Vol. 36 No. 4　　情　報　処　理　　Apr. 1995

解　説

プログラミング言語最新情報-II

5. Ada-9X
—大規模ソフトウェア向きの手続き型言語†—

筧　捷彦††　石畑　清†††　西田晴彦††††

1. はじめに

Adaは，アメリカ国防総省（DoD）が，省内で使用する各種組込み型ソフトウェアの開発を主目的として策定した言語である．この目標とはやや矛盾するが，実際に完成したAda言語は，狭い応用範囲に限定されるものでなく，1970年代のソフトウェア工学の成果を集大成した汎用言語と見なすことができる．

Adaの規格には，1983年のANSI規格[1]，1987年のISO規格[2]，1991年の日本工業規格[3]がある．これらの内容は，すべて同一である．

日本での利用は，NTTがソフトウェア生産に試用した例，宇宙開発事業団，KDDなどで通信制御に利用している例，自衛隊からの受注による特殊な実時間システムに利用している例など，まだ多いとは言えないのが現状である．アメリカでも，国防や宇宙開係ではかなりの量のソフトウェアがAdaで記述されている（DoDが発注するシステムはすべてAdaで書くように義務付けている）ものの，それ以外の分野への普及は思うように進んでいないのが実情らしい．また，ヨーロッパにおいても，国防関係のほか，航空管制，通信制御，プロセス管理などの分野での利用が中心になっている．

1.1 Ada 9X計画

国際規格をはじめとする規格は，ふつう5年ないし10年ごとに見直しを行う．Adaの場合は，この見直しの機会を利用して言語の改訂を行うことになった．オブジェクト指向などの新しい考え方を取り入れて，より競争力の強い言語を制定することが目的である．改訂の作業は1988年ごろから始められ，1994年の末に完了した．この結果として新しく制定された言語をAda 9Xと呼ぶ．

Ada 9X計画を推し進めた主体はDoDである．ANSI規格の改訂作業，つまりアメリカ国内のプロジェクトとして，組織を整備した上で作業が始まった．ただしDoDも，現実には国際的な合意が得られなければ，新しい言語を作っても意味がないことは十分承知している．実際の作業は，国際規格に責任を持つISOと緊密に連絡を取り合って進められた．言語の大まかな枠組みはISO側が主導権を持って決め，細かな部分のつめをDoD側が行うという形態である．作業開始前に，ISOとDoDの間で，ANSIとしての改訂が完了するのと時期を合わせてISOの改訂としても受理されることを目指す，との合意文書[4]を取り交わした．

Ada 9X計画は，改訂すべき項目の決定，要求に従った言語改訂，そして移行措置という三つの段階に分けて，順次進められた．

改訂項目の決定は，DoD内部に限らず，広く一般から改訂要求を募ることから始められた[5]．このような大規模な体制を組んで作業を進めるのがAdaの伝統である．集まった改訂要求は多岐にわたった．それらの中で実際の改訂で取り上げるべき項目だけを残す整理の作業を行った結果，44の項目を今回の改訂案件として取り上げることを決め，22の項目を研究項目として残すことにした．

Ada 9X計画の当初の方針は，Ada利用者の要求に応えるための本質的な項目だけに限って改訂

† Ada 9X—a Procedural Language for Developing Large Scale Software—by Katsuhiko KAKEHI (Department of Information and Computer Science, Waseda University), Kiyoshi ISHIHATA (Department of Computer Science, Meiji University) and Haruhiko NISHIDA (Information and Communication Systems Laboratories, NTT).

†† 早稲田大学理工学部
††† 明治大学理工学部
†††† NTT情報通信研究所

することだったと思われる．プログラムの互換性や言語としての整合性に問題を生じるような大改訂は避けるはずだった．しかしながら，改訂要求を一般から集めたためもあって，この方針が守られたとは言い難い．結果としてかなり大規模な改訂になった．

新しいAdaの規格化の作業は，1994年末までにすべて完了し，IS（国際規格）として認めるかどうかの投票も，参加国すべてが賛成という結果が出た．これを受けて，1995年2月にAdaの新しい国際規格が制定された．こうしたISOとしての手続きと並行して，ANSIとしての規格化の手順も着々と進められている．こちらも間もなく新しい規格が制定されるはずである．

Ada 9Xの処理系の開発も進んでいる．現在利用可能なのはNew York大学のGNATプロジェクトの処理系だけだが，ここ1，2年のうちに数多くの処理系が市場に出回ると予想される．

1.2 Ada 9Xの主な改訂点

Ada 9Xでは，提案された多くの要求に基づいて，様々な機能追加が行われたが，大きなものは次の3点である．

（1） OOP（オブジェクト指向プログラミング）
（2） ライブラリの階層化
（3） 共有変数に基づく並行タスクの制御

これら以外にも，符号なし整数型の導入，手続きを指すポインタ型の導入など，いろいろな機能追加が行われた．その中で，我々日本人にとって意義の大きいものに文字の国際化がある．これまでは，ASCIIの文字しか扱うことができなかったが，今回の改訂で日本語などの字種の多いテキストまで対処できるようになった．

改訂案件とそれに従った実際の改訂内容については，以下の章で詳しく述べることにする．ただし，すべての項目を取り上げることは不可能なので，上にあげた主要項目だけに限ることにする．

1.3 分野別附属書

Ada 9Xに要求された新規機能は，実に様々なものがあった．これらをすべて一つの言語の中に取り込むと，言語仕様が巨大になりすぎる．処理系作成者の負担が大きくなるだけでなく，プログラマが言語を修得することも難しくなる．ほとんど使わない機能を提供することによって，実装上の無駄が生じることも問題である．また，要求された機能の中には，特定の分野での利用しか見込めないものも多かった．このような実情を踏まえて，Ada 9Xでは，分野別附属書（specialized needs annex）という考え方を導入した．

言語規格を核言語（core language）と分野別附属書の二つの部分に分ける．すべての処理系は，核言語を完全に実装しなければならない．しかし，附属書の機能については，それぞれ実装するかどうかを選択できることにした（どれも実装しない，という選択も認める）のである．処理系作成者は，その処理系の想定するユーザ層を考えに入れた上で，それぞれの附属書の取捨を決めればよい．一方，附属書に定められている分野の機能を導入する場合は，附属書に従うことが義務付けられる．

分野別附属書は，二つの目標の妥協として生まれたと考えることができる．一方で，Ada言語の浸透を図るには，広い応用分野のいずれにも対応する必要がある．しかし，もう一方では，処理系作成者やプログラマの負担を考えて，あまりに大きな言語とすることは避けたい．分野別附属書の導入によって，この二つの要求をともに満たすことができる．

Adaは，処理系間の互換性を完全に保証するために，言語のサブセットを認めない方針をとってきた．処理系が言語仕様を拡張したり縮小したりすることは認めない．分野別附属書は，この方針を継承するための便法と見なすこともできる．どの処理系でも同じAda言語のプログラムを受け付けるようにしたいのだが，応用範囲が広くなった現状では無理である．そこで，それぞれの応用分野ごとの標準仕様を定める．その分野の機能を提供しないことは自由だが，提供する場合にはこの仕様に従わなければならない．これによって，処理系ごとにまちまちな言語仕様となることが避けられる．

Ada 9Xでは，システムプログラミング，実時間システム，分散システム，情報システム，数値計算，高信頼性システムの6つの分野の附属書を用意している．

2. オブジェクト指向プログラミング

プログラミングパラダイムに関係する要求として上がってきたのが，オブジェクト指向プログラミングを支援する機能の強化であった．Ada 83（従来のAdaのことをこう呼ぶ）の設計は，1978年にその大枠が終わっていた．ANSI規格となる1983年までの間は，もっぱら規定の方法や細目について洗練を行っただけである．その時点で，オブジェクト指向は，言語設計の柱としてさほど大きなものとは意識されていなかった．

Adaが規格となってからの10年の間に，オブジェクト指向はパラダイムとして広く受け入れられ，大規模なソフトウェアの開発に欠かせないものとなっていた．実際，Adaを利用してシステムの設計をオブジェクト指向風に行うBoochの方法は，標準的な技法として受け入れられている．しかし，それをプログラム化する段になるとAda 83では言語機能の点で役不足であった．組込み型ソフトウェアという，Adaが当初から目標としてきた分野において直接の競争相手であったC言語が，C＋＋という新しい衣装のもとに一層普及してきているだけに，OOPへの対応は，Ada 9Xの大きな柱となった．Adaは，大規模ソフトウェアの開発を主目的とする言語であるため，ソフトウェア工学の成果の一つであるオブジェクト指向を取り入れることは自然なことと考えられる．

Ada 9Xの対処の大筋は，いわゆるOOP風の構文を設計するのではなく，オブジェクト指向にAda 83では対応しきれなかった，いわば不足していた機能を洗い出し，それらを既存の枠組みと両立する形で取り入れることにある．オブジェクト指向の特徴は，既存のクラスが持つ特性を継承するクラスを新しく追加することによって，既存部分に手を加えずにシステムを拡張して行けることにある．継承にあたっては，新しい属性を追加したり，メソッド（操作）を変更・追加したりできることが必要になるし，同じメソッド名の指定に対してこの継承関係に応じた操作が呼び出せるという，ある意味で動的なバインディングが必要となる．

2.1 継承とフィールド付加

Ada 83には，型の派生という仕組みがある．すでに宣言してある型と同種の新しい型を作り出すのだが，その際に元の型の属性や操作をそのまま継承する．操作については，新しいものに置き換えたり，追加したりすることもできる．Ada 9Xにオブジェクト指向を導入する際の基本的な考え方は，派生の仕組みを拡張して，継承に相当する機能を持たせることである．

単なる型の派生だけでは不足する機能としては，属性の追加がある．オブジェクトをAda 83でのオブジェクト（変数・定数）と見れば，利用者が多数の属性を与えることのできる対象は，レコード型になる．そこで，レコード型の一部として，そのフィールドが追加できるものを新たに設けることにした．これをタグ付き型（tagged type）という．

```
type Point is tagged
  record X, Y : Real; end record;
type Pixel is new Point with
  record Color : Integer; end record;
```

この例では，タグ付き型Pointに属性Colorを追加した型としてPixelを宣言している．

Ada 83では，派生関係にある型同士では，相互に型変換を行うことができた．タグ付き型の派生については，元の型に向かっての型変換は無条件に許す．この場合は，不必要なフィールドを無視するだけである．逆向きの変換は，付加したフィールドの値を新たに指定する形式を使って行う．型変換を利用することで，継承するにあたって変更した操作についても，変更前の操作を再利用することが可能となる．

2.2 クラスとディスパッチ

タグ付き型Tに対して，TとTから直接・間接に派生した一連の型は，いわば特性を共有するオブジェクトのクラスを構成する．このクラスを特別な性格の型と見なしてT'Classと表す．T'Classは，そのオブジェクトの具体的な構成が確定していないから，Ada 83でいう未制約型となり，直接にT'Class型の変数を宣言することはできない．しかしながら，T'Class型の引数を持つ操作を用意したり，T'Class型を指すアクセス型（Adaでのポインタ型）を設けたりすることはできる．これによってクラス全体を対象とするプログラミングを可能としている．

たとえば，Original型を継承して拡張したEn-

hanced 型があり，どちらの型に対しても手続き Operate が使えるものとしよう．このとき，次のような手続きを書くことができる．

```
procedure Dispatch (p: Original'Class)
  is begin Operate(p); end;
```

この手続きは，仮引数の型が Original'Class なので，Original 型でも Enhanced 型でも実引数として受け付けることができる．このとき，手続きの中で呼び出している Operate(p) は，実引数の型に応じて違う手続きを動的に切り替えて呼び出す必要がある．実際には，コンパイラがタグ付き型のオブジェクトそれぞれに，その型を表すタグを添加しておき，そのタグによって切替えを行うことになる．この動的な手続きの切替えをディスパッチという．

2.3 抽象型と抽象操作

クラスの構成を行う際に，その出発点となる型は，単なる雛型を示すだけで，その型のオブジェクトを実際に宣言して使うことを意図しないことが多い．「この型を先祖とするクラスのオブジェクトは，これこれの属性を持ち，これこれの操作を持つ」ことを示すのに，操作の内容が付加される属性に依存する場合は，具体的にその本体を書き示すことができない．ただ単に，「これこれの操作がある」としか書きようがない．こうした操作を抽象操作といい，抽象操作を持つ型を抽象型という．抽象型の変数は，宣言したり生成したりすることができない．

たとえば，次の抽象型を考えてみよう．

```
type Root is abstract tagged
  record … end record;
procedure Display (object: in Root) is
  … end;
procedure Operate (object: in out Root)
  is abstract;
```

手続き Display は，そのまま継承することができる．一方，手続き Operate は派生した型ごとに必ず定義を与えてやらなければならない．

2.4 アクセス型の一般化

操作の動的な切替えは，ディスパッチによるものだけでは対応しきれないこともある．そこで，アクセス型を拡張して，手続きや関数もアクセス型で指せるようにした．もちろん，Ada は強い型付けを持つ言語であり，その特徴を失うことがないように十分な配慮が行われている．

手続きや関数へのアクセス型を使うことで，オブジェクトの操作内容を動的に差し替えることも可能となる．その効用はそれだけに留まらない．Ada 以外で作られたシステムとのインタフェースをとるのにも大いに役立つ．たとえば，ウィンドウシステムの呼出しにあっては，いわゆるコールバック操作を引数として渡してやらなければならないことが多い．これが，Ada の特徴である型の安全性を保った形で可能となった．

他のシステムとの共存という点で言えば，手続きや関数ばかりでなく，通常の変数や定数も，それを指すアクセスデータを介して受渡しする必要が生じることが多い．そこで，手続き，関数，変数，定数などのいずれに対しても，'Access という属性を持たせることにした．ただし，静的な（つまり宣言で用意した）変数・定数については，あらかじめ 'Access 属性を利用することを予約語 aliased を添えて宣言した上でだけ使えるようになっている．また，静的な変数も指し示すアクセス型の場合には予約語 all または constant を明示する約束である．予約語 constant は，そのアクセス型を通じて指しているオブジェクトの値が更新できないことを同時に示す．

Ada 83 では，アクセス型で指せるものは，動的に生成したオブジェクトに限られていた．このため，リスト処理などのプログラミングで，静的な変数が指せないとか，初期設定でアクセス値で連なった一連のオブジェクトが宣言できないとかといった不便があったが，これらも同時に解消できることになった．

2.5 プログラム例および他言語との比較

Ada のオブジェクト指向機能の具体例として，汎用のリストパッケージを示す．これを使えば，任意の型のデータをリストに入れることができる．

```
package Linked_List is
  type Node_Type is
    tagged limited private;
  type Node_Ptr is
    access all Node_Type'Class;
  procedure Add (Item: Node_Ptr;
    Head : in out Node_Ptr);
  procedure …
```

```
private
  type Node_Type is tagged limited
    record
    Next: Node_Ptr: = null;
  end record;
end Linked_List;
```

特定の型のデータをリストに入れたい場合は，その型をNode_Typeに追加する形での継承を行えばよい．たとえば，整数データをリストに入れる場合の準備は次のようになる．

```
type Integer_Node is new Node_Type with
  record
  Integer_Element: Integer;
end record;
```

Ada 9Xでは，オブジェクトがどのクラスに属するかをコンパイル時に静的に把握できるようにしている．これがSmalltalk-80などの言語との大きな違いである．タグによるディスパッチだけは，実行時の動的な解釈が必要だが，それ以外の部分はすべてコンパイル時に処理できる．これによって，プログラムの安全性（型チェックが可能）と実行時の効率の両方を高めることができる．

Ada 9Xでは，言語の中のクラスの扱いが他のオブジェクト指向言語とはずいぶん違っている．まず，クラスという概念を独立したものとはせずに，型の一種と見なしている．複数の型を包含する上位概念としてクラスを定義することが自然だと思われるが，この概念を新しく導入すると，従来のAdaとはまったく違ったものになってしまう．そこで，動的切替えなどの機能を持つ，一種特別な型としてクラスの機能を実現することにした．また，他の言語のクラスには，名前の有効範囲を制御するプログラム単位としての機能があるが，Ada 9Xにはない．Adaでは，名前の制御は，もっぱらパッケージを使って行う．クラスは，データとその操作を記述し，これらの継承を行う機能しか持っていない．

オブジェクト指向言語では，特定のオブジェクトへメッセージを送る形の構文を使って仕事を依頼することが普通である．普通の手続き呼出しの用語で言えば，引数のうちの一つに特別な地位を与えて，ディスパッチなどの機能を持たせていることになる．2番目以降の引数には，これらの機能はない．Ada 9Xでは，このようなメッセージ授受の形の構文を採用せずに，手続き呼出しの構文をそのまま利用した．ディスパッチの機能はすべての引数が平等に持っている．もちろん，複数の引数がディスパッチに関与する場合は，それらのタグが等しくなければならない．

3. ライブラリの階層化

Adaでは，パッケージを基本単位としてプログラムを作成することが多い．パッケージは，名前を初めとする詳細を外部から隠す機能を持っている．大規模なプログラムの開発の場合，それぞれの部分ができるだけ独立になるように，相互作用ができるだけ小さくなるように努めることが重要である．パッケージは，これを実現するために有効である．従来の抽象データ型やオブジェクト指向を標榜する言語と違って，情報を隠蔽するための専用の機能を設けたのがAdaの特徴の一つである．

パッケージやライブラリに関する変更点について以下にまとめる．

3.1 子ライブラリ

Adaでは，すでに組み上がって外部に存在しているパッケージや手続き・関数などのプログラム単位の集まりをライブラリという．Ada 83でのライブラリには構造がなく，プログラム単位はすべて同列に扱われる．しかし，それではライブラリが大規模になるにつれて，その管理運営が難しくなる．

特に，ライブラリの一部に改変が加えられたとき，その影響を受ける利用者プログラムを自動的に見つけ，再コンパイルを施す作業が膨大になることが問題点として指摘されていた．ライブラリの仕様を変更した場合，これを利用するプログラムはすべてコンパイルし直さなければならない．ライブラリの一部だけを利用していて，該当する変更箇所には関係のないプログラムでも同じである．ライブラリが大きくなるにつれて，このような現象が頻繁に起こることは明らかである．

この問題を解決するには，個々のライブラリパッケージにも構造を持たせるのがよいと考えられる．Ada 9Xでは，ライブラリに置くパッケージの間に階層関係を設ける．つまり，ライブラリを木の形に配置する．この配置での親子関係を利用

してこれらの問題に対処することにした．これは，同時に言語に取り入れたOOPにも関係する．クラスを提供する単位は，ふつうパッケージである．OOPは，クラスの階層化によってシステムを構成していくから，対応するパッケージの階層化も必須なのである．

ライブラリの木の上では，基本的な機能や共通の定義を親の頂点に置き，細分化された機能を子供の頂点に置く．ライブラリを利用するプログラムは，木の中の特定の頂点を指定して利用する．この場合，その頂点の祖先にあたるライブラリはすべて自動的に利用できるが，子孫にあたるライブラリは利用できない．こうすると，そのパッケージと，そのパッケージの先祖のパッケージに改変が加わったときだけ再コンパイルすればよいことになり，再コンパイルの対象を限定することが容易になった．ライブラリの中の特殊な機能に相当する部分に変更があっても，その機能を利用するユーザプログラムだけを再コンパイルすればよく，大部分のプログラムには影響しない．

なお，言語で規定しているライブラリパッケージも，この機能を生かして階層化した形に整理された．その階層構造の一部を例として示しておこう．

```
Ada
  Text_IO
    Complex_IO
    Editing
  Strings
    Bounded
    Unbounded
  Real_Time
  Interfaces
    C
    COBOL
```

3.2 クラスとパッケージの関係

これまでに見てきたように，Ada 9XでのOOPでは，クラスは型に，オブジェクトは変数に対応する．Adaでは，プログラム上の大きな構成単位はパッケージである．パッケージでは，複数のデータや型，手続き，関数などを定義することができる．したがって，複数のクラスを一度に定義することも可能である．パッケージには，また，その実現のためだけに使って外部には隠蔽したデータや型なども用意することができる．AdaでのOOPでは，クラス階層を展開していくのに，このパッケージという単位を並べていくことになる．

パッケージや手続き・関数には，それを多様型（polymorphic）として提供するための汎用体と呼ぶ仕組みがある．OOPのために追加拡張したいろいろな機能にあわせて，この汎用体の引数機構が整理拡張された．たとえば，特定のクラスに属する型を汎用体の引数として受け付けることができるようになった．

4. 共有変数に基づく並行タスクの制御

Adaの並行処理において，タスク間の同期，通信はもっぱらランデブーを使って行うことになっていた．ランデブーは，CSP（communicating sequential processes）の流れをくむ強力な機能である．並行処理のたいていの問題は，ランデブーを使ってエレガントに記述できる．しかし，ランデブーは，二つのタスクが互いに相手を待つことによって同期をとるという形態であるため，オーバヘッドが大きくなるという欠点を持っている．ごく単純な同期を実現するにも，タスク切替えが何回も起こり，少なからぬ時間がかかる．悪いことに，ランデブーを使って問題を記述すると，小さなタスクを数多く使うプログラミングスタイルになりがちで，オーバヘッドの大きさがいっそう深刻な問題となる．実時間処理など性能面の要求の厳しい環境では，適用が難しかった．

4.1 防護レコード

同期，通信のメカニズム（プリミティブ）には，大別して，共有変数を利用するものとメッセージ通信に基づくものとがある．このうち，メッセージ通信だけですべてを片付けるというのがAdaの思想であった．これでは不十分なことが分かったので，共有変数に基づくメカニズムを追加しようというのが今回の改訂の趣旨である．

新しく導入された機構は，防護レコード（protected record）という名前を持つ．これは，その中にいくつかの副プログラム（関数，手続き，エントリ）を持つプログラム単位の一種である．他のプログラム単位からこれらの副プログラム類を呼び出して実行することができる．このとき，その副プログラムの本体を実行する主体は，

基本的には呼び出したタスク自身である（ランデブーの場合は呼ばれた側のタスクが実行を請け負う）．防護レコードは，タスクと違って受動的な単位であり，独自の制御の流れを持たない．これによって，タスク切替えの回数を減らし，オーバヘッドを小さくすることができる．

4.2 相互排除

防護レコードの中の副プログラム類は，ある瞬間にはどれか一つしか実行できない．一つの副プログラムを実行しているタスクがあれば，他のタスクは防護レコードの中に入れない．相互排除の機能がプログラム単位自身によって保証されているわけである．ただし，関数と関数であれば，データの値を読み出すだけなので，同時に走っても互いに影響を与えない．この場合に限って，複数のものが同時に走ってよいことになっている．

相互排除は，ロックの状態を表す変数を利用して実現する．このとき，各タスクはこの変数の値が変わるまでループしながら待つ（busy wait）のが標準的な姿である．セマフォなどの機構を利用するよりも，この方が効率的だとされている．

防護レコードの中のエントリと手続きは，ほぼ同じ機能を持つが，エントリの方にはバリアと呼ぶ条件式をつけることができる．これは論理型の式で，その値が真のときに限って，そのエントリを実行することができる．偽であれば，真になるまで呼び出したタスクは待たされる．

例として，計数セマフォ（counting semaphore）を実現する防護レコードを示す．

```
protected type Counting_Semaphore
              (Initial : Integer : = 1) is
  function Value return Integer ;
  procedure V ;
  entry P ;
private
  Counter : Integer : = Initial ;
end Counting_Semaphore ;

protected body Counting_Semaphore is
  function Value return Integer is
    begin return Counter ; end Value ;
  procedure V is
    begin Counter : = Counter+1 ; end
    V ;
  entry P when Counter>0 is
    begin Counter : = Counter-1 ; end
    P ;
end Counting_Semaphore ;
```

防護レコードの考え方は，特に目新しいものではない．1970年代に数多く提案された同期，通信のプリミティブの一つにモニタがあるが，防護レコードはモニタに若干の変更を加えたものにすぎない．

防護レコードがモニタと違うのは，中に入った後で何らかの条件を待つ必要が生じた場合の扱いである．モニタの場合は，条件変数に対してsignalおよびwait操作を行うことで，待ちを実現した．これに対して，防護レコードの場合はrequeue文を使うことになっている．requeue文を実行したタスクは，実行中のエントリや関数の実行を放棄して，別のエントリなどの実行を先頭から始めることになる．

5. 文字データの国際化

従来のプログラミング言語は，比較的少数の文字だけを文字データとして許していた．具体的には，ASCIIやEBCDICなど1バイトで表現できる128～256字程度の文字セットを採用していることが多かった．英米ではこれで用が足りていたかもしれないが，近年のように計算機がいろいろな国で広く使われるようになってくると，これだけでは不十分である．それぞれの国固有の文字を含めて，より広い範囲の文字を利用できるようにすることが必要である．このように，文字の種類を増やす動きを文字の国際化と呼ぶことが多い（国際化と呼ぶのはやや不適切だと思うが）．

文字の国際化の必要性は，欧米でも広く認識されている．言語の規格化を担当するISO/IECのJTC 1/SC 22では，その下の各言語担当のWGに対して，文字の国際化を促進するよう求める決議をすでに採択している．日本をはじめとするアジア各国から強い要求があったのはもちろんだが，ヨーロッパにも国際化を望む国が多かった．ただし，ヨーロッパ諸国の場合は，アクセント記号がついた英字などの比較的少数の文字を256字種の中に入れる（文字の選び方は国ごとに違いうる）という要望がほとんどなのに対して，アジア特に漢字使用国の場合は，一つの文字を2バイト

以上で表現したものを念頭に置いている点で，発想にかなりの違いがある．

5.1 文字データの型

プログラミング言語で文字の国際化を行う場合は，文字データを表す型をどのように扱うかが設計方針の分かれる点である．最も簡単な方式は，従来と同じ文字データのための型（標準の文字型，たとえばCharacter）で2バイト文字まで扱えるとしてしまうことである．今のところ，Modula-2やPascalがこの方針をとっている．

たいていのプログラミング言語は，文字型に含まれる文字の種類を文法書では決めていない．処理系ごとにOSその他の環境を判断して決めることになっている．したがって，必要ならば処理系の責任で2バイト文字を文字の範囲に含めることにしてもいっこう差し支えない．確かに，理論的にはこれで十分である．この方式をとれば，言語の文法書をほとんど書き替える必要がないので，言語の設計者にとって最も労力のかからないやり方である．

しかし，この方式で現実に2バイト文字がうまく扱えるかどうかは疑問である．実際に市販される言語処理系が，標準の文字型を2バイト文字に拡張するとは思えない．1バイトですむデータをすべて2バイトで表現するのでは，メモリの使用効率が悪すぎるからである．欧米では，ほとんど1バイト文字しか使わないため，処理系がそれに合わせた設計方針をとるのは当然である．これでは，2バイト文字を使用したいユーザの要望は満たされない．

より現実的なのは，従来の標準の文字型はそのまま残して，拡張文字セットを扱うための型を別に設けることであろう．Fortran, Cなど実用言語の多くは，この方式を採用する方向にある．Ada 9Xも，同じ方針であり，Wide_Characterと呼ぶ型を設けて，2バイト文字に対応させることになっている．

5.2 文字データとその表現

2バイト文字が入ってくると，文字の外部表現が複雑になる．2バイト文字を1バイト文字と区別して表現するための約束が必要だからである．日本でも，ISO 2022に基づいたエスケープシークエンスを使う方法のほかにshift-JIS，EUCなど各種の規約が使われている．言語の規格を定める際に，これらの規約（広く言えば文字の表現方法）を考慮に入れる必要があるかどうかが問題になる．結論から言えば，Adaの場合は，文字の表現方法は言語規格の範囲外の問題と見なしている．外部表現を適当に解釈して，正しい文字データを構成するのは，コンパイラまたは実行時システムの責任であって，ユーザに表現を意識させることがあってはならない．別の言い方をすると，プログラムが扱うのは，抽象的なデータとしての文字の列であり，それを構成する具体的なバイトデータの列は見えないようにすべきである．ただし，このあたりの考え方は言語の性格によるところが大きい．Cは，システム記述言語という性格から，エスケープシークエンスなどの表現までユーザが意識してプログラムを書くことを可能にしている．

欧米では，2バイト文字をよく知らない人も多く，1バイト文字型でバイト列を扱えるのだから，特に2バイト文字を扱う機能が必要とは思えないという議論がしばしばある．しかし，この考え方は受け入れ難い．たとえば，文字列の探索を行うときに，一つの文字の2バイト目と次の文字の1バイト目を合わせたものを別の文字と見なすようなボタンの掛け違えが起こる．やはり，具体的な表現から独立した抽象的なデータとして文字を扱う機能が必要である．

Adaでは，他の言語と違って，文字型に対応する文字の種類を規格で定めてしまう方針をとっている．プログラムの互換性を重視した結果である．Wide_Character型に対応するのは，ISO 10646のBMP（basic multi-lingual plane）と定めることにした．BMPは，漢字も含めてほとんどの国の文字を2バイトで表現できるようにした文字コード体系である．10646が文字の国際規格として制定された以上，言語の側がこれに準拠するのは当然だと考えられる．ところで，日本では今のところ10646が広く普及するかどうかはっきりしない状況にある．10646 BMPしか受け付けないような言語では，使い物にならない．そこで，規格では標準仕様としてBMPに決めるが，処理系ごとに違う文字セットをWide_Characterにあててもかまわないことにしてある．ただし，BMP以外の文字セットを使った場合は，他の処理系との互換性は保証されない．

なお，Wide_Character 型はその環境で使えるすべての文字を含むのが大前提である．Fortran のように拡張文字用の型を複数用意する考え方もあるが，これでは違う型に属する文字をいっしょに扱うことができなくなる．文字の型を Wide_Character 一種類に限れば，この問題を避けられる．Wide_Character のほかに従来の型 Character も残っているが，これは Wide_Character の一部の文字だけをより能率よく扱うための一種の便法と考えるべきである．

5.3 文字データの操作

以上のように，文字のための型が決まれば，後はほとんど自動的に仕様が決まる．文字操作の副プログラム類は，Character とほとんど同様のものが Wide_Character にも定義してある．文字定数の書き方も同じである．Widc_Character と Character の間の型変換関数も用意されている．入出力は，Characterが並んだファイルと Wide_Character が並んだファイルを別の型として扱うことにした．整数その他の入出力は，どちらの型のファイルに対しても同じ方法で行える．

プログラミング言語における文字の国際化をさらに推し進めると，プログラム中の識別子や予約語まで英字以外の文字で書けるようにすることが考えられる．これは確かにプログラムの可読性の向上の面で望ましいことではあるが，プログラムの互換性を損ねるおそれがある．特に，どの文字を識別子に許すかについて国際的な合意を得ることが難しい．そこで，今回の改訂ではそこまで踏み込まなかった．将来の改訂の際に再び話題になると思われる．

6. 分野別附属書

はじめに述べたとおり，Ada 9X では特定の分野で必要となる機能の集合を分野別附属書と呼ばれるもので規定している．

分野別附属書で規定しているのは Ada 9X の言語仕様に対する拡張であるが，ユーザや他の標準化団体にも行える程度の拡張しか行っていない．たとえば，標準ライブラリの設定，属性の追加などである．言い替えれば，Ada 9X の構文的，文法的な拡張は避けている．このことは今後新しい分野向けの附属書を作成する場合や，既存の分野別附属書を改訂する場合の処理系への影響を小さくする狙いがある．

以降，分野別附属書それぞれについて，概要を紹介する．

6.1 システムプログラミング

ハードウェアに密接に関係する低位機能を定義している．これらの機能は，オペレーティングシステムや実行時システムなどを Ada で記述する際に必要となる．

従来からあった機械語を直接に記述する機能や，割込み処理のための機能，プログラム間で共有される変数のためのプラグマ，タスクの一意性を識別するための識別番号を取得するための機能などが含まれている．

6.2 実時間システム

もともと Ada は DoD で組込み型システム作成用として設計された言語である，という経緯もあり，実時間システムや組込みシステムへの適用が多かった．この分野でサポートの要求の最も多かったのがタスクの優先順位やスケジューリングをユーザが制御するための機能であった．これらの要求に応えるため，実時間システム向け附属書では，以下のような機能を提供している．

- ●優先順位の宣言，動的な優先順位
- ●二つのタスクが関係する場合の優先順位の制御
- ●タスクのスケジューリング算法の宣言
- ●防護レコードとの関係の指定
- ●エントリ呼出しの待ち行列の処理方法の宣言
- ●タスクに対する制限
- ●同期/非同期のタスク制御

タスクの実行に関するきめ細かな制御を行うものがほとんどである．これらの機能は，実時間システムにおける厳しい時間的制約を満たすために必要とされる．

このほか，abort 文の実際的な働きについて詳しい情報を利用者に提供することを処理系作成者に要求している．また，実時間システムの実際の時間的要求に則すために，様々な言語機能を制限してプログラムを作ることで，処理系がより高速のコードを生成できるようにするための，一連のプラグマが用意してある．

6.3 分散システム

Ada で書いた一つのプログラムが 1 台の計算機上で実行されるとは限らない．多数の計算機上

に分散して配置され，実行されることも多い．その場合に，分散配置することに関係して生じる事柄を，プログラムの中核部分とは切り離して記述できるようにするのが目的である．

Ada 83では，一つのAdaプログラムは一つの名前空間，すなわちアドレス空間を持つ．直観的にはこれが一つの実行プログラムに相当する．Ada 83ではすでにタスクなどの並列実行に関する概念は導入されているが，一つのシステムが複数の（独自のアドレス空間を持つ）プログラムで構成されるような場合に対応する概念はなかった．

これに対して，Ada 9Xは実行プログラムの単位に関する概念としてパーティション（partition）を導入している．一つのパーティションは，一つの計算機の上でのプログラムを意味すると考えてよい．Ada 9Xのプログラムは複数のパーティションから構成される．

この附属書では，複数の名前空間を持つシステムの概念と，その間で可能な操作に関する規定を定めている．

パーティションには，能動的なもの（プログラムに対応する）と受動的なもの（データに対応する）がある．コンパイル後のパッケージや副プログラムなどのライブラリ単位からどうパーティションを作るかは処理系依存であるが，基本的には必要なライブラリを列挙することになる．

パーティションの間で可能な操作には次のものがある．

- 受動的なパーティションを使用した共有変数
- 遠隔呼出し（Remote Call）
- パーティション間のデータの受渡し（通信）

もちろんこれ以外の仕組みを処理系で提供することは許される．

6.4 情報システム

Adaは大規模システム向けの言語として設計されたが，大規模システムの一つの典型である会計や給与計算システムなどの情報システムにはあまり適用されていない．今までCOBOLなどで実現されていたこれらのシステム向けの機能が用意されていなかったことが原因の一つとしてあげられてきた．また，大規模システムでは全体の更改よりも一部の設計変更や，作り直しが多くなるので，ほかの（他言語で書かれた）部分とのインタフェースも重要な要素となる．情報システム向け附属書では，

- 10進整数の入出力（COBOLのピクチャー形式による書式制御を含む）
- 固定小数点数の入出力
- 10進数に関するパッケージ

などがサポートされている．このほか，他の言語とのインタフェースとして，COBOLやCで書いてある副プログラムを呼び出すための規約を定めることによって，この分野の機能強化を図っている．

6.5 数値計算

精密な数値計算を行う場合に必要となる事項が盛られている．まず，浮動小数点計算，固定小数点計算の計算モデルを定義する．それを使って，実行時システムの計算についての詳細な情報を調べるための，各種の属性を定義する．また，言語定義の基本関数パッケージについて，その計算精度についての要件をここで規定する．

複素数型の基本パッケージ，入出力パッケージ，基本関数パッケージも，この附属書で定義している．

なお，従来のAdaの場合，数値計算関係のパッケージは言語では定めていなかった．Ada 9Xの数値計算ルーチンの定義は，精度などの定め方が精密であり，他の言語の同種ルーチンのモデルになりうると考えられる．

6.6 高信頼性システム

交通管制，原子炉制御など，システムの保安・保全が最重要な応用分野では，プログラムの検証が欠かせない．そのためには，処理系が生成する機械語について，かなり詳しい知識も必要となる．ところで，核言語では，処理系による最適化の余地を大幅に認めている．このため，Adaプログラムの字面だけでは，生成される機械語列が（利用者にとって）予測できないこともある．

この附属書では，予測可能な，そして検証可能な機械語列の生成を保証するためのプラグマを導入するとともに，処理系作成者に対して，生成する機械語列についての詳しい情報を開示することを義務付けている．また，複雑な機能を削って必要最小限の部分だけを残したサブセット言語を定義している．このサブセットの範囲でプログラムを書けば，プログラムの検証が可能であり，安全

性を保証することができる．もちろん，処理系に対してもこのサブセット言語を正しく処理するという保証が必要である．

7. おわりに

やや主観的になるが，全体を通しての感想を述べることで，まとめにかえたい．

今回の改訂は，Ada利用者からの要求もさることながら，Adaの利用が当初に予想したほど伸びていないことに対する，DoD，ベンダなどの巻き返し作戦のようなものを強く感じさせるものになっている．従来のAdaが技術的には優れた言語であったにもかかわらず，十分には市場で受け入れられなかったことを反省して，ユーザの要求にできるだけ応える姿勢をとった．余談になるが，Ada関係者のC＋＋に対する対抗意識には強烈なものがある．C＋＋に負けるなということが，Adaの改訂作業の隠れた合言葉になっていたと言って過言でない．

このような態度をとった結果，改訂の規模は相当に大きなものになった．Ada 9X計画の出発時点では小幅な改訂という触れ込みだったのだが，これは絵に描いた餅だったようである．でき上がった言語の規模も大きい．実際，参考文献1)が280ページ程度であったものが，6)では450ページ（このうち核言語部分は350ページ）に膨れ上がった．

現在の時点でこの結果を評価することは難しい．一方では，ユーザの要求に屈してまとまりのない言語になってしまったと見ることもできるだろうし，一方では様々な機能を含んだ非常に強力な言語ができ上がったと見ることもできる．筆者らの目から見れば，大規模ではあっても言語全体としてのまとまりは保たれていると思えるが，この評価が市場で受け入れられるかどうかは，今後の関係者の努力にかかっていると言えるだろう．

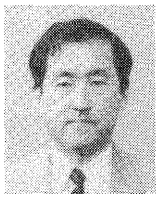
参考文献

1) American National Standard Institute, Reference Manual for the Ada Programming Language, ANSI/MIL-Std-1815 a (1983).
2) International Standard Organization, Reference Manual for the Ada Programming Language, ISO/8652-1987 (1987).
3) 日本工業技術院，電子計算機プログラム言語Ada, JIS X-3009-1991 (1991).
4) International Standard Organization, Memorandom of Understanding between the Ada 9X Project Office and ISO-IEC/JTC 1/SC 22/WG 9 Ada, ISO-IEC/JTC 1/SC 22 N 844 (1990).
5) Ada 9X Requirements, Office of the Under Secretary of Defence for Aquisition, Washington DC. (1990).
6) Ada 9X Mapping/Revision Team, Ada 9X Reference Manual, ANSI/ISO DIS 8652 (draft version 5.0) (1994).

（平成7年1月11日受付）

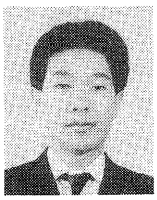
筧　捷彦（正会員）

1968年東京大学工学部卒業．早稲田大学理工学部情報学科教授．プログラミング言語の設計，処理系，支援環境に興味を持つ．日本数学会，日本ソフトウェア科学会，ACM各会員．



石畑　清（正会員）

1976年東京大学大学院修士課程修了．明治大学理工学部情報科学科助教授．プログラミング言語とその処理系，プログラミング環境，アルゴリズムなどに興味を持つ．日本ソフトウェア科学会，ACM各会員．



西田　晴彦（正会員）

1987年東京工業大学大学院修士課程修了．NTT情報通信研究所に勤務．プログラミング言語処理系，開発支援環境，汎用コンピュータプラットフォーム仕様などの開発検討に従事．