コンピュータリテラシ

久野　靖
電気通信大学

2019.12.24 改訂

目 次

 # 1 コンピュータの利用と認証 1

1.0 はじめに 1
1.1 コンピュータとは何か 2
1.2 認証とパスワードの安全性 2
1.2.1 認証とその必要性 ex 2
1.2.2 パスワードとそれにまつわるトラブル ex 3
1.2.3 安全なパスワードのために ex 4
1.3 キーボード入力の重要性 5
1.4 付録: タッチタイピング入門 7
1.4.1 はじめに 7
1.4.2 キーボードとソフト 7
1.4.3 ホームポジション 7
1.4.4 打ってみよう! 8
1.4.5 ホームポジションの指づかい 8
1.4.6 g と h の追加 8
1.4.7 左手の上下段 9
1.4.8 右手の上下段 9
1.4.9 総合練習 10

 # 2 インターネットの原理 11

2.1 Unix システムの利用 11
2.1.1 Unix システムへの接続 11
2.1.2 CUI と GUI ex 11
2.2 ネットワークの基礎概念 12
2.2.1 イントロ: 地球の裏側までどのくらい? ex 12
2.2.2 ネットワークとその目的 ex 14
2.2.3 回線交換とパケット交換 ex 14
2.2.4 システムと階層構造 ex 15
2.2.5 プロトコルとその考え方 ex 15
2.3 インターネットのプロトコル 16
2.3.1 TCP/IP のプロトコル群 ex 16
2.3.2 IP アドレス ex 17
2.3.3 インターネットの構造 ex 18
2.3.4 IP と経路制御 19
2.3.5 ドメイン名と DNS 21

 # 3 ネットワークと安全性 23

3.1 情報セキュリティ 23
3.1.1 セキュリティと情報セキュリティ ex 23
3.1.2 暗号技術 ex 24

3.1.3 PKI と証明書の連鎖 ex . . . 25
3.2 ネットワークサービスとその構成 ex . . . 26
3.3 遠隔ログインと SSH . . . 27
3.4 World Wide Web . . . 28
3.4.1 WWW とリンク ex . . . 28
3.4.2 Web アプリケーション ex . . . 29
3.5 電子メール . . . 30
3.5.1 電子メールサービスの構成 ex . . . 30
3.5.2 IMAP クライアントの設定内容 . . . 31
3.5.3 メールメッセージの形式 ex . . . 32
3.6 ネット上のコミュニケーション . . . 34
3.6.1 ネット上のコミュニケーションが持つ性質 . . . 34
3.6.2 電子メールに関する留意事項 . . . 35
3.6.3 SNS などのコミュニケーションサービスに関する留意事項 . . . 35

 # 4 コンピュータの動作原理 39
4.1 コンピュータとデジタル情報 . . . 39
4.1.1 デジタル情報とビット ex . . . 39
4.1.2 2 進法 — 0 と 1 による数値の表現 ex . . . 40
4.1.3 2 の補数による負数の表現 ex . . . 42
4.2 コンピュータと情報処理 . . . 43
4.2.1 コンピュータとプログラム ex . . . 43
4.2.2 小さなコンピュータのシミュレータ ex . . . 44
4.2.3 ループのあるプログラム . . . 46
4.3 コンピュータの性能向上とその意義 . . . 48
4.3.1 CPU の製造技術 . . . 48
4.3.2 デジタル革命の本質 . . . 49

 # 5 ファイルシステムとファイル操作 51
5.1 ファイルシステム . . . 51
5.1.1 2 次記憶装置とファイルシステム ex . . . 51
5.1.2 情報の整理とファイルシステムの階層構造 ex . . . 51
5.1.3 パス名と現在位置 ex . . . 52
5.1.4 ls — ファイル名一覧の表示 ex . . . 54
5.1.5 ファイルの中身を調べる ex . . . 55
5.1.6 コンプリーション . . . 55
5.2 ファイルとディレクトリの操作 . . . 56
5.2.1 echo と cat ex . . . 56
5.2.2 ファイルの操作 ex . . . 56
5.2.3 ディレクトリの操作 ex . . . 57
5.3 ファイルの属性と保護設定 . . . 58
5.3.1 ファイルの各種属性 ex . . . 58
5.3.2 ファイルの保護設定 ex . . . 59
5.3.3 ディレクトリに対する保護設定 ex . . . 60

＃6 テキストファイルとエディタ 63

6.1 テキストエディタ ex . . . 63
6.2 テキストファイル . . . 64
6.2.1 テキストファイルと文字コード ex . . . 64
6.2.2 日本語の文字コード ex . . . 65
6.2.3 日本語のエンコーディング ex . . . 66
6.2.4 UNICODE と UTF8 ex . . . 66
6.2.5 文字コードの変換 . . . 67
6.3 Emacs の詳細 . . . 68
6.3.1 ファイル読み書き等 . . . 68
6.3.2 カーソル移動 . . . 68
6.3.3 日本語入力 . . . 68
6.3.4 削除とコピーペースト . . . 69
6.3.5 キーボードマクロ . . . 70
6.3.6 探索と置換 . . . 70
6.3.7 窓・フレーム・バッファの操作 . . . 71

＃7 コンピュータシステムと OS 73

7.1 コンピュータシステムの構造 . . . 73
7.1.1 ハードウェアの一般的な構成 . . . 73
7.1.2 ソフトウェアの一般的な構成 . . . 74
7.1.3 OS とその働き ex . . . 74
7.2 プロセスとその観察・操作 . . . 76
7.2.1 マルチタスクとプロセス ex . . . 76
7.2.2 ps — Unix でのプロセス観察 ex . . . 77
7.2.3 シェルによるプロセスの生成 ex . . . 79
7.2.4 kill によるプロセスの操作 ex . . . 80
7.3 シェルの制御機能 . . . 80
7.3.1 リダイレクションとパイプ ex . . . 80
7.3.2 ジョブコントロール . . . 82

＃8 フィルタとシェルスクリプト 85

8.1 ユティリティとフィルタ . . . 85
8.1.1 「大きなユティリティ」と「小さなユティリティ」 ex . . . 85
8.1.2 フィルタ ex . . . 86
8.1.3 tr — 文字の置換 ex . . . 86
8.1.4 sort と uniq — 整列と重複除去 ex . . . 87
8.1.5 head と tail と wc — 先頭/末尾/数える ex . . . 89
8.2 正規表現 . . . 90
8.2.1 grep 族 — パターンにあてはまる行を探す ex . . . 90
8.2.2 sed — 文字列を置き換える ex . . . 91
8.3 シェルの進んだ機能とシェルスクリプト . . . 93
8.3.1 シェル変数と変数展開 . . . 93
8.3.2 既定義なシェル変数と実行パス . . . 94
8.3.3 ファイル名展開 . . . 95
8.3.4 シェルスクリプト . . . 95
8.3.5 スクリプトの引数と変数 . . . 96
8.3.6 スクリプトの for ループ . . . 97

 # 9　マークアップによるテキスト整形 99

9.1 文書作成 99

9.1.1 テキストと文書の違いと位置付け 99

9.1.2 文書整形のアルゴリズム 99

9.1.3 見たまま方式とマークアップ方式 ex 101

9.1.4 LaTeX を動かしてみる 102

9.2 文書整形系 LaTeX 104

9.2.1 文書の基本構造 ex 104

9.2.2 表題、章、節 ex 105

9.2.3 いくつかの便利な環境 ex 105

9.2.4 脚注 ex 107

9.2.5 文字サイズ ex 107

9.2.6 表 ex 107

9.2.7 数式 ex 108

 # 10　グラフィクス/図と表 111

10.1 ピクセルグラフィクス 111

10.1.1 画像の表現 ex 111

10.1.2 ピクセルグラフィクスの得失 ex 112

10.1.3 PBM 画像を作成する ex 113

10.2 ベクターグラフィクス 114

10.2.1 ベクターグラフィクスとその特徴 ex 114

10.2.2 PostScript — ベクターグラフィクス記述言語 115

10.3 LaTeX 文書における図や表の扱い 118

10.3.1 LaTeX 文書に画像を入れる ex 118

10.3.2 図や表の扱い ex 119

10.3.3 ラベルと参照 ex 120

10.3.4 文献の参照 120

 # 11　アカデミックリテラシ (総合実習) 123

11.1 情報の検索 123

11.1.1 ネット検索と検索エンジン 123

11.1.2 文献や書籍の検索 124

11.2 著作権と引用 124

11.2.1 著作権について 124

11.2.2 著作権の制限と引用 125

11.3 アカデミックな文書の作成 126

11.3.1 アカデミックな文書とその要件 126

11.3.2 先行研究や文献の重要性 127

 # 12　HTML/CSS による Web ページ記述 129

12.1 HTML と CSS 129

12.1.1 Web とマークアップ ex 129

12.1.2 HTML の基本部分 ex 129

12.1.3 構造と表現の分離、スタイルシート ex 132

12.1.4 CSS の指定方法 ex 133

12.2 より進んだ HTML と CSS の指定 135

12.2.1 HTML をファイルとして保存する 135

12.2.2 ブロック要素とインライン要素 ex . . . 136
12.2.3 id と class ex . . . 136
12.2.4 箇条書と表 ex . . . 137
12.2.5 CSS 指定の追加 . . . 138

 # 13 Web と情報アーキテクチャ 141
13.1 外部ページ/外部メディアの参照 . . . 141
13.1.1 絶対 URL と相対 URL ex . . . 141
13.1.2 画像の使用 ex . . . 142
13.2 情報アーキテクチャ . . . 143
13.2.1 情報アーキテクチャとサイト構造 ex . . . 143
13.2.2 ページナビゲーション ex . . . 144
13.3 CSS と HTML によるページレイアウト . . . 145
13.3.1 ページの構成要素のグループ化 . . . 145
13.3.2 CSS グリッドによるレイアウト . . . 146
13.3.3 CSS のマージンとパディング . . . 148

 # 14 Web サイトの設計/製作 (総合実習) 151
14.1 Web サイトの設計と制作 . . . 151
14.1.1 Web サイトの設計/制作とは . . . 151
14.1.2 コンセプト (企画立案) . . . 151
14.1.3 デザイン・スタイルガイド . . . 152
14.1.4 設計・製作 . . . 153
14.1.5 公開・運用・保守 . . . 153

 # 15 ソフトウェア開発とテストケース 157
15.1 プログラミングと手順 . . . 157
15.1.1 高水準言語と低水準言語 ex . . . 157
15.1.2 JavaScript 言語 ex . . . 158
15.1.3 JavaScript による枝分かれの記述 ex . . . 159
15.2 ソフトウェア開発 . . . 160
15.2.1 ソフトウェア開発とは ex . . . 160
15.2.2 要求仕様の問題 ex . . . 161
15.2.3 テストとテストケース ex . . . 162
15.2.4 ソフトウェア開発プロセス . . . 165
15.3 Web ページ上の JavaScript プログラミング . . . 166
15.3.1 JavaScript によるページ要素へのアクセス . . . 166
15.3.2 JavaScript コードを自動的に動かす . . . 167

(empty page)

＃1　コンピュータの利用と認証

1.0　はじめに

本書は「コンピュータについて学ぶ」ことを目的としたテキストとなっています。ひとくちに「コンピュータについて学ぶ」といっても、その切り口は様々ですが、ここでは次のことが主題となります。

主題: コンピュータや情報システムは情報を処理する装置として、今日の社会で重要な役割を担うとともに、個人の知的活動に不可欠なツールとなっている。ここでは、全ての学生が学士力の一環として学んでおくべき、コンピュータと情報システムの基本原理を取り上げ、また、専門を問わず必要となる、コンピュータを用いた知的活動の基本的素養を身につける。

そして、学ぶ上での達成目標は次のようになっています。

達成目標: 簡単なプログラミングを通じてコンピュータの本質を理解し、コンピュータおよび情報システムの主要な構成要素と基本原理を Unix オペレーティングシステムを題材として学ぶとともに、情報倫理・情報セキュリティの考え方を身につけ、コンピュータを用いた情報の収集・交換・創出が行えるようになることを目標とする。

図 1.1 に 15 章から成る内容の構成を示します。内容は大きく 5 つの流れに分かれて各々が比較的独立していますから、(望ましくはないですが) どれか 1 つで分からなくなっても他にはあまり影響しないはずです。15 章のうち 2 章については「総合課題」となっていて、比較的大きな課題にグループで取り組み、長めのレポートを作成することを想定しています (その分、知識的な内容は少なくなっています)。また、各章のセクションに ex と記されているところは、必須内容 (例: 試験範囲) を表します。[1]

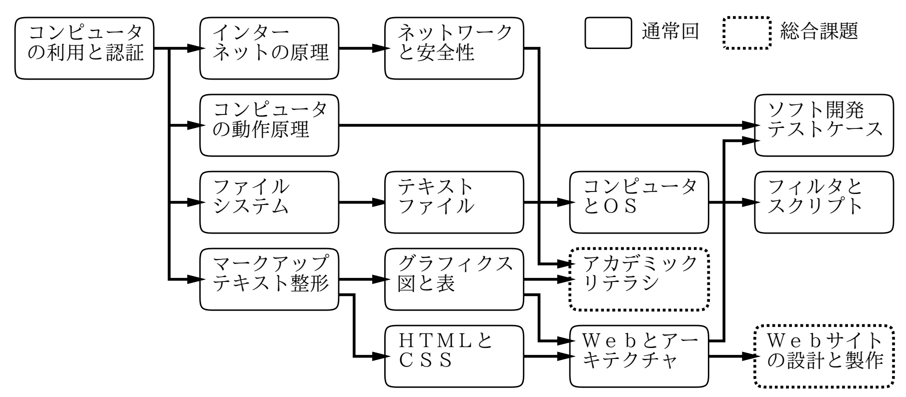


図 1.1: 本書の構成

勘違いしがちなのですが、これらの内容では「操作方法」を学ぶのは目的でなく、それぞれのテーマの「原理」を学ぶことが目的です。しかし原理はお話だけでは絶対に身につかず、自分で試行錯誤して体験することで始めて納得できます。

そのため、テキストは原理の説明のあと、具体例は「大まかなこと」しか書いてなくて、「具体的にやるとどうなるかを試して報告すること」が課題になります。ですから、各回の課題は五里霧中で苦労すると思いますが、その結果が学習成果になるものと思って頑張ってください。

[1] もちろん、 ex のついていない箇所も含めて一通りマスターして頂くことが望ましいです。

演習 1 皆様はこれまでの自分の学修活動を振り返って「どのようなスタイルの授業のときに」最もよく学んだと思いますか? また、それはなぜだと思いますか? さらに前記の質問において「よく学んだかどうか」をどのようにして評価していますか? 簡潔に述べなさい。

1.1　コンピュータとは何か

それではここから本題です。本テキストではコンピュータを中心的な題材として扱っています。自分なら次の質問にどう答えますか?

コンピュータとは何でしょうか?

たとえばまったくコンピュータのない世界からやってきた人に上のような質問をされたとして、あなたはコンピュータをどのように「定義」して説明しますか? なお、「○○するのに使うもの」という定義のしかた (実際に使う場面を例示するだけの定義) は避けてください。これは、コンピュータが使える場面が非常に多くあるため、有用な定義にならないためです。

次の演習は必ず自分でやってみてください。その上で資料の先を読むようにしてください。

演習 2 「コンピュータとは何でしょうか?」という質問に対する自分なりの解答を紙などに書き記してください。

コンピュータ (computer) という言葉はもともと「計算をする人」という意味でした。お金の計算でも技術的な計算でも「計算」が必要な場面は沢山ありますが、コンピュータ以前は人が計算するしかありませんでしたから、「このような計算をしなさい」という指示をうけて計算をする仕事の人がコンピュータだったわけです。そして今日のコンピュータも、この計算を人間より速く正確におこなうために作り出されました。

ではコンピュータは「計算を速く正確に行う機械」でしょうか? 誕生した時の意図はそうだったとしても、今はそうではありませんね。現在あなたは計算のためにパソコンや (そのポータブル版である) スマホを持ってはいないと思います。

では実際のところ、あなたは PC やスマホを何に使っていますか? (「○○するのに使う」を除外しておいて今更ですみません。) たとえば、天気予報などさまざまな情報を調べたり、音楽や映像を鑑賞したり、仲間とメッセージをやりとりしたり、文書を作成したり、というあたりかと思います。

確かに非常にバラバラですが、実はこれらには共通点があります。それは「**情報**を扱う」ということです (正確にはディジタル情報ですが、この点はしばらく置いておきましょう)。調べるものも、音楽や映像も、メッセージも、文書も、目や耳から入って来て、あなたの頭の中で取り扱い、時としてあなたの頭の中で作り出す「情報」には違いありません。

現代は情報社会であり、情報に大きな価値が置かれます。そして、コンピュータはそのさまざまな情報を取り寄せたり、よそに送ったり、保管したり、提示したりして、取り扱わせてくれる装置だと言えます。納得いただけましたか?

1.2　認証とパスワードの安全性

1.2.1　認証とその必要性 [ex]

認証 (authentication) とは「この人が確かに誰それであることを証明する」という意味の用語です。共用のコンピュータやサイトを利用するときには、最初に**ログイン**処理を行い、利用しようとしている自分が誰であるかを認証させます。その後実際の利用をおこない、利用が終ったら**ログアウト**処理により認証状態を取り消します (図 1.2)。

なぜこのような面倒なことが必要なのでしょうか? それは、コンピュータが情報を取り扱う装置だからに他なりません。つまり、あなたがコンピュータを使っている間には「あなたの」情報を保管したり受け取ったりするわけです。その中には他の人に見られたり変更されたくないものも含まれてい

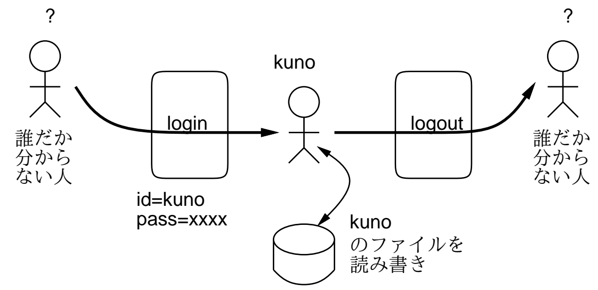


図 1.2: ログインとログアウト

ますね? そのような事態を防ぐには、まず「今使っているのが誰か」ということをきちんと確認するのが前提です。そのために認証が必要になるわけです。

なお、情報の保護について言えば、認証が済んだあと、こんどはデータごとに「これは持ち主だけが見てよい」「これはだれでも見られるが書き換えは持ち主だけ」のような情報が個別に設定されていて、それに基づいて制御されます。これを**許可** (authorization) と呼びます。そして、許可をきちんとするためには、今使っているのが持ち主かどうか分からなければ話になりませんから、まず認証がきちんとしている必要があるわけです。

さて、では認証は具体的にどのようにすればいいのでしょうか。たとえばあなたがスマホを持っていて**パスコード**を設定していない場合、その「機器を持っている」ことが認証として作用しています。いつも持っていて他人に渡さないなら、それを使っている人はあなた、ということです。

しかしこれでは、置き忘れたりしたときにそれを手に取った人が持ち主として操作できてしまい、大変危険です (よくあることですが)。そこで、パスコードを設定します。機器を持っていて、なおかつパスコードの秘密の数字 4 桁を正しく打ち込める人は、認証を通過し、持ち主として振舞うわけです。

では次に、パソコン室に置いてある共用の機器とか、ネット経由で接続できるサーバ上に、あなたの情報が置いてある場合はどうでしょうか? そのときは、まず機器やサーバの管理者が、そこを使うことを許された人に対して「**ユーザ ID**」を発行しますから、それが「あなた」を表します (そんなものを覚えるより姓名をそのまま使いたいでしょうけれど、管理が面倒ですし同姓同名というものもありますので)。そして次に、それと組になる秘密の**パスワード**があり、その 2 つを正しく入力できる人が「あなた」だと認証されるわけです。

なお、「アリババと 40 人の盗賊」では「ひらけごま」という「秘密の合言葉」だけで認証を行なっていました。この場合の「秘密の合言葉」は、いわばユーザ ID とパスワードを兼ねていますが、それがその 1 つしかないので、秘密の洞窟は「誰が今あけたのか」を認識できません。今日のコンピュータはそれだと困るので「誰」というユーザ ID と「確かにその人」というパスワードを分けているのです (さらに、システムを破ろうとする人にとってのハードルを高くする効果もあります)。

認証にはこのほかに「指紋」「網膜パターン」のように生体的な特徴を使ったり、物理的な「カード」「タグ」を持参したり、さまざまな方法があります。しかし読み取り機器が必要だったり設定が手間だったりするので、ID とパスワードとの組が一番広く使われるわけです。

1.2.2 パスワードとそれにまつわるトラブル ex

共用のシステムを使う場合には、自分のユーザ ID と、それと対になったパスワードを入力し、それが正しければ認証を通過できて (つまり確かにあなただと確認されて) システムを使える、というのがここまでの話でした。

パスワードにまつわるトラブルはおおむね次の 2 つがあります。ちなみに、(a) のようなことが起きやすいパスワードは「脆弱な (**脆弱性**のある)」パスワード、という言い方をされます。

(a) パスワードが破られてしまい、他人に認証を通過される。

(b) パスワードを忘れてしまい、自分が認証を通過できなくなる。

どちらも困ることですが、(a) はそんなに頻繁に起きないのに対し (b) はよくあることなので、つい (b) への対策として「簡単で覚えやすいパスワード」を使いがちです。しかし実際には、(b) が起きたときは管理者にお願いしてパスワードを再設定してもらえばよく、不便であっても危険はないのに対し、(a) は実際に起きたときは非常に危険です。どのような危険があるかは、ここで延々と説明しても身につかないと思うので、課題ということにしましょう。

演習 3 パスワードが破られ、他人に認証を通過された場合、どのような「よくないこと」が起きるか、思い付くものをできるだけ多く挙げてみなさい (コンピュータシステム以外の暗証番号なども含めてよい)。また、Web を検索して「実例」を 1 つ以上探してその要約を記し、自分が挙げたもののどれかに相当するか検討しなさい。

1.2.3　安全なパスワードのために ex

さて、パスワードが破られては困る、という前提は受け入れるものとして、ではパスワードが破られないようにするにはどのようなことが重要でしょうか? これは重要なことなので以下に記します。

- 誕生日や電話番号など他人に推測できるものを使わない。
- パスワードを見える場所に書き留めておいたりしない。
- 文字の種類を増やし桁数を増やす。
- 英単語など辞書に載っているものをそのまま使わない。
- 複数の箇所で同一のパスワードを使いまわさない

簡単に解説しましょう。まず、推測できるものを使えばパスワードが簡単に破られてしまうことは明らかですね。見える場所に書き留めておくのがダメなのも明らかです。

文字の種類はどうでしょうか? 今日のコンピュータは非常に高速ですから、沢山の場合を試してみることは簡単です。たとえばパスワードが 6 桁だとして、それが数字だったらたった 100 万通りしかないので、それを全部試すことは可能なのです。しかしもし、数字に加えて英字 (大文字、小文字) と記号を加えた 64 種類 (たとえば) の文字が使えるとすると? すべての場合は次のようになります (1 文字目が 64 通り、その全てについて 2 文字目が 64 通り、…なので、全部で 6 回かけ算します)。

$$64^6 = 68,719,476,736$$

さすがに 687 億通りを試すのはコンピュータでも一瞬というわけには行きません。桁数も多くしたらさらにこの数字が大きくなります。[2]

次の英単語はどうでしょう。辞書に載っている項目数というのは数万とか十数万とかですから、それだけ試すのは上記のように簡単なのです。では辞書に載っていないものを頑張って作って、覚えるのが大変だからそれをあちこちで使う? そうすると、1 箇所でパスワードが破られたときに (安全なパスワードであっても、たとえば入力時に覗き見されるとか、打鍵を記録する罠がしかけてあるなどで、他人に知られる可能性があります)、他のサイトまで一編に破られてしまいます。

しかし沢山の英単語でも何でもないものを覚えるなんて無理、と思うでしょうね。1 つの方法として、パスワードを管理するソフトやツールを使い、そこを見るためのパスワードだけは覚え、あとはそのソフトやツールに覚えさせる、というのがあります。ただし、そのソフトやツールが壊れた時の備えはお忘れなきように。

[2]では銀行の暗証番号が数字 4 桁なのはどうなの、というふうに思われるかも知れません。確かにかなり問題なのですが、歴史的事情でこうなっています。ただ、銀行カードを持参して ATM の前まで行った時だけが 4 桁の暗証番号のみで取り引きでるのであって、ネット経由などの場合はもっと別の認証も必須になっていて、一応安全のための配慮がされています。

そして、パスワードの作り方も工夫すれば、安全で覚えやすいものにできます。電通大では、共通アカウントのパスワードは「大文字、小文字、数字および記号がすべて含まれていて 12 文字以上」という要件があるので、それを満たすように考えます。

おすすめは、英単語、固有名詞または日本語のローマ字綴りを「2 つ」くっつけ、間に記号をはさんだり、一部を数字に置き換えたりすることです (日本語は語呂合わせが簡単なのでどんな語でも 1 つか 2 つは数字に置き換えられます)。大文字については語の頭だけでなく末尾とか、片方の単語全体を大文字にするなどの方法があります。

```
電通大 調布 → Den2Dai+CHOUFU --- (通→ two → 2)
湾岸線 首都高 → 1GANSEN@shutokoU --- (湾→ one → 1)
```

ぜひ自分でいくつかやってみてください。

演習 4 自分にとって覚えやすく脆弱でないパスワードを考え、共通アカウントのパスワードをそれに変更しなさい。

1.3 キーボード入力の重要性

本テキストで使用を前提とするコンピュータには**キーボード**がつながっていて、文字はキーボードから入力します。当り前だと思いましたか? しかし今日、既に多くのコンピュータユーザはキーボードで文字入力をしていません。そう、スマホのことです。仮想キーボードとか、フリック入力とか、音声入力とかを使うわけですね (音声入力は相当精度が向上しています)。

ではなぜ今更、ここでキーボード入力をやるのでしょうか。それは、コンピュータに対する情報入力手段としてキーボードより効率のよい方法はまだ見つかっていないからです。

文章だけであれば音声入力はかなり強力になって来ましたが、プログラムを打ち込んだりデータを打ち込んだり、また文章と一緒に数式や整形のためのコマンド (マークアップ) を打ち込んだり、ということになるとまだまだです。それに会合の場や大部屋の研究室など、「勝手に喋れない」場所では音声入力が使えません。

というわけで、以下ではキーボード入力を前提としてさまざまな演習をやって頂きます。そのとき大切なのは、「5 本の指を使って」「キーボードを見ないで」入力する方法 (**タッチタイピング**、タッチメソッド) をマスターしておくことです。入力時にいちいちキーボードを見ていると、視線の移動が大きく疲労が増しますし、入力速度も「見る→場所を確認→打つ→見る」というサイクルの反復のために速くできません。

本テキストでは今回タッチメソッドの原理と練習方法を示すだけで (付録参照)、以後毎回タッチタイプの練習をするようなことはしませんが、最後の章が終わるまでに各自で練習して身につけることを強く勧めます (タッチメソッドは合計数時間程度の練習で身につきますし、その結果従来よりずっと疲労が少なく高速にキーボードが打てるようになったら絶対にお得です)。

課題 1A

今回の課題は「演習 3」を実施し、結果をレポートとして報告して頂くことです。LMS の「assignment ＃ 1」の入力欄に入力してください (直接打ち込むとログイン時間切れになった時に内容が消失するので、メモ帳など他のソフトで作成して完成後にコピーすることを勧めます)。以下の内容がこの順に含まれるようにしてください。

- 冒頭に題名「コンピュータリテラシレポート＃ 1」、学籍番号、氏名、提出日付を書く。
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 以下のアンケートに対する回答。

Q1. 今回の内容についてどれくらい既知でしたか。

Q2. 科目の運用や進め方についてどう思いますか (何がよい/悪い等)。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

1.4 付録: タッチタイピング入門

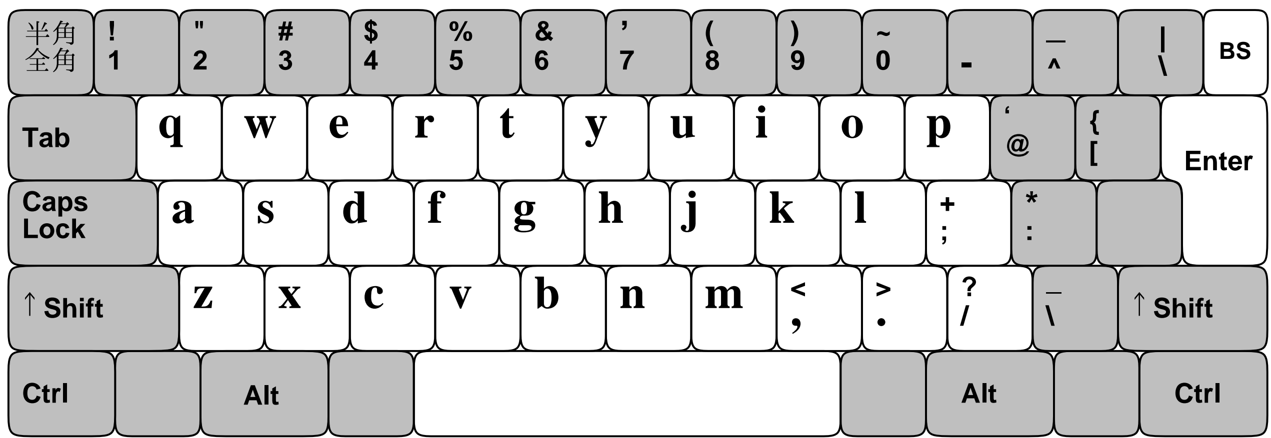
1.4.1 はじめに

この付録では、「タッチタイピング」を学ぶやり方を説明しています。タッチタイピングとは、コンピュータのキーボードを「キーを見ないで」打つ方法です。かっこいいけど、私にはむずかしそう! と思いますか? いいえ、そんなことはありません。誰でも、あわせて数時間、きちんとしたやり方で学べばマスターできます。

タッチタイピングは、いちばん疲れずに、スムーズに、速くキーボードを打つやり方ですから、体のためにもよいのです。そして、考えていることが滑らかにコンピュータに打ち込めますから、考えたことを書きとめる、ベストなやり方の一つです。ぜひ、マスターしてください。

1.4.2 キーボードとソフト

ここでは、皆さんの使っているパソコンのキーボードのうち、一部のキーだけを使います。具体的には、下の絵で白く描かれているキーです。パソコンによってキーボードは少しずつ違いますが、白くないところは使わないので、違っていても構いません。英字のキーには大文字が書かれていますが、この練習では小文字だけ打つので、絵では小文字が書いてあります。



練習には、「メモ帳」など打った字がそのまま入るソフトを使います。最後は日本語を打つことになるでしょうけれど、ここでは練習なので英字がそのまま入る状態でやってください。

1.4.3 ホームポジション

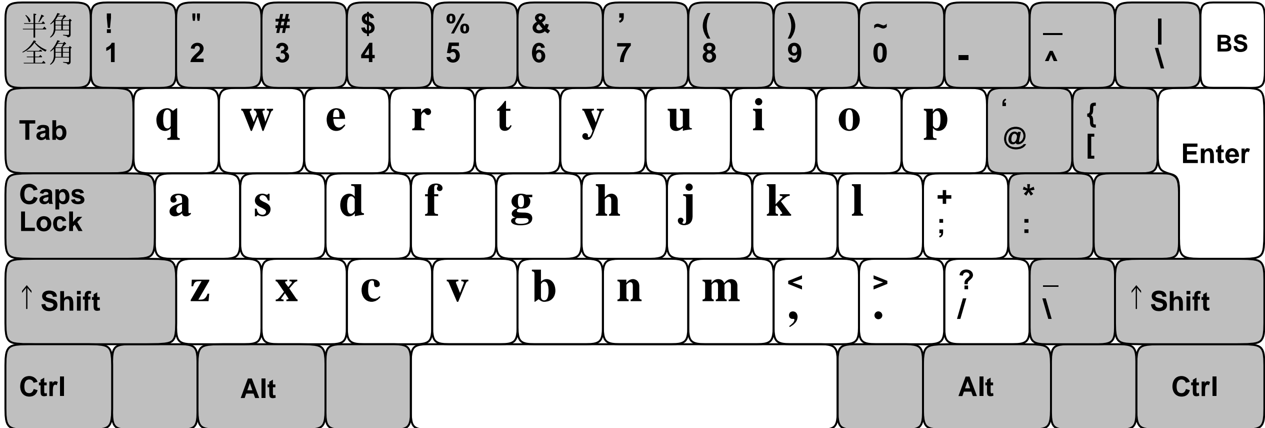
タッチタイピングで大切なのは、キーボードを見ずに練習することです。そこで:

練習は必ず「両手の上にハンカチをかぶせて手もとが見えない状態で」行ってください。

どこにどのキーがあるか忘れた時は、手もとは見ないで、絵を見てください。そのため、どのページにもキーボードの同じ絵を載せています。

皆さんが使う指は、親指を除く左手と右手の 4 本ずつの指です。親指は、スペース文字を打つためのキーであるスペースバーにだけ使います。そして皆さんがまず覚えるのは、「**ホームポジション**」です。左手の 4 本の指を`a`、`s`、`d`、`f`、右手の 4 本の指を`j`、`k`、`l`、`;`に置いてください。親指は軽くスペースバーに載せます。目をつぶって、確かめてください。人さし指のところのキーが、出っぱりがあったり、窪んでいて、それと分かるはずです。

キーボードから手をは離してください。次に、キーボードを見ないで、ホームポジションに戻してください。見なくてもさっともどせるように、繰り返しやってみてください。



1.4.4 打ってみよう!

では、ホームポジションに手を置いたまま、キーを打ってみましょう。ハンカチはかぶせてありますね。ゆっくりでいいので、次のとおり打ちましょう。

```
asdf jkl; asdf jkl; asdf jkl; asdf jkl;
asdf jkl; asdf jkl; asdf jkl; asdf jkl;
```

間をあけるところは、好きなほうの親指でスペースバーを打ってください。適当な長さまできたら、右はしのエンターキーを打って行をかえてください。エンターキーを打つには、右手の小指をえいっと右に伸ばせばよいです。打ち終わったら、小指をホームポジション (;のところ) に戻すのを忘れないこと。これを何回か繰り返し、練習してください。繰り返すことで指が覚えるので、このテキストの枠内の練習課題はすべて、「10 回」繰り返して打つことをすすめます。

1.4.5 ホームポジションの指づかい

では次は、ホームポジションに指を置いたままで打てるローマ字づかいを練習してみます。ローマ字なので、読みあげながら打ってみてください。打とうとしているキーがどの指だったか分からなくなったら、絵を見て確認してください。

```
sasasa dadada dasadasa asasa asada fasafasa fadafasa dasafasa
akasaka kasakasa jakajaka akada kakada fakaja jakasa dakasa kajada
```

急ぐ必要はないので、ゆっくりと、何回かくりかえして、間違えずに打てるようになるまでやってください。間違えたときは、打ちなおさなくてもよいです。そのかわり、その言葉のはじめから、もういちど打ってください。エンターキーでの行かえは、好きなところで入れてかまいません。

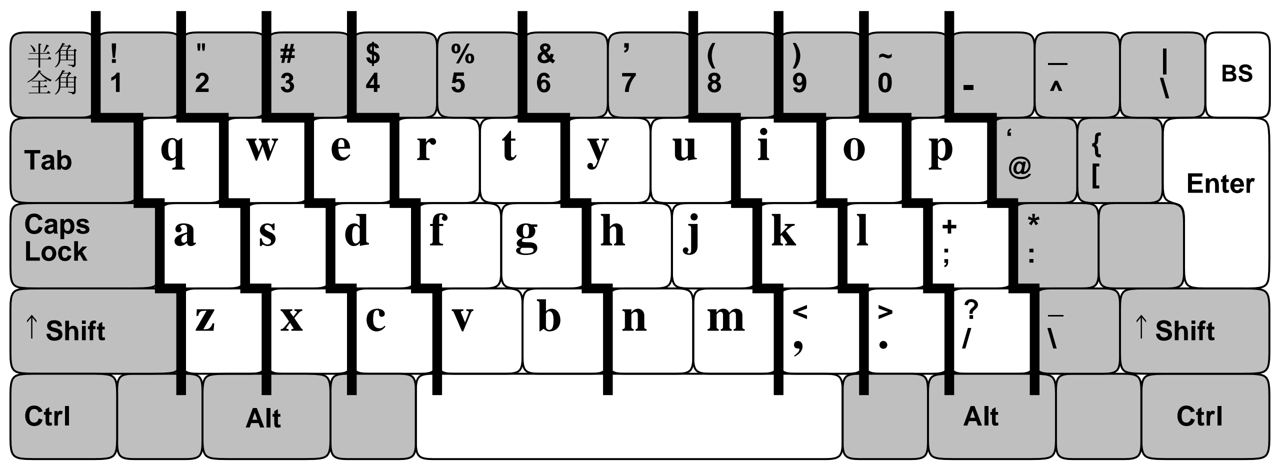
1.4.6 g と h の追加

では次に、ホームポジションから人さし指を内側に動かしたところにある、gとhを練習しましょう。

```
fgf jhj fgf hjh fggf jhhj fggh jhhj fghj fghj jhgf jhgf
asdfg hjkl; asdfg hjkl; asdfg jhkl;
```

では次に、gとhを入れたローマ字と (ローマ字で使わないキーもあるので) 英単語を練習します。ゆっくりでいいので、繰り返し、練習してみてください。

```
gasagasa gadagada gaka hakka hakasa haga saga asaja kaga hada
fall flash dags flag gala half lask shall salad slash gas glass
```



1.4.7 左手の上下段

中段をマスターしたところで、上と下の段に進むことにします。この時は必ず、図にあるように指を「斜めに」移動して対応する指で打ってください。これを守らないと覚えられません。

左手からやりましょう。指の受け持ちは、a担当の小指が、その上のq、その下のzも打つ、というふうになります。動かして 1 文字打ったら、必ずホームポジションに戻すこと! どの指か分からなくなったら、絵で確認してね。繰り返し打って、覚えましょう。

```
aqa aza sws sxs ded dcd frf fvf ftf fgf fbf
aqa aza sws sxs ded dcd frf fvf ftf fgf fbf
```

次はローマ字の指づかいと、(ローマ字で使わないキーもあるので) 英単語です。だいぶいろいろ打てますねこれも、繰り返して練習してください。

```
wakasa zakka wakka zawazawa wazawaza dekedeke dekadeka chacha kesa
sake eda eja eka eeka rakka rekka taka geta baka barabara garagara
gabagaba taraba saraba bakara bakera kabara taraba karate hatarake
far wave serv zee fez dave call beg dart cart vent qere bad ward
bed save zebra get cat fast west better east far test taste best
```

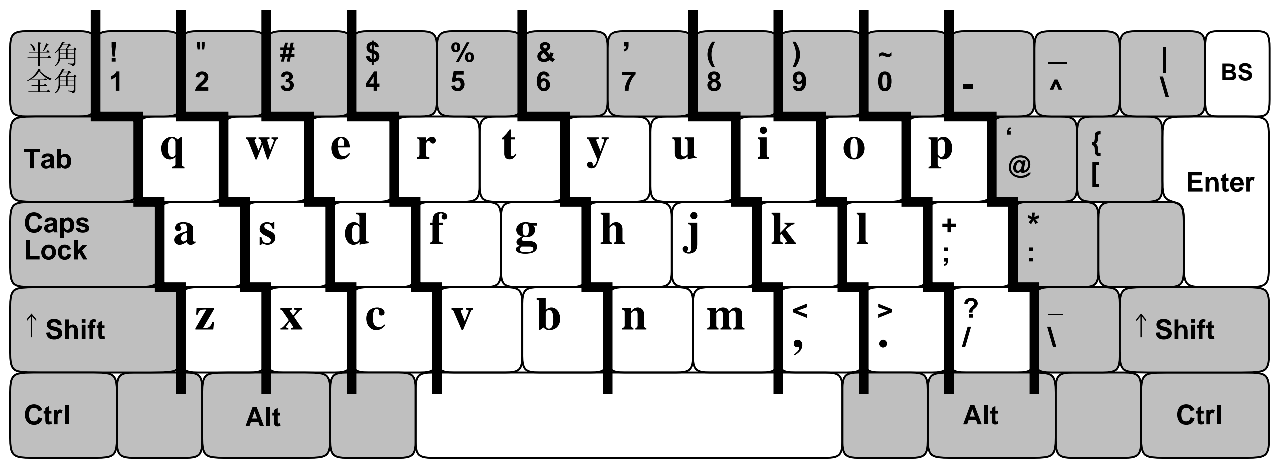
1.4.8 右手の上下段

いよいよ、右手の上と下の段をやって、これでおしまいです! ,と.は「、」と「。」を打つのに使います。繰り返し練習してください。

```
juj jyj jhj jnj jmj kik k,k lol l.l ;p; ;/;
juj jyj jhj jnj jmj kik k,k lol l.l ;p; ;/;
```

右手の上段には「u」「o」「i」の母音があるので、多くの単語が右手で打てます。左手中段まで合わせて、練習してみましょう。

```
nippon koujou himiko nomi yuki konomi uni yoko jiku moji mikon
honkon yuuki mikonn miyuki koiji hako umi yuuhi yuuki yumomi kippu
yudann jikann nigiyaka namiki jama yonaka kodama kandan hondana
yukiyama nakayama mamorigami oodama sonoki panda sippo happyou
yukiha, samui. kumoha, ukanda. yamaha, ookii. umiha, aoi.
```



1.4.9　総合練習

全てのキーを使ったローマ字と英文の指づかい練習です。少し多めに用意したので、あせらずに繰り返し練習してください。

```
watasiga asa okiruto, sotodeha amega futteita.  youjiga attanode,
kasawo sasite dekaketa.  machiniha ookuno hitotachiga kaimonowo
siteita.  watasiha honnyani haitte zassiwo busshokusita.

tokaide kurasite iruto, tamani sizenno nakani dekaketaku naru.
sikasi, inakani sumitaikato iuto souiu kimichiniha naranainoda.

there was a port in front of my house.  when very large ships arrive,
the port was extremely crowded with a long queue of passengers.
in everyday life, it was very quiet and was my best playing place.

in such event as college entrance examination, we work hard
just to attain our goals.  however it is questionable whether
we can live without any relaxations.
```

演習 T 各レッスンが仕上ったと自分で判断したら、「タッチタイピング練習ページ」で 1 分あたり打鍵数を計測してみなさい (手にハンカチをかぶせること)。期末までに「L5. 総合課題」で「100 文字/分」に到達した場合、評価に加点がある。

＃2　インターネットの原理

今回の目標は次の通りです。

- Unix システムへのログインとコマンド実行ができるようになる — 本科目でも他の科目でも Unix 上で実行したり Unix を題材とすることは多いので Unix を使いこなせる必要があります。
- インターネットに関する主要な概念を理解する — ネットワークは使うだけでなく研究の対象や実験の土台ともなるので、原理や特性を知っておく必要があります。

2.1　Unix システムの利用

2.1.1　Unix システムへの接続

前回、コンピュータは「情報を処理する装置」ということをお話ししました。また後の回で改めて出て来ますが、コンピュータの動作はすべてプログラムを実行するかたちで行われます。

ということは、あなたが手元のコンピュータの電源を入れた後、マウスで画面を操作したりするのも、すべて「そのような動作をしてくれるプログラム」が動いて行われているわけです。この部分を OS (**オペレーティングシステム** — Operating System) と呼びます。コンピュータの操作方法や主要な機能は OS によって変わってきます。ここで説明する **Unix** もその 1 種別です。

皆様が普段目にする OS は PC(パソコン) 用の OS である Mac OS や Windows だと思いますが、Unix はネットワーク関係の機能が豊富に準備されていて、インターネット上でさまざまなサービスを実現するために多く使用されています。また、皆様が研究室で研究したり論文を書くのにも Unix が多く使われています。そういうわけで、この科目の実習を行う手段として Unix を主に用い、ついでに Unix に習熟して頂きます。

演習には、演習室の PC 上で動いている Unix システムと、情報基盤センター内で動いている共用の (大きな)Unix システムの 2 つを使います (演習ガイド参照)。多くの演習はどちらでも行えます。共用のシステムは学外からでも接続でき、演習を行うことができます。[1]

2.1.2　CUI と GUI ex

Unix では、多くの作業は**端末** (Terminal) と呼ばれるプログラムの窓を開き、そこで**コマンド** (command、指令) を打ち込んで実行します。ここでできることは、コンピュータにやらせたいことを、文字の並びであるコマンドの形でコンピュータに与えることだけです。そして、コマンドの出力も同じ窓に続いて出て来ます。このようなコンピュータの使い方を **CUI**(Character User Interface または Command User Interface) と呼びます (図 2.1)。

これに対し、普段皆様が PC で使っているような、画面にデスクトップやメニューなどが表示され、それをマウスなどで指して操作するようなものは **GUI**(Graphical User Interface) と呼びます。

GUI の方が分かりやすそうなのに、なぜ CUI があるのでしょうか。それは、次の理由によります。

- ネットワーク経由で遠隔地から操作する場合は、手前側と向こう側の間で通信に時間が掛かったり、両者のソフトウェア (たとえば OS) の違いがあり、GUI が使えないことがある。

---

[1] 共用システムに学外から接続するためには「SSH プロトコルに対応した遠隔ログインソフト」が必要です。Mac OS では標準の `ssh` コマンドが使えます。Windows では、たとえば PuTTY というフリーソフトを導入して使えます。

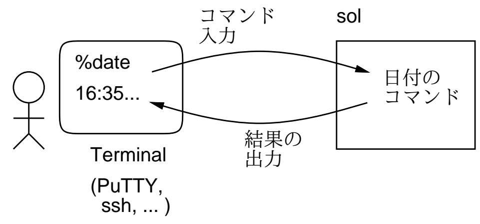


図 2.1: CUI の概念

- コンピュータを操作することの基本は「これこれの動作をしなさい」とコンピュータに指示することであり、それにはコマンドで指示するのが一番直接的かつ応用範囲が広い。[2]
- コマンドは覚える手間が掛かる代わりに、いちど覚えてしまえば GUI で操作するよりも高速である。つまり CUI の方が「プロ向き」である。

コマンドを覚えるのが大変そうだと思われるかも知れませんが、これから毎回テーマごとにいくつかのコマンドが出て来るだけですから、演習をきちんとやっていれば問題ありません (皆様がこれまでに英単語を覚えて来た数と比べたら、合計でも 100 分の 1 未満だと思われます)。

演習 1 Unix システムの端末を開きなさい。すると「`...$`」のような文字列が表示されると思います。これを「プロンプト」と言い、コンピュータからの「さあコマンドを入力してください」というメッセージになっています。その状態で次のことをやってみなさい。

a. 「**date**`[RET]` [3]」を実行する。このキーの通常の用途は「1 行終り (改行)」だが、CUI では「コマンドを実行せよ」という意味になる。`date` は日付と時刻を出力するコマンドなので、画面に日付と時刻が表示され、その後再度プロンプトが表示される (以後は `[RET]` の記載は省略します)。
b. 「**exit**」を実行。これは CUI の実行を終了するコマンドなので、端末が閉じたり、ネットワーク経由で使う場合は接続が切れる (確認したら再度開始)。
c. 「**sleep** `5`」を実行。これは「5 秒間待って終了する」コマンドなので、少し待たされ、コマンドが終了すると (出力はないので) 次のプロンプトが現れる。このように、コマンドの後「 」(空白文字) で区切って「追加指定」(オプションやパラメタと呼ぶ) を指定する場合がある。
d. 「`sleep 100`」を実行する。そして (100 秒は待ちたくないので)、「`^C`」(Control キーを押しながら C のキー) を打つと、コマンドが中止され次のプロンプトが現れる。間違って長くかかるコマンドを開始してしまった際はこのように「`^C`」を使う。

なお、「**man** コマンド名」で指定したコマンドのマニュアルが見られます (`date`、`sleep` で試してください)。`man` は 1 画面より長い情報を見るためのコマンド `less` を呼び出します。`less` では次の 3 つのキーが使えます: [SP](空白) →次の画面を見る、「`b`」→ 1 画面戻る、「`q`」→終了。

2.2 ネットワークの基礎概念

2.2.1 イントロ: 地球の裏側までどのくらい? ex

今日の社会では、世界のすみずみまにでまたがるネットワーク (インターネット) は欠かせない機能です。しかし皆様は、単に便利に使っているだけで、仕組みを学んだり、調べたりしたことは無いと思います。一方、この科目は原理・仕組みまで考えることがテーマですから、そのようなことをしていきます。まず手始めに、次の疑問を考えてみることにしましょう。

[2]GUI ではメニュー等に含まれていない機能は使えない。このため、Windows や Mac OS でも、頻繁に使わない機能はコマンドで起動する必要がある。
[3]RETURN ないし ENTER と書かれたキーを意味する

インターネットで地球の裏側 (ブラジル) まで行ってくるのに、どのくらいの時間で済むの?

少し手をとめて、片道どれくらいの時間だと思うか、あなたの予想をメモしてください。

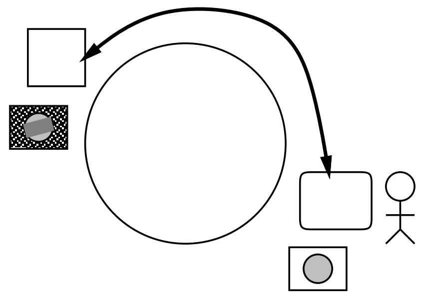


図 2.2: ブラジルまで行って帰ってくる時間は?

では実際に測ってみましょう。それには次のようにすればよいのです (図 2.2)。

- Google などで「ぜったいにブラジル現地にありそうな施設」の Web ページを検索して探す。ここでは「ブラジル行政府」を見つけました。
- そのサーバ名 (ブラウザのアドレス窓に出る名前) の部分にある名前「`brasil.gov.br`」を指定して **ping** コマンドを実行し、「パケット往復時間」を調べる (以下の表示にある**バイト** (byte) というのはデータ量の単位で、1 バイトは 8 ビットです。パケットについては後述)。

```
...$ ping brasil.gov.br
PING brasil.gov.br (170.246.252.243): 56 data bytes
64 bytes from 170.246.252.243: icmp_seq=0 ttl=237 time=308.461 ms
64 bytes from 170.246.252.243: icmp_seq=1 ttl=237 time=309.451 ms
64 bytes from 170.246.252.243: icmp_seq=2 ttl=237 time=308.437 ms
^C
--- brasil.gov.br ping statistics ---
4 packets transmitted, 3 packets received, 25.0% packet loss
round-trip min/avg/max/stddev = 308.437/308.783/309.451/0.472 ms
...$
```

- 往復時間が 300msec 程度だから、片道は 150msec 程度と分かる。

演習 2 各自の端末から直接外部へ ping はできないので、学内システムへの到達時間を計ってみてください。

- `joho.g-edu.uec.ac.jp` — 情報部会 Moodle
- `www.uec.ac.jp` — 電通大対外 Web サーバ
- `www.office.uec.ac.jp` — 電通大事務系 Web サーバ

また、「ping テスト」で検索すると複数の外部の ping サービスサイトが見つかります。そのページで調べたい相手先を (上記と同様に) 指定すると、そのサイトで ping を実行して結果を表示してくれます。試してみてください。また、スマホで ping や traceroute(後述) を実行できるアプリを使うこともできます

注意! 多人数で 1 つのマシンからあちこち `ping` すると攻撃と勘違いされるので、共用システムではやらないこと! 個人の PC やスマホ、または外部のテストサービスでやること。

a. どこか「ネット的に遠い」と思う地域を選び、そこにある施設 (複数) までのパケット往復時間を計測して「遠さ」を確認してみなさい。できれば、そこまで行くのにどこを通っているのか予想してみて、その地域までの時間も計測して確認みるとなおよいでしょう。

b. 世界の複数地域までのパケットの往復時間を調べ、およその傾向をまとめなさい。できればそれらを整理して世界のネットワークのつながり方の予測できるとなおよいでしょう。[4]
c. 本来なら遠くにあると思われる施設のサーバへの往復時間が異常に速いものの例を調べて挙げてみなさい。できれば、どのようなサイトついてそのようなことがなされているかの傾向を分析できるとなおよいでしょう。

2.2.2　ネットワークとその目的 ex

コンピュータ**ネットワーク** (computer network) とは何でしょうか? 形としては、複数のコンピュータシステムが互いに通信できるように接続されたものがネットワークです。では、ネットワークを構成する目的は何でしょうか? 次のものが挙げられます。

a. 資源の共有 — あるコンピュータ内のデータを別のところから利用する (データの共有)、あるコンピュータの能力が不足したら一部をよそで処理する (処理能力の共有)、など。
b. 信頼性 — 1 台のコンピュータだとそれが止まったら処理全体が停止するが、複数のコンピュータをネットワークで結合したものなら、1 台が壊れても残りで処理を進めるようにできる。
c. 経済性 — 大きなコンピュータ 1 台で色々やらせるより、量産型のマシンを必要な台数だけネットワークで結合する方が廉価。
d. 段階的成長 — 1 台のコンピュータで能力が不足したら、もっと大きいマシンに置き換える必要があり大変だが、複数台のネットワークであればマシンを追加することで能力を増やせる。
e. 通信媒体 — 距離的に離れたシステムどうしを接続することにより、新しいタイプの応用が可能になる。

どの目的を主とするかによって、ネットワークの形態は異なります。たとえば、信頼性や経済性のためにネットワークを構成するのなら、その各コンピュータ間の距離は近く、それらの間を高速なネットワークで結ぶことになります (**LAN** — Local Area Network、局所ネットワーク)。一方、離れたところとのデータ共有や通信が目的なら、そのネットワークは長い距離を結ぶものでしょう (**WAN** — Wide Area Network、広域ネットワーク)。

2.2.3　回線交換とパケット交換 ex

古くからあるネットワークの代表例は固定電話です。電話はもともと、2 つの機器の間を直接繋ぐと遠隔地で通信ができるというところから始まりました。そして、個人の家から電話が引けるようになった後も、電話局で交換手が配線を繋ぎ、最終的に自分と相手との間に線がつながると通話できる、という点は同じでした。その後技術の進歩により、番号をダイアルすると自動で相手につながる等の変化はありますが、「最初に線を確保し、その後通信し、終わったら切断する」という点はずっと変わっていません。これを**回線交換** (line switching) と呼びます。[5]

回線交換の弱点は、最初に線を確保する手間が必要で、線が確保できないと通信もできないことです。利点は、一度つながった線は自分たち専用に確保され、常に 100%使えるということです。[6]

一方、インターネットの方式はこれとは違っています。データは上限サイズが決まった**パケット** (packet) という単位で、ネットワークに常時送り出せます。各パケットには宛先が書いてあり、中継点ごとに宛先の方向に中継することで最終目的地まで到達します。これを**パケット交換** (packet switching) と呼びます。パケットとは「葉書」「小包み」だと思えばよいでしょう (図 2.3)。先の `ping` コマンドはこのパケットの往復時間を測っていたのでした。

パケット交換の利点と弱点は先の裏返しです。パケット交換では接続の手間が無く、経路の途中では多数の通信のパケットが「相乗り」するので回線が有効活用できます。ただし、通信量が多くなり

[4] 後から出て来る `traceroute` を使ってその正しさを確認することができます。
[5] 携帯電話も固定電話も今日では実態はパケット交換になりました。
[6] 話している二人が沈黙している間も線は確保されたままなので、無駄があるとも言えます。

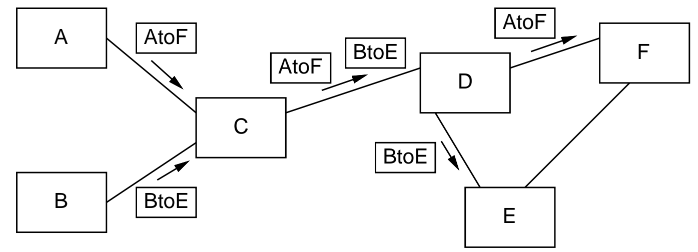


図 2.3: パケット交換の原理

すぎると、既に通信している相手どうしでも通信しにくくなります (到達時間が長くなったり、パケットが失われます)。また、1 パケットに入らない大きなデータを送る場合には、複数のパケットに分けて送り、相手先で組み立てる必要があります。そして、パケットは独立に送られるので、一部だけ無くなったり、途中で追い越しが起きたりします (それにどう対処するかは、すぐに説明します)。

2.2.4 システムと階層構造 ex

コンピュータシステムもそうですが、多くのシステムは「**階層構造**」(layered architecture) を持ちます。その基本的な考え方は、「全体を層状に構成し」「下の層の詳細は上の層に影響しないようにする」ことです。

たとえば、あなたが遠方の相手方と取り引き契約の交渉をするために、その書類を秘書に送らせるとします。秘書が文書を送るのには「郵便」「クロネコメール」「FAX」など複数の手段がありますが、それはあなたは関知する必要がありません。秘書はどの手段を使うかで作業内容が違いますが、郵便を使うとして、郵便ポストに入れたら、郵便局が集配に自動車を使うか、バイクかといったことは関知する必要がありません (図 2.4)。逆に秘書はあなたの検討の内容には関知しないし、郵便局は秘書が送ったものの内容には関知しないわけです。

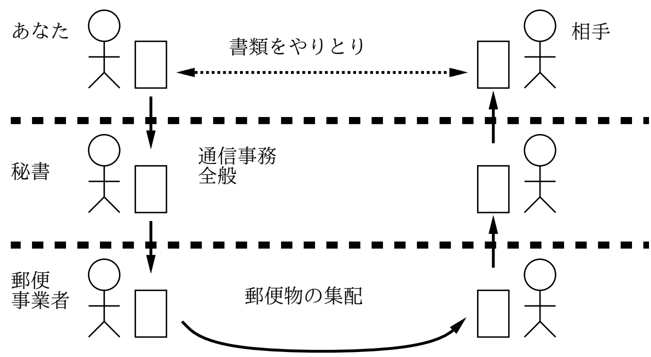


図 2.4: 階層構造

階層構造を用いると「それぞれの層の中をうまく構成する」ことと「隣接する層とのやりとりをする」ことだけに限定して考えれば済むので、問題の複雑さが小さくなります。ネットワークシステム全体もそういう風に構成されているわけです (そうしないと作れないくらい込み入っている)。

2.2.5 プロトコルとその考え方 ex

次に**プロトコル** (protocol) について説明しましょう。プロトコルとは「通信規約」などと約されることもありますが、「同じ階層の相手どうしでうまくやりとりするための約束ごと」です。先の例では、値段の交渉のしかたは「あなた」と「相手」のプロトコルですし、宛先を指定して書類を受け渡すのはそれぞれの「秘書」さんどうしのプロトコルです。

ネットワークは非常に多数の機能の集まりですから、それらの機能を実現する多数のプロトコルを必要とします。それらのプロトコルは、上で説明したように階層構造になっています。1 つの層を 1

つのプロトコルが占めている場合もありますし、層によってはさまざまな用途に応じて複数のプロトコルが使い分けられるものもあります。

先に「パケットは失われたり追い越しがあったりする」という話をしました。しかし実際には、たとえば音楽ファイルを送った時に、曲の途中が抜けたりフレーズが入れ替わったりすることはありませんね? これも、プロトコルの機能によってエラーのない伝送が実現されているからです。

どうしたらそんなことが可能になるのでしょう それは例えば、次のようにするのです (図 2.5)。

1. 送り側は送り出すパケットに一連番号をつける。
2. 受け側は相手からパケットが来たら**確認**パケットを送る。
3. 送り側は一定時間待っても確認が来なかったら**再送**する。
4. 受け側は一連番号の順にパケットを正式に受け取る。

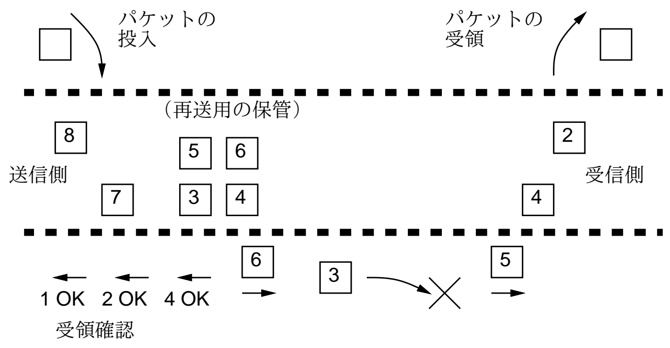


図 2.5: エラー制御のある伝送

このようにすると、下の階層でエラー (パケットの**喪失**等) があっても、上の階層にはエラーのないサービスが提供できます。ただし、下の階層でのエラーに対処するためには「再送」「確認待ち」などが必要であり、エラーが多くなればなるほど伝送は (上の階層から見て)「遅く」なるように見えます。

演習 3 3〜4 人の 1 グループで、うち 2 人が送信側と受信側になって背中合わせに座り、英字 10 文字程度のメッセージを送るゲームをします。残った人はネットワーク役としてパケット (メモ紙) を送信側から受信側またはその逆に運びますが、時々(2 回に 1 回くらい?)、a〜c にある「普通でないこと」を起こします。1 パケットには「4 文字」までしか書けず、紙の手渡し以外の動作 (合図等) は禁止です。a〜c それぞれについて、正しく送るための (送信側と受信側の間での) 約束を工夫しなさい。[7]

a. 送ったパケットは必ず 10 秒以内に到着するが、時々「×」印だけのパケットに入れ替わっていることがある。
b. 送ったパケットは到着することもしないこともある。到着する場合は 10 秒以内に到着する。
c. 送ったパケットは到着することもしないこともあり、到着する場合も遅延することがある (後から送ったパケットに追い越されたりすることもある)。

2.3 インターネットのプロトコル

2.3.1 TCP/IP のプロトコル群 ex

ここからは、インターネットで使われているプロトコル群である **TCP/IP** を題材として見ていきます。TCP/IP にはこれまで長く使われてきた **IPv4** と、近年普及してきている IPv6 とがありますが、以下では IPv4 について説明します。ここで述べる範囲では、アドレス表記やプロトコルやプログラムの名前が違う程度でして、両者に原理的な違いはありません。

[7]ヒント:「正しく受信した」ことを示す確認パケット (ACK) を返送することはどの場合も必要になります。ACK が×になったり喪失・遅延する可能性もあることに注意。

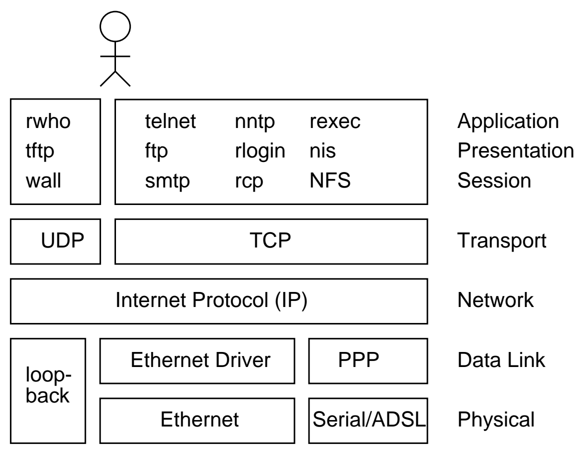


図 2.6: TCP/IP のプロトコル構成

TCP/IP の階層構造とそこに含まれる主要なプロトコルを図 2.6 に示します。中間の「**ネットワーク層**」として書かれている **IP**(Internet Protocol) はすべてに共通していて、その上や下では層ごとに複数のプロトコルがあります。

下側では、どのような媒体 (無線 LAN とか光ファイバーとか) であるかで異なるプロトコルになります。IP より上側のプロトコルは「**トランスポート層**」のところで **UDP**(User Datagram Protocol) と **TCP**(Transmission Control Protocol) に分かれます。UDP は単独のパケットを送受するもので、少量のデータを迅速にやりとりしたい場合に向いています。一方 TCP は前の節で説明したような考え方でエラーの無い通信を実現しています。それより上では、電子メールとか、WWW とか、サービスによってプロトコルが違っています。

2.3.2 IP アドレス ex

先にパケットの説明で、各パケットに「宛先」が書いてあると述べました。TCP/IP ではこの宛先として指定するものを **IP アドレス**と呼びます。パケットを届ける先は個々のマシン (**ホスト**) ですから、IP アドレスはホストごとに別の値が割り当てられています。

TCP/IP が今の形になった 1980 年ごろには、ネットワークの規模はごく小さく、通信は遅く、コンピュータも非力でした。パケット方式ではすべてのパケットに宛先を入れるため、その形がコンパクトであることが大切でした。そこで IPv4 では、アドレスとして「32 ビットの値」を使うことにしました。我々が普段目にする「210.154.96.162」のような書き方は、32 ビットを 8 ビットずつに区切って、各バイトの値を 0〜255 の範囲の数値として表記したものです (図 2.7 上)。

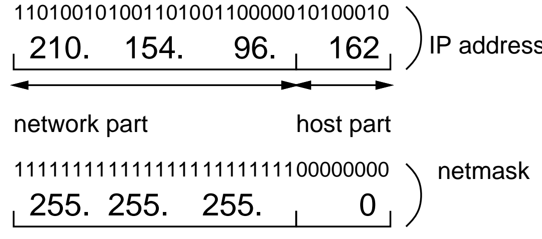


図 2.7: IP アドレスとネットマスク

32 ビットの IP アドレスは**ネットワーク部** (network part) と**ホスト部** (host part) に分かれています。これはどういうことかというと、1 つの LAN では、そこに接続されている複数のホストや機器にそれぞれ違うホスト部を割り当て、ネットワーク部はすべて共通にします。これによって、外部からはネットワーク部だけ見てパケットを送れば済むようになるのです。

ネットワーク部とホスト部の境界はネットワークごとに違うので、必要のつど指定する必要があります。その指定は、ネットワーク部が全部「1」、ホスト部が全部「0」のビットになった 32 ビットの

値を使うのが普通です。これを**ネットマスク** (netmask) ないし**サブネットマスク** (subnet mask) と言います。ネットマスクも通常、8 ビットずつに区切って各バイトを 0〜255 の 10 進数として表現します (図 2.7 下)。[8]

IP アドレスは、世界中 (インターネット中) で重複がないように管理されていて、ある特定の IP アドレスを持つホストは世界中でただ 1 つしか存在しないようになっています (実際にはネットワーク番号を重複しないように各組織に割り当て、ホスト番号はその組織のネットワーク管理者が割り当てます)。これによって、世界中のどこからでも「あのコンピュータ」という指定をすればそのコンピュータに向かってパケットが転送されて来るようになるわけです。[9]

2.3.3　インターネットの構造 ex

IP アドレスが分かったところで、次はインターネットの全体像を見てみましょう。皆様は自分の使うマシンからインターネットまで、どうつながっているのか考えたことがありますか?

図 2.8 に簡単な概念図があります。左上の方にある「C」と書かれた箱のあたりが、自分のコンピュータだと思ってくださささい。多数のホストを持っているところでは、それらのマシンは**スイッチ** (switch) ないし**ハブ** (hub) という装置によって相互接続されていて、1 つの LAN を構成しています。個人の家のように、マシンが少数しかない場合でも、今日では無線 LAN を使うのでやはり LAN になっていることが多いでしょう。

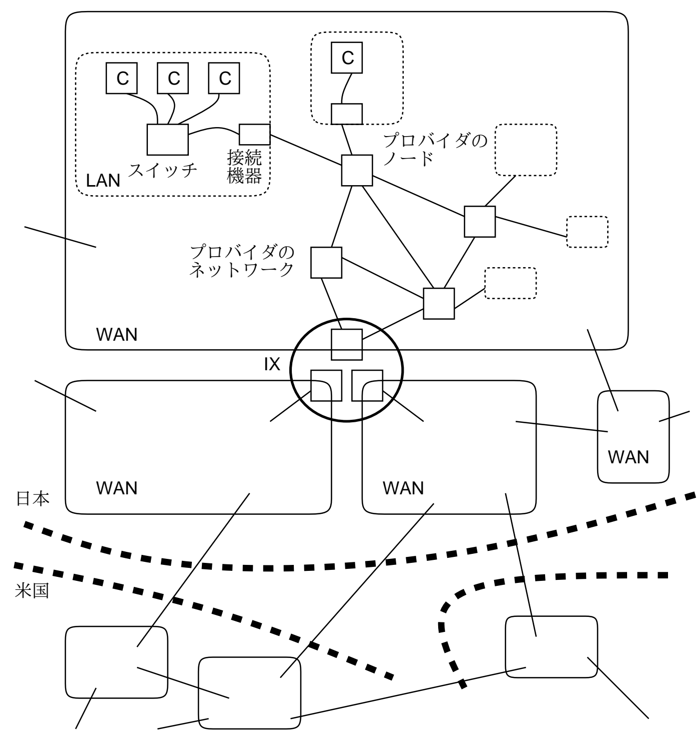


図 2.8: インターネットの構造

LAN のような 1 つのネットワークと別のネットワーク (や外部の回線) とがつながるところでは、ハブやスイッチではなく、次節で述べる経路制御を行う中継装置が必要です。

TCP/IP ではこのような装置のことを**ルータ** (router) と呼びます。学校や家庭などの LAN が直接接続する相手先は **ISP**(Internet Service Provider) ないし「プロバイダ」と呼ばれる事業者のネットワークです。ISP は契約者から料金を取ってインターネットへの接続を提供するのが商売であり、そ

[8] `255.255.255.0` ではネットワーク部 24 ビット、(全体が 32 ビットなので残りの) ホスト部 8 がビットとなります。ネットワーク部が 29 ビットであれば、ネットマスクは `255.255.255.248` になります (2 進表記に直してみてください)。

[9] このほか、ある LAN の中だけでパケットをやりとりする場合には自由につかっていい「プライベートアドレス」も用意されています。プライベートアドレスを含んだパケットはその LAN から外に出さないようにしなければなりません。

のために大規模な WAN を自社の設備によって (または他社の設備を論理的に借りるなどして) 構築しています。

しかしこれだけでは、私がプロバイダ A と契約していて、知合いがプロバイダ B と契約しているときに、相互に通信できないことになります。そこで、プロバイダはそれぞれの WAN を相互に接続しており、そこを通じてパケットを相互に流通させます。[10]

A と B が直接繋がっていなくても、間接的につながってさえいれば互いに通信ができるのです。このようにして、多数のネットワークが相互につながってできた集合体がインターネットです。そういう意味で、インターネットは「草の根」的な構造をしており、どこかに「本部」があるわけではないのです。

2.3.4 IP と経路制御

先に説明したように、TCP/IP の中では IP というプロトコルがインターネットの中核となっています。IP の最大の「魔法」は、行き先を指定するだけで、それが世界中どこであっても、パケットをその行き先に届けてくれることです。この機能を**経路制御** (routing) と呼びます。

IP の経路制御がなぜそんなに偉いのでしょう? 郵便物の場合を考えて見ましょう。ある郵便局に集まってきた葉書に「東京都調布市調布ケ丘 1-5-1」という宛先が書いてあったとすると、そこが東京都以外の郵便局であれば、東京に「調布市」という市があるかどうか知らなかったとしても、とにかく東京中央郵便に送れば済みます。東京中央郵便局では、東京都のどの区や市はどの局の受け持ちか知っていますから、「調布ケ丘」という地名を知っていてもいなくてもとにかく調布郵便局に送れば済みます。これが可能なのは、住所が「都道府県→区市町村→地名→番地」という**階層構造**になっているからです。アドレスが階層構造になっていれば、各中継地点では「自分の受け持ち範囲でないアドレスはとにかく上位の中継地点に送る」「受け持ち範囲のアドレスはより小さい受け持ち範囲の中継地点か個々の宛先に送る」という方法で経路制御が行えます。

しかし IP ではアドレスは 32 ビットの数値であり、上で述べたような階層構造にはなっていません。前述のように、32 ビット中で上位何ビットかがネットワークアドレス、残りがホストアドレスであり、LAN 内のホストはネットワークアドレスが共通ということは決っていますが、ネットワークアドレスのつけ方は任意です。[11]

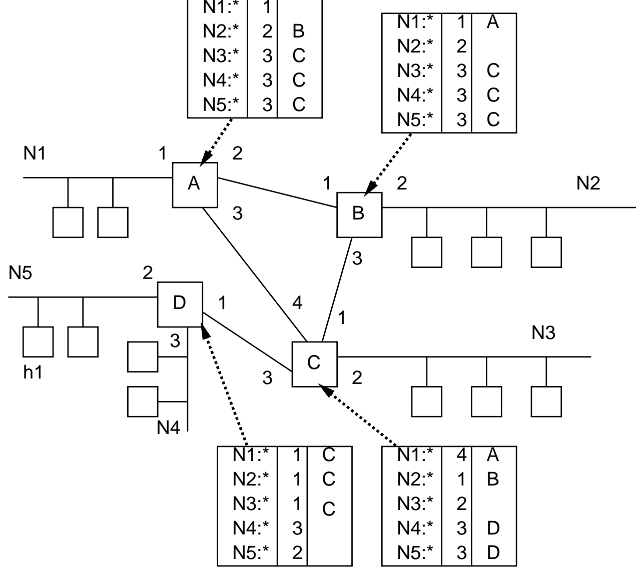


図 2.9: 経路表

[10] これを効率よく行うため、多数の ISP が相互に接続を取るための地点を設けることもあります。これを IX(Internet eXchange) と呼びます。

[11] 厳密にはそれだと経路表が大きくなって大変なので、できるだけ隣接したネットワークには隣接した番号をつけ、外部からはあたかも 1 つのネットワークに見えるようにします。これを経路の集約 (aggregation) と呼びます。

つまり「深大寺元町」「布田」などの名前だけが与えられ、それを見て送り先を決める必要があります (布田 N 丁目、というのが近くにまとまっていることは保証されますが)。つまり、すべての主要な郵便局には全国のあらゆる地名の一覧表があり、その一覧表に「この地名はこちらの方に送る」と書かれていて、それに従って中継するわけです。このデータのことを**経路表** (routing table) と呼びます。

たとえば図 2.9 のように 4 つのルータでネットワーク N1〜N5 がつながっているとします。それぞれのルータは、自分が直接つながっているネットワークについてはそこにつながるインタフェースにパケットを送ればいいと分かりますが、それ以外については経路表を参照してパケットの転送先を決めます。たとえば、ルータ A に「N5:h1」あてのパケットが来たとすると、表を参照して「インタフェース 3 番を通じて C に送ればいい」と分かります。C にそのパケットが来ると、C も同様にして「インタフェース 3 番を通じて D に送る」と分かり、D に到達するとそこから直接 N5 に送るわけです。このような経路表は、インターネット上で経路情報を交換する複数のプロトコルを用いて、常に最新の状態に保たれています。

先の ping では RTT しか分かりませんでしたが、**traceroute** [12] というコマンドを使うと相手先までの経路を (できる範囲で) 調べることができます。ただし、このコマンドはパケットを多数出すので ping よりも厳しく規制されていて、使えないことが多いです。以下ではとあるサイトからブラジル行政府あてを調べてみたものです。l

```
...$ traceroute brasil.gov.br
traceroute to brasil.gov.br (170.246.252.243), 64 hops max, 40 byte packets
 1  plalagw (210.154.96.161)  1.139 ms  0.652 ms  0.626 ms
 2  i114-191-248-67.s99.a049.ap.plala.or.jp (114.191.248.67)  3.267 ms  2.775 ms  2.341 ms
 3  114.191.248.89 (114.191.248.89)  3.584 ms  3.542 ms  2.416 ms
 4  118.21.174.65 (118.21.174.65)  4.068 ms  3.385 ms  3.192 ms
 5  i118-21-197-33.s99.a049.ap.plala.or.jp (118.21.197.33)  4.486 ms  3.646 ms  4.270 ms
 6  211.6.91.169 (211.6.91.169)  5.310 ms  4.597 ms
    211.6.91.173 (211.6.91.173)  4.674 ms
 7  122.1.245.65 (122.1.245.65)  4.825 ms  4.622 ms  4.342 ms
 8  ae-6.r03.tokyjp05.jp.bb.gin.ntt.net (120.88.53.29)  5.987 ms  4.778 ms
    ae-6.r02.tokyjp05.jp.bb.gin.ntt.net (120.88.53.21)  5.763 ms
 9  * * *
10  * ae-1-3501.ear3.SanJose1.Level3.net (4.69.203.121)  164.538 ms *
11  * * *
12  ae3-20G.csr1.BSA1.BSB.gblx.net (209.130.131.158)  361.155 ms  355.460 ms *
13  46.25.125.189.static.impsat.net.br (189.125.25.46)  344.007 ms
    ae3-20G.csr1.BSA1.BSB.gblx.net (209.130.131.158)  353.085 ms
    46.25.125.189.static.impsat.net.br (189.125.25.46)  342.928 ms
14  161.148.25.165 (161.148.25.165)  350.539 ms  349.462 ms  347.646 ms
15  189.9.201.22 (189.9.201.22)  334.180 ms
    161.148.25.165 (161.148.25.165)  351.838 ms
    189.9.201.22 (189.9.201.22)  333.211 ms
16  189.9.201.22 (189.9.201.22)  332.341 ms *  333.437 ms
17  * * *
18  * * *
19  ^C
... $
```

このコマンドは指定した宛先までの通信が通っている中継点とそこまでの往復時間を順次リストアップします (場所によっては中継点が複数マシンから構成されていてタイミングでどのマシンかが代わるものがあり、このような時は中継点が複数表示されます。また、セキュリティのため情報を返さない中継点もあり、その場合は「* * *」のような表示になります)。18 ホップ目から先は「* * *」(応答なし) ばかりなので`^C`で止めています。

これを見ると、おおよそ次のようなことが分かります。

[12] Windows ではコマンド名は `tracert` になっています)。

- まずこのサイトが使っているプロバイダ plala のネットに入り、そこの内部で数段の中継があり、NTT のネットに行く。数 ms 程度のパケット往復時間なので国内。
- 次に 2 つほど中継があって米国サンノゼ (西海岸) に到達する。米国西海岸だとパケット往復時間は 160ms 程度。
- そのあとまた中継があってブラジルに入る模様。ブラジルまで行くとパケット往復時間は 300ms を超える。
- そこでいくつか中継された先は応答がなくて分からないない。

演習 4 演習 2 でパケット到達時間を調べたホストまでの経路を `traceroute` で確認してみなさい。それができたら、次の事項から 1 つ以上やってみなさい。

注意! この演習は個人 PC やスマホからできることもありますが、できないことも多くあります。その場合は無理せずに別の課題を選んでください。なお、Windows では `traceroute` コマンドは `tracert` という名前になっています。

a. 日本国内のさまざまな箇所までの経路を調べ、自分のいるところから国内がどのようにつながっているかを整理してまとめなさい。
b. 世界の複数の地域までの経路を調べ、日本から世界各地にどのようにつながっているかを整理してまとめなさい。
c. 個人 PC とスマホなど複数の出発点から各地までの経路を比較し、どこまでが違ってどこからは共通かなどを整理しなさい。

2.3.5 ドメイン名と DNS

ここまで見て来たように、IP での通信はあくまでも IP アドレスを用いて行われますが、人間が見て理解するにはもっと「普通の」(意味のある) 名前を使うことが望まれます。このため、インターネット上では**ドメイン名** (domain name) と呼ばれる文字列の名前もサポートし、ドメイン名から IP アドレスを検索するためのシステムとして **DNS**(Domain Name System) と呼ばれるサービスが運用されています。ドメイン名は複数の名前を「`.`」でつなげたもので、右側の名前がより広い範囲に対応する階層構造になります (日本での住所表記とは反対ですが、英語では住所、街、州、国の順で書くのでそれと同様です)。たとえばドメイン名「`sol.cc.uec.ac.jp`」は次のような階層に対応しています。

```
sol.cc.uec.ac.jp
             ↑日本
          ↑教育研究組織
      ↑電気通信大学
   ↑計算センター
↑ホスト名
```

DNS は内部的にもこの階層構造を利用して「最右側の名前 (**TLD**、Top-Level Domain) の管理元をすべて知っているサーバ (**ルートサーバ**)」、「各 TLD の管理をするサーバ (その子供のドメインの管理元をすべて知っている)」、「第 3 レベルのサーバ」等々のように分担してドメイン名と IP アドレスの対応を把握し、検索に応答できる状態を維持しています。

各 TLD は、その TLD を管轄する組織や事業者が持っていてそのサブドメイン以下を管理運営しています。これを**レジストリ** (registry) と呼びます。自分が特定の TLD の下にサブドメインを持ちたい場合は、レジストリに個別に頼むのでは自分も相手も大変なので、**レジストラ** (registerer) と呼ばれる管理・申請代行業者に頼むのが普通です。[13]

[13] レジストラの下請け・孫受けな業者も多数あります。

さまざまなドメイン名を使う権利は、レジストラから購入できます。実際にそのドメイン名が使えるようになるためには、そのドメイン名による検索に応答するサービスが必要ですが、それもレジストラにお金を払って頼むことができます (技術力があれば自分で DNS サービスを動かしてもよい)。

DNS に対する検索は通常、ユーザが (メールや Web などのアドレスの形で) ドメイン名を指定した時に自動的に行われますが、コマンド **nslookup** を使って次のようにユーザが直接検索することもできます。

```
...$ nslookup www.yahoo.co.jp
Server:  130.153.26.5
Address: 130.153.26.5#53
Non-authoritative answer:
www.yahoo.co.jp   canonical name = edge12.g.yimg.jp.
Name:    edge12.g.yimg.jp
Address: 183.79.219.252
...$
```

演習 5 自分が使いたいと思うサブドメイン名 (たとえば `gogkigen.de` みたいなもの、自分の名前がらみを勧めます) をいくつか選び、取得できるか、その場合いくらかを調べてまとめてみよ (「ドメイン名 取得」などで検索して調べる)。それを実際に使用する場合、取得費用以外に何にお金が必要かも調べること。

課題 2A

今回の課題は「演習 2〜5」に含まれる (小) 課題 (合計で 10 個) の中から 1 つ以上を選択し、結果をレポートとして報告して頂くことです。LMS の「assignment ＃ 2」の入力欄に入力してください (直接打ち込むとログイン時間切れになった時に内容が消失するので、メモ帳など他のソフトで作成して完成後にコピーすることを勧めます)。以下の内容がこの順に含まれるようにしてください。

- 冒頭に題名「コンピュータリテラシレポート＃ 2」、学籍番号、氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も別途書く。)
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 以下のアンケートに対する回答。

  Q1. コンピュータネットワークの仕組みについてどれくらい知っていましたか。新たに知ったことで面白かったことは何ですか。

  Q2. Unix システムを使って操作しながら調べたりするのは慣れましたか。

  Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

なお、グループ課題以外の課題もグループでやって構いません。いずれの場合も、(メンバー全員の氏名を明記した上で) レポートは必ず各自で執筆してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。

＃3　ネットワークと安全性

今回の目標は次の通りです。

- ネットワークにおける安全性の概念やそのための技術を理解する— 様々な活動の前にまず安全であることが必須の前提です。
- WWW、電子メールというネットワーク上の基本的な通信サービスを理解する — どちらも必ずお世話になりますが、その原理まで知っておくことが安全性や使いこなしの鍵です。
- ネットワークコミュニケーションの特性やその注意点を理解する— これらを意識しておかないと社会的トラブルになります。

3.1　情報セキュリティ

3.1.1　セキュリティと情報セキュリティ ex

セキュリティ(security, 安全性) とはその名前通り安全が保たれること、たとえば身体的危険や窃盗等の危険に逢わないことです。一般にはそのため、家屋に施錠したり、ガードマンを頼んだり、銀行に預けたりしますね。では情報セキュリティ(情報に関する安全性) はどうでしょう。たとえばあなたは、無線 LAN 経由で情報をやりとりする際、その内容が周囲の人に傍受され放題だと知っていましたか? [1] 多くの人は意識せずに無線 LAN 経由で重要な情報を送ったりしていますが、実際には情報セキュリティについて意識しておかないと、思わぬトラブルに遭遇する可能性があるのです。

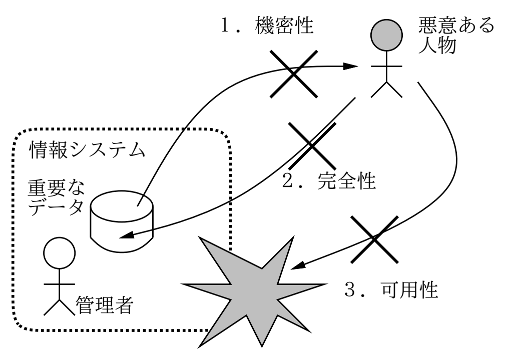


図 3.1: 情報セキュリティの 3 要素

一般に、情報セキュリティに関する目標としては、次の 3 つが挙げられます (図 3.1)。

- **機密性** (confidentiality) — 情報が、取得できるべきでない対象に漏洩しないこと。
- **完全性** (integrity) — 情報が、完全な状態である、つまり改ざんされないこと。
- **可用性** (availability) — 情報が、必要なとき利用可能、つまりシステムがきちんと動作すること。

これらから、情報セキュリティの多くは人的問題、管理の問題だと分かります。たとえば機密性は、ある情報が「誰は」見てよく「誰は」だめか明確でなければ適正に保てません。完全性については、ある情報を権限を持つ人が業務の必要に応じて書き換えることは必要ですが、権限がない人が書き

[1] 暗号化通信を使っていれば別です。なお、暗号といっても無線 LAN の暗号化機能は除外して考えてください。同じ無線 LAN ステーションを使っている他人も、あなたと同じキーを使うので、それらの人には解読し放題ですから。

換えたり、権限がある人でも業務上の必要以外で書き換えるのは問題です。実際、企業等におけるセキュリティ侵害の大多数は外部の人ではなく内部の人によるという調査結果があります。

可用性については、いくらネットワーク経由の攻撃を防御しても、サーバ室に誰かが入ってきて電源を切ったり装置を破壊できたら無意味です。地震・火事など自然災害への対策も重要です。可用性に対する対策としては、システムを多重化してどちらかが壊れても運用を続行可能とする、重要なデータは定期的にバックアップして安全な場所に保管する、などがあります (そしてバックアップが流出すると機密性が損なわれます)。

今日、多くの情報システムや個人にとって情報セキュリティ侵害の大きな要因となっているものに、不正アクセスとマルウェアがあります。なお、これらの侵害においては、当事者の認識不足がきっかけである場合が少なからずあるので、そのようなきっかけにならないよう心してください。

不正アクセスとは、(1) 他人の ID/パスワードを使ったり、(2) ソフトの欠陥を衝くなどして、本来のアクセス制限を回避してシステムを利用することを言います。これに対する対策は、(1) についてはパスワードを適正に管理すること [2]、(2) についてはシステムソフトウェアのアップデートを適正に行うことが挙げられます。[3]

マルウェア (malware) とは、システム所有者の意図しないことを行うような有害ソフトウェア全般を指す言葉であり、**ウィルス** (プログラムや文書の中に自分自身を埋め込み、、ユーザがそのファイルを開くことで感染し広まるもの)、**ワーム** (ネットワーク経由で自分のコピーを他のホストに送り込むもの)、**トロイの木馬** (有用に見えるソフトウェアで、ユーザが騙されてそれを使うことで広まるもの)、などの種類があります。マルウェア対策としては、ウィルス対策ソフトウェア (既知のマルウェアを監視して排除/警告する) の使用と、他人から送られて来たりネットワーク上で入手したソフトウェアを安易に実行しないことが重要です。

3.1.2　暗号技術 ex

暗号 (cryptography) は昔から情報の機密性・完全性の維持に使われて来ました。その基本は秘密の通信に際し、もとのテキスト (平文) を一定の規則によりランダムに見えるような文字列 (暗号文) に変換 (**暗号化**) して送り、受け手はそれを逆に変換 (**復号**) して読む、というものです。通信途中の暗号文を誰かが盗み見ても暗号が解読できなければ秘密を知ることができませんし (機密性の保持)、暗号化の規則が分かっていなければ偽物のメッセージでだますこともできません (完全性の保持)。

ところでこの暗号化や復号は昔は人間がやっていて大変でしたが、今では当然コンピュータでやります。だったら、暗号化や復号をおこなうプログラムを秘密にしておくのでしょうか (悪い人に復号プログラムが渡ると秘密が解読されますし、暗号化プログラムが渡ると偽物の秘密通信が来るかも知れません)。実際にはプログラムを秘密にするというのは難しいので、プログラムは誰もが共通に使うことにして、そのプログラムに**鍵** (key) と呼ばれる情報 (実際には 128 ビットとか 512 ビットとかの長さの 0/1 の列です) を与えて暗号化/復号をおこないます。この鍵を秘密にしておくことで、悪い人に解読されないようにするわけです。

上記が暗号の基本でしたが、その方式には大きく分けて、次の 2 種類があります。

- **対称鍵暗号** (symmetric key cryptgraphy) ないし**共通鍵暗号**— 暗号化と復号に同じ鍵を用いる。
- **公開鍵暗号** (public key cryptgraphy) — 2 つの鍵の対を使い、片方で暗号化、他方で復号を行う。

歴史的には対称鍵暗号の方が古くからありますが、これには「鍵をどうやって安全に伝達するか」という問題があります。ネットワークで送る? 送っているのを盗聴されたら後の暗号化通信も解読されてしまいます。USB メモリなどに入れて手渡たせば安全ですが、面倒です。

そこで考案されたのが公開鍵暗号です。この方式では、鍵を秘密鍵 (secret key) と公開鍵 (public key) の対で生成し、公開鍵は皆に知らせます。秘密の通信をしたい人は、公開鍵を使って通信を暗号

[2]推測されやすいパスワードを避ける、複数サイトに同じパスワードを使わない、紙に書いて放置しないなど

[3]システムには多くの欠陥が含まれていて絶えず新しいものが発見されているので、その対策を施したものに入れ替えておかないと、その発見された欠陥を利用して不正アクセスを受ける危険があります。

化して送り、受け手は自分だけが持つ秘密鍵で解読します。他人は秘密鍵を知らないので、暗号を解読できません (図 3.2)。

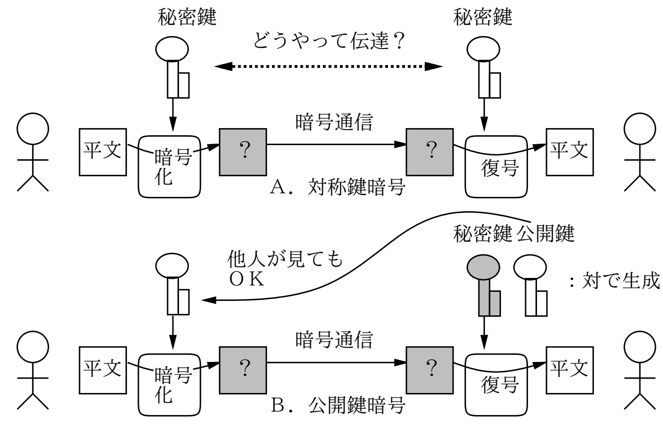


図 3.2: 対称鍵暗号と公開鍵暗号

また、公開鍵暗号の別の用途として、持ち主が持つ秘密鍵によって情報に印をつけ、対応する公開鍵を用いてそれが OK であることを検証する (偽物の秘密鍵だと NG となる) ことにより、情報を発信したのが確かに秘密鍵を持つ本人だということを証明する、というものもあります。こちらは書類にサインするのと似ていることから、**電子署名** (digital signature) と言います。

どんな暗号であっても、十分な時間を掛ければ…すべての鍵の可能性を「総当たり」で試せば、破れます。ただしそれには、すごく長い時間が掛かります。512 ビットの鍵であれば 2^{512} 通りの場合があり、これはすごく大きな数なので、全部試すよりも前に太陽が燃え尽きてしまうくらい時間が掛かるから安全だ、と考えるわけです。もちろん、鍵のビット数が増えるほど、試す場合の数は多くなりますから、より安全になります (その代わり処理は遅くなります)。

3.1.3 PKI と証明書の連鎖 ex

先に公開鍵を皆に知らせると簡単に言いましたが、途中で誰かがその公開鍵を別のものにすりかえたりすると、秘密のつもりの通信がそのすりかえた人に解読されてしまいます。ですから、公開鍵が確かに本ものかを確認することが必要ですが、それを通信する相手ごとにやるのは大変そうです。

そこで **PKI**(Public Key Infrastructure) という枠組みが用意されました。これは、**CA**(Certification Authority) という組織が他の組織や個人の公開鍵を証明書により「正しいと保証」するという制度で、保証してもらいたい側は CA に自分の出自を示す書類を送るなどして確認してもらいます (もちろん、CA も商売なのでお金が掛かります)。

では、CA 自体の正しさは…それは、その CA を保証する「親 CA」があり、そのまた親…とつながって行き、最後は「根元」の CA(ルート CA) にたどりつきます。ルート CA はそう沢山はないので、あらかじめ確認しておくなどの手段が取れます。なお、CA の仕事は「当該組織が存在していること」を確認し保証することまでであり、その組織の業務内容や社会的信頼性を調査して保証するわけではない点は注意しておいてください。

WWW では普通の (暗号化しない) サイトが「`http://`」で始まるアドレスなのに対し、ネットショップや Web メールなどログインを要するサイトは「`https://`」を使用します。これが暗号化されたデータ転送を用いる **TLS** [4] プロトコルを使うサイトで、通信に公開鍵暗号を使用し、PKI を用いたチェックをおこないます。そのため、代表的なルート CA について、ブラウザを配布する時に最初からその証明書情報がブラウザ内に組み込まれています。ブラウザで `https://`のページを開いた時は、このページの証明書情報からルート CA までの連鎖をたどれるかどうかをブラウザが自動的にチェックしています (図 3.3)。大切なのは、単に「暗号化が使われている」(通信内容が傍受されない) ことだけ

[4] 以前は **SSL** プロトコルが使われていましたが、安全でないことが分かり、現在は TLS に移行しています。

が重要なのではなく、「通信相手が本当に意図した相手なのか」(偽サイトにつながされていないか) も同じくらい重要だということです。

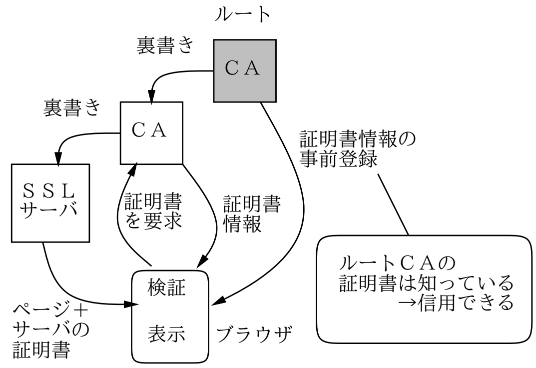


図 3.3: PKI と CA の連鎖

しかし、お金の節約その他 (?) の理由で、この連鎖に加わらずに `https:`のページを動かすサイトもあります。このようなサイトの証明書を (自分で勝手に正しいと言っているニュアンスで)「オレオレ証明書」と呼びます。その種のサイトを表示させると、ブラウザは「ルート CA への連鎖がたどれない」という警告を出し、次の選択肢を提示します (原則として (2) を選びましょう)。

(1) このサイトを恒久的に信用する。

(2) 表示をとりやめる。

最後に、普段のシステムへのログインなどパスワードに基づく認証も暗号技術が基本になっていることを注意しておきます (指紋や静脈パターンなどの生体認証技術は別です)。

演習 1 ブラウザで適当な `https:` サイトを開き、鍵マーククリックなどの方法で「証明書」の情報を表示させなさい (Firefox では「鍵マーク」→「>」→「more info(詳細を表示)」→ (別窓) →「View Certificate(証明書を表示)」で見られます)。表示できたら、以下のテーマから 1 つ以上やってみなさい。

a. 自分が開いている `https:`サイトからルート CA までの CA の連鎖を確認してみなさい (Firefox では「Details」タブに切替えると見られます)。できれば、複数のさまざまな国の `https:`サイトについて、連鎖がどうなっているかを整理してみられるとよいです。

b. ブラウザの「証明書管理」画面を開き、そのブラウザにどのようなルート CA の証明書が標準で入れられているか見てみなさい (Firefox では「メニュー」→「Preferences(設定)」→「Advanced(詳細)」→「Certificates(証明書)」→「View Certificaties(証明書を表示)」で見られます)。できればその中のいくつかについて、ネットで検索してどのような企業か調べてみるとなおよいでしょう。

c. 例えば `https://kotonet.com/mail/inquiry1.html` などのオレオレ証明書サイトをブラウザで開き、警告の様子を調べなさい。

3.2 ネットワークサービスとその構成 ex

私達がネットワークを使うのは、メッセージを送るなど、何か「役に立つ機能」を使うためです。ネットワーク上でこのような機能を提供するもののことを**ネットワークサービス**と呼びます。

ネットワークサービスはどんな形で提供されているのでしょう? 通信を行うためには、互いに通信する 2 つのプログラムが「あちら」(リモート側/ネットワークの向こう側) と「こちら」(ローカル側/自分の手もと) で動く必要があります。「こちら」は自分のマシンだから自由になりますが、「あちら」はどうしましょうか? この問題に対する 1 つの解は、プログラムを次の 2 種類に分けることです (サーバが「あちら」、クライアントが「こちら」になります)。

- サーバ — サービスを提供するマシンで常時稼働していて、サービス要求があるまで待ち、要求があったらサービスを提供する。
- クライアント — サーバに要求を出して、そのサービスを受ける。

これを**クライアントサーバ方式**と呼びます (図 3.4)。この方式では、上述の通信の問題が自然な形で解決できますし、サービス提供のために必要な資源 (データ等) はサーバのところで一括管理できるので、各種サービスの実現が比較的簡単に行えます。このため、クライアントサーバ方式はネットワークの初期から今日に至るまでネットワークサービスの構成方式として広く使われています。

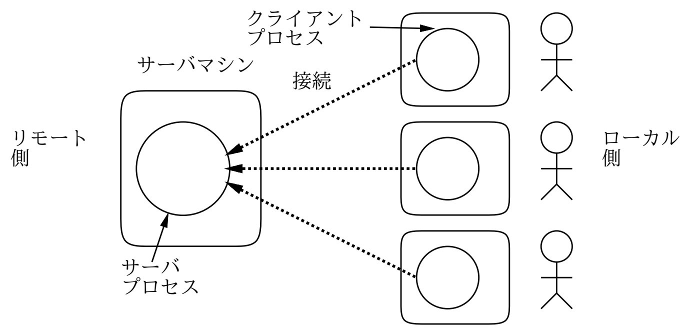


図 3.4: クライアントサーバ方式

クライアントはサービスを利用するユーザの数だけ動きます。ユーザはサービスを使うときにクライアントを起動し、そのプログラムがサーバと接続しサービスを受けます。つまりクライアトサーバ方式では、クライアントが各ユーザのマシンで動き、サーバはサービスを提供するための専用マシンで動く形になります。

1 つのマシンで複数のサービスを提供することも普通にあるので、クライアントは特定のサービスを利用するのにサービスを提供するマシンの IP アドレスに加えて「ポート番号」と呼ばれる番号を指定します。WWW やメールなどの標準的なサービスは、標準化されたポート番号を使うようになっていますが、サービスやクライアントによっては初期設定でこれらの情報を個別に設定する必要がある場合もあります。

ところで、クライアントサーバ方式ではない構成のネットワークサービスもいくつかあります。これらは**ピアツーピア方式** (P2P) と呼ばれ、特定のサーバはなく、各プログラムが対等な立場で通信します。このようなシステムで大規模なものの代表例としては、Napster、Winny、Skype などが挙げられます。これらはいずれも草の根的な通信やファイル転送を目的としているため、どこか 1 箇所に皆がアクセスするよりも、互いにやり取りをしたい組をうまく設定できさえすれば、それぞれが直接やり取りすることで負荷の集中を避けられるのです。[5]

3.3 遠隔ログインと SSH

これまでやってきたように、Unix では端末にコマンド (=文字) を打ち込んでシステムを利用し、コマンドの結果も文字で表示されます。ということは、「自分が座っている」システムを使うかわりに「遠隔地にある」システムを使うことも、文字さえやりとりできれば同様におこなえることになります。

これを可能にするサービスは**遠隔ログイン** (remote login) と呼ばれ、ネットの初期から存在しています。なぜかというと、離れた場所にあるマシンの機能を使いたいと思ったときに、そこまで歩いて行かなくても「座ったままで」使えて便利だからです。

遠隔ログインは今日では **ssh** (Secure SHell) と呼ばれるサービスがおもに使われます。図 3.5 にその概要を示しています。普段は私たちはマシンの前に座ってそのマシンであれこれコマンドを動かしていますが (左: コマンド群を小さな○で表しています)、よそのマシンでコマンドを動かしたい (右) 場合は、たとえば次のようにして `ssh` コマンドを使用します。

[5]中央サーバが無いので不法コピーなどが発見されにくいという点もあります。Winny は暗号化と組み合わせてこの部分を売りものにしたソフトですが、その秘密主義がさまざまな不幸をもたらしました。

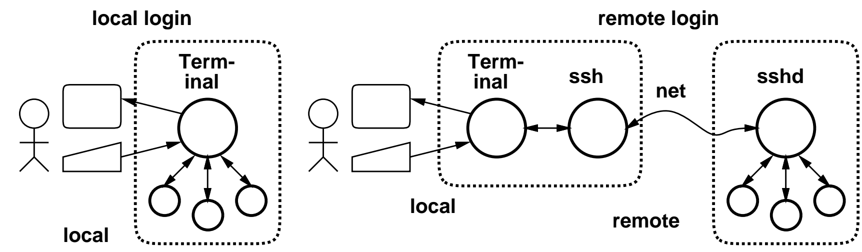


図 3.5: ssh による遠隔ログイン

```
...$ ssh ka002689@sol.edu.cc.uec.ac.jp  ←「ssh ユーザ名@ホスト名」
ka002689@sol.edu.cc.uec.ac.jp's password:    ←パスワードを応答
... メッセージ ...
sol$ hostname      ← sol のプロンプトが出る。hostname コマンド
sol.cc.uec.ac.jp   ←ホスト名は sol
sol$ uptime        ← uptime コマンド (実行状況の表示)
 17;59:00 up 106 days, 42 users  ← 106 日間稼働。ユーザ 42 名
sol$ exit          ←終わる
logout
Connection to sol.edu.cc.uec.ac.jp closed.
...$  ←元のマシンにもどった
```

sol は共用サーバであり、処理能力が大きく、学外から接続して使えます。自分のファイルはどちらでも同じに使えるように管理されていますから、演習室のマシンと同じように使えます。学外からセンターの Unix を使いたい場合は `ssh` ないしこれと同等の機能を持ったソフトを使って sol に接続すれば使うことができます。その役割は「キーボードから入力したものを相手先のホストに送って実行させる」ことと、「相手先のホストで出力された文字を送り返してもらって手元の端末に表示させる」ことです。

なお、`ssh` の「secure」はどこから来ているかというと、手元と相手先で文字を送受する際、すべての情報を暗号化して傍受されないようにするなどの点によります。このほか、ログイン時の認証を単なるパスワードではなく、より厳密な公開鍵認証にするなど、さまざまな安全のための機能が用意されています。

演習 0 「`ssh ユーザ名@sol.edu.cc.uec.ac.jp`」を実行し、パスワードを応答して、sol に遠隔ログインしてみなさい。`date`、`sleep 5`、`hostname`、`uptime` などのコマンドを実行してみて、手元のマシンとどこが違うか観察してみなさい (注意: 手元のマシンでも実行してみないと比較できない)。終わったら `exit` で抜けること。

3.4 World Wide Web

3.4.1 WWW とリンク ex

今日、最も広く利用されているネットワークサービスは **WWW**(World Wide Web ないし Web) です。皆様もそのクライアントである **Web ブラウザ**を使わない日はないと思います。しかし、その見慣れた画面の裏側の仕組みを考えたことのある人は、多くないかも知れません。

Web の基本的な枠組みは**ハイパーテキスト**です。ハイパーテキストとは次のようなものです。

- 計算機の画面上にテキストや画像などの内容 (コンテンツ) が表示されている。
- コンテンツの中には、他のコンテンツやその特定箇所を「指し示して」いる箇所が埋め込まれている。これを**リンク**という。

- リンクの箇所を何らかの方法で選択すると、画面はそのリンク先の内容に切り替わる。
- このようなリンクで互いに結び合わされたコンテンツの集まりは、リンクを自由にたどりながら読み進んでいくことができる。

ハイパーテキストの概念そのものは WWW よりずっと前から存在しましたが、それをネットワークサービスの形に組み立てたのが WWW なのです。

WWW ではネットワーク上にある様々なモノ (資源ないしコンテンツ) を指す手段として **URL**(Uniform Resource Locator) と呼ばれる形式を使用しています。URL は一般に次の形をとります。

スキーム:アドレス

スキーム中で最も一般的なのは、Web サーバからコンテンツを取り寄せるプロトコル **HTTP**(HyperText Transfer Protocol) を使う場合で、次の形を取ります。[6]

`http://ホスト指定/ディレクトリ/.../ディレクトリ/ファイル`

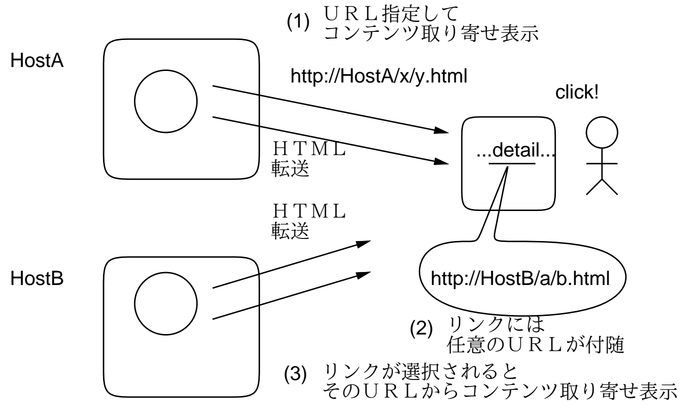


図 3.6: リンクの仕組み

なぜここで URL について長々と説明しているのかというと、URL こそが WWW の「肝」だからです。WWW でコンテンツを伝送しているのはごく簡単なプロトコルであり、ブラウザがやっていることは基本的には次の 2 点だけです。

a. 指定された URL からコンテンツを取り寄せる。

b. 取り寄せたコンテンツを適切な形で画面に表示する。

コンテンツを取り寄せたとき、それが **HTML**(HyperText Markup Language) 形式であれば、中にリンクが含まれていることがあります (そこに別コンテンツを指す URL 情報が含まれています)。そしてユーザがリンクを選択すると、ブラウザはリンク先のコンテンツを取り寄せて表示し (上記 a+b の動作)、結果的に「別のページへ飛ぶ」わけです (図 3.6)。これだけで済むのは、「リンク先」が URL の形で表され、世界中のどのサーバのどのコンテンツでも自由に指し示せるから、なのです。

3.4.2 Web アプリケーション ex

前の節では WWW をサーバからコンテンツを取って来るだけのシステムとして説明しましたが、実際にはそうではありませんね。私たちは Web ブラウザを使ってネットショップの買物をしたり掲示板などで互いにコミュニケーションしたりします。それはどのような仕組みによるのでしょうか。

実は HTML によって記述されるページの中にはボタン、入力欄などの GUI 部品を含めることができます。そして、ブラウザ上でユーザが入力欄などにテキストを記入したり選択メニューで選択肢を設定したあと、「送信」ボタンを押すことでこれらの情報を Web サーバに送ることができるのです。

[6]既に説明したように、サーバとブラウザの間で暗号化通信を行うために TLS/SSL を使うこともでき、その場合はスキームとして `http:` の代わりに `https:` を使います。

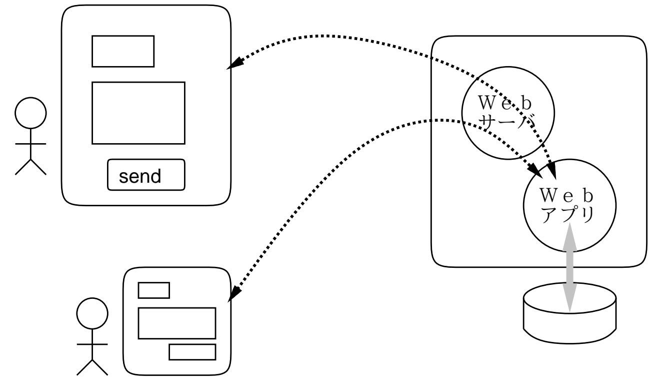


図 3.7: Web アプリケーション

この送信されたデータは、Web サーバと連携して動いているプログラムによって受け取られ、このプログラムがさまざまな処理を行います。そしてその結果は再び HTML としてブラウザに送られ、ブラウザの画面で処理結果を見ることができます。これがネットショップや掲示板などのサイトを使っている時に起こっていることなのです。[7]

このような、Web サーバと Web ブラウザの上で動くプログラムのことを **Web アプリケーション**と言います (図 3.7)。Web アプリケーションは PC 上にブラウザさえあればそれ以上特別なプログラムを入れなくても使えるので、WWW の普及とともに広く使われるようになりました。また、スマートフォンやタブレットなどでもブラウザが動くため、これらのモバイル機器からも使えるという利点も生まれました。このため、今日では非常に多くの Web アプリケーションが使われています。[8]

3.5 電子メール

3.5.1 電子メールサービスの構成 ex

電子メール (e-mail) はネットワークの初期からの情報交換サービスです。電子メールは遠隔地との情報交換を基本的に「サーバどうしで」行う設計になっていて、ユーザは「自分の手元の」サーバと通信してメッセージを投入したり取り出したりします (図 3.8)。ユーザがメッセージの読み書きを行う際に用いるクライアントソフトのことを**メールリーダ**ないし **MUA**(Mail User Agent) 呼びます。これと対比してメールサーバは **MTA**(Mail Transfer Agent) とも呼びます。

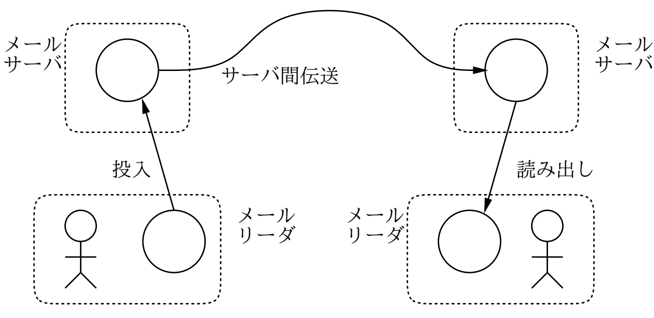


図 3.8: メールの伝送経路

電子メールでは、MUA からメッセージを送信すると、メッセージは手元の MTA がまず受け取り、そこから相手側の MTA に送られます (場合によっては複数の MTA が順次中継します)。これらのデー

[7]実際には処理結果が返されたときにページ全体が切り替わるのでなく、部分的に変化するようになっている場合もあります。この場合は最初の HTML の中に連携用のプログラムが含まれていて、このプログラムが処理結果を受け取って画面の一部を変更するなどの処理をします。

[8]スマートフォンなどでは画面が小さいため、ブラウザでは使いづらい場合もあります。そのため、とくに人気がありユーザ数の多いサービスでは、ブラウザ版に加えて Android や iOS 用に専用のアプリを用意しているものも増えています。

タ伝達には **SMTP**(Simple Mail Transfer Protocol) と呼ばれるプロトコルが使われます。

メールアドレスは一般に、「`someone@example.com`」のような形を取り、「`@`」の後ろ側はドメイン名 (この場合は `example.com`) になっています。そこで DNS で検索することで、宛先 MTA の IP アドレスが得られます。[9]IP アドレス分かれば、そのホストの MTA に接続して「`someone` さん宛ですよ」といってメッセージを送れます。

メールがメールサーバに到着した後、ユーザはサーバから自分宛のメッセージを取り出して読みます。その際ユーザがサーバとやりとりするのに、大きく分けて 2 通りの方法があります。

- ユーザが手元のマシンにメッセージを格納し管理する — MTA から MUA にメッセージを取り寄せる際に **POP**(Post Office Protocol) を使う。
- メッセージは常時サーバ内に格納しておき、ユーザは読みたいメッセージだけ取り寄せて眺める — MTA と手元の MUA の間での通信に **IMAP**(Internet Message Access Protocol) を使う。

POP ではメッセージが手元のマシンにあってすぐ探せ、サーバの通信が受信/送信時の短時間で済みます。一方、あるマシンで読んだメールは手元のマシンに移動してしまうので、後で別のマシンから読むことはできません。

IMAP はそのような不便がない代わりに、ネットワーク経由でサーバとつながっていないとメッセージを見ることができません。

3.5.2 IMAP クライアントの設定内容

では、自分の PC やスマホで IMAP を使えるメールクライアント (MUA) を設定するのに何が必要かを、Unix 上で定番の MUA である Thunderbird を例に見てみます。

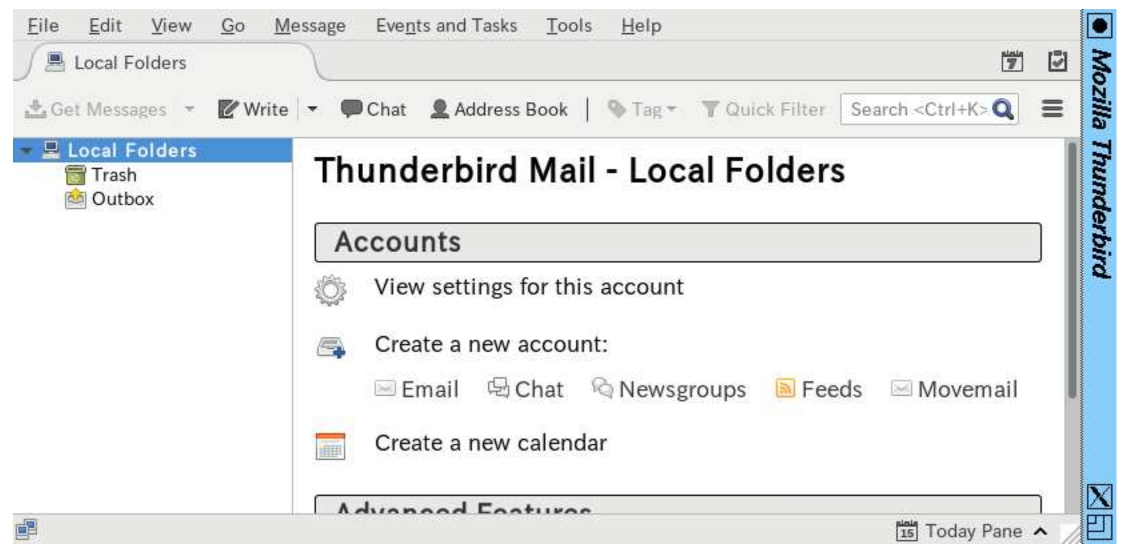


図 3.9: Thunderbird のアカウント管理画面

まず「`thunderbird &`」により起動すると、最初は何も設定がないので「アカウント管理」の画面になります (図 3.9)。この状態で「Create a new account: → email」のところを選択すると、「新しいメールアカウントが欲しいですか?」の画面になりますが、皆様は電通大アカウントを持っているので「Skip this and use my existing email」を選んで「名前」「メールアドレス」「パスワード」入力画面に進みます (図 3.10)。

ここで OK すると自動設定を試みますが、電通大の場合は自動では済まない設定になっているので (勉強にもなります)、自動設定に失敗して手動設定画面になります (図 3.11)。ここで、プロトコル関係を次のように設定ます。

[9]ただし、メール伝送の時は DNS 上で「MX(Mail Exchange)」レコードと呼ばれるメール専用種別のデータを検索します。これは、メールアドレスの場合、あるアドレス (つまりメールサーバ) が停止していたら代替のサーバに送るなど、メール固有の扱いをする場合があるためです。

図 3.10: Thunderbird のアカウント設定画面

- Incoming: のプロトコル IMAP、サーバホスト名 `imap.cc.uec.ac.jp`、ポート番号 993、プロトコル SSL/TLS、認証「normal passowrd」
- Outgoing: のプロトコル SMTP、サーバホスト名 `mx-delivery.uec.ac.jp`、ポート番号 587、プロトコル STARTTLS、認証「normal passowrd」

図 3.11: Thunderbird のアカウント設定画面 (手動設定)

これで「OK」すると、図 3.12 のようにサーバ名の下にフォルダ類が表示され、Inbox(受信箱) フォルダを選ぶと送られて来たメッセージ一覧が見え、さらに一覧から選択するとメッセージ内容が見られるようになります。送信は「返信」ボタンを使うか「作成」ボタンで画面を開いて宛先、件名、本文を打ち込んで送信します。

3.5.3　メールメッセージの形式 ex

個々のメールメッセージには、送信者 (From:)、件名 (Subject:)、日付 (Date:) などの情報があり、そして本文があります。通常 MUA でメッセージを見ているときはこれらは別々に見えていますが、実際にインターネット上でメッセージが送られているときは **RFC822** と呼ばれる形式でこれらの情報が一緒になって送られます (図 3.13)。

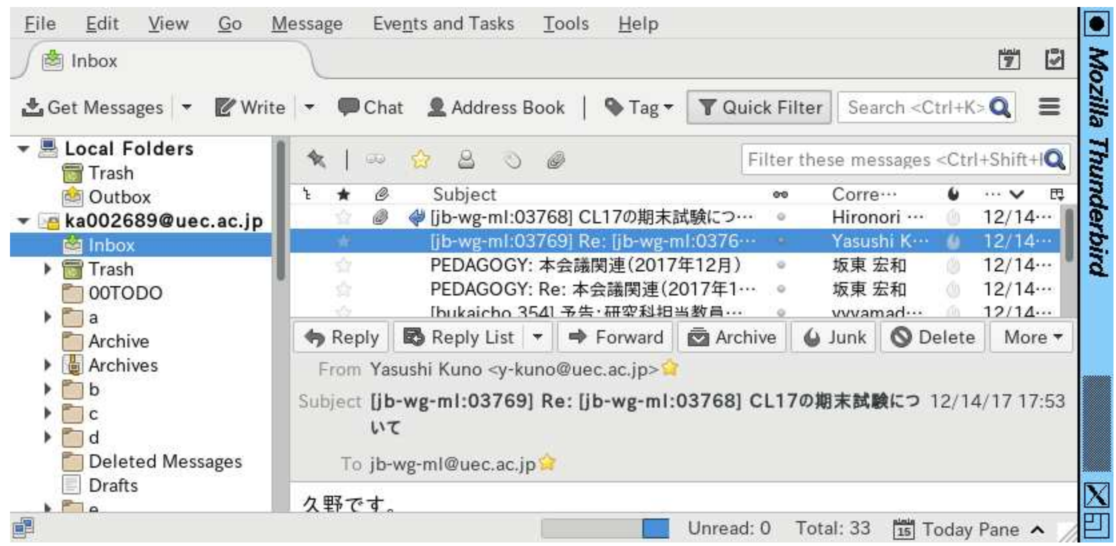


図 3.12: Thunderbird の通常画面 (受信箱表示)

RFC822 形式メッセージの冒頭には**メッセージヘッダ**と呼ばれる部分があり、ここに上述のもを含めた各種付加管理情報が格納されています。その後ろにメッセージ本文がありますが、ヘッダと本文の間は 1 行の空白行 (長さ 0 の行) で区切られています。

ヘッダの情報を見ると、メッセージがいつどこで投入され、どのように中継されてきたかが分かります。メールヘッダの一例を見てみましょう (図 3.13)。これはあくまでも例なので本文が 1 行しかありませんが、実際にはもちろん本文の方に肝心の用件が書かれています。

```
Return-Path: <kuno@gssm.otsuka.tsukuba.ac.jp>
X-Original-To: y-kuno@uec.ac.jp
Delivered-To: ka002689@uec.ac.jp
Received: from localhost (localhost [127.0.0.1])
    by mx-east.uec.ac.jp (Postfix) with SMTP id 49A144E348E
    for <y-kuno@uec.ac.jp>; Sun, 29 May 2016 12:26:59 +0900 (JST)
Received: from utogwpl.gssm.otsuka.tsukuba.ac.jp (utogwpl...[210.154.96.162])
    by mx-east.uec.ac.jp (Postfix) with SMTP
    for <y-kuno@uec.ac.jp>; Sun, 29 May 2016 12:26:58 +0900 (JST)
Received: (qmail 17906 invoked from network); 29 May 2016 03:26:58 -0000
Received: from OneOfLocalMachines (HELO gssm.otsuka.tsukuba.ac.jp) (127.0.0.1)
    by localhost with SMTP; 29 May 2016 03:26:58 -0000
From: "Yasushi_Kuno" <kuno@gssm.otsuka.tsukuba.ac.jp>
To: y-kuno@uec.ac.jp
Subject: test
Mime-Version: 1.0
Content-Type: text/plain;charset="ISO-2022-JP"
Date: Sun, 29 May 2016 12:26:58 +0900
Sender: kuno@gssm.otsuka.tsukuba.ac.jp
Message-Id: <20160529032659.49A144E348E@mx-east.uec.ac.jp>
                ←この 1 行がヘッダと本文の区切り
これはテストです
```

図 3.13: SMTP によりやりとりされる RFC822 形式メッセージ

ヘッダは「フィールド名: 値」の形の行が並んだもので、Subject:や From:などの情報もすべてヘッダフィールドとして伝達されています。ただし、誰が送信者かの情報はこれとは別に SMTP プロトコルでも伝達しています。この情報を**エンベロープ From** と呼び、最後に Return-Path:ヘッダに格納されます。また、Received:ヘッダを見ると、メッセージがどのような経路を通って中継されてきたかを追跡できます (途中のサーバがここに嘘を書き込んでいない限り)。

上の例のメッセージは筆者が筑波大内から電通大の自分のアドレスにあてて送信したものです。この場合はReceived:が4つありますが、受信相手が「127.0.0.1」のものは同一マシン内での受渡しなので、中継しているメールサーバは2つです。

なお、RFC822はあくまでもこのような単純な文字の並びから成るメッセージを格納する規格なので、音や画像など、また場合によっては日本語の長い行など、この規格に収まらないデータを送る場合はそのデータをまずビット単位のデータと考えてから英字の列に変換(**符号化**)して送ることになります。このような多様なデータを送る際の約束は**MIME**(Multipurpose Internet Mail Extension)と呼ばれる規格で定められています。

演習2 Thunderbirdを動かし、自分のアカウントを設定しなさい。設定が終わったら、「メール作成」を選び、宛先「自分の`ID@edu.cc.uec.ac.jp`」に簡単なメールを送りなさい。次に隣に座っている人のIDを尋ね、同様に送りなさい。送れたら、以下の演習を1つ以上やってみなさい。

a. 「メール受信」を選択し、送ったメールが読めることを確認する。さらに本文表示領域の上の「操作を選択」から「ソース表示」を選び、RFC822形式メッセージ全体を表示する。どのようなメッセージヘッダがあるか、そこにどんな情報が含まれているかを検討する。

b. 普段自分が使っている(電通大以外の)アカウントから「自分の`ID@edu.cc.uec.ac.jp`」に簡単なメールを送りなさい(または知合いに送ってもらうのでもよい)。送られたメッセージについて上と同様にメッセージ全体を表示し、上とはどのような違いがあるか、どのような経路でメッセージが送られて来ているかを中心に検討しなさい。

c. 今度は件名が日本語だったり、短い/長い日本語の行が入っていたり、画像などの添付されたメールを同様に送りなさい。送られたメッセージについて上と同様にメッセージ全体を表示し、上とはどのような違いがあるか、データがどのような形で送られて来ているかを中心に検討しなさい。

電子メールで注意するべきことは、「内容は(暗号を使わない限り)覗き見られたり改ざんされる可能性が常にある」点です。前者は、世界中のメールサーバは基本的にRFC822形式のメッセージを中継するので、サーバの管理者が中身を見るのは容易だからです。後者も、サーバはRFC822形式に合致したメッセージであれば単純にそれを受け入れて宛先に配送するので、「中身が本もの」という保証は何も無いことによります。ですから、そのメッセージに嘘がないかどうかの判断規準は基本的に「知っている相手とのやりとりで」「その相手がいつも言っていることと矛盾しないことを言っているか」に尽きるわけです。

3.6 ネット上のコミュニケーション

3.6.1 ネット上のコミュニケーションが持つ性質

ここまでで電子メールやWebアプリケーションについてひととおり見て来ました。過去においてはネット上のコミュニケーションは電子メールや**メーリングリスト**[10]が主流でしたが、今日ではWebアプリケーションから発達してきた掲示板(2ちゃんねる等)、SNS(Facebook, Twitter, Instagram、…)、メッセージ交換サービスの比重が高まって来ています。そして、これらの上で毎日膨大な量のトラブルが発生し、多くの人が嫌な目にあったり時間を無駄にしています。このような問題に巻き込まれないようにすることも、この科目で学んで頂きたい重要な内容です。

ここではまず、ネット上の**コミュニケーション**手段一般について考えてみます。ネット上のコミュニケーション手段は一般に次のような性質を持つと言えるでしょう。

- 文字が中心である — 通話や動画もありますが、やはり文字によるメッセージ中心になります。そして文字だとつい「言葉足らず」になりやすく、その結果誤解や食い違いを生みがちです。

[10] メーリングリストサーバのメールアドレスにメッセージを送ると全参加者にそのメッセージが転送される仕組みで、これによって全員による情報交換が可能になります。

- 手段ごとに特性が異なり、それによる影響がある — たとえば電子メールに基づくコミュニティでは1つのメッセージを全員が読むまでに時間が掛かるので「ゆっくり」議論をするような慣習ができます。逆にチャットやTwitterのように短いメッセージを前提とした場では頻繁なやりとりが行われ「その場かぎり」の話題が中心となります。
- **コミュニティ**ごとに「常識」が異なる — たとえば巨大掲示板サイトの技術的なテーマを題材としたあるスレッドでは、「初心者です」という書き出しは良くないとされていました。それは「初心者を免罪符に自分で調べることなく質問してくる」人がいたためですが、このように一般には問題ない行動でも場によっては問題とされることがしばしばあります。[11]
- **炎上**が起きる — ネット上には他人を攻撃することを趣味にする人が多数いて、問題行動や問題発言を見つけるとその場に集まって来て非難の書き込みを集中させたりします。
- **匿名**による勘違い — ネット上の場では「自分が誰だか明かさずに発言できる」ものも多くあり、自分が誰だか知られないという安心感から問題行動や問題発言がよく見られます。[12]
- **忘れられない** — 日常ではなにか失言をしてもその場にいた人の記憶が薄れるまで待てばあまり問題にならないのですが、ネット上の情報はコピーが簡単なため「記録」されてしまいやすく、そうなるといつでも蒸し返されてくる恐れがあります。[13]

3.6.2 電子メールに関する留意事項

電子メールは今日では高校生以下にはあまり使われていませんが、逆に大学や社会では広く使われており、社会に参加する上で不可欠となっています。そこで知らないままに「LINEやSNSののりで」メールを使って失敗する例が多く見られます。次のことを注意しましょう。

- 内容を表す件名 (Subject:) を記入する — 忙しい人は日に何十もメールを受け取るので、MUAに表示される件名で内容を把握できないと大変迷惑になります。[14]
- 必ず名乗り丁寧な言葉づかいをする — 出だしで「○○大学×年の△△です。」のように自分が誰かを知らせるようにします。こうすることで、本題の予測がつけやすくなり誤解を防げます。本題も丁寧な言葉づかいで簡潔に用件を述べるようにします。最後は「よろしくお願いします。」などでしめくくればよいです。メールは目上の相手に出す機会が多くなるため、常に丁寧な言葉づかいを使う習慣をつけた方が結局得です。
- 添付ファイルは注意して使う — メールは大量データの扱いには向いていないため、大きなデータ (写真や圧縮アーカイブなど) を添付することは避けるべきです。また受け取る際も知らない相手からの添付ファイルはマルウェアの可能性が高いので開いてはいけません (ということは、自分の添付ファイルもそう扱われる可能性があることを理解するべきなわけです)。
- メールは発信者の著作物 — メールのメッセージも文章である以上著作物で、あなたが読むのは了解の上でしょうけれど、他人に勝手に公開することはできません (許可があればOK)。公開されていない著作物は「引用」もできないことに注意。

3.6.3 SNSなどのコミュニケーションサービスに関する留意事項

Twitter、インスタグラム等の**コミュニケーションサービス**は高校生から大人までが使いますが、とくに大学生・若者が問題を起こすことが頻繁にあります。次のことを注意しましょう。

---

[11] このため、ネット上の場に参加する前にまず1か月くらいは「ROMる」(書き込まずにやりとりを読んで勉強する) ことがよいとされます。

[12] しかし一旦炎上が起きると、「発言者が誰だか特定する技能に長けた人」がやってきてあちこちの情報をもとに実名や住所などを明らかにしてそれを勝手に公開することが多く起きています。

[13] 米国で自分でSNSに貼った「恋人とのキス写真」が学校に見付かった保育専門学校の女子学生が退学になった例があります。

[14] 内容を表す件名としてダメなものは「こんにちは」のように情報の無いものです。よい例は「○月○日の××の授業内容の質問」のように本題を表すようなものです。

- 投稿内容は仲間内だけでは済まない — ふだん何も問題がない内容を交換している間は、反応するのは仲間だけなので、仲間しか読まないと勘違いしがちです。しかし実際には他人でも読める場合が多く、影で「あの人は実はこんなことを言ってる」と思われている可能性があります。
- **SNS** の「友人のみ」は保証されない — SNS ではメッセージの見える範囲を「友人のみ」「友人の友人」等限定できますが、それを広めたいと思った「友人」がコピーすればいくらでも広まります。いちど投稿したらその内容は全世界から見られるものだと考えるべきです。[15]
- 起きる可能性のある悪いことは必ず起きると思え — あなたがネットに悪ふざけ写真を流そす前に必ず「これを親が/学校が/将来の就職希望先が見たら」と想像すべきです。実際にプライベートな悪ふざけ写真のため希望する職業の道を断たれた若者は多くいます。
- 「炎上」は大きなダメージ — 「問題発言」「問題写真」などを投稿したとたん、それを見つけて炎上させる人が一斉にやって来ますし、匿名のはずでも「名寄せ」などの技術であなたが誰かを明らかにし、まとめサイトに載せたりします。ネットに投稿するときには、常にその危険を想像した上で「問題ない」と思うことだけにとどめるべきです。

このほか追加として、授業関係のものをネットに書く際の注意を記しておきます。

- 配布物の公開 — 配布物は当然教員等の著作物であり、許可なく公開してはいけません。[16]
- 授業の「実況」 — 担当教員の許可を取りましょう。教員によっては、その場にいる学生にとどまること前提で「重要な」ことを話してくれるので、公開されると困ります。また授業内容は教員の工夫の結果であり、勝手に持ち出されたくないと考えるかも知れません。
- 自分のレポートの「公開」 — レポートは自分の著作物ですが、それを丸写しされたら科目運営として困るわけで、授業の妨害をしていると判断されるかも知れません。

演習 3 ネット上で次のような問題事例を 1 つ以上探して報告しなさい。探した事例についてレポートを見る人が検証できるように URL を明記し、起こったことを分かりやすく整理して示すこと。また、「何が原因で悪い結末に至ったか」をきちんと考察すること。

a. 2 ちゃんねるなどの掲示板サイトで、最初は楽しく対話ができていたのに、途中から喧嘩になって後味悪い結末となった事例。
b. Twitter や SNS などのサイトで、仲間内だけだと思って気軽に書いた内容が想定しない相手に見られていて不幸な結末となった事例。
c. SNS やブログなどでの発言に対して、非難が集中して炎上や本人特定などに至った事例。

課題 3A

今回の課題は「演習 1」「演習 2」「演習 3」に含まれる小問 (合計で 9 個) の中から 1 つ以上を選択し、結果をレポートとして報告して頂くことです。LMS の「assignment ＃ 3」の入力欄に入力してください (直接打ち込むとログイン時間切れになった時に内容が消失するので、メモ帳など他のソフトで作成して完成後にコピーすることを勧めます)。以下の内容がこの順に含まれるようにしてください。

- 冒頭に題名「コンピュータリテラシレポート＃ 3」、学籍番号、氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も別途書く。)
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。

[15]「ネットに投稿する前に、そのメッセージを紙に書いてあなたの家の玄関に貼ってもよいか想像してみよう。そうしてもよいと思うなら、投稿してもよい」という指針があります。
[16] スライドや板書の撮影も同じです。撮影自体が複写なので、撮影しないように言われた場合は従うこと。

- 以下のアンケートに対する回答。

  Q1. セキュリティ、WWW、電子メール、ネットコミュニケーションの難しさについてどれくらい知っていましたか。新たに知ったことで面白かったことは何ですか。

  Q2. 電子メールの仕組みについてどのように思いましたか。

  Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

なお、課題はグループでやって構いません。その場合も、(メンバー全員の氏名を明記した上で) レポートは必ず各自で執筆してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。

＃4　コンピュータの動作原理

今回の目標は次の通りです。

- 整数の表現 (符号なし、2 の補数) とその操作について理解する — コンピュータ上の演算が持つ特性について知っておくことで実験や研究のための計算での失敗を減らすことができます。
- コンピュータの基本的な動作原理について理解する — コンピュータの動作原理を知ることで何が得意/不得意かという的確な判断ができ研究がスムーズに進められます。
- 機械語 (アセンブリ言語) レベルの簡単なプログラミングを体験する — プログラムを動かしてみることでコンピュータの特性が具体的に理解できます。

4.1　コンピュータとデジタル情報

4.1.1　デジタル情報とビット ex

皆様は**デジタル** (digital)、**アナログ** (analog) という用語を聞いたことがあるはずです。いちばん身近なのはおそらく「デジタル時計」(文字で表示される時計) と「アナログ時計」(針が連続的に動く時計) という言葉かも知れません。また、体重計などもデジタル式 (数字で表示されるもの) と、アナログ式 (昔ながらの、針が動くもの) がありますね。

では、デジタルとアナログの区別は何でしょう? 上の例からだと数字と針の違い、ということになりそうですが、もっと一般的に言えば、デジタルとは値が有限個の決まった値のどれか 1 つという形で表されるもの、アナログとは値が連続的に変化し得るもの、ということになります。

デジタル式体重計は「○○○. ○ Kg」という表示窓がついているとすれば、表示できる体重は全部で 10,000 通りしかありません (その代わり、ぱっと見てすぐ分かります)。つまり数字で表したものは常にデジタルです。そして、コンピュータが扱う情報はすべてデジタル情報なのです。

デジタルな情報の最小単位は「ある」「ない」のどちらか、「はい」「いいえ」のどちらか、「0」「1」のどちらか、といったものだと考えられます。これを「0」「1」で代表させ、**ビット** (bit) と呼びます。そして、すべてのデジタル情報はこの「0」「1」を沢山並べるだけで表現可能です。

たとえば、現在の天気を「雨が降っていない」「雨が降っている」の 2 通りの場合に分けたとすると、その情報をたとえば次のように 1 ビットの情報として表すことができます:

ビット表現	意味
0	雨が降っていない
1	雨が降っている

1 ビットはデジタル情報の最小単位ですが、複数のビットを並べた**ビット列**とすることで、より多くの情報を表現できます。たとえば、雨が降っている/いないでは大まかすぎるので、もっと詳しい情報として「晴れ」「曇」「雨」「雪」のどれであるかが知りたいとします。これは、たとえば次のように 2 ビットに対応させて表現できます。

ビット表現	意味
00	晴れ
01	曇
10	雨
11	雪

このように、ビット列の長さを1増やすと、表せる場合の数は2倍になり、一般にNビットのビット列では2^N通りの場合を表すことができます。そして、デジタル情報とは「いく通りかの場合のうちのどれか」という情報なので、すべてのデジタル情報は(必要なだけの長さを決めることによって)ビット列で表すことができます。

コンピュータとはひらたくいえば、ビット列を蓄積/転送/加工するための装置であり、その機能によってあらゆるデジタル情報を取り扱うことができます。さらに、これから実際に見ていくように、人間の介在なしに自動的に処理を行える、という点も重要です。

4.1.2　2進法 — 0と1による数値の表現 ex

コンピュータでは数を扱うことも多いので、ビット列を数に対応させる方法があると便利です。図4.1のような5枚のカードのそれぞれの状態を、表だったら1、裏だったら0として、合わせて5ビットで表すものとします。そして、この5ビットが表す「値(整数)」は、見えている丸の個数であるものと定義します。たとえば「01010」だと、「8」「2」のカードが表なので、この2つを足した値「10_{10}」を表すわけです。[1]

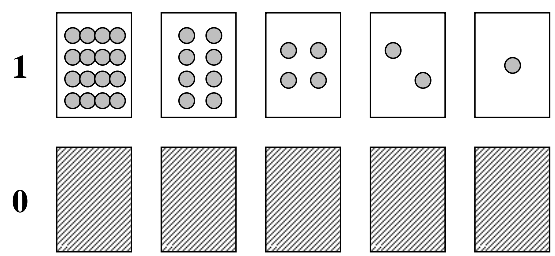


図4.1: 5枚のカード

この方法で数を表すやり方を**2進法**(binary system)と呼びます。なぜそうなのかを、私たちが普段使っている数の表し方である**10進法**(decimal system)と対比させて説明します。10進法では、数字として「0」〜「9」の十種類を使い、数字を並べて数を表しますね。そして、数を並べて書いたとき、最も右の桁が「$1(=(10)^0)$の桁」、次が「十$(=(10)^1)$の桁」「百$(=(10)^2)$の桁」「千$(=(10)^3)$の桁」と進んで行きます。そして

```
235
```

と書いた場合、これは「百が2個、十が3個、1が5個」を意味します。3桁で表せる最も大きい数は「999」で、それより1多いのは「1000」つまり次の位の値1個(千)ということになります。

これに対し2進法では、数字として「0」「1」の2種類を使い、数を並べて書いたとき、最も右の桁が「$1(=2^0)$の桁」、次が「$2(=2^1)$の桁」「$4(=2^2)$の桁」「$8(=2^3)$の桁」と進んで行きます。そして

```
101
```

と書いたとき、これは「4が1個、2が0個、1が1個」(ということは5)を意味します。数字が0と1しかないので簡単ですね。そして、3桁で表せる最も大きい数は「111」で、それより多いのは「1000」つまり次の位の値1個(8)ということになります。

なぜコンピュータでは2進法を使うのでしょうか。それは、電子回路では「信号のある/ない」だけを区別するようにすることで、回路の設計が単純化され、高速化や高密度化が容易になるからです。

たとえば、2進法による足し算を見てみましょう。1桁の足し算の表は次のようになります。

+	0	1
0	0	1
1	1	10

[1] このように基数を添字で表現することがあります。例えば「10_2と2_{10}は同じ」は「2進法の10と十進法の2は同じ」と読むわけです。

つまり、数字が 0 と 1 しかないわけですから、「0+0=0」「0+1=1」「1+0=1」「1+1=10」これだけです。最後のがびっくりしたかも知れませんが、1+1 の結果は 2 進表現の 1 桁では表せないので上の桁に桁上がりが起きて「10」になるわけです (そして先に学んだようにこれが「2」に相当します)。

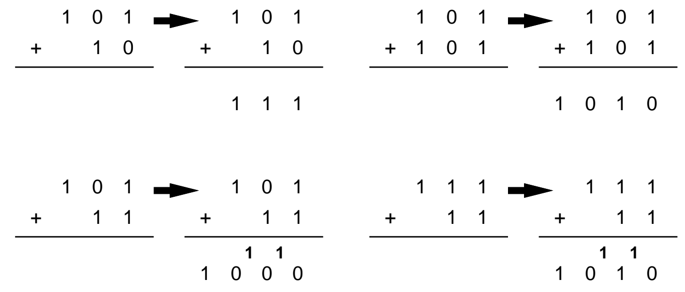


図 4.2: 2 進法の足し算

皆様は小学生のときに「0〜9 の値と 0〜9 の値を足すといくつになるか、また桁上がりがあるかどうか」を覚えさせられたはずですが、それに比べると上の規則はずっと簡単ですね? ですからコンピュータの回路にもしやすいわけです。桁上がりをつかって複数桁の足し算をしている例を図 4.2 に示しておきます。

表 4.1: 4 ビットのビット列とその値

ビット列	値	16 進	ビット列	値	16 進
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	a
0011	3	3	1011	11	b
0100	4	4	1100	12	c
0101	5	5	1101	13	d
0110	6	6	1110	14	e
0111	7	7	1111	15	f

2 進法は、ビット数が増えると、見るのも書き写すのも大変になります。そこで、4 ビットを 1 つの桁と考えて表 4.1 の右側にある `0` から `f` までの 1 文字で表す方法がよく使われます (`9` から先は数字がないため、`a`、`b`、`c`、`d`、`e`、`f` を充てているわけです)。これを **16 進** (hexadecimal) 表記と呼びます。

たとえば

```
1010 0001 0010 1011
```

であれば

```
 a    1    2    b
```

になるわけです。なぜ「16 進」かというと、これを数値として見た場合、1 つ桁があがるごとに (2 進法では 4 桁ぶんなので) 値は $16 (=2^4)$ 倍になるからです。そして、「`a12b`」は値としては

$$\underline{10} \times 4096 + \underline{1} \times 256 + \underline{2} \times 16 + \underline{11} \times 1 = 41253$$

を表すことになります。このほか、2 進法を 3 桁ずつ区切って表現する **8 進法**が使われることもあります。8 進法の場合は各桁の数字は 0〜7 の範囲ということになります。

4.1.3　2の補数による負数の表現 ex

実際にコンピュータで整数を扱うときには、演算回路の大きさやメモリに格納するデータの大きさに応じて32ビット、64ビットなどの決まったビット数で扱うことが普通です。一般に前節で説明した形の2進表現がNビットの場合、2^N通りのものが表せるので、0〜2^N-1までの範囲の整数が表現できます。これを負の数を含まないことから**符号なし表現**といいます。

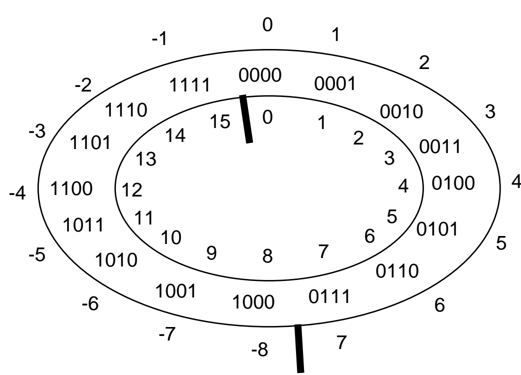


図4.3: 4ビットの2の補数表現と符号なし表現

では負の数を扱う場合はどうでしょうか。ここでは多くのコンピュータで使われている**2の補数** (two's complement)と呼ばれる表現を説明します。2の補数表現では、一番上位の(左の)ビットが符号ビットであり、これが0だと「0以上の値」、1だと「負の値」を表します。たとえば4ビットの2の補数であれば、1ビットが符号ですから、残り3ビットで8通りの場合が表せるので、0以上の数としては「0〜7」、負の数としては「-8〜-1」のそれぞれ8通りが表せます。[2]

これを具体的には図4.3のように割り当てます。ドーナツ型の上には4ビットの2進表現、内側にはそれを符号なし整数として解釈した場合の値、外側には2の補数として解釈した場合の値を記してあります。つまり、負の数の側は符号なし表現としての値から16を引く(マイナス側に16ずらす)ようになっています。

この表現がなぜ広く使われているかというと、負の数の計算がこれまでにやった符号なし計算と同じ回路で済むからです(そして引き算は符号を反転して足せばよい)。それが分かるためには、符号反転の方法がまず分かる必要がありますね。符号の反転は「0と1を交換して、それから1を足す」ことでできます。たとえば3は「0011」ですから、その0と1を入れ換えて「1100」、それに1を足して「1101」これが-3ということになります。

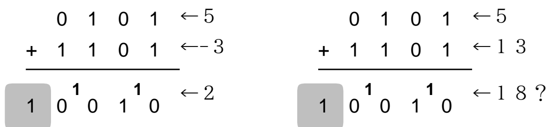


図4.4: 4ビットの2の補数表現での足し算

では「5+(-3)」をやってみましょう。図4.4左のように、これまでの足し算と同じ操作により、確かに期待通りの値「2」が得られます。しかし一番上の桁上がりは? それは、回路のビット数が4ビットなのですから、この網掛けになっている5ビット目は失われて消えるので、単に無視すればいいのです。

なんだか疑問ですね? 同じビットを符号なしとして解釈したらどうなのでしょう? それは図4.4右にありますが、「1101」を符号なしで解釈すると13ですから、5と足すと18になるはずです。しかし4ビットの符号なし表現では0〜15しか表せないはずです。つまり桁上がりが無視された結果、正しくない答えである2が結果になります(これは正しい答え18を16ずらした値です)。

[2]0以上の方には0が含まれるので、2の補数で表せる正の数は負の数より1つ少ないことに注意。

このように、コンピュータは有限のビット数で計算をするため、その範囲内で表せない結果になるような計算は正しく求まりません。これを**オーバフロー** (overflow、あふれ) と呼びます。たとえば4ビットの場合、図4.3の太線のところを計算結果が超えるとオーバーフローになります (上で見たように、符号なしの場合と2の補数の場合でオーバーフローの起こる境界は違います)。

演習0 2進表現の練習として次のことをやりなさい。

a. 次の10進表現を2進表現に直しなさい。8, 17, 25, 107

b. 次の2進表現を10進表現に直しなさい。1011, 100100, 111010

c. 次の10進表現を4ビットの2の補数表現に直してから足し算しなさい。3+7、7+(-5)、(-6)+(-3)

d. 4ビットの2の補数表現した2つの数を足すとオーバーフローになり正しく結果が表現できない例を作りなさい。「正の数+正の数」の例と「負の数+負の数」の2例作ること。

4.2 コンピュータと情報処理

4.2.1 コンピュータとプログラム ex

この講義の＃1において「コンピュータとは情報を扱う機械である」と述べましたが、前節の説明から、その情報はデジタル情報であり、ビット列として表されていることが分かりました。そこで定義を次のように具体化します。

◎ コンピュータとはビット列を処理する装置である

ここで「処理する」というのは、転送する (あちこち移動する)、蓄積する (記録・保管する)、そして加工することを表します。とくに最後の加工が大切で、これによってコンピュータは新しい情報を作り出したりできるわけです。

では、コンピュータの内ではどのようにして「ビット列の加工」を行うのでしょうか。数値の計算など基本的なものについては、そのための回路が組み込まれています。しかし、コンピュータに行わせたい「ビット列の加工」のバリエーションすべてをあらかじめ電子回路として組み込むことは不可能です。

そこで、コンピュータでは特定の計算を電子回路に組み込む代わりに、電子回路ではごく基本的なビット列の加工だけを用意しておき、それらを後で自由に組み合わせることによってさまざまな加工を行います。

しかし、各種の加工を行う回路を「組み合わせる」には、そういう配線を行う必要があるのでは? そこが実は重要なポイントで、現代のコンピュータでは配線を行う代わりに「どう組み合わせるか」を「命令として与える」ことで自由な加工を実現しています。ここがまさに、コンピュータを作り出した人たちの偉大なアイデアです。具体的には、次のようにしています (図4.5)。

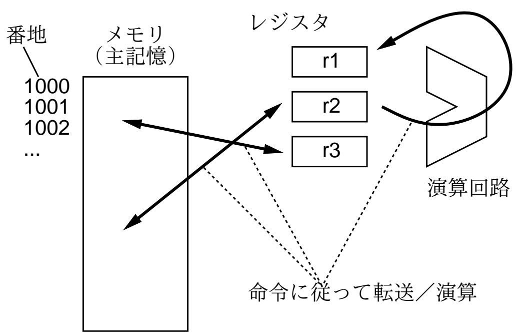


図 4.5: コンピュータと命令

- すべてのデータは**メモリ** (memory) ないし**主記憶** (main storage) に格納する。メモリには**番地** (address) がついていて、番地を指定してデータを格納したり、取り出して来たりできる。
- **CPU**(central processing unit、中央処理装置) はデータをメモリから取り出し、**レジスタ** (register) と呼ばれる記憶場所を利用しながら、さまざまな加工を行い、結果をまたメモリに戻す。
- データの移動や加工はすべて**命令** (instruction) で指定する。

実際には 1 つの命令では簡単な動作 1 つしかできないので、命令を並べてそれを順番に実行していくことで、より込み入った動作を行わせます。この、命令を並べたものが**プログラム** (program) なのです。

プログラムもまたメモリに格納されており、データの一種として扱えます。ということは、新しいプログラムをよそから持って来ることも、さらにはプログラムを動かすことで新しいプログラムを作り出すことも、可能になるのです。非常に画期的だと思いませんか? そういうわけで、この「プログラムもメモリに格納されたデータである」という構成を、**プログラム内蔵** (stored program) 方式、ないし考案した科学者の名前から**ノイマン型** (Neuman architecture) と呼びます。

4.2.2 小さなコンピュータのシミュレータ ex

本ものの CPU の命令は複雑なので、ここでは簡単化した (架空の)「小さなコンピュータ」を想定して、その命令を動かしてみましょう。この小さなコンピュータは JavaScript 言語で記述されていて、ブラウザ上で動作します (図 4.6)。とりあえず使ってみましょう。

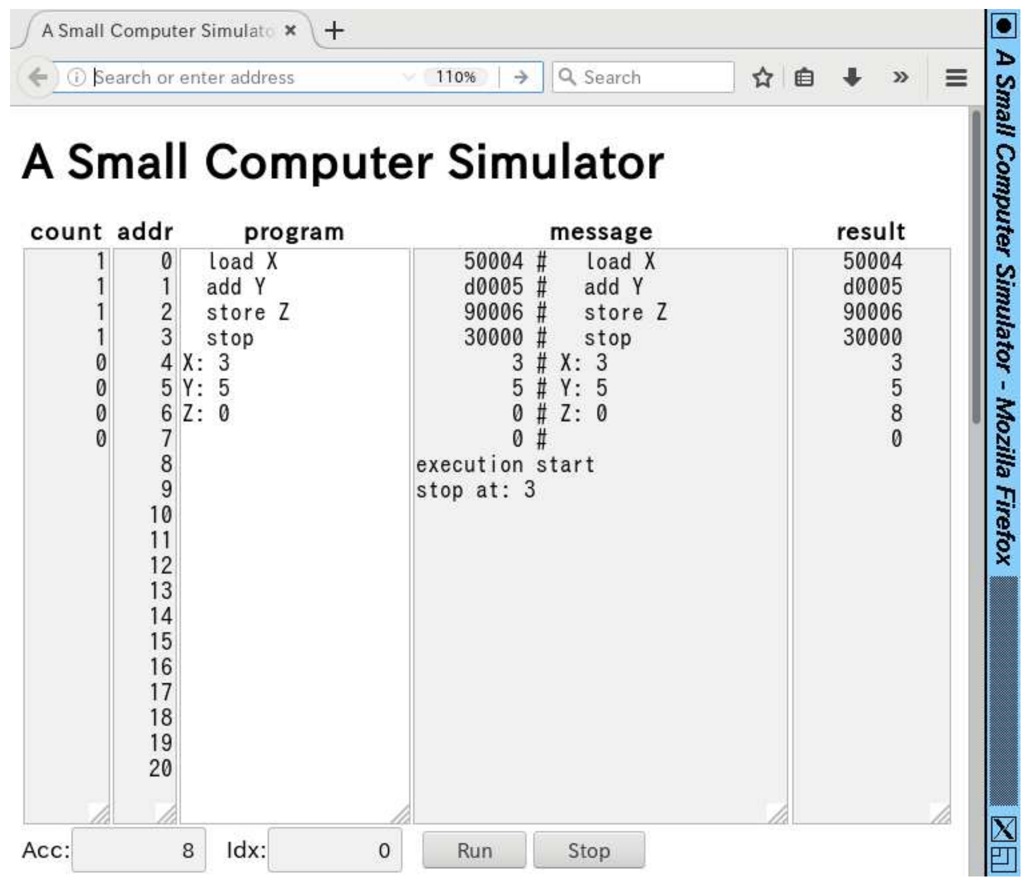


図 4.6: 「小さなコンピュータ」の画面

ここで「プログラム (program)」と記された欄に、1 行に 1 つずつ、命令を書いて行きます。たとえば、次の 7 行のコードを打ち込んでから「実行 (run ボタンをクリック)」してみましょう。なお、**コード** (code) というのは、「プログラムないしその断片」を表す一般的な用語です。

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

ここで「load」は、数値をメモリの指定場所 (この場合は X という名前のついている番地) からアキュムレータ (Acc) というレジスタに取り出して来る命令 (図 4.7 上)、「add」は、メモリを指定場所 (この場合や Y という名前のついている番地) から取り出し、それを Acc の内容に足し込む命令 (図 4.7 中)、「store」は、Acc の内容をメモリの指定場所 (この場合は Z という名前のついている番地) に格納する命令です (図 4.7 下)。これらの命令はいずれも「どこからどこへ」のように 2 つの場所を本来は指定する必要がありますが、その一方は Acc になっているので場所を 1 つ指定すれば済むのです。

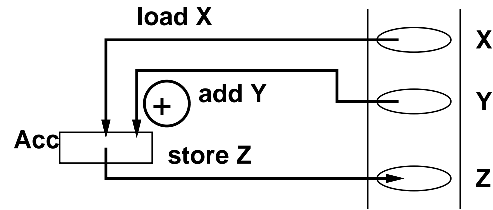


図 4.7: ロード命令、加算命令、ストア命令

そして最後の「stop」は、その名前通りプログラムの実行を停止する命令です。この命令が無いと、コンピュータは次の番地の内容を命令だと思って実行してしまいます (上の例だと次には「3」「5」等が入っていて、これらを実行して行き、正しく止まりません)。

プログラムの後には、X、Y、Z という名前のついた場所を用意し、それぞれに 3、5、0 という値を入れておきます。そうすると、プログラムを実行した結果、X と Y の値を足した結果 (8 ですね) が Z の場所に格納され、確かに足し算ができていることが分かります。

実行開始時には、**プログラムカウンタ** (program counter、PC) というもう 1 つのレジスタ (画面には見えていない) に 0 を入れてから開始します。CPU は PC が指す番地 (最初は 0 番地) から命令を取り出し、PC をその命令の次の番地に変更します。最初の命令を実行し終ったらまた PC の指している番地から命令を取り出し、PC を次の番地にします。これを繰り返して実行が進みます。

なお、表示の見かたですが、message の欄はコードを命令に翻訳した様子や実行開始/停止の情報が表示されます。result の欄はプログラムが停止したときのメモリの内容が表示されます。

私たちが普段使っているコンピュータでは、画面やマウスなどを使ってデータをやりとりしますが、この「小さなコンピュータ」ではごく基本的な命令しか用意していないので、このようにメモリに直接データを用意して動かすようにしています。

実は、このように命令やメモリの場所に名前をつけて表すプログラムの書き方を**アセンブリ言語** (assembly language) と呼びます。アセンブリ言語のプログラムは**アセンブラ** (assembler) と呼ばれるプログラムによってビット列、つまり 0 と 1 だけから成るプログラムに変換され、CPU はそれを実行します。この、CPU が直接実行する形のプログラムのことを**機械語** (machine language) のプログラムと呼びます。「小さなコンピュータ」では、実行開始時に message 欄に機械語 (アセンブラの変換出力) を表示するようになっています。

上の例は「足し算」でしたから順番に命令を 4 つ実行すれば終わりでした。しかし実は、プログラムでは「計算した結果によって処理を切り替える」ということが可能であり、これによって複雑な処理が行えます。そのために、命令の中に「分岐 (ジャンプ) 命令」「条件分岐命令」があります。具体的には、通常の命令はその命令を実行し終わると「次の」命令に進むのに対し、分岐命令は番地を指

定し、次はその番地の命令の実行に進むようにさせます。そして条件分岐命令は、「Acc が 0 でないならば」のように条件を指定して、その条件が成り立っている時だけ分岐します (成り立っていなければ、次の命令に進む)。

たとえば、今度は X と Y のうち「より大きい値を」求めてそれを Z に入れることを考えます。そのためには、X から Y を引いてみて、マイナスなら Y の方が大きいと分かります。この考えに基づいてプログラムを作ってみましょう。

```
      load X
      store Z
      sub Y
      ifp Skip
      load Y
      store Z
Skip: stop
X:  3
Y:  5
Z:  0
```

「sub」は引き算の命令です。このプログラムではまず、X を Acc に取り出し、とりえあず Z に入れます。次に、Acc の内容から Y を引き算します。ここで、もしプラスなら X の方が大きくて OK ですから、Skip という場所に「飛びます」(ifp は Acc の内容がプラスなら指定した場所に分岐する条件分岐命令です)。プラスでないなら、もう 1 回 Y の値を Acc に持って来て、Z に格納します。いずれにせよ Skip の所に合流して、そこでプログラムは止まります。このように、条件判断して自動的に処理を切り替えることで、コンピュータは複雑な処理が行えているのです。

演習 1　「小さなコンピュータ」でここまでに出て来た例題をそのまま動かしてみなさい。うまくできたら、以下の処理を実行するプログラムを作成してみなさい。

a. 5 つの値を A、B、C、D、E というラベルの番地に入れておき、その合計を Z というラベルの番地に入れて止まる。データ例: 1, 2, 3, 4, 5 → 結果 15 (16 進では F)。
b. A というラベルのついた番地に入れておいた数値を 5 倍し、結果を Z というラベルのついた番地に入れて止まる。データ例: 4 → 20 (16 進では 14)。
c. 3 つの値を A、B、C というラベルのついた番地にそれぞれ入れておき、その 3 つの最大値を求めて、Z というラベルのついた番地に入れて止まる。データ例: 3, 6, 2 → 結果 6。

いずれにおいても、プログラムがどのように動作するか説明すること。

4.2.3　ループのあるプログラム

先の課題では、5 つの値の合計とか、5 倍とかなので、その数だけ足し算命令を使えば済んでいました。

では、合計に戻って、もっと沢山の数を合計したければどうしましょうか? A、B、C…のように沢山変数を並べていてはプログラムが長くなって大変そうです。そこで、Acc のほかにもう 1 つ、インデックスレジスタ (Idx) というものが用意されています。そして、load 命令の拡張版 loadx では「指定した番地より Idx の値だけ先の場所」からデータを取り出すことができます (図 4.8)。これを使って、並んだデータを順番に取り出し、合計して行けばよいのです。プログラムを見てみましょう。

```
      iload 0
Loop: loadx Data
      ifz End
      add Sum
```

```
        store Sum
        iadd 1
        jump Loop
End:    stop
Sum:    0
Data:   1
        2
        5
        0
```

Data というラベルの後に数行ぶんのデータがありますが、これを順番に持って来て合計するわけです。「iload」命令は Idx に指定した値 (この場合は 0) をロードします。次に、loadx 命令で Acc に

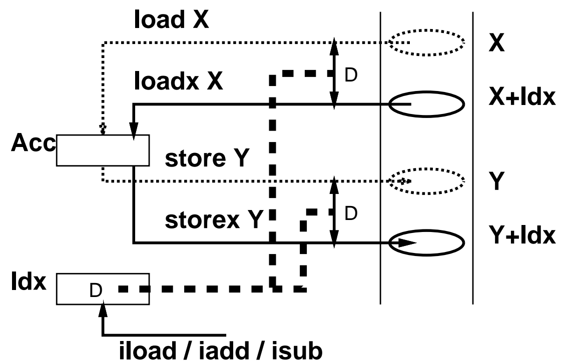


図 4.8: Idx を使うロード命令とストア命令

値を持って来ますが、最初は Idx は 0 なので、ちょうど Data の場所の値が持って来られます。もしその値が 0 なら、これは終わりの印なので (合計を取りたいのに 0 をデータに入れる必要はないでしょうから)、End へ分岐します。そうでなければ、Acc の値と Sum を加え、その結果を Sum に入れます。続いて、「iadd」命令で Idx を 1 増やしてから、「jump」命令で無条件に Loop へ戻ります。すると、次は Data の次の場所から値が取り出せるわけです。これを繰り返して次々に値を足して行き、0 が現れたら End へ来て止まりますが、そのときには Sum には総計が入っています。

このプログラムの「肝」は、手前 (上) 方向への分岐命令を使って一群の命令列を「繰り返し」実行させることで、短いプログラムでも沢山の処理を行わせられる、というところにあります。これが、今日のコンピュータの重要な原理だと言えます。最後に、この「小さなコンピュータ」が持っている命令の一覧を掲載しておきます (表 4.2)。[34]

演習 2 「小さなコンピュータ」で上に説明した N 個のデータの合計プログラムを動かし、さらにデータの個数を増やしたり減らしたりして、動作を確認しなさい。うまくいったら、以下の課題から 1 つ以上やりなさい。プログラムがどのように動作しているのか説明すること。

a. 正負とりまえぜて複数の整数を並べて与え (0 が終わりの印)、それらの数値の「絶対値の合計」を求めるプログラムを作りなさい。データ例:「1 -2 3 -4 5 0」→結果例: 「15」

b. 正の整数を複数個並べて与え (0 が終わりの印)、その「最大値」を求めるプログラムを作りなさい。データ例:「3 5 9 1 4 0」→結果例:「9」

---

[3]条件分岐命令が沢山ありますが、その覚え方は次のようになります。ifz〜「if zero」、ifnz〜「if not zero」、ifp〜「if positive」、ifn〜「if negative」

[4]「コード」はその命令を表現するビット列です。すべて 2 つずつコードがありますが、大半の命令はどちらでも動作は同じです。値を取り出す命令 (add、sub、mul、load) のみ、「命令の後に指定した数値を取り出す」「命令の後に指定した名前をつけたメモリの場所から取り出す」の 2 通りに分かれています。

表 4.2: 「小さなコンピュータ」命令一覧

名前	コード	命令の動作
nop	00,01	何もしない
stop	02,03	プログラムの実行を停止
load	04,05	Acc に値を持って来る
loadx	06,07	〃 (値/番地に Idx を足す)
store	08,09	Acc の値を格納する
storex	0a,0b	〃 (番地に Idx を足す)
add	0c,0d	Acc に値を足す
sub	0e,0f	Acc から値を引く
iload	10,11	Idx に値を持って来る
iadd	12,13	Idx に値を足す
isub	14,15	Idx から値を引く
ifz	16,17	Acc ＝ 0 なら分岐
ifnz	18,19	Acc ≠ 0 なら分岐
ifp	1a,1b	Acc ＞ 0 なら分岐
ifn	1c,1d	Acc ＜ 0 なら分岐
jump	1e,1f	無条件に分岐
neg	20,21	Acc の符号を反転

c. 2 つの整数 (いずれも 0 以上) を与えておき、その 2 つの積 (掛けた結果) を求めるプログラムを作りなさい。データ例:「X: 3, Y: 5」→結果例:「15」。ヒント: この問題では Idx レジスタは使う必要がありません。つまり iload や lodax 命令は使う必要がありません。[5]

4.3　コンピュータの性能向上とその意義

4.3.1　CPU の製造技術

コンピュータの CPU は「電子回路」ですから、もともとは**真空管** (vaccuum tube) や**トランジスタ** (transistor) などの個別素子を相互に配線して作っていました。しかし、CPU には非常に多数の素子や配線が必要なので、装置が巨大で高価だったり、信頼性が低いなどの問題がつきまとっていました。

今日ではこの問題は、小さな結晶の上に多数の素子と回路を直接作り込む、**VLSI**(Very Large Scale Integration) という技術によって克服されています。その原理を簡単に説明しましょう。VLSI を作るには**シリコン** (硅素、Si) の単結晶のうす切り (ウエハー) を用意します。これ自体は電気を通しませんが、この上に**ホウ素**や**リン**の分子をごく微量加える (拡散させる) と、その部分は電気を通すようになり、またその配置に応じてトランジスタの働きをするようになります。だから、微細な模様をデザインしてその模様を焼きつけたマスクに沿って拡散を行うことで、好きな形の素子の配置と電気配線がウエハー上に作れます (図 4.9)。

VLSI の何がそんなにすごいのでしょう。それは、個別の素子を使って作るのと比べると、ずっと小さい手間で多数の回路が量産できること、そして回路を細かくすることで「焼き付けて現像する」という同じ作業のままで同じ 1 つのチップに一層多くの回路を詰め込めるようになります。さらに、技術が進歩して回路が小さくなると、高速で動作させることが可能になります。

1965 年に、ゴードン・ムーア (後の Intel 社の共同創立者) が「ある一定サイズのチップに搭載できる素子数は、毎年 2 倍ずつになっている」という記事を発表しました。具体的な比率はその後、もう少し低い値 (たとえば 18 か月ごとに 2 倍) に修正されていますが、この「一定期間ごとに倍」という知見は**ムーアの法則** (Moore's law) として知られ、近年まで成り立ち続けてきました。このためにコ

[5] Idx には中の値を調べたりどこかに保存する機能が無く、メモリの内容を取り出す以外の役には立てにくいため。

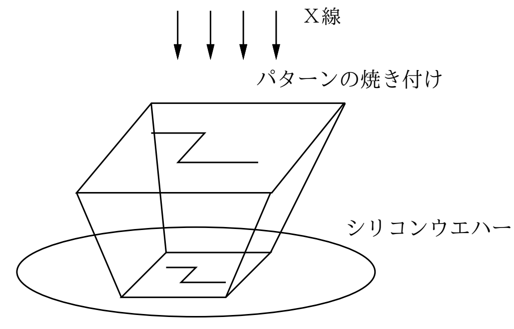


図 4.9: VLSI の製造方法

ンピュータはすごい勢いで進歩してきたのです。[6]

もちろん、最先端のチップを量産するには原理は同じ「焼き付けて現像」であっても、極めて高価な設備が必要です。日本のメーカーはどこも、もはやこの設備投資について行けていないわけですが…

4.3.2 デジタル革命の本質

コンピュータの原理や CPU の機能について説明してきましたが、結局コンピュータの何がこれほどのインパクトを世の中に与えているのでしょうか? いくつか挙げてみましょう。

- 汎用的なデジタル情報 — 世の中の多くの情報は、デジタル表現でき、その結果、コンピュータで扱うことができる。
- 汎用的な処理装置 — コンピュータは、プログラムを取り替えることで、デジタル情報の「どのような」処理であっても、(その処理の方法が分かっている限り) 行うことができる。
- コンピュータの低価格化と小型化 — VLSI 製造技術の発達により、コンピュータはどんどん安価に作ることができるようになってきた。たとえば、数ミリ角のチップで完全なパソコンと同じ機能を持つ CPU が (速度は遅いが) 実現できる。その結果、どこでも専用の機械や電子回路を組み立てて使うより、コンピュータを組み込んでソフトで処理を記述する方が安くて柔軟に処理できるようになった。
- コンピュータの高速化・大容量化 — ハードウェア技術の進化により、これまでは「扱えなかった」「計算できなかった」処理がどんどん実用的な時間でこなせるようになってきている。
- 通信とコンピュータの融合 — 以前の通信は「文字や音声を伝達する」だけのものだったが、インターネットに代表されるコンピュータネットワークでは、ネットワークを介してやり取りされるデジタル情報を直接コンピュータにより制御・加工することで、人間の介在なしに膨大かつ多種多様な情報をあらゆる場所に廉価に流通させることを可能とした。今やインターネットの存在しない世界はほとんど考えることすらできない状況である。

演習 3 (チャレンジ問題) 以下の課題から 1 つ以上を選んでやってみなさい。

a. 「小さなコンピュータ」の 1 命令あたりの実行時間 (または同じことですが 1 秒間あたりの平均命令実行数) を測ってみなさい。どのようにして計測したか、また答えの求め方の根拠を必ず記すこと。
b. 「小さなコンピュータ」で、これまでにまだ使ったことのない命令を使うプログラムを作りなさい。どのようなプログラムか、どのように動作するかの説明と、実行例をつけること。
c. 本もののコンピュータの 1 命令あたりの実行時間 (または同じことですが 1 秒間あたりの平均命令実行数) を測ってみなさい。対象とするコンピュータや測定方法はお任せしますが、レポートを読んだ人に数値の根拠が分かるように説明すること。

[6] さすがに今日では物理的な限界に到達し、ムーアの法則は成り立っていないと考える人が多い状況です。

課題 4A

今回の課題は「演習 1」「演習 2」「演習 3」に含まれる小問 (合計で 9 個) の中から 1 つ以上を選択し、結果をレポートとして報告して頂くことです。LMS の「assignment ＃ 4」の入力欄に入力してください (直接打ち込むとログイン時間切れになった時に内容が消失するので、メモ帳など他のソフトで作成して完成後にコピーすることを勧めます)。以下の内容がこの順に含まれるようにしてください。

- 冒頭に題名「コンピュータリテラシレポート＃ 4」、学籍番号、氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も別途書く。)
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。必ず自分が書いたプログラムを掲載すること。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 以下のアンケートに対する回答。

  Q1. コンピュータの動作原理やアセンブリ言語によるプログラムについてどれくらい知っていましたか。新たに知ったことで面白かったことは何ですか。
  Q2. 「小さなコンピュータ」でプログラムが組めるようになりましたか。
  Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

なお、課題はグループでやって構いません。その場合も、(メンバー全員の氏名を明記した上で) レポートは必ず各自で執筆してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。

＃5　ファイルシステムとファイル操作

今回の目標は次の通りです。

- ファイルシステムやディレクトリ階層の概念について理解する— Unix 上でのファイルの保管/整理について知っていることは実験/研究を効率よくこなす上で必須です。
- ファイルのコピーや名前変更など基本的な操作について理解する— ファイルの操作ができることは実験データの保管や加工のために必須です。
- ファイルやディレクトリの保護モードとその設定について理解する — 実験等でデータを保護する必要がある場合もありますし、逆に保護設定をゆるめないと使えない場面もあります。

5.1　ファイルシステム

5.1.1　2 次記憶装置とファイルシステム ex

メモリ (主記憶) はコンピュータシステムの「主要な」記憶装置ですが、容量が限られていて、電源を切ると内容が消えます。コンピュータシステムは電源が切られてもきちんと残って欲しい情報を沢山格納する必要があるので、次の条件を満たす **2 次記憶** (secondary storage) 装置を使用します。

- 安定記憶 — 電源を切っても消えず、内容が勝手に書き変わったりしない。
- 容量/コスト — 大量のデータを格納でき、1 ビットあたりのコストが低い。

今日のシステムで広く使われている 2 次記憶装置として、磁気コーティングした円盤に磁気的に情報を書き込み保存する**磁気ディスク装置** (hard disk drive、HDD) があります (図 5.1)。

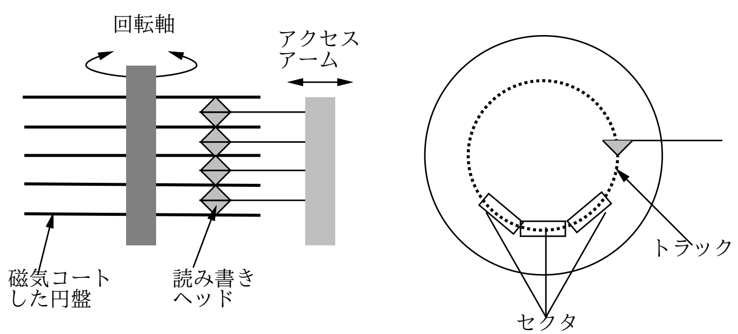


図 5.1: 磁気ディスク装置の構造

他に、電源を切っても内容が保持される半導体メモリである**フラッシュメモリ** (flash memory) も、モバイル機器を中心に使われます。これはコストが高めですが、HDD より高速なので、通常の PC でも磁気ディスク装置の代りに使うことがあり、その場合 **SSD**(solid state drive) と呼びます。[1]

5.1.2　情報の整理とファイルシステムの階層構造 ex

2 次記憶装置には大量の情報が入れられるので、それらに名前をつけて、構造を持たせて管理できるようにする必要があります。コンピュータシステムの中で、このような機能を提供するソフトウェア

[1] フラッシュメモリは書き換え回数の上限が数千回〜数万回程度で、それを超えると壊れてしまうので、実質何回でも書き換え可能なディスクの代わりに使うためには書き換え回数を押える工夫が不可欠になります。

のことを**ファイルシステム** (file system) と呼びます。そしてこの名称は「沢山のファイルが集まった構造全体」という意味でも使われます。以下では Unix を例に、後者の意味でのファイルシステムを説明します。ファイルシステム中の主要な要素は、次の 2 種類のものです。

- **ファイル** (file) — 任意のビットのデータ (メモリ同様にバイト単位でアクセスするので長さもバイト単位)。
- **ディレクトリ** (directory) — 「登録簿」という意味の英語で、ファイルやディレクトリをいくつでも入れられる。

あなたが多数の情報 (書類など) を保管し、必要に応じて取り出したり追加するとしたら、どうますか? 書類が少量なら、適当に積み上げておくだけでも済むかも知れません。しかし、何百、何千もの書類となると、それでは無理ですね。普通は、書類をバインダーなどに綴じ込み、キャビネットなどに入れて保管することでしょう。このとき、キャビネットには「学校関係」「保険関係」など大まかな分類を記し、バインダーには「○○年度」「××生命」などさらに細かい分類を記すはずです。こうすれば、情報のありかは分類をたどって探すことができます。

この「大分類→中分類→小分類」と次第に細かく分類していく構造を**階層構造** (hierarchy) と呼び、広く使われています。情報の整理に階層構造を使うことには、次の利点があります。

- 新たな情報を置くべき場所を、明確に決めることができる。
- 入っている情報を取り出すとき、ありかが簡単に分かる。

このようなことから、コンピュータのファイルシステムでも、階層構造が一般的に使われています。つまり、ディレクトリが容れ物でその中にファイルやディレクトリが入るわけです (情報には物理的な大きさはないので、バインダやキャビネットなど複数種類の容れ物にする必要がないのです)。

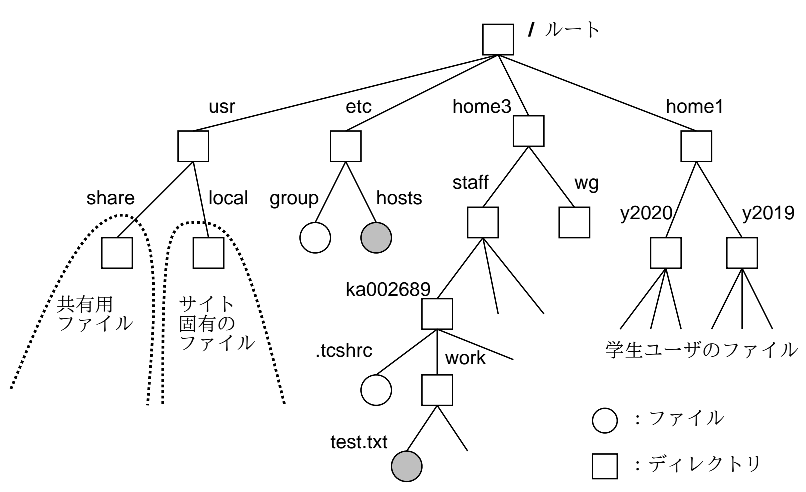


図 5.2: ファイルシステムの階層構造

図 5.2 に、本学共通システムのファイルシステムの階層構造を簡単化して示します。○で表したのがファイル、□がディレクトリです。一番上にあるディレクトリは**ルートディレクトリ** (root directory) と呼ばれます。これは、枝分かれした構造が木の根元 (root) を想像させるからです (植物の木と上下が反対ですが)。

ルートの下にはシステムのさまざまな共有ファイルを集めたディレクトリ `usr`、管理用ファイルを入れたディレクトリ `etc`、利用者のファイルを集めたディレクトリ `home1`〜`home3` 等があります。それぞれのディレクトリの下にはさらに、ディレクトリやファイルが多数置かれています。

5.1.3　パス名と現在位置 ex

次の問題は、多数あるファイル (やディレクトリ) をどのようにして「これ」と指定するか、です。このために Unix では**パス名** (pathnames) と呼ばれるものを使います。次の例を見てください。

```
/
```

これは、ルートディレクトリを表しています。そして、その次に名前を指定することで、そのディレクトリに直接置かれているものを指定できます。

```
/etc
/home3
```

これにより、管理用ファイルの置き場 etc と、教員のファイルの置き場 (の 1 つ)home3 が指定できます。さらに「/」で区切って名前を指定することで、それらの下のものが指定できます。

```
/etc/hosts
/home3/staff/ka002689/work/test.txt
```

これらは図 5.2 で灰色で塗って示したファイルです (指定方法はファイルでもディレクトリでも同じ形)。このような、「/」で区切った名前を並べてファイルやディレクトリを指定する記法がパス名です。そして上の例の「/」(ルート) から始まるものは**絶対パス名** (absolute pathnames) と呼びます。

Unix で動いているプログラムは、常にディレクトリのうちの 1 つを「現在位置」ないし**カレントディレクトリ** (current diretory) として覚えています。コマンド **pwd** は現在位置を表示します。

```
...$ pwd
/home3/staff/ka002689
```

各ユーザがログインした直後の現在位置を**ホームディレクトリ** (home directory) と呼び、各ユーザのファイルは基本的にその下に置かれます。このため、システム上に多数のユーザが居ても、それらのファイルが混ざることは無いわけです。

パス名は「/」以外に名前から始まる場合もあり、これを**相対パス名** (relative pathnames) と呼びます。相対パス名は、ルートではなく現在位置から、それぞれの名前をたどる意味になります。たとえば、先のファイル test.txt や、ユーザ ka002689 のホームディレクトリにあるファイル.tcshrc は、現在位置が/home3/staff/ka002689 の場合、次のように指定できます。

```
work/test.txt
.tcshrc
```

現在位置は「cd ディレクトリ」により変更できます。何も指定しない「cd」で自分のホームディレクトリに戻ります。あちこち移動して迷子になったら「cd」を実行してください。

```
...$ cd work     (cd /home3/staff/ka002689/work と同じ意味)
...$ pwd
/home3/staff/ka002689/work
```

この場合、現在位置が変わったので、先の 2 つのファイルは次のように指定できます。

```
test.txt
../.tcshrc
```

「..」は「ディレクトリ階層における 1 つ上のディレクトリ」という意味になります。ついでに、現在位置のディレクトリは「.」で指定できます。

5.1.4 ls — ファイル名一覧の表示 ex

個々のファイル操作で、まず必要なのは、ディレクトリに入っているファイル (とディレクトリ) の一覧表示です。それには ls コマンドを使います。ただの ls は名前の最初が「.」で始まるものを表示しないので、それらも見たい場合には「ls -a」とします。より詳しい (ファイル種別、ファイルサイズや変更日付などを含む) 情報を見たければ「ls -F」「ls -l」です。どの使い方でも、ディレクトリ名を指定すればそのディレクトリの、指定しなければ現在位置のディレクトリの一覧を表示します。

- ls — ディレクトリにあるファイルの一覧を表示
- ls -a — 「.」で始まるファイルも含めて表示
- ls -F — 名前の末尾に種別を表す文字 (ディレクトリなら「/」等) を付加して表示
- ls -l — ファイルサイズ、変更日付等の情報も表示

これらを使っているようすを見てみましょう。

```
...$ ls   ←「...$」は向こうが表示するコマンドプロンプトなので打ち込まない
WWW  a.out  bin  cl20  lib  public_html  sbin  test2.c  work  zz
...$ ls -F
WWW@  a.out*  bin/  cl20/  lib/  public_html/  sbin/  test2.c  work/  zz/
...$ ls -a
.             .claws-mail  .gconfd          .profile      a.out        test2.c
..            .config      .gimp-2.8        .ssh          bin          work
.Xauthority   .dbus        .gnome2          .tcshrc       cl20         zz
.ced          .emacs       .gnome2_private  .thunderbird  lib
.ced_centos   .emacs.d     .local           .viminfo      public_html
.ced_ubuntu   .gconf       .mozilla         WWW           sbin
...$ ls -l
tltal 20
lrwxrwxrwx  1 ka002689 faculty   11  5月  3  2016 WWW -> public_html
-rwxr-xr-x  1 ka002689 faculty 8352  5月 17 13:45 a.out
drwxr-xr-x  2 ka002689 faculty  217  4月  1 14:28 bin
drwxr-xr-x  2 ka002689 faculty   10  5月 17 13:46 cl20
drwxr-xr-x  4 ka002689 faculty   41  4月  2  2018 lib
drwxr-xr-x 35 ka002689 faculty 4096  5月  3 11:10 public_html
drwxr-xr-x  2 ka002689 faculty   71 10月 29  2018 sbin
-rw-r--r--  1 ka002689 faculty   35  5月 17 13:45 test2.c
drwxr-xr-x  2 ka002689 faculty  244  5月 16  2018 work
drwxr-xr-x  2 ka002689 faculty   99  5月 17 13:46 zz
...$
```

「ls -a」の結果を見ると「.」で始まるファイルが多数あると分かります。これは、Unix では各種のプログラムごとに、固有のオプション設定などを「.」で始まるファイルに書く習慣があるためです。そして、いつもそれらが表示されているとうるさいので、-a を指定しない限り ls はそれらを表示しません。「ls -F」の結果を見ると、ディレクトリは「/」つき、実行可能なプログラムは「*」つきで表示されます (その他については後で)。「ls -l」だと細かい情報が表示されますが、中央付近にファイルサイズ (バイト単位)、その右に最終変更日時があります (残りの情報は後で)。

5.1.5 ファイルの中身を調べる ex

ファイル名が分かったら、次は内容を見るコマンドを学びます (less については、man コマンドのところでも説明しました)。[2]

- **file** ファイル — ファイルの種別を表示
- **cat** ファイル … — 1 個以上のテキストファイルを次々に表示
- **more** ファイル — 長いテキストファイルを 1 画面ずつ表示 ([SP] で次の画面、「q」で終了)
- **less** ファイル — 戻ることもできる more ([SP] で次の画面、「p」で前の画面、「q」で終了)
- **eog** ファイル、**display** ファイル — (画像ファイルのみ) 画像を表示

5.1.6 コンプリーション

長いパス名やファイル名を打つのは面倒ですし、打ち間違いがあるとエラーになり、やり直すはめになります。そこで、入力をサポートする**コンプリーション** (completion、補完) という機能があります。これは、ファイル名等を入力中にどこでも [TAB] を打つと、現在入力中の文字列に続けて「曖昧さがない範囲でできるだけ長く」勝手にコンピュータが入力をしてくれる、という機能です。そして、このときもう 1 回 TAB を打つと、ファイルやディレクトリとして何があるのかを表示してくれます。

実際に先のディレクトリで試すと、次のようになるはずです (コマンドは必要なので、すぐ次の演習で使う cp コマンドを打つことにしました。cp の詳細も後で)。

```
cp /c[TAB]      ←「c」「p」「 」「/」「c」だけですぐ [TAB]
cp /class/      ←「/class/」まで入力された。
cp /class/[TAB]     ←この先何があるの?
... 多数の表示...   ←「これらの候補があるよ」
cp /class/j[TAB]   ←「j」を打ってまた補完
cp /class/jb/      ←ディレクトリの「jb/」が補完された
cp /class/jb/[TAB]   ←この先何があるの?
... 多数の表示...   ←「これらの候補があるよ」
cp /class/jb/cl20/w[TAB]   ←「cl20/w」を打ってまた補完
cp /class/jb/cl20/work/    ←ディレクトリ work まで補完された
cp /class/jb/cl20/work/[TAB]    ←この先何があるの?
iwai.jpg   otaku.jpg  umi.png    yama.png   zannen.png ←「これら
iwai.txt   otaku.txt  umi.txt    yama.txt   zannen.txt ←の候補…」
cp /class/jb/cl20/work/iwai.j[TAB]   ← iwai.jpg にするつもり
cp /class/jb/cl20/work/iwai.jpg  .   ←「.」(現在位置に同名コピー) で RET
```

演習 0 端末を開き、すぐ「mkdir cl20」(シー・エル・2・0) でサブディレクトリを作りなさい (このコマンドについては後でまた説明)。次に以下のことを順次やりなさい。

- 「pwd」で現在位置を確認。「ls」で cl20 の存在確認。
- 「cd cl20」でディレクトリを移動し再度「pwd」
- 上のコンプリーションの例を参考に、好きなファイルを現在位置に 1 つコピー。
- 「ls」で一覧表示しコピーしたファイルを確認。
- 「file ファイル名」でそのファイルの種別を確認。
- テキストなら「cat ファイル名」画像・動画なら「eog ファイル名」で内容表示し確認。
- あと 2〜3 個のファイルをコピーしてきてやってみる。

[2] cat、more、less はファイルを指定し忘れるとキーボードから読み込もうとして入力待ちになります。そうなったときに強制終了させるには Ctrl-C を打ってください。

5.2　ファイルとディレクトリの操作

5.2.1　echo と cat [ex]

ではいよいよ、ファイルやディレクトリの操作について取り上げます。その前に、ファイルを簡単に作る方法から説明しておきます。それには **echo** というコマンドを使います。

- echo 文字列… — 文字列を画面に表示

これだけでは画面に表示するのでファイルと関係ないですが、Unix ではコマンドの末尾に「>ファイル」という指定をすることで「画面に表示する代わりにファイルに保存」できるので、これでファイルを作れます。

```
...$ echo 'This is a pen.'
This is a pen.                ←指定した文字列が画面に
...$ echo 'This is a pen.' >t1 ←今度は画面に現れない
...$ cat t1                   ←ファイルの中を見ると
This is a pen.                ←その文字列が
```

上の方法だと 1 行のファイルしか作れません。2 行以上の場合には、cat を使うことができます。cat は先に「1 個以上のテキストファイルを次々に表示」と説明しましたが、ファイルを 1 個も指定しない場合は「キーボードからの入力を表示」になり、何行でも入力できます (入力を終わらせたいところで Ctrl-D を打ちます)。

```
...$ cat
This is a pen.  ←入力した行
This is a pen.  ←表示
That is a dog.  ←入力した行
That is a dog.  ←表示
^D              ←入力の終わりは Ctrl-D
...$
```

これと「>ファイル」を利用すると、次のように複数行のファイルが作れます。

```
...$ cat >t2
This is a pen.  ←入力した行
That is a dog.  ←入力した行
^D              ←入力の終わりは Ctrl-D
...$ cat t2     ←こんどは中身の表示に cat を使用
This is a pen.
That is a dog.
...$
```

画像ファイルもコマンドだけで作ることは可能ですが、それは今回は大変なので、画像ファイルを作りたい場合は gimp などを使ってください。

5.2.2　ファイルの操作 [ex]

次に cp、mv、ln、rm という 4 つのコマンドを説明します。

- **cp** ファイル 1 ファイル 2 — ファイル 1 の内容をファイル 2 にコピー (ファイル 2 が無ければ新たに作られる)。

- **mv** ファイル 1 ファイル 2 — ファイル 1 の名前をファイル 2 に変更する (パス名を指定することで別のディレクトリに移動できる)。
- **ln** ファイル 1 名前 2 — ファイル 1 の「別の名前」として名前 2 を追加する (パス名を指定することで別のディレクトリに名前を登録することもできる)
- **rm** ファイル — ファイルを消去する (正確には名前を 1 つ消去し、名前が 0 個だと本体も消える)

実際に先の続きからやってみましょう。

```
...$ cp t1 t3   ←コピーする
...$ cat t3     ←確かに同じ内容
This is a pen.
...$ mv t3 t4   ←名前変更
...$ cat t3     ←古い名前は…なくなっている
cat: t3: そのようなファイルやディレクトリはありません
...$ cat t4     ←新しい名前はある
This is a pen.
...$ ln t4 t5   ←新しい名前をつける
...$ cat t4     ←古い名前もそのまま使える
This is a pen.
...$ cat t5     ←新しい名前でも同じ内容
This is a pen.
...$ rm t4      ←古い名前を消す
...$ cat t4     ←確かになくなっている
cat: t4: そのようなファイルやディレクトリはありません
```

ファイルについては、`mv` は `ln` で別名をつけてから `rm` で古い名前を消すのとほぼ同等です。

演習 1 上のサンプルを実際に操作してみなさい。毎回「`ls`」等で状況を確認しながらおこなうこと。納得したら次のことをやってみなさい。

a. `echo` や `cat` でファイルを作ったときの「文字数」と「`ls -l`」でそのファイルを調べたときに表示されるファイルサイズ (バイト数) の関係がどうなっているか観察し検討しなさい。

b. 「**gimp** `&`」を実行して、適当な画像をお絵描きして、好きなファイル名 (ただし `.jpg` か `.png` で終わらせること) で保存しなさい。そのファイルサイズと元の絵の関係について観察し検討しなさい。

c. 「`cp`」でファイルをコピーしたとき、元のファイルとコピーとで同じこと、違うことを「`ls -l`」の表示で検討しなさい。

d. 「`ln`」でファイルに別名をつけるのと「`cp`」でコピーするのでどのように違うか (=別名かコピーかを区別する方法があるか) 検討しなさい。

5.2.3 ディレクトリの操作 ex

ディレクトリ作成/削除のコマンドも説明しておきます。また、前に説明した `mv` はディレクトリの名前を変更したりディレクトリの場所を変更するのにも使用できます。

- **mkdir** ディレクトリ — 新しいディレクトリを作る
- **rmdir** ディレクトリ — ディレクトリを消す (空っぽである必要がある)

- rm -r ディレクトリ — ディレクトリを消す (その下にファイルやサブディレクトリがあればそれらも消す)

そして、ファイルを操作するときに cp、mv、ln の 3 つのコマンドを使う場合、「行き先」としてディレクトリを指定すると、そのディレクトリの下に「元と同じ名前で」コピー、移動、名前作成を行なう効果になります。

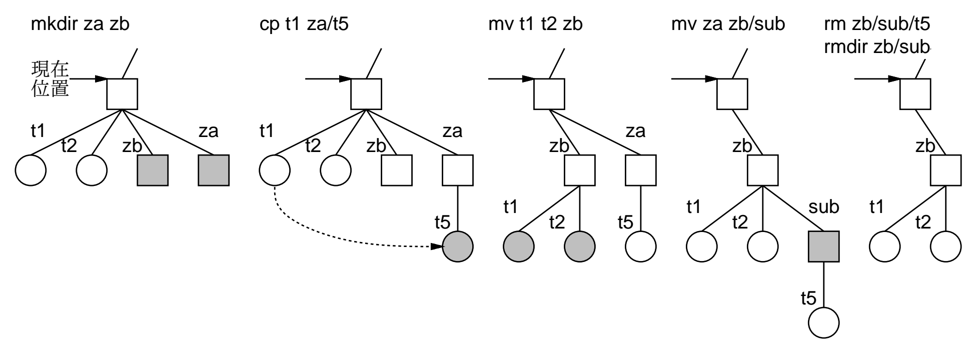


図 5.3: ディレクトリ操作の様子

これらを実際にやってみましょう (操作の様子を図 5.3 に示します)。

```
...$ mkdir za zb   ←ディレクトリを 2 つ作成
...$ cp t1 za/t5   ←ファイルを 1 つコピー (名前も変更)
...$ mv t1 t2 zb   ←ファイルを 2 つ移動 (名前は同じまま)
...$ mv za zb/sub  ←ディレクトリ za を zb の下に移動 (名前も変更)
...$ ls -F zb      ← zb の下のファイル一覧
sub/   t1    t2    ←ファイル 2 つとサブディレクトリがある
...$ ls -F zb/sub  ←サブディレクトリの下は
t5                 ← t5 だけ
...$ rm zb/sub/t5  ← t5 を消す
...$ rmdir zb/sub  ←ディレクトリを消す
```

演習 2 /class/jb/cl20/bin/get-files1 というコマンドを実行すると、自分のホームディレクトリに「zz」というサブディレクトリが作られ、その下にいろいろなファイルが置かれる。それらのファイルを調べて「自分がこれらを取っておくとしたらどのように整理するか」を検討して決め、そのように整理しなさい。どのような方針で整理することにしたかの説明と、実際に整理した様子が分かる表示をレポートに含めること。

5.3 ファイルの属性と保護設定

5.3.1 ファイルの各種属性 ex

ファイルには名前に加えて、さまざまな属性がついています。それらの属性を表示するのが、既に学んだ「ls -l」です。ここまでで、ファイルサイズと最終変更日時について学びましたが、そのほかに何があるでしょうか。

```
-rw-r--r--  1 ka002689 faculty   35  5月 17 13:45 test2.c
(mode)  (links) (user) (group) (size) (lastmod)   (name)
```

右から行くと、name(ファイル名) がまずあります。次に lastmod(最終変更日時)、size(バイト数)、ここまでは既知ですね。次の user、group というのは、ファイルの持ち主と所属グループで、保護のため使います。次の lins(リンク数) は「そのファイルがいくつ名前を持っているか」で、先の演習で利用したかも知れません。最後の mode(モード) は以下で。

5.3.2 ファイルの保護設定 ex

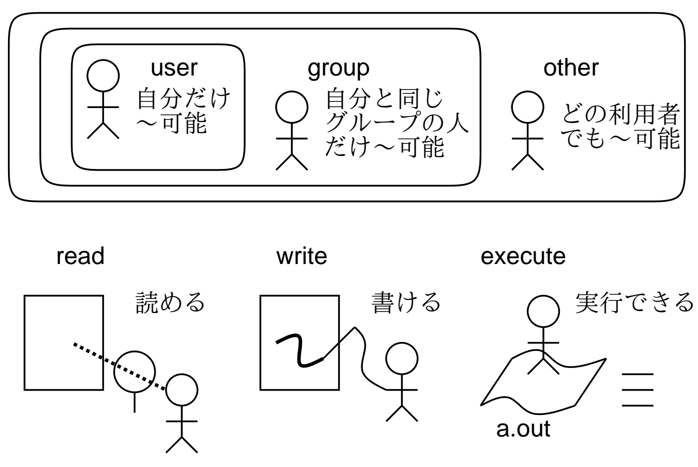


図 5.4: ファイルの保護設定

Unixでは、ファイルの保護設定を**モード** (mode) と呼び、各ファイルごとに「user(持ち主)」「group (グループメンバー)」「other (その他の人)」それぞれについて「read(読む)」「write(書く)」「execute(実行する)」の可否を各々設定できます (図 5.4)。モード情報は `ls -l` の表示の最初の部分に含まれます。

```
-rw-r--r-- 1 ka002689 faculty 30  6 22 16:12 2018 test1.txt
```

これは先の例ですが、持ち主 (ka002689) は読み、書きは可能だが実行は不可、それ以外の人には読みだけ可という設定を意味しています。実行というのは、実行可能な (マシン語のプログラムの入った) ファイルに対してのものなので、C 言語で「hello.」と打ち出すプログラムを作って試します。

```
...$ echo 'main() { puts("hello."); }' >test.c
          ↑ 1 行の C プログラム
...$ gcc test.c  ←コンパイル
...$ ls -l a.out ←ファイル a.out に実行形式ができる
-rwxr-xr-x 1 ka002689 faculty 6626  6月 22 16:18 2018 a.out
...$ ./a.out   ←「現在位置の a.out」の指定で実行
hello          ←プログラムの出力
```

Unixではコンパイラは「`a.out`」という決まった名前の実行形式ファイルを作るので、これのモードを見ると確かに実行可能になっています。次はまた別のファイルを見ます。

```
...$ ls -l t1
-rw-r--r-- 1 ka002689 faculty 15  6 24 15:56 2018 t1
```

ファイル `t1` は、持ち主 (ka002689) は読み書きともに可能ですが、グループ `faculty` の人、および他の人には読むことのみ可能です。モードの変更は、**chmod** コマンドで次のような形で指定します。

- `chmod` 対象 (`+|-`) 許可 ファイル … — モードを設定する

「対象」は `u`、`g`、`o` どれか 1 つ以上 (それぞれ持ち主、グループメンバ、その他の人を表す)、「許可」は `r`、`w`、`x` どれか 1 つ以上 (それぞれ読み、書き、実行を表す) で、「`+`」はその許可を出し、「`-`」はその許可を取り除きます。たとえばさっきのファイルでやってみましょう (`a` は全員という意味で `ugo` と同じです)。

```
...$ chmod a-rwx t1      ←全員に対して「R」「W」「X」を OFF
...$ ls -l t1
---------- 1 ka002689 faculty 15  6 24 15:56 2018 t1
...$ chmod u+rx t1       ←自分に対して「R」「X」を ON
...$ ls -l t1
-r-x------ 1 ka002689 faculty 15  6 24 15:56 2018 t1
...$ chmod go+x t1       ←グループ/他人に対して「X」を ON
...$ ls -l t1
-r-x--x--x 1 ka002689 faculty 15  6 24 15:56 2018 t1
```

chmod ではもう 1 つの方法として、rwx の 3 ビットを 1 桁の 8 進表現で指定することもできます。8 進表現と 2 進表現 (3 ビット) と保護設定の対応を表 5.1 に示しておきます。

表 5.1: 8 進表現による保護モードの指定

8 進	ビット	保護設定
0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

実際には User/Group/Other の順に 8 進表現 3 桁で指定をおこないます。

```
...$ chmod 750 t1    ← 7: rwx, 5: r-x, 0: ---
...$ ls -l t1
-rwxr-x--- 1 ka002689 faculty 15  6 24 15:56 2018 t1
...$ chmod 642 t1    ← 6: rw-, 4: r--, 2: -w-
...$ ls -l t1
-rw-r---w- 1 ka002689 faculty 15  6 24 15:56 2018 t1
```

このように、chmod コマンドを使うことで、さまざまな保護モードが自由に設定できます。

5.3.3　ディレクトリに対する保護設定 [ex]

ここまではファイルに対するモードを見て来ましたが、ディレクトリに対しても同じようにモードが設定されています。実際に見てみましょう。ディレクトリ自体の情報を表示させるには、ls にオプション「-d」を与える必要があります (そうしないと、そのディレクトリの「中にあるファイル」の情報を表示します)。

```
...$ ls -ld bin
drwxr-xr-x 2 ka002689 faculty 16  5 18 14:45 2018 bin
```

これまでずっと「-」だったモードの先頭が「d」になっていますが、これが「ディレクトリである」ことを表しています。そして、このディレクトリは「自分には読むことも書くことも実行もできる」「グループや他の人には読むことと実行だけできる」というモードになっています。

しかしディレクトリに対する読み/書き/実行とは、どういう意味でしょう? それは次のようになります (順番を入れ替えました)。

- 実行 (x) — そのディレクトリの下にあるファイルやディレクトリを (パス名で指定して) アクセスできる。このモードが OFF だと、そこにあるファイルは対象の人 (一般/グループ/自分) がアクセスすることは一切ないので保護という点では安全になる (もちろん、自分がアクセスしたいときはディレクトリのモードを変更する)。
- 読み (r) — そのディレクトリの一覧を ls コマンドなどで表示できる。x モードが ON でも r モードが OFF だと、ファイル名を直接知っている人しかその中のファイルにアクセスできないので、知っている人だけにアクセスさせるという効果がある。(中のファイルを読める、書ける等はそのファイル自体のモードによる。)
- 書き (w) — ディレクトリの変更とは、ディレクトリに新しいファイルを追加したり、そこにあるファイルを削除すること。このため、このモードが OFF だと、ファイルを追加したり削除したりは不可能になる (中のファイルの読める、書ける等はそのファイル自体のモードによる)。

ディレクトリ自体に対する保護設定の変更は、ファイルと同様にして `chmod` コマンドで行えます。

```
...$ chmod go-rx bin
...$ ls -ld bin
drwx------ 2 ka002689 faculty 16  5 18 14:45 2018 bin
```

演習 3 自分の持っているファイルを `ls -l` で調べ、保護モードを確認しなさい。その後、以下の課題をやってみなさい。

a. 自分のファイルのモードを変更し、「読めない」ものの内容を (`cat` などで) 表示させようとしたり、「書けない」ものに (`cp` で内容をコピーするなどで) 書こうとした場合にどうなるか観察しなさい。「誰にでも読めるが自分には読めない」などの矛盾した設定の効果も調べるとなおよい。
b. 自分のディレクトリについて、その保護モードを変更するとどのような効果があるか探究してみなさい。通常は自分に対して「rwx」になっているはずだが、r/w/x をそれぞれ OFF にするとどのような効果があるか調べること。なお、`ls` でディレクトリ自体の情報を表示させるには「`ls -ld` ディレクトリ」のように「`-d`」を指定する必要がある。
c. (複数人で協力してやる課題) 自分のホームディレクトリの下にサブディレクトリを作り「グループメンバーや他人に書ける」モードにしてから、他人にそこにファイルを作ってみてもらいなさい。そのファイルのモードやディレクトリのモードをさまざまに変更して、どういう設定だと何ができて何はできないかを調べなさい。
d. `/class/jb/cl20/bin/get-files2` というコマンドを実行すると、自分のホームディレクトリに「yy」というサブディレクトリが作られ、その下にいろいろなファイルが置かれる。今度はファイルやディレクトリにさまざまな保護設定がなされていて、そのままでは内容が調べられないものもある。中にどのようなものがあるか確認して報告しなさい。

課題 5A

今回の課題は「演習 1」「演習 2」「演習 3」に含まれる小問 (合計で 9 個) の中から 1 つ以上を選択し、結果をレポートとして報告して頂くことです (直接打ち込むとログイン時間切れになった時に内容が消失するので、メモ帳など他のソフトで作成して完成後にコピーすることを勧めます)。以下の内容がこの順に含まれるようにしてください。

- 冒頭に題名「コンピュータリテラシレポート＃ 5」、学籍番号、氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も別途書く。)
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。

- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 以下のアンケートに対する回答。

  Q1. ファイルシステムやファイルの属性についてどれくらい知っていましたか。新たに知って面白かったことは何ですか。

  Q2. `ls` によるファイルの表示やその他のコマンドによるファイルの操作を (日付や保護モードなどの参照も含めて) GUI による操作と比較してどう思いますか。

  Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

なお、課題はグループでやって構いません。その場合も、(メンバー氏名を明記した上で) レポートは必ず各自で執筆してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。

＃6　テキストファイルとエディタ

今回の目標は次の通りです。

- コンピュータ内部でのテキストの表現について学ぶ — 実験データなどもコンピュータ上ではテキストファイルなのでその原理を知っておくと有益です。
- 日本語の複数の文字コードとその問題について学ぶ — 日本語の文字化けトラブルは多いので文字コードの知識は必須です。
- テキストエディタ Emacs の多様な機能について学ぶ — 実際にプログラムやレポートを書くのに使う道具なので有効に使えるようになっておくことは価値があります。

6.1　テキストエディタ ex

ここまで、サンプルファイルを作るのには echo を使ったりしてきましたが、実際に Unix で作業をする場合はあまりそうはしません。まず、ファイルには人間が読める文字が入ったテキストファイル (text file) と、それ以外の (任意のビット列が入った) バイナリファイル (binary file) とがあります。そして、テキストファイルを作成したり修正するためのプログラムのことをテキストエディタ (text editor) と呼びます。

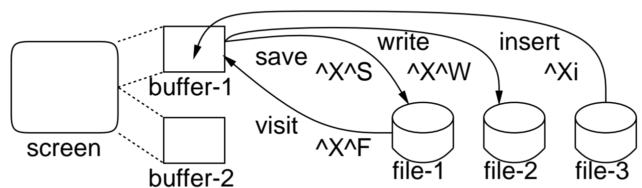


図 6.1: Emacs の編集モデル

ここでは Unix で広く使われている Emacs というテキストエディタについて、その基本部分を取り上げます (後の節でもっと細かいことも扱います)。Emacs の基本的なモデルを図 6.1 に示します。ファイルを作ったり修正するのは、すべてバッファに持って来て行います。Emacs は多数のバッファを備えていて複数のファイルを同時に編集できます (今回は簡単のためバッファを 1 つしか使いませんが)。バッファの一部 (編集している付近) が画面に常時見えています (このあたりは Word などと同じですが、実際には Word より Emacs の方が先輩です)。

Emacs の特徴の 1 つは、コマンドがすべて Ctrl-□や Alt-□ (Ctrl キーや Alt キーを押しながらどれかのキーを打つ) で始まることです。なぜかというと、エディタで一番多くやるのは「普通の文字を打ち込む」ことなので、文字キーをそのまま打ったら「その文字が入る」ことにしたからです。

Word も文字キーを打ったらそのまま入るのでそこは同じですが、ただし Word ではさまざまな機能はメニューやボタンで指示しますね? それをやると、マウス操作が必要になって時間が掛かり効率が落ちます。そこで Emacs では上述のように Ctrl-□や Alt-□を使うのです (Word でもよく使うコマンドはキーボードショートカットとして同様になっていますが、Emacs は「キーボードショートカットしかない」わけです)。なお、以下では「Ctrl-□」を「^□」と表記します。また、Alt キーがない環境もありますが、Alt-□は常に「ESC キーを打ってから□のキーを打つ」ことで代替できます。

さて、再び図 6.1 を見てください。ファイルの読み書き関係その他のコマンドを説明します。

- `^X1` — 画面が縦 2 分割されていて狭い時 1 分割に変更。
- `^X^F` ファイル名 `[RET]` (Find File) — ファイルの内容をバッファに読み込んで編集に備える。そのファイルがまだ無い場合は空っぽのバッファからはじまる。または「`emacs ファイル名 &`」でファイルを指定して起動しても同じこと。
- `^X^S` (Save File) — バッファからファイルに書き戻す (まだ無いファイルを指定していたならこのとき作られる)。
- `^X^W` ファイル名 `[RET]` (Write File) — バッファの内容を別の (ここで名前を指定している) ファイルに書き出す。
- `^Xi` ファイル名 `[RET]` (Insert File) — 指定したファイルの内容をカーソル位置に挿入する。このコマンドでは 2 文字目が Ctrl のない「ただの i」なのに注意。
- `^X^C` (Kill Emacs) — Emacs を終了する。

Emacs を起動してみましょう。それには編集したいファイルを指定してコマンド「`emacs ファイル &`」を実行します (最後の「&」を忘れると、Emacs が終るまで次のコマンドが打てません)。すると新しい窓が現れ、指定したファイルを編集できる状態になっています。[1] 画面の一番下の行は**ミニバッファ**と呼び、ファイル名入力などのときに使われます。そしてその 1 つ上の行を**モードライン**と呼び、ここにはファイル名や文字コードなどが常時表示されています。

編集コマンドは今回は最低限の説明ですが、矢印キー、`[BS]` キー、文字キーだけでも編集はできます (ただし矢印キーより Ctrl-□の方が高速です)。最後は上記の「`^X^S`」でファイルを保存します。

- 通常の文字キー (self insert) — その文字がカーソル位置に挿入される。
- `^N`, ↓ (Next) — カーソルを 1 行下へ。
- `^P`, ↑ (Previous) — カーソルを 1 行上へ。
- `^F`, → (Forward) — カーソルを 1 文字右へ。
- `^B`, ← (Backward) — カーソルを 1 文字左へ。
- `^D`, `[DEL]` (Delete) — カーソル位置の文字を消す。
- `[BS]` — カーソル位置の直前の文字を消す。
- `^\` (キーボードによっては`^¥`) — 英語/日本語切替え。

演習 0a これまでに作ったテキストファイルのどれかを指定して「`emacs ファイル名 &`」で Emacs を起動し、編集を開始しなさい。ファイルの内容を任意に変更し、Unix コマンドで内容やファイルの長さの変化を確認しなさい。

6.2 テキストファイル

6.2.1 テキストファイルと文字コード ex

テキストファイル (text file) とは、人間に読める文字を内容とするファイルのことでした。しかしコンピュータ上の情報はすべてビット列で表現されているわけですから、「文字を内容とする」というのはどういうことか、もう少し考えて見ましょう。

文字の種類は有限ですから、その文字の一覧表を作り、それぞれの文字に重複がないように番号を割り当てます。この番号を**文字コード** (character code) と呼びます。次に、実際にファイルに文字を入れるときに、その文字コードをビット列でどのように表現するかを決めます。これをその文字の**符号化**ないし**エンコーディング** (encoding) と呼びます。あるひとまとまりの文字群 (英語の文字と

[1] 逆に、X Window が使えない状態では「&」なしで起動しないと動作しません。ややこしいですが、そうなっています。

か日本語の文字とか) について文字コードと符号化を併せたものを**文字コード体系** (character coding system) と呼びます。

コンピュータの初期には、扱える文字の種類が限られていたので、文字コードと符号化の区別をする必要がありませんでした。このころ作られ、現在でも使われている文字コード体系に **ASCII コード**があります (図 6.2)。これは英数字・記号と制御文字から成ります。

ASCII の各文字は 7 ビットで表せますが、ファイルはバイト (8 ビット) 単位で扱うので、「0」のビットを追加し、1 バイトに 1 文字を入れます。これが ASCII のテキストファイルです。テキストファイルとは、何らかの文字コード体系のビット列が入っているファイルなのです。

<table>
<tr><td colspan="2">上位 3ビット → / 下位 4ビット ↓</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>１６進</td></tr>
<tr><td></td><td></td><td>000</td><td>001</td><td>010</td><td>011</td><td>100</td><td>101</td><td>110</td><td>111</td><td>二進</td></tr>
<tr><td>0</td><td>0000</td><td></td><td></td><td>SP</td><td>0</td><td>@</td><td>P</td><td>‘</td><td>p</td><td></td></tr>
<tr><td>1</td><td>0001</td><td></td><td></td><td>!</td><td>1</td><td>A</td><td>Q</td><td>a</td><td>q</td><td></td></tr>
<tr><td>2</td><td>0010</td><td></td><td></td><td>"</td><td>2</td><td>B</td><td>R</td><td>b</td><td>r</td><td></td></tr>
<tr><td>3</td><td>0011</td><td></td><td></td><td>#</td><td>3</td><td>C</td><td>S</td><td>c</td><td>s</td><td></td></tr>
<tr><td>4</td><td>0100</td><td></td><td></td><td>$</td><td>4</td><td>D</td><td>T</td><td>d</td><td>t</td><td></td></tr>
<tr><td>5</td><td>0101</td><td></td><td></td><td>%</td><td>5</td><td>E</td><td>U</td><td>e</td><td>u</td><td></td></tr>
<tr><td>6</td><td>0110</td><td></td><td></td><td>&</td><td>6</td><td>F</td><td>V</td><td>f</td><td>v</td><td></td></tr>
<tr><td>7</td><td>0111</td><td></td><td></td><td>'</td><td>7</td><td>G</td><td>W</td><td>g</td><td>w</td><td></td></tr>
<tr><td>8</td><td>1000</td><td></td><td></td><td>(</td><td>8</td><td>H</td><td>X</td><td>h</td><td>x</td><td></td></tr>
<tr><td>9</td><td>1001</td><td>TAB</td><td></td><td>)</td><td>9</td><td>I</td><td>Y</td><td>i</td><td>y</td><td></td></tr>
<tr><td>A</td><td>1010</td><td>NL</td><td></td><td>*</td><td>:</td><td>J</td><td>Z</td><td>j</td><td>z</td><td></td></tr>
<tr><td>B</td><td>1011</td><td></td><td>ESC</td><td>+</td><td>;</td><td>K</td><td>[</td><td>k</td><td>{</td><td></td></tr>
<tr><td>C</td><td>1100</td><td>FF</td><td></td><td>,</td><td><</td><td>L</td><td>\</td><td>l</td><td>|</td><td></td></tr>
<tr><td>D</td><td>1101</td><td>CR</td><td></td><td>-</td><td>=</td><td>M</td><td>]</td><td>m</td><td>}</td><td></td></tr>
<tr><td>E</td><td>1110</td><td></td><td></td><td>.</td><td>></td><td>N</td><td>^</td><td>n</td><td>~</td><td></td></tr>
<tr><td>F</td><td>1111</td><td></td><td></td><td>/</td><td>?</td><td>O</td><td>_</td><td>o</td><td>DEL</td><td></td></tr>
<tr><td>１６進</td><td>二進</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

図 6.2: ASCII コードの表

実際にファイルにどのようなビット列が入っているかを調べるには「**od -t x1 ファイル**」というコマンドが利用できます。やってみましょう。

```
...$ echo 'This is a pen.' >test.txt
...$ cat test.txt
This is a pen.
...$ od -t x1 test.txt
0000000 54 68 69 73 20 69 73 20 61 20 70 65 6e 2e 0a
0000017 ←ここの表示はファイルの長さ (ただし 8 進表記)
```

図 6.2 と照合すると、確かに先頭の「54」は「T」ですし、以下同様で、最後の方の「2e」は「.」です。一番最後の「0a」は改行文字ですね。

演習 0b 各自が英数字 15 文字までの 1 行のファイルを作り、3〜4 名のグループで互いに「`od -x t1` ファイル」の結果を見せて内容を (紙に書いて) 当てるまでの時間を競う競技を開催してみなさい。最も解読に時間が掛かる問題を出した人の優勝とする。[2]

6.2.2 日本語の文字コード ex

さて、英字は分かったとして、では日本語の文字はどうでしょうか。その前に少し歴史的な話をします。ASCII では前述のように左端のビットは「0」ですから、そこを「1」にした場合に、1 文字を 8

[2] ヒント: 記号を沢山入れたり、普通の英文に見えてわざと違うスペルを入れるなどの技があると強いかもしれない。

ビットで表すというところは変えないままで、ASCII とは別の文字を最大 $2^7 = 128$ 種類、追加する余地があります。ヨーロッパ各国の言語は英語にない文字 (や、英語の文字にアクセント記号を追加した文字) があるので、それをこの部分に追加した文字コードを使用しました。

日本では、漢字を表示できる画面や打ち出せるプリンタが当初なかったので、カタカナの 50 音と「゛」「゜」をこの部分に追加した文字コードを使用しました。これを「**8 ビットカナ**」や (俗称ですが)「半角カナ」と呼びます。今でも幅が英字と同程度に狭くて濁点・半濁点が別の文字になっているものを見掛けますが、これが 8 ビットカナ文字です。

やがて漢字の表示や印刷ができるようになると、こんどは漢字を含めた日本語文字コード体系が必要になり、**日本工業規格** (Japan Industrial Standard、JIS) として規格 **JIS C6226** が定められました (現在では改訂されて **JIS X0208** となっています)。これを **JIS 漢字コード**と呼びます。

漢字は 256 よりはるかに個数が多いため、日本語の 1 文字は 2 バイトで表す必要があります。そこでこの規格では、1～94 の「区番号」と 1～94 の「点番号」を組み合わせた「**区点番号**」で 1 つの文字を表すようになっています (区番号が 1 バイト目、点番号が 2 バイト目に対応)。なぜ 94 かというと、ASCII の図形文字 (制御文字でない、目に見える文字) が「！」～「～」までの 94 あるので、それに対応させるという意図があったためです (図 6.2 を見てください)。

6.2.3 日本語のエンコーディング ex

JIS 漢字コードを実際にファイルに格納するときには、ASCII の文字と混ぜて使用することが必要とされました。そこで次の 3 つのエンコーディングが開発され使用されるようになりました。

- **iso-2022-jp**(いわゆる JIS) — ISO-2022 というのは制御文字 ESC で始まる 3 バイトでさまざまな文字コードを切替えるという規格であり、それを用いて「ESC-$-B」で JIS2 バイトコード、「ESC-(-B」で ASCII コードに切替える。
- **euc-jp**(日本語 EUC) — 一番左のビットが 0 のものは ASCII、1 のものは 2 バイトずつ JIS2 バイトコード (の最左端のビットを 1 にしたもの) とする。
- **shift-jis**(シフト JIS) — 日本語 EUC に類似しているが、8 ビットカナと共存させるようにずらしたもの。このため 2 バイト文字の 2 バイト目に最左端のビットが 0 のものもある。

iso-2022-jp は初期の電子メールなど 8 ビット文字が使えない場面で、euc-jp は Unix 系のシステムで、shift-jis は PC 系 (Windows、Mac) で使われてきました。さらに面倒なことに、行末の文字も Unix 系では LF(0a)、Windows 系では CR+LF(0d0a) の 2 バイト、Mac では CR(0d) が使われて来ました。そして今は UTF8(後述) が多くのシステムでも使われるようになりました。改行文字も Mac では OS-X 以降、Unix と同じになってきています。つまり、日本語の文字コード (とエンコーディング) は大変混乱しているというのが現状です。

6.2.4 UNICODE と UTF8 ex

以前は、複数の国の文字を扱うとき、ISO-2022 規格に従い、言語ごとに ESC で始まる 3 バイトで切替えていく、という方法しかありませんでした。しかしこれではコンピュータによる処理が面倒なため、「すべての国のすべての文字に 16 ビットの文字コードを割り当てて扱おう」という考え方が現れました。これが **UNICODE** で、この考え方に基づく国際規格 **ISO 10646** が制定されています。[3]

UNICODE における符号化には複数の方式があり、そのうちで、1 バイト (8 ビット) ずつの処理を前提としたものが **UTF8** です。UTF8 は、00～7F の範囲は ASCII と完全に一致しており、さらにすべての国の文字をまとめて扱えるため使いやすく、その使用が広まっています。UNICODE の文字コードの範囲は 21 ビットで表現でき、000000～1FFFFF までの範囲になります。そしてそれを UTF8 で符号化すると 1 文字が 1～4 バイトで表現されます (表 6.1)。

[3]ただし実際にやってみるとすべての文字を 16 ビットのコードに収録するのは無理だとわかり、現在では一部の文字は 16 ビットを 2 つつなげた値として表現しています。

表 6.1: UTF8 による符号化の形式

UNICODE の値	bits	ビット列 (UTF8 による符号化)	bytes
00〜7F	7	`0xxxxxxx`	1
80〜07FF	11	`110yyyxx 10xxxxxx`	2
0800〜FFFF	16	`1110yyyy 10yyyyxx 10xxxxxx`	3
10000〜1FFFFF	21	`11110zzz 10zzyyyy 10yyyyxx 10xxxxxx`	4

6.2.5 文字コードの変換

上記のように、日本語文字の入ったファイルは iso-2022-jp、euc-jp、shift-jis、UTF8 と 4 種類あり、それぞれ使われているので、それらの間で変換する必要が生じることがあります。ここでは (1) コマンド **nkf** を使う方法と、(2)Emacs の操作によるものの 2 種類を紹介します。

まず `nkf` の方は使い方は次のように簡単です。

```
nkf -j 入力ファイル >出力ファイル     ← iso-2022-jp に変換
nkf -e 入力ファイル >出力ファイル     ← euc-jp に変換
nkf -s 入力ファイル >出力ファイル     ← shift-jis に変換
nkf -w 入力ファイル >出力ファイル     ← UTF8 に変換
```

出力側のコードしか指定していませんが、入力側はファイル内容から自動判別されます。ただし、自動判別に失敗することがありますので、そのときは入力側のコードも指定してください。それには「出力指定の小文字を大文字にしたもの」をくっつけて指定します。たとえば、入力が UTF8、出力が euc-jp だと「`nkf -We t1 >t2`」などのようにします。

次に Emacs ですが、以下では読みやすさのため、Ctrl-X (`^X`) の代わりに C-x のように記します。M-x は Meta-X ですが、常に [ESC]x でも同等に使えることに注意。Emacs は次の 3 つについてそれぞれ別個に「どの文字コード、どのような改行規則」を設定できます。

f. ファイル読み書き — ファイルを読む/書く時の文字コード

t. 画面 — 画面表示をおこなうときの文字コード

k. キー — キー入力をおこなうときの文字コード

Emacs を no window モードで使うときにはこれら 3 つともが問題になりますが、GUI で直接窓を開いているときは、画面表示もキーの入力も Emacs が直接おこなうので、問題になるのはファイル読み書きだけになります。[4]

- C-x [RET] f コード [RET] — ファイル読み書きのコード設定
- C-x [RET] t コード [RET] — 端末画面のコード設定
- C-x [RET] k コード [RET] — キー入力のコード設定

設定できるコードは「`iso-2022-jp`」「`euc-jp`」「`shift_jis`(これだけ語のつなぎが下線!)」「`utf-8`」のいずれかで、さらに改行コードを指定する場合にはその直後に「`-unix`」(LF)、「`-dos`」(CR+LF)、「`-mac`」(CR)」のいずれかを指定します (指定しない場合はこれまでと同じになる)。これらの指定時には `[SP]` キーで**コンプリーション**が使えます。

演習 1 日本語のひらがな「あいうえお」をファイルに書き込み、「`od -t x1 ファイル`」で表示し、どのような規則性があるかを検討しなさい。続いて以下のことをやってみなさい。いずれも表を作るだけでなくどうなっているか説明すること。

[4]ただし GUI であっても、Emacs の画面からコピーペーストするときは、コピーペーストする先の文字コードに合わせる必要があります。

a. コード系を変更しない状態 (UTF8) でのひらがなのビット表現一覧表を作る。

b. UTF8 以外の文字コードでのひらがなのビット表現一覧表を作る。UTF8 とどのように違っているかを検討すること。

c. 文字コードを 1 つ決め、ひらがな、カタカナ、英字、数字、漢字などの字種ごとの文字コードがどうなっているか検討する。

演習 2 日本語の文章を iso-2022-jp でファイルに書いてから「`tr -d '\033' <入力 >出力`」で変換し、文字化けしていることを確認する。できたら、次の課題から 1 つ以上やってみなさい。

a. この化けたファイルから日本語として読めるファイルを復元してみなさい。[5][6]

b. 単純に復元するかわりに、ひらがなをカタカナに、カタカナをひらがなに入れ換えてから復元してみなさい。[7]

c. (チャレンジ問題)1 バイト文字 (いわゆる半角) と 2 バイト文字 (いわゆる全角) に同じ文字があるものがいくつかある。そのどちらを使うかは自分の好みだが、自分の好みでないファイルを好みであるように変換する方法を検討しなさい。

6.3 Emacs の詳細

6.3.1 ファイル読み書き等

先に Emacs の基本を説明した際ファイル関係で学んだものに、C-x C-f (ファイルを編集)、C-x C-s (保存)、C-x C-w (名前をつけて保存)、C-x i (ファイルを挿入)、C-x C-c (終了) がありました。

一般に複雑なコマンドを途中で中止するのは C-g (中止) でできます。また、実行し終ったばかりのコマンドの効果を取り消すのには C-x u (アンドゥ) が使えます (取り消せない動作もありますが)。

6.3.2 カーソル移動

C-p C-n C-f C-b (上下左右に 1 文字) は以前学びましたが、左右については 1 単語ずつ移動する M-f M-b も使えます。また行頭/行末に移動するのに C-a C-e が使え、文の先頭/末尾に移動するのに M-a M-e が使え、バッファの先頭/末尾に移動するのに M-< M-> が使えます。表示の制御として、C-l によりカーソルのある行が画面中央になるようスクロールし、1 画面進む/戻るのに C-v M-v が使えます。

現在いるのが何行目かはモードラインに表示されていますが、特定行にカーソルを移動するには「M-x goto-line [RET] 行数 [RET]」でできます。コンプリーションは `[SP]` なので、「M-x got[SP]l[SP][RET]」のように短く打てます。また、一般にコマンドには「C-u 数値」で数値プレフィクスをつけることができ、多くの移動コマンドではこれは反復数を意味します。たとえば「C-u 5 C-n」で 5 行下に行けます。

6.3.3 日本語入力

日本語入力モードは C-\ または C-￥ で ON/OFF できます。ON のときは打ち込んだものはローマ字かな変換されるので、入力したい単語の読みが打ち込めたら [SP] で漢字変換します。漢字変換中は次の候補に進む/前の候補に戻るのに C-n C-p が使えます。正しい漢字になったら [RET] によって確定します。単語によっては変換する単位が (文節) 複数に分かれることがあります。このとき、文節の区切りが正しくない場合は C-i C-o で文節を 1 文字ずつ伸ばす/縮めることができます。現在の文節 OK で先の文節に進むのは C-f また前の文節に戻るのは C-b です (図 6.3)。

---

[5]Emacs では改行文字や ESC 文字は「C-q C-j」、ESC は「C-q C-[」で入力できます。

[6]ヒント: ASCII と日本語の切替えは ESC-$-B、ESC-(-B であったところを ESC を削除したために文字化けしている。ファイルに現れない文字 X を 1 つ選び、`sed` で$-B や (-B を X-$-B や X-(-B に置き換えてから `tr` でこの X を `'\033'` に置き換えれば戻るはずである (が、間違った置き換えが起きる可能性もあるかも)。

[7]ヒント: `tr` でひらがなの 1 バイト目をカタカナの 1 バイト目に、またカタカナの 1 バイト目をひらがなの 1 バイト目に置き換えればよい。ただし、2 バイト目にそれらの文字が出て来るのにも対応するとしたら少しやっかいかも。

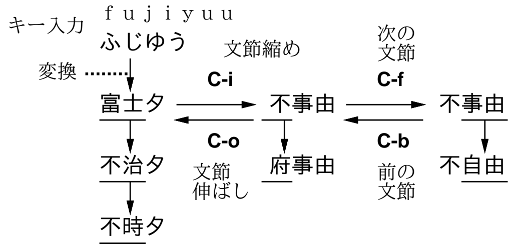


図 6.3: 文節伸ばし/縮めと文節移動の使い方

6.3.4 削除とコピーペースト

1 文字消すのに [BS] [DEL] C-d がありましたが、大量に消すのは大変です。そのときは、C-k が「行末まで消す、既に行末なら改行を消す」なので、これを連打することで行単位で消していけます。

Emacs では**キルリング** (kill ring) と呼ばれる見えないバッファがあり、ここにテキストを一次保存して別の位置に移したりします。C-k の連打で消したテキストは複数の連打の (他の操作が間に入らない) 範囲でまとまってキルリングに入ります (図 6.4 上)。

Word などで沢山消すには消す範囲をマウスでドラグして塗りますが、Emacs では (1) カーソルの現在ある位置を「**マーク**」し、(2) 次にカーソルを任意の場所まで移動することでマークとの間が範囲として指定できます。この範囲のことを**リージョン** (region) と呼びます。マークを設定するコマンドは正規には C-[SP] ですが、演習室の環境ではこのキーが漢字変換に割り当てられているので代わりに C-@を使う必要があります。リージョンの範囲を確認するのには、マークの位置とカーソルの位置を交換するコマンド C-x C-x が使えます。

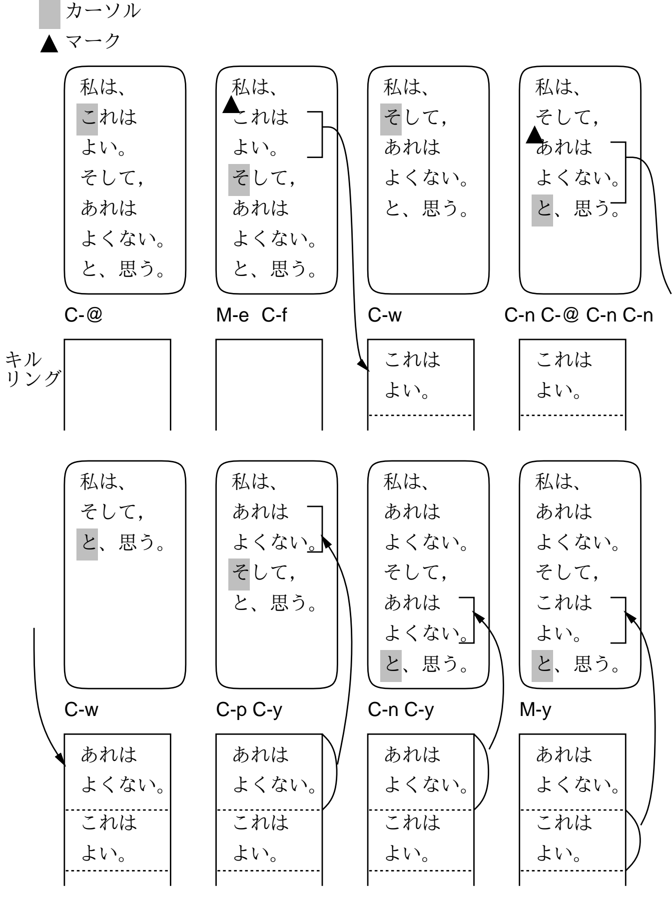


図 6.4: マーク・リージョンとキルリング

リージョンの内容は C-w で削除されキルリングに入ります。また M-w ではリージョンは削除されずにその内容がキルリングにコピーされます。C-y はキルリングの先頭 (最後に消したりコピーした範囲) がカーソル位置に挿入されるので、これを用いてテキストを移動できます。そして、C-y に続いての M-y は、いま挿入されたものをリングの「1 つ前に」あった範囲に置き換えます (図 6.4 下)。M-y を連続して使うことで、さらに 2 つ前、3 つ前…とできます。

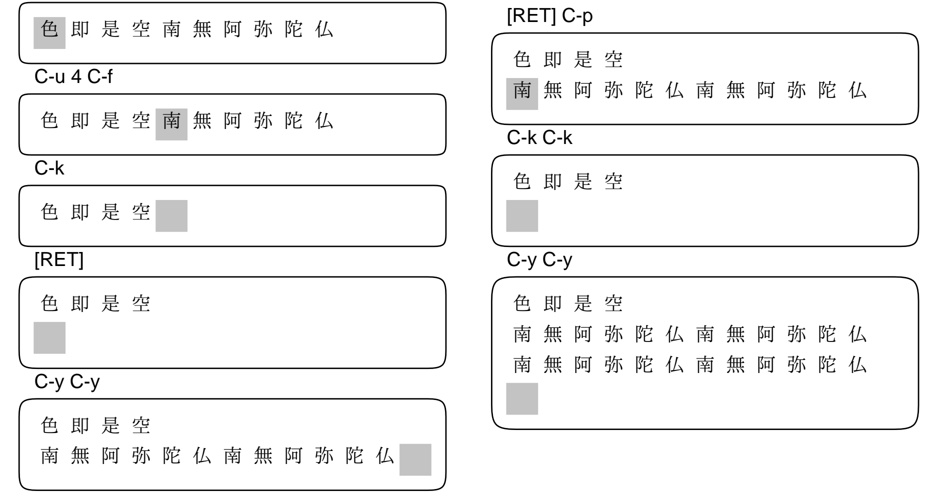


図 6.5: カット＆ペースト

なんだか非常に面倒そうですが、普段よく使うのは C-k の方なのでその使用例を図 6.5 に示しました。左側ではまず行の途中までカーソルを移動しそこで C-k を打つと行末までがカットされ、カーソル位置が行末になります。[RET] を打つと改行が挿入され、そこで C-y を 2 回打つと、先にカットした内容が 2 回挿入されます。

右側では再度 [RET] で改行を挿入し、1 つ上の行頭に入って C-k を 2 回打つと、今度は行末まで+改行がカットされ、今度 C-y を 2 回打つと行末まで含めたものが 2 回挿入されるので、2 行が挿入できます。

6.3.5　キーボードマクロ

Emacs を使っていると「同じ打鍵列を何回も使いたい」と思うことがあります。そのようなときは「C-x (……任意の打鍵…… C-x)」のように「**キーボードマクロ**開始コマンド」「キーボードマクロ終了コマンド」で囲むことで、間の打鍵を「覚えて」、そのあと「C-u 数値 C-x e」により指定した回数それを実行させられます (1 回でよければ「C-x e」)。「任意の打鍵」の中には通常の文字もコマンドも入れられるので、工夫すると色々変わったことができます。

たとえば、ある行の中の 20 文字に 1 文字おきに「.」を入れたいとすると、まず「C-x (. C-f C-x)」でマクロを定義し、「C-u 20 C-x e」で実行すればできます (やってみましょう)。

6.3.6　探索と置換

Emacs で英語テキストを扱っているときは**イクリメンタルサーチ** (incremental search) 機能が使えます。これは前方向なら C-s 後方向なら C-r で開始し、その後文字を打つごとにそこまでの文字のある位置にカーソルが移動します (そして入力を [BS] で取り消すとカーソルもそのぶんだけ前の位置に戻ります — 図 6.6)。めざすものが見つかった時は [ESC] で終了できます。せっかく便利なのですが、日本語の場合は漢字変換と一緒に使えないので、もっと原始的な M-x search-forward [RET] 文字列 [RET] や M-x search-backward [RET] 文字列 [RET] を使ってください。

文字列の置換には M-x replace-string [RET] 文字列$_1$ [RET] 文字列$_2$ [RET] が使えます。ただしこれは一気に全部置換してしまうので、M-x query-replace [RET] 文字列$_1$ [RET] 文字列$_2$ [RET] が使いやすいかも知れません。こちらは 文字列$_1$ が見つかるごとに止まってどうするか聞いてくるので、

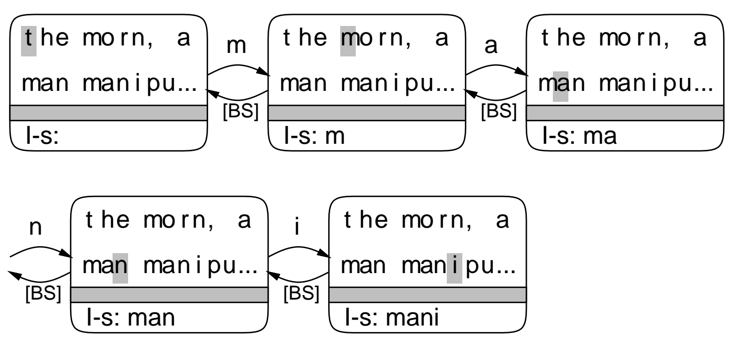


図 6.6: インクリメンタルサーチ

[SP] または y で「置換して次の場所」、[BS] または n で「置換せずに次の場所」、q で「置換せず終了」、. で「現在位置を置換して終了」、! で「残りを全部置換」、^ で「誤って置換したので 1 つ前に戻る」のいずれかが行えます。

上の 2 つはいずれも文字列を探索して置換しますが、文字列の代わりに正規表現を探索するコマンド M-x replace-regexp [RET] パターン [RET] 文字列 [RET]、M-x query-replace-regexp [RET] パターン [RET] 文字列 [RET] も使えます。これらではマッチした範囲の一部を\(...\) で囲み、置き換え文字列側で\1 等で参照する機能が使えます。

文字列やパターンの入力字に制御文字などはそのままでは入れられませんが、C-q を前置することで入れられます。たとえば改行文字はコントロール記法だと C-j になるので、改行をすべて削除するのは M-x replace-string [RET] C-q C-j [RET] [RET] でできます。

6.3.7 窓・フレーム・バッファの操作

Emacs では C-x 2 と C-x 3 で画面を水平/垂直に 2 分割できます (そのそれぞれの部分を窓と呼びます)。その状態でカーソルを移動すると同じバッファの違う場所を見ながら編集することができます。カーソルを他の窓に切替えるのは C-x o でできます。現在いる窓を少し大きくしたい場合は C-x ^ でできます。最後に、C-x 1 と C-x 0 でカーソルの無い側/ある側の窓を消すことができます (図 6.7)。

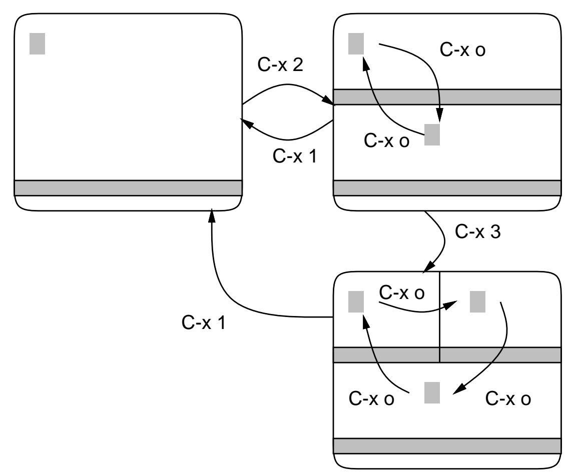


図 6.7: Emacs の窓分割機能

GUI の環境では、ウィンドウ (Emacs の用語では**フレーム**) を増やすこともできます。C-x 52 で新しいフレームを作ることができ、C-x 50 で現在カーソルのあるフレームを閉じます。

前に説明したように、Emacs では内部に複数の**バッファ**を持ち、それぞれで別のファイルを編集できます (そのほか Emacs の作業領域としても使ったりしています)。C-x C-f でファイルを指定すると、そのファイル名と同名のバッファができて、それが現在の窓に表示されます。C-x C-b で現在

存在しているバッファの一覧が見られます。そして C-x b バッファ名 [RET] で現在の窓を指定したバッファに切替えられます。

演習 3 Emacs で以下のことをやってみなさい。

a. 次の模様 (?) を Emacs でファイルに入力しなさい (横に並べる必要はない)。打鍵は少ない方がよい。どのように工夫して入力したかも説明すること。

```
目目目目目　田田田田田田田　縞縞縞縞縞縞縞縞
目　　　目　田　　田　　田
目目目目目　田　　田　　田　縞縞縞縞縞縞縞縞
目　　　目　田田田田田田田
目目目目目　田　　田　　田　縞縞縞縞縞縞縞縞
目　　　目　田　　田　　田
目目目目目　田田田田田田田　縞縞縞縞縞縞縞縞
```

b. 次の模様 (?) を Emacs でファイルに入力しなさい (横に並べる必要はない)。打鍵は少ない方がよい。どのように工夫して入力したかも説明すること (キーボドマクロはたぶん必要)。

```
.AAAAAAA   BBBBBBB+   CCCCCCCC
A.AAAAAA   BBBBBB-B   CCCCCCC
AA.AAAAA   BBBBB+BB   CCCCCC
AAA.AAAA   BBBB-BBB   CCCCC
AAAA.AAA   BBB+BBBB   CCCC
AAAAA.AA   BB-BBBBB   CCC
AAAAAA.A   B+BBBBBB   CC
AAAAAAA.   -BBBBBBB   C
```

c. (自由課題) Emacs の置換機能の「面白くて役に立つ」利用例を 1 つ考案して報告しなさい。

課題 6A

今回の課題は「演習 1」「演習 2」「演習 3」に含まれる小問 (合計で 9 個) の中から 1 つ以上を選択し、結果をレポートとして報告して頂くことです。以下の内容がこの順に含まれるようにしてください。

- 冒頭に題名「コンピュータリテラシレポート＃6」、学籍番号、氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も別途書く。)
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 以下のアンケートに対する回答。

Q1. テキストファイルや文字コードについてどれくらい知っていましたか。面白いところはありましたか。

Q2. Emacs の新たに知った機能で興味深かったものはありましたか。または別のエディタの方がいいと思っていますか (それはなぜですか)。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

なお、課題はグループでやって構いません。その場合も、(メンバー氏名を明記した上で) レポートは必ず各自で執筆してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。

＃7　コンピュータシステムとOS

今回の目標は次の通りです。

- コンピュータの一般的な構造について理解する — コンピュータの組み立てや設置を必要とする研究もありますので予備知識として。
- オペレーティングシステム (OS) とその役割について理解する — ソフトウェアの基本構造を知ることは計測実験などで必要になります。
- Unix を題材にプロセスやコマンドインタプリタなどの概念を理解する — プロセスについて知ることはソフト的な実験ではまず基本的な前提となります。

7.1　コンピュータシステムの構造

7.1.1　ハードウェアの一般的な構成

コンピュータの動作はすべてプログラムによって定められていますが、プログラムが動いて実際に仕事をするためには、CPU をはじめとする物理的な装置が必要です。この、目にみえるコンピュータの装置のことを**ハードウェア** (hardware) と呼びます。ハードウェアの中核部分はプログラムを動作させる **CPU** とプログラムやデータを保持する**メモリ**ですが、このほかに**入出力装置** (I/O devices) が必要です。その役割は、ユーザや他のコンピュータとやりとりしたり、長期的なデータ保持などの機能を提供することです。主要な入出力装置を挙げておきます。

- キーボード、マウスなど — 人間がシステムに指示やデータを渡す**入力装置** (input devices)
- ディスプレイ、プリンタなど — システムが人間に情報を提示する**出力装置** (output devices)
- ディスク、フラッシュメモリ — 長期的にデータを保持する **2 次記憶装置** (secondary storage)
- ネットワークインタフェース — 他のシステムとデータをやりとりする**インタフェース** (interface)

これらの装置の配置のされ方は、コンピュータの種類や用途によってさまざまですが、普段目にする PC などを例に描いたものが図 7.1 です。

中核となるのが VLSI 技術で作られた CPU です。今日の CPU の多くは、内部に**コア** (core) と呼ばれる独立した CPU の機能を果たす部分が複数入っていて、これらが並列に動作することで性能を向上させています。これを**マルチコア** (multi-core) と呼びます。そしてメモリ (これも VLSI 技術で作られます) は、CPU と密接にやりとりする必要があるので、専用の配線で CPU とつながっています。

コンピュータのケースを開けると、中には**マザーボード** (mother board) と呼ばれる基板が入っていて、その上には CPU・メモリとバスが搭載されています。バスはただの配線なのですが、その上にたくさん端子のついたソケットが数個くっついています。そして、CPU とメモリ以外の要素 (つまり入出力のための回路) はこのソケットに基盤を差し込むことで接続します。こうしておけば、このソケットを抜き差しするだけで、さまざまな入出力装置を増設したり外したりできるわけです。

この、抜き差しする基盤の反対側には、また別のさまざまな形のソケットがついていて、ここと実際の入出力装置をケーブルやコネクタで接続します。基盤の上に載っている回路はコントローラ、アダプタなどと呼ばれ、バス上の信号と各入出力装置の間の橋渡しを行う機能を持ちます。このようにすることで、CPU からはどの入出力装置でもバス上の同じ信号で制御でき、それぞれの装置に固有の信号の送受はコントローラにまかせることができるのです。

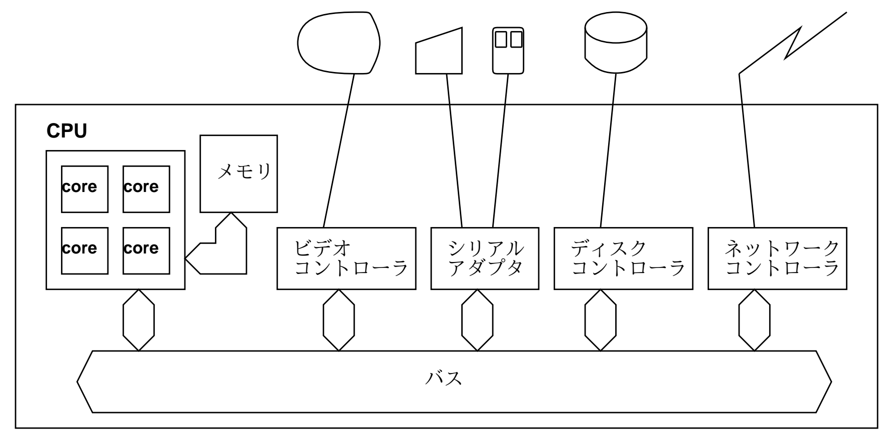


図 7.1: PC などのハードウェアの構成

PC などのシステムにはおおむね、ビデオコントローラ (画面)、シリアルアダプタ (キーボード、マウス)、ディスクコントローラ、ネットワークコントローラなどが備わっています

7.1.2　ソフトウェアの一般的な構成

ハードウェアは以上だとして、それを動かすソフトウェアはどのような構造になっているのでしょうか? あなたがたとえばブラウザで Web を見ているとき、コンピュータ上で動いているのはブラウザだけでしょうか? ブラウザを動かした状態でさらに別のプログラムを動かしたり、ブラウザの窓の位置を変更したりする時は、「ブラウザではない何か」を使って操作をしていますね?

この「何か」の部分は、使っているソフトが違っても同じように使え、さまざまなソフトを使う手助けとなってくれます。そしてその手助けも、プログラムを動かす、窓を操作する、など沢山の方面に渡っています。つまり一般に、ソフトウェアには次の 2 種類があるわけです。

- **アプリケーションソフト** (application software) — ユーザが各々の仕事を実行するためのソフト。
- **システムソフト** (system software) — アプリケーションを使って仕事をする上で必要な手助けや土台となったり、コンピュータを使う上で必要となる仕事を行うソフト。**基本ソフト** (primitive software) とも呼ぶ。

具体的には、どのような「手助け」が必要でしょうか? それはこれから見ていきますが、その前にシステムソフトを次のように分類しておきます (この分類は人により違うかも知れません)。

- **オペレーティングシステム** (operating systems、OS) —アプリケーションが動く下ざさえとなる機能を提供するひとまとまりのソフトウェア。
- **ミドルウェア** (middleware) — データベース管理システム、Web サーバなど、(OS と同様の意味で) アプリケーションが動く基盤となるが、OS とは独立に開発・提供されているもの。
- **言語処理系** (language processors) — プログラムを記述して動かすためのソフトウェア。
- **ユティリティ** (utilities) — ファイルの操作やデータ形式の変換など、(特定目的に特化した「アプリケーションソフト」とは対照的に) 汎用的な作業を手助けする。

今回はおもに Unix を題材として、まず OS について見ていきます。

7.1.3　OS とその働き ex

上で OS の役割りは「アプリケーションが動く下ざさえ」と書きましたが、それは具体的にはどういうことでしょうか? もう少し具体的に考えてみましょう。

コンピュータのハードウェア命令は、メモリとレジスタの間のデータ転送や四則演算など、ごく基本的な機能しか提供しません。ですから、多くのプログラムで必要とするような、画面の表示、文字の入力などの機能の実現には、かなり長いコードが必要です。

それを各アプリケーションを作る人が毎回用意するのは大変ですし、個々のアプリケーションが勝手に入出力機器を制御しはじめると、入力の混乱や画面の破壊など、様々な問題が起きそうです。ですから、以下は OS の重要な役割りだと言えます (図 7.2)。

- 多くのプログラムが必要とする標準的機能を一括して提供する

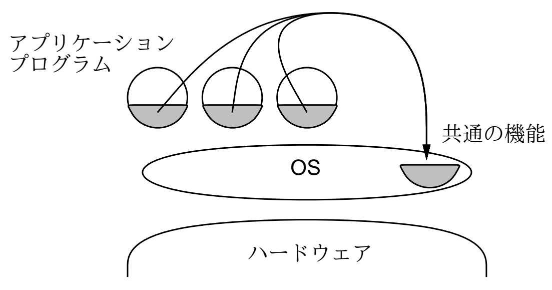


図 7.2: OS による標準的機能の提供

次に、現在のコンピュータシステムでは、ある窓で計算をさせながら、別の窓では待ち時間に Web を見るなど、複数プログラムを並行して動かす**マルチタスク** (multitask) 機能が使われます (図 7.3)。

- 複数のプログラムが並行して動作するのを管理する

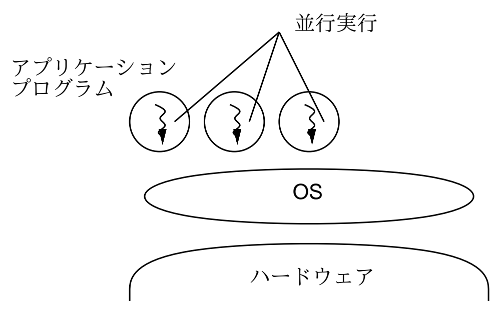


図 7.3: OS によるマルチタスク機能の提供

とすると、あるプログラムを動かすときに、そのつど CPU を止めてプログラムをメモリに書き込むわけには行きません (それでは現在進行中の別の仕事も止まってしまいます)。ですから、以下も OS の基本的な役割りです (図 7.4)。

- ユーザが指定したプログラムを読み込んで実行開始させる

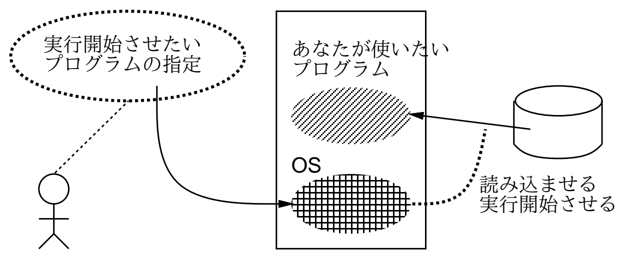


図 7.4: OS によるプログラムの実行開始

さて、そうして並行動作している複数のプログラムが同じメモリ番地やディスク上の領域を使おうとしたら、やはり混乱が起きます。ですから、以下のことも OS の重要な仕事です。

- コンピュータのメモリを各プログラムにうまく割り当てて調整する

- 入出力装置に対するアクセス (読み/書き) を管理する

見かたを変えると、コンピュータに備わった入出力装置、メモリ、CPU などはすべて、数が限られた貴重な**資源** (resource) だと言えます。そこで、OS の働きは次のようにまとめることができます。

- コンピュータ内部の各種資源を管理する

つまり、動作中のコンピュータでは常に OS が動き、ハードウェアと一体となりすべてを管理しています。そして、あなたやあなたのプログラムがコンピュータを使おうとする時、必要な資源はすべて OS に頼んで使わせてもらい、それによって仕事ができるのです (図 7.5)。

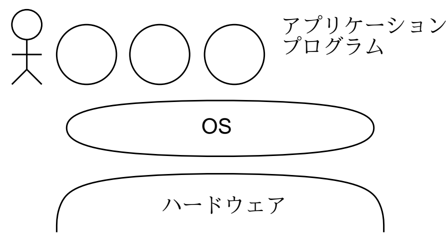


図 7.5: OS とユーザ、アプリケーションの関係

7.2 プロセスとその観察・操作

7.2.1 マルチタスクとプロセス ex

まず OS で一番目だつ機能であるマルチタスク、つまり複数のプログラムを並行して走らせる機能について見てみます。どのようにしてそんなことが可能なのでしょう?

コンピュータのメモリ上には命令の列 (プログラム) が置かれていて、CPU はプログラムカウンタ (PC) が指している番地から順に命令を取り出しては実行して行きます (図 7.6)。

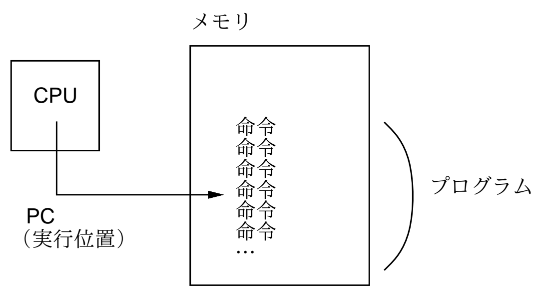


図 7.6: CPU による命令の実行

そこで、複数のプログラムを並行して動かすには、それらをメモリに一緒に入れます。そして、CPU についている**タイマー** (timer) と呼ばれる機能を動かした状態で、まずプログラム A の実行を始めます。タイマーは一定時間たつと CPU に信号を送ります。CPU はタイマーから信号を受け取ると、プログラム A の実行を一時中断してプログラム B の実行に切り替わります。またしばらくすると実行はプログラム C に、そして次は A に戻ります。このようにすると、複数のプログラムが「小刻みに切り替わりながら」実行されますが、CPU は非常に高速なので、ユーザにとってはすべてのプログラムが同時に動いているように見えます。

正確には、A〜C のうち 1 つは OS で、タイマー信号が来ると CPU は常に OS の実行に切り替わります。OS は各プログラムの使用時間や優先順位を調べ、次に実行するプログラムを選択し、実行開始させます。このため、OS は各プログラムへの CPU 割り当てを自由に制御できるのです。

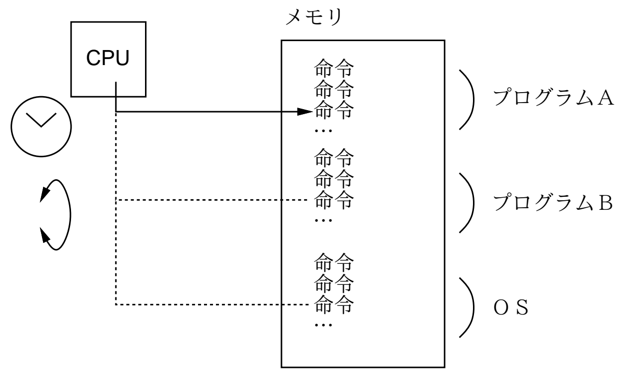


図 7.7: マルチタスク機能の実現

一般に、この「動いている状態のプログラム」のことを**プロセス** (process) と呼びます。現在は CPU を複数搭載したシステム [1] も多く使われていますが。そのようなシステムでも、動かしたいプログラム数 (プロセス数) は CPU 数より多いのが普通ですから、上で述べた「小刻みな切り替わり」があることは同じです。こうして、CPU は 1 個、メモリは 1 枚しかなくて、その上で直接プログラムを走らせるなら 1 個だけしか走らせられなくても、OS がその上にプロセスという「プログラムが走るいれもの」を作り出すことで、多数のプログラムを並行して走らせられるのです。

一般に「実際にはないけれどソフトウェア (OS など) の働きによって役に立つものを作り出す」ことを**仮想化** (virtualizaiton) と言い、コンピュータシステムでは多く使われる考え方です。そして、プロセスは OS の働きによって作り出れた「仮想化された CPU とメモリ」なのです。[2]

7.2.2 ps — Unix でのプロセス観察 ex

Unix では、**ps**(process status) コマンドによって、現在動作中のプロセスを観察できます (指定するパラメタによって、表示するプロセスの範囲や詳しさを制御できます)。

Unix システムの系統ごとに ps のパラメタ指定は違っています。ここでは sol の OS である GNU/Linux の場合について、基本的なもののみを説明します。なお以下で「端末」というのはコマンドを打ち込んでいる「窓」に相当するものと思ってください。

- `ps` — 簡潔な表示。使っている端末から動かしたプロセスのみ
- `ps -f` — 各プロセスについてのより詳しい表示
- `ps -A` — 他人のものも含めたすべてのプロセス表示
- `ps -u` ユーザ名 — 指定ユーザの全プロセス表示

たとえば sol で `ps` を実行した結果を次に示します。

```
...$ ps
  PID TTY          TIME CMD
49299 pts/77   00:00:00 tcsh
84487 pts/77   00:00:00 ps
```

`PID` というのは**プロセス ID** で、システム内の各プロセスにつけられた固有番号です。そして `TTY` が端末番号で、現在自分は `pts/77` という端末から使っていることが分かります。`TIME` というのはそのプロセスがどれくらい CPU 時間を使用したかの累計、`CMD` はコマンド名です。

[1] 1 つの CPU 中に複数のコアが入っているマルチコアのシステムもありますし、さらに CPU チップを複数搭載した**マルチプロセサ** (multiprocessor) と呼ばれるシステムもあります。
[2] OS のうちでプロセスを作り出す機能の部分を**プロセス管理**と呼びます。

これを見ると、現在の端末からは 2 つのプロセスを動かしていると分かります。1 つは ps のプロセスですが、これは「現在自分は ps のプログラムを動かして自分のプロセスを調べているので、その調べるという作業をしているプログラムである ps も当然調べた中に入っている」ということです。[3]

もう 1 つの tcsh は**コマンドインタプリタ** (command interpreter) といい、ユーザからコマンドを受け付け、対応するプログラムを動かす働きのプログラムです (「コマンドを解釈してくれる」からこのような名前がついている)。コンピュータが行う全ての動作はプログラムによって実行されるので、コマンドを打ち込んだらそのコマンドが動作する、ということ自体もそれをおこなうプログラムがあるから起きてくれる、というわけです。なお、Unix ではコマンドインタプリタのことを「貝殻のようにユーザを護ってくれている」というイメージから**シェル** (shell) とも呼びます。

プロセスが 2 つしか見えないのではつまらないので、全部のプロセスを見てみましょうか。sol の上で合計いくつくらいプロセスがあると思いますか?

```
...$ ps -A
   PID TTY          TIME CMD
     1 ?        00:01:42 init
     2 ?        00:00:04 kthreadd
     3 ?        00:14:24 migration/0
(途中略…)
130173 ?        00:00:00 httpd
130175 pts/126  00:00:00 a.out
131069 ?        00:00:00 Web Content
```

この例をやってみた時は 2725 個のプロセスが表示されました。[4]sol は多数のユーザが共同で使い、またさまざまなサービスも提供しているシステムなので、常時このように多くのプロセスが動作しているのです。なお、TTY が?のプロセスは端末から切り離されて単独で動き続けているもので、その中には CPU 使用時間が長いものも含まれています。

しかし 2 千以上では見ても分かりませんから、今度は自分が動かしているプロセスを見ることにします (皆様も自分の ID を指定してください)。

```
...$ ps -u ka002689
   PID TTY          TIME CMD
 49298 ?        00:00:00 sshd
 49299 pts/77   00:00:00 tcsh
 67891 ?        00:00:00 sshd
 67896 pts/162  00:00:00 tcsh
 68171 pts/77   00:00:00 ps
```

このときはもう 1 つ別のマシンから sol に入っていたので、pts/162 の tcsh が増えています。あと、sshd というのが 2 つありますがこれは何でしょうか? 外部からネットワーク経由で (より正確には SSH プロトコルで)sol に接続すると、その接続は sshd というプログラムにつながり、そこで認証処理が行われて OK だと、新しい端末を 1 つ割り当てて tcsh を起動します。この処理をやっているのが sshd のプロセスです。このプロセスは使用を終わるまでずっと待っていて、終ったら接続を切ります。ためしに、pts/162 の側を exit してから再度見てみます。

```
...$ ps -u ka002689
   PID TTY          TIME CMD
 49298 ?        00:00:00 sshd
```

[3]鏡に向かって写真を撮ったら撮っている自分も写るようなものだと思ってください。
[4]実際は画面で数えられるわけはないので、ファイルに保存して数えましたが、そういう話題はまた後で。

```
49299 pts/77   00:00:00 tcsh
93685 pts/77   00:00:00 ps
```

確かに tcsh と sshd が 1 つずつ減りました。ところで sshd は TTY が?ですが、これは sshd がログイン処理の後に端末を割り当てるので sshd 自体は端末の中で動作していないことによります。では最後に、より詳しい表示を見るため、オプションに-f を追加します。

```
...$ ps -f -u ka002689
UID         PID   PPID  C STIME TTY          TIME CMD
ka002689  49298  49213  0 12:33 ?        00:00:00 sshd: ka002689@pts/77
ka002689  49299  49298  0 12:33 pts/77   00:00:00 -tcsh
ka002689 103662  49299 17 14:55 pts/77   00:00:00 ps -f -u ka002689
```

いくつかフィールドが増えています。まず PPID は「Parent PID」つまりそのプロセスを作り出したプロセスの ID になります。よく見ると、sshd が tcsh を作り出し、そして tcsh が ps を作り出していますね。C は CPU 使用率で、そのプロセスが実行している全時間のうち CPU を使っていた割合 (%) です。ps は少し CPU を使っていますが、あとはほとんど使っていないことが分かります。また CMD のところはコマンド名だけでなくオプションまで表示されています。

このように、ps を使うことで現在のプロセス群の動作状態をかなり詳しく調べることができます。ここで説明した以外のオプションについては、man コマンドなどで調べることができます。

7.2.3 シェルによるプロセスの生成 ex

ここまでプロセスの観察について見て来ましたが、プロセスを作るのにはどうすればいいのでしょう? まず外部から sol にログインするとそれに対応するシェル tcsh のプロセスができることが分かりました。次に、シェルに対してコマンド「ps」や「emacs &」などを打ち込むと、そのコマンドを実行するためのプロセスがそのつど生成されます (図 7.8)。

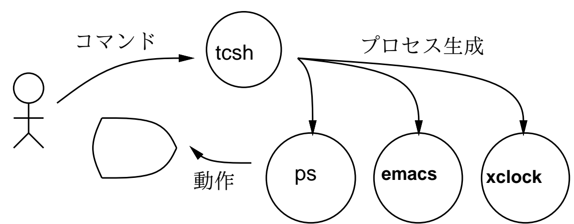


図 7.8: コマンドインタプリタ

毎回 ps を実行するごとに、PID が異ることに気づきましたか? つまり、ps のプロセスは 1 回表示を行うだけで終わってしまい、必要のつど新たに作られています。一方、tcsh は同じままです。この様子を図 7.9 に示します。つまり、tcsh はずっと動いていますが、コマンドの方は利用者がコマンドを打つたびにそれを実行する新しいプロセスが tcsh によって作られるのです。[5]

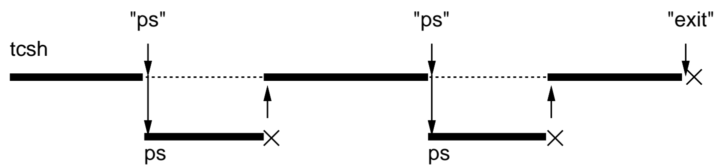


図 7.9: tcsh によるコマンドの発行と待ち合わせ

[5] tcsh が終るのは、利用者が exit という特別なコマンドを打ち込んだ時だけです。ということは、exit というのは他のコマンドのように新しいプロセスとして実行されるのではなく、tcsh 自身によって実行されることになります。このような「特別な」コマンドがいくつか存在します。

図の点線は、tcsh が子プロセスの完了を待つ箇所です。通常は一つコマンドを打ったらその完了を待って次のコマンドを打つちますね。ただし時間が掛かったり別の窓として動きはじめるコマンドの場合には、待ちたくないですね。そのときはコマンドの最後に「&」をつけて「待たずにすぐ次のコマンドを打ちたい」と指示します。つまり、&をつけると新しいプロセスができるのではなく、常に新しいプロセスはできて、ただ&をつけないとそのプロセスの完了を待つので、次のコマンドを打つときには先のプロセスは消滅しているわけです。

7.2.4 kill によるプロセスの操作 ex

プロセスは生成されたあとは仕事が終れば自分で終了しますし、ブラウザなど対話的プログラムであれば終了メニューなどによって終了します。しかし時には、プロセスがおかしくなって外部から操作する必要が生じることもあります。そのときは **kill** コマンドを使って、プロセスに次のような「信号」を送ります (PID は `ps` で調べたものを指定します)。

- `kill -STOP PID` — プロセスの実行を一時凍結する
- `kill -CONT PID` — 凍結したプロセスの実行を再開する
- `kill -TERM PID` — プロセスに「終わってほしい」と信号する
- `kill -KILL PID` — プロセスを強制終了させる

なお、コマンド実行中に`^C` を押すとそのコマンドを実行しているプロセスに`-TERM` 相当のシグナルが、`^Z` を押すと`-STOP` 相当のシグナルが、それぞれ送られます。[6]

演習 1 プロセスの生成や操作について、次の内容から 1 つ以上 (できれば全部) 試してみなさい。

a. 「`xclock -update 1 &`」により秒針つきの時計を画面に表示し、その PID を調べなさい。次に、`kill` を使ってこのプロセスを操作してみなさい。操作に先立ち、調べるべきことがらを複数考えてから試すこと (たとえば、凍結してから再開した場合に時計は「遅れたまま」になるかどうかなど)。

b. Emacs エディタを動かし、同様に操作してみなさい。操作に先立ち、調べるべきことがらを複数考えてから試すこと (たとえば、凍結した状態で Emacs の窓の上に別の窓を重ねて隠すと、上の窓をどけたときにその部分の表示は見えるかとか、凍結した状態で文字を打ち込むとその文字は入るかとか、その後解凍した時どうなるかなど)。

c. プロセスの親子関係について (PID と PPID の関係を見て) 確認したあと、「プログラムを起動すると、そのプロセスの PPID は起動したシェルになる。そのプログラムを動かしたままログアウトすると、起動したシェルが消滅するが、動かしたままのプログラムの PPID はどうなるか」を調べる実験をおこないなさい。動かしたままのプログラムとしては「`sleep 秒数 &`」を推奨するが、そのほかのものを使ってもよい。実験に先立ち、どうなるかについての想定とその理由を書き留めておき、結果と照合すること。

7.3 シェルの制御機能

7.3.1 リダイレクションとパイプ ex

リダイレクション (redirection) とは、データの流れを切り替える機能です。Unix では各プログラムは、**標準入力** (stdin)/**標準出力** (stdout)/**標準エラー出力** (stderr) という 3 つのチャネル (データの通り口) を使ってデータを読み書きするのが標準になっています。通常状態では、標準入力はキーボード入力、標準出力と標準エラー出力は画面につながります (図 7.10 左)。

しかし先の例のように、出力をファイルに保存したいとき、あるいは入力があらかじめ用意されているのでファイルから読ませたいときは次のように指定することでおこなえます。

[6]「相当の」というのは、いちおう区別のためにシグナル番号は違えてあるけれど機能的には同じという意味です。

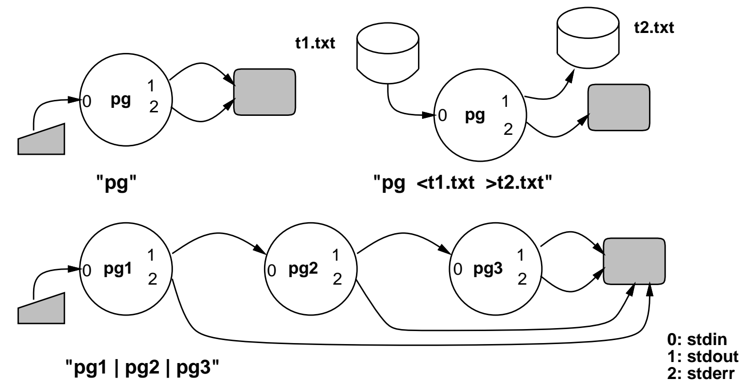


図 7.10: リダイレクションの考え方

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

3 番目は「出力の追加」で、これを使うと既にあるファイルの末尾に出力内容を追加できます。そして図 7.10 右のように、入力と出力のリダイレクションは併用もできます。

ところで、stderrって何でしょう? 実は各プログラムは通常の出力は stdout に送りますが、エラーメッセージだけは stderr に送ります。なぜかというと、たとえば長い時間掛かる処理をファイルに保存するようにリダイレクションを使った時、何か問題が起きてプログラムがそれを教えてあげようとメッセージを出してもファイルに入ってしまったらユーザがメッセージを見るのは翌朝だったりして不便だからです。そこで、stdout をリダイレクションしても stderr はもとのまま画面につながっていて、メッセージがすぐ見られるようにしているわけです。

ところで、上でプログラムと書いたのは実行している状態のプログラムですから、実際にはプロセスです。つまり stdin、stdout、stderr はプロセスごとに接続先を制御できます。そこでシェルでは

コマンド | コマンド | コマンド

のように「|」を使って複数のコマンドを区切ったときは、その前側のプロセスの stdout を後側のプロセスの stdin につなぐ、という制御が実行されます。これを**パイプライン** (pipeline) と呼びます (図 7.10 下)。stderr はこの場合も画面のままです。

ですから先程の例に戻ると、多数の行がある `ps -A` の出力をゆっくり見るためには、ファイルにリダイレクトする代わりに「`ps -A | less`」としてもよいのです。`less` は `man` コマンドで呼び出される「1 画面ずつ見るためのプログラム」であり、ファイル名を指定したらそのファイルを、指定しなければ stdin の内容を 1 画面ずつ表示してくれます。[7]

シェルの機能の中でパイプラインは代表的なものですが、ほかにもコマンドの実行され方を制御する方法が多数用意されています。ここでは主要なもの 3 つを説明します。

- コマンド ; コマンド — 前のコマンドに続けて後のコマンドを実行する。たとえば `ps` を実行する前に日時も表示させたい場合は「`date; ps`」と 1 行に打つことができる。
- コマンド & — コマンドを並列に実行する (終了まで待たない)。
- (コマンド…) — コマンドをサブシェルで実行する。サブシェルというのは、シェル自身のそっくりコピーなのでコマンドが動作する様子は同じだが、それらの実行や入出力は 1 つにまとめられて行われる。たとえば次のようにすることで、1 分間をおいて 2 回プロセスをファイルに記録できる (そしてその間待っていなくてよい)。

```
...$ (date ; ps -f ; sleep 60 ; ps -f) >rec.txt &
```

[7] `less` の中では `n`(次の画面)、`p`(前の画面)、`q`(終了する) などのコマンドが使えます。

7.3.2　ジョブコントロール

ジョブ (job) とは、コンピュータ用語としては、プログラムやプロセスよりも大きな (それらがいくつか連なった)「ひとまとまりの仕事」という意味で使われます。

一方で Unix では、1 行のコマンド行で複数のプログラムを起動したり、それらの間で入出力のやりとりを指定するなど、かなり複雑なことができます。しかし、複数のプログラムということは複数のプロセスから成るわけなので、それらを `ps` で探したりして操作するのは面倒です。

そこで、コマンド行で指定したひとまとまりのプロセス全体 (ジョブ) に対して停止や再開などの処理ができる機能が作られました。これが Unix の**ジョブコントロール**機能です。ジョブコントロール的にはジョブは次の 3 つの状態を持ちます。

- **フォアグラウンド** (foreground) — 端末入出力 (キー入力/画面出力) ができる通常の状態
- **バックグラウンド** (background) — プロセスは実行しているが端末入出力はできない
- **中断** (suspend) — プロセスが実行を中断している (凍結されている) 状態

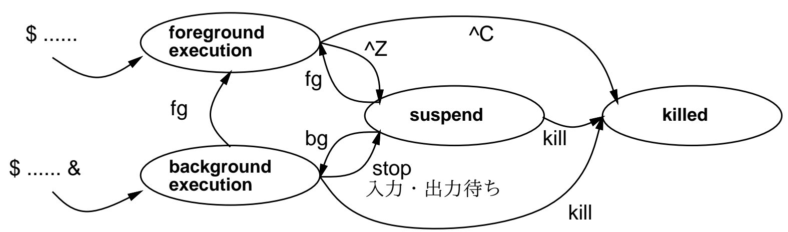


図 7.11: ジョブコントロールの状態遷移

これらの間の遷移を図 7.11 にまとめます。通常コマンド (&なし) で実行開始したジョブはフォアグラウンド状態です。この状態で終了したり Ctrl-C で止めるこもありますが、実行中に Ctrl-Z を打つと中断状態になります。次に&をつけて実行開始したジョブはバックグラウンド状態です。この状態では端末入出力はできません。このジョブも自然に終了することもありますが、止めるには「`kill`」を使います。また「`fg`」コマンドを使ってフォアグラウンドに移すことや、「`stop`」コマンドを使って中断にすることもできます。また、ジョブ中のどれかのプログラムが端末入出力を行おうとすると自動的に中断になります。最後に中断状態のジョブは、終了させるのには `kill`、フォアグラウンド実行に移すには `fg`、バックグラウンド実行に移すには `bg` を使います。

現在あるジョブを調べるには `jobs` というコマンドを使います。そしてその番号をもとに、ジョブを操作するコマンド `fg`、`bg`、`stop`、`kill` では「`%番号`」という形でジョブを指定します (`kill` で`%`を指定しない場合は先にやった PID を指定する方の機能になります。例を見てみましょう。

```
...$ emacs &   ← emacs をバックグラウンドで
[1] 33893      ←ジョブ番号と PID を表示
...$ man ps &  ← man コマンドをバックグラウンドで
[2] 36457      ←ジョブ番号と PID を表示
...$           ← [RET] だけを打つ
[2]  + 中断 (tty 出力) man ps  ← man は出力を試み中断
...$ xclock -update 1   ← xclock をフォアグラウンドで
(&を忘れたのでコマンドが打てない)
^Z     ← Ctrl-Z でフォアグラウンドの xclock を中断
中断
...$ jobs    ← ジョブ一覧を確認
[1]     実行中です      emacs
[2]  - 中断 (tty 出力) man ps
```

```
[3]  + 中断          xclock -update 1
...$ bg %3  ←時計を動かすためバックグラウンドに
[3]  xclock -update 1 &  ←確認のためコマンド行表示
...$ fg %2  ← man の表示を見るためフォアグラウンドに
(man の画面になる。しばらく見てから q で終了)
man ps      ← さっき fg した対象のコマンド行が表示される
...$ jobs   ← ジョブ一覧を確認
[1]  + 実行中です    emacs
[3]  - 実行中です    xclock -update 1
...$ kill %3
...$        ← [RET] だけを打つ
[3]    終了          xclock -update 1
...$        ← emacs を^X^C で終了してから [RET] を打つ
[1]    終了          emacs
...$        ←ジョブはなくなった状態
```

演習 2 ここまでに出て来た「ps -A >test.txt」「ps -A | less」「(date ; ps -f ; sleep 60 ; ps -f) >rec.txt &」および上のジョブコントロールの使用例をそのままやって見なさい。納得できたら、次の課題から 1 つ以上試してみなさい。

a. stdout をリダイレクトしても stderr は画面に出ることを確認できるような例を考案して実行し確認しなさい。また stderr も stdout と一緒にリダイレクトする方法を man tcsh で探してこちらも確かにそうなっていることを確認しなさい。

b. コマンド「tee ファイル」は、stdin から入って来た文字をそのまま stdout にコピーするが、ただしそのデータを指定したファイルにも記録する。「ps -A | tee test2.txt | less」で 1 画面ずつ表示している途中で q で less を終了したとき、表示された量とファイルに記録した量は同じか違うか。違うとしたらその差はどれくらいか、また何回か実行したとき変化するか。1 つのパイプライン内で tee を複数回使った場合はどうか。などの問題を検討しなさい。これをもとに、この現象の理由を推理してみなさい。[8]

c. ジョブがバックグラウンドで実行中に、端末入出力が必要になると、そこで中断するはずである。実際にそのようなことが起こる例を考案して実行し確認しなさい。できれば、そのようなことが複数回起きる (例: バックグラウンド実行→出力要求 (中断) → fg →出力→ Ctrl-Z → bg →バックグラウンド実行→入力要求 (中断)、など) 例だとより望ましい。

いずれの課題も、実際にやったことや画面表示の記録 (長い場合は途中を適宜省く) をきちんと示すこと。

最後の課題をやるときにキーボードから入力するコマンドが必要であるが、それには cat を使えばよい。cat は「cat ファイル…」で指定された 1 つ以上のファイルを順次 stdout に出力するが、ファイル名の代わりに「-」を指定するとそこでは stdin からの入力を出力する (または 1 つもファイルを指定しない場合も stdin からの入力を出力する。キーボード入力を終らせるには Ctrl-D を打てばよい。次の例を参照。

```
...$ cat >test3.txt
aaa
bbbbbb
^D     ← ctrl-D を打ったところ
...$   ← test3.txt には aaa と bbbbbb が入っている
```

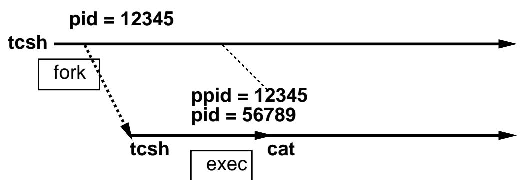
[8] 出力の量が多くないとうまく調べられない場合は、(ps -A; ps -A; ps -A) のようにして複数回 ps を動かすとよい。

演習 3 `ps` で PPID フィールドを表示させせることで、あるプロセスがどのプロセスによって作り出されたが分かる。これを利用して、次のことを検討してみなさい。単に憶測を書くのでなく、まず仮説を立て、次にそれを確認するテスト実行を行いその結果を示して検討すること。

a. 「`(sleep 10; sleep 11; sleep 12) &`」のような複合コマンドがどのようにして作られているのかを検討する。

b. 「`(ps -f | cat | cat | cat | cat) &`」のようなパイプラインがどのようにして作られているのかを検討する。

c. シェルが一般に「`;`」「`|`」「`(...)`」「`&`」などの機能をどう実現しているか検討する。



なお、Unix 内部でプロセスを作る際には、(1)1 つのプロセスが親子の 2 つに分裂する (プログラム名は同じまま)、(2) あるプロセスが別のプログラムに「変身」する (PID は同じまま)、という 2 つの機能を組み合わせて実行しています (図)。(1) は分かりやすいですが、(2) は「なぜか `tcsh` であるべきプロセスが `cat` になっている」のような分かりにくい結果をもたらします。

課題 7A

今回の課題は「演習 1」「演習 2」「演習 3」に含まれる小問 (合計で 9 個) の中から 1 つ以上を選択し、結果をレポートとして報告して頂くことです。以下の内容がこの順に含まれるようにしてください。

- 冒頭に題名「コンピュータリテラシレポート＃ 7」、学籍番号、氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も別途書く。)
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 以下のアンケートに対する回答。

Q1. OS の機能やプロセスについてどれくらい知っていましたか。新たに知って面白かったことは何ですか。

Q2. `ps` によるプロセスの観察や `kill` による操作などについてどのように思いましたか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

なお、課題はグループでやって構いません。その場合も、(メンバー氏名を明記した上で) レポートは必ず各自で執筆してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。

＃8　フィルタとシェルスクリプト

今回の目標は次の通りです。

- フィルタを用いた柔軟なユティリティの構成について理解する— Unix で多様な作業をこなす原動力がユティリティです。
- シェルの変数や展開機能、実行パスについて学ぶ — Unix を使う上ではこれらの概念を知っていると効率が高められます。
- シェルスクリプトの考え方について理解する — 繰り返しおこなう作業をスクリプトにすることで作業効率がいっそう高くなります。

8.1　ユティリティとフィルタ

8.1.1　「大きなユティリティ」と「小さなユティリティ」 ex

ユティリティとはコンピュータの世界では「汎用的な作業をこなすプログラム」を意味します。そしてユティリティには「大きなユティリティ」と「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、アーミーナイフ (図 8.1 右) のように、1 つのプログラムが多数の機能をこなすようなものをいいます。便利そうですが、次の弱点があります。

- 指定が沢山必要で、使い方が複雑になりがちである。
- 1 つの機能だけ使いたくても大きなプログラムが動くので遅くなりやすい。
- 機能を増やしたり修正するのが大変である。

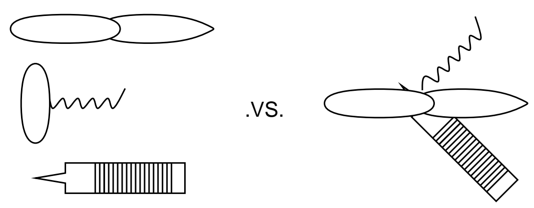


図 8.1: 単機能の道具と多機能の道具

一方小さなユティリティとは、個別の道具 (図 8.1 左) のようなもので、1 つのプログラムは単純な機能だけを提供し、複雑なことはプログラムを組み合わせて対応する考え方です。この方式では、大きなユティリティの弱点が克服できます。

- 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。

8.1.2　フィルタ ex

Unix は伝統的に小さいユティリティの文化です。小さいユティリティのためには複数のプログラムを組み合わせて動かす仕組みが必要ですが、それがパイプラインです。

プログラム | プログラム | … | プログラム

パイプラインの途中にあるプログラムはどれも「標準入力から入力データを読み取り、処理を行なって、標準出力にデータを書き出す」形で動作します。その形がちょうど、空気や水をろ過するフィルタに類似しているので、この種のプログラムを Unix では**フィルタ** (filter) と読びます。既に学んだ `cat` もフィルタの 1 つです。

このやり方がうまくいくためには、どのプログラムの出力も別のプログラムの入力として使える必要があります。そのためには色々な考え方がありますが、Unix の場合は「テキストファイル」つまり人間が読み書きできるようなデータを扱う、という形で共通化しています。以下では代表的なフィルタをいくつか見ながら、これらの原理がどのように具体化されているかを学んでいきます。

8.1.3　tr — 文字の置換 ex

tr は入力の各文字を (原則として)1 対 1 で別の文字に変換 (TRanslate) して出力するフィルタです。キーボードが壊れていて小文字の「`a`」が大文字になってしまうマシンでファイルを打ち込んだとしましょう (何てわざとらしい例!)。あとで大文字の「`A`」を小文字に直すのには次のようにします。

```
...$ cat test.txt
ThAt is A cAt.
...$ tr A a <test.txt
That is a cat.
```

`tr` の基本的な使い方は次のとおりです。

- `tr` 文字列$_1$ 文字列$_2$ — 文字を別の文字に変換

「文字列$_1$」の 1 文字目は「文字列$_2$」の 1 文字目、2 文字目は 2 文字目、というふうに対応する文字どうしの変換が行われるわけです。ほかの多くのフィルタと違って、`tr` はファイル名指定を受け付けないので、ファイル入力が必要なら入力リダイレクションを用います。

しかし、注意深い人は「そのキーボードは小文字の「`a`」が入らないはず」と気づいたかも知れません。別マシンに移ったことにもできますが、壊れたマシンでやりたければ、任意の文字を 8 進文字コードで指定することができます (文字と 8 進コードの対応は「`man ascii`」で見てください)。

```
...$ tr A '\141' <test.txt
That is a cat.
```

もうちょっと実用的なのは、すべての大文字を対応する小文字に直すことでしょう。

```
...$ tr A-Z a-z <test.txt
that is a cat.
...$ tr ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz <test.txt
that is a cat.
```

どちらも同じ動作ですが、「`-`」で文字範囲を指定する方が楽ですね。ここまでは指定する 2 つの文字列の長さが同じでしたが、もし文字列 2 が短ければ、文字列 1 のあぶれた文字には全て文字列 2 の最後の文字が対応します。

```
...$ tr A-Za-z a <test.txt
aaaa aa a aaa.
```

tr にもいくつかのオプションがあります。

- -c — 文字列 1 に「ない」すべての文字を集めたものを改めて文字列 1 だと思う。

例を見てみましょう。

```
...$ tr -c A-Za-z / <test.txt   ←英字以外をすべて「/」に
ThAt/is/A/cAt//...$ ←改行が「/」になったので改行されない
```

「すべての」文字に改行文字も含まれるのが不都合なら、次のように改行も除外すればよいです。

```
...$ tr -c 'A-Za-z\012' /<test.txt   ← 012 は改行の 8 進コード
ThAt/is/A/cAt/
...$             ←改行されてから次のプロンプト
```

- -d — 文字列 1 に現われる各文字を消去 (delete)。この場合、文字列 2 は指定する必要がない。

こちらは不要な文字を削除するのに便利です:

```
...$ tr -d ' ' <test.txt   ←空白文字をすべて削除
ThAtisAcAt.
```

- -s — 置き換えが連続して起こる場合には、最初の 1 個だけ出力 (squeeze)。

たとえば単語の「数だけ」知りたい場合は、次のようにできます。

```
...$ tr -s A-Za-z a <test.txt
a a a a.
```

これらの指定を組み合わせて様々なことができます。たとえば「tr -cd 'A-Za-z \012'」で英字と空白と改行以外をすべて消して「きれいな」単語だけにできますし、「tr -cs A-Za-z '\012'」で英字以外の並びをすべて改行 1 個に置き換えて各単語を 1 行ずつバラバラにできます。

8.1.4 sort と uniq — 整列と重複除去 ex

ここから先で説明するフィルタはどれも、ファイルを指定すればそのファイルから読み込み、指定しなければ標準入力から読み込みます。**sort** は、ファイルの行を指定した順番に並べ替えてくれます。

- sort ファイル… — 行単位での並べ替え

いちばん簡単には、何もオプションを指定しないと、sort は行全体を比較して、その文字コード大小順にもとづき、行を小さい順に並べます:

```
...$ cat test.txt
this
is
a
pen
...$ sort test.txt
a
is
pen
this
```

しかし、文字コード順だと $A < B < \ldots < Z < a < b < \ldots < z$ なので、大文字と小文字が混ざっているとあまり嬉しくないかも知れません。

```
...$ cat test.txt
This
is
a
pen
...$ sort test.txt
This
a
is
pen
```

このようなときには「-f」(大文字小文字統合) オプションを指定すれば、対応する大文字と小文字の文字は同一とみなしてくれます。また、数値データも文字コード順で比較されるとあまり嬉しくありません。

```
...$ cat test.txt
10
2
1
...$ sort test.txt
1
10
2
```

文字コード比較だと「1」で始まるものが全部終わってからはじめて「2」で始まるものが来ます。このような場合には「-n」(number) オプションを指定すると、数値としての順に並べられます。いずれの場合も、「-r」(reverse:逆) オプションを追加すると小さい順でなく大きい順に並べられます。

行全体ではなく、行の特定の部分にもとづいて並べ変えを行わせることもできます。その指定方法は少し面倒ですが、いちばん簡単には「-k 1」「-k 2」などと指定することで (空白で区切られた)「1番目の欄」「2 番目の欄」などを指定できます (詳しくは「man sort」を見てください)。

ふたたび、行全体を整列する場合に戻りますが、「どのような行があるか」だけ知りたい場合には重複を除く (同じ行が複数あった場合に 1 行だけ残してあとは消す) ほうが望ましいですね。一般のファイルについてこれをやるのは面倒ですが、整列後なら同じ内容の行が隣り合っているので簡単です。それをやってくれるのが **uniq** というフィルタです。

- uniq ファイル… — 整列後のファイルの重複を除く

たとえば単語リストでこれを行ってみましょう。

```
...$ cat test.txt
this is a pen.
what is this?
...$ tr -cs A-Za-z '\012' <test.txt
this
is
a
pen
```

```
what
is
this

...$ tr -cs A-Za-z '\012' <test.txt | sort
a
is
is
pen
this
this
what
...$ tr -cs A-Za-z '\012' <test.txt | sort | uniq
a
is
pen
this
what
```

なお、uniqに-c(count) オプションを指定すると、同じ行がいくつずつあったか数えてくれます:

```
...$ tr -cs A-Za-z '\012' <test.txt | sort | uniq -c
  1 a
  2 is
  1 pen
  2 this
  1 what
```

8.1.5 head と tail と wc — 先頭/末尾/数える ex

あと少しだけ、行数関係のフィルタについて簡単に説明します。

- head -行数 — 先頭から指定した行数のみ取り出す
- tail -行数 — 末尾から指定した行数のみ取り出す

これらはファイルの先頭 N 行または末尾 N 行だけを取り出すフィルタで、長いファイルの頭だけ/終わりだけ見たい場合に使います。行数を省略すると 10 行分取り出されます。また、入力の行数が指定行数より少ない場合はその全部が取り出されます。

また、入力の文字数や行数などが知りたい場合には、**wc** (Word Count) を使うことができます。

```
...$ ps | wc
      7      39       241
行数↑      ↑単語数   ↑文字数
```

数字が 3 つ表示されますが、順に「行数」「単語数」「文字数」です。「wc -l」「wc -w」「wc -c」の各オプションにより、行数、単語数、文字数だけを出力させることもできます。

演習 1 まず「echo This is A cAt | tr A z」など echo とパイプを使って tr の説明されている機能をひととおり確認。次は英語の文書が欲しいで、「setenv LANG C」を実行する (man を実行してみて英語になったことを確認。logout すると元に戻る)。「man コマンド名 >ファイル」でマニュアル内容をファイルに保存できるのでやってみる。もしファイルに行番号がついている方がよければ「cat -n ファイル」でできる。以上をふまえて、以下の課題をやってみなさい。

a. 行数の多いファイルの途中 (たとえば 100 行目から 110 行目まで) を取り出して表示するにはどうしたらいいか考えてやってみる。
b. 自分が選んだコマンドの man に出て来る単語の一覧 (出現回数つき) を作ってみる。たとえば「the」や「that」などのよく出て来そうな単語の出現回数は何回か調べてみる。
c. 自分が選んだコマンドの man に出て来る単語の出現頻度トップ 10 の表 (のようなファイル) を作成してみる。

8.2 正規表現

8.2.1 grep 族 — パターンにあてはまる行を探す ex

grep、**fgrep**、**egrep** の 3 つのコマンドはいずれも「入力のなかに指定したパターン (Unix の用語では**正規表現** (regular expression) にあてはまる行があったら、その行全体を打ち出す」という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。

そして、3 つの違いはパターンとして何が書けるかの違いになります。まず fgrep の場合、パターンとしてたんなる文字列のみが書けます。たとえば「that」をいつも「taht」打ってしまうくせがある人が、そのまちがい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:

```
...$ 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。なお、「[」と「]」はシェルのメタキャラクタ (後述) なので、grep にパターンとして渡すときには「'…'」で囲む必要があります。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。表 8.1 にパターンについてまとめておきます。(1) は grep ではパターン α が 1 文字に対応するパターンでなければならないという制約があります。(2) は egrep のみの機能で

表 8.1: grep 族のパターンのまとめ

パターン	説明
c	c という文字そのもの。
`[…]`	…のうちどれか 1 文字 (`tr` 同様 a-z のようにも書ける)。[1]
`.`	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。(2)
α`*`	α というパターンの 0 個以上の繰り返し。(1)
`(…)`	くくり出し。(2)
α `|` β	α または β。(2)
`\(…\)`	…のところを一時的に覚える。(3)
`\1`、`\2`	覚えたものの 1 番目、2 番目、…。(3)

す。一方 (3) は `grep` のみの機能です。また、「`.`」の長い連続など、組み合わせが爆発的に多くなるパターンは `egrep` では実現上うまく扱えません。というわけで、`grep` と `egrep` は適材適所で使い分ける必要があるわけです。

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見ましょう (`less` を使うのは、多数見つかったときゆっくり見るため)。

```
...$ grep ’パターン’ /usr/share/dict/words | less
```

パターンの例をあげておきます (うろ覚えの単語をさがすという実用的な使い方も可です)。

`’[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

8.2.2 sed — 文字列を置き換える ex

`tr` による文字置換は強力ですが、「自分は `that` を `taht` と打ってしまうので、これを正しく直したい」のような仕事には無力です。`tr` は各文字をバラバラに扱うので「特定の文字の並び」には対応できないからです。そのような文字列の置き換えには **sed**(stream editor) が適役です。

- `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?
...$
```

[1] `[^…]` のように先頭に`^`があると「…以外の文字」となる。

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 (他のコマンドは man sed で見てください)。なお、上の 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、「s」コマンドによる指定には 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。例を見てみましょう。

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

これは、「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」わけです。

場合によっては、こういう置き換えを多数やりたいかもしれません。その場合は

```
sed -e 's/Thsi/This/g' -e 's/Taht/That/g'
```

のように各コマンドのまえに-e オプションをつければ、いくつでもコマンドが書けます。あるいは、

```
s/Thsi/This/g
s/Taht/That/g
```

のように多数のコマンドを並べたファイルを準備しておき、「sed -f ファイル」の形で指定することもできます。sed は入力の各行についてファイルのなかにある命令を 1 行ずつ「実行」してくれます。つまり、これは一種の「プログラム」なわけです。

演習 2 正規表現を扱うフィルタの課題をいくつか挙げておきます。

a. /usr/share/dict/words から次のような単語を探しなさい (3 つ以上やってみること)。
- 「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。[2]
- 末尾が「otion」で終わる単語で、「e」が含まれないようなもの。[3]
- 先頭が「z」で最後が「tion」で終わる単語。[4]
- 母音を 5 つ連続して含む単語。[5]
- 5 文字のまったく同じ文字の並びが 2 回出て来る単語。[6]

b. sed を使って以下のことをやりなさい (2 つ以上やること)。
- 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。[7]
- 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。[8]

---

[2] 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。
[3] ヒント: 「終わる」は、grep の$の機能を使います。その後でパイプで「-v」つきの grep で「e」を排除すればよい。
[4] ヒント: 途中に任意文字が 0 個以上あるわけです。
[5] ヒント: 母音とは「a」「e」「i」「o」「u」のどれかですね。
[6] 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。
[7] 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。
[8] 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。

- 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…)。[9]
- `ls -l` の出力からファイル名前と大きさ (バイト数) の部分だけ抜き出し表示する。[10]
- 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。[11]

c. (チャンレンジ問題)「`aaaaabbbcc`」のように、「a が N 個、b が M 個、c が T 個」並んだ行について「できるだけ多くの a の並びと b の並び (a と b 同数つまり $min(M, N)$ 個) を消し、その個数だけ c の並びに追加する」置き換えを `sed` で作りなさい。[12]

8.3 シェルの進んだ機能とシェルスクリプト

8.3.1 シェル変数と変数展開

変数 (variable) とは (コンピュータ業界では) おもにプログラミングの用語であり、値を入れておける (そして変化させられる) 「入れ物」「箱」のようなものを意味します。そして、シェルにもそのような意味での変数があります。このあたりはシェルの種別によって違うのですが、ここでは sol で標準に設定されている `tcsh` を前提とします。

シェルでは変数は値を入れたときに作られます。変数の名前は英字 (A〜Z, a〜z, _) で始まり、英数字が並んだものである必要があります。変数に値を入れるのは `set` コマンドを使います。

```
...$ set ax1 = 'abc'
```

次に変数から値を取り出すときは、**変数展開** (variable substitution) を使います。具体的には何かというと、「`$変数名`」という形のものがあると、その部分が変数の中身で置き換わるのです。これを確認するのに便利なコマンドが `echo` です。

```
...$ echo "$ax1 + $ax1"
abc + abc
```

文字列の中の「`$ax1`」が先程入れた「`abc`」に置き換わっています。なお、この例のように文字列`"..."`の中では変数展開は置きますが、`'...'` で囲んだ場合は置きません。また、何も囲まない場合はもちろん展開が起きます。1 つ注意するのは、変数のうしろにすぐ英数字がくっついていると変数名が変わってしまうので、そのような場合は変数名を「`{...}`」で囲む必要があるということです。

```
...$ echo '$ax1 + $ax1'
'$ax1 + $ax1'          ← '...' の中は変数展開なし
...$ echo This is $ax1
This is abc            ← 文字列の外は当然変数展開
...$ set ax1x = 'def'  ← 新たな変数に値をセット
...$ echo ${ax1}x is not $ax1x
abcx is not def        ← 2 つの変数は別
```

あと、変数に入れる値はリスト (かっこで囲んだ値の並び) であってもよいです。このときは全体として取り出す以外に「`$変数名[番号]`」で特定の要素だけを参照もできます。

```
...$ set a = (x1 x2 x3 x4) ←リストを入れる
...$ echo $a
```

[9] 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。
[10] 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。
[11] 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。
[12] ヒント: a と b の個数に上限を決めることにすれば、その数だけ`-e` を使ってコマンドを並べる (か、またはその数だけパイプラインでつなげる) ことでできます。上限を決めない方法については「`mand sed`」を熟読する必要あり。

```
x1 x2 x3 x4            ←表示では () は現れない
...$ echo $a[2]        ←2番目だけ参照
x2
...$ set a[3] = zzz    ←3番目を変更
...$ echo $a
x1 x2 zzz x4           ←確かに変化した
```

8.3.2　既定義なシェル変数と実行パス

シェル変数の中には最初から値が入っているものがあります。代表的なものを挙げておきます。

- `$user` — 自分のユーザ名が入っている
- `$home` — 自分のホームディレクトリの絶対パス名が入っている
- `$cwd` — 現在位置の絶対パス名が入っている
- `$path` — 実行パスが入っている

実行パスとは「コマンドとして扱うプログラムが入っているディレクトリのリスト」です。シェルはコマンドが打ち込まれると、実行パス中のディレクトリを順番に取り出し、そのディレクトリに「コマンドと同じ名前で」「実行可能な」ファイルがあるか調べます。そしてそのようなファイルがあったら、それを実行させせます (図 8.2)。つまり Unix では、コマンドとは単なる「実行するプログラム」なのです。コンピュータでやることはすべてプログラムによって行うので、この考え方はとても合理的だと思います。

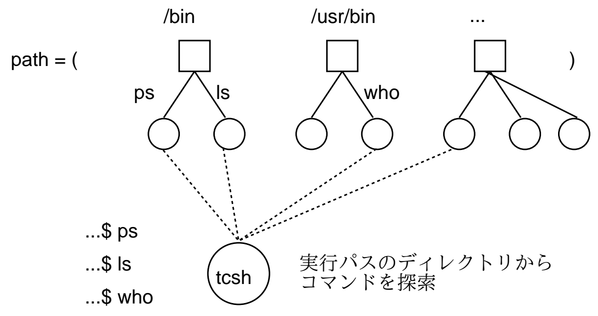


図 8.2: シェルによるコマンド探索

sol のユーザは管理者が設定した実行パスが最初から有効になっているため、標準のコマンドが利用できるわけです (試しに「`echo $path`」を実行してみるとよい)。そして、自分独自にコマンドを増やしたい場合は、たとえば次のようにすればできます。

```
...$ mkdir $home/bin ←ホームの下にディレクトリ bin 作る
...$ echo 'main() { puts("hello."); }' >test.c
...$ gcc -o $home/bin/hello test.c   ← hello 用意
...$ set path = ($home/bin $path)    ←パスに bin 追加
...$ rehash   ←パスやその内容を変更したことを tcsh に通知
...$ hello    ←新しいコマンドが
hello.        ←動くようになっている
```

自分のコマンドを入れておくディレクトリを毎回パスに追加したければ、後で出て来る設定ファイル `.tcshrc` のどこかに「`set path = ($home/bin $path)`」を入れておけば、いつでもここに置いた実行可能なファイルが使えるようになります。

8.3.3 ファイル名展開

ここまでで変数展開について説明しましたが、それと類似した機能である**ファイル名展開** (filename expansion) についても説明しておきます。これまで現在位置にあるファイルの一覧を表示するのには ls を使って来たと思いますが、それ以外に次の方法もあります。

```
...$ echo *
CL16 Desktop IED_HOME WWW WindowsEdu a.out bin public_html
t1 t3 t4 test.c test1.txt test2.txt
```

これは何でしょう? 実は、コマンド行に次のようなパターンを書くとシェルはそのパターンにあてはまるファイル名を探して来て置き換えてくれるのです。

- * — 任意の文字の並び (ただし先頭の「.」は除く)
- [...] — かぎかっこ内に書いた文字のうちいずれか (正規表現と同じ)。

これを使うことで、「すべての C プログラムを消す」であれば「rm *.c」、「a〜d いずれかで始まるファイルだけ消す」であれば「rm [a-d]*」のように一気に指定できるわけです。
そのほかに普段使う展開として次のものもあります。

- ホームディレクトリの展開 — 「~」は自分のホームディレクトリに、「~ユーザ名」は指定したユーザのホームディレクトリに展開される。
- 分配法則 — 「x{a,b,c}」→ xa xb xc のように隣接する文字列とくっつけた状態で展開。たとえば「.txt と.c と.s のファイルを全部消す」だと「rm *.{txt,c,s}」のように書ける。
- コマンド置換 — 「‘コマンド…‘」はコマンドを実行してその出力結果に置き換えられる。見ても分かりにくいけれど「‘」はバッククォート文字なので「バッククォート置換」とも呼ぶ。

最後のコマンド置換はわりあい便利で、たとえば「rm *.c」で全部消すのではまずくて残したいファイルがある場合は、次のようにすればよいわけです。

(1) 「echo *.c >t1」で候補のファイル名をファイル t1 に入れる。
(2) テキストエディタで t1 を開いて消したらまずいファイルの名前は削除。
(3) 「rm ‘cat t1‘」で t1 に入っている名前のものだけ削除。

なお、ここまでに「特別な働きを持つ文字」として$*[]'‘~\などが出てきました。このような文字を**メタ文字** (meta character) と呼びます。メタ文字を普通の文字として打ち込みたいときは、(1)「'...'」で囲むか (2) 直前に「\」を置くことで、特別な働きを止めることができます。

8.3.4 シェルスクリプト

ここまでに見てきたように、シェルのコマンドを使うとかなり複雑なことができます。そういう複雑なことを定期的にやる場合に、毎回考えるのは大変ですし、打ち間違えると面倒そうです。ではどうすればいいかというと、「ファイルに入れておけば」いいですね。この、コマンドをファイルに入れたものを**シェルスクリプト** (shell script) と言います (スクリプトとは台本という意味)。
たとえば、自分のプロセスを観察するのに「ps -u ka002689」と毎回打つのは面倒ですから、次の内容を「psme」というファイルに入れておくことにします。

```
#!/bin/sh
ps -u $USER
```

1 行目は「このコマンドは/bin/sh 用」という意味です。これまでコマンドの実行には tcsh を使って来ましたが、スクリプト用には/bin/sh という (古くからある方の) シェルを使うことが多いです。これまでに tcsh 前提で説明したことと違うのは次の点です。

- 組み込みシェル変数は$USER、$HOME、$PATHと大文字を使う ($cwdは無し)。
- 値の設定は「A="this is"」のように「変数名=値」をくっつけて (空白なしで) 使う (値が文字列でその中に空白が含まれるのは構わない)。
- 配列はないので$PATHなどはパスを「:」で区切ってくっつけた形になっている。

さて、2行目が本体です。ユーザ名を固定してしまうと自分にしか使えませんが、シェル変数$userを書いておけば、これを動かした人のユーザ名に展開されますから便利ですね。そして、それを動かすには次のようにします (この場合は自分で/bin/shを指定しているので、1行目は何も効果を持ちません)。

```
...$ /bin/sh psme
   PID TTY          TIME CMD
 52964 ?        00:00:00 sshd
 52965 pts/202  00:00:00 tcsh
 76133 pts/202  00:00:00 ps
```

こうしてシェルスクリプトにしておけば、もっと長いコマンドや、数行にわたるコマンドも、覚えておかなくてすみます。

これまで実行可能なファイルはCなどの言語で書いて作っていましたが、実はシェルスクリプトも実行可能にできます。

```
...$ chmod a+x psme  ←誰でも実行可能にする
...$ ./psme          ←パス名指定して実行
   PID TTY          TIME CMD
 52964 ?        00:00:00 sshd
 52965 pts/202  00:00:00 tcsh
118550 pts/202  00:00:00 psme ←確かに実行中
118567 pts/202  00:00:00 ps
...$ mv psme $home/bin  ←自分の実行パスに入れると
...$ rehash             ←パスの内容変更したらrehash
...$ psme             ←コマンドとしてどこでも実行可能
(実行結果はおなじなので略)
```

そしてこの形で実行するときは、1行目の「#!/bin/sh」は「このスクリプトは/bin/shで実行してね」という指定として意味を持ちます。そのために入れておいたわけです。そして今は「./psme」のように「/」が入る形で指定していましたが、先に説明した実行パスのどこかに入れておけば、名前だけで (つまり普通にコマンドとして) 使えるようになります。

なお、tcshは各ユーザによって起動されたとき、$home/.tcshrcの内容を読み取ってスクリプトとして実行します。なので、ここに書いておくことで毎回実行したい設定を自動的におこなわせることができます。たとえば、シェル変数promptはプロンプト文字列なので、これを自分の好みに変更することなどはおすすめです。

8.3.5　スクリプトの引数と変数

シェルスクリプトによってコマンドが作れるとしても、それに対してオプションや引数を渡せなければあまり面白くはありません。実は、スクリプトをコマンドとして起動したときに、その1番目、2番目、... の引数の値は$1、$2、... というシェル変数に予め設定されます。従って、それらを参照したスクリプトを書くことにより、引数の値を活用できます。たとえば「ls -F -l ファイル…」をよく使うとしましょう。次のようなシェルスクリプトをコマンドにしたらどうでしょうか。

```
...$ cat $bin/lsf
#!/bin/sh
ls -F -l $1
...$ ./lsf t1 t2
-rw-r---w- 1 ka002689 faculty 39395 Jun 29 16:45 t1
```

あれ? ファイルを 2 つ指定したのに 1 つしか表示されない…のは当然で、スクリプト内部で引数`$1`だけしか使っていませんね。ここは次のようにすればよいです (指定しない変数は無視)。

```
#!/bin/tcsh
ls -F -l $1 $2 $3 $4 $5 $6 $7 $8 $9
```

実際にはこのように並べるかわりに「`$*`」と書けば、全引数を埋め込むことができます (次の例参照)。

8.3.6 スクリプトの for ループ

シェルスクリプトは普通のプログラミング言語同様、枝別れや分岐なども記述できます。ここでは 1 つだけ、引数を順に処理するループの例を扱います。それには次の形を用います。

```
for i in $*
do
  繰り返し動作 ←この中で変数$i が各引数に設定されている
done
```

画像ファイルを複数指定するとそれらを表示する HTML ファイルを出力する、という例を示します。

```
#!/bin/sh
echo "<body>"
for f in $*
do
  echo "<img src='$f'><br>"
done
echo "</body>"
```

これを `imghtml` という名前で保存したとして、使い方は次のような感じになります (これで `test.html` をブラウザで開くと 3 つの画像が並んだページが見られる)。

```
...$ ls IMG
a.jpg   a.txt   b1.jpg   c3.jpg
...$ ./imghtml IMG/*.jpg
<body>
<img src='IMG/a.jpg'><br>
<img src='IMG/b1.jpg'><br>
<img src='IMG/c3.jpg'><br>
</body>
...$ ./imghtml >test.html
```

演習 3 ディレクトリ`$home/bin` を作成し、自分の実行パスに含めるようにしなさい。続いて、`psme` をそこに置き、(`rehash` を実行した後) 普通のコマンドとして使えるようになっていることを確認しなさい。OK なら、以下のことをやってみなさい。

a. `imghtml` を作成してパスに入れ、実際に自分が持っている画像を指定して HTML を生成して動かせ。さらに画像だけだとそっけないので、`<hr>`(横線の指定) を追加して区切るようにしてみよ。

b. 複数のテキストファイルを区切りの横線「----------」で区切って連続して出力するスクリプト delimtxt を作成してみよ。たとえば2つの(それぞれ1行の)ファイルを与えると次のようになることを想定している。

```
...$ ./delimtxt a.txt b.txt
----------
This is a.
----------
This is b.
----------
...$
```

c. 数値を複数指定すると、その和を出力してくれるスクリプト sum を作成してみよ。次のような実行例になることを想定している。

```
...$ sum 1 3 5
9
...$
```

ヒント: ループに入る前に「sum=0」を実行しておき、ループ変数が i だとして、ループ内で「sum=‘expr $sum $i‘」を実行すればできる。expr の意味については自力で調べること。

d. 自分があると便利だと思うスクリプトを構想し作りなさい。

課題 8A

今回の課題は「演習1」「演習2」「演習3」に含まれる小問(合計で10個)の中から1つ以上を選択し、結果をレポートとして報告して頂くことです。以下の内容がこの順に含まれるようにしてください。

- 冒頭に題名「コンピュータリテラシレポート＃8」、学籍番号、氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も別途書く。)
- 課題の再掲を書く(どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容(やったこととその結果)を書く。
- 考察(課題をやった結果自分が新たに分かったことや考えたこと)を書く。
- 以下のアンケートに対する回答。

Q1. さまざまなフィルタやその機能について、どれくらい知っていましたか。新たに知って面白かったことは何ですか。

Q2. シェルの様々な機能やシェルスクリプトについてどう思いましたか。

Q3. リフレクション(今回の課題で分かったこと)・感想・要望をどうぞ。

なお、課題はグループでやって構いません。その場合も、(メンバー氏名を明記した上で)レポートは必ず各自で執筆してください。レポート文面が同一(コピー)と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。

＃9　マークアップによるテキスト整形

今回の目標は次の通りです。

- マークアップ方式によるテキスト整形の概念を理解する — ワープロソフトよりも効率よく作業できることを理解しておくと、効率を高める工夫がしやすくなります。
- LaTeX による文書の作成方法について理解する — 理系では LaTeX によってレポート・論文を書くことが普通です。
- 数式や表なども含む、LaTeX のさまざまなマークアップについて学ぶ — 数式や表はレポート・論文で非常によく使います。

9.1　文書作成

9.1.1　テキストと文書の違いと位置付け

コンピュータは情報を処理する装置ですが、人間の知的活動 (情報) のアウトプットは大半が**文書** (書籍、レポート等) です。このため、文書を作成する**ワープロソフト**や**文書整形系** (formatter、後述) は、コンピュータで最も活躍するアプリケーションの 1 つです。ここで、**テキスト**と文書の違いについて触れておきます。テキスト情報は「文字で表された情報」で、テキストを扱うソフトの代表テキストエディタです。エディタでは文字の並びは自由に修正できますが、作成したファイルの文字はどれも同じ大きさで、美しく読みやすいとは言えませんね。我々は大量の情報を取り入れるために文書を読むので、それが美しく読みやすいことは大切です。その具体例に入る前に、次の問いを考えてみましょう。

- 美しい文書とはどういうものを言うのか?
- なぜ美しい文書が望ましいのか?

1 番目の問いに対して「文字がきれいな形である」「多様な字形 (フォント) が使われてる」などが浮かんだ人もいそうですが、文字がきれいでもでたらめな規則で並んでいたら嬉しくないですね。

そこで 2 番目の問いですが、なぜ美しい文書が望ましいのでしょう? それは、文書が美しいと、内容を読み取る効率がよいからです。具体的には、文字の並び方が読みやすく、「ここは見出し」「これは図」などの構造が把握しやすく、必要な箇所が探しやすいこと、つまり次のことです。

- 文書の構造が読み手に的確に伝わること。

逆にいえば、「美しい文書」では、内容であるテキスト (文字) に加え、文書のどこが何であるの**付加情報**も含まれています。これによって「見出しだから大きく」などの処理が可能になるわけです。

9.1.2　文書整形のアルゴリズム

ワープロソフト (や文書整形系) が「どのように」各文字を紙面に配置しているかを言葉で説明できますか? たとえば、初期のワープロソフトなどでは「原稿用紙」モデル、つまり画面や紙の上に縦横のサイズが決まったます目があり、そこに 1 文字ずつ文字を詰めて行く、というアルゴリズムが使われていました。それだとどんな美しくないことが起きるか分かりますか?

- 禁則処理 (行頭に「。」などが来ないための処理) の結果、右端が「でこぼこ」になる (図 9.1 上)。

- 英字を日本語と同じ文字間隔で詰めるとえらく間延びになる。かといって日本語 1 文字ぶんに英字 2 文字を詰めると窮屈 (しかも英字が奇数だと半分のアキが…)。
- 箇条書きなど字下げの段落で文字詰め幅を変更すると空白がずれて悲惨になる (図 9.1 下)。

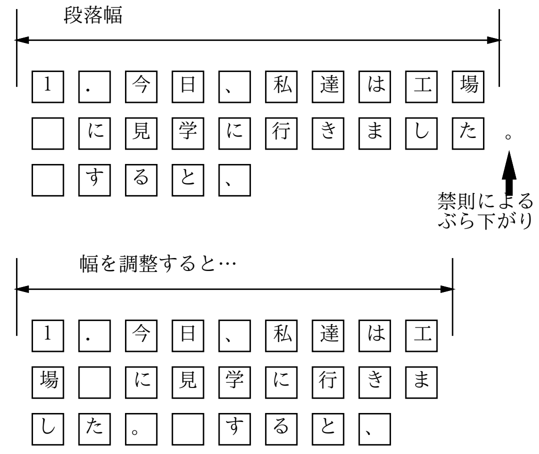


図 9.1: 「原稿用紙方式」の困った点

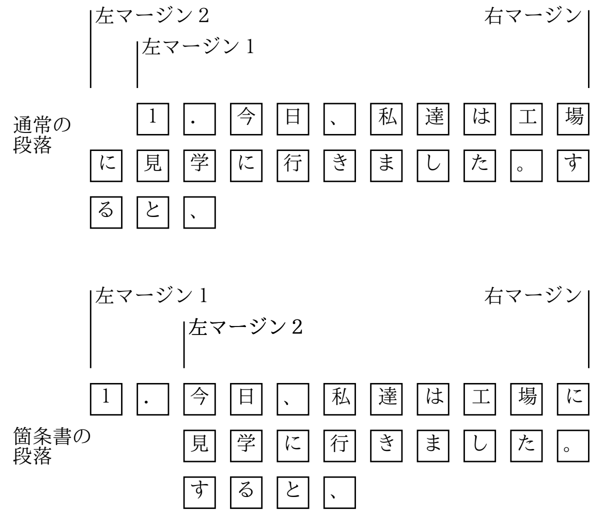


図 9.2: マージン指定による段落の詰め合わせ

今日のソフトではこのようなことはなく、段落は「1 行目の左マージン」「2 行目以降の左マージン」「右マージン」の位置で形が指定されており、文字を普通に打って行くことでこれらのマージン内で自動的に文字が詰め合わされて行くようになっています (図 9.2)。ただし、これをちゃんと使うには、段落の形を変える時に「ルーラ」を出してマージンを設定する必要があります (これを知らずに字下げを空白で調整している人もまだ多いようです)。

さらに禁則なども考慮して整形するため、現在は次のようなアルゴリズムが使われます (図 9.3)。

- 段落ごとに「詰め合わせる幅」があり、それぞれの文字は「四角い箱」と考えてその箱を幅一杯まで詰めていく。
- 文字と文字の組み合わせごとに適切な「あき」が変わってくるので、組み合わせごとに適切な「にかわ (のり)」を詰めていく。
- 禁則処理などでその位置で行かえできないなら、少し余分に詰めるか詰めたものを取り除く。

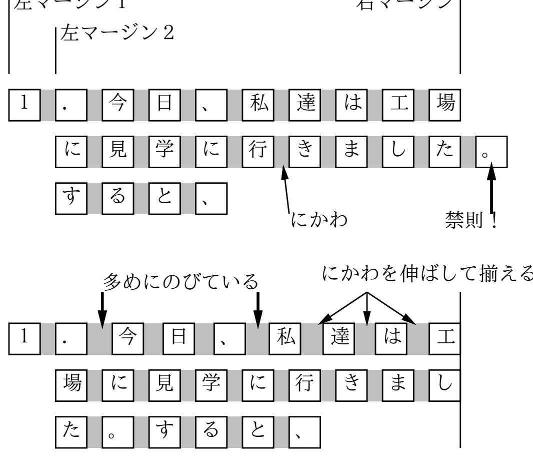


図 9.3: 整形のアルゴリズム

- 右端が揃うように全体を詰める/伸ばする (この時にかわ部分が伸び縮みする)。
- すべてのにかわが均等に伸び縮みするわけではなく、「(」の左、「)」の右、「。」の右など、伸びてもおかしくないところが余分に伸びるようにする。
- 各行が出来てくると、今度はページに行を上から詰めていく。ここでも「にかわ」がはさまれ、ページの切れ目の禁則 (節タイトルがページ下端に来ない等) に応じて同様の処理が行われる。

ここで重要なのは、「文字どうしのアキ (にかわの幅や伸び)」「段落間のアキ」「マージンの値」などをどう調整すれば見やすい文書になるか、ということです。これについては、長い歴史のある印刷・出版界が多くのノウハウを持っており、それを駆使して読みやすい書籍を作っています。そこで、LaTeX などの整形システムでも、これらのノウハウを借りて来てパラメタを調整することで、できるだけ見やすい整形となるように努めているわけです。

9.1.3 見たまま方式とマークアップ方式 ex

美しい文書を作成するためには、テキストと付加情報をともに入力したり修正するなどの必要があることまでは分かりました。ではそれを具体的にどのようにして行なったらよいでしょうか? コンピュータの世界では、その方法として次の 2 種類が考えられ、現に使用されています (図 9.4)。

- 見たまま方式、**WYSIWYG**(What You See Is What You Get — 「あなたが見るものがあなたの得るものである」) 方式。この方式では、実際にプリンタで印刷したイメージにできるだけ近いものを生成し、画面表示します。そしてマウスやメニューなどを用い、画面上に見える文書を対象に修正を施します。配置やフォントを変更したり、文章を直したりすると、その結果は直ちに画面に反映されます。Word などもこの仲間です。
- **マークアップ** (markup) 方式 — テキストエディタを用い、ファイル中にテキストに混ぜて特別な「印」(マークアップ) を入れ、整形系と呼ばれるソフトウェアがこれに従ってレイアウトを行います。今日のマークアップは「ここは見出し」「ここは引用」など、個々の部分の意味を指定する**意味づけ方式** (semantic encoding) を採用しています。マークアップではコマンドによって細かい制御が行えますが、レイアウトをチェックする際に「整形+プレビュー」操作をおこなう必要があります。

世の中のワープロソフトでは見たまま方式が主流なのに、ここではマークアップ方式を取り上げるわけですが、その理由について説明しておきましょう。

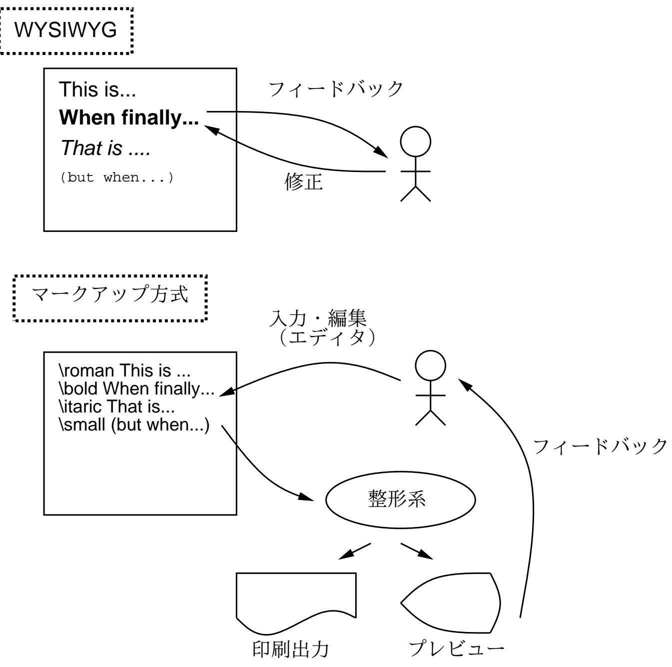


図 9.4: WYSIWYG 方式とマークアップ方式

- 見たまま方式は、個所ごとに「このスタイル」という指定を繰り返し行い、そのマウスやメニュー操作は時間が掛かります。マークアップなら、どこはどういうマークアップをするか覚えればあとは「打ち込む」作業の一部であり、時間が掛かりません。
- 見たまま方式は (多くの場合)「こういう体裁に」ということを人間がデザインし制御するので、センスがないと美しくできません。意味づけ方式では、スタイルは標準化されているので、そのようなことに頭を悩ます必要はありません。[1]
- 見たまま方式は、1 つの体裁に合わせて文書を作るため、後で体裁を変更するのは大変です。意味づけ方式なら、整形用スタイルを複数用意することで、多様な体裁に対応できます。
- 見たまま方式は、ファイル形式が特定ソフトに依存しがちで、コンテンツの流用が難しくなります。[2] マークアップはもともと全部テキストファイルなので、(マークアップの統一的な書き換えなどの処理を行うなどでして) 流用が容易です。

「文書を作るのにコマンドを覚えるなど耐えられない」「そんな面倒な方式は旧態依然だ」と思ったでしょうか? しかし実は、大量に文書を作成するプロほど、LaTeX のような整形系をメインに使っています。それは大量に文書を作る人にとってはコマンドの方が楽で効率的だからです。また、LaTeX はスタイルのチューニングが徹底しているので、「おまかせ」で比較的美しい文書ができます。

9.1.4　LaTeX を動かしてみる

ではまず LaTeX を「とにかく」体験してみて頂きます。コマンドの書き方は後回しにして、先にサンプルファイルを「そのまま」整形してみましょう。LaTeX の入力ファイルは「`.tex`」で終わる名前をつけます。以下ではそれが「`sample.tex`」であるものとして説明します。整形のためのコマンドは `platex` ですので、入力ファイルを指定してこれを起動します。

```
...$ platex sample.tex
This is e-pTeX, Version 3.1415926-p3.4-110825-2.6 (utf8.euc) (TeX Live 2013)
```

[1]WYSIWYG でもスタイルシート機能を活用すれば意味づけ方式を GUI でやる形になりますからスタイル設計者次第にできますが、その分マークアップに近くなり難しくなるので使わない人も多いです。
[2]テキストファイルに変換すれば他のソフトに持っていけますが、付加情報は失われてます。

```
 restricted \write18 enabled.
entering extended mode
(./sample.tex
pLaTeX2e <2006/11/10> (based on LaTeX2e <2011/06/27> patch level 0)
Babel <3.9f> and hyphenation patterns for 78 languages loaded.
(/export/opt2/texlive/2013/texmf-dist/tex/platex/base/jarticle.cls
Document Class: jarticle 2006/06/27 v1.6 Standard pLaTeX class
(/export/opt2/texlive/2013/texmf-dist/tex/platex/base/jsize12.clo))
(./sample.aux) (/export/opt2/texlive/2013/texmf-dist/tex/latex/base/omscmr.fd)
[1] [2] [3] (./sample.aux) )
Output written on sample.dvi (3 pages, 7524 bytes).
Transcript written on sample.log.
```

自分でファイルを作った際は、エラーがあると途中で「?」というプロンプトが出て止まります。その時は「x[RET]」と打って実行を終わらせ、エラーメッセージを見て間違いを直し、再度実行してください。整形が成功すると.dviで終わる名前のファイル(この場合だとsample.dvi)ができます。

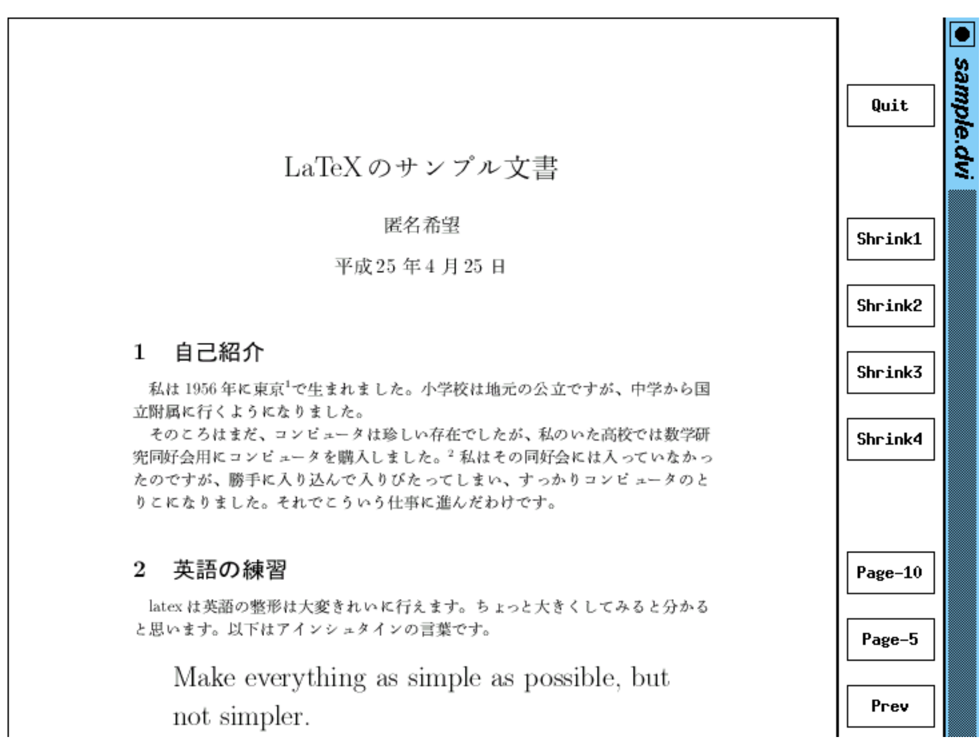


図 9.5: pxdviによるプレビュー

整形結果はまず画面で確認します。それにはプレビューアpxdviを「pxdvi sample.dvi &」で起動します。すると、図9.5のような窓が現れます。右側に縮尺やページめくりを指定するボタンがあるので、これらをマウスで操作し、各ページが意図どおりか確認します。なお、他の人に渡したりプリンタに出すには「dvipdfmx sample.dvi」によりPDF形式を生成してください。

大変だったでしょうか? 確かに最初はとっつきにくいですが…では、先頭の行を次のように直すとどうなるか、試してみてください。

```
\documentclass[12pt,a4j]{jarticle} ←文字サイズ変更
\documentclass[twocolumn,a4j]{jarticle} ←2段組み
```

このような取り換えが簡単なのが意味づけ方式の利点だということが理解して頂けるかと思います。

演習 1 `sample.tex` を入手して整形し、結果を画面で確認してみなさい。うまく行ったら、冒頭の documentclass 指定について次の変更を 1 つ以上 (できれば全部) 行ってみなさい。

a. twocolumn 指定の有無による違いを比べてみる。
b. フォントサイズ指定 (`9pt`、`10pt`、`11pt`、`12pt` のどれか) を変えた時の違いを調べる。
c. 紙サイズ指定 `a5j`、`b5j`、`a4j`、`b4j` のどれか) を変えた時の違いを調べる。

いずれも、指定を変更した時にきれいに見えるようにどのような工夫が見られるか、文字サイズに比べて整形幅が狭くなった場合に整形時にどのようなメッセージが出てどのような整形結果になるかを見ること。

9.2　文書整形系 LaTeX

9.2.1　文書の基本構造 ex

ここでは、意味付け方式の文書整形システム **LaTeX** を実例中心で一通り解説します。まず、LaTeX の文書に最低限必要な基本構造の例を見てみます。次のものは、一つの完結した LaTeX 文書です (内容も読んでください)。

```
\documentclass{jarticle}
\begin{document}
ここに示すように、LaTeX の文書はまずこの文書がどんなスタ
イルの文書であるかを宣言する部分から始まる。スタイルの例
としては「記事 (jarticle)」、「本 (jbook)」、「報告
(jreport)」などがある。ここでは一番簡便な jarticle を例に
使用している。これに加えて字の大きさ、図形の取り込み機能
などをオプションとして指定できる。

続いて「文書開始」の宣言があり、この中に文書の本体が入る。
この部分には様々な記述が可能だが、一番簡単にはここに示す
ように段落ごとに 1 行あけて次々に文章を書いて行くだけでも
地の文が普通にできる。つまり、「特に指定がない」ならば
「地の文」である。

あとは文書の本体が終わったら最後に必ず「文書終了」の宣言
がある。最低限必要なのはこれだけである。
\end{document}
```

ここに示すように、LaTeX の文書はまずこの文書がどんなスタイルの文書であるかを宣言する部分から始まる。スタイルの例としては「記事 (jarticle)」、「本 (jbook)」、「報告 (jreport)」などがある。ここでは一番簡便な jarticle を例に使用している。これに加えて字の大きさ、図形の取り込み機能などをオプションとして指定できる。

続いて「文書開始」の宣言があり、この中に文書の本体が入る。この部分には様々な記述が可能だが、一番簡単にはここに示すように段落ごとに 1 行あけて次々に文章を書いて行くだけでも地の文が普通にできる。つまり、「特に指定がない」ならば「地の文」である。

あとは文書の本体が終わったら最後に必ず「文書終了」の宣言がある。最低限必要なのはたったこれだけである。

図 9.6: pLaTeX の出力

これを LaTeX に掛けて整形すると図 9.6 になります。ここで少し補足すると、まず宣言 (指令) は

```
\指令名 [オプション指定]{パラメタ}
```

の形をしています。オプション指定がないときは [...] はなく、またパラメタがないときは{...}も不要です。ということは「\」はそのままでは文書に含められないわけです。LaTeX では、このような特別な (そのままでは使えない) 記号として次のものがあります。[3]

```
# $ % & ~ _ ^ \ { }
```

9.2.2 表題、章、節 ex

本文だけの文書では見づらいので、表題と章立てをつけましょう。その場合の例を示します。

```
\documentclass{jarticle}
\begin{document}
\title{あなたがつけた表題}
\author{書いた人の名前}
\maketitle

\section{表題について}
表題をつけるには title、author など必要な事項を複数指定し
た後、最後に maketitle というとそれらの情報をもとに表題が
生成される。その際指定されなかった事項は出力されないかま
たは適当なものが自動生成される (たとえば日付を指定しない
と整形した日付が入る。)

\section{章立てについて}
ここにあるように section 指令を使って各節の始まり、および
その表題を指定する。節の中でさらに分ける場合には section
というのも使え、さらに subsection というのまで可能である。
ちなみに jbook/jreport スタイルでは section の上位に chapter
があるが jarticle では section からである。ところで、節番号
etc. は自動的に番号付けされるのに注意。

\section{より細かいことは}
より細かいことは、参考書を参照してください。
\end{document}
```

これを整形すると、文書の冒頭にタイトル、著者、日付 (整形した日付) [4] がそれなりの形式で用意され、見出しも前後にアキを取ってそれなりのフォントで出力されます。なお、section より小さいレベルの見出しとして subsection、subsubsection も使えます。

9.2.3 いくつかの便利な環境 ex

表題と章建てがあれば普通の文書がかなり書けますが、全部地の文ではめりはりがないですね。それに、プログラム例など行単位でできているものまで詰め合わされては困ります。このように、部分的にスタイルが違う範囲は次のような形 (**環境** — environment — と呼びます) を使います。

---

[3] この他に<、>など記号自体に特別な意味はないのですが、TeX の標準フォントの関係で出力すると別の字になってしまうものがいくつかあります。

[4] 日付を自分で記入したければ「\date{ 日付表記 }」を author の後あたりに入れます。

```
\begin{環境名}
....
....
\end{環境名}
```

では、よく使う環境について、説明していきましょう。まず、**verbatim 環境** (そのまま) というのは文字通り入力をそのまま整形せずに埋め込みます。たとえば

```
\begin{verbatim}
This is a pen.
  That is a dog.
\end{verbatim}
```

は、つぎのような出力になります。プログラム例などを示すのに最適です。

```
This is a pen.
  That is a dog.
```

ちなみに、独立した行にする代わりに、文章のなかに一部「そのまま」を埋め込みたい場合もあります。そのような時には verbatim の類似品で`\verb|....|`という書き方を使うことができます。これで縦棒にはさまれた部分がそのまま出力できます。中に縦棒を含めたい時は、両端に縦棒以外の適当な記号を使います。なお、`verbatim` も `verb` も脚注 (後述) の中には入れられません。

つぎに、**itemize 環境** (箇条書き) について説明しましょう。この場合は、環境のなかに複数「`\item ....`」というものが並んだ格好になっていて、その一つずつが箇条書きの 1 項目になります。たとえば

```
\begin{itemize}
\item あるふぁはギリシャ文字の一番目です。
\item ベータはギリシャ文字の二番目です。
\item ガンマはギリシャ文字の三番目です。
\end{itemize}
```

は、つぎのような出力になります。

- あるふぁはギリシャ文字の一番目です。
- ベータはギリシャ文字の二番目です。
- ガンマはギリシャ文字の三番目です。

次に、この itemize を **enumerate 環境** (数え上げ) に変更すると、出力の際に項目ごとに 1, 2, 3... と自動的に番号付けされるようになります。入力はほとんどまったく同じだから出力のみ示します。

1. あるふぁはギリシャ文字の一番目です。
2. ベータはギリシャ文字の二番目です。
3. ガンマはギリシャ文字の三番目です。

点や番号でなくタイトルをつけたい場合には **description 環境** (記述) を使います。これは

```
\begin{description}
\item[あるふぁ] これはギリシャ文字の一番目です。
\item[ベータ] これはギリシャ文字の二番目です。
\item[ガンマ] これはギリシャ文字の三番目です。
\end{description}
```

のように、各タイトルを [] の中に指定するもので、上の整形結果は次のようになります。

あるふぁ これはギリシャ文字の一番目です。

ベータ これはギリシャ文字の二番目です。

ガンマ これはギリシャ文字の三番目です。

この他にもいくつか環境がありますが、とりあえずこれくらいで十分使えると思います。

9.2.4 脚注 ex

脚注の作り方は、単に好きなところに`\footnote{.....}`という形で注記をはさんでおけばそれがページの下に集められて脚注になり、そこへの参照番号は自動的につけられます。[5]

9.2.5 文字サイズ ex

文字サイズ指定は、`{\LARGE 大きく}`のように書くと大きくなります。このように LaTeX では`{...}`が範囲を区切り、字体や文字サイズを変更した効果はその範囲を出ると終わります。文字サイズとしては huge、LARGE、Large、large、normalsize、small、footnotesize、scriptsize、tiny が指定できます。字体は`\textbf{Bold}`で **Bold**(ボールド体)、`\texttt{Typewriter}`で `Typewriter`(タイプライタ書体)、`\textit{Italic}`で *Italic*(イタリック)、そして`\textrm{Roman}`で Roman(立体 — 普通の書体) です。

9.2.6 表 ex

表は情報を整理して伝えられまです。LaTeX では表は **tabular 環境**で作り出します。その先頭では、

- 表のカラム数
- それぞれのカラムを左/中央/右揃えのどれにするか
- 各カラムの境界および左右に罫線を引くかどうか

を文字列で指定します。1 つのカラムごとに揃え方を `l/c/r` のうち 1 文字で指定し `l/c/r` の文字数がカラム数になります)、これらの文字の間や左右端に「`|`」を挿入するとそこに縦罫線が入ります。具体例で見てみましょう (**center 環境**は表全体を中央揃えするものです)。

```
\begin{center}            ←見ばえのため
\begin{tabular}{c|ll}
AND 演算 & 0 & 1 \\
\hline                    ←横罫線
0        & 0 & 0 \\
1        & 0 & 1 \\
\end{tabular}
\end{center}
```

上のようにすると、次のように表示されます。見れば分かる通り、表の内部では「`&`」がカラムの区切り、「`\\`」が 1 行終わり、「`\hline`」が横罫線を表します。

AND 演算	0	1
0	0	0
1	0	1

[5]たとえば、こんな具合になりますね。

ところで、表の一部について「セルを結合」したいことがよくあります。横方向の結合はmulticolumnコマンドを使ってできます (縦結合もできますがここでは省略)。たとえば、先の例の2つ並んだ「0」を1個にまとめたい場合は、0の行を次のようにします。

```
0 & \multicolumn{2}{|c}{0} \\ ←2セル結合、内容は「0」
```

結果は次のようになります。

AND演算	0	1
0	0	
1	0	1

9.2.7 数式 ex

TeX属の整形系はもともとは**数式**を美しく打ち出したい、という目的のもとに開発されたものなので、数式機能がとても充実しています (そのため凝り出すと大変ですから、ここでは簡単に説明します)。まず、数式を文中にはさむときは$で囲みます。たとえば、

```
…と書くと$x^2 - a_0$のような具合になります。
```

と書くと $x^2 - a_0$ のような具合になります。ここで出てきたように、肩字は^、添字は_で表せます。その他、数学に出てくる記号はたとえば

```
\equiv \partial \subset \bigcap \cdots \sum \int
```

のように書くと $\equiv \partial \subset \bigcap \cdots \sum \int$ のように出てきます。

また、文中ではなく独立した行にしたければ、$...$の代わりに\[...\]ではさむか、\begin{displaymath} … \end{displaymath}で囲みます (どちらでも同じ意味)。たとえば

```
…と書くと\[f(t) = \sum_{j=1}^m a_j e^{i\lambda_j} t\]になります。
```

と書くと

$$f(t) = \sum_{j=1}^m a_j e^{i\lambda_j} t$$

になります。このように、肩字や添字のグループ化には{}を使います (丸かっこは数式本体のために使います)。もう1つよく使うのは分数で、$\frac{a}{b}$は $\frac{a}{b}$ となります。もっと網羅的な記号一覧などはTeXの入門書などを参照してください。

演習2 LaTeXの数式機能を使って、次のようなものを1つ以上 (できれば全部) 指定してみなさい。

a. $$a_0 + a_1 + \cdots + a_n$$

b. $$\sum_{i=1}^N \int_0^i f(x)dx$$

c. $$x = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3}}}$$

できれば、それぞれについてもう少し複雑な「技」を追加してみるとなおよいです。

演習3 LaTeXを使って自分の自己紹介の文書を作成してみなさい (フィクション可)。タイトルと章立ては必ずおこなうこと。加えて次の機能から1つ以上 (できれば全部) を活用すること。文書内容は「あああ」みたいな意味のないものではなく、それらの機能を使う必然性のある形で使ってください。

a. 箇条書きや引用。できれば、箇条書の中に箇条書を入れて何が起きるか観察できるとなおよいでしょう。[6]
b. 脚注。できれば、脚注の中で使えない機能として何があるか試してみるとなおよいでしょう。
c. 表。できれば、複数カラムの結合も使ってみられるとなおよいでしょう。

[6]引用文は \begin{quote}…\end{quote} の内側に段落を1つ以上入れて表します。

課題 9A

今回の課題は「演習 1」「演習 2」「演習 3」に含まれる小問 (合計で 9 個) の中から 1 つ以上を選択し、結果をレポートとして報告して頂くことです。ただし、「演習 3」から必ず 1 つ以上選ぶこと (演習 1、演習 2 の内容を含める場合はこの中に一緒に書き込む)。そして、LaTeX による整形結果を PDF ファイルにして、LMS の「assignment ＃ 9」の箇所からアップロードしてください。以下の内容がこの順に含まれるようにしてください。

- 題名「コンピュータリテラシレポート＃ 9」、学籍番号と氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も脚注などで別途書く。)
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 以下のアンケートに対する回答。

  Q1. 普段どのくらい文書を作成していますか。またそのときに使うソフトや心がけていることなどを教えてください。
  Q2. LaTeX のようなマークアップ型の文書作成系についてどのように感じましたか。
  Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

なお、課題はグループでやって構いません。その場合も、(メンバー氏名を明記した上で) レポートは必ず各自で執筆してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。

＃10　グラフィクス/図と表

今回の目標は次の通りです。

- 画像の基本概念、ピクセル画像とベクター画像について理解する— 画像は実験の写真や図などの形で多数使うので知っておくことで後で有効に作業できます。
- PPM、EncapsulatedPostScript などのファイル形式について学ぶ— 画像ファイル形式は多数ありますがピクセルとベクターの例を 1 つずつ詳しく知っておくと他に応用できます。
- LaTeX 文書における図や表の扱いを理解する — レポートに図や表を入れられることは必須のスキルです。

10.1　ピクセルグラフィクス

10.1.1　画像の表現 ex

インターネット上で最も多く使われている情報の形式はテキスト情報ですが、2 番目に多いのは**画像** (イメージ) ですね。画像の存在しないネットはほとんど考えられないと思います。ここでは画像とその表現について取り上げます。

画像を入力する装置の代表は**デジタルカメラ**や**スキャナ**、画像を出力する装置の代表は**ディスプレイ**と**プリンタ**ということになるでしょう。ではデジタルカメラやスキャナはどのような形で画像を取り込み、ディスプレイやプリンタはそれをどのようにして復元しているのでしょうか。

まず**モノクロ画像**から考ます。画像は平面的な広がりを持つ、空間的に連続したものです。これをデジタル化して取り込むには、まず平面を縦横のます目に十分細かく区切ります。続いて、それぞれのます目の明るさを電圧や電流に変換する素子を使って測り、その明るさを決まった範囲の整数値として取り込みます (**AD 変換** — Analog-Digital 変換)。つまり、画像は縦横に並んだ多数の点の集まりであり、それぞれの点ごとに、ます目の範囲内の元画像の明るさを**サンプリング** (sampling、計測し代表値を取ること) および**量子化** (段階にあてはめ何番目かの値にすること) した値を持つわけです。この「点」のことを**ピクセル** (pixel) と呼びます (図 10.1)。

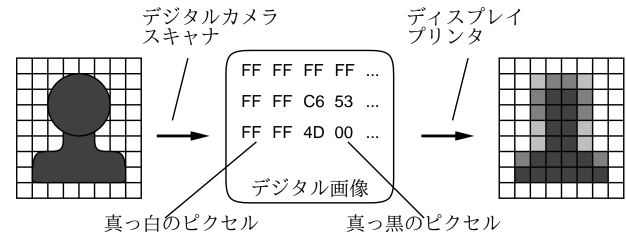


図 10.1: デジタル画像の原理

ディスプレイはこの点の集まりを画面に表示し、それぞれのピクセルごとにその値に応じた明るさ/暗さで光らせます。また、プリンタは紙の上にそれぞれの点の集まりを色素を使って定着させますが、ピクセルごとにその値に応じた量の色素を定着させます。そこで、人間がこれらを見ると元の画像 (をデジタル化して復元したもの) が見られるわけです。

カラー画像の場合も画像がピクセルの集まりであることは同じですが、色の情報を取り込むために、各ピクセルごとに赤 (Red)、緑 (Green)、青 (Blue) の光の 3 原色のフィルタを通して 3 つのサンプル値を取り込みます。

つまり、各ピクセルは 3 つの値の組 (**RGB 値**) として表現します。広く使われるのは、RGB 値の各色ごとに 8 ビット (0〜255 の値) を使い、1 ピクセルあたり 24 ビット (3 バイト) でカラー画像を表す方法です。これを **24 ビットカラー** 24 びっとからーといいます。カラーディスプレイは各ピクセルごとに RGB の 3 つの光る点を制御し、それぞれを RGB 値に応じた明るさで光らせますし、プリンタは各ピクセルごとに 3 色 [1] の色素を配合して各ピクセルが RGB 値に対応する色になるよう制御します。

10.1.2　ピクセルグラフィクスの得失 ex

画像は多数のピクセルの集まりですから、画像を加工することは、個々のピクセルごとにその RGB 値を変化させてやることで行えます。また、マウスやタブレットペンなどを使って「お絵描き」をする場合も、描画範囲上でマウスポインタがなぞった部分のピクセルの色を変化させることで「インク」のようにそこの部分の色を変えることができます。

このように、画像を構成するピクセルを直接取り扱うようなグラフィクスの方式を一般に**ピクセルグラフィクス**と呼びます。ピクセルグラフィクスに基づく作画ソフトのことを**ペイントソフト** (比較的簡単な絵を描く場合に使う) や**フォトレタッチソフト** (写真などの細密な絵を加工するような場合に使う) と呼びます。代表的なフォトレタッチソフトとしては Photoshop、Gimp などがあります。

ピクセルグラフィクスに基づく処理には、次のような利点があります。

- ○ 画面の表示能力に見合う細かさと色数のデータならその画面で表せるすべての画像が表現可能。
- ○ 「ぼかし」「にじみ」などの効果が使え、中間的な色合いや独特のタッチを持った絵が作れる。

その一方で、次のような弱点もあります。

- △ ピクセル数を大きくすると (これは絵を大きくする場合だけでなく、点の取り方を細かくする場合も含まれる)、ファイルも巨大になりやすい。
- △ 描いた絵を拡大したり回転するなどの加工に弱く、大きくするとぎざぎざが目立ったりする。
- △ ある場所に絵を「描いてしまう」と、そこのピクセルの色を設定してしまうので、後から消したり動かしたりが難しい。(消しゴムで消すというのは結局「背景で塗っている」のと同じ。そして動かすと後が「空白」になる。) [2]

最後の弱点に対しては、画像を複数の**レイヤ** (層) に分けて扱うことである程度対処可能です。レイヤ機能を持つソフトでは、透明なシート (レイヤ) を複数使ってそれぞれに絵を描き、それらを全て重ねて眺めたものが最終的な絵になります。重ねる順番や位置などは描いた後でも変更できますし、描き損なった場合はそのレイヤだけ消してやり直せば済みます。

ここまではソフトの機能を中心に説明してきましたが、作成した画像は最終的にはファイルに出力します。ピクセルグラフィクスのファイル形式は多数あり、またソフトごとにそのソフトで扱いやすい独自形式を持ったりしますが、共通に使われる代表的なものを挙げておきます。

- **BMP**(Windows Bitmap) — Windows 固有の、圧縮のないピクセル画像ファイル。あまり使われることはない。
- **PBM**(Portable Bitmap) — Unix 文化で普及している、圧縮のないピクセル画像ファイル。形式が簡単なので後の実習で使う。

---

[1] 正確には、印刷の場合は「真っ黒」を表現するため 4 色目として黒の色素を持たせます。また 3 色も印刷の性質上、シアン、マゼンタ、イエロー (色の三原色) の組合せを使います。

[2] ただし、間違って塗った場合には、元の状態を記憶しておくことである程度戻すことは可能。大量には難しい。

- **GIF** WWW で最初に使われた圧縮のある画像ファイル形式。色数が最大 256 色という制約があるが、アイコン等には使いやすい。
- **JPEG** — WWW で GIF に続いて普及した、写真などの画像に適した、損失のある圧縮を主に用いる画像形式。[3]
- **PNG** — WWW で最も新しく普及した形式。GIF の 256 色という制約をなくし、JPEG と異り損失のない圧縮を用いている。

Web で画像ファイルを使う場合は、最後の 3 つ (GIF、JPEG、PNG) のどれかの形式を使います (どのブラウザでも対応しているため)。

10.1.3 PBM 画像を作成する ex

PBM 形式の画像には「テキスト形式」つまり普通の文字だけで記述する形式が用意されています。ここで画像の原理を実感していただくために、テキストエディタでこの形式を打ち込んでみます。最初は一番簡単な **2 値画像** (モノクロで真っ白と真っ黒しかない) を作ってみます。sol 上で Emacs を起動し、`t1.pbm` という名前で次の内容から成るファイルを作成し保存してください。

```
P1 6 6          ← P1 は「2 値画像」、6 6 は幅と高さ
1 1 0 0 1 1    ←ピクセルの値 (1 か 0) が 36 個必要
1 1 0 0 1 1    ←値の間には空白か改行が必要
0 0 0 0 0 0
0 0 0 0 0 0
1 1 0 0 1 1
1 1 0 0 1 1
```

次に「`gimp t1.pbm &`」により Gimp を起動します。窓が複数開きますが、小さい画像が表示されている窓を選び、拡大率を最大にしてください。図 10.2 のように確かに白黒の画像になっていることが分かります。

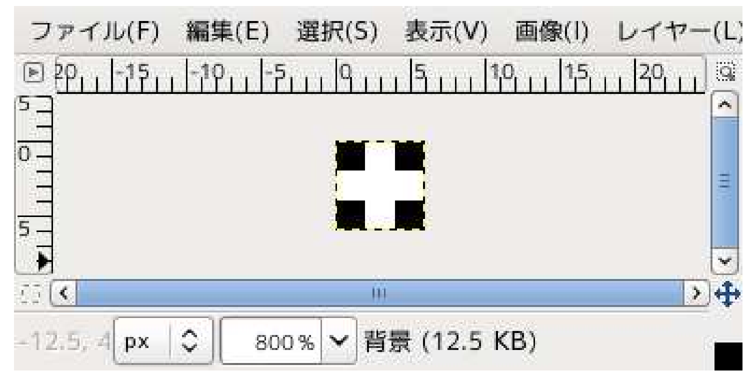


図 10.2: 2 値画像を表示する

しかし色がないとつまらないですね。そこで次はカラー画像にしてみましょう。ファイル名としてこんどは「`t2.ppm`」を指定し、次の内容を打ち込んでください。こんどは RGB で指定するので数値の数が 1 ピクセルあたり 3 個になります (全部で幅×高さ×3 個の数値が必要)。RGB の値は 0〜255 の範囲で指定します。

[3]損失のある圧縮とは、圧縮したものを展開したときに完全に元のデータとは一致しないような圧縮方式をいう。その代わり、圧縮率を高くしやすい。JPEG では圧縮率を指定することで「ファイルサイズが小さいが品質が落ちる」「ファイルサイズが大きいが品質が高い」などの制御ができる。

```
P3 6 6 255   ← P3 はカラー画像、255 は RGB 値の最大数
0 0 255 0 0 255 0 0 255 0 0 255 255 0 0 255 0 0
0 0 255 0 0 255 0 0 255 0 0 255 255 0 0 255 0 0
0 0 255 0 0 255 0 0 255 0 0 255 255 0 0 255 0 0
0 0 255 0 0 255 0 0 255 0 0 255 255 0 0 255 0 0
0 0 255 0 0 255 0 0 255 0 0 255 255 0 0 255 0 0
0 0 255 0 0 255 0 0 255 0 0 255 255 0 0 255 0 0
```

こんどは Gimp で開いてみると、色がついている。元データを見ると分かるように、左側の 4 ピクセルは (0,0,255) つまり真っ青、右側の 2 ピクセルは (255,0,0) つまり真っ赤になっている (左から RGB の順なので)。

図 10.3: カラー画像を表示する

演習 1 上の 2 つの例を打ち込んで Gimp(フリーソフトなので Windows 用や Mac 用も無料で入手可能) で表示してみなさい。うまくいったら、次の演習をやってみなさい。

a. 白黒でもカラーでもよいので、もう少し大きな画像に挑戦してみる (PBM のテキスト形式では数値は並んでいる順だけに意味があるので、扱いやすくするために好きなところで行をかえてよい)。ただし、最初にどのような画像にするか紙にスケッチし、その形を再現するように作ること。

b. 「色が徐々に変わって行く」「図形のふちがぼけている」「半透明な図形が重なっている」などの効果から 1 つ選び、カラー画像として作成してみよ。ただし、最初にどのような画像にするか紙にスケッチし、その形を再現するように作ること。

c. (自由課題) テキスト形式の白黒またはカラー画像を作るやり方で、自分の好きな画像を作ってみなさい。どのような画像であるかはあなたに任されるが、例題で示したものよりは「美しい」ことが期待される。エディタで打ち込む以外に、プログラムを書いて出力するなどの方法を用いてもよい。

10.2　ベクターグラフィクス

10.2.1　ベクターグラフィクスとその特徴 ex

ピクセルグラフィクスとは全く違う絵の表し方として、図形などの位置や輪郭を数値的/数式的に覚えておき、絵が必要になる瞬間にその式に応じて絵を生成して表示するという方式があります。このようなモデルを (位置、方向などの「ベクトル」を用いて絵を表すことから) **ベクターグラフィクス**と呼びます (図 10.4)。ベクターグラフィクスでは、絵は円、直線、矩形などの比較的単純な図形の集まりで表すのが普通ですが、高度なソフトになると 3 次曲線、ベジェ曲線などの数式に基づく曲線を活用してもっと柔軟な形を取り扱うこともできます。

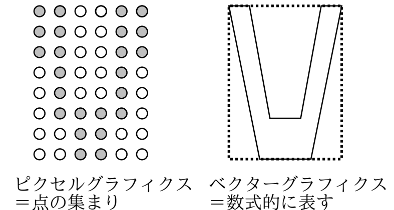


図 10.4: ベクターグラフィクスとピクセルグラフィクス

ベクターグラフィクスで絵を描くのは、「無限に伸び縮み可能な針金で作った図形にスクリーントーンを貼って好きな順に重ねて行く」ようなものだと思えばよいでしょう。針金ですから、あとで自由に置き場所や大きさを調整することができるわけです。ベクターグラフィクスに基づく作画ソフトで代表的なものは Illustrator ですが、より簡便なものはドローソフト、ドローツールと呼ばれます。たとえば PowerPoint 等の中の作画機能はドローツールになっています。これは、いちど描いた図を後で動かしたりしたいのでドロー系が使われているわけです。

ベクターグラフィクスの得失はだいたいピクセルグラフィクスの裏返しと考えればよいでしょう。

○ 図形の拡大・縮小・回転・重なり順の変更などは単にその変更に基づいて絵を表示し直すだけなのでいくらでも自由に行なえる。

○ 絵は数式的に表されているので、拡大してもぎざぎざになることはない。

○ 絵の情報は座標や形などの情報なので、ファイルの大きさは小さくて済むし、拡大/縮小してもファイルサイズは変わらない。

△ 絵の細かさはソフトに用意されている階調機能や模様機能などで決まってしまい、細かい色合いは使いにくい。

△ ぼかし、にじみなどの効果は使えない。[4]

ベクターグラフィクスのファイル形式としては、**PostScript** があります (TeX の出力に使いましたね)。PostScript はもともとはプリンタで出力するページを記述するための言語であり、手で書くこともできます。現在広く使われている **PDF**(Portable Document Format) は PostScript の技術を土台に、よりコンパクトになるように設計されていて、印刷形式の文書を配布するのに有用です。このほか、WWW などのページ内容としてベクターグラフィクスを記述できるように設計された言語として **SVG**(Scalable Vector Graphics) があります。

10.2.2 PostScript — ベクターグラフィクス記述言語

前述のように、PostScript はベクターグラフィクスのファイル形式ですが、正確には「言語」でもあります。今回は、PostScript ファイルを次のような形で作ります。長さの単位はすべて pt(ポイント、1pt = $\frac{1}{72}$inch) です。

```
%!PS-Adobe-2.0                ← PostScript であることを表す
%%BoundingBox: 0 0 400 300 ←絵の範囲を表す。
                              (今回は 400x300 にした。)
…(ここにさまざまな図形記述を入れる)…
showpage      ←プリンタに送った場合にページを出力する
```

線を引くには次のコマンドを使います。

- newpath — 新しい線引きを開始する。

[4]図形を塗りつぶすときに、階調 (グラデーション) や模様 (テクスチャ) などを使うことはできます。

- X Y `moveto` — ペンを指定した座標 (X, Y) に移動。
- X Y `lineto` — 座標 (X, Y) までの線を登録しながら移動。
- `stroke` — `lineto` で指定した線引きを一気に実行する。
- W `setlinewidth` — 線の太さを Wpt にする。

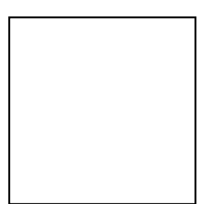


図 10.5: 正方形を描く

では正方形を描くという簡単な例を挙げておきます (図 10.5)。これをたとえば sol で `t3.eps` というファイルに Emacs で打ち込み、「`gv t3.eps &`」とすると見ることができます (`gv` は PostScript を画面で見るためのツールです)。また、PostScript プリンタに送るとそのままプリントされます。

```
%!PS-Adobe-2.0
%%BoundingBox: 0 0 400 400
newpath 100 100 moveto 100 200 lineto
200 200 lineto 200 100 lineto 100 100 lineto stroke
showpage
```

ところで、PostScript は言語だと書いたのはどういうことでしょう? それは、たとえば「ループ」を使って先の正方形を繰り返し描いたりできます。同じ場所に描いてもつまらないので、次のコマンドも知っておいてください。

- T_x T_y `translate` — 絵を描くときの原点を X 軸方向に T_x、Y 軸方向に T_y だけずらす。
- S_x S_y `scale` — 絵を描くときの大きさを X 方向に S_x 倍、Y 方向に S_y 倍する。
- D `rotate` — 絵を描くときの角度を原点のまわりに (反時計回りに)D 度回転する。

繰り返し自体は次のようにします。

- N { 動作列 } `repeat` — 「動作列」を N 回繰り返す。

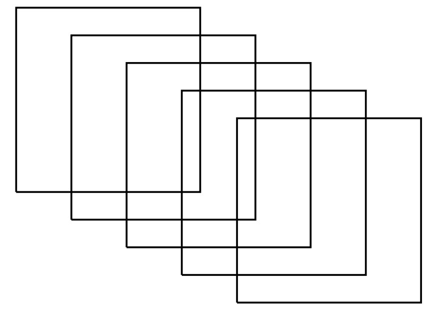


図 10.6: ループで正方形を描く

では、これを使って正方形を 5 回ずらして描いてみます (図 10.6)。

```
%!PS-Adobe-2.0
%%BoundingBox: 0 0 400 400
5 {
  newpath 100 100 moveto 100 200 lineto
  200 200 lineto 200 100 lineto 100 100 lineto stroke
  30 -15 translate
} repeat
showpage
```

最後に文字について説明しましょう。文字の表示には、次の 2 つのステップが必要です。

- `/フォント名 findfont` S `scalefont setfont` — 指定した名前のフォントを探してきて、サイズ (ポイント数)S に設定し、使用するフォントとしてセットする。
- X Y `moveto (文字列) show` — 指定位置 (X, Y) に文字列を表示する。

一度フォントをセットしたら、別のものに変更しないで繰り返し `show` を使って構いません。文字列を丸かっこで囲んで指定するというのはちょっと変わっていますね。もし文字列の中に丸かっこを入れたければ、「\(」「\)」のようにバックスラッシュを前につけてください。

ゴシック体です。
これは明朝体です。
This is Courier-Bold 12pt.
Times-Roman 36pt.
This is Helvetica 24pt.

図 10.7: さまざまなフォント

例 (図 10.7) では「灰色の濃さ」や「色」を変更しているので、その説明もしておきます。

- G `setgray` — 灰色の明るさ (0.0〜1.0、大きいほど明るい) を設定。
- R G B `setrgbcolor` — 色を RGB で指定 (どれも 0.0〜1.0、大きいほどその色が強い)。

```
%!PS-Adobe-2.0
%%BoundingBox: 0 0 400 300
/Helvetica findfont 24 scalefont setfont
20 20 moveto (This is Helvetica 24pt.) show
/Times-Roman findfont 36 scalefont setfont
20 60 moveto (Times\(-Roman\) 36pt.) show
/Courier-Bold findfont 12 scalefont setfont
20 110 moveto (This is Courier-Bold 12pt.) show
0.5 setgray
/Ryumin-Light-EUC-H findfont 40 scalefont setfont
20 140 moveto (これは明朝体です。) show
```

```
5 rotate 0.8 0.3 0.2 setrgbcolor
/GothicBBB-Medium-EUC-H findfont 48 scalefont setfont
20 180 moveto (ゴシック体です。) show
showpage
```

なお、日本語については必ず EUC コードを使用してください。[5]Emacs であれば「Ctrl-X [RET] f euc-jp [RET]」でファイルの文字コードを EUC に設定できます。

ところで、PostScript のフォントは**アウトラインフォント**、つまり数式で輪郭を表現したフォントを使っていることが特徴で、このため任意のポイント数に大きさを設定できます。今日では他のソフトでもこれが普通ですが、PostScript がこれを始める前は、フォントはすべて「ある大きさで」デザインされていたので、自由にさまざまなサイズを指定することはできませんでした。実際にさまざまな大きさにフォントを変更してみてください。

演習 2 PostScript の例題を打ち込んで動かしてみなさい。様子が分かったら、以下の小課題から 1 つ以上やってみなさい。

a. 単純な図形を繰り返し表示させて模様を作りなさい (拡大縮小や回転も使うとよい)。
b. 複数サイズの文字や線を組み合わせて自分の名刺を作ってみなさい。
c. (自由課題) 自分が描きたいと思う好きなものを描いてみなさい。絵の内容についてはあなたに任されますが、例題よりは「美しい」ことが期待されます。

10.3 LaTeX 文書における図や表の扱い

10.3.1 LaTeX 文書に画像を入れる ex

画像が作れるようになったので、これを LaTeX 文書に入れる方法を説明しておきます。

1. TeX に取り込むには、ファイル形式は **EPS** 形式 (Encapsulated PostScript — `%%BoundingBox:` の指定された PostScript) であることが必要です。[6] 先に作った PostScript の例題は既にそのようにしてありましたね。それ以外の形式のファイルであれば、次のようにして変換できます (または Gimp の機能として画像を EPS で保存もできます)。

   ```
   convert test.ppm fig1.eps
   ```

2. LaTeX では`\documentclass`と`\begin{document}`の間に次のような行を追加します。

   ```
   \usepackage{graphicx}
   ```

   そして本文中の図を入れたい場所に次のようなコマンドを入れます。

   ```
   \begin{center}
   \includegraphics[scale=0.5]{fig1.eps}
   \end{center}
   ```

   見て分かる通り、「0.5」は縮小比率、「`fig1.eps`」は PS ファイル名です。scale の代わりに `width=8cm` のように出来あがりの幅を指定もできます (中央そろえは必須ではないです)。

あとはこれまで通りに `platex` で整形します。`platex` コマンド使用時だけでなく、`pxdvi` や `dvipdfmx` を使うときも、画像ファイルが同じ場所に置いてある必要があります。

[5]日本語フォントとして EUC フォントを指定しているためです。
[6]最近は 1 ページの PDF が取り込める LaTeX 整形系も増えています。

10.3.2 図や表の扱い ex

ここまでで LaTeX の文書に図や表が入れられることは分かりましたが、この資料などを見ていただくと分かるように、実際の文書では図や表の扱いが少し違いますね? 具体的には次のようになっています。

(1) 図や表の位置は「ページの上端」とか図表が多いときは「別のページ」など、本文とは別に配置されていることが多い。

(2) 図や表にはタイトル (**キャプション**) と図表番号がついていて、本文では「図 1」などのようにその番号で参照される。

ここではこれらの扱いについて説明しましょう。[7]

上記 (1) や (2) のためには、図や表のここまでに述べた「本体部分」を figure 環境、table 環境で囲み、caption コマンドでキャプションをつけます。全体がどのような感じになるかを見て頂きましょう。

```
\begin{figure}[htbp]
\begin{center}
\includegraphics[scale=0.7]{FIGS/c7-ps01.eps}
\end{center}
\caption{正方形を描く}\label{fig-rectangle}
\end{figure}

\begin{table}[htbp]
\caption{AND 演算の結果}\label{tbl-bitwiseand}
\begin{center}
\begin{tabular}{c|ll}
AND 演算 & 0 & 1 \\
\hline
0       & 0 & 0 \\
1       & 0 & 1 \\
\end{tabular}
\end{center}
\end{table}
```

`\begin{table}`や`\begin{figure}`の後の `[htbp]` とは何でしょうか? それは、LaTeX に「この図や表は次の方針で配置してください」という希望を指定するものです。

- `h`(here) — なるべくこの table 環境や figure 環境を置いた場所の近くに配置してほしい
- `t`(top)/`b`(bottom) — なるべく現在のページの一番上/一番下に配置してほしい
- `p`(page) — 本文とは別に図表のページを作ってそこに集めてほしい

これを「希望する順」に指定しているので、上の例だと「できるだけここに、だめならページの上、だめならページの下、それでもだめなら別ページ」となります。実際にやってみると希望を述べても思い通りにならないことが多いのですが…

次にキャプションは表では表の「上に」、図では図の「下に」配置するのが通例ですので、そのようにしてください。さて、キャプションの後ろに`\label{fig-rectangle}`などとあるのは何でしょうか。それは次の節でまとめて紹介します。

---

[7] とくに (1) のように、位置が本文とは別に配置されるもののことを、LaTeX では**フロート** (float) と呼びます。

10.3.3　ラベルと参照 ex

図や表は「図 1」のように番号で参照すると述べましたが、それを手で管理するのは大変です。文書を改訂していると図表が増えたり位置を入れ換えたりとかはよくあるので。

そこで、上で示したように図や表の箇所で「好きな名前」を指定した label コマンドを置きます。そのうえで、参照する箇所には

```
…図\ref{fig-rectangle}や表\ref{tbl-btiand}のように…
```

のように、ref コマンドを使って参照を指定します。すると、この ref コマンド全体が「1」「2」のように図や表の番号に置き換わって整形されます。それは LaTeX にとっては簡単なことで、整形しているときに順番に出て来る図や表に連番を割り当てていけばよいだけですから。

ただ 1 つ注意する点があります。それは「下の方にある図表の番号を手前の方で参照する」ときは、最初に整形した時には (まだその図表まで到達していないので)LaTeX には番号が分からないことです。そのときは ref コマンドのところには「?」が現れますので、「もう 1 回」整形を実行してください。図表番号の情報は 1 回実行するごとに.aux ファイルに書き出され、次の処理でこれを読むため、2 回やれば先の方の図表番号も正しく分かります。ただし、図表の入れ換えなどがあると番号が変わりますが、そのときは「変わってからあとにもう 1 回」整形する必要があります。

そのほか「undefined reference」というメッセージが出ることもありますが、それは「ref コマンドで指定した名前のものが見付からない」で、だいたいはタイプミスによります。対応する label コマンドと同じ名前になっているか確認してください。

なお、label コマンドを置くのは図表だけに限定されません。節や章の番号も参照したいことがありますが、そのために次のように section などの箇所に label を指定しておくとよいわけです。

```
\setion{ラベルと参照}\label{sec-labelref}
```

こうしておけば「第\ref{sec-labelref}節を見てください」のようにして ref で参照できるわけです。

10.3.4　文献の参照

文献もやはり、番号などで参照し、その管理がややこしいものの 1 つです。文献については LaTeX まわりに様々なツールがあるのですが、ここでは一番簡単なものを説明しておきます。

```
\begin{thebibliography}{99}
\bibitem{dolbook} 兼宗，久野，ドリトルで学ぶプログラミング、
イーテキスト研究所，2008.
\bibitem{wing} Wing, Jannet M.:~Computational Thinking,
CACM, vol.~49, no.~3, pp.~33-35, 2006.
\end{thebibliography}
```

このように (通常文書の末尾に)thebibliography 環境を置き、その中で bibitem コマンドでラベルを指定した項目を区分すると、文献に連番が振られます。参照するときは文献は「\cite{dolbook}」のように cite コマンドを使用します。するとその箇所に対応するラベルを持つ bibitem の番号が埋められます。

演習 3 図や表を 1 つ以上含んだ LaTeX 文書を作成しなさい。本文中から図表番号を参照すること。

課題 10A

今回の課題は「演習 3」ですが、その内容として「演習 1」「演習 2」に含まれる小問 (合計で 6 個) から 1 つ以上を選択し、結果をレポートとして報告するものとしてください。これらは図を作る課題なので、それらを LaTeX 文書の図として入れてください。LaTeX による整形結果を PDF ファイルにして、LMS の「assignment # 10」の箇所からアップロードしてください。以下の内容がこの順に含まれるようにしてください。

- 題名「コンピュータリテラシレポート＃ 10」、学籍番号と氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も脚注などで別途書く。)
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 以下のアンケートに対する回答。

Q1. ピクセルグラフィクスとベクターグラフィクスについてどれくらい知っていましたか。新たに知って面白かったことは何ですか。

Q2. LaTeX での図表の扱いや参照機能についてどう思いましたか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

なお、課題はグループでやって構いません。その場合も、(メンバー氏名を明記した上で) レポートは必ず各自で執筆してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。

＃11　アカデミックリテラシ(総合実習)

今回の内容は「総合実習」であり、資料調査とグループ討議に基づいてレポートを作成して頂きます。今回の目標は次の通りです。

- 情報の受信・発信に際して気を付けるべき点について知る — 効率よく情報を探すことで研究の品質を高め時間を節約できます。
- アカデミックな文書作成における約束ごとを理解する — 約束ごとに従っていないと、たとえよい研究をしてもちゃんと評価してもらえませんし、不法行為にはペナルティがあります。
- LaTeX を用いて適切な内容・スタイルの文書を作成できる — 理系スタイルのレポートや論文を書けることは必須のスキルです。

11.1　情報の検索

11.1.1　ネット検索と検索エンジン

今日のネットワーク上には極めて多くの情報が掲載されています。その多くは Web ページという形態を取っていますが、今日の Web ページの総数は「兆」をはるかに超えていて、その中からゆきあたりばったりで必要な情報を見つけることは困難です。

Google に代表される**検索エンジン** (search engine) を利用することで、**キーワード検索**によって必要な情報のあるページを見つけ出すことができます。検索エンジンはロボットないし**クローラ** (crawler) と呼ばれるプログラムを実行し、各ページに含まれているリンクを次々にたどって多数のページから情報を収集します。具体的には、それぞれのページに含まれているテキストやリンクの情報をもとに、「どの語はどのページとどのページに含まれているか」「どのページが多く参照されているか」などの情報を抽出し、これらを整理してデータベースに保管しています。

検索エンジンは利用者が検索を行った時にこのデータベースから情報を取り出し、利用者のニーズに合っていると思えるページを上位から並べて表示してくれます。

検索エンジンには多数のページの情報が収録されているため、1 つの単語だけを検索すると大量のページが見付かってしまいます。条件をうまく指定して必要な情報の範囲をできるだけ正確に示すことで、不要なページが除外され、有効な情報に到達しやすくなります。

具体的には Google の場合でいうと、次のような検索の指定ができます。

- 複数の単語を空白文字 (半角) で区切って並べることで、「これらすべての語を含む」という意味になり (AND 検索)、見付かるページを絞ることができる。
- 上記において単語の前に「`-`」(半角マイナス) をつけることで「この語を含まない」という意味にできる。
- 場合によっては「A または B」を指定したいときがある。この場合は「単語 `OR` 単語」で指定できる (OR は半角で大文字、空白も半角)。
- 複雑な条件はかっこをつけることができる。

たとえば、「電気通信大学または電通大という名称で、東京にない」という検索をしたければ次のように指定します。

```
( 電気通信大学 OR 電通大 ) -東京
```

検索エンジンで見付けられるのはあくまでもその内容を含む Web ページであり、そのページ内容が正確かどうかについては自分で判断する必要があります。現代では Web ページは誰にでも作れるので、個人が「まったくの嘘」をページとして公開していても、それを止めるものは無いのです (その人に悪意があるとは限らず、単にファンタジーを公開したいだけかも知れません)。

ページに書かれている情報が正確かどうかを判断する方法としては、たとえば次のものがあります。

- 公的機関などが公開しているデータであるか (またはそれと合致するか)
- 一次情報 (また聞きでない情報) か、他の情報源と合致するか
- 論拠が明確か、根拠となる文献やデータをきちんと示しているか

11.1.2 文献や書籍の検索

アカデミックな (学術的な) 主題について調べたり論文・レポートを書く場合には、参照する情報も学術雑誌や学術書などに掲載されているものであることが望まれます (テーマによってはそれだけで済まないこともありますが)。

論文や学術書の検索を行えるネット上のサービスも多数あります。

```
http://scholar.google.com  --- Google Scholar
http://ci.nii.ac.jp        --- CiNii
http://www.amazon.com (co.jp)  --- Amazon
```

Google Scholar は Google が提供している学術検索サービスで、検索の方法が Google と一緒だという利点、検索だけでなく本文のありかが分かって読めるものが多いという利点があります。CiNii は国立情報学研究所の提供するサービスで、日本の学術文献には強いです。Amazon はもちろんあらゆる本が探せますが、洋書であれば米国サイトが強いかもしれません。

図書館は昔から多くの情報の集積場所になってきました。大学の図書館はその大学の研究分やに関する論文雑誌や蔵書を大量に収集しています。大学図書館の利点は、見付かった文献を所蔵していれば出向くだけですぐに見られることです。電気通信大学図書館のサイトで「資料を探す」→「学内資料」を選び、学内資料を検索してみてください (簡易検索と詳細検索の両方を体験してください)。

```
http://www.lib.uec.ac.jp  --- 電気通信大学附属図書館
```

情報技術の発達とともに、**電子図書館** (digital library) も増えています。とくに多くの学会は、その学会の分野に関連する電子資料を収集した電子図書館を運営していて、学会の発行する雑誌の内容を有償・無償で公開しています。また、学術雑誌出版社も自社の雑誌コンテンツを販売しています。

```
https://ipsj.ixsq.nii.ac.jp/ej/ ---情報処理学会図書館
http://dl.acm.org   --- ACM Digital Library
https://www.computer.org/web/csdl/ --- IEEE Computer Society DL
http://www.sciencedirect.com ---Elsvier 社検索サイト
```

有償のものであっても、大学が一括契約している場合には学内から無償でコンテンツを取得できる場合があります。具体的な相談は附属図書館カウンタでどうぞ。

11.2 著作権と引用

11.2.1 著作権について

私たちが学術的な活動をする上で、著作物と**著作権** (copyright) について意識することは頻繁に必要になります。それは、学術活動は基本的に研究して分かったことを論文等の形でまとめる作業が不可欠だということと、自分や他人が書いた文章はすべて著作物であり、著作者の権利を侵して利用することはできないことによります。

ここで著作権法の条文のなかから、とくに重要と考える点を紹介します (http://law.e-gov.go.jp で、すべての法律の最新の条文が見られます)。

まず目的について見てみます。

> (目的) この法律は、著作物並びに実演、レコード、放送及び有線放送に関し著作者の権利及びこれに隣接する権利を定め、これらの文化的所産の公正な利用に留意しつつ、著作者等の権利の保護を図り、もつて文化の発展に寄与することを目的とする。

著作権法は著作物を作り出した人の権利を保護することで「文化の発展に寄与」することを目的としています。たとえば苦労して創作してもそれをすぐコピーされたら創作で食べていけなくなり、創作する人がいなくなりますね。そのようなことを防ぐのが目的です。

次に著作物の定義について見てみましょう。

> 著作物 思想又は感情を創作的に表現したものであつて、文芸、学術、美術又は音楽の範囲に属するものをいう。

重要なのは著作権法では「表現」したものを著作物として保護するのであり、形のないアイデアなどは保護されないということです (アイデアは特許や実用新案などの形で保護を受けられます)。

この先はいちいち条文を示さず、要点だけ記します。

- 無方式主義 — 登録手続きは不要で、著作物ができた時点で自動的に権利が生じる
- 著作物の種別 — 言語著作物 (文書、演述)、音楽、舞踏、美術、建築、図形、映画、写真、プログラム、2 次著作物 (翻訳、編曲、脚色、翻案)、編集著作物がある
- 著作権の内容 — 複製権、上演権、演奏権、公衆送信権 (放送、有線放送、自動公衆送信)、口述権、展示権、頒布権、譲渡権、貸与権、改作利用権 (翻訳、編曲、翻案)、2 次著作物利用に関する許諾権 — これらは著作財産権であり譲渡できる
- 著作者人格権 — 公表権、氏名表示権、同一性保持権から成り、著作者の人格を保護するものであって譲渡できない

11.2.2 著作権の制限と引用

著作物の利用には原則として著作者の**許諾**が必要ですが、それだけでは不便なのでいくつかの場合には著作者の権利を制限して自由に利用できるようにしています。

- 保護期間 — 著作者の死後 (団体名義では公表後)50 年まで (映画の場合は公表後 70 年)
- 図書館における複製 — 図書館 (公共図書館、大学図書館) の利用者の求めに応じ、調査研究のために、所蔵図書などの一部分の複製を、1 人につき 1 部供する場合は許諾不要
- 私的使用 — 個人的に家庭内やこれに準ずる限られた範囲内で使用するため自分で複製する場合は許諾不要 (テレビ番組などを録画できるのもこの規定のおかげ)
- 引用の条件を満たす場合は許諾不要

とくに**引用** (quotation) は私たちが論文等を執筆するときに重要になるので詳しく説明します。自分の著作物 (基本的に文章) に他人の著作物を引用する場合条件とは次の 3 つです。

- 公表された著作物であること
- 公正な慣行に合致
    - どの範囲が引用物か明確で出典が明示されていること
    - 引用する著作物を改変していないこと (同一性保持)
- 引用の目的上正当な範囲内であること

- 自分の著作物が主、引用される部分が従の関係にあること
- 自分の著作物にとって必要最小限の範囲であること

引用とはこれらの要件を満たして許諾なしに著作物を利用することなので、「無断引用」という言葉は意味をなさないことになります (引用であれば無断でよい)。逆に、引用にはこのようにかなり厳しい条件が課せられますが、著作者の許諾を得て使用するのであれば、これらの条件を満たす必要はありません。

また、著作権法はあくまでも表現を保護するものなので、他人の考えなどを紹介するのに、「誰それは、このようなことを述べた (主張した)」などのように自分のことばで紹介する (**パラフレーズ**) 場合は引用の条件などは関係ありません。

ただし、他人が考えたり述べたことを、あたかも自分のオリジナルであるかのように書くことは**剽窃**と呼ばれ、犯罪になります。引用の条件を守っているうちは剽窃にはなり得ませんが、パラフレーズの場合は剽窃にならないように十分な注意が必要です。また逆に、パラフレーズに際して元の文章に無いような意味や内容をつけ加えることは意味や内容を改変することになり、これも不適切な行為になります。[1]

11.3 アカデミックな文書の作成

11.3.1 アカデミックな文書とその要件

アカデミック (学術的) な文書とは、人により定義や範囲が違うかも知れませんが、ここでは論文、レポートなど「学術的な活動 (やその練習) のために作成する文書」であるものとします。

アカデミックな文書と一口にいっても非常に幅広いものがありますが、それでも小説やポエムのような文芸、あるいはビジネスレターなどとは明確に違っていると言えます。違うと思われる点を次に示します。

(1) 何らかの「新しいこと」を記述するために作成する。

(2) 「**事実** (fact)」と「**意見** (opinion)」を明確に区分する。

(3) 論理的な論述により結論を導き出す。

(1) ですが、たとえば論文であれば、(たとえごく小さなことでも)「これまで知られていなかった事柄を示す」ことがその目的であり内容となります。[2] では皆様が 1 年次の授業で実験をやってその結果をレポートにするのはどうでしょうか。授業における実験はだいたい、よく知られていることを確認するためにやるわけですが、この場合はその (皆がやる定番な) 実験を「あなたが」やった結果どのようだったか (あなたがちゃんとやったか、どう工夫したか、何を考え何を学んだか) という「あなた以外だれも知らないこと」を (教員がチェックしたりあなた自身の学習に役立つために) 記述するわけです。

上述の目的のためには、(2) が重要となります。計測をした結果計測値がどうだったか、実験をした結果どのような事象が観測されたか、などの「事実」をまず報告し、その事実から導かれる「新しいこと」が文書の内容となるわけですから。一方、それらの活動をやった結果「どのように思った」「このような理由だと感じた」などのことも、文書に記述することは普通です。ただしこれらはあなたの頭の中で考えた「意見」であり、「他の人であれば別の意見を持つかもしれない」という点で「新しい事実」ではないわけです。[3]

---

[1] 元の著作物を受け手が「これはあの作品だ」と分かるような同一性を保持しつつ、登場人物や場面を変えたり表現を変更したりして別の著作物を作る場合は「翻案」となり、著作者の許諾が必要です (翻案権の内容)。

[2] 小説でも音楽でも「新たな感情をもたらす」という点では新しいことを目的とすると言えますが、ここで言っているのは「学術的な事柄」だと考えてください。

[3] 実験レポートであなたが感じたこと、というのは感じたという点では事実かも知れません。しかし、そのようなレポートは学習や練習のために書くのであり、卒業論文などステージが進んだ段階では「感じただけ」のものを事実として記載することはなくなります。

そして、先の 2 項のための「道具」となるのが (3) の「論理」です。誰の目にも明らかなことは既に先人が報告してしまっている可能性が高いので、現在書かれる論文のほとんどは、すぐには明らかでない (直接的には観測・計測できない)「事柄」を、さまざまな計測、実験、分析に基づいて間接的に示していくことになります。その際には、論理的な論証が必要になり、それを正しく使うことが重要となります。

11.3.2 先行研究や文献の重要性

論文では新たに見出した事柄を述べるので、「これまでに知られていた事柄」と「今回新たに見出した部分」の違いをきちんと示すことが重要です。これまでに知られていた事柄は、別の研究者が論文等で発表しているはずですから、それらを紹介して「ここまでは知られていた」という説明をする必要があります。これが論文の「**先行研究**」セクションです。そしてその紹介する論文等については文献として記載し、先行研究のセクションではその参照を記載する必要があります。

きちんと先行研究が書けていないと、論文を読む人にとって「この論文は何が新しいのか?」が読み取れず、せっかくよい内容であっても伝わらなかったり、論文雑誌投稿の場合は却下されたりします。十分注意しましょう。必要なことは、自分の研究に関連すると思われる文献をもれなく探して読み、その上で先行研究セクションとして何をどこまで紹介すればよいかを考えて書くことです。

「もれなく探して読む」ことは実は研究をはじめる前にやっておく必要があります。ぜっかく新しい研究をしたと思ったのに、実は同じことが既に発表済みだったら、研究の価値はゼロです。そうならないためにも、ある分野の研究を志したら、常にその分野の論文を探し、重要そうなものには目を通して、自分の**文献リスト**を最新のものにしておくことが大切です。

文献を参照するのは先行研究のセクションに限りません。論文では「未定義なことば」があっては困ります。しかし、自分はよく知っていて、自分の論文ではいちから説明すると長くなりすぎるけれど、その論文の読者は知らないかもしれない用語というものもたまに現れます。そのような場合に、その用語について説明している文献を参照すれば、いちいち定義を説明しなくてもすみます。

もっとも、すべての用語について定義か参照を記載するというのは大変すぎますから、どこからかはその論文の分野の常識として説明せずに使う言葉もあります。それは、その分野や論文の対象読者の水準によります。

課題 **11A**

今回の課題のために、3〜5 人のグループを構成してそこでディスカッションをおこなって頂きます。欠席などで乗り遅れた人は、クラスメートを 2 名探してディスカッションだけやってもらってください。やるべきことを記します。

(1) グループ内で相談して、ディスカッションのテーマを決めます。テーマは本資料で解説されている事柄なら何でもいいですが、人によって立場が異るテーマの方がやりやすいと思います。例をいくつか挙げます (あくまでも例ですが採用したければご自由にどうぞ)。

   - 今のように何でもすぐ検索して調べられる方がいいのか? 昔のように時間を掛けて考えたり図書館で本を探したりした方がよくなかったか?
   - 現在、紙の本には価値があるか? ネットで詳しく新鮮で動画なども含んだ情報を見た方がずっとよくないか?
   - ミュージシャンは自分の著作権をもっと自分で管理した方がよいか? 今のような委託システムはいいことか?
   - 作家とミュージシャンは著作権による保護という点でどちらが幸せ?
   - 情報は無償であるべきではないか? 有償であってもよいか?
   - 著作権法による保護はもっと緩くすべき? 厳しくするべき?
   - 引用という制度はよいことか? もっと緩くする/厳しくするのがよい?
   - 現代の図書館の役割は何か? 紙の本を沢山持つことでいいのか?

- 論文と文学のどちらが好きか? それはなぜか?
- すべての主張に論拠が必要とは窮屈ではないか? むしろ好ましいか?

(2) テーマが決まったら自分の考え (立場) を決め、その裏付けとなるような資料を収集します (最低でも 20 分くらい)。

(3) グループ内でテーマに対する討論を行います。時間は自由ですが、それぞれで自分および他人の発言内容をメモしてレポートに掲載していただくので、長すぎると大変になります。録音はしてもかまいません。

(4) レポートを作成してください (下記参照)。調べた資料や他の人が言及していた資料は参考文献セクションに掲載すること (後で互いに収集した資料の情報を交換するといいでしょう)。

レポートは LaTeX により整形し、PDF ファイルにして、LMS の「assignment # 11」の箇所からアップロードしてください。以下の内容がこの順に含まれるようにしてください。

- 題名「コンピュータリテラシレポート# 11」、学籍番号と氏名、提出日付を書く。
- グループで決めたテーマ、およびグループ全員の名前と学籍番号を書く。
- 討論の前にどのようなことを考えたか、どのような資料を探して調べたか、その他どのような準備をしたかを書く。
- 討論の内容を書く。全文書き起こす必要はない。誰がどのような発言をしたか、どのような流れの討論になったかが分かる程度に要約を書く。この部分も各自それぞれに (自分のメモに基づいて) 書くこと。
- 討論が終って自分がどのような結論に至ったかをその論拠・理由等も含めて筋道立てて書く。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 参考文献セクション。
- 以下のアンケートに対する回答。

  Q1. 調べる、討論する、考えるというプロセスはあなたにとってどのように有効でしたか。一人で考えるのと比較して述べてください。
  Q2. 今回のようなレポートは何がよかったですか。何が大変でしたか。
  Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

今回はグループ作業が前提ですが、レポートは各自で作成してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。

＃12　HTML/CSSによるWebページ記述

今回の目標は次の通りです。

- HTML による Web ページのマークアップについて学ぶ — Web サイトは研究その他の活動において情報を共有/公開する有力な手段でありサイトがすぐ作れるようになっておくと有益です。
- CSS による表現指定について学ぶ — 論理構造と表現の分離という考えは情報技術において重要であり、またページの表現を調整できるとサイトの視認性のために有効です。

12.1　HTML と CSS

12.1.1　Web とマークアップ ex

LaTeX でマークアップの仕組みは理解したけれど、見たまま (WYSIWYG) 方式が分かりやすい、と思う人もいるかも知れません。しかし実は、見たまま方式が「使えない」場合も存在します。しかも、皆様がいつも目にしている Web ページがまさにそうなです。なぜでしょう?

ワープロであれば、出力する紙サイズを決め、紙に合わせて配置や文字詰めやフォントを決めて「見たまま」で作業できます。しかし Web ページはどうでしょう? Web には「紙サイズ」は存在せず、「1 行何文字」も決められません。マシンによって使えるフォント違ってきますから、「ここは MS 明朝」と指定しても、そのフォントがないかも知れません。

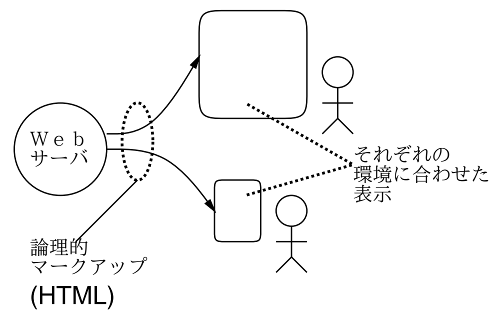


図 12.1: Web と HTML による論理的マークアップ

ではどうすればいいのでしょうか? できることは「ここは表題」「ここは段落」「ここは箇条書き」など、文書の「構造を」指定しておいて、「ブラウザが画面に表示する時に」窓の幅や使えるフォントに合わせて整形してもらうことです。つまり意味づけ方式の (論理的な) マークアップを行う必要があるわけです (図 12.1)。そのための書き方が **HTML**(HyperText Markup Language) です。

しかし LaTeX に続いてまた別のマークアップを学ぶのか、と思う人もいるかも知れません。今日では、WYSIWYG ではないけれど GUI でページを作成できるソフトや **CMS**(contents management system) も多く使われていますから、HTML なしでもサイトは作れます。しかし、Web アプリケーションなどでプログラムから HTML を生成する場合には HTML を知っている必要があります。また、ソフトも何もなくても、少し練習すれば Web サイトが簡単に作れるようになり、便利です。

12.1.2　HTML の基本部分 ex

これから HTML とはどのようなものか学んで行きますが、すぐ実践してみるために「練習ページ」を使用します (図 12.2)。このページの下方にコピー用のテンプレートがありますので、それをコピー

して先頭の入力欄に貼ってから修正してください。表示してみたい時はいつでも Run ボタンを押すことで、そのページを整形したようすがすぐ下の領域に表示されます。

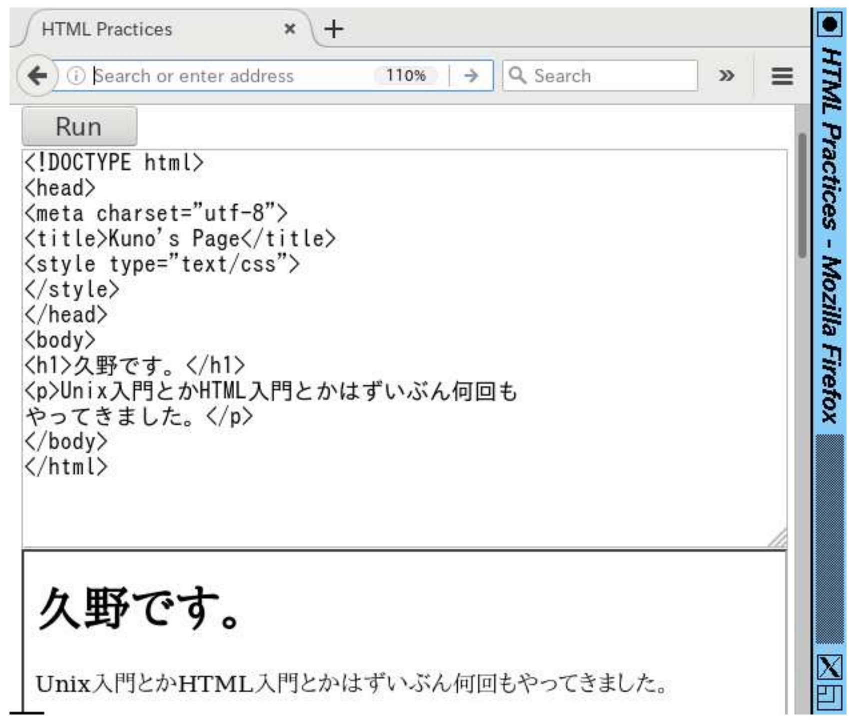


図 12.2: HTML 練習ページを使っているところ

では、最初に作成してもらう内容を見ていただき、説明します。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>○○'s page</title>
</head>
<body>
<h1>○○です。</h1>                  ←ここを好きに作る
<p>…挨拶ないし自己紹介を書く…</p> ←ここを好きに作る
</body>
</html>
```

冒頭の「<!DOCTYPE html>」は以下のファイルが HTML であることを宣言します。以後、HTML では<名前>…</名前>という形でさまざまな種類のものを指定します。これを**要素**、<名前>を**開始タグ**、</名前>を**終了タグ**と言います。開始タグから終了タグまでが 1 つの要素。ただし終了タグのない単独の要素もります。また、開始タグの中に「<名前 属性名="値" 属性名=" 値" …>」のように 1 つ以上の追加指定 (**属性**) を書く場合もあります。

なお、このように HTML では「<」「>」およびすぐに説明する機能に使う「&」は特別な意味を持つので、この 3 つの文字を使いたい場合は代わりに「<」「>」「&」という書き方を使う必要があります (この書き方を**文字エントリ**または**文字参照**と言います)。

要素の中には別の要素が入ることもあります (入れ子構造)。上に出て来る HTML の要素 (タグ) の意味は次の通り。

- <html>…</html> — HTML 文書全体をあらわす。

- <head>…</head> — ヘッダ (この文書に関する情報を記述する部分) をあらわす。
- <meta charset="符号化"> — ファイルの符号化を指定 (閉じタグのない単独のタグ)。
- <title>…</title> — 文書のタイトルをあらわす。
- <body>…</body> — 文書本体 (ブラウザの窓の内側に表示される内容全体) をあらわす。
- <h1>…</h1> — レベル 1 の見出し (大見出し) をあらわす。さらに小さいレベルの見出しとして<h2>…</h2>〜<h6>…</h6>まで使うことができる。
- <p>…</p> — 通常の段落をあらわす。

段落等の中はすべて窓の幅に合わせて詰め合わせされますから、空白や改行を入れても意味がありません。空白や改行で整形したい場合は pre 要素を使います。

- <pre>…</pre> — この内側は空白や改行をそのまま残す。

次のような形のものを末尾 (ただし</body>の前) に入れてみると図 12.3 のようになります。最低限、h*n* 要素 (見出し)、p 要素 (段落)、pre 要素 (そのまま) さえ覚えればいちおうページは作れます。

```
<pre>
  柿食へば
          鐘が鳴るなり
                  法隆寺
</pre>
```

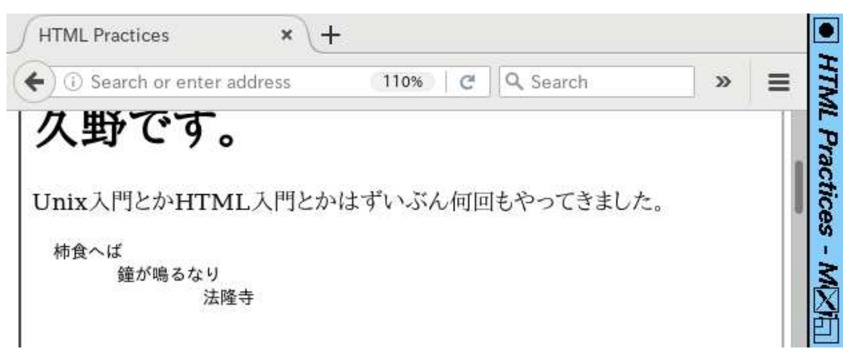


図 12.3: pre 要素は空白と改行が残る

次はリンクのつけ方と、数個追加のタグを説明します。

- リンクテキスト — 「リンクテキスト」の部分がリンクとして表示され、そこを選択すると *URL* のページを表示する。
- <blockquote>…</blockquote> — 引用文を表す。
- <div>…</div> — ディビジョン (一連の範囲) を表す。

このように、a 要素の開始タグは「href=...」という指定 (**属性**と呼びます) がつけられ、それによってリンク先を指定します。これを利用して、「リンク集」を作ってみましょう。1 項目を 1 段落とします (リンクは段落等の中に入れることになっています)。例を示しましょう (図 12.4)。

```
<h2>私のリンク集</h2>
<p><a href="http://www.fujitv.co.jp/">フジテレビ</a>
のサイトはよく見に行きます。ドラマ好き。</p>
<p><a href="http://www.nhk.or.jp/">NHK</a>はあまり
見ません。</p>
```

図 12.4: リンクを入れたページ

演習 1 好きな内容の Web ページを作成しなさい (まず最小限のもので表示してみて、少しずつ書き足すのがよいです)。表示できたら、次のテーマから 1 つ以上選んで作成・検討してみなさい。

a. 見出しのタグは`<h1>`〜`</h1>`から`<h6>`〜`</h6>`まで 6 レベル用意されていますが、それらがどのように見た目を違えているか (大見出しはより大見出しらしいか) 調べてみなさい。できれば、複数のブラウザでやってみて比較できるとなおよいです。

b. 段落のタグ`<p>`〜`</p>`はどのような機能を持っているか (たとえば字下げとか複数並べたときの様子とか)、検討してみなさい。ディビジョンのタグ`<div>`〜`</div>`や引用文のタグ`<blockquote>`〜`</blockquote>`についても同様に調べ、比較しなさい。

c. ページ内にリンクを作りなさい。URL はブラウザの URL 窓からコピーするのが間違えにくくていいでしょう。ページによってはうまくリンクできないものもあると思いますが、どのようなものはうまく行ってどのようなものはうまく行かないか検討してみなさい。

12.1.3 構造と表現の分離、スタイルシート ex

ここまで HTML 要素をいくつか見てきましたが、ページを作ってみると白地に黒で変わりばえしません。しかし世の中の Web ページを見ると、色や配置などの表現がざまなまに工夫されています。自分のページでもこれらの表現を行なうにはどうすればよいでしょうか? 実は過去には HTML にも「色をつける」「フォントを変える」「中央そろえ」など表現を指定する要素がありました。しかし、HTML 4.0 からはこの種の機能はすべて「非推奨」になり、代りに「スタイルシート」と呼ばれる方法で表現を指定するようになりました。

なぜでしょうか? たとえば HTML では「大見出しを全部集めて来て一覧を作る」などの作業は (`grep` などのツールを使って) 簡単に行なえますが、そのとき見出しの中に「ここは青い色」など別のタグが混ざっているとうまく取り出せなかったり、または取り出したものにタグが混ざるなど、面倒なことが起きます。

それに、大見出しを青い色にするとしたら、全部の大見出しをそのように統一したいわけですが、すべての大見出しの所に余分に「ここからここまで青」というタグをつけて行くのも無駄な話です。コンピュータで処理するのだから、「すべての大見出しは青」と「ひとこと」言えば済むようであるべきではないでしょうか? (図 12.5)。スタイルシートとはちょうどそのように、つまり文書の構造のそれぞれについて「このような部分はこのような表現」という形で表現を指定する機能なわけです。

HTML と組み合わせるスタイルシート指定言語としては **CSS** (Cascading Style Sheet) が使われます。HTML に CSS の指定を追加する方法としては、次の 3 通りがあります。

(1) CSS 指定を次のように **style 要素**の内側に書く。style 要素はヘッダ部分に入れる必要がある。

```
<style type="text/css">
CSS 指定…
…
```

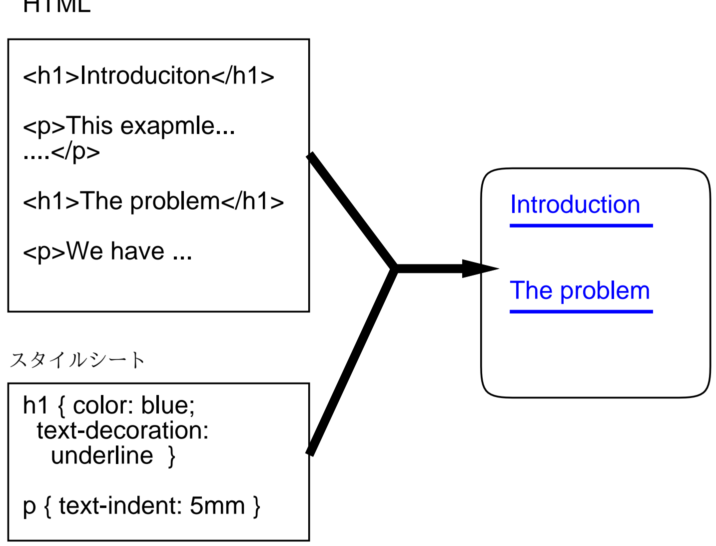


図 12.5: スタイルシートの概念

```
</style>
```

(2) CSS 指定を別ファイルに入れ、HTML のヘッダ部分に次のような **link** 要素を入れる (ここでは CSS 指定が `mystyle.css` というファイルに入っているものとしました)。この方法はやや繁雑だけれど、1 つの CSS ファイルを複数の HTML ページに適用させられる。

```
<link rel="stylesheet" href="mystyle.css" type="text/css">
```

(3) HTML の各要素に **style** 属性を指定し、その値として CSS 指定の本体部分を書く。特定の要素だけに表現を指定する場合に使う。

```
<p style="color: blue">この段落は青い。</p>
```

以下では (1) の方法を使うようにします (練習ページのテンプレートにも style 要素は用意してあります。先程作ったページの style 要素の中に次のものを入れて表示した様子を図 12.6 に示します。本体部分 (body 要素の中) は一切変更していないけれど、色々な表現ができていることが分かります。

```
…前略…
<style type="text/css">
h1 { color: blue; text-decoration: underline }
p { text-indent: 5mm; background: rgb(180,200,255) }
pre { border: double green 4px }
</style>
…後略…
```

12.1.4 CSS の指定方法 ex

順序が逆になりましたが、CSS の指定方法について説明しましょう。まず、CSS の指定は「規則」の集まりで、1 つの規則は次の形をしています。

```
セレクタ { プロパティ: 値; プロパティ: 値; … }
```

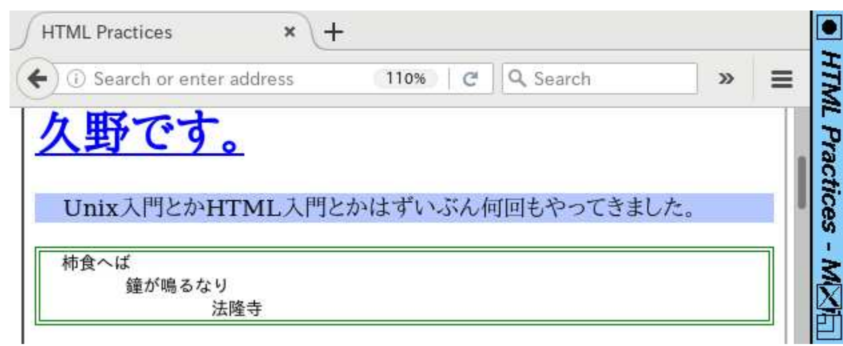


図 12.6: スタイルシートを適用した場合

セレクタは、とりあえず HTML のタグと思ってください。つまり「この要素はこう表現する」という指定です。**プロパティ**は、色や字下げなどです (すぐ後で説明します)。そして、それに対する値を指定します。値の指定方法は次の通り。

- 文字サイズの指定方法: `12pt` (ポイント数)、`xx-small`、`x-small`、`small`、`medium`、`large`、`x-large`、`xx-large`、百分率 (本来のサイズの何%か)。
- 長さの指定方法: `1px`(画面上の点)、`1cm`(センチ)、`1em`(文字「m」の幅 1 個ぶん) などがある。
- 百分率: `%`をつけた数値。ページ幅に対する割合などの指定に使用。
- 色の指定方法: `black`、`blue`、`gray`、`green`、`maroon`、`navy`, `olive`、`purple`、`red`、`silver`、`white`、`yellow`、`rgb(赤,緑,青)` ただし赤/緑/青は 3 原色の強さを 0〜255 の数値で表す。
- ファイルや URL: `url(ファイル名)`、`url(`*URL*`)`。

CSS プロパティの代表的なものとしては次のものがあります。

- `color:` 色 — 文字色を指定。
- `background:` 色 — 背景色を指定。
- `margin:` 長さ — 要素の周囲のマージン (余白) 幅を指定。4 つの長さを指定すると「上、右、下、左」の長さを指定したことになる。2 つだと「上下、左右」、1 つだと 4 周全部がその長さに。また個別に指定したければ `margin-buttom`、`margin-top`、`margin-left`、`margin-right` という指定もできる (padding、border も同様)。
- `padding:` 長さ — 要素の周囲のパディング (詰めもの) 幅を指定。指定方法は margin と同様。
- `border:` 形状 色 長さ — 要素の枠を指定。形状として、`solid`(均一)、`dashed`(点線)、`double`(2 重線)、`ridge`(土手)、`groove`(溝)、`inset`(くぼみ)、`outset`(出っぱり) 等が指定できる。色は枠の色、長さは枠の幅を指定。
- `text-indent:` 長さ — 段落先頭の字下げ幅を指定。
- `text-align:` 種別 — `left`、`right`、`center` で左そろえ、右そろえ、中央そろえを指定。
- `text-decoration:` 文字飾り。`underline`(下線)、`overline`(上線)、`line-through`(抹消線) 等を指定。
- `font-style:` 傾き。`normal`、`italic`、`oblique` 等を指定。
- `font-size:` 文字の大きさを指定。

演習 2 HTML の練習ページに先の例の CSS 指定を (1 行ずつ順番に) 追加して効果を確認してみなさい。その後、以下の課題から 1 つ以上やってみなさい。

a. h1〜h6 の見出しの要素から 1 つ以上選び、自分がかっこいいと思うデザインになるように CSS で表現を工夫してみなさい。たとえば次のような調整が可能だと思われます (どうするかは自分で選択)。
   - 枠線で囲んだりする。下だけ、または下と上だけ枠線にするなども。
   - 周囲のあきを調節したり、色、背景色や文字サイズを変えたりする。

b. p 要素、blockquote 要素、div 要素から 1 つ以上選び、本文の文章をこれらの要素として表示されたときにかっこいいデザインになるように CSS で表現を工夫してみなさい。たとえば次のような調整が可能だと思われます (どうするかは自分で選択)。
   - 段落先頭の字下げ量や周囲のあきを調整する。
   - 枠線で囲んだりする。左だけ、または左と右だけ枠線にするなども。

c. 「`p:hover { color: red }`」のようにセレクタに「`:hover`」を指定するとその要素上にマウスポインタが乗った時だけの表現を指定できるのでやってみる。[1]

d. (自由課題) レポート課題で提出する Web ページについて、CSS を調節して全体として「統一された、美しい見栄え」になるように工夫してみなさい。

12.2 より進んだ HTML と CSS の指定

12.2.1 HTML をファイルとして保存する

ここまでは「HTML 練習ページ」を使ってきましたが、これではブラウザを止めたら内容が無くなってしまいますし、複数のページを作ることもできません。そこで、これまで作った HTML をファイルに保存して「普通の形で」ブラウザで表示してみましょう。

sol では各自のホームディレクトリの下に `public_html` というサブディレクトリが用意されていて、そこに HTML を (誰でも読める保護設定で) 置くと、次の URL で学内限定で公開されます。

```
http://www.edu.cc.uec.ac.jp/~ユーザ名/ファイル名.html
```

また、学外では上記の方法は使えませんが、ブラウザでファイルを直接開くことで表示させることは常にできます。両方やってみることにして、次の手順でファイルを保存してください。

(1) エディタで「`~/public_html/mypage.html`」を開く (最初は空っぽのファイルであるはず)。

(2) 「HTML 練習ページ」の自分が作成した HTML をコピーし、エディタに張り付け、エディタを保存する。

(3a) ブラウザの URL 窓で「`http://www.edu.cc.uec.ac.jp/~ユーザ名/mypage.html`」を打ち込んで開くか、または

(3b) ブラウザの「ファイルを開く」機能を使って今保存したファイルを直接開く。

エディタで作業するようになれば、作りかけの状態で保存しておくことも、前に作ったページをコピーして修正することも、自由に可能です。以下ではこの方法で課題ごとに別の HTML ファイルを作って頂くことを前提とします。[2]

---

[1] さらにプロパティとして「`transition: 3s`」のように変化に要する秒数を指定すると色等がその時間かけてゆっくり変化する。戻るときもゆっくりにしたければ`:hover` をつけない側の指定にも `transition` を指定する。

[2] ファイル名は好きにしてよく、`index.html` にするとファイル名なしで (「`/`」までで) 表示されるページとなります (ただし URL 窓で開いた場合)。

12.2.2 ブロック要素とインライン要素 ex

HTML は文書の意味に従ったマークアップをするので、「どの要素の中にはどの要素が入れられるか」ということがきちんと定まっています。たとえば h1 要素を使うと文字が大きくできますが、では p 要素 (段落) の中に h1 を入れていいかというとそれは駄目です。h1 はあくまでも大見出しであり、段落の途中に大見出しが挿入されるというのは文書としてあり得ないからです。

間違った HTML を書かないために、次のような原則は理解しておきましょう。

- ブロック要素とは、段落などと同じ位置に置けるもので、p、h1〜h6、blockquote、table、div、ul、ol、dl などがある。
- ブロック要素の中にはブロック要素とインライン要素と文字が入れられる (p だけ例外で、中にブロック要素が入れられない)。
- インライン要素は、ブロック要素の中に入っている必要があり、中にインライン要素と文字が入れられる。

このほか、li は ul と ol の中、th と td は tr の中など (いずれも後述)、いくつか要素の機能から決まっている規則がありますが、それは原則というより個別の規則ですね。

インライン要素としてはこれまで a 要素しか出てこなかったので、他の主なものを挙げます。

- `<br>` — ここで行かえをする。単独のタグ。
- `<em>…</em>` — この範囲を強調表示する。
- `<b>…</b>`、`<i>…</i>`、`<code>…</code>`、`<var>…</var>`、`<sup>…</sup>`、`<sub>...</sub>` — それぞれ、ボールド、イタリック、プログラムコード、変数、肩字、添字のスタイルとする。
- `<img src="...">` — 埋め込み画像 (後述)。
- `<span>...</span>` — 任意のインラインの範囲 (後述)。

12.2.3 id と class ex

ここまでは CSS のセレクタとして要素 (タグ) の名前を使用して来ましたが、それだと当然「全部の○○」の表現を変更することになります。それでいい場合も多いですが、「ここだけ」「このいくつかだけ」のような指定も行いたですね。そのため、各要素の開始タグには次の 2 種類の属性が指定可能です。

- `<タグ名 … id="固有名">` — 固有名を指定する。
- `<タグ名 … class="クラス名">` — クラス名を指定する。

固有名 (**id**) は「この特定の箇所」を示すのに使うのもので、従って同じものを 2 つ以上指定してはいけません。**クラス名**は複数を指定することができます。これらを CSS で指定するときはセレクタとして次の書き方を用います (その先の書き方はこれまでと同じ)。

```
#固有名 { .... }
.クラス名 { ... }
```

そして、ここまでに出て来た **span 要素** (インラインの範囲) と **div 要素** (ブロックの範囲) はいずれも、特定の表現はついていなくて、主に上記の方法で表現を設定するのに使われます。たとえば重要なところを赤字の下線つきにしたいとすると、HTML 側では span を用いて次のように記述します。

```
HTML 入門では、<span class="imp">スタイルシート (CSS)</span>、
<span class="imp">意味と表現の分離</span>が重要です。
```

対応する CSS 側の記述は次のようにします。

```
.imp { color: red; text-decoration: underline }
```

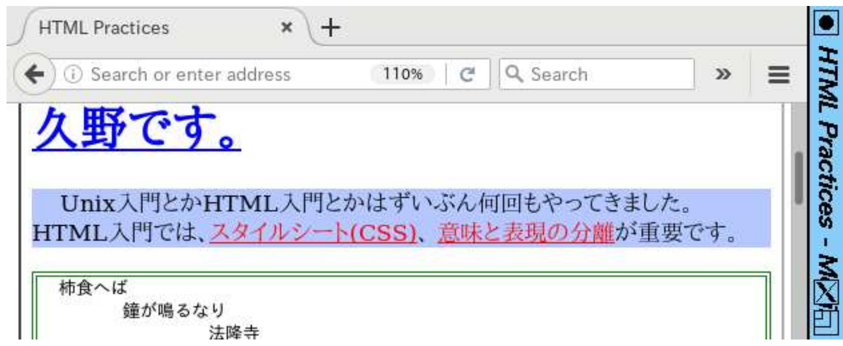


図 12.7: span とクラス指定による範囲の指定

12.2.4 箇条書と表 ex

HTML の追加として、LaTeX にもあった箇条書きと表について説明しておきます。まず番号なしの箇条書きは次のようになります。

```
<ul>
<li>スタイルシートは、表現の豊かなページを作るために必要。</li>
<li>意味と表現の分離は、スタイルシートの適切な使用のために必要。</li>
</ul>
```

`<ul>...</ul>`を`<ol>...</ol>`にすると番号がつきます (図 12.8)。LaTeX でいうと ul 要素は itemize 環境、ol 要素は enumerate 環境に相当します。箇条書きの項目はどちらも li 要素として含めます。

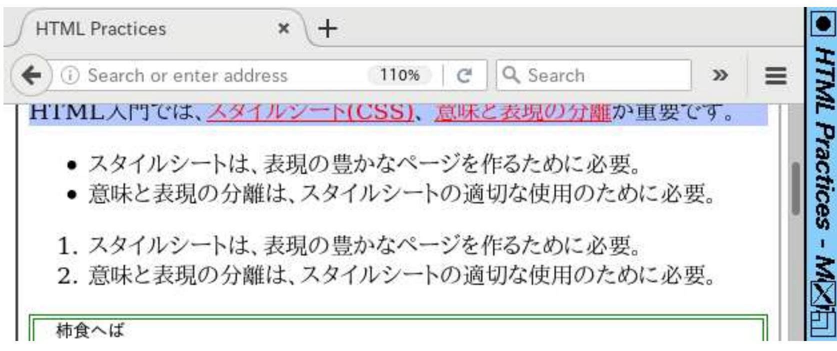


図 12.8: ul 要素と ol 要素による箇条書

一方、LaTeX にある description 環境に相当するものもあります。

```
<dl>
<dt>Boolean 値</dt>
<dt>論理値</dt>
<dd>「真偽値」とも呼ばれる。true/false のいずれか。</dd>
<dt class="imp">整数値</dt>
<dd>一定の範囲の正・負・ゼロの整数。</dd>
</dl>
```

こちらは LaTeX と異り、1 つの項目に見出しを複数つけることもできます (図 12.9 上)。

次に表について説明しましょう。こちらはもう少し複雑で、表全体を表す **table** 要素、その中の 1 行ずつを表す **tr** 要素、そして行の中に入る各セルを表す **th** 要素または **td** 要素の 3 レベルから成ります。HTML の上で次のように縦横に揃えて書くと分かりやすいでしょう (「`border="2"`」という指定は表の罫線を幅 2 で描くという指定。無いと罫線も描かれない)。

```
<table border="2">
<tr><th>A</th><th>B</th><th class="imp">C</th></tr>
<tr><td>111</td><td>2222</td><td>3333</td></tr>
</table>
```

th と td の違いは、前者が見出し用、後者が一般用で、見出し用の方が少し強調表示になります (図 12.9 下)。また、th も td も「`colspan="個数"`」「`rowspan="個数"`」という属性が指定でき、これによって LaTeX と同様横や縦にまたがったセルが作れます。

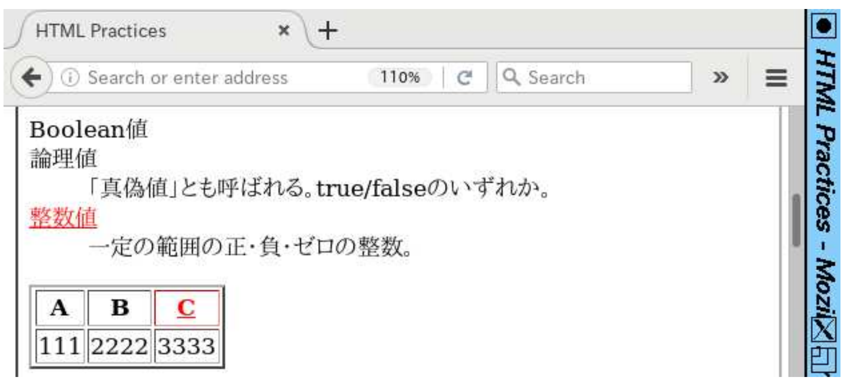


図 12.9: dl 要素の箇条書、table 要素

12.2.5 CSS 指定の追加

最後に、主に table と div で活用する CSS 指定についても追加しておきます (最初のものが table 用、残りは div でよく使います)。

- `border-collapse: collapse` — table 要素専用で、これを指定した表は隣接するセルの境界が 1 本の線になる (これを使う時は th や td の border も併せて指定することが多い)。
- `position:` — 位置指定。`absolute`(絶対位置を指定)、`fixed`(画面上での絶対位置を指定)、`relative`(本来あてはまる位置からのズレを指定) のいずれかが指定できる。
- `top:`、`left:`、`bottom:`、`right:` — 要素の上端、左端、下端、右端の位置を指定。
- `width:` 長さ、`height:` 長さ — この要素を整形する幅や高さを指定できる。div に対して使うことが多い。
- `float:` — 流し込みの指定。`left`(左に寄せて右に流し込む)、`right`(右に寄せて左に流し込む)、`none` が指定できる。
- `overflow:` — 高さを指定したとき、その中に入る内容が高さを超えたときの扱い。`visible`(そのまま見える)、`hidden`(見えなくする)、`scroll`(スクロールさせる) 等が指定できる。

演習 3 自分が作成した Web ページに関する説明をおこなう LaTeX 文書を作成しなさい。Web の画面を取り込むこと。また、Web ページには演習 1、演習 2(の小課題) の内容、または以下のいずれか 1 つ以上が含まれること。

a. id 機能または class 機能を使って「特定の何かだけ表現が他と違う」ようにしてみる。
b. 箇条書のどれかの形を使ってみる。
c. 表を使ってみる。

LaTeX 文書に画面を取り込むのは次の方法をおすすめします。

1. ブラウザで取り込みたいページ画面を表示させる。図が大きくなりすぎないようにブラウザの窓はかなり小さくした方がよい。

2. 「`import -compress rle fig1.eps`」のようにファイル名を指定して `import` コマンドを発行し (マウスカーソルが「＋」形になる)、続いて取り込みたい窓の上をクリックするとその窓の画像が指定したファイルに取り込める。[3]
3. LaTeX のソースの中で図を入れたい箇所に次のように挿入指定 (幅とかファイル名は適宜変える。もっと別の入れ方でも入っていればよい)。

```
\begin{center}
\includegraphics[width=12cm]{fig1.eps}
\end{center}
```

画面を取り込む方法は `import` コマンド以外でも構いません。その場合は取り込んだ画像は.png や.jpg になると思いますので、次のコマンドで.eps に変換してください (LaTeX には eps しか入れられないので)。

```
convert -compress rle ファイル名.png ファイル名.eps
```

課題 **12A**

今回の課題は「演習 3」ですが、その文書の内容として「演習 1」「演習 2」「演習 3」に含まれる小課題 (合計で 10 個) の中から 1 つ以上を選択し、結果をレポートとして報告するものとしてください。そして、LaTeX による整形結果を PDF ファイルにして、LMS の「assignment # 12」の箇所からアップロードしてください。以下の内容がこの順に含まれるようにしてください。

- 題名「コンピュータリテラシレポート # 12」、学籍番号と氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も脚注などで別途書く。)
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 以下のアンケートに対する回答。

  Q1. HTML によるページの記述はどれくらい知っていましたか。今回やってみてどうでしたか。
  Q2. CSS による表現の指定はどれくらい知っていましたか。今回やってみてどうでしたか。
  Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

なお、課題はグループでやって構いません。その場合も、(メンバー氏名を明記した上で) レポートは必ず各自で執筆してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。

[3] `-comppress rle` はランレングス符号化 (run-length encoding) による圧縮指定で、指定しないとファイルが大きい。

＃13　Webと情報アーキテクチャ

今回の目標は次の通りです。

- 相対 URL や外部メディアの参照について学ぶ — ページに画像を使えるようになることで、見栄えのする (見てもらいやすい) サイトが作れます。
- Web サイトの構成および情報アーキテクチャの考え方について学ぶ — 定番なサイト構造を知っておくことで、労力をかけずにわかりやすいサイトが作れます。
- CSS によるページデザインについて学ぶ — CSS を用いてどのページでも統一された見え方のページ群が作れるようになり、また 1 箇所の変更で全部の表示を調整することも可能になります。

13.1　外部ページ/外部メディアの参照

13.1.1　絶対 URL と相対 URL ex

これまで、Web ページを表示させたり HTML 中で a タグに記載する URL は次のようなものでした。

```
http://example.com/aa/bb/mypage.html
スキーム　　ホスト　パス
```

このような (`http:`などの) スキームから始まる URL を**絶対 URL** と呼びます。すべてのサイトは絶対 URL で指定できますが、次のような場合には絶対 URL は不便です。

- 1 つのサイトが複数のページからできていて、相互にリンクで行き来するような場合。
- ページの HTML に加えて、そこで使用する画像などのファイルを参照している場合。

どのように不便かというと、(1) リンクなどで指定する URL が長くなり記述が繁雑であることと、(2) サイト内のファイルをまとめてよそのサーバに引越したときに、全部リンクを直さなければいけないことです (絶対 URL が書いてあるということは、リンクをたどると元のサーバから読みに行きますから、引越しの意味がありません)。

そこで Web では**相対 URL** も使うことができます。たとえば上の URL のファイル `mypage.html` にリンクを書くとき、相対 URL ではその置いてあるディレクトリ `http://example.com/aa/bb/` が「起点」となります。そして、あとは Unix のパス名と同じようにそこを起点に対象とするファイルを指定します (`..` も使えますが、ただしホストやスキームは変更できません)。表 13.1 に上の例のページを起点としたときの相対 URL とそれを解釈して絶対 URL に直したものの対照を示します。

基本的に Unix のパス名のようなものだと考えていいのですが、(1) まったく「`/`」がなくても相対 URL であること、(2)「`/`」で始まるものも相対 URL であること (その Web サーバの公開用トップディレクトリからたどる意味になる)、などが違います。

なお、一番下の 2 つのようにパスが「`/`」で終わる場合はディレクトリを指しますが、そのときに何が表示されるかはサーバの設定次第で、多くのサーバではそこにある `index.html` というファイルの内容を返すように設定されています。

表 13.1: 「http://example.com/aa/bb/」を起点とする相対 URL の解釈

相対 URL	絶対 URL
page2.html	http://example.com/aa/bb/page2.html
fig1.png	http://example.com/aa/bb/fig1.png
FIGS/fig1.png	http://example.com/aa/bb/FIGS/fig1.png
../FIGS/fig1.png	http://example.com/aa/FIGS/fig1.png
../../FIGS/fig1.png	http://example.com/FIGS/fig1.png
/aa/FIGS/fig1.png	http://example.com/aa/FIGS/fig1.png
./	http://example.com/aa/bb/
../	http://example.com/aa/

13.1.2 画像の使用 ex

だいぶお待たせしましたが、ここで Web ページにおける画像の使用方法について説明します。なぜここまで待ったかというと、画像は HTML の中に直接は入れられないのでサーバ上に別のファイルとして置く必要があり、その参照はふつう相対 URL によるからです。

さて、まず画像ファイルの形式についてですが、ピクセル画像では **PNG**、**GIF**、**JPEG** のいずれかの形式、ベクトル画像では **SVG** 形式のものを使用してください。これらは多くのブラウザで共通にサポートされています (SVG は未サポートなブラウザがまだあります)。

図 13.1: リンク先として画像を指定した場合

次に、Web ページで画像を使うやり方ですが、次の 3 種類があります。

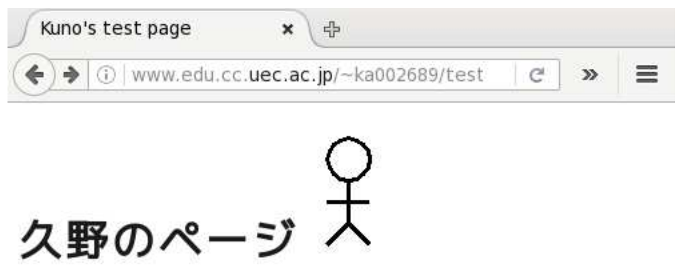


図 13.2: 埋め込み画像 (img 要素) として画像を指定した場合

(1) リンク先として — `<a href="fig1.jpg">...</a>`のようにリンク先として画像を指定した場合、リンクを選択するとブラウザ画面にその画像が単独で表示されます (図 13.1)。

(2) 埋め込み画像 — `<img src="fig1.png" alt="説明">` のようにして、**img 要素**を使うことで、ページのその場所に画像が埋め込めます (図 13.2。この要素はインライン要素です)。なお、`alt` 属性は画像が表示できない時に表示したり、目に障害がある人が読み上げブラウザを使っているときに読み上げたりするのに使います。

(3) 背景画像 — CSS で「`background-image: url(fig1.gif)`」などのように指定すると、その指定した要素の「背景模様」として指定した画像が繰り返し敷き詰められて表示されます (図 13.3)。ページ全体の背景にしたければ、セレクタで `body` や `html` を指定してください。なお、繰り返し敷き詰めたくなければ、同じセレクタに対して `background-repeat:` プロパティとして `no-repeat`(繰り返さない)、`repeat-x`(横方向のみ繰り返す)、`repeat-y`(縦方向のみ繰り返す) を指定できます。通常は `repeat`(縦横とも繰り返す) です。

図 13.3: 画像を背景画像として指定した場合

演習 1 Web ページに画像を入れる演習として、次のものから 1 つ以上やってみなさい (題材の画像ファイルは各自工夫して入手のこと)。

a. 複数の画像をリンク先としたリストを含むページを作ってみる。なお、`<a href="..." target="名前">...` のように a タグに **target 属性**を指定するとどのような効果があるか試して検討してみること。

b. 画像を img 要素を使ってページ内に埋め込んでみなさい。このとき、この要素に対して CSS で枠や空白あけを指定して見やすくしてみなさい。さらに「`float: left`」「`float: right`」を指定するとどのような効果があるか試して検討してみること。

c. 画像を適当な要素の `background-image` として指定して効果を観察しなさい。複数の要素 (たとえば `body` とページの中にある `h1` や `p` など) に別々の背景画像を指定して、見た目だけでなくページ内容の読みやすさのためにはどういう配慮が必要か検討しなさい。画像のつぎ目の柄合わせのために位置を微調整する機能が CSS にあるか探してみるとなおよい。

13.2 情報アーキテクチャ

13.2.1 情報アーキテクチャとサイト構造 ex

情報アーキテクチャ(information architecture) とは、広義には情報を分かりやすく伝え、受け手が情報を探しやすくするための技術全般を指します。また狭い意味では、Web サイトの構築に先立ち、情報を分類し、意味を整理し、サイト構造やサイトの使われ方を検討する分野を指す場合もあります。

なぜこのような分野が必要になったのでしょうか。もともと WWW はハイパーテキストから生まれており、どのページのどこにでも他のページへのリンクを埋め込めることが特徴でした。これを活かして、多数のページが互いにつながり合った構造が**ネットワーク構造**です (図 13.4 左)。しかし

WWW でこのような構造を作ると、今居る場所が分からず (迷子)、混乱が生じることが分かってきました。そこで、ページを直線状につなげた**線形構造** (図 13.4 中)、ページを大分類→中分類→小分類のようにテーマに従って掘り下げていく**階層構造** (図 13.4 下) が生み出され使われるようになりました。

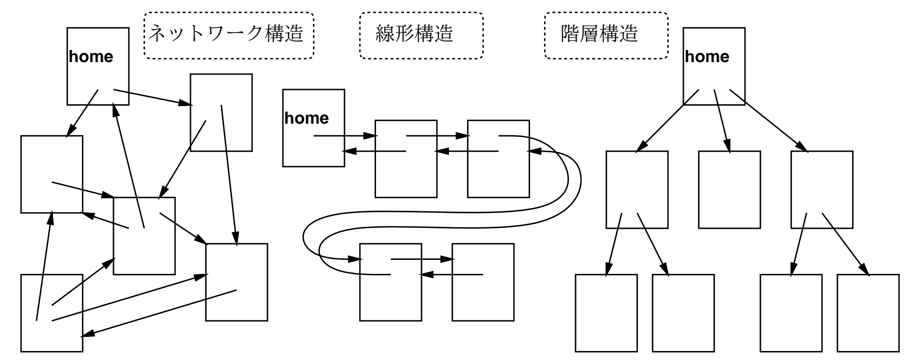


図 13.4: Web サイトの代表的な構造

線形構造はインタビュー記事のように流れが直線的で分量がさほど多くない場合は分かりやすいのですが、分量が多くなると先の方までたどって行くのが大変です。そのために目次を用意してそこまで飛べるようにするなどの工夫がなされます。

階層構造はファイルシステムのところでも述べたように、大量の情報を整理して納めるには適しています。そのかわり、すべての情報をひととおり見るようなことはやりにくいです。この対策として、情報はすべて一番下の「葉」のところにいれて、そこを線形構造で結ぶなどの工夫もあります。しかしそうすると、中間の分類を下までたどっていくのに手間がかかります。またそもそも、何を規準に階層構造に分類するべきか、ということも自明ではないかも知れません。

このような構造の問題や、個々のページの使いやすさの問題まで含めて、どのようにしたらよいかを考えて実現するのが、情報アーキテクチャの分野であるといえるでしょう。

なおついでですが、図で home と書いてあるのは一連のページの入口となるページです。本来はこのようなページを「ホームページ」と呼ぶはずですが (またはブラウザの home ボタンを押したときに移動するページの意味もある)、日本ではなぜか WWW 自体のことを「ホームページ」と呼ぶ習慣があるので、どのような意味でこの語を使っているのか、そのつど注意が必要です。

13.2.2 ページナビゲーション ex

ナビゲーション (navigation、航行) とは一般には進路を制御して目的地に向かう作業をいいますが、Web の場合は「行きたいページに行かせてくれるための機能全般」を指します。

とくに、前節で述べた「線形構造」「階層構造」の場合は、どのページにも同じような形のリンクが必要になります。具体的には、線形構造であれば各ページに「前へ」「次へ」が必要ですし、階層構想では中間のページにはそれぞれの「子供」に行くリンク、そしてトップ以外の全てのページには「上へ」行くリンクが必要です。また、**パンくずリスト** (crumb list) と呼ばれる、一番上から現在のページまでの経路のリストもあると位置が分かりやすくなります (図 13.5 の囲んだ部分)。

このような、サイトの設計に応じてどのページでも同じように使えるリンクのことを**ナビゲーションリンク**と呼びます。ナビゲーションリンクは頻繁につかうので、ページ内でナビゲーションリンクのありかがすぐ分かることは使いやすさのために重要です。このため、ナビゲーションリンクを集めた領域である「ナビゲーションバー」を作成し、どのページでも同じ場所 (ページの先頭、末尾など) に配置するという工夫は多くのサイトでなされています。

演習 2 ページ構成に関する次のテーマから 1 つ以上やってみなさい。

a. 線形構造のサイトの例として「浦島太郎」のストーリーをもとにしたサイトを作る。入口ページのほかに「亀を助ける」「竜宮城に連れていってもらう」「乙姫と楽しくすごす」「乙

図 13.5: パンくずリストの例

姫に別れを告げる」「故郷に戻ってみると」の 5 ページを作り線形構造につなぐ。ページ内容は自由だが「前へ」「次へ」のリンクは必須とする (最初と最後は適宜修正)。浦島太郎を知らない場合はクラスメートに説明してもらうか、別のストーリーに変えてもよい。

b. 階層構造のサイトの例として「食材」のサイトを作る。入口ページのほかに「魚介類」「魚」「あじ」「しゃけ」「貝」「あさり」「しじみ」「肉」「牛肉」「鶏肉」「野菜」「キャベツ」「たまねぎ」の 13 以上を作る (個々の食材は別のものにしてもよいし、追加してもよい)。ページ内容は自由だが、すべてのページに「上へ」のリンク、そして分類のページにはその中の各種別へのリンクが必須とする。

c. この課題は a または b をやった後にそれに加えて選ぶこと。a の線形構造をやった場合は、入口ページに「目次」を追加しなさい。b の階層構造をやった場合は、個々の食材のページだけを順番に線形構造でながめて行けるようなリンクを追加するか、またはパンくずリストを追加しなさい。

13.3 CSS と HTML によるページレイアウト

13.3.1 ページの構成要素のグループ化

今回の Web ページ制作では CSS を使ってページ内容の配置を行います。また、CSS では (既に学んだように) 背景色、文字色の配色設定も行えます。ここではこれらを実際に (簡単なものですが) やってみましょう。

まず、CSS でレイアウトする場合、各ページの内容はいくつかの「部分」に分けて作成することになります。たとえば次のような部分が考えられます (デザインに応じて、これらのうちから 2〜3 個を用意するのが普通ですが、さらにこれら以外のものを加えることもあるかも知れません)。

- トップバナー — ページの先頭に置き、タイトルやロゴなどを目立つように配置する。
- メイン (本体) — ページの本体つまり主たる内容を入れる。
- フッター — ページの最後に置き、著作権表示やナビゲーションバーなどを入れることが多い。
- 左サイドバー — ページの左端に細長く置き、ナビゲーションバーやその他各ページで共通に使う機能、リンクを置くのに使う。
- 右サイドバー — ページの機能が多くて左サイドバーだけでは縦長になりすぎる (スクロールしないと見えなくなる) 場合に、右にもサイドバーを置くことがある。

分かりやすく無難な方法として、ページを縦/横に区切りながらこれらの要素をはめ込んで行く**ブロックレイアウト**と呼ばれる方法があります (図 13.6)。これに対し、縦横の配置にこだわらずもっと自由なデザインで配置する方法もありますが、デザインの難易度は高くなります。以下ではブロックレイアウトを前提とします。

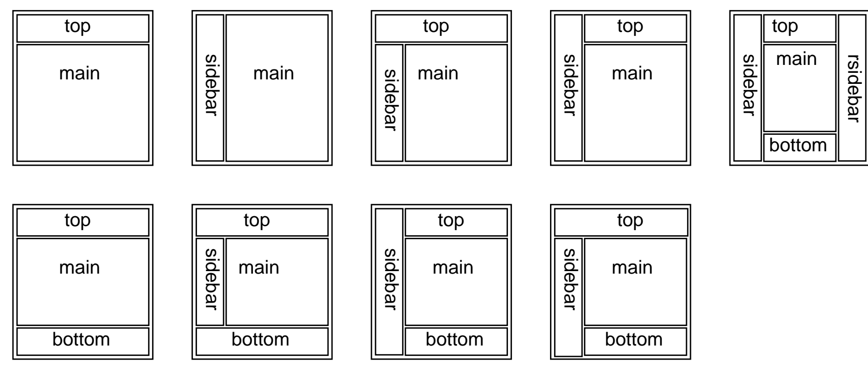


図 13.6: 標準的なブロックレイアウト

これまで HTML ページはいきなり見出し等から始まっていましたが、レイアウトを行う場合はその各ブロックをそれぞれ何らかの要素で囲む必要があります。

HTML 4(1 つ前のバージョンの HTML) まででではそのような目的に使えるものが `div` しかなかったので、すべて `div` で囲み、`id` または `class` 属性を指定して、それぞれ「`#id` 名」または「`.` クラス名」を CSS のセレクタで指定しました。それでは分かりづらいので、HTML 5 (現在のバージョン) になったときに、目的別に次のような要素が追加されています (これらで不足する場合は従来の方法を併用します)。

- `<section>...</section>` — 1 つの本文セクションを表す。
- `<article>...</article>` — 1 つの本文記事を表す。
- `<aside>...</aside>` — 本文とは別の内容を表す。
- `<nav>...</nav>` — ナビゲーション部分を表す。
- `<header>...</header>`、`<footer>...</footer>` — ヘッダ、フッタを表す。

13.3.2 CSS グリッドによるレイアウト

実際に図 13.6 のような配置を実現するのには、以前は CSS のさまざまな「技」を駆使する必要があり面倒だったのですが、今は **CSS グリッド**と呼ばれる機能を使って簡単に構成できるようになっています。それには、たとえば図 13.6 の最後の配置を実現するのであれば、ページ全体 (body 要素の内側) を 1 つの div 要素にした上で、次の CSS を指定します (ページ全体を囲む div 要素に `class="main"` を指定することにします)。

```
.main { display: grid; width: 100vw; height: 100vh;
        grid-template-areas: "a a" "b c" "b d";
        grid-template-columns: 8em 1fr;
        grid-template-rows: 6em 1fr 3em }
```

まず 1 行目で、この範囲をグリッドで配置することを指定し、またこの要素をブラウザの画面全体 (幅 100%、高さ 100%) と指定します。次の `grid-template-areas` で、図 13.7 のように、横 2 カラム、縦 3 行のグリッドを作り、そのそれぞれのます目に入る要素に a, b, c 等の名前をつけています (同じ名前をつけたものは複数のグリッドにまたがって配置される)。指定の`"a a" "b c" "b d"`と図 13.7 の各セルの記号が対応していることを確認してください。[1] その次の `grid-template-columns`、`grid-template-rows` は各グリッドの幅と高さの指定で、「`1fr`」と指定されている部分が直接大きさを指定した部分の「残り」になります。

[1] 1 つの文字列が横 1 行を表し、文字列が 3 つあることで縦 3 行、文字列の中に名前が 2 つずつあることで横 2 カラムであることを表します。

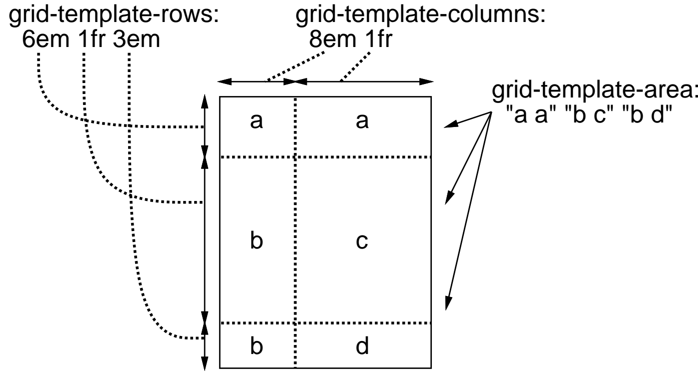


図 13.7: グリッド指定の意味

なお、長さの指定においては単位として「em(文字 m の幅)」「ex(文字 x の高さ) など文字サイズを基準にした指定値を使う方がよいでしょう。というのは、「px」(ピクセル数) や「cm」などの指定だとその大きさに固定されてしまうのに対し、文字サイズ指定なら読み手がブラウザのメニューで文字を大きくしたとき対応して長さが大きくなって釣合いが保てるからです。

これらでグリッドを定義したあとは、それぞれの要素に grid-area として先の a, b, c 等の名前を指定すれば、その要素が指定したグリッドの位置にはめ込まれます。実際に見てみましょう。

```
<!DOCTYPE html>
<html><head>
<meta charset="utf-8">
<title>sample</title>
<style type="text/css">
body { margin: 0px }
.main { display: grid; width: 100vw; height: 100vh;
        grid-template-areas: "a a" "b c" "b d";
        grid-template-columns: 8em 1fr;
        grid-template-rows: 6em 1fr 3em }
header { grid-area: a; background: rgb(200,240,180); padding: 1em }
aside  { grid-area: b; background: rgb(180,220,220); padding: 1em }
section { grid-area: c; background: rgb(210,255,240); padding: 1em }
footer { grid-area: d; background: rgb(200,240,180); padding: 1em }
address { text-align: right }
</style>
</head><body><div class="main">
<header><h1>A Sample Page</h1></header>
<section><p>A</p><p>B</p><p>C</p></section>
<aside><p>a</p><p>b</p><p>c</p></aside>
<footer><address>Computer Literacy</address></footer>
</div></body></html>
```

この例を表示したようすを図 13.8 に示します。この例ではバナー、本体、サイドバーの内容は仮に入れたものであり、本来ならこれらの部分には、見出し、段落、箇条書き、リンクなどのタグを使って、それぞれの内容を記述していきます。

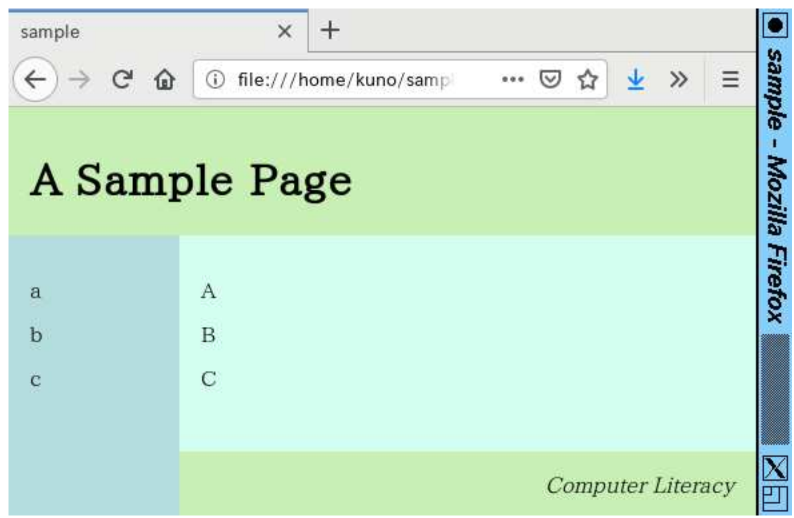


図 13.8: ブロックレイアウトの例

13.3.3 CSS のマージンとパディング

それぞれの要素内を見やすく配置するのは、やはり CSS の機能を使って行います。このとき知っておく必要があるのは次のプロパティ程度です (マージンとパディングの関係は図 13.9 を見てください)。少なくともグリッドに使った要素にパディングを指定しないと、要素がグリッドの端にぴったりくっついてしまいます。

- `margin:` 上 右 下 左 — 要素の「外側の」アキを上/右/ 下/左の順で指定。数値が 2 個の時は「上下」「左右」をそれぞれ指定、1 個のときは 4 つとも同じ値を指定。
- `padding:` 上 右 下 左 — 要素の「内側の」アキを指定。指定方法は margin と同様。
- `background:` 色 — 背景色の指定は必要に応じて。

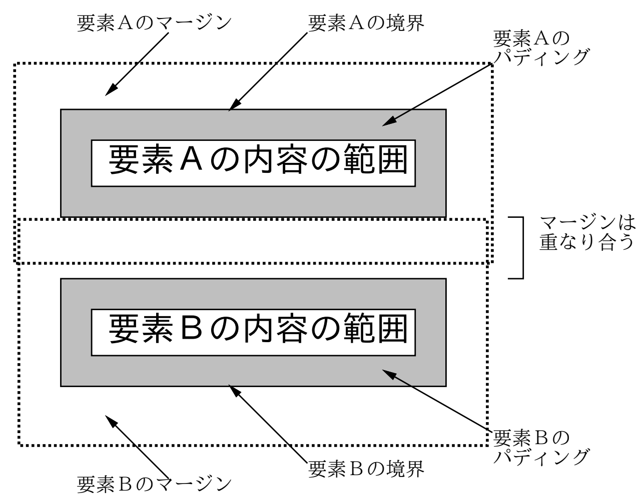


図 13.9: マージンとパディングの関係

演習 3 ブロックレイアウトの例をそのまま動かしてみなさい。うまく動いたら、`grid-template-area` を変更して配置を修正したり、サイドバーを右側にしたり、ブロックの数を増減してみなさい。最終的には、自分が自分のサイトを作るときに使いたいと思う色づかいと配置のレイアウトを構成してみなさい。

課題 13A

今回の課題は「演習 1」「演習 2」に含まれる小問か「演習 3」(合計で 7 個) の中から 1 つ以上を選択し、結果を LaTeX によって整形したレポートとして報告してください。そして、整形結果を PDF ファイルにして、LMS の「assignment # 13」の箇所からアップロードしてください。以下の内容がこの順に含まれるようにしてください。

- 題名「コンピュータリテラシレポート # 13」、学籍番号と氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も脚注などで別途書く。)
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 以下のアンケートに対する回答。

  Q1. リンクや画像の使用についてどれくらい知っていましたか。
  Q2. CSS によるブロックレイアウトについてはどうでしたか。どのようなページデザインがよいデザインだと思いますか。
  Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

なお、課題はグループでやって構いません。その場合も、(メンバー氏名を明記した上で) レポートは必ず各自で執筆してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。

＃14　Webサイトの設計/製作(総合実習)

今回の内容は「総合実習」であり、グループで分担して Web サイトを設計・製作し、その結果をレポートにして頂きます。今回の目標は次の通りです。

- Web サイトの設計の流れや行うべき作業を理解する — 今後研究や仕事でサイトを作成する際に有用な知識です。
- 共同での設計作業と分担しての製作作業を経験する — 課題限りでのグループ作業や分担作業を経験して頂くことは今後の研究等での同様の活動の下地となります。
- 共同部分と分担部分を明確に区分して報告を作成する — 研究等でも同様の活動でも必要となるスキルです。

14.1　Webサイトの設計と制作

14.1.1　Webサイトの設計/制作とは

HTML と CSS の基本的な機能についてひととおり学びましたが、ではそれで Web サイトが作れるようになったと感じられるでしょうか? 多くの人は NO と答えると思います。何が足りないのでしょう?

たとえば、授業に関して告知をするとか、読み手がどうしても見ないと困るような情報を提供するのなら、とにかく HTML を書いてそこに必要な情報が含まれていさえすれば、最低限の用事は足りるかも知れません。しかしそのようなサイトでは、見た目も楽しくなく、企業等が「顧客に向けて」情報発信するという目的には不十分です。

たとえば、商品告知のパンフレットを作るとしたら、(1) 企画立案、(2) デザインやスタイルの決定、(3) 内容の制作、(4) 印刷・配布、のようなワークフローがあるはずです。たとえば、いきなりワープロソフトで必要な情報だけとにかく打ち込んでプリントし、そのまま印刷所に回すことなど考えられませんね?

それと同様に、「真面目な目的で」Web サイトを制作するのであれば、次のような**ワークフロー**を持つのが普通です。

1. コンセプト (企画立案) — サイトの目的、ターゲット読者、その他の前提を明確に設定。
2. デザイン/仕様策定/スタイルガイド作成 — サイトの構成、ページのデザインの枠組み、コンテンツの中で守るべき約束などを決定し、これらをまとめて文書化する。
3. 設計/制作 — 実際に作成するページを決めて内容を用意し制作。
4. 公開/運用 — ページを公開し、フィードバックや情報の変化に応じて手直しして行く。

今回はこれらについて、ひととおり簡単に見て行くことにします。

14.1.2　コンセプト(企画立案)

何の制作でも同じですが、Web サイト制作に当たり、次のことがらを明確にする必要があります。これが**企画立案** (conceptual planning) です。

- 制作 (情報伝達) 目的 — 何を目的として/どんな情報を伝達するために、サイトを作るのか。
- ターゲット — 情報伝達の対象となる相手はどのような層か。

自分で自分のために制作するなら、これらは自分で考えて決めればよいのですが、注文を受けて制作する場合は顧客 (注文者) と十分話し合い、何が求められているのかを明確にしておく必要があります (さもないと制作が終わってから「何か違う」という話に…)。**ターゲット** (対象読者) については、つい「なるべく多くの人」と考えてしまいますが、ターゲットが絞れていないとサイトの方向づけも万人向けとなってしまい、焦点が定まらずに最も必要な層にアピールし損ないます。ターゲットの分類基準としてはたとえば次のようなものがあります。

- 性別、年齢層 (子供、生徒、学生、社会人、シニア、…)
- 働いている/家庭にいる、都会/地方などのカテゴリ
- コンピュータ環境 (PC、スマホ、タブレット…)

コンピュータ環境のなかでも、画面サイズはデザイン上大きな制約となります。ユーザは画面一杯にブラウザを開くわけではないので、たとえ幅が 2048 ピクセルの画面が普通だったとしても、デザイン幅は 1200 ピクセル程度にした方がよいでしょう。普通の PC であればこれより狭い画面は少ないでしょうけれど、タブレットやスマホでも見てもらうには、より小さい画面サイズを前提とします。

一方で、小さいサイズで大丈夫なように設計したとしても、大きいディスプレイを使用しているユーザにとって間延びして見えるのもよくありません。このあたりは印刷デザインなどと異なる Web デザイン固有の悩みと言えるでしょう。

14.1.3 デザイン・スタイルガイド

コンセプトが明確になったら、コンセプトに合ったデザインを考えます。顧客がいる場合は、いくつか案を用意してプレゼンテーションを行い、顧客が OK を出したものを選ぶことになるでしょう。デザインについてはここでは詳しくは述べませんが、たとえば次のような要素があります。

- ロゴデザイン — 顧客の紋章やロゴマークに既定のものがあればどう活かすか、活かすとしてもどうアレンジするか、タイトルや本文などに使うフォント (フォントごとに印象が決まってくる) の選定なども課題です。
- 色彩計画 — 顧客のテーマカラーなどがあれば参考にして、どのような**配色** (カラースキーム) を使うかを決めます。ターゲットやコンセプトに応じて決まる部分もあります (活動的→目立つ色やコントラスト、癒し→アースカラー、パステルカラー等)。
- ページに盛り込む情報量や見せ方 — たとえば子供に見せるページなら、文字は少なく、絵や図を多くして、なおかつ 1 ページ当たりの情報量が多すぎないような配慮が必要です。

これらをデザインしている段階では、実際に HTML で制作するよりも、見え方をチェックすることが目的なので、お絵描きソフトでページの見た目を作ってみてチェックすることが普通のようです。ページデザインについては既に取り上げていますが、CSS で (そして自分の技能で) 記述できるようなデザインを選ぶことも実務上は大切です。

サイトの構成 (情報アーキテクチャ) についても既に学んで来ましたが、実際にサイトを作るにあたっては、具体的なページ内容を (箇条書程度のものでもよいので) 描いたメモ紙を机上に並べてみて、構造を決めるなどの方法がおすすめです。

全体の構造に加えて、個々のページのデザインもこの段階で決めて行きます。小規模なサイトではページデザインも 1 種類だけ (または入口ページだけ別デザインで 2 種類) にするかも知れませんが、大規模になってくると複数のデザインを用意するかもしれません。ただし、デザインが複数になっても全体として「1 つのサイトである」というまとまり感が損なわれないように、レイアウトや色づかいに共通性を持たせることが必要です。

ナビゲーションリンクについても既に学んでいますが、具体的なサイト構成を決めるときに、どのようにナビゲーションするかも検討しているはずですから、それに合わせて各ページにどのようなナビゲーショリンクを入れるかを決め、ページデザインに含める必要があります。

これら見た目のデザインに加えて、中身に入る文章や図などにも**指針** (ガイドライン) が必要です。たとえば「だ・である」「です・ます」のどちにするか、「お客様」か「あなた」か、その他複数の言い方がある用語は何に統一するか (用語集を作って基準を示す)、図や写真のサイズや注意すべき内容 (個人の顔が判別できる場合など)、各ページに入れるコピーライトや個人情報保護の注記の書き方、などがこれに相当します。

このように、1 つのサイトを作るだけでも非常に沢山「決めるべきこと」があるわけですが、それらをまとめて 1 つの文書 (**スタイルガイド**) とします。実際の制作に入ったら多くの人が協力/分担して作業するわけですが、スタイルガイドをきちんと決めておくことで、各ページごとにバラバラにならず、統一した方針のサイト作成が可能になるわけです。

14.1.4　設計・製作

大枠が決まったら、いよいよ個別のページについて検討します。このときも、いきなりコンピュータに向かって製作するのではなく、次のような段階を踏みます。

(1) ページ構成・ページ内容の設計 — 実際に作成するページとそれらのつながりを決める。

(2) 素材の収集と作成 — ページに入れる文章を準備したり、画像 (写真・図) などを用意する。

(3) 製作 — 設計に合わせて内容を組み込み、ページを作る。

(4) チェック・レビュー — 完成したページをチェックする。

まず (1) については、製作する 1 ページごとにメモ紙を用意し、そこにページタイトル・内容や使用する素材などを書き込んで並べます (既に前の段階でやっているのならそれを流用できます)。これによって、個別のページの取捨選択ができ、ページのつながりが確認できます。どこへのリンクを貼るかもメモ紙に書き込んでおきます。

次に (2) として、実際に文章を打ち込んだり画像を用意します。他人の文章を流用するわけにはいかないので、自分で構成を考えて作文する必要があります。内容によっては、この段階でインタビューや取材を行うことになります。画像については、写真は撮影し、図は描いて、サイズやファイル形式を調整します)。

これらの準備ができてからようやく (3) に進み、ページを製作します。ページの大枠はデザイン段階で決まっているはずなので、それに従って文章や画像を入れていけばよいはずです。また設計時に決めたリンクも忘れずに含めるようにします。

単に製作するだけでなく、製作した内容が正しいかどうかチェックすることも必要です。その際には、漫然と見るだけでなく、次のような項目を含めた**チェックリスト**を用意して、項目ごとに OK かどうか見て行くのが普通です。

- 各ページに内容に対応したタイトルがついているか。
- 各ページの内容は設計時に予定したものになっているか。
- 各ページに含まれる画像には説明文や alt 属性がついているか。
- 各ページに含まれるリンクは正しくたどれるか。

自分で作ったページを自分で見ると甘くなるので、他人に**レビュー**してもらう方が問題を見つけやすくなります。

14.1.5　公開・運用・保守

チェックが終わって OK になったら、サイトを実際に公開します。サイトの公開は通常、サイトを構成する各ファイルをまとめて Web サーバに転送し、そのサイト用に用意したディレクトリに置くことで行います。設置したら、各ページが計画通りに見られ、リンクも正しく働くことを再度チェックします。

サイトの**運用** (operation) については、動的処理を行うページであればユーザからのデータがきちんと処理できていることを確認するわけですが、今回のように単に HTML と画像を公開するだけ (静的なページだけ) であれば、サーバが止まって見えなくなっていないかを監視する程度でしょう。

ただし、いつまでも公開した時のままだと誰も見てくれなくなりますから、定期的に新しい内容を追加したり古い内容を更新するなどの作業が必要となります。また、外部のページに対するリンクの場合、その外部ページが移動したりなくなったりしたときには、相応に対処しないと「たどれないリンク」になってしまいます。このような、ページ公開後の**メンテナンス** (**保守**) 作業も、使いやすいサイトを維持する上では重要です。

課題 14A

今回の課題のために、3〜5 人のグループを構成してグループ単位で 1 つずつ Web サイトを作っていただきます。欠席などで乗り遅れた人は、どこかのグループに加えてもらって、「分担」を分けてもらってください。やるべきことを記します。

(1) グループ内で相談して、コンセプト (企画立案)・デザイン・設計までの作業をやってください。

(2) テーマは任意ですが、テーマに関心を持てないと苦痛ですから、全員がそれならやりたいと思うものにしてください。参考までに皆様がシンパシーを持てそうなテーマの例を挙げます。

- 学内または学校周辺の「ちょっと変わったところ」を紹介する。
- おすすめな飲食店の紹介 (カテゴリは広くとっても絞ってもよい)。
- スポーツ、工芸、音楽、エンタメ、食材、料理、旅行など。

(3) デザインはこれまでに CSS でやってきて「できそう」なものを念頭に配色も含めて皆で提案し、1 つ選んでください。選んだ設計は写真をとっておいてください (またはコンピュータ上の画像ならあとで皆で共有)。選ばれた人はデザイン担当として CSS を作りますから、担当ページ数は減らしてもいいです。なお、入口ページが必要ですが、入口ページのデザインは他と変えても (たとえば CSS なしの真っ白でも) 同じでもいいです。入口以外のページは全部同じデザイン (CSS) を用いてください。入口ページにはグループ全員の氏名と学籍番号を記載してください (入れ方はちゃんと読み取れる限り自由)。

(4) 設計は入口ページと個別ページを含めて 1 人あたり 2〜3 ページ担当してください (増やしたければご自由に)。製作する 1 ページごとにメモ紙に内容をおおまかにメモして並べて構成を決めてください。今回は小規模なのでサイト構成と設計を一緒にやって構いません。各メモごとに HTML ファイル名と担当者名を書き込んでおくこと。メモを机に配置したところを写真にとっておいてください。

(5) 製作から先は個別作業となります。デザイン担当は CSS 記述部分 (`<style>`〜`</style>`) を全員に送付してください。誰か 1 人が収集担当となり、皆で製作した HTML や画像をその人が受け取り、その人の `public_html` 以下 (サブディレクトリを作ることを勧めます) に設置してください。

(6) サイトが完成したらレポートを作成してください (下記参照)。なお、諸般の事情によりサイトが完成しなかったとしても最初の話し合いさえしてあれば個人のレポートは出せるように配慮しました。

レポートは LaTeX により整形し、PDF ファイルにして、LMS の「レポート # 14」の箇所からアップロードしてください。以下の内容がこの順に含まれるようにしてください。

- 題名「コンピュータリテラシレポート # 14」、学籍番号と氏名、提出日付を書く。
- グループで決めたテーマ、およびグループ全員の名前と学籍番号を書く。完成したサイトの URL を書く。

- グループ作業 (コンセプト、デザイン、設計) の内容を報告する。画像や表など共同作業の成果は同じものを使ってよいですが、説明する文章はそれぞれで書くこと。
- 自分が担当したページの報告を書く。各ページともブラウザで表示しているようすを図として取り込むこと。どのように考え、どのように製作し、どういう工夫をしたかなどを書くこと。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 参考文献セクション (あれば)。
- 以下のアンケートに対する回答。

Q1. Web サイトをグループで協力して製作してみて、どのようなことが分かりましたか。
Q2. 今回のようなレポートは何がよかったですか。何が大変でしたか。
Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

今回はグループ作業が前提ですが、レポートは各自で作成してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。

＃15　ソフトウェア開発とテストケース

今回の目標は次の通りです。

- 高水準言語によるプログラムの記述について理解する — プログラムとはどのようなものか知っておくことはコンピュータの理解に有用です
- ソフトウェア開発とその難しさについて理解する — ソフトウェア開発は仕事としてのほかに研究活動として行うこともあるのでその難しさを知っておくことは有用です
- テストの考え方とテストケースについて理解する — ソフトウェアをその「仕様」に基づいて判断するという考え方は実際にプログラムを書く時に大変有用です。

15.1　プログラミングと手順

15.1.1　高水準言語と低水準言語 ex

以前の回で、CPU がメモリに格納された命令を順に取り出し実行して行く装置であることを説明し、「小さなコンピュータ」の命令を使ってプログラムを組み立ててみました。CPU の命令は機種ごとに違っているので、機械語やアセンブリ言語のプログラムは別機種でそのまま動かすことができません。このようなプログラミング言語のことを**低水準言語** (low level language) と呼びます。低水準言語のプログラムは、繁雑で書くのが大変という欠点も持ちます。

そこで今日では、特定 CPU の命令に依存せず、人間の考え方に近い記法でプログラムを書きます。これを**高水準言語** (high level language) と呼びます。皆様は C++、Java、JavaScript、Ruby などの名前に見覚えがあると思いますが、これらは高水準言語の名前です。

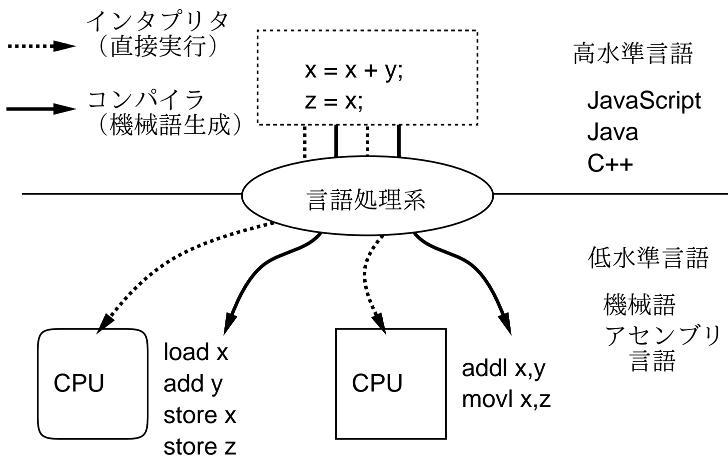


図 15.1: 高水準言語と低水準言語

高水準言語で記述したプログラムは、**言語処理系**と呼ばれるソフトウェアによって実行可能になります (図 15.1)。言語処理系の種別として、高水準言語をアセンブリ言語や機械語に変換する**コンパイラ** (compiler)、変換する代わりに高水準言語に記述された動作を直接実行する**インタプリタ** (interpreter) があります。

15.1.2　JavaScript 言語 ex

以下では皆様に、**JavaScript** という言語でプログラムを作る経験をして頂きます。この言語は処理系が Web ブラウザに標準的に内蔵されており、Web システム開発で広く使われています。

今回はこの特徴を利用して、皆様に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 15.2)。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。

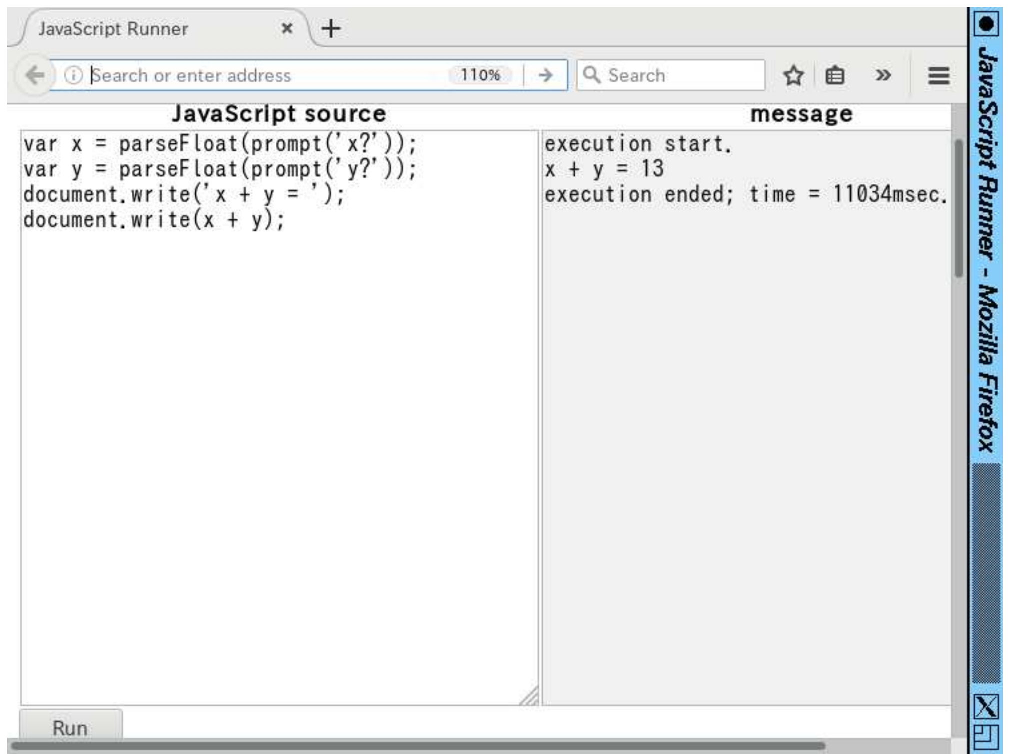


図 15.2: JavaScript 実行用のページ

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△△の部分で計算した値を入れる、という意味です。「=」は等しいという意味ではなく「値を代入する」ことを表します (機械語の store 命令)。これ以外は全て「関数名 (…)」という形で、いずれも、JavaScript 環境に予め用意された機能です。今回使うものの一覧を表 15.1 に示します。

表 15.1: 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

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、足し算などが普通の計算の式らしく書けるのが、高水準言語のよい所です。

15.1.3 JavaScript による枝分かれの記述 ex

次に、これも以前やった例のやき直してすが、「2 つの数のうち大きい方を表示する」例題を示します。以前は「条件によって枝分かれ」するのに条件分岐命令を使いましたが、高水準言語では「枝分かれ全体を表す構文」(if 文) 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 なら 1 番目、NO なら 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 (見にくいと間違えますから、必ず守りましょう)。

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。この形を使った、プログラムの別バージョンを示します (枝が短ければこのように 1 行に書いて構いません)。

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

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、さまざまな書き方が可能です。では、どの書き方がいいでしょう? それは、プログラムを書く人のセンスで、「この方が分かりやすい」「将来直すときに直しやすい」などの見通しに基づいて選びます。皆様なら、上の 2 つのどちらを選びますか?

JavaScript には繰り返しを記述する構文などもありますが、本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の説明までにします。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 行に書くために適宜「(…)」で囲むことなどが必要です。数値演算、比較演算、論理演算については表 15.3 を参照してください。上に書かれたものほど結び付きの強い演算となります。

演習 1 JavaScript の例題を動かしてみなさい。動いたら、次の小問から 1 つ以上やってみなさい。

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

表 15.2: 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)

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。2 つの違う書き方のものを作成できるとなおよい。

c. 3 つの値を読み込み、その最大を表示するが、ただし 2 つ以上の値が等しい場合は代わりに「error」と表示するプログラム。2 つの違う書き方のものを作成できるとなおよい。

表 15.3: JavaScript の主要な演算子

<table>
<tr><th></th><th>演算子</th><th>意味</th></tr>
<tr><td>数値の演算</td><td>*、/、%
+、-</td><td>乗算、除算、剰余
加算、減算</td></tr>
<tr><td>比較演算</td><td>>、>=、<、<=
==、!=</td><td>より大、以上、より小、以下
等しい、等しくない</td></tr>
<tr><td>論理演算</td><td>&&
||</td><td>～かつ～
～または～</td></tr>
</table>

15.2　ソフトウェア開発

15.2.1　ソフトウェア開発とは ex

ここまでは「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言えば「ソフトウェア」です。**ソフトウェア** (software) とは、コンピュータ上で何らかの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、ハードウェア以外のすべての部分を指します。そしてハードまで含めた動くもの全体が**システム** (system) 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)、多様な文書のテンプレート等も必要なわけです。

ソフトウェアを作る際は、やみくもにプログラムを書くという方法ではうまく行きません。ここまでの体験でも分かるように、プログラムは非常に緻密であり、どこか少しでも間違っていたら、それ

だけで全体として動作しないこともたびたびです。数行のプログラムでも正しく作るのは大変なのに、何万行ものプログラムを作って動かすのが簡単なわけはありません。

これには、多くの理由があります。まず、製品となるソフトウェアを作る時には、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、等の付帯作業が多数必要です。そのため、単にプログラムを書くだけなら 1 日に数十行・数百行が書ける人でも、ソフトウェア開発プロジェクトとして書くと「開発者 1 日あたり数行」になることも多いのです。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 人以上必要になります。人が多くなると、それらの人の間で意思疎通する手間 (ミーティング・会議・連絡の手間) が大きくなり、その分だけコードを書く時間が削られます。これも「開発者 1 日あたり数行」の原因の 1 つです。

このように、ソフトウェア開発で「人が多くなる」ことは多くのマイナスをもたらします。しかし、ソフトウェア開発に詳しくない人には、それが分かりません。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。

しかしそれは勘違いで、100 人増員したら、その人たちがいかに優秀でも、(1) これまでのメンバーが新しい人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)、(2) 人数増加による意志疎通の手間の増大により、ソフトウェアの完成時期はかえって伸びます。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」と呼ばれます。

15.2.2 要求仕様の問題 ex

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。

15.2.3 テストとテストケース ex

作成したコードには大抵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です。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が必ず必要です。これには**レビュー** (review)、**インスペクション** (inspection)、**ウォークスルー** (walkthrough) など複数の呼び方があります。ですが、普通まずやるのは、実際にコードを走らせて正しく動作するか調べることですよね? これを**テスト** (test) と呼びます。そして、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、自分で書いたプログラムの結果を見たら、それで合っていると勘違いしがちなので)。

では、どのようにテストしますか? 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。[1]
- **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) と呼ばれるものがある。[2]
- **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。

---

[1] 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。

[2] 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。

- **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 15.3)、これを用いてテストを行ってみましょう。

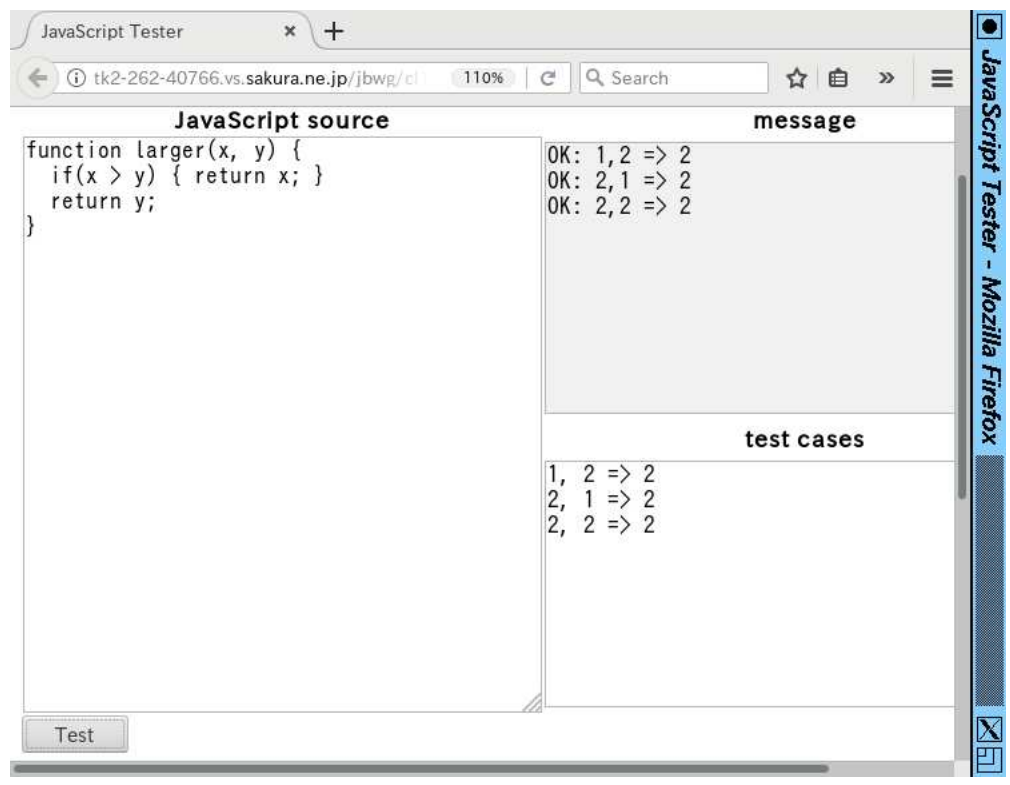


図 15.3: JavaScript のテスト機能つきページ

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します。ここでいう関数とは数学の関数とは違い、単に「まとまったコードに名前をつけて呼び出せる」もので、次の形を取ります。

```
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、その機能を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…」の部分はこれまで同様に処理を書きますが、最後に結果を「`return 式;`」という文によって返します。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すね (理由を考えてみること)。そこで、テストケースとして次の 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 2 `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 問から 1 つ以上やってみてください。必ず「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。関数のコードの正しいものを完成させることは望ましいですが必須ではありません (テストケースを体験することが目的)。

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。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。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。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];
  }
```

15.2.4 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 15.4)。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。

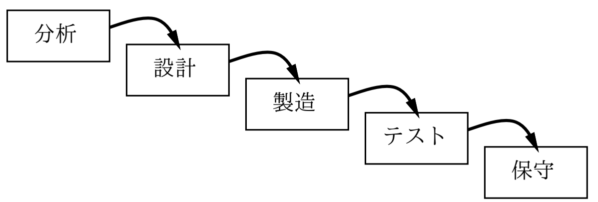


図 15.4: ウォーターフォール型プロセス

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実情です。

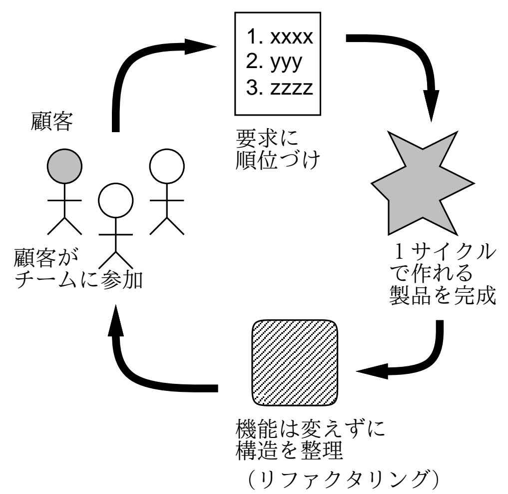


図 15.5: アジャイル型プロセス

そこで現在では、これとは別の**アジャイル** (agile) 型のプロセスが普及してきました。アジャイル型では、要求仕様や設計に時間を掛ける代わりに、顧客にまず要求を聞き、一定期間 (数週間程度) で作れる箇所を選んでそこをまず開発・稼働し、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、のように多数のサイクルを繰り返して開発を進めます (図 15.5)。これなら、後からの要求追加も容易ですし、動いているリリースを見れば顧客も何が足りないか確認できます。[3]

[3]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

15.3 Web ページ上の JavaScript プログラミング

15.3.1 JavaScript によるページ要素へのアクセス

最後のお楽しみな話題として、HTML により作成する Web ページの中に JavaScript プログラムを含めることで様々な動作ができることを見てゆきましょう。いきなり最初の例題から見てゆきます。

```
<!DOCTYPE html>
<html>
<head>
<title>Test</title>
<meta charset="utf-8">
<script type="text/javascript">
var msgs = ["Very Lucky", "Lucky", "Usual", "Unlucky"];
function show() {
  var i0 = document.getElementById("i0");
  var num = Math.floor(Math.random() * msgs.length);
  i0.value = msgs[num];
}
</script>
</head>
<body>
<div>
<input type="text" id="i0">
<button id="b0" onclick="show()">Push Me</button>
</div>
</body>
</html>
```

新たに出て来た HTML 要素が 3 つあります。

- `<script type="text/javascript">...</script>` — JavaScript プログラムを HTML 中に含めるための要素。
- `<input type="text" id="固有名"` [`size="文字数"`]`>` — 入力欄を表す要素。入力欄の幅を文字数で指定してもよい。
- `<button onclick="コード">...</button>` — 押しボタンを表す要素。その中身はボタンのラベルとなり、ボタンが押されたときは「コード」部分 (JavaScript プログラムの断片) が実行される。

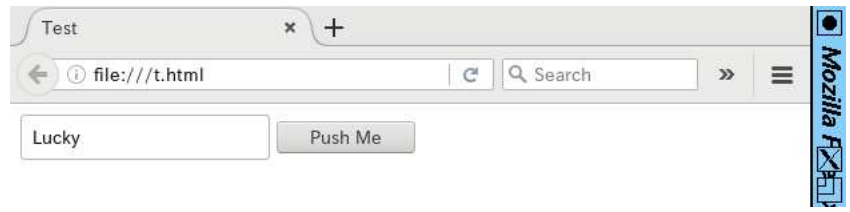


図 15.6: 入力欄と押しボタンがあるページ

まる危険があります。

そういうわけで、HTML 部分について見ると図 15.6 のように入力欄と押しボタンがあるページができるわけです。では押したときに何が起きるでしょう。「コード」の内容は、関数 `show` を呼び出す、となっていて、その関数は script 要素の中にあります。script 要素を見てみましょう。

まず関数の外に、`msgs` という名前で配列 (データの並び) を用意しています。並びの長さは 4 で、それぞれの値は見ての通り文字列です。この配列はあとで `show` の中から参照します。

関数 `show` ですが、パラメタは 0 個です。そして実行開始するとまず 1 行目で、変数 `i0` に「HTML 要素中の固有名が `i0` である要素を持って来て格納」します。`documentl.getElementById(...)` は固有名を指定して要素を取得する機能です。このように、HTML 中の JavaScript は HTML のさまざまなものを取り出して来て扱えるのです。

2 行目ですが、「`Math.random()`」は「0 以上 1 未満の数値をランダムに (さいころを振って) 返す」機能です。「`msgs.length`」は配列 `msgs` の要素数 (ここでは 4) を返します。「`*`」は乗算なので、結果として「0 以上 4 未満のランダム数」ができます。「`Math.floor(...)`」は整数への切捨てなので「0、1、2、3 のいずれかがランダムに生成」され、それが変数 `num` に入ります。

そして 3 行目ですが、配列 `msgs` の `num` 番目 (先頭が 0 番目) を取り出し、それを…`i0` に保持している入力欄の「値」プロパティとして格納します。そうすると、図 15.6 のように、入力欄にその文字列が表示されます。つまりこれは、ボタンを 1 回押すと運勢 (?) が表示される「おみくじ」プログラムなのでした。

演習 3 おみくじプログラムをそのまま動かせ。動いたら次のように手直ししてみよ。

a. メッセージの数を増やしたり、変更して試せ。

b. おみくじを何回でも押せたらありがた味がない。そこで、ボタンを押した回数を数える変数の定義 (最初は 0) を `msgs` の次の行に追加し、`show` が動くたびに 1 増やすようにしてみよ。数が増えていることの確認には `i0.value` にメッセージのかわりにその変数の値を入れてみればよい。

c. 上記ができたら、「回数が 0 より大きかったらおみくじは引かない」ようにしてみよ。

d. まったく違う問題として、「2 つの入力欄から値を読み込み、大きい方の値を 3 番目の入力欄に出力する」ページを作ってみよ。入力欄を 3 つにする必要がある。入力欄の文字列を取り出すのも `i0.value` 等を参照すればできる。既に学んだ `parseFloat(...)` で数値に変換する必要あり。

15.3.2 JavaScript コードを自動的に動かす

前節では入力欄やボタンなどの部品を使って動作をつけていましたが、実は JavaScript で HTML 中の任意の要素を (id 属性で固有名をつけておくことで) 参照して制御できます。また、クリックしたときだけでなく、「勝手に」動かすようなこともできます。そのようなサンプルを見てみましょう。

```
<!DOCTYPE html>
<html>
<head>
<title>Test</title>
<meta charset="utf-8">
<script type="text/javascript">
var count;
function start() { count = 20; step(); }
function step() {
  if(--count > 0) {
    var num = Math.random()*40-20;
    var hd0 = document.getElementById("hd0");
```

```
    hd0.style.position = "relative";
    hd0.style.top = num + "px";
    setTimeout(step, 100);
  }
}
</script>
</head>
<body>
<h1 id="hd0" onclick="start()">Click Me</h1>
</body>
</html>
```

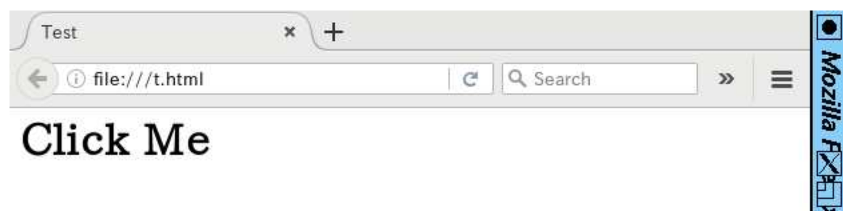


図 15.7: ゆれ動く見出しのページ

HTML はさらに簡単になり、h1 要素が 1 つだけです (図 15.7)。それを参照するため、id で固有名を指定しています。また、`onclick` 属性でこの要素がクリックされたとき関数 `start` が呼ばれるようにします。

script 要素を見ましょう。こんどは変数 `count` を用意しています。そして、関数 `start` は `count` を 20 に設定してから `step` を呼びます。あとの動作はすべて `step` の中です。

`step` ではまず、`count` の値を 1 減らし、続いてそれが 0 より大きいか調べ、大きい時だけ動作本体を実行します。本体の 1 行目では、0〜1 の乱数を取り、40 倍して 20 を引くので、-20〜20 の乱数ができます。次に、h1 要素を固有名を指定して取得します。そして、h1 要素の位置指定の方法を「相対位置」に変更します。

そのあと、h1 要素の縦位置を先に用意した乱数に設定します。実はこの「.style. なんとか」というのはすべて、CSS のプロパティを設定する機能を持っています。CSS プロパティでは位置指定は必ず単位をつけるので、乱数値のあとに「`"px"`」(ピクセル) をくっつけてそれを設定しているのです。

そして最後に、「100 ミリ秒後に再度 `step` を呼び出してくれ」とシステムに頼みます。このため、100 ミリ秒間隔で繰り返し `step` が動き、乱数のため見出しの縦位置がゆらゆらゆれ動きますが、20 回実行すると `count` が 0 になってそれ以上動作しなくなります。再度クリックすればまた動きます。面白いでしょう?

演習 4 揺れ動く見出しの例をそのまま打ち込んで動かせ。動いたら、次のように修正してみよ。

a. 動きを速くしたり (ヒント: 間隔を 100 ミリ秒より短くする)、大きくしたり、横方向に動くようにしたり (ヒント: 横位置のプロパティは「`style.left`」)、縦横両方に動くようにしてみよ。

b. body 要素に id をつけて取得し、`body` 要素の `style.backgroundColor` プロパティを変更することで、「クリックされるとランダムに、または徐々に色が変化」するページを作れ。(ヒント: 色は「"rgb(赤, 緑, 青)"」という文字列で指定できるので、赤/緑/青のどれかを計算してあとは固定した値にして「+」で文字列連結する。赤/緑/青は 0〜255 の整数である必要がある。`count` を整数にして 1 ずつ減らして行くとかならもともと整数だが、乱数を使う場合は実数を整数にするため `Math.floor(...)` を使う。)

c. HTML 要素の「`innerHTML`」プロパティに書き込むことで、その要素の内側に任意の HTML を書き込むことができる (ただの文字列であれば中身がその文字列の内容になる)。これを使って、クリックすると勝手に一部の内容が変化していく…たとえば 100 から 1 ずつカウントダウンしていくような…ページを作ってみよ。
d. JavaScript を使って任意の好きな変化をおこなうページを作れ。

課題 15A

今回の課題は「演習 1〜4」に含まれる小問 (合計で 14 個) の中から 1 つ以上を選択し、結果をレポートとして報告して頂くことです。LMS の「assignment # 15」の入力欄に入力してください。以下の内容がこの順に含まれるようにしてください。

- 冒頭に題名「コンピュータリテラシレポート# 15」、学籍番号、氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も別途書く。)
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。必ず自分が書いたコードやテストケースを掲載すること。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 以下のアンケートに対する回答。

Q1. JavaScript のような高水準言語と前にやった機械語 (アセンブリ言語) を比較してどのように感じますか。
Q2. テストケースを書くなどのソフトウェア開発の考え方についてどのように思いましたか。
Q3. ここまでの科目全体を通して、学べたこと、学びたかったけど学べなかったことは何ですか。その他感想や、この科目の今後改善した方がよいこと、今後も維持したことがよいことの指摘もどうぞ。

なお、課題はグループでやって構いません。その場合も、(メンバー氏名を明記した上で) レポートは必ず各自で執筆してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。

索 引

10 進法, 40
16 進, 41
24 ビットカラー, 112
2 次記憶, 51
2 次記憶装置, 73
2 値画像, 113
2 の補数, 42
2 進法, 40
8 ビットカナ, 66
8 進法, 41
AD 変換, 111
ASCII コード, 65
BMP, 112
CA, 25
cat, 55
center 環境, 107
chmod, 59
CMS, 129
cp, 56
CPU, 44, 73
CSS, 132
CSS グリッド, 146
CUI, 11
date, 12
description 環境, 106
display, 55
div 要素, 136
DNS, 21
echo, 56
egrep, 90
enumerate 環境, 106
eog, 55
EPS, 118
euc-jp, 66
exit, 12
fgrep, 90
file, 55
GIF, 113, 142
gimp, 57
grep, 90
GUI, 11
HTML, 29, 129
HTTP, 29
id, 136
IMAP, 31
img 要素, 143
IP, 17
IPv4, 16
IP アドレス, 17
ISO 10646, 66
iso-2022-jp, 66
ISP, 18
itemize 環境, 106
JavaScript, 158
JIS C6226, 66
JIS X0208, 66
JIS 漢字コード, 66
JPEG, 113, 142
kill, 80
LAN, 14
LaTeX, 104
less, 55
link 要素, 133
ln, 57
ls, 54
man, 12
MIME, 34
mkdir, 57
more, 55
MTA, 30
MUA, 30
mv, 57
nkf, 67
nslookup, 22
od, 65
PBM, 112
PDF, 115
ping, 13
PKI, 25
PNG, 113, 142
POP, 31
PostScript, 115

ps, 77
pwd, 53
RFC822, 32
RGB 値, 112
rm, 57
rmdir, 57
sed, 91
shift-jis, 66
sleep, 12
SMTP, 31
SNS, 36
sort, 87
span 要素, 136
SSD, 51
ssh, 27
SSL, 25
style 属性, 133
style 要素, 132
SVG, 115, 142
table 要素, 137
tabular 環境, 107
target 属性, 143
TCP, 17
TCP/IP, 16
td 要素, 137
th 要素, 137
TLD, 21
TLS, 25
tr, 86
traceroute, 20
tr 要素, 137
UDP, 17
UNICODE, 66
uniq, 88
Unix, 11
URL, 29
UTF8, 66
verbatim 環境, 106
VLSI, 48
WAN, 14
wc, 89
Web アプリケーション, 30
Web ブラウザ, 28
WWW, 28
WYSIWYG, 101
アウトラインフォント, 118
アジャイル, 165
アセンブラ, 45
アセンブリ言語, 45
アナログ, 39
アプリケーションソフト, 74
暗号, 24
暗号化, 24
意見, 126
意味づけ方式, 101
イクリメンタルサーチ, 70
インスペクション, 162
インタフェース, 73
インタプリタ, 157
引用, 125
ウィルス, 24
ウォークスルー, 162
ウォーターフォール, 165
運用, 154
遠隔ログイン, 27
エンコーディング, 64
炎上, 35
エンベロープ From, 33
オーバフロー, 43
オペレーティングシステム, 11
オペレーティングシステム, 74
回帰テスト, 163
回線交換, 14
階層構造, 15, 19, 52, 144
鍵, 24
確認, 16
仮想化, 77
可用性, 23
カレントディレクトリ, 53
環境, 105
関数, 163
完全性, 23
画像, 111
キーボード, 5
キーボードショートカット, 63
キーボードマクロ, 70
キーワード検索, 123
機械語, 45
企画立案, 151
機能テスト, 162
基本ソフト, 74
機密性, 23
脚注, 107
キャプション, 119

共通鍵暗号, 24
許可, 3
許諾, 125
キルリング, 69
区点番号, 66
クライアントサーバ方式, 27
クラス名, 136
クローラ, 123
経路制御, 19
経路表, 20
検索エンジン, 123
言語処理系, 74, 157
コード, 44
コア, 73
公開鍵暗号, 24
高水準言語, 157
構造テスト, 162
コマンド, 11
コマンドインタプリタ, 78
コミュニケーション, 34
コミュニケーションサービス, 35
コミュニティ, 35
コンパイラ, 157
コンピュータ, 2
コンプリーション, 55, 67
再送, 16
サブネットマスク, 18
サンプリング, 111
シェルスクリプト, 95
シェル, 78
資源, 76
指針, 153
システム, 160
システムソフト, 74
主記憶, 44
出力装置, 73
シリコン, 48
真空管, 48
磁気ディスク装置, 51
事実, 126
情報, 2
情報アーキテクチャ, 143
ジョブ, 82
ジョブコントロール, 82
スイッチ, 18
数式, 108
スキャナ, 111
スタイルガイド, 153
スモークテスト, 162
正規表現, 90
セキュリティ, 23
セレクタ, 134
線形構造, 144
先行研究, 127
脆弱性, 3
絶対 URL, 141
絶対パス名, 53
喪失, 16
相対 URL, 141
相対パス名, 53
ソフトウェア, 160
ソフトウェア開発プロセス, 165
属性, 130, 131
ターゲット, 152
対称鍵暗号, 24
タイマー, 76
タッチタイピング, 5
端末, 11
チェックリスト, 153
中断, 82
著作権, 124
低水準言語, 157
テキスト, 99
テキストエディタ, 63
テキストファイル, 63, 64
テスト, 162
テストケース, 162
ディスプレイ, 111
ディレクトリ, 52
デジタル, 39
デジタルカメラ, 111
電子署名, 25
電子図書館, 124
電子メール, 30
匿名, 35
トランジスタ, 48
トランスポート層, 17
トロイの木馬, 24
ドメイン名, 21
ナビゲーション, 144
ナビゲーションリンク, 144
日本工業規格, 66
入出力装置, 73
入力装置, 73

認証, 2
ネットマスク, 18
ネットワーク, 14
ネットワーク構造, 143
ネットワークサービス, 26
ネットワーク層, 17
ネットワーク部, 17
ノイマン型, 44
ハードウェア, 73
配色, 152
ハイパーテキスト, 28
ハブ, 18
バイト, 13
バイナリファイル, 63
バックグラウンド, 82
バッファ, 63, 71
番地, 44
パイプライン, 81
パケット, 14
パケット交換, 14
パスコード, 3
パス名, 52
パスワード, 3
パラフレーズ, 126
パンくずリスト, 144
被覆テスト, 162
標準エラー出力, 80
標準出力, 80
標準入力, 80
剽窃, 126
ビット, 39
ビット列, 39
ピアツーピア方式, 27
ピクセル, 111
ピクセルグラフィクス, 112
ファイル, 52
ファイルシステム, 52
ファイル名展開, 95
フィルタ, 86
フォアグラウンド, 82
フォトレタッチソフト, 112
付加情報, 99
復号, 24
符号化, 34, 64
符号なし表現, 42
不正アクセス, 24
フラッシュメモリ, 51
フレーム, 71
フロート, 119
ブロックレイアウト, 145
文献リスト, 127
文書, 99
文書整形系, 99
プリンタ, 111
プログラム, 44
プログラムカウンタ, 45
プログラム内蔵, 44
プロセス, 77
プロセス ID, 77
プロセス管理, 77
プロトコル, 15
プロパティ, 134
変数, 93
変数展開, 93
ベクターグラフィクス, 114
ペイントソフト, 112
ホームディレクトリ, 53
ホームポジション, 7
ホウ素, 48
保守, 154
ホスト, 17
ホスト部, 17
傍受, 28
マーク, 69
マークアップ, 101
マザーボード, 73
マルウェア, 24
マルチコア, 73
マルチタスク, 75
マルチプロセサ, 77
ミドルウェア, 74
ミニバッファ, 64
ムーアの法則, 48
メーリングリスト, 34
メールリーダ, 30
命令, 44
メタ文字, 95
メッセージヘッダ, 33
メモリ, 44, 73
メンテナンス, 154
モード, 59
モードライン, 64
文字エントリ, 130
文字コード, 64

文字コード体系, 65
文字参照, 130
モノクロ画像, 111
ユーザ ID, 3
ユティリティ, 74, 85
要求仕様, 161
要素, 130
ランダムテスト, 162
ランレングス符号化, 139
リージョン, 69
リダイレクション, 80
量子化, 111
リン, 48
リンク, 28
ルータ, 18
ルートサーバ, 21
ルートディレクトリ, 52
レイヤ, 112
レジスタ, 44
レジストラ, 21
レジストリ, 21
レビュー, 153, 162
ログアウト, 2
ログイン, 2
ワークフロー, 151
ワープロソフト, 99
ワーム, 24
忘れられない, 35

コンピュータリテラシ