境界を越える: Lisp の美しさ

プログラミング言語の黄金郷

Bruce Tate

2007年 2月 06日

Lisp は長年の間、偉大なプログラミング言語の 1 つと考えられてきました。その長い歴史 (ほぼ 50 年) をとおして多くの熱狂的な信奉者を生んだという事実は、Lisp が何か特別なものであることを物語っています。MIT では、すべてのプログラマーに対するカリキュラムの中で、Lisp が基本的な役割を担っています。Paul Graham などの起業家は、新規事業を成功へと推進する力として、Lisp の持つ信じられないほどの生産性を活用しました。しかし Lisp の信奉者にとっては残念なことに、Lisp が主流になることはありませんでした。皆さんも Java™ プログラマーとして、この失われた黄金郷である Lisp を少しばかり研究してみると、コーディング方法をより良い方向に変えるための多くのヒントを見つけられるはずです。

このシリーズの他の記事を見る

私は最近、初めてマラソンを完走し、走ることが想像以上に報いの多いものであることを知りました。私は一歩進むという単純な動作を、26.2 マイルを走るという、人体にとって一種特別なことに変化させたのです。Smalltalk や Lisp のような一部の言語も、これと似ているように思います。Smalltalk の場合、一歩という基本に当たるのはオブジェクトです。つまり Smalltalk では、すべてがオブジェクトとメッセージ交換を扱います。Lisp では、この基本となる一歩がさらに単純です。この言語は、完全にリストで構成されています。しかし、この単純さに惑わされてはいけません。この、登場して 48 年になる言語には、Java 言語が太刀打ちできないほどの恐るべきパワーと柔軟性が備わっているのです。

私が最初に Lisp に出会ったのは大学生の時でしたが、その出会いは順調なものではありませんでした。私は、Lisp の関数型構造を利用しようとせず、当時私が知っていたプロシージャー型の枠組みの中に Lisp を押し込もうと必死で Lisp と格闘したのです。Lisp は厳密には関数型言語ではありません。Lisp の機能のいくつかを見ると、Lisp が厳密な意味での関数型ではないことがわかります。しかし Lisp のイディオムや機能の多くは、関数型の素質を強く感じさせます。それ以来私は、リストと関数型プログラミングを積極的に受け入れることを学んだのです。

このシリーズについて

この、境界を越えるシリーズでは、著者である Bruce Tate が、「今日の Java プログラマーは、他の手法や言語を学ぶことから多くを得ることができる」という概念を押し進めます。

プログラミングの世界の様相は、あらゆる開発プロジェクトにとって Java が最善の選択肢であることが明確だった頃から変わってきています。他のフレームワークも Java フレームワークと同じ構築形態をとりつつあり、また他の言語での概念を学ぶことによって、それを Java プログラミングに生かすこともできます。皆さんが書く Python (あるいは Ruby や Smalltalk、その他何であれ) コードによって、皆さんの Java コーディングに対する取り組みも変わる可能性があるのです。

このシリーズでは、Java 開発とは大幅に異なりながら、同時に Java 開発にも直接応用できるプログラミング概念や手法について紹介します。場合によると、そうした技術を利用するためには、それら統合する必要があるかも知れません。また場合によると、そうした概念は直接利用できるものかもしれません。他の言語やフレームワークが、開発者やフレームワーク、Java コミュニティーでの基本的な手法にまで与えうる影響に比べると、個々のツールはそれほど重要ではありません。

今回の「境界を越える」では、この失われた宝を取り出します。最初に Lisp の基本的な構成体を簡単に紹介し、それから急速にスピードを上げ、ラムダ式や再帰、マクロなどを見ていきます。この短いツアーだけでも、Lisp の生産性と柔軟性を十分に理解できるはずです。

はじめに

この記事では、多くのオペレーティング・システム用に無料のダウンロードがある、GNU の GCL を使います。しかし、どんなバージョンの Common Lisp であっても、少しいじれば使えるはずです。利用可能な Lisp のバージョンに関しては、「参考文献」を参照してください。

他の大部分の言語と同様、Lisp を学ぶための最良の方法は、実際にいじってみることです。インタープリターを開き、私と一緒にコーディングしてみてください。Lisp は基本的にコンパイル言語なので、単純にコマンドを入力するだけで容易に試すことができます。

リストの言語

基本的に、Lisp はリストの言語です。Lisp では、データを始めアプリケーションを構成するコードまで、すべてがリストです。各リストは、アトムとリストで構成されます。数字はアトムです。数字を入力すると、単純にその数字が結果として返されます。

リスト 1. 単純なアトム

```
>1
1
>a
Error: The variable A is unbound.
```

もし文字を入力すると、インタープリターはエラーを出力します (リスト 1)。文字は変数なので、使う前に値を割り当てる必要があります。変数ではなく文字や単語を参照したい場合には、引用符で囲みます。変数の前に単一引用符を置くと、それに続くリストまたはアトムの評価を遅らせるように Lisp に命令していることになります (リスト 2)。

リスト 2. 評価の遅延と引用符

```
>"a"
"a"
>'a
A
```

Lisp が A を大文字にしていることに注目してください。A が引用符で囲まれていないため、Lisp は、A がシンボルとして使われるのだと想定します。後で割り当てを説明しますが、その前にリストについて説明する必要があります。Lisp のリストは単純に、括弧で囲まれ空白で区切られたアトムのシーケンスです。リスト 3 のようなリストを入力してみてください。リストの前に ' を置かない限り、このリストは動作しません。

リスト 3. 単純なリストを入力する

```
>(1 2 3)
Error: 1 is invalid as a function.
>'(1 2 3)
(1 2 3)
```

リストの前に ' を置かない限り、Lisp は各リストを関数として評価します。最初のアトムは演算子であり、リストの中の残りのアトムは引数です。Lisp には、多くの数学関数 (たとえば皆さんが想像されるような + や *、sqrt など) を含め、数々の基本関数があります。 (+ 1 2 3) は 6 を返し、(* 1 2 3 4) は 24 を返します。

リストを操作する関数には、コンストラクターとセレクターという 2 つのタイプがあります。コンストラクターはリストを作成し、セレクターはリストを分解します。中心となるセレクターは、first と rest です。first セレクターはリストの最初のアトムを返し、rest セレクターは、最初のアトム以外の、リスト全体を返します。リスト 4 は、これらのセレクターを示しています。

リスト 4. 基本的な Lisp 関数

```
> (first '(lions tigers bears))
LIONS

> (rest '(lions tigers bears))
(TIGERS BEARS)
```

どちらのセレクターも、リスト全体を調べ、リストの中の戦略的な部分を返します。後ほど、こうしたセレクターが再帰で利用されることを説明します。

リストを分解するのではなく作成したい場合には、コンストラクターが必要です。Java 言語の場合と同じく、コンストラクターは新しい要素を作成します。Java 言語ではオブジェクトが要素ですが、Lisp ではリストが要素です。cons や list、append などはコンストラクターの一例です。コア・コンストラクターである cons は、アトムとリストという、2 つの引数をとります。cons は、リストの最初の要素としてアトムをリストに追加します。nil に対して cons をコールすると、Lisp は nil を空のリストとして扱い、1 つの要素から成るリストを作成します。append は 2 つのリストを連結します。list は引数すべてのリストを含んでいます。リスト 5 は、こうしたコンストラクターの実際を示しています。

リスト 5. コンストラクターを使う

```
> (cons 'lions '(tigers bears))
(LIONS TIGERS BEARS)

> (list 'lions 'tigers 'bears)
(LIONS TIGERS BEARS)

> (append '(lions) '(tigers bears))
(LIONS TIGERS BEARS)
```

cons を first と rest と組み合わせて使うと、任意のリストを作成することができます。list 演算子と apend 演算子はちょっと便利な性質をもっているだけですが、頻繁に使われます。実際、cons と first、rest を使うことで、任意のリストを作成する、あるいは任意のリスト・フラグメントを返すことができます。例えば、リストの中の 2 番目、あるいは 3 番目の要素を取得するには、rest の first を取得するか、あるいは rest の rest のfirst を取得します (リスト 6)。あるいは、2 つまたは 3 つの要素から成るリストを作成するには、 cons を first と rest と組み合わせて使うことで、list と append をシミュレートすることができます。

リスト 6. 2 番目、3 番目の list と append を作成する

```
>(first (rest '(1 2 3)))
2

>(first (rest (rest '(1 2 3))))
3

>(cons '1 (cons '2 nil))
(1 2)

>(cons '1 (cons '2 (cons '3 nil)))
(1 2 3)

>(cons (first '(1)) '(2 3))
(1 2 3)
```

これらの例を見ても興奮することはないかもしれませんが、こうした単純なプリミティブからクリーンできれいな言語を構築できるという、この原理が、一部のプログラマーを魅了するのです。こうした、リスト作成のための単純な命令が、再帰や高階関数、さらにはクロージャーや継続など高次の抽象化の基礎となっています。では次に、高次の抽象化の説明に移りましょう。

関数を作成する

Lisp の関数宣言は、ご想像のとおり、リストです。リスト 7 はリストの 2 番目の要素を返す関数を作成するものですが、このリストは関数宣言の形式を示しています。

リスト 7. 2 番目の関数を作成する

```
(defun my_second (lst)
  (first (rest lst))
)
```

defun は、カスタム関数を定義する関数です。最初の引数は関数名、2 番目は引数リスト、そして 3 番目が実行対象となるコードです。Lisp コードがすべてリストとして表現されていることがわかると思います。この柔軟性と強力さを利用すれば、アプリケーションを操作できるだけではなく、他の任意のデータも操作することができます。後ほど、コードとデータの区別があいまいになる例をいくつか見ることにします。

Lisp は、if 文のような条件構成体も処理することができます。その形式は、(if condition_statement then_statement else_statement) です。リスト 8 は、2 つの入力変数の最大値を計算する単純な my_max 関数を示しています。

リスト 8. 2 つの整数の最大値を計算する

```
(defun my_max (x y)
  (if (> x y) x y)
)

MY_MAX
(my_max 2 5)

5
(my_max 6 1)

6
```

ここまでのところを整理してみましょう。

- Lisp は、リストとアトムを使ってデータとプログラムの両方を表現します。
- リストを評価する場合には、最初の要素をリスト関数として扱い、他の要素を関数の引数として扱います。
- Lisp の条件文は、コードと合わせて真と偽による表現を使います。

再帰

Lisp には繰り返し用のコード構造がありますが、リストを扱う上では、再帰の方がはるかに一般的です。first と rest の組み合わせは、再帰でも問題なく使えます。リスト 9 の total 関数を見ると、この関数の動作がわかります。

リスト 9. 再帰を使ってリストの合計を計算する

```
(defun total2 (lst)
(labels ((total-rec (tot lst)
(if lst (total-rec (+ tot (first lst)) (rest lst)) tot)))
(total-rec 0 lst)))
```

リスト 9 の total 関数は、リストを 1 つの引数として扱います。最初の if 文は、リストが空の場合には再帰を停止し、ゼロを返します。空ではない場合、total 関数は最初の要素をリストの残りの要素の合計に加えます。これを見ると、なぜ first と rest が、正に最初と残りとして作られているのかがわかります。つまり first はリストの最初の要素を取り出し、また rest によって残り部分に対する末尾再帰 (リスト 9 のタイプの再帰) が行いやすくなります。

Java 言語での再帰は、パフォーマンスの問題から制限があります。一方 Lisp には、末尾再帰最適化と呼ばれるパフォーマンス最適化が用意されています。Lisp コンパイラー、あるいは Lisp インタープリターは、ある形式の再帰を繰り返しに変換できるため、これを利用すると再帰的なデータ構造 (ツリーなど) をより単純でクリーンな方法で処理することができます。

高階関数

データとコードの区別があいまいになると、Lisp は一層興味深いものになります。このシリーズの前回と前々回では、JavaScript での高階関数と Ruby でのクロージャーを見てきました。これらの機能はどちらも、関数を引数として渡します。Lisp では、関数は他のどんな種類のリストとも何ら違いがないため、高階関数は重要なものではありません。

高階関数の最も一般的な使い方はラムダ式でしょう。ラムダ式はクロージャーの Lisp 版です。ラムダ関数は関数定義であり、これを使って高階関数を Lisp 関数に渡します。例えば、リスト 10 のラムダ式は 2 つの整数の和を計算します。

リスト 10. ラムダ式

```
>(setf total '(lambda (a b) (+ a b)))
(LAMBDA (A B) (+ A B))

>total
(LAMBDA (A B) (+ A B))

>(apply total '(101 102))
203
```

クロージャーあるいは高階関数を使ったことのある人であれば、リスト 10 のコードをよく理解できるかもしれません。1 行目はラムダ式を定義し、それをシンボル total にバインドします。2 行目は、total にバインドされたラムダ式を単純に表示します。そして最後に、最後の式は、このラムダ式を (101 102) を含むリストに適用します。

高階関数は、オブジェクト指向の概念よりも高次の抽象化を実現します。高階関数を使うと、より簡潔に、より明確に概念を表現することができます。プログラミングの大きな目標は、読みやすさやパフォーマンスを犠牲にすることなく、より少ないコード行で、より大きな力を発揮し、より高い柔軟性を実現することです。高階関数によって、そのすべてが現実となるのです。

まとめ

登場からの年数を見れば Lisp は古く、その構文も古めかしいものです。しかし、少し掘り下げてみると、Lisp が高度の抽象化を備えた信じられないほど強力な言語であり、その誕生時の 50 年前と同じくらい有効で生産的なことがわかります。Lisp よりも新しい多くの言語が Lisp の概念を借用しており、それらの多くは、まだ Lisp に匹敵できるほどの能力を提供できていません。Java や .NET の何分の一かのマーケティングが Lisp に関して行われ、多くの大学でも MIT と同じように積極的に教えられていれば、私達全員は今でも Lisp を書いているかもしれないのです。

関連トピック

- 『Beyond Java』(2005年、O'Reilly刊) は、この記事の著者による本です。Java 言語の台頭と停滞について、また、一部のニッチな領域で Java プラットフォームに対抗しうる技術について解説しています。
- GNU Common Lisp は一般的な Lisp 実装の 1 つであり、その Lisp インタープリターがこの記事で使われています。
- Carl de Marcken: Inside Orbitz は、実際に使用されている Lisp 機能について議論しており、また現実の世界で Lisp で何ができるかを紹介しています。
- 『計算機プログラムの構造と解釈』(Harold Abelson らの共著、1996 年、(原書: McGraw-Hill 刊、邦訳: ピアソン刊)) は時代を越えた古典であり、Lisp の哲学を素早く学ぶことができます。
- Association of Lisp Users は、Lisp コミュニティーをサポートする国際的な組織です。
- The Common Lisp Directory は、Lisp と Lisp 関連のリソースを豊富に提供する素晴らしいサイトです。ここには、Common Lisp で実装される無料ソフトウェアや他の ALU Lisp リソースにリンクする CLiki もあります。
- Java technology ゾーンには Java プログラミングのあらゆる側面を網羅した記事が豊富に用意されています。
- Common Lisp Implementations には、商用および無料の Common Lisp 実装がリストされています。