人工知能と対話技法 16－1
（1980. 7. 18）

Concurrent LiSP と そのインタプリタ

杉本重雄† 田畑孝一‡ 大野 豊†
†京大工学部 ‡京大情報処理教育センター

1. はじめに

従来、人工知能の研究分野においては、LiSP及びLiSPを基礎としたプログラミング言語が広く用いられてきた。これらの言語は研究対象となる問題の複雑さが増すにつれ、性能、機能の両面で多くの改良が行われてきた。機能面、特に制御構造を見た場合、バックトラッキング機能やコルーチン機能を導入してより柔軟な制御構造を目差した言語が提案されている。このような制御構造が導入された背景には、問題解決のためにたどるべき探索の道筋がたくさん存在し、それらを適当に選択しながら探索を進めていかねばならないという事情があるからである。そこでは、一つの道筋の探索が一つの探索プロセスとして取扱われている。我々は、上記のような探索問題において、並行プロセスの概念を導入することでより自然な理解しやすい記述が可能になると考える。すなわち、従来は探索プロセス間における制御の移行をプログラムテキスト上に記述していたのに対し、探索プロセスを一個の独立したオブジェクトとして記述することで、各探索プロセスの構造がより簡単になると考える。

我々は、LiSP 1.5を基礎として並行プロセス記述に必要でかつできるだけ少数の並行処理関数を導入した並行プログラミング言語Concurrent LiSPを開発した。Concurrent LiSPはLiSP 1.5の持つ性質を損なっておらず、しかも基本的で簡単な並行処理関数を与えることで、非常にたやすく並行プロセスシステムを記述できるように考案されている。

本稿では、従来の制御機構であるコルーチン等と並行プロセスシステムとの比較を行い、Concurrent LiSPの言語仕様、Concurrent LiSPインタプリタの構成について述べる。

2. 並行プロセスとLiSP系言語

現在までに開発されてきたLiSP系のプログラミング言語は、それぞれの目的に応じていろいろな特徴を備えている。特に制御機構の面においては、バックトラッキング機能やコルーチン機能が代表的なものであろう。これらは、柔軟で高度な記述機能をユーザに提供し、プログラミングにおけるユーザの負担を軽減している。たとえばPLANNER[2]はパターンマッチングにともなう自動バックトラッキング機能を持ち、また多くのLiSP処理系が、fail, failset 等のバックトラッキング用関数を備えている。一方、CONNIVER[5]やINTERLISP[8]では、柔軟性の乏しいバックトラッキングに代わり、より自由な制御構造を記述し得るコルーチン機能を準備している。コルーチン機能を用いることで、ユーザは擬似的にではあるがマルチプロセッシングを行なうことが可能になっている。

プログラミング言語の持つ機能から見た場合、上記のようにより自由な制御機構を取り入れるべく発展してきていると見ることができる。一方では、Hewitt等によって、制御構造をactorと呼ばれる自立したオブジェクトの間でのメッセージ交換として見るactor理論[9]が提唱されている。actorを用いたシステムは、actor本来の性質によって並列処理が仮定されており、また問題解決のためのプ

ログラムを、部分問題に対するスペシャリストの集りとして記述することが提案されている。

問題解決のためになされる探索において、探索すべき道筋が数多く存在し、しかも探索すべき道筋の効率的な選択が困難な場合，解に到達しそうないくつかの道筋を交互に，あるいは並行に探索していくことが望ましい。こうした要求に対してはコルーチン機能が有用であり CONNIVER や INTERLISP に導入されている。たとえば，INTERLISPのコルーチン機能を用いた例として、agendaなるスケジュール表によって最も望ましそうな探索の道筋を決定しながら探索を進めていくという方法が用いられる。以上のようにLISP系言語の対象分野において，並行した処理は制御機構として重要な役割を演ずるものであることがわかる。このように，複数の道筋を探索するような場合，一つの探索プロセスを独立したプロセスとして考えることは、論理を組み立てる上で必要なことであろう。しかしながら、コルーチン機能を利用した場合、プログラムテキスト上にコルーチン間での制御の移行を表わす操作を記述しなければならない。これは、コルーチンの数が増えた場合や、実行中にコルーチンの数が変化する場合において，制御の記述を非常に複雑なものにする可能性がある。

我々は、コルーチンを用いて表現していた問題を並行プロセスを用いて表現することで、より記述性に優れた記述方法が得られると考える。すなわち，コルーチン間での制御の移行を記述する代りに、各プロセスの動作と、それらの動作の共通環境に注意を向け，一つのプロセス内の制御構造を独立させる方が望ましいと考える。また，プロセス間でのデータのやりとりに関しても，プロセス間での通信機能を利用することができる。図1はINTERLISPのコルーチンプログラムとConcurrent LISPの並行プロセスプログラムの例である。

I.
```
(EQLEAVES  (L1 L2)
    (bind  LHANDLE1 LHANDLE2 PE EL1 EL2
       first (COUROUTINE PE LHANDLE1 (LEAVESC L1 LHANDLE1 PE) 'NO_MORE)
             (COUROUTINE PE LHANDLE2 (LEAVESC L2 LHANDLE2 PE) 'NO_MORE)
       do (EL1 ← (RESUME PE LHANDLE1)
          (EL2 ← (RESUME PE LHANDLE2)
          (if EL1 ~= EL2
              then  (RETURN  NIL))
       repeatuntil EL1 = 'NO_MORE finally (RETURN T)))
```

C.
```
(EQLEAVES (LAMBDA (L1 L2) (PROG (EL1 EL2)
      (STARTEVAL ('LHANDLE1 (LEAVESC L1 'EL1))
                 ('LHANDLE2 (LEAVESC L2 'EL2)))
    LOOP
     <CCR (AND (NOT (NULL EL1))(NOT (NULL EL2)))
          (PROG ()
           (CRPRINT (LIST 'EL1 EL1 'EL2 EL2))
           <COND ((NOT (EQUAL EL1 EL2)) (TERMINATE NIL))
                 ((EQUAL EL1 'NO_MORE)(TERMINATE T)))>
           (SETQ EL1 NIL) (SETQ EL2 NIL))>
    (GO LOOP) ) ) )
```

IがINTERLISP，CがConcurrent LISPのプログラム例である。関数LEAVESCは，与えられたリストの左端のアトムをIではresumeを使って，Cでは共有変数を介してEQLEAVES側へ伝える。

図1　INTERLISP と Concurrent LISP のプログラム例

3. Concurrent LiSP

Concurrent LiSPはLiSP1.5を基礎にした並行プログラミング言語であり，LiSP1.5本来の機能に加えて，並行プロセスシステムを記述する上で必要な機能を備えている。我々はConcurrent LiSPを開発するに当りLiSPの持つ単純さを失わないよう留意し，LiSP1.5の記法との間に差異が生じないようにした。本章では並行プロセス記述のための機能を中心に述べる。

◎ プロセス

まずはじめに，Concurrent LiSPにおけるプロセスの定義を示す。

「一つのプロセスはトップレベルダブレットまたはstarteval関数に示された形式を評価する。」

また，プロセスは以下の1～7の性質を持つ。

1. トップレベルダブレットを評価するプロセスを主プロセスと呼ぶ。
2. プロセスはstarteval関数を用いてプロセスを生成することができる。この時，前者を親プロセス，後者を子プロセスと呼ぶ。
3. 子プロセスは，それが生成された時点での親プロセスの環境を初期環境として持つ。すなわち，親プロセスの環境は子プロセスからも利用可能である。
4. プロセスは，与えられたダブレットまたは形式の評価を終了した時，あるいは親プロセスが終了した時終了する。
5. プロセスはユーザが与えた名前とインタプリタが与えた番号によって識別される。（主プロセスの場合，名前＝MAiN，番号＝1である。）
6. プロセスはプロセス値を持つ。プロセス値は，プロセスが評価するダブレットまたは形式の評価値である。終了していないプロセスのプロセス値は"UNDEF"なるAPVAL値が与えられる。
7. プロセスは，他のプロセスのプロセス値を参照することができる。

◎ 環境，セル，アトム，プロセス制御ブロック

次に，Concurrent LiSPにおける環境，セル，アトム，およびプロセス制御ブロック(pcb)の特徴を示す。

環境：主プロセスの初期環境はNil，非主プロセスの初期環境はその親プロセスがプロセス生成を行った時の環境である。従って，プロセスはその親プロセス内に束縛された変数への参照が可能であり，親プロセス内の変数を介して他のプロセスと通信することができる。

セル：全セル領域が全プロセスによって共有される。

アトム：全ての文字アトムはuniqueである。

pcb：プロセスの管理のために必要な情報，プロセス値等を格納する。プロセス間での直接的な通信に用いるMailboxが設けられている。pcbの構造は次章で詳しく述べる。後述のプロセスデータ処理関数は，主にpcbの持つデータを取り扱うことを目的としている。

◎ 並行処理関数

Concurrent LiSPにおいて最も重要な役割を演じる3つの並行処理関数を示す。並行処理関数は，プロセスの生成を表わすstarteval，プロセス間での同期，通信を表わすcr，ccrの3関数である。これらの評価規則はFSUBR型関数の評価規則に似るが，必ずスケジューラが介入するためCSUBRなる型を与えている。付録にstartevalの定義を示している。

1. starteval [fpl1; fpl2; ・・・; fplm]
　　ただし　fpli = (formi1 formi2)
意味：形式 formi1 の評価値を名前とし，formi2 を評価するプロセスを生成する。
　　formi2 の引数部の評価は関数の型に応じて異なる。（付録参照）
　　評価値はプロセス名のリスト（list [form11; form21; ・・・; formn1] ）。
例　f[x] と g[x] を評価するプロセス P と Q の生成

(STARTEVAL ('P (F X)) ('Q (G X)))

2. cr [form]　　（critical region 関数）
意味：この関数を実行するプロセスは，全セル領域の占有権を得て form を評価する。評価値は form の値。

3. ccr [form1; form2] （conditional critical region 関数）
意味：全セル領域の占有権を得た後，form1 を評価し Nil でなければ form2 を評価する。Nil の時は，占有権を放棄して Nil でなくなるまで待つ。
　　form1 には任意の形式を書くことができるが，副次的効果を持つものを書いてはいけない。（次章参照）　評価値は form2 の値。
例　変数 X を介しての同期，通信

(CR (SETQ X NIL))　　・・・プロセス P

(CCR (NULL X) (FUN V))・・・プロセス Q

　　プロセス P の動作でプロセス Q の待ち合せ条件が成立する。

(注)　cr, ccr の入れ子は許される。

◎ プロセスデータ処理関数

上記の並行処理関数は，並行プロセスシステムを記述する上で必要不可欠と考えられるものであり，これらだけで並行プロセスシステムを記述することは可能であるが，我々は記述能力をより良くするために，以下に示すプロセスデータ処理関数を用意した。これらは6種類に大別され，全体で21関数ある。

a. プロセスの状態（実行可，待ち合せ，終了）を調べる関数
termp[p]：プロセス p* が終了していれば T，そうでなければ Nil。
　　（*. p はプロセス名 or はプロセス番号）
asonterm[p]／osonterm[p]：プロセス p のすべての／いずれか一つの子プロセスが終了していれば T，そうでなければ Nil。
waitp[p]：プロセス p が待ち状態であれば T，そうでなければ Nil。
asonwait[p]／osonwait[p]：プロセス p のすべての／いずれか一つの子プロセスが待ち合せ状態であれば T，そうでなければ Nil。

b. プロセスの名前または番号を取り出す関数
self[]：この関数を実行したプロセスのプロセス番号を取り出す。
parent[p]：プロセス p の親プロセスのプロセス番号を取り出す。
firstson[p]：プロセス p の第一子プロセス* のプロセス番号を取り出す。
　　（*. プロセスが最初に実行した starteval 関数の第1番目の引数（fpl1）に対応する。）
brother[p]：プロセス p の弟プロセス（fpli に対しては fpli+1）のプロセ

ス番号を取り出す。

sonlist[p]: プロセスpの全ての子プロセスのプロセス番号から成るリストを作る。

procnum[name]: プロセス名=nameであるプロセスのプロセス番号を取り出す。

procname[num]: プロセス番号=numであるプロセスのプロセス名を取り出す。

c. プロセス値を取り出す関数

procval[p]: プロセスpのプロセス値を取り出す。

sonval[p]: プロセスpの全子プロセスのプロセス値のリストを作る。

d. プロセス間での直接的な通信(Mailing)のための関数

mail[mes; p]: プロセスpにメッセージcons[self[]; mes]を送る。
(*mailbox*(p) ← append[*mailbox*(p); cons[self[]; mes]])

recmail[]: この関数を実行するプロセスのMailboxが非空ならばT, そうでなければNil。

getmail[]: この関数を実行するプロセスのMailboxの内容を取り出す。実行後Mailboxの内容はNilになる。

e. プロセスを終了させる関数

terminate[f]: この関数を実行したプロセスは終了する。そのプロセス値はfの値になる。

f. pcbを解放する関数

一度割り付けられたpcbは, 主プロセスが終了するまで解放されない。ここで示す2関数は, pcbを解放し再利用可能にするためのものである。

free_pcb[p]: プロセスpのためのpcbを解放する。プロセスpは既に終了していなければならない。

term_free[f]: fを評価した後, プロセスは終了し, そのpcbは解放される。

◎ プロセス間での同期, 通信

プロセス間における同期や通信は, cr, ccr関数と適切な述語を組み合せることで実現する。同期, 通信のための媒体には下記のように共有変数, P-list, Mailboxを用いることができる。

1. 共有変数: 前述のように, 親プロセスの環境は, 親子プロセス間で共有されている部分が有り, そこに束縛された変数を親子あるいは兄弟プロセス間での共有変数として同期, 通信に用いることができる。
2. P-list: アトムの性質を利用して、任意のプロセス間での同期, 通信に用いることができる。
3. Mailbox: pcb内に設けられたMailboxは、上記のMailing機能を用いることで、相手を指定したメッセージの伝達に利用される。

LISPにおいて, 変数の値はセルを要素とするリストであり, このリストは複数の変数の値として共有される。従って, リストを書きかえるrplaca, rplacd等の関数の使用に関しては, 通常のLISPの場合以上に注意を払わなければならない。また, プロセスの生成時の引数の受渡し規則はevalの場合と同じである。(詳しくは付録) 従って, 親子プロセス間で引数値のリストを共有することがあり、ユーザは必要に応じて実引数値をコピイしてから子プロセス側に渡すという操作

をしなければならない。

4. Concurrent LispインタプリタConcurrent LispインタプリタはConcurrent Lisp

Concurrent Lispインタプリタは，複数のプロセスの動作をinterleaveさせることで並行動作を実現している。本章では，インタプリタの内部構造，スケジューリング等について述べる。

◎ 内部構造

インタプリタは，図2に示すように，プロセススケジューリングを主な仕事とするスケジューラ部，LISP1.5インタプリタとほぼ等価でありスケジューラ部の管理下で関数を実行する解釈実行部から成る。また主なデータ領域としては制御スタック領域，一時変数領域，pcb領域，セル領域，オブリスト領域がある。

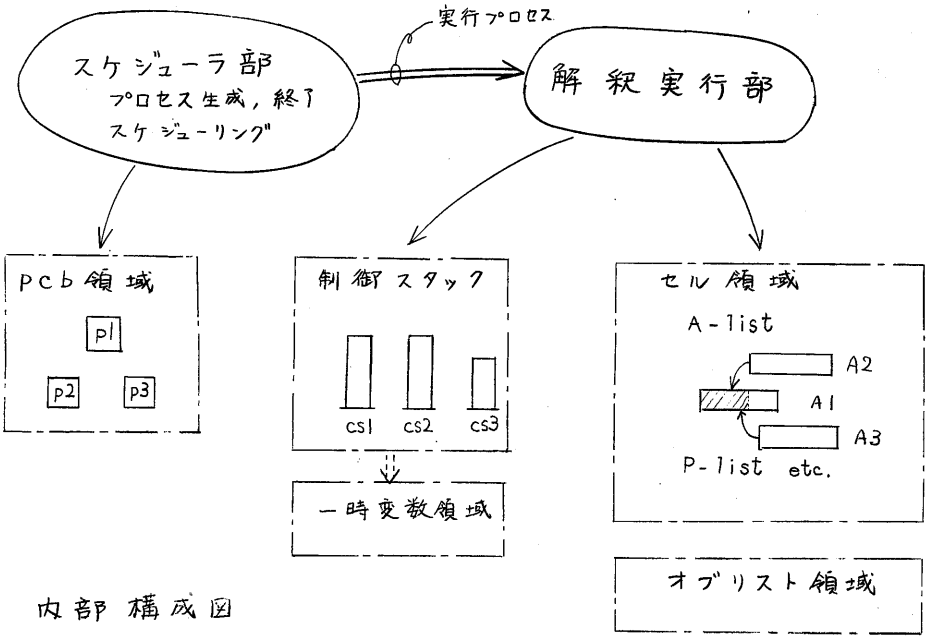


図2 内部構成図

次に，インタプリタの主構成要素を概説する。

スケジューラ部：プロセスの生成時，終了時におけるpcb領域の管理，プロセスの状態遷移（実行可，待ち合せ，終了）の管理，実行プロセスの選択，セル領域の占有権の管理を行う。現バージョンではラウンドロビン方式によるスケジューリングを行っている。また，プロセスに対して制御が与えられる時間は，制御を得てから待ち合せ状態になるまで，または一定回数の関数呼出しがなされるまでの間である。

解釈実行部：スケジューラ部によって決定された実行プロセスを動作させる。スケジューラの管理下で動作するLISP1.5インタプリタと見なしても良い。

A-list：環境を保持するために，セル領域上に作られたA-listを使用する。環境が共有されている場合、共有部分はすべてのプロセスから参照されなくなるまでGarbageとは見なされない。

P-list：P-listはセル領域上に作られる。

制御スタック：プロセス毎に独立した制御スタックが作られる。図3にスタックフレームの構成を示す。一時変数領域にはインタプリタが実行する関数の引数や一時変数の値を格納する。

pcb：プロセススケジューリングに必要なデータ，pcb間のリンク，プロセスの状態，待ち条件等を格納する。図4にその構成を示す。

オブリスト：文字アトムの印字名とP_listへのポインタを保持する。

◎ 実行

Concurrent LiSPインタプリタはトップレベルダブレットを入力すると主プロセスを生成し，解釈実行を開始する。主プロセスの実行が終了すると，主プロセスをはじめ各プロセスのプロセス値を出力する。（図5） 解釈実行中にデッドロックが生じた場合は、主プロセスの実行を終了する。

◎ スケジューリングに関する補足

スケジューラ部は待ち合せ状態のプロセスの待ち合せ条件をテストし，実行可状態にすべきかどうかを判定する。一方，ユーザは待ち合せ条件（ccr関数のform1）として任意の述語を書くことが許されている。しかしながら，この述語が副次的効果を持つ場合，スケジューラ部による述語の評価がプログラムの他の部分に影響を及ぼすため，重大な誤りを生じる危険がある。待ち合せ条件に対し強い制限を加えるか、あるいは，スケジューラ部が待ち合せ条件をテストする際には環境を凍結し、テスト終了後元に戻すといった解決策も考えられるが，それぞれ柔軟性やオーバーヘッド等の問題点がある。現バージョンではリストの付けかえを行う関数の実行に対し警告を発する。

continuation point
return value
environment
pointer to temp. var. area
number of temp. var's

図3 スタックフレーム

process name
pointer to the control stack
pointer to the parent process
pointer to the first son process
pointer to the last son process
pointer to the brother process
scheduling link (forward)
scheduling link (backward)
status (run, wait, and stop)
continuation point
environment
initial environment
process value
mailbox
wait condition
function names to be traced
trace flag (start-stop)
frame number for cr and ccr fun.

図4 pcb

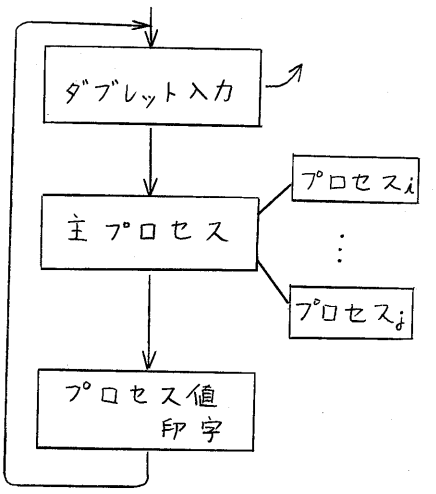


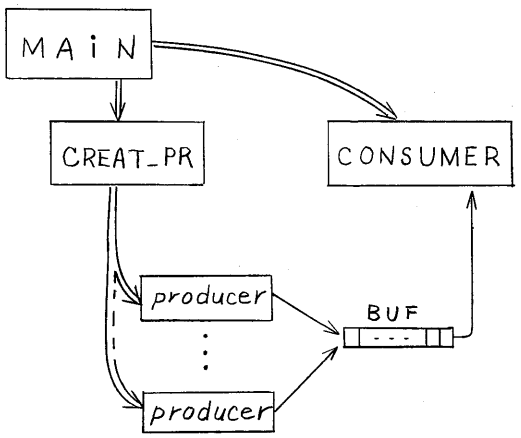
図5 インタプリタの動作

5. 例題

producer-consumer問題を題材としてConcurrent LiSPのプログラム例と実行結果を示す。次ページに示す例では，入力S式数に応じた数のproducerプロセスと1つのconsumerプロセスが存在し，これらは，長さ10のバッファ（BUF）を介して通信する。なお，主プロセスは，producerプロセスの統轄プロセスであるCREAT_PRとCONSUMERプロセスを生成する。CREAT_PR

プロセスは，主プロセスから送られてくるリストの数だけのproducerプロセスを生成する。以下にプログラム，実行結果，およびプロセスの構成図を示す。

```
INPUT =     DEFINE<(
INPUT =         <MASTER (LAMBDA () (PROG (D BUF)
INPUT =           (STARTEVAL ('CREAT_PR (PR_MASTER))('CONSUMER (CONSUMER)))
INPUT =         LOOP
INPUT =           (CR  (MAIL (SETQ D (READ_FILE 1))  'CREAT_PR))
INPUT =           (COND ((NOT (EQUAL D 'END)) (GO  LOOP)))
INPUT =           (CCR  (AND (TERMP 'CREAT_PR) (NULL BUF)) (SETQ BUF '(END)))
INPUT =           (CCR  (TERMP  'CONSUMER)(RETURN 'OK))  >
INPUT =         <PR_MASTER (LAMBDA () (PROG (M)
INPUT =         LOOP
INPUT =           (CCR  (RECMAIL) (SETQ  M  (GETMAIL)))
INPUT =           (WHILE  (NOT (OR (NULL M)(EQUAL (CDAR M) 'END)))
INPUT =                 <PROG () (STARTEVAL ((MAKEPRNAME)(PRODUCER (CDAR M))))
INPUT =                          (SETQ M (CDR M))  >)
INPUT =           (COND ((NULL M) (GO LOOP)))
INPUT =           (CCR  (ASONTERM (SELF)) (RETURN  'OK))    >
INPUT =         <PRODUCER (LAMBDA (X) (PROG (Y)
INPUT =           (SETQ Y *X)                          (注  *X は (COPY  X)に等しい)
INPUT =         LOOP
INPUT =           (COND ((ATOM X)(RETURN   Y)))
INPUT =           (CCR  (LESSP (LENGTH BUF) 10)
INPUT =                 <SETQ BUF (APPEND BUF (CONS (CONS (PROCNAME (SELF))
INPUT =                                                   (CAR X)) NIL) >)
INPUT =           (SETQ X (CDR X))
INPUT =           (GO  LOOP)   >
INPUT =        <CONSUMER   (LAMBDA () (PROG (L V)
INPUT =        LOOP
INPUT =          (CCR   BUF   <PROG ()
INPUT =             (SETQ   V   (CAR   BUF))
INPUT =             (SETQ   BUF   (CDR   BUF))
INPUT =             (CRLF 1)(PRINT  (LIST 'CONSUMER V))
INPUT =             >)
INPUT =           (COND  ((EQUAL V 'END) (RETURN  L)))
INPUT =           (SETQ  L  (APPEND  L  (CONS  (CDR V) NIL)))
INPUT =           (GO   LOOP)  >
INPUT =         >
PROCESS NAME =  MAIN
 ( MASTER PR_MASTER PRODUCER CONSUMER )
INPUT = MASTER()
INPUT =     (A B C)
INPUT =     (X Y Z)
INPUT =     (1 2 3)
INPUT =     END
 ( CONSUMER ( PR#00000 . A ) )
 ( CONSUMER ( PR#00000 . B ) )
 ( CONSUMER ( PR#00000 . C ) )
 ( CONSUMER ( PR#00001 . X ) )
 ( CONSUMER ( PR#00001 . Y ) )
 ( CONSUMER ( PR#00001 . Z ) )
 ( CONSUMER ( PR#00002 . 1 ) )
 ( CONSUMER ( PR#00002 . 2 ) )
 ( CONSUMER ( PR#00002 . 3 ) )
 ( CONSUMER END )
PROCESS NAME =  MAIN
 OK
 PROCESS NAME =  CREAT_PR
 OK
  PROCESS NAME =  PR#00000
 ( A B C )
  PROCESS NAME =  PR#00001
 ( X Y Z )
  PROCESS NAME =  PR#00002
 ( 1 2 3 )
 PROCESS NAME =  CONSUMER
 ( A B C X Y Z 1 2 3 )
:
```



producer は入力S式の数だけ生成され，それぞれ PR#xxxxx なる名前を与えられる。

6. おわりに

Concurrent LiSPは、複数の探索プロセスが独立して，あるいは共同して問題を解決していくような応用に対して開発された。現在までに開発されてきたLiSP系の言語では、コルーチン等の機能を持つものはいくつかあるが，並行処理を目標にしたものはあまりない。（[6]は，主目標ではないが，並行処理を取扱っている。）探索プロセスの記述において，コルーチン間での制御の移行はプログラムの構造を複雑にするため、各プロセス毎に独立した制御構造を持つ並行プロセスの方が望ましいと考える。

我々は，Concurrent LiSPを開発するに当り次の2点に留意した。

1. LiSP 1.5の持つ記法を損わない。
2. 並行処理を表現するための機構はできるだけ簡単な構造を持ち，しかもできるだけ少数にとどめる。

これによってConcurrent LiSPは，LiSPのユーザにとって理解しやすく，また，その簡潔な並行処理機構は，必要に応じていろいろな応用に適用できるものと考える。

一方，Hewittによるactor理論にもとづいた並行処理モデルの記述や，シミュレーション言語を用いて表わすような，同時に動作する複数のオブジェクトからなるモデルの記述といったものに対しても，Concurrent LiSPは十分な能力を備えていると考える。

現在，Concurrent LiSP インタプリタは，京都大学大型計算機センターFACOM M-200上にPL/Iを用いて実現されている。現在のバージョンは，前述の並行処理関数，プロセスデータ処理関数を含めて約140の関数を持ち，セル領域は10Kセルである。またインタプリタのプログラムサイズは約4500行である。今後，現バージョンをもとにして，デバッグ機能付加，処理の高速化等、改良を進めていく予定である

参考文献

1. BOBROW D. G. : A Model and Stack Implementation of Multiple Environment, CACM Vol. 16, No. 10, Oct. 1973
2. Hewitt C. : PLANNER : A Language for Proving Theorems and Robots, A.I. Memo No. 137, M.I.T, May 1969
3. McCARTHY J. et al : LiSP 1.5 Programmer's Manual, The MIT Press, 1962
4. 杉本，田畑，大野 : Concurrent LiSP とそのインタプリタ，情報処理学会第21回全国大会予稿集，1980
5. SUSSMAN G. J. and McDERMOT D. V. : The CONNiVER Reference Manual, A.I. Memo No. 259a, M.I.T., May 1972
6. SUSSMAN G. J. and STEEL G. L. : The Revised Report on SCHEME, A.I. Memo No. 452, M.I.T., Jan 1978
7. 田畑，杉本，真崎，伊藤，大野 : Concurrent LiSP，情報処理学会第20回全国大会予稿集，1979

8 TEITELMAN W. : INTERLISP Reference Manual, XEROX Palo Alto Research Center, Nov 1975
9 米澤 明憲 : ACTOR理論について, 情報処理, Vol. 20, No. 7, July 1979

付 録 starteval 関数の定義

```
starteval [ fpl ] ≡c
  prog [[form;dummy;proc_name];
    [null[fpl] → return[]];
     proc_name := eval[caar[fpl];a] ;
    [ atom[cadar[fpl]] → <C:proc_name;eval[cadar[fpl];a*]> ;
      atom[caadar[fpl]] →
          [get[caadar[fpl];EXPR]→
             {dummy := evlis[cdadar[fpl];a];
              <C:proc_name; apply[expr;dummy;a*] >;
           get[caadar[fpl];FEXPR]→
             {dummy := list[cdadar[fpl];a*];
              <C:proc_name; apply[fexpr;dummy;a*] >;
           get[caadar[fpl;SUBR]→
             { dummy := evlis[cdadar[fpl];a];
               <C:proc_name; call subr(fun*,dummy,a*)>;
           get[caadar[fpl];FSUBR]→
             { dummy := cdadar[fpl];
               <C:proc_name; call fsubr(fun*,dummy,a*)>;
           get[caadar[fpl];CSUBR]→
             { dummy := cdadar[fpl];
               <C:proc_name; call csubr(fun*,dummy,a*)>;
           t   →
             { dummy := evlis[cdadar[fpl];a];
               form := cons[cdr[sassoc[caadar[fpl];a;u]]; dummy];
               <C:proc_name; eval[form;a*]>;];
      t    →{ dummy := evlis[cdadar[fpl];a];
              <C: apply [caadar[fpl];dummy;a*]>];
      return[cons[proc_name;starteval[cdr[fpl]]]];].
```

- ≡c は CSUBR の定義を示すものとする。
- <C : proc_name ; function> は function がプロセス名 proc_name である新しく生成されたプロセスによって評価されることを示す。
- a^* は新しく生成されたプロセスに与えられる初期環境である。
- fun^* は関数名を示す。
- u はエラー処理用関数であるとする。