先にＡ０上に生成し，後からそのｃｄｒ部に [R | nil] を生成する方法である。
unify_variable_t では一旦リストの要素を undf に初期化し，その要素へのポインタをＸ３上に乗せる。その直後の get_list で，その要素へ，新しく生成するリスト [R | nil] へのポインタをセットするために，いかにも無駄である。
ＰＳＩ－Ⅱでは，上記 unify_variable_t X, put_list X をマージした unify_list 命令を用意し，最適化している。unify_list を使用する場合は inner first の場合よりも高速で，かつ，コード量も削減する事ができる。

```
        put_list          A0        % A0 <-- New List (outmost)
        unify_value_t     X2        % car of A0 <-- L
        unify_list                  % cdr of A0 <-- New List
        unify_value_t     A1        % car of (cdr of A0)<-- R
        unify_nil                   % cdr of (cdr of A0)<-- nil
```

４．４　ローカル変数

クローズのゴール部に初めて現れる変数の事である。変数セルはそのクローズの環境フレーム内もしくはグローバル・スタック上に取られる。
どちらに変数セルが取られるかは，以下の基準による。

グローバル・スタック上 ... 環境の必要のないクローズ，または，環境はあったが，最後のゴールで初めて現れる変数のため，そのゴールの呼出しまでに環境が捨てられる場合
パーマネント変数として環境フレーム内 ... 上記以外の場合

ただし，一旦パーマネント変数として生成されたローカル変数で，最後のゴール呼出しにも使用されるものには注意を要する。即ち，その変数の値が最後のゴール呼出し時点でもまだ未定義状態（他の未定義変数とのユニフィケーションが行われている場合は除く。ここでは本当のバージン・セルの場合だけを指す。）であるとすると，その変数セルは間もなく環境と共に捨てられてしまうため，グローバル・スタック上に引越しさせなければならない（グローバライズと言う）。その変数が定義済みならば，その内容を引数レジスタにコピーするだけで，その変数セル自身はもはや不要である。このような，最後のゴールとそうでないゴールの両方に現れるローカル変数を unsafe と言い，put_unsafe_value と言う特別な命令で動的な判断を行う。

（例７）　p(X,1) :- q(X,Y,Z), r(W,Z).
　　　　　　　- -　　- -

```
        put_variable_p    A1, Y0    % A1 <-- REF!(init(Y0=Y))
        put_variable_p    A2, Y1    % A2 <-- REF!(init(Y1=Z))
        call              q         % q の呼出し
        put_variable_t    A0, X0    % A0 <--
                                          REF!(init(global top))
        put_unsafe_value  A1, Y1    % A1 <-- Y1 (if Y1=Z bound)
              % A1 <-- REF!(init(global top)) (if Y1=Z unbound)
        execute_with_deallocate     % 環境フレームの廃棄と
                                    % r の呼出し
```

Y,Z,W はヘッド部に現れない変数，即ちローカル変数である。
Y と Z はパーマネント変数にされ，W はグローバル・スタック上の変数とされる（グローバライズ）。W は，このクローズの環境がなくなった以後の参照可能な変数となる事からグローバル変数とも呼ばれる。しかし，他言語のでのグローバル変数とは意味が異なるので，注意を要する。
なお，r の呼出し時の Z の値が未定義の場合はグローバライズする必要がある事から put_unsafe_value 命令が用いられている。

ローカル・スタックは出来るだけ早く畳もうとするスタックであるので，グローバル・スタックからローカル・スタックに向かってポインタを張ってはいけない。これを守らないといわゆる”ダングリング・ポインタ”ができてしまうからである。
従って，ローカル変数の扱いには注意が必要である。例えば，２つの未定義変数をユニファイする場合，このポインタの向きに注意しなければならない。

```
（例８）p :- q(X,Y).
        q(X,[f|X]) :- r(Z), s(Z), t([X|Z]).
              ---          ---      ---
        allocate              2      % 2 permanents for X and Z
        get_variable_p      A0, Y0   % (X,
        get_list            A1       %    [
        unify_atom           f       %     f|
        unify_local_value_t A0       %       X]),
        put_variable_p      A0, Y1   %  (Z),
        call                 r       % r
        put_local_value     A0, Y1   %       (Z),
        call                 s       %      s
        put_list            A0       %            ([
        unify_value_p       Y0       %              X|
        unify local_value_p Y1       %                Z])
        execute_with_deallocate t    %           t
```

ヘッド部の最初の X は呼出しもとの未定義ローカル変数の可能性があるため，[f|X] とのユニフィケーション時に注意を要する。即ち，呼出し元 Y が未定義で Write Mode となった場合，グローバル・トップにリストを生成するが，リストの ｃｄｒ部に X の値を入れるにあたり，もし X が未定義ローカル変数ならば，その変数へのポインタを入れる事が出来ない。従って，その変数をグローバライズしなければならない。
s(Z) の Z は自環境の未定義変数の可能性があるため，通常の put のように，パーマネント変数の内容をそのまま引数レジスタにセット出来ない（レジスタ上に未定義変数が乗ってしまうため）。
put_local_value は，パーマネント変数が自環境の未定義変数かどをテストし，そうである場合はその変数へのポインタをセットするようにしている。

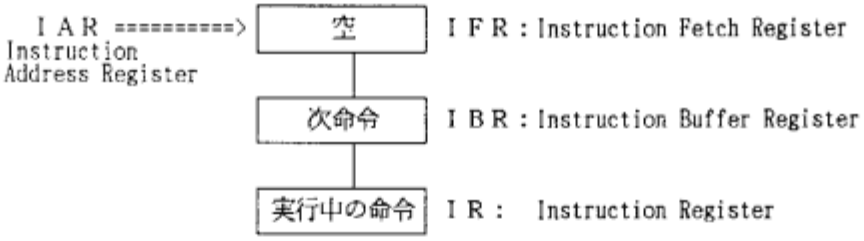
５．ファームウェア

ＰＳＩ－Ⅱはマイクロ・プログラムで制御される３段の簡単なパイプライン機構と，
ＥＯＰ（Op Code 分岐）時の割込み判定機構を備えている。
スタックの伸長に伴うメモリの割り当て要求処理など，例外的処理はすべて割込み処理の一環として行われるため，通常の処理は極めて高速に実行できるようになっている。

この章では，ファームウェアによる機械語実行や割込み処理の一環として行われるシステム制御機構などについて概説する。

５．１　パイプライン制御

ＰＳＩ－Ⅱには３段のパイプライン用命令レジスタＩＲ，ＩＢＲ，ＩＦＲを備えている。
ＩＲは実行中の命令を，ＩＢＲは次の命令を，ＩＦＲは更に次の命令を保持するために用いられる。
各機械語命令の実行開始時点では，ＩＲに現命令，ＩＢＲに次の命令がそれぞれ入っており，プログラム・カウンタであるＩＡＲは次の次の命令のアドレスを保持している事を保証している。



各機械命令の中では本来の処理の他に，次命令への移行の準備を行わなければならない。
それには２つの方法がある。

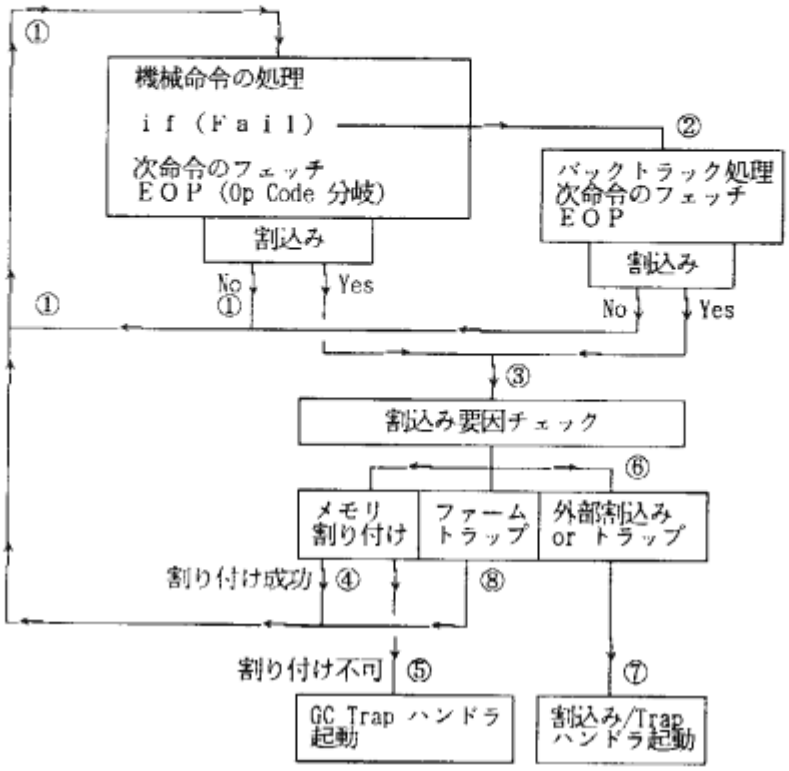
(1) ＩＦＲに次々命令を読み込みＩＡＲを１つ進める（ｐｒｅ＿ｆｅｔｃｈ）。
最後のマイクロ命令でＥＯＰを発行する。
ＥＯＰのマイクロ・オーダはハードウェア的には以下のような動きとなる。
- ＩＢＲの内容をＩＲに，ＩＦＲの内容をＩＢＲにシフトする。
- シフト時にＩＢＲに入っていた命令（次命令）のオペレーション・コードに基づき，マイクロ・ルーチンの飛び先アドレスを決定する。

(2) 最後のマイクロ命令でＥＯＰを発行すると同時に，ＩＢＲに次々命令を読み込みＩＡＲを１つ進める（ｉ＿ｆｅｔｃｈ＿ｅｏｐ）。
この場合はＩＢＲには，ＩＦＲの内容ではなくメモリから読み込んだ次々命令が入る。

通常は（1）の方法が用いられるが，ＩＦＲを最後まで作業用に使用する場合や，１ステップで命令の処理を終了させる場合などに（2）の方法が使用される。

５．２　基本処理フロー

ＰＳＩ－Ⅱの機械命令の基本処理フローを以下に示す。



機会命令を次々に実行している通常の場合は，① のルートで処理されている。
ユニフィケーションの失敗時には，バックトラック環境の復帰を行い，オールタネイト・クローズにプログラムの流れ（プログラム・コンテクスト）を切り換える（②）。

もし，機械命令実行中にメモリ割り付け要求が発生（ハードウェア検出）したならば，ＥＯＰ時の割込みチェックにかかり（③），ファームウェアによるメモリ割り付けが成功した場合には（④）次命令の実行に戻る。実メモリの不足，あるいはアドレス変換用テーブル（Page Pap Memory）のエントリの確保の失敗などにより，メモリ割り付けが不成功の場合には，ＧＣ　Ｔｒａｐハンドラが起動される（⑤）。

外部割込みの発生時には，やはりＥＯＰの時点で検出され，割込み要因を判別した後に要因別のハンドラを起動する（③,⑥,⑦）。

上図には示していないが，１命令で１ページ以上のメモリを消費する処理や，非常に長い時間を要する処理中には，ファームウェアでも割込み検出を行う事により，メモリの割り付けの保証や，レスポンスの確保を行っている。

その他の例外的処理（⑧）としては，トレース・エクセプション，バインド・フック・トラップ，コンソール・ブレークなどがあり，やはり，割込みと同様に扱われる[9]。即ち，例外事項を検出したファームウェア（コンソール・ブレークの場合はユーザ自身）は割込みフラグを立てるだけとし，その処理を命令の切れ目まで遅延させるようにしている。

トレース・エクセプション発生時は，次命令がトレースすべき種類の命令かどうかを判定し，もしそうであればトレース・ハンドラへコンテクスト切り換えを行う。もしそうでなければＮＯＰ，すなわち，そのまま次命令へ進む。

バインド・フック・トラップでは，ヘッド用命令を実行中にはＮＯＰとして次命令へ進み，ボディ用命令に入る直前にフックされているハンドラを起動する。

コンソール・ブレークはファームウェア・レベルのデバッグに使われる，機械命令単位のトレーサで，コンソール画面に次命令を表示したり，予め指定された条件が成立した時点でＣＰＵを停止させる機能をサポートしている。プログラム・コンテクストの切り換えが行われる事は（停止時にユーザが行わない限り）ない。

おわりに

水平型マイクロプログラム制御マシンＰＳＩ－Ⅱは，最小限のハードウェアにより最大限のＫＬ０（Ｐｒｏｌｏｇ）性能を引き出す事を目標に設計された実用マシンである。本メモでは，ＰＳＩ－ⅡにおけるＫＬ０処理系の機械命令と処理方式の概要を解説した。

実行プログラムの性質にもよるが，ＰＳＩ－Ⅱの実行時間の大部分は本メモで解説した処理内容で占められる。実際にセルフ・コンパイラの実行時間を評価したところによると，全実行時間の約３０％が ｐｕｔ／ｇｅｔ／ｕｎｉｆｙなどの引数操作とユニフィケーション，約１４％が ｃａｌｌ／ａｌｌｏｃａｔｅ系，約２１％が ｔｒｙ／クローズ・インデックス系，約１３％がバックトラック処理であった。この数字からも，基本処理部の効率が全体の性能に及ぼす影響がいかに大きいかがわかる。

本メモでは述べなかったが，ＰＳＩ－Ⅱのファームウェアは１６ＫｗのＷＣＳ容量を利用し，ＥＳＰ／ＳＩＭＰＯＳマシンとしての非常に多くの機能をサポートしている。たとえば，ＥＳＰのメソッド呼出し／スロット・アクセス機能，エクセプションを利用した高速なトレース機能，マーク／コンパクション方式のＧＣ，ＫＬ０特有のＢｉｎｄ＿ｈｏｏｋ／Ｏｎ＿ｂａｃｋｔｒａｃｋ／Ｒｅｍｏｔｅ＿ｃｕｔ述語，ＳＩＭＰＯＳのプロセス切り換えをはじめとするシステム制御機能，リスト／スタック・ベクタ／ヒープ・ベクタ／ストリング等の操作を行う組込み述語などである。これらの実装にあたって特に留意し，工夫した事は，基本処理の途中に突然，例外的に起動される機能，即ち，Ｂｉｎｄ＿ｈｏｏｋやエクセプション機能などを，通常の処理効率を低下させることなく実現することであった。

これらの機能やその実装方式については，また機会を改めて解説する予定である。また，これらの諸機能のうち，全体性能に本当に寄与しているもの，ファームウェアでしかできないものの他に，ソフトウェアなどの他の手段を用いても十分であるものなどがあるはずで，それらの評価も今後の課題と考えている。

参考文献

[1]:K.Taki, et.al., "Hardware Design and Implementation of the Personal Sequential Inference Machine(PSI)", Proc. of FGCS'84, 1984.

[2]:M.Yokota, et.al., "A Microprogrammed Interpreter for the Personal Sequential Inference Machine", Proc. of FGCS'84, 1984.

[3]:K.Nakajima, et.al., "Evaluation of PSI Micro-interpreter", COMPCON spring '86, 1986.

[4]:D.H.D Warren, "An Abstract Prolog Instruction Set", TR309, SRI, 1983.

[5]:中島 他, "マルチＰＳＩ要素プロセッサＰＳＩ－Ⅱのアーキテクチャ", 第３３回情報処理全国大会, 1986.

[6]:H.Nakashima, K.Nakajima, "Hardware Architecture of the Sequential Inference Machine : PSI-II", SLP'87, 1987.

[7]:池田 他, "ＰＳＩ－Ⅱシステムのファームウェア", 第３４回情報処理全国大会, 1987.

[8]:稲村 他, "ＰＳＩ－Ⅱの性能評価（２）", 第３５回情報処理全国大会, 1987.

[9]:中島 他, "ＰＳＩ－ⅡにおけるＫＬ０実行順序系組込述語の実現方式", 第３５回情報処理全国大会, 1987.