並列処理システムのための言語について†

宮村 勲†† 榎本 肇†††

本論文は，プログラムの生産，理解，修正を容易にするような新しい並列処理言語について述べている．この目的を達成するため，プログラムに人工的な制約を課さず，プログラムの構造が問題の論理構造を静的に反映できなければならない．プログラム・テキスト中に問題の階層構造を表現するため，process, resource, procedure, function という4種類のモジュールを用意し，それらの任意の入れ子を可能にした．共有変数に起因するデッドロック問題を避けるため，変数のスコープはそれが宣言されたモジュールだけに制限する．モジュール間の相互作用はパラメータ転送，メッセージ交換，resource を用いて行われる．process の動作は，通常動作と例外的事象に対する動作に分けて記述される．例外的事象とは他モジュールから緊急メッセージを受信することである．そのようなメッセージは常時受信可能である．また，メッセージ交換の相手は静的と動的の両方で制限される．resource は共有 object に対するアクセスを制御する．resource procedure 間の並列動作の可能性は，個々の手続きとは分離して静的に記述する．個々の手続きが呼び出されるときの前提条件も分離して静的に記述される．非決定的表現として，if 文，do 文，while 文の三つを用意する．これらは Dijkstra の提案した guarded command に基づくが，guard に入出力文のどちらでも書けるように拡張されている．本言語の有効性を明らかにするため，フィボナッチ数列を計算する procedure と，Readers/Writers 問題のプログラム例を示している．

1. まえがき

計算機で扱う問題が大規模になり，高速化のために並列処理が必要である．並列処理言語にはコンカレント・パスカル[1]，Modula[2]，CSP[3]，分散プロセス[4]，Ada[5] らが提案されているが，これらは並列処理に特有な同期や排他制御を簡潔に表現する手段をもたず，それが原因でデッドロックその他のエラーを起こす．また，大規模ソフトウェアの開発・保守は困難であり，これを容易にするには，簡潔表現のできる並列処理言語が必要である．われわれはそのような言語を開発[10),11)]したので，その概略を説明する．

大規模ソフトウェアは階層的に設計する．そこで，階層構造をプログラム・テキスト中に保存し，プログラムの理解・保守を容易にする．プロセスをはじめ，任意モジュール間の入れ子を許す．階層構造をモジュールの入れ子で表現する．モジュール間の共有変数を禁止し，相互作用をテキスト中に陽に表現する．可能な限り静的記述を増やすため，個々の手続き実行の前提条件を condition 文として，手続き間の並列実行の可能性は並列定義として，動作記述と分離して記述する．

2. 設計方針

言語は概念を表現する手段を提供するだけでなく，人間の思考方法までも制約する．問題の論理構造をそのまま表現するだけでよく，手続き化するために余分な制約を持ち込むべきでない．ハードウェアの高速化により，プログラムの実行効率はあまり問題にならない．その反面，ソフトウェア開発の大半はデバッグや修正に費される．そこでプログラム言語としては，次のような特徴をもつべきである．

1) 簡潔に表現でき，エラーのないプログラムの作成が容易である．
2) プログラムがドキュメントの働きをし，理解・修正が容易である．

言語仕様は純粋に論理的必要性から決定し，ハードウェア上の制約は持ち込まない．決定された言語仕様を実際の計算機上でどう実現するかは別の問題であり，ソフトウェア上の要求に基づいて，ハードウェア・アーキテクチャを決定すべきである．

問題の論理構造をそのまま表現できるだけの機能は必要であるが，あまり多くの機能をもつと，言語が複雑になり，理解しにくくなる．また，エラーの原因になりやすい機能は提供しない．これらの目的を達成するため，次のような方針を立てる．

2.1 階層的記述

大規模な問題は階層的に分割して，プログラム化する．書かれたプログラムを理解・修正するには，プロ

† On a Language for Parallel Processing Systems by ISAO MIYAMURA (Department of Information Engineering, Faculty of Engineering, Niigata University) and HAJIME ENOMOTO (Department of Computer Science, Faculty of Engineering, Tokyo Institute of Technology).

†† 新潟大学工学部情報工学科
††† 東京工業大学工学部情報工学科

グラム化の過程を再構築しなければならない．理解や修正の容易なプログラムとは，問題の階層構造がテキスト中に静的に表現されているものである．

従来の並列処理言語では，変数のスコープにアルゴル流の規則を採用している．そのため，あるモジュールの内部で子プロセスが宣言された場合，変数のアクセスについて同期や排他制御の問題が無制限に生じ，これがエラーの原因になっている．これを避けるため，プロセスを他のモジュールの内部で宣言することが禁止されていた．

本言語ではアルゴル流のスコープを採用せず，あるモジュールで宣言された変数のスコープは，そのモジュールだけに限定される．内部モジュールであってもその変数を自由にアクセスすることはできない．これにより，内部にプロセス宣言を含んでいても何らの問題も生じない．そのため，プロセスをはじめとする任意のモジュール間の入れ子を許し，問題の階層性が，モジュール間の親子関係に反映できるようにする．

2.2 並列性の指定

並列処理言語であっても，すべての並列性を指定する必要はない．不必要に詳細な指定はプログラムを煩雑にし，理解を困難にする．文レベルの並列性はコンパイラ等による自動検出がある程度可能であり，その利点も実行効率の向上だけに限られる．プログラムの簡潔化，エラー防止の観点から，文レベルの並列性は特別な場合を除き，指定しない．

並列処理言語の利点は処理効率の向上だけでない．複数のモジュール間の相互作用として全体を記述できることも大きな利点である．順次処理言語では，制御の流れを中心にして，モジュールに分割する．そのため，プログラムが問題の論理構造を反映せず，理解は困難である．一方，並列に動作するモジュール間の相互作用として全体の作業が記述できると，個々の論理機能に対応してモジュールが作れるため，プログラムの開発・理解・修正が容易になる．そこで，モジュール間の並列性記述に重点を置く．

2.3 プロセス間の相互作用

プロセス間の相互作用を記述する方法が適当でないと，デッドロックその他のエラーの原因になる．相互作用が暗黙のうちに行われるのは望ましくなく，テキスト中に陽に表現されるのがよい．各モジュールの独立性を最大限に保障するため，相互作用の影響を必要最小限に抑える手段が必要である．

相互作用の実現は4種類の方法が考えられる．a）変数を直接共有する．b）Monitor[6] のような媒介物を介して間接的に影響を与える．c）相手を指定してデータを送る．d）相手プロセス内の手続きにアクセスする．

共有変数は実行効率の点で優れるが，相互作用が陰に隠れてしまい，アクセスの同期や排他制御が問題となる．より高級な機能が必要である．相手プロセス内の手続きにアクセスする方法は，アクセスされる側に選択権がない．起動の後，プロセスは独立した存在として，自分の動作に完全な決定権をもつべきである．

Monitor等を使用する方法は長期的共有に適しているが，一時的なデータの授受に適さない．逆に，データを送る方法は一時的なデータの授受に適しているが，長期的な共有に適さない．これらはパラメータ転送における名前どりと値どりの関係と同じである．論理的に異なった概念であるので，別々の手段を用意する．一時的共有に対してメッセージ交換，長期的共有に対して，Monitor の概念を拡張した resource を提供する．

2.4 非決定性

非決定性とは，複数の命令系列が実行可能なことであり，2通りの解釈が存在する．一つはオートマトン理論等における解釈であり，実行可能な命令系列のなかに一つでも解を導くものが存在すればよい．他方の解釈ではすべての命令系列が解を導かなければならない．

第一の解釈は人工知能用言語で使用される．解を得るまでにいくつかの命令系列を実行する必要があり，バックトラック等のために能率が非常に悪い．第二の解釈では適当に一つの命令系列を選択すればよく，実現も容易なので，本言語はこの解釈を採用する．

非決定的表現は計算効率が悪い．しかし，扱う問題の多くは非決定的性質をもち，プログラム化の過程でその性質を失うと，プログラムの理解や修正が困難になる．また，非決定的表現は簡潔になるので，非決定性を表現できるようにする．

2.5 制約条件の静的記述

プログラムは計算機の動作を指定するが，指定には動的記述と静的記述がある．動的記述は，幾通りも存在する解法のなかから効率のよい方法を選択し，それを手続き化しなければならないために，プログラムの開発がむずかしい．また，書かれたプログラムから，それが何を行うプログラムであるかを理解しにくい．その反面，最も効率のよい方法で実現でき，プログラ

ムの実行効率はよい．静的記述は解の満たすべき条件を指定するだけでよく，手続きへの変換はコンパイラが自動的に行う．解の条件式をそのまま記述するだけでよいので，記述や理解は容易であるが，手続きへの変換がむずかしい．高級言語としては可能な限りプログラマの労力を削減すべきなので，静的記述を採り入れる．

本言語では排他制御の記述を，個々の手続きの動作記述と分離して，静的に記述する．また，手続きには実行のための前提条件が存在する場合がある．たとえば，スタックが空だとポップは実行できない．空でなくなるまで待たねばならない．これらの条件なども，動作記述と分離して記述する．

3. 言語仕様

モジュールには process, resource, procedure, function の4種類がある．問題の階層構造を，モジュール宣言の包含関係として，テキスト中に静的に表現し，各種オブジェクトのスコープを静的に制限する．

プログラムは process, procedure, function, type の宣言をいくつかまとめたものである．

〈プログラム〉::=〈プログラム要素〉
　　|〈プログラム要素〉〈プログラム〉
〈プログラム要素〉::=〈process 宣言〉
　　|〈procedure 宣言〉|〈function 宣言〉|〈type 宣言〉

変数が複数モジュールに共有されると，同期や排他制御の問題が生じる．これを防ぐため，パラメータ転送された場合を除き，共有させない．また，メッセージ交換の相手を制限するため，process 名をパラメータ転送された場合を除き，親・子・兄弟モジュール以外とはメッセージ交換できない．

変数と process 名以外の，型名，定数名，procedure 名，function 名などは，複数のモジュールに共有されても問題は生じないので，これらのスコープはアルゴル流の規則に従う．とくに，プログラムを構成する4種類の宣言のうち，process 宣言以外は任意のところで参照できる．

3.1 process

process は並列実行されるモジュールであり，宣言部と実行部で構成される．process は識別のために名前をもつが，同じような処理をする複数の process を起動し，それらを区別するために，名前に添字をつけてもよい．添字の値は離散的であり，その変域を宣言する．

〈名前〉{[〈名前〉:〈変域〉,〈名前〉:〈変域〉,…]}*

```
〈プロセス名定義〉=PROCESS{(〈仮引数の並び〉)};
        〈定数定義〉
        〈型定義〉
        〈プロセス宣言〉
        〈procedure 宣言〉
        〈function 宣言〉
        〈変数宣言〉
        〈例外定義〉
        〈文列〉
END 〈名前〉;
```

図1 プロセス宣言
Fig. 1 Process declaration.

図1に示すように，process 内では任意の順に process, resource, procedure, function 等を宣言できる．あるモジュール内で宣言されたモジュールは子モジュールと呼ばれ，同じモジュール内で宣言されたモジュールは兄弟と呼ばれる．このように，モジュールの親子・兄弟関係は，宣言の包含によって静的に決まる．

process は init 文により起動されるが，いちばん外側で宣言された process だけは例外で，プログラムが実行開始した時点で自動的に起動される．ある process に対する init 文の実行は，その親または兄弟モジュールに限られる．process は起動されると，init 文を実行したモジュールとは独立して，並列に処理される．しかし，起動時に値どりのパラメータを受け取れる．

INIT 〈プロセス名〉{(〈実パラメータの並び〉)}

terminate 文を実行するか，最後の end 文を実行すると，process は正常終了する．正常終了する前に親モジュールが終了すると，その process は異常終了する．

process の動作は例外定義部と実行部に分離して記述される．実行部では通常動作を記述し，例外的事象が生じた場合は例外定義部で記述する．

共有変数が存在しないため，process 間の相互作用は resource を介するか，メッセージ交換による．メッセージを交換するには，相手の process 名を指定しなければならない．そのため，同じ名前の process を複数個起動することはできない．メッセージ交換の相手やアクセス可能な resource は，モジュール宣言の入れ子関係による静的な制限と，実行時の動的な制限を受ける．

* {A}はAが省略可能であることを示す．

3.2 メッセージ交換

メッセージ交換は二つのモジュール間でデータを交換する機能であるが，任意の相手とメッセージ交換できるわけでなく，その組合せは次の3種類に限られる．

a）親モジュールと子 process 間

b）兄弟の process 間

c）上記以外で，双方とも相手の process 名をパラメータ等で受け取り，互いの存在を知っている process 間

メッセージ交換用に input 文と output 文を用意する．input 文は相手からデータを受け取り，対応する変数に代入する．output 文は式の値を相手に送る．実際には output 文と input 文との対応をとらねばならないため，output 文で指定されたラベルをデータといっしょに送り，input 文で指定されたラベルと比較して，対応を判定する．ただし，ラベルを省略することも可能である．

INPUT {〈ラベル〉:}
〈変数の並び〉 FROM 〈送信者〉
OUTPUT {〈ラベル〉:}
〈式の並び〉 TO 〈受信者〉

メッセージ交換の相手を指定するには，その process 名を書くか，"PARENT" と書く．"PARENT" は相手が自分の親モジュールであることを示す．親モジュールは必ずしも process とは限らない．また，process でも添字をもっていれば，親の添字の値を知らない場合が多いからである．

相手の process 名を指定する場合，相手に添字があればその値も指定する．しかし，メッセージ交換の相手が唯一に決定せず，いくつかの process のうちのどれでもよい場合もある．それを表現するため，とくに値を指定しない添字に don't care を表す記号"?"を付けた識別名を書く．相手 process としては，"?" の付けられた添字の値は何でもよい．メッセージが交換されると，"?" の付けられた識別名には，相手 process の対応する値が代入され，後で参照することができる．

メッセージ交換の際，自動バッファリングはしない．input 文と output 文がマッチングしてから，実際の転送が行われる．先に出された入出力文は，マッチングがとれるまで待たされる．input 文と output 文は次の二つの条件が成立すると，マッチングする．
1) それぞれを実行しようとしているモジュールを，メッセージ交換の相手として指定している．2) 両方のラベルが一致するか，両方のラベルとも省略されている．

```
EXCEPTION
    {P=〈整数〉:} 〈入力文〉→〈文列〉,
    {P=〈整数〉:} 〈入力文〉→〈文列〉,
                  ⋮
    {P=〈整数〉:} 〈入力文〉→〈文列〉
END;
```

図 2 例外定義
Fig. 2 Exception definition.

自動バッファリングはせず，メッセージが実際に交換可能かどうかに応じて，その後の動作を変更できるようにする．これは CSP の考えと同じであり，融通性に富み，デッドロックの危険を減少できる．

3.3 例外処理

非決定的動作も含め，各モジュールの動作はテキストで完全に指定されなくてはならない．自分のモジュール内で，何がいつ起こりうるかは事前にわかり，対処するのは容易である．一方，他モジュールに関連することになると，何がいつ起こるかは予測できない．そこで，他モジュール内の出来事を緊急に知りたいときの記述方法が問題となる．これを実現するには，特定のメッセージを常に受け取れるような記述方法が必要である．

process は input 文を用いて他モジュールからメッセージを受け取る．従来の言語は，特定のメッセージが常に受信可能であるように記述できない．並列処理言語はそのような機能をもつべきなので，本言語では緊急時の処理を通常の処理と分離し，図2に示すような例外定義として記述する．

例外定義はいくつかの例外処理記述から成っている．個々の例外処理記述は，その優先度を表す整数値と，緊急メッセージを受け取るための入力文，そのメッセージを受け取ったときの動作を指定する文列の三つのもので構成される．指定された整数値が小さいほど，優先され，整数値が省略されると，いちばん低い優先度が与えられる．

通常の処理をしているときに緊急メッセージが到着すると，通常の処理は中断され，メッセージに対応した処理がとられる．例外処理中に，それより高い優先度のメッセージが到着すると，処理は中断され，新たに到着したメッセージに対応した処理が始まる．

その処理が終了すると，前に中断したところに戻る．例外処理中に，それと同じかそれより低い優先度のメッセージが到着しても，高い優先度の例外処理が

```
TYPE 〈リソース名〉=RESOURCE {(〈仮引数の並び〉)};
                〈定数定義〉
                〈型定義〉
                〈プロセス宣言〉
                〈変数宣言〉
                〈resource procedure 宣言〉
                〈並列定義〉
                〈文列〉
END 〈リソース名〉;
```

図 3 resource の宣言

Fig. 3 Resource declaration.

```
{GLOBAL} PROCEDURE 〈手続き名〉 {(〈仮引数の並び〉)};
        〈定数定義〉
        〈型定義〉
        〈プロセス宣言〉
        〈procedure 宣言〉
        〈function 宣言〉
        〈変数宣言〉
        {CONDITION : 〈論理式〉; }
        〈文列〉
END 〈手続き名〉;
{GLOBAL} FUNCTION 〈関数名〉{(〈仮引数の並び〉)} : 〈型名〉
                ⋮
END 〈関数名〉;
```

図 4 resource procedure 宣言

Fig. 4 Resource procedure declaration.

```
PARALLEL
    〈論理式〉→〈手続き名〉,〈手続き名〉,…,〈手続き名〉;
                ⋮
    〈論理式〉→〈手続き名〉,〈手続き名〉,…,〈手続き名〉
END;
```

図 5 並列定義

Fig. 5 Parallel definition.

続行され，それが終了してから低い優先度の処理に移る．

緊急メッセージを受け取る側では，それを普通のメッセージと区別して受け取るため，例外処理の左辺で指定される入力文と同じ出力文にマッチングする入力文が他に存在してはならない．緊急メッセージの送信側では，緊急メッセージと通常メッセージを区別することはできず，すべて同一レベルのメッセージとして送る．緊急かどうかは受信側の判断に任せられる．

3.4 resource

resource は Monitor の概念を拡張したものであり，**図 3** のようにタイプとして定義される．resource 内では，共有 object を実現するためのグローバル変数，それに初期値を与えるための初期化レコード，その他の宣言等を記述する．resource 内で宣言された procedure と function は，resource 以外で宣言されたものと違い，resource 内のグローバル変数にアクセス可能である．そこで，これらをとくに resource procedure と呼んで区別する．resource procedure には，resource 内だけで呼び出し可能なものと，グローバル属性が与えられて，外部からも呼び出し可能なものがある．共有 object に外部からアクセスするには，グローバル属性をもつ resource procedure を呼び出す．

resource procedure はグローバル変数にアクセスするため，通常の procedure や function と違い，常に実行可能とは限らない．実行の前提条件が存在する場合もある．たとえば，スタックに対するポップ操作はスタックが空であると実行できず，空でなくなるまで待たされる．このような記述を実行部に埋め込むと，プログラムが煩雑になるので，**図 4** のように condition 文として静的に記述する．実行時には，condition 文で指定された論理式が真になるまで，手続きの実行は待たされる．

resource は複数モジュールからアクセスされるので，複数の resource procedure が同時に呼び出されることがある．それらはグローバル変数にアクセスするので，排他制御の問題が生じる．Monitor では，すべての手続きの実行が排他的に行われる．一方，セマフォ[7] やパス式[8] を用いて並列性を指定する方法もあるが，プログラムを煩雑にする．本言語では並列実行可能な resource procedure の組合せを個々の手続きとは分離して，**図 5** のように並列定義として与える．

resource procedure はその組合せにより，常に並列実行可能な場合，特定の条件が成り立つときのみ並列実行可能な場合，常に並列実行不可能な場合に分かれる．従来は，とくに指定しない限り並列実行可能であり，排他制御する場合はそれを記述していた．このようにすると，指定を忘れたりしたときに内容不一致などのエラーが生じる．そこで，本言語では原則として排他制御を行い，並列実行可能な場合を記述するようにした．こうすれば，記述を忘れても効率が悪くなるだけで，エラーを生じない．

並列定義の右辺に，並列実行可能な resource procedure の組合せを記述し，左辺にそのための条件を記述する．すなわち，左辺に書かれた条件が成立していると，右辺で指定された resource procedure は並列に実行可能である．常に並列実行可能な場合は左辺の条件が空になる．また，異なった resource procedure の組合せでなく，同一の resource procedure が複数個並列に実行可能であるかどうかの指定も必要である．並列実行可能な場合には，右辺に一つだけその名前を書く．

並列定義部は静的に記述された複数の並列定義から成る．これにより，resource procedure 間の関係が明確になり，動作を追跡しなくても条件の把握が可能である．アルゴリズムの変更による影響も容易に吸収できる．

resource は複数モジュールからアクセスされるので，resource にアクセス可能なモジュールの集合を制限するだけでなく，アクセスの方法までも制限する．そのため，resource 型の変数を特別に扱い，capability と呼ぶ．capability は，宣言された resource の instance へのポインタと，instance に対するアクセス権から成る．アクセス権とは，その capability を介してアクセス可能な resource procedure の名前と，その capability を他の capability にコピーできるかどうかの権利から成っている．

resource の instance は，その型の capability を引数としてもつ create 文で生成される．その際，値どりの引数を与えることもできる．instance が生成されると，グローバル変数の領域がとられ，初期化コードによって初期値が与えられる．次に，指定された capability に対して，その instance へのポインタと，その resource のすべてのグローバルな resource procedure に対するアクセス権と，コピーする権利が与えられる．

CREATE (〈capability〉[:〈実引数の並び〉])

capability のコピーは代入文の形式をとり，同じ型の capability 間で行われる．その際，右辺の capability がコピーの権利をもっている必要がある．コピーにより，左辺の capability は右辺と同じ instance を指し，同じアクセス権をもつようになる．ただし，同じ権利を与えるのでなく，もっている権利のうちのいくつかだけを与える場合，与えるアクセス権の名前を，右辺の capability の後に指定する．

〈capability〉
　:=〈capability〉[[〈アクセス権の並び〉]]

capability に対して NULL が代入されると，それはどの instance も指さなくなる．ある instance を指す capability の数が0になると，その instance は消滅する．

グローバル属性をもつ resource procedure を外部モジュールから呼び出す場合，アクセスすべき instance を指す capability を実引数の前に書く．その際，その capability が呼び出す resource procedure へのアクセス権をもっていなければならない．

CALL 〈手続き名〉(〈capability〉
　:〈実引数の並び〉)

3.5 非決定性のための primitive

非決定的表現として，if 文，do 文，while 文の3種類を用意する．これらは Dijkstra によって提案された guarded command[9] を拡張したものである．

guarded command は guard と文列の二つで構成される．guard は文列を実行するための十分条件を表し，これが真である guarded command だけが実行される．Dijkstra は0個以上の論理式をコンマで区切って並べたものを guard とした．CSP では論理式の最後に入力文が書けるように拡張された．しかし，出力文を書くことは禁止されているため，入出力文の対称性は失われている．対称性を保存し，記述力を高めるため，本言語では論理式の最後に入出力文を書けるようにする．

〈論理式の並び〉[,〈入出力文〉]→〈文列〉

guard の構成要素がすべて真のとき，guard は真となる．論理式の真偽はふつうに評価される．guard に含まれる入出力文は，それとマッチングする入出力文が実行可能であるとき，真とみなす．メッセージ交換の相手モジュールがすでに実行を終了して，消滅していると，そのモジュールとメッセージ交換することは不可能なので，偽とみなす．そのどちらでもないと，値が決まるまで未定となる．guard が真となり，対応する文列が実行される場合，guard に入出力文が含まれていれば，まず最初にその入出力文が実行される．

if 文は guarded command の並びを "if" と "fi" で囲ったものである．真となる guard を一つ見つけ，対応する guarded command を実行する．すべての guard が偽である場合，どの guarded command も実行されず，if 文はスキップされる．真である guard が存在せず，真偽が未定な guard が存在するときは，guard の真偽が決定し，動作が決まるまで待たされる．

IF 〈guarded command の並び〉 FI

do 文は guarded command の並びを "do", "od" で囲ったものである．指定された guarded command のなかで，guard が真であるものをすべて一度ずつ実行する．実行すべき command がなくなると，do 文は終了する．通常は逐次的に command を処理するが，"IN PARALLEL" が指定されると，複数の command を並列に実行してもよい．

DO [IN PARALLEL]
　〈guarded command の並び〉 OD

while 文は "while do", "od" で囲ったものである．真である guard が存在する間は何度でもその command を実行し，すべての guard が偽になるまで続ける．通常は一度に一つの command を逐次実行するが，"IN PARALLEL" が指定されると，複数の command を並列に実行してもよい．ただし，同じ command を複数個並列に実行することはできない．

WHILE DO {IN PARALLEL}
〈guarded command の並び〉 OD

guarded command の guard に入出力文が書けるようになり，動作選択の自由度が大きく向上した．すなわち，相手とのメッセージ交換が可能かどうかに応じて，自分の動作が変更できる．その反面，非決定性のインプリメントが複雑になり，効率も悪くなる．

do 文や while 文は繰返しを表す．これらの繰返し文から強制的に抜け出す手段として，exit 文を使う．

EXIT {〈正の整数〉}

抜け出すべきループの深さを示す整数値を，exit の後に指定する．ただし，1のときは省略してもよい．指定された整数値が，ループの深さの最大値を越えた場合には，いちばん外側のループから抜ける．

4. プログラム例

言語機能の有効性を明らかにするため，二つのプログラム例を示す．最初に，フィボナッチ数列のm番目の値を計算する関数を，二つの子 process を用いて記述したものを**図6**に示す．フィボナッチ数列とは，次のように帰納的に定義される．

$f(n+2)=f(n+1)+f(n)$ for $n\geqq 0$

$f(0)=0,\ f(1)=1$

新しい値を計算するのに，直前の二つの値を必要とする．そこで，二つの子 process を起動し，process 間のデータ交換により，交互に数列を計算する．片方の process が求める結果を計算すると，その process はその結果を親モジュールに知らせた後，二つの process は終了する．

次に，resource を用いて readers/writers 問題を記述する．複数の process がバッファを共有し，そこにデータを格納したり，そこからデータを読み出したりする．データの読出しは並列に実行できるが，格納は排他的である．バッファを管理する resource を作り，データを読み出す関数 read と，格納のための手続き write の二つの resource procedure を用意する．格納されるデータの型は何でもよいが，整数型とする．

```
FUNCTION fibonacci (m): integer;
    child [i: 1…2]=PROCESS (n);
        VAR index, val, temp, j, n: integer;
        index:=i-1;
        val:=index;
        IF i=1-> j:=2,
           i=2-> j:=1; OUTPUT val TO child [1] FI;
        WHILE DO
            index<n-2-> INPUT temp FROM child [j];
                        val:=val+temp;
                        index:=index+2;
                        OUTPUT val TO child [j],
            index=n-2-> INPUT temp FROM child [j];
                        val:=val+temp;
                        index:=index+2;
                        OUTPUT val TO PARENT
        OD
    END child;
    VAR val, j, m: integer;
    INIT child [1…2] (m);
    INPUT val FROM child [?j];
    RETURN val
END fibonacci;
```

図6 フィボナッチ数列を計算する手続き
Fig. 6 Procedure for fibonacci sequence.

```
TYPE buffer=RESOURCE;
    VAR content: integer;
    GLOBAL FUNCTION read: integer;
        RETURN content
    END read;
    GLOBAL PROCEDURE write (d);
        VAR d: integer;
        content:=d
    END write;
    PARALLEL
        ->read
    END;
    content:=0
END buffer;
```

図7 Readers/Writers 問題 (Readers が優先する場合)
Fig. 7 Readers/writers' problem—Readers' priority—

読出しを優先する場合を**図7**に示す．並列定義では，複数の read 要求が並列に処理されてもよいが，それ以外の組合せは並列処理できないことを示す．そのため，後から出された read 要求が，前に出された write 要求よりも先に処理されることがある．

次に，格納を優先する場合を**図8**に示す．格納要求が出されたときに，まだ実行されていない読出し要求が存在しても，格納要求を先に処理する．これを実現するため，格納の仕事は要求を登録する write と，実際に格納作業を行う wwrite に分ける．並列定義によれば，write は自分自身を含めてすべての手続きと並列に実行でき，read は wwrite 以外の手続きと並列実行できる．しかし，格納要求を優先させるため，condition 文を用いて，格納要求が到着しているときは，

```
TYPE buffer=RESOURCE;
    VAR content, nw: integer;
    GLOBAL FUNCTION read: integer;
        CONDITION nw=0;
        RETURN content
    END read;
    GLOBAL PROCEDURE write (d);
        VAR d: integer;
        nw:=nw+1;
        CALL wwrite (d);
        nw:=nw-1
    END write;
    PROCEDURE wwrite (d);
        VAR d: integer;
        content:=d
    END wwrite;
    PARALLEL
        ->read, write;
        ->write, wwrite;
        ->read;
        ->write
    END;
    content:=0;
    nw:=0
END buffer;
```

図 8 Readers/Writers 問題 (Writers が優先する場合)
Fig. 8 Readers/writers' problem—Writers' priority—

新たな read 要求が実行できないように指定してある.

5. むすび

筆者らが開発した並列処理言語の特徴を説明した. プログラムの開発・理解・修正のすべてを容易にするため, 次の3点を設計の基本方針とした. 1) プログラム化する際に, 余分な制約をアルゴリズムに持ち込まない. 2) 扱う問題の論理構造を, プログラム・テキストに反映させる. 3) 可能な限り静的な記述を増やし, 動的な記述を減らす.

これらの目的達成のため, モジュールごとに変数の壁を設け, モジュール間の任意の入れ子を許す. モジュール間の相互作用の手段として, resource とメッセージ交換の二つを提供し, 相互作用が自然な形で表現できるようにした. 手続きを実行するための前提条件や, 手続き間の並列実行の可能性などは, 動作記述とは分離して宣言の形で静的に記述する. 静的記述はそのまま実行することはできないので, 指定されたような動作ができるようなコードを, コンパイル時に発生する. 動作選択の自由度を大きくするため, guarded command の guard に出力文も書けるようにする.

従来の並列処理言語と違い, 以上の機能をもたせることにより, エラーの少ないプログラムの開発が容易になるだけでなく, 書かれたプログラムの理解・修正も容易になる. しかし, これで満足であるという状態ではなく, さらに新しい記述法の研究を進めていかなければならない.

最後に, 処理系の作成について述べる. われわれの目的は, 現存する計算機システム上で動作する並列処理言語を開発することでない. 現在, 直面しているソフトウェア危機を解決するために必要な言語機能を研究することである. ハードウェアとしては, そのような言語機能を実現できるアーキテクチャを開発すべきである. このような立場から, 処理系について考える. 現存する計算機システムに処理系を作成することは可能であるが, 効率は悪くなる. condition 文や並列定義は, それを実現するためのコードを, 各手続きの前後に挿入すればよい. また, 非決定性については, 各モジュールごとに非決定性を管理する部分を作り, そのなかで非決定性を解決するようにすればよい.

参考文献

1) Hansen, B.: The Programming Language Concurrent Pascal, *IEEE Trans. Softw. Eng.*, Vol. SE-1, No. 2, pp. 199-207 (1975).
2) Wirth, N.: Modula: A Language for Modular Multiprogramming, *Softw. Pract. Exper.*, Vol. 7, No. 1, pp. 3-35 (1977).
3) Hoare, C. A. R.: Communicating Sequential Process, *Comm. ACM*, Vol. 21, No. 8, pp. 666-677 (1978).
4) Hansen, B.: Distributed Processes: A Concurrent Programming Concept, *Comm. ACM*, Vol. 21, No. 11, pp. 934-941 (1978).
5) Preliminary Ada Reference Manual, Sigplan Notice, Vol. 14, No. 6, Part A (June 1979).
6) Hoare, C. A. R.: Monitor: An Operating System Structuring Concept, *Comm. ACM*, Vol. 17, No. 10, pp. 549-557 (1974).
7) Hansen, B.: *Operating System Principles*, Prentice-Hall, Englewood Cliffs (1975).
8) Cambell, R. H. and Habermann, A. N.: *The Specification of Process Synchronization by Path Expression*, *LNCS*, Vol. 16, pp. 89-102 Springer-Verlag, New York (1974).
9) Dijkstra, E. W.: Guarded Command, Nondeterminacy and Formal Derivation of Programs, *Comm. ACM*, Vol. 18, No. 8, pp. 453-457 (1975).
10) Enomoto, H., Yonezaki, N., Miyamura, I. and Sunuma, M.: A Parallel Programming Language and Description of Scheduler, The 14th IBM Computer Science Symposium (Oct. 1980).
11) 宮村, 米崎, 榎本: プロセス間通信による並列処理システム記述用言語について, 信学会 オートマトンと言語研究会資料, AL 79-35 (1979).

(昭和57年7月20日受付)
(昭和58年3月11日採録)