段階的詳細化，データ抽象化を支援する言語 SPL のコンパイル技法†

中所武司†† 野木兼六††
林 利弘††† 森 清三†††

ソフトウェアの信頼性および生産性向上のためのプログラミング方法論として，プログラムの構造化，データ抽象化などの研究が行われ，すでにサポート言語もいくつか提案されている．ところが，これらの言語処理系の開発時に新たに必要になるコンパイル技法は明らかでない．

本論文では，プログラムの段階的詳細化，データ抽象化などをサポートするために新たに開発された構造化プログラミング言語 SPL の処理系のコンパイル方式について述べる．

まず，厳しいデータ型チェックと良いコンパイル効率を確保しながら分割コンパイルを実現するために，モジュール間情報を保存するライブラリが導入され，そのデータ構造が工夫された．また，オブジェクト効率の低下を防ぐため，手続きのインライン展開とコンパイル時実行機能を備えたが，定義よりもその参照が先行するトップダウン開発を可能とするために，これらの処理はソースプログラムの解析処理後に行うようにした．ユーザ定義データ型の同一性のチェックは，その用途に合わせて名前による場合と構造による場合を使いわけた．

最後に，コンパイル時間とオブジェクト長に関する性能解析を行い，上記コンパイル技法の効果を確認した．

1. まえがき

ソフトウェアの信頼性および生産性向上のためのプログラミング方法論は，ソフトウェア工学の主要テーマの1つであり，プログラムの構造化，データ抽象化などの研究[1]が盛んである．これらの技法を適用して開発するプログラムは，いくつかの理解容易なモジュールに分割され，プログラミング，テスト，保守を容易にするほか，情報のカプセル化により信頼性も向上するなどの効果が期待される．

このような技法としては，これまで，Dijkstra の階層構造設計[2]，Wirth 等の段階的詳細化法[3),4]，Parnas の情報隠ぺいの概念によるモジュール分割法[5]，Myers の複合設計[6]，Liskov のデータ抽象化[7]などが提案されている．

一方，これらの方法論を支援するプログラミング言語としては，Simula 67 の影響を受けている CLU[7]，Alphard[8]，Mesa[9]，あるいは Pascal を基本にした Modula[10]，Euclid[11]などが提案されている．また，現在米国国防総省によって開発が進められているリアルタイム用高級言語 DoD-HOL（新名称 Ada）の要求仕様書[12]でもカプセル化機能が必須とされている．

しかしながら，これらの言語の処理系の開発に際しては，従来の Fortran コンパイラなどと異なり，相互に関連の深い各モジュールの分割コンパイルと厳しい型チェックを両立させる処理方式，ユーザ定義データ型の処理方式，あるいは，オブジェクト効率とコンパイル速度の向上策などに関して新しいコンパイル技法が要求されると思われるが，処理系の開発された言語が少ないこともあり，まだこれらの技法は明らかではない．

本論文では，筆者らが新たに開発した，段階的詳細化技法やデータ抽象化技法などを支援する構造化プログラミング言語 SPL (Software Production Language) の処理系における上記問題の解決方法について述べる．

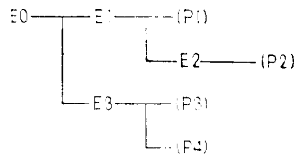
2. 構造化プログラミング言語 SPL

SPL は，段階的詳細化によるトップダウン技法，データ型とそのオペレーション集合を合せてカプセル化するデータ抽象化技法などをサポートするプログラミング言語であり，共通データの宣言の独立化と階層化によってプログラムの階層構造を表現する点に大きな特徴がある．

すなわち，SPL は環境モジュールと処理モジュー

† Compiling Techniques for SPL Programs Based on Stepwise Refinement and Data Abstraction by TAKESHI CHUSHO, KENROKU NOGI (Systems Development Laboratory, Hitachi, Ltd.), TOSHIHIRO HAYASHI, and KIYOMI MORI (Omika Works, Hitachi, Ltd.).

†† (株)日立製作所システム開発研究所
††† (株)日立製作所大みか工場



En : an environment module
(Pn): a process module

図 1 SPL プログラムにおけるモジュール構造の例
Fig. 1 A sample structure of an SPL program.

```
   ......
process P1(E1);
  func EVAL opt (sub);
    var S1: STACK (100);
    ......
    PUSH (VALUE) TO (S1);
    ......
end P1;
environment E2(E1);
  specification;
    func ERROR (CODE: int) opt (sub);
  end;
  declaration;
    type STACK (LENGTH: int)
               =(TOP: int,
                 STK (LENGTH): real);
  end;
end E2;
process P2 (E2);
  func PUSH (V) TO (S) opt (open);
    par V: real,
        S: STACK (*);
    S. TOP=S. TOP+1;
    % if LEVEL
      is 1 then
            if S. TOP. gt. LENGTH
              then ERROR (E013);
              else S. STK (S. TOP)=V;
            end;
      is 2 then S. STK (S. TOP)=V;
    end;
  end PUSH;
   ......
end P2;
```

図 2 SPL におけるモジュール定義の例
Fig. 2 Samples of SPL modules.

ルの2種類のモジュールを有し，環境モジュールでは変数，定数の宣言，データ型の定義などが行われ，これらのスコープを正確に表現するために，環境モジュールは図1のような木構造を構成する．一方，処理モジュールは手続きの集合から成り，その機能を果すのに適した環境モジュールの下に位置する．SPLのプログラム例として，図1のモジュール P1，E2，P2の定義の一部を図2に示す．

このような言語 SPL が有する特徴的な機能の中で，特にその処理系に新たなコンパイル技法が必要となるものは，主に，

(1) 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) 手続きのインライン展開機能
(4) コンパイル時実行機能
(5) 外部参照手続きとデータ型の仕様明記機能

である．これらのうち(1)は先に述べたようにプログラムの階層構造を構成するものである．(2)はトップダウン開発時のデータの段階的詳細化に必要なもので，図2の例では，モジュールP1で新しいデータ型 STACK とその処理手続き PUSH TO が導入され，その下位モジュール E2，P2 で各々の定義が行われている．この場合，SPL では，STACK 型の定義本体はそれを定義しているE2の下位モジュールからのみ参照できるので，これらのE2とP2はデータ抽象化の例でもある．

(3)と(4)はモジュール化されたプログラムのオブジェクト効率向上のためのもので，図2のP1内で引用された手続き PUSH TO はオプション指定が **open** なのでインライン展開される．この時，% **if** 文が実行されて，エラーチェックの要否に応じたオブジェクトを出力する．これらは PL/I のコンパイル時機能に似ているが，SPL はトップダウンプログラミングを志向しているため，定義よりもその参照が先行する場合が多く，これらの処理は個々のモジュールの解析処理後に行われる必要がある．

(5)はコンパイル時の型チェックを厳しく行うためのもので，図2のP1内で引用された手続き ERROR は未だ定義されていないので，E2でそのインタフェース仕様が明記されている．

3. 言語処理系の特徴

3.1 処理方式概要

SPL コンパイラの処理内容は大別すると以下の4項になる．

解析：ソースプログラムの構文解析と意味解析．
展開：手続きのインライン展開．
解釈：コンパイル時実行文の解釈実行．
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．そこで，処理方式の設計時に考慮すべき課題として，

（1） 解析，展開，解釈の処理順序．

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

がある．

第1の点については，たとえば，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すでに定義されている必要があり，ボトムアップ的な開発を志向する言語では可能であるが，SPL のように展開されるモジュールよりもそれを引用するモジュールの方が先に定義されることの多いトップダウン志向言語には適さない．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のは，エラーの発見が遅れるほか，展開されるモジュールは機能的断片でも良いためドキュメント性が悪くなる．

そこで，ここではまず初めに解析処理を行うこととし，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評価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ことにした．

第2の点については，手続き，データ型の定義や変数，定数の宣言を行うモジュールとそれらを参照するモジュールが異なることが多く，解析処理が別々に行われること，特に手続きとデータ型については，図2のようなトップダウン開発の場合に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ことなどのために，定義および宣言と参照の対応づけは複雑な問題である．

ここでは，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，その管理を強化することでこれらの問題の解決を計ったが，詳細は後述する．

以上のような観点から設計された SPL コンパイラの基本処理方式を**図3**に示す．環境モジュールおよびインライン展開される手続きの定義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情報を用いて解析処理を行い，結果を各々のライブラリに登録する．

オブジェクトプログラムの出力を必要とする処理モジュールに対しては，同様の解析処理後，SPL ライブラリからインライン展開手続きの本体を取込みながら展開および解釈処理を行い，最後に生成処理を行う．

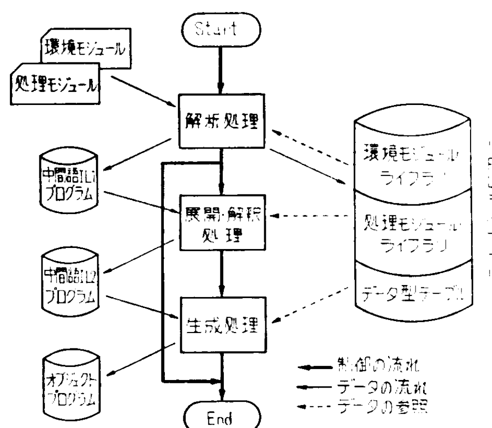


図3 SPL コンパイラの処理の流れ
Fig. 3 Process flow of the SPL compiler.

3.2 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で大きな効果をもたらすが，こうした分野では，プログラムのデバッグ，テスト，修正を効率良く行うために，モジュール単位でコンパイルできる必要がある．たとえば，現在，大学を中心に急速に普及しつつある Pascal は，もともと教育用に設計されたため，一括コンパイルを前提にした言語であるが，実用的見地から，モジュール化機能を付加して分割コンパイルを可能にする拡張が行われている[15),16)]．

なお従来の言語処理系は外部手続き単位で独立にコンパイルできるものが多いが，共通データをその都度宣言したり，実引数の型チェックをしないなど，プログラムの信頼性低下要因が多く，各モジュールを外部手続きとして作成するとオブジェクト効率も悪いため，モジュラプログラミングには適さない．

SPL では，これらの欠点を除き，かつ分割コンパイルを可能にしているが，その代償として各モジュールの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．そこで，SPL コンパイラでは，各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，モジュール間情報の受渡しはこれを経由して行うようにした．このような方式の必要性は，CLU[7)] や DoD-HOL[12)] でも言及されている．

SPL ライブラリは，主に，

- 環境モジュールライブラリ (EML)
- 処理モジュールライブラリ (PML)
- データ型テーブル

から構成されるが，これらは頻繁に参照されるため，その内部構造がコンパイル処理効率に与える影響は大

きい．

（1） 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重要である．最も単純な方法として，個々のモジュールを独立にコンパイルし，木構造は別に保存した場合，変数名や定数名がどこで宣言されたものかを調べるために下位のモジュールから順に探すことになり，処理が遅くなる．

そこで，本システムでは，トップダウン型開発の利点を生かし，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ライブラリから取込み，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．したがって，EML 内の各モジュールはあたかもその上位環境情報がすべて自分の中で宣言されていたようになるため，迅速な参照が可能である．

すなわち，EML は**図4**のような構造で，識別子テーブルには，自分およびその上位環境モジュールで宣言されたすべての識別子が含まれ，1つのハッシュテーブルを経由してアクセス可能になっている．そして，同名の識別子間では後から登録された方に先に出会うようにハッシュチェインを結ぶ．したがって，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．

なお，図中のモジュール管理テーブルは，識別子テーブル内のモジュールごとの境界を管理するもので，

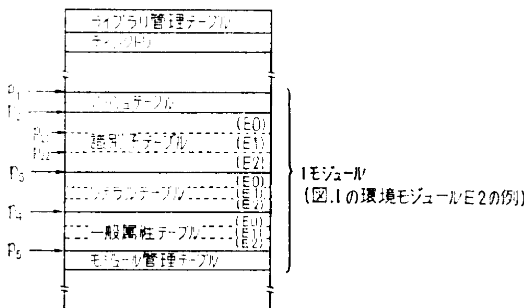


図 4 環境モジュールライブラリ (EML) の基本構造
Fig. 4 Basic structure of EML.

3.4 で述べるインライン展開処理時に用いられる．

（2） 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じように，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．ただ，処理モジュール内に複数の手続きが定義されている時は，各々のコンパイル開始時に初期環境を設定し直す必要があるが，2度目からはハッシュテーブルだけ元に戻せば良いので処理は簡単である．

コンパイル結果を保存する PML の内部構造は EML と似ているが，さらに手続きの中間語プログラムとインタフェーステーブルが加わる．このインタフェーステーブルは，手続き名，引数と返す値のデータ型，手続き本体へのポインタなどの情報を管理し，手続き引用文のデータ型チェックに用いられる．本テーブルは処理モジュールとは独立に保存されるが，それは，

（i） 手続きがその定義に先んじて引用される時は，そのインタフェース仕様だけが保存される，

（ii） 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．

（3） データ型テーブル

本テーブルは，ユーザが定義するデータ型のインタフェース情報と定義本体を保存するもので，手続きのインタフェー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なっている．

3.3 ユーザ定義データ型のチェック

データ型定義機能を有する言語の処理系の問題として，それに関連するデータ型チェックの方法がある．たとえば，

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

と宣言されている時，代入文

A＝B

を両辺のデータ型が異なるエラーとするか否かの問題である．

データ型の等価性の判断基準としては，一般に，

（i） データ型名による識別，

（ii） データ型定義本体の構造による識別，

の2種類が考えられている[17]．上の例の場合，(i)ではエラー，(ii)では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 はこの点があい

まいであるが，その系統の Euclid[11], Stony Brook Pascal/360[15]などは(ii)，Mesa[9]は(i)の方式である．

しかしながら，SPL などのモジュラプログラミング用言語においては，データ型定義機能はデータ抽象化，情報のカプセル化などに用いられる重要なものであり，一律にいずれかの方法をとると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形態に依存するもので，基本的には，

(i′) 使用者がそのデータ型の定義本体の構造を知る必要のない場合は(i)の方式，

(ii′) 使用者がその構造を知る必要のある場合は(ii)の方式，

とするのが自然である．たとえば，図2のユーザ定義データ型 STACK に関するプログラム例で，処理モジュールＰ１ではスタックを利用する立場なので，その処理手続きは引用してもデータ構造を知る必要はないが，処理モジュールＰ２ではスタックの処理手続きを定義しているので，その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データ型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，

(i″) データ型名だけ引用できる所では(i)の方式，

(ii″) その定義本体の参照可能な所では(ii)の方式，

とした．

3.4 インライン展開

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 SPL のインライン展開は，3.1 節で述べたように，解析処理後の中間語プログラムに対して行うため，アセンブリ言語や PL/I のテキストマクロの展開のように簡単ではない．すなわち，通常の実引数と仮

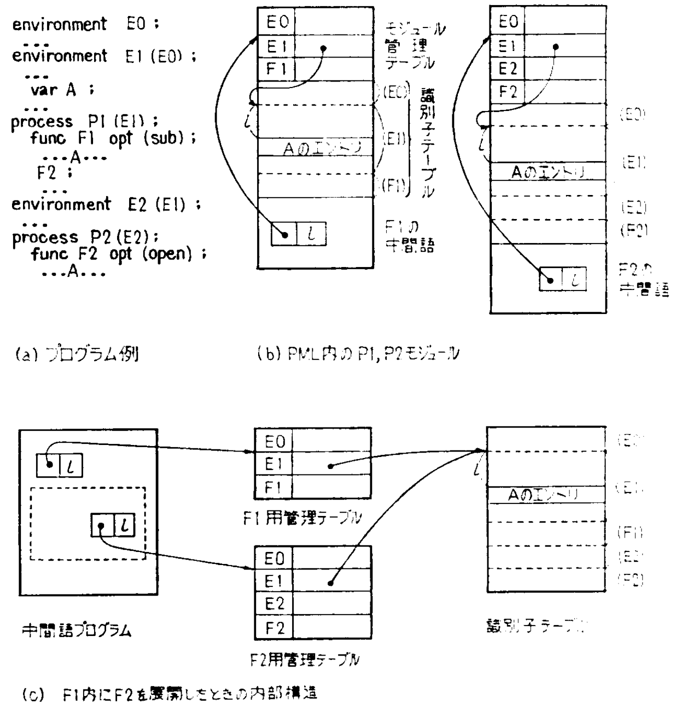


図 5 手続きのインライン展開のメカニズム
Fig. 5 A mechanism of inline substitution.

引数の置き換えのほかに，引用される手続きの環境情報と引用する側の手続きの環境情報を統合し，中間語プログラムをこの統合された環境情報に関連づける必要がある．

そこで，SPL コンパイラでは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

- 各環境モジュー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，
- 各環境モジュールの先頭からの相対アドレス，

の２アドレス形式とし，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で行うことにした．

処理例を図5に示す．これは手続きＦ２を手続きＦ１にインライン展開した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を指すようにするメカニズムを示している．

このような環境情報の統合方法により，中間語をスキャンしてポインタを書き換える処理が不要になり，インライン展開処理が効率良く行える．

なお，環境情報の統合時には，共通部分のバージョンの一致をチェックする．すなわち，3.2 節で述べたようなライブラリ構造では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り，図5の場合，Ｐ１

とＰ２が取込んだ共通環境Ｅ０，Ｅ１が各々同じバージョンである保証はない．そこで，ライブラリ化されるモジュールは誕生日として通し番号を与え，これを用いてバージョンの管理を行う．

4. 性能解析

4.1 コンパイル効率

SPL は日立の制御用計算機 HIDIC 80 用に設計された構造化プログラミング用言語[18),19)]で，その処理系は，制御用 Fortran (PCL)[20)]のプリプロセッサとして開発された．

処理内容別の時間比では，**図 6** に示すように解析処理が全体の 3/4 を占めている．このデータは環境モジュール２個，**main** 型手続きと５個の **open** 型手続きを含む処理モジュール１個から成る比較的単純な階層構造をしたプログラムの実測値である．SPL コンパイラに固有の展開・解釈処理は 10% 前後のものが多いが，コンパイル時実行文や **open** 型手続き引用文を多用すれば増加する．

各処理時間に占める CPU 時間，各種入出力時間の割合は**図 7** のようになっており，全体では CPU : I/O はほぼ 1 : 1 である．解析処理に占める補助記憶装置（磁気ドラム，M/D）入出力時間比が他の２倍近くになっているのは，SPL の特徴である環境モジュールの独立化と階層化により，SPL ライブラリへの参照が頻繁に行われることを示している．そのため，本処理系ではライブラリとの入出力処理に用いるページングアルゴリズムの改良を行ったが，詳細は別に報告[21)]した．

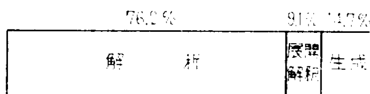


図 6 各処理フェイズ間のコンパイル時間比
Fig. 6 Compile time of each process.

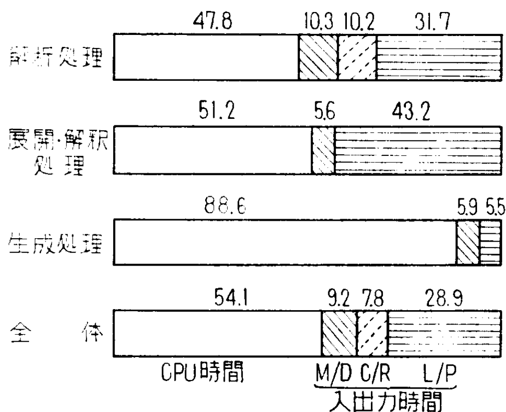


図 7 各処理フェイズごとの CPU および入出力時間比
Fig. 7 CPU time vs. I/O time of each process.

4.2 オブジェクト効率

モジュラプログラミングによって開発されたプログラムの場合，その構成要素である各々のモジュールがそれ自身で閉じたプログラムとして記述されるため，いくつかのオブジェクト効率低下要因がある．その主なものは，

（ⅰ） リンケージオーバヘッド，
（ⅱ） 処理の冗長性，
（ⅲ） 局所変数（作業エリア）の増加，

であるが，SPL ではこれらの対策がなされている．

（ⅰ）についてはすでに **3.4** で述べたように，すべての手続きを外部手続きにするとリンケージオーバヘッドが大きくなるので，手続き本体をその引用部分にインライン展開する機能がある．本機能の効果は CLU でも確められている[22)]．

（ⅱ）の処理の冗長性とは，トップダウン開発において，上位プログラム作成時には下位プログラムの機能とインタフェース仕様を知る必要はあるが，その処理手順まで知る必要はないため，下位プログラムを上位プログラム内にインライン展開した時に不要な処理部分が生じることをいう．これを避けるためには，下位プログラム作成時に，SPL のコンパイル時実行機能を用いて冗長な処理を除くような記述をするのが有効である．

（ⅲ）は，プログラムのモジュール化によって処理の記述が局所化されることにより，作業エリア的に用いられる変数が増加する問題である．これは変数宣言時にメモリ属性の **equivalent** 指定を用いてユーザ側で解決することもできるが，信頼性低下の要因となるので好ましくない．この問題の多くはコンパイラの最適化処理で解決される．

SPL を用いてプログラムのモジュール化を行った場合のオブジェクト効率の低下度合を定量的に把握するために，PCL（日立の制御用 Fortran）で約 200 ステップの既存のプログラムを SPL で書き直したところ，機械語オブジェクトは 8.7% 大きくなった．その他の類似実験，分析などからも，大半のプログラムのオブジェクト効率の低下は PCL 比で 10% 以下と推定される．

5. むすび

以上，プログラムの構造化，抽象化，カプセル化などの新しいモジュール化技法を支援する言語処理系に必要とされるコンパイル技法とその性能解析について述べた．特に考慮すべきことは，厳しいデータ型チェックとコンパイル効率を確保しながら分割コンパイルを実現する方式とオブジェクト効率低下の防止策である．

ここで述べた構造化プログラミング用言語 SPL は，日立の制御用計算機 HIDIC 80 用に開発されたもので，すでに多くのアプリケーションプログラムに適用された実績[23]を有する．

今後は，SPL が支援するモジュール化技法の適用をより容易にするために，プログラム構造の再構成，最適化，ライブラリ保守などに関する会話型プログラミングツールの開発が重要であり，すでにその検討[24]を始めている．

最後に，本研究の機会を与えていただいた日立製作所大みか工場高井兵庫副技師長ならびに，日頃ご指導頂く，同社システム開発研究所中田育男博士に深謝します．

参考文献

1) 中所武司：プログラムのモジュール化技法，信学会誌，Vol. 62, No. 1, pp. 91〜93 (1979).
2) Dijkstra, E. W.: The Structure of the "THE"—Multiprogramming System, Comm. ACM, Vol. 11, No. 5, pp. 341〜346 (1968).
3) Wirth, N.: Program Development by Stepwise Refinement, Comm. ACM, Vol. 14, No. 4, pp. 221〜227 (1971).
4) Dijkstra, E. W.: Notes on Structured Programming, Structured Programming, Academic Press, pp. 1〜82 (1972).
5) Parnas, D. L.: On the Criteria to be Used in Decomposing Systems into Modules, Comm. ACM, Vol. 15, No. 12, pp. 1053〜1058 (1972).
6) Myers, G. J.: Reliable Software through Composite Design, p. 159, Petrocelli/Charter, New York (1975).
7) Liskov, B. and Snyder, A.: Abstracton Mechanisms in CLU, Comm. ACM, Vol. 20, No. 8, pp. 564〜576 (1977).
8) Wulf, W. A. et al.: An Introduction to the Construction and Verification of Alphard Programs, IEEE Trans. Software Eng., Vol. SE-2, No. 4, pp. 253〜265 (1976).
9) Geshke, C. M. et al.: Early Experience with Mesa, Comm. ACM, Vol. 20, No. 8, pp. 540〜553 (1977).
10) Wirth, N.: Modula: a Language for Modular Multiprogramming, Software-Practice and Experience, Vol. 7, No. 1, pp. 3〜35 (1977).
11) Lampson, B. W. et al.: Report on the Programming Language Euclid, SIGPLAL Notices, Vol. 12, No. 2, pp. 1〜79 (1977).
12) Requirements for High Order Computer Programming Language "STEELMAN", p. 22, Department of Defence, Arlington (1978).
13) 林，野木，中所，浜田ほか：制御用ストラクチャドプログラミング言語 SPL，情報処理学会第17回全国大会，pp. 1〜8 (1976).
14) 野木，中所，中田，浜田，林：トップダウン・プログラミング言語 SPL におけるモジュール概念について，情報処理学会ソフトウェア工学研究会資料 3-1, pp. 1〜10 (1977).
15) Kieburtz, R. B. et al.: A Type Checking Program Linkage System for Pascal, Proc. the 3rd Int.. Conf. on Software Eng., pp. 23〜28 (1978).
16) LeBlanc, R. J.: Extentions to Pascal for Separate Compilation. SIGPLAN Notices, Vol. 13, No. 9, pp. 30〜33 (1978).
17) Welsh, J. et al.: Ambiguities and Insecurities in Pascal, Software-Practice and Experience, Vol. 7, No. 6, pp. 685〜696 (1977).
18) 林，野木，中田，浜田：制御用トップダウン・ストラクチャードプログラミング言語-SPL-，日立評論，Vol. 60, No. 3, pp. 59〜64 (1978).
19) HIDIC 80 SPL 言語仕様編，ソフトウェアマニュアル，p. 282，日立 (1977).
20) HIDIC 80 PCL 言語（文法編）ソフトウェアマニュアル，p. 176，日立 (1975).
21) 中所，林：ページングアルゴリズムの性能に関する実験的および理論的解析，情報処理学会論文誌，Vol. 20, No. 6, pp. 460〜467 (1979).
22) Scheifler, R. W.: An Analysis of Inline Substitution for a Structured Programming Language, Comm. ACM, Vol. 20, No. 9, pp. 647〜654 (1977).
23) 野木，中所，林，森：階層化プログラミング言語 SPL の評価，情報処理学会ソフトウェア工学研究会資料 9-1 (1979).
24) Chusho, T. and Hayashi, T.: Two-stage Programming: Interactive Optimization after Structured Programming, Proc. the 3rd UJCC, pp. 171〜175 (1978).

(昭和54年1月22日受付)
(昭和55年3月21日採録)