第５章　言語処理系の作成技法

第５章　言語処理系の作成技法[C14)]

5.1　概　　要

高級言語で記述されたプログラムは，一般にそれを実行する計算機用の機械語プログラムに変換して実行される。この変換を行うコンパイラ[A1,G4)]は，大きくは２つの処理から成る。即ち，前半部では，入力されたソースプログラムがその記述言語の文法に合っているか否かを解析し，中間語と呼ばれる内部形式に変換する。そして，後半部では，この中間語を機械語に変換して，オブジェクトプログラムを生成する。前半の解析処理は，さらに構文の正しさをチェックする構文解析とその他の文法規則をチェックする意味解析に分けられる。このうちの構文解析については，理論化が進み，技法も確立している。

ＳＰＬ用のコンパイラの開発に際しては，既存技術を利用できる部分もあったが，前章までで述べてきたような新しい言語機能の処理方式については独自の方式を研究する必要があった。その対象となる主な言語機能は次のようなものである。

(1)　共通データ宣言の独立化と階層化機能

(2)　データ型定義機能

(3)　外部参照の手続きとユーザ定義データ型の仕様明記機能

(4)　手続きのインライン展開機能

(5)　コンパイル時実行機能

これらの項目のうち，(1)～(3)は解析処理に関するものである。即ち，新言語は大規模ソフトウェアへの適用を前提にしているので，モジュール単位に分割コンパイルできることが必須条件である。そこで，変数，ユーザ定義データ型，手続きなどの宣言や定義を含むモジュールとそれらを参照するモジュールが異なる場合には，宣言，定義部分と参照部分の対応付けが重要な課題である。

一方，(4)，(5)は生成処理に関するもので，その実行のタイミングおよび効率的な処理方式が課題である。この生成処理については，この他にコンパイラ独自の課題がある。即ち，本コンパイラは，早期開発の必要から，機械語をオブジェクトコードとする通常の方式をとらず，既存の高級言語である制御用 Fortran （ＰＣＬ）をオブジェクトコードとするプリプロセッサ方式を採った。そのため，上記処理を終えた中間語プログラムからＰＣＬソースプログラムへの効率的な変換方式が必要である。

本章では，以上の課題を解決するために，ＳＰＬ用のコンパイラの開発に際して研究開発されたコンパイル技法について述べる。

5.2　基本処理方式[C11)]

コンパイラの設計において，その基本的な処理方式を決めるものは，コンパイラの処理機能を逐次的に分割するフェイズ構成とそのフェイズ間で受け渡す中間語プログラムの形式，およびフェイズ間で共通に参照する記号テーブルのデータ構造である。なかでもフェイズ構成は，コンパイラの機能と

性能への影響が大きく，また中間語や記号テーブルの形式などの設計の基準となる重要なものである。そこで，本節では，まずＳＰＬコンパイラの基本構成を述べる。

ＳＰＬコンパイラの処理内容は大別すると次のようになる。

(1) 解析：ソースプログラムの構文解析と意味解析

(2) 展開：手続きのインライン展開

(3) 解釈：コンパイル時実行文の解釈実行

(4) 生成：オブジェクトプログラムの生成出力

他の多くのコンパイラが，主に解析処理と生成処理から構成され，各モジュール単位に独立にコンパイルを行うのに対し，ＳＰＬコンパイラは，上述のように展開，解釈処理も行うほか，コンパイル時に他モジュールの情報を参照する必要がある。そのため，フェイズ分割時に考慮すべき課題として，次のようなものがある。

(1) 解析，展開，解釈の処理順序

(2) 変数と定数の宣言や手続きとデータ型の定義とそれらの参照部分との対応づけ

第１の点については，例えば，アセンブリ言語の可変マクロ機能やＰＬ／Ｉのコンパイル時機能の実行は，まずソースプログラム上で解釈，展開処理を行い，しかる後に解析処理を行う方法をとっている。しかしながら，このような方法では，あるモジュールをコンパイルする時にはそこに展開されるモジュールが既に定義されている必要がある。そのため，ボトムアップ的な開発を志向する言語では問題ないが，ＳＰＬのように，展開されるモジュールの定義よりもそれを引用するモジュールの定義の方が先行することの多いトップダウン志向言語には，この方式は適しない。

さらに，このように文法的正しさのチェックを展開処理後に行う方式では，誤りの検出が遅れるほか，展開されるモジュールについては，その展開された形が正しいものでさえあれば，その定義側は機能的断片で良いことになる。これでは，プログラムの信頼性や理解容易性の低下を招き，構造化プログラミングの精神に反する。特にＳＰＬでは，手続きのインライン展開機能は処理性能上の対策として導入したものであり，手続きの定義形式はインライン展開指定の有無に拘らず同じにして，後から指定の変更ができるようにしているため，この方式は適しない。

そこで，ＳＰＬコンパイラでは，まず初めに解析処理を行うことにした。そして，展開および解釈処理については，コンパイル時変数の値の設定時期と参照時期の関係を複雑にしないために両方を同時に行うようにした。

次に，第２の課題については，変数と定数の宣言や手続きとデータ型の定義を行うモジュールとそれらを参照するモジュールが異ることが多く，解析処理が別々に行われるため，両者の対応づけは簡単ではない。特に，手続きやユーザ定義データ型を段階的詳細化によるトップダウン開発に用いた場合は，その定義モジュールよりも参照モジュールが先に解析処理されるため，問題はより複雑である。

この最も簡単な処理方法は，個々のモジュールの解析処理時には構文解析とモジュール内の局所的意味解析のみ行い，型チェックを中心とした全体の意味解析はオブジェクトプログラム生成時に行う

ような方法である。しかし，この方式では，誤りの検出が遅くなるほか，コンパイラの処理が複雑になる。そこで，ＳＰＬコンパイラでは，各モジュールの解析処理結果を保存するＳＰＬライブラリを設け，その関連モジュールの解析処理時に必要に応じて参照できるようにすることにより，上記問題の解決をはかった。この詳細については次節で述べる。

以上のような基本方針に基づいて設計されたＳＰＬコンパイラの基本処理方式は，図5.1に示す通りである。まず，フェイズ構成は，解析処理，展開，解釈処理，生成処理の順に各々が数フェイズに分けられる。中間語はいずれも，オブジェクトコードが高級言語であることを考慮して，逆ポーランド記法を基本にした形式にした。そして，環境モジュールおよびインライン展開される手続きを含む処理モジュールに対しては，ＳＰＬライブラリに登録されている上位環境モジュールの解析情報を用いて解析処理を行い，その結果をライブラリに登録する。一方，オブジェクトプログラムの出力を必要とする手続きを含むモジュールに対しては，同様の解析処理後，インライン展開される手続きの本体をライブラリから取込みながら，展開，解釈処理を行い，最後に生成処理を行う。

5.3　共通ライブラリを用いた分割コンパイル方式

5.3.1　分割コンパイル

プログラムのモジュール化は特に大規模ソフトウェアの開発時に大きな効果をもたらすが，このような分野では，プログラムのテスト，デバッグ，修正作業の繰返しが多い。その場合，プログラムの一部分を修正する毎にプログラム全体を再コンパイルしていたのでは作業効率が悪いため，モジュール単位のコンパイル機能が必須である。例えば，最近，急速に普及したPascalは，もともとはプログラミング教育用の言語として設計されたものなので，常にプログラム全体をまとめてコンパイルする一括コンパイル方式を前提にしているが，実用化されたPascalコンパイラでは，モジュール化機能を付加して分割コンパイルを可能にしているものが多い。

一方，従来の言語では，外部手続き単位で独立にコンパイルできるものが一般的であるが，その代償として，共通データは参照側でその都度宣言するとか，外部手続きの引用部分の実引数の型チェックを行わないなど，プログラムの信頼性低下要因が多い。また，モジュール化を徹底した場合，その各モジュールを外部手続きとして作成するため，オブジェクトの実行効率やメモリ効率が低下する。なお，本論文では，モジュール間の型チェックを行わない従来方式を独立コンパイル方式と呼び，分割コンパイル方式と区別する。

ＳＰＬでは，このような従来方式の欠点を除き，かつ分割コンパイル方式を実現している。即ち，プログラムの開発期間中，一貫して管理されるライブラリを設定し，これを経由してモジュール間情報の受け渡しを行う。従って，各モジュールのコンパイル時には他モジュール情報を参照して，一括コンパイル方式と同様の型チェックを行うことができる。

このＳＰＬライブラリは次の３つの構成要素から成る。

(1)　環境モジュールライブラリ（ＥＭＬ）

(2) 処理モジュールライブラリ（ＰＭＬ）

(3) データ型テーブル

これらは，ＳＰＬコンパイラによって頻繁に参照されるため，その内部データ構造がコンパイル処理効率に与える影響は大きい。そこで，本節では以降，これらの内部構造について述べる。

5.3.2 ＥＭＬの内部構造

ＥＭＬの設計では，環境モジュール間の木構造の表現形式が基本となる。その最も単純な方法として，個々のモジュールの情報および木構造情報を独立に保存することが考えられる。しかし，この方式では，参照された変数名や定数名がどこで宣言されているかを調べる場合，下位のモジュールから順に探していく必要があり，処理が遅くなる。

そこで，ＳＰＬコンパイラでは，個々の環境モジュールのコンパイル時には，その上位環境情報を初期環境としてＥＭＬから取込み，それに追加する形で現在のモジュールをコンパイルするようにした。従って，ＥＭＬ内の各モジュールは，あたかもその上位環境モジュール情報がすべて自分のモジュール内で宣言されていたかのように情報を保存するので，迅速な参照が可能となる。

即ち，ＥＭＬは図5.2のような構造とした。この図は，図3.1のモジュール階層における環境モジュールＥ２の場合の例である。モジュール単位の情報は次のような構成要素から成る。

(1) ハッシュテーブル

(2) 識別子テーブル

(3) リテラルテーブル

(4) 一般属性テーブル

(5) モジュール管理テーブル

図の(a)のＥ２用の情報は次のような過程で作成される。即ち，まず最上位のＥ０がコンパイルされ，ＥＭＬに登録される。次に，Ｅ１がコンパイルされる時，Ｅ０のライブラリ情報を初期状態として用い，結果をＥＭＬに登録する。最後に，Ｅ２は，同様にＥ１のライブラリ情報を初期状態としてコンパイルする。その結果，Ｅ２の識別子テーブルには，その上位環境Ｅ０，Ｅ１で宣言された識別子がすべて含まれる。そして，これらのアクセスは１つのハッシュテーブルを経由して行うと共に，同名異種の識別子間では後から宣言されたものに先に出会うようにハッシュチェインを結ぶ。この結果，Ｅ２の下に結合された手続きの中で参照された各識別子は，１回のハッシュ演算でそのエントリに結びつけられる。

この識別子テーブルの各エントリは図の(c)のような固定長のデータ構造とし，基本データ型などの基本属性フィールドの他に，リテラルテーブルおよび一般属性テーブルへのポインターとハッシュチェインフィールドを有する。リテラルテーブルは識別子の名前を文字定数として保存するテーブルであり，最大３２文字までの可変長データなので識別子テーブルから分離した。一般属性テーブルは配列体や構造体などの複雑なデータ型およびユーザ定義データ型に関する詳細情報，あるいは初期値や

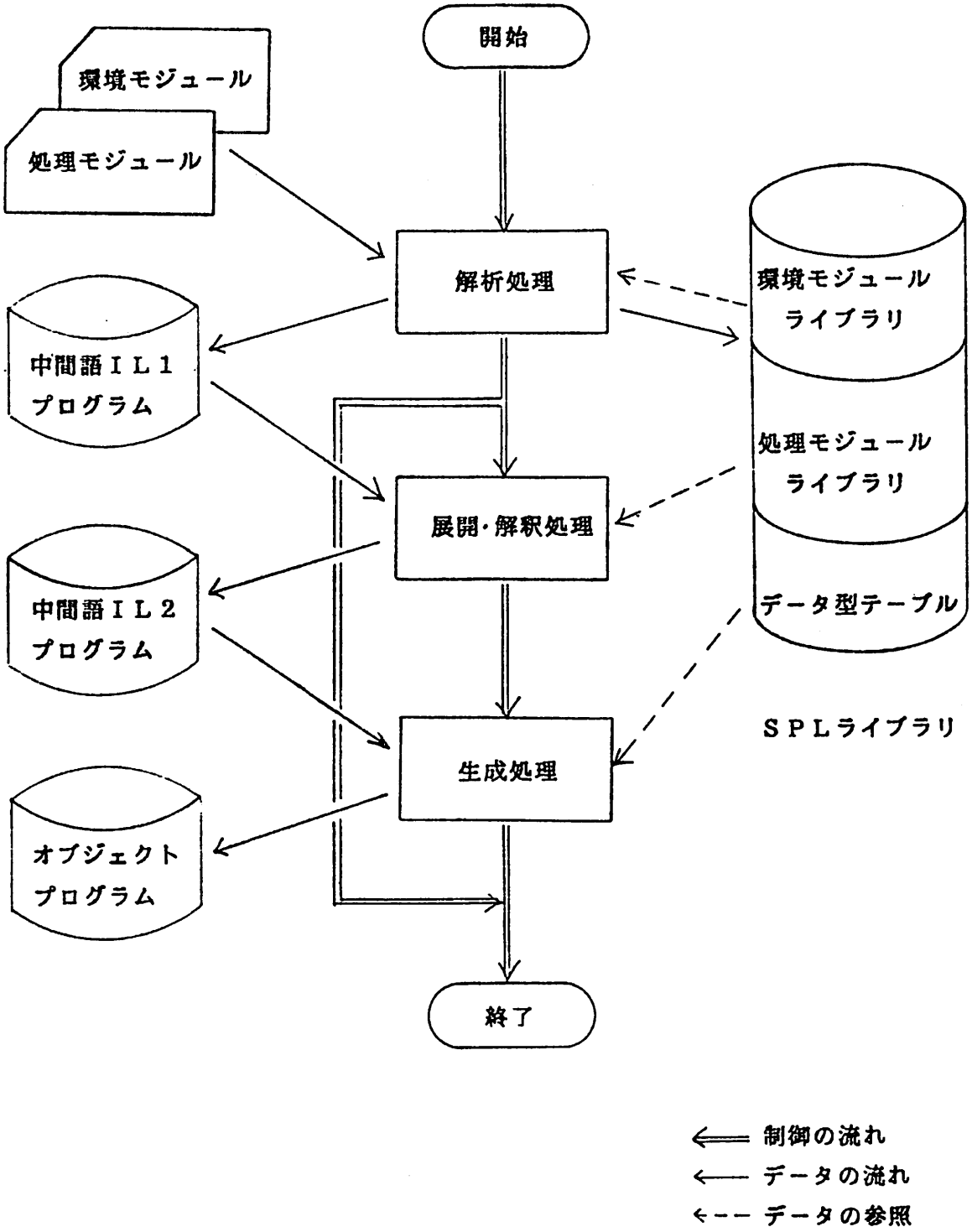
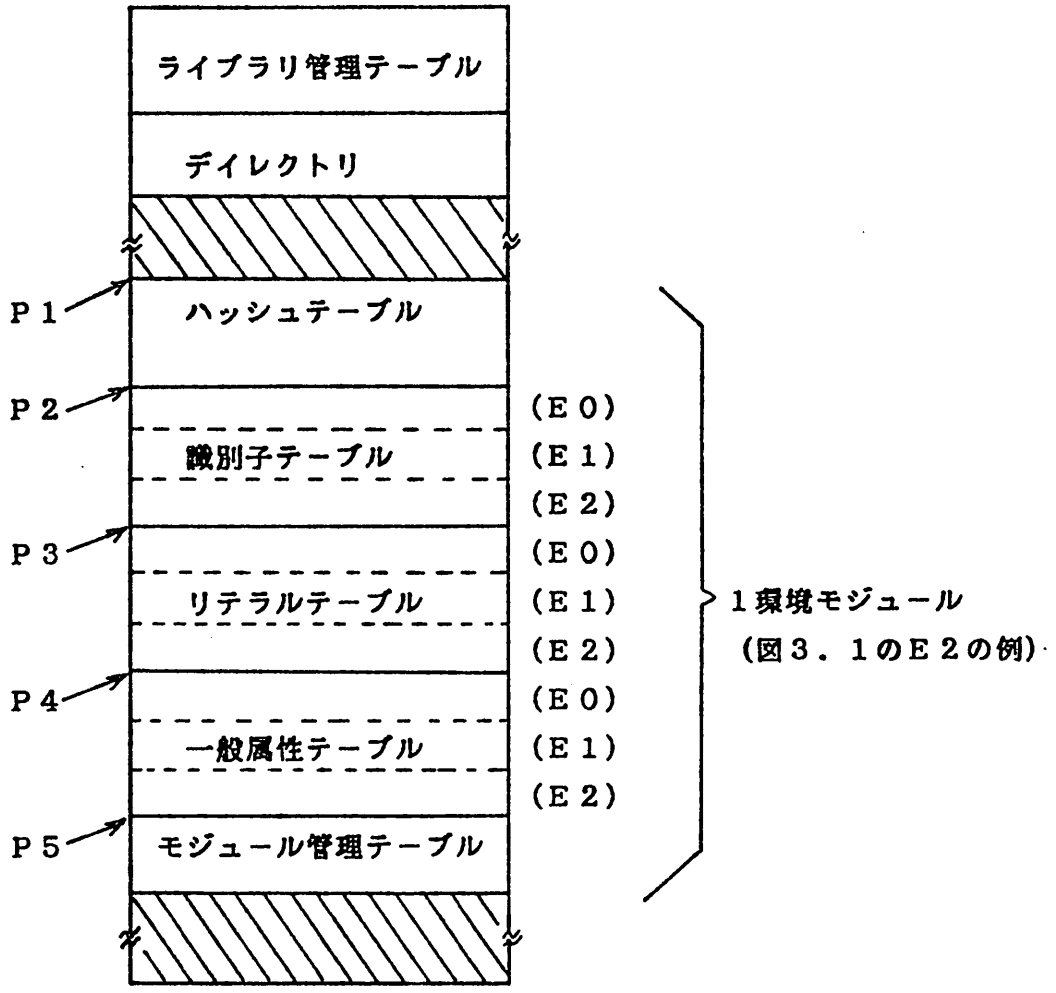


図５．１　ＳＰＬコンパイラの基本処理方式



(a)　EMLのストレージマップ

モジュール名 ('E2')	親モジュール（E1）のディレクトリアドレス	作成番号	P1	P2	P3	P4	P5

(b)　ディレクトリの1エントリ

リテラルテーブルへのポインタ	ハッシュチェイン	基本属性	一般属性テーブルへのポインタ

(c)　識別子テーブルの1エントリ

図5．2　環境モジュールライブラリ（EML）の基本構造

メモリ属性など，基本属性に含まれない一般的な属性を保存するものである。最後尾のモジュール管理テーブルは，識別子テーブル，リテラルテーブル，一般属性テーブル内のデータについて，その対応する環境モジュール毎の境界を管理するものであり，5.5節で述べる手続きのインライン展開処理時に用いる。

なお，ＥＭＬ内のモジュール単位の情報を管理するディレクトリは図の(b)に示す構造とした。ここで，モジュール名は6文字以下なのでディレクトリ内に固定長フィールドを設けた。第2フィールドは親の環境モジュールのディレクトリを指すことにより，システム全体での環境モジュールの木構造を表現している。第3フィールドは，手続きのインライン展開時の版管理に用いるもので後述する。他のフィールドは各テーブルの先頭アドレスを示す。

5.3.3　ＰＭＬの内部構造

処理モジュールは，環境モジュールの場合と同様に，その上位環境情報をＥＭＬから取込み，これを初期環境としてコンパイルする。この時，処理モジュール内で定義されている複数の手続きを連続的に処理するために，各手続きのコンパイル開始時に初期環境を設定し直さなければならない。しかし，そのために上位環境情報を毎回ＥＭＬから取込む必要はなく，ハッシュテーブルだけ初期状態に戻しておけば良いので処理は簡単である。

このコンパイル結果を保存する処理モジュールライブラリ（ＰＭＬ）の内部構造は図5.3のようにした。ＥＭＬと同じ部分が多いが，さらに手続きの中間語プログラムとインターフェイステーブルが加わる。このインターフェイステーブルは，手続き名，引数と返す値のデータ型などの情報を保持し，手続き引用時のデータ型チェックに用いられる。本テーブルは，各手続きを定義している処理モジュールとは独立に保存されるが，それは，

(1)　手続きがその定義に先じて引用される時は，インターフェイス仕様だけが保存される，

(2)　手続きは，その定義モジュールの位置には関係なく，どこからでも引用可能である，

などの理由による。

このインターフェイステーブルの構造を示す図5.3(b)において，手続き名を構成する単語の並びは木構造形式で表現している。これは，ＳＰＬのように手続き名を自然語風に記述できるようにした場合，名前の前半が同じで後半が異なるものが多くなるのでその識別処理を速くするためである。

5.3.4　データ型テーブル

データ型テーブルは，ユーザが定義するデータ型のインターフェイス情報とその定義本体を保存するもので，手続きのインターフェイステーブルと同様の理由で，その定義モジュールとは独立にしている。また，データ構造もほぼ同じである。なお，データ型の定義モジュールが削除された時には，本テーブルへの登録を取消す必要があるので，本テーブル側で定義モジュール名を保持している。そこで，本テーブルの各エントリには，次のような3種類の状態を記憶する領域がある。

(1) データ型の外部仕様宣言はされているが，未定義である。

(2) データ型が定義されている。

(3) データ型の定義モジュールが削除された。

これは手続きのインターフェイステーブルも同様であるが，特に(3)の状態は重要である。即ち，データ型や手続きの定義と参照のモジュールが異なる場合，分割コンパイル方式では，定義モジュールが削除された後も，参照モジュールから本テーブルへのポインタは存在するため，そのエントリは残しておく必要がある。

5.4 ユーザ定義データ型のチェック方式

言語機能としてデータ型定義機能を導入した場合には，それに関連して，型チェックの方法が問題となる。例えば，

```
type   T  = integer ;
var    A  : T ;
var    B  : integer ;
```

という宣言がなされている時に，代入文

```
A = B ;
```

があったとしよう。この場合，代入文の両辺のデータ型を同じとみれば正しい文として処理されるが，異なるデータ型と考えれば誤った文となる。即ち，この型チェックの方式は，データ型定義機能に大きな制約を与えることになるので重要な課題である。

一般に，データ型の等価性の判断基準としては，次の2方式がある。

(1) データ型名による識別

(2) データ型定義本体の構造による識別

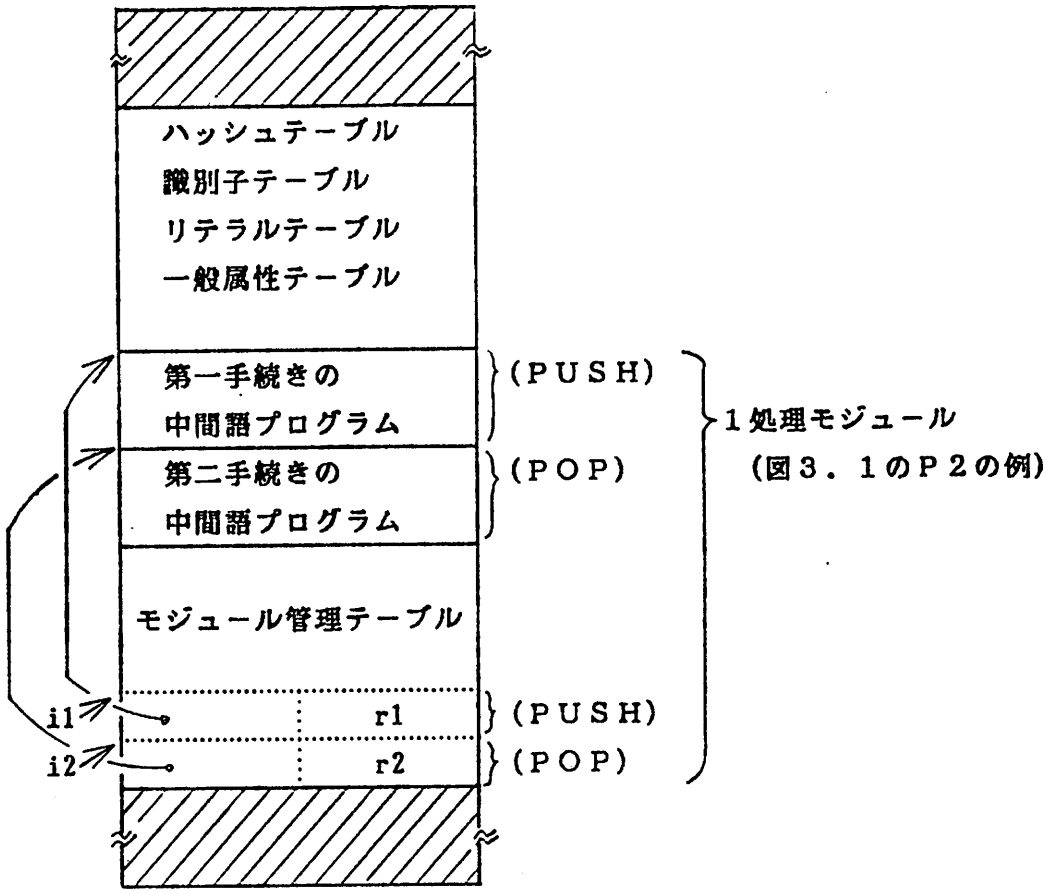
上の例の場合，(1)の方式では，Tとinteger は型名が異なるので誤った代入文となる。一方，(2)の方式では，Tはinteger と考え，正しい代入文となる。ＳＰＬと似たデータ型定義機能を有するPascalは，この点について言語仕様上はあいまいであるため，いずれの方式を採る処理系も存在する。また，PascalをベースにしたEuclidは(2)の方式を採っている。

しかしながら，ＳＰＬのようにモジュラープログラミングに用いられる言語では，データ型定義機能は，データ抽象化や段階的詳細化などに使用される重要なものであり，一律にいずれかの方式を採用すると，その用途によって矛盾が生じる。この問題は本来，定義されたデータ型の使用目的に依存するものであり，基本的には次の規則が自然である。

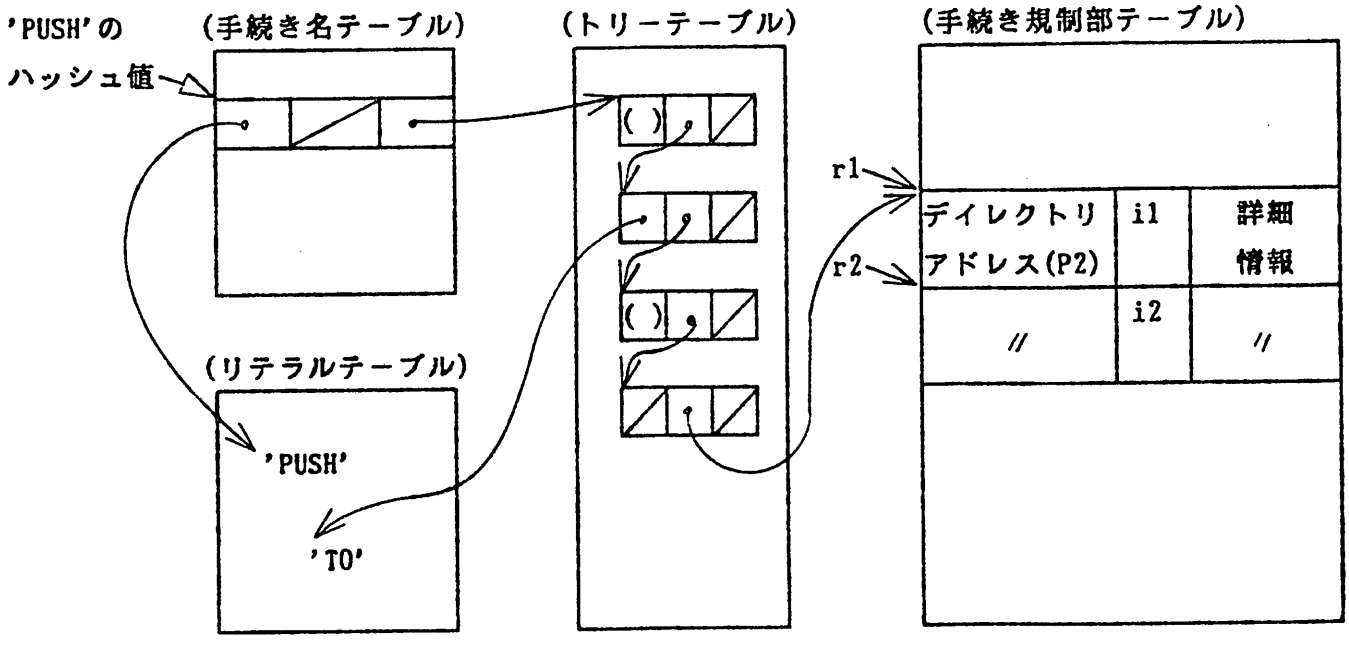
(1′) 使用者がそのデータ型の定義本体を知る必要がない時には(1)の方式，

(2′) 使用者がその構造を知る必要のある時は(2)の方式。

例えば，図4.1のＳＴＡＣＫ型のプログラム例の場合，処理モジュールＰ１ではそのＳＴＡＣＫ型データを利用する立場なので，そのデータ操作手続きさえ引用できれば，データの構造を知る必要はな



(a)　PMLのストレージマップ



(b)　手続きインターフェイステーブル

図5．3　処理モジュールライブラリ（PML）の基本構造

い。一方，処理モジュールＰ２では，そのデータ操作手続きを定義する立場なので，データ構造を知る必要がある。

そこで，ＳＰＬコンパイラでは，ＳＰＬの言語仕様上，データ型名はどこからでも引用できるが，その定義本体の参照は，その定義モジュールまたはその下位モジュールに限られることに対応して，次のような(1)と(2)の混合方式とした。

(1″) データ型名だけ引用できる所では(1)の方式

(2″) その定義本体の参照可能な所では(2)の方式

従って，ユーザ定義データ型変数の参照部分の意味解析処理時には，その変数参照場所とデータ型の定義場所の関係を認識する必要がある。しかしながら，5.3.4項で述べた，ユーザ定義データ型のインターフェイス情報と定義本体を保存するデータ型テーブルでは，その定義場所とモジュール階層の関係を直接的には表現していないため，この識別処理は複雑になる。そこで，この繁雑さを避けるため，その定義に使用されたデータ型名は，データ型テーブルの他に，その定義を含むモジュールの識別子テーブルにも登録するようにした。そのため，上記の識別処理は，変数の有効範囲のチェックと同様に識別子テーブル内の有無で判別できるので，容易である。

なお，特別のデータ抽象化機構を導入しているＣＬＵ，Alphard，Ada などでは，ユーザ定義データ型のチェックはデータ型名による識別方式を採っているが，そのデータ型の操作手続きの定義は特定の場所でのみ行うため，不都合は生じない。例えば，ＣＬＵでは，cluster と呼ばれるデータ抽象化機構を有し，データ型の定義と操作手続きの定義はcluster の中にまとめて行う。そして，操作手続きの中でデータ型の定義本体を参照する場合には，cvtという型変換指定を付加するような明示的方法を採っている。

5.5　手続きのインライン展開方式

プログラムをトップダウンに開発したり，機能分割を徹底し，それに対応したモジュール分割を行ったような場合には，一度だけ引用される手続きが多く出現する。このような手続きを外部手続きとして作成すると，手続き呼び出し時のリンケージオーバーヘッドのためにプログラムの実行速度が遅くなる。そこで，ＳＰＬでは，手続き本体をその引用部分にインライン展開するかどうかを手続きの定義時に指定できるようにしている。

このインライン展開処理は，5.2節で述べたように，解析処理後の中間プログラムに対して行うため，アセンブリ言語やＰＬ／Ｉにおけるテキストマクロの展開処理のように簡単ではない。即ち，通常の実引数と仮引数の置き換えの他に，引用される手続き側の環境情報と引用する側の環境情報を統合し，双方の中間語プログラムをこの統合された環境情報に関係付ける必要がある。

この簡単な処理方法としては，双方に共通な環境情報については引用する側のものを共有し，その他のものについては引用する側のものの後に引用される側のものを追加すると共に，中間語から識別子テーブルへのポインターの書き換えを行うことが考えられる。しかし，この方式では，中間語の走

査と書き換え処理に時間がかかる。特にインライン展開される手続き内でまたインライン展開すべき手続きが呼ばれた場合は，書き換え処理が増加するが，ＳＰＬの段階的詳細化機能を用いた場合はこのようなケースが多くなる。

そこで，ＳＰＬコンパイラでは，この処理を容易にかつ迅速に行うために，解析処理後の中間語から環境情報（識別子テーブル）へのポインタを次のような２要素で構成する２アドレス形式とした。

(1) 各環境モジュールおよび手続きに対応する識別子テーブルの先頭アドレスを保存するモジュール管理テーブル内の該当エントリへのポインタ。

(2) そのエントリが示す識別子テーブルの先頭アドレスからの相対アドレス。

そして，環境情報の統合はこのモジュール管理テーブルの書き換えによって行うことにより，中間語プログラムは全く変更しないで済むようにした。

この処理例を図5.4に示す。これは，図の(a)のようなプログラム例において，異なる処理モジュール内で定義された手続きＦ２を手続きＦ１内にインライン展開する時に，双方で参照している共通データＡの中間語が同じ環境情報のＡを指すようにする機構を示している。即ち，展開される側のモジュール管理テーブルの各エントリのうち，展開する側の環境と共通の部分については，そちらを指すように書き換える。また，独自の部分は後に付加してその番地を書き込む。そして，展開される手続きの中間語プログラム内の仮引数を実引数で置換しながら引用側の中間語プログラム内に埋め込む。この処理後，共通データのＡの参照に対応する中間語はすべて，同じ識別子テーブルの同じエントリを指すようになっている。

このような環境情報の統合方式により，中間語を走査して識別子テーブルポインタを書き換える必要がなくなり，インライン展開処理を効率良く行える。

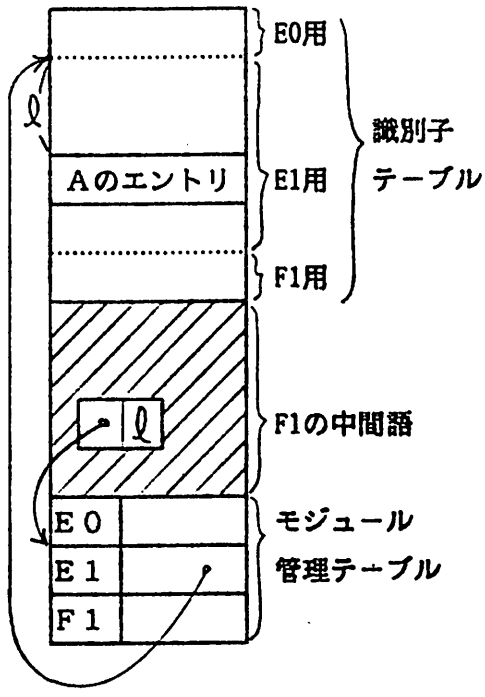
なお，この環境情報の統合時には，共通の環境モジュールについてはバージョンの一致をチェックする必要がある。即ち，5.3節で述べたように，各モジュールはその上位環境情報を自分の中に取り込んだ形でライブラリに保存されているため，同一環境モジュールの新旧のバージョンの混在が可能である。図5.4の例では，Ｐ１とＰ２が取り込んでいる共通環境モジュールＥ０およびＥ１のバージョンが同じである保証はない。そこで，図5.2(b)に示すように，ライブラリのディレクトリ内に作成番号フィールドを設け，ライブラリ化されるモジュールに通し番号を与えてここに保存する。そして，これを用いてバージョンの一致をチェックする。

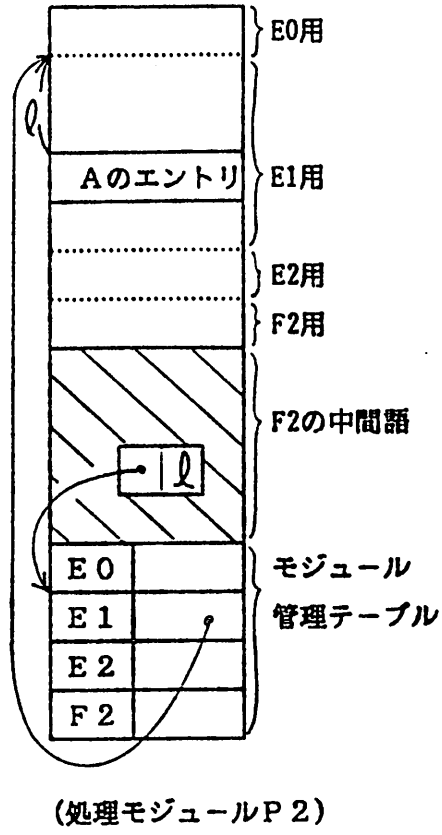
手続きのインライン展開処理におけるもう１つの問題は，異なる識別子が同じ名前を持つ場合の処理である。即ち，先の例において，双方の手続きＦ１とＦ２が同じ名前の局所変数を有する場合，コード生成処理時にそのままの名前でオブジェクトプログラムを作成すると２重宣言エラーが発生する。そこで，このような識別子については少くともどちらか一方の名前を変更する必要がある。その方法としては，

(1) 一方だけ変更する。

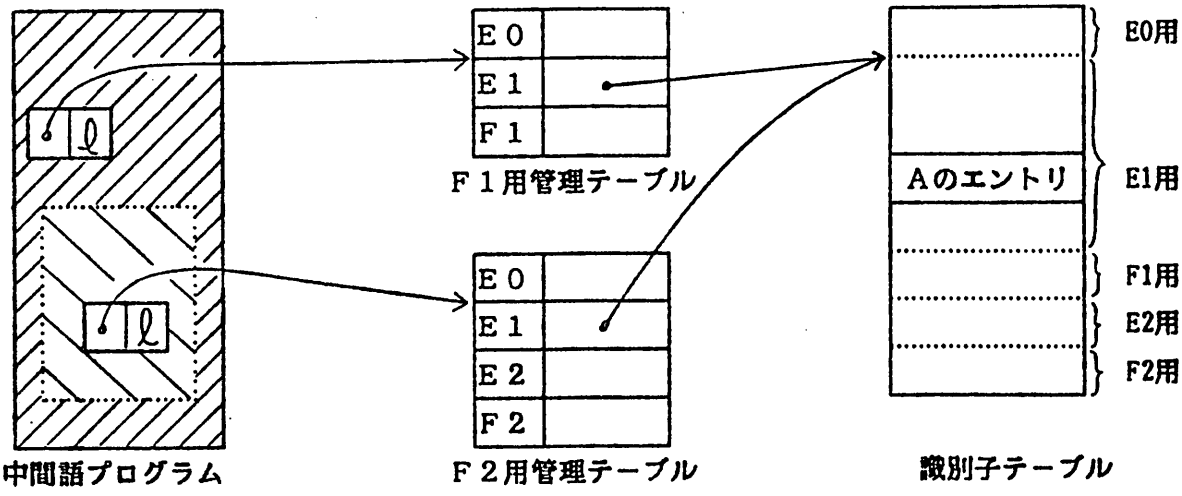
(2) 双方共に変更する。

```
environment E0;
 .....
environment E1(E0);
 .....
  var A;
 .....
process P1(E1);
  func F1 opt(sub);
    ...A...
      F2;
    .....
environment E2(E1);
 .....
process P2(E2);
  func F2 opt(open);
    ...A...
 .....
```





(a) プログラム例　　(b) PML内の処理モジュール



(c) 手続きのインライン展開処理後の内部構造

図5．4　手続きのインライン展開のメカニズム

(3)　名前の変更可能なものはすべて変更する。

などが考えられるが，コンパイラの処理容易性からは(3)が望ましい。また，デバッグはすべてＳＰＬソースプログラムのレベルで行うべきものなので，ＰＣＬ形式のオブジェクトリストをユーザに提供する必要はないため，(3)の方式で問題はない。しかしながら，実際には，プログラム開発時にオブジェクトリストを参照する場合が皆無とは言えないため，ＳＰＬコンパイラでは，できるだけソースプログラムの情報を残すという方針を採り，(1)の方式とした。

5.6　中間語からソースへの逆変換方式

5.6.1　コード生成方式

ＳＰＬコンパイラは，その早期開発の必要性から，オブジェクトコードを既存の高級言語である制御用Fortran（ＰＣＬ）とするプリプロセサ方式をとっている。そのため，コード生成処理時には，計算機向きの逆ポーランド記法に変換[N2)]された中間語を元のインフィックス形式で表現されたソース形式に戻す処理が必要である。例えば，元のソースプログラム内に，

Ａ＊Ｂ＋Ｃ／（Ｄ－Ｅ）

という式があった場合，ＳＰＬコンパイラの解析処理によって，

ＡＢ＊ＣＤＥ－／＋

という逆ポーランド記法に変換される。コード生成処理では，これを再び元に戻す逆変換処理を行う。

本節では，この逆変換処理の基本アルゴリズムおよびそれを改良した処理効率の良い逆変換方式について述べる。

5.6.2　基本アルゴリズム

まず，逆ポーランド記法の式の意味を示すために，その式の評価方法について述べる。即ち，式を左から右へ走査していき，演算子があればその左側のオペランドに対して演算を行い，その結果で置き換える。そして，更に走査を続けて，最後の演算の結果がその式の値となる。このアルゴリズムは，実際にはスタックを用いて次のような規則に基づいて実行される。

（Ａ１）　式を左から走査していき，識別子または定数があれば，スタックに積む。

（Ａ２）　２項演算子があれば，スタックの先頭とその次のオペランドに対して演算を行い，それらの代りにその演算結果をスタックに積む。

（Ａ３）　単項演算子があれば，スタックの先頭のオペランドに対して演算を行い，その結果をスタックに積む。

これは，逆ポーランド記法の式を直接計算する場合であるが，一般のコンパイラのように機械語オブジェクトを生成する場合は，演算を実行する代りにそのための機械語命令を生成すれば良い。

逆変換処理の場合も基本的には同じ手順で良いが，演算を実行したり，機械語命令を生成する代りに，インフィックス形式の演算式を生成する。その場合，演算子の優先順位を明確にするために，括

弧でくくることが必要で，具体的な処理は次のようになる。

（Ｂ１）　式を左から走査していき，識別子または定数があれば，スタックに積む。

（Ｂ２）　２項演算子があれば，スタックの先頭とその次のオペランドを取り出して，インフィックス形式の２項演算式を作成し，その全体を括弧でくくったものを新たにスタックに積む。

（Ｂ３）　単項演算子があれば，スタックの先頭のオペランドを取り出して，その直前に単項演算子を付けたものを括弧でくくり，新たにスタックに積む。

これは，最も単純な逆変換アルゴリズムであり，先の逆ポーランド記法の例である

ＡＢ＊ＣＤＥ－／＋

に本方式を適用すると，結果の出力は，

（（Ａ＊Ｂ）＋（Ｃ／（Ｄ－Ｅ））），

となる。

5.6.3　演算子の優先順位を考慮したアルゴリズム

先に述べた基本アルゴリズムは，例えば，加減算よりも乗除算を先に実行するというような演算子の優先順位を考慮していないため，出力結果に冗長な括弧が多く含まれる。これは，オブジェクトリストをユーザに提供しなければ問題ない。しかし，5.5節のインライン展開処理における名前替えの問題のところで述べたように，できるだけソースプログラムの情報を残すという方針からは好ましくない。

そこで，このような冗長な括弧の生成を避けるためには，先の基本アルゴリズムのＢ２，Ｂ３の規則適用時に，演算子の優先順位を考慮して括弧の必要性の有無を調べれば良い。その判別には，優先順位文法の構文解析に用いる優先順位関数を用いることができる。その例を表5.1に示す。例えば，先の規則Ｂ２はこの関数を用いて次のように変更すれば良い。

（Ｂ２′）　２項演算子ｘがあれば，スタックの先頭とその次のオペランドを取り出して，インフィックス形式の２項演算式を作成し，新たにスタックに積む。但し，以下の条件を満たすオペランドは括弧でくくる。

［左側のオペランドを括弧でくくる条件］

そのオペランドが演算子ａの２項演算式で，かつ，

ｆ（ｘ）＞ｇ（ａ）

［右側オペランドを括弧でくくる条件］

そのオペランドが演算子ｂの２項演算式で，かつ，

ｇ（ｘ）＞ｆ（ｂ）

なお，単項演算子を含めた全体のアルゴリズムについては，後で詳しく述べる。

この方式を先の逆ポーランド記法ＡＢ＊ＣＤＥ－／＋に適用した例を図5.5に示す。この例では，括弧の有無の判定はステップ８および９で行われる。即ち，ステップ８では，／演算子の右側オペラ

表５．１　優先順位関数

演算子 x	左側優先順位関数 g（x）	右側優先順位関数 f（x）
単項	１５	１５
巾乗演算子	１３	１４
乗除演算子	１２	１１
加減演算子	１０	9
比較演算子	8	7
否定	5	6
論理積	4	3
論理和	2	1

No.	適用規則	入力 x	出力スタック
1	B 1	A	
2	B 1	B	A
3	B 2'	*	A B
4	B 1	C	A * B
5	B 1	D	A * B C
6	B 1	E	A * B C D
7	B 2'	−	A * B C D E
8	B 2'	／	A * B C D − E
9	B 2'	+	A * B C ／ (D − E)
10			A * B + C ／ (D − E)

(注)　入力データ：　ＡＢ＊ＣＤＥ－／＋

図５．５　優先順位関数を用いた逆変換方式の適用例

ンドＤ－Ｅについて調べると，

ｇ（／）＝１２　＞　ｆ（－）＝９

となり，上記条件を満たすため括弧でくくる。ステップ９では，まず＋演算子の左側オペランドＡ＊Ｂについて調べると，

ｆ（＋）＝９　＜　ｇ（＊）＝１２

となり，上記条件に反するため括弧は不要である。次に＋演算子の右側オペランドＣ／（Ｄ－Ｅ）について調べると，

ｇ（＋）＝１０　＜　ｆ（／）＝１１

となり，同様に括弧は不要である。

本方式により，冗長な括弧の生成を避けることはできたが，処理効率上の問題が残されている。即ち，図5.5からわかるように，出力スタックに対する処理が多く，しかも，これらの処理はテキストの移動と編集が主であることから，処理時間を多く費すことが予想される。さらに，スタックの各要素の長さが可変長であるため，スタック管理の処理も複雑になる。そこで，ＳＰＬコンパイラでは，これらの問題を解決するために，次に述べるような方式を採った。

5.6.4　効率の良いアルゴリズム

先に述べた処理方式では，特に処理効率に関して，次のような重要な問題があった。

(1)　テキストの移動，編集を伴うスタック処理が頻繁に行われる。

(2)　スタックの各要素が可変長のため，スタック管理が複雑になる。

そこで，この問題を解決するために，逆変換アルゴリズムの有する次の性質に注目して，処理効率の良い方式を考案した。

(1)　逆ポーランド記法およびインフィックス記法において，定数および識別子の相互の相対的位置関係は同じであること。

(2)　インフィックス記法においては，定数および識別子の相互の間には，必ず１個以上の区切り記号が存在すること。

(3)　オペランドを括弧でくくる必要性の有無は，演算子の優先順位によっていること。

この性質を利用した処理方式は次のようなものである。即ち，逆変換アルゴリズム上で必要となるスタックの要素の形式を出力と同じにしていたものを改めるため，スタックと出力エリアを別々に設ける。そして，入力である逆ポーランド記法の式を走査していきながら，定数および識別子（またはその中間語）は出力エリアの方へ出力していく。この時，これらの間には，後で区切り記号を挿入するエリアを確保する。一方，スタックの各要素は，１オペランドに対応する出力エリアの要素列とその演算子コードを記憶する。そして，演算子を走査した時の処理としては，まず，スタックの先頭からの２要素に対応する出力エリアの間の区切り記号エリアに演算子を挿入し，新たに生成したオペランドに対応する要素をスタックに積む。

以下では，本方式の具体的な実現方法を幾つかの項目に分けて詳しく述べる。

〔スタックと出力エリアのデータ構造〕

スタックの各要素は，図5.6(a)に示すように，４つのフィールドで構成され，１つのオペランドを表わす。第１フィールドは演算子コードを保存し，オペランドを括弧でくくる必要性の有無の判断に用いる。例えば，オペランドが，単項，Ａ＋Ｂ，Ｘ＊Ｙ－Ｚなどの時，各々，単項，＋，－のコードになる。第２フィールドはこのオペランドが必要とする右括弧の数を表わす。例えば，Ａ＊（Ｂ＋Ｃ）ならば１となる。第３フィールドは，このオペランドに続く後のオペランドが必要とする左括弧の数を表わす。例えば，後のオペランドが（（Ｉ＋Ｊ）＊Ｋ－Ｌ）／Ｍならば２となる。第４フィールドは，このオペランドの実体が出力エリア上に存在する位置を示すポインタであるが，実際には処理の都合上，図5.6(c)に示すように，空の区切り要素分だけ後を指す。なお，これらのフィールドに各々，ＯＰ，ＲＰ，ＬＰ，ＰＴＲの略称を与える。

一方，出力エリアは，区切り要素と単項要素を１組にしたものの列とする。単項要素は定数および識別子を表わす中間語と考えれば良い。区切り要素は，図5.6(b)に示すように，３つのフィールドから成り，その両側にある単項要素間の区切り記号を表わす。即ち，第１フィールドは区切り記号のコードで，第２，第３フィールドはその左側および右側に必要な右括弧および左括弧の数を表わす。例えば，Ａ＊（Ｂ＋Ｃ）の＊の区切り要素は，＊コード，０，１となる。なお，各フィールドをＯＰ，ＲＰ，ＬＰと名付ける。

このようなスタックと出力エリアの関係は，例えば，図5.5の第５ステップの状態では図5.6(c)のようになる。

〔処理概要〕

処理手順の概要は図5.7に示す通りである。各部分の処理詳細は以下に述べる。

〔初期処理〕

式の最左端の左括弧の数を保存するために，図5.8(a)に示すように，スタックおよび出力エリア内に１要素を確保する。

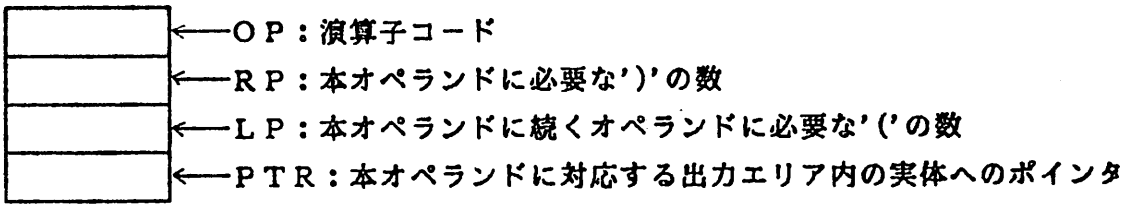
〔定数および識別子の処理〕

定数および識別子は，先に基本アルゴリズムの規則Ｂ１で述べたように，スタックに積むだけで良い。従って，ここでは，まず定数または識別子の中間語を単項要素として出力エリアに出力し，その後に，空の区切り要素エリアを確保する。スタックには，この単項要素をオペランドとするスタック要素を積む。その結果は図5.8(b)のようになる。

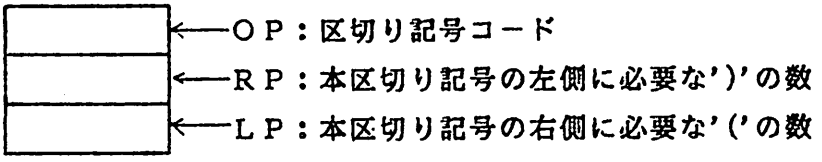
〔２項演算子処理〕

２項演算子が入力された時は，原則として，5.6.3で述べた規則Ｂ2′を行えば良い。本方式での処理手順は，図5.9に示すように，まず入力ｘとその第１オペランド（左側オペランド），即ちスタックの先頭から２番目の演算子コードＯＰ１の優先順位を比較し，

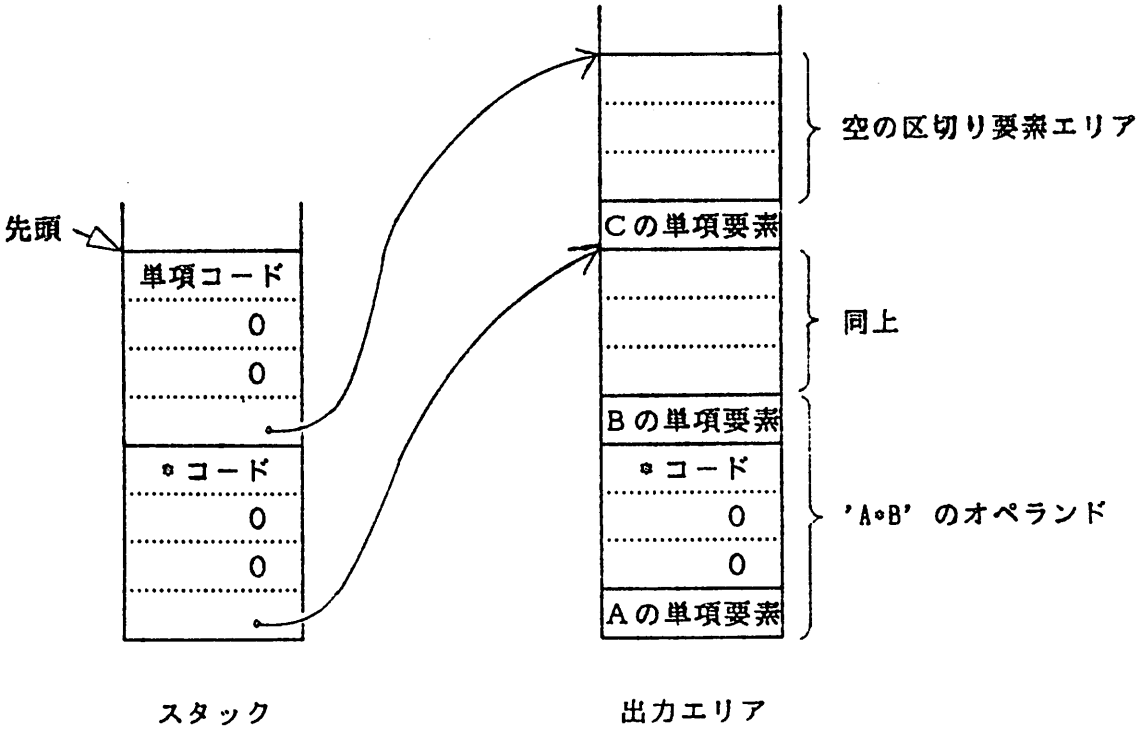
$$g(OP1) < f(x)$$



（ａ）　スタック要素の構造



（ｂ）　出力エリア内の区切り要素の構造



（ｃ）　例（'AB∘C'を入力済み）

図５．６　逆変換に用いるスタックと出力エリアのデータ構造

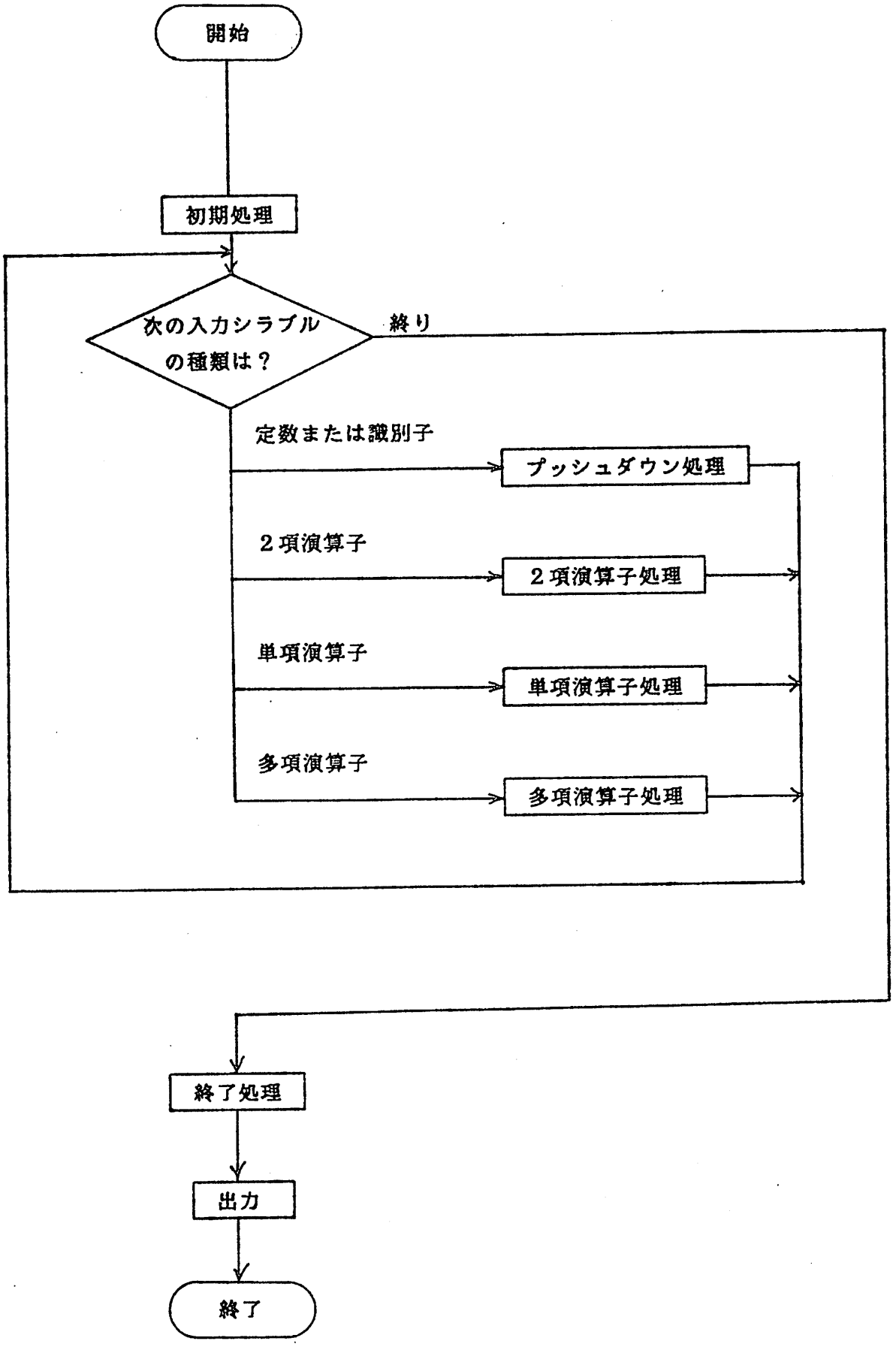
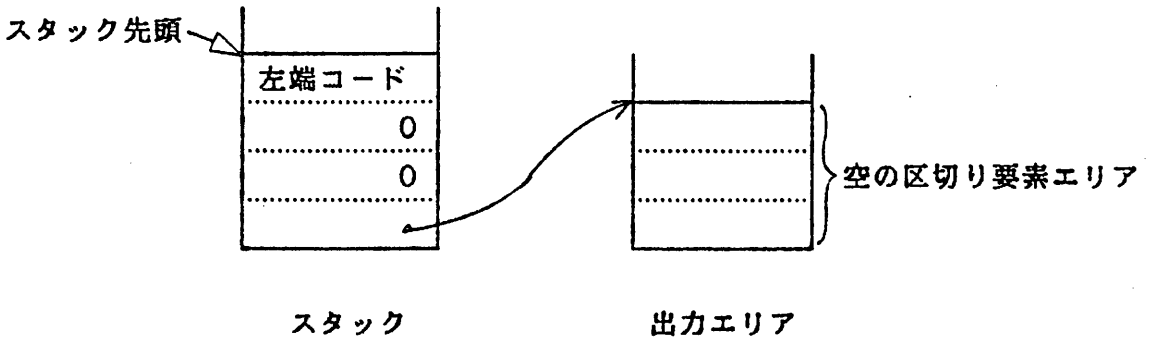
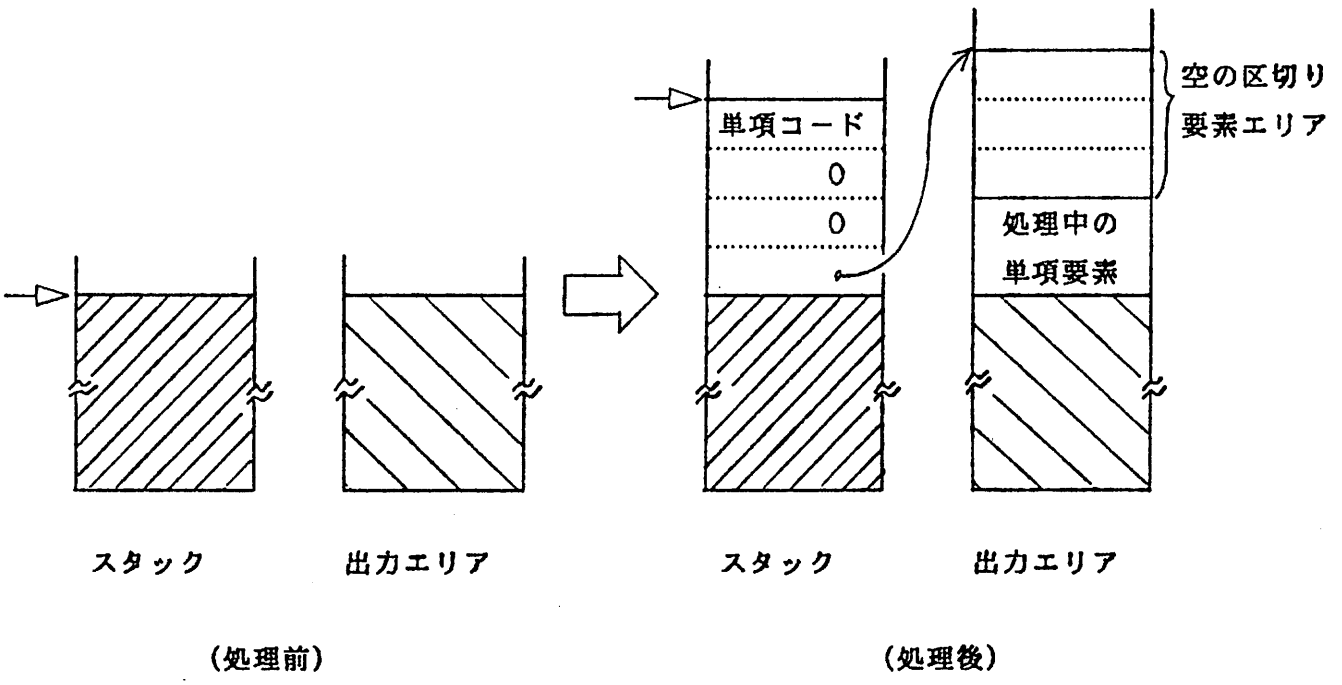


図５．７　逆変換の処理方式概要

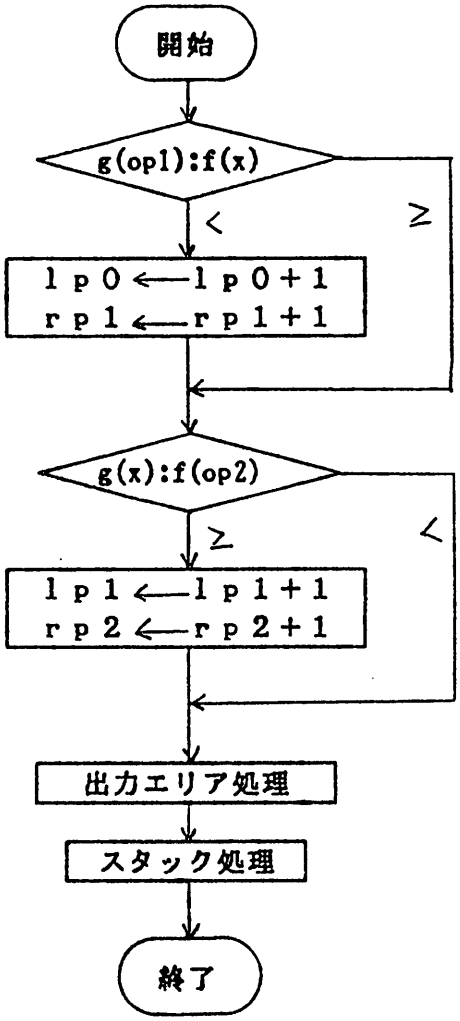


（a）　スタックおよび出力エリアの初期処理



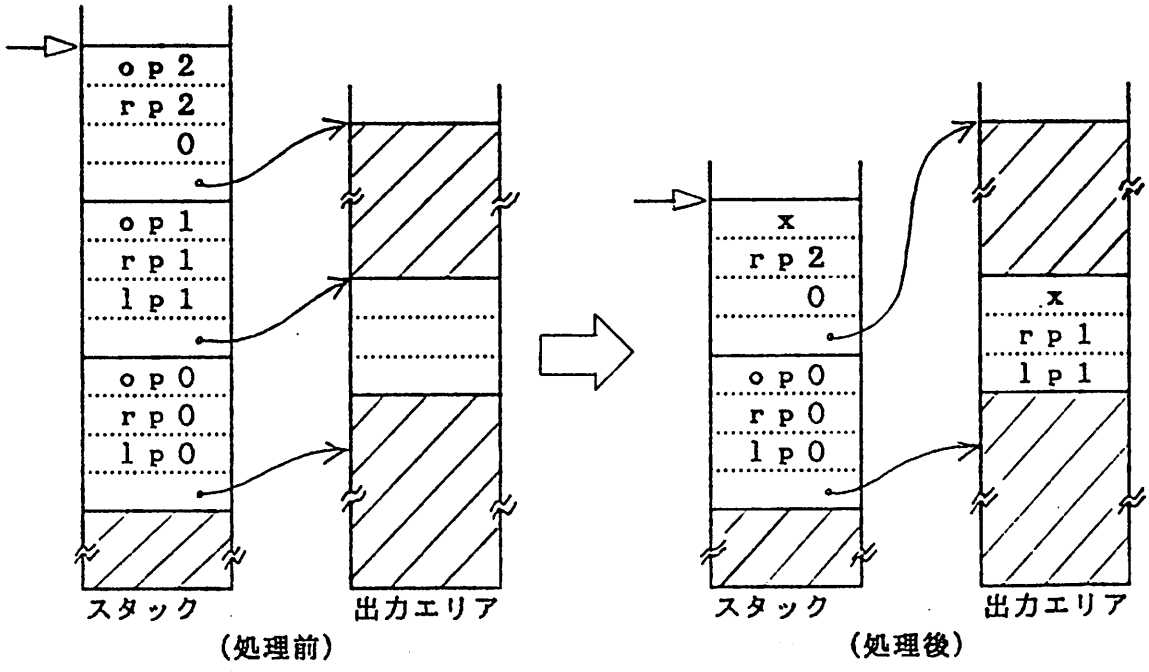
（b）　定数および識別子の処理

図５．８　初期処理および単項要素処理



f：右側優先順位関数
g：左側優先順位関数
x：入力された２項演算子コード
op2：スタック先頭の要素の演算子コード
op1：スタックの２番目の要素の演算子コード
lp1：スタックの２番目の要素のＬＰ
lp0：スタックの３番目の要素のＬＰ
rp2：スタック先頭の要素のＲＰ
rp1：スタックの２番目の要素のＲＰ

（a）　２項演算子処理手順



（b）　スタックと出力エリアの状態変化

図５．９　２項演算子処理

ならば，第１オペランドを括弧でくくる必要がある。そのため，その代りに第１オペランドのＲＰとスタックの先頭から３番目要素のＬＰを１だけ増加させる。次に，第２オペランド（右側オペランド）の演算子コードＯＰ２についても同様にして，

$$g(x) \geq f(OP2)$$

ならば，第２オペランドのＲＰと第１オペランドのＬＰを１だけ増加させる。

なお，判定処理に用いる優先順位関数は，前述の表5.1とする。一般の演算子では，同じものが隣り合った場合，左から右へ評価していくため，左関数ｇの方が右関数ｆよりも大きいが，巾乗（＊＊）では右から左へ評価するため逆になっている。単項演算子である否定演算子の場合も同様である。

このような括弧処理が終ると，出力エリアの第１オペランドと第２オペランドの間にある区切り要素エリアに，入力ｘのコードおよびスタック上の第１オペランドのＲＰとＬＰを書き込む。そして，最後にスタックから第１，第２オペランドを除き，代りに新しく生成したオペランド要素を積む。その様子は図5.9(b)に示す。

〔単項演算子処理〕

ＳＰＬの場合，単項演算子としては，＋，－，．ＮＯＴ．（否定）があるが，＋については解析処理時に逆ポーランド記法に変換する時に削除している。

単項演算子の処理は，基本的には２項演算子の場合と同じであるが，１つの大きな相違は，単項演算子の場合は出力エリア内に区切り記号が連続して現れることになるということである。例えば，Ａ＋（－Ｂ＊Ｃ）は，ＡとＢの間に２個の演算子が存在する。そこで，この式の逆ポーランド記法であるＡＢＣ＊@＋（但し，@は－の単項演算子）を処理する場合，出力エリアのＡとＢの間には２個の区切り記号エリアを確保する必要があるが，Ｂを入力した時点ではその個数がわからない。そこで，この場合には残念ながら出力エリア内のデータの移動が必要である。その様子と処理手順については図5.10に示す。

〔多項演算子処理〕

一般に式の項として配列要素や関数引用が含まれるが，これらは逆ポーランド記法で表現する場合は多項演算子として扱う。例えば，Ｓ（Ａ，Ｂ，Ｃ）はＳＡＢＣf_4とする。ここで，f_4は４つのオペランドを有する多項演算子である。

この処理としては，f_4が入力された時，出力エリア内のＳ，Ａ，Ｂ，Ｃの各々の間に確保された空の区切り要素エリアに，各々，左括弧，幾つかのコンマ，そして右括弧のコードを挿入すれば良い。この時，各々のＲＰ，ＬＰエリアに，スタック内の対応する要素のＲＰ，ＬＰの値をセットした後，これらのスタック要素を除き，代りに新たに作成したオペランドを単項要素として積む。

〔終了処理〕

入力された式の走査処理終了時のスタックには，初期処理で作成した左端コード要素も含めて２要素が残っている。これらは，各々，式全体の左端および右端に必要な左括弧と右括弧の数を保存している。従って，これらの値を，各々の要素が指す出力エリアの区切り要素に書き込むことにより，出

力エリアが完成される。

〔出力処理〕

これまでの処理結果である出力エリアを先頭から走査していき，各要素について以下の処理を施せば，最終目的であるインフィックス形式の式が得られる。

(1) 先頭の左端コード要素のＬＰの数だけ左括弧を出力する。

(2) 以降の各要素に対して，

(i) 単項要素ならばその名前を出力する。

(ii) 区切り要素ならば，

(a) まず，ＲＰの数だけ右括弧を出力する。

(b) 次に，区切り記号コードに対応する区切り記号を出力する。

(c) 最後に，ＬＰの数だけ左括弧を出力する。

(3) 出力エリアの最後尾の区切り要素のＲＰの数だけ右括弧を出力する。

5.7 結　言

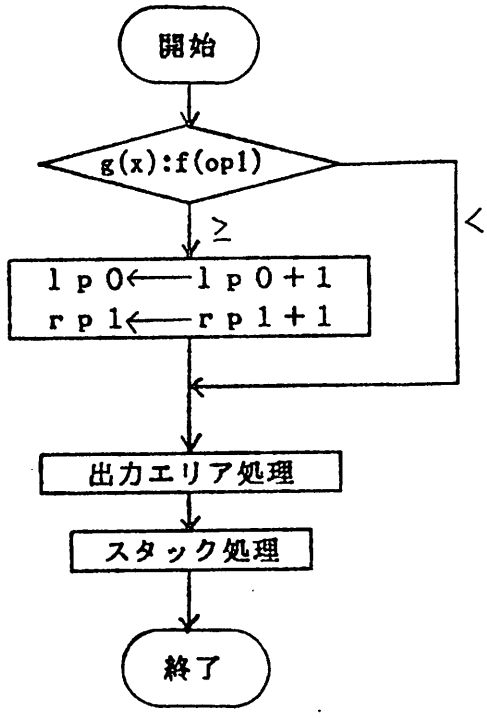
新言語ＳＰＬは，従来のブロック構造を変形して，共通データの独立化と階層化を基本にしたモジュール階層構造を導入し，この上に段階的詳細化やデータ抽象化などの構造化プログラミング機能を実現している。そのため，ＳＰＬコンパイラの開発に際しては，従来技法とは異るコンパイラ作成技法が必要とされた。

その第１は，プログラムのトップダウン開発およびその各段階での完全な型チェックを実現することであり，そのために，手続きのインライン展開を解析処理後の中間語プログラムに対して行うような基本処理方式を設定した。そして，各モジュールは解析処理後には共通ライブラリに保存され，他モジュールのコンパイル時に必要に応じて参照可能とすることにより，システム全体での無矛盾性を確保するようにした。

第２には，そのような基本処理方式の中で効率的な分割コンパイルを実現するために，共通ライブラリのデータ構造に工夫をこらした。まず，環境モジュールについては，その環境情報の参照手順が階層構造のために複雑化するのを避けるために，各モジュールの解析処理時には，その上位環境情報を初期環境として取り込み，あたかも自分のモジュール内の情報のように扱う方式を実現した。また，処理モジュールについては，その中で定義されている手続きが常に外部手続きとしてどこからでも引用可能であるため，処理モジュールの削除や再コンパイルに伴なうシステム全体の無矛盾性のチェックが複雑であった。そのため，手続き本体の中間語プログラムと手続き名や引数のデータ型などの外部インターフェイス保存用テーブルを分離し，関連処理が迅速に行えるようにした。なお，ユーザ定義データ型についてもほぼ同様の処理方法をとった。

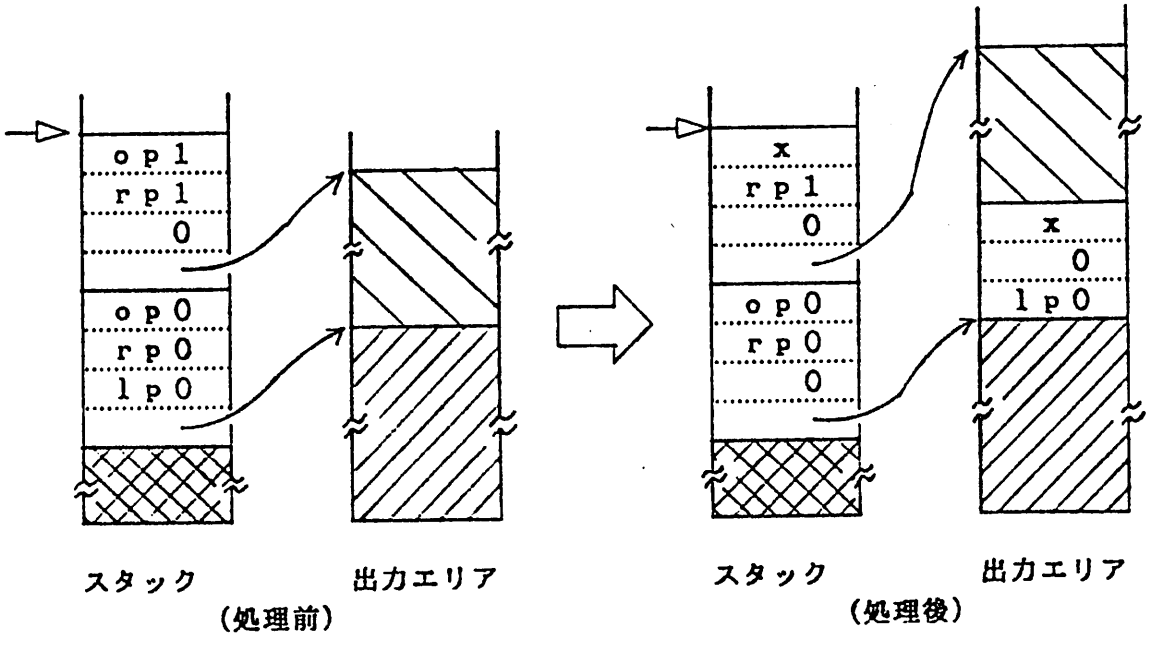
しかしながら，このような分割コンパイル方式では，次のような欠点が存在する。

(1) 共通ライブラリのメモリ容量が大きい。



x：入力された単項演算子コード
op1：スタック先頭の要素の演算子コード
lp0：スタックの２番目の要素のＬＰ
rp1：スタック先頭の要素のＲＰ

（a）　単項演算子処理手順



（b）　スタックと出力エリアの状態変化

図５．１０　単項演算子処理

(2) 上位モジュールの修正時にその下位モジュールの再コンパイルが必要になる。

第1項は，上位環境情報を初期環境として取り込む方式により，上位の環境情報が幾つも複写されて，共通ライブラリ内に複数個存在することに起因する。この問題はコンパイル効率とのトレードオフの関係にあり，ＳＰＬコンパイラが稼動する計算機システムの条件に応じて方式変更が可能である。一方，第2項は，システム全体の無矛盾性を保証するために必要なことであり，高信頼化プログラミングがＳＰＬ開発の目的でもあるため，止むを得ない。従って，無矛盾性を保証しながら，部分的再コンパイルを実現する方式は今後の課題である。

次に，ＳＰＬコンパイラの第3の特徴として，ユーザ定義データ型の同一性のチェック方式がある。即ち，従来のように，データ型名による識別方法およびデータ型定義本体の構造による識別方法のうちのいずれか一方を採用すると，いずれの場合もデータ型定義機能の用途によっては矛盾が生じる。そこで，ＳＰＬでは，データ型名のみ参照可能な所は前者の方法，またその定義本体の参照可能な所では後者の方法を適用するような混合方式を採用して，上記矛盾を解消した。

第4には，中間語プログラムレベルでの手続きのインライン展開を効率良く行うため，中間語から識別子テーブルへのポインタを2アドレス方式とした。即ち，モジュール単位の識別子テーブルの先頭アドレスを保存するモジュール管理テーブルへのポインタとその先頭アドレスからの相対アドレスによって特定の識別子テーブルエントリを指定する。そのため，手続きのインライン展開時には，このモジュール管理テーブルだけ書き換えれば，中間語プログラム自身の書き換えが不要になり，処理効率が良い。

最後に，第5の特徴として，逆ポーランド記法の中間語からインフィックス形式への変換方式がある。これは，ＳＰＬコンパイラのオブジェクト言語を既存の高級言語ＰＣＬとするプリプロセッサ方式のために必要とされたものである。これは，通常の逆ポーランド記法の式の評価手順と同様にスタックを用いて実現できるが，単純に行うと冗長な括弧が生成される。そこで，まず，演算子優位文法の構文解析技法を応用して，演算子の優先順位関数を用いることにより，冗長な括弧の生成を防いだ。しかしながら，この方式においても，テキストの移動や編集を伴なうスタック処理が多いことや，スタックの各要素が可変長のためにその管理が複雑になることなどの処理効率上の問題は解決していない。そこで，逆ポーランド記法およびインフィックス記法において，定数および識別子の相互の相対的位置関係は同じであることや，インフィックス記法においては定数および識別子の相互の間には必ず区切り記号が1個以上存在することなどに注目した，処理効率の良い方式を考案した。即ち，定数および識別子は直接に出力エリアに出力すると共に，それらの間に後から区切り記号を挿入するためのエリアを1個分いつも確保する。そして，各区切り記号の前後に必要な括弧の数は，この区切り記号用エリア内に保存するようにした。この方式により，スタック処理やテキスト操作数を大巾に削減できた。

なお，本方式においても，単項演算子が存在する場合は，区切り記号用エリアを連続的に複数個確保する必要があるため，出力エリアに最後に出力した単項を後へ移動する処理が生じる。この移動処

理を避けるためには，単項を出力エリアに出す前に，次の入力要素が単項演算子か否かを先読みをすれば良い。しかし，この先読み方式は，入力が補助記憶装置にある場合は処理が複雑化するため，ＳＰＬコンパイラでは採用していない。従って，本方式の具体化は今後の課題である。