第８章　言語支援系の考察

第８章　言語支援系の考察

8.1　概　　要

ソフトウェアの開発は，プログラミング言語によるプログラムの記述を以って終了するものではなく，その後に全開発費用の約半分を占めるプログラムの検証作業が必要である。そのため，大規模ソフトウェアの信頼性と生産性の向上のためには，単に言語機能の高級化とその処理系の開発にとどまらず，プログラム開発支援環境の充実が重要である。

ところが，従来のプログラム開発支援ツールは，テキストエディタやデバッガに代表されるように，プログラミング言語とは独立した機能を持ち，汎用性が高い反面，個々の機能は比較的簡単なものに限られている。そこで，新しい言語の適用をより効果的なものにするためには，その言語に反映されたプログラミング方法論を促進するような支援ツールや言語機能に適応したプログラム生成，検証，修正機能を有するツールが必要であると思われるので，本章では，これらの言語支援系について検討する。

まず第１には，構造化プログラミングを促進するために，最初の段階ではプログラムの構造化に徹し，その後に性能要求に応じて最適化を行う２段階プログラミング法を検討する。そして，この方法論を支援する会話型システムに備えるべき構造化コマンドと最適化コマンドの機能抽出を行うために，その適用実験を行い，効果を確める。

次に，プログラムの生成と修正時にＳＰＬの文法を誘導する構造エディタ機能を検討する。従来はテキストエディタが用いられてきたが，これは，プログラムを意味のない文字列とみなし，文字単位あるいは行単位のテキスト操作を行うものである。そのため，文法的な誤りを防げないことや操作効率が悪い等の欠点があった。そこで，この問題の解決のためには，構文テンプレートの自動表示や構文要素単位の編集機能が有効である。また，プログラムの生成時に，ＳＰＬが支援するプログラミング技法である段階的詳細化技法や構造化コーディング技法の適用を誘導することにより，新言語をより効果的に使用できる。本章では，これらの機能の具体化を行う。

次に，プログラムの検証に効果的なテストデータを効率的に作成するための構造テスト法について検討する。従来の代表的技法として，プログラム内の分岐がどの程度テスト実行されたかを調べる分岐テスト法があるが，これは，冗長なテストデータ作成や品質の過大評価を招き易いため，この改良方式を考える。

最後に，従来のようにプログラミングツールを個別に開発する方式はユーザの使い勝手を損うため，本章で述べる言語支援系を１システムとする統合プログラミング環境の構築技法について検討する。

8.2　２段階プログラミング法[C6,C7)]

8.2.1　プログラミングの構造化と最適化

構造化プログラミングは，前章で述べたように，オブジェクト効率の低下要因を伴っている。その

ため，実時間処理プログラムや主記憶容量の小さい小型計算機システムなどのように厳しいオブジェクト効率を要求する分野では，次のような２つの相反する評価基準に従ってプログラムを作成しなければならない。

(1) きれいな構造のプログラムを作ること。

(2) 効率の良いプログラムを作ること。

この問題に対する最も簡単な解決法は，構造化と最適化を分離し，第１段階では構造化に専念した後，第２段階で必要に応じて最適化を行うことである。本論文では，このようなプログラミング方法論を２段階プログラミング法と名付ける。この方法は，構造化プログラミングの提唱者であるE.W. Dijkstra の「１度に１つの決定をする」という段階的詳細化技法の原理を「１度に１つの評価基準を用いて１つの決定をする」ように拡張したものである。

このような方法論に基づく具体的なプログラミングの手順は，次のようになる。

(1) 初期プログラム作成

(2) 構造化のための再構成

(3) 検証と修正

(4) 性能評価

(5) 実行頻度分布測定（プロフィール解析）

(6) 最適化

この手順において，(1)～(3)は構造化，(4)～(6)は最適化処理段階である。即ち，まず，第１ステップで構造化プログラミングを心掛けて初期プログラムを作成後，第２ステップではその見直しを行う。そして，第３ステップにおいて，プログラムの機能的正しさを検証することによって，構造化プログラミングの段階を終了する。次に，第４ステップで性能評価を行い，所定の性能が得られない場合は，第５ステップにおいて，各文単位の実行頻度を測定する。そして，その結果から改良効果の大きい所を調べ，第６ステップで最適化処理を行う。この(4)～(6)を繰返し，所定の性能に達すれば，最適化処理段階を終了する。

ここで，最適化処理の方法としては，人手作業，会話型処理，自動化の３種類が考えられる。このうち，人手作業による方法には，次のような致命的な欠点がある。

(1) 最適化処理後のプログラムに対して，再び検証が必要である。

(2) 構造化プログラムにおける理解容易性の利点が失われ，プログラムの保守が難しくなる。

一方，コンパイラ等による自動化の方法では，最適化の内容が複雑で高度なものになるほど，その処理コストに対する効果が小さくなることから，最適化処理には限界がある。そこで，この問題を解決するためには，会話型処理の方法を採用し，各最適化コマンドの実行によってプログラムの機能的な等価性が保たれることを自動証明をするのが良いと考えられる。

8.2.2　適用実験

2段階プログラミング法の効果の確認と最適化コマンド機能の抽出のために，2段階プログラミング法の適用実験を行った。

(1)　例題プログラム

この適用実験には，6.3節のページングアルゴリズムの分析に必要とされたプログラムを例題として用いた。従って，本プログラムの機能は6.3.1で述べている。

(2)　初期プログラム作成

最初に段階的詳細化法を用いてプログラムを作成したが，その様子は既に4.3節および図4.2で述べた通りである。プログラムのモジュール構造の全体は図6.2に示してある。

(3)　プログラムの再構成

初期プログラムに対して，構造化プログラミングの観点から見直しを行い，次のような項目についてプログラム構造の変更を行った。

(a)　名前の有効範囲を正確に表現するため，あるレベルの環境モジュールにある変数，定数名などの宣言を他の環境モジュールに移動する。

(b)　データ抽象化を徹底するため，そのデータアクセス用手続きをしかるべき処理モジュールに追加する。

(c)　各処理モジュールや手続き自身の機能強度を高め，お互いの結合度は弱めるというG.J. Myersの尺度に従って，モジュールや手続きの統合と分割を行う。

その結果，この構造化プログラムは，検証終了時には，１２個の環境モジュールと１３個の処理モジュールで構成され，３３個の手続きを含む。そして，そのオブジェクト効率は，表8.1の(a)に示すように，メモリ容量が５９１０Ｗ，実行時間は３７７秒であった。

(4)　最適化処理

ここでは，メモリを節約する場合と時間を節約する場合に分けて，各々の最適化処理を徹底して行った。その結果，表8.1の(b)と(c)に示すように，メモリに関しては３４％の削減，実行時間に関しては４７％の短縮を達成した。ここでは，以下のような最適化項目を適用した。

(a)　引用順序に依存して冗長となる手続き内処理を削除する。

(b)　繰返し処理部分の順序変更により，初回判定処理を削除する。

(c)　類似処理をサブルーチン型の手続きとして統一する。

(d)　類似の手続きをコンパイル時機能を用いて統一する。

(e)　ｉｆ文のthen節とelse節に共通に含まれる処理をｉｆ文の外へ出す。

(f)　メモリ属性equivalentを指定して，変数のストレージを共有する。

(g)　複数箇所で引用されているopen型手続きをサブルーチン型に変更する。

(h)　サブルーチン型手続きをopen型に変更する。

これらの項目のうち，(c)～(g)はメモリ節約，(h)は時間節約，(a)～(b)はその両方の節約に効果があ

る。

(5) 結果の考察

第1段階では，プログラムの構造化にのみ着目したため，その作成は容易であった。また，論理的誤りが2件あったが，いずれもデバッグは容易であった。一方，最適化処理による効果の度合は，そのプログラムの性質により異なるが，本実験の結果は，2段階プログラミング法の実用性を示していると思われる。

8.3 構造エディタ[C10)]

8.3.1 概　　要

プログラムの作成と修正に用いるツールとして，従来の行エディタに代り，最近では画面エディタが普及しはじめて操作性が向上した。しかしながら，これらはいずれもプログラムを単なる文字列とみなし，文字単位あるいはあるまとまった文字列からなる行単位のテキスト操作を行うテキストエディタである。そのため，それを使用するプログラマは，プログラムを操作対象とする時はプログラムを意味の無い文字列とみるが，思考対象とする時は意味のある文章としてみる。即ち，書くレベルと読むレベルの間に大きなギャップがあり，これが，単にプログラミング効率の低下ばかりでなく，誤り発生の原因になっている。

そこで，このような意味的ギャップを除き，操作対象のレベルを思考対象のレベルにまで引き上げるためには，プログラミング言語に関する知識を有し，プログラム構造を常に把握する構造エディタが有効と思われる。特に，プログラム作成時には，対象言語が支援するプログラミング方法論に沿った誘導機能を設けることにより，言語とツールの相乗効果が期待できる。

このような目的から，ＳＰＬを対象とした構造エディタとして，次のような機能を有するものを開発した。

(1) 段階的詳細化機能

(2) 構造化コーディング機能

(3) 構文要素単位の編集機能

(4) 上記項目のための画面編集機能

これらの詳細については次項で述べる。

8.3.2 基本機能

(1) 段階的詳細化機能

ＳＰＬが支援する段階的詳細化技法を次の方法で誘導する。

(a) 構造エディタが扱うプログラムは次のような詳細未定義部分（Refinable と呼ぶ）を含んでいて良い。

(i) ＳＰＬ文法を記述したＢＮＦ規則に用いた非終端記号（例：＜statement＞）

表８．１　２段階プログラミング法の実験結果

		メモリ容量（語数）	同左比率	実行時間（秒）	同左比率
a	構造化プログラム	５９１０	１．００	３７７	１．００
b	メモリ容量に関する最適化プログラム	３９２６	０．６６	３７８	１．００３
c	実行時間に関する最適化プログラム	５９２７	１．００３	２００	０．５３

(ii) ＳＰＬの文の代りとなる擬似文（例："ＰＵＳＨ　ＶＡＬＵＥ　ＴＯ　ＳＴＡＣＫ"）

(iii) ＳＰＬの手続き引用文（例：ＰＵＳＨ（ＶＡＬＵＥ）ＴＯ（Ｓ１））

(iv) ＳＰＬのユーザ定義データ型参照（例：var Ｓ１：ＳＴＡＣＫ（１００））

これら４項目のうち，後の２項目はＳＰＬ自身の有する段階的詳細化機能であるが，第１項は文法規則に沿った詳細化を誘導するために導入した。第２項は第３項に似ているが，構文規則に制約されないでコメント風に記述できるようにしたものである。

(b) その詳細未定義部分を逐次的に画面に表示し，その詳細化を促す。表示形式は端末の機種に依存するが，例えば高輝度表示，特定色表示などとする。なお，その対象となっている詳細未定義部分をCurrent Refinable と呼び，ＣＲＥと略す。

(c) その際，可能な詳細化の形式をメニュー表示し，その１つを選択できるようにする。特にＣＲＥが非終端記号の時は対応するＢＮＦ規則の右辺の他に，直接的なテキスト入力の可否も示す。即ち，＜module＞や＜function unit＞のようなプログラムの基本構造に対応するものは必ずＢＮＦ規則の右辺で置換させるが，＜declaration＞や＜expression＞のような下位レベルのものは，ＢＮＦ規則右辺への置換の他に，熟練者用に直接テキスト入力も許した。

構造エディタを用いた段階的詳細化の例を図 8.1 に示す。まず，手続き引用文の「ニュウリョク　チエック」がＣＲＥの時に refine キーを押すと，手続き引用文はコメントに変わり，その下に文の非終端記号＜statement＞が表示される。そこで，ｉｆ文を選択するとその構文テンプレートが表示され，条件式の部分がＣＲＥになる。さらに，ここでテキストを入力すれば条件式の所へ置換され，ＣＲＥはその下の then 節の＜statement＞に移る。なお，テキスト入力時には即時に構文チエックを行う。一方，最初に手続き引用文がＣＲＥの時に define キーを押すと，その定義が誘導される。

(2) 構造化コーディング機能

ＳＰＬは，goto 文を排し，基本制御構造を選択文，反復文，複合文の３種類に限定している。そこで，構造エディタでもこのような構造化コーディングを支援するため，メニュー選択方式により，その構文テンプレートを自動表示するようにした。その１例が，図 8.1②のｉｆ文である。そして，この目的のために，次のような他と異なる特徴を持たせた。

(a) 一般の文は，＜statement＞がＣＲＥの時にはテキスト入力可能にしているが，選択文，反復文，複合文はテキスト入力を許さず，常に構文テンプレートを用いて詳細化する。

(b) その代り，メニュー選択の操作の手間を省くため，テンプレートコマンドを用意した。例えば，ｉｆ文の選択時にはそのメニュー選択の代りに「．ｉｆ」を入力しても良い。

(3) 構文要素単位の編集機能

プログラムの修正は，プログラムの生成の場合と同じく，構文要素単位に行えるようにして，修正時の誤りを防止する。その基本機能である挿入，削除，追加の操作は次のように行う。

(a) 挿入は，まず，該当部分にカーソルを移動後，insert キーにより，カーソル位置に挿入可能な非終端記号を挿入する。その後，プログラム生成時と同じ方法で追加したいテキストに置換する。

(b) 削除は，まず，該当部分にカーソルを移動後，reduce キーによって，カーソル位置のテキストに対応する非終端記号で置換する。その後，その非終端記号が文法的に省略可能な場合は空行入力によって削除する。

(c) 置換は，まず，削除の場合と同じ方法で該当部分を非終端記号に置換した後，プログラム生成時と同じ方法で新しいテキストに置換する。

このように，プログラムの修正はすべて非終端記号を経由して行い，一般のテキストエディタのような文字列操作を禁止した。ただし，これらの機能はあくまで基本となるものであり，実用上は操作性向上のための種々の拡張が考えられる。

(4) 画面編集機能

この構造エディタはディスプレイ端末を使用した画面編集を基本にしており，入力はできるだけファンクションキーやメニュー選択方式で行う。特に構造エディタに特徴的な画面操作機能として次のようなものがある。

(a) プログラムの中の詳細化したい部分を自由に選べるように，ＣＲＥの移動用のファンクションキーを設ける。

(b) モジュールや手続き単位の画面表示や画面スクロールを行うファンクションキーを設ける。

8.4 構造テストツール

8.4.1 動的テスト法の課題

ソフトウェアプログラムのテストとデバッグは，ソフトウェア開発費用の約半分を占め，生産性向上に重要であるばかりでなく，信頼性向上のための重要課題である。このプログラムテストは，「プログラムが仕様通りに作られている」ことを確めるのが目的なので，仕様書とプログラムが等価であることを自動検証するような方法が理想的であるが，まだ実用には程遠い。そのため，実際には，多くのテストデータを用いてプログラムを何度も実行し，各々の結果を調べるという動的テスト法が行われている。

このような手間のかかる動的テストを効率良く行うためには，システム全体のテストを行う前に，部品テストとも言うべきモジュールテストを徹底して行う必要があるが，この方法には実用面で次のような問題がある。

(1) 効果的なテストデータを効率的に作成する方法がない。

(2) テストの準備や結果の確認に手間どる。

このうち，第２項は，ＳＰＬのような構造化プログラミング支援言語で記述されたプログラムでは，あまり大きな問題ではない。即ち，段階的詳細化とデータ抽象化によってモジュール化が促進される

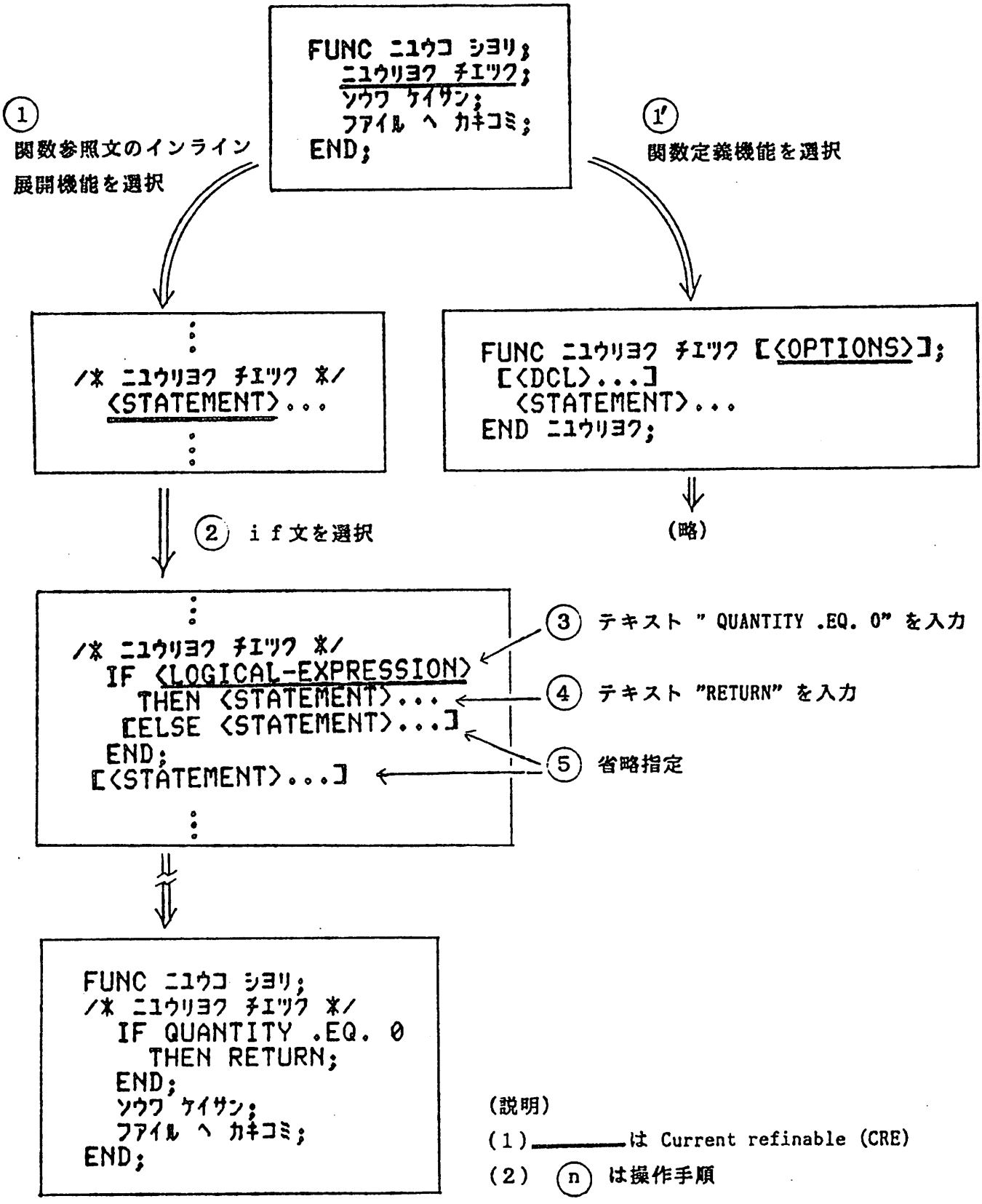


図８．１　構造エディタを用いた段階的詳細化の例

と共に，各モジュールの機能を単純化できるため，テストのための入力データ作成や結果の確認は比較的容易である。

一方，第１項は動的テストに本質的問題である。即ち，構造化プログラミングの提唱者であるE.W.Dijkstraが「テストは誤りの存在を示すことはできるが，誤りのないことは示せない」と明言するように，テストは「誤りを見つける」目的で行われる。そのため，プログラムの品質は，もし誤りがあれば必ずそれを検出できるような効果的なテストデータを用いてテストしたか否かに依存してしまうからである。

8.4.2 構造テストの問題点

効果的なテストデータを作成する技法の１つとして構造テスト法がある。これは，プログラムの制御構造を制御フローグラフと呼ばれる有向グラフで表わし，そのパス解析に基づいてテストケースを選ぶもので，その時の選択基準としてテスト網ら性を表わす尺度が用いられる。本方式による基本的なテストデータ作成手順は次のようなものである。

(1) あるテスト網ら基準を満たすように，パスの最小セットを選ぶ。

(2) 各パスを通過するためのパス条件を選ぶ。

(3) 各パス条件を満たす入力データ値を求める。

そこで，この方式を実施するためには，まずテスト網ら基準を定める必要があるが，その基本となるものはすべてのパスを網らする基準である。しかし，通常のプログラムでは繰返し処理を含むため，パスの数が膨大となり，全パス実行は不可能な場合が多い。そのため，実際には，パスの構成要素，即ちある分岐点から次の分岐点までの部分パスに着目した次のような尺度が用いられる。なお，この部分パスをｄｄパス（decision − to − decision path）と呼ぶ。

$$C_{dd} = \frac{\text{実行済みのｄｄパスの数}}{\text{ｄｄパスの総数}}$$

これは，被テストプログラム内のすべての分岐方向のテスト実行を意図することから分岐テスト法と呼ばれ，次の用途に用いられる。

(1) テスト十分性の指標として用いられ，この網ら率が高いほど被テストプログラムの品質も高いとみなす。

(2) テストデータの漏れの検出に用いられ，未実行のｄｄパスを通過するようなテストデータを追加する。

ところが，この基準は各々の用途に対して次のような問題点を有する。

(1) 品質評価に用いた場合，実際より過大な評価値になる。

(2) テストデータ選択に用いた場合，冗長なデータが混入しやすい。

例えば，図8.2(a)のプログラムについて考えてみる。図の(b)はその制御フローグラフである。この図において，パスabcegを通過するようなテストデータとして「１０１」を入力したとする。この場

合，５個のｄｄパスのうちの３個が実行されるので，C_{dd}は６０％となるが，テストが６０％終了したとは言い難い。また，２つ目のテストデータとして，未実行ｄｄパスｆを通過するようなパスabcefcegを選び，「１，１０１」を入力したとする。この場合，２番目のテストデータは最初のテストデータのｄｄパスをすべて含むため，分岐テストの視点からは最初のテストデータは冗長になってしまう。

このような問題が生じた理由は，パス網ら性に本質的でないｄｄパスａｂ，およびｇを他のｄｄパスｃｅ，ｆおよびｄと同等に扱ったためである。

8.4.3 テスト網ら基準の改良[C15)]

先に述べた分岐テストの問題点は，テスト網ら性に本質的な分岐とそうでない分岐を分離し，前者にのみ着目する方式によって改善できる。ここでは，その具体的方法について述べる。

まず，有向グラフの２つのアークｐとｑの間の関係について，次の概念を導入する。

〔定義〕　ｐを含む任意のパスが必ずｑを含む時，ｑをｐの相続子アークと呼ぶ。

ここで，ノードｘからノードｙへのアーク（ｘ，ｙ）が相続子アークとなるのは，（ｘ，ｙ）なるアークが１つで，かつ次の４条件のいずれかを満たすものである。なお，ＩＮ（ｘ），ＯＵＴ（ｘ）はノードｘの入力および出力アーク数，ｗは隣接ノード，ＤＯＭ（ｗ）とＩＤＯＭ（ｗ）はｗの支配子および逆支配子の集合，Ａはアークの集合とする。

（Ｒ１）　$IN(x) \neq 0 \wedge OUT(x) = 1$

（Ｒ２）　$IN(y) = 1 \wedge OUT(x) \neq 0$

（Ｒ３）　$OUT(x) \geq 2 \wedge x \in IDOM(w)$，但し $\overset{\forall}{w} \in \{ w \mid (x, w) \in A \wedge w \neq y \}$

（Ｒ４）　$IN(y) \geq 2 \wedge y \in DOM(w)$，但し $\overset{\forall}{w} \in \{ w \mid (w, y) \in A \wedge w \neq x \}$

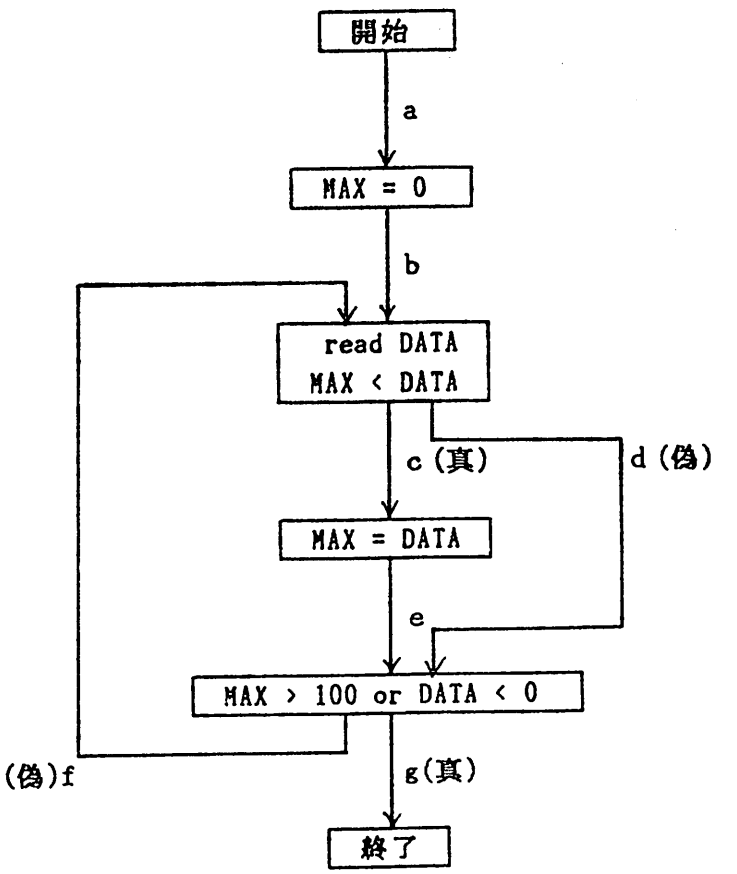
パステストの観点からは，このような相続子アークは被相続子アーク（上記定義のｐ）の網ら情報を相続するため注目する必要がない。そこで，これら４条件を相続子消去規則として適当な手順で適用することにより，制御フローグラフを相続子アークのないグラフに簡約化できる。このような相続子簡約グラフ上でのアークを原始アークと呼び，次のようなテスト網ら基準を導入する。

$$C_{pr} = \frac{\text{実行済みの原始アークの数}}{\text{原始アークの総数}}$$

この基準は，C_{dd}がすべての分岐に着目するのに対して，原始アークに対応する分岐，即ちパステストに本質的な分岐にのみ着目することにより，従来の分岐テストの欠点を改善している。例えば，図8.2の例では，この方式を適用すると，制御フローグラフは図8.3のように簡約化され，本質的な分岐は，ｃ，ｄ，ｆの３個であることがわかる。従って，テストデータ「１０１」を入力した時C_{dd}は６０％だが，C_{pr}　では３３％となる。

```
func MAXIMUM : int ;
  var DATA : int ,
      MAX : int ;
  MAX=0 ;
  repeat
    read(5,F100) DATA ;
    F100 : format(F10.2) ;
    if MAX .lt. DATA
       then MAX=DATA ;
    end ;
  until (MAX .gt. 100 .or. DATA .lt. 0)
  end ;
end MAXIMUM ;
```

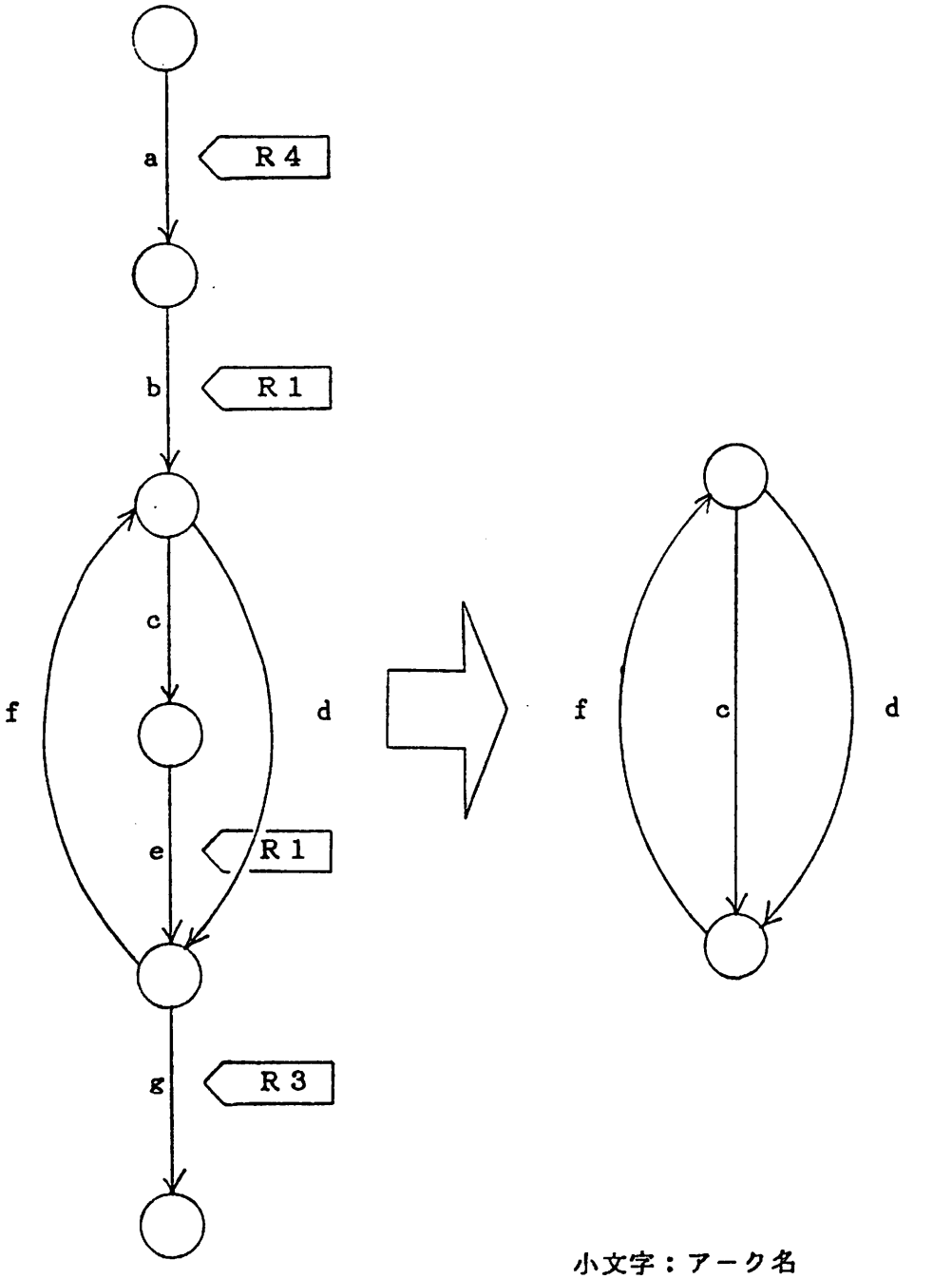
（a）　ＳＰＬプログラム例



a－g：アーク名

（b）　制御フローグラフ

図８．２　プログラムから制御フローグラフへの変換例



（a）　元の有向グラフ　　（b）　簡約化されたグラフ

図8．3　Cprのための有向グラフ簡約法

8.5 プログラミング環境の統合化[C10]

8.5.1 統合化の思想

本章では，これまでに幾つかのプログラミングツールについて述べてきたが，言語支援系としては，個々のツールを充実させる他に，これらのツール群を1つのプログラミング環境として統合化していくことが重要である。この点に関して，従来は各ツールを個別に開発し，ユーザに提供していたため，次のような問題があった。

(a) 各ツールの相互独立性が強いため，お互いの入出力仕様が異なるとか，機能が重複したり，欠除したりする不都合が生じ，複数のツールを用いる場合の使い勝手が悪い。

(b) エディタやデバッガなど，多くのツールは，プログラミング言語に依存しないようにして汎用性を持たせているために，機能的には低水準である。

そこで，このような問題を解決するためには，

(a) 各ツールの有機的結合による1システム化，

(b) 各ツールの言語適応化による高機能化，

の方針に基づいた統合的プログラミング環境の構築が望ましい。このうちの第2の方針については，既に本章でとり上げた会話型最適化ツール，構造エディタ，構造テストツールなどに反映されている。一方，第1の方針の「有機的結合」は，次に述べるように，方法論，技法，データ，ユーザインタフェイスの共有によって実現しうる。

(1) プログラミング方法論の共有

プログラミングの主要な作業である設計，作成，検証の間に一貫性を持たせるために，各部分の支援ツールに同一のプログラミング方法論を反映させる。例えば，既に述べたように，段階的詳細化法やデータ抽象化法に基づいて設計されたプログラムモジュールは，その支援機能を有する言語で記述できること，さらにその設計法を誘導する構造エディタや導かれたプログラムのモジュール構成に適した構造テストが行えることが望ましい。

(2) プログラムの解析技法の共有

特定言語向きのツールはプログラム解析処理を伴うが，ツール間で共通な部分も多い。例えば，構文解析は，コンパイラの他にも，構造エディタや構造テストツールでも必要である。そこで，構文解析やフロー解析などの処理を共有化することにより，各ツールの高機能化が容易に行える他，対象とする言語の仕様がツール間でくい違うのも防止できる。

(3) データの共有

プログラムや各ツールの入出力データなどをデータベース化し，それらの間の関係を保持しておくことにより，ツール間インターフェイスの不一致を防止できる他，プログラム変更に伴う再コンパイルや再テストの自動化，あるいは関連データの問合せなどの機能を付与できる。

(4) ユーザインターフェイスの共有

ツール毎にコマンド言語や端末操作法が異なっていてはシステムとしては使いづらい。ユーザ

インターフェイスの一様性は統合化の必須条件である。

8.5.2 ソフトウェア構成法

以上に述べた統合化の思想に基づく言語適応型プログラミング環境のソフトウェア構成を図8.4に示す。図の(a)はプログラム解析処理をバックエンド，コマンド解析処理をフロントエンドとして，従来の個別ツールを両者の間に隠ぺいした形を表現している。また，図の(b)は，図の(a)をユーザ側から眺めたもので，ユーザとプログラムの間の階層的データ抽象構造を表現している。即ち，各ツールはプログラム解析処理を経由しないとプログラムを参照できない構造を示している。

8.6 結　言

新しい方法論を反映したプログラミング言語の適用をより効果的に行うために，ＳＰＬプログラムの開発支援環境について検討した。

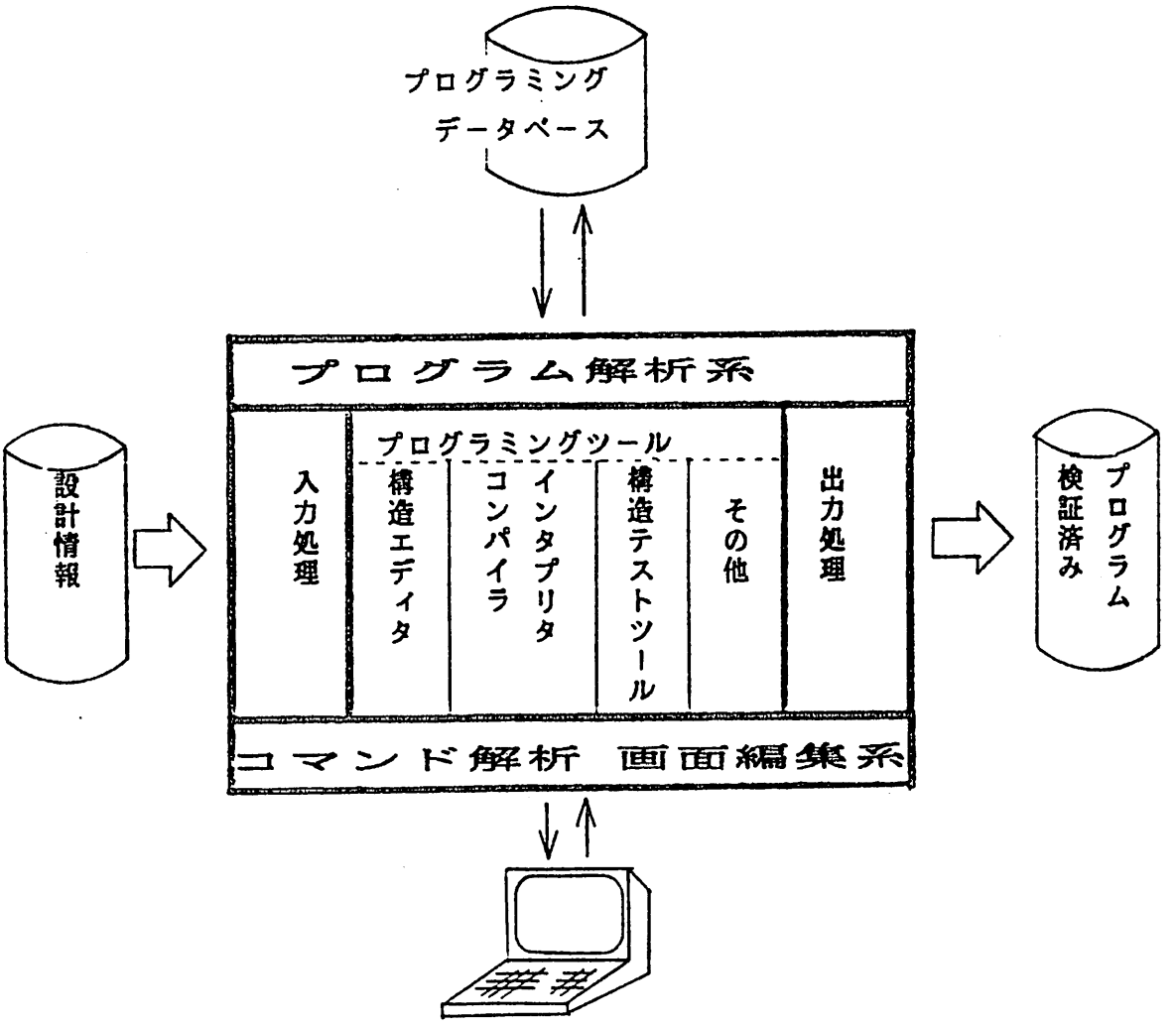
第1には，オブジェクト効率の厳しい分野に構造化プログラミングを適用するために，第1段階ではプログラムの構造化に専念し，第2段階で性能要求に応じてプログラムの最適化を会話型で行う2段階プログラミング法を考案した。そして，その適用実験を行い，実用性を確認すると共に，最適化コマンド機能を抽出した。

第2には，ＳＰＬの有する段階的詳細化機能や構造化コーディングを誘導する構造エディタについて，基本機能の設計を行った。そして，文法的誤りを防止しながら効率良くプログラムを作成したり，修正したりする方式を具体化した。

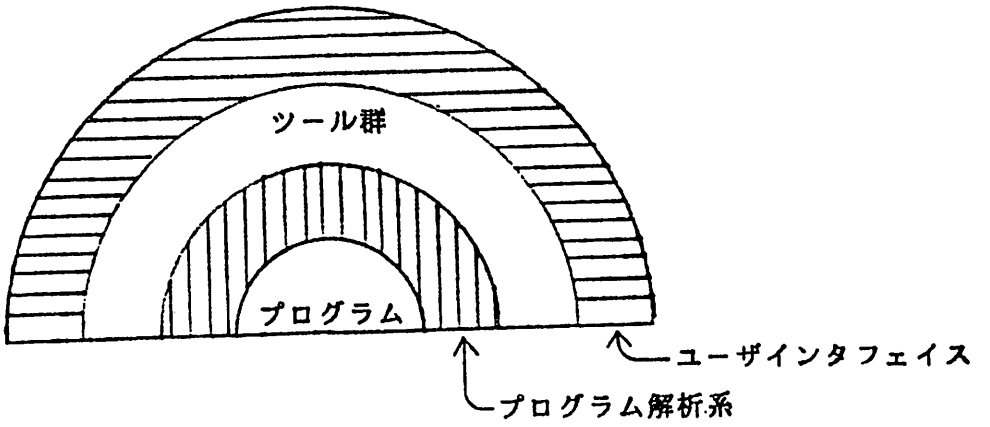
第3には，従来の構造テストに用いられてきたテスト網ら基準，即ち，全分岐に対するテスト実行済み分岐の割合に着目する方法は，品質の過大評価や冗長なテストデータ選択などの欠点を有することを明らかにすると共に，パステストに本質的な分岐にのみ着目する新しい基準を考案した。そして，この本質的な分岐をその他のものと識別する方法を導いた。

最後に，以上の言語支援系ツールをプログラミング環境として1システムに統合する方法を検討した。そして，各ツールの有機的結合のためには，プログラミング方法論，プログラム解析技法，データおよびユーザインターフェイスの共有化が必要であることを示すと共に，それを実現するソフトウェア構成を設計した。

なお，構造エディタについては既にプロトタイプを開発済みであるが，会話型最適化システムについては，最適化コマンド設計およびその正当性の自動検証方式設計を終了している段階で，実際のインプリメントは行っていない。また，構造テストツールについては，従来の分岐テストの網ら基準を用いたものは開発済みであるが，新しい基準に対するものは未開発である。従って，本章で述べた支援ツールすべての実用化と適用評価は今後の課題である。



(a) ソフトウェア構成



(b) 階層的データ抽象構造

図８．４ 言語適応型プログラミング環境のシステム構成