プログラミング言語論2012 # 2 —
構文と意味/記号処理とLisp

久野　靖*

2012.4.19

1　構文と意味

1.1　構文定義とその役割

通常、プログラミング言語を定義する際には通常、その構文を BNF(Backus Naur Form、ないし Backus Normal Form) で定義します。[1]たとえば、次のような具合です (ちなみに、ϵ は「何もない」空列を意味しています)。

```
文　::=　代入文　|　IF 文　|　WHILE 文　|　{　文列　}
文列　::=　文　|　文列　文
代入文　::=　変数　=　式 ;
IF 文　::=　if(　式　)　文　ELSE 部
ELSE 部　::=　ϵ　|　else　文
WHILE 文　::=　while(　式　)　文
```

さて、このような構文定義はどのような役割を果たしているのでしょう。まず思い付くのは、この構文定義に合致するようなコードだけが、その言語の正しいプログラムとして受け入れられる可能性がある、という点です。[2]つまり「形が合っているかどうか」は構文定義に照らして定まります。誰もが「syntax error」で悩まされた経験をお持ちだと思います。

しかしもちろん、プログラムの目的は「計算を記述する」ことであり、そのとき「構文に合った記述だけが許される」ということは単に形を限定する、という以上の意味があります。それはつまり「ある形の構文に合致する記述は、その構文に対応する意味を持つ」ということです。

具体的にはたとえば、次の記述があったとします。

```
if(x > y) { z = x; } else { z = y; }
```

これは if 文にあてはまるので、まず条件部の式を評価し、そしてそれが真なら then 部の文を、そうでなければ (かつ else 部があれば)else 部の文を実行する、という「意味」を持ち、そのように実行される (処理系がコンパイラであればそのような動作をするように翻訳される)、ということになります。

このように、構文に基づいて動作ないし意味が決まることを「構文主導」(syntax-directed) と呼び、Alogol-60 以後、今日の多くの言語ではこれが普通になっています。それに限らない変わったものが

*経営システム科学専攻

[1]BNF で定義するというのは、文脈自由文法を与えるというのと同じことですが、文脈自由文法の説明を始めると長くなるので、ここではその説明は省略します。なお、FORTRAN や COBOL は BNF の発明以前にできた言語なので、そのような構文定義を持ちません。だから今見るとヘンな言語だなと思います。

[2]もちろん、形が合っていてもさらに別の面で間違っていて受け入れられない可能性はあります。しかし形が合っていなければ、それだけで拒否されるわけです。

より言語としてより良い特性を持つ、という可能性も無くはないのですが、現在のところ、この考え方が人間にとって素直で分かりやすい言語につながっている、ということは (実証的に) 否めないように思えます。

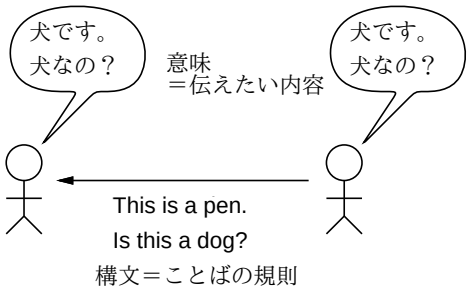

図 1: 構文と意味の対応づけ

実はこのことは、自然言語においてもある程度あてはまります。自然言語の文法でも、ある特定の形 (たとえば英語の BE 動詞の文なら、動詞と主語の入れ替え) によって特定の意味 (疑問) を表す、というふうに、構文と意味が対応しているのはよくあることです。

ただし、日本語みたいにあまり語順などに重みのない言語では、むしろ特定のフレーズの有無「例: ～なんですか?」だけで疑問の有無が決まったりするかも知れません。しかしそのような言語では、必然的にボキャブラリ (とりわけ動詞や形容詞や副詞) の役割が大きくなります。このような「違い」はプログラミング言語の設計にもある程度あり得ると考えます (そのような言語の例はまた別の回に)。

1.2 ラムダ計算

チューリング完全であることが示されている計算モデルの 1 つに、Alonzo Church によって提唱されたラムダ計算 (lambda calculus) があります。ラムダ計算は関数 (ラムダ式) とその適用に基づく手順の表記法であり、関数型プログラミング言語となぞらえて理解しやすいです。また型付きラムダ計算など型理論の土台としても多く参照されます。

ラムダ計算の特色として、「構文の種類が少ない」ということが挙げられます。具体的には、ラムダ計算の「プログラム」は次のように変数、ラムダ式、適用の 3 つの書き方だけしか使用しません。

$$
\begin{array}{ll}
t ::= x & \text{(変数)} \\
\quad \mid \lambda x\ .\ t & \text{(ラムダ式)} \\
\quad \mid t\ t & \text{(適用)}
\end{array}
$$

ラムダ式は抽象化 (いわゆる関数) を表し、適用は関数に引数を与えて実行することを表します (このほかに順序を表すかっこを適宜用いる)。その意味するところは、λ の次に記された変数が引数であり、適用時には $.$ の後ろに書かれた項の中の引数を適用時に渡された式で置き換えること (β 簡約と呼びます) を意味します。β 簡約の例を挙げておきます。

$$(\lambda x\ .\ x\ y)\ z = z\ y$$

ところで、上の例の左辺に現れていたラムダ式のパラメタ名を別のものに変えても意味は変わらないでしょうか。

$$(\lambda t\ .\ t\ y)\ z = z\ y$$

一見良さそうですが、たとえば次のようにしたらまずいですね。

$$(\lambda y\ .\ y\ y)\ z = y\ y$$

何が問題かというと、変数 y は元のラムダ式の中では何にも束縛されておらず、外から自由に意味が与えられる状況にある (これを自由変数と呼びます) ものだったのに、それをラムダ式のパラメタに変えてしまったら意味が変わってしまう、ということです。つまり、λ 式のパラメタ名を別のものに置き換える (α 変換と呼びます) ときには、(1) 本体内に自由に出現する変数名をパラメタ名にしない、(2) 逆に、本体内に自由に出現しているパラメタ名のみを新たな名前に置き換える、という 2 点を守る必要があります。このように、自由変数のを持つラムダ式は環境まで考慮する必要があって煩わしいので、自由変数を持たないラムダ式 (これを「コンビネータ」と呼びます) のみを扱う理論も発達しています。

ラムダ計算の興味深いところは、上で定義した項だけで計算と計算対象となるデータの両方を表せる点です。これは、プログラムもまたデータであり、プログラムによって加工できるという、現代のコンピュータがもつ特性にそのまま対応していると言えます。

たとえば自然数の集合やその加算をラムダ式だけで表現し定義する様子を見てみることにします。まず、自然数を次のような形の項によって表すものとします (これをチャーチ数と呼びます)。

$$
\begin{aligned}
c_0 &= \lambda s\,.\,\lambda z\,.\,z \\
c_1 &= \lambda s\,.\,\lambda z\,.\,s\ z \\
c_2 &= \lambda s\,.\,\lambda z\,.\,s\ (s\ z) \\
c_3 &= \lambda s\,.\,\lambda z\,.\,s\ (s\ (s\ z)) \\
&\cdots
\end{aligned}
$$

このとき、加算を行うラムダ式 $plus$ は次のように定義すればよいのです。

$$
\mathrm{plus} = \lambda m\,.\,\lambda n\,.\,\lambda s\,.\,\lambda z\,.\,m\ s\ (n\ s\ z)
$$

これに対してチャーチ数の 1 と 2 を与えて適用した様子を次に示します。

$$
\begin{aligned}
&\mathrm{plus}\ c_1\ c_2 \\
&\quad = \lambda m\,.\,\lambda n\,.\,\lambda s\,.\,\lambda z\,.\,m\ s\ (n\ s\ z)\ (\lambda s\,.\,\lambda z\,.\,s\ z)\ (\lambda s\,.\,\lambda z\,.\,s\ (s\ z)) \\
&\quad = \lambda s\,.\,\lambda z\,.\,(\lambda s\,.\,\lambda z\,.\,s\ z)\ s\ ((\lambda s\,.\,\lambda z\,.\,s\ (s\ z))\ s\ z) \\
&\quad = \lambda s\,.\,\lambda z\,.\,(\lambda s\,.\,\lambda z\,.\,s\ z)\ s\ (s\ (s\ z)) \\
&\quad = \lambda s\,.\,\lambda z\,.\,s\ (s\ (s\ z)) \\
&\quad = c_3
\end{aligned}
$$

確かに、1 と 2 を足した結果として 3 が得られていますね。

また、論理値とそれに基づく枝分かれも同様にラムダ式で定義できます。まず、真と偽を次の項によって表現します (これをチャーチ論理値と呼びます)。

$$
\begin{aligned}
\mathrm{true} &= \lambda t\,.\,\lambda f\,.\,t \\
\mathrm{false} &= \lambda t\,.\,\lambda f\,.\,f
\end{aligned}
$$

次に、真偽値とさらに 2 つの引数を受け取り、真偽値が真なら 2 つの引数の前者、偽なら後者を値とするラムダ式 $test$(if-then-else 演算子に相当) は次のように定義できます。

$$
\mathrm{test} = \lambda b\,.\,\lambda y\,.\,\lambda n\,.\,b\ y\ n
$$

実際に真ないし偽と 2 つの引数 p、q を与えた様子を示します。

$$
\begin{aligned}
&\mathrm{test\ true\ p\ q} \\
&\quad = (\lambda b\,.\,\lambda y\,.\,\lambda n\,.\,b\ y\ n)\ (\lambda t\,.\,\lambda f\,.\,t)\ \mathrm{p\ q} \\
&\quad = (\lambda t\,.\,\lambda f\,.\,t)\ \mathrm{p\ q} \\
&\quad = \mathrm{p} \\
&\mathrm{test\ false\ p\ q} \\
&\quad = (\lambda b\,.\,\lambda y\,.\,\lambda n\,.\,b\ y\ n)\ (\lambda t\,.\,\lambda f\,.\,f)\ \mathrm{p\ q} \\
&\quad = (\lambda t\,.\,\lambda f\,.\,f)\ \mathrm{p\ q} \\
&\quad = \mathrm{q}
\end{aligned}
$$

このように、数と枝分かれまではラムダ式の項によって素直に表現できることが分かります。次に繰り返しについては、再帰を用いることで表現します。

無限の再帰は、たとえば次のラムダ式で表現できます。

$$\mathrm{Y} \;=\; \lambda f\,.\,(\lambda x\,.\,f\;(x\;x))\;(\lambda x\,.\,f\;(x\;x))$$

これは名前渡しの Y コンビネータと呼ばれ、次のように任意の関数を繰り返し適用させられます。

$$\begin{aligned}
&\mathrm{Y\ g}\\
&\quad = (\lambda f\,.\,(\lambda x\,.\,f\;(x\;x))\;(\lambda x\,.\,f\;(x\;x)))\;\mathrm{g}\\
&\quad = (\lambda x\,.\,\mathrm{g}\;(x\;x))\;(\lambda x\,.\,\mathrm{g}\;(x\;x)))\\
&\quad = \mathrm{g}\;(\lambda x\,.\,\mathrm{g}\;(x\;x))\;(\lambda x\,.\,\mathrm{g}\;(x\;x)))\\
&\quad = \mathrm{g}\;(\mathrm{Y\ g})
\end{aligned}$$

ただしこれが使えるのは、評価の順序が名前渡しの場合に限られ。値渡しだと (Y g) の部分が展開されてしまうので発散しまする。このため「名前渡しの」という限定がついて呼ばれるわけです。値渡しの Y コンビネータは次のようにもう少し複雑になります。

$$\mathrm{Y'} \;=\; \lambda f\,.\,(\lambda x\,.\,f\;(\lambda y\,.\,x\;x\;y))\;(\lambda x\,.\,f\;(\lambda y\,.\,x\;x\;y))$$

ラムダ計算そのものは理論的な検討のためのものですが、これに基づいて Lisp 系の言語やその他の関数型のプログラミング言語が多数作られています。

プログラミング言語ではもちろん、数値やその演算、論理値やそれに基づく枝分かれなどは予め組み込みの機能として備わっています。また、再帰についても、名前をつけた関数を定義し、その本体で名前を用いて自分自身を参照することで、Y コンビネータのようなものを使用しなくても記述することができます。さらに、Lisp など副作用を忌避しない言語では再帰を使用しなくても直接的に繰り返しを実現する機能を備えていることが普通なわけです。

2 Lisp 入門

2.1 Lisp 言語の由来と特徴

Lisp 言語は John McCarthy によって 1958 年に考案された、世界で 2 番目の (1 番目は FORTRAN) 高水準言語であり、ラムダ計算にヒントを得て「少数の基本構造だけでプログラムを組み立てる」ことが基盤にありました。また、リスト処理、記号処理などの考え方を取り入れることで (これらはそれ以前にあった IPL と呼ばれる言語から来ているらしいです)、柔軟なデータの扱いを得意とし、人工知能分野で広く使われるようになりました。

McCarthy の Lisp (Lisp-1、Lisp-1.5) の成功により、多くの Lisp 処理系が作られるようになりましたが、それぞれに独自の拡張を行ったため、多様な言語仕様を持つ「Lisp 属」が形成されました。これらの流派は最終的に統合されて CommonLisp となりましたが、CommonLisp とは別の方向性をめざした Lisp 属の言語は現在でも多数あります (Scheme、Clojure など)。

Lisp の特徴としては、次のものがあげられます。

- プログラムとデータをともに「S 式」と呼ばれる共通の記法で表現している。これにより、プログラムを生成し実行するプログラムが容易に記述できる。
- S 式は記号 (symbol) と呼ばれるデータ要素とその並びを中心として構成されており、他の手続き型言語のような面倒なデータ構造定義によらずに柔軟かつ動的なデータを扱うことができる。
- ごみ集め (garbage collection) を標準として備えた最初の言語であり、これによりメモリ管理に煩わされることなしに動的なデータを扱うことができる。

ここでは、フリーの CommonLisp 実装である CLisp 処理系を用いて簡単に Lisp 入門をしたあと、Lisp 処理系の基本原理がどのようなものであるかを理解するための題材として、「Lisp で書いた Lisp」を取り上げて行きます。

2.2 CLisp の REPL

Lisp 処理系は伝統的に、「Read-Eval-Print Loop」(REPL) の形で実行環境が提供されています。これはつまり、「S 式を読み込み、それを評価 (計算) し、結果を印字し、また最初に戻る」ということです。Clisp で見てみましょう。

```
% clisp
...
[1]> (+ 1 2 3)          ←S 式を入力
6
[2]> (* (+ 1 2 3) 5.5) ←別の S 式を入力
33.0
[3]> (setq x 100)       ←変数 x を設定
100
[4]> x                  ← x を参照
100
[5] (/ x 1000)          ←S 式の中で x を参照
;;;;; どうなると思います?
[6]> y                  ←未定義の変数参照
;;;;; y は未定義だというエラーと選択肢
ABORT :R3     ABORT
Break 1 [7]> :R3        ←中止してトップに戻るという選択
[8]> (bye)              ←終わるという関数
Bye.
%
```

先に書いたように、Lisp の入力は「S 式」ですが、これは次のように定義されると思ってください (とりあえず)。

S 式 ::= リスト | アトム
リスト ::= (S 式…)
アトム ::= 記号 | 数値 | 文字列 | …

S 式をプログラムとして実行 (評価) するときは、次のようになります。

- 数値は、その数値、文字列はその文字列に評価される。
- 記号は、その記号が表す変数の値に評価される。
- リストは、その先頭は記号であり、関数名として扱われる、残りの部分はそれぞれ評価された後、関数の引数として渡される。

実はリストにはこの規則に従わない別のものもあるのですが、それはおいおい説明します。とにかく大事なのは、C などの言語では「`func(1, 2)`」のように書いていたのが Lisp ではすべて「`(func 1 2)`」のようにかっこを外に書く、カンマは使わず空白で区切る、ということです。慣れるまでちょっととまどうかも知れません。

あと、記号のうちでも `nil` と `t` は特別な意味を持っていて、それぞれ「偽/リストの終わり/空リスト」「真」を意味しています。ですからこれらを勝手な変数名として使うべきでありません。[3]

[3]ただし条件判定のときには `nil` 以外の値はすべて「真」として扱われます — C で 0 以外は真なのと同様。

2.3 関数定義

Lisp のプログラムを作るというのはだいたい、必要な関数を定義していくことに相当します。関数は次の形になっています。

```
(defun 関数名 (引数…) 本体…)
```

これも全部 S 式の形になっていることに注意。なので、引数もすべて空白で区切ります。本体の式を複数並べた場合は順番に実行して最後の値が関数値として返されます。が、だいたいは本体の式は 1 個だけです。なお、この defun というのは S 式ではありますが関数ではないですね。このような特別な機能を持つ S 式を「特殊形式 (special form)」と呼びます。

もう 1 つ、if 文に相当する特殊形式 cond を紹介してしまいましょう。

```
(cond (式 1 本体…)
      (式 2 本体…)
      …
      (t 本体…))
```

これは、まず式 1 を評価し、それが真なら対応する本体を実行し、結果を返します。そうでなければ、式 2 を実行し、それが真なら対応する本体を実行し、結果を返します…のように続き、最後は t なので、どれでもなければ最後の本体を実行して結果を返します。つまり if-elsif の連鎖になっているわけです。[4]

では、絶対値と階乗の関数を書いてみましょう。直接打ち込むのだと、間違いがあったときに直せませんから、エディタで書いてファイルに入れるのがよいと思います。

```
(defun absolute (x)
  (cond ((< x 0) (- x))
        (t x)))

(defun factorial (n)
  (cond ((< n 1) 1)
        (t (* n (factorial (- n 1))))))
```

これらをたとえば test1.lisp に書いておいたとして、CLisp に読み込ませ実行するのは次のようにします。

```
[1]> (load "test1.lisp")
;; Loading...
;; Loaded...
T
[2]> (absolute -3)
3
[3]> (factorial 5)
120
```

以下の演習で必要な関数を整理しておきます。

- 四則演算と剰余 — +, -, *, /, mod (-は単項の符合反転も)
- 数値の比較 — >, <, >=, <=、=、/=

[4]なお、2 分岐の if「(if 式 式 式)」もありますが、今回は後で実装してみる都合上、cond を優先して使います。

演習 1 CLisp を起動し、上の例題群をそのまま実行してみよ。エラーも出してみること。階乗でちょっと大きな値も試してみること (中断したければ Ctrl-C)。できたら、次のような関数も定義して実行してみよ。

a. 円の半径を与えると、面積を返す。
b. 2 次元の 2 点の座標 x1、y1、x2、y2 を与えると、距離を返す。
c. 整数を与えると、奇数か偶数かに応じて「"even"」「"odd"」を返す。

2.4 S 式と記号処理

さて、ここまでは例題として数値の計算ばかりやってきたが、いよいよ Lisp の特徴である記号処理に進むことにします。そのためには、S 式をデータとして扱う必要がありますが、そのために quote という特殊形式から始めます (' というのは打ち込むのを楽にするための略記法です)。

`(quote` S 式`)` ;;; または 'S 式

こう書くと、「そのままの S 式」を指定することができます (これがないと、S 式が関数として実行されてしまうのでしたね)。次に、S 式を操作するための「5 つの基本関数」を覚えて頂きます。

- `(eq` X Y`)` — X と Y がともに記号のとき、それらが同じ記号なら `t`、そうでなければ `nil` を返す。
- `(null` L`)` — リストが空 (=`nil`) のとき `t`、そうでなければ `nil` を返す。
- `(car` L`)` — リストが空でないとき、その先頭要素を返す。
- `(cdr` L`)` — リストが空でないとき、その先頭を除いた残りのリストを返す。
- `(cons` X L`)` — リスト L の頭に要素 X を追加したリストを返す。

実際に例を見てみましょう。

```
[1]> (eq 'a 'a)
T
[2]> (eq 'a 'b)
NIL
[3]> (null '())  ←空のリストは
T
[4]> (null nil)  ←nil と同じ
T
[5]> (null '(a b))
NIL
[6]> (car '(a b c))
A
[7]> (cdr '(a b c))
(B C)
[8]> (cons '(a b) '(c d e))
((A B) C D E)
```

最後のが意外に思えるかも知れませんが、`(c d e)` というリストの先頭に「`(a b)`」というリストを 1 要素として追加するので、こうなるのです。ここで基本 5 関数とは別なのですが、便利に使える関数を 2 つ紹介しておきます。

- `(append` L M`)` — 2 つのリスト L と M を連結したリストを返す。

- (list A B C …) — すべての引数を並べたリストを返す。

後者は quote でいいのではと思った人、演習をやってから言うように。

あと、car と cdr は連続して使うことが多いので、(car (cdr X)) は (cadr X)、(car (car (cdr X))) は (caadr X) のように「合成した」関数が「a」「d」の並び 4 個まですべて用意されています。

演習 2 まず、(setq w '(a b (c d e) f)) を実行してください。そのあと、quote は使わず、基本 5 関数だけで次のものを取り出してみてください。

a. A

b. (C D E)

c. NIL

d. (E F)

演習 3 まず、(setq x '(a b c))、(setq y '(d e)) を実行してください。そのあと、基本 5 関数と append と list だけで次のものを生成してみてください。

a. (A B C D E)

b. ((A B C) (D E))

c. (A B C (D E))

d. (A B C A D E)

2.5 リスト処理の再帰関数

リスト処理に慣れたところで、こんどは再帰関数を使ってリストを処理する流儀を見て頂きます。その前に、条件判定に必要な述語をいくつか追加しておきます。

- (atom X) — X がアトム (数値か記号) なら t。
- (numberp X) — X が数値なら t。
- (symbolp X) — X が記号なら t。
- (listp X) — X がリスト (空リストを含む) なら t。
- (consp X) — X がリスト (空リスト以外) なら t。

では基本的な例として、リストの長さを求める関数を作ってみます。

```
(defun listlen (l)
  (cond ((null l) 0)
        (t (+ 1 (listlen (cdr l))))))
```

考え方は次のようになります。このようにリストを先頭からはがしていく再帰は基本なのでよく味わってください。

- リストが空リストなら、長さは 0。
- それ以外なら、長さはリストの先頭を除いた残りのリストの長さ+1。

実行例を示します。トレースしてみると、徐々にリストを短くしつつ計算していることがよく分かります。

```
[2]> (listlen '(a b c d))
4
[3]> (trace listlen) ← トレースモードにする
...
;; Tracing function LISTLEN.
(LISTLEN)
[4]> (listlen '(a b c d))
1. Trace: (LISTLEN '(A B C D))
2. Trace: (LISTLEN '(B C D))
3. Trace: (LISTLEN '(C D))
4. Trace: (LISTLEN '(D))
5. Trace: (LISTLEN 'NIL)
5. Trace: LISTLEN ==> 0
4. Trace: LISTLEN ==> 1
3. Trace: LISTLEN ==> 2
2. Trace: LISTLEN ==> 3
1. Trace: LISTLEN ==> 4
4
[5]> (untrace listlen) ← トレース解除
(LISTLEN)
```

演習 4 次のような処理をする再帰関数を書け。既に定義した関数を下請けにしてもよいことに注意。

a. 数値のリストを渡すとその数値の合計を返す関数 `listsum`。例: `(listsum '(1 2 3 4))` → 10。

b. リスト中に含まれているすべてのアトムの数を返す関数 `countatoms`。例: `(countatoms '(a (b c) d (e)))` → 5。

c. 記号とリストを受け取り、リスト中からその記号をすべて除去したリストを返す関数 `remove`。例:`(remove 'e '(e v e n t))` → `(V N T)`。

d. リストとリストを受け取り、2 番目のリストの中から 1 番目のリストに含まれている記号をすべて除去したリストを返す関数 `remset`。例: `(remset '(a e i o u) '(n a g a s a k i))` → `(N G S K)`。

e. 2 つの長さが同じリストを受け取り、その各要素の対を長さ 2 のリストとして並べたリストを返す関数 `pair`。例: `(pair '(a b c) '(x y z))` → `((A X) (B Y) (C Z))`。

f. 2 つのリストを受け取り、両者を連結したリストを返す関数 `concat`。実はこれは `append` と同じもの (当然、`append` を利用してはいけない)。

3 Lisp の秘密 (?)

3.1 cons の内幕

さて、ここまで `cons` の第 2 引数は常にリストだという話で通してきました。しかし実はそうでなくてもよいのでして、次の例を見てください。

```
[1]> (cons 'a 'b)
(A . B)
```

この . というのは何でしょう? 実は、これまで Lisp に打ち込んできた式 (S 式、と呼ばれています) の内部構造は次の図 2 のように「cons セル」とよばれる要素から成っています。各 cons セルは「下」と「右」の 2 方にたどることができます。実はこれが car と cdr なのです。

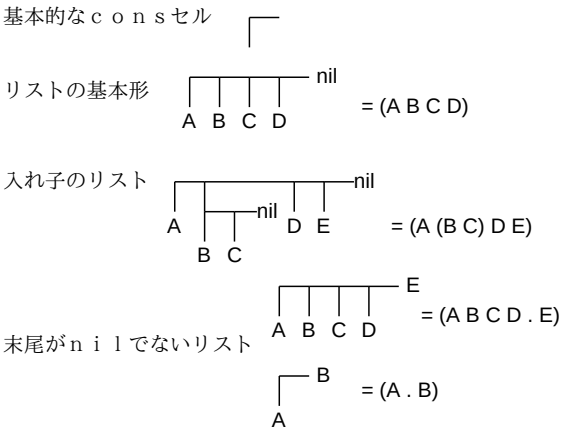

図 2: cons セルの概念図

リストというのは実はこれを n 個つないで、末尾に終りの印 nil をつけたものだったわけです。そして、末尾が nil でないことも可能であり、その場合には . のあとにその末尾の要素を書くことで表します。より厳密に言えば、次のどちらの書き方も正しいわけです。

```
(A . nil)   ==  (A)
(A . (B . (C . nil)))  ==  (A B C D)
```

ただし、. を毎回つけていると煩わしいのも確かなので、普段はできるだけ . をつけない記法を使うのが一般的だということです。