46 － S 004

オンラインシミュレーション言語SIMBOL

－遠隔情報処理システムの研究開発－

昭和 47 年 3 月

JIPDEC

財団法人　日本情報処理開発センター

この事業は、日本自転車振興会の機械工業振興資金による「昭和46年度 情報処理に関する調査・研究補助事業」のうち「遠隔情報処理システムの研究開発」の一部として実施したものであります。

序

当財団は，情報処理に関する調査および研究開発の一環として遠隔情報処理システムの研究開発を進めておりますが，この報告書はオンライン・アプリケーションの一つとして当財団で開発したタイムシェアリング端末から会話型式で使用できるシミュレーション・システムの開発成果を報告するものであります。

シミュレーションはコンピュータの代表的なアプリケーションの一つであると同時に，マン・マシンの会話的インタラクションによってより効果のあがる仕事であると言われてはおりますが，システムの大きさや，オンライン・システムとしての実用上の効果等に関してはまだかなりの問題を含んでおります。

この研究開発は，これらの問題点を解明する試みの一つとして取り上げたものであります。

本報告が広く利用され，わが国のソフトウェア発展の一助として寄与できるよう念願いたす次第であります。

昭和 47年3月

財団法人　日本情報処理開発センター

会長　難　波　捷　吾

まえがき

オンライン・シミュレータ SIMBOL（Simulation Model Builder For On－Line Usage）はタイムシェアリング・システムの元で端末からインタラクティブにシミュレーションをおこなう会話型システムである。

シミュレーションのオンライン利用の効果が云々されてすでに久しい。モデルの作成や変更，部分的な，あるいは試行錯誤的な実行の繰返し。これらを会話的にまたはインタラクティブにおこなえることの効果は確かにあるであろう。しかし一方シミュレーションの仕事そのものは，相当なCPUタイムと豊富なメモリースペースを必要とする代表的なものの一つでもある。TSSやオンラインシステムの環境下で，これらの要求を充分に満足させようとすれば，必然的にかなり大型のシステムを対象とせざるを得ないであろう。

MITのCTSSのもとで使用されていたというOPS－3，その後MULTICSのもとで使用される予定といわれたOPS－4等は，数少ないオンライン・シミュレーション・システムの例であろう。

我々が計画したシステムは，会話型的処理とバッチ型処理の両方の機能を持ち，使用者の希望により任意の切りかえが可能なものとした。シミュレーションという仕事の性質から，会話型処理のみではスピードの点で，実用性がやゝ薄いのではないかと判断したからである。またTSS端末としてCRTディスプレイを主体に考えたのは，簡単なグラフ表示，モデルのステータスのダイナミックなモニタリング等に新しい効果を出させようと試みたためである。

このプロジェクトは前年度からの継続であり，45年度は既存のシミュレーション言語の調査およびSIMBOLの言語仕様の設計を中心におこない，46年度は具体的なインプリメンテーションをおこなった。この報告書はシステムの設計およびプログラム設計の概要をまとめたものである。

目　　次

Ⅰ．システムの概要

Ⅰ. システムの概要

1. SIMBOLの特徴

○　会話型言語であること

このシステムの一番大きな特徴は，タイムシェアリングシステムのもとでオンライン・インタラクティブなシミュレーションを行う為の言語であると言う事である。

特に従来，シミュレーションをインタラクティブにおこなう事の効果が強調されてはいたが,必ずしもまだ多くのインプリメントが行われておらず，このシステムによる実用実験を大きな主眼としている。

○　プログラムの修正が簡単である

インプットされたステートメントはその場でコンパイルされエラーがあると即座に，そのむね表示される。ユーザはただちにプログラムの修正をする事が出来る。

○　部分部分を確めながらモデルを組立てられる

バッチ処理用の言語では，一応プログラムが完結していなくては，実行する事が出来ない。SIMBOL ではプログラムが完結していなくても，一部分だけを実行する事が出来る。従って大きなプログラムでも，部分部分のテストを行いながら全体を組立てて行く事が出来，能率的にデバッグをする事が出来る。

○　言語の表現能力が豊かである

一般に会話型言語はバッチ処理用言語に較べて，機能的に劣ると見られがちである。しかし少なくとも実用的なシミュレーション言語を目ざす以上モデルの記述能力が十分にある事は是非とも必要な事である。

一般に，シミュレーションを行わねばならぬようなものはかなり複雑なシステムである。従って複雑なシステムでも十分に記述出来るだけの表現能力を持っていなくては役に立たない。 GPSS がトランザクションを SIMSCRIPT がイベントを中心にシステムを記述するのに対して SIMBOL はプロセスを中心にシ

ステムを捕え記述する。この様なタイプの言語としてはSIMULAが有名でありプロセスタイプの言語は記述の簡明さと，表現能力の豊さを兼ね備えているとして高く評価されている。

○　モデル実行中にその様子を適格に摑む事が出来る

シミュレーション言語をオンライン化する事によって得られるメリットの一つは，実行中のモデルの様子を自由に参照出来る事である。

SIMBOLでは，モデルの実行中にクイットを掛けて，実行を一時中断して変数の値やシステムクロックを参照したり，スケジュールテーブルの様子を調べる事が出来る。もし必要なら，変数の値を変えてから実行を再会したり，場合によってはシミュレーションをそこで打切ったりする事など，適格な処置を取る事が出来る。又，SIMBOLには豊富なトレース機能が備わっており，モデルのデバックやモデルの動作を知る上で便利になっている。

トレースは単に機能が豊かなだけでなく，その扱いが自由である点で，バッチ処理用のシミュレーション言語が持っているこの機能とは，ハッキリ異っている。即ち，バッチ用シミュレーション言語でトレースを取る為にはモデルの実行する以前に何をトレースするかを指示すると，途中でもはや必要ない事がわかっても実行を終了するまで情報を取り続けてしまい不要な情報が大部分で，本当に必要な部分は，ほんのわづかであると言う経験は良くある。この点SIMBOLでは，トレース情報を見ながら，必要ない事がわかれば，ただちにコマンドを入れて打切る事が出来る。

○　モデルの実行方法に融通性がある

シミュレーションモデルの実行は普通のプログラムの場合と異り，出来上ってしまったプログラムを1回実行して，それで終りと言ったものでなく，パラメータの値を変えては，実行を繰り返したり，アルゴリズムを変えて実行を繰り返すと言った様に，色々な方法で取り扱われるものである。その為には，モデルを実行させる手段に充分な融通性が要求される。これを従来のバッチ処理用シミュレーション言語に期待するのは困難な事である。この点シミュレーションをオンラインで行う事に依って得られるメリットの1つであろう。

SIMBOLでは，実行の開始や終了を指示する手段はもちろんであるが，モデルのステータスを実行途中でセーブして，後にそれをロードし，実行を再開したりする手段も用意されている。

○　実行スピードを上げる工夫がされている。

会話型言語では融通性を大事にする為，どうしても実行スピードがそこなわれがちである。シミュレーションプログラムは一般に長時間CPUを使うプログラムが多く，実行スピードの遅速は大事である。SIMBOLの場合，デバックが済んだプログラムの一部，又は全部をコンパイルして実行スピードの速いオブジェクトを作成し，これを解決している。

○　データ集収機能が優れている。

一般のシミュレーション言語，例えば，GPSSとかSIMSCRIPTなどを見ても，特別に必要な統計データを得るにはTABULATEブロックやACCUMULATEステートメントなどを使って，あらかじめモデルの中に組込んでおかなくてはならない。モデルを実行して見てトレースを取る様に，統計量を得る事が出来れば，モデルのバリディティチェックにも大変役に立つ。SIMBOLにはGPSSのTABULATEブロックに相等するステートメントも，もちろん用意されているが，それ以外に，この様な便宜をはかるコマンドが用意されている。

○　キャラクターディスプレイを端末として使っている。

プログラムのインプットや結果の表示はキャラクターディスプレイ端末から行っている。シミュレーション・ランの結果や，統計データの分布をキャラクターディスプレイを使って簡単なグラフ表示が出来るので，その様子を一目で捕える事が出来，即座に，次に取るべき処置を決定する事が出来る。

2　モデル記述の基本概念

シミュレーション言語に於ける，モデル表現の考え方は，word－viewと言われる。GPSSはトランザクションタイプ，SIMSCRIPTはイベントタイプと呼ばれている。SIMBOLは，この意味で，プロセスタイプの言語と言う事が出来る。このタイプの言語としてはNorwegian Computing CenterのNygaardらによって開発されたSIMULA[1)]を挙げる事が出来る。この言語は非常にパワーフルな言語として高く評価されているが，残念ながらバッチ処理のもとでしか使う事が出来ない。数少ないオンラインシミュレータの試みのうちMITに於けるOPSシリーズは有名であるが，この内，最新版と言えるOPS－4もやはりプロセスタイプの言語である。我々が，SIMBOLの言語仕様を設計するに当って，どの様なworld－viewにするのが適当であるか大変迷ったあげく，どうもプロセスタイプが適当なのではないかと判断したのは，その言語の持つ表現力の豊さ故である。

一般に会話型の言語によって大きなプログラムを実行するのは不適当であるとされているが，少くとも我々のシステムは実用的なシミュレーションが出来る事を目的としている以上，言語自身の持つ記述能力が十分にある事はぜひとも必要である。

SIMBOLの言語仕様はプロセスタイプである点でSIMULAやOPS－4と共通しているが，特にこれらの言語の仕様にこだわってはいない。

一応，フリーな立場に立って設計している。ただ，プロセスタイプであるが為にどうしても類似する部分も出て来たが，これは仕方が無いと思っている。

2.1 プロセス

プロセスの概念を一言で述べるのは困難であるが，強いて言うならお互いに関連付されたイベントの集合と言う事が出来る。他のランゲージに於てもそうであるが，一般にシミュレーション言語に取り入れられているこうした概念（例えばSIMSCRIPTのイベント，GPSSのトランザクション）は，現実のシステムに於て，これがプロセスであると言ったハッキリした性格を持っておらずむしろ何をプロセスで表わしたら良いかはモデル化のテクニックとしてとらえる事が出来るものである。

つまり，複雑なシステムを記述するには，なるべく単純なサブシステムに分解して考え，あらためて全体を構成する方法を取る為にプロセスとかトランザクションとかイベントなどの単位が導入されており，どの様なサブシステムに分割したら，より見通しよくシステムを記述出来るかは，この様な単位をいかに上手に使うかと言った技術としての性格が強いと言えるであろう。

さて，プロセスには，個々に取扱うデータを割り当てる事が出来，一つのプロセス中でお互のイベント間の情報を受渡したり，プロセスの特性を示すアトリビュートとして使う事が出来る。このデータの事をローカルデータと呼び，各プロセスから共通に扱われるグローバルデータと区別されている。

SIMBOLの中で，プロセスを定義するのはアクティビティプログラムである。アクティビティプログラムは，一般のサブルーチンなどと同じ形式をしていて，唯，SUBROUTINEステートメントがACTIVITYステートメントで置き換えられている。もちろん，たとえ形式が似ていても取扱いは異にしている。

アクティビティプログラムとプロセスはかならずしも1対1に対応しておらず言わばプロセスの類が定義されたと考えられる。1つのアクティビティプログラムによって定義された，この類の事を単にアクティビティと呼んでいる。SIMBOLには，こうして定義されたアクティビティプログラムから個々のプロセスを発生する手段が用意されていてユーザが自由にコントロールする事が出来る。

プロセスが発生すると，アクティビティプログラムのコピーと，ローカルデー

タのエリアが割当てられる。（図１参照）

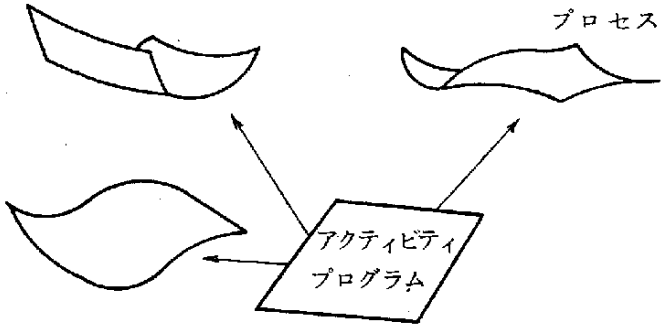


図１　アクティビティプログラムとプロセスの関係

システムの場にはこの様なプロセスが発生したり消滅したりしながら幾つものプロセスがコンカレントに実行される。もちろんコンカレントとは言うもののコンピュータの扱う事であるから，ある一時点では，ただ１つのプロセスしか実行されておらず，この実行手順はＳＩＭＢＯＬのスケジューラーに依ってコントロールされる。

2.2 セットとキュー

離散系システムのシミュレーション言語では，集合の表現や，その扱いについては十分の機能が要求される。GPSSではUser's chainが，この役割をはたしているが，十分な機能を持っておらずモデル記述に複雑な手間を要する一因となっている。最近のGPSSではこの点がGroup entityの導入などによって一部改良されつつある。これもその必要性を示す，良い例として受取る事が出来る。一般に，表現力の豊かな言語では集合の表現や扱いが，パワーフルである。これはSIMSCRIPTやSIMULAに見る事が出来る。SIMBOLに於けるSetは丁度，これらの言語に於けるSetと同じ様な概念である。Setは，エレメントを要素（メンバーと呼ぶ事にする）とし，順序化された集合である。もちろんSetに対してはエレメントをメンバーとして登録したり，Setのメンバーを削除したりするステートメントが用意されていて，シミュレーション実行中にダイナミックに変る集合の扱が出来るようになっている。

SIMBOLではSIMSCRIPTなどと異って，セットのメンバーとなるエレメントはあらかじめ，そのむね定義しておく必要もないし，1つのエレメントが同時に，幾つものセットのメンバーとなる事も出来る。これもあらかじめ定義しておく必要はない。SIMSCRIPTはSetのdata構造が異っている為に，この様な事が可能となっている。しかし，どちらの扱い方が優れているか一概に決められない問題である。

つまり，ユーザーが扱いに気を使う分だけ，メモリースペースと実行スピードの向上に役立っているからである。唯，ON-LINEインタラクティブな言語である事を考えると，なるべくUserが手軽に扱える方がまさっていると考えられる。

Setのメンバーは種々の方法によって参照される。一つは，システムが用意した関数があって先頭メンバー，最後尾のメンバー，第n番目のメンバーなどを値とするもので，これを使ってそのメンバーを取り出したり，attributeを参照したりなどが出来る。一方，プロセスやブロックの名前には特別な概念としてメ

ンバー名と言ったものが導入されており，直接その名前によって参照する事も出来る様になっている。

キューはエレメントを要素とした集合を表わし，メンバーが順序化されている事や，参照の仕方などセットとまったく同じである。唯，異る点は，キューでは，キューを構成する要素が変る度に，種々の統計情報を自動的にカウントしてくれる点と，キューをスケジューリングと関連して取扱う為の 2，3 のステートメントが用意されている事だけである。

SIMBOLではセットとキューを総称してグループと呼んでいるが，どのプロセスからも共通に参照されるグローバルなものとして扱われるだけでなく，各プロセスにローカルなグループを定義する事が可能である。

この為，グループのメンバーはエレメントに限られていても結果的にグループ自身をメンバーとするグループと言ったグループの階層化も出来る様になっている。

3　SIMBOLの言語体系

この章ではSIMBOLでモデルプログラムをコーディングする為に使うステートメントについて説明する。

言語体系と言う言葉を広く解釈するなら，直接，モデルをプログラムする為だけでなく，モデルの実行を指示したり，プログラムのエディトを行ったり，などするコマンドと呼ばれるものも含めて考えなくてはならないが，これは別の章に譲る事にしてここでは触れていない。

又，ステートメントやコマンドをSIMBOLにインプットする時の形体をラインと呼んでいるが，ステートメントについては単にラインナンバーが前に付くだけなので，この事も，ここでは無視している。

SIMBOLは前にも述べたが，GPSSなどと異ってSIMULAやSIMSCRIPTと同じ様に言語としての体裁をとっている。従って変数だとか式といった言語の中で扱われる概念についても述べなくてはならない。

3.1 モデル・ストラクチャー (Model structure)

SIMBOLのステートメントによってコーディングされたプログラムというのは，一般にはシミュレーションの為に作られたモデルである。そこでSIMBOLでコーディングされたプログラムをモデルと呼ぶことにする。

SIMBOLのモデルは，前に述べたアクティビティを基本に組立てられているが，これをサブルーチンとか関数と言った単位に分割してプログラミング出来る様にしてある。SIMBOLはサブルーチンに相当するプログラムをプロセデュアー（Procedure）と呼び関数はALGOLの様にプロセデュアーにタイプ（例えばREALとかINTEGER）を付けて表わす事にしている。又，グローバルなデータを定義したり，シミュレーション実行時のイニシャライズをしたり，などする為にメインプログラムが設けられている。結局SIMBOLのモデルはメインプログラム，アクティビティ，プロセデュアー，関数などを基に組立てられるが，このうち，メインプログラムはかならず組込まれていなくてはならない。（唯1つだけ）。

もちろん，これらのプログラムはSIMBOLのステートメントによってコーディングされていて，それぞれ，アクティビティならACTIVITYステートメントが，プロセデュアーならPROCEDUREステートメントが，関数であれば，REAL PROCEDUREステートメントが，と言うように，これらを区別するステートメントで始まり終りを示すENDステートメントによって区切られている。唯，メインプログラムだけは特に決ったステートメントが先頭にくる様な事はなく，終りを示すENDステートメントだけが用いられる。これは他のプログラムとの関連で解る様になっている。

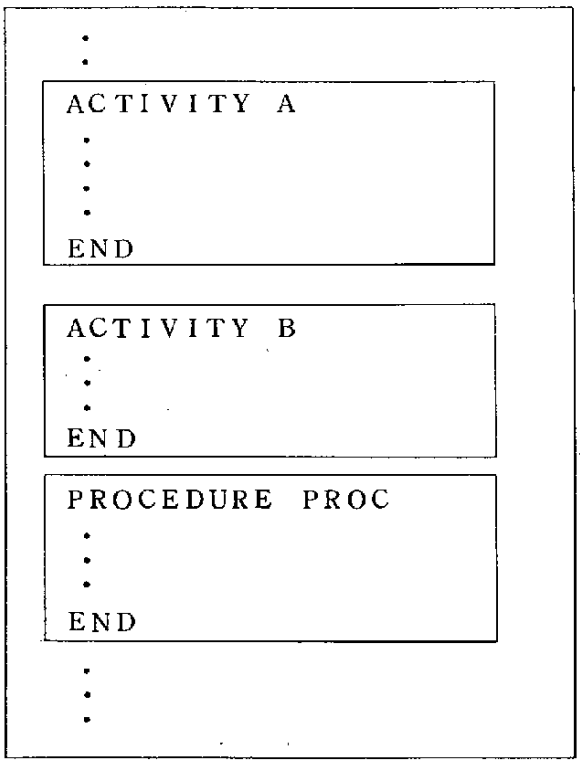


図2　プログラムの構造

3.2 SIMBOLで扱うデータ

SIMBOL では，実数型データ，整数型データ，論理型データと言った一般の汎用言語で扱われるもの以外に，エレメントデータ，グループ，統計用テーブルといった新しいデータタイプを扱う事が出来，シミュレーションの為に便宜を図っている。

実数型データ，整数型データ，論理型データ，エレメントデータには，これらを扱う為に変数が用意されていて，宣言文で名前を定義して使う事が出来る。一方，グループと統計用テーブルは特にこれを扱う為の変数は用意されていないがやはり宣言文によって，これらのデータを定義し，そこで与えた名前によって，これらを扱う事が出来る様になっている。これら変数は単純変数としてだけでなく，もちろん配列として使用が可能であるが，配列は2次元までしか許していない。（グループや，統計用テーブルも配列で扱う事が出来る）

実数型や整数型データの定数はFORTRANとまったく同じパターンで書く事が出来，論理定数としては，真，疑を表わすTRUE,FALSEがある。

エレメントデータは，エレメント，即ちプロセスやブロックを表わすもので，これらを参照する為の名前として扱われる。エレメント定数とでも呼ばれるべきNONEは存在しないエレメントを示し，いわば数値のゼロに相当するものである。一方，システム変数として用意されたSELFは，その時点でコントロールが渡っているプロセスを表わしている。

又，グループについては後の章で詳しく説明するが，セットとキューを総称したもので，実際には，SETステートメントとQUEUEステートメントと別々のステートメントによって定義される。

統計用のテーブルには2つあって，1つはデストリビューションテーブル（Distribution table）と呼ばれ，サンプルしたデータのヒストグラムを作る時に使い，もう1つのプロパティーテーブル（Property table）は，データ群の平均，分散，max，minといった，統計量だけを必要とする時に便うと便利である。これもグループと同じく，後の章で詳しく述べる事にする。

3.3 データの定義

SIMBOLで扱う変数やグループ，テーブルなどは全部宣言ステートメントによって定義しなくてはならないが，宣言ステートメントとしては次のものがある。

1. REALステートメント
2. INTEGERステートメント
3. LOGICステートメント
4. ELEMENTステートメント
5. SETステートメント
6. QUEUEステートメント
7. DTABLEステートメント
8. PTABLEステートメント

宣言の仕方はどのステートメントも同じで，ステートメント名の後にデータの名前を書く事によって行う。

例えば，

```
REAL  A,B,C; とか
SET  S(20);
```

の様に書く。

3.4 データの参照

データにはそれを識別する為に名前が与えられていて，名前によってreferする事が出来る。一般の汎用言語では，そのまま名前によって，そのデータが参照されるのが普通である。しかし，SIMBOLの場合，ローカルデータの参照に関して特別な扱いがなされる為に，変数の名前と，参照の仕方をハッキリ区別して考える必要がある。

SIMBOLのデータは，プロセデュアーとか関数を除いたアクティビティ，及びメインプログラムを考えると，グローバル，又はローカルという Scope attribute がかならず与えられている。

このうちグローバルデータは各プロセスから共通に参照されるもので，変数の名前と実体は1対1に対応している。従って，この場合には単にその名前を書けば，変数の内容を参照する事が出来る。

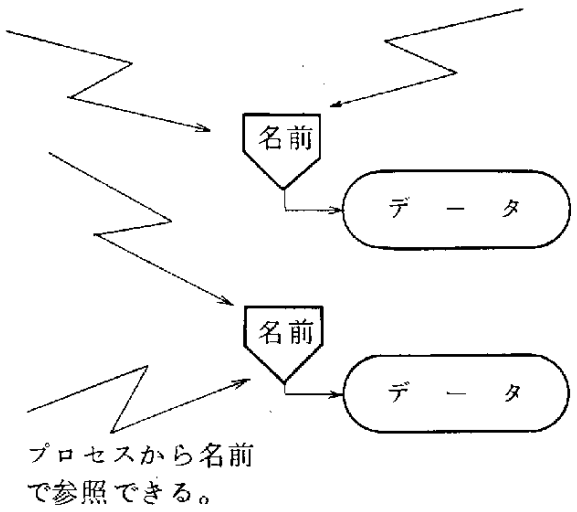


図3　グローバル変数の参照

ところが，アクティビティプログラムで定義されたデータ，即ちローカルデータは，1つのアクティビティから一般には複数個のプロセスが発生し，同じ名前の変数に対して，プロセスの数だけ実体が関係付られている。

例えばアクティビティが，

```
ACTIVITY CAR ;
REAL WEIGHT, SPEED ;
 .
 .
 .
END
```

と定義されると，プロセスが発生する度に，WEIGHT，SPEEDに対応するデータエリアが確保される。その関係は次の図の通りである。

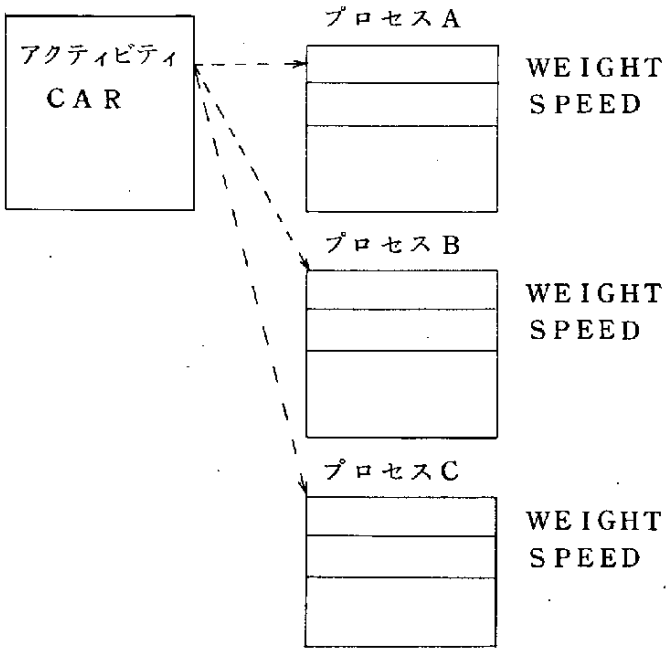


図4　ローカル変数とアクティビティプログラム

図で見れば明らかな様に各プロセスのデータに対してWEIGHT，SPEEDと言う同じ名前が付いているので，単に名前だけでは，どのWEIGHTやSPEEDであるか判別が付かず，参照する事ができない。そこで，プロセスにローカルなデータは，プロセスの名前と変数の名前を組合せて参照する事にしている。唯も

う一つ，やっかいな問題があって，こんどは異ったアクティビティで同じ名前を使う事があり，結局アクティビティの名前も含めて指定する方法を取っている。もちろん，これは言語仕様を制限して，異ったアクティビティに於ては，同じ名前を使う事を禁止すればこの指定が無用になる。

以上の事から，図の例ではプロセスAのWEIGHTを参照するには，

A：CAR．WEIGHT

プロセスBのWEIGHTは

B：CAR．WEIGHT

プロセスCのSPEEDは

C：CAR．SPEED

と書く。

この様にプロセスが他のプロセスローカルなデータ，又はブロックのローカルデータを参照するときの仕方をエクスターナルリファレンス（external reference）と言う。

エクスターナルリファレンスの一般型は，

＜エレメント＞：＜クラス名＞．＜ローカルデータ＞

で表わされる。

ここでエレメントと書いてあるのは単にエレメント変数であってもよいし，エレメントを示す函数（これをエレメント函数と呼ぶ）であっても良いし，これ自身エクスターナルリファレンスの形をしていてもかまわないが，その値がエレメントである事を意味している。

又，クラス名というのはアクティビティ名，又はブロック名を総称して呼んだ名である。

ローカルデータを参照する，特別のケースとして，プロセスが自分自身にローカルなデータを参照する場合が考えられる。この場合には，プロセスの属すアクティビティを示す名前はあらかじめ明らかなので指定する必要がなくグローバル変数の場合と同じ様にデータ名だけを書けば良い。

この参照の仕方をローカルリファレンスという。

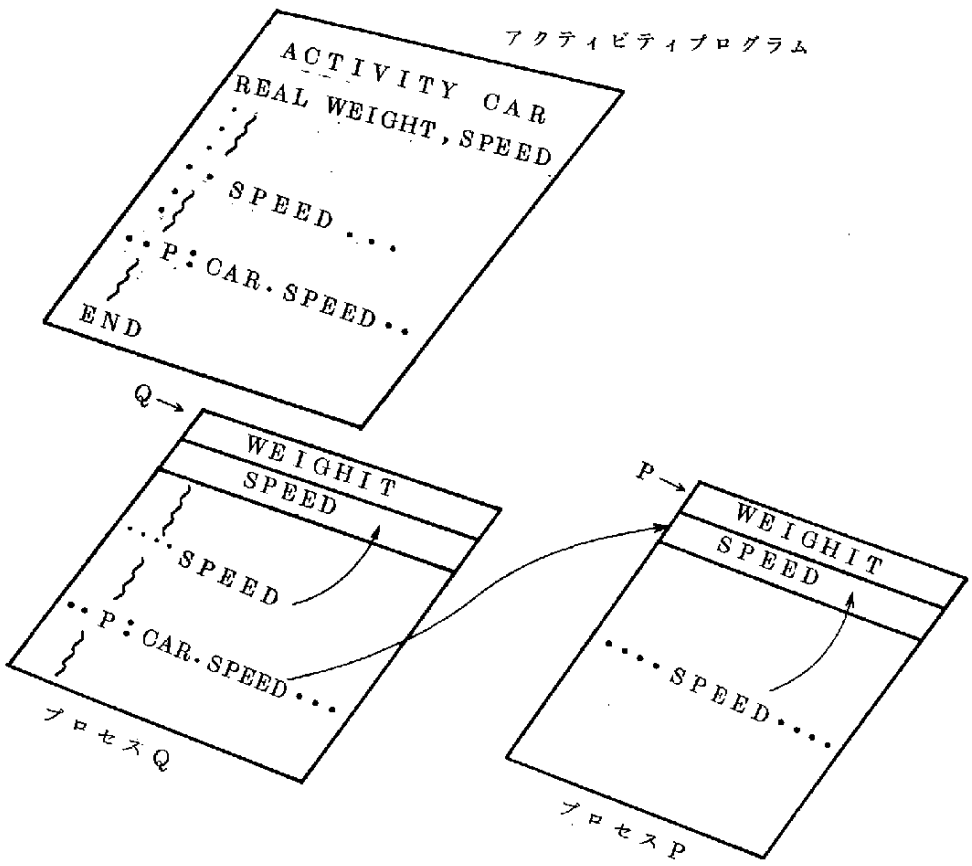


図5　ローカル変数の参照

3.5 エレメント式

エレメント式は，その名の通りエレメント，即ちプロセス又はブロックを値として取り，これらを参照する手段として使われる。

この式は，

1. エレメント変数
2. エレメント関数
3. 生　成　式

のいずれかである。

ここでエレメント変数はエクスターナルリファレンスの形をしていてもかまわない。

生成式については次に説明する。

3.6 生成式 (Generative expression)

生成式と言うのはエレメントを値として取る式であるが，この式が算定される度に新たにエレメントを１つ発生させ，そのエレメントが式の値となる。アクティビティ名か，ブロック名の前にNEWオペレータを付け，必要な発生するエレメントのローカル変数に初期値を設定する為に，パラメータ部を書き加えたものを生成式という。

例えば，アクティビティMESAGEに対して

```
NEW MESAGE
```

は生成式で，MESAGEから発生した１つのプロセスを値としている。
MESAGEがもし次の様に定義されているとする。

```
ACTIVITY MESAGE
INTEGER LENG
  .
  .
  .
END
```

新に発生するプロセスのLENGと言うローカル変数にはグローバル変数LENGTHの値を初期設定したければパラメータ部を付け加えて

```
NEW MESAGE(LENG=LENGTH)
```

と書けば良い。もし初期設定したいローカル変数が２つ以上あるならカッコの中にカンマで区切って並べれば良く，その際順番は問題とされない。

生成式は後述するSCHEDULEステートメントとか，INSERTステートメントNAMEステートメントなどの中に書かれ，例えば

```
SCHEDULE NEW MAN DELAY 0 ;
INSERT NEW BOY -> CLASS ;
NAME TOM:=NEW BABY ;
```

の様にして使われる。

3.7 グループ（セット及びキュー）

グループはエレメントを要素とする集合を表わす一種のデータ構造である。SIMBOLで扱うグループは他の変数と同じ様にあらかじめ宣言しておかなくてはならない。セットはSETステートメントで，キューはQUEUEステートメントで定義をする。配列型のセットやキューを扱う事が出来その定義の仕方は，普通の変数の場合とまったく同じであって，例えば，

SET S(5);

とすれば，Sという配列型のセットが定義され，1つ1つのセットはS(1)，S(2)，S(3)，S(4)，S(5)という様にして参照する事ができる。定義されたばかりのセットやキューはメンバーをまったく含まない，いわゆるエンプティな状態にある。内部的にはエンプティなグループでも何も存在していないわけでなく，セットヘッド或は，キューヘッド（以後これを総称してグループヘッドと呼ぶ事がある）が作られている。（実をいうと，これは正確な表現でない。後で明らかにする。）

グループはグループヘッドを基点にしてエレメント（プロセス又はブロック）が対称リストでチェインされていると考えられる。SIMSCRIPTやGPSSのキューは，それ自身にFirst-in First-outとか，Last-in Fast-outと言ったメンバーの扱いに関する性格を与えているが，SIMBOLでは，キュー自身には与えず，キューを扱うステートメントの側で指定する事になっている。

3.7.1 セットとキューの相異

セットもキューもエレメントを要素とする集合を表わす事では変りないが，キューでは出入りしたメンバーに関連した統計量を自動的にカウントしてくれる点で異っている。セットでは，この様な処理は一切行われない。キューがカウントする情報はキューヘッドにセーブされていて，システムが用意するプロセデュアーを使って統計情報を得る事ができる。

その内容は以下の通りである。

1. キューに入ったメンバーの総数

2. キューの平均メンバー数

3. 1時点でキューにいたメンバーの最大数

4. 待ちなしで出て行ったメンバーの総数

5. 上記のメンバーを除いた，平均待時間（キューに入ってから出ていくまでの時間の平均）

3.7.2 エレメントにローカルなグループとグローバルなグループ

グループが定義された場所がメインプログラム中であるか，アクティビティであるかによって前者をグローバルなグループ，後者をローカルなグループという。これは次の理由による。

グローバルなグループというのは，グループの定義がなされると同時にグループヘッドが実際に作成されるが，ローカルなグループは普通のローカル変数と同じく，それを定義しているアクティビティやブロックから実際にプロセスやブロックが発生するとローカルデータの一部としてグループレッドが作成される。その為，定義された時点ではグループヘッドすら存在していない。グループにグローバル，ローカルの概念を導入する事で，グループ自身をメンバーとするグループの扱いがスッキリする。即ち，グループが直接グループをメンバーとしないで，グループのメンバーはあくまでエレメントであり，そのエレメントにローカルなグループが結果的に前のグループのメンバーとなっていると考えられる点である。これを概念的に示したのが次の図である。

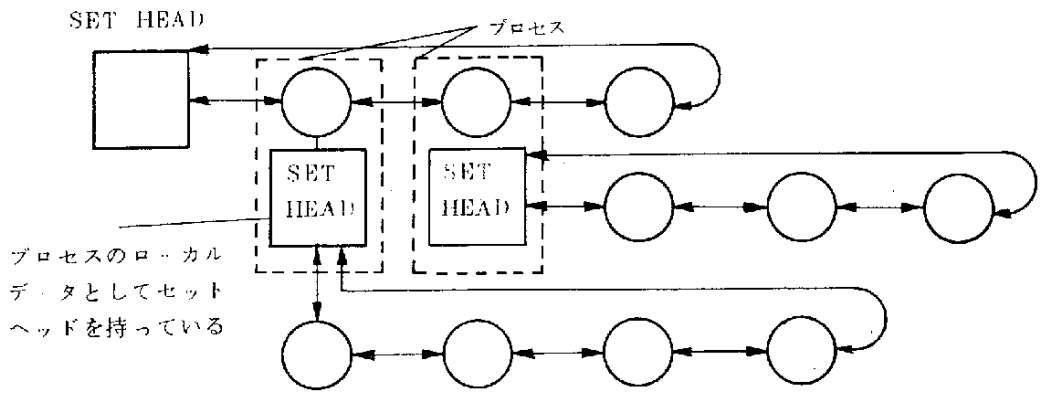


図6 エレメントにローカルなセットの例

3.7.3　グループの参照

グループにも，エレメントにローカル，グローバルと言った概念がある為にグループ自身を参照する場合にも実数型，整数型，エレメント型などのデータと同じ扱いがなされる。

従って，例えばアクティビティACTで定義したセットSが自分自身にローカルなセットとして参照する場合には単にSと書けば良いが，他のプロセスがこのセットを参照する場合には，そのプロセス名をPとすると，

P：ACT．S

としてエクスターナルリファレンスの形体を取らねばならない。

3.7.4　グループを扱うステートメント

この種のステートメントには

(1)　INSERTステートメント

(2)　REMOVEステートメント

があり，この両者はセットであってもキューであっても対象とする事が出来るが，キューについてはスケジューリングステートメントと関連してWAITステートメントがある。

ここではINSERTステートメントとREMOVEステートメントについて述べ，WAITステートメントはスケジューリングステートメントの項にゆずる事にする。

(1)　INSERTステートメント

このステートメントは，セットやキューにエレメントを挿入し，これらのメンバーとする時に用いる。対象とするセット（又はキュー）とエレメントの他に，挿入する位置を指定する事が出来，この指定の仕方を位置指定と呼んでいる。

位置指定には，セット（又はキュー）の先頭，最後尾，特定メンバーの前又は後といった4つの方法がある。この形式は次の通りである。

① FIRST

② LAST

③ BEFORE ＜メンバー＞

④ AFTER ＜メンバー＞

特に位置指定が無かった場合には，LASTと指定されたと解釈される。

図7は上記の位置指定とエレメントの挿入される位置との関連を示したものである。

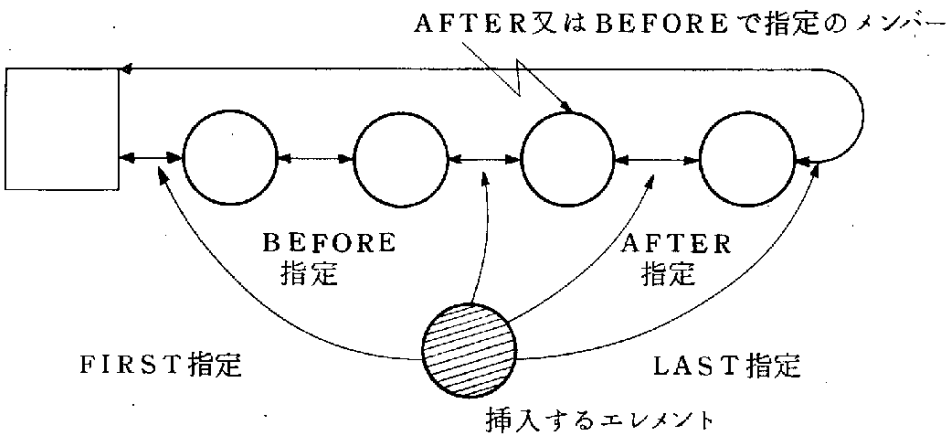


図7　位置指定と実際に挿入される場所

この位置指定も含めてINSERTステートメントの形式をBNFで書くと，

＜INSERTステートメント＞ ::= INSERT ＜エレメント式＞->＜グループ＞

〔＜位置指定＞〕

となる。

(2) REMOVEステートメント

このステートメントはINSERTステートメントと反対に，セットやキューのメンバーとなっているエレメントを，取り出す為に使われる。どのメンバーを取り出すのかを指定するには，前に説明したメンバーを参照する為の関数（例えばTOP，BOTOMなど）とか，エレメントに付けられたメン

バー名などによって行う。REMOVEステートメントには2つの形式があって，1つは，単にメンバーをグループから取り出すもので，もう1つは取り出したエレメントに名前を付ける機能が追加されているものである。

後者は，グループに登録されているメンバーを取り出した後に，なんらかの扱いをする為に名前を付ける必要がある場合には便利である。つまり，前者の機能だけでは，一度取り出してしまったエレメントは，予め名前が付いていれば良いが，そうでない場合，その後の扱いが出来なくなってしまうので是非とも必要な機能である。

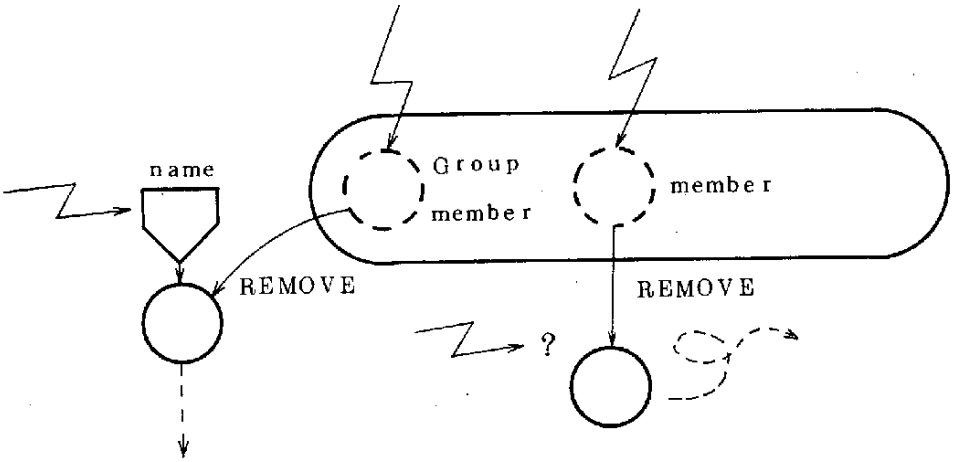


図8　メンバーの削除と名前付け

REMOVEステートメントの形式は次の通りである。

REMOVE〔＜メンバー＞〕＜－＜グループ＞

REMOVE＜エレメント変数＞：＝＜メンバー＞＜－＜グループ＞

3.8　スケジューリング

3.8.1　スケジューリング

一般に離散系のシミュレーション言語では，本来全く同時に起るはずのイベントでも，コンピュータが特性から言って，適当な順番に並べシリーズに実行しなくてはならない。こういった事は最近，特に問題となっており色々と議論のある所である。しかし，これを解決するのはなかなか難かしい問題である。こんな意味もあって，シミュレーション言語のuserは，シミュレータ自身の持っているスケジューリングメカニズムについての十分な知識を要求される羽目になる。

ここでSIMBOLが持つスケジュール機能とスケジューリングメカニズムについて触れる前に，プロセスについて振り帰っておく必要がある。これまでにプロセスについては，プロセスが持つ概念的な面を説明してきた。アクティビティプログラムと関連してプログラムサイドからながめておかねばならない。プロセスには個有のローカルデータと，プログラムコードのコピーが対応して作成され，これを定義する。言い換えるならコピーの原紙と言ったものが，アクティビティであるものを述べて来た。しかしこれは，あくまで概念的な説明で，実際にプログラムコードのコピーが作成される事はない。

実際には，1つのプログラムコードを各プロセスが共通に使う。従って，このプログラム（つまりアクティビティプログラム）はリエントラントに作られている。一方，ローカルデータは各々のプロセスについて，実際に場所（メモリー）を必要とするのでコピーに相当するものが作成される。この事は第2編に於て触れるが，プロセスについてはプロセスコントロールブロック（PCBと略記）が作成されて，ローカルデータのエリアとか，プロセスに付帯する種々の情報を記入する場所が確保されている。

図9はこの関係を表わしたのである。

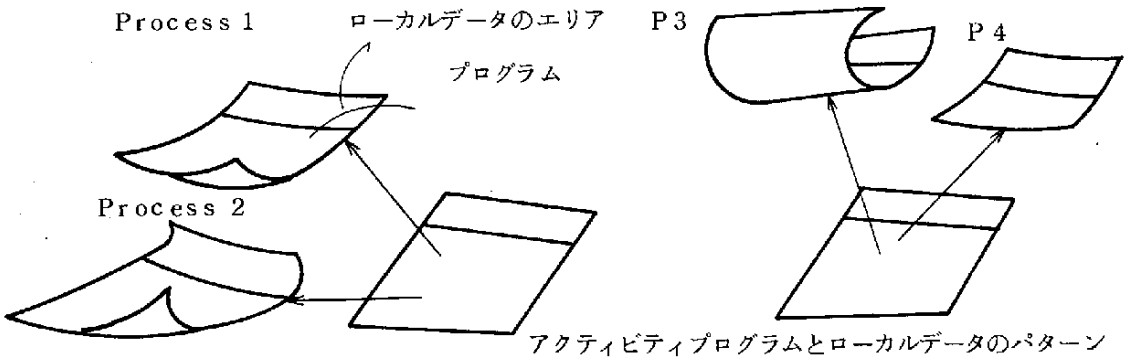


図9　アクティビティプログラムとプロセス

各プロセスが同じプログラムを共通に実行する時に，１つのプロセスの実行（これはアクティビティプログラムの実行である）は，後述する，ＤＥ－ＬＡＹステートやＷＡＩＴステートメントなどによって中断され，他のプロセスの実行に切り換えられる。この為，各プロセスがどこまで実行したかを示すポインターを，それぞれに持たせてコンカレントな実行が，うまく行くようにしてある。このポインターのことをreactivation－pointと呼んでいる。つまりリアクティベイションポイントは各プロセスの実行中断点を示してコントロールが戻ると，ここから実行が再会される。従って１つのプロセスについて考えると，自分に対応するアクティビティプログラムの実行は単に途中で中断されると言うだけで，実行順序は普通プログラムが実行されるのとまったく同じである事がわかる。この様な実行順のことをローカルシーケンス（local　sequence）と呼び，全てのプロセスを考えた時の実行順序はオーバオール・シーケンス（Over－all－sequence）と呼ぶ。

オーバオールシーケンスには最低次の原則がまもられている。つまり１つのプロセスにコントロールが渡ったとき実行が中断される様なステートメントは決っていて，スケジューリングステートメントのみである。これも全てのスケジューリングステートメントが対象となるわけでない。この事については今後の説明で明らかにしていく。従って，これ以外のステートメントを実行しているかぎりコントロールが他のプロセスに移る事は絶対にない。この様に１つの

プロセスで中断が起らない区間のことを active-phase と呼ぶ。もちろん1つの active-phase が実行されているあいだはシミュレーションブロックは進まず，同一時刻に於いて実行されている。この意味で active-phase はプロセスに於けるイベントと言う事が出来る。

SIMBOL が持つスケジューリングメカニズムの行う主な仕事は，オーバオールシーケンスに従って各プロセスにコントロールを渡しシミュレーションクロックを管理する事である。

オーバオールなシーケンスで各プロセスにコントロールが渡り，実行されて行く過程というのは，即ちシステムの動きをそのまま反映したものである。これはもちろん SIMBOL の User が意識して組立てたものであって，その為の手段が用意されていなくてはならない。この様なシステム流れをモデルの中で関連づけるのがスケジューリングステートメントである。

スケジューリングステートメントは，基本的な機能としてイベントを特定の時刻に実行されるべく，スケジューラー（スケジューリングメカニズムを司さどる部分をこう呼ぶ）が持っているスケジュールテーブルに登録したり，あるいは削除したりする機能を持っている。スケジュールテーブルというのは，スケジューラーがオーバオールシーケンスとかシミュレーションブロック進行の管理をする為のテーブルで，テーブルにはイベントに対応してイベントノーティスが登録されている。イベントノーティスの具体的な形体については第2編で詳しく述べてあるのでここでは触れないでおく。ただ，イベントノーティスには，そのイベントが実行されるはずの時刻（これを Execution Time,ET と略記する）が書かれていて，スケジュールテーブルに登録されているイベントノーティスはETの順にソートされている。オーバオールシーケンスはこのスケジュールテーブルによって決定される。つまり，一番ETの小さいイベントにいつもコントロールを渡して行けば，Time sequence に従ってイベントが起ると言った過程が実現される。ここでイベントと呼んでいるのは前に述べた様に丁度プロセスの active-phase に対応している訳で，イベントにコントロールを渡すと言う表現は，プロセスの active-phase にコントロール

を渡す事の意味である。

SIMBOL 言語で又，イベントはあくまで概念的な単位であって，直接，これを示すイベントデータといったものは存在しない。従ってイベントはプロセスによって implicit に表現される。即ち，プロセスを指定してスケジュールすると，そのプロセスのローカルシーケンスに従った，「今度実行される active phase（イベント）」が一意に定まっているからである。

3.8.2 プロセスのステータス

プロセスにはスケジューリングと関連してステータスが定義される。

プロセスのステータスには次の6つがある。

(1) アクティブ（Active）

(2) スケジュールド（Scheduled）

(3) パッシブ（Passive）

(4) インタラプティド（Interrupted）

(5) ターミネイト（Terminate）

(6) デッド（Dead）

以上のうち，(1)〜(5)までは少なくともプロセスが存在する時のステータスで(6)はプロセスがシステムの場に存在していない事，例えば，一度発生したプロセスが消滅してしまった様な場合などに相当している。そこで(1)〜(5)については，(1)と(2)はイベントノーティスが作られている事，即ちスケジュールテーブルに登録されている場合で，アクティブと言うのは，現在コントロールが渡っているプロセスの状態で，ある一時点で見る限り，唯一つのプロセスしかこの状態に成り得ない。

(4)のインタラプティドな状態と言うのは，まさに割込まれた状態の事であるが，これについては後のスケジューリングステートメントの章で INTERRUPT ステートメントを説明する時に述べる事にする。

前後するが，(3)のパッシブの状態とは，スケジュールテーブルに登録されていず，従ってイベントノーティスも持たないで，キューに入って順番を待って

いる時などこの状態になる。従って，この状態にあるプロセスは，他のプロセスによってスケジュールされなければ実行されない訳である。

又，(5)のターミネイト状態の場合では，すでにプロセスの実行を完全に終了してしまって，再びスケジュールしても何にももう実行する部分がない状態である。しかし，(6)と異る所は，(5)ではローカルデータのみシステムの場に残っている事である。これらの関係をまとめたのが次の表である。

表1　プロセスのステータス

プロセスのステータス	イベントノーティス	リアクティベイションポイント	ローカルデータ	実　行
アクティブ	あ り	あ り	あ り	実行中
スケジュールド	あ り	あ り	あ り	予　定
パッシブ	ナ シ	あ り	あ り	予定ナシ
ターミネイティド	ナ シ	ナ シ	あ り	予定ナシ
インタラプティド	あ り	あ り	あ り	予定ナシ
デッド	ナ シ	ナ シ	ナ シ	予定ナシ

3.8.3　スケジューリングステートメント

SIMBOLのスケジューリングステートメントには次のものがある。

(1)　SCHEDULEステートメント

(2)　RESCHEDULEステートメント

(3)　CANCELステートメント

(4)　TERMINATEステートメント

(5)　DELAYステートメント

(6)　WAITステートメント

(7)　WAKEステートメント

(8)　INTERRUPTステートメント

(9) RESUMEステートメント

まず，SCHEDULEステートメントは，プロセスを新にスケジュールする為に使われる。従って新に発生したプロセスをスケジュールする時などに使われる。これに対してRESCHEDULEステートメントでは，既にスケジュールされているイベントの予定変更を行う。つまりスケジュールし直す時に使う。CANCELステートメントも同じくスケジュールの変更であるが，スケジュールし直すのでなくキャンセルする点が異る。キャンセルされたプロセスはパッシブな状態になる。TERMINATEステートメントはプロセスをターミネイト状態にする機能を持っている。

(7)(8)も含めて以上のステートメントは，原則として他のプロセスを～するといった，いわば他動詞的な表現に使われる。これに対してDELAYステートメントとWAITステートメントは，このステートメントを実行するプロセスをスケジュールする為のものでセルフスケジューリングと呼ばれるものである。DELAYステートメントは，このプロセスを一定時間後にスケジュールし直す機能，即ち，時間経過を表わす為のステートメントである。又，WAITステートメントは，このプロセスをキューに入れ待ち状態（正確にはパッシブな状態）にし，他のプロセスによってスケジュールされる（例えば，キューにおける順番が回ってくる）まで，この状態に停っている事を表わしている。

(1) SCHEDULEステートメント

このステートメントは，パッシブ状態にあるプロセスをスケジュールする時に使われる。パッシブ状態のプロセスというのは，2つのケースが考えられる。第1は，全く新たにプロセスが発生された時で，第2は，過去に少なくとも一度，スケジュールされたがCANCELステートメントなどでスケジュールをキャンセルされている状態である。

この様なプロセスがスケジュールされると対応してイベントノーティスが作成され，スケジュールテーブルに登録される。将来，順番がくると実行される事になる。スケジュール方法の指定は大きく分けると2通りの方法で行

う事ができる。第１は，時間を指定する方法で，この場合には直接実行すべき時刻を指定するか，現在の時刻からの相対時間を指定する。第２は既にスケジュールされているイベントに関連して，その直前とか直後を指定する方法である。第１の場合にはスケジュールされたプロセスの ET は指定された時刻なり相対時間によって計算できるし，第２の場合には，スケジュールされているイベントは ET の値が確定しているから，その値と同じにする事で結局どちらの場合にも，この値が決定される。

以上のスケジュール方法の指定をスケジュールステートメントの中で与える形式をスケジュール節と呼び，次の形式をしている。

① AT ＜時刻＞

② DELAY ＜時間＞

③ AFTER ＜プロセス＞

④ BEFORE ＜プロセス＞

これらの意味は上記で明らかなので説明は省略する。
唯，＜時刻＞とか＜時間＞というのは一般の算術式でおえられる。

①と②の場合に，この後にカンマで区切ってPRIORと書く事ができて，これは，スケジュールテーブルに登録されている今スケジュールしようと思っているイベントと同じ ETを持つ，イベントがあると，それらの一番先頭に登録する事を指定している。

従って，同時刻に起る幾つかのイベントの内，一番始めに実行されるわけである。特に，

AT TIME* , PRIOR とか

DELAY 0 , PRIOR

と書かれた場合，このステートメントを実行したプロセスからスケジュールしたプロセスに直接コントロールが移行してしまう。この意味でPRIORの指定をダイレクト（direct）スケジューリングと呼んでいる。

以上，PRIOR指定も含めてスケジュール指定を行う形式をスケジュール句と呼び，BNF で記述すると，

＜スケジュール句＞：：=＜時間によるスケジュール節＞|＜時間によるスケジュール節＞，PRIOR|＜イベントによるスケジュール節＞

＜時間によるスケジュール節＞：：＝AT ＜時刻＞|DELAY ＜時間＞

＜イベントによるスケジュール節＞：：=AFTER＜プロセス＞|BEFOR＜プロセス＞

＊ TIMEはシステム変数と呼ばれ現在の時刻（シミュレーションクロックによる）を表わしている。

SCHEDULEステートメントは，このスケジュール句とプロセスの指定を加えて次の様な形式で書く。

SCHEDULE ＜プロセス＞ ＜スケジュール句＞

例えばJOB－SHOPモデルで注文が，発生した場合には

SCHEDULE NEW ORDER DELAY O ；

と書けば良い。

(2) RESCHEDULEステートメント

このステートメントは現在スケジュールされているイベントのスケジュールを変更する為のもので，新たに行うスケジューリングの方法はSCHEDULEステートメントと同じ要領で指定する。

一般的な形式は，

RESCHEDULE ＜プロセス＞ ＜スケジュール句＞

で，例えばプロセスPは時刻Tに実行すべくスケジュールされていたが，T_1に実行される様，変更したい時には，

RESCHEDULE P AT T_1 ；

とすれば良い。

RESCHEDULEステートメントが実行されると内部的にはスケジュールテーブルに登録されている。そのプロセスのイベントノーテスを取り出し，実行時刻を指定に従って書き換え，再びスケジュールテーブルにするという操作が行われる。

(3) CANCELステートメント

このステートメントも，RESCHDULEステートメントと同様に，現在スケジュールされているイベントのスケジュール変更を指定するが，この場合にはその名の通り，スケジュールをキャンセルする。

キャンセルされたイベントを再びスケジュールされなければ，実行される事はない。もし，再びSCHEDULEステートメントやWAKEステートメントによってスケジュールされると，以前，実行するはずであったイベントがされる。つまり，このプロセスが持つプログラムコードで，今まで実行してきた「続き」が実行されると言う意で，これは，プロセスのリアクティベイションポイントが，スケジュールをキャンセルされても，変更されないという事でもある。

(4) TERMINATEステートメント

アクティブ，スケジュールド，パッシブ，この内のいづれかの状態にあるプロセスをターミネイト状態にする時使う。

アクティブな状態にあるプロセスを対象とする事は，TERMINATEステートメントを実行しているプロセス，つまり自分自身をターミネイトさせる事である。

この場合には対象プロセスの名前は書かないが，一般には，

TERMINATE ＜プロセス＞

の形式で使い，ターミネイトするプロセスを指定する。

(5) DELAYステートメント

このステートメントの形式は

DELAY ＜時間＞

で，現在時刻から指定した後に自分自身をリスケジュールする機能を持っている。これはGPSSのADVANCE BLOCKに相当するステートメントで，一つのプロセスに於ける時間経過を表わすのに使われる。例えば10単

位時間CPUを使う時には

DELAY 10 ;

とする。

もし同じ事をRESCHEDULEステートメントを使って表わすには，

RESTHEDULE SELF DELAY 10 ;

と書く事ができる。

(6) WAIT ステートメント

WAIT ステートメントはこのステートメントを実行したプロセスをキューに接ぎ，待ち状態にする機能を持っている。従ってそのプロセスは，他のプロセスがキューから取り出してスケジュールしてくれるまで実行されない事になる。ステートメントの形式は，

WAIT ->＜キュー＞〔＜位置指定＞〕

で指定したキューにこのステートメントを実行したプロセスを挿入する。挿入する位置は位置指定で行うが，これはINSERT ステートメントの書き方とまったく同じである。この指定が省略されるとキューの最後尾に挿入される。

例えば，

WAIT -> RDYQ ;

が実行されるとこのプロセスはRDYQと言う名のキューの最後尾に挿入され，プロセスのステータスはアクティブからパッシブに変る。

(7) WAKE ステートメント

このステートメントはWAITステートメントと組合せて使うと便利で，WAIT ステートメントがプロセスをキューに挿入し待ち状態にするのに対して，この様に待ちにあるプロセスをキューから取りだしてスケジュールする機能を持っている。スケジュールの仕方については一般のスケジュールステートメントと同じくスケジュール句で指定できるが省略する事もできる。その場合にはDELAY 0でスケジュールされたと見なされる。

ステートメントの形式は次の通りである。

WAKE 〔＜メンバー＞〕＜－＜キュー＞〔＜スケジュール句＞〕

メンバーの指定はREMOVEステートメントの場合と同じ意味で，省略すればキューの先頭メンバーとみなされる。

従ってWAKEステートメントの一番簡単な形は

WAKE <- RDYQ ;

とだけ書けば良く，この例ではRDYQというキューにある先頭のプロセスを取り出しDELAY 0でスケジュールする事を指定している。そのプロセスは現時刻で実行される事になる。

(8) INTERRUPTステートメント

このステートメントはその名の通りインタラプションを起す機能を持つものである。インタラプションとは言え計算機で起るインタラプションとはちょっと趣を異にしている。これはシミュレーション言語で言うこの機能に共通して言える事である。即ちシミュレーションプログラムの実行している最中に起すインタラプションではなく，モデルで現実のシステムを表わす時の「インタラプションの表現」である。

例えば，計算機システムをシミュレーションするのにCPUを使っていると言う表現は一つの変数の値を例えば5秒（シミュレーションクロックで）間だけ1にしておく事でも表わせる。その間，つまり5秒間はそのプロセスをスケジュールテーブルに登録しておく事に相当し，CPUを使っているプロセスがアクティブな状態でなくスケジュールドの状態にあるわけで，現実に実行されているのは別のプロセスである。従ってCPUを使っているプロセスにインタラプトを掛ける事はスケジュールテーブルに登録されているそのプロセスに対してスケジュールされている状態を保留する事に相当する。さてINTERRUPTステートメントの形式は，

INTERRUPT ＜エレメント変数＞－＞＜キュー＞

で，その機能は，エレメント変数で指定されたプロセス（このプロセスはス

ケジュールされていなくてはいけない）を指定したキューの先頭に挿入し，インタラプトが解消されるまでこのキューでスケジュールを保留して待っている。キューを指定していてもこのキューは普通のキューとは別の扱いを受け，プロセスの挿入や取り出しはＬＩＦＯ（Last In First Out）ベースで行われる。

(9) RESUMEステートメント

このステートメントは前の INTERRUPT ステートメントでインタラプト状態にされたプロセスのインタラプトを解消する機能を持っている。

ステートメントの形式は，

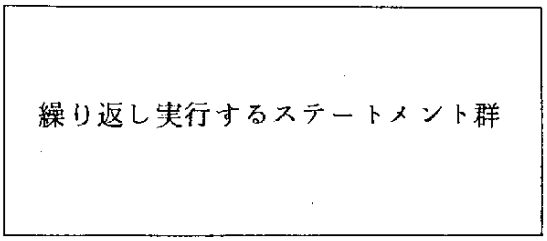
RESUM ＜エレメント変数＞ ＜－＜キュー＞

と書きキューの先頭にあるプロセスのインタラプトを解除し，このプロセスをスケジュールする。スケジュールの仕方は，インタラプト状態にあった分だけ修正を行って，つじつまが合う様，システム側で自動的に行う。即ち，ある時刻にT時間後に実行される様にスケジュールされたプロセスは T_0 時間インタラプト状態にあったら始めの時刻から $T+T_0$ 時間後に実行される様にスケジュールされる。

3.9 繰り返しステートメント

一連のステートメントを繰り返し実行するにはDOステートメントを用いると便利である。DOステートメントには2つの形式があって，1つはFORTRANやALGOLなど一般の言語が持っている，数値によって繰り返し指定を行う形式で，例えば

```
DO LABEL FOR I=1,200,2
```




```
LABEL. LOOP
```

と書くと，　形で囲まれたステートメント群を，始めは I=1 で，次にI=I+2とし，この操作を続けてI>200となるまで繰り返し実行する。ここで LOOPと書かれたステートメントはDOステートメントの繰り返し範囲を指定するもので，かならず書かなくてはならない。この形式の一般型は

DO ＜ステートメントラベル＞ FOR ＜コントロール変数＞=＜初期値＞,
＜終値＞,＜増分＞

でコントロール変数には実数型変数でも整数型変数でも書く事ができる。
又，＜初期値＞,＜終値＞,＜増分＞には算術式を書く事ができる。

第2の形式は一般に汎用言語に無い形式で，セットやキューを構成する全てのメンバーについて何かを行う時などに便利な機能を持っている。

例えばセットSに属しているメンバーを全部取り出すには

```
DO RMV FOR MEN:=FIRST(S),LAST(S),SUCC(MEN);
REMOVE MEN <-S ;
RMV. LOOP;
```

と書く。ここで，MENはあらかじめELEMENTと宣言されたエレメント変数

で1回目はSの先頭メンバーを示し，2回目は函数SUCC(パラメータが指すメンバーのSUCCESSOR を表わす)の値を，以下，毎回SUCCの値を入れてLAST(S)と等しくなるまで対応するLOOPステートメントによって示されるステートメント群を繰り返し実行する。

この形式をエレメントによる繰り返しと呼び，一般形式は以下の様になる。

DO ＜ステートメントラベル＞ FOR ＜エレメント変数＞：＝
＜初期値エレメント＞，＜終値エレメント＞，
＜次のエレメントを決るエレメント式＞；

3.10 統計データの収集

シミュレーションの目的は単にモデルを作ってそれを動かして見るだけでなく，種々の情報を得る事である。離散系のシミュレーション言語では少なからず，このための手段が用意されている。

GPSS では FACILITY，STORAGE，QUEUE などの使用状況を自動的に取り出し，後でレポートを作成してくれる。又，TABLEを定義して必要な情報を TABULATE ブロックでカウントする機能を持っている。

一方 SIMSCRIPTⅡ では，自動的なレポート作成機能は持っていないが，TABULATE ブロックと似た機能を持つTALLYステートメントがあり，あらかじめ定義しておくと，変数の値が変る度に，その情報を取り出しカウントしてくれる。これらの機能を整理してみると次の様に考える事ができる。

(1) データの集収もレポートも自動的にシステムが作成するもの

(2) データだけ変化が起ると自動的に取るもの

(3) データを取り出す事も，レポートを作成する事もユーザが自分で行うもの

(1)は GPSSのFACILITY などの扱いに相当する。これはユーザがいちいちプログラムをする必要がなく便利であるが，反面，レポートの形式が標準化されてしまい，予想以上に使いにくい面も持っている。

(2) は SIMSCRIPTⅡ の TALLYステートメントの機能に相当する。データを取り出すタイミングは指定した変数の値が変った時である。レポートの作成はまったくユーザの手にゆだねられている。

(3)は GPSSのTABLE と TABULATEブロックの機能に当る。

(2)と(3)を較べた場合，データを取り出すタイミングが非常に複雑であったりする場合には(2)では不十分な事がある。

SIMBOL ではこれらの事を考慮した上で基本的には3つの立場を取る様にした。

データをカウントする為のテーブルとして Distribution table と Property table を設け，

1. Distribution tableはデータの分布まで知る必要がある場合に

2. Property tableはデータの平均，分散，標準偏差，最大値，最少値と言った統計量だけ得られれば良い時に

使える様にしている。

a テーブルの宣言

Distribution tableはDTABLEステートメントで，Property tableはPTABLEステートメントで定義をするが，どちらも配列として宣言する事ができる。

1. DTABLEステートメント

Distribution table の宣言は，

・その名前

・配列であれば，そのサイズ

・ランクの幅

・ランクの代表値の最小値

・ランクの代表値の最大値

を与える。例えば

DTABLE HIST＜1，100，10＞；

はHISTと言う名のディストリビュションテーブルを宣言し，〔0，10），〔10，20），〔20，30），…，〔90，100）と言う10個のランクを持つ事を表わしている。

一般には1つのテーブルの宣言は

＜テーブル名＞＜＜ランクの代表値の最少値＞，

＜ランクの代表値の最大値＞，＜ランク幅＞＞

の形をしていてDTABLEの後にカンマで区切って並べればよい。

配列形のテーブルでも同じ様にして

DTABLE AHIST(5)＜0，100，10＞；

これはHISTと同じ形のテーブルが5個定義され個々のテーブルはAHIST(1)，AHIST(2)，などの様にして識別される。

b　PTABLE ステートメント

プロパティーテーブルについてはデストリビューションテーブルの様なランクの指定はまったくいらない。単に名前と配列型であれば，そのサイズを定義するだけである。

例えば

PTABLE　PT，PS(5)；

はPTと言うプロパティテーブルとPSと言う配列型のプロパティテーブルを定義している。

c　データのカウント

データを実際にカウントするのはCOLLECT ステートメントで行う。

このステートメントは丁度GPSSの TABULATEブロックに相当する機能を持っている。このステートメントでは一般に，

COLLECT　＜算術式＞，＜テーブル＞；

の形で記述し，ステートメントが実行されるとその時の算術式の値を指定したテーブル（デストリビューションテーブルでも，プロパティテーブルでも良い）にカウントする。

具体的にはデストリビューションテーブルなら，算術式の値がおさまるランクを探し，みつかればそのランクのカウントを1つ増やし，平均値や，分散などを求める部分のアップデートを行う。

プロパティテーブルでは後半分の処理だけが行われる。

d　レポートの作成とテーブルのリセット

データをカウントしたテーブルに対してレポートを得る為に必要なプロセデュアーをシステムで用意してある。単にテーブルの名前を指定してスタンダードなレポートを作成するプロセデュアー以外に，テーブルの細い情報を取り出し，ユーザが適当なフォーマッティングを行ってレポートを作成する為に用意されているプロセデュアーは各々のテーブルから次の情報を参照する事ができる。

デストリビューションテーブル

・各ランクのカウント

・分布の平均値

・　〃　分散

・　〃　標準偏差

・　〃　最大値

・　〃　最少値

・サンプル数

プロパティテーブル

・分布の平均値

・　〃　分散

・　〃　標準偏差

・　〃　最大値

・　〃　最少値

・データの時間に関する積分値

（これは，$\Sigma V_i (T_i - T_{i-1})$として求めたもので$V_i$は時刻$T_i$にカウントしたときのデータの値で$T_{i-1}$は1回前のカウントをした時刻である）

テーブルにカウントした値を全てキャンセルし，又次のカウントを開始する事が必要な時にはRESETステートメントで行う。

RESETステートメントではキャンセルしたいテーブルの名前をカンマで区切って幾つでも書く事ができ，例えば

RESET HIST, AHIST, PT ;

などと書く。

4　オペレーショナル　エンバイロメント

シミュレーションを行う為には，単にモデルを組立てる為のステートメントだけでなく，モデルの実行を指示したり，モデルの修正をする為のエデットを行ったりモデルのセーブやロードを行う事などを指示する手段が必要である。SIMBOLではこれをコマンドと呼んでいる。

この章はSIMBOLのコマンドと関連したオペレーショナル　エンバイロメントについて述べてある。

4.1　シミュレーションステージとステータス

オンラインシミュレーションの特徴は普通のバッチ処理に於けるシミュレーションと異ってモデルの作成，修正，実行など色々な取扱が随時行える点にある。しかしもちろん技術的な問題から言っても，まったく勝手に何んでも行えるわけではなく適当なルールが設けられている。

インタラクティブに行うシミュレーションの過程は，幾つかのステージと，ステータスに分けて考える事が出来る。ステージと呼んでいるものはSIMBOLに対して，ステートメント又はコマンドのインプットが出来る状態でステータスと呼んでいるのはSIMBOLが何かを行っている状態をさしている。次の図はこれらを整理して示したものである。

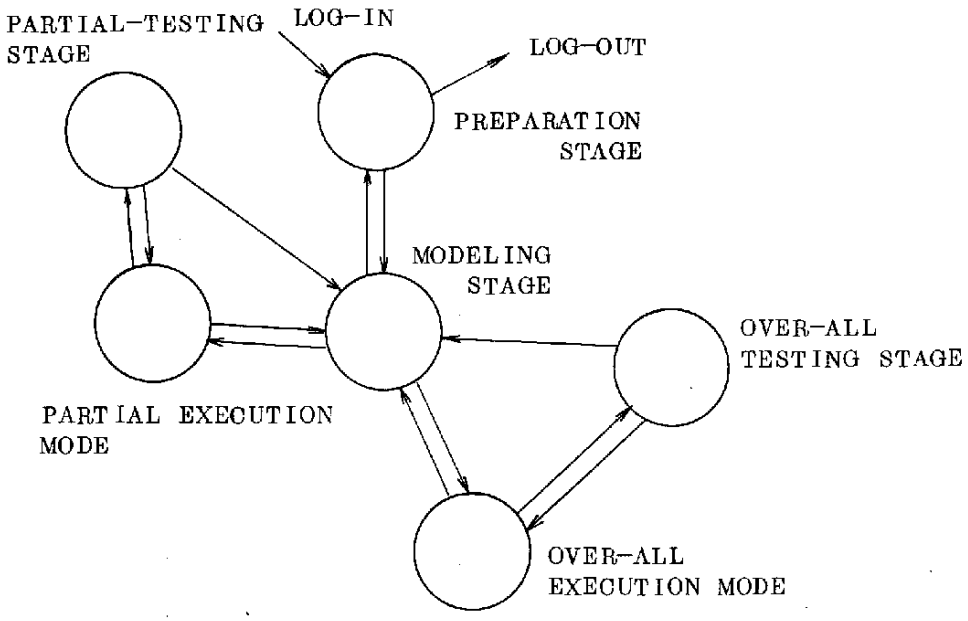


図10　ステージとステータス

1. プリパレーションステージ

ユーザが，タイプライター端末からタイムシェアリングシステムに対してSIMBOLの機動を依頼して，LOG－INが完了した状態である。

このステージでは，直接モデルの作成や実行と関係なくソースファイルのプログラムを消去したり，プログラムファイルの内容を調べたりする作業を行う事が出来る。

2. モデリングステージ

プリパレーションステージでMODELコマンドをインプットするとモデリングステージにステージがある。

モデリングステージは，その名の通りモデルビルディングをインタラクティブに行う為のステージである。新たにモデルを作成する事も，すでに登録してあるモデルを再ロードして，その次をインプットする事も可能である。又，すでに完成したモデルを実行して，その途中のステータスをセーブしたファイルからステータスをロードして，実行の続きを行う事も出来る。このステージはインタラクティブシミュレーションの中心となるステージである。

3. パーシャルテスティングステージ

モデリングステージで作成中のモデルはまだ完成していないが，一部分を実行して見たが，どうも様子がおかしいので，途中で止めて調べたい。この様な時にこのステージで調べる事が出来る。

4. オーバオールテスティングステージ

モデルを実際にシミュレーションクロックを進めながら実行を始めてから，途中で，モデルの様子を見たり，パラメータの値を変えたり，トレースを取ってみたりなどの実行中のモデルとインタラクションを取る為のステージである。

4.2 ステートメントとコマンド

ユーザがSIMBOLシステムにキャラクターディスプレイからインプットする単位をラインと呼んでいる。ラインの構成は2通りあって，ラインナンバーから始まるものと，%記号で始まるものである。

前者はステートメントをインプットする時の形で，ステートメントにはかならずラインナンバーと呼ばれる数値を対応付けて，ステートメントを検索する時のキーとして使われる。後者はコマンドをインプットする時に使われる。コマンドとステートメントの相異は，ステートメントはその場で，コンパイルされオブジェクトが作成されるが，後に指示があるまでその実行を保留しているのに対してコマンドは，その場で実行される点である。従って，ステートメントはモデルそのものを構成するが，コマンドはモデルの実行や中止を指示したり，プログラムのエディットを行ったりなど直接モデルに組込まれないで，その取扱いをする為のものである。

4.3 ステートメントインプットとエディット機能

ステートメントは原則としてキャラクターディスプレイからインプットする。キャラクターディスプレイの画面は次の図の様に区分されていて上半分にはインプットし終ったステートメントが表示され，下半分に現在キーインしているステートメントやコマンドがそのまま表示される。

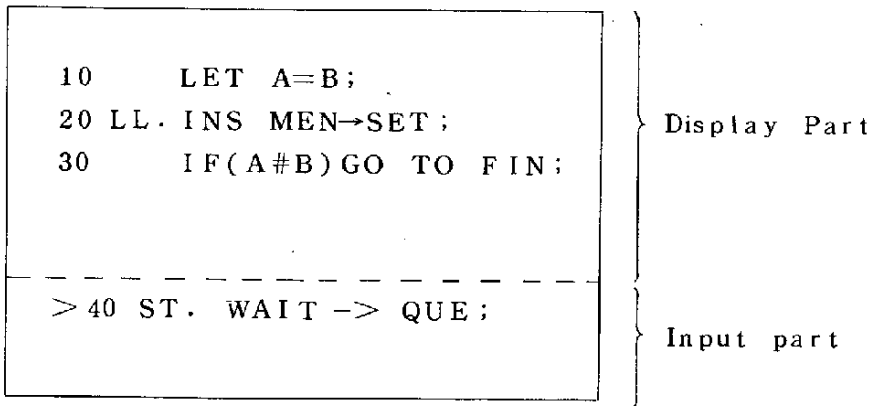


図11 ディスプレイ画面

ステートメントをインプットする時にはまず，ラインナンバーを入れる。その後に，ステートメントラベルが必要ならこれをキーインし，以下ステートメント本体を続けてセミコロンでしめくくり，SENDキーを押す。

ステートメントインプット手順は，ラインナンバーが増加する様に行う必要は特になく，任意の順番でインプットしてさしつかえない。インプットしたステートメントにシンタックス・エラーがなければ，ディスプレイ画面の上半分にラインナンバーの順がそろう位置に表示される。

インプットする順番が任意であってもシステム内部ではインプットされたステートメントをラインナンバーの順にソートしてセーブしておく，従ってプログラムの順番はラインナンバーによって支配される。

インプットしたステートメントにエラーが見つかると画面の一番下にエラーメッセージが表示される。エラーのあったステートメントはそのまま画面に残っているので，キャラクターディスプレイのハードウェアが持っているエディット機

能を生かして，一部分の修正も簡単に出来る。

再び SEND キーを押せば，全体をキーインする事なく，エラーの修正が出来る。

4.4 ラインナンバー

SIMBOLにはメインプログラム，アクティビティプログラム，プロセデュアー，関数と言ったプログラム要素があるが，ラインナンバーは次のルールで与えられる。

1. メインプログラムと一連のアクティビティはひと束めにして通し番号で与える。従ってこれらのステートメントに同じ番号を与えられない。
2. プロセデュアーや関数は個々にラインナンバーを与える。従って異るプロセデュアーどうしでは，同じラインナンバーを持ってステートメントが在ってもかまわない。

ステートメントのインプットについてはラインナンバーはユーザが自分でキーインする事になっているが，もし，直前にインプットしたステートメントのセミコロンの後に英字のLをキーインすれば10だけ増えたラインナンバーをシステム側から表示してくれる。後はステートメントをインプットするだけで，手間をはぶく事が出来る。

4.5 プログラムインプット手順

SIMBOLに依るモデルはメインプログラム以外にアクティビティプログラム，プロセデュアー，関数と言ったプログラム要素によって構成されている。もちろん，メインプログラム以外は必要なければ省略してかまわない。

SIMBOLの特徴の1つは，こういった幾つかのプログラム要素をコンカレントにインプットして行ける事である。即ち，1つのプロセデュアーを組み終った後でなくては，次のプロセデュアーをインプット出来ないと言った事はなく，始めは，アクティビティAのステートメントを，次にはアクティビティBのステートメントをと，まったく任意の順序でステートメントをインプットして行けるわけである。シミュレーション用のプログラムではお互のプログラム要素が複雑に関連し合っているので，この様に出来る事は，プログラミングをする上で，大変便利である。

プロセデュアー間とか，メインプログラムとプロセデュアーなど個々のプログラム要素の間ではお互に同じラインナンバーを持つステートメントがあってかまわない事もあって，現在インプットしているステートメントはどの要素に属すのかを，ハッキリ区別しなくてはならない。

そこでインプットモードと言った考え方を導入している。

例えば，プロセデュアー SUB のインプットをしている時にはインプットモードが SUB であると言う。インプットモードは，コマンドに依って変える事も出来るがそれ以外に次の場合に自動的に変更される。

1. PROCEDURE ステートメントがインプットされると，そこで定義されたプロセデュアー名のインプットモードが新しく設定され，そのモードに合せられる。

2. タイプを前に付けた PROCEDURE ステートメント，これは即ち関数の宣言である。この場合もプロセデュアーと同じ扱いを受ける。

4.6 未完成なモデルの実行

EXECUTE コマンドはモデルの部分的な実行を行う為のコマンドである。まだ完成していないモデルでも部分的なチェックを行っておきたい時にこのコマンドを使うと便利である。このコマンドで行うチェックは，あくまで未完成なモデルに対して行うもので，シミュレーションクロックを実際に進めながら，イベントの動きを追う様な事は対象としていない。

従って，スケジューリングと関連したステートメントなど，一部のステートメントは無視して実行される。

EXECUTE コマンドのパラメータとして，実行範囲をラインナンバーで指定する。例えば，

```
%EXECUTE  10, 100;
```

ではラインナンバー10のステートメントから実行を始めて，10から100の範囲外のステートメントを実行したら打切る事を指示している。

4.7 モデルの実行

STARTコマンドはシミュレーションクロックを進めながらのシミュレーションを指示する為のコマンドである。このコマンドでは実行直前にモデルのグローバル変数のクリアーとかシステム変数，スケジュールテーブルなどの初期設定などを行ってからモデルの実行を開始する。

シミュレーションの打切りは実行中にSTOPコマンドで行う事も出来るが，あらかじめSTARTコマンドのパラメータに指定しておく事も出来る。

パラメータでは，

1. 時刻によって
2. イベント数によって

などの打切り条件を書く事が出来る。

スタートコマンドの特別な機能として初期設定終了後に一度PAUSEする機能と，グローベル変数のクリアーを行わずに実行を開始する機能を持っている。これらはグローバル変数を特別の値にセットしてからシミュレーションランを開始する為の機能である。

4.8　モデル実行中のインタラクション

SIMBOLのモデルとはその実行中に，いつでもクイットを掛けてインタラクションを取る事が出来る。キャラクターディスプレイ端末はハードウェア的にもオペレーティングシステムの元でもクイット機能を持っていないので，ペアーで使っているタイプライター端末のクイットキーによって行う。

シミュレーションクロックを実際に進めながらモデルを実行している最中にクイットが掛けられると，SIMBOL システムステージは，直ちにオーバオールティステイングステージに移される。このステージに移されると，ユーザは，モデルのステータスを調べる為に，DISPLAY コマンドを入れたり，LET コマンドでグローバル変数やローカル変数の値を変えたり，スケジューリング関係のコマンドを使ってスケジュールテーブルを変更したりする事も可能である。もし，この状態からモデルの実行を何回も繰り返し実行する必要があったらCHECKコマンドを使ってモデルのステータスをセーブしておき，次に実行する時には，RESTART コマンドを使えば良い。

このステージの処理が終って，そのまま実行を続ける場合にはGO コマンドをインプットすれば良い。

4.9 ステータスモニタリング

実行中のモデルの様子を見る事をモニタリングと呼んでいる。モニタリングには大きく分けると2通りの方法があって，第1は，実行中のモデルにクイットを掛け，モデルの実行を一時中断した状態で，モデルの様子を調べるものである。第2はトレースに依る方法で，この場合にはモデルの実行を中断する必要はない。第1の方法では，この為に特別に用意されたコマンドやステートメントがある訳でなく，DISPLAYステートメントやWRITEステートメントの前に%記号を付けて，コマンドとして用いればモデルの様子を全て参照する事が出来る。一方，第2の方法，つまりトレースは，その為にTRACEコマンドがあり，このコマンドを使って種々のトレースを行う。

4.10　トレース機能

SIMBOLのトレース機能はトレース種類が豊富であるだけでなくその取扱いが自由である点で特徴がある。その為にトレース指定はトレース対象の指定と実際にトレースを開始したり打切ったりする為の指定が分離して行える様に成っている。

a　トレース種類

SIMBOLで行うトレースの種類は大きく分けると2つある。

第1は実行シーケンスのトレースで，これはプログラムの実行がどの様な順で行われているかを調べる為のものである。SIMBOLのプログラムはアクティビティやプロセデュアーなどによって構成されているが，プログラムの実行シーケンスについては，個々のプロセデュアーやアクティビティプログラムだけのローカルシーケンスが知りたい場合もあるし，コンカレントに実行されるプロセスについてプロセス間の実行手順，つまりイベントの発生順について知りたい場合もある。その為実行順についてはローカルシーケンスのトレースとイベントシーケンスのトレースと2つに分けて行える様になっている。

ローカルシーケンスのトレースでは，プログラム単位を選んでトレースしたり，1つのプログラム単位中でも一部分や，特定ステートメントだけを対象としてトレースする事が出来る。

第2はモデルステータス変化のトレースで，これはモデルのグローバル変数やシステム変数の値が変るとその値を表示するものである。

トレース対象とする変数の数は1つだけとは限らず幾つでも指定する事が出来る。

b　トレースの実行

実際にトレース情報を取り出す為の指定は別に行い，トレースの開始，終了は任意の時点で行う事が出来る。一度トレース指定を行っておけば，取り消されるまで有効で，その間トレースの開始，終了の指定を何回でも繰り返す事が出来る。

c　トレース情報の表示

トレース情報は原則としてキャラクターディスプレイ端末に表示される。キャラクターディスプレイ画面に表示出来る情報には限りがあるので、表示スピードが速すぎると人間が見る事が出来なくなってしまう。そこでトレース情報を画面に順々に表示していき画面が一ぱいになったらモデルの実行を一時中断しユーザから指示のあるまで待っている。

画面情報を見終って次を実行したければGOコマンドをインプットする。以下この様な手順が繰り返される。もしいちいちディスプレイに表示する必要がなく情報量も多く後でゆっくり見たい場合にはセンターのプリンターに出力する事も出来る様にしてある。

4.11　プログラムのコンパイル

一応完成したプログラムは実行スピードを上げる為にバッチ処理言語で行っている様な方法でコンパイルする事が出来る。これはCOMPILEコマンドを使って行う。このコマンドは，メインメモリーにあるプログラム，又は，セーブファイルにセーブされているソースプログラムのどちらかを対象とし，指示のあったプログラムをコンパイルして，ユーザズライブラリーファイルに登録する。
（図 13 参照）

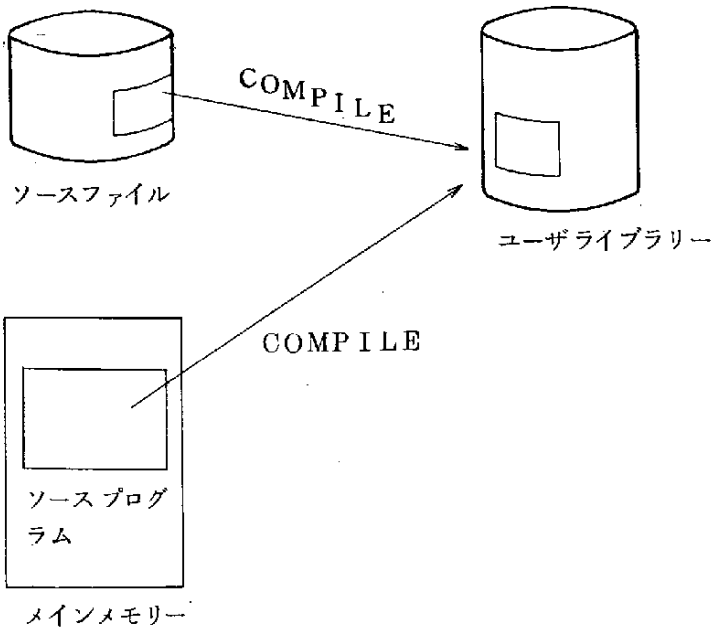


図 13

コンパイルの対象とするプログラムには次の制限がある。

1. プロセデュアー，又は関数を単独で

2. メインプログラムと一連のアクティビティプログラムをまとめて，以上

以上の様なプログラムがコンパイルされて出来たオブジェクトプログラムの形式はFACOM 230／60 に於ける相対形式プログラムと同じ形式になっている。その為，FACOMのオペレーティングシステムが持つ，リンクエディターLIEDやライブラリーエディターＬIBE による処理が可能である。

4.12　プログラムのセーブとロード

SIMBOLのユーザに対しては，個々にソースプログラムと相対形式プログラムをセーブする為のファイルが割当てられていて，それぞれ，ソースプログラムファイル，ユーザズライブラリーと呼んでいる。

ソースプログラムファイルには，端末からインプットしたプログラムをSAVEコマンドでセーブする事が出来る。現在の所SAVEコマンドでセーブするプログラムはソースプログラム部分だけでインクリメンタルコンパイルがされたオブジェクト部分は対象としていない。そこでセーブの対象は，特にプログラム単位とは限らずプログラムの一部ステートメントだけをセーブする事も出来る。セーブの際にはSAVEコマンドのパラメータに名前を指定し，ソースプログラムファイル上でその名前に依って識別できる様に成っている。途中までインプットしたプログラムをセーブし，後日この続きを行う時には，LOADコマンドを使ってソースプログラムファイルからメインメモリーにロードしなくてはならない。このコマンドでロードされたソースプログラムは，丁度，端末からインプットされたのと同じ効果を持っている。

4.13　アルゴリズムの変更

幾つかのアルゴリズムがあって，この内，最適なものを見つける為にシミュレーションを行うと言った事は良く行われる。例えばジョブショプ問題で，デスパッチングルールが幾つかあってその内もっとも良いものを選ぶとか，タイムシェアリング・システムのシミュレーションでスケジューリング・アルゴリズムを決定する問題等である。

ＳＩＭＢＯＬではプログラムのリンクをランタイムに行う，いわゆるダイナミックリンクを行っている。その為，ランタイムに論理的関係を乱さない限り，プロセデュアーの置き換えが可能である。そこで，前記の様な問題に対して，個々のアルゴリズムに対応してプロセデュアーを作成しておき，それぞれのケースについて全体のリコンパイルをしなくても，アルゴリズムを変更しながらその比較をする事が出来る。（会話型の言語であるから，なにもランタイムに変えずとも，プログラム自体を修正すれば良い様なものであるが，内部的にはリコンパイルされる事になって，やはり手間がかかってしまう。）

モデルを実行している時に呼び出そうとした函数やプロセデュアーが無いとＳＩＭＢＯＬは見つからなかったルーチンの名前を表示してユーザの処置を求める。もちろん，ユーザのミスでプログラムを入れ忘れたのであれば，一度実行を中止して，あらためてプログラムのインプットをしなくてはならない。しかし，たまたま呼び出す側の名前を間違えていてＦＵＮＫと入れていたが実はＦＵＮＣと言う名前であった，という場合には，ＵＳＥコマンドを使って，

　％ＵＳＥ　ＦＵＮＫ　ＡＳ　ＦＵＮＣ；

とインプットして実行を続けることが出来る。

ＵＳＥコマンドを，この様にミスがあった時だけに使うのではなく，もっと積極的に使う事が出来る。つまり，あらかじめ幾つかのプロセデュアーを用意しておき，呼び出し側は適当な名前を使っておく。もちろん，この様な名前のプロセデュアーが存在していないので，実行を開始すると，リンクフォールトが発生する。そこで，第１のプロセデュアーの名前をＵＳＥコマンドで与えて実行する。

次に別のプロセデュアーを組込んで実行する時には，実行直前にUSEコマンドで組込みたいプロセデュアーの名前を今使ったものと取換えておく。以下USEコマンドで順番に名前を変換して行けば，それぞれのシミュレーションランを行う事が出来る。

Ⅱ. SIMBOLプロセッサー

Ⅱ. SIMBOLプロセッサー

1　インプリメント

SIMBOLはFACOM230／60のTSSモニター，Monitor－V(以下M－Vと略記）のもとで稼動するシステムで，プロセッサーはデマンドジョブとして起動される。次の図はSIMBOLがインプリメントされたFACOM230／60のハードウェアー構成図である。

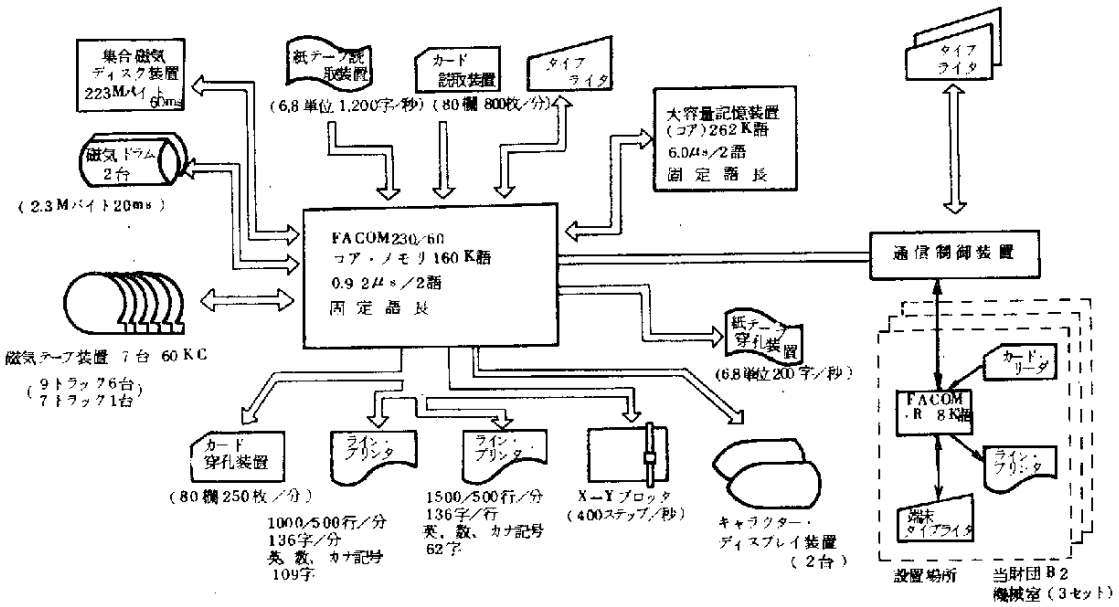


図14　ハードウェア構成図

SIMBOLは，端末として普通のタイプライター端末とキャラクターディスプレイを組合せて，2台一組にして使っている。プログラムインプットがエディット・モデルの実行などほとんどはキャラクターディスプレイを使って行うが，たまたまこのシステムに於けるキャラクターディスプレイは端末としての資格を持っておらず，ジョブの起動や実行中のQuit機能を持っていない。そこで，タイプライター端末から，これを行い，合せてキャラクターディスプレイのハードコピー用にも使っている。将来端末としての資格を持つディスプレイが開発されしだいタイプライター端末の使用を取りやめたいと思っている。

2　プロセッサーの構造

SIMBOLのプロセッサーは幾つかの大きなモジュールと，サブルーチン群から出来ている。そのうち主なものを挙げると，

(1)　コマンドディスパッチャー

(2)　ステートメントプロセッサー

(3)　各種ステートメント処理サブルーチン

(4)　算術式処理サブルーチン

(5)　エレメント式処理サブルーチン

(6)　各種コマンド処理モジュール

(7)　エディター

(8)　コンパイラー

(9)　リンクローダー

(10)　エグゼキュータ

(11)　スケジューラー

(12)　その他のランタイムシステム

などである。以下にこれらの機能を簡単に述べておく。

(1)　コマンドディスパッチャー

これはSIMBOLシステムの中核を成すモジュールで，その機能を列挙すると

①　キャラクターディスプレイから，ユーザがインプットするラインを読み込んで，ステートメントか，コマンドかを識別する。

②　ステートメントである事が解ると，ステートメントプロセッサーにコントロールを渡す。

③　コマンドである事が解ると，各コマンド処理モジュールへのディストリビューションを行う。

などである。

(2) ステートメントプロセッサー

ステートメントプロセッサーは，いわゆるインクリメンタルコンパイルの制御モジュールである。このモジュール自身，ステートメントを解読して，オブジェクトコードを作成しないが，各種ステートメント処理サブルーチンのうち，必要なものを呼び出して，オブジェクトコードの作成を依頼する。出来たオブジェクトを，ステートメントプロセッサーが管理するオブジェクトエリアにストアし，必要なチェイニングを行う。

又，このモジュールは，ソースエリア（ソースステートメントをセーブして置く為のエリア）の管理も行っており，簡単なエディト機能も備えている。その為にラインナンバーの管理もこのモジュールが行っている。

(3) 各種ステートメント処理サブルーチン

これは全体で1つのサブルーチンとなっているのではなく，ステートメントごとに対応して1つずつサブルーチンが存在するサブルーチン群である。

それぞれのサブルーチンは，ステートメント本体（インプットされたラインから，ラインナンバー，ステートメントラベルを除いた部分）をもらい，そのオブジェクトコードを作成する機能を持っている。

コンパイル作業に必要があれば算術式処理サブルーチンとか，エレメント式処理サブルーチンを呼び出す事もある。

これらのサブルーチンは単にステートメント・プロセッサーから呼ばれるだけでなく，コマンドディスパッチャーから直接，呼び出される事もある。これは，LETステートメントの様にコマンドとしても使えるものが，%記号を付けてコマンドとして使われると，オブジェクトコードを作成し，その場で実行しなくてはならない。どちらの場合でもオブジェクトコードは決ったワークエリアに作り出されるが，ステートメントの場合は，オブジェクトエリアで動くべくアドレス調整がされ，コマンドではワークエリアで，そのまま動く様になっていなくてはいけない。従ってこれらのサブルーチンは，指定された場所で動くオブジェクトコードが作成出来る機能も備えている。

(4) 算術式処理サブルーチン

このサブルーチンは各ステートメントの中に出てくる算術式の処理をするもので，与えられた算術式からそれに相当するオブジェクトコードを作成する為のものである。従って，このサブルーチンは各ステートメント処理サブルーチンから必要な時に呼び出される。

(5) エレメント式処理サブルーチン

このサブルーチンもその名の通り，エレメント式の処理をする為のもので，パラメータとしてもらったエレメント式に対してオブジェクトコードを作成する機能を持っている。しかしエレメント式の処理と言ってもエクスターナル・リファレンス等の処理は一部算術式処理サブルーチンが受持っている部分もあり，主に生成式の処理を行っている。生成式の中にはパラメータ部に制限した形で算術式が書けたりする為に，このサブルーチンが算術式処理サブルーチンを呼び出しその処理を任せている。

(6) 各コマンド処理モジュール

EDIT，COMPILE，SAVE，LOAD などのコマンドとしてしか使われないものに対して，1つずつ設けられたモジュールで，コマンドパラメータの解読とコマンド機能の処理を行う。EDITコマンドや，COMPILE コマンドの様に自分自身では処理せず，エディターや，コンパイラーを呼びだし処理を任せる場合もある。

(7) エディター

エディターはメインメモリーにセーブされているソースステートメントのエディトを行うためのモジュールである。ソースステートメントがセーブされているエリアをソースエリアと呼び，フリーストレッジ（Free storage）として管理されている。このエリアに登録されているステートメントは，ラインナンバーで検索出来る様にその為のテーブルが設けられている。エディターはステートメ

ント単位にエディットを行い，個々のステートメントの一部だけを修正したりする機能は持っていない。それは端末として使っているキャラクターディスプレイ自身がハード的に持っているエディット機能が使えるのでその必要がないからである。

エディターはステートメント・プロセッサーや，EDITコマンド処理モジュールから呼び出されるが，それぞれの場合によって一部処理が異っている。

(8) コンパイラー

端末からラインごとにインプットされたステートメントは，その都度コンパイルされオブジェクトコードが作成されるが，このコンパイラーはこの処理と直接関連はない。コンパイラーはCOMPILEコマンドがインプットされた時にCOMPILEコマンド処理モジュールから呼び出されると，ソースエリア又はソースプログラム・セーブファイルにあるプログラムを，普通のバッチ処理で行うコンパイルと同じ様にしてオブジェクトプログラムを作り出し，プログラムファイルに書き出す。

コンパイラーモジュールではインクリメンタルコンパイルで使われるモジュールを出来るだけ共有し，プログラムサイズの縮少と，開発工数の短縮を計っている。

(9) リンクローダー

このモジュールの機能は，前記のコンパイラーに依って作り出された，プログラムファイルか，SIMBOLシステムが用意するシステムライブラリーから必要なプログラムを探し出し，メインメモリーにロードする事である。

ロードするプログラムは相対形式プログラムで実際に動き得る様にする為にはアドレス調整の必要がある。リンクローダは次の処理の他に，プログラムテーブルの管理も行っている。プログラムをロードすると，プログラムテーブルに入口番地を登録し，プログラム間のリンクを付ける。

この処理については後述してあるので，ここでは省略する。

(10) エクゼキュータ

エクゼキュータはスケジューラと共にモデル実行をコントロールする為のモジュールである。この両者の機能的な相異は，エクゼキュータはローカルなプログラムシーケンスに従って実行手順を決めるのに対して，スケジューラはコンカレントに実行されるプロセスの流れを時間軸にそってコントロールする点である。

エクゼキュータの主な機能を次に挙げてみると

① 上記のプログラム実行シーケンスの決定と制御

② トレース処理

③ 統計データの収集機能

④ パーシャルエクゼキューションのコントロール

などである。尚エクゼキュータが行う処理については，処理別に詳しく後述してある。

(11) スケジューラー

スケジューラーはランタイムシステムの一部として考える事が出来る。インタラクティブなシステムで，ランタイムシステムと言った呼び方が適当でないかもしれないが，SIMBOL のモデルプログラムは SIMBOL プロセッサーがまったく介入せずに，モデルだけが一つのプログラムとして実行する事が出来る様に成っていて，その場合にはオブジェクトプログラムの中で，スケジューラを呼び出すので，この様な呼び方をしている。

スケジューラの機能は，プロセスの実行順序のコントロールが主なものであるが，トレース処理の一部も受け持っている。スケジューラについてはエクゼキュータと同様，後に詳しく述べてあるのでここでは省略する。

3　ステートメント処理

ＣＲＴディスプレイからインプットされたラインが数字で始っていれば，その数字はラインナンバーでそれ以下のストリングはステートメントである事がわかる。コマンドディスパッチャーがこの事を識別すると，ただちにステートメントプロセッサーへコントロールを渡す。

ステートメントプロセッサーはまずラインナンバーだけ取り出し，同じものか，すでに登録されているか否かを調べる。もしあればそのステートメントを表示しユーザーの処置を待つ。もしない事がわかれば，ステートメントのコンパイル処理に入る。ステートメントプロセッサーは，後でエディトをしたり，コンパイルをする時の為にソースステートメントをそのままのイメージでメインメモリーのソースエリアに登録しておく。

ステートメントのコンパイルはステートメントプロセッサーがステートメントの種類を識別した後に各ステートメントごとに用意された処理サブルーチンを呼び出しコンパイル処理をまかせる。

これらサブルーチンは共通に使うワークエリアにオブジェクトコードを作成し，ステートメントプロセッサーにコントロールを戻す。ステートメントプロセッサーはオブジェクトコード以外に必要な情報を付けて，１つのレコードとしオブジェクトエリアにセーブし，すでに登録されているこの様なレコードとチェイニングする。

１ステートメントのコンパイルが終了するとステートメントプロセッサーはコマンドディスパッチャーにコントロールを戻し，次の入力を持つ，次に入力されたものが，ステートメントであれば再びステートメントプロセッサーにコントロールが渡るがコマンドであればコマンドディスパッチャーが自分で処理をする。以上の関係を図にまとめて次に示しておく。

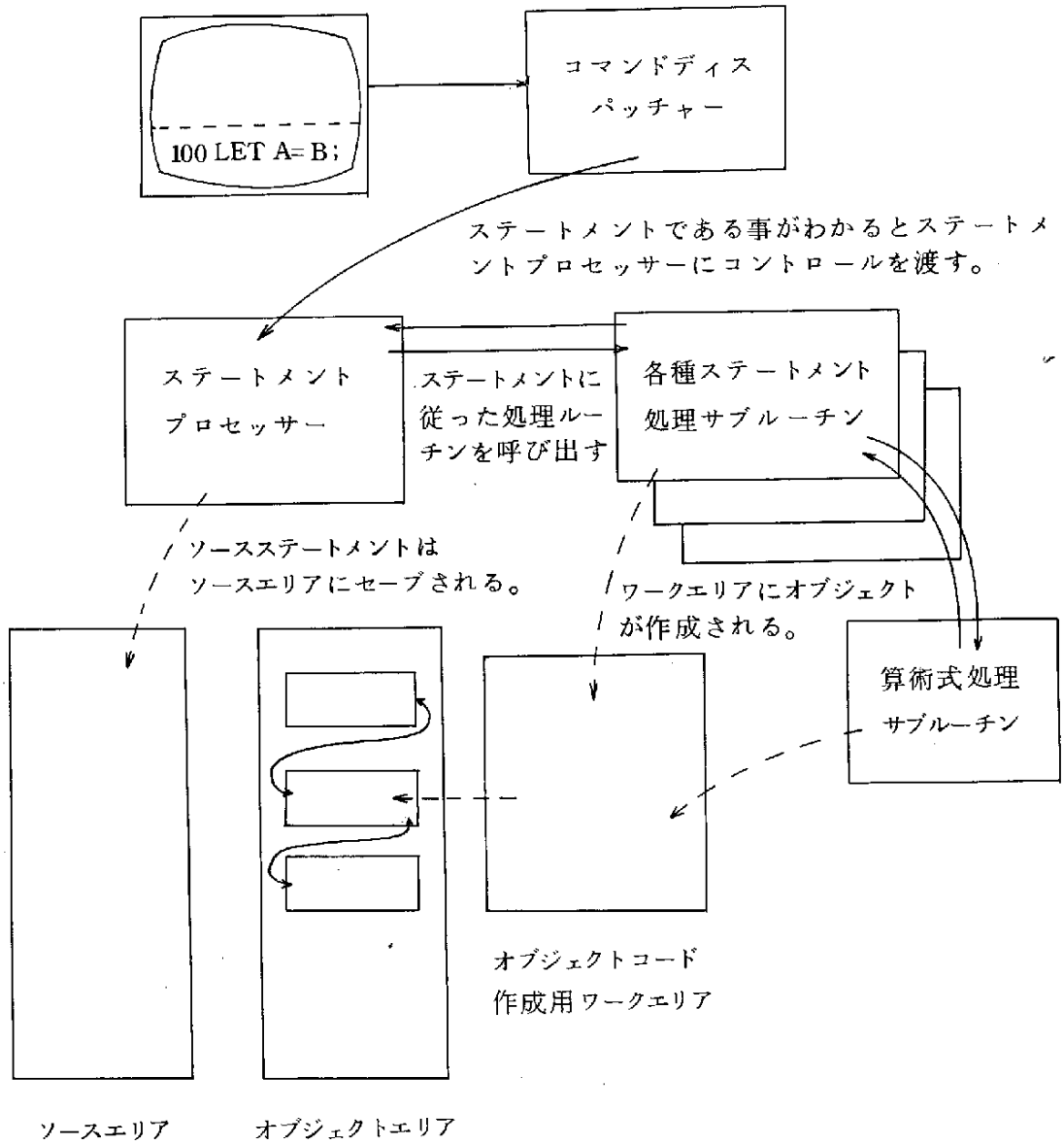


図15　ステートメント処理の概略図

4　コマンド処理

ラインの先頭が%記号で始まるものはコマンドである。コマンドディスパッチャーが，インプットされたラインがコマンドである事を判別すると自分自身でこれを処理する。自分で処理すると言っても，個々にコマンドに関連する処理はそれぞれの処理モジュールにコントロールを渡し，完全に任してしまう。まずコマンドディスパッチャーは，コマンドの種類を調べるが，次の２つのケースが考えられる。

1. EDITやCOMPILEなどの様にコマンドとしてだけ使われるもの
2. LET，SCHEDULE，INSERTなどの様にステートメントとしてもコマンドとしても使われるもの

これら２つのケースによってコマンドディスパッチャーは別の処理を行う。

・　第１のケース

この場合には，それぞれのコマンドに対応してコマンドパラメータの解読と，コマンドの機能に相応する処理を行うモジュールがあるので，コマンドディスパッチャーはコマンドに従って，その処理モジュールにコントロールを渡す。個々のモジュールではパラメータの解読を行い，その場で，指示に従った処理を行う。SIMBOLのコマンドはサブコマンドを持つ事が許されていて，例えばEDITコマンドではLIST，DEL，などと言ったサブコマンドを持っている。この様なサブコマンドの処理もコマンドディスパッチャーではなく個々のコマンドモジュールが一切の処理を行っている。

・　第２のケース

第１のケースと異る点は，第１のケースではオブジェクトコードを作成したりしないのに対して，この場合には一度，オブジェクトコードが作成される事である。コマンドディスパッチャーがこのケースである事を知るとステートメントプロセッサーが呼び出して，ステートメントのコンパイルに使ったサブルーチンを

直接呼び出しオブジェクトコードをワークエリアに作成する。ステートメントプロセッサーから各ステートメントの処理サブルーチンが呼ばれてオブジェクトコードを作る時には，物理的に同じワークエリアであっても，オブジェクトエリアで動く様にアドレス調整したものが作られており，ワークエリアでランする事は出来ない。

一方，コマンドディスパッチャーから呼ばれた場合には，ワークエリアでラン可能なオブジェクトコードが作られる。オブジェクトコードが出来るとコマンドディスパッチャーは直接，このオブジェクトにコントロールを渡し実行させコマンドとして使われた時の機能をはたす。

5　インクリメンタルコンパイレーション

キャラクターディスプレイからインプットされたラインは，%記号が付いたコマンドであるか，ライン番号から始まるステートメントであるかに従って，コマンドデスパッチャーが自分で処理すべきかステートメントプロセッサーにコントロールを渡すべきかを判断して処置を行う。

インプットされたものがステートメントである事がわかると，後はステートメントプロセッサーによって処理される。ステートメントプロセッサーはステートメントの種類に応じて，対応する処理サブルーチンを呼び出し，オブジェクトコードを作成する。CRTディスプレイ端末からインプットされるステートメントに対しては，いわゆるインクリメンタルコンパイレーションを行っている。即ち，ラインがインプットされると，それをセーブしておき，プログラム全体のステートメントが出揃ってからまとめてコンパイルするのではなく，一ステートメントごとにコンパイルをし，オブジェクトコードを作成する方法である。一般に，この様なコンパイルではマシンコードをオブジェクトして出さずに，ランタイムにインタプリートしながら実行する方法が取られていたが，最近は，なるべく，マシンコードに近いオブジェクトを出す方法が用いられる様になって来ている。

SIMBOLの場合にも，実行スピードは出来るだけ速くする必要があるのでほとんどマシンコードのオブジェクトコードを作り出している。

一つのステートメントに対応するオブジェクトコードはメモリーの連続したエリアに作り出されるが，一連のステートメントがかならずしも連続したエリアに作り出されるとは限らない。

一つ一つのステートメントに対して作り出されたオブジェクトコードがまとまって，いわば1つのデータブロック（これをコードブロックと呼ぶ）の形をしており関連するステートメントがお互にポインターでチェインされる事によって1つのプログラムが出来上っている。

今ステートメントについて，ロジカルシーケンス，フィジカルシーケンス，イン

プットシーケンスと言ったシーケンスを考えてみる。

ロジカルシーケンスと言うのは，ステートメントが実行される順序，フィジカルシーケンスは，ラインナンバーに従った順序で，1つ1つのプログラム（例えばプロセデュアーとかアクティビティ）に於けるステートメントのならびを示す順序である。又，インプットシーケンスと言うのは，SIMBOL でプログラムのインプット順は，1つのプロセデュアーをインプット完了した後に別のプロセデュアーをインプットしなくてはならないと言う事はなく，プロセデュアーやアクティビティなど平行してインプットしていく事を許している為に，この様なインプットシーケンスを問題としなくてはならない。以上のシーケンスに従って言うなら，ラインごとに出来たコードブロックはフィジカルシーケンスに従ってチェインされている。

1ステートメントがコンパイルされるとコードブロック以外に，このステートメントの種類や，トレース処理に使うスイッチ類，使われている変数名など，コンパイラー自身とか，エクゼキュータが処理の為に必要な情報を書き込んだ，ステートインフォメーションと呼ばれるブロックが作成される。コードブロックとステートメント・インフォメーションはお互にポインターでチェインされている。

5.1 オブジェクト作成の基本方針

インクリメンタルコンパイレーションの結果作り出されるオブジェクトは，SIMBOL ではインタープリティブ形式と呼んではいるが，実際これを実行する際にシンボルテーブルを参照したり，中間形式になっているオブジェクトといちいち解読しながら実行する方法は取っていない。コンパイル時や，実行時の融通性をそこなわない程度に，マシン・インストラクションに近いオブジェクトを作る様にしている。

次にオブジェクトコードを作成する上での基本的な方針を列挙してみると，

- ○ 1ステートメントごとにコンパイルし，1つのオブジェクトを作り出す。
- ○ 原則として1ステートメントを実行するごとにエクゼキュータへコントロールを戻し，エクゼキュータが次に実行するステートメントにコントロールを渡す様にしてある。
- ○ 変数に対しては直接アドレスに変換されていて，シンボルテーブルを経由する方法を取らない。
- ○ 算術式や論理式，エレメント式に対しては完全にマシン・インストラクションにおとしてしまう。従ってランタイムにオペレータを解読しながら実行する方法を取らない。
- ○ ステートメントラベルの参照は直接行わず，ラベルテーブルを経由する。
- ○ プロセデュアー呼び出しや函数呼び出しについては，直接リンクせず，プログラムテーブルを経由して行う。

などである。

5.2 変数に対する処理

SIMBOLで扱う変数には，コンパイルタイムにアドレスを確定出来るものとランタイムでなくては実際のアドレスが対応づけできないものがある。グローバル変数は，1つの変数名に対して1つのメモリーセルが対応するのでコンパイルタイムに決定してしまう事が出来る。しかしローカル変数は，それを含むプロセスやブロックが実際に作成された時に始めて，メモリーが割当てられるし，1つの変数名に対しては1つ以上のメモリーが対応するのが普通である。従ってグローバル変数と同じ様にしてアドレスを決める事は出来ない。ローカル変数の処理についてはプロセスやブロックの処理及び，スケジューラの処理とも関連するので後述するのでここでは一応省略する。ただ，アクティビティプログラムやブロック宣言で定義されたローカル変数には相対番地を与える事によってコンパイルタイムに，この変数と関連するステートメントのオブジェクトコードが作成出来る点だけ述べておく。

又，プログラムで使う変数については原則として，使う以前に定義されていなくてはならない事になっている。もし定義しないで変数を使うとその時点でエラメッセージを出し，変数の定義を要求する。従ってステートメントをコンパイルする時には，かならず，変数は定義済みとして扱う事が出来る。

5.3　ステートメントラベル処理

ステートメントラベルの場合は変数の時と違って，定義される以前に参照されるケースが起り得る。その為に，例えばGO TOステートメントに対するオブジェクトを出すのに直接，相手ラベルを定義しているステートメントへのジャンプ命令を出すわけにゆかない。そこでラベルテーブルを作り，新らたにラベルが出現すると，定義するか，参照するかにかかわらず，そのラベルをテーブルに登録し，オブジェクトコードにはラベルテーブルへのジャンプ命令を出しておく。一方ラベルテーブルの側には，もし，そのラベルがまだ未定義であるなら，エラールーチンへのジャンプ命令を入れておく。従って未定義のままプログラムが実行されれば，エラールーチンへ飛込んでくる事になる。ラベルの定義がされると，そのラベルを定義しているステートメントをアドレスを書き込んでおき，正常に実行出来る状態となる。どちらにしても，ラベルを参照するステートメントのオブジェクトとしてはラベルテーブルへのジャンプ命令が出せる事になる。ラベルテーブルの扱いはエクゼキュータの処理と密接な関連があるので，後章（９章）に譲る事にし概念的な図だけを次に示しておく。

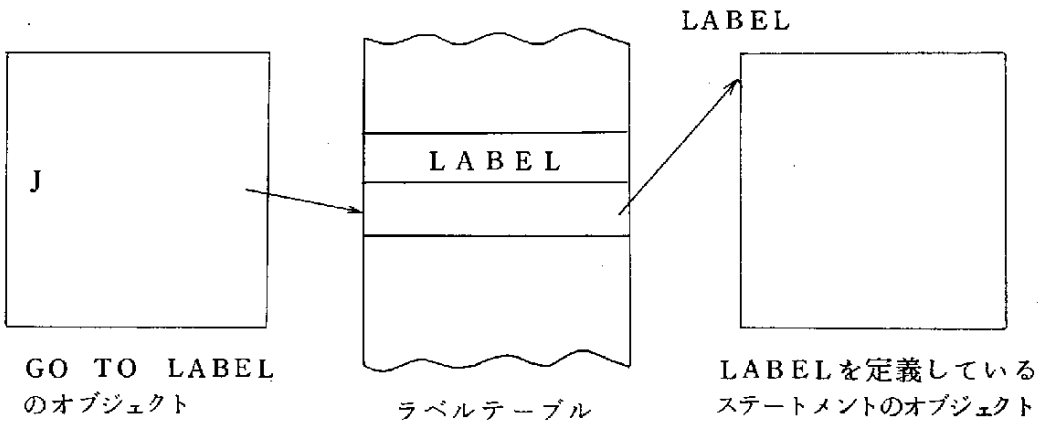


図16　ラベルテーブルとオブジェクトとの関連

5.4　プロセデュアーと函数の呼び出し処理

プロセデュアーや函数は，ステートメントラベルと同様に，参照するステートメントが出て来た時に定義されている保障は全く無いし，ランタイムにロードしてリンクする事もSIMBOLでは許している。そこでこの処理もラベルの処理と似た方法を取らなくてはならない。プロセデュアーや関数が，参照されたり，定義されたりすると，プログラムテーブルに登録される。このテーブルにはプロセデュアーや関数に対して1つずつのアイテムが対応して，その名前，定義済みであれば入口番地などを示す情報が記入されている。オブジェクトプログラムの中でサブルーチンコールはFACOM 230／60 のSXJ命令が使われるが，この命令は指定したインデックスレジスターにこの命令の次のアドレスをセットしアドレス部で指定した番地にジャンプする機能を持っている。

プロセデュアーや関数の呼び出し側のオブジェクトとしてはSXJ命令を出しジャンプ先はプログラムテーブルの対応するアイテムにしておく。

6 コンパイル

ここで言うコンパイル処理はインクリメンタルコンパィレーションとハッキリと区別して考えている。インクリメンタルコンパイレーションによって出来たプログラムが，ステートメント単位のオブジェクトになっていて，実行も，エクゼキュータが介在しないと出来ないのに対して，コンパイルされたプログラムは，コントロールが渡るとまったくエクゼキュータの介入なしで実行出来る点で異っている。これは，つまり今までバッチ処理で行っているコンパイルとまったく同じものである。インクリメンタルコンパイレーションに依って作成されたオブジェクトプログラムをＩ形式プログラム，コンパイルされて出来たオブジェクトプログラムをＣ形式プログラムと呼んでいる。

前にも簡単に触れておいたが，コンパイルするプログラム単位については次のものに制限されている。

1. プロセデュアーを単独で
2. 関数を単独で
3. メインプログラムと関連するアクティビティプログラムをまとめて

この制限については色々と議論のある所であろうが，１，２，は特に問題ないとして3.については，アクティビティを単独に出来る様にする方が実用上，有益ではないかとも思われる。しかし，技術的な問題と，本当に必要か否かを総合的に判断して見ると，むしろ，この様にする方が，良いのではないかと考えている。

コンパイル処理によって作成されるＣ形式プログラムはＦＡＣＯＭのオペレーティングシステムに於ける相対形式プログラムと完全に同じ形式になっていてリンクエディターやライブラリーエディターで扱える様になっている。

7 プロセスとPCB

プロセスが発生するとフリーストレッジにプロセスコントロールブロック（以下PCBと略記する）が作られる。PCBにはプロセスをコントロールする為に必要な情報が記入されていて，それは，

1. プロセスの発生時刻
2. プロセスのステータス
3. リアクティベイションポイント
4. プライオリティ
5. 最後にアクティブであった時刻
6. イベントノーティスへのポインター

などである。

PCBにはこれ以外にこのプロセスにローカルなデータがあればその為のエリアがとられており，実際には2つのブロックから成っている。

第1のブロックをPCBFと呼び，これには上記の情報が書き込まれている。又第2のブロックをPCVBと呼び，ここにはローカル変数の為のエリアが割当てられている。これら2つのブロックはお互にポインターによりチェインされている。PCBFはどのアクティビティに属すプロセスかについては，そのサイズは固定長であるが，PCBVにローカル変数の数によって変るので，別のアクティビティに属すプロセスの間ではサイズが異なるのが普通である。しかし，同じアクティビティに属するプロセスについては，もちろん同じ長さになっている。もしローカルを全全持たないプロセスについては，PCBVは作成されない。

次の図にPCBのフォーマットが示してある。

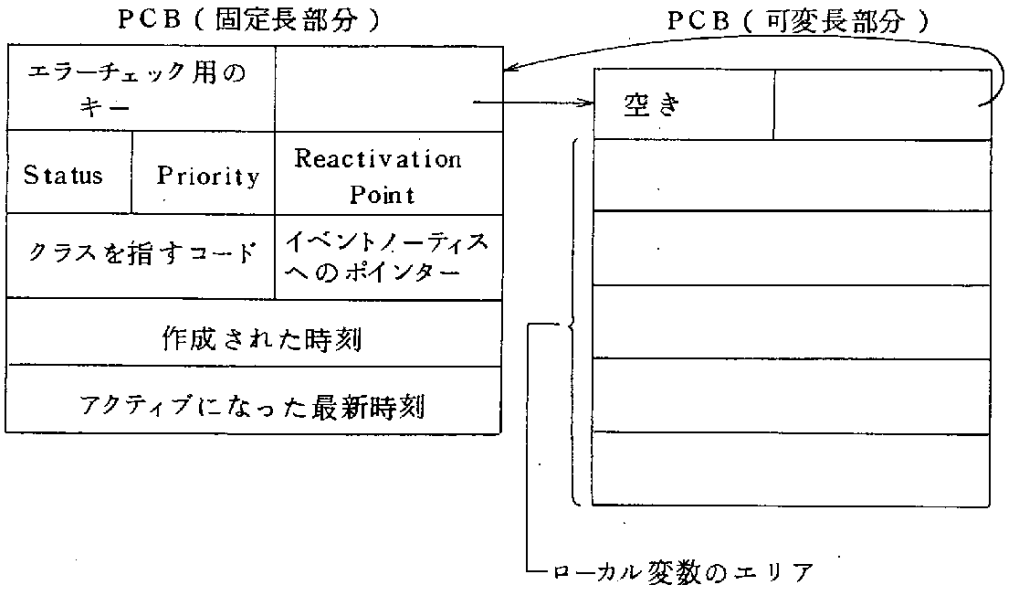


図17　PCBのフォーマット

PCBの情報のうち，リアクティベイションポイントは，アクティビティプログラムで，次に実行するラインを示すポインターで，詳しくはスケジュール処理の項で説明してある。

又，イベントノーティスへのポインターは，プロセスがスケジュールされると対応して作成されるイベントノーティスを示すポインターであり，従って，プロセスがスケジュールされていない時には，まったく意味を持たない情報である。

8　プロセスとローカル変数

アクティビティプログラムで宣言されたローカル変数に割付けられるメモリーセルは，このアクティビティから発生するプロセスのＰＣＢＶに取られる。ローカル変数には出現した順にＰＣＢＶの先頭から相対番地が割当てられる。配列，セット，キューなどがローカルに定義された場合にはもちろんＰＣＢＶ上には各々に必要なサイズ分のメモリーが割当てられるので，次に定義される変数の相対番地はそれまでに定義された変数の数でなくメモリーサイズ分だけ取られた値となる。（図18）

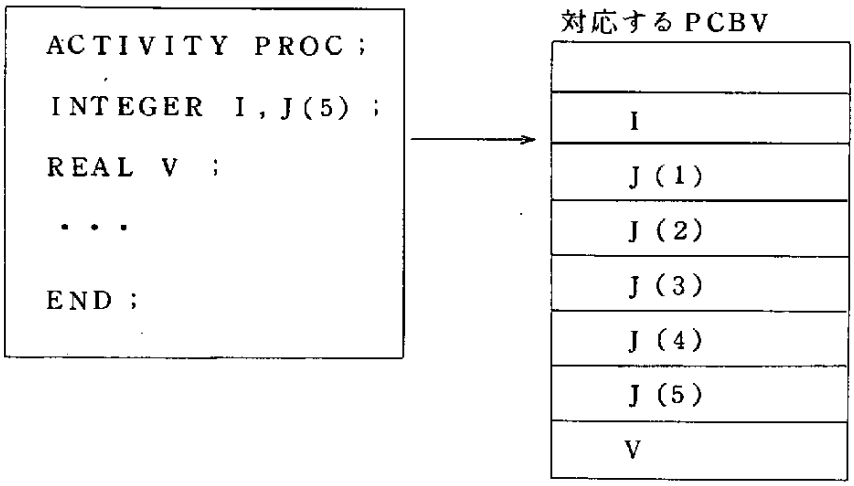


図18　ローカル変数の割付け

ローカル変数についてはその変数が宣言されると同時に，ＰＣＢＶに於ける相対アドレスを決める事が出来る。セットやキューがプロセスにローカルに宣言された時でも，ヘッドがＰＣＢＶに取られるのでヘッドの先頭位置を同じ方法で与える事が出来る。そこでローカル変数を使っているステートメントに対しては，特定のインデックスによるモディファイとＰＣＢＶに於ける相対アドレスによってオブジェクトコードを出す事が可能である。

このインデックスには，

1. ローカルリファレンスの場合，ローカル変数を参照するのは今実行しているプロセスのはずであり，スケジューラーがこのインデックスの値をプロセスが変る度に，その PCBV アドレスをセットしているのでローカル変数を参照出来る。

2. エクスターナルリファレンスの型でローカル変数が参照される場合にはエクスターナルリファレンスのオブジェクト自身のそのプロセスの PCBV アドレスを求める部分が作られており，インデックス（ローカルリファレンスの時とは別のインデックス）にオブジェクト自身がセットしている。

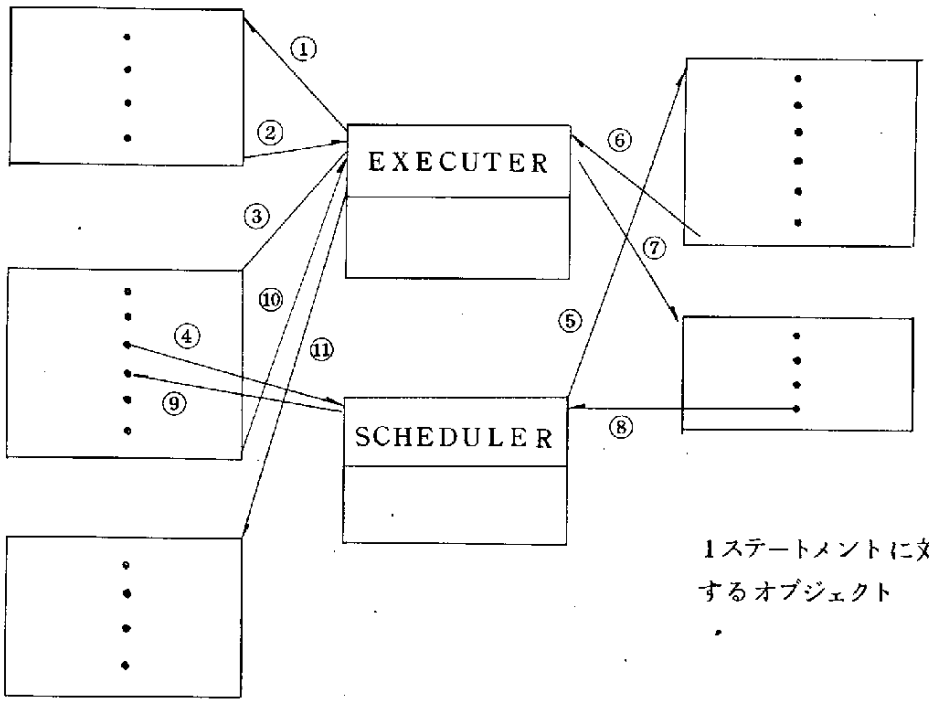
従って，この場合にも対象としているローカル変数をまちがえなく参照する事が出来る。

9　モデルの実行とエクゼキューター

モデル，即ちSIMBOLによって作られたオブジェクトプログラムは，SIMBOLプロセッサーが介入して実行する場合と単独で実行する場合がある事を述べたが，前者の場合，具体的には主にエクゼキュータがこれを行う。この様な形式では，アクティビティやメインプログラムは常にI形式のオブジェクトであるが，プロセデュアーや関数は，I形式のものも，C形式のものも同時に存在する事が許されている。

1.　エクゼキュータから見たプログラムストラクチャー

I形式のプログラムでは原則として1ステートメント実行する度にエクゼキュータにコントロールが戻り，エクゼキュータが次に実行すべきステートメントを決めてそこにコントロールを渡すと言ったルールで行われるもちろん，これらローカルシーケンスに従って実行する時の事であって，スケジューリングと関連して途中から他のプロセスにコントロールが移される様な場合には，間にスケジューラが介入する事になる。しかし，一度，他のプロセスにコントロールが渡っても，そこからは又，ローカルシーケンスに従って実行される事になるので，エクゼキュータが再び実行シーケンスの決定を行う事になる。この様な繰り返しによってモデルの実行が行われる。これらの関係を概念的に示したのが次の図である。



——→ はコントロールフローを表わし円印の数字実行順を表わす。

図19　モデルの実行過程

エクゼキュータはステートメントの種類によって違ったルールで，次に実行すべきステートメントを決定する様な事をなるべくさけて，どのステートメントであっても同じルールで，単独にコントロールを渡す場所が求められる事が望ましい。この様な理由からステートメントの方で少し工夫をし，エクゼキュータは相手がどんなステートメントであっても，同じルールで次に実行するステートメントを決められる様になっている。これは以下の方法に依っている。

(1) LETステートメント，INSERTステートメント，REMOVEステートメントなどの実行手順を変更する事のないステートメントでは，ステートメント実行後に，エクゼキュータにSSJ命令でリターンする。SSJ命令はサブルーチンリンクの為に使われるFACOM230／60のハードウェア命令で，この命令を

実行した次の番地をジャンプ先のアドレスにセットし，その次の番地にコントロールを渡す機能を持っている。従ってＳＳＪ命令の番地にフィジカルシーケンスに従って，次のステートメントがあるアドレスを書き込んでおけば，エクゼキュータは簡単に参照する事が出来る。

この関係を次の図で示してある。

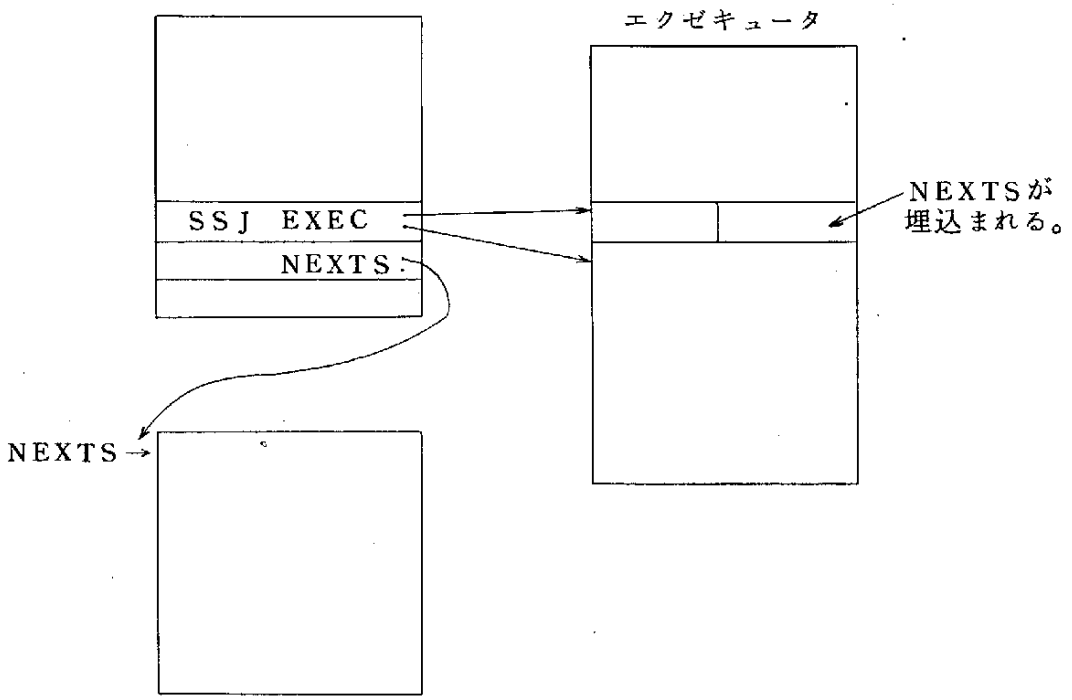


図20　エクゼキュータとオブジェクトの関係

(2)　GO TOステートメントの場合

GO TOステートメントに対するオブジェクトは無条件ジャンプ命令で，対応するラベルテーブルのアイテムへ飛び込む様になっている。

ラベルテーブルには次の図で示す通りＳＳＪ命令がありアドレス部にはエクゼキュータへの入口番地が書き込まれている。

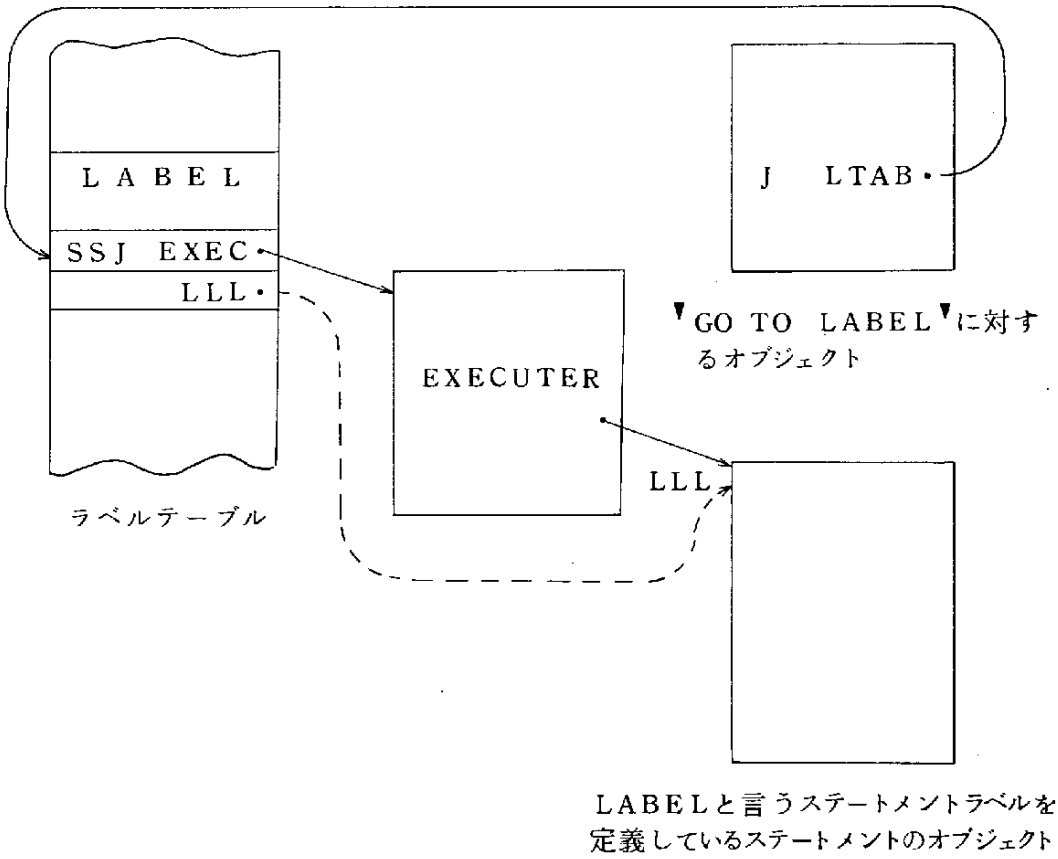


図21　GO TO ステートメントのオブジェクトとエクゼキュータ

GO TO ステートメントにコントロールがエクゼキュータから渡されると，ジャンプ命令を実行するのでラベルテーブルのSSJ命令にコントロールが渡る。次にSSJ命令が実行される事になり，エクゼキュータにコントロールが戻される。GO TO ステートメントに書かれたラベルは，定義されると，そのアドレスがSSJ命令の次のアドレスに書き込まれているので，普通のステートメントと同じルールで次に実行するステートメントを決めれば，GO TO ステートメントが示すステートメントにコントロールを渡す事が出来る。

(3)　プロセデュアーコールの場合

プロセデュアーコールに対して出されるオブジェクトプログラムについては前

にも述べたが，ＳＸＪ命令のジャンプ先をプログラムテーブルの対応するアイテムにしてあり，そのアイテムがサブルーチンであるかの様にして扱われている。プログラムテーブル側では無条件ジャンプ命令が出されていて，飛び先が実際にプロセデュアーの入口番地になっている。もちろんこれは，入口番地が定義されている場合で，まだ未定義であればリンクフォールトハンドラーのアドレスが植込まれている。従ってもし，この様な状態でプロセデュアーコールが行われるとリンクフォールトハンドラーに飛込む事になる。もし正常に定義されていれば呼び出したいプロセデュアーにコントロールを渡す事が出来る。

中介となるプログラムテーブルに植込まれている命令が無条件ジャンプ命令である為に，プロセデュアー側でパラメータのあるアドレスや，復帰番地は直接呼ばれた時と同じ様にインデックスを参照して求める事が可能である。次の図は以上の関連を示したたものである。

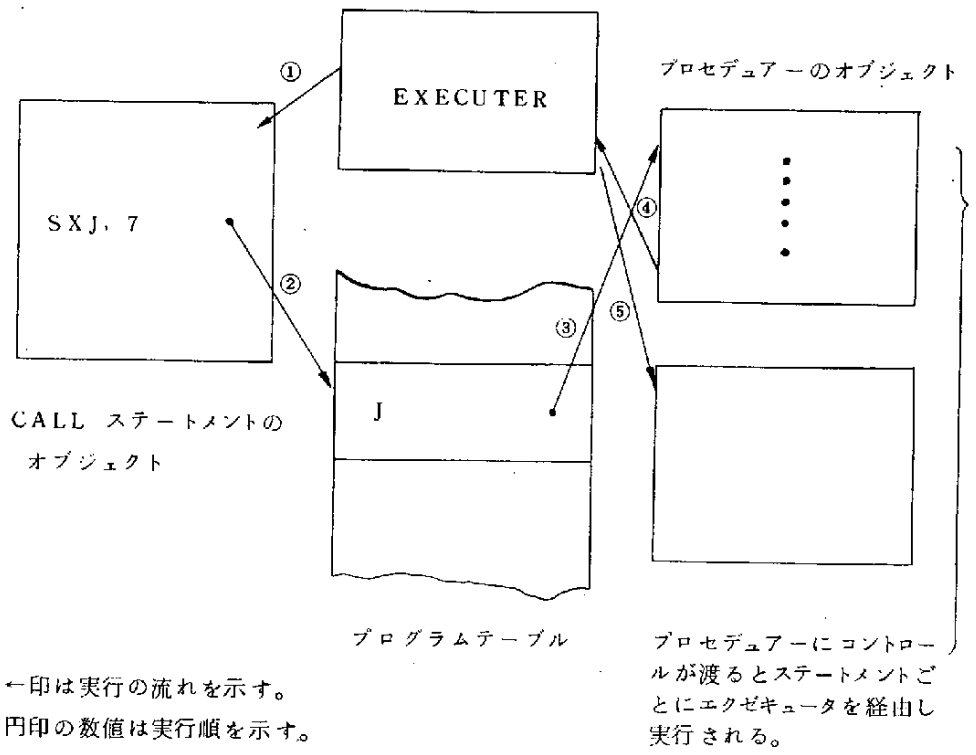


図 22　プロセデュアーコールとエクゼキュータとの関係
（呼ばれる側がＩ形式の場合）

この様なルールでプロセデュアーが呼び出されるので，CALLステートメントのコードブロックにエクゼキュータがコントロールを渡すと，プロセデュアーの先頭ステートメントを実行し終るまでエクゼキュータにコントロールは戻ってこない。プロセデュアーにコントロールが渡る以前にエクゼキュータを経由してから行う様にする事も処理の上からは簡単に出来るが，プロセデュアーがC形式（コンパイルされたオブジェクト）である事も考慮すると直接コントロールを渡す方が良いと思っている。

次に，プロセデュアーがC形式のオブジェクトである場合について説明する。次の図はこの様なケースのコントロールの流れを示したものである。

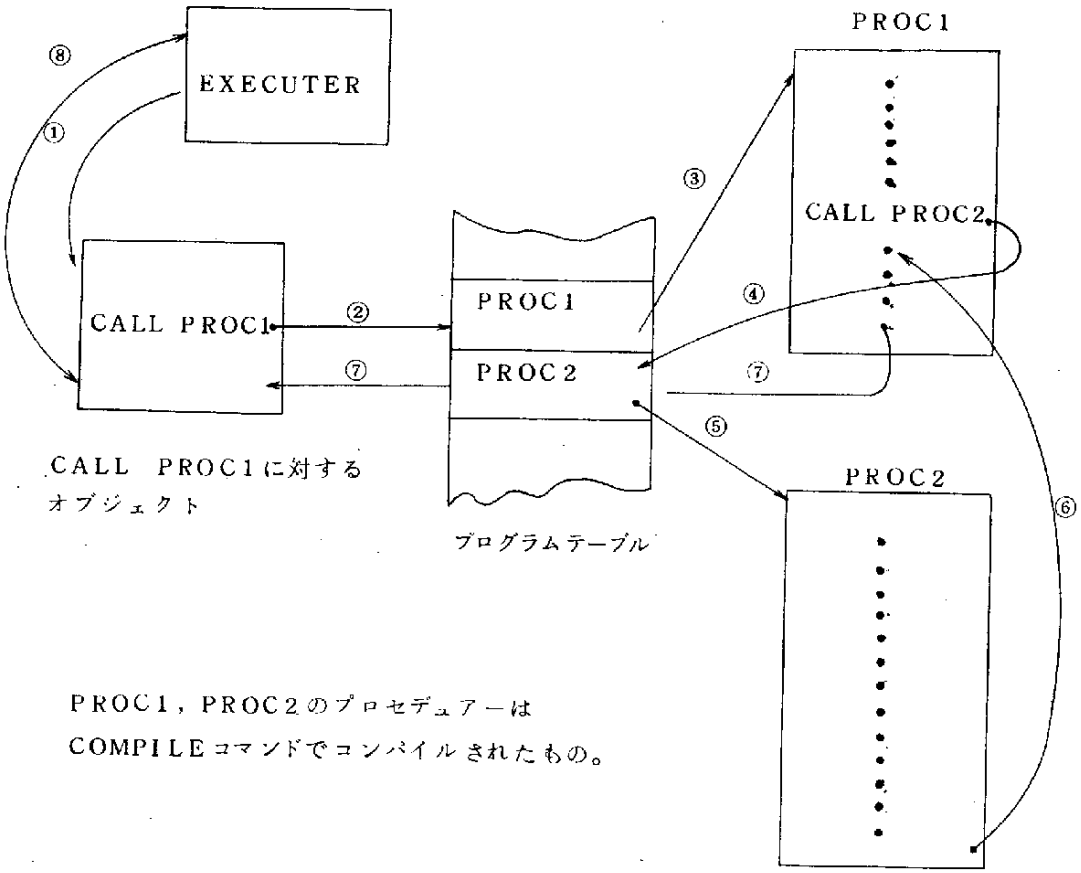


図23　プロセデュアーコールのエクゼキュータの関係
（呼ばれる側がC形式の場合）

図で解る様にエクゼキュータからCALLステートメントのコードブロックにコントロールが渡されるとプログラムテーブルを経てC形式のプロセデュアーを完全に実行し終ってから再びコントロールがCALLステートメントのコードブロックに戻りその後，普通のステートメントと同じ様に，エクゼキュータへSSJ命令で制御が戻される。もしC形式オブジェクトの中で他のプロセデュアーを呼び出していて，それがI形式である場合にはもちろん1ステートメントと実行する都度エクゼキュータに制御が戻されるが，C形式であればエクゼキュータはこれにまったく関与しない。

10　セットやキューの取扱い

(1)　セットとキューの構造

セットとキューはそれぞれセットヘッド（Set head）とキューヘッド（Queue head）を基点として，これらを構成するメンバーがチェインされ対称リスト構造になっている。これらのメンバーとなるのはプロセスやブロックと言ったエレメントであるが，エレメント自身にポインターを持たせ，直接リンクする方法は取らず，セットやキューに登録されたエレメントに対しては，セットであればセットメンバーエレメント（ＳＭＥと略記する），キューであればキューメンバーエレメント（ＱＭＥと略記する）が作成されエレメントの代りにこれらがお互にチェインされている。その様子を次の図に示してある。

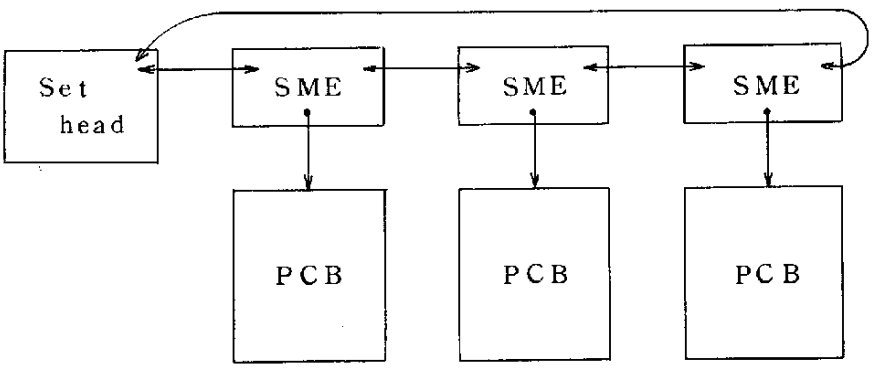


図24　セットの構造

SME，QMEは，それぞれ２ワード，３ワードで構成されていて次の図にその構造を示してある。

エレメントへのポインター	
先行 SME へのポインター	後続 SME へのポインター

図25　ＳＭＥの構造

エレメントへのポインター	
先行 QME へのポインター	後続 QME へのポインター
キューに登録された時刻	

図26　ＱＭＥの構造

又，セットやキューの基点となるヘッドはセットの場合３ワード，キューでは８ワードで構成されている。その構造を次の図で示してある。

セットヘッドを示す ID	
先頭 SME へのポインター	最終 SME へのポインター
現在のメンバー数	

図27　セットヘッドの構造

キューヘッドを示す ID	
先頭 QME へのポインター	最終 QME へのポインター
キューに入ったメンバーの総数（M）	
最　大　待　数（Lmax）	現在のメンバー数（Lc）
メンバー数の総和（ΣLc）	
待ち時間の総和　（Σw）	
待ち無しで通過したメンバー数（Mo）	
待ち時間の最大　（Wmax）	

図28　キューヘッドの構造

(2) メンバーの登録と削除

セットやキューに登録したり，メンバーを削除したりする事はINSERTステートメントやREMOVEステートメントが実行されると行われる。

これらの扱いはINSERTやREMOVEステートメントに対するオブジェクトがランタイムシステムのサブルーチンコールの形で出されているので，全てランタイムシステムで行っている。

新しくセット又はキューにエレメントが登録される時には，SMEやQMEを作り出さなくてはならない。これらはFree storageにSME同し，QME同しがそれぞれお互にチェインされまとめて管理されている。必要な時に取り出し使い終れば再びFree storageに返却される。

従ってSMEやQMEはメンバーの登録の際作成され，メンバーが削除されると消去される。WAITステートメント，WAKEステートメント，INTERRUPTステートメント，RESUMEステートメントなどが後に説明するイベントノーティスとQMEを兼ねて使用するのは，その都度Free storageから取り出したり返却する手間をはぶき実行スピードを上げる為の処置である。

a セットの場合

セットに新しくメンバーを登録する時にはFree storageからSMEをもらい登録するエレメントとのリンクを付ける。その後でSMEをセットの必要な場所に挿入しSME同しのチェイニングを行う。

このセットのメンバーが一つふえたのでセットヘッドにあるメンバー数を示す部分をアップデートする。

セットからメンバーが取り除かれる場合には丁度これとは逆に，対応するSMEをセットから取り出しチェインを修正した後にFree storageに返却する。セットヘッドのメンバー数は1だけ減らされる。

b キューの場合

キューにメンバーを登録したり，削除する場合にはセットと同じ様な処理の他に，簡単な統計処理も行わなくてはならない。

キューに登録される場合はQMEをFree storageからもらいエレメント

とリンクを取った後，QMEに登録時刻を書き込む。この時刻はその時点でのTIMEの値が入れられる。QMEはセットの時と同じ様にキューに挿入される。登録時にはキューヘッドの各アイテムに対して次の様な処理が行われる。
（記号は図28に従っている）

1. Mを1だけ増す
2. Lcを1だけ増す
3. LmaxとLcを比較してLcが大きければLmaxにLcを入れる。そうでなければそのままにする。
4. ΣLcにLcを加える。

キューからメンバーが削除される時にはセットの時と同様な処理以外に登録の時と同じくキューヘッドに対して次の処理が行われる。
（削除されるメンバーのキューに登録された時刻をITで表わす。）

1. IT＝TIMEならMoを1だけ増し処理を終る。

そうでない場合

2. Σwに（TIME－IT）を加える。
3. Wmaxと（TIME－IT）を比較しWmaxが小さければ（TIME－IT）をWmaxに入れ，そうでなければなにもしない。

11　スケジューリングメカニズム

イベントを時刻に従って順次実行させるのが，スケジューリングメカニズムの基本的な機能である。SIMBOLのスケジューリングメカニズムは，スケジューラーとスケジュールテーブルに依って構成される。

スケジュールテーブルは，イベントがスケジュールされると，対応して作られるイベントノーティスが先行イベントノーティス，後続イベントノーティスの両方向へのポインターを持つ，対称リスト構造にチェインされている。

イベントが，スケジューリングステートメントによってスケジュールされると，実行予定時刻（以下ETと略記する）が与えられイベントノーティスに書き込まれる。スケジュールテーブルに登録されたイベントノーティスは，ETの順にsortされており，スケジューラーはこのテーブルを参照しながら次のイベントを見つける事が出来る。（図 29 参照）

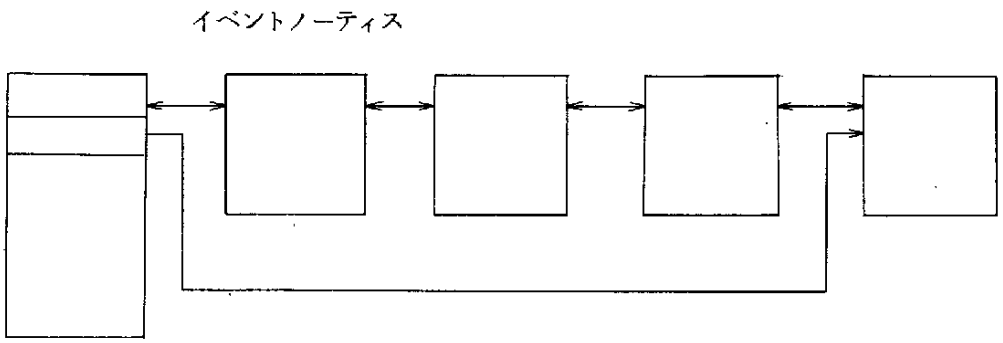
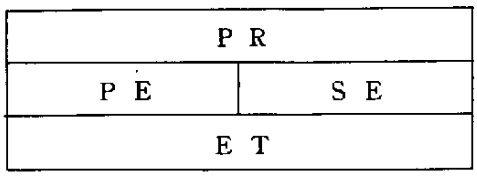


図 29　スケジュールテーブルの概略図

スケジュールテーブルに登録されるイベントノーティスは3ワードで構成され，次の図にその構造を示してある。



- PR は対応するプロセスへのポインター
- PE は Preceding event, 即ち先行イベントノーティス へのポインター
- SE は Succeeding event, 即ち後続イベントノーティスへのポインター
- ET はこのイベントの実行予定時刻である。

図30　イベントノーティスの構造

スケジューラーの処理

スケジューラが行う基本的な処理は，スケジュールテーブルを参照して，次に実行するイベントを決定する事であるが，実際には

(1) システムクロックの管理

(2) スケジュールテーブルの管理

(3) プロセスコントロールブロック（PCB）の更新
（リアクティベイションポイントなど）

(4) プロセスにコントロールを渡す為の準備

(5) スケジューリングと関連したトレース処理

なども行っている。

又，スケジュールステートメントがコンパイルされると，そのオブジェクトは全てランタイムシステム，即ちスケジューラに対するサブルーチンコールの形が作り出される。従ってスケジューラは各スケジュールステートメントの処理に対応した入口を持っている。

以下に各ステートメントごとに行う処理について順に述べてある。

(1) SCHEDULE

対象とするプロセスはまだスケジュールされていないので，イベントノーティスは作られていない。そこでまずイベントノーティスを作り PCB とリンクする。

スケジュール句の指定がＡＴ又はDELAYであれば，オブジェクトプログラム側で，ＥＴが算定されているのから，ＥＴの値を入れた後に，スケジュールテーブルに登録する。

又，スケジュール句の指定がBEFOR又はAFTERであると，ここで指定されたプロセスはすでにスケジュールされており，イベントノーティスが登録されている。これはＰＣＢから探す事が出来るので，このイベントノーティスの直前又は直後にスケジュールするイベントを登録し，ETはAFTER，又はBEFOREで指定されたプロセスに合せる。

次の図はこれらの関係を示してある。

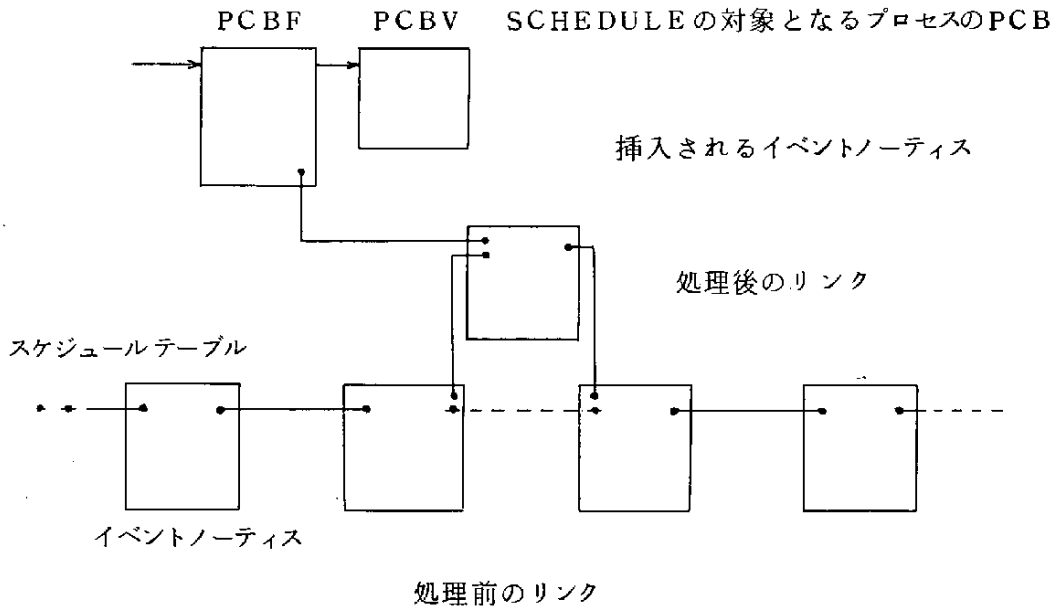


図31　SCHEDULE処理の概略（ＡＴ又はDELAY指定の場合）

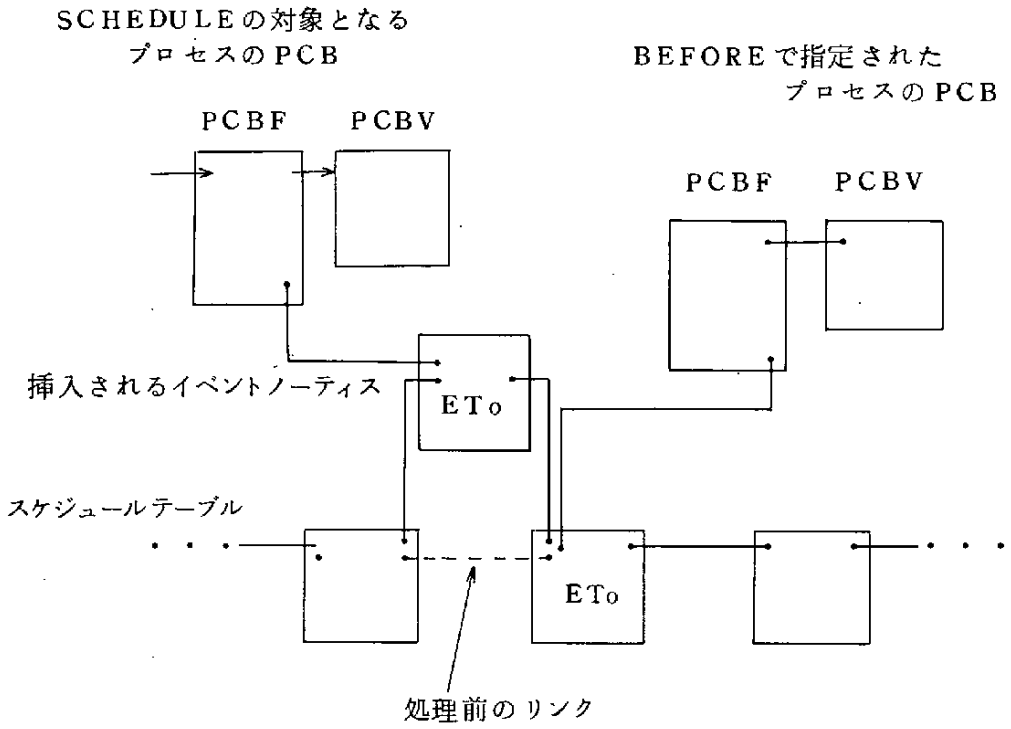


図32　SCHEDULE処理の概略（BEFORE指定の場合）

(2)　RESCHEDULE

このステートメントで指定されたプロセスはすでにスケジュールされているからイベントノーティスが存在する。イベントノーティスはプロセスのPCBから探す事が出来る。このイベントノーティスを一度スケジュールテーブルから取り外した後，スケジュール句で指定された方法に従ってスケジュールテーブルに再登録する。この処理はSCHEDULEステートメントの場合と全く同じである。

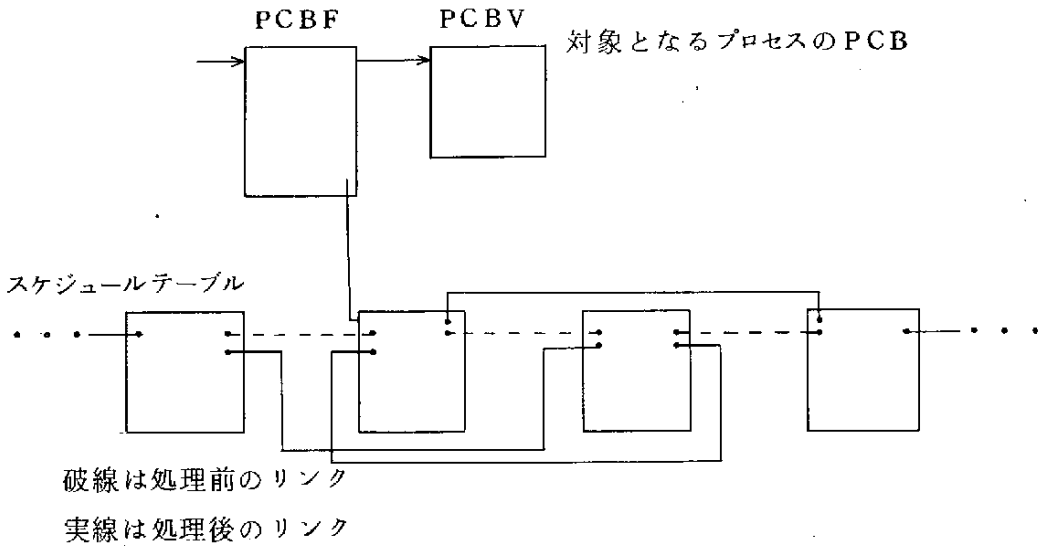


図33　RESCHEDULE処理の略図

(3)　CANCEL

指定されたプロセスのPCBからイベントノーティスを探し，スケジュールテーブルから取り外し，Free storageに返却する。プロセスの指定が省略されたり，或は直接SELFの指定があって結果的にこのステートメントを実行しているプロセス自身が対象となった場合には，スケジュールテーブルの先頭にあるイベントノーティスが対象となり，このイベントノーティスを取りはずしスケジュールテーブルの一部をアップデートし，システム変数SELFの値を書き換える。

(4)　TERMINATE

ターミネイトするプロセスはスケジュールされている必要はないので，PCBのreactivation pointをターミネイト状態を示すコードにし，statusをターミネイトにした後，イベントノーティスの有無を確め，有ればスケジュールテーブルから取り外し，Free storageに返却する。

(5) WAIT

WAIT ステートメントの機能は，このステートメントを実行したプロセスをスケジュールテーブルから取り外し，指定したキューに接ぎ，待ち状態にする事である。従って，スケジューラが行う内部処理は，

1. SELF，即ち現在実行中のプロセスのイベントノーティスをスケジュールテーブルから取りはずす。

2. 他のスケジュールステートメントの処理ではスケジュールテーブルから取り出されたイベントノーティスは消去し Free storageに返却されるが，WAIT ステートメントの処理ではこれを返却せずにそのままキューのデータストラクチャーに組込んでしまう。

この為にキューに於けるQME（第11項参照）とイベントノーティスのサイズもパターンも同じ様な形式にしてある。

キューに挿入する処理はセットやキューを扱う為の INSERTステートメントの処理ルーチンに任せてしまうのでこの点は，やはり11項に譲る事にする。

3. CANCEL ステートメントの SELF 指定の時と同様に，WAIT ステートメントの場合にもスケジュールテーブルの先頭にあるイベントノーティスが削除されるので同じ処理が必要である。

次の図は以上の処理をまとめたものである。

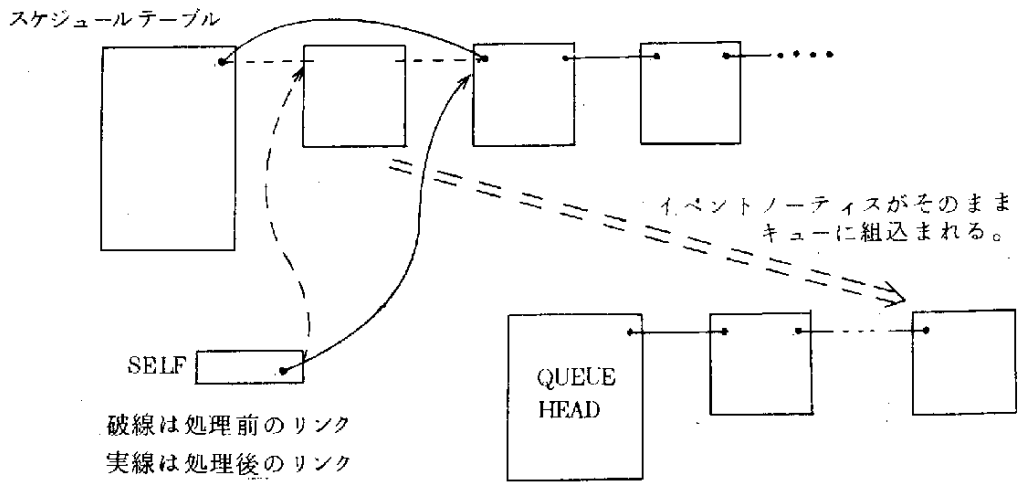


図34　WAIT処理の略図

(6) WAKE

WAKE ステートメントの機能は，キューに入っている先頭のプロセスをキューから取り出し，スケジュールする事である。スケジューリングに関しては，SCHEDULE ステートメントや，RESCHEDULE ステートメントと同じ様にスケジュール句を指定出来，この辺の処理はこれらのステートメント処理と共通である。WAIT ステートメント処理で述べている通り，キューに於ける QME は，イベントノーティスと同じく3ワードで構成されて，共用されている。WAKE ステートメントでも，キューの先頭にあるメンバーの QME を取り外し Free storage に返却する事無しに，直接イベントノーティスとして取り扱う。従ってこれ以後の処理は，SCHEDULE ステートメント処理とまったく同じなので省略する。

(7) INTERRUPT

このステートメントではエレメント変数と，キューが指定され，エレメント変数が示すプロセスをキューの先頭に登録した後エレメント変数の値を，INTERRUPT ステートメントを実行したプロセス（即ち自分自身）に変更する機能を持っている。そこで内部処理は次の手順で行われる。

1. エレメント変数の示すプロセスはかならずスケジュールされているので，PCBからイベントノーティスを探す。
2. 1で見つけたイベントノーティスをキューの先頭に直接チェイニングする。従って，この時 QME は作り出さず，イベントノーティスがそのまま QME の替りをする。
3. イベントノーティスのETの部分には，現時刻 TIME から ET を引いた値を入れる。即ち，

   ET←（TIME−ET）とする。
4. 指定されたエレメント変数には，システム変数 SELF をそのままコピーする。

(8) RESUME

このステートメントの処理は INTERRUPT ステートメントの処理と対を成している。RESUMステートメントではパラメータとして，キューとエレメント変数が指定されるが，その処理は，

1. 指定されたキューの先頭にある QME を取り出して，キューに対する REMOVE ステートメントと同じ処理をする。
2. QME に書き込まれている ET をもとにこのプロセスを現時刻からDELAY ET でスケジュールする。

このステートメントの処理でもスケジュールに対しては新しくイベントノーティスを作成する事はせず QME をそのままイベントノーティスとして使用する。

昭和47年3月 発行

発行所　財団法人　日本情報処理開発センター
東京都港区芝公園3丁目5－8
機械振興会館内
TEL（434）8211（代表）

印刷所　株式会社　チャンスメーカー
東京都千代田区神田錦町3－19
TEL（295）1177（代表）

46－S004