

2024 機械情報夏学期 ソフトウェア第二

担当：岡田 慧 k-okada-soft2@jsk.t.u-tokyo.ac.jp

3. ソフトウェアの計算モデル

ソフトウェアの計算モデル及び手続きとデータの特に抽象化の持つ重要性について「計算機プログラムの構造と解釈」(Structure and Interpretation of Computer Programs) (<https://mitpress.mit.edu/sites/default/files/sicp/index.html>) の 1, 2 章に基づいて紹介する。

日本語訳は東大名誉教授の和田英一先生による <https://sicp.iiijlab.net/fulltext/xcont.html> が全文公開されている。また真鍋氏による新訳版もある <https://github.com/hiroshi-manabe/sicp-pdf/blob/japanese/jsicp.pdf>。

<https://uents.hatenablog.com/entry/sicp/index> もよくまとまっている。

原書は Scheme 言語を用いた説明がなされているが、ここでは Python で説明し直している。Python コードは <https://wizardforcel.gitbooks.io/sicp-in-python/content/> を参考にした。

1 宣言的知識 (declarative knowledge: what is) と命令的知識 (imperative knowledge: how to)

1.1 数学 (宣言的) と計算機科学 (命令的/手続き的)

数学ではある出力を得るため手続きを、その出力の性質を持って表そうとするもの (what is true) であり、コンピュータ・サイエンス (計算機科学) はその出力を得るための手続きを示すもの (how to) といえる。

前者は宣言的知識 (declarative knowledge) と呼ばれ、例えば、 x の平方根 y を例に取ると、

$$\sqrt{y} \geq 0 \text{ かつ } y^2 = x \text{ であるような } y$$

と示すことができる。

一方、後者はと命令的 (規範的) 知識 (imperative/procedural knowledge) であり、例えば以下のニュートン法による平方根の計算手続きがある。

1. x の平方根の予測値を $y = x$ とする。
2. y^2 が十分に x に近ければ、 y を平方根の値とする。
3. 予測値 y を更新し 2) に戻る (更新方法は例えば y と x/y の平均など)

21. 宣言的知識 (declarative knowledge: what is) と命令的知識 (imperative knowledge: how to)

1.2 宣言型プログラム言語と命令型プログラム言語

このことから、ほとんどのプログラム言語は命令的と言えるが、一方で一部の論理プログラミング言語のような宣言的なプログラム言語が存在する。つまり (これは非常にややこしいが) 数学と比較すれば命令的な計算機プログラム言語の中でも、宣言型プログラム言語 (Declarative programming language) と命令型プログラム言語 (Imperative/Procedural programming language) の2つがある。

例えば Python を用いた平方根の計算手続きはニュートン法を用いて以下のように記述することができる。

```
def my_sqrt(x):
    y = x

    while True:
        if (y * y) - x < 0.001:
            return y

        y = ( y + x / y ) / 2.0
```

一方、Prolog と呼ばれる論理型言語では

```
my_sqrt(X, Y) :- member(Y,[1,2,3,4,5,6,7,8,9,0]), X is Y * Y.
```

として、それぞれ Y が X の平方根かどうか、調べることができる¹。

以下は、4 が 16 の平方根かどうかの真偽 (true or false) を調べており、

```
?- my_sqrt(16, 4). # . (ピリオド) を打った後にリターンを押す。
true.             # true が表示された後にリターンを押す。
```

また、以下は 4 が X の平方根となるときに、X の値を示すプログラムである。

```
?- my_sqrt(X, 4).
X = 16.
```

さらに、Y が 16 の平方根となるときに、Y の値を示すプログラムは以下のように書ける。

```
?- my_sqrt(16, Y).
Y = 4;                # ; (セミコロン) を打つと、複数の解が存在すれば続けて表示される
false.               # false が表示されるとこれ以上解は無い
```

1.3 宣言文的プログラミング (記述) と命令文的プログラミング (記述)

さらに (これも非常にややこしいが) 命令文的プログラム言語におけるプログラムの記述方法の中でも what is をプログラムとして記述する宣言文的な書き方と、how to をプログラムとして記述する命令文的な書き方が存在する。

¹Prolog のインタプリタは gnuprolog ではなく、swi-prolog を使うこと。すなわち、`sudo apt install swi-prolog-nox` とする。すでに gnuprolog がインストールされてる場合は、`sudo apt remove gprolog` としてアンインストールしておくこと。実行方法は、この宣言をファイル、例えば `sqrt.pl` に記述し、`prolog -f sqrt.pl` としてインタプリタと立ち上げると `?-` とコマンドプロンプトが表示されるので、`my_sqrt(16, 4).` 等の、命令 (述語: predicate) を記入すると実行できる。このプログラムでは $y \geq 0$ の部分を 0 から 10 までの整数と制約を強めている。

例えば、階乗の計算を考えると、この数学的な定義は

$$n! = n \cdot (n-1)!$$

となる。

この定義は、以下の再帰関数で計算することができ、これは階乗の定義をそのままプログラムで記述した what is を記す宣言文的な書き方といえる²。

```
def my_factorial_1(n):
    if n == 1:
        return 1
    else:
        return n * my_factorial_1(n - 1)
```

一方で、以下のように、 1×2 その結果に 3 を掛け、その結果に 4 を掛ける ... という計算の過程を手続きとしてプログラムとして記述した how to を記す命令文的な書き方も可能である³。

```
def my_factorial_2(n):
    def iter(product, counter, max_count):
        if counter > max_count:
            return product
        else:
            return iter(counter * product, counter + 1, max_count)
    return iter(1, 1, n)
```

これをまとめると以下ようになる。

宣言的知識	数学				
命令的知識	計算科学	宣言型プログラム言語	論理型言語や制約型言語等		
		命令文的プログラム言語	その他	宣言文的記述	定義を記す
				命令文的記述	手続きを記す

2 コンピュータプログラム言語の要素

2.1 基本式 (primitive expressions) と合成式 (compound expression)

式は基本式 (primitive expressions) と合成式 (compound expression) からなる。

基本式は、数値や文字列など評価⁴すると自分自身の値が返る Self-evaluating primitives や、四則演算や論理式などの Built-in procedure(組み込み手続き) からなる。

例えば数値は以下のように評価すると自分自身が返る。以下は Python を立ち上げインタプリタを実行した例になっている。

² プログラムをファイル、例えば fact.py に記述し、python2 としてインタプリタと立ち上げると >>> とコマンドプロンプトが表示されるので、exec(open('fact.py').read()) としてファイルをロードし、引き続きコマンドプロンプトから print(my_factorial_1(n)) を記入すると実行できる。Python2 の場合は、execfile('fact.py') とする。

³ このとき関数 iter は関数 my_factorial_2 の中で定義されており、そのスコープだけ有効である。また、product, counter, max_count は関数 iter の中だけで有効であり、外部から隠蔽 (hiding) しているという。オブジェクト指向の特徴の一つであるが、このように関数を使っても実現できる。

⁴ evaluation. 乱暴に言うところインタプリタ上でリターンを押して実行し、値を返すこと、例えば Python 自身にも eval という関数がある。eval('1000-334'); が実行できる。

```
>>> 486
486
```

一方で、合成式は基本式を組み合わせて新しい式を作ることができる。式を評価すると、その計算結果が返ってくる。

```
>>> 1000 - 334
666
```

$+$, $-$, $*$, $/$ などを演算子 (operator) と呼び、これを作用される要素を被演算子 (operands) という。Pythonにおける数値の四則演算は演算子が被演算子の間に配置される中置記法 (infix notation) である。

2.2 式の呼び出し (call expressions) と部分式 (subexpressions)

式の呼び出しは手続き/関数 (procedure/function)⁵を被演算子の値である引数 (arguments) に作用させて、返り値を得る。

これは入力引数に対して出力の値を決定する数学の関数と同様であり、例えば `max` は引数の数値の集合から、最も大きい値を出力とする関数である。

```
>>> max(486, 666)
666
```

これは、関数名を演算子として捉えると、演算子を被演算子の前に置く書き方であり、前置記法 (prefix notation) とか、ポーランド記法 (polish notation) と呼ばれている。この記法のメリットは任意個の引数をとる手続きを許し、また入れ子にして式を拡張する入れ子式 (nested expressions) の記述が容易であるという点にある⁶。

2.3 合成式の呼び出しの評価 (evaluation of compound expression) と置き換えモデル (substitution model)

合成式の呼び出しの評価は次の処理になる。

- 部分式 (subexpressions) を評価し値を得る。
- 演算子 (operator / 関数) を、被演算子 (operand / 引数) に作用させる。

例えば中置記法 (infix notation) で書かれた以下の式は

$$(2 + (4 * 6)) * (3 + 5)$$

は前置記法 (prefix notation) を使うと以下のように書くことができる。

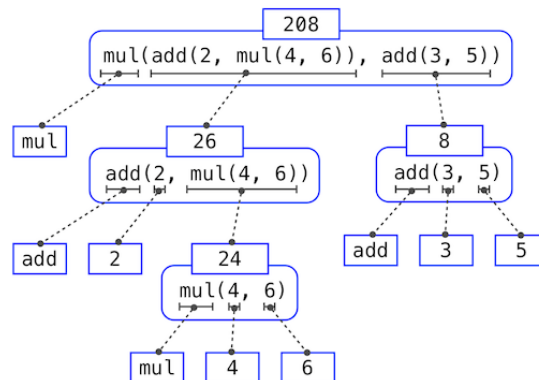
⁵SICP では手続きと関数は同じ意味で扱っている

⁶Scheme など前置記法が徹底されているプログラム言語では、四則演算や論理演算の計算も関数の呼び出しも同じ記法になり、合成式と式の呼び出しの区別がなく、シンプルな説明が可能になっている

2.3. 合成式の呼び出しの評価 (evaluation of compound expression) と置き換えモデル (substitution model)⁵

```
from operator import *  
mul(add(2, mul(4, 6)), add(3, 5))
```

この評価 (一番外側の mul) を行うためには, 組み合わせの各要素 (`add(2, mul(4, 6))` と `add(3, 5)`) の評価を行う必要がある. すなわち, 評価は本質的に再帰的なプロセスとなる.



また, 演算子が関数である場合, また, その関数 (手続き) に引数を作用させるためには, 各仮パラメタ⁷を対応する引数で取り替え, 手続き本体を評価する.

例えば,

```
def square(x):  
    return x * x  
def sum_of_squares(x, y):  
    return square(x) + square(y)  
def f(a):  
    return sum_of_squares(a + 1, a * 2)
```

という定義がある際に,

```
f(5)
```

を評価する場合, まず, `f` の本体を取り出し,

```
sum_of_squares(a + 1, a * 2)
```

を得る. 次に, この仮パラメタ `a` を引数の `5` で取り換え,

```
sum_of_squares(5 + 1, 5 * 2)
```

となる.

このとき, `sum_of_squares` に作用させるべき被演算子は, それぞれ `5 + 1`, `5 * 2` となり, それぞれ評価すると最終的には `6`, `10` となり, 関数 `sum_of_squares` に `6` と `10` を作用させることになる. この値は関数 `sum_of_squares` の本体の仮パラメタ `x`, `y` に置き換えられ, 式は

```
square(6) + square(10)
```

⁷仮引数: 関数定義字に使用される引数

に帰着する．`square` の定義を使えば

```
6 * 6 + 10 * 10
```

となり，乗算により

```
36 + 100
```

となり，最後に

```
136
```

となる．

このプロセスを置き換えモデル (substitution model) と呼び，プログラムの意味を定めるモデルと考えて良いが，実際のインタプリタの働きはこれとは異なる場合がある．また，このモデルはもともと単純なものでより詳細なモデルが必要となる手続きも存在する．

2.4 作用的順序 (applicative-order evaluation) と正規順序 (normal-order evaluation)

前節で示した評価方法では，まず演算子と被演算子を評価し，次に演算子の評価の結果の手続きに対して，被演算子の評価の結果を引数として作用させる．これを作用的順序と呼ぶ．

一方で，その値が必要になるまで被演算子を評価しない方法を正規順序の評価，と呼ぶ．ここでは，基本式 (primitive expression) だけを持つ式が出てくるまで，パラメタへの被演算子の式の置き換えを繰り返し，評価を行う．

```
f(5)
sum_of_squares(5 + 1, 5 * 2)
sum_of_squares(6, 10)
  square(6) + square(10)
    6 * 6 + 10 * 10
    36 + 100
    136
```

作用的順序による置き換えの進行

```
f(5)
sum_of_squares(5 + 1, 5 * 2)
  square(5 + 1) + square(5 * 2)
(5 + 1) * (5 + 1) + (5 * 2) * (5 * 2)
  6 * 6 + 10 * 10
  36 + 100
  136
```

正規順序による置き換えの進行

正規順序の評価では `5 + 1`，や `5 * 2` の評価が二度ずつ行われているが，双方で結果は変わらない．

一方，作用的順序と正規順序の評価が同じ結果にならない例も存在する．

```
def p():
    return p()

def test(x, y):
    if x == 0:
        return 0
    else:
        return y

test(0, p())
```

では、作用的順序で評価されると無限ループに陥る。すなわち、まず、引数の 0 と `p()` が評価される。 `p()` は、その中で `p()` 自体を呼び出すため無限ループとなっている。

一方で正規順序の評価では、必要な要素のみが評価される。すなわち、`test` 関数の第一引数が 0 であり、これを評価し `x == 0` が真となることから `test` から 0 が返る。

```
File "<stdin>", line 2, in p
File "<stdin>", line 2, in p
File "<stdin>", line 2, in p
RuntimeError: maximum recursion depth exceeded
```

正規順序で評価する言語としては Scheme から派生した Racket 言語 (PLT Scheme) が存在する⁸。ターミナルから

```
$ racket -i -l lazy
```

としてインタプリタを実行し、

前述のプログラムと等価の以下のプログラムを例えばファイル (`test.l`) に記述し、これを (`load "test.l"`) として読み込み、(`test 0 (p)`) を評価すると 0 が返ってくる。

```
(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))
```

2.5 静的スコープ (static scope, lexical scope) と動的スコープ (dynamic scope)

スコープはある変数や関数の名前を参照できる範囲のことである。通常、変数は関数やブロック内で定義し利用することが望ましい。また、内側のブロックで定義されていない変数がある場合は、その外側のブロックで定義されている変数を参照する。例えば以下のプログラムでは、変数 `a` は関数 `bar()` 内では定義されていないが、大域変数として文頭に `a = 10` が定義されているため、これを参照する。

この様にスコープが構文上から決めることができるものを静的スコープと呼ぶ。

```
a = 10

def foo():
    # global a # global と宣言すると大域変数进行操作することになる
    a = 100
    print("foo: a = {}".format(a))
    bar()

def bar():
    print("bar: a = {}".format(a))

foo()
print("global: a = {}".format(a))
```

⁸<https://racket-lang.org/>, Linux では `apt install racket` としてインストールできる。

一方、スコープが、その実行状態によるものを動的スコープと呼ぶ。すなわち、呼び出され側から、呼び出した側のスコープを参照できる。

例えば Bash は動的スコープを採用している。以下の例では bar を呼び出した際に、foo の局所変数を参照している。foo, bar の呼び出しが終わった後は、大域変数の a は変化なく、最初にセットされた 10 のままである。

```
a=10

foo() {
    local a=100 # local をつけないとただ単に大域変数进行操作することになる
    printf "foo: a = $a\n"
    bar
}

bar() {
    printf "bar: a = $a\n"
}

foo
printf "global: a = $a\n"
```

2.6 純粋関数 (pure function) と非純粋関数 (non-pure function)

純粋関数とは戻り値が一意に定まり、標準入出力やファイル入出力を行わない関数。例えば、`abs(-2)` はその例である。

```
>>> abs(-2)
-2
```

一方、非純粋関数の例として `print` がある⁹

```
>>> print(-2)
-2
```

`abs` も `print` も同様の出力を表示しているように見えるが、`print` の戻り値は常に `None` であり、関数の中で `-2` を標準出力に書き出している。



以下の例では `-2` は `print(-2)` の副作用として標準出力に書き出した結果であり、`2` `None` は `abs(-2)`, `print(-2)` のそれぞれの戻り値である。

⁹Python2 の `print` は文であり、戻り値がないが、`from __future__ import print_function` とすることで、関数として扱える。


```
>>> print(abs(-2), print(-2))
-2
2 None
```

3 再帰 (recursion) と反復 (iteration)

3.1 手続き (Procedure) とその際に生成するプロセス (processes they generate)

前掲した階乗計算の2つの手続きについて、それぞれ再帰関数（手続き）として定義されているが、プログラムコードが実際の計算機の中で実行されるステップの展開、具体的には呼び出されるステップの回数と、そこで必要となる記憶の量、すなわち計算プロセス (computational process) の様子は大きく異なる。

```
def my_factorial_1(n):
    if n == 1:
        return 1
    else:
        return n * my_factorial_1(n - 1)
```

```
def my_factorial_2(n):
    def iter(product, counter, max_count):
        if counter > max_count:
            return product
        else:
            return iter(counter * product,
                        counter + 1,
                        max_count)
    return iter(1, 1, n)
```

それぞれ置き換えモデルを用いて計算のプロセスを見てみると以下ようになる。それぞれ、線形再帰プロセスと線形反復プロセスと呼ばれている。

```
my_factorial_1(6)
6 * my_factorial_1(5)
6 * 5 * my_factorial_1(4)
6 * 5 * 4 * my_factorial_1(3)
6 * 5 * 4 * 3 * my_factorial_1(2)
6 * 5 * 4 * 3 * 2 * my_factorial_1(1)
6 * 5 * 4 * 3 * 2 * 1
6 * 5 * 4 * 3 * 2
6 * 5 * 4 * 6
6 * 5 * 24
6 * 120
720
```

```
my_factorial_2(6)
iter(1, 1, 6)
iter(1, 2, 6)
iter(2, 3, 6)
iter(6, 4, 6)
iter(24, 5, 6)
iter(120, 6, 6)
iter(720, 6, 6)
720
```

線形再帰プロセスでは膨張と収縮の形を取っている。膨張している時はプロセスの遅延演算 (deferred operations) が発生し、収縮していくときに実際の演算が行われている。

この形のプロセスは再帰的プロセスと呼ばれ、ステップ数に加えて遅延演算の列、すなわち、それを覚えておくのに必要な情報の量は n に比例して線形に成長する。このような再帰プロセスを線形再帰プロセスと呼ぶ。

上の例では `my_factorial_1(5)` が呼ばれるときに、この結果と 6 を掛けなければいけないというふうに、後で実行する演算を覚えておく必要があり、この n に比例して記憶量が必要となる。

一方、後者の反復的プロセスでは、各ステップで覚えておくべき変数は状態変数 (product, counter, max-count) の数だけである。計算途中のどの時点でも状態変数でプロセスの状態を持っており、計算を途中で止めても、それを再開することが容易である。反復プロセスにおいても、計算に必要なステップ数が n に比例して線形に成長する場合、線形反復プロセスと呼ぶ。

3.2 再帰 (linear recursion) プロセスと反復 (linear iteration) プロセス

ここで、再帰的プロセス (process) と再帰的手続き (procedure) を混同しないように注意してほしい。2つの階乗の計算は両方とも再帰の手続きである。手続きが再帰的というのは、手続き定義 (関数定義) の構文上の事実をいう。一方プロセスが再帰的あるいは反復的、というのは、その手続きを実行する際にプロセスがどのように進化していくか、という点を示している。

ちなみに、通常の言語では再帰手続きの実行時に必要となる記憶量が常に手続きの数だけ増えるように設計されており、たとえそのプロセスが原理的に反復的であっても、手続きの呼び出し数に比例した記憶量を必要とする。そのため、これらの言語では、do, for, while などの反復プロセス (繰り返し) 用の特殊目的の文が用意されている。

3.3 末尾再帰 (tail recursion) と末尾再帰最適化 (tail-call optimization))

再帰的手続きにおいて、自身の再帰呼び出しが、その計算の最後のステップになるような再帰のパターンを末尾再帰 (tail recursion) と呼ぶ。my_factorial_2 関数は末尾再帰の形になっていることを確認しよう。

末尾再帰の形になっている手続きは、インタプリタ / コンパイラにおいて反復的プロセスとなるような実行に最適化することができる。このような実装を末尾再帰最適化 (tail recursion optimization / TCO: tail-call optimization) と呼ばれる。

具体的には、末尾再帰の手続きは不要な再帰呼び出しをすることなく、ジャンプ文により結果を得ることができる。例えば以下の my_factorial_3 関数では、label .begin, goto .begin のジャンプ命令¹⁰により再帰を使わないプロセスとなっている。これは、結局これは my_factorial_4 を行っていることとなり、見慣れている繰り返し文となっている。

¹⁰Python はジャンプ文をサポートしていないが、<https://github.com/snoack/python-goto/> など有志による拡張を利用することができる。これは pip モジュールと呼ばれる Python のモジュールパッケージマネージメントシステムにより提供されており、pip3 install goto-statement として、インストールする。このときモジュールは ~/.local/lib/python2.8/site-packages 以下にダウンロードされる。挙動がおかしくなったら、最悪 ~/.local 以下を消せばよい。なお、調べると sudo pip3 install と書いてある場合があるが、ubuntu の apt でもインストール可能な python モジュールもあり、非常に危険である。指定された以外のコマンドで sudo を使わないこと、というルールを思い出してほしい。まあ、wget <https://raw.githubusercontent.com/snoack/python-goto/master/goto.py> として同じディレクトリにファイルをダウンロードし利用するのも安全である。

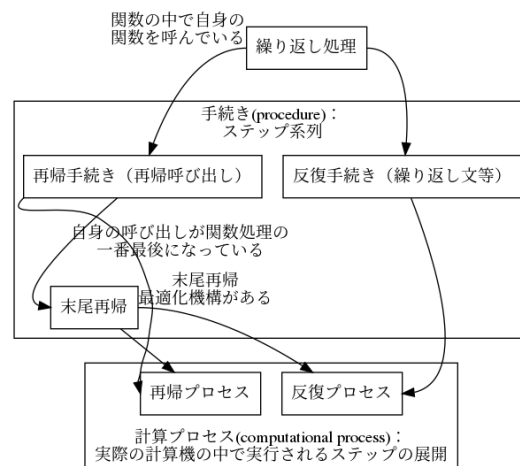
```
from goto import with_goto
```

```
def my_factorial_3(n):
    @with_goto
    def iter(product, counter, max_count):
        label .begin
        if counter > max_count:
            return product
        else:
            product = counter * product
            counter = counter + 1
            goto .begin
    return iter(1, 1, n)
```

```
def my_factorial_4(n):
    product = 1
    counter = 1
    max_count = n
    while True:
        if counter > max_count:
            return product

        product = counter * product
        counter = counter + 1
```

以上，本節をまとめた図が以下ようになる．



4 宿題

提出先：ITC-LMS を用いて提出すること

提出内容：以下の問題の実行結果の画面をキャプチャしファイル名は「問題番号.png」とし，また講義中にてきたキーワードについて知らなかったもの，興味のあるものを調べ「学籍番号.txt」としてアップロードすること．テキストファイルはワードファイルなどだと確認出来ないことがあるため，emacs/vi 等のテキストエディタを使って書こう．プログラムが長くなりキャプチャ画面に入り切らなくなってきたらプログラムファイルと実行結果を「問題番号.txt」にまとめてアップロードしてよい．

画像で提出する場合は，各自のマシンの Mac アドレスが分かるようにすること．例えば画面中に ifconfig というコマンドを打ち込んだターミナルを表示すればよい．

ITC-LMS にアップロードする際には講義・宿題の感想を必ずコメントに記すこと．また授業中に質問した者はその旨を記すこと．質問は成績評価時の加点対象となる．

キーワード：宣言的，命令的，再帰関数呼び出し，反復文（繰り返し構文），静的スコープ，動的スコープ，再帰プロセス，反復プロセス，末尾再帰最適化，

1. 次の二つの手続きはどちらも、引数を 1 増やす手続き `inc` と、引数を 1 減らす手続き `dec` を使って、二つの正の整数を足す方法を定義している。置換えモデルを使い、それぞれの手続きが $(4 + 5)$ を評価する時に生成するプロセスを示せ。そのプロセスは反復的か再帰的か。

```
def inc(x):
    return x + 1
def dec(x):
    return x - 1

def plus_a(a, b):
    if a == 0:
        return b
    else:
        return inc(plus_a(dec(a), b))

def plus_b(a, b):
    if a == 0:
        return b
    else:
        return plus_b(dec(a), inc(b))
```

2. 以下のフィボナッチ数の計算アルゴリズムを実装し実行してみよ（言語は何でも良い）、その時の計算プロセスが、ステップ数は指数関数的に増加するが、必要な記憶領域は線形に増加する木再帰 (tree recursion) と呼ばれていることを確認せよ。

```
Fib(n) = 0                if n = 0
Fib(n) = 1                if n = 1
Fib(n) = Fib(n-1) + Fib(n-2) otherwise.
```

3. C 言語はコンパイル時に末尾再帰最適化が行われているか？確認してみよう。

例えば、末尾再帰で書かれた階乗のプログラムを `fact-toc.c` として、

```
$ gcc -S -c -O2 -o fact-toc.s fact-toc.c
```

として、アセンブラとしそのコードを見ると良い。例えば、`-O2` を `-O0` として最適化オプションを無効にした場合とで違いがあるかを調べてみる事ができる。

また、実行ファイルから C 言語を逆アセンブルするツールも存在する。以下のようにして使ってみよう。

```
$ gcc -g -O2 -o fact-toc fact-toc.c
$ docker run --rm -v $(pwd):/samples cincan/snowman-decompile -v /samples/fact-toc
```

また、JavaScript ではオンライン最適化ツール (<https://babeljs.io/repl>) もあるので興味あるものは確認するとよい¹¹。

¹¹Plugins から `babel-plugin-tailcall-optimization` を選択すること



次の二つの手続きはどちらも、引数を1増やす手続き`inc`と、引数を1減らす手続き`dec`を使って、二つの正の整数を足す方法を定義している。置換モデルを使い、それぞれの手続きが $(4+5)$ を評価する時に生成するプロセスを示せ。そのプロセスは反復的か再帰的か。

```
def inc(x): return x+1
def dec(x): return x-1
def plus_a(a,b):
    if a==0:
        return b
    else:
        return inc(plus_a(dec(a),b))

def plus_b(a,b):
    if a==0:
        return b
    else:
        return plus_b(dec(a),inc(b))
```



まず、手続き `plus_a` のプロセスを示します。



```
SCSS Copy code

plus_a(4, 5)
inc(plus_a(3, 5))
inc(inc(plus_a(2, 5)))
inc(inc(inc(plus_a(1, 5))))
inc(inc(inc(inc(plus_a(0, 5)))))
inc(inc(inc(inc(5))))
inc(inc(inc(6)))
inc(inc(7))
inc(8)
9
```

これは再帰的なプロセスです。プロセスは、再帰呼び出しが戻ってくるときにスタックにプッシュされたままになります。そして、最後に戻ってきた呼び出しで計算が完了します。

次に、手続き `plus_b` のプロセスを示します。



フィボナッチ数の計算プログラムの計算プロセスが、ステップ数は指数関数的に増加するが、必要な記憶領域は線形に増加する木再帰(**treerecursion**)と呼ばれていることを確認せよ。



フィボナッチ数列は、以下のように定義されます。



scss

Copy code

```
fib(n) = n                if n = 0 or n = 1
      = fib(n-1) + fib(n-2) if n > 1
```

これを再帰関数として実装すると、以下のようになります。

arduino

Copy code

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

このプログラムの計算プロセスは、フィボナッチ数 n に対して指数関数的に増加します。つまり、 n に対して $O(2^n)$ のステップ数が必要になります。

しかし、必要なメモリ領域は線形に増加します。**treerecursion**の場合、再帰呼び出しが木のような構造を形成するため、深さが n である再帰呼び出しの数は最大でも $O(n)$ です。つまり、必要なメモリ領域は $O(n)$ です。

これは、フィボナッチ数列の再帰関数が、同じフィボナッチ数を複数回計算することがあるためです。しかし、計算されたフィボナッチ数を配列や辞書に保存することで、再計算を回避することができ、ステップ数を $O(n)$ に減らすことができます。



C言語はコンパイル時に末尾再帰最適化が行われているか？



C言語の標準規格であるC11や、C言語の前身であるK&R Cには、末尾再帰最適化についての言及はありません。そのため、Cコンパイラが末尾再帰最適化を行うかどうかは、コンパイラの実装依存となります。

一般的に、Cコンパイラは末尾再帰最適化をサポートしていません。そのため、再帰的な関数を使ったアルゴリズムを設計する場合、スタックオーバーフローのリスクがあるため、注意が必要です。

しかし、一部のCコンパイラは、末尾再帰最適化を行うオプションを提供しています。例えば、GCCコンパイラには、`-foptimize-sibling-calls`オプションがあります。このオプションを有効にすると、末尾再帰最適化が有効になります。

また、C言語の代替として開発されたSchemeやHaskellなどの関数型プログラミング言語では、末尾再帰最適化が重要な最適化手法の1つとして取り扱われています。

付録

SICP(「計算機プログラムの構造と解釈」)で用いられているプログラミング言語として EmacsLisp/Scheme/Python について概説する。

それぞれ、式、変数、関数、条件式と述語、スコープ、反復プロセス、非ローカル脱出、無名関数/クロージャの順番で説明してある。それぞれ相互に参照しながら理解して言ってほしい。また、将来新しいプログラム言語を使うことになっても、この順番で一通り仕様を追いかけることで、必要最低限の情報を得ることが出来るように成るはずである。

A Emacs と Emacs Lisp

Emacs はテキストエディタであり、拡張性高く自由にカスタマイズ出来る点が特徴であり、UNIX 文化では vi と共に最も利用されるエディタである。

オリジナルは 1975 年にガイ・スティールとリチャード・ストールマンが開発した TECO エディタのエディタ・マクロ (Editor MACroS) であり、現在の Emacs は 1984 年にリチャード・ストールマンが開発を始めた GNU Emacs が現在も使われている。

ちなみに、ガイ・スティールは Scheme 言語、Java 言語の開発にも携わった計算機科学者であり、C 言語、FORTRAN、CommonLisp の ANSI 標準化にも参加した。

一方、リチャード・ストールマンは GNU プロジェクトの主催者であり、様々な GNU フリーソフトウェアを開発してきている。先週紹介した gdb も GNU Debugger である。GNU のソフトウェア全てインターネット上のサーバ (<ftp://ftp.gnu.org/gnu/>) で公開されダウンロードできる。

A.1 式

式により以下の様に四則演算が利用可能である。また、値その物を評価し、その値を得ることが出来る。

```
486
486 <-- 返り値
(- 1000 334)
666 <-- 返り値
(* 5 99)
495 <-- 返り値
(/ 10 5)
2 <-- 返り値
(+ 2.7 10)
12.7 <-- 返り値
```

ここで括弧ではさまれた部分の左端の要素を演算子 (operator) と呼び、他の要素を被演算子 (operands) という。式の評価は演算子が指定する手続き (procedure) を被演算子の値である引数 (arguments) に作用させて得る。

この様に演算子を被演算子の前に置く書き方を前置記法 (prefix notation) とか、ポーランド記法 (polish notation) と呼ぶ。この記法のメリットは任意個の引数をとる手続きを許し、また入れ子にして式を拡張することが容易であるという点にある。


```
(+ 21 35 12 7)
75 <-- 返回值
(* 25 4 12)
1200 <-- 返回值
(+ (* 3 5) (- 10 6))
19 <-- 返回值
```

C-j 以外にも C-x C-e (Ctrl キーを押しながら x を打ちその後, Ctrl キーを押しながら e を打つ) でカーソルのある場所から左の最初の式を評価する。たとえば,

```
(+ (* 3 5) (- 10 6))
```

で, 数値 5 の次の括弧にカーソルがある場合には, 評価結果は 15 になり, 6 の次の括弧の上にカーソルがある場合には, 結果は 6 になる。また, 最後の各個にカーソルがあれば, 結果は 19 になる。

A.2 変数

Emacs-Lisp では以下の組み込み関数 `setq` を使って変数を定義する。一旦変数を定義したら, それ以降は変数を使ってプログラムを書くことができる。

```
(setq size 2)
2 <-- 返回值
(* 5 size)
10 <-- 返回值
```

EmacsLisp では `pi` があらかじめ組み込まれているので, それを使い円周を以下の様に計算できる。

```
(setq radius 10)
10 <-- 返回值
(* pi (* radius radius))
314.1592653589793 <-- 返回值
(setq circumference (* 2 pi radius))
62.83185307179586 <-- 返回值
```

A.3 関数

関数は `defun` を用いて以下の様に定義する。

```
(defun square (x) (* x x))
square <-- 返回值
```

関数の定義は以下の様に一般化できる。

```
(defun <name> (<arguments>) <body>)
```

<name> は関数の定義を対応づける名前であり, arguments は仮パラメタと呼ばれ, 関数の引数を

指す名前である．<body>は仮パラメタを用いて関数の値を計算する式である¹²．

定義した関数（手続きは）以下の様に利用できる．square の引数に他の手続きを利用することも，square を他の手続きの引数に利用することも可能である．

```
(square 21)
441 <-- 返回值
(square (+ 2 5))
49 <-- 返回值
(square (square 3))
81 <-- 返回值
(+ (square 3) (square 4))
25 <-- 返回值
```

また，以下の様に自分で定義した関数を呼び出す関数もつくる事が出来る．

```
(defun sum-of-squares (x y) (+ (square x) (square y)))
sum-of-squares <-- 返回值
(sum-of-squares 3 4)
25 <-- 返回值
```

ここで，自分で定義した関数も，+や-などの組み込まれた関数と同様に扱うことが出来る点に注意して欲しい．

A.4 条件式と述語

EmacsLisp では cond を使った制御構造を利用できる．例えば，値の絶対値を計算する手続きは以下の様に定義する．

```
(defun abs (x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

あるいは，

```
(defun abs (x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        (t      (- x))))
```

と書く．t は cond の最後の節に書くことができ，常に条件が成り立つ．

また，場合分けが2つであれば，if を使える

¹²関数の呼び出しは関数を評価することで行う．関数を評価するとその結果が値として得られる．プロンプトにはその結果が表示される．

```
(defun foo () (print 10) (+ 1 2))
```

foo というのは foo という関数定義をして foo という値が帰ってきている．ややこしいのは，関数の中で値を表示している場合である．

```
(foo)
```

```
10
```

```
3
```

とある場合，最初の 10 は関数のなかで表示された値，次の 3 は関数を評価した結果の値である．

```
(defun abs (x)
  (if (< x 0)
      (- x)
      x))
```

この一般型は

```
(if <predicate> <consequent> <alternative>)
```

とかける．述語 <predicate> が成立（評価した結果が真）すれば <consequent> を評価しその値を返し，それ以外の場合は <alternative> を評価し値を返す．

述語には $<, >, =$ などの他に, and, or, not 等がある. ここで, and, or は2項演算子でなく, 多項演算子である.

A.5 スコープ

スコープとは変数や関数が参照できる範囲のこと。ある範囲外に置いた変数などは、スコープ外となり、その名前では参照できない。

EmacsLisp では let 式を使い，ローカル変数を定義することが出来る．let 式が作るローカル変数は let 式の内側だけで値を保持する．

```
(setq x 10)
(defun foo () (print x))
(foo)
(let ((x 2)) (print x))
(defun bar (z) (let ((x 1) (y 10)) (+ x y z)))
(bar 10)
(defun bar2 (z) (let ((y 10)) (+ x y z)))
(bar 10)
```

これは以下の様にスコープを表すことができる。

```

variable x <-----|
|                  |
|  -- function foo --|
|                  |
|  |                  | |
|  |  print x  -----|-----|
|  |                  |
|  |                  |
|  |-----|
|
|  ---- let  x  -----|
|  |              ^      | |
|  |              |      |
|  |              |      |
|  |  print x      |
|  |              |

```

細かく言うとスコープには静的スコープと動的スコープがある．静的スコープは構文スコープとも呼ばれ，構文上から決定できるスコープである．一方で動的スコープでは実行時の関係から決定されるものになっている．

例えば以下のプログラムを見てみよう．

```
(defun foo () (let ((a 100)) (bar)))
(defun bar () (print a))
(setq a 10)
(foo)
```

(foo) を評価した結果が 100 になる場合は動的スコープを採用している言語，一方で 10 となる場合は静的スコープを採用している言語となる．EmacsLisp では動的スコープを採用している¹³．

A.6 反復プロセス (while 文と do 文)

プログラムのある部分を何回も実行することを繰り返し (反復プロセス) と呼ぶ．ここでは while, do を紹介する．

while 文は以下の様にかける．

```
(setq num 0)
(while (< num 4)
  (print num)
  (setq num (+ num 1)))
```

この一般型は

```
(while <condition> <body>)
```

となり，<condition> を評価した結果が真であれば <forms> を評価し再度，<condition> を評価する．

while 文以外には反復を実現する方法として，do 文¹⁴や dotimes 文等があり以下の様を書くことができる．ただし内部的にはこれらの命令は while 文に置き換えて実行されている¹⁵．

```
(do ((i 0 (+ i 1))) ((>= i 4)) (print i))
```

```
(dotimes (i 4)(print i))
```

A.7 非ローカル脱出 catch と throw

throw 関数を呼ぶと対応する catch に制御がもどり，その catch は直ちに終了する．throw 以下のコードは実行されず，catch の戻り値は throw の戻り値となる．先ほどの while 文の例は

¹³一方 clisp というコマンドで実行できる CommonLisp は静的スコープになる

¹⁴emacs-lisp で do 文を使うには (require 'cl) とする

¹⁵(pp (macroexpand-all '(dotimes (i 4)(print i)))) とするとよい

```
(setq num 0)
(catch 'loop
  (while t
    (print num)
    (setq num (+ num 1))
    (if (>= num 4) (throw 'loop nil)))))
```

と書くことが出来る¹⁶。一般形は以下の様に見える。

```
(catch <tag> <body>)
(throw <tag> <value>)
```

catch と throw を使った例としては以下の 2 重ループからの脱出があげられる。

```
(defun foo (i j) (> i (+ j 2)))
(catch 'loop
  (let ((i 0))
    (while (< i 10)
      (let ((j 0))
        (while (< j 10)
          (if (foo i j)
              (throw 'loop (list i j)))
          (setq j (+ j 1)))))
      (setq i (+ i 1)))))
```

より複雑な例として以下の例を示す。式を順番に評価し何が起きているか理解してほしい。

```
(defun catch2 (tag) (catch tag (throw 'hack 'yes)))

(catch 'hack (print (catch2 'hack)) 'no)
(catch 'hack (print (catch2 'quux)) 'no)
```

A.8 無名関数, クロージャ

関数はラムダ式と呼ばれる lambda から始まるリストで定義されている。関数を呼び出すには operator にラムダ式を, operands に引数を入れれば良い

```
(lambda (a b c) (+ a b c))
((lambda (a b c) (+ a b c)) 1 2 3)
```

ラムダ式を使うことで名前を持たない関数を定義することが出来る。funcall は関数を呼び出す関数である。

```
(setq f #'(lambda (a b c) (+ a b c)))
(funcall f 1 2 3)
(funcall #'(lambda (a b c) (+ a b c)) 1 2 3)
```

¹⁶ここで出てきた 'loop というのは, (quote loop) という式の短縮形で loop という記号 (シンボル, 識別子) を表す。これは, 変数と変数名が違うことに注意。(setq x 10) は x という変数名 (記号, シンボル, 識別子) をもつ x という変数に 10 という値を代入している。

動的スコープをもつ言語では実行時に変数等の環境が決まるが、静的（レキシカル）スコープでもクロージャ(closure)を利用することで、実行する関数とアクセス可能なレキシカル変数をまとめたものをつくることができる。

Emacs Lisp ではクロージャは対応していないが cl という機能拡張モジュールを利用すると以下の様にエミュレートすることが出来る。

```
(require 'cl)
(lexical-let ((counter 0)) (defun new-id () (setq counter (+ counter 1))))
```

後は (newid) 関数を呼ぶたびに返り値が1つつ増えていく

```
(new-id)
1 <-- 返り値
(new-id)
2 <-- 返り値
```

CommonLisp では以下の様に定義できる。CommonLisp はターミナルから clisp というコマンドで起動する。

```
$ clisp
( 省略 )
[1]>
```

インタプリタが立ち上がったら以下のようにしてクロージャを利用できる。

```
[1]> (setq new-id (let ((counter 0)) #'(lambda () (setq counter (+ counter 1)))))
#<FUNCTION :LAMBDA NIL (SETQ COUNTER (+ COUNTER 1))>
[2]> (funcall new-id)
1
[3]> (funcall new-id)
2
[4]> (funcall new-id)
3
[5]> (print counter)
*** - SYSTEM::READ-EVAL-PRINT: variable COUNTER has no value
```

B Python を使った例

Python は近年非常によく利用されているプログラム言語であり、オブジェクト試行のスクリプト言語に分類できる。本章では基本的な Python 言語の構文を紹介する。

B.1 式

```
$ python
>>> 486
486
>>> 1000 - 334
666
>>> 5 * 99
495
>>> 10 / 5
2
```

```
>>> 2.7 + 10
12.7
>>> 21 + 35 + 12 + 7
75
>>> 25 * 4 * 12
1200
>>> (3 * 5) + (10 - 6)
19
```

B.2 変数

```
>>> size = 2
>>> 5 * size
10
>>> radius = 10
>>> import math
>>> math.pi * radius * radius
314.1592653589793
>>> circumference = 2 * math.pi * radius
>>> circumference
62.83185307179586
```

B.3 関数

```
$ cat square.py
def square(x):
    return x * x

def sum_of_square(x, y):
    return ( square ( x ) + square ( y ) )
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> exec(open('square.py').read())
>>> square(21)
441
>>> square(2+5)
49
>>> square(square(3))
81
>>> square(3)+square(4)
25
>>> sum_of_square(3,4)
25
>>> def square(x): return x * x
...
>>> square(21)
441
```

B.4 条件式と述語

```
$ cat abs.py
def abs(x):
    if x > 0:
        return x
    elif x == 0:
        return 0
    elif x < 0:
        return -x

def abs(x):
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```

```
def abs (x):
    if x > 0:
        return x
    else:
        return -x

$ python
>>> exec(open('abs.py').read())
>>> abs(10)
10
>>> abs(-10)
10
```

B.5 スコープ

```
$ cat scope.py
x = 10
print(x)
def foo ():
    print(x)
foo()
def bar (z):
    x = 1
    y = 1
    return x + y + z
```

```
print bar(10)
def bar2 (z):
    y = 1
    return x + y + z
print bar2(10)

>>> exec(open('scope.py').read())
10
10
12
21
```

B.6 反復プロセス (while 文と for 文)

```
$ cat while.py
num = 0
while num < 4:
    print(num)
    num = num + 1

for num in range(4):
    print(num)
```

```
>>> exec(open('while.py').read())
0
1
2
3
0
1
2
3
```


B.7 非ローカル脱出 catch と throw

```
$ cat catch.py
num = 0
print(num)
try:
    while True:
        print(num)
        num = num + 1
        if num >= 4:
            raise ValueError("exit loop")
except ValueError as e:
    print(e)
```

```
>>> exec(open('catch.py').read())
0
0
1
2
3
exit loop
```

B.8 無名関数, クロージャ

```
>>> (lambda a, b, c: a + b + c)(1,2,3)
6
>>> apply(lambda a, b, c: a + b + c, [1,2,3]) # this is old
6
>>> f = lambda a, b, c: a + b + c
>>> f(1,2,3)
6
```

```
$ cat closure.py
def f():
    counter = 0
    def g():
        nonlocal counter # local to 'f'
        counter = counter + 1
        return counter
    return g
```

```
#c = 0
new_g = f()
print(new_g())
print(new_g())
print(new_g())

$ python3 closure.py
1
2
3
```

C JavaScript を使った例

JavaScript は Web でよく利用されているプログラム言語であり, オブジェクト試行のスク립ト言語に分類できる. 本章では基本的な JavaScript の構文を紹介するが, より詳しい内容は http://www.geocities.jp/m_hiroi/light/javascript.html などの資料を参照するとよい. 以下は js というインタプリタプログラムを利用した例であるが, FireFox などのブラウザで Tools - WebDeveloper - WebConsole からインタプリタが利用できる.

C.1 式

```
$ js
> 486
486
> 1000 - 334
666
> 5 * 99
495
> 10 / 5
2
```

```
> 2.7 + 10
12.7
> 21 + 35 + 12 + 7
75
> 25 * 4 * 12
1200
> (3 * 5) + (10 - 6)
19
```

C.2 変数

```
> size = 2
2
> 5 * size
10
> radius = 10
10
```

```
> Math.PI * radius * radius
314.1592653589793
> circumference = 2 * Math.PI * radius
62.83185307179586
```

C.3 関数

```
> function square(x) { return x * x; }
undefined
> square ( 21 )
441
> square ( 2 + 5 )
49
> square ( square ( 3 ) )
81
> square ( 3 ) + square ( 4 )
25
> function sum_of_square(x, y) { return ( square ( x ) + square ( y ) ); }
undefined
> sum_of_square (3, 4)
25
```

C.4 条件式と述語

```
> function abs ( x ) {
... if ( x < 0 ) { return - x; }
... else { return x; }
... }
undefined
> function abs ( x ) {
... if ( x > 0 ) { return x; }
... else if ( x == 0 ) { return 0; }
... else if ( x < 0 ) { return - x ; }
... }
undefined
```

```
> function abs ( x ) {
... if ( x > 0 ) { return x; }
... else if ( x == 0 ) { return 0; }
... else { return - x ; }
... }
undefined
```

C.5 スコープ

```
> x = 10
10
> function foo () { console.log( x ) }
undefined
> foo()
10
undefined
> function bar ( z ) { var x = 1; var y = 1; return ( x + y + z ); }
undefined
> bar()
NaN
> bar( 10 )
12
> function bar2 ( z ) { var y = 1; return ( x + y + z ); }
undefined
> bar2( 10 )
21
```

C.6 反復プロセス (while 文と do 文)

```
> while ( num < 4 ) {
... console.log(num);
... num = num + 1;
... }
```

C.7 非ローカル脱出 catch と throw

```
> num = 0
0
> try {
... while ( true ) {
..... console.log ( num );
..... num = num + 1;
..... if ( num >= 4 ) { throw "exit loop"; }
..... }
... } catch ( e ) {
... console.log ( e );
... }
0
1
2
3
exit loop
```

C.8 無名関数, クロージャ

```
> (function (a, b, c) { return ( a + b + c ); })(1,2,3)
6
> f = (function (a, b, c) { return ( a + b + c ); })
[Function]
> f(1,2,3)
6
> function f() { var counter = 0; function id () { counter = counter +
1; return counter; } return id; }
undefined
> var new_id = f()
undefined
> new_id()
1
> new_id()
2
> new_id()
3
```

D Scheme

Scheme は Lisp の方言の一つとして、1975 年頃に当時 MIT AI Lab の学生だったガイ・スティーラとジェラルド・サスマンにより設計された。末尾再帰や継続といった技術が採用されている。

Scheme は多くの実装が有るが、ここでは川井史郎氏が開発している Gauche(ゴーシュ) を利用してみよう。

インストールは

```
sudo apt install gauche
```

とする。インタプリタは gosh として立ち上げる。

D.1 式

```
$ gosh
gosh> 486
486
gosh> (- 1000 334)
666
gosh> (* 5 99)
495
gosh> (/ 10 5)
2
```

```
gosh> (+ 2.7 10)
12.7
gosh> (+ 21 35 12 7)
75
gosh> (* 25 4 12)
1200
gosh> (+ (* 3 5) (- 10 6))
19
```

D.2 変数

```
gosh> (define size 2)
size
gosh> (* 5 size)
10
gosh> (define radius 10)
radius
```

```
gosh> (use math.const)
gosh> (* pi radius radius)
314.1592653589793
gosh> (define circumference (* 2 pi radius))
circumference
gosh> (print circumference)
62.83185307179586
```

D.3 関数

```
gosh> (define (square x) (* x x))
square
gosh> (square 21)
441
gosh> (square (+ 2 5))
49
gosh> (square (square 3))
81
```

```
gosh> (+ (square 3) (square 4))
25
gosh> (define (sum-of-square x y)
        (+ (square x) (square y)))
sum-of-square
gosh> (sum-of-square 3 4)
25
```

D.4 条件式と述語

```
gosh> (define (abs x) (if (< x 0) (- x) x))
abs
gosh> (abs 10)
10
gosh> (abs -10)
10
```

```
gosh> (define (abs x)
        (cond ((> x 0) x)
              ((= x 0) 0)
              (else (- x))))
abs
```

D.5 スコープ

```
gosh> (define x 10)
x
gosh> (define (foo) (print x))
foo
gosh> (foo)
10
#<undef>
```

```
gosh> (define (bar z) (let ((x 1)(y 1)) (+ x y z)))
bar
gosh> (bar 10)
12
gosh> (define (bar2 z) (let ((y 1)) (+ x y z)))
bar2
gosh> (bar2 10)
21
```

D.6 反復プロセス

```
gosh> (define num 0)
num
gosh> (while (< num 4)
        (print num)
        (set! num (+ num 1)))
gosh> (do ((i 0 (+ i 1)))
        ((>= i 4) #t)
        (print i))
```

フォーム `set!` は変数のレキシカルな束縛 (バインディング) を変更する

D.7 非ローカル脱出

```
gosh> (define num 0)
num
gosh> (call/cc (lambda (break)
                (while #t
                        (print num)
                        (set! num (+ num 1))
                        (if (>= num 4) (break 'loop))))))

0
1
2
3
loop
```

D.8 無名関数, クロージャ

```
gosh> (apply (lambda (a b c) (+ a b c)) '(1 2 3))
6
gosh> (define f (lambda (a b c) (+ a b c)))
f
gosh> (f 1 2 3)
6
gosh> (define (f) (let ((counter 0)) (lambda () (set! counter (+ counter 1)))))
f
gosh> (define new-id (f))
new-id
gosh> (new-id)
1
gosh> (new-id)
2
gosh> (new-id)
3
```

D.9 末尾再帰 (tail recursion)

再帰関数のうち、自分自身の関数が、その計算の一番最後に評価される物を末尾再帰、と呼ぶ。末尾再帰のコードは関数の戻り先をスタックに保存することなく実行する最適化が可能であり、Scheme は仕様で最適化を実施するパターンを形式的に定めている。

```
(define (fact-t n r)
  (if (<= n 1)
      r
      (fact-t (- n 1) (* n r))))
```

```
(define (fact n)
  (if (= n 1)
      1
      (* n (fact (- n 1)))))
```

lua 言語も末尾再帰最適化をサポートしている。Lua は 1993 年にリオデジャネイロ・カトリカ大学のコンピュータ科学科の Roberto Ierusalimsky, Waldemar Celes, Luiz Henrique de Figueiredo により開発されたスクリプト言語であり、C 言語のホストプログラムに組み込まれることを目的に高速動作、高移植性、組み込み容易さを特徴としている。Lua のコードはバイトコードにコンパイルされ Lua VM で実行される。

```
sudo apt-get install lua5.3
```

としてインストールし lua として実行できる。ファイルの読み込みとして dofile 関数がある。

```
$ lua
> dofile("myfile.lua")
> print(fact_r(10))
> print(fact_t(10, 1))
```

```
function fact_t(n, r)
  if n <= 1 then
    return r
  else
    return fact_t((n - 1), (n * r))
  end
end
```

```
function fact_r(n)
  if n == 1 then
    return 1
  else
    return (n * fact_r(n - 1))
  end
end
```

D.10 継続 (continuation)

継続とは「次に行われる計算」

```
(define (foo) (display "foo"))
(define (bar) (display "bar"))
(define (baz) (display "baz"))
(define (test) (foo) (bar) (baz))
```

とあると、(test) は (foo), (bar), (baz) を順番に呼び出す。foo を呼び出した後の「継続」は (bar), (baz) になる。

```
(* (+ 1 2) (- 10 5))
```

の場合は (+ 1 2) の評価が終わった段階の継続は、この評価の結果を a とすると、(* a (- 10 5)) となる。また、2 が評価された時点の継続は、この結果を a (=2) とすると、(* (+ 1 a) (- 10 5)) となる。

call/cc を用いることで、この継続を取り出すことができる。

```
(call/cc (lambda (cc) [式 . cc は call/cc が呼ばれた時点での「継続」が入る]))
```

であるから、上記の式は

```
(* (+ 1 (call/cc (lambda (cc) 2))) (- 10 5))
```

とかける。継続は保存することができるので、

```
gosh> (define *save* #f)
*save*
gosh> (* (+ 1 (call/cc (lambda (cc) (set! *save* cc) 2))) (- 10 5))
15
gosh> (*save* 2)
15
gosh> (*save* 10)
55
```

となる．以下のような lambda 式で表される関数が継続として *save* に保存されていると考えて良い．ただし、トップレベル以外で呼ばれた場合は、その時点でトップレベルに戻るという振る舞いをする．

```
gosh> ((lambda (a) (* (+ 1 a) (- 10 5))) 2)
15
gosh> ((lambda (a) (* (+ 1 a) (- 10 5))) 10)
55
```

```
gosh> (*save* 5)
30
gosh> (/ (*save* 5) 0)
30
```

参考文献

http://www.geocities.jp/m_hiroi/func/abscscm20.html,

<https://practical-scheme.net/wiliki/wiliki.cgi?Scheme:使いたい人のための継続入門>

<http://www.shido.info/lisp/callcc.html>

E リスト

リストは基本的なデータ型であり現在のプログラミング言語では標準で使えるものが多い¹⁷．Lisp, Python でそれぞれリストを作るのには以下のようにする¹⁸．

```
(setq a '(1 2 3 4))
(1 2 3 4)
(setq a (list 1 2 3 4))
(1 2 3 4)
(setq a '(this is a pen))
(this is a pen)
```

```
>>> a = [1, 2, 3, 4]
>>> a
[1, 2, 3, 4]
>>> a = ['this', 'is', 'a', 'pen']
>>> a
['this', 'is', 'a', 'pen']
```

リストの基本処理として、リストの長さ返す、連結、リバース、ソートがある．

```
> (length '(1 2 3 4))
4
> (append '(1 2) '(3 4))
(1 2 3 4)
> (reverse '(1 2 3 4))
(4 3 2 1)
> (sort '(1 2 3 4) #')
(4 3 2 1)
```

```
>>> len([1, 2, 3, 4])
4
>>> sorted([1,2,3,4],reverse=True)
[4, 3, 2, 1]
>>> a = [1, 2]
>>> a.extend([3,4])
>>> a
[1, 2, 3, 4]
>>> a = [1, 2, 3, 4]
>>> a.reverse()
>>> a
[4, 3, 2, 1]
```

¹⁷最新の C++0x/C++11 対応の C コンパイラでもリストは容易に利用できる

¹⁸Python と Lisp の比較は <http://norvig.com/python-lisp.html>

Python では len, sorted などの関数と extend, reverse 等のメソッドが混在している。さらに、スタックやキューを作るためのメソッドも準備されている。

```
> (setq a '(1 2 3 4))
(1 2 3 4)
> (pop a)
1
> a
(2 3 4)
> (push 1 a)
(1 2 3 4)
> a
(1 2 3 4)
```

```
>>> a = [1, 2, 3, 4]
>>> a.pop()
4
>>> a
[1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
>>> a.pop(0)
1
>>> a
[2, 3, 4]
```

これらの更に基本的な処理として、リストの第一要素を取り出す関数を car、リストの第一要素を取り除いたリストを解す関数を cdr と呼ぶ、それぞれ Contents of the Address part of the Register、Contents of the Decrement part of the Register の略であり、機械語の名残である¹⁹。

最新の言語仕様では car, cdr の代わりに同じ仕様を持つ first, rest の利用が推奨されている。また、Python では a[0], a[1:] として同様の処理を行う事が出来る²⁰。EmacsLisp では first, second caadr の代わりに cl-first, cl-second cl-caadr を利用すること

```
> (car '(1 2 3 4))
1
> (cdr '(1 2 3 4))
(2 3 4)
> (car '(this))
this
> (cdr '(this))
nil
> (cdr '(this is))
(is)
> (cdr '(this is a))
(is a)
> (car nil)
nil
> (cdr nil)
nil
> (car '((1 2 3 4) (this is a pen)))
(1 2 3 4)
> (cdr '((1 2 3 4) (this is a pen)))
((this is a pen))
> (car (cdr '((1 2 3 4) (this is a pen))))
(this is a pen)
> (car (car (cdr '((1 2 3 4) (this is a pen)))))
this
> (caadr '((1 2 3 4) (this is a pen)))
this
> (cadr '(1 2 3 4))
2
> (second '(1 2 3 4))
2
```

```
>>> a = [1, 2, 3, 4]
>>> a[0]
1
>>> a[1:]
[2, 3, 4]
>>> ['this'][0]
'this'
>>> ['this'][1:]
[]
>>> ['this', 'is'][1:]
['is']
>>> ['this', 'is', 'a'][1:]
['is', 'a']
>>> a = []
>>> a[0]
IndexError: list index out of range
>>> a[1:]
[]
>>> [[1, 2, 3, 4], ['this', 'is', 'a', 'pen']][0]
[1, 2, 3, 4]
>>> [[1, 2, 3, 4], ['this', 'is', 'a', 'pen']][1:]
[['this', 'is', 'a', 'pen']]
>>> [[1, 2, 3, 4], ['this', 'is', 'a', 'pen']][1:][0]
['this', 'is', 'a', 'pen']
>>> [[1, 2, 3, 4], ['this', 'is', 'a', 'pen']][1:][0][0]
'this'
```

car, cdr がリストの分解だとすると、リストの合成を行う基本関数は cons になる²¹。car, cdr で分解されたリストは cons で合成することができる。(cons 1 '(2 3 4)) -> (1 2 3 4) car, cdr, cons の実行過程をセルとしてみると以下のように考えることが出来る。

¹⁹address は計算機のアドレスだが decrement は何であるか、調べてみると良い

²⁰<http://legacy.python.org/dev/peps/pep-3132/> も読んでみる

²¹constructs の略

```

car
|
| cdr
| |
[ | ]->[ | ]->[ | ]->[ | ]->nil
|       |       |       |
1       2       3       4

(car '(1 2 3 4)) => 1
(cdr '(1 2 3 4)) => (2 3 4)

```

```

[ | ] -> nil
|
1
(cons 1 nil) -> (1)

[ | ] -> [ | ]->[ | ]->[ | ]->nil
|       |       |       |
1       2       3       4
(cons 1 '(2 3 4))

```

car, cdr を使って append, reverse, length は以下のように定義することが出来る .

```

(defun my-length (a) (if (null a) 0 (+ 1 (length (cdr a)))))
(defun my-append (a b) (if (null a) b (cons (car a) (my-append (cdr a) b))))
(defun my-reverse (a) (if (null a) nil (append (my-reverse (cdr a)) (list (car a)))))

```

(cons 0 1) は (0 . 1) となるがこれをドット対と呼ぶ . リストは終端の cdr 部が nil になっているものであり , それが nil である必要はない . (cons 0 nil) -> (0)

関数には「破壊的」な関数と「非破壊的」な関数がある . (reverse a) では a の値は変更されないが . (nreverse a) では a の値は変更されてしまう²² . Python の場合も a.sort() は a の値を変更するが sorted(a) は変更しない .

```

> (setq a '(1 2 3 4))
> (reverse a)
(4 3 2 1)
> a
(1 2 3 4)
> (nreverse a)
(4 3 2 1)
> a
(1)
> (setq a '(1 2 3 4))
> (setq a (nreverse a))
(4 3 2 1)
> a
(4 3 2 1)

```

```

>>> a = [1, 2, 3, 4]
>>> a.sort(reverse=True)
>>> a
[4, 3, 2, 1]
>>> sorted(a)
[1, 2, 3, 4]
>>> a
[4, 3, 2, 1]
>>> a = sorted(a)
>>> a
[1, 2, 3, 4]

```

リストの繰り返しとして dolist や for ループが用意されている

```

> (dolist (x '(1 2 3 4))
>   (print x))
1
2
3
4

```

```

>>> for x in [1,2,3,4]:
...     print(x)
...
1
2
3
4

```

これ以外に map 関数がある . 例えば mapcar は mapcar func &rest arg-list と定義されており , arg-list のそれぞれの要素に func を適用しその全ての結果のリストを作る . 無名関数を利用すると便利である .

²²このとき a は元の a の逆順にもなっていない点に注意

```
> (mapcar #'(lambda (x) (sort x '<))
  '((2 1) (4 3 2 1)))
((1 2) (1 2 3 4))
> (mapcar #'car '((1 2) (2 3)))
(1 2)
```

```
>>> map(sorted, [[2, 1], [4, 3, 2, 1]])
[[1, 2], [1, 2, 3, 4]]
>>> map(lambda x: x[0], [[1, 2], [2, 3]])
[1, 2]
```

リストに関する繰り返し関数として map 以外に reduce, filter がある。reduce は 2 つの引数を取る関数 f と、リストを引数にとり、リストの各要素を関数を

```
(a1 a2 a3 a4 a5)
=> f( f( f( f( a1, a2 ), a3 ), a4 ), a5 )
```

のように適用する。これを畳み込みと呼ぶ²³。

また、filter は引数のリストの内、関数が返した要素のみから新しいリストを作成する。mapcar は関数とリストを引数にするが、関数が返したリストを append して返す。なお、EmacsLisp 上では mapcar 関数は cl-reduce, cl-mapcar という名前になっている。

```
(reduce #' + '(1 2 3 4))
10
(mapcar #'(lambda (x)
  (if (= (mod x 2) 1) (list x))) '(1 2 3 4))
(1 3)
```

```
>>> reduce(lambda x, y: x+y, [1, 2, 3, 4])
10
>>> filter(lambda n:n%2==1, [1, 2, 3, 4])
[1, 3]
```

リストの探索は member を使う²⁴。

```
> (member 0 '(1 2 3 4))
nil
> (member 3 '(1 2 3 4))
(3 4)
> (member-if #'(lambda (x) (= (car x) 2)) '((1 2) (2 3) (3 4) (4 5)))
((2 3) (3 4) (4 5))
```

```
>>> 0 in [1, 2, 3, 4]
False
>>> 3 in [1, 2, 3, 4]
True
```

連想リストは Lisp ではドット対を要素とするリストであり、car がキー、cdr がバリューになる。連想リストからキーを用いてバリューを求める関数が assoc である。Python ではディクショナリ型として利用できる。

```
> (setq a '((a . 1)(b . 2)(c . 3)(d . 4)))
((a . 1) (b . 2) (c . 3) (d . 4))
> (assoc 'c a)
(c . 3)
> (cdr (assoc 'c a))
3
> (setq a (append a (list (cons 'e 5))))
((a . 1) (b . 2) (c . 3) (d . 4) (e . 5))
```

```
>>> a = {'a':1, 'b':2, 'c':3, 'd':4}
>>> a['c']
3
>>> a['e'] = 5
>>> a
{'a': 1, 'c': 3, 'b': 2, 'e': 5, 'd': 4}
```

²³MapReduce と呼ばれるフレームワークが存在するので調べてみると良い

²⁴EmacsLisp では member-if は cl-member-if

```
== test.l ==  
(print "input vlaue ? ")  
(setq x (parse-integer (read-line)))  
(print (+ x 10))  
=====
```

\$ clisp
[1]> (load "test.l")
"input vlaue ? " 10
20

```
== test.py ==  
x = int(raw_input("input value ? "))  
print(x+10)  
=====
```

>>> exec(open('test.py').read())
input value ? 10
20