16. システム記述言語†

武　市　正　人††

1. はじめに

システム記述言語は，ハードウェアとともに計算機システムを構成するソフトウェアの中核となるプログラム——システムプログラム——の記述・作成に用いられるものである．しかしながら，一般に，システムプログラムとそうでないものとの境界を定めることが困難なこともあり，また，それらの区別をする必要のない場合も多い．本稿では，とくに，システム記述言語の定義を明確にすることはしないで，むしろ，実用的なシステムの記述に用いられている言語を対象とすることとする．

計算機システムの規模が小さい頃には多くのシステムプログラムはアセンブリ語で書かれていたが，システムが大きくなり複雑になるにつれて論理的に複雑になったシステムプログラムを記述するために高水準言語とその処理系が必要とされるようになった．高水準言語による記述には，プログラムの生産性を向上させ，保守を容易にすることはもちろん，何よりも信頼性を高めることが期待される．一方で，こうした要求に応える言語処理系の要件の一つとして，効率の良い目的プログラムを生成することがあげられる．それは，対象とするプログラムがシステム全体の性能に影響を与えるものであることによる要請である．言語の水準の高さと，目的プログラムの質の良さとは相容れない要求であり，言語とその処理系の設計にあたっては，それらのあいだに妥協点を見出すことや，新たな処理方式（たとえば最適化の手法）の開発が繰り返されているといえよう．

システム記述言語のなかには対象とする計算機を定めた機械向き（machine-oriented）の言語と呼ばれるPL/360, Bliss など[1]があり，その一方では汎用言語である Fortran, Algol 60, PL/I, Pascal など[1]がシステム記述に用いられた例も多い．機械向きの言語は汎用性を犠牲にして効率の良い目的プログラムを期待するものであり，汎用言語は機械に依存しないプログラムの記述を望むことにあるといえよう．

こうしたなかで，1970年頃から現在まで，多くの計算機のメインフレーマで使用されているシステム記述言語は，主として PL/I 系のものであろう．これらは PL/I の部分言語にシステムを記述するために必要とされる機能や，オペレーティングシステムに依存する部分を追加して作られている[2]．

PL/I という“大きな”言語をもとにして作成されたこれらのシステム記述言語とは別に，近年，Pascal や BCPL といった“小さい”言語を基礎として機械に依存しない言語が設計されている．上にあげた機械向き言語，汎用言語，PL/I 系のシステム記述言語についてはほかに譲り，本稿では最後にあげたような形で設計された言語についてその特徴を見ることにする．

2. 最近の動向

1970年頃の構造的プログラミング，ことにデータの抽象化の概念や制御構造は，Pascal をはじめ，多くの言語の設計に影響を与えてきた．Pascal 以後にも，プログラムで扱われる対象の存在期間がプログラムテキストの上の静的な構造と制御の流れとの関係から定まるという Algol 流の強い制約は抽象的なデータ型を実現するには適切でないことが指摘されたり，いくつかの制御の流れが並行しているような多重プロセスのプログラムを記述する機能などを取り入れる動きも出てきた．これらをはじめとするプログラミング言語の機能に関する新しい概念が新しい言語の設計の動機になっている．こうした流れのなかで，Pascal をもとにして Concurrent Pascal, Modula, Euclid, Mesa, Ada, などが設計されてきた．これらの言語の概要は学会誌 1981 年2月号の小特集[3]を参考にされたい．また，Ada については本特集でも解説されている[4]．本稿では Pascal の設計者である N. Wirth 自身が Pascal

† System Description Languages by Masato TAKEICHI (Dept. of Computer Science, The University of Electro-Communications).

†† 電気通信大学計算機科学科

の10年後に設計した言語 Modula-2[4] の概要を 2.1 で述べることとする.

一方では,こうした新しい概念の影響を受けつつも,どちらかといえば保守的な,"アセンブリ語に代わる"言語にも新たなものが現われてきた.この種の言語の代表的なものに言語C[5]がある.Cは,BCPL を基礎として Unix オペレーティングシステムとともに成長してきた言語ということができよう.本稿では実用的なシステムの記述に成功している代表的な言語Cの特徴を 2.2 で述べる.

2.1 Modula-2

Modula-2[4] は,Pascal と Modula をもとに 1980 年に定められたものである.細部については異なるものの,おおよそ,Pascal にモジュール (module) 構造を取り入れ,擬似的な並行プロセスを実現するためのコルーチン (coroutine) の機能を追加したものであるといえる.言語は特定の計算機やオペレーティングシステムには依存しないものとして定義されている.機械に依存する基本的なデータの単位や操作については,一つのモジュールとして与えられるものとし,機械に依存しない部分とは分離するという方法がとられている.基本的な制御構造やデータ構造については,いくらか変更があるものの,Pascal と同様のものと考えてよい.ここでは,Modula-2 に特徴的なモジュールの概念と,それがシステム記述の上で果たす役割について簡単に述べることとする.

Modula-2 のプログラムは,いくつかの (論理的に分割された) モジュールから構成される.処理系 (コンパイラ) によって翻訳される単位はモジュールである.(論理的な)一つのモジュールには,プログラムテキストでは,一つの定義モジュール (defining module) と,一つのプログラムモジュール (program module, implementation module) とが対になって対応する.定義モジュールは,そのモジュールから別のモジュールに持ち出される (export される),すなわち,別のモジュールが持ち込む (import する) 変数,型,手続きなどの対象を定義するもので,ほかのモジュールとの連絡のために用いられる.プログラムモジュールは処理の手順を記述するもので,対応する定義モジュールに含まれる対象の具体的な記述を与える.また,プログラムモジュールには,モジュール内部でのみ操作する対象をも含みうる.このように,2種類のモジュールの記述を用いることによって,いわゆる情報隠蔽 (information hiding) の機能が実現されるとともに,プログラムの開発における分割処理が可能となる.処理系 (コンパイラ) は,ほかのモジュールを参照しているモジュールの翻訳の際には,必要な定義モジュールを利用するという方法を用いている.モジュールを(ほかのモジュールとは独立に)個別に翻訳する独立翻訳 (independent compilation) の方式に欠けているモジュール間の対象参照の適合性検査が Modula-2 の分割翻訳 (separate compilation) では完全に行われ,プログラムの静的な検査によって信頼性を高める効果が期待される.

特定の計算機やオペレーティングシステムに関する低水準の対象 (たとえば語 (word),番地 (address) などのデータの型や手続きなど) の記述は,特別に用意されるモジュール SYSTEM に置かれ,その上で動作するプログラムは,必要な対象を SYSTEM モジュールから持ち込んで記述される.こうして,Modula-2 では,プログラムは,より基本的なモジュールの上にモジュールを構築するという上昇型 (bottom up) の構成をとることになる.

図-1 に,簡単なモジュールの記述例を示す.(a)が定義モジュール,(b)がプログラムモジュールである.このモジュールは,手続き *allocate* が外部から参照され,呼び出されると,大きさ n の記憶領域を与えるものである.プログラムモジュール(b)の最後に置かれている文は,モジュールの初期設定である.

擬並行プロセスを実現するコルーチンの機構もまた SYSTEM モジュールのなかに用意される手続きによる.また,割込み処理のモジュールの記述にも SYSTEM モジュールにある手続きが使われる.このよう

```
DEFINITION MODULE storage;
  FROM SYSTEM IMPORT ADDRESS;
  EXPORT allocate;

  PROCEDURE allocate(VAR a: ADDRESS; n: INTEGER);
END storage
```

(a)

```
MODULE storage;
  FROM SYSTEM IMPORT ADDRESS;
  EXPORT allocate;

  VAR lastused: ADDRESS;
  PROCEDURE allocate(VAR a: ADDRESS; n: INTEGER);
    BEGIN a:= lastused;  lastused:= lastused+n
    END allocate;

  BEGIN lastused:= 0
  END storage
```

(b)

図-1 Modula-2 のモジュールの例

に，言語そのものに機械依存性は含まれないが，特別のモジュールには，(通常の言語処理系の) ライブラリに相当するプログラムが用意されているものとしている．

Modula-2の処理系は，Zürich工科大学で，PDP-11のRT-11オペレーティングシステムの上に作られているほか，Pascalによるクロスコンパイラなども作成されている．Modula-2システムは，既存のオペレーティングシステムの一部(たとえばファイルシステム)を借用する形で動くが，このオペレーティングシステムとは独立したシステムを記述することも可能である．

2.2 C

言語C[5]は，“とくに高水準というわけではなく，また大きくはない”言語である．BCPLをもとに，言語Bを経て1972年頃に作られた．米国Bell研究所で作成され，プログラムの研究・開発に数多く用いられてきているUnixシステム[6]自身や膨大なユーティリティプログラムのほとんどすべてのものが，(1,000行程度のアセンブリ語の部分を除いて) Cで記述されている．また，Cで記述されたプログラムの移植性の高いことも報告されている．

言語Cには，BCPLから受け継いだ特徴として，基本的な制御構造を表現する機構 (while, if など) や，指標 (pointer) のデータと記憶場所の番地を表わす対象参照値 (lvalue) の概念に基づく記憶場所の取り扱い方などがあるほか，BCPLやBには存在しなかったデータの型が導入され，多くの計算機で扱うことのできる基本的な対象 (バイト，語，倍長語，……) を表現することができるように拡張されている．また，基本的な型のデータに対しては，豊富な演算子が用意されていて，ビット操作などの細かい操作も記述することができる．ここでは，言語Cの基礎の一つである指標と対象参照値の概念について述べ，Cの設計のなかに見られる主張について触れることにする．

Cでは，通常の式の値は，算術演算や論理演算の結果として得られるものであるが，ある種の式は，値のほかにも対象参照値を持つ．BCPLで，値を右辺値 (rvalue, right side value)，対象参照値を左辺値 (lvalue, left side value) と呼んで，それぞれ代入演算の右辺と左辺の文脈で用いられる値と，変数に割り当てられる記憶場所を示すものとを区別していたものを継承している考え方である．変数に割り当てられた記憶場所を指定することができるような式は，式の値とともに対象参照値をも持つものとされる．たとえば，名前xは，(レジスタに割り当てられていないときには) xに割り当てられた記憶場所の番地を示す対象参照値と，そこに保持されている値とを持つ式である．演算子の多くのものは値に対する演算を表わすものであるが，なかには，被演算子として対象参照値をとるものや，結果として対象参照値を与えるものがある．式

$$\& x$$

は，xの対象参照値に対応する指標型の値を与えるものであり，式

$$*p$$

は，指標型の値pに対応する対象参照値と，その対象に保持される値とを表わすものである．対象参照値と指標型の値とは，実際には属性が変わるだけであり，同一のビット構成で表わされるものと考えて差し支えない．これらの区別は，言語を定義する上で必要なものである．演算子 & は，“……の番地”を与える演算であり，*は“間接演算”の演算子である．関数の引数はすべて値によって結合されることになっているので，値を変更するような引数は，演算子 & を用いて，指標型の値を通じて処理することになる．

指標型はまた，配列を取り扱う際にも重要な役割を果たしている．配列に対して与えられる名前は，式のなかでは，その配列に割り当てられた第0番目の要素を指す指標の値を表わす．指標型の値と数値との加算も用意されていて，配列要素

$$a[i]$$

は，式

$$*(a+i)$$

と等価であると定義されている．指標型の値と数値との加算の際には，指標の指し示す対象の大きさに応じて，数値が変換されて加えられる．

図-2に示すプログラムは，文字列の複写を行うものである．関数 *strcpy* の引数 s, t は文字データの配列，すなわち文字列，への指標である．文字列は，“空”文字 '\0' で終了するという約束である．変数や引数の型は

型規定子　宣言子

の形で表わされ，型規定子 (type specifier) は，宣言子 (declarator) の形のものが式のなかに現われたときに，それがとる値の型を表わすものである．すなわち，型は，Pascalや多くの言語のように，陽に示されるものではない．たとえば

char $*s, *t$;

```
strcpy(s, t)
char    s[], t[];
{
        int     i;

        s[0]= t[0];
        i= 1;
        while(t[i] != '\0'){
                s[i]= t[i];
                i= i+1;
        }
}
                (a)


strcpy(s, t)
char    s[], t[];
{
        int     i;

        i= 0;
        while((s[i] = t[i]) != '\0')
                i++;
}
                (b)


strcpy(s, t)
char    s[], t[];
{
        register        int     i;

        i= 0;
        while((s[i]=t[i])!='\0')
                i++;
}
                (c)


strcpy(s, t)
char    *s, *t;
{
        while((*s = *t) != '\0'){
                s++;
                t++;
        }
}
                (d)


strcpy(s, t)
char    *s, *t;
{
        while((*s++ = *t++) != '\0')
                ;
}
                (e)


strcpy(s, t)
char    *s, *t;
{
        while(*s++ = *t++)
                ;
}
                (f)
```

図-2 Cの関数の例

は，s と t とが，$*s, *t$ の形で用いられると，*char* 型（文字型）であることを宣言している．つまり，s と t とは文字型のデータを指し示す指標型ということになる．

代入演算子 = は，式の値として代入される値を与えるとともに，副作用として代入の効果を生じる．また，i++ のように書かれる式も副作用を伴うもので，i の値を式の値としたうえで，i には，i の値を1殖やしたものを代入する操作を表わす．!= は等しくない関係を示す．また，$*s$++$=*t$++ は $(*(s++))=(*(t++))$ と同じである．

while 文による繰返しの反復条件に用いられる論理値としては，0が偽，0でない値が真として扱われる．

図-2のプログラムのうち(a)は，Pascal 風の記述，以下(b)，…，(f)の順に，“Cらしい”記述になっているといえよう．

この例で見られるように，演算子が強力であることと，表現形式が豊かなことから，Cのプログラムは，ややもすると読みづらくなることもあるが，プログラマが，多様な表現のなかから適切な記述を選択し，ひいては目的プログラムが効率の良いものになるように工夫することもできる．また，計算機に本来備わっている演算に対しては多くの演算子が用意されているがプログラマが関数として定義することのできる演算は言語では定義せず，プログラマに委ねることによって，処理の記述の“量”と処理の“内容”とが対比できるということができよう．つまり，計算機にとって簡単な処理は簡潔に記述されるが，複雑な処理は複雑な記述になるわけである．

言語 C にも，最近，Pascal にあるような列挙型 (enumeration type) が導入されてきているが，多重プロセスの制御を記述する機能を言語に取り入れることは頑固に拒んでいる．これらの機能を言語の基本的な要素として定めることは，特定のオペレーティングシステムを仮定することになって好ましくないという主張である．

言語Cの処理系（コンパイラ）は，Bell研究所で，PDP-11, Honeywell 6000, IBM 370, Interdata 8/32, VAX 11 に作成され[5),6)]，また，マイクロプロセッサ 8080 や，NOVA 3[7)] でも作られている．

3. 今後の見通し

本稿では，代表的な二つの言語の特徴的な概念を見た．これらの言語からは，機能の充実を目差して，“大きい”言語とするのではなく，基本的な概念を整理した簡潔な言語を定めるという主張がうかがわれる．どちらの処理系も，当初はミニコンピュータに作成され

ている程度の規模である．今後のシステム記述言語の一つの方向であるといえよう．同時に，“何でもできる”言語について検討を加える必要もあろう．

また，処理系に関しては，単にコンパイラだけではなく，プログラムの設計・作成を支援する環境を整備することが重要な課題であるといえよう．

参 考 文 献

紙面の関係で，本稿で取りあげた言語に関するものを中心に挙げる．引用した多くの言語については 1)，3) の参考文献を参照されたい．

1) 井田昌之，中田育男：基本ソフトウェアの記述ツール，情報処理，Vol. 20, No. 6, pp. 519-526 (1979).
2) 筑後道夫：システム記述用言語，情報処理，Vol. 16, No. 10, pp. 864-870 (1975).
3) 和田英一，武市正人，疋田輝雄編：プログラミング言語—Pascal と Ada 小特集，情報処理，Vol. 22, No. 2, pp. 86-148 (1981).
4) Wirth, N.: Modula-2 Berichte des Instituts für Informatik Nr. 36, Eidgenössische Technische Hochschule Zürich (1980).
5) Kernighan, B. W. and Richie, D. M.: The C Programming Language, Prentice-Hall (1978).
6) The Bell System Technical Journal Vol. 57, No. 6, Unix 特集号 (1978).
7) 黒田壽祐，辻野喜宏，萩原兼一，荒木俊郎，都倉信樹：システム記述用言語Cのポータブルコンパイラの作成，情報処理学会論文誌，Vol. 21, No. 6, pp. 461-468 (1980).

（昭和 56 年 3 月 2 日受付）