第２章　従来の諸研究との関係

第２章　従来の諸研究との関係

2.1　概　　要

１９５０年代後半から６０年代前半にかけて，Fortran，Cobol，Algol，ＰＬ／Ｉなどの手続き型言語が次々と開発され，記述量の削減によるプログラムの生産性向上をもたらした。しかしながら，７０年代にソフトウェアの大量化と大規模化が進み，その保守，拡張の費用が増大するにつれて，プログラムの書き易さよりも読み易さの方が重要になってきたが，この点では従来言語は不十分なものであった。

そこで，７０年代には，本研究との関連の深い段階的詳細化技法やデータ抽象化技法をはじめとして，種々の新しいプログラミング方法論[C12]が提案された。そして，言語設計においても，これまでのような機械語への展開率を向上させる「量的高級化」よりも，プログラミング方法論を反映させた「質的高級化」を実現している。これらの研究経過は大まかには次のように考えられる。

(1)　７０年代初期　：　方法論の提案

(2)　　同上中期　：　方法論の具体化と言語の提案

(3)　　同上後期　：　言語処理系と関連ツールの開発

本章では，この分野における従来の諸研究と本論文との関係について述べる。

2.2　プログラミング方法論

以下に述べる幾つかのプログラミング方法論はいずれもプログラムのモジュール化を促進し，理解容易なプログラム構造を導くという共通点がある。そして，その目的は，プログラムの作成，検証，保守，拡張の容易化による生産性および信頼性の向上である。

(1)　段階的詳細化

これは，７０年代初めにE.W. Dijkstra[D10]とN. Wirth[W7,W9]によって提案されたプログラムのトップダウン設計技法であり，まずはじめに大まかなことを決め，次第に詳細な設計に進む過程をそのままプログラム化するものである。即ち，各段階の抽象レベルで閉じたプログラムを作成し，次の段階ではその詳細化を行うという過程を繰返し，最後に実際の計算機で処理できるレベルに達すれば終了する。その結果，プログラムはモジュールの階層構造になる。なお，このようなプログラムの階層構造化は，以前にE.W. Dijkstra[D9]が多重プログラミングシステムＴＨＥの開発に適用している。これは，オペレータ操作からハードウェア制御までの間を５段階の機能レベルに階層化している。

ＳＰＬでは，本技法が信頼性の高いプログラムを導くこと，およびプログラムのドキュメント性を高めることから，その支援機能を具体化して実用化した。

(2)　データ抽象化

これは，プログラムを各抽象レベルで表現されたモジュールの階層構造とする点で前項と似て

いるが，手続きよりもデータの抽象化を重視する。即ち，データの利用者はそのデータの詳細構造や操作アルゴリズムを知る必要はないので，それらをまとめて定義し，カプセル化する。そして，データの利用者にはデータ抽象型名とその操作手続き名の集合だけが与えられるので，その型のデータへのアクセスはこれらの操作手続きを通じてのみ行われる。このようなデータ抽象化技法の言語化の研究は，B. Liskov[S6,L6]をはじめとして７０年代中期以降に盛んに行われた。[B3] なお，この方法の基本的概念は，それ以前にD.L. Parnas[P1,P2]によって提案されたInformation hidingの概念と同じである。この概念は，モジュール分割の評価基準として導入されたもので，各モジュールの設計時の詳細情報を他のモジュールから隠すと共に，そのインターフェイスをインプリメンテーションとは無関係に設定する方法である。

ＳＰＬでは，本技法は，共通データへのアクセス誤りが多いという従来方式の問題解決に有効であるという考えに基づき，その支援機能を具体化して実用化した。

(3) 複合設計

これは，モジュール分割において，個々のモジュールの強度は強くなるように，そしてモジュール相互の結合度は弱くなるようにする方法であり，L.L. Constantine, W.P.Stevens,[S13] G.J. Myers[M14,M15]によって提案されている。即ち，モジュール強度は，強い順に機能的，情報的，連絡的，手順的，時間的，論理的，暗合的レベルが設定され，モジュール結合度は，弱い順にデータ結合，スタンプ結合，制御結合，外部結合，共通結合，内容結合に分類される。従って，個々のモジュールは機能単位に構成し，情報の受け渡しは引数で行うのが良いとされる。[C3,S5]

ＳＰＬでは，モジュール化を促進する機能を導入することにより機能的モジュールの作成を容易にしている。

(4) 構造化コーディング

これは，プログラムの制御構造を逐次処理，選択処理，繰返し処理の３形式の組合せで表現し，goto文のような無条件分岐を排する方法であり，最初に１９６８年にE.W. Dijkstra[D8]によって提案された。以来，その是非をめぐって"goto論争"が行なわれたが，基本的考え方としては定着した。[L12] 但し，D. Knuth[K4]によってgoto文を用いた構造化プログラミングが提示され，例外処理などの限定された使用の必要性が指摘されている。なお，すべてのプログラムはgoto文無しで記述できるという理論的根拠は，１９６６年にC.Böhm と G. Jacopini[B9]によって与えられている。

ＳＰＬでも，プログラムの理解容易性向上のために本機能を導入し，goto文を排すと共に，例外処理のためのブロック抜け出し文を設けて，記述性の低下を防いだ。

(5) 複雑度

これは，個々のモジュールの処理の複雑度を測る定量的尺度を導入するもので，その度合が低いほど良いと考える。その代表的なものは，T.J. MaCabe[M4]が提案したcyclomatic numberである。この尺度は，プログラムを制御フローグラフＧで表現した時，節点数ｎ，辺数ｅ，連結

要素数pとして，V(G)＝e－n＋pで定義される。これは，構造化コーディングの概念と合うが，データに関する要因が無視されている。そこで，分岐に用いられる制御変数やデータ使用形態に注目した尺度[M6]など，種々の尺度[B1]が提案されている[C4]。

ＳＰＬでは，段階的詳細化やデータ抽象化によるモジュール化の促進，および個々のモジュールの処理記述への構造化コーディングの適用により，プログラムの複雑度を低くするようにしている。

(6) その他

上記以外にも種々の方法論に関する提案や議論がなされている。特にトップダウン方式[M7,W2]による設計やプログラミングについては，その利点[M9]が強調される一方で，その困難性[B5,P3]も指摘され，ボトムアップ方式との併用[M5]が提案されるなど，議論の多い所である。ＳＰＬでは，段階的詳細化支援機能の他にデータ抽象化機能も備え，いずれの方式も可能としている。

2.3 言語と処理系

2.3.1 設計言語

モジュール設計時に用いる言語としては，モジュールインターフェイスを明記できるものと各モジュールの処理概要を手続き的に記述できるものがある。前者の例としては，F. DeRemer[D7]が，変数や手続きをリソースとみなし，その共有関係を明記させる言語ＭＩＬ(Module Interconnection Language)を提案，また，R. Bridge[T3]らは，グローバル変数の有効範囲の限定やそのアクセス形式を明記するモジュールインターフェイス記述言語を提案している。しかしながら，このような方法はプログラムの静的構造と動的構造のずれが大きくなるため，プログラムの理解容易性に欠ける。そこで，ＳＰＬでは，グローバル変数の有効範囲は環境モジュールの階層構造で表現し，プログラムの静的構造から容易に理解できるようにしている。

一方，処理概要の記述言語の例としては，S.H. Caine[C1]らの開発したＰＤＬがある。これは，制御構造は従来のプログラミング言語の制御文と同様に表現するが，個々の処理内容は自然語風に記述できる。しかしながら，この方法では最終的には人手によってプログラミング言語に変換するため，設計書とプログラムの対応が不明確になる。そこで，ＳＰＬでは，手続き名を自然語風の構文とすると共に，そのインライン展開機能を設けることにより，処理概要の記述からプログラム記述までを１つの言語で済むようにした。

2.3.2 プログラミング言語

１９７０年代には新しいプログラミング方法論を反映した言語が数多く設計，開発された[C2,T4]。まず，データ抽象化機構を備えた言語としては，シミュレーション言語Simula 67のclassの概念[D1,I2]を基礎にしたＣＬＵ[L7,L9]，Alphard[W12,W13,W14]，Mesa[G2,M12]などがある。B. LiskovのＣＬＵは，clusterと呼ばれる抽象化機構を導入し，データ型とその型のデータへの操作手続きをまとめて定義する。この時，データ

型を引数にできるため，例えば，整数型スタックと実数型スタックを統一的に定義できるなど，抽象化機能は強力である。W.A. Wulfらの Alphard は，form と呼ばれる抽象化機構を導入している。その中は3部分に分かれ，specification 部では外部インターフェイス，representation 部ではデータ構造など，オブジェクト生成に必要な情報，implementation 部では操作手続きの本体が記述される。この言語は検証機能を重視しているため，操作手続きの前提条件や終了条件を宣言する機能がある。Xerox 社の Mesa は，外部インターフェイスは Abstraction 部で，また操作手続きの本体は implementer 部で行う。

一方，当初教育用として N. Wirth の設計した Pascal[W8,J1] は，構造化コーディング用の制御文と豊富なデータ型を備えている他，データ抽象化の基本となるデータ型定義機能に特徴があり，その後，Euclid[L1] や Ada[D4,D5] に影響を与えた。Euclid はプログラムの検証を容易にするために B.W. Lampson らによって設計された言語で，Pascal を基本にしている。この言語では，名前のアクセス方法を明示的に記述するため，定義側と参照側で各々 exports, imports の宣言を行う。Ada は，米国々防総省が内部で使用する統一的高級言語として開発したもので，データ抽象化，例外処理，非同期処理などの機能が充実している。言語仕様は上記のデータ抽象化支援言語や Pascal の影響を受け，package と呼ばれる抽象化機構を持つ。

この他，goto 文を排して構造化コーディングを支援する言語も多く，その1つとして，Pascal と同じ時期に W.A.Wulf によって設計された Bliss[W11] がある。この言語では goto 文を排除した代りに例外処理用にループやブロックから抜け出す exitloop 文や leave 文がある。その後，従来言語に対しても「構造化 Fortran」，「構造化 Cobol」などの改良版が開発された。また，通常の Fortran プログラムを自動的に構造化するツール[B2] なども開発されている。一方，これらの構造化コーディング用の制御文の構文に関する議論も多く，繰返し処理[F1] や多方向の選択文[S2,D12] などに関する種々の方法が発表されている。

ＳＰＬでは，制御文については Bliss などと同様に，goto 文を排し，ブロック抜出し文を導入したが，データ抽象化と段階的詳細化の支援機能は，特別な抽象化機構を導入することは避け，従来のブロック構造に近い方式を採った。

2.3.3 コンパイル技法

従来の手続き型言語に対するコンパイル技法はほぼ確立しているが，プログラミング方法論を反映した言語には新たな技法が要求される。

その代表的なものとして分割コンパイル方式がある。即ち，従来の言語では，外部手続きなどのコンパイル単位に独立に処理され，前のコンパイル結果を後で使用することはしなかった。しかし，型チェックを徹底して行うためには，1モジュールのコンパイル時にもシステム全体の情報が必要になる。そこで，Pascal のように常に全体をまとめて処理する一括コンパイル方式が良いが，この方式はトップダウン開発や大規模ソフトの開発には適さない。そのため，何らかの方法でコンパイル情報

を受渡す機構が必要となる。

そこで，本研究では，モジュール単位のコンパイル情報を保存する共通ライブラリを導入し，関連モジュールのコンパイル時にそれを参照するような分割コンパイル方式を考案した。

本方式は，その後にＣＬＵの処理系[A3]でも採用されており，Adaでもその必要性[D4]が認められている。Mesaの場合[L2]もコンパイル対象モジュールが参照（ imports ）する外部モジュールのコンパイル情報を最初に読み込む方式をとっているほか，これらのモジュールの結合は専用のバインダーで行う。また，一括コンパイル方式を前提にして設計されたPascalに対しても，実用的見地から分割コンパイル方式への拡張[L3,L4]が試みられている。

次に，データ抽象化などのために導入されるデータ型定義機能に関連して，データ型の一致のチェック方法の問題[B7,D2,H2]がある。これは，基本的には，データ型名の一致をチェックする厳しい方法とデータ構造の一致で良いとする緩い方法がある。Pascalの場合は当初，その仕様があいまいであったため，処理系毎にまちまちであるが，その系統のEuclidやStony Brook Pascal／３６０[K2]などは構造によるチェック方式である。特別の抽象化機構を有するＣＬＵでは，基本的には名前による方式をとる代りとして，抽象データ型定義本体で明示的にcvt宣言をした型に対しては構造による方法を採っている。本研究では，いずれか一方に決めると，いずれの場合も不都合が生じるという考えから，使用目的に合せて使い分ける方式を採った。

また，構造化，モジュール化を徹底したプログラムのオブジェクト効率低下の問題については，Mesaが，本研究と同様の手続きのインライン展開指示機能を１９７９年の改訂版で追加している他，ＣＬＵでは，コンパイラの最適化処理として手続きのインライン展開処理を行っており，その効果を実験的に確めている[S3]。その他，goto文を排した言語ＳＩＭＰＬを対象にコンパイラによる最適化の試みがM.V. Zelkowitz[Z1]らによってなされている。

2.3.4　ページングアルゴリズム

分割コンパイル方式を実現するためには，各モジュールのコンパイル結果をライブラリに保存し，他の関連モジュールのコンパイル時に参照するような方法が有効である。大規模ソフトウェアの開発時には，このライブラリは大容量になるため，補助記憶装置に置き，必要に応じて主記憶装置にロードするようなページング処理が必要である。

このページング処理に関する研究は，主にプログラムの手続き部を対象に行われてきており，実験的にも理論的にも多くの研究成果[C18,C19]がある。例えば，L.A. Belady[B6]は，主記憶バッファの小さい所では，ページフォールトから次のページフォールトまでの平均実行時間は，バッファサイズをｓ，パラメータをａ，ｋとして，as^kで近似でき，ｋ≒２であることを示した。また，R.L. Mattson[M3]はＬＲＵのようなスタックを用いたアルゴリズムを数学的に定式化し，ページングアルゴリズムの特性の理論的解析を行った。この他にも簡易ＬＲＵ方式の研究や代表的なページングアルゴリズムであるＬＲＵとＦＩＦＯの性能差の実験分析[M1,O1]，あるいは，一部分のページをバッファに常駐する方式の提案[G1,Y1]

など，多岐にわたって研究が行われてきた。

この分野の研究が充実している一因として，汎用オペレーティングシステムにおける仮想記憶方式[D3]の実現があると思われるが，その反面，データを対象にしたページング特性の研究は少ない。その中で，最近，研究が開始されたのはデータベースの分野においてである。しかしながら，例えば，関係データベースマシンＲＡＰでは，ページの局所参照性が高いというS.A. Schuster[S4]らの主張がある一方で，階層型データベースでは局所参照性は無いという実験報告がJ. Rodrigues-Rosell[R3]やS.W. Shermann[S7]らによってなされるなど，未だページング特性は十分には解明されていない。

本研究で取り上げた，分割コンパイル時のライブラリ参照特性やＬＲＵ方式と簡易ＬＲＵ方式あるいはＬＲＵ方式とＦＩＦＯ方式の性能差に対する理論的分析についての研究は，これまで皆無であると思われる。

2.4　言語支援系

2.4.1　ソース変換システム

本研究で提案する構造化と最適化の２段階プログラミング法に関連する研究として，ソースプログラムの変換システムに関するものが幾つかある[D11,S8]。

まず，１９７４年にD. Knuth[K4]が会話型プログラム操作システムの概念を提案した後，T.A. Standish[S10]らがこの概念を具体化したシステムを開発した。このシステムでは，可能なプログラム変換操作の一覧表をカタログとしてまとめている[S11]。D.B. Loveman[L11]も同様の技法を提案しているが，これらはいずれもカタログが大きくなるため，実用上は必要な操作コマンドの検索に手間取るという問題がある。そこで，J.J. Arsac[A2]は最も単純なプログラム変換だけを扱うカタログ作成を試みている。

本研究では，すべての変換コマンドについて，その適用前後のプログラムの機能的等価性を自動証明すること，およびカタログの肥大化を防ぐためにプログラムの最適化は単純な変換コマンドの組合せで行うことを基本方針とした。

2.4.2　特定言語向きプログラミング環境と構造エディタ

既存のプログラミング環境[H7]の代表的システムとしてベル研のUnix[K1,R2]がある。これは，Shellと呼ばれるコマンド解析系に基づく統一的ユーザインターフェイスと単純明快な木構造形式のファイルに特徴があり，７０年代の優れたシステムと言える。しかしながら，プログラム開発支援機能は個別ツールの寄せ集めで，一貫性はない。

一方，特定言語向きの統合プログラミング環境[C16]としては，Xerox社のLisp言語用Interlisp[S1,T2]，ＩＮＲＩＡのPascal用Mentor[D13]，コーネル大学のＰＬ／ＣＳ言語用Cornell Program Synthesizer[T1]，カーネギーメロン大学のIncremental Programming Environment（ＩＰＥ）[M8]，京大のIota言語用プログラミングシステム[N1]などがある。これらはいずれも構造エディタやデバッガを備え，プロ

グラムの開発はすべてこのシステムだけで行える。しかしながら，対象言語の制約やテスト機能の不備などから，大規模ソフトウェアの開発には適していない。また，現在，米国々防総省が推進している言語Ada用のプログラミング支援環境ＡＰＳＥ(Ada Programming Support Environment)でもツールの統合化を試みている。

プログラミング環境の代表的ツールであるプログラム編集用エディタとしては，プログラムを文字列として扱うテキストエディタが最も一般的であるが，最近は画面エディタが普及しはじめている。その一例として，熟練者向きのＭＩＴのＥＭＡＣＳ[S9]やＵＣＢのvi,初心者向きのＩＢＭのＳＰＦ[I1]や日立のＤＥＳＰなどがある。

先に述べた特定言語向きプログラミング環境では，いずれもテキストエディタの代りに，対象言語に依存した編集機能を有する構造エディタを含んでいる[L5,M13]。この構造エディタは，３種類に分類できる。即ち，第１は，Mentor や Interlisp が有する初期の構造エディタであり，これらは，プログラムの編集中に常にその構文の正しさを保証する構文チェック型である。第２は，構文テンプレートを自動生成する文法誘導型であり，ＩＰＥやＣＰＳがこれに属する。本研究の構造エディタはこれらの機能に加えて，段階的詳細化技法によるプログラム開発を促進する構造化技法支援型である。なお，構造エディタにも従来のテキスト編集機能を残すべきか否かについては賛否両論[C20,W1,W3]があるが，本研究では，文単位以下の構文要素についてはテキスト編集を許して操作量を減らす一方，その構文チェックは即座に行い，常に文法的に正しいプログラムを保証するようにした。

2.4.3　構造テスト

ソフトウェアのテストの方法として，種々の技法[M11]が考えられている。まず，静的テスト法として，プログラムの制御フローを解析して実行されない命令を検出したり，変数の値の定義と参照に関するデータフロー解析[B4,H1,L10]によりその矛盾を検出するツールがあるが，この方法で検出できる誤りは限られている。プログラム内の変数に数値などの定数の代りに記号値を与えてプログラム動作解析を行う記号実行[B10,H5]は，実行経路分析やテストデータ生成[V1]に応用される他，プログラムの正当性証明[K3]にも用いられる。また，プログラムの入出力[L8]やループ不変式の表明に基づいてプログラムの正当性証明[G5]を行うものもある。しかしながら，いずれも大規模ソフトウェアのための一般的な検証技術としては実用的でない。

そこで，実際にはテストデータを用いてプログラムを実行し，その結果を調べるという動的テスト法[C17]が行なわれているが，この方法においては，効果的なテストデータセットを選択することが重要である。J.B. Goodenough[G3]らは reliable と valid という概念を用いて理想的なテストデータセットを定義したが，その具体的手法は見つかっていない。実用的手法[H6]としては，プログラムの機能仕様に基づいて選択する機能テスト法[W5]とプログラム構造に基づく構造テスト法がある。前者については，原因結果グラフ法[M16]，領域法[W6]，状態遷移図法[C5]などがあるが，自動化ツールは少ない。

一方，構造テスト法としては，プログラムの入口から出口に至るすべての実行可能なパスのうちのできるだけ多くをテストすることを目標とするパステスト法がある。この場合，実用的なプログラム

の多くは繰返し処理を含み，パスの数が膨大になることから，テスト網ら率の尺度としては，パスよりも簡単なもの[R1)]が用いられる。その代表的なものは，プログラム内の分岐に着目した網ら率尺度を用いるもので，E.F. Miller[H8,M10)]によって提案され，多くのテスト支援ツールに採用されている。しかしながら，このように全分岐を対等に扱う尺度は，品質の過大評価や冗長なテストデータ選択などの欠点があるため，本研究では，パステストに本質的な分岐にのみ着目する新しい網ら率尺度を提案した。