4 プログラミング

4.1 概　　要

プログラミングとは，設計仕様書の内容をプログラミング言語を用いて計算機入力可能な形で記述することである．すなわち，このプログラミングは次章のテストとともに，要求定義から始まったプログラム開発過程の最終生産物を作ることである．このプログラミングとテストの作業量は，プログラム開発の全作業量の半分以上を占めており，ツールの良し悪しで効率が大きく左右される．

このツールは，計算機が世の中に登場し始めた1950年代から絶えず進歩してきた．最近の十数年をみても，プログラミング言語はアセンブラから高級言語へ，プログラムの入力，修正はカードパンチから画面エディタの利用へ，プログラムデバッグは16進ダンプリストからデバッガの利用へと大きく変化してきた．

このツールの目的には，プログラミングに本質的な知的作業を支援することと計算機利用に伴なう手作業を軽減することがあるが，従来は手作業軽減が主になっていた．そのため，プログラミングに本質的な知的作業はプログラマ個人の能力に依存し，プログラム開発効率に対する個人差は数倍にもなっている．最近では，この個人差をできるだけ少なくするために知的作業支援型のツールが開発され，普及し始めている．

本章では,

(1) プログラムの記述に用いるプログラミング言語.

(2) プログラムの計算機への入力と修正に用いるエディタ.

(3) プログラムを計算機の実行可能な機械語形式に変換するコンパイラ.

(4) プログラムを解析して誤り検出などに利用する静的プログラム解析ツール.

などについて述べる.

4.2 プログラミング言語

4.2.1 プログラミング言語の種類

プログラミング言語の主目的は,人間と計算機とのインタフェースをできるだけ人間側に近づけることである.そのため,言語は記述水準の高級化という形で発展してきたが,その具体的な開発目的が多種多様であることから,世の中には数千に及ぶ計算機言語が存在するといわれている.その開発目的を分類すると次のようなものが考えられる.

(1) 特定の問題に適した言語

計算機言語は程度の差はあってもすべて特定の問題に適しているといえるが,代表的な例としては,科学計算用の FORTRAN,事務処理用の COBOL,リスト処理用の LISP,文字列処理用の SNOBOL,シミュレーション用の SIMULA などがある.

(2) 特定のプログラミングパラダイムを実現した言語

記述対象となる問題の範囲がある程度汎用向きで,特定のプログラミングパラダイムと強く関連した言語がある.それらに共通の特徴は,現在主流となっているフォンノイマン型計算機のアーキテクチャ,すなわち,メモリ上にプログラム自身およびデータ領域を確保し,中央演算装置(CPU)がプログラムを読み出して実行する方式とは無関係に,プログラミング言語はあくまで人間の思考に合った表現形態を支援すべきであるという考え

に基づいていることである．実際，誰でも最初に学んだプログラミング言語に基づくプログラミングパラダイムからなかなか逃れられないことはよく知られており，よいプログラム作りはよいプログラミング言語の選択に始まるともいえる．

代表的な例としては，述語論理をベースに定理証明の過程をプログラム

```
  INTEGER FUNCTION FACT(N)
    FACT=1
    DO 10 I=1,N
10    FACT=I*FACT
    END
```

(a) FORTRAN(手続き型言語)

```
① function FACTORIAL(N : integer) : integer ;
    var F, I : integer ;
    begin
      F : =1 ;
      for I : = 1 to N do F : =I*F ;
        FACTORIAL : =F
    end
② function FACTORIAL(N : integer) : integer ;
    begin
      if N=0 then FACTORIAL : =1 ;
             else FACTORIAL : =N*FACTORIAL(N-1)
    end
```

(b) Pascal(手続き型言語)

```
(DEFUN FACTORIAL(NUM)
  (COND ((ZEROP NUM) 1)
        (T (TIMES NUM (FACTORIAL (SUBI NUM))))))
```

(c) LISP(関数型言語)

```
factorial(0, 1).
factorial(N, M) : -T1 is N-1. factorial(T1, T2), M is N*T2.
```

(d) Prolog(論理型言語)

```
factorial : x
  | fact |
  fact←1.
  1 to : x do: [ :n | fact←n* fact].
  ↑fact
```

(e) SMALLTALK-80(オブジェクト指向型言語)

図 4.1 いくつかのプログラミング言語による階乗計算プログラムの記述例

実行とみる論理型プログラミング言語 Prolog，ラムダ計算をベースに入力と出力の関係表現をプログラムとみる関数型言語 LISP，メソッドと呼ばれる手続きの集まりからなるオブジェクトの間でのメッセージ送受信によりプログラムが実行されるオブジェクト指向型プログラミング言語 SMALLTALK-80* などがある．

図 4.1 に階乗計算のプログラムをいくつかのプログラミング言語で記述した例を示す．通常，n の階乗 n！を計算するのに，手続き型言語ではループを用いて，

$$n!=n(n-1)\cdots 1$$

の計算をするが，再帰的定義機能を有する言語では，

$$n!=n\{(n-1)!\}$$

をそのまま簡潔に表現できる．

(3) 特定の利用者向けの言語

ビジネス分野への計算機の浸透とともに計算機にやらせたい業務は飛躍的に増加しているがプログラマの数は限られている．そこで，プログラマのような専門的教育を受けなくても，日常，事務系の業務担当者が簡単に計算機を利用できるための言語が必要とされている．エンドユーザ言語，データベース問い合わせ言語，事務処理用簡易言語，第 4 世代言語などと呼ばれるものがこの分類に入る．

(4) 特定のハードウェアに依存した言語

特定のハードウェア装置を制御する目的で開発された言語がいくつかある．プロセス制御用言語は，鉄鋼，電力，化学等のプラントのプロセスに用いられ，プラントと制御用コンピュータのインタフェースとなるプロセス入出力機能や高速の応答時間を実現するリアルタイム処理機能などに特徴がある．数値制御用言語は，工作機械の制御に用いるもので先駆的な例として APT(Automatically Programmed Tools)がある．最近では，製造工程の自動化をめざす CAM(Computer Aided Manufacturing)の発展

* SMALLTALK-80 は Xerox 社の登録商標である．

とともにロボット制御言語なども実用になっている.

(5) 汎用プログラミング言語

ここでの汎用とは万能という意味ではなく，元来は上記(1)～(4)に含まれている特定の目的で開発された言語であっても，たとえば，FORTRANのように比較的広範囲に用いられているものを含み，一般的な手続き的処理が無理なく記述できるものをいう．これらの言語を総称して手続き型高級言語と呼び，現在普及しているものを中心に次の項で述べる．特に，その中でプログラムの信頼性と生産性の向上に有効と思われる構造化プログラミング言語については項を分け，4.2.3 で述べる.

4.2.2 手続き型高級言語

現在使用されている計算機の大半が採用しているフォンノイマン型アーキテクチャが確立された1950年前後のプログラムの記述は，計算機の各機械命令に対応した「1」と「0」のビット列の符号を用いて行なわれた．このような機械語によるプログラミングはまもなくビット列の符号を記号化してニーモニックコードで表現するアセンブリ言語の使用へと発展したが，計算手順を手続き的に記述するというプログラミング方式はこの当時から今日まで変わっていない.

しかしながら，「計算機もプログラムがなければただの箱」といわれるように，よいプログラムを効率的に作成することは当初からの課題であった．1950年代後半から1960年代前半にかけて，FORTRAN, COBOL, ALGOL, PL/I などの手続き型高級言語が次々と開発され，プログラムの生産効率が飛躍的に向上した．その後，1970年代はじめまでに BASIC, Pascal, C 等も開発され，現在も使用されている代表的な手続き型高級言語が出揃った.

これらの言語の特徴を以下に簡単に述べる.

(1) FORTRAN

1955年に IBM 社の J. Backus らが開発したもので，1966年には標準規格化された．最新の FORTRAN 77 は1978年に標準規格化されたもの

で，従来の科学計算向きの機能に加え，プログラムの実行の流れを示す制御構造を分かりやすく記述するための構造化プログラミング用の制御文などがつけ加えられた．

実用面では，他機種への移行性が高いこと，高性能のオブジェクトプログラムを生成する最適化コンパイラがあること，プログラミング支援ツールが豊富であることなどの利点がある．図 4.1（a）に記述例を示す．

(2) COBOL

1959年に計算機関係のユーザとメーカで組織する CODASYL(the COnference on DAta Systems Language) の委員会で開発されたもので，1968年に標準規格化された．比較的簡単な構文仕様で，事務処理プログラムを英語（自然語）風に記述できる．計算機の利用目的では事務処理が圧倒的に多いということもあり，COBOL は世の中で最も広く使われている言語である．1970年代後半には構造化プログラミング用の機能がつけ加えられた．

実用面では，FORTRAN と同様に他機種への移行性が高いこと，プログラミング支援ツールが豊富であることなどの利点があるが，特に定形的な業務プログラムの開発手順の標準化と自動化が進んでいる．以下に階乗計算のプログラム記述例を示す．

```
PROCEDURE DIVISION
      USING N FACT.
   MOVE 1 TO FACT.
   PERFORM
      VARYING I FROM 1 BY 1 UNTIL I>N
      COMPUTE FACT=I*FACT
   END-PERFORM.
```

(3) ALGOL

1960年頃に開発され，Algorithmic Language に由来する ALGOL 60 という名前で規格化されている．この言語の開発目的は，FORTRAN などの既存の言語の仕様にはあいまいなところが多く，細かい仕様がコンパ

イラごとに異なるという不便を解決するために，言語仕様を精密に定義したきれいな言語とすることであった．

その結果，プログラム内で使用する変数の名前とデータ型はすべて明示的に宣言する機能，変数や手続きの名前の有効範囲を明確にする複合文および標準的な制御構造を表わす条件文や反復文などの構造化プログラミング機能，手続き呼び出しにおける実引数と仮引数の間の対応関係を明確にする機能，などの重要な概念を導入した．

実用面では，画一的な仕様定義の難しい入出力機能を言語仕様の対象外にしたことやよいコンパイラがあまり開発されなかったことから普及の程度は低い．しかしながら，PL/I, Pascal 等，その後に開発された言語に与えた影響は大きい．以下に階乗計算のプログラム記述例を示す．

```
integer procedure factorial(n);
  value n;
  integer n;
    factorial:=if n=0 then 1
                      else n×factorial (n−1)
```

(4) PL/I

1965年に IBM 計算機のユーザ団体である SHARE が中心になって開発した言語で1979年に標準規格化されている．この言語は，科学計算から事務処理まで広範囲の応用プログラムを記述できるようにするために，FORTRAN, COBOL, ALGOL の機能を包含する仕様にすること，およびハードウェアやオペレーティングシステムの機能をアセンブリ言語を用いないで直接利用できるようにすることを目的に開発したため，言語仕様が大きくなっている．

実用面では，適用範囲が広く，ほとんどのプログラムはこの言語だけで記述できる利点があるが，コンパイラが大規模になるため大型計算機でないと利用できないことや不用意なプログラミングをするとオブジェクトプログラムの効率が極端に悪くなる場合があるので注意する必要がある．以下に階乗計算のプログラム記述例を示す．

```
FACTORIAL : PROC(N) RETURNS(FIXED BIN(31)) ;
   DCL N FIXED BIN(31) ;
   DCL I FIXED BIN(31) ;
   DCL F FIXED BIN(31) ;
      F= 1 ;
      DO I= 1 TO N ;
         F=I*F ;
      END ;
      RETURN(F) ;
END FACTORIAL ;
```

(5) BASIC

1964年にダートマス大学の T. E. Kurtz らによって開発された言語で1978年に標準規格化されている．この言語は，修得が容易で，計算機を簡単に利用できるようにすることを目的としており，タイムシェアリングシステム下で動くものが最初であった．その後，マイクロコンピュータの発展とともに普及が進み，小さなプログラムの開発に用いられてきた．そのため，処理系もインタプリタ型が主流であったが，マイクロコンピュータの高性能化とともに BASIC も実用的な規模のプログラムに用いられるようになり，コンパイラも開発され，言語仕様も徐々に拡張されている．以下に階乗計算のプログラム記述例を示す．

```
10 INPUT N
20 LET F= 1
30 FOR I= 1 TO N STEP 1
40 LET F=I*F
50 NEXT I
60 PRINT F
70 RETURN
80 END
```

(6) Pascal

1970年頃にチューリッヒ工科大学 (ETH) の N. Wirth によって開発された言語で1983年に標準規格化された．学生にプログラミング技法を教えるのに適した教育用言語として開発されたため，言語仕様が簡潔なわ

りに，データ型が豊富で，構造化プログラミングも可能になっている．コンパイラもコンパクトになるように工夫されている．マイクロコンピュータの発展とともに急速に普及したが，標準規格化が遅れたため，実用的規模のプログラム開発のための拡張機能部分が異なるものが出まわっている．また，実行時に配列変数のインデックスの範囲チェックをするなどのため，オブジェクトプログラムの性能は必ずしもよくない．図 4.1（b）に記述例を示す．

(7) C

1972年頃に Bell 研究所の D. Ritchie によって開発された言語で，1983年以来，標準規格化を推進中である．この言語は UNIX* システムの記述に使用されていることからも分かるように，アセンブリ言語の代わりとなるシステム記述言語として開発されたもので，他の高級言語に比べ，細かい記述が可能である．オブジェクトプログラムの実行効率もよい．

実用面では，マイクロコンピュータを中心とした種々の機種間での移行性が高いという利点が大きい．以下に階乗計算プログラムの記述例を示す．

```
factorial(n)
  int n;
{
  if(n==0) return(1);
  else return(n*factorial(n-1));
}
```

4.2.3 構造化プログラミング言語

計算機の応用分野は，ハードウェア技術の飛躍的な進歩を背景に，拡大し続けている．その結果，計算機の利用形態はますます複雑化，多様化の傾向を強めており，この要求に応えるためのソフトウェアは大規模化の一途をたどっている．しかしながら，従来のプログラミング技術だけでは信頼性の高い大規模

* UNIX は AT&T の米国での登録商標である．

ソフトウェアを効率よく作成することには限界があった．この問題は1970年頃から顕著になり始め，「ソフトウェアの危機」と呼ばれた．

そこで，ソフトウェアの信頼性と生産性向上のために，1970年代に二つのアプローチがとられた．その第1は「構造化プログラミング」という言葉に代表されるプログラミング方法論と言語に関する研究である．第2のアプローチは「ソフトウェア工学」という言葉に代表される研究で，ソフトウェア開発工程のあらゆる局面，すなわち，要求定義，設計，プログラミング，テスト，保守・運用の各々を対象とした方法論，技法，ツールの開発が行なわれてきた．後者は本書全体に対応するものであるが，ここでは特に前者の構造化プログラミング言語について説明する．

(1) 書きやすさ vs. 読みやすさ

よいプログラミング言語の評価基準としては，マニュアルが薄い（修得容易性），誤りが簡単に検出できる（検証容易性），種々の計算機で使用可能（機種独立性，移行性），コンパイラが作りやすい（実現容易性）など，数えあげればきりがない．どの基準を重視するかは，その言語で記述するプログラムの開発目的，開発体制，開発環境，開発者のレベル，規模あるいは開発後の稼動環境，要求性能，保守体制，使用期間など種々の要因で決定される．

ここでは，特にプログラミング言語の基本的な評価基準である記述容易性（writability）と理解容易性（readability, understandability）について述べる．

記述容易性とはプログラム化しようとしている処理手順がいかに簡単に記述できるかという指標であり，前節で述べた手続き型高級言語の主目的はここにあったといえる．すなわち，アセンブリ言語よりもできるだけ少ない行数で記述できること，言い換えれば，高級言語で記述したプログラムからコンパイラによる機械語プログラムへの展開率の大きさを重要視するという意味で「量的高級化」が追求された．

しかしながら，プログラムの信頼性と生産性の向上への要求の高まりと

開発後の拡張や改造を含む保守コストの増大とともに，以下のような理由でプログラムの読みやすさの方が書きやすさよりも重要になった．

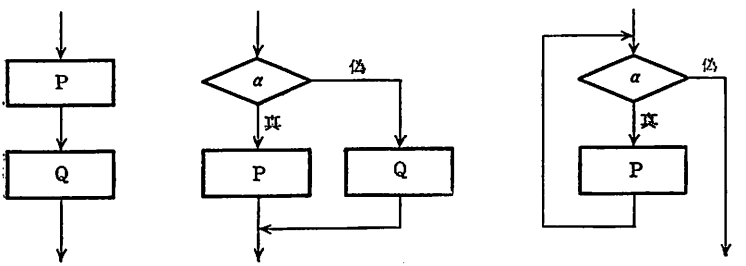
① 開発時に，プログラムを書くのは一度でも，読む回数は多い．

② 開発後のプログラムの改造は開発者と別の人が行なうことが多い．

プログラムの読みやすさ，すなわち理解容易性とは一言でいえばプログラムの構造がきれいであることである．従来の「量的高級化」に代わり，このような「質的高級化」を実現する言語をここでは構造化プログラミング言語と呼ぶ．

(2) go to 文の排除

プログラミング言語の重要な機能として，プログラムの実行の流れを制御する命令がいくつかあるが，その一つである goto 文のような無条件分岐命令を多用するとプログラムの制御構造の理解が非常に難しくなる．そこで，goto 文は使用せず，プログラムの制御構造は，図 4.2 に示すような逐次処理，選択処理，反復処理の組み合わせで表現する方式が，1968年に E. W. Dijkstra によって提案された．ただし，例外処理の記述などは goto 文を用いた方が素直な場合があるので，goto 文を全面禁止にする必要はない．



(a) 逐次処理　(b) 選択処理　(c) 反復処理

図 4.2 構造化コーディングのための制御構造の基本形

(3) 情報隠蔽 (information hiding)

前項の goto 文の排除は一つのモジュール内のプログラム構造をきれい

にするものであるが，プログラム全体の規模が大きくなると個々のモジュール内の構造よりもモジュールの分割の仕方およびその結果として得られるモジュール間の関係の方がプログラム理解に重要である．情報隠蔽はこのようなモジュール分割の評価基準として導入された概念であり，

① 各モジュールの機能，すなわち他のモジュールから呼び出されるときに必要となる外部インタフェースをその機能の実現方式（インプリメンテーション）とは独立に決定すること．

② 各モジュールの実現方式に関係する詳細情報を他のモジュールから参照できないように隠すこと．

の2点がポイントである．このようにモジュール分割を行なえば，あるモジュールのプログラム構造を理解するときにはそれが呼び出しているモジュールの中味まで調べる必要がないことからプログラムの理解容易性が向上する．また，あるモジュールの実現方式を変更しても，外部インタフェースさえ変更しなければそのモジュールを呼び出しているモジュールを変更する必要がないので，プログラムの保守性も向上する．

このような情報隠蔽の考えに基づいた言語として抽象データ型を有するものがあるが，詳細は後の第(5)項で説明する．

(4) 段階的詳細化（stepwise refinement）

これは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．このとき，重要なことは，各段階の抽象レベルで閉じたプログラムを記述することである．たとえば，ある抽象レベルでデータを導入したとき，そのデータの構造に関する宣言のレベルとそのデータを参照する処理記述のレベルが同じ抽象度でなければならない．そして，次の段階ではまたそれらの詳細化を行なうという過程を繰り返し，最後に実際の計算機で処理できるレベルに到達すればプログラムが完成する．

たとえば，最初の1000個の素数を求めるプログラムを開発する場合を考えてみよう．

まずはじめにプログラムを次のように記述する.

[ステップ1]

"print first thousand prime numbers"

これはこれから作成するプログラムの要求仕様といえるレベルのものである. 次に, その具体的な処理方法として, テーブルPを導入し, まず1000個の素数をこのテーブルに書き込んだ後, まとめて印字出力をすることに決めるとステップ1のプログラムは次のように詳細化される.

[ステップ2]

"table P"
"fill table P with first thousand prime numbers"
"print table P"

さらにこのテーブルPのデータ構造を配列変数とする決定によりステップ2のプログラムは次のように詳細化される.

[ステップ3]

"integer array P [1 : 1000]"
"make for K from 1 to 1000
P [K] equal to the Kth prime number"
"print P [K] for K from 1 to 1000"

このような段階的詳細化を最終的なプログラムが得られるまで繰り返す.

この段階的詳細化技法の基本思想は, プログラミングというものは非常に知的で難しい作業なので, 「1度に一つの決定」(one decision at a time) だけをすることを繰り返しながら最終的なプログラムに到達することによってはじめて誤りのないプログラムを作れるというものである. これは従来, 詳細設計とコーディングの二つの工程に分かれていたプログラミングを一つの統一的手法で一体化したものと考えることができる.

ここでプログラムの理解容易性の観点から重要なことは, 詳細化の各段階がプログラムとして記述され, プログラム全体がモジュール階層構造となることである. そのためにはサブルーチンのような処理手順の詳細化だ

けでなく，処理手順の抽象度に合わせたデータ定義を可能とするためのユーザ定義データ型機能を有するプログラミング言語が必要である．その例を次に示す．

(5) データ抽象化 (data abstraction)

データ抽象化の基本的な考え方は，データの詳細な構造とデータ操作手続き群をまとめて定義し，カプセル化 (encapsulation) するとともに，そのデータへのアクセスはデータ操作手続きを通してのみ可能とするものである．このような方式は次のような利点がある．

① そのデータの参照側では，詳細なデータ構造を知る必要がなく，データアクセス時にはその処理内容に対応する操作手続きを呼び出すだけでよいので，より抽象的な水準でプログラムを記述できる．

② そのデータの定義側では，操作手続きの外部インタフェースさえ変更しなければ参照側への影響がないので，データ構造や操作手続きの処理手順（アルゴリズム）を自由に決めてよい．そのため，所要メモリ量や処理速度などの外部条件に応じて内部を変更するのも容易である．

データ抽象化の基本思想は先に述べた情報隠蔽と同じである．しかしながら，このような方法だけでは，類似の機能を有するデータがいくつも必要になったとき，その度にデータ構造とデータ操作手続きの詳細を定義する必要が生じる．そこで，実際には，データ抽象化機構は抽象データ型 (abstract data type) のユーザ定義機能として実現されることが多い．

ここでは，MIT の B. Liskov が開発したプログラミング言語 CLU の抽象データ型定義機能を紹介する．

たとえば，スタックは図 4.3 のように定義される．スタックとは複数のデータを下から上に順々に積んで保存し，データを除去するときは逆に上からとっていくものである．図の例では，1行目で stack という名前の抽象データ型は t というデータ型の引数を一つもち，操作手続きとしては create, push, pop を用意することを示している．2行目では stack 型を実際には t というデータ型の配列で実現することを示している．3行目

```
stack=cluster [t : type] is create, push, pop
      rep=array [t]
  create=proc ( ) returns (cvt)
          return (rep$new( ))
          end create
  push=proc (s : cvt, x : t)
        rep$addh(s, x)
        end push
  pop=proc (s:cvt) returns (t) signals (empty)
       if rep$empty(s)
         then signal empty
         else return (rep$remh(s))
         end
       end pop
end stack
```

図 4.3 CLU による抽象データ型スタックの定義（文献38)より引用）

以降では三つの操作手続きの各々について，入力引数や出力結果のデータ型や実際の処理内容を定義している．ここで定義された stack 型の使用例を次に示す．プログラムの中で，

```
stk : stack[int] : =stack[int]$create( ) ;
```

と宣言することにより，整数型データの収納に用いるスタックの実体（オブジェクト）が生成される．そして，この stack 型変数 stk の操作は，push および pop を用いて，

```
stack[int]$push (stk, 1) ;
v : =stack[int]$pop(stk) ;
```

のように行なう．CLU のデータ抽象化機能の主な特徴は次のようなものである．

① データ構造とその操作手続きを cluster と呼ぶ入れものの中にカプセル化できる．なお，この入れものの名前は言語により異なり，SIMULA や SMALLTALK-80 では class, Ada* では package, Alphard では form と呼ばれる．

② 抽象データ型の引数にデータ型を指定できるので，整数型データを収

* Ada は米国国防総省の登録商標である．

納するスタックと実数型データを収納するスタックを別々のデータ型として定義する必要がなく，上の例のように変数宣言時に引数で指定すればよい．Ada ではこのような仕掛けをジェネリックパッケージという機能で実現している．

③ 操作手続きの呼び出しは「抽象データ型名$手続き名」の形式で行なうため，手続き名は異なる抽象データ型間で同じ名前を用いてよい．

④ 操作手続きの定義内では，抽象データ型の詳細データ構造に対する操作が必要なので，**cvt** という特殊な型指定によりそれを可能としている．上の例では push という手続きの第1引数は stack 型であるが，定義側の仮引数の型指定を **cvt** とすることにより，stack 型データ s を配列とみなし，配列の最後にデータを追加する addh という配列型操作を可能にしている．

4.3 テキスト編集

4.3.1 エディタの種類

プログラムを計算機へ入力する物理媒体として，昔は紙テープやパンチカードが使用されていたが，磁気テープを経て，今は磁気ディスクが主になった．その結果，プログラムのソーステキストはディスク上のファイルとして保存され，プログラムの入力や修正はエディタを用いて行なう．

このエディタは，プログラム開発時に最も頻繁に使用されるプログラミングツールであり，この使い勝手の良し悪しがプログラムの開発効率に与える影響は大きい．それだけにエディタの評価基準には多面性がある．一応の基準としては，

① テキストおよびコマンドの入力方法

② テキストおよびメッセージの出力方法

③ 基本コマンドの充実度

④ 応答時間

⑤ 拡張機能の有無

⑥ 他のツールとの連係

⑦ ソーステキストファイルの保全性

などが考えられるが，エディタの機能は，それが稼動する計算機や端末装置のハードウェア仕様，通信回線の仕様，オペレーティングシステムの機能などの外部環境に制約される．さらに，利用者が初心者か熟練者か，頻繁に利用するか否か，端末は個人で専有できるか共有か，などの利用環境に加え，各自の好みの違いにまで影響される．

したがって，一口にエディタといってもその種類は多いが，プログラムの編集を主目的にしたものとして以下の5種類が考えられる．これらの違いは主にプログラムテキストをどういう概念で捕えるかに基づいているが，実際には外部環境の影響も受けている．

(1) 文字エディタ

これはプログラムテキストを切れ目のない一つの文字列として扱うもので紙テープのイメージに近い．行とかレコードの区切りを入れたければ改行文字を打ち込む．代表的なものとして TECO およびその流れをくむ EMACS がある．

(2) 行エディタ

これはプログラムテキストを行単位の文字列として扱うものでパンチカードの束のイメージである．代表的なものとして QED がある．

(3) 画面エディタ

これはプログラムテキストを横幅が一定の長さの紙に上から順に横書きしたものとして扱うもので，プリンタ出力したリストあるいは巻紙のイメージである．実際は，ディスプレイ画面上に表示されたテキストの任意のところを編集可能であることが最大の特徴である．したがって，画面エディタは文字エディタや行エディタと排他的な関係にはなく，むしろそれらの一つの実現形態と考えてよい．それぞれに対応する代表的なものとして EMACS と vi がある．

(4) 構造エディタ

これはプログラムテキストをそれを記述するのに用いたプログラミング言語の構文規則に合った構造をしているものとして扱う．実際にはプログラムを構文木へ変換したものを直接編集させるものや，構文要素の枠組み（テンプレート）を表示して穴埋めを誘導するものなどがある．先駆的なものとしては Interlisp がある．

(5) 図式エディタ

これはプログラムをテキストとして直接編集することはせず，プログラムの制御構造を図式表現したものを扱うもので，先の構造エディタをビジュアル化したものといえる．先駆的なものとして SDL/PAD がある．

以上のプログラムを主対象としたエディタの分類はまとめて表 4.1 に示す．本節では現在主流となっている画面エディタおよび将来性が期待される構造エディタおよび図式エディタについて詳しく述べる．

表 4.1 プログラムを主対象としたエディタの種類

種　類	テキストの概念	テキストのイメージ	例
文字エディタ	切れ目のないひとつながりの文字列	紙テープ	TECO
行エディタ	行単位に区切られた文字列	パンチカードの束	QED
画面エディタ	横幅一定の用紙に上から横書きされた文字列	プリンタ出力リスト（巻紙）	EMACS vi
構造エディタ	プログラミング言語の構文規則に合った構文要素の列	構文木	Interlisp
図式エディタ	プログラムの制御構造が図式表現されたもの	フローチャート（流れ図）	SDL/PAD

4.3.2 画面エディタ

現在，最も広く使用されているのは画面エディタである．ここでは，その代表的なものとして，EMACS と vi について述べる．

(1) EMACS

これは，MIT の R. M. Stallman が文字エディタ TECO を拡張して作成したもので，DEC の TOPS-20 上で稼動する．その後，LISP で記述し直したものやCで記述して UNIX 上で稼動するものもある．ここでは

DEC マシン上で端末に VT 100 を用いる場合を想定して機能概要を説明する.

画面構成は，上方部がソーステキストの表示領域であり，下方の数行がシステムメッセージの表示領域とコマンドオペランドの入力領域になっている.

画面エディタとして次のような特徴的な機能がある.

① カーソルがソーステキスト表示領域にあるときは常にカーソル位置の左側に文字挿入が可能な状態になっており，文字キーをたたくと即時にその文字が挿入され，カーソル位置以降のテキストが右へ1文字ずれる.

② コマンドは文字キーに割り当てられており，以下の3種類の入力に分けられる.

- コントロールキーと文字キーを同時に押す.（以下では「C—」と略す.）
- メタキーと文字キーを同時に押す. VT 100 端末ではメタキーがないので，代わりに ESCAPE キーの後に文字キーを押す2打鍵方式をとる.（以下では「M—」と略す.）
- コントロールキーとメタキーと文字キーを同時に押す. VT 100 端末ではその代わりに「C—Z」の後に文字キーを押す.

③ カーソルの移動コマンドは，プログラムテキスト向きのものとして，文字単位，行単位に1単位分だけ前後させたり，行やテキスト全体の先頭や最後尾へ位置づけるものがある. たとえば「C—P」を入力するとカーソルが1行上へ移動する.

④ 削除，移動，探索などの基本コマンドも豊富であるが，特に上記①で述べたように，カーソルの左側に常に文字挿入可能であるということから，カーソルは論理的にはカーソル位置の文字とその左側の文字の間にあると考えられるため，1文字削除については左右どちらの文字も削除可能となっており，削除キーを押せばカーソルの左側の文字，「C—D」

を押せばカーソル位置の文字が削除される.

⑤ ヘルプメッセージも充実しており，使用中に手元にマニュアルがなくても困ることはほとんどない.

EMACS では，このほかにも種々の便利な機能がライブラリに納められており，これらの豊富な機能を使いこなせば，プログラムの開発効率が大幅に向上するが，使い慣れるには少し時間がかかるため，玄人好みのする熟練者向きエディタといえる. EMACS の基本コマンドの一部を表 4.2 に示しておく.

表 4.2 画面エディタのコマンドの例

コマンドの機能	EMACS (文字エディタ)	vi (行エディタ)
カーソル移動： 右へ	C—F	スペースキー
左へ	C—B	h
上へ	C—P	k
下へ	C—N	j
画面スクロール： 1行上へ		C—E
1行下へ		C—Y
1画面上へ	M—V	C—B
1画面下へ	C—V	C—F
挿入： カーソルの左へ	常 時	i
カーソルの右へ		a
カーソルの上へ1行		O
削除： 1文字	C—D	x
1行		dd

(2) vi

これは UCB (カルフォルニア大学バークレイ校) で開発した画面エディタであり，UNIX の行エディタ ex を拡張したものになっている.

画面構成は，画面の上方部がソーステキストの表示領域になっており，その下にコマンド入力やシステムメッセージ出力の領域がある.

特徴的な機能としては次のようなものがある.

① EMACS はカーソルの左側に文字挿入が可能な状態が基本になっているため，コマンド入力にはコントロールキーやメタキーが用いられた

が，vi ではコマンド入力モードが基本になっているので，コマンド入力には普通の文字キーを用いる．たとえば文字列挿入コマンド「i」を入力すると挿入モードに変わり，それ以降の入力文字がカーソルの左へ挿入されていく．最後にエスケープキー（ESC）を入力するともとのコマンド入力に戻る．このように vi にはモードの概念があり，使用中に常に現在のモードがコマンド入力モードか特定のコマンドモードかを意識しておく必要がある．

② カーソルの移動コマンドは，プログラムテキスト向きの機能としては，文字単位と行単位の前後への移動，行の先頭，最後尾への移動，画面の最上位，最下位，中間への移動のためのコマンドがある．たとえば，「H」を入力するとカーソルは画面の先頭へ位置する．

③ 削除，探索，置換などの基本コマンドも豊富であり，同じコマンド名でも大文字と小文字で機能が異なるものもある．たとえば，「ta」は次にある文字 a を探索し，「Ta」は逆方向に文字 a を探索するコマンドである．また，「aa」と入力するとカーソルの右側に文字 a が挿入され，「Aa」では行の終りに文字 a が挿入される．

④ vi は行エディタ ex の拡張になっているため，互いに呼び出しが可能であるが，vi から最初に「:」を入力することで ex のコマンドを利用できる．

vi ではこのほかにもプログラム編集に便利な機能が多く，EMACS と同様に慣れれば大変効率のよい熟練者向きエディタといえる．代表的なコマンドの一部を表 4.2 に示す．

4.3.3 構造エディタ

(1) 構造エディタの目的

これまでプログラムの編集には行エディタや画面エディタが用いられてきたが，これらはいずれもプログラムを単なる文字列とみなして編集をするテキストエディタである．ところが編集の対象となるプログラムは実際

には，その記述に用いたプログラミング言語の文法規則に従った構文構造と意味をもっている．そして，プログラマがプログラムをみるときは常にそのプログラムが表現している処理手順やアルゴリズムがなにを意味しているかを考えている．

ところが，従来のようなテキストエディタを用いてプログラムを作成する方法では，プログラマはプログラムを思考対象とするときは意味のある文章としてみているにもかかわらず，操作対象とするときにはプログラムを意味のない単なる文字列としてみるという不自然さがある．このようなプログラムを読むレベルと書くレベルの間の意味的ギャップがプログラミング効率の低下や誤り発生の原因になっている（図4.4）．ここで述べる構造エディタはプログラミング言語の文法規則に従って構文要素単位のプロ

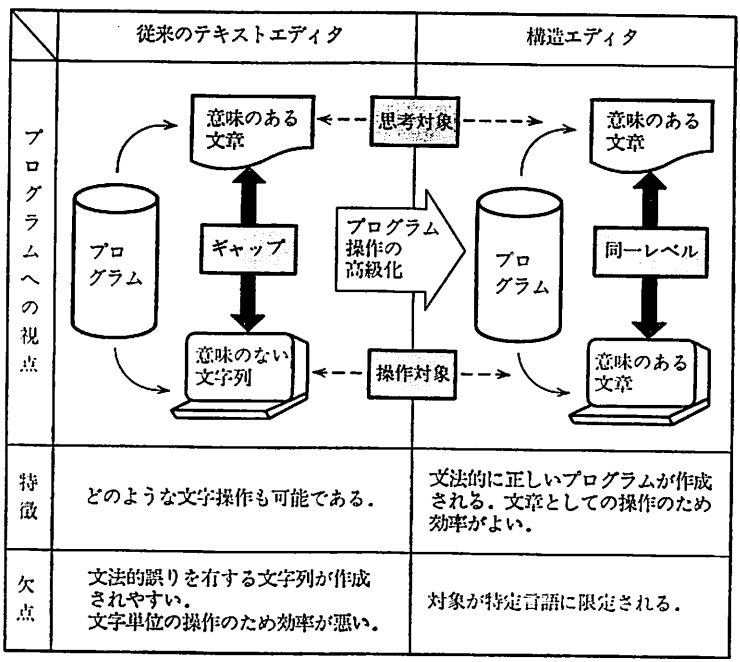


図 4.4　構造化エディタとテキストエディタの比較

グラム編集を行なうことにより，この問題を解決しようとするものである．

このような構造エディタが出現したもう一つの背景にプログラミング環境の発展がある．計算機が発展する初期の段階では，4.2 節でも述べたようにプログラミング言語の高級化によるプログラミング効率の向上が大きかったが，1970 年代のソフトウェアの大規模化とともに，このような言語の高級化だけではソフトウェアの生産性向上に限度があることが認識され，プログラミング環境の充実が重要視されるようになった．そして，このプログラミング環境は単なる個別ツールの寄せ集めでなく，次項以降で述べるように，構造エディタを含んだ特定言語向きの統合的環境とすることが重要である．

(2) 特定言語向きプログラミング環境

プログラミングを効率的に行なうためには次のような条件を満たす統合プログラミング環境が適している．

① 各ツールの有機的結合による 1 システム化

② 各ツールの言語適応化による高機能化

すなわち，これまで個々に開発されてきた単品ツールの結合と対象プログラムの記述言語に適応した支援機能の付与により強力な支援環境を構築できる．条件 ① の「有機的結合」は次に述べるような，方法論，技法，データ，ユーザインタフェー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プログラムの設計，作成，検証の間に一貫性をもたせるために，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．たとえば，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機能を利用できることが望ましい．

② プログラム解析技法の共有

特定言語向きツールはプログラム解析処理を伴なうが，ツール間で共有な部分も多い．たとえば，構文解析はコンパイラのほかにも，構造エディタやテスト網ら率計測用コード埋込み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可能になるほか，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，ツール間インタフェースの不一致を防止できるほか，プログラム変更に伴な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い合わせなどの機能を付与できる．

④ ユーザインタフェースの共有

ツールごと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．ユーザインタフェースの一様性は統合化の必須条件である．

以上のような有機的結合は必然的に先の統合化の条件②である言語適応化を実現する．以下では，このような特定の言語を対象とした統合プログラミング環境の例をいくつか紹介する．これらはすべて構造エディタを

表 4.3 特定言語向きプログラミング環境とその構造エディタの一覧表

No.	システム名*	開発元	開発時期	対象言語	内部表現
1	Interlisp	Xerox, BBN	1973	LISP	テキスト
2	MENTOR	INRIA	1975	Pascal	抽象構文木
3	Cornell Program Synthesizer	Cornell Univ.	1978	PL/CS	構文木と逆ポーランド記法
4	IPE (ALOE)	CMU	1979	GC	構文木
5	PERFECT (PARSE)	日立	1983	SPL ほか	抽象構文木

*：( ) 内は構造エディタ名

備えているが，その特徴を表4.3にまとめておく．

(3) Interlisp

リスト処理用言語 LISP の処理系の一つとして，MACLISP などとともに広く用いられている Interlisp は単なる言語処理系ではなく，構造エディタやデバッガを内蔵した統合プログラミング環境である．

構造エディタは構文要素単位の操作機能を有する．すなわち，プログラム内の操作対象は常にその部分式であり，ネスト構造をしたリストの上位や下位の式への移動や式単位の削除，挿入，置換，探索ができる．またプログラム構造を表現している括弧の削除，挿入，移動コマンドでは常に左右の括弧の対が保たれるようにしているほか，プリティプリンティングの機能も用意されている．

(4) MENTOR

MENTOR は，プログラムの設計，作成，検証，保守を支援する会話型プログラミング環境をめざしたシステムで，最初に Pascal 用構造エディタが開発されている．これは，抽象構文木をその木操作言語を用いて編集する方式で，たとえば，次のような Pascal プログラムは図4.5の抽象構文木として保存される．

```
if X>0 then P(X, Y)
       else begin Y : =Y * 2 ; X : =0 end
```

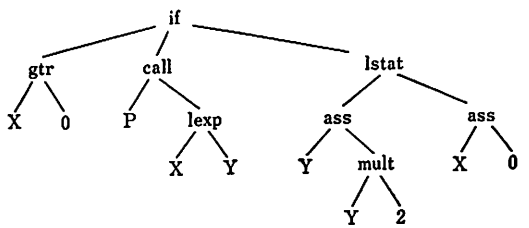


図 4.5 MENTOR の抽象構文木の例

木操作言語はコマンド形式で，操作対象とする節の位置指定やその上下

左右への移動，および指定位置での部分木の削除，挿入，置換ができ，構文誤りを生じるような指示は受けつけない．たとえば，図 4.5 の抽象構文木を操作して次のようなプログラムに変更する場合を考える．

```
if X>0 then begin
            Y : =Y * 2 ; P(X, Y) ; X : =0
            end
       else Z : =0
```

編集コマンドは次のようになる．

① S2 X S3 ——then 句と else 句の交換

② S2 S1 I S3 ——then 句に else 句を複写挿入

③ S3 C & ——else 句を次の端末入力で置換

④ Z : =0 ; ——端末入力テキスト

⑤ P ——アンパースしてプリント出力

なお，Sn は n 番目の子供をさすので，②は then 句の 1 番目の文の後への else 句の複写を意味する．

このほかにも，抽象構文木を上位の n レベル分だけアンパースしてテキスト表示したり，あるパターンを探索する機能がある．

(5) Cornell Program Synthesizer

これは，PL/I 風の教育用言語 PL/CS を対象にプログラムの作成，編集，実行，デバッグを支援する会話型プログラミング環境で，すでにコーネル大のほかにもいくつかの大学で教育用に使われている．プログラムの作成は，代入文などの一部の文を除いた文や宣言以上の構文要素は構文テンプレート挿入用コマンドを用い，そのほかは直接テキストを入力する．これらの入力可能位置は placeholder と呼ばれる非終端記号の表示位置である．構文要素単位の編集機能として，このほかにも要素の削除や削除した要素をファイルに保存しておき任意のところへ挿入するためのコマンドや隣接する要素，同一レベルでの前後の要素，上位または最上位の要素へのカーソル移動用コマンドがある．

このほか，コメント用コマンドを介して文の列を挿入すると桁下げ表示

したり，その文の列の代わりにコメントだけ表示する機能や対応する宣言がないと変数参照部分が高輝度表示になるような型チェック機能なども特徴的である．また，デバッグ機能として，未完成プログラムを実行しながらプログラムを詳細化する機能や，制御トレースやデータトレースの常時画面表示，構文要素単位の部分実行などの機能がある．

(6) Incremental Programming Environment

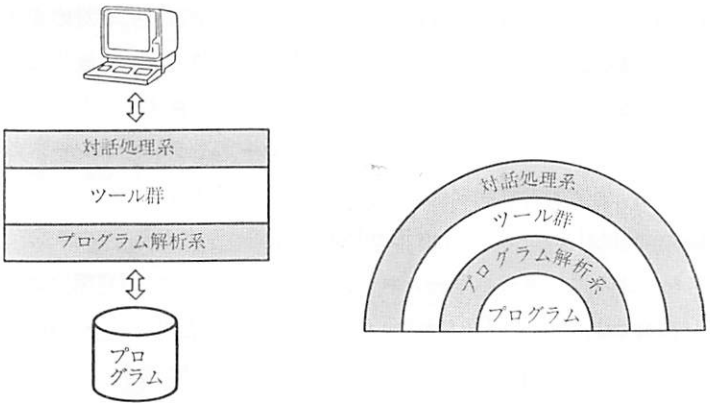
本システムは，カーネギーメロン大学のソフトウェア開発環境プロジェクト Gandalf の中で開発された，単一プログラムを扱う単一プログラマ用プログラミング環境である．対象言語はもともとは Ada で，プログラムの内部表現には Ada 用 TCOL が用いられているが，最初に開発されたものは C を変形した GC である．

その構造エディタは ALOE と呼ばれ，プログラムの作成はプログラムテンプレートの中のメタノード（非終端記号）にテンプレートまたは識別子や定数を挿入することを繰り返して行なう．この ALOE は構造エディタ生成システムによって各言語対応に作られるようになっており，そのための文法記述には，構文のほかに各メタノードに挿入可能な構文要素集合，プリティプリンティング情報，アクションルーチン名などが含まれる．

プログラムの実行は構文木をコンパイルして行なうが，そのときに条件付中断設定や変数値表示用コードあるいは不完全な手続き用スタブコードなどを埋め込む方式でデバッグを支援する．

(7) PERFECT

言語適応型統合プログラミング環境 PERFECT (Programming Environment For Editing, Compiling and Testing) は著者らが本節(2)で述べた統合プログラミング環境の設計思想に基づいて開発したものである．基本的なシステム構成は図 4.6（a）に示すようにプログラム解析系をバックエンド，対話処理系をフロントエンドとし，従来のツール群を両者の間に隠蔽したサンドウィッチ型ソフトウェアアーキテクチャとしている．図の（b）は，ユーザ側から眺めた概念図で，ユーザとプログラムの間の



（a）サンドウィッチ型ソフトウェアアーキテクチャ　　（b）階層的データ抽象構造

図 4.6　言語適応型プログラミング環境 PERFECT

階層的データ抽象構造を示している．

本システムはツール群の一つとして構造エディタ PARSE (Production And Reduction Structure Editor) を有している．この主な特徴を次に述べる．

(i)　段階的詳細化機能

E. J. Dijkstra が提唱する「1度に一つの決定をする」という原則に沿った段階的詳細化技法については先に 4.2.3 (4) で詳しく述べたが，PARSE はこの技法を支援するため最初に処理概要を自然語で記述できるようにしている．そのため，たとえば，

“データの平均を求める”

のような擬似文を導入している．

実際の段階的詳細化を図 4.7 の例で説明する．① ～ ⑦ は図の画面番号に対応する．

① ポップアップメニューから「擬似文」を選択し，「データの平均を求める」というテキストを入力する．

② 画面の最上段に表示されているコマンドビューポートの中から「詳細化」コマンドを選択すると，先の擬似文はコメントに変わり，「文」の

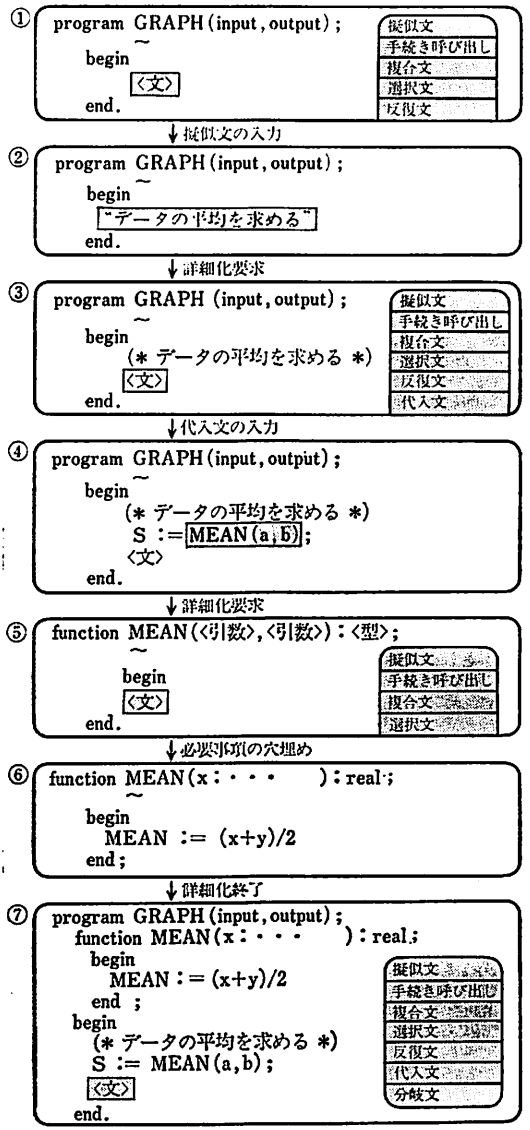


図 4.7 PARSE による段階的詳細化の例

入力要求状態になり，処理概要の詳細が記述可能となる．

③ 「代入文」を選択し，その右辺で「関数参照」を選択し，「MEAN (a, b)」を入力する．

④ その関数参照に対し，「詳細化」コマンドを選択すると関数定義の枠組みが表示される．

⑤ 同様にして，関数 MEAN の詳細化を行なう．

⑥ その終了後，もとの画面に戻る．

⑦ 引続き次の処理概要の入力状態になる．

このように従来は設計者が机上で行なっていた段階的詳細化を計算機が誘導してくれるため，経験の浅いプログラマでも段階的詳細化が可能である．さらに，詳細化の過程がプログラム内にコメントとして保存されるため，ドキュメント性が向上する．

(ii) 構造化コーディング機能

選択文，反復文，複合文を用いた構造化コーディングを支援する．そのため，入力可能な「文」として，先に示した擬似文，手続き呼び出し文のほかに，基本制御文の一覧をメニューとして表示する．ユーザはメニューから1項目を選択するだけで，希望する文が入力できる．このとき，PARSE は選択された文の種別に応じてその枠組みを自動生成し，清書（プリティプリンティング）を行なって表示する．

このように，初心者でも文法の複雑な基本制御文の入力が簡単に行なえるため，文法誤りがなくなり，キー操作数が削減されるとともに，従来はプログラマに任されていた字下げをエディタが自動的に行なうことにより，ソースプログラムの書式の標準化が図れる．

(iii) 構文要素単位の編集機能

「宣言文」，「選択文」，「条件式」などの構文要素単位のプログラム編集を可能とするため，各々の構文要素をその定形的な枠組みを示すテンプレートとユーザが入力すべきホールで構成する．ホールのうち，現在，詳細化しようとしているものを注目点と呼ぶ．

ホールに入力するには，まず注目点をホールへ移動し，ポップアップメニューに表示された入力可能なテンプレート一覧から一つを選択すると，注目点は選択されたテンプレートで置換される．プログラムの修正も構文要素単位に挿入，削除，移動，複写，交換，探索などができ，修正時の文法誤りの発生を防止している．

これらの機能により，従来は分厚いマニュアルをみながら行なっていたコーディングが，計算機の誘導により正確な文法を知らないユーザにも簡単に行なえるとともに，編集されたプログラムは常に文法的に正しいことが保証されるため，プログラムの信頼性が向上する．

なお，初心者から熟練者まで，その習熟度に応じた柔軟な操作方式を提供するため，プログラムおよびコマンドは，それぞれメニュー選択方式とテキスト直接入力方式の両方式で入力できるようにしている．

以上，構造エディタ PARSE の特徴について述べたが，このようなツールを各々のプログラミング言語対応に開発するのは大変なので，実際には，構造エディタジェネレータ PARSE-G を開発し，言語仕様，対話仕様および端末種別を入力すればそれに対応する構造エディタが生成されるようにしている．これまでに Pascal，C など4種類の言語への適用実績がある．

4.3.4 図式エディタ

(1) 図式エディタの目的

プログラムの開発に図式を利用することは古くから行なわれている．アセンブリ言語でプログラムを記述するのが普通であった初期の頃は，最初から細かい処理をプログラムに記述することは難しいため，まず処理の流れを図4.8に示すような流れ図（フローチャート）で表わし，処理手順の正しさを確認してからコーディングをした．また，開発後のプログラム修正や拡張などの保守作業にもソースプログラムの処理内容を理解するためにこの流れ図は有用である．

このような流れ図は，プログラムを高級言語で記述するようになっても使用されてきたが，構造化プログラミングの観点では大きな欠点があった．すなわち，従来の流れ図では，制御の流れを示す矢印を自由に使用できるため，goto 文が多用され，プログラムの制御構造が複雑になってしまう．さらに，goto 文をもたない構造化プログラミング言語では自由に作成された流れ図をプログラムとして記述できないという支障が生じる．

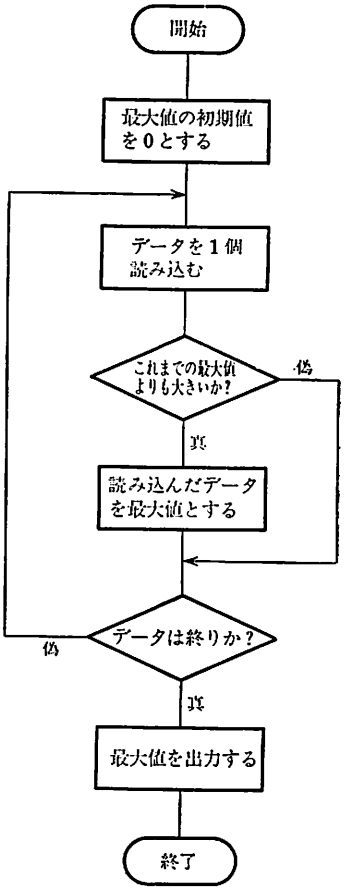


図 4.8　最大値を求めるプログラムの流れ図の例

そこで，これまでの流れ図に代わり，構造化プログラミングに対応する図式表現がいくつか発明された．そのうち，NS チャート，HCP チャート，PAD，SPD などは現在国際標準規格の設定が検討されている．その 1 部を図 4.9，図 4.10 に示す．これは，図 4.8 に対応するプログラムの例である．

ここでいう図式エディタとは，このような図式で表現されたプログラムの編集ツールであり，図式そのものの編集のほかに，図式からソースプログラムを自動生成したり，逆にソースプログラムを図式に逆変換する機能などを有するものが多く，図式を中心としたプログラム開発が可能になっている．ここでは，その 1 例として PAD の図式エディタを紹介する．

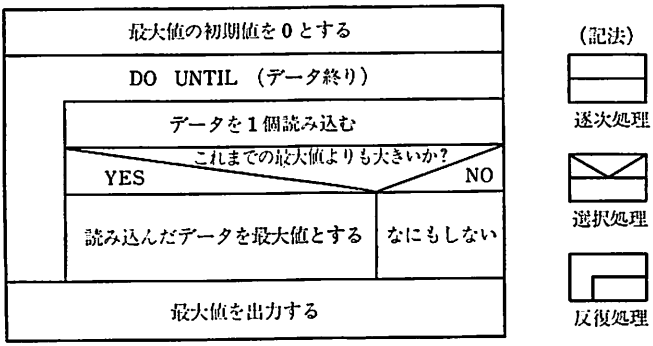


図 4.9 NS チャートの例

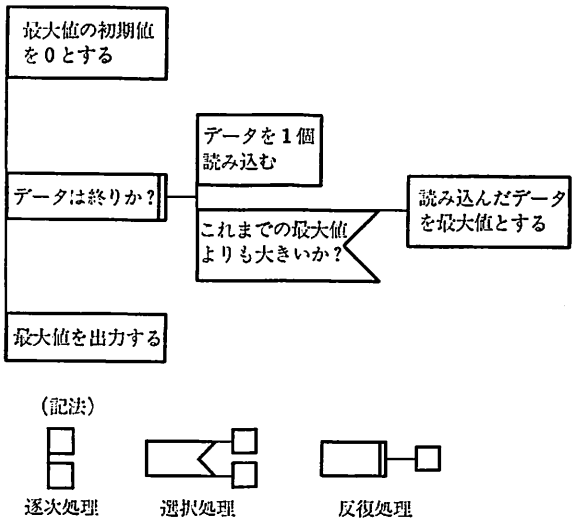


図 4.10 PAD の例

(2) 図式エディタ SDL/PAD

SDL/PAD(Software Design Language/Problem Analysis Diagram)は日立製作所で開発されたビジュアルプログラミング支援の対話型システムであり，図式エディタ，図式デバッガ，図式とソースプログラム間の双方向変換機能などを備えている．

このうちの図式エディタ機能の主な特徴を以下に示す．

(i) 構造化設計技法を基本としており，図4.11に示すように，プログラムの構造を，

① モジュール間の関係を定義するモジュール構造図（木構造）

② モジュールのインタフェースを定義する入出力データ（表形式）

③ モジュール内の処理手順を定義する PAD 図

④ モジュール内で用いる内部データ（表形式）

の4種類の図面で記述する．

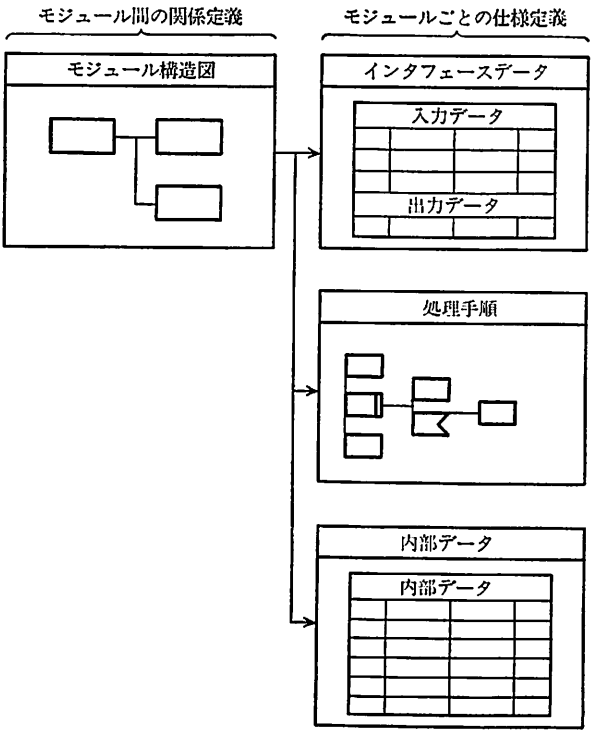


図 4.11　SDL/PAD のプログラム仕様定義機能

(ii) 処理手順の記述は構造化プログラミング技法を基本としており，処理内容を概要から詳細へと段階的に記述する段階的詳細化技法および制御の流れを逐次処理，選択処理，反復処理の3要素で記述する構造化コーディン

グ技法を支援している．たとえば，図 4.10 の PAD 図は図 4.12 のような手順で作成される．

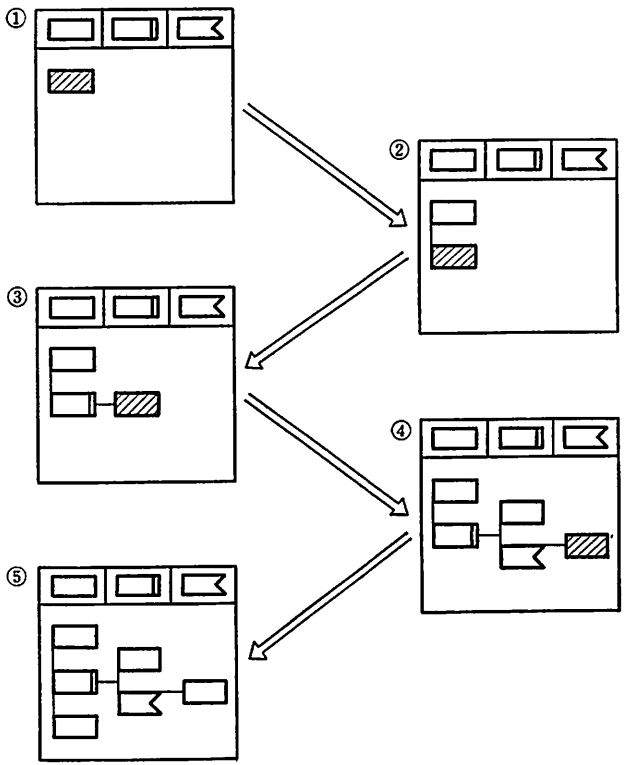


図 4.12 SDL/PAD の誘導方式

① 初期画面に入力を誘導するナビゲータカーソル ▨ が表示されている．

② 画面上段のシンボルメニューから左端の逐次処理をマウスで選び，テキストを入力し，ポップアップメニューから挿入コマンドを選ぶと逐次処理の二つの箱の上段にテキストが挿入され，下段の箱にナビゲータカーソルが移動する．

③ 次に反復処理のシンボルを選び，同様の操作で反復条件を入力する．

④ 次に逐次処理と選択処理のシンボルを選び，判定条件を入力する．

⑤ このような手順を繰り返して，最終的な PAD 図を得る．

このように，シンボル選択，テキスト入力，コマンド選択の操作をナビゲータカーソルの誘導に従って繰り返すことにより，容易に PAD 図を作成できる．

(iii) システムとの対話インタフェースはオブジェクト指向を基本としており，画面でまず「もの（オブジェクト）」を指定し，次にそれへの操作を選択する方式で統一されている．図 4.13 は，図 4.11 の最大値を求めるプログラムの例において，「最大値を出力する」処理を反復処理の中に移動する操作の例で，[1]，[2]，[3]の順に選択すればよい．

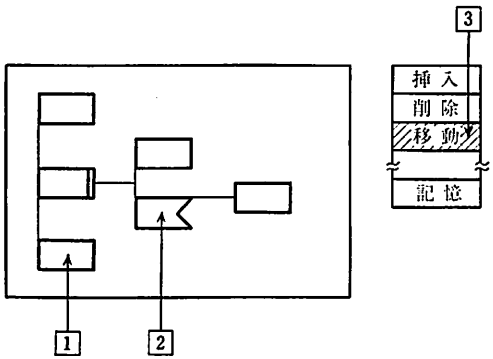


図 4.13 SDL/PADでの操作方式

4.4 言語処理

4.4.1 言語処理プログラムの種類

言語処理プログラムとは，ソースプログラムを実際に計算機が実行可能な機

機械語プログラムに変換するコンパイラやソースプログラムを直接解読して実行するインタプリタなどをいう．この言語処理プログラムは，以下に述べるように，その使用目的や作り方により，いろいろな種類のものが存在する（表4.4）．

表 4.4 言語処理プログラムの種類

視点	プログラム実行方式	使用目的	処理方式	コンパイル単位	稼動マシン
種類	コンパイラ	実行型コンパイラ	1パスコンパイラ	一括コンパイル方式	レジデントコンパイラ
	インタプリタ	チェック型コンパイラ	多パスコンパイラ	独立コンパイル方式	クロスコンパイラ
		最適化コンパイラ	プリプロセッサ	分割コンパイル方式	

(1) プログラム実行方式

プログラムを実行する方式の違いから，言語処理プログラムは，主に，

① コンパイラ

② インタプリタ

の2種類に分けられる．

コンパイラは，ソースプログラムの文字列テキストを読み込み，そのプログラミング言語の文法に合っているか否かを解析し，計算機が実行可能な機械語に変換する．

インタプリタは，読み込んだソースプログラムが文法に合っているか否かを解析した後，そのプログラムの各文の意味を解釈して，直接実行する．コンパイラに比べて一般に実行時間は10倍から100倍ぐらい遅くなるが，小さなメモリ空間で動くとか，ソーステキストの編集機能やデバッグ機能と連動した対話型システムを容易に構築できるなどの利点がある．

(2) 使用目的

コンパイラはその使用目的によって付加的機能が異なり，

① 教育用の実行型コンパイラ

② デバッグ用のチェック型コンパイラ

③ 実稼動用の最適化コンパイラ

などがある．

教育用とは，大学などでプログラミング教育の一環として学生が作成した小さなプログラムを効率よく処理するコンパイラである．

プログラムを実行させる場合，一般のコンパイラを用いるとソースプログラムを機械語プログラムに変換するだけなので，その後，リンケージエディタを用いてモジュール間の結合や実行時ルーチンの付加を行なったり，ローダを用いてプログラムを主メモリ上へ読み込んだりする処理が必要となる．これらの処理時間は大きなプログラムを実行させる場合は全体の時間に占める割合が小さいので問題ないが，実習プログラムのような小さなものを実行させるときは大きなオーバヘッド時間となる．このような無駄な時間をなくすために，教育用の実行型コンパイラでは，多くのプログラムを一度に入力し，各々のコンパイル処理から実行処理までを繰り返し実行できるようになっている．著者らが開発した実行型 FORTRAN の例では，一般のコンパイラを用いて実行処理まで行なった場合に比べて20倍以上の高速化が実現している．

実行型コンパイラは，そのほかにも，エラーメッセージが詳しく出力されたり，デバッグ機能が豊富であるなどの特徴があり，プログラム実習向きになっている．

第2のチェック型コンパイラは，主にプログラム開発時に用いるもので，プログラムのテスト機能やデバッグ機能が豊富である．プログラムの開発時は，頻繁にソースプログラムが修正され，再コンパイルが行なわれることから，チェック型コンパイラでは，実行効率のよい機械語プログラムを生成することよりも，テスト，デバッグ機能を充実させて開発効率をよくすることが目的である．具体的な機能は 4.4.3 で述べる．

最後の最適化コンパイラは，チェック型コンパイラとは逆に，できるだけ実行効率のよい機械語プログラムを出すことを目的にしている．これは，プログラムの実稼動時には，1回のコンパイル処理により出力された機械語プログラムが何回も繰り返して実行されるため，コンパイル処理に

時間をかけても，実行速度の速い方がよいという考えに基づいている．詳細は 4.4.4 で述べる．

(3) 処理方式

コンパイラは内部の処理方式の違いによって，

① 1パス方式

② 多パス方式

③ プリプロセッサ方式

などの種類がある．

1パス方式とは，コンパイラの入力となるソースプログラムを上から順に解析していくと同時に機械語プログラムを生成してしまうものである．プログラムを上から順に処理していく操作を1回しか行なわないことから「1パス」と呼ばれる．この方式では，プログラム内の実行文に対応する機械語を生成するために必要な情報は，プログラムの先頭からその実行文の直前までの間の処理中に得られていなければならない．そのため，プログラム内で使用される変数や手続きはそれらが使用される前に宣言をしておかなければならないという言語仕様上の制約が必要である．

このようなプログラミング言語の例として Pascal がある．Pascal の設計は，よいプログラミング言語の条件は仕様が簡潔でコンパイラが簡単に作れることであるという考えに基づいている．そこで，1パス方式でコンパイルできる言語仕様にするため，プログラム内で用いるラベル，定数名，型名，変数名，手続き名，関数名などはすべて使用する前に宣言させたり，その宣言の順序を規程している．

多パス方式は最もよく用いられている方式であり，構文解析，意味解析，コード生成などの主要な処理を別々のパスで行なうものである．たとえば，最初のパスではソースプログラムを上から順に読み込み，構文解析をしながら木構造形式などの中間語に変換する．2番目のパスではその中間語プログラムを上から順に読み込み，意味解析をしながら四つ組形式などの中間語に変換する．最後のパスではその中間語プログラムを上から順

に読み込み，機械語プログラムに変換する．このような多パス方式は，複雑な言語仕様のプログラムでもコンパイルできること，エラーチェックや最適化処理を詳細に行なえること，処理が分割されるため主メモリが少なくてよいこと，などの利点がある．詳細は 4.4.2 で述べる．

プリプロセッサ方式は，通常のコンパイルのようにソースプログラムを機械語プログラムに変換する代わりに，他のプログラミング言語のソースプログラムに変換するものである．この方式では機械語プログラムへの変換を他のコンパイラに任せるため，処理は簡単になる．

たとえば，FORTRAN の言語仕様に新たに構造化プログラミングのための機能を付加して新しい言語を作った場合，その拡張部分を解析して通常の FORTRAN プログラムに変換するプリプロセッサを作るようなときに用いられる．

(4) コンパイル単位

コンパイラへの入力となるソースプログラムの最小単位およびその相互関係に関連して，

① 一括コンパイル方式

② 独立コンパイル方式

③ 分割コンパイル方式

などの種類がある．

一般に高級プログラミング言語には，手続き，サブルーチン，関数などと呼ばれるモジュール化機能があり，それらを別々にコンパイルしておき，実行するときに互いに結合するような機能をもつものが多い．ある程度以上の大きさのプログラムの開発時には必須の機能である．

一括コンパイル方式とは，このような機能をもたず，常にソースプログラム全体をコンパイラに入力するものである．この方式では，手続き呼び出しの実引数とその手続き定義側の仮引数の数およびデータ型の不一致あるいはグローバル変数の参照誤りなどをコンパイル時に検出できるため，プログラムの信頼性向上の利点がある．

その反面，頻繁にソースプログラムの修正，コンパイル，実行の一連の処理が繰り返されるプログラム開発時に，ソースプログラムを1文字でも修正すれば全体を再コンパイルする必要があるため，プログラムが少し大きくなると開発効率が悪くなり，不便である．さらに，コンパイラが内部で使用するテーブルやスタックなどのデータエリアのメモリ容量の限界を越えるような大きさのソースプログラムはコンパイルができないという致命的な欠点をもつ．

たとえば，Pascal はもともと教育用に用いるプログラミング言語として開発されたため，あまり大きなプログラムは作らないという前提で一括コンパイル方式に適した言語仕様になっている．しかし，実用的プログラムの開発に用いられるようになると，この方式では無理があるため，手続き単位でもコンパイルできるように言語仕様を拡張したものが多い．

独立コンパイル方式は，FORTRAN，PL/I などで一般的に用いられているもので，サブルーチンや手続きなどのモジュール単位のコンパイルが可能な方式である．この方式は，一括コンパイル方式で述べた欠点が解消される反面，モジュール間のインタフェースの不一致によるプログラム不良がコンパイル時には検出されない．一方，この種の誤りをプログラム実行時に検出するような仕掛けを設けると実行速度が低下するため，一般には実行時にシステムが自動的に検出することはしない．そのため，デバッグに手間のかかるモジュール間インタフェース不良が多く発生するという欠点がある．

分割コンパイル方式は，一括コンパイル方式と独立コンパイル方式の双方の利点をもつように，モジュール間インタフェースの一致のチェックを行ないながらモジュール単位のコンパイルを可能とする方式である．これは，米国国防総省が開発したプログラミング言語 Ada で採用されている方式で，信頼性の高い大規模プログラムの開発に適している．しかしながら，この方式では，一つのモジュールをコンパイルするときにそのモジュールが参照する他のモジュールの情報が必要になるため，

① 先にコンパイルしたモジュールの情報を他のモジュールのコンパイル時に参照できるようにライブラリに保存，管理する．

② そのときに情報を提供するモジュールが参照するモジュールより先にコンパイルされるようにコンパイル順を管理する．

③ あるモジュールのソーステキストを修正したときには直接的および間接的にそのモジュールの情報を参照するモジュールはすべて再コンパイルする．

などの処理が必要となり，コンパイラおよびその使用方法が複雑になる．

(5) 稼動マシン

コンパイラが稼動する計算機とその対象となるユーザプログラムを実行する計算機の機種が同じか異なるかによって，それぞれ，

① レジデントコンパイラ

② クロスコンパイラ

と呼ばれる．通常は同じ機種の場合が多いが，たとえば，ユーザプログラムを実行するターゲットマシンがマイクロコンピュータの場合で，

① ターゲットマシンの能力が小さくて，コンパイラが動かない．

② ターゲットマシンのプログラム開発環境が貧弱で，開発効率が悪い．

③ ターゲットマシンの台数が少なくて，多人数での開発に支障がでる．

などの問題があるときには，そのプログラムの開発は大型計算機などをホストマシンとして用いるのがよい．このとき，ホストマシン上で稼動し，ターゲットマシン用の機械語プログラムを生成するクロスコンパイラが必要になる．

4.4.2 コンパイラの構造

コンパイラの構造は先に述べたようなコンパイラの種類や稼動環境によって千差万別であるが，概念的には図 4.14 のような構造をしたものが多い．コンパイラに関連する項目を以下で簡単に説明する．

(1) 全体構成

コンパイラの処理は大まかには文法規則に従ってソースプログラムを解析する処理と機械語プログラムを生成する処理からなる．実際にはかなり細かく処理を分割し，各々をフェーズと呼ぶが，複雑なものではフェーズの数が数十に及ぶものもある．その中でプログラムを先頭から順に処理していくパスの数はフェーズより少ない．図 4.14 は概念的な例なのでパスとフェーズが同じで四つである．

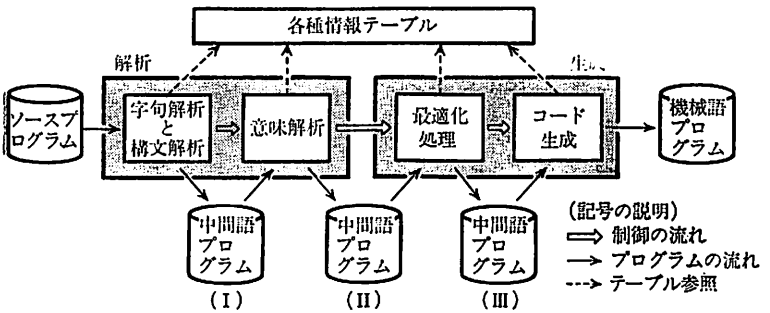


図 4.14 コンパイラの概念的構造

(2) 中間語プログラム

多パスコンパイラにおいて，ソースプログラムを機械語プログラムに変換するまでの間に各パスが出力する中間段階のプログラムを中間語プログラムという．中間語の形式には，たとえば，a × b + c という算術式を ab × c + のように演算子 × と + をそれらの演算オペランド a，b，c のあとにもってくる逆ポーランド記法，演算子を節，オペランドを枝で表現する木構造，(演算子，第 1 オペランド，第 2 オペランド) で表現する三つ組形式，(演算子，第 1 オペランド，第 2 オペランド，結果) で表現する四つ組形式などがある．

(3) 字句解析

字句解析では，ソースプログラムの文字列を解析し，プログラミング言語の基本要素である定数，変数，キーワード，区切り記号などに切り分ける処理を行なう．たとえば，

if data=0 **then** l : =3.14 * r

というテキストは,

if／data／=／0／**then**／l／:=／3.14／*／r

という 10 個の字句に分けられる.

(4) 構文解析

構文解析は字句解析によって得られた字句の並びがプログラミング言語の構文規則に合っているか否かのチェックを行ないながら，構文構造を表現した中間語に変換する．先の if 文の例では，字句の並びが,

if <式> **then** <文>

の構文規則に対応しており，<式>および<文>はそれぞれ,

<変数> = <定数>

<変数> := <式>

の形をしていることを解析する.

(5) 意味解析

意味解析では，プログラミング言語の文法規則のうち，構文規則で表現できないような規則について解析する．これらの規則は通常マニュアルの中では日本語で表現されている部分であり，データ型の一致や goto 文の分岐先の制約などがチェックされる．先の if 文の例では変数 data のデータ型は整数型であり，l と r は実数型であることをチェックする.

(6) 最適化処理

最適化処理では，少しでも実行速度の速い機械語プログラムを生成するため，中間語プログラムの中から最適化項目の条件に合う部分を探し出し，変更する．たとえば，先の if 文の例で，変数 r の値がその if 文の実行の時点で 2 となることが分かれば，l : =3.14 * r という代入文は l : = 6.28 という代入文に変換され，実行時の乗算をなくすことができる．詳細は 4.4.4 の最適化コンパイラのところで述べる.

(7) コード生成

最後のコード生成処理において，中間語プログラムが機械語プログラム

に変換される．そのために，まず各々の変数のストレージを変数用のメモリエリアに割当てたり，レジスタの使い方を決めたりする処理を行ない機械語命令を生成するために必要な情報を決定する．

(8) 各種情報テーブル

上記の(3)から(7)までの各処理の間で得られた種々の情報はテーブルに保存し，必要に応じて他の処理時に参照可能にする．たとえば，名前テーブルにはプログラム内で使用されるすべての名前が一覧表になっており．data という名前は整数型変数で，手続き P の中で宣言されており，そのストレージは手続き P 用のメモリエリアの先頭から n バイト目に確保されているなどの情報が保存される．

(9) 文書化

図 4.14 には明示していないが，コンパイラは，プログラム開発時に役に立つと思われる種々の情報を出力する．基本的なもの以外はオプション機能になっており，ユーザがその出力の要否を指定できる．出力情報の例としては次のようなものがある．

① ソースプログラムリスト

② 名前リスト（種別，データ型などを含む）

③ オブジェクトリスト（機械語プログラムをアセンブリ言語レベルで表現したもの）

④ クロスリファレンスリスト（各名前の宣言場所と参照場所を対応づけたもの）

⑤ メモリマップ（各手続きや変数ストレージの相対的配置関係）

⑥ エラーメッセージリスト

4.4.3 デバッグ用コンパイラ

デバッグ用のコンパイラは，テスト，デバッグ機能を充実させ，プログラムの開発段階での作業効率をよくする目的のものである．その実現方式は，コンパイラのオプション機能で指定するもの，ソースプログラムの中にテストデバ

ッグ用の文を挿入するもの，対話型実行時に端末からコマンド入力するものなどがある．

誤りが自動的に検出されるものとしては次のようなものがある．

① 添字つき配列変数の添字の値がその変数宣言時の範囲を越えたことによる変数領域外参照

② 値が代入されていない変数の参照

③ 手続き呼び出しにおける，実引数と仮引数のデータ型の不一致

一方，デバッグ用の機能としては次のようなものがある．

① 変数や式の値の表示

② 変数の値の変更

③ 変数の値が変更されたときにその情報の表示（データトレース）

④ 制御の流れを示すために，分岐が生じたときにその情報の表示（制御トレース）

⑤ 対話型実行時の割込みキーによる実行中断と上記①～④機能のコマンド入力

4.4.4 最適化コンパイラ

最適化コンパイラは，できるだけ実行速度の速い機械語プログラムを生成するのが目的であり，コンパイル処理の比較的前段階で実施されるものから機械語命令生成直前で実施されるものまで種々の最適化項目がある．

まず，局所的な最適化項目としては次のようなものがある．

① 冗長な処理の削除：たとえば，次のような二つの代入文が連続していた場合，2番目の代入文を削除する．

```
work := a
a:= work      ⇨   work : = a ;
```

② 定数の演算：次の例のような定数同士の演算はコンパイル時に行なう．

```
l : =2 * 3. 14 * r   ⇨   l : =6. 28 * r
```

③ 数学的単純化：次の二つの例のような数学的に冗長な演算を削除する．

```
a := b + 0   ⇨   a := b
a := b * 1   ⇨   a := b
```

④ 演算種別の変更：次の二つの例のように演算の種類をできるだけ簡単なものに変更する．

```
2 * a    ⇨   a + a
 a²      ⇨   a * a
```

⑤ 多重分岐の単純化：次の例のように無条件分岐が重なるときは最後の分岐先へ直接分岐させる．

```
     goto  l              goto  m
       ≀        ⇨           ≀
l : goto  m          l : goto  m
```

⑥ 実行不可能な処理の削除：次の例のような実行されない処理を削除する．

```
const  flag  = 0
   ≀
if flag = 1  then goto  l
```

次に何度も繰り返して実行されるループ内の最適化項目として次のようなものがある．

⑦ ループ内不変式の移動：ループの実行回数に無関係に常にループ内で同じ演算結果になる式は，ループに入る直前で1回だけ演算を行ない，ループ内ではその結果を用いるようにする．

⑧ ループ制御変数の除去：ループの実行回数を制御するためだけに導入された変数は，それを用いた配列変数の添字計算を工夫することにより除去する．

⑨ 配列変数の添字計算の簡単化：ループ制御変数を含む添字つき配列変数のメモリ上のアドレス計算は，一般的方法では乗算を含むがループ実行回数に応じて一定値だけ増加することに注目して加算に変換する．

さらにデータフロー解析を行ない，各変数の値がどこで代入され，それがどこで参照されるかを把握することにより，次のような最適化が可能である．

⑩ 定数の伝播：定数が代入された変数を参照している部分を定数で置換するとともに，上述した最適化項目が適用可能になったならばそれを実施する．

⑪ 共通式の削除：同じ演算式が2ヶ所で記述され，それらの式が時系列的に前後して実行されるようになっており，かつその間でその式に用いられている変数の値が不変ならば，あとの式は前の式の値を用いるように変更する．

以上，ソースプログラムのレベルで説明の容易な最適化項目を中心に紹介したが，このほかにも複数のレジスタをもつターゲットマシンの機械語プログラムを生成する場合は，計算の順序を工夫してレジスタを有効に利用する方式などが実施されている．

4.5 静的プログラム解析と文書化

プログラムの文法エラーをコンパイラを用いて検出，除去した後，実行テストに入る前に，もう一度ソースプログラムの見直しを行なうのが望ましい．これは，実行テストによってプログラムの誤りを検出，除去する方法は作業効率が悪いだけでなく，テストデータの不備による誤りの見逃しが発生しやすいことが経験的によく知られているからである．また，プログラムの誤りの検出，除去のための費用は，その検出が遅れるほど，指数関数的に増大するという事実もある．そこで，実行テストに入る前にプログラムの見直しを行なうことは非常に重要であり，この作業はコードレビュー，机上レビュー，机上テストなどと呼ばれ，大規模ソフトウェアの開発時には必ず実施されている．

このときに役に立つのが静的アナライザであり，ソースプログラムを細かく分析し，その構造をチェックしたり，誤りの可能性の大きな不自然な処理を検出したり，コードレビューの助けになる種々の情報を提供する．その主なものとして次のようなものがある．

① コードオーディタ：プログラムの構造を分かりやすくしたり，誤りを未

然に防止するための種々のコーディング規則を守っているか否かをチェックする．

② プログラム複雑度の計算：プログラムの構造の複雑さを測定することにより，きれいな構造のプログラム作りを奨励し，不良作り込みのポテンシャルを減少させる．複雑度の尺度としては，分岐数や構造化コーディング違反件数などのように制御構造をベースにしたものや変数の定義参照関係や制御変数の使用などを含めて分析するもの，モジュール間の関係やモジュールの外部インタフェースに着目するものなど，種々の方式が提案されている．

③ データフロー解析：各々の変数への値の代入がどこで行なわれ，その値がどこで参照されているかという分析を行なうことにより，代入された値がどこでも参照されていないとか，値が代入されていない変数が参照されている，などの不自然な処理を検出する．

④ 手続き呼び出しグラフ：ある手続きとその本体の中で呼び出している手続きとの間の呼び出し関係をプログラム全体の中のすべての手続きについて分析し，一つのグラフにまとめる．このとき，できれば実引数リストと仮引数リストの対応関係も表示する．

⑤ 制御フローグラフ：モジュール単位に制御の流れを分析し，4.3.4で述べたような図式で表示する．流れ図を出力するオートフローやPADを出力するAuto-PADなどがある．

⑥ プリティプリンタ：コンパイラが出力するソースプログラムリストは行番号，文番号，ブロック構造やそのネストの深さレベルなどの情報を付加するものが多いが，テキスト自身はユーザが入力したときの形で表示する．それに対し，プリティプリンタは，プログラムの制御構造をみやすくするためにブロックのネストが深くなるにつれて対応するテキストの先頭カラムを右へ桁下げして表示する．この操作は自動インデンテーションと呼ばれる．

5 テスト

5.1 概　　要

5.1.1 プログラム検証の方法

プログラムの検証は，

「プログラムが仕様どおりに作られている」

ことを確かめるのが目的である．その方法としては，図5.1に示すように，

① 仕様書からプログラムを自動生成

② 仕様書とプログラムの等価性証明

③ テストデータによる動的テスト

などが考えられる．

最初の自動プログラミング方式はプログラム開発方式の理想形態であり，常に正しいプログラムが生成されるので，生成されたプログラムを改めて検証する必要はない．この方式を実現するためには，仕様を形式的に記述するための仕様記述言語とその言語で記述された仕様から計算機で実行可能なプログラムへの変換規則あるいは仕様を満たす既開発のプログラム部品の検索技術などが必要である．本方式はまだ研究段階にあり，実用的規模のプログラムの自動生成実現に至っていない．

次に仕様書とプログラムの等価性を自動証明する方式はプログラム検証方式