|連載|

記述の科学

第1回 記述とは

木下佳樹　高井利憲
(産業技術総合研究所)

今月から3回にわたって，記述の科学という題で書き物を連載させていただくことになった．データを計算機で自動処理するために，まず計算機で処理可能な形，つまり形式言語における文あるいは項として，データを表現しなければならない．そのように表現することを記述と呼んでいる．本連載を通して，(1) 記述の形式，(2) 記述の処理，(3) 記述の方法などについて考えてみたい．第1回は，記述とはどういうものか，について，2人の研究者の対談を交えながら記す．

苦迦(クカ)と羅茶(ラーチャ)は，いずれも計算機科学の研究に携わっている，なかよしの同僚で，苦迦・羅茶の連名で共著論文もたくさん書いている．羅茶が最近，記述こそ大切，とあちこちで言い出したので，苦迦が訝って議論を始めた．

苦迦：早速ですが，ずばり記述の科学とは何ですか．たとえば，情報科学とはどのような違いが……

羅茶：あ，ちょっと待ってください．

苦迦：はい？

羅茶：いや，そんな大上段にふりかぶってものを言うのではなく，身近なところから出発して話していきたいと思うのです．記述の科学ということばで何を言いたいのか，については，話していくうちに明らかになればいいのではないでしょうか．

苦迦：はあ，ちょっと気が短かったかな．

羅茶：いいえ，そんなことはありません．

ライブラリ

羅茶：さて，我々が計算機を使うのは，いろいろなデータの処理をするため，と言ってよいでしょうが，計算機で処理するためにはまず，処理できる形でデータを表現しなければなりません．

苦迦：もちろん．

羅茶：そのように表現することを，記述，と言っ

てみたいのです．データの処理をしたいのは人間です．しかし，データの自動処理を行うのは計算機です．人間のしたいことを計算機に伝えるのが記述という行為です．

苦迦：なるほど．

羅茶：人間同士のコミュニケーションのためには，言葉を使います．

苦迦：ふむふむ．言語ですね．

羅茶：言葉を計算機と人間の間でのコミュニケーションにまで広げて考えたいというわけです．

苦迦：すると，プログラミング言語を考えようということでしょうか．

羅茶：確かにプログラミング言語はプログラムを記述して計算機に実行させるためのものです．同じようなことを，プログラム以外のものの記述にも拡げて考えたいものだ，と言いたいわけです．

苦迦：確かに，プログラミング言語処理の技術は，情報処理技術全体のなかで大きな役割を果たしていますね．しかし，プログラム以外のものの記述とはどのようなものの記述でしょうか．

羅茶：たとえば，命題や述語，いわゆる論理式の記述です．

苦迦：はあ，命題ですか．命題というのは，論理学などで出てくる命題と考えてよいでしょうか？真偽が定まるような言明というか，主張というか……

羅茶：はい．現在普通に見られるプログラミング言語では，データの処理は扱うけれど，論理式の処理は行いません．しかし，抽象データ型の取り扱いを突き詰めると，論理式を処理する必要が自然に出てきます．

苦迦：抽象データ型ですか．なんだか実行が遅くなりそうですね．

羅茶：処理系やコンパイラの出力コードの実行効率もさることながら，現状では抽象データ型の支援が十分になされている言語があまりありません．しかし，命題の処理を考えることによって，より強力なプログラミング支援が可能になります．

苦迦：うーむ，強力な支援は自動的な支援とは限らないので……

羅茶：さすがにするどい！　確かに，現状のプログラミング言語が強力な支援を提供しない理由の1つは，自動的な支援を行いたいから，ということだろうと思われます．しかし自動的な支援を与えながらも，もっと強力な支援を提供することも可能です．

苦迦：それはぜひうかがいたいですね．支援を強力にすることができる，というのは，たとえば，どのような点においてなのでしょうか？

羅茶：たとえば，プログラムの記述における，ライブラリの“利用条件”のようなものを考えてみましょう．

苦迦：ライブラリですか．C言語で言うとたとえばstdioのようなもののことですね．

羅茶：そうです．ここでは，総和や総積を例えにして考えてみましょう．

苦迦：総和や総積のライブラリというのは，あまりイメージがわきませんが……

羅茶：foldrやfoldlという関数は聞いたことありませんか？

苦迦：ああ，畳み込み関数と呼ばれるものですね……

計算機の黎明期のことを思い起こすと，プログラミングの自動化のために，いろいろな記号処理がなされるようになった．コンパイラが作られるようになり，プログラミング言語の間の翻訳が計算機によって自動的に行われるようになった．アセンブラ言語から始まって，50年前にはすでにLispもAlgolも出現していたのは驚くべきことのように思われる．プログラミング言語はその後も発展を続けて，現在では，パラメータ付きモジュール，クラスといった高度な機能がいろいろな言語に備えられている．プログラムライブラリの柔軟な構成を，誤用を防ぎながら可能にすることが，これらの機能の大切な役割だと思われる．

ライブラリの誤用の防止とは，結局のところ，それが使われる状況を限定することであろう．仮引数

を2つ要求するのに実引数が1つしか与えられないようでは困る（もちろんわざとそのようなことをする場合もあったが）ので，この場合は実引数も必ず2つと限定したい．整数型の仮引数の場所に実数型の値をいれては具合が悪いので，実引数も必ず実数型にするように限定したい．ライブラリの利用条件の限定を行うために，データ型が導入され，モジュール，クラスなどが導入され，さらにそれらのパラメータが導入された．これらの言語機能によって，ライブラリ関数に対して，込み入った利用条件を課すことができるようになった．しかし，まだ表現しきれないような複雑な利用条件もある．

例として総和や総積の一般化を考えよう．二項演算の加算から，n個のものの総和をとる演算を導いたり，乗算からn個の総積を導いたりすることができる．これを一般化して，一般に二項演算@を与えると，n個の数 $x_1, x_2, \ldots, x_n$ に対して，$x_1 @ x_2 @ \ldots @ x_n$ を計算する演算を考えることができる．仮にここでは総@と呼ぶことにしよう．

苦迦：ちょっと待ってください．$x_1 @ x_2 @ \ldots @ x_n$ はアヤシイですね．@は左結合的なのですか？　それとも右結合的なのですか？　どのように括弧をつけるのかが分かりません．$(\ldots(x_1 @ x_2) @ \ldots x_{n-1}) @ x_n$ なのか，$x_1 @ (x_2 @ \ldots (x_{n-1} @ x_n)\ldots)$ なのか，それとも他の括弧のつけ方をするのか．

羅茶：その通りです．いやはや，これから説明しようとしたのだけれど，先を越されてしまいました．@は結合法則 $x @ (y @ z) = (x @ y) @ z$ を満たすこと，としておかないと，総@の考え自体が曖昧になってしまいますね．

苦迦：なるほど，結合法則を満たすのなら，右結合的であろうが，左結合的であろうが，出てくる答えは等しいことになりますね．もっとも途中の項の計算中に桁あふれのような例外が発生すると厄介ですが．

羅茶：さしあたって，そのような，いわゆる副作用は考えないことにしておきましょう．

苦迦：分かりました．総和や総積では，たまたま加算も乗算も結合法則を満たすので，変なことが起こらなかったわけですね．

羅茶：はい．たとえば減算は結合法則を満たしませんので，たとえば $((30 - 9) - 8) - 7 = 6 \neq 20 = (30 - 9) - (8 - 7)$ ですから，$30 - 9 - 8 - 7$ と書くと，このどちらを指すのか分かりません．もっと他の括弧付けもあり得ます．もっとも，減算の場合は，左結合的だ，つまり $30 - 9 - 8 - 7$ は $((30 - 9) - 8) - 7$ を表すもの，という約束をする場合が多いわけですが，それは単に約束事ですから．

苦迦：お話に戻ると，どんな二項演算を持ってきてもいいわけではなくて，結合法則を満たすような二項演算を持ってこなければならない，ということになりますか？

羅茶：はい．結合的な演算でないと総@の考えが成り立たないというわけです．実はもう1つあります．n個のもの，と言ったけれど，0個のものが与えられたときには何を返すべきか．

苦迦：なるほど，空集合の場合ですね．今はプログラミングの話だから集合じゃなくてリストと考えたほうがいいかな．

羅茶：そうです．『$x_1, x_2, \ldots, x_n$ の総@』＝『$x_1, x_2, \ldots, x_{n-1}$ の総@』$@ x_n$ ですから，特に $n = 1$ の場合を考えると『$x_1, x_2, \ldots, x_{n-1}$ の総@』は空リストの総@になりますが，左辺は x_1 に等しくなりますから，x_1 ＝『x_1 の総@』＝『空リストの総@』$@ x_1$ が得られます．これがどんな x_1 についても成り立たなければなりませんから，『空リストの総@』は，少なくとも二項演算@の左単位元だと定めるのが妥当です．

苦迦：ふむふむ，『$x_1, x_2, \ldots, x_n$ の総@』＝『$x_1, x_2, \ldots, x_{n-1}$ の総@』$@ x_n$ から始めると，『空リストの総@』は左単位元でなければならない，という結論が出たわけですね．しかしもちろん，『$x_1, x_2, \ldots, x_n$ の総@』＝ x_1@『$x_2, x_3, \ldots, x_n$ の総@』ででもなければいけませんが，これから出発すると，『空リストの総@』は右単位元でもあるとするのがよい，ということになりますね．

函数 foldr は，3つの引数をうけとって，自然数を返す．第一引数は，二項演算である．第二引数は，自然数ならば何でもよい．第三引数は，自然数をいくつか（有限個）並べてできる リスト（並び）である．

foldr (@) m (n1, ...) の値は，第三引数のリストの形によって場合わけして次のように定義される．

1. 第三引数が空のリストの場合．この場合は，すでに述べたように第二引数を値として返す．空のリストを () と書くと，

   foldr (@) m () ← m

   となる．ここで，← は，その右辺の値によって左辺の値が定められる，ということを意味するために使っている．

2. 第三引数のリストが空でない場合．この場合，その先頭の自然数を n とし，リストの残りを...と書くと，

   foldr (@) m (n, ...) ← m @ (fold (@) m (...))

   となる．

この定義に従って，たとえば `foldr (+) 0 (3, 2, 1)` を計算してみると，

```
    foldr (+) 0 (3, 2, 1)
= 3 + (foldr (+) 0 (2, 1))
= 3 + (2 + foldr (+) 0 (1))
= 3 + (2 + (1 + foldr (+) 0 ()))
= 3 + (2 + (1 + 0))
= 3 + 2 + 1
```

となり，確かに，結果は，第三引数 (1, 2, 3) の総和になっている．

図 -1　foldr の定義

羅茶：その通りです．つまり『空リストの総 @』は両側単位元だとするのが妥当です．

苦迦：なんだか話しが込み入ってきましたね．空リストなんてものがあるから単位元の話が出てくるわけですが，リストは長さ 1 以上のもの，という風に最初から決めておけば，単位元の話は消えますね．

羅茶：これまた，まったくその通り．

苦迦：分かりました．ともあれ，両側単位元を持ち，結合法則を満たすような二項演算 @ をもらうと，それを n 項演算に拡張することを考える，というわけですね．待てよ，聞いたことがありますね．なんだ，入力データは，要するにモノイドを 1 つもらう，ということになりますね．

総和や総積は，函数型言語では，たとえば foldr という函数を使って計算される．これを例にとろう．foldr は**図 -1** のように定義される．第三引数のリストの各自然数に第一引数の演算 @ を次々に施して，総 @ を結果として返すのが foldr である．第二引数は，第三引数が空のリストであるときに返す値である．

foldr は，一般に二項演算を n 項演算に拡張しようとするものである．たとえば，二項演算である加算 + を，任意の個数の総和に拡張する．

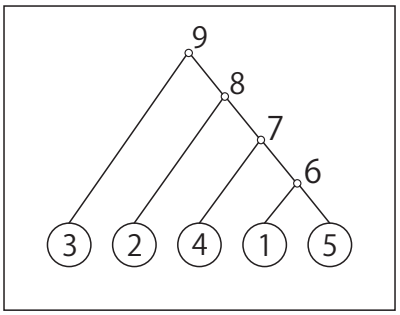
```
foldr (+) 0 (5, 9, 7, 4, 6) = 5 + 9 + 7 + 4 + 6
```

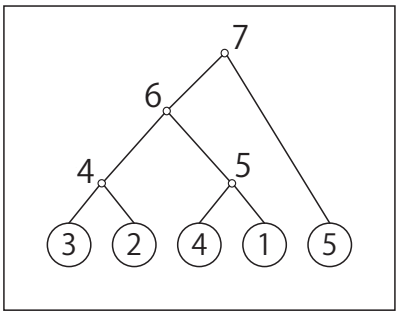
という具合である．ここで，加算 + を実引数として与えるときに，括弧で囲って (+) と記している．さもないと foldr と 0 (4, 9, 7, 4, 6)（これは意味不明の式になってしまうが）に加算を適用するかのようにみえてしまうからである．二項演算はもちろん加算でなくともよく，たとえば

```
foldr (×) 1 (5, 9, 7, 4, 6) = 5 × 9 × 7 × 4 × 6
```

となる．

さてここで，一歩立ち止まって考えてみたい．総 @，つまり $x_1 @ x_2 @ \ldots @ x_n$ は，第一引数に与えられる二項演算が結合的であるときにしか意味が確定しない．foldr は，特別な括弧のつけ方 $x_1 @ (x_2 @ (x_3 \ldots @ x_n)\ldots)$ を指定した計算を行っている．次に，上記の式が総積を表すためには，第二引数が第一引数の二項演算の右単位元でなければならない．





各節に記されている数字の順に，節の値の計算が終了するとしよう．左側では，右端の節の計算が終わるまで，総積の計算がはじまらない．右側では，隣同士の計算が終了し次第，積が計算される．さらに，下図の場合は，隣同士でなくとも，2つの計算が終わり次第，それらの積を計算していく．

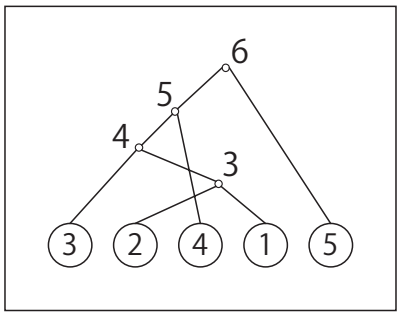


図 -2　総 @ の計算

苦迦：なんだか理屈を並べましたね．しかし，二項演算が結合的でなくっても，始めに計算したかった総 @ の考えは，確かに崩れるかもしれませんが，foldr の意味はきっちり決まっているから，それでいいのではありませんか．

羅茶：計算の注文を出す側で，順序がどうでもいいのなら，計算の注文を受ける側では，勝手に決めてもよかろう，というわけですね．そのような場合も，確かにたくさんありますが，計算する側でも前もって決めておく場合もあります．たとえば，$x_1, x_2, \ldots, x_n$ の計算を並列に行って，それらの結果の総 @ をとる，という場合を考えましょう．

苦迦：ふむ，そういう場合は，計算が終わったものから順に，というか，隣同士の項の計算が共に終わったら，構わず @ を計算していく，というやり方が早いでしょうね(**図 -2** を参照)．

羅茶：その通り．foldr のように右結合的，あるいは左結合的などと決めてしまうと，早く計算の終わった項が待たされるかもしれませんからね．

苦迦：隣同士で計算が終わったものからどんどん，@ の計算をしていきたいですね．そうすると，この場合には，確かに，あらかじめ括弧のつけ方を定めておくわけにはいきませんね．

羅茶：そうです．しかも，このようなやり方で高速化を図るためには，二項演算 @ が結合的なものでなくてはなりません．

苦迦：そして，そのような制限を二項演算に課すには，結合法則に相当する等式を引数への条件としておかなければならないわけですね．なるほど，等式は，論理式の特別な場合というわけか．ところで，@ が結合的である，という条件しかない場合には，隣同士の項の計算が終わったときにだけ @ の計算に渡せるわけですが，@ がさらに交換律を満たせば，つまり $x \mathbin{@} y = y \mathbin{@} x$ が成り立てば，もっと効率を上げられますね．

羅茶：その通り．

苦迦：@ が交換律も満たせば，1 番目に計算が終わったものと 2 番目に終わったものの @ を計算し，さらにその結果と 3 番目の @，さらに 4 番目との @，というように一般に n 番目までの総 @ と

n+1 番目のものの @ を計算していく．この算法が一番効率がいいでしょうね．

苦迦：総和，総積の例はよく分かりました．foldr ともう 1 つ foldl があれば何も考えることはない，と思っていましたが，二項演算に条件を加えたり，単位元であるという条件を加えたりすることによって，コードの効率化の可能性が広がるわけですね．

羅茶：その通りです．条件が増えれば，それを満たすものたちは限定されていくわけですから，効率化の可能性が広がるのは当然と言ってもよいでしょう．

苦迦：ところで，引数として与えられる二項演算が結合律や交換律を満足することは，どのようにして指定するのですか？

羅茶：第一引数の実引数として二項演算を与えれば，それが結合律を満たすのかどうか，あるいは交換律を満たすのかどうかを，処理系が自動的に調べてくれるのが理想ですね．

苦迦：しかし，そのような自動化を可能にするためには，入力データに課す条件は，あまり一般的なものにするわけにはいかないのではないでしょうか．

羅茶：はい．そこで，たとえば，実引数とともに，求められる条件を実引数が満たすことの証明を一緒に渡すことが考えられます．

苦迦：ふーむ，証明をデータとして渡すわけですか．

羅茶：証明を記述する形式言語を作ることができます．そのようなものは一般に証明図と呼ばれています．証明を表す図というわけです．

苦迦：なるほど，証明と言うと日本語や英語で記されているもの，と考えてしまいましたが，そうではなく，何か人工的に作った言語で記した証明なのですね．

羅茶：そうです．実引数が求められる条件を満たすことの証明を，人工言語で記して，それをもう 1 つの引数として渡してやります．

苦迦：あっ，そうか．実引数が条件を満たすことを証明するのは，一般には決定不能になるかもしれませんが，実引数とその証明が与えられたときに，証明が妥当なものかどうかを調べるのは，ほとんどパターン照合で，高速に検査できますね．答えを求めることは大変だが，答え合わせは簡単にできる，というわけか．

羅茶：その通りです．

苦迦：このような検査が，始めにおっしゃっていた，プログラミングへのより強力な支援，ということなのですね．

ところで，ここではデータの処理と命題の処理を分けて考えていますが，命題の処理もデータの処理も，計算という意味では同じことではないのですか？

羅茶：どういうことでしょうか？

苦迦：つまり，計算は，函数によって表現できますが，命題も，真か偽かを返す特殊な函数と見ることができますよね．

羅茶：おっしゃる通りです．命題というのは，真あるいは偽，どちらかの値を返す特別な函数と見ることができますね．その意味で，命題を扱うことも，函数を扱うことも高階の対象を扱うことと言えます．しかし，ここでは，特に命題という種類の函数，これは一種の抽象データ型と考えることもできますが，そのようなものを特に扱うことに意味があるのではないか，という話をしたいのです．

苦迦：ははあ，なるほど．ところで，命題を抽象データ型と考える，というのはどういうことなのでしょうか．

羅茶：函数型言語では，函数をモノとして扱っているのはご存知の通りです．そこでは，函数を値とするデータ型があります．同様に，命題を値とするデータ型を定義することができます．先ほどの言い方を借りれば，List 型とか，Queue 型がいろいろ役に立っていると思いますが，これと同じように，命題型を導入しよう，というわけなのです．

苦迦：List 型などと同じように，命題型を導入するわけですね．

羅茶：命題だけでなく，一般に論理式をデータ型にすることができます．命題は自由変数を含まない論理式，特別な論理式と考えることができますので．そうすると，論理式そのものだけでなく，その真偽，また先ほどでてきた論理式の証明データ型，といったものが一緒に出てきます．そこで，それらをまとめて一種の抽象データ型を定義することができる，というわけです．

苦迦：なるほど．そして，始めにおっしゃっていた命題の処理がここにでてきますね．

羅茶：はい．命題を扱う多くの場合，自動証明ができればそれに越したことはないのですが，それが簡単にできない場合でも，人間が証明を与えることにして，ある程度の支援が可能になる可能性があります．あなたもおっしゃったように，自動証明は一般には不可能だけれど，与えた証明が正しいかどうかの検査は，証明の長さに対して線形時間で解ける問題ですから．

苦迦：すると，Prologのような，いわゆる論理型言語のことは，今のお話の文脈では，どのように考えたらいいでしょう？

羅茶：Prologの指導原理は，ホーン節という特別な形をした論理式が与えられたときに，それを真にする解釈を見つける，という操作によってチューリング機械を模倣することができる，というものです．したがって命題の真偽だけではなく，論理式の形も関係しますので，先ほどよりも話は複雑になりますが，命題あるいは論理式の処理が行われている，という意味ではPrologは，今お話したいこと，つまり命題の処理の１つの例になっています．

苦迦：ふむふむ．

羅茶：しかし，Prologでは解釈が全自動で見つかるわけで，逆に言うと，取り扱う論理式を，全自動で解釈が見つかるようなものに限定しているわけです．しかし，先ほどのように人間が証明を与えたり解釈を与えたりしてやる，あるいは答えを全部人間が与えないまでも，計算機による処理の要所，要所で人間が知恵をつけてやれるような枠組みに興味があるのですよ．

苦迦：なるほど，必ずしも自動証明を目指すわけではなく，できない場合でも計算機による処理があり得るというわけなのですね．

羅茶：はい．ここで大事なのは，証明の自動的な検査を計算機にさせるためには，証明もデータ化する必要がある，ということだと思います．この『データ化』という操作，言い換えると，『記述』が計算機による処理に本質的に重要だと思うのです．記述の対象はプログラムだけではないのはもちろんです．事務文書や法律，契約書などでも，計算機による自動処理を行うための記述があります．計算機による処理を可能にするときに記述という操作の側面から一般的に考察していく必要があるのではないか，と思うのです．

苦迦：なるほど．記述の科学というゆえんですね．

プログラムの仕様

プログラミング言語では，算法(アルゴリズム)や，算法が取り扱うデータの構造(データ型)を表現したが，算法やデータの持つ性質を直接表すことはあまりなかった．もちろん，データ型を指定することは，データ構造の性質を規定しているし，算法を指定することはおのずから，その性質の指定につながる．しかし，これらの仕方では指定できないような性質がたくさんある．仕様記述や抽象データ型の考えが現れるとともに，データの性質で，これまでのプログラミング言語では指定できないようなものも指定できると便利だ，という考えがでてきた．

仕様記述の考え方は，プログラムを直接書くのではなく，まずプログラムに何をさせたいのかを書こう，というものである．何かをコンピュータにさせたいとき，それを行うプログラムは１つとは限らない．また，させたいことを行うプログラムは，たいていの場合ほかのいろいろなことも副作用として行う．そこで，当面注目したいプログラムの部分(函数やデータ)の名前と，それに要求することがら，

つまり，何に何をさせたいのかをまず書き下ろし，それさえ行ってくれれば，ほかのことはしなくてもよいし，余計なことをしても（させたいことさえしてくれれば）よろしい，と考えるのが，仕様記述の考えである．何をさせたいか，は指定するが，それを，どのようにして行うのか，までは指定しない考え方，ということもできるだろう．さらに言い換えると，させたいことを行ってくれるプログラムがいくつかあったとしても，そのうちのどれか１つがあればよく，どれを選ぶかまでは指定しない，という考え方だと言ってもよい．

ここで，「何に何をさせたいのか」を記したものが仕様で，それを行うものが実現である．仕様と実現の考えで大事なのは，１つの仕様を実現するものは，一般にはいくつもあり得る，という事実である．仕様には何に何をさせたいかを書く．このうち「何に」にあたるところの指定は簡単である．名前さえ決めればよい．しかし，「何をさせたいか」の指定をプログラミング言語で書くのは難しい場合がある．

見方を変えると，プログラミング言語における文，つまりプログラムは，算法を１つ記すものであった．しかし仕様は，させたいことをしてくれるプログラムをいくつか（0 個，有限個，無数個）まとめて記すものである．だから，プログラミング言語によって仕様を記すことには無理があるわけである．

さて，プログラムの仕様は，プログラムを１つだけ指定するとは限らず，一般にはプログラムの集まりを規定するものであった．仕様の記述とプログラムの記述はそこが違う．一般に，ものの集まりを指定する方法として広く用いられるのは，ものがその集まりに属するための必要十分条件を指定することである．さらに何か１つの述語を与えるとそれによって１つの集まりが決まる，と素朴には考えられる．ここで注意しないと，すべての集合の集合を指定することになって，有名なパラドックスに陥ってしまうけれども，大筋では述語＝集まりと考えてよい．つまり仕様を記すのはプログラムについての述語を指定すること，と考えられる．プログラムを指定するのがプログラム言語だとすると，プログラムの仕様記述の言語は，プログラムについての述語を指定する．

苦迦：プログラムの性質を表すものとして，仕様記述や抽象データ型というものがあるのは，よく分かります．ところで，どうして仕様記述の話が出てきたのでしたっけ？

羅茶：さっきの，ライブラリの利用条件の話では，データの性質を指定して，実引数として渡してもよいデータを制限する，という話題が出てきました．そこではデータの性質がすなわちデータの集まりでした．で，データの話からプログラムの話に目を移してみたのです．

苦迦：なるほど，データの性質にあたるのは，プログラムの性質，つまりプログラムの集まりですが，それを記すのは仕様記述にほかなりませんね．

羅茶：ココロは，どちらも集まりが出てくる，というわけです．

苦迦：寄席のなぞかけじゃあるまいし．まあよろしい．仕様記述などは述語として表されることもわかりました．ただ，前には，命題によって性質を表す，という話でしたが，今度は命題じゃなく述語がでてきましたね．命題と述語はどう違うのですか？

羅茶：命題は，それ自体で完結しているもの，つまり命題だけを持ってきても真偽が決められるものです．述語は，主語と対で語られるのが普通です．このことから分かるように，述語にはその主語を指定しないと真偽が決まりません．つまり述語は P(x) の形をしていて，x に具体的に何かを入れることによって真偽が決まります．もう一歩進んで，述語は，それを真にするようなものの集まりと考えてもよいわけです．

苦迦：なるほど．すると，正確を期すとすれば，ライブラリの引数の条件も，命題ではなくてデータに関する述語と言うべきなのですね．

羅茶：まったくそのとおりです．

プログラムのライブラリに渡す引数に関する条件

を，より詳細に指定することによって，より強力なライブラリの構成を導く可能性があること，また，プログラムの仕様は，結局のところ，プログラムに関する述語の指定に帰着することなどを考察した．いずれの話題にも，基本的なモノ（ライブラリに渡すデータやプログラム）からできる集まりを述語で指定する，という筋書きがあった．

述語が出現するのはプログラミングの世界に限るわけではなく，いろいろな世界で述語が現れる．昨今，食の安全などに関連して求められるトレーサビリティ（traceability）の管理では，「このものは，どこそこで作られた」などという，データの出所を表す述語が基本的に重要だし，書類の決裁では，その書類が，誰それによって（課長によって，部長によって，あるいは社長によって）承認済みであるという述語が本質的である．法律や条例をはじめとする規則も述語をたくさん集めたものと見ることができる．

苦迦：たとえば，法律や規則ではどう述語と見ることができるのでしょうか．規則そのものがデータで，ある規則に従っている，というのが述語かな．

羅茶：いろいろな見方があると思います．たとえば，法律の場合は，1つ1つの条文が述語になり得ます．

苦迦：たしかにそうですね．

羅茶：ただここで，何でも述語と見なせるということを言いたいのではなく，目的に応じて述語をたてていくということが重要です．

苦迦：目的って何の目的ですか？

羅茶：計算機による自動処理の目的です．たとえば，法律の場合は，法律自体の整合性の検査のためであれば，条文1つ1つの述語を用意すれば十分だろうし，ある行動が法律に従っているかどうかを示すためには，Bさんが言った，ある規則に従っている，という述語が必要です．

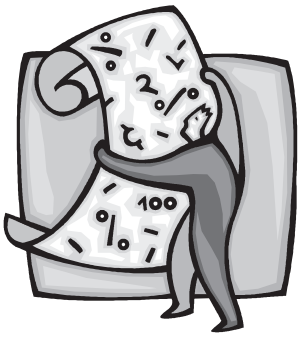
苦迦：観点というか視点を明確にすることが必要ということですね．視点によって，必要な語彙も変わってくるわけだ．

羅茶：はい，その通りです．次回は，その視点をどう表すか，また複数の視点の間の関係などについて，考えていきましょう．

苦迦：にわかに大問題になってきましたね．楽しみにしています．

今回の2人の会話によって，記述というものに対して，彼らがどんな問題意識を持っているのかが明らかになった．次回はいよいよ，本連載の主題の1つ，記述の形式に議論がすすむ．何らかの形式があってこそ記述が可能になる．彼らの言う視点，観点こそ，その形式である．

（平成22年6月1日受付）



木下佳樹（正会員）
yoshiki@m.aist.go.jp

平成元年東京大学大学院理学系研究科博士課程情報科学専攻修了．理学博士（情報科学）．テキサスインスツルメンツ，産業技術総合研究所システム検証研究センター長等を経て現在，同組込みシステム技術連携研究体主幹研究員．

高井利憲（正会員）
t-takai@aist.go.jp

平成13年奈良先端科学技術大学院大学博士後期課程単位取得認定退学．博士（工学）．科学技術振興機構CREST研究員等を経て現在産業技術総合研究所組込みシステム技術連携研究体研究員．