計算機科学基礎'13 ＃3 – ソフトウェア開発

久野　靖*

2013.4.24

今回は「ソフトウェア」の2回目として、前回取り上げられなかったユティリテリの話をまずした後、ソフトウェアとはどのようにして作られるか、ソフトウェアの中心となる「考え方」とはどのようなものか、ソフトウェア開発においてどんなことが問題になるのか、について取り上げて行きます。

1　シェルとフィルタ

1.1　コマンド入力のユーザインタフェース

我々の使っている Unix 環境では、コマンドを打ち込むとそれに応じてプログラムが実行される、という形でユーザがシステムに指示を与えています。この方式を**コマンド行インタフェース** (command line interface、CLI) と呼びます。そして、コマンドを解釈してそれに対応する動作を起動してくれるプログラムがコマンドインタプリタでした。[1]この方式について「コマンドを覚えるのが大変だ」「コマンドを打ち込むのが負担だ」「だから Unix は使いづらい」と思われている人が多いかも知れません。これに対し、今日の PC ソフトの多くは GUI によって操作しますが、GUI は「使いやすい」でしょうか? 「使いやすい」とは、厳密にはどういう意味なのでしょうか?

たとえば、「操作時間が短い」というのは効率につながりますから、「使いやすさ」の1つの指針だと思っていいですよね? 実際に時間を計測してみると、たとえばファイルを削除するのに (GUI でよくある)「ファイルアイコンを選択後、メニューを出して『削除』を指定」という操作と、Unix のコマンド「`rm` ファイル名 `RET`」を比べた場合、後者の方が圧倒的に速いのが普通です (図 1)。

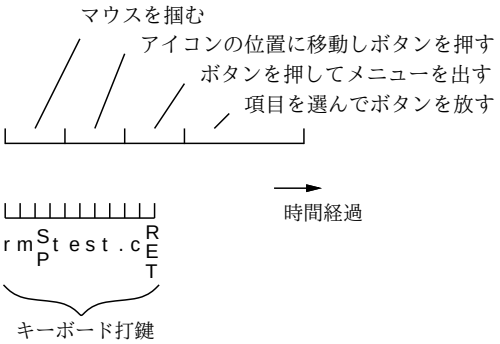

図 1: アイコン+メニュー操作とコマンド入力の時間比較

なぜそうなるかというと、人間がマウス等で画面上の対象を選んだり、メニュー項目を選択するには、少なくても 1 秒くらい、普通は 2〜3 秒の時間が掛かるからです。これに対し、キーボードは習熟すると 0.2 秒くらいで 1 文字打てるので、コマンド 1 行を打っても全部で 1〜2 秒で済みます。

だからコマンドの方がいい、というわけではありません。コマンドは覚えていないと何を打っていいか分かりませんが、GUI だと画面を見たりメニューを出してみることでやり方が分かることもあ

*経営システム科学専攻

[1]Unix では伝統的に、コマンドインタプリタのことを「シェル」と呼びます。利用者を囲って守ってくれる「貝殻」になぞらえてそう呼ぶようになったそうです。

ります。一般に、GUI はカジュアルユーザ (あまり長時間は使わない人) 向け、CLI は熟練ユーザ (頻繁につかっていて短時間で操作できる効率が欲しい人) 向けと言われています。

演習 3-1 コマンドインタフェースと GUI で同じ作業を行う時の所要時間を比較するため、以下の 3 つのことがらから 1 つ以上 (できれば全部) やってみなさい。計測は、「キーボードに手が置かれている状態」から始め、最後に「キーボードに手が置かれている」状態で終わること。

a. コマンドでファイルを削除するのと、GUI でファイルを削除するので、時間を比較する。できれば、GUI での削除は 2 つ以上の方法について行うとなおよい。
b. コマンドであるプログラムを起動するのと、GUI であるプログラムを起動するので、時間を比較する。できれば、GUI での起動は 2 つ以上の方法について行うとなおよい。
c. 上の a または b(または両方) について、同じ操作を何回も行った場合に習熟によって時間が短くなるか、なるとしたらどれくらいかを調べる。できれば、「最終的に」どれくらいまで速くなるのか予測できるとなおよい。

いずれも、操作する人と計る人で組になってやるのがよい。何を計ることを意図したか/どのような実験を行ったかの説明や、得られたデータをきちんと書くこ。

「◎」の条件: (1) 小問を 2 つ以上解答しており、(2) いずれも「できれば」の部分までやってあって、(3) 適切な (と担当が考える) 分析がなされていること。

1.2 コマンドとは?

そもそも、コマンドとは一体何でしょう? ps コマンドの表示を見ると ps というプログラムがプロセスとして動いているのが観察できました。emacs、xcolck などもやはり同名のプログラムでした。つまりシェルでは、コマンドとはプログラムの名前なわけです。より正確に言うと「プログラムの実行形式が格納されているファイルの名前がコマンド名でもある」ということになります。

たとえば gcc で C のソースプログラムをコンパイルすると、a.out という実行形式ファイルになりました。そしてその実行形式を動かすには、ファイルの名前 a.out をコマンドとして打ち込みましたね。では、a.out ではなく別の名前だったらどうでしょうか?

```
% echo 'main() { puts("hello."); }' >test.c
% gcc test.c          ←コンパイルする
% ls
a.out   test.c        ←実行形式ファイル: a.out
% a.out               ←ファイル名を打つと実行
hello.
% mv a.out hello      ←ファイル名を変更してみる
% ls
hello   test.c        ← hello という名前に変更
% hello               ←ファイル名を打つと実行
hello.
%
```

つまり、ファイルの名前を取り換えれば、その取り換えた名前が新しいコマンドの名前になるわけです。OS の目的を「プログラムを実行させること」と考えるなら、「実行したいファイルの名前を言うことがすなわちコマンド」というシェルの方針はとても明快だと言えるでしょう。

それにしても、自分は ls とか ps とかいうファイルは持っていないけど、と思うかも知れません。もちろん、これら共通のコマンドの実行形式ファイルを各自が持つのではディスクの無駄ですから、共通のコマンドに対応する実行ファイルは共通の場所 (/bin、/usr/bin、/usr/local/bin など) に

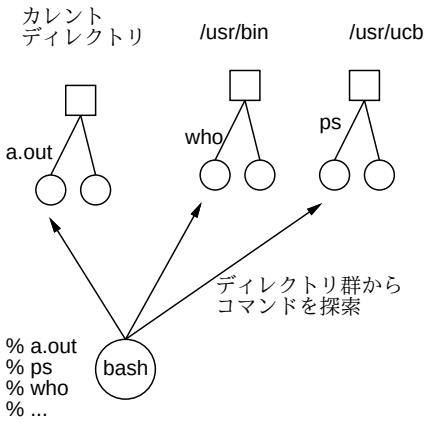

図 2: シェルによるコマンド探索

おいてあり、シェルはそこを順番に探して実行形式ファイルを見つけます。具体的にどこにあるかを知りたければ `type` というコマンドを使います。

- `type` コマンド名 — コマンドの種別やありかを表示

```
% type ls
ls is /bin/ls
%
```

つまり `ls` というコマンドは`/bin` というディレクトリにあると分かったわけです。[2]

1.3 コマンドの組み合わせとリダイレクション

シェルの特徴の 1 つは、1 つのコマンド行で複数のプログラムの実行を指示したり、それらのプログラム群の入出力関係などを制御できることです。まず、1 つのコマンドに対してその入力や出力の接続先を切り替えるには**リダイレクション** (redirection) を用います。

コマンド <入力ファイル
コマンド >出力ファイル
コマンド >>出力ファイル

出力の場合、「>」では既に出力ファイルに内容が入っている場合には一担からっぽにされますが、「>>」では「末尾に追加」になります。さらに、あるコマンドの出力を別のコマンドの入力に接続することもできます。これを**パイプライン** (pipeline) と呼びます。

コマンド`1` | コマンド`2` | ... | コマンド`n`

1.4 ユティリティとフィルタ

システムソフトのうち、汎用的な作業をこなすものをユティリティと呼ぶのでしたね。そして「小さな政府」「大きな政府」という概念があるのと同様、ユティリティにも「大きなユティリティ」「小さなユティリティ」があります。大きなユティリティとは、1 つのプログラムに多数の機能をつけることを言います (図 3 右)。機能が豊富だと便利そうですが、次の弱点もあります。

[2]なお、この「順番に探す」という動作はコマンドに「/」が含まれている場合には行われません。「/」が入っている場合は、ファイルを直接指定しているものとして扱われます。

- どの機能を使うかといった指定が沢山必要で、使い方が複雑になりがちである。
- どれか 1 つの機能だけ使いたいときでも全機能を備えたプログラムが動くので遅くなりがちだし計算機資源の無駄づかいである。
- 機能をちょっと増やすとか訂正するといったことは、その巨大なプログラムを直さなければならず面倒だし実際上不可能なこともある。

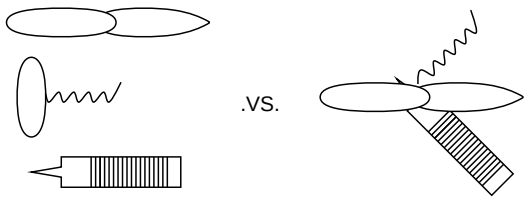

図 3: 単機能の道具と多機能の道具

これに対し、小さなユティリティとは、各プログラムは単純な機能だけ持ち、それらを組み合わせて多様な仕事をこなす、という考え方です (図 3 左)。その利点は前の裏返しです。

- 1 つのユティリティはごく単純で使い方もすぐわかる。
- 使いたい機能に対応するユティリティだけ動かせば済む。
- 足りない機能があったら、その機能だけを行う小さなプログラムを書いて追加すれば既存のユティリティと組み合わせて使える。

Unix は伝統的に「小さいユティリティ」文化です。そこでは、単機能のユティリティは標準入力からデータを読み込んで標準出力に結果を出力します。これを (水や空気を漉すのになぞらえて) **フィルタ** (filter) と呼びます。そして、複数のフィルタを接続するのにパイプラインを使うわけです。以下では例として grep 属と sed の 2 つを説明します。これらは**正規表現** (regular expression) と呼ばれる記法を活用する代表的なフィルタです。正規表現は文字列の「パターン」を指定するためのもので、ソフトウェアのさまざまな場面で多く用いられます。

1.5 grep 族

grep、**fgrep**、**egrep** の 3 つのコマンドはいずれも「入力のなかにあるパターンを含む行があったら、その行全体を出力する」機能を提供する同類のフィルタです (指定方法は 3 つとも同じ)。

- `grep` パターン ファイル… — ファイル中でパターンを含む行を出力

オプションも次のものが 3 つ共通に使えます。

- `-v` — パターンを「含む行」の代わりに「含まない行」を打ち出す。
- `-n` — 行を打ち出す際に行番号を一緒に打ち出す。

3 つの違いはパターンとして書けものの違いです。`fgrep` の場合、パターンは単純な文字列です。たとえば「`that`」をいつも「`taht`」打ってしまう人がそれをチェックしたければ次のようにします。

```
% fgrep taht wrong.txt
What is taht?
%
```

しかし、「`That`」のように文の頭にくるときは大文字なのでこれではみつかりません。そのような場合には `grep` のパターンを使えばよいのです。

表 1: grep/egrep が扱うパターン (正規表現)

パターン	説明	注記
c	c という文字そのもの	
[⋯]	⋯のうちどれか 1 文字	
.	任意の 1 文字	
^α	行の先頭の α	
α\$	行の末尾の α	
α?	α または空	grep では α は 1 文字のみ
α*	α というパターンの 0 個以上の繰り返し	〃
α+	α というパターンの 1 個以上の繰り返し	〃
(⋯)	くくり出し	egrep のみ
α|β	α または β	〃
\(⋯\)	⋯のところを一時的に覚える	grep のみ
\1、\2	覚えたものの 1 番目、2 番目、⋯	〃

```
% grep '[Tt]aht' wrong.txt
Taht is a cat.
What is taht?
%
```

「[⋯]」は「⋯」の部分のどれか 1 文字にマッチするパターンです。[3]では、fgrep の存在意義は何でしょうか? それは、たとえば「[」という字を探したければ fgrep で探すほうが簡単なわけです。grep で探したい場合には「\[」のように前に「\」をつけなければなりません。

さて、that だけでなく this も thsi と打ってしまう人が両方探したい場合はどうでしょうか。そのときは grep でも力不足で、egrep で次のように指定します。

```
% egrep '[Tt](aht|hsi)' wrong.txt
Taht is a cat.
What is taht?
Thsi isn't a dog.
% ...
```

丸かっこは「くくり出し」を、縦棒は「または」を表わしています。この丸かっこと縦棒が egrep で加わった機能なわけです。パターンの書き方を表 1 にまとめました。

興味深い練習として、/usr/share/dict/words という英単語が多数入ったファイルからパターンに合った単語を取り出してみるとよいでしょう。[4]

```
% grep 'パターン' /usr/share/dict/words | less
```

おもしろそうなパターンの例をあげておきます。

'[aeiou][aeiou][aeiou]'	母音が 3 つ続く単語
'tion$'	末尾が tion で終わる単語
'^z'	z で始まる単語
'^.........$'	長さ 10 文字の単語
'\(...\)\1'	3 文字の反復を含む単語
'\(.\)\(.\).\2\1'	5 文字の回文を含む単語

[3]括弧や空白などを指定するときは、「'⋯'」(シングルクォート) で囲む必要があるのに注意してください。
[4]less を使うのは、たくさんあてはまったとき、1 画面ずつ止まりながら見るためです。

1.6 sed

上の例では間違った箇所は見つかりましたが、それを直すことはできません。エディタで直してもいいですが、100 も 200 もあったら大変ですね。フィルタの中にある、文字列の置き換えを行うコマンド **sed** がこのような場合の道具です。

- `sed` コマンド ファイル… — コマンドに従って入力を加工する

たとえば上の例題は次のようにしてできます。

```
% cat wrong.txt
What is taht?
% sed 's/taht/that/' wrong.txt
What is that?
%
```

この「`s`」(substitute) コマンドだけ覚えておけばほとんど十分でしょう。なお、ただの `s` コマンドは 1 行に 1 回しか置き換えを行ないませんが、`taht` が全部 `that` になるまで繰り返しやりたければ「`'s/taht/that/g'`」のように末尾に「`g`」をつけてください。

たったこれだけ…? と思うかもしれませんが、実はこの「`s`」コマンドによる置き換え指定のなかには `grep` と同じパターンが書けるので、これだけでかなり強力な修正ができます。

```
% cat test.txt
a 21
is 10
this 3
% sed 's/\(.*\) (.*\)/\2 \1/' test.txt
21 a
10 is
3 this
```

これはどう読むかというと、「入力行を任意の文字列 1 と、空白と、また別の任意の文字列 2 にマッチさせ、それ全体を 2、空白、1 の順でつなげたものに置き換える」という意味になるのです。このように、Unix のパターンはそれ 1 つで強力な処理が指定でき、それを搭載したフィルタを組み合わせることで多様な処理が行えるわけです。

演習 3-2 フィルタに関する以下の設問から 1 つ以上 (できれば全部) やってみなさい。

a. 英単語が多数入ったファイル (たとえば`/usr/share/dict/words`) を材料に、grep 族を利用して以下から 3 つ以上調べてみなさい。できれば、全部やるとなおよいでしょう。

- 途中に「otion」が含まれている単語の例。[5]
- 「aho」というつづりと「ya」というつづりが両方含まれている単語。[6]
- 末尾が「otion」で終わる単語で、「e」が含まれないようなもの。[7]
- 先頭が「z」で最後が「tion」で終わる単語。[8]
- 母音を 5 つ連続して含む単語。[9]
- 5 文字のまったく同じ文字の並びが 2 回出て来る単語。[10]

[5]ヒント: 「otion」の前後に任意の文字が 1 つあればいいわけですね。
[6]ヒント: まず「aho」が含まれているものを取り出し、その出力の中からさらに「ya」が含まれているものを探します。
[7]ヒント: 「終わる」は、grep の$の機能を使い、さらにパイプで「-v」つき grep で「e」を排除すればいいのです。
[8]ヒント: 途中に任意文字が 0 個以上あるわけです。
[9]ヒント: 母音とは「a」「e」「i」「o」「u」のどれかですね。
[10]ヒント: 5 文字の並びを覚えて、そのあとに任意文字が 0 個以上あり、その後で覚えた並びがあればいいですね。

b. sed を使って、以下から 2 つ以上やってみなさい。できれば、全部やるとなおよいでしょう。

- This や this を Thsi や thsi、That や that を Taht や taht と打ってしまう可哀想な人の間違いを修正する。[11]
- This や this をすべて That や that に、逆に That や that をすべて This や this に一括して修正する。[12]
- ファイルの各行の頭にある空白の数を 2 倍にする (0 個なら 0 個、1 個なら 2 個、2 個なら 4 個…)。[13]
- `ls -l` の出力からファイルの名前と大きさ (バイト数) の部分だけを抜き出して表示する。[14]
- 上と同じだが、ただし名前が左側、バイト数が右側に来るようにして、なおかつバイト数を右そろえする。[15]

c. 打ち間違いをする人が、どのようなパターンのときに間違えるのかを分析するため、grep と sed を組み合わせて、「Thsi、thsi、That、taht」が含まれている行を取り出した後、その前や後ろにある単語と一緒に (ただし並び順は変えないで) 打ち出すようにしなさい。できれば「Thsi、thsi、That、taht」の位置が縦に揃うように空白を入れるとなおよいでしょう。

「◎」の条件: (1) 小問を 2 つ以上解答しており、(2) いずれも「できれば」の部分までやってあって、(3) どのようにやったかの適切な (と担当が考える) 解説がなされていること。

2 プログラミングとアルゴリズム

2.1 低水準言語と高水準言語

第 1 回で CPU とはどのようなものかを学びました。その内容は、次のようにまとめられます。

- CPU はメモリに格納された命令を順番に取り出し実行していく。
- このメモリに格納された命令の並びがプログラムである。
- それぞれの命令はごく単純な機能しか持たない。それを非常に多数、高速に、間違いなく実行していくことが、コンピュータの本質である。

裸のプログラムは、命令やメモリ上の場所に名前がついていて、それらの名前や数値を組み合わせて 1 つずつの命令を記述し、それを並べて行く、というものでした。このような記法を**アセンブリ言語**と呼ぶのでした。アセンブリ言語による記述は CPU が持つ命令を 1 つずつ正確に指定していくという点では明確なのですが、書くのも読むのもとても大変です (それでも、CPU の命令を直接 16 進法などで書き表すよりはだいぶ楽ですけれど)。

そこで、コンピュータがいくらか発達してくると、もっと人間にとって分かりやすい書き方を使ってプログラムを記述し、その記述したプログラムを「あるソフトウェアを使って」アセンブリ言語のプログラムに変換して、その先はこれまでと同じように動かす、という方法が使われるようになりました。この、プログラムを作成するときに用いる「人間にとって分かりやすい書き方」のことを**高水準言語** (high level language) と呼びます。たとえば、前回とりあげた「小さなコンピュータ」のアセンブリ言語で「2 つの数を合計する」プログラムは次のように書いていました。

[11] ヒント: sed の出力をパイプでまた sed に接続することで、いくつもの置き換えを指定できます。
[12] ヒント: 最初の置き換え先を「**%%%**」等の目印にしておき、2 番目の置き換え後に本来のものに置き換え直します。
[13] ヒント: 行頭にある空白の列を覚えて 2 回出力すればできます。
[14] ヒント: 地道に名前や大きさの部分だけ取り出すパターンを作るだけです。
[15] ヒント: 揃えるには、十分多くの空白にしてから、長すぎるものを削除します。

```
      load X
      add Y
      store Z
      stop
X:    5
Y:    3
Z:    0
```

ただし、データは最初からメモリに入っていて、結果もメモリに入れて終わるようになっています (実際にはコンピュータの「外部」とデータをやり取りしなければなりませんが、その部分は非常に厄介なので省略してきました)。

これに対し、FORTRAN という高水準言語では同じことを次のように書きます (FORTRAN はコンピュータの歴史上最初に作られた高水準言語です)。

```
integer i, j, k    # 変数の宣言 (場所の確保)
read *, i          # 変数 i に値を読み込む
read *, j          # 変数 j に値を読み込む
k = i + j          # i の値と j の値の和を計算し、k に入れる
print *, k         # k の値を出力
stop
end
```

「大して変わらない」と思いますか? これだけであればそうかも知れませんが、FORTRAN のような高水準言語で書いたプログラムには次のような特徴があります。

a. 「変数」「計算式」「制御文」などの形で、処理内容を分かりやすく/短く書くことができる。

b. データの読み込み、書き出し、格納などについて直接コンピュータの命令で記述するよりも簡潔で使いやすい機能を提供する。

c. 1 つのプログラムを、さまざまな CPU の命令に変換して実行できる。

なお、上で出て来る「=」は等しいという意味ではなく、「その場所に値を (store 命令などを使って) 格納する」という意味になります。これを**代入** (assignment) と呼びます。数学と違うので混乱の元なのですが、今日の言語の多くは代入を「=」で表します。

実は「高水準言語」の定義は、特定のコンピュータの機種に依存しないようなプログラムの書き方、ということです (c に対応)。ただ、特定のコンピュータの命令を書かないで済むとすれば、人間にとって書きやすくするための工夫を色々取り込むことができ、その結果 a や b が達成されるわけです。

ちなみに、アセンブリ言語やコンピュータの命令そのものを記述する形のプログラム (機械語) は、個々の CPU の種類に直接依存することから**低水準言語** (low level language) と呼びます。高水準言語を実際にコンピュータ上で実行するためには、**言語処理系** (language processor) が必要です。言語処理系は大きく分けて 2 つの方式に分類されます。

- **コンパイラ** (compiler) — 高水準言語の記述を低水準言語に変換する。変換後の低水準言語は (必要ならアセンブラを経て)CPU が直接実行する。
- **インタプリタ** (interpreter) — 高水準言語の記述を直接解釈して記述する動作を実行する。

この 2 方式を組み合わせた方式もあります (仮想的な機械語に変換し、その機械語をインタプリタ実行するなど)。ともかく、ある言語 (たとえば FORTRAN) を特定のコンピュータ/OS 上で動かすには、その OS 上に FORTRAN の言語処理系があればよいのです。

2.2 JavaScript 言語入門

今回は皆様に、JavaScript という高水準言語を用いて、プログラムを作る経験をして頂きます。この言語は言語処理系が Web ブラウザに標準的に内蔵されているので、どこででも簡単に動かせるという特徴があります。

ただしそれでも通常は、JavaScript コードを HTML に埋め込んでブラウザに読ませるなどの処理が必要なのですが、ここでは簡単に練習していただくため、ブラウザ内の入力欄に直接 JavaScript コードを打ち込めるページを作成しました (図 4)。使い方は簡単で、左側の欄に JavaScript コードを打ち込み、「Run」ボタンを押すと実行が始まり、出力などは右側の欄に表示されます。

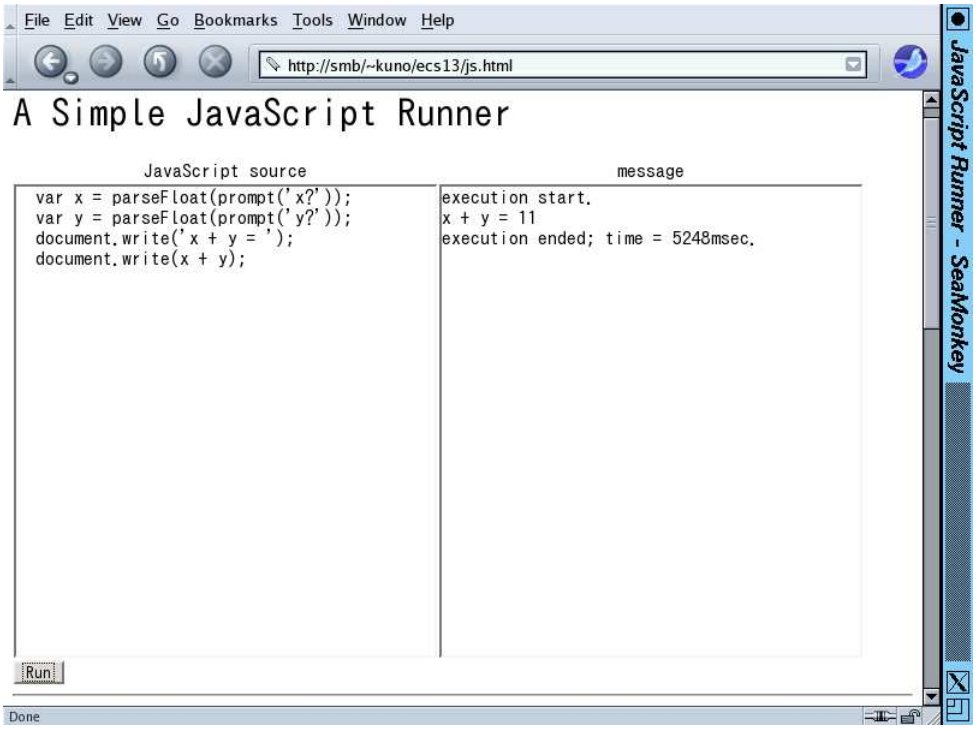

図 4: JavaScript 実行用のページ

ではさっそく、2 つの数の和を計算して表示する例題を見て頂きましょう。

```
var x = parseFloat(prompt('x?'));
var y = parseFloat(prompt('y?'));
document.write('x + y = ');
document.write(x + y);
```

ここで「`var x = ...`」というのは、`x` という名前の変数 (値を入れておく場所) を用意し、そこに「`...`」の部分で指定した値を入れる、という意味です。先に書いたように、「`=`」は等しいという意味ではなく「代入する」という意味です。そのほかはすべて「関数名 (…)」という形をしたものですが、これらはいずれも、JavaScript 環境に予め用意されているものです。上のものも含め、今回使うものの一覧を表 2 に示しておきます。

実際に実行すると、`prompt(...);` を実行したところでメッセージを表示したダイアログボックスが現れ、そこに文字列を打ち込んで OK を押すと先に進む、というふうに動いていきます。`prompt(...)` で打ち込んだ文字列は `parseFloat(...)` によって数値に変換され、変数 x と y に入ります。次に、「`x + y =`」というメッセージを表示し、それに続いて x+y を計算した結果を表示します。このように、足し算は「足し算の式」のように書けるのが、高水準言語のよい所です。

表 2: JavaScript のライブラリ関数一覧

書き方	説明
prompt(s)	s を表示して文字列を入力してもらいそれを返す
parseFloat(s)	文字列 s を数値に変換して返す
document.write(s)	文字列 s を出力する
document.writeln(s)	文字列 s を改行つきで出力する

では次に、これも前回やった例の書き直してすが、「2 つの数のうち大きい方を表示する」例をやりましょう。前回は「条件によって枝分かれ」するには条件分岐命令を使っていましたが、高水準言語では「枝分かれ全体を表す構文」(if 文) というものを使います。具体的に見てみましょう。

```
var x = parseFloat(prompt('x?'));
var y = parseFloat(prompt('y?'));
var max;
if(x > y) {
  max = x;
} else {
  max = y;
}
document.write('largest = ');
document.write(max);
```

if 文ではかっこ内の条件 (ここでは x > y) の成否 (YES/NO) によって分岐が起こり、YES だったら 1 番目の枝、NO だったら 2 番目の枝の中を実行します。それぞれの枝の中は字下げして書いてありますが、これは「プログラムを見やすくするための習慣」です (しかし見にくいと絶対に間違えますから、守った方がよい習慣です)。

ところで、if 文で「NO の時にやること」が無い場合は、else 以降は書かなくて構いません。そちらの形を使った、「大きい方」のプログラムの別バージョンを示しておきます。このように、枝も短ければ 1 行に書いてしまって構いません。

```
var x = parseFloat(prompt('x?'));
var max = x;
var y = parseFloat(prompt('y?'));
if(y > max) { max = y; }
document.write('largest = ');
document.write(max);
```

ところで、上の 2 つのプログラムはやることは同じで、どちらも「正解」ですが、処理の進み方や見た目は見るからに違います。このように、ソフトウェアは動作や効果が同じであっても、さまざまな「正解」があり、さまざまに書くことができます。では、どの書き方を選ぶか? それは、プログラムを書く人のセンスによって、「この方が分かりやすく、間違いにくい」「将来直すときに直しやすい」などの見通しを持って選ぶわけです。皆様だったら、上の 2 つの例のどちらを選びますか?

このほか、JavaScript には繰り返しを記述する構文などもありますが、この科目はプログラミング入門ではないので今回は枝分かれまでの説明としています。なお、「式」についても説明していませんが、「＋」「- 」以外に掛け算「＊」、割り算「／」、剰余「%」などの演算子を書くこと、1 行に書く必要があるので適宜「(…) で囲むことなどが違っています。

演習 3-3 JavaScript の例題を実際に動かしてみなさい。動いたら、次の小問から 1 つ以上 (できれば全部) やってみなさい。

表 3: JavaScript の主要な構文

書き方	説明
式;	単に式を計算する (関数呼び出しなど)
変数 = 式;	式の値を計算して変数に入れる
var 変数 = 式;	変数を定義し、初期値を入れる

if(条件) {
  文…
}
	条件が成り立った時だけ「文…」を実行する

if(条件) {
  文 1…
} else {
  文 2…
}
	条件が成り立った時「文 1…」、そうでない時「文 2…」を実行する

if(条件 1) {
  文 1…
} else if(条件 2) {
  文 2…
} else {
  文 N…
}
	条件 1 が成り立った時「文 1…」、そうでなく条件 2 が成り立った時「文 2…」、どれも成り立たなかった時「文 N…」を実行する (条件と文はいくつでも追加できる)

a. 数値を 1 つ読み込み、それが正/負/零のときそれぞれ「plus」「minus」「zero」と表示するプログラムを作る。できれば、2 つの違う書き方のバージョンを作成してみられるとなおよい。

b. 3 つの値を読み込み、その最大を表示するプログラムを作る。できれば、2 つの違う書き方のバージョンを作成してみられるとなおよい。

c. 3 つの値を読み込み、その最大を表示するプログラムを作る。ただし、どれか 2 つ以上の値が等しい場合は代わりに「error」と表示する。できれば、2 つの違う書き方のバージョンを作成してみられるとなおよい。

「◎」の条件: (1) 小問を 2 問以上解答しており、(2) いずれも「できれば」の部分までやってあって、(3) まともな (と担当が考える) 考察が書いてあること。

2.3 アルゴリズムとその記述方法

ここまでに見て来たように、プログラミング言語で書かれたコードはコンピュータ上で動かすために極めて詳細な部分にまで渡って記述がなされています。しかし、私たちがコンピュータにどのような処理をさせるか検討している時は、詳細な部分は必要ではないので、「処理手順の本質部分」だけに注目して検討する方が好ましいのです。この、処理手順の本質部分のことを**アルゴリズム** (algorithm) と言います。

アルゴリズムの書き表し方には**フローチャート** (flowchart) を始めとして色々なものがありますが、ここでは日本語を使ってプログラムふうに記述することにします。これを**擬似コード** (pseudocode) と呼びます。たとえば、先の `larger` を擬似コードで記述してみます。

```
larger(x, y): x と y のうち大きい方を返す関数
      もし x > y なら、x を返す。
      y を返す。
```

枝分かれはこれまで散々やったので、今度はくり返しのある例を見てみましょう。**素数** (prime number) とは、「1 とその数自身でしか割り切ることのできない正の整数」ですが、パラメタ n が素数かどうかを判定する関数を書いてみましょう。

```
isprime(n): n が素数か否かを返す関数
    result ←「はい」。
    i を 2 から n-1 まで 1 ずつ増やしながら繰り返し。
        もし n が i で割り切れるなら、result ←「いいえ」。
    ここまで繰り返し。
    result を返す。
```

見て分かるように、ここでは代入を「←」で表現しています (やはり=では混乱しやすいので)。

このアルゴリズムの要点は、変数 result に最初「はい」を入れておきますが、2〜n-1 の数で割ってみて割り切れたらそれを「いいえ」に書き換えることです。このため繰り返しを終わった後では、result には 1 回でも割り切れた場合は「いいえ」、そうでなければ「はい」が入っていて、それが素数か否かの答えとなっているわけです。このような変数の使い方を**旗** (flag) と言います。つまり、最初に旗を立てておき、いろいろ処理をした後、最後に旗が降りていれば、誰が降ろしたかかは分からなくても、誰かが旗を降ろしたことは確実に分かるというわけです。[16]

2.4 アルゴリズムとデータ構造

先に例示したアルゴリズムは、少数の変数だけを扱うものでしたが、多くのアルゴリズムはもっとまとまったデータを扱います。そのときは、データをどのような形でプログラムの中に保持し扱うかを決める必要があります。プログラムが扱うデータの形のことを**データ構造** (data structure) と呼びます。

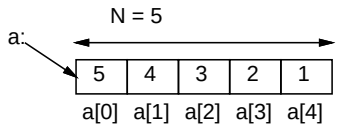

図 5: 配列

最も基本的でよく使われるデータ構造に、**配列** (array) があります。これは、N 個の「入れる場所」が並んでいて、その何番目かを番号で指定できるようなものです。たとえば、a が配列で、そこに N 個の値が入っていたとすると、その最初のものは a[0]、次は a[1]、…のようになっていて、最後は a[$N-1$] まであるわけです (図 5)。

いくつか、配列の上で動作するアルゴリズムを見てみましょう。まず、配列に入っている n 個の値のうち最大を求めるアルゴリズムです。[17]

```
arraymax(a, n): 配列 a に入っている n 個の数値の最大値を返す
    max ← 0。
    i を 0 から n-1 まで 1 ずつ増やしながら繰り返し。
        もし max < a[i] なら、max ← a[i]。
    ここまで繰り返し。
    sum を返す。
```

[16] ところで、このアルゴリズムは少し工夫すると、処理時間をずっと短くできます。どうすればよいか、分かりますか?

[17] ただし、このアルゴリズムは「全データが負だと、その最大ではなく 0 を返す」バグがあります。修正は演習で。

次は、配列の中に指定された値 x があれば、それが何番目にあるかを返し、無ければ -1 を返すアルゴリズムです。なお、このようにどこにあるかを順番に調べて行く方法を**線形探索** (linear search) と呼びます。

linearsearch(a, n, x): 配列 a の n 個の中に x があればその位置、無ければ-1 を返す
    pos ←-1。
    i を 0 から n-1 まで 1 ずつ増やしながら繰り返し。
        もし a[i] = x ならば、pos ← i。
    ここまで繰り返し。
    pos を返す。

ところで、配列の中の値が「小さい順」に並んでいれば、線形探索よりも効率のよい方法である **2 分探索** (binary search) が使えます。これは、最初に left を配列の先頭、right を配列の末尾に設定しておき、その中間の位置 $\lfloor \frac{left+right}{2} \rfloor$(平均を整数に切捨てたもの) を c と起き、a[c] と x の大小に応じて枝分かれします。

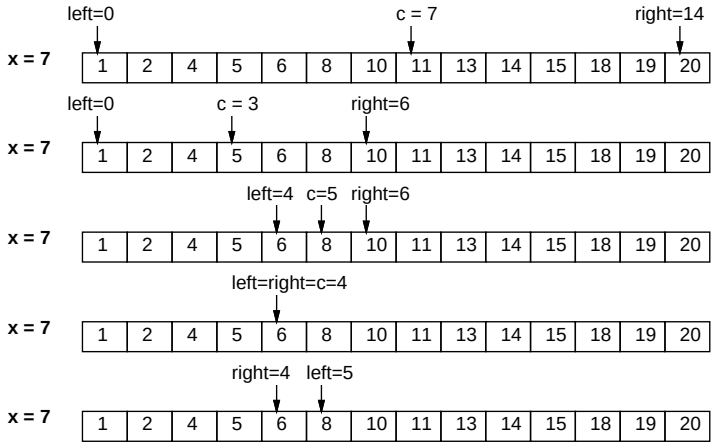

図 6: 2 分探索

具体的には、a[c] が x より大きければ、求める値は「左半分」にありますから、right を c の 1 つ手前にして、続けます。a[c] が x より小さければ、求める値は「右半分」にありますから、left を c の 1 つ先にして、続けます。等しければ単に c を返せばよいですね。さて、これをやっていくと、left と right の間 (区間) が毎回半分ずつになっていき、求める値が見つかるか、または区間長が 0 になります。0 になったら繰り返しは抜けだし、見つからなかったということで-1 を返します (図 6)。

binsearch(a, n, x): 昇順に並んだ配列 a 中の x の位置 (無ければ-1) を返す
    left ← 0。
    right ← n-1。
    left ≦ right である間繰り返し。
        c ← floor((left + right) / 2)。
        もし a[c] > x なら、right ← c-1。
        そうでなくてもし a[c] < x なら、left ← c+1。
        そうでないなら、c を返す。
    ここまで繰り返し。
    -1 を返す。

演習 3-4 上で示した配列のアルゴリズム (擬似コード) について、その「紙の上での実行」をしていただきます。以下の小問から 1 つ以上 (できれば全部) やってみなさい。

a. `arraymax` を次のデータで実行するものとする。

```
n = 10, a = [3, 7, 6, 4, 1, 8, 9, 2, 3, 5]
n = 5, a = [-3, -1, -2, -5, -4]
```

それぞれの場合について、「ここまで繰り返し」の直前に到達した各時点での i と max の値を記入しなさい。できれば、すべてのデータが負でも正しい答えが求まるようにアルゴリズムを手直しする方法を考案し解説するとなおよい。

i	0	1	2	…
max				

b. `lenaersearch` を次のデータで実行するものとする。

```
x = 2, n = 10, a = [3, 7, 6, 4, 1, 8, 9, 2, 3, 5]
x = 4, n = 5, a = [1, 4, 2, 4, 7]
```

それぞれの場合について、「ここまで繰り返し」の直前に到達した各時点での i と pos の値を記入しなさい。できれば、「複数同じ値があった場合」どうなるかについて示し、「複数あった場合は最初の位置を答える」ようにアルゴリズムを手直しする方法を考案し解説するとなおよい。

i	0	1	2	…
pos				

c. `binsearch` を次のデータで実行するものとする。

```
x = 2, n = 15, a = [1, 2, 4, 5, 6, 8, 10, 11, 13, 14, 15, 18, 19, 20]
x = 21, n = 15, a = [1, 2, 4, 5, 6, 8, 10, 11, 13, 14, 15, 18, 19, 20]
```

それぞれの場合について、「ここまで繰り返し」の直前に到達した各時点での left、right、c の値を記入しなさい。できれば、「複数同じ値があった場合」どうなるかについて示し、「複数あった場合は最初の位置を答える」ようにアルゴリズムを手直しする方法を考案し解説するとなおよい。

left	
right	
c	

「◎」の条件: (1) 小問を 2 問以上解答しており、(2) いずれも「できれば」の部分までやってあって、(3) まともな (と担当が考える) 考察が書いてあること。

2.5 アルゴリズムと計算量

先の問題で、2 分探索の方が「効率が良い」と書いてありましたが、これはどういう意味なのでしょう? もちろん、実際にプログラムを書いて特定のコンピュータで動かして見れば、時間を測ることができます。しかし、それをしなくても優劣を判断することが可能なのです。実際のところ、プログラムを書いて動かして計測することは、プログラムの上手下手とかコンピュータの性能などに左右されますから、あまりよい判断方法ではないのです。

以下で述べる方法まず、次の観察に基づきます。「コンピュータで命令を実行する時間は、命令ごとに違うとしても、それぞれの命令についていえば、おおよそ一定である。」もちろん、命令についても (データがキャッシュにあるかどうかなどで) 変動しますが、キャッシュに当たる率が一定なら、平均するとこれくらい、という値は出るはずです。

そうすると、`linearsearch` を実行するときには、繰り返しの範囲内を N 回、それ以外の部分を 1 回実行することになります。1 回しか実行しない部分はあっという間に終わるでしょうから、それは

省略するとすると、このアルゴリズムの実行時間はおおよそ N に比例すると言えるでしょう。これを $O(N)$(オーダーが N である)、というふうに言います。または、**計算量** (computational complexity) が $O(N)$ である、とも言います。

一方、`binsearch` を実行するときには、繰り返しの回数は何回でしょうか。毎回範囲が半分ずつになって、(途中で運よく終わるのでなければ) 範囲長が 0 になったところで終わるので、回数はおおよそ $\log_2 N$ 回といえます (なお、$\log_2 N$ とは、$2^x \sim N$ であるような x、もっと平たく言えば「2 を何回掛け算したら N になるかの回数) です。つまり、こちらのアルゴリズムは $O(\log N)$ です。

ここで、N と $\log_2 N$ の対応表を示しておきましょう (N の方は概数)。

N	1	2	4	8	1000	1000000	1000000000	10000000000000
$\log_2 N$	0	1	2	3	10	20	30	40

つまり、N が 1000 倍になるごとに、$O(N)$ のアルゴリズムの実行時間は 1000 倍ずつになりますが、$O(\log N)$ のアルゴリズムの実行時間は「一定ずつしか増えない」わけです。ということは、多少ループ 1 回あたりの処理の手間が多くても、プログラムの書き方が下手でも、コンピュータが遅くても、そんなことは問題になりません。このように、計算量というのはアルゴリズムの「効率」を判断するよいやり方なのです。

さて、ここで出て来たアルゴリズムは $O(N)$ とか $O(\log N)$ でしたが、一番速いのは何でしょう。それは、データ量に係わらず一定時間で終わる処理であり、$O(1)$ と書きます。たとえば「並びの最初のものを打ち出す」処理は読んで打ってれで終わりなので $O(1)$ です。

逆に、世の中の複雑な問題を解くアルゴリズムではもっと大きな計算量が必要です。たとえば、$O(N \log N)$ とか、$O(N^2)$ とかもよくあります。一般に $O(N^i)$ の形で書ける計算量のアルゴリズムのことを**多項式時間** (polynominal time) と言います。さらに困難な問題になると、$O(2^N)$ のようなものもあります。これを**指数時間** (exponential time) と呼びます。指数時間のアルゴリズムしかない問題だと、データの数を 1 個増やすだけで、処理に要する時間が倍、10 個増やすと 1000 倍になってしまい、大きな N について実際に実行するのは困難ということになります。

一般にはこれは困ったことですが、役に立つこともあります。たとえば、今日の暗号アルゴリズムでは「鍵を知らない人がしらみ潰しに鍵を探す計算をしたときの」計算量が指数時間であるようになっています。これにより、悪い人が無理矢理解読しようとしても実用的な時間では解読できないようになっているわけです (ただし、時々、もともと指数時間掛かると思っていた問題を、もっと短い時間で — たとえば多項式時間で— 解く方法が発見されたりします。そうなると、その問題を利用していた暗号アルゴリズムは危険だということになってしまいます)。

3 ソフトウェア開発

3.1 ソフトウェア開発とは

ここまでは高水準言語によって「プログラム」を書いてきましたが、我々が普段耳にする用語はどちらかというと「ソフトウェア」ですよね。**ソフトウェア** (software) とは、コンピュータ上で何らかのタスク (仕事) をこなすために必要とされるプログラム、データ、手引き書など、ハードウェア (装置) 以外のすべての部分を指して使われます。そしてハードまで含めた動くもの全体が**システム** (system) なわけです。

たとえば Word などを考えてみても、動作するプログラムコードに加えて、そのプログラムの画面に表示されるさまざまなアイコン (絵) とか、さまざまな文書を作るためのテンプレートなどのデータも、製品としては必要なわけです。また、ソフトウェアの種類によっては、マニュアルなどがきちんと揃っていないと役に立たないかも知れません。

ソフトウェアを作る作業は、単にやみくもにプログラムを書きまくっていくという方法ではまず成功しません。ここまでやって見ただけでも分かるように、プログラムは非常に緻密なものであり、ど

こかがちょっとでも間違っていたら、それだけで全体としての動作が台無しになることもよくあります。ほんの数行のプログラムでも正しく作るのに苦労することがあるのに、何万行ものプログラムをきちんと作って動かすというのは本当に大変なことなのです。

これには、さまざまな理由があります。まず、製品となるソフトウェアを作る時には、きちんと進め方を文書化し、設計も文書化し、コードも文書化し、テスト計画を立ててテストし、という付帯作業が多数必要です。そのため、ただ単にプログラムを書くだけなら 1 日に数十行とか数百行とか書ける人は沢山いますが、ソフトウェア開発プロジェクトとして実施するとなると、「開発者 1 日あたり数行」になってしまうのが普通なのです。

次に、そうなると数万行のプログラムを 1 年程度で完成させるには、開発者が 100 人とかもっと必要になります。人が多くなると、それらの人の間で意思疎通する手間、具体的にはミーティングとか会議とか連絡とかの手間がどんどん大きくなり、その分だけ実際にコードを書く時間が削られてしまいます。これもまた、「開発者 1 日あたり数行」になってしまう大きな要因の 1 つです。

こうして見ると、ソフトウェア開発で「人が多くなる」というのはさまざまな点でマイナスをもたらすことが分かります。しかし、ソフトウェア開発に詳しくない人にはそのことが分かりません。たとえば、100 人でソフトの残りを仕上げるのに 2 か月掛かるとします。しかしそれでは期限に遅れるとなると、ではもう 100 人増員して 200 人にすれば 1 か月で仕上がるとか思うわけです。

しかしそれはとんでもない勘違いで、新たに人を 100 人増員したら、その人たちがどんなにできる人であっても、(1) これまでやっていた人たちが新しく来た人たちに情報を伝えて仕事ができるようにする手間 (教える側・教わる側とも) と、(2) 人数が増えることによるオーバヘッドの増大によって、ソフトウェアの完成時期は当初予定よりかえって伸びるのが関の山です。この「遅れているプロジェクトに追加人員を投入すると遅れは一層ひどくなる」という知見は、フレデリック・P・ブルックスという人が最初に指摘したことから「ブルックスの法則」と呼ばれています。

3.2 要求仕様の問題

そもそも一番最初に、「どのようなプログラムを作る」ということを決めることがまず大変です。このことを決めたものを**要求仕様** (requirements specification) と呼びますが、世の中のソフトウェア開発プロジェクトの多くは要求仕様がきちんと決まっていなかったためにトラブルに陥っています。

たとえば、次の要求仕様を見てください。

> 2 つの数値を入力し、その大きい方を打ち出す。…(A)

これが先に動かしたプログラム (2 バージョンありましたね) の元となる要求仕様だったとします。プログラムの片方を再掲します (動作はどっちでも同じです)。

```
var x = parseFloat(prompt('x?'));
var y = parseFloat(prompt('y?'));
var max;
if(x > y) {
  max = x;
} else {
  max = y;
}
document.write('largest = ');
document.write(max);
```

このプログラムは完璧ですか? 要求仕様に合致していますか? 当然じゃないか、と思われるかも知れませんが、次のことを考えてみてください。

- 要求仕様 (A) は、2 つの数値が同じだったときの動作について規定していない。

ということは、どうすればいいのでしょうか。2 つの数値が同じだったらまずいので、同じ場合をチェックしてエラーメッセージを出しますか? 待ってください。勝手にそんなことを決めてはいけません。お客さんに確認したところ、要求仕様を次のように改訂したものとします。

> 2 つの数値を入力し、その大きい方を打ち出す。2 つの数値が同じである場合は、その数値を打ち出す。…(B)

これだったら、先のプログラムで完璧です。早まって直さなくてよかったですね。でも次のようになるかも知れません。

> 2 つの数値を入力し、その大きい方を打ち出す。2 つの数値が同じである場合は、何が出力されてもよいものとする。…(C)

このプログラムがもっと大きなシステムの一部であって、値が同じだったら別の場所で別の処理をするので、このプログラムの出力は気にしないという場合はこのようになります。これも先のプログラムで OK です。では次の場合はどうでしょうか。

> 2 つの数値を入力し、その大きい方を打ち出す。2 つの数値が同じであることは決してないはずである。…(D)

決してないのだから、考えないでよい? それは違います。決してないことが起きていると分かったら、それは「おかしいことが起きている」と教えてあげるのが筋です。もっとも、本当にそうした方がよいかどうかは再度お客さんの意向を確認した方が安全ですが。このように、ごく簡単なプログラムでも「要求仕様をきちんと決める」「決めた要求仕様に正しく合致するコードを作る」ことはとても大変なのです。

3.3 テスト

作成したコードには大抵の場合、間違いが含まれているので (人間は必ず間違いを犯します)、この間違いを防いだり発見して修正する作業が必要となります。その 1 つの方法は、コードをよく見直すことです。当り前みたいですが、ソフトウェア開発プロセスの中に複数人で集まってコードを見直す作業 (**レビュー** (review)、**インスペクション** (inspection)、**ウォークスルー** (walkthrough) などと呼ばれます) が組み込まれていることは普通です。

ですが、普通まず考え付くやり方は、実際にコードを走らせて正しく動作するか確認することですよね? これを**テスト** (test) と呼びます。ちなみに、テストのためにプログラムに与える入力と、その入力に対応して出力されるであろう「正しい」結果を組にしたものを**テストケース** (test case) と言います。テストケースでは、プログラムを動かしてみる前に正解を用意しておくことがポイントです (間違った結果を出すプログラムでも、あなたがプログラムを書いたのだとすると、出て来た結果を見たらそれで合っているような錯覚に陥るかも知れませんから)。

では、どうやってテストしますか? 思い付いた入力を与えて動かしてみて、結果が想定とあっているかをいく通りかやる? 最初のテストとしてはそれもいいですが、それだけでは全然不足です。実際に開発プロセスの中で行われるテストとしては、次のようなものがあります (これでも一部です)。

- **スモークテスト** (smoke test) — コードがとりあえず一通り実行されることを確かめる。上で「最初のテスト」と呼んだもの。[18]
- **機能テスト** (blackbox test) — コードの「仕様」に基づき、さまざまな可能性をひととおり網羅するようなテストケースを用意し、結果を確認するもの。
- **構造テスト** (whitebox test) — コードの「構造」に基づき、分岐などの境界の条件を調べるテストケースを作って確認するもの。この中に、「プログラム中のできるだけ多くの箇所を実行する**被覆テスト** (coverage test) がある。[19]

[18] 電気のスイッチを入れてみて、「煙」が出てこないかどうか確認するという意味でこう呼ばれています。

[19] 実際には大きなコードになるとその全ての箇所を実行するというのは不可能です。その場合はカバー率を設定してそれを目標に被覆テストを行います。

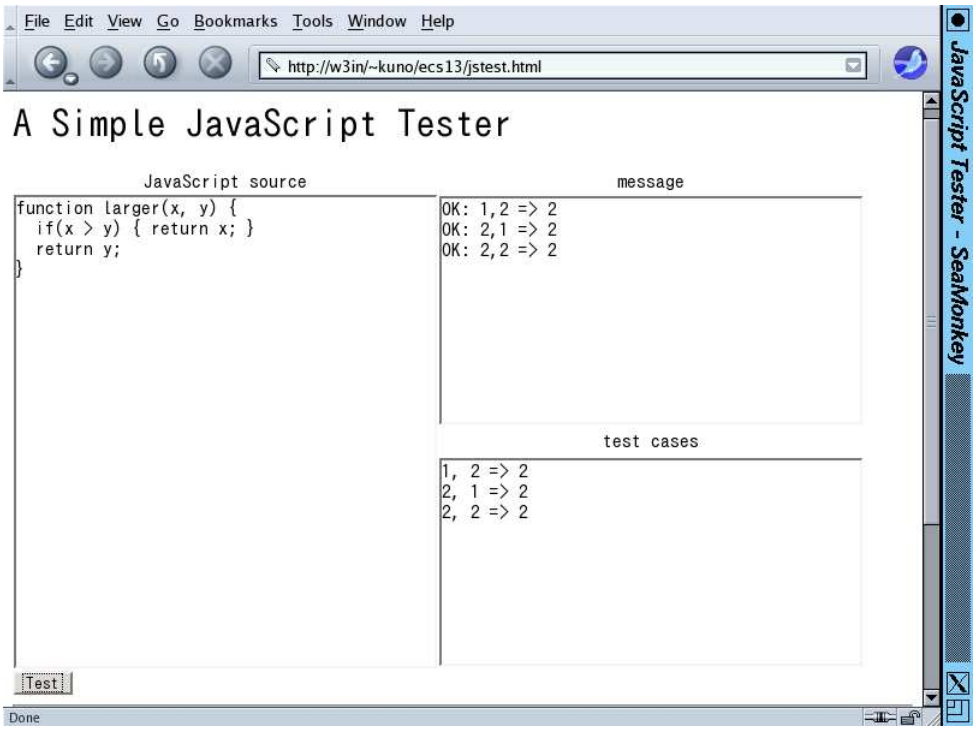

図 7: JavaScript のテスト機能つきページ

- **ランダムテスト** (random test) — 乱数などで生成した値を与え、不具合が起きないか調べる。
- **回帰テスト** (regression test) — これまでに確認したテストケースをすべて保管しておき、プログラムを修正したあと再度すべて実行しなおして修正によって壊れた箇所がないか確認する。

近年では開発プロセスの一種として「テストファースト」つまり、コードを 1 行でも書くよりもまず先にテストを記述するという流儀も提案されており、それなりに使われています。

また、テストケースを予め格納しておき、自動的にテストを実行して結果を報告してくれるツール (自動テストツール) の使用も一般的です。ここでは簡単なテスト実行機能を提供してくれる Web ページを用意したので (図 7)、これを用いてテストを行ってみましょう。

このページでは、テスト対象のコードを**関数** (function) という単位でパッケージして用意することにしたので、その説明をします。ここでいう関数とは数学の関数とは違って、単に「まとまったコードに名前をつけて呼び出せる」ようにしたものであり、次の形をしています。

```
function 関数名 (変数名, ...) {
  文…
}
```

「関数名」はコードに対してつける任意の名前であり、その機能をよく表す分かりやすい名前をつけます。変数名の並びは、この関数に対する「パラメタ」であり、ここに処理してもらうデータを渡します。そして、「文…」の部分はこれま同様に処理を書きますが、最後に結果を「`return 式;`」という文により返します。具体例ですが、先の「より大きい値」を関数にした「渡されたパラメタのうち大きい値を返す」関数 `larger()` は次のようになります (書き方は色々有り得ますので、あくまで一例です)。これをテスト対象とします。

```
function larger(x, y) {
  if(x > y) { return x; }
  return y;
}
```

次にテストケースですが、パラメタとして渡す 2 つの数の前者が大きい場合、後者が大きい場合、両方とも同じ場合の 3 つの場合をテストすれば十分でしょう (理由を考えてみてください)。そこで、テストケースとして次のように 3 つを記述します (各ケースでは「=>」の左にパラメタ、右に想定出力を指定します— この書き方はこのツール用に決めたものです)。

```
1, 2 => 2
2, 1 => 2
2, 2 => 2
```

これらを入力した状態で Test ボタンを押すと、テストが実行され、3 つとものケースをパスする (実行結果とテストケースの想定結果が一致する) ことが確認されます。

演習 3-5 `larger()` のテストを自分でも実行してみなさい。「正しくない」テストケースを設定したら NG が出ることも確認しなさい。終わったら、次の 3 小問から 1 つ以上 (できれば全部) を選んでやってください。いずれも、「なぜこのテストケースで十分と考えるか」を記述すること。

a. 「3 つの数をパラメタとして受け取り、最も大きいものを返すが、ただし 3 つの数値がすべて同じなら-3、2 つだけが互いに同じなら-2 を返す」という関数を書くものとします。テストケースを書きなさい。その後で、関数の実装を書き、テストしなさい。できれば、2 通りの書き方でやってみるとなおよいです。自力で書くのがつらい人向けにバグあり版をつけておきます。

```
function largest(a, b, c) {
  if(a == b && b == c) { return -3; }
  var max = a;
  if(max < b) { max = b; }
  if(max < c) { max = c; }
  return max;
}
```

b. 「3 つの数をパラメタとして受け取り、小さい順 (等しいものがある場合も含めるので正確には大きくない順) に並べて返す」という関数を書くものとします (並びは「`[1, 2, 3]`」のようにかぎかっこで囲んだ数値のリストとして表現します)。テストケースを書きなさい。その後で、関数の実装を書き、テストしなさい。できれば、2 通りの書き方でやってみるとなおよいです。自力で書くのがつらい人向けにバグあり版をつけておきます。

```
function order3(a, b, c) {
  if(a > b) { var x = a; a = b; b = x; }
  if(b > c) { var x = b; b = c; c = x; }
  return [a, b, c];
}
```

c. 「h 時 m 分から、h_1 時間 m_1 分たったら、何時何分かを計算し、答えを (時と分の並びとして) 返す関数を書くものとします (24 時制とし、23:59 を過ぎたら翌日の 0 時になります)。」ただし、$0 \leq h, h_1 < 24$, $0 \leq m, m_1 < 60$ とします。テストケースを書きなさい。その後で、関数の実装を書き、テストしなさい。できれば、2 通りの書き方でやってみるとなおよいです。自力で書くのがつらい人向けにバグあり版をつけておきます。

```
function etime(h, m, h1, m1) {
  h = h + h1;
  if(h > 23) { h = h - 24; }
  m = m + m1;
```

```
    if(m > 59) { m = m - 60; }
    return [h, m];
}
```

「◎」の条件: (1) 小問を 2 問以上解答しており、(2) いずれも「できれば」の部分までやってあって、(3) まともな (と担当が考える) 考察が書いてあること。

3.4 ソフトウェア開発プロセス

要求仕様の獲得から始めて、最後にソフトウェアがテストを通過し納品されるまでの過程のことを**ソフトウェア開発プロセス** (software development process) と呼び、さまざまな流儀のやり方があります。一番古典的でしかし現在でも多く使われているのが、**ウォーターフォール** (waterfall) 型のプロセスです。これは、分析→設計→製造→テスト、というふうにステップが決まっており、各ステップとも一旦終わったら後戻りしない、ということが原則です (図 8)。それぞれ前の段階が終わってなかったら次の段階の作業はできない、というのは言われてみれば当然なので、このモデルは自然のように思えます。

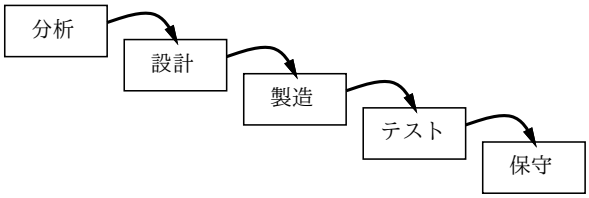

図 8: ウォーターフォール型プロセス

でも実際には、要求を確定してこれ以上変更しない、ということになっていたのに、後からお客さんの無理な要求が追加されたとか、途中までやってみたらできないことが明らかになったとかで、手戻りが発生しがちです。手戻りが発生すると、その分だけスケジュールが遅れてあとの方で大変になります。では一切変更を認めないのがいいのでしょうか? そうすると、最後に製品を納品した後で「肝心なところの機能が要求から落ちていて役に立たないと分かった」というふうな失敗に至ります。現実にはこの両方の失敗が組合わさって起きるというのが、ソフトウェア開発の現実です。

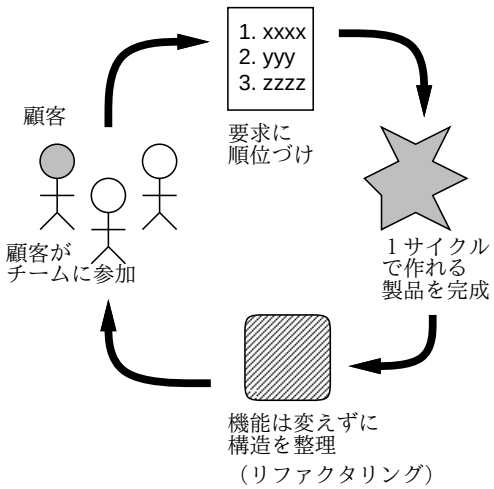

図 9: アジャイル型プロセス

そういうわけで、現在ではもう 1 つ別の**アジャイル型** (agile) 型のプロセスが人気を博しています。アジャイル型では、要求仕様とか設計とかに延々と時間を掛けて決定する代わりに、顧客にまず要求を聞いて、一定期間 (数週間程度) でできそうなところを選んでそこをまず開発して製品として動かし、コードをきれいに整理した後、また次にやることを決めて開発し、というふうに多数回のサイクルを繰り返して開発を進めて行きます (図 9)。こうすれば、後から要求の追加を受け入れることも容易ですし、動いているリリースを見れば顧客も何が足りて何が足りないか確認できます。ただしその一方で、じっくり設計して作るわけではないので、大きなものを作るときに設計が練れていなくて途中で行き詰まる危険があります。

ソフトウェアプロセスやその改良も現在ホットな話題であり、さまざまな研究がなされています。そのあたりは「ソフトウェア工学」という分野なので、興味があればそちらの授業を取りましょう。

3.5 ソフトウェアと知的財産

知的財産 (intellectual property) とは、人間の知的活動の成果として生み出され、価値を持つようなもの全般を指す言葉です。たとえば絵や音楽作品や TV 番組のような**著作物** (copyrighted materials)、**特許** (patent) の対象となるようなアイデアなどはその代表です。

そして、ソフトウェアもまた、知的財産の一つです。コンピュータに関して言えば、ハードウェアが高価でありその製造が大きな利益を産んでいた時代はとっくに終わり、現在ではソフトウェアの方がはるかに大きな価値をもたらすようになっています。

これは、ハードウェアが VLSI 技術の進歩により急激に大容量化・低価格化したのに対し、その上で動作するソフトウェアは依然として人間が知的労働で作るしかなく、しかもハードウェアが大容量になり、大規模なソフトウェアを動かすようになった分だけ、作るのも大変になっていることによります。その大変さについては先に述べた通りです。

では、ソフトウェアという知的財産はどのようにして守られているのでしょう。多くの国では、ソフトウェアもまた著作物の一種として、**著作権** (copyright) によって守られています。著作権は表現を保護するものですが、ソフトウェアは大量のコードを組み立ててできあがっており、そのコードに価値があると考えれば、これはうなずける考えです。

ただし、著作権は「アイデア」は保護しないので、ある問題を解決するすばらしい方法を思い付いた人がいたとしても、それが保護されないという問題がありました。そのソフトウェアの機能を観察して、それと同じに機能するソフトウェアを「まったく別に」書けば、それは著作権法的には OK なのです (画面などの見た目はもちろん真似てはいけません)。

それでは困るので、今日ではソフトウェア上のアイデアも特許によって、つまり**ソフトウェア特許** software patents により保護するということが一般化してきています。ただし、ソフトウェア特許はまだ歴史が浅いため、何が特許として認められ、何は認められないかについて、多くの論争や紛争があります (一見簡単なアイデアでも特許として認めさせてしまえば、多くの企業を特許違反として訴えてお金を出させることが可能になりますから)。

3.6 所有ソフトウェアとオープンソース

ソフトウェアはディジタル情報であるため、それを物理的にコピーするのはごく簡単です。このため、ソフトウェア開発者がその利用者から収益を得るためには、通常の商品のように「お金を渡したら、品物をあげて、もらった方はそれを自由にしてよい (売り切り主義)」は通用しません。買った誰かがそれをネットで公開したら他の人は無料でそれを取り寄せて使えてしまいますから。

そこで、ソフトウェア開発者は対価を払った利用者に対し「ソフトウェアを使用する権利」を許諾します。当然この権利は「使用すること」についてだけで、他人にコピーさせたりはできませんし、自分のマシンで使うにしてもインストールできる台数などは制限されます。今日ではこのような「利用許諾」による方式が一般的なモデルとなっています。

このようなモデルを**占有ソフトウェア** (proprietary software) といいます。しかしこのモデルは、権利者にとってはいとしても、利用者にとっては不便です。皆様もこのモデルに対してはいろいろ嫌な思い出があるかと思います。Richard Stallman という人は優秀なプログラマでしたが、占有ソフトウェアが自分の思うように修正/改良できないことを嫌悪し、**フリーソフトウェア** (free software) という運動を提唱しています。

彼の考え方では、ソフトウェアは皆の共有財産であり、作成されたすべてのソフトウェアは無償で公開され誰もが自由に使えるべきだ、というものです。ではソフトウェア開発者はどこから収入を得るかというと、「このようなソフトウェアが欲しい」という顧客が彼を雇って開発させ、その労働対価が収入となるわけです。しかし完成したソフトウェアは上述のように無償で公開されます。

彼はこの考えに基づいて **GNU**(Gnu's Not Unix) というプロジェクトを開始し、当時占有だった Unix OS の互換ソフトウェアやテキストエディタなど多数のフリーソフトウェアを公開しています。GNU プロジェクトには対しては賛同者から多くの寄付がなされてきました。

1 つ問題なのは、そうやって無償で公開されているソフトを入手し、「ちょっとだけ改良した」ソフトを旧来の許諾モデルで「販売」して儲ける人がでると困る、ということでした。そこで Stallman は **GNU GPL**(GNU General Public Licence) というライセンス条項を制定し、フリーソフトウェアはこの条件下で公開されることを推奨しています。GPL では「このソフトはソースコードを無償で公開し、誰がどのように使ってもよいが、改良などを加えた場合はそのソフトもまた、同じ GPL の条件下で公開されなければならない」というものです。こうしておけば、せっかくの無償ソフトウェアが途中から有償になってしまうことは起きません。

GPL はこのようにかなり厳しい哲学を持っているので、開発者によってはこれよりは緩い条件を用いているところもありますが、今日ではこのようにソースコードを公開しているソフトウェアが多数あります。これらを総称して**オープンソース** (open source) と呼びます。そして、Linux OS、Apache Web Server など、オープンソースだけれども対応する有償ソフトウェアに負けないくらい広く使われているソフトウェアも多数あります。オープンソースの利点としては、無償であることに加え、ソースが公開されているので誰でも改良に参加でき、また不具合があった時に自分でその原因を究明したり対処することも可能なことが挙げられています。

演習 3-6 ネット上で「ソフトウェア開発プロジェクトが失敗した」事例を探し、2 つのまったく異なる事例で「失敗した原因が同様である」ようなものを探し、それらについて解説しなさい。それぞれの事例について記載されている URL を明記すること。

「◎」の要件: それぞれのプロジェクトやその共通点・相違点について適切な (と担当が考える) 検討がなされていること。

4 まとめ

この回では高水準言語の話題から始めて、ソフトウェアを書いたりテストしたりすることはどのようなことか、何が難しいのか、どのような問題があるのか、ということがらを説明しました。さらに、プログラムを抽象化した「本質」であるアルゴリズムと、アルゴリズムの性能を測る指標である計算量についても説明しました。プログラムを速くしようとして細かいことで頑張るよりも、よいアルゴリズム (と適切なデータ構造) を見つけて採用することの方がずっと効果的だということは、知っておくとよいでしょう。