平成 15 年 11 月 20 日
京都工芸繊維大学コンピュータ部

Lime

はじめに

本年度のコンピュータ部の部長の田村というものです。今年も無事 Lime の発行することができました。今回は大先輩である AXE 社社長の竹岡さんに原稿を寄稿していただいたこともあり、内容も充実した色濃いものとなったと思います。僕たちの代になり、部内の環境がよくなり喜ぶ反面、年々部員全体の実力低下に悩むところもあり、部長として不安に思うこともあります。士気や部内の雰囲気によるものもありますれば、一概に実力低下と騒ぐのもよくないかもしれませんが、先輩方からかつての黄金期の話を聞くとやはりがんばりが足りない部分も否めません。今年はそういったいきさつもあり、松ヶ崎祭の参加には本来の部活動に沿った研究発表・展示に力を注ごうという考えのもと、展示に力をいれました。僕も稚拙ではありますが、現在製作中の語彙連続認識エンジン Julius を利用して製作する言語モデルを展示に間に合わせ、研究展示するつもりでいます。毎年ささやかれる「もっと情熱を！」とマシンに対するピュアな愛情を叫べるようになりたいと思うしだいです。最後になりますが、毎年編集作業に携わってくれている山本先輩をはじめ、Lime 執筆に携わった部員、スタッフに感謝したいと思います。稚拙な文章ですが、これをもってはじまりの挨拶とさせていただきます。

平成 15 年 10 月 20 日
京都工芸繊維大学コンピュータ部部長 田村 航

目 次

第I部

寄稿記事

コンピュータ部の OB で、業界の最前線で活躍しておられる方の寄稿記事です。

1 速い計算機は素敵なのだわわわ、誰がなんと言っても

コンピュータ部 OB たけおか ナヲミ

つれづれなるままにその日暮らし、男もすなるという随筆という物を綴ってみようと、潮も満ちてさあ漕ぎい出んと思うほどに、盛者必衰のことわりこそいとおかしけれ。

というわけで、火星大接近とほぼ同じ周期でしか優勝せず、自分が生きているうちに次の優勝があるのかどうか、心配しているタイガースファンの皆さんいかがお過ごしですか?

火星よりも地球に興味を持つ人達が作った地球シミュレータは今日も世界で一番速いコンピュータなのですわ。

Macintosh G5 を並べると、世界で 3 番目に速いスーパーコンピュータが作れるらしいわ。
PowerPC G5 おそるべしやわ。
あるちべっく様々やわぁ。←ありし日の吉川ひなの風に
スパコン Top500 で 1 位の座から不動の地球シミュレータを働かせる事で横浜近辺がヒートアイランドになり、地球シミュレータのシミュレーション結果と横浜近辺の実際の気候が食い違う、という噂もちらほらだわ。

ああ、ベクターマシンには夢があるわ。(あったわ? (号泣))
浮動小数点は、小数点が「浮動」するらしいのだけれど、字面の上では、常に、整数一桁の次にあり、「不動」だとも言えるわ。

DEC Alpha は無くなるし、もう、Power アーキテクチャぐらいしか、心のよりどころがないわ。
Opteronって、結局 x86 命令を実行できたりするし。
IA64って、何?
でも、組み込み用 CPU でも、演算パワーのある機械がいくつも出て来そうで頼もしいわぁ。

例えば、これ

「超並列シミュレーションサーバ「BioServer」の実証実験を開始」
http://pr.fujitsu.com/jp/news/2003/11/5.html
http://www.zdnet.co.jp/enterprise/0311/05/epn08.html

BioServerって、戦隊ものの基地にある機械じゃないわよ。
OS には、axLinux を採用だって、素敵ぃー

あたし達、某キャリア様と一緒に Linux 入り携帯電話を作ってるの。
去年の年末には、そのキャリア様から頂いたカレンダーが、女子社員の間で取り合いになったわ。
今年はベッカム様のカレンダー、もっと下さい > 某ダフォン様

2G な松下先輩が置いてけぼりになる、仲間由紀恵な 3G Phone な会社も Linux 電話器を作りたいそうよ。
どうでもいいけど、あたしも持ってる Cdma Oneって、3G じゃなかったの???
Cdma One はアメリカで使えてとても便利。GSM よりよっぽどつながる。ハワイもグァムもつながるわ。便利だわ。

ボーダフォン様の電話器はヨーロッパで使えるらしい。でも、ヨーロッパに行く機会はなかなかない。ああ、ギリシャとトルコに行ってみたい。(連れていってくれよだわ > K 本)
KDDI も Vodafone もワールドワイドに使えるのは、極一部の機種だけよ。そんなの持ってるのは、割と物好き。おかげで、KDDI のは写真の撮れないケータイだったりする訳なのよ。(最近、写真付きケータイで国際対応が出たけど)

瓜田に冠を直すがごとく、明白なのは、家電には WindowsCE は入らないという事。日本の家電な方々は、かなりケチなので、「Wintel にええようにされてタマルかい」てなものなのよ。
家電には Linux で決りでんなぁ、だわよ。

WinCE を担いでいた人々も、PocketPC という裏切りのせいで、行き場所がなくなり、もう、死ぬしかない状態。

- PocketPC は、CPU は XScale だけ
  これは、CPU を Intel からだけ、言い値で買えという事
- 台湾メーカにも日本語版 PocketPC を卸す
  これは、セットメーカに、台湾と価格競争をしろということ

こんな状態では、日本メーカは、なんのために PocketPC を作るのか、訳判らないわ。なので、WinCE を担いできた人々は、もう、死ぬ程に元気が無くなっている。

WinCE は車に生き残りを賭けているわ。
GBook…
しかし、BSD はいいわよねぇ。
SCO に訴えられる事も未来永劫ないし (たぶん)

ああ、そうだわ、悪魔崇拝よ。
不道徳よ。
信号無視をしたり、部室でお菓子を食べたり、左側通行をしたり、廊下を走ったりするのよ。
ああ、素晴らしく甘美な行為…

これでメイド衣裳を来たり、バニーになったり、アラビアンナイトな衣裳を着れれば最高だわよ。悪魔様万歳!!!

ちなみに、ベクトルマシンというのは、複数の演算を一遍にやる機械の事じゃないのよ、そこんとこ、よ・ろ・し・く・ね、坊や達。

以上、オチ無し

第II部

ロボット

近年ロボット技術の発展はめまぐるしいものがあります。ここではそれに関連した基礎技術などについて話していくことにします。

1 ロボットアームプロジェクト

00230111 電子情報工学科 4 回生 山本 大介

1.1 はじめに

昨年の 9 月ごろ、別の部員が立ち上げたロボットアームプロジェクトをいつのまにか先導することになってしまいました。昨年の松ヶ崎祭では、作品をいろんな方に楽しんでもらい大盛況に終ったわけですが、今回の `Lime` にはそのロボットアームプロジェクトの概要を書きたいと思います。今回の制御対象になるロボットアームは図 1.1 のようなもので、市販されているイーケイジャパンの MR-999 を改造しました。

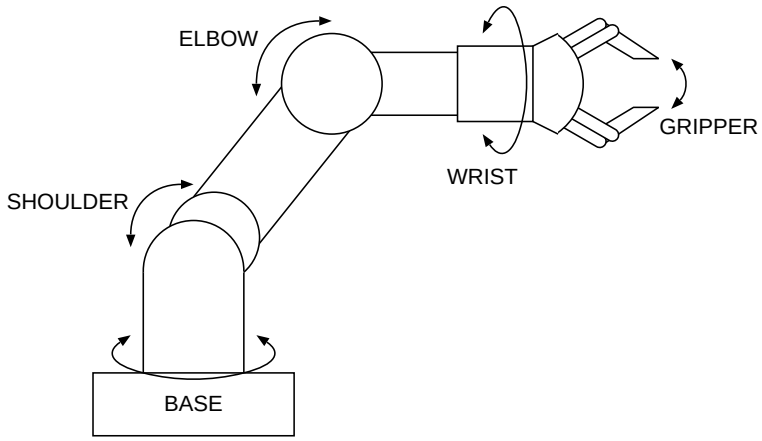

図 1.1: ロボットアーム

1.2 ディジタル回路設計

ロボットアームは市販のロボットアーム組み立てキットを使用することにしました。本プロジェクトはこれに電子回路をとりつけて改造することが主題となっています。

そして、以下のように計画を立てることとしました。ロボットアーム組み立てキットには最初からボタン操作でロボットアームを動かすことのできる簡単な操作板がありましたが、これはモータ

に電流を流したり流さなかったりするだけのスイッチであったので、そこで、これをパソコンからできるようにすることを考えました。手順としてはこのスイッチをとりはずし、新たに制御用の電子回路をとりつけ、パソコンと接続するというものです。

PC/AT 互換パソコンの出力装置としては、主にシリアルポート、パラレルポート等が考えられましたが、回路を簡単なものとするためにパラレルポートを出力装置とすることにしました。

しかし、パラレルポートの出力端子は 8 つしかありません。モータの正転・逆転・停止を 2 端子で制御するとすれば、2 つ足らなくなってしまいます。そこで、図 1.2 のように D-FF を使用して、時間的に変化させることにより、出力の種類を増やすことにしました。これは、最良の方法とは言えませんが、D-FF チップひとつで済ませることができるので安価です。状態遷移図は図 1.4[1]のようになっています。

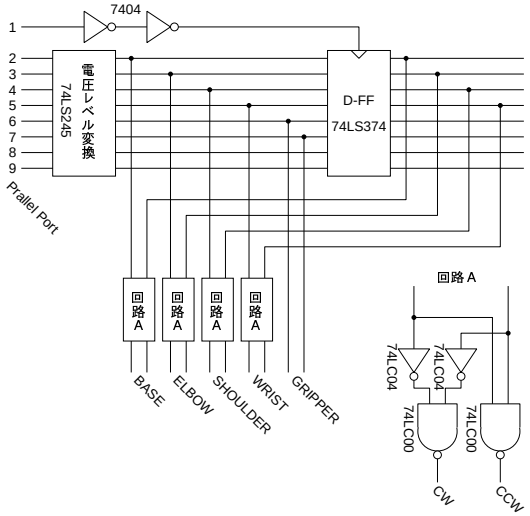

図 1.2: 論理回路

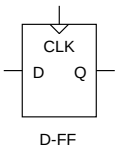

図 1.3: D-FlipFlop

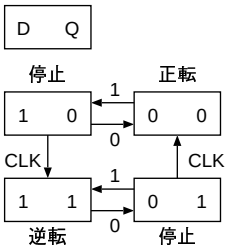

図 1.4: 状態遷移図

1.3 ドライブ回路設計

汎用ディジタル IC の出力ではモータを動かすような大きな電流は流すことができません。そこで、ディジタル IC の電圧制御信号をもとに、電流を増幅させてモータを回すドライブ回路が必要になります。このロボットアームは直流モータで関節を制御するので正負両方向[2]へ電流を流せるような回路にしなければなりません。それを実現するには片方を GND に片方を $+V_{CC}$ と $-V_{EE}$ の切替えにする方法があります。しかし、これは $-V_{EE}$ を用意するために印加電圧を半分に分ける必

[1] ここの CLK はモータ全部への指令となってしまうため、あるモータが正転から逆転へ移行しようとした場合一度すべてのモータを止める必要がある。

[2] 本記事では正方向への回転を正転 (CW)、負方向への回転を逆転 (CCW) と書くことにします。

要があるため、モータにかける電圧が低下してしまいます。そこで、こういう場合によく用いられるのが図 1.5 の H-bridge 回路と呼ばれているものです。

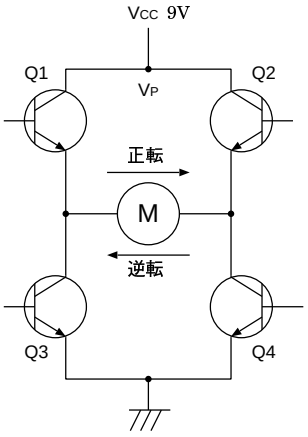

図 1.5: H-bridge 回路

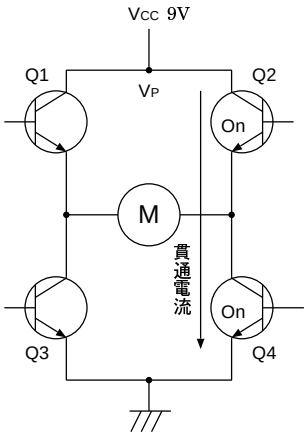

図 1.6: 貫通電流

表 1.1: モータの回転方向

モータ	Q1	Q2	Q3	Q4
停止	Off	Off	Off	Off
正転 (CW)	On	Off	Off	On
逆転 (CCW)	Off	On	On	Off
ブレーキ	Off	Off	On	On

この回路の使用にあたっては注意することがいくつかあります。そのひとつが図 1.6 に示すように片側のトランジスタが両方とも On になった場合 V_{CC} と GND がショートして回路を壊してしまうおそれがあることです。

そこで、作成した H-bridge 回路は図 1.7 のようにしました。この回路の上部を見て下さい。5Ω の抵抗が繋がっていると思います。通常、この抵抗はモータの抵抗値に比べて小さいので消費電力はわずかです。しかし、貫通電流が流れてしまった時には電源電圧のほとんどがこの抵抗にかかります。トランジスタの CE 間電圧を 1V と仮定すると、この抵抗で消費される電力は $P = V^2/R = 9.8[\mathrm{W}]$ という大きな値になってしまいます。電子回路で良く利用されるカーボン抵抗は 0.25W 程度しか耐えられないのでここにつければ一瞬で壊れてしまいます。そういう時に利用されるのがホウロウ抵抗と呼ばれるものです。これは大電力にも耐えることができ、電力消費にともなう発熱[3]にも耐

[3]実際に電圧がかかると 100℃ くらいになります。もちろん触ると火傷します。ただ、触ってみて熱かったら暴走してい

えられます。よって、この回路ではこの部分だけホウロウ抵抗を利用しています。

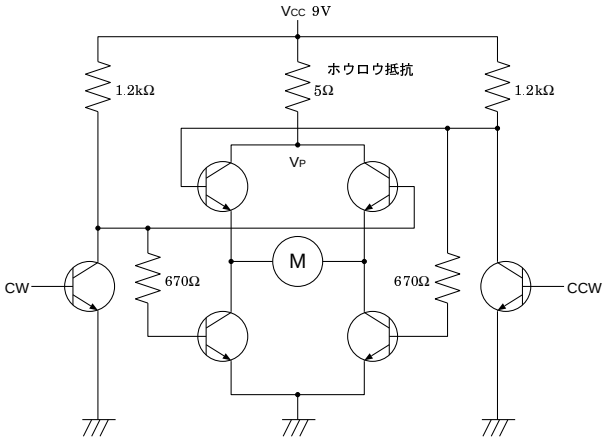

図 1.7: 作成した H-bridge 回路

図 1.8 は実体配線図です。これをモータ 5 つ分作ったので最終的にトランジスタはなんと 30 個にもなりました。

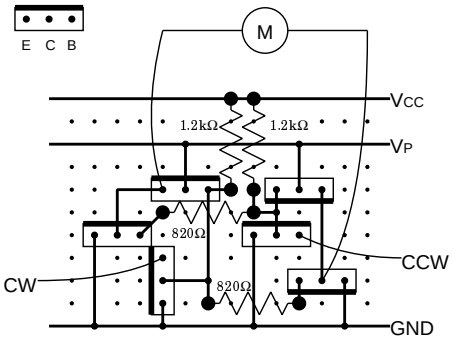

図 1.8: H-bridge 実体配線図

1.4 諸問題

1.4.1 ノイズの問題

論理回路の実装にあたっては汎用のディジタル IC をそのままつなげて実現しましたが、そのためにノイズによって暴走することがしばしばありました。特に図 1.7 の 670Ω の部分は流れる電流

るとわかるので便利なのです。

によってたまに On, Off が不安定になるようで、よく貫通電流を誘発しました。

ディジタル IC は一瞬で電圧が上がったり下がったりするため、電流の供給がおいつかなくなってノイズが発生します。この回路を製作したころは、そのノイズを解消する手段である「ディジタル IC 付近にはバイパスコンデンサ（通称：パスコン）をつける」というようなテクニック[4] を詳しく知らなかったのです。これは今後の製作にとってよい経験となりました。

1.4.2 電源の問題

ノイズを考えディジタル IC に供給する電源とドライブ回路に供給するのは別々としていました。ディジタル回路で急速な変化が発生すると一時的に電圧が低下したりしますが、それがドライブ回路に波及するのを防ぐことができたからです。

しかし、思わぬところで問題が起きました。モータはひとつにつき 500mA ほど必要としますが、5 つのモータが同時に動くと 2500mA もの電流が必要となりました。今回利用した直流電源は市販の AC アダプタで限界は 1600mA でしたので、5 つ同時に動かすと AC アダプタの内部抵抗で電圧降下が発生してロボットアームの性能が著しく低下しました。

1.4.3 制御の問題

今回製作した H-bridge 回路は図 1.7 に示す通り、CW および CCW がともに Low 電圧となると貫通電流が流れて回路が暴走します。つまり、CW、CCW を制御する電源回路の電源が入っていない状態で、この回路に電源が入るだけで暴走してしまうという致命的欠陥をかかえています。両端をプルアップしてそういう事態を避けるべきでした。

1.5 拡張計画

昨年にロボットアームが完成した後に実行された計画や、これから実行する予定の計画などを書いていきたいと思います。

1.5.1 汎用モータドライバ取り付け

昨年のロボットアームの計画では 1.8 のように手で H-bridge を作りましたが、将来的に安定した動作が必要となるため汎用の DC モータドライバ（H-bridge 回路の要素がすべて入った汎用チップ）を使用することにしました。これにより、暴走はほとんどなくなりました[5] 。

[4] モータの逆起電力による電流を逃がすためのダイオードなどノイズ除去には他にも様々なテクニックがある。
[5] でも、せっかく作った H-bridge 回路は不要となってしまいました。

1.5.2 ロータリーエンコーダ取り付け

ロボットアームを自律駆動させたりする時に重要になるのが、関節がどれだけ曲がっているのかという情報です。ステッピングモータは与えたパルスの数によって回転角がわかりますが、DC モータは回転角を知る術がありません。よって、センサなどを取り付けて回転角を調べる必要があります。そこで、ロータリーエンコーダを各関節につける計画が進んでいます。

1.5.3 その他の計画

現在は長いケーブルを通じてロボットアーム操作していますがそれを赤外線リモコンに変える計画があります。また、動作パターンをプログラムとして組み込んだマイコンによって、ロボットアームを動かす計画もあります。

1.6 さいごに

電子回路製作はプログラミングなどと違い費用がかかります。また、失敗すると最初からやりなおさなくてはならなくなったり、とりかえしのつかないことになったりします。そういう意味で、プログラミングのように、頻繁に試行錯誤を繰り返すことが難しいため、ある程度の計画が必要になります。しかし、計画通りにいった時の感動はすばらしいものです。

今回の場合、コンピュータのプログラムとディジタル処理の部分を重森が行ない H-bridge は私が作ることにし、インタフェースだけ決定して同時並行で作業を進めました。そして、文化祭当日の朝に接続して完璧に動いた時は感動ものでした。電子回路製作はとっつきにくいものですが、一度やると他では味わえない感動があります。ぜひやってみてはいかがでしょうか？

参考文献

[1] 桜庭一郎/大塚敏/熊耳忠 共著,「電子回路」, 森北出版株式会社, 1986

[2] 後閑哲也,「誰にでも手軽にできる電子工作入門」, 技術評論社, 2001

2 ロボットの目のこれまでと今後の推移

02220077 機械システム工学科 2 回生 森本 勇次

2.1 始めに

近年ヒューマノイドロボットの開発が盛んに行われています。例としては、ホンダの ASIMO や P3、富士通の HOAP-1、2 などがあり、これらはいずれも二足歩行を実現しています。

これらのようなヒューマノイドロボットの開発を目指した理由としては、ヒューマノイドロボットが社会と共存、協調しやすく、それでいて、ロボットの新たな可能性を見出すことが出来るからであり、ヒューマノイドロボットは将来人間を生活の様々な部分でサポートし、安心感を与えるパートナーとなってくれるでしょう。

ここでは、そのロボットのパーツの一つである「目」と、それから得られる情報をロボットの制御部がどう解析、処理するのか、といったことを説明していきます。

2.2 ロボットの「目」

ロボットの目は、人とコミュニケーションを取ったり、自分のおかれている環境を認識するうえで、非常に重要な役割を持っており、現在はほとんどのロボットがカメラを目として使っています。カメラを使う利点としては、画像を解析することによって、対象物の形や種類、更に特定の個人かどうかまで判断することが出来うることがあげられます。これは、ヒューマノイドロボットとしては非常に好都合といえるでしょう。これによってコミュニケーションを取る為の「互いを認識すること」ステップをほとんどの場合クリアできるからです。

一方、悪い点としては、特に暗闇の場合、普通のカメラでは対象物を照らし出す照明等がないと、画像から情報を得ることがかなり困難になります。ですが、これに対する対処法として、赤外線カメラを使うことによって明暗に関係なく画像を鮮明に得ることは出来ます。赤外線カメラを設置する頭の大きささえ考えれば不可能ではない話ではあります。

設置するカメラの数については、多ければいいというものではありません。現在２または３眼が主流ですが、片目をつぶって周囲を見回しても分かるとおり、１眼でも立体感覚はあまり失われません。これは、人間の頭の中に、基本的な立体の情報が入っていて、そこからある程度判断することができるからです。そのようなデータをロボットにもたせてやれば、１眼でも物体の形を予想ことはある程度可能なのです。情報の中で数の影響を受けるのは距離感なのです。

対象物との距離は非常に重要な情報であり、そういった意味では１眼は頼りないといえるでしょ

う。１眼で得られるのは平面だけで、画像を連続的に取って解析をしないと距離は得られません。

２眼になると、１眼より大分距離が得やすくなっています。ここで使われる解析方法は三角測量法と呼ばれているもので、ある一点について、右と左の画像の座標を調べることによって距離を求めることが出来る距離測定方式です。この方法は距離測量は正確ですが奥行についてはあまり精度がよくありません。特にカメラの真正面で、カメラと同じ高さにある物体では、奥行きを測ることが出来ません。

これを補うためにもう一つカメラをつけて３眼にすることによって、この問題はある程度改善されます。ただ、画像の解析の手間が更に一つ分増えるため、三つ目（この場合、カメラは大体において三角形に配置される）の上の部分のものは距離の測定にのみ用いて、後に説明する画像の解析を行わないようにする、等の工夫をすれば制御部への負担は多少軽減されると思います。

これ以外にも、片方にスリット光発射装置、もう片方にカメラといったもの等、様々な方式のものがあります。

2.3 画像の解析

上記のようにして得られた画像は、制御部において色々な方法で解析されます。下に示すのはその一例です。

2.3.1 表面による表現

厳密な表面形状を解析するのではなく、物体の表面をいくつかの部分平面や曲面で表現し、それらの接続関係を示すことで画像を解析する方法です。この方法が有効な例としては、平面で囲まれた物体であり、逆にあまり有効でないのは単一曲線で囲まれた物体（球など）です。これは、球面を例として説明すると、球面と単一で表現する場合には、簡潔に表現できますが、別途「球面」という概念をインプットしてやる必要があり、逆に多数の小さな平面の集合と考えると、平面を多数用意しなくてはいけませんが、「平面」と言う概念は既に入っているのでそれを用いることが出来る、といったようになります。

もう一つの問題点は、見えている面同士の接続関係がどういったようになっているかを表現する方法です。これには色々な方法があるのですが、ここでは一つの方法を例として紹介します。

その方法とは、エッジリンクと呼ばれる方法で、物体を頂点、面、エッジ（これは頂点同士を結んだ線、通常「辺」）の三つの要素にわけ、それらが互いにつながって（リンクして）いる場合、それら同士を線で結んで、それが更に他の頂点や面、エッジとつながっていて、といったように展開し、物体を一つの平面図形として見ることが出来るようにするのです。

ただ、この方法を用いた場合、制御部が見えている部分をもとに作り出したモデルのエッジリンクと、実際の物体のエッジリンクが整合するかどうかが分からないので、それを確認するための作業がないと確実性が落ちてしまうことになります。

表面による表現についていえることは、平面で囲まれた物体に対しては強いが、曲面、特に球等に弱く、これらを克服するためにはあらかじめこれらを示す概念をインプットしてやる必要があり、

その分他の方法よりあらかじめインプットすべきデータが多少増える、といったことです。

2.3.2 一般化円筒による表現

物体をある軸と、そこをある形の図形が軸に沿って移動した結果としてみる方法です。この場合、物体は軸、その上を移動する図形の形、及び大きさ、という三つの要素で物体を表現することが出来ます。

この方法の利点は、３つあります。

１つ目は、自然、人口問わず、物体には軸があることが分かっており、この方法はその軸を用いて物体を簡潔に表すことが可能です。例えば、動物で言う骨格などがそれに当たり、それを線を用いて表現することにより、その動物の動きを正確に表現することが可能です。

２つ目は、物体の移動や回転を軸を同じだけ移動させて肉付けすることによって移動後の物体の形も正確に再現できる点です。前項の表面による表現などは、物体が移動するとそのつどほぼ全表面の解析をしてやらなければならなくなり、非常に効率が悪く、制御部にも負担をかけますが、軸の移動だけを考えるとかなり負担は軽減されると思われます。

３つ目に、物体の軸に適当な肉厚をつけることで、ある物体の基本的な形を示すモデルとすることが出来、これを用いて多くの物体を区別することが出来ます。

なお、一般化円筒を得る方法については、スリット光で物体の外縁を調べ、ある程度詳細な形を求めて、そこから一般化円筒を導き出す方法など、いくつか方法があります。

一般化円筒による表現について言えることは、軸を検出するために別途装置が必要となりますが、立体への復元、移動後の物体の形状を出す方法の容易さ、前提となるデータがあまり多くなくて済む、といった点から、有効な方法です。

2.3.3 体積による表現

ある空間領域を物体が占めるかどうかでその形状を表現する方法です。例えば、ある物体を格子状の小空間に分割し、その半分以上を物体が占有していた場合に、その空間に物体があるとする方法で、物体の大まかな形を知るのに便利な方法であるといえます。これ以外にも、物体に完全に占有されている部分のみに物体があると定義したり、逆に少しでも物体が占有していれば、その小空間は物体に占有されている、といったようにすることも可能です。後者の二つは物体の非常に大雑把な形を見ることが出来る程度の精度なので、正確な解析には向きません。

もう少し効率のよい方法として、「八分木」というものがあります。

これは、空間を大きく分割しておいて、全て物体が占有している場合はそこに物体があるとして、一部を物体が占有している場合は、その部分を更に２×２×２に分割して、といったことを繰り返し、最終的には、最初の一点から８本の枝がいくつもつながっていくようなグラフで表現することが出来ます。

体積による表現の全ての方法について言えることは、精度が他の二つに比べ落ちるということです。その反面、処理は比較的簡単（小空間に物体がある or ない）なので、負担は軽くなるといえます。

2.4 それ以外の「目」

これまでロボットの普通の目について説明してきましたが、ヒューマノイドロボットという形状上、どうしても死角が出来てしまいます。体右、左半分直近や、真後ろ、更に首間接の性能によっては足元も見えなくなることがあります。これらの補佐をするのが、それ以外の目たちです。

例えば、脛の辺りにつけ正面のものとの距離を測る測定器、足の裏につけ足元の凹凸を感知するセンサー、真後ろや右左をカバーするカメラなどです。これらを装備することによって、ロボットの安定した歩行や人身事故などの発生をいくばくか防いだりすることが出来るでしょう。

2.5 最後に

人間が脳内でどのような画像処理を施しているのかまだ解明されていません。ですが、推測することは可能です。ヒューマノイドロボットの開発にはその実験の要素も含まれていると思います。これから更に技術が発達していき、人工知能が開発され、人間とのコミュニケーションがとても円滑になったとき、このロボットの目の真価が発揮されるでしょう。

人間は感情を表情に出すことがあります。それを正確に汲み取って（つまり、表情の変化を認識して）その時の感情に適した対応を取れるようになったとき、ヒューマノイドロボットは私達人間の新しいパートナーとなるのです。

始めの「安心できるパートナー」というのは、人間のように人間に接してくれるロボット、という意味です。人に似せているからこそ安心感があり、パートナーになりやすいのです。今後のロボット開発に期待したいところです。

参考文献

[1] 出口光一郎 「コンピュータビジョン　ロボットの「目」を作る」 丸善株式会社

[2] 「富士通オートメーション株式会社」
http://www.automation.fujitsu.com/products/products07.html

第III部

プログラミング

ソフトウェアを製作するときに欠かせないのがプログラミングです。ここでは、プログラミングに関する記事を載せています。

1 工芸的プログラミング

00230042 電子情報工学科 4 回生 越本 浩央

1.1 はじめに

ここではみなさんにいくつかの提言をしようと思います. それは, 端的に言うとプログラミングに関することであり, そして正確に言うならば言葉の問題だということを始めに申し上げておきます. 同時に断っておくと, この文章を読んで何か新しい事実が確認されるわけではありませんし, 扱うべき問題に新しい視点を与えるわけでもありません. それでもなお私がこのようにエネルギーを消費して, またみなさんの貴重な時間を割いている理由は, つまりそれこそがここで語られる事実なのですが, 今まさに私達の目の前に巨大な問題がそびえ立っているからであります. あと付け加えておくと, 当コンピュータ部の 1-2 回生には是非読んでもらいたいと思います.

では早速語ることとしましょう. プログラミング, その基礎論や方法論についてです. まるで水と油のように両者は互いを分け隔てていることでしょう. ご冗談を!と叫ぶ方もいらっしゃるでしょうか. ここは一先ず声を抑えて, しばらく私の話にお付き合い下さい.

1.2 準備

議論には手順がつき物です. そして, この場合もしかり. まずは問題を説明するための準備があり, 続いて問題が定義され, そして明晰なる論理の上で仮説と検証が行われ, ついに結論を得るわけです. これがまず準備という段階です.

現在のコンピュータは本質的にチューリング機械と同値である. というようなことをこれまでに一体どれほど聞いてきたでしょうか? 一体どこの誰がその事実を確認したのか, 全くの虚言です. 計算可能性という定義において現在の多くのコンピュータがチューリング機械と同値である, という事実は承諾致しましょう. ですが, 実行コードとデータが homogeneous に扱われることはほとんど無く, そしてこれからはますます逆方向 (heterogeneous) へと進展しようとしています. そのことが一体どれほど重要なのかとお疑いの方も居られることでしょう. この疑問への解答は一つの重要なテーマとなります.

次にソフトウェア開発を巡る現在の流行と資産についても触れておきましょう. みなさんも十分ご存知でしょう. 建築学とのアナロジーで展開されるソフトウェア工学です. ここでは複雑な議題を持ち上げる前に簡単なエピソードをお話しましょう. プロジェクトチーフを務めるあるソフトウェアエンジニアが建築プロジェクトのマネージメント講習会に参加しました. 講師が参加者に対し “

貴方は何を目的に参加していますか?" と尋ねたとき, そのエンジニアが "ソフトウェア業界では建築業界の方法論を学び応用することが, 一つの解決策になるのだと信じられ, そして実際の成果も上がっているのです" と答えると, 場の一同が爆笑したそうです. 建築の現場では実際非常に多くの問題が現在も (そして振り返るならば過去数千年に渡っても) 存在するのです.

1.2.1 計算原理

チューリング機械にしろ, 帰納関数にしろ, 本質的なことは計算限界を指示するための, 操作の表示的公理系だと見ることが出来るでしょう (意味論はまた別の次元で議論しなければなりません). これら公理系は半順序集合として一般化され, より制約を緩めれば先行順序集合となり, 並列システム等を記述することが可能となります.

過去, 計算原理はプログラミングパラダイムの原動力となり多くの成果を生んで来ました. 現在にも通用する成果の一つは, 副作用という問題の認識でしょう. 今簡単に手に入る副作用が無い強力な関数言語は Haskell(その処理系は GHC が有名) だと思われますが, 特に理論的な成果はモナドとしてソフィスティケイトされています. モナドについてここで議論はいたしませんが, 簡単に言うと参照透過性を維持したまま破壊的代入を実現する仕組みです. ユーザに対して非明示に代入対象を保持し, 同時に更新結果への参照を可能とするために, モナド関数は先行するモナドの結果を後続するリストに加え, ベルトコンベアーのように処理を手続き的に実現します. 代入履歴を更新履歴として見れば, まるでジャーナリングシステムの様ではないでしょうか?

計算原理のもう一つ重要なテーマに, 複雑さを効率的に扱うことが挙げられます. この点においてタイプ理論は極めて魅力的 (例え幻想的ではあっても) と言えるでしょう. もっとも, 正確に言うならばリカーシブタイプ (再帰的記述) とオブジェクトタイプ (ユニークな参照を持つタイプ) を前提としたオブジェクト指向とした方がいいでしょうか? いずれにしろその原点は自己言及に関する数学的パラドックスを解決するため, ラッセルとホワイトヘッドによって持ち込まれた階の概念にあります. 簡単に言ってしまえば, タイプとタイプの写像をレコードとして持つタイプ (オブジェクト) によって複雑な対象を正確に記述しようというものです. このオブジェクト上で成り立つ推論規則 (もっとも重要なのは, スーパークラスの出現はサブクラスで置換可能という Liskov 置換原則です) を考えると, これはまさにオブジェクト指向パラダイムとなります. 但し忘れてはいけないことがあります. 数学的なモデルは現実的複雑さの前に崩れ去ってしまいます. もしこのままオブジェクト指向を突き進めるならば, モデルの構築を演算能力によって実現する方向に進まなければ生き残る術は無いでしょう.

計算原理はその限界を明示的に記述する手段でもって, 自らの可能性を具体的に扱うことを実現します. 前者の主張は非常に重要ですが, 今の私達にはナンセンスな問題です. 後者の主張は, まさに昔も今も向かい合っている問題です. それは複雑さをより経済的に扱うことを競うゲームと言えるかもしれません. チャイティンに示唆を仰ぐことが出来ます. 彼はアルゴリズム情報理論で, あるサイズを超えるプログラムは還元不能な情報を含むと主張します. 彼の言葉を借りれば, "神は物理的事象だけでなく算術演算においてさえサイコロを降る" のだそうです. 彼の結果は, 自身も認めているように, ゲーデルやチューリングの総括の上に成り立っていますが, しかし絶望的な主張と言えるでしょう. あるいは遺伝的アルゴリズムは有力な梃子となるかもしれませんが, それが解決となる

ことは未来永劫ありえないことも論理的帰結です.

基礎論について重要な提言をして終わることとしましょう. ウルフラムは全く異なる希望を持っているということです. 彼が計算等価性原理と呼んでいるものです. Cellur Automata の計算能力は万能チューリング機械と等価であることが示されていますが, ウルフラムはさらに進んで次のように主張します. “機械的構造のみならず, 自然界の複雑な構造も等価的に Celluer Automata 上で再現可能である”. 流石に,Cellur Automata が直接的な解決となるようには思えませんが, Celluer Automata に対する意味論が可積分系として与えられつつある現状は, 少なからずの希望が残っていると言えるでしょう. また可積分系は物理的な性質のみならず, 数多くのアルゴリズムにも潜んでいることも明らかになってきました. 可積分の定義さえ数学的には不明瞭な部分が多いこともあり, 学術的な話題性は極めて高いと言えるでしょう.

1.2.2 デザインパターン

アレグサンダーが建築において用いられる構造的造形的パターンのリストを記した建築書に由来します. 広く一般の共通認識として, 建築とコンピュータが相互にアナロジーとして使われ始めたのもこれが始まりではないでしょうか? (驚くべきことに, 昨今の建築書ではコンピュータアーキテクトをアナロジーとして議論されることがあります). デザインパターンには様々な議論が交わされていますが, とりあえずの共通理解としてはそれが実装段階での言語であるということです. デザインパターンそのものについての多くの誤った議論がありますが, 利点や欠点という評価は全く無駄だと言えるでしょう. 各種パターンについても, 利用場面における適合という点を除けば利点欠点という問題は存在しないと言っても過言ではありません.

さて, デザインパターンについて語ることはどれほどあるでしょうか? 例えばパターンをクラス図で記述すると, それはグラフとして扱うことが出来ます. Rational のツールなどは設計段階でのパターンの適用などを機械的に処理出来るそうです. 話を少し戻せば, オブジェクト指向の推進力を演算能力に頼るべきだと述べました. パターンについてはまさにこれが体現されようとしているわけです. ライブラリとそのメタデータ, ユースケースに対応した設計モデル, それら複数の階層を繋ぐようなシステマティックなデータフローが実現されれば, ソフトウェア開発の現場は大きく変わるでしょうか?

デザインパターンが実装段階での言語であるという主張は, 非常に重く受け取るべきだと思われます. つまり, 我々の思考はまさにパターンで制限付けられるのだということです. GoF 本以降もパターンは発見されていますが, 多分貴方が記述するコードはほとんどが現存するパターンで整理されることでしょう. これはとても重大な事実です.

1.2.3 アナリシスパターン

デザインパターンと違ってアナリシスパターンはあまり有名ではないかもしれません. M. ファウラーのアナリシスパターンで紹介されている方法論であり, デザインパターンと違って定型的なパターンカタログなどはありません. そして, 言葉ではありません. 複数のビジネスモデルやアーキテクチャで現れる共通の構造がアナリシスパターンです. 対象は非常に抽象的で, パターンも実装とは

かけ離れた記述を得ることとなるでしょう. しかしそれはやはり構造であり, それ以外の何者にもなりません. 私達はその構造を設計で肉化することとなるのです.

言葉でないものをパターンとして発見することに重要性があるのかどうか, みなさんはお疑いのことだと思います. ファウラー本人から聞いたわけではありませんから, これは推測ですが, 発見するという行為がとても重要なプロセスであると言えるでしょう. 考えてみてください. 言葉とは発見に対する名に他なりません. 発見とは事実の認識であり, そこには真に新しい事実があるわけです. 重要なのは対象となるモノです. ビジネスモデルやアーキテクチャと言った, 非常に広範で抽象的な, あるいは全く未開の事実に対し, 発見という行為は自らの意識思考の拡張へと繋がることでしょう. まさにこの点で, 言葉とその前段階である発見のそれぞれの意義は分けられます. 同時にデザインパターンとアナリシスパターンの存在意義が示されるわけです.

1.2.4 アーキテクチャ

最後に語られることはアーキテクチャです. 私達は特にイメージとしての理解でアーキテクチャを捉えることと思います. これこそアーキテクチャの性質を露にしていると言えます. 人の脳は右脳と左脳によって, 大域的なイメージと緻密な論理を同時に扱うことが出来ます. 論理構造から精密に構築されていく計算原理を左脳の担当だとすれば, アーキテクチャはまさに右脳の専門領域だと言うことが出来るでしょう.

アーキテクチャに関する記述は, 古代の建築家ウィトルウィウスによってなされたものが最古だと言われます. ソフトウェアにおけるそれを, 古代の建築書に求めることは不毛ですが, 古代ギリシャ建築におけるアーキテクチャに関する理想論から学ぶことはあるでしょう. ギリシャ建築, 特に有名なのはアテナイのパルテノン神殿でしょうか. パルテノン神殿に限らずギリシャ建築では全ての神殿建築が一つの理想を追求しています. 曰くオーダーを基本として構成し, 地理的要因を踏まえてリファインメントを行うべし. オーダーとは柱の様式であり, それには力学的な根拠と装飾的な目的から複数の方式が存在し, またそれらはリファインメントを正確に行えるように体系化されていました. リファインメントは建築が全体として齟齬を来たさないように行われる微調整で, 最終的な造型が黄金率を見事に体現することを可能としたのです. 古代ギリシャにおいて建築とは神殿建築を指し, 神殿は神の居住まいでした. そのため, ロマネスクやそれ以降の建築に見て取れる多様性は全くありませんが, しかし神殿に求められた建築という点では既に完成を迎えていたとされます. そしてその方法論もまた, 現在における私達の進む道を見事に示していたと言えるでしょう.

さて, ウィトルウィウスの建築書ではアーキテクチャが次の三要素から成ると述べています. ユーティリタス (ニーズ) とベヌスタス (デザイン) とファーミタス (ストラクチャ). これらはまさに現在のアーキテクチャに求められているものと合致するのではないでしょうか? すなわち, ユースケースを達成するシステムデザインを設計し, それを実現する具体的な構造・実装を検討する, と.

それでは次から問題へと取り掛かることにしましょう.

1.3 問題

私がこれから述べる問題は一見全くの関連性を欠いていると思われるかもしれません. それは多分正解です. 実際, 私達は多くの問題を解決する必要があります. しかしそれは相互に様々な影響を与えるはずです. いずれそれらの相互干渉を, ネットワークとして高い視点で意味が与えられることでしょう.

1.3.1 トレードオフ

一体何が問題でしょう?トレードオフが問題だとはおかしな話です. そう, 問題はそこにはありません. 問題は私達の中にあります. トレードオフという分析が一体何を生むのでしょうか? 問題とはまさに私達の知識の体系化にあるのです.

ソフトとハードのトレードオフに代表される論理. ほぼあらゆるコンピュータ関係の話についてまわるのが, この論理です. 過去の歴史を見る限り, 問題に対する新しい解決はその多くがトレードオフを打ち破っています. そして実際私達が問題に直面したとき, トレードオフの論理は何の力も生み出しません. 極端な物言いをすれば, トレードオフに代表される論理はその多くが思考を停止されると言えなくないでしょう.

さて, ここまで非難を続けて, しかし私はそして貴方達も, 明日から変わりなくトレードオフという使うことでしょう. まるでセールストークのような言葉ですが, 知識の体系化でこれほど便利な制約はありません. 論理的整合性と定量的比較に基づき, 肥大するシステムの構造はトレードオフで片付けられます. トレードオフをまやかしのように使うアブストラクトナンセンスは問題外ですが, しかしそれ以外で一体どこが問題なのか? まさにそれが問題以外の何者でしょうか?

1.3.2 セマンティック web

バーナーズ・リーに限らず, 最近は猫も杓子もセマンティック web です. セマンティック web に付きまとう問題とは何でしょうか? この場合は技術的な問題よりむしろ形態について問題があると言えます. つまり, セマンティック web で生まれるビジネスモデルです.

セマンティック web で実現されることの多くは, 現在のオンラインサービスにはほとんど適合しません. Amazon しかり,Sparco しかり, あるいはホテル・チケットのオンライン予約など, その大部分が顧客に密着し, 人の優柔不断な性格を上手く利用して商売を行っています. バーナーズ・リーの言うことが実現されることで利益が上がるビジネスはとは何なのでしょうか?

もっともこの問題の見通しはまだ良いと言えます. 新しいビジネスモデルの構築がその解答でしょう. ウェアラブルデバイス (果たしてそんなものが将来的にも意味付けられるのかは一つの小問題です) やユビキタスコンピューティングと関連して新しいモデルを期待したいところです. P2P モデルにおけるデジタル証券というのが巷で話題に上がっていますが, コミュニティを超えたワールドワイドな浸透は疑問です. もっと違う解答が必要となるでしょう. 逆に, セマンティック web に関しての様々な規制などが出来ることは最悪の解答です.

ではセマンティック web のもう一つの素晴らしい側面を考えてみましょう. それは例えば世界図

書館キサナドゥかもしれませんし, ウィトゲンシュタインの描く論理形式の模倣かもしれません. 何が可能か, まさに全てでしょう. 不可能なことは語りえません. 与えうる限りの事実が与えられれば, それは表現しうる限りの表現を出力することが可能です. 当然の主張でしょうか?しかしそれは容易に受け入れられない問題を抱えます. 事実が有限である以上に, 表現しうる全ては全事実の直積空間から抜け出ることは無いでしょう. これはすなわち二つの大きな問題を意味します. 解空間の爆発的増加と創造性の欠落です. 前者には強力な推論機構を, 後者にはより活発なコミュニケーションを. セマンティック web の発展を望むばかりです.

1.3.3 プログラミングパラダイム

今の流行りと言えば Generic Programming でしょう. 代表する技術的項目を挙げるとしたら,C++の template を真っ先に述べましょう. LISP のマクロにしろ,Smalltalk のメタオブジェクトにしろ, そして template にしろ, その目指す先は非常に似たところであるのは事実だと思います. Meta Programming が Generic の行き着く先だとは明言出来ませんが, それらに共通してある何かが私達の目を向ける先にあると確信しています. そしてこれらのアブストラクトな構造が, 実践的な場面で本当に活躍する様子が幾つも確認されています.

ではこの流行の方向で他の要素技術と結びついて広大なプログラミングフレームワークが実現されるのでしょうか? ある意味でそうであり, またそうでないと考えます. キーとなるのはスケーラビリティだと, みなさんの見解はそんなところではないでしょうか? 恐らく template が継承 (によるポルモーフィズ) に対して優位な結果を示すのは, スケーラビリティに起因しているためだと思われます. スケーラビリティは他の要素技術の側面においても同様な比較をもたらします. 私達は経験的にも, そのように理解することでしょう. つまりスケーラビリティこそがメルクマールであり, 複雑に肥大化するソフトウェアの構築に経済的な形態を導いてくれる, と. しかしそこに問題を見ようと考えています.

私達の科学技術の屋台骨を支えているのは非常に簡潔な, しかし柔軟で拡張性豊かな法則です. 特に線形性として与えられる構造は現存する技術のほぼあらゆる場面に現れると言っていいでしょう. 必然的に,(現在のところ一見は) 非線形なものが踏み入るべき未開の地であり, 眼前にそびえる壁なわけです. 果たして手の出しようがないのでしょうか?そんなことはありません. 最近の数理科学で得られた非常に強力な成果により, 指数関数の有理多項式によって一部の (しかし頻出する) 非線形現象を線形化することが可能となりました. この操作で得られた対象は無限次元の多様体上での恒等式 (実は Plucker 恒等式そのもの) に従うことが分かり, 結局は線形的法則の上で完全に正確な取り扱いが現実のものとなったのです. これをソフトウェアの世界でもやることとなるでしょう. 急速に肥大する複雑さをこれまでの方法論で扱えるような (チャイティン流に言うと経済的な圧縮を得るような), 包括的な抽象的概念がこれまでの, そしてこれからのソフトウェアに潜んでいることでしょう. よりジェネラルに, よりアブストラクトに. 恐れることなくナンセンスの領域に夢を見ましょう. 勿論目的を失ってはいけません. しかし着地点さえ理解していれば, 想像力の許す限り高い次元へと飛び立つことが私達に求められているのです. かつて数学界の若者がこぞって従ったように, カテゴリーやスキームの基礎を全くのゼロから夢想したグロタンディークに追随するのです.

余談ですが追記しておくと, 私が面白いと思い人にすすめる言語は Haskell です. 新しい GHC で

は Meta Programming もサポートされています. 謳い文句を付けるとしたら, 恐れることなく最先端の言語と言いうことでしょう. おそらくまだ見たこともない何かがそこにあると思います. 興味の無い人に是非学んで頂きたい言語です.

1.3.4 インターフェース

インターフェースの問題は特に語るまでもなく意識されていることでしょう. 現在ほとんど盲目的に利用されているファイルの概念から始まり, 入出力デバイスや, ユーザインタラクションの定義をどう決定付けるか等々. カンブリア期のように問題が爆発し, そして多様な方向性が模索されることでしょう. 最も楽しみで, 最も楽観視することが出来る問題だと私は思います.

1.3.5 コンピュータ

最後に語るべき問題はコンピュータです. パーソナルコンピュータを想像しないでください. 携帯端末から個人用コンピュータ, デジタルカメラ, ヴィジュアリゼーションワークステーションやスーパーコンピュータ等々. 医療分野での特殊機器は最新技術が常に投入される魅力的なコンピュータです. 最近の自動車には多くなると 70 の MPU を搭載するのだそうです. 私達の生活の回りには至る所にコンピュータが潜んでいます. 最後に問題として訴えたいのはパーソナルコンピュータでしょうか? 確かにそれもそうです. 私達は過去に追われても, あるいは今を生かされるだけでもいけません. 未来に向かって生きるプライドを持つべきです. プライドは傲慢さではありません. 貴方は 20 年後も Intel の入ったデスクトップ PC でゲームとチャットと web サーフィンをしているのですか?

歴史を学び, 未来を作りましょう. 古代, コンピュータが一切の手作りで, 他の有識者が馬鹿にする中, 自分の頭にあるヴィジョンを信じて夢を紡いだ偉人たち. チューリングに学びましょう. 74 シリーズがなければ何も出来ないわけではありません. 自然界は宝の宝庫です. そしてみなさんはまさに今存在し, そして物理法則を感覚しているのです. 流れる水を工夫しても計算機は出来ます. 今年, 微量流体を用いることで実用レベルの計算速度を実現させたプロジェクトもあります. 一昨年は DNA コンピュータも実現されました. 量子コンピュータも時間の問題でしょう. 計算の基礎も変わります. 私の研究室に同室する連中は何やら量子コンピュータのアルゴリズムを勉強しています. 高名な数学者アーノルドは “物理学は最先端の学問だ. 数学はその流行遅れをやる” と言います. さしずめ, 工学は洗練された数学的実在で遊ぶと言ったところでしょうか?

1.4 言葉と思考

突然不思議な項目だと思われたでしょうか? しかしここまでめったやたらと話をしてきて, 最後にどうしても述べなければならないことがあります. 言葉と思考についてです. 紙面も尽きてきました. 簡潔に申しましょう. 貴方が思考するその限界は, その言葉に縛られます. 言葉は社会に影響され, 社会は貴方を含めたコミュニティの意志です. 規範と思想は螺旋を描くように相互再帰を繰り返します. 思想から言葉は生まれます. その無限連鎖から逃れることは不可能です. それは問題ですら

ありません. もし逃れてしまえばそれは全くの無意味でしょう.

ウィトゲンシュタインは完璧な正確さを持って事態を描写します. “およそ語られうることは明晰に語られうる. そして, 論じえないことについては, 人は沈黙せねばならない” と.

1.5 おわりに

おそらくこの文章は, ここに表されている思想, ないしそれに類似した思想, をすでに自ら考えたことのある人だけに理解されることでしょう. それでもなおこれら事実が述べられている限りにおいて, この文章は価値があるでしょう.

最後に, ここまでお読み頂いた方々へお願いがあります. 私の言葉を鵜呑みにする愚挙だけは, 決して犯さないで下さい. 常に問い続けるのです. そして願わくば事実が明晰に語られたことを祈りつつ...

2 オブジェクト指向プログラミングの基礎の基礎

00230713 電子情報工学科 4 回生 重森 晴樹

2.1 序文

こんにちは。部員なのにあまり活動できていない重森です。夜間主の学生かつ社会人でもあって忙しいこともあるのですが，一回くらい部誌である `Lime` に寄稿しようと思い、キーボードを取らせて頂きます。

私が C++を長いことやって[1]慣れていてサンプルコードを書きやすいので、C++で説明します。しかし、基本的な概念は Java でも C#でも同じですから、読み替えるのは容易でしょう。最後に付録として C#のソースコードも載せておきます。また、ここで言語仕様を説明しているといくらページと時間があっても足りないので、その辺りのことは大きく割愛します。お許しください。

2.2 オブジェクト指向の三本柱

オブジェクト指向には、いくつかの重要なキーワードがありますが、その中でも次の 3 つは中核を成す存在でしょう。

- カプセル化
- 継承
- 多態性

このセクションでは、この 3 つのキーワードを説明します。ですが、その前にオブジェクトとクラスについて少し述べておきます。

2.2.1 オブジェクトとクラス

オブジェクト指向の世界では、あらゆるものをオブジェクトと考えることが出来ます。イメージしにくいかもしれませんが、ある対象に名前を付けられれば、それは対象を独立した 1 つの「もの」

[1] だらだらと無駄にやっていただけともいう。

として考えているわけで、どんなものでもいいのです。例えば、具体的にイメージしやすいファイルや文字列もそうですし、状態のような抽象的なものもオブジェクトと考えることが出来ます。

あなたがプログラミングする場合、オブジェクトの雛型となるクラスというものを設計することになります。雛型ですから、1 つのクラスからは複数のオブジェクトを生み出すことができます。この生み出した実体をインスタンスといいます。感覚的には普通の変数と似たようなものです。違いがわからない場合は、独立した固有の「もの」がオブジェクトで、その設計図がクラス、組み立てた実体（変数）がインスタンスとイメージすればいいのではないでしょうか。

オブジェクト指向では、そのオブジェクト固有のデータと、データを操作する手続きをひとまとめにして扱います。言語によっていろいろな呼び方がありますが、C++ではデータをメンバ変数、手続きをメンバ関数といいます。まずメンバ変数をオブジェクトの中心要素と考えるのがポイントで、メンバ関数はメンバ変数を操作するための機能といった位置づけにあります。もしもメンバ変数を「オブジェクトの機能に必要なデータ」と考えてしまうと、メンバ関数が中心要素になり、良いクラスの設計をすることが難しくなります。メンバ変数はただのグローバル変数ではありません。オブジェクトにとって、もっと重要な本質であることを常に忘れてはいけません。

2.2.2 カプセル化

このように、プログラム全体を「オブジェクト」=「データと操作をひとまとめにしたもの」の集合と考えます。しかし、いくら分割しても全体としては何らかの動作をするわけですから、オブジェクト間で何らかの連携をとらなければいけません。この、他のオブジェクトに対して送る連携のための合図が、C++では相手のオブジェクトが公開しているメンバ関数のことです。

ここで「公開しているメンバ関数」という言葉がでてきましたが、「非公開なメンバ関数」もあります。これは基本的には他のクラスから呼び出されることはなく、クラス内部のメンバ関数からしか実行することは出来ないメンバ関数です。つまり、外部から実行されても異常をきたさないような公開メンバ関数（public メンバ[2]関数）と、勝手に実行されては困るメンバ関数（private メンバ[3]関数）をしっかり分けることで、そのオブジェクトが安全なものであることを保障しようというのが、カプセル化です。

完全にカプセル化するには、いつ、どんな順序で public メンバ関数を呼び出されても問題が起こらないようにしなければいけません。自分が設計したクラスを他人が使うような場合は特に、十分にカプセル化できているか注意するようにしましょう。そうでなければ、思いもよらぬトラブルに見舞われるかもしれません。また、以前のプログラミング言語でも、モジュールの結合度はなるべく低い方が良いとされてきました。それはオブジェクト指向でも同じです。うまくカプセル化することで、クラス間の結合度を低くすることもできるでしょう。

では、カプセル化された小さいクラスのサンプルを示します。

```
1: class Fruit {
2: protected:
3:     int value;
```

[2]どのクラスからでもアクセス可能。
[3]自クラス内でしかアクセスできない。

```
4: public:
5:     Fruit():value(0){}
6:     virtual ~Fruit(){}
7:     int getValue() { return value; }
8:     char* getName() { return "Fruit"; }
9: };
```

これは果物を表すクラスです。メンバ変数 value は価格で、初期値は 0 です。getValue() で価格が、getName() で名前が得られます。value は、まだ説明していない protected メンバ[4]ですがここでは非公開と考えてください。このように、メンバ変数は非公開にして、値を得るときはメンバ関数を介するべきです。そうすることで、メンバ変数が異常な値をとることを防止できます。

2.2.3 継承

今、Fruit クラスという大雑把に分類したクラスを作りましたが、これを大分類として、小分類に当てはまるリンゴとオレンジを表すクラスを考えます。ただし、せっかく Fruit クラスというすべての果物の基礎になるクラスを作ったのですから、これを有効利用したいのです。さて、次のような感じでしょうか。

```
 1: class Apple : public Fruit {
 2: public:
 3:     Apple(){}
 4:     virtual ~Apple(){}
 5:     char* getName() { return "Apple"; }
 6: };
 7: class Orange : public Fruit {
 8: public:
 9:     Orange(){}
10:     virtual ~Orange(){}
11:     char* getName() { return "Orange"; }
12: };
```

2 つのクラスを見ると、value や getValue() が定義されていません。サブクラス（子クラス）は、スーパークラス（親クラス）のすべてのメンバを持つので、Fruit クラスを継承しているこれらのクラスには書く必要がないからです。また、サブクラスの getName() のように同じ関数を書けば、スーパークラスから受け継いだ関数をわざと上書きすることが出来ます[5]。このように、継承という機能を使えば、既存のクラスを元に変更・拡張することができます。

Fruit クラスの value はサブクラスで書き換えられることを想定していたので、private ではなく protected にしました。private にした場合は、サブクラスからも直接書き換えることはできません。

[4] 自クラス内と、子クラスからのみアクセス可能なメンバです。
[5] これを関数のオーバーライドという。

2.2.4 多態性

「継承は機能の変更・拡張のためにある」と書きましたが、最初からその目的で使うことを想定してクラスを設計すると、継承のデメリットである「クラス間の結合度が高い」ことが悪さをして、後で苦労する可能性があります。継承には、もっと有効な利用法があるのです。

次のコードは今作ったサブクラスを使っていますが、もっとサブクラスの種類が多いと思ってコードを見てください。

```
1: void Main() {
2:     Apple apple;
3:     Orange orange;
4:     //...
5:     printf("%s %d\n",  apple.getName(),  apple.getValue());
6:     printf("%s %d\n", orange.getName(), orange.getValue());
7:     //...
8: }
```

同じことばかり書いていてスマートじゃないなぁ、と感じてもらえますか？他の多くの場合と同様に、同じような処理はまとめてしまうべきです。これは次のように書き直せます。

```
 1: void Main() {
 2:     Fruit* fruit[2];
 3:     fruit[0] = new Apple;
 4:     fruit[1] = new Orange;
 5:     //...
 6:     for(int i=0; i<2; ++i) {
 7:         printf("%s%d\n", fruit[i]->getName(), fruit[i]->getValue() );
 8:         delete fruit[i];
 9:     }
10: }
```

まず、`fruit` というポインタ型の配列に、`Apple` クラスと `Orange` クラスのインスタンスへのポインタを代入しています。すいませんが、`new` と `delete` の説明は省きます。ところが、`fruit` の型をよく見ると、`Fruit` クラスとなっています。違う型なのに、どうして代入することが出来るのか？実は、サブクラスは、スーパークラスの変数に代入することができるのです！では、実行してみると…。

```
Fruit 100
Fruit 80
```

サブクラスのインスタンスのはずなのに、スーパークラスの名前が表示されました。なぜでしょうか。それは、呼び出される関数は、インスタンスの型に依存するからです。`fruit` は `Fruit` クラス型ですから、呼び出されるのは `Fruit` クラスの関数です。それでは代入できても意味がないと思うでしょう。確かに、このままではせっかくサブクラスで `getName()` をオーバーライドした

意味がありません。そこで、Fruit クラスの getName() の定義を少し変更します。定義の前に、virtual と付け足してください。

```
8: virtual char* getName() { return "Fruit"; }
```

さて、今度はどうでしょう。

```
Apple 100
Oraneg 80
```

思惑通りにインスタンスの中身に応じて呼び出される関数が変わりました。virtual 指定の関数は仮想関数といい、スーパークラスのポインタにサブクラスのインスタンスを代入した場合、インスタンスの型によって呼び出される関数が実行時に動的に変わります。これを多態性といいます。

2.3 インタフェースと実装の分離

2.3.1 インタフェースという考え方

クラスには public メンバ関数がありますが、これはオブジェクトのデータに対して何らかの操作を行うためのものとすでに述べました。つまり、public メンバ関数というのは、そのクラスと外部とのインタフェースです。外部からは、そのインタフェースを用いてでしか、そのオブジェクトにアクセスできません。ですから、外部に公開するインタフェースさえ同じであれば、クラス内部の実装がどんな方法で行われても、変更されても、そのクラスを使う側のコードは変更を受けません。よって、インタフェースが同じクラスを丸ごと入れ替えることもできるようになります。

2.3.2 仮想関数と純粋仮想関数

2.2.4 で仮想関数を使いました。これはスーパークラスが継承される場合に、サブクラスで上書きされる関数を仮想関数にすることで、多態性を用いて多数のサブクラスをまとめて扱えるというものでした。

これをインタフェースと実装の分離という側面から見て、スーパークラスでインタフェースだけ定義しておいて、サブクラスではそのインタフェースに従って好きに実装できないか、ということが考えられました。このインタフェースとしてだけの関数を純粋仮想関数といい、次のように $= 0$ をつけ、中身は実装しません。

```
8: virtual char* getName() = 0;
```

また、純粋仮想関数を含むクラスは、共通のインタフェースを持ったサブクラスを定義するため

の雛形としての意味を持ち、抽象基底クラスと呼ばれます。抽象基底クラスは実装されていない関数を持つため、インスタンスを作れないので、必ず継承して、サブクラスですべての関数を実装してからインスタンス化することになります。

2.3.3 Java や C#との関係

Java/C#では、その名もズバリ interface というものがあり、これは C++では純粋仮想関数しか持たないクラスに相当します。Java/C#は単一継承のみで多重継承[6]ができないので、変わりに interface が用意されているわけです。また、C++ではスーパークラスのポインタ型に代入しましたが、Java/C#には言語仕様上ポインタがないので、

```
Fruit fuit = new Apple();    // Java/C#での変数の代入
```

という書き方で代入することができます。が、そもそも Java/C#のクラスは参照型[7]なので、そこまで把握していれば同じようなものだったりします。わざわざポインタを意識しなくてもいい辺り、Java/C#は新しいだけあって便利かもしれません。ただ、ポインタなどのバグの引き金になりやすい機能をプログラマに対して隠蔽しただけですから、内部で何をやっているのかはしっかり理解しておく必要があります。

また、Java は実行速度が遅いという問題もかなり改善されたようですし、速度に関しては C#は元から C++と大きな差はありません。Java/C#は言語仕様からして OOP[8]に向いた仕様になっています。

2.4 最後に

C++は 20 年も前の言語ですし、後から仕様を拡張してきた歴史を持つので、言語仕様が一貫していません。また、STL[9]などの標準ライブラリを知り尽くすのはかなり大変です。しかし、C++は自由度が高く大きな力を秘めています。使いこなせるようになれば、「なんでもできる感」の虜になりますよ。

[6]複数のクラスをスーパークラスとして継承すること。C++では可能。
[7]参照型の変数はヒープ上のオブジェクトを参照しており、変数の代入時には参照先のオブジェクトが共有される。また、対である値型の変数はスタック上にその領域が取られ、代入によって値がコピーされる。
[8]オブジェクト指向プログラミングの略
[9]標準テンプレートライブラリ

2.5 付録：C#でのソースコード

```
 1: using System;
 2: namespace CSharpSample {
 3:     class AppClass {
 4:         static void Main(string[] args) {
 5:             Fruit[] array = {new Apple(), new Orange()};
 6:             foreach( Fruit i in array ) {
 7:                 Console.WriteLine( "{0} {1}", i.getName(), i.getValue() );
 8:             }
 9:         }
10:     }
11:     abstract class Fruit {
12:         protected int m_value;
13:         public Fruit() { m_value = 0; }
14:         public virtual string getName() { return "Fruit"; }
15:         public int getValue() { return m_value; }
16:     }
17:     class Apple : Fruit {
18:         public Apple() { m_value = 100; }
19:         public override string getName() { return "Apple"; }
20:     }
21:     class Orange : Fruit {
22:         public Orange() { m_value = 80; }
23:         public override string getName() { return "Orange"; }
24:     }
25: }
```

参考文献

[1] “日系ソフトウェア 2003 年 7·8·9 月号”, 日経 BP 社

[2] “++C++; //未確認飛行 C++”

http://www-ise2.ist.osaka-u.ac.jp/˜iwanaga/

3 コンピュータプログラムのしくみ

03230715 電子情報工学科 1 回生 久保 達彦

えー、超ビギナーな私ですが、プログラムについてかたらせていただきます。まずコンピュータプログラムの働きについて考えてみましょう。例えば、ワープロプログラムは、文章入力や編集・整形・紙への印刷・スペルチェックまでも可能にします。ビデオゲームは、異星人の宇宙船を撃ち落としたり、物語の主人公になれたりします。このように可能性は無限ですが、どのプログラムにも共通していることがあります。それは「プログラムはデータを操っている」ということです。どんなコンピュータ言語でもデータを格納する方法を提供しなければいけませんが、それはわかりやすく名前をつけた変数がその目的を果たします。

3.1 変数

変数はいわば、コンピュータプログラムのメールボックスです。それぞれの変数にはデータが入っていきます。メールボックスに手紙が入っていくのと同じように。また各変数にアドレスがあるのもメールボックスに住所が割り当てられているのと同じです。変数には通貨型変数、テキスト型変数、実数型変数、ユーザー定義型変数などがあります。

通貨型変数　小数点の右から 2〜4 桁に置かれた、小数点固定の数。

テキスト型変数　1 つ以上の文字をしまっていく。

実数型変数　非常に小さい値か非常に大きい値を表すのに使われる。

ユーザー定義型変数　どんなサイズでも設定でき、他のタイプの変数を組み合わせることもできる。

変数がデータを格納することは言いましたが、次にこれら変数を使って作業する方法についてお話しましょう。変数に格納されたデータを処理するには、コンピュータには命令のリスト (プログラム) が必要です。プログラムの中の命令は、プログラムの「コード」と呼ばれます。コンピュータはコードが表している命令を処理することで、コードを「実行」します。

3.2 プログラムの流れ

もしプログラムが一連の命令を次から次へと実行するだけのものなら、たいした仕事はできないでしょう。コンピュータの真のパワーは、プログラムがデータに基づいて、実行するコードブロッ

クを選ぶときに発揮されます。プログラマは、どの操作で実行するか決めるときに、プログラムの使う比較やルールを定義します。例えば、if 文では if という言語の後ろに指定された比較式が正しいと判ったときだけ書かれた命令を実行します。

3.3 変数の宣言とスコープ

どのプログラミング言語も変数を定義するメカニズムを提供しています。このプロセスを変数の宣言と言います。変数を宣言することはプログラム言語にメモリ内に変数のための場所を確保させることであり、プログラマに変数のタイプを指定させることです。どのコードがどの変数にアクセスするかを決めるルールは、変数のスコープを決めるので、言語のスコープ規則と呼ばれています。一般的な変数のスコープには、グローバル変数、スタティック変数、ダイナミック変数とパラメータです。

グローバル変数は通常、プログラムが実行を開始した時に宣言されます。これらはプログラムの生涯に渡り存続し、プログラム内のどの変数からでもアクセスできます。

スタティック変数もプログラムの生涯を通じて存続しますが、たいてい一つの関数か、プログラムの一部にある関数からしかアクセスできません。

ダイナミック変数は一時変数で、特定の関数かコードブロックの実行中にだけ存在します。一度、コードブロックが実行をとめるとこれらの関数は消え、確保していた変数のメモリスペースが解放されます。

パラメータは特殊な変数で、関数に情報を渡すために使われます。関数パラメータはダイナミック変数であり、呼び出された関数の中でのみ存在します。

さて、ここからはコンピュータ言語の話をしてみます。まずは C 言語から。

3.4 コンピュータ言語のはたらき

プログラマにとって最も重要な道具は、特定の言語で書いたプログラムのソースコードを、コンピュータが実際に実行できる機械語に変換できる道具です。この道具はそれ自体がプログラムであり、あるコンピュータ言語を「実装している」と表現できます。コンピュータ言語の実装のタイプは大きく分けて二つ。コンパイラとインタープリタです。「コンパイラ」はプログラム全体を機械語に変換する。一度変換されたらプログラムは CPU で直接で実行します。「インタープリタ」は、プログラムのソースコードを読み込み、実際にコードをマシン語に変換することなく、指定された処理を実行します。インタープリタは作成したプログラムを実行するのでプログラムには常にインタープリタが必要です。コンパイラとインタープリタとの大きな違いは、インタープリタはコードが一度に 1 行か 1 文ずつ実行されるので、プログラムを対話的に実行することができます。どの時点でも実行をやめて、変数やコードの変更ができます。インタープリタの不利な点は、コンパイルされたプログラムよりも実行速度が遅いことです。なぜならば、インタープリタは実行するたびにプログラムをマシン語に翻訳するからです。

3.5 コンピュータ言語

では次に C、BASIC、アセンブリ言語について

3.5.1 C 言語

C 言語には 2 つの素晴らしい力があります。一つは高度な移植能力 (あるコンピュータ上で書いた C プログラムは別のコンピュータ上でコンパイルし直せば実行できる)。もう一つは C 言語は高水準言語であるのと同時に低水準言語でもあるということです。C 言語の強みはここにあります。まあ例えば、低水準言語は卓越した処理速度を提供しますが、作業がメンドクサイわけです。逆に高水準言語はメモリ管理などの細かい部分が操作が不要なのです。そして、高水準の命令は効率的に実行できる多くの低水準の命令組合せへ変換され、処理速度を大幅に下げます。しかし C はこれら両方を実現します。この低水準と高水準のミックス機能が C を理想的な言語にしています。なぜなら多くのプロジェクトが、プログラミングの効率性と処理速度の速さを必要とするからです。

3.5.2 BASIC

BASIC は最も有名な言語の 1 つです。少し前までの BASIC は機能に制限があり、非常に遅いなどの欠点がありましたが、今の BASIC はこれらの欠点を克服しています。BASIC はインタープリタで使用されることが多いですが、BASIC コンパイラを利用して、プログラムを高速に実行することがあります。また、BASIC は C よりも文字列を操作しやすいという特徴をもっています。しかし、最も標準化が進んでいない言語のままであるという欠点があります。BASIC の特定の処理系のために書かれたプログラムは、別の BASIC の処理系では正しく作動しないことがあります。

3.6 アセンブリ言語

アセンブリ言語は最も低水準のプログラミング言語です。また、プロセッサと密接に結びついているので、CPU は種類ごとに独自のアセンブリ言語を持っています。しかし CPU によってはわざと他の CPU の命令のすべてか一部を実行するように設計されています。アセンブリ言語を使うには、コンピュータが内部でどのように動くかを理解しないといけません。C などの高水準言語は、変数がどこにどのように格納されているか知らなくても、宣言すれば、作業を実行でき、変数を確保するスペースの処理も言語の方でやってくれますが、アセンブリ言語を使う場合、メモリ内の変数の位置と順番を指定しなければなりませんし、CPU 自体が操作できる変数型は制限されています。アセンブリ言語を使って作業する場合、CPU 自体に組み込まれた特殊なメモリのタイプである「レジスタ」を直接操作できます。レジスタには様々な種類があり、ハードウェアによって異なります。

多くの CPU は、少なくとも 1 つ「アキュムレータ」をを持っています。これは「データレジスタ」とか「汎用レジスタ」と呼ばれることもあります。アキュムレータは計算と情報の一時的な格納のために使われます。レジスタのビット数は、プロセッサ (CPU) のサイズを決めます。例えば、

18 ビットのレジスタを持つ CPU は、18 ビットプロセッサと呼ばれたりします。「フラグレジスタ」の個々のビットは異なる意味を持ち、CPU の処理によってセットされます。現在、全体がアセンブリ言語で書かれたアプリケーションはほとんどありません。たいていの場合、プログラム中のどうしてもスピードが必要な箇所にのみ使われています。ほかのコンピュータ言語と違って、アセンブリ言語はある時刻にどのデータをレジスタ内に保持するか操作できるので、CPU を最大限活用することができます。ただし、アセンブリ言語のプログラミングは、高水準言語に比べると難しく、より時間がかかります。

3.7 最後に

なにかと無知な私ですので、何か間違いがあるかもしれませんが、一応一通りのことは書いたと思います。これからも勉強して精進して行きたいとおもいます。

参考文献

[1] ダニエル・アップルマン, 「コンピュータ・プログラムのしくみ」, ソシム

[2] 椋田 實, 「はじめての C」, 技術評論社

[3] 矢沢 久雄, 「プログラムはなぜ動くか」, 日経 BP 社

第IV部

ソフトウェア

今日、ソフトウェアには様々なものがあります。ここではソフトウェアに関する記事を載せています。

1 PerlでSQLを操ろう

00230098 電子情報工学科 4 回生 松村 宗洋

1.1 SQLってそもそもなんだ？

という質問をいきなり浴びせられてもいいように、念のため説明しておきます。最初に断っておきますが、「SQLっていえば …?」…「2 種 (基本情報処理試験) でいつも点落とすムカつくあれだよね？」なんていう DQN[1]な返事をしないでください (笑)。SQL とは Structured Query Language(構造化問合せ言語) の略であり、とあるデータベースに特定のデータを操作するための問い合わせするための言語です。「問い合わせ」という部分では近いところで話をするとメールを読むときに使ったりする POP3 やらに似ているイメージを持ってもらったらいいです。データベースと聞いてなんやら難儀なイメージをお持ちの方は、そんな難儀なものじゃないと思っていただけたらいいです。詳しい話は割愛しますが、ぶっちゃげて話すと数学でいう行列みたいなのがデータベースだとおもってもらったらいいです。SQL はそんなデータベースが複数個あったとして、そのタグづけ (名前つけ)、データの共通の属性 (関連した同じ要素)、そしてそのデータベースの操作や保守 (定義、追加、更新、削除) をひっくるめた全体を管轄する機構「DBMS(DataBase Management System)」を操作する言語なんだな、というイメージでいいと思います (図 1.1)。

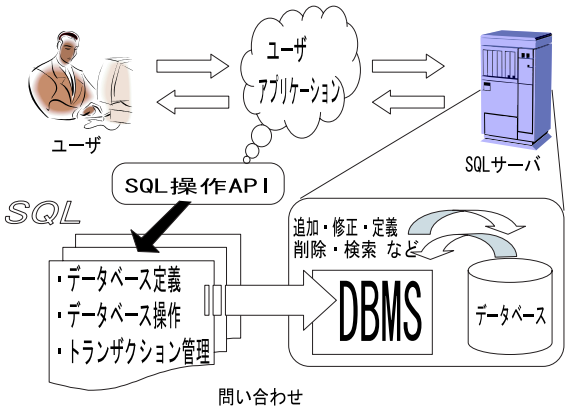

図 1.1: SQL の位置づけ

[1]人生に確固たる目的も持たず、反社会的な行動をとったり、自堕落な生活を送る者の蔑称。この言葉を最初に使ったのは「マミー石田」氏とされる。

えっと、最初に断っておきますが、この記事では SQL の仕組みがどーのこーのとか、データベースにはこんな種類があって云々ということを特集するものではありませんので悪しからず。とりあえず SQL で遊んでみるためにはこうすればいいんだよという感覚で書いていきますんでよろしくです。

1.2 とりあえず SQL な環境を作ってみましょうか

SQLってのは言語の名前だからソフト名でもなんでもないわけです。SQL な環境をつくるには**SQL サーバ**と、SQL サーバに問い合わせをするための **SQL クライアント**の二つがいるわけです。SQL サーバとして代表的なものは **MySQL**(マイエスキューエル) や **PostgreSQL**(ポストグレス・ポスグレエスキューエル)、**Microsoft SQL** とかがあります。前者二つはフリーで、商用にも使用可能です。フリーだからといって機能的な劣りはまったくないと思います。個人的には PostgreSQL が好きなんでこれをインスコ[2]したいと思います。FreeBSD だったら ports コレクションからコマンド一発で入れたら楽なんですが、一応ちゃんとソースからコンパイルして入れることにします (データベース保存場所とかいろいろ決めないといけないしね)。今回は Postgres-7.3 を使います。ソースをダウソしてきて解凍します。ユーザアプリケーションとして今回はプログラム言語 Perl を使用します。そのため Perl のモジュールも入れるように Configure オプションで指定します。

```
% tar xzvf postgresql-7.3.tar.gz
% cd postgresql-7.3/
% ./configure '--prefix=/usr/local/pkgs/postgresql-7.3' \\
  '--enable-recode' '--with-perl'
% make
% su -
# make install
# adduser pgsql
# mkdir /usr/local/pgsql/data
# chown pgsql /usr/local/pgsql/data
# su - pgsql
$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
$ /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data >logfile 2>&1 &
$ /usr/local/pgsql/bin/createdb test
$ /usr/local/pgsql/bin/psql test
```

面倒くさいんでとりあえず流れだけです。コンパイルしたら PostgreSQL 本体をインストールして、次にデータベースを保存する場所を決めます。データベースがちゃんと作れるか「test」っていう名前のデータベースを作ってみます。あ、あとシステムが起動したときに postmaster が自動起動するように起動スクリプト[3]でも作っておきましょうか。

/usr/local/etc/rc.d/pgsql.sh

```
PREFIX=/usr/local
```

[2]インストール・install・セットアップ の俗語

[3]起動スクリプト … UNIX 系では OS を起動した時に自動的に起動させるプログラムを登録しておく場所がある。FreeBSD では/usr/local/etc/rc.d/* 以下、Linux や Soralis は /etc/rc.d/* 以下

```
PGBIN=${PREFIX}/bin
logfile=/var/log/pgsql

case $1 in
start)
    touch ${logfile}
    chmod 600 ${logfile}
    chown pgsql:pgsql ${logfile}
    [ -x ${PGBIN}/pg_ctl ] && {
        su -l pgsql -c \
            "[ -d \${PGDATA} ] && exec ${PREFIX}/bin/pg_ctl start -s -w -o -i \\
                                                    -o -N15 -l ${logfile}"
        echo -n ’ pgsql’
    }
    ;;

stop)
    [ -x ${PGBIN}/pg_ctl ] && {
        su -l pgsql -c "exec ${PREFIX}/bin/pg_ctl stop -s -m fast"
        echo -n ’ pgsql’
    }
    ;;

restart)
    [ -x ${PGBIN}/pg_ctl ] && {
        exec su -l pgsql -c "exec ${PREFIX}/bin/pg_ctl restart -s -m fast"
    }
    ;;

status)
    [ -x ${PGBIN}/pg_ctl ] && {
        exec su -l pgsql -c "exec ${PREFIX}/bin/pg_ctl status"
    }
    ;;

*)
    echo "usage: ‘basename $0‘ {start|stop|restart|status}" >&2
    exit 64
    ;;
esac
```

これで PostgreSQL サーバが稼動することになります。説明荒くてすいませんです。メインじゃないので … 。それでは次にちょっとばかしコマンドラインから SQL を呼び出してテストしてみましょう。

1.3 SQL コマンドを少しは覚えよう

1.3.1 前準備 〜テーブルの定義〜

さっそくお手軽にデータベースでも作ってみるとします。PostgreSQL をインスコ[4]すると **psql** というクライアント用コマンドもインストールされます。このコマンドラインから SQL 文法に従った文字列をパチパチと入力するとちゃんとデータベースにアクセスしてデータを読み込み、書き込み、修正、削除とかできるのです。もちろん最初のデータベースの定義もできます。ではさっそくデータベースを作ってみましょう。まず PostgreSQL データベース上の root アカウントともいえる pgsql ユーザになりましょう。以下のコマンドで「test」というデータベースを作成します。

```
$ createdb test
CREATE DATABASE
```

「test」というデータベースを psql で開くには

```
$ psql データベース名
```

で開けます。

```
$ psql test
Welcome to psql 7.3.2, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

test=#
```

「`test=#`」となってるのがプロンプトです。このプロンプト部分に SQL 文をポチポチと打ち込んでやると色々な操作ができるわけです。さて早速データベースの中身を作ってみましょう。データベース定義を直で打ち込むのは骨がおれるので、SQL 文を書いたファイルをひとつ用意して、シェルスクリプトみたいに実行できますんで、その方法で読み込ませましょう。とりあえずテストとして、超簡単な Web カウンター用 SQL スクリプツを書いてみます。

test.sql

```
-----------------------------------------------------------------------------
-- SQL scentences for "chou kantan Web counter"
--
-- [how to use]: just type the commands bellow
```

[4] インスコ … インストール、セットアップ の俗語

```
--  pgsql@host % psql test -f ’./test.sql’
------------------------------------------------------------------------------

DROP TABLE choucounter;          -- delete table if exsist
CREATE TABLE choucounter (
        id SERIAL,                      -- internal serial id
        count INTEGER,                  -- count number
        date TIMESTAMP,                   -- access date
        cookie BOOLEAN,                 -- cookie flag
        ipaddr INET,                    -- ip address
        useragent TEXT Default NULL,  -- useragent
        PRIMARY KEY (id)
);

--CREATE USER httpd WITH PASSWORD "gomi01";
CREATE USER httpd;
ALTER USER httpd WITH PASSWORD ’gomi01’; -- change password of "httpd" user
REVOKE all ON choucounter FROM public; -- disable all authorities
REVOKE all ON choucounter_id_seq FROM public; -- disable all authorities
GRANT all ON choucounter TO httpd;    -- put all authorities to "httpd"
GRANT all ON choucounter_id_seq TO httpd;    -- put all authorities to "httpd"
```

ちなみに、SQL スクリプツでは行中の「--」以降がコメントとなります。SQL 文は区別しやすいように全部大文字にしてあります。id, date, cookie, ipaddr, useragent という 5 つの属性を作りました。それが前半の部分です。DROP 文はもし同じ名前のテーブルがすでにあった場合消去するための文です。後半の CREATE USER 文はこの「test」データベースにアクセスできるユーザを作ります。PostgreSQL ではデータベースにアクセスできるユーザの概念があります。又、ユーザごとのアクセス権限も設定できます。その操作をする文が REVOKE や GRANT です。初期状態ではデータベースユーザならだれでもテーブルの内容を読み込みできる権限になっています。REVOKE で権限を奪う、GRANT が権限を与えるコマンドなので、これらを使って適切な権限を設定しましょう。今回は httpd というユーザを作って、このユーザ以外はテーブルに触れることも触ることもできなくするようにします。

また、PostgreSQL には SERIAL というデータ型があり、これは何かしらのデータ列が INSERT されたときに自動的にインクリメント (+1 増加) してくれるという超便利な型です。カウント数のデータ属性に使いたいところですが、あとの保守のことも考えてぐっと歯をくいしばって普通に INTEGER 型 (整数型) を使いましょう。

あとはこのスクリプトを「test.sql」と名前をつけて保存し、以下ように実行すると、

```
$ psql test -f ’./test.sql’
```

スクリプトが読み込まれそれぞれの行が実行されます。

```
test=# \z
           Access privileges for database "test"
 Schema |       Table        |        Access privileges
--------+--------------------+---------------------------------
```

```
 public | choucounter        | {=,pgsql=arwdRxt,httpd=arwdRxt}
 public | choucounter_id_seq | {=,pgsql=arwdRxt,httpd=arwdRxt}
(2 rows)
```

\z コマンドで確認してみましょう。Access privileges の arwdRxt の部分が権限を表しています。最初の「=」の左の空白は public アクセス、つまり「誰でも」に相当する権限です。「=」右側が権限を表します。みごとにすっぱ抜かれてます。ルートユーザ pgsql とあとで作ったアクセス用のユーザ httpd には全権限が与えられているのがわかると思います。

1.3.2 データの読み出し、追加

\d コマンドでテーブルの定義を表示できます。

```
test=# \d choucounter
                          Table "public.choucounter"
  Column   |  Type     |                         Modifiers
-----------+-----------+------------------------------------------------------------
 id        | integer   | not null default nextval('public.choucounter_id_seq'::text)
 count     | integer   |
 date      | timestamp |
 cookie    | boolean   |
 ipaddr    | inet      |
 useragent | text      |
Indexes: choucounter_pkey primary key btree (id)
```

カウンターの最初、とりあえずダミーの行を追加しておかないとスタートできないので、テストがてら以下の SQL 文で追加してみましょう。新規追加には INSET 文を使います。

```
test=# INSERT INTO choucounter(count, date, cookie, ipaddr, useragent)
                       VALUES(0, '2003-01-01 00:00:00', '1', '0.0.0.0', 'test-user');
INSERT 25372 1
```

これで追加完了。追加できてるのかの確認を SELECT 文使ってやってみましょう。

```
test=# SELECT * FROM choucounter;
 id | count | date | cookie | ipaddr  | useragent
----+-------+------+--------+---------+-----------
  1 |     0 | 1000 | t      | 0.0.0.0 | test-user
(1 row)
```

OK。無事に追加できています。えっと、SQL 文の文法がわからないっていう人はネットで検索してみてください。たくさんでてきますんで。さてこれで下準備は完了。次のステップにいきましょうか。

1.4 さっそくプログラムを書いてみましょ

図 1.1 にも描いておきましたが、ユーザプログラムと SQL がおしゃべりするには何かしらのライブラリやモジュール (API 類) が必要です。今回 Perl を使うので「DBI」と「DBI::Pg」っていう CPAN[5] モジュールというものを使います。余談ですが、自分でなにかしらのモジュールを作ろうとしている方は車輪の発明を 2 度しなくてもいいように CPAN で検索をかけてみるといいでしょう。これらのインストールも make と make install で済みますので解説は省略です。

さて、それではさっそく DBI を使ってアクセスしてみましょう。せいぜい使用する DBI メソッドは表 1.1 くらいです。

表 1.1: よく使う DBI メソッド

メソッド	解説
connect	データベースサーバに接続する
disconnect	データベースサーバから切断
prepare	SQL 文をセットする
execute	セットされた SQL 文を実行する
fetch	フィールドのリファレンス配列として次行を取り出す。最速
fetchrow_array	フィールドの配列として次行を取り出す。扱いやすい
fetchrow_hashref	ハッシュテーブルへのリファレンスとして次行を取り出す。簡単

これらだけでも非常に楽な API とは思えますが，もっと楽をするために (笑) こんな Perl モジュールを作ってみました．PgWorks.pm というファイル名で保存し，実行スクリプトと同じディレクトリ，もしくは Perl の@INC パスの通った場所に置いてください．

PgWorks.pm

```
;#############################################################################
;#  SQL(PostgreSQL 用) 関連を扱うモジュール
;#
;#   PgWorks.pm  (PostgreSQL Perl API)    Sowyo Matsumura 2003 4/21
;#
;#############################################################################
package PgWorks;

use strict;
use DBI;
use Data::Dumper;

sub new
{
    my $class = shift;
    my %arg = @_;
    my $self  = {
    _dbname => $arg{dbname},
```

[5] http://www.cpan.org/ — Perl モジュールを総合的に取り扱っている機関。検索からインストールまでほぼ全自動でできるようなインストーラーが熱い。

```
    _dbuser => $arg{dbuser},
    _dbpass => $arg{dbpass},
    @_,
    };
    bless $self, $class;

    $self->{_dbh} = $self->_connect_DB();
    $self->{_dbe} = $DBI::errstr;
    return $self;
}

# アクセサー
sub dbh # データベースハンドラ保持用
{
    my $self = shift;
    return $self->{_dbh};
}
sub dbe # エラーメッセージ読み書き用
{
    my $self = shift;
    my $new  = $_[0];
    $self->{_dbe} = $_[0] if defined $_[0];
    return $self->{_dbe};
}

# 内部メソッド。Postgres に接続後、接続ハンドラを返す
sub _connect_DB
{
    my $self = shift;

    my $dsn = "dbi:Pg:dbname=$self->{_dbname}";
    my $hDb = DBI->connect($dsn, $self->{_dbuser}, $self->{_dbpass},
   { RaiseError => 1, AutoCommit => 0})
or die $DBI::errstr;
    $hDb->commit;
    $hDb;
}

# Postgres から切断するときはこれ、半分アクセサーみたいなもの
sub disconnect
{
    my $self = shift;
    $self->dbh->disconnect;
    $self;
}

# 汎用 SQL 実行関数。実行結果シンボル $PgWorks::dbr を作成
sub pg_exec
{
    my $self = shift;
    my $sql = shift;

    $self->dbe(0); # エラー消去
    $self->{_dbr} = $self->dbh->prepare($sql);
    $self->dbr->execute();
    $self->dbe($DBI::errstr);

    $self;
```

```
}

# アクセサー
sub dbr # データベースリクエストの結果リソース ID 保持用
{
    my $self = shift;
    return $self->{_dbr};
}

1;
```

以上！これを作っておくだけで，以下のようなメソッドの使い方ができます．まず，このモジュールを使う宣言をしなければなりません．実行スクリプトのソース中で，

```
use strict;   # ←超基本
use PgWorks;  # ←ここで PgWorks.pm モジュールを使う宣言をします
  .
  .
  .
# 新しいオブジェクトを作ってあげます
# このオブジェクトを作った瞬間に SQL サーバに接続してくれます
# PgWorks ならでは！
my $db = new PgWorks(
     dbname => 'test',
     dbuser => 'httpd',
     dbpass => 'gomi01'
     );
```

という$db オブジェクトを PgWorks クラスとして作ってあげると，あとは 2，3 種類くらいしかないオブジェクトメソッドを使ってあげるだけで (ほとんど) なんでもできてしまいます．もっとトランザクション管理[6]とかして遊びたい人にはものたりないかもしれないですが，掲示板や集計サイトやカウンターくらいに使う程度なら十分すぎるくらいです (なぜならば，PostgreSQL ではデフォルトで AutoCommit[自動実行] を行わないからである．ちなみに PgWorks ではその記述を明記しているため操作一つ一つがトランザクションで守られた形となっている)．PgWorks で使えるメソッドを表 1.2 にまとめました．

では早速，以上のメソッドを使って，超簡単カウンターを作ってみましょう．

chou-counter.pl

```
#!/usr/bin/perl

use strict;
use PgWorks;
```

[6] トランザクション管理 – じつは DBMS の重要な機能の一つがこのトランザクションである．トランザクションとは，とある (一連の) 操作が完全に完結するか，または全く実行されなかったものとするかのブロックの役目をします．データベースに局所的にあるタイミングでロックをかけたりといった使い方をします．

表 1.2: PgWorks のメソッド一覧

メソッド	説明
pg_exec($sql)	$sql の SQL 文を即実行してくれます
dbh->commit	実行した SQL 文をデータベースに反映 (更新，追加，削除時)
dbr->"DBI メソッド"	実行後の読み取りに使う DBI メソッドがそのまま使えます
dbe	エラーが生じた時にエラーメッセージを返してくれます
disconnect	SQL サーバから切断します

```
my $db = new PgWorks(
                     dbname => 'test',
                     dbuser => 'httpd',
                     dbpass => 'gomi01'
                     );

# まずは現在カウントを読み込み
my $sql = qq|SELECT MAX(count), host(ipaddr) FROM choucounter LIMIT 1;|;

$db->pg_exec($sql); # 読み込み実行
my @ref = $db->dbr->fetcharray(); # SELECT 文の結果を配列として格納

$ref[0]++; # カウント UP！

print($ref[0]); # 表示

# 次、新しくアクセスしてきた人を登録
my $sql = qq|INSERT INTO choucounter(count, date, ipaddr, useragent)
    VALUES($ref[0], now(), "$ENV{'REMOTE_ADDR'}", "$ENV{'HTTP_USER_AGENT'}");|;

$db->pg_exec($sql); # SQL 文実行！
$db->dbh->commit;   # データベースに反映 (トランザクション解除)

exit;
```

以上がそのソースです。本当は俗なウェッブカウンターに使われてる無駄な flock とか多様しまっくてる香具師みたいに最初の SELECT 文のところから、SQL チックにトランザクション文（BEGIN[7]とか）入れて SELECT する前にロック掛けたいところですが、今回は超簡単なってのが趣旨ですので省略です。

上記のスクリプトは SSI[8]とかで呼び出してくださいね。print 文の部分が出力になります。あとはコメントに書いておきました。今回はアクセスがあるたびに上記スクリプトが SSI から呼び出されて、実行され、SQL サーバから現在の最高カウント値を SELECT 文で読み出してきてそれをカウント UP して表示、つぎに現在のアクセス情報を新しく INSERT 文で挿入します。これが一連

[7]トランザクションのロックは Perl の flock みたいにロックが漏れたりしないので結構信用できる

[8]SSI … Server Side Include。HTML 中に命令文を書くと、HTML を呼び出すときに Web サーバがその命令を実行して結果をその箇所に挿入してくれる機能。ファイル拡張子が .shtml とかになってるのはたいていそれ。

の処理です。

1.5 長くなったのでそろそろまとめをw

1.5.1 結局のところ SQL 使うと何が嬉しいの？

超簡単 Web カウンターを動かしてみるとなんら普通のそこらじゅうに出回ってるそいつと何も違うことなく動作します。「SQL で動いてるんだぜ〜」と自慢したくなったら、「**Hogege SQL カウンター created by xxx all right reserved**」とかをちょっくりと活きがりのサイトみたいに Web サイトのどこかに書いておくといいでしょう。今回の応用で多量のデータベースから何かを整形して HTML に吐かせたいっていう CGI をいとも簡単に作ることができます。悩むところは SQL 文作るところだけ、なんですから。実際に掲示板やらチャットやらのプログラムを書いたことがある方ならわかるかもしれませんが、保存するデータ構造に悩んだことはありませんか？SQL なら複雑なものでもリレーショナル (とある属性名が他のテーブルに関連している) データベースを定義すれば瞬殺なんです。そしてデータベースの並べ替え、部分抜き出しも **SQL 文に書くだけ**で**プログラム上の処理はまったく不要**なんです。これが SQL プログラムのとてもうれしいところだと思います。先ほどにも前述したトランザクションのロックもプログラム上の処理が不要です。しかもシステムコール関数とか使うものよりバグが少なくほとんど完璧にロックしてくれます (完璧じゃないそんな SQL サーバはそれだけで存在価値がなくなるので)。

1.5.2 おわりに

私は Web 関連のスクリプトいじりからこういう世界に入ったもので、CGI 関連が非常に好きです。これからも CGI プログラムを書きたいですが、最近は `open close if else` しか使ってなくて見るにも耐えないソースコードをシェアウェアとして公開している方々が多い感じがします。シェアウェアだけは勘弁してほしい … です。私は何か作るときはなるべく新しい技術とか方法とか使って書いていこうみたいな、スローガンを掲げているので、いろいろ勉強できて、しかも CGI はそんなに難しいことはしないので楽しくプログラムに触れるほうかなと思っています。ただ、今回は Perl を使いましたが、Perl の Object Oriented なプログラムは記述面からプログラマを苦しめるだけのような気がしてきました。別の言語でもいろいろ模索したいと思っているこの日この頃です。

こうして、私の KITCC の 4 年目の Lime も終わりですが全部 CGI 関係の記事しか書いてなかったカモ … (笑)。今までこれらのプログラムを書くにあたって助言していただいた先輩、OB の方々、部員の方々にこの場をかりて感謝したいと思います。

参考文献にお勧めのリファレンスと SQL 文法の勉強サイトを書いておきます。

参考文献

[1] SQL　-ＳＱＬの基礎から応用までーリレーショナル型データベース (RDBMS) を操作する
http://www.techscore.com/tech/sql/

[2] 鈴木啓修 著 「Advanced Reference Postgre SQL 全機能リファレンス」 技術評論社

[3] Object-Oriented Programming with Perl
http://perl-oop.hp.infoseek.co.jp/

2 jailの構築

01230704 電子情報工学科 3 回生 池野 直樹

2.1 はじめに

この稿では FreeBSD の jail 機能を利用したバーチャルサーバの構築に関して、それが度の程度のものなのか調べるとともに、構築運用に関して一通りのやりかた等を解説することを目的に筆を進めてみます。 なおここで用いる環境は FreeBSD 5.1-RELEASE で、メールサーバ、web サーバ等に関してはある程度の知識と構築の経験があることを前提にしておきます。（そこまで解説する余力があるとは思えない...）

また jail を構築する環境を「ホスト環境」、jail の環境を「jail 環境」として、jail 環境は /usr/jail/jail1 、/usr/jail/jail2 の二つ作ることにする。 またホスト名は jail1.test と jail2.test でローカルの DNS に一通りの設定をしておく。

2.2 jail の構築

まずは jail をインストールするディレクトリを作る

```
$ mkdir /usr/jail1
```

/usr/src に移動して

```
$ make world DESTDIR=/usr/jail/jail
```

(まえもって make buildworld してあれば、

```
$ make installworld DESTDIR=/usr/jail/jail
```

でもよい)

```
$ cd /usr/src/etc
$ make distribution DESTDIR=/usr/jail/jail
```

を行なう。
/usr/jail1 以下には 157M のファイルができる。

ここで /usr/jail/jail を /usr/jail/jail1 と/usr/jail/jail2 にコピーしておく。
/usr/jail/jail を最初に作り、それをコピーすることで複数の jail 環境を楽に準備できる。

jail 環境が用意できれば、それを順次設定していくことになる。

```
$ mount -t procfs proc /usr/jail/jail1/proc
$ jail /usr/jail/jail/jail1 jailhost1 192.168.0.80 /bin/sh
```

で jail で起動する。
sh のみが表示されるので、環境を整える。

- 空の /etc/fstab を作成
- /etc/rc.conf に

```
rpcbind_enable="NO"
network_interfaces=""
inetd_enable="YES"
inetd_flags="-wW -a 192.168.0.80"
sshd_enable="YES"
```

などを追加。

- newaliases を実行しておく
- /etc/resov.conf を設定

```
search  mydomain.com
nameserver 192.168.0.1
```

- root のパスワードの設定
- タイムゾーンの設定
- jail 環境のユーザアカウントの追加

(● 必要なら /etc/group の編集とユーザのホームディレクトリの作成)

- jail に必要なパッケージの追加

(● 必要なら syslog.conf の設定)

(● FreeBSD 4.x の場合は /bin/sh /dev/MAKEDEV でデバイスファイルを作成しておく)

shell から抜けることで jail をシャットダウンすることになる。
同様にして、jail2 の環境も整える。
環境構築が終れば jail の起動

```
$ ifconfig xl0 inet alias 192.168.0.80/32
$ mount_devfs devfs /usr/jail/jail1/dev
$ mount -t procfs proc /usr/jail/jail1/proc
$ jail /usr/jail/jail1 jailhost 192.168.0.80 /bin/sh /etc/rc
```

で jail 環境が通常起動するはず。jail を終了する時には

```
$ kill -TERM -1
```

で終了させる。

これで jail 環境を起動することができるようになった。 jail を自動的に起動させたいのなら、ホスト環境の /etc/rc.conf に次のような感じで書いて

```
ifconfig_xl0_alias0="inet 192.168.0.80
   netmask 255.255.255.255 broadcast 192.168.0.255"
ifconfig_xl0_alias1="inet 192.168.0.81
   netmask 255.255.255.255 broadcast 192.168.0.255"
jail_hosts0="/usr/jail/jail1 jailhost1.test
   192.168.0.80 /bin/sh /etc/rc"
jail_hosts1="/usr/jail/jail2 jailhost2.test
   192.168.0.81 /bin/sh /etc/rc"
```

/usr/local/etc/rc.d/jail.sh として以下をおいておく

```
#!/bin/sh

if [ -r /etc/defaults/rc.conf ]; then
        . /etc/defaults/rc.conf
        source_rc_confs
elif [ -r /etc/rc.conf ]; then
        . /etc/rc.conf
fi

if [ -z "${jail_prog}" ]; then
jail_prog=/usr/sbin/jail
fi

processjail()
{
/bin/ps aux | /usr/bin/perl \
-ane 'print $F[1],"\n" if($F[7] =~ /J/);'
}

stopjail()
{
        jailp=`processjail`
        if [ -n "${jailp}" ]; then
                echo kill -TERM $jailp
                kill -TERM $jailp
        fi
        sleep 1;
        jailp=`processjail`
        if [ -n "${jailp}" ]; then
                echo kill -KILL $jailp
```

```
                kill -KILL $jailp
        fi
}

case "$1" in
start)
echo 'jail: Starting'
stopjail
#
        jailc=0
        while : ; do
eval jail_args=\$jail_hosts${jailc}
if [ -n "${jail_args}" ]; then

jail_dir=`echo $jail_args | awk '{print $1}'`
jail_host=`echo $jail_args | awk '{print $2}'`

echo "jail: Starting host : $jail_host"

if [ -d "${jail_dir}/proc" ]; then

# mount /proc
if [ ! -d "${jail_dir}/proc/curproc" ]; then
mount -t procfs proc "${jail_dir}/proc"
fi

#mount /dev
if [ ! -d "${jail_dir}/dev/fd" ]; then
mount_devfs devfs "${jail_dir}/dev"
fi

# start
$jail_prog ${jail_args}
else
echo "jail: Error: $jail_host"
fi

jailc=`expr ${jailc} + 1`
else
break;
fi
done
echo 'jail: Finish'
;;

stop)
echo 'Stopping jail hosts'
stopjail

;;

*)
echo "Usage: `basename $0` {start|stop}" >&2
        exit 64
        ;;
esac
exit 0
```

これで jail がホスト環境の起動時に実行されてるようになる。

2.3 管理

jail 環境も必要に応じてバージョンアップしていく必要があるわけで、

```
$ cd /usr/src
$ make buildworld
$ make installworld DESTDIR=/usr/jail/jail1
$ mergemaster -D /usr/jail/jail1
```

とやっていく。 複数 jail 環境がある場合は最後の 2 つのコマンドを jail 環境毎に実行していく。

ちなみにホスト環境はマルチユーザでも問題ないが、jail 環境は停止した状態で上記を実行する必要がある。

2.4 総括

ホスト環境、jail 環境 1、jail 環境 2 のそれぞれに httpd サーバ、 smtp サーバ等をインストールしてためしてみたところ、jail がきちんと IP フォワード？してくれているらしく、ホスト名を指定して、個別にアクセスすることができた。smtp-source を使用してホスト環境と jail 環境のメール送信時の CPU 負荷を測定してみたところ、

ホスト環境 44.9%

jail 環境 83.2%

となった。

jail によるバーチャルサーバはかなり CPU 負荷が高くなることがわかる。 また jail でバーチャルサーバを作る時には IP アドレスが必須になるので、IP アドレスが潤沢にある環境でもなければ、IP アドレスにかかるコストも無視できなくなる。 こう考えていくと、jail によるバーチャルサーバはバーチャルサーバ使用者に root 権限を渡す必要があるというのでもなければ、あまり使い勝手が良いとはいえないという結論に達した。

3 画像圧縮について

03230114 電子情報工学科 1 回生 若松 健

3.1 はじめに

今，インターネット上には多種多様の Web ページが存在し，その Web ページにはたくさんの画像が使われています．その画像の殆どは JPEG[1]や GIF[2]がよく使われています．以下この 2 つについて詳しくみていくことにしませう．

3.2 GIF について

GIF は米国の大手パソコン通信サービス会社だった CompuServer 社が，権利上自由な画像形式として開発し公開した画像圧縮技術です．インターネットの急速な普及に伴い，パソコン通信サービスは淘汰され，1997 年に CompuServe 社は AOL[3]社に吸収されることになりましたが，GIF 形式は現在でもインターネットの Web ページなどで広く利用されています．

3.2.1 仕様

GIF ファイルの仕様は，GIF87 と，その後一部の仕様を改良した GIF89 があります．これら GIF87 および GIF89 フォーマットは，いずれも仕様書[4] がインターネット上で公開されています．

GIF フォーマットでは，画像用のパレット データ（最大 256 色）をファイル内部に持ち，このパレットの色を使って画像を表現しています．したがって本来は 256 種類以上の色のピクセルを含む画像を GIF フォーマットで保存するには，減色処理を行わなければなりません．この減色処理の方法として，256 色パレットにある近い色をもって代用する方法と、ディザ法[5]を使う方法の 2 つがあります．また，アニメーションや透明色のサポートもされています．GIF フォーマットでは，LZW 法[6]（3.4.1 参照）によってファイルに格納するデータを圧縮保存します．

---

[1] Joint Photographic Experts Group の略.「ジェイペグ」と読む.
[2] Graphics Interchange Format の略.「ジフ」と読む.
[3] America OnLine.
[4] http://www.geocities.co.jp/SiliconValley/3453/gif_info/index_jp.html
[5] 1 つのピクセルではなく，複数の色のピクセルを組み合わせることによって，擬似的に異なる色を表現する方法.
[6] 法律ではありませぬ.

3.2.2 問題の浮上

1990 年代前半，GIF の圧縮に LZW 法圧縮伸長技術が使われていることが周知のこととなり，この圧縮伸長技術の特許をもつ米 UNISYS 社が，特許使用料請求の交渉を始めたのです．LZW に関する動きを表 3.1 に示しておきます．

UNISYS 社の LZW 法の特許については，サブマリン特許という解釈をしている人が多いようです．しかし，この GIF 形式等に含まれる LZW 法の特許使用料請求問題は，サブマリン特許ではありません．UNISYS 社が特許を取得したのが，CompuServe への交渉を開始した時期以前のようだからです．

この時期に GIF の代替となる受け皿として，パテント[7]フリーの PNG[8]なるものが開発されました．PNG フォーマットは，W3C において標準化が進められており，1996 年 10 月には W3C[9]推奨フォーマットとなりました．GIF とは異なり，当初は PNG を標準サポートする Web ブラウザがあまりなかったため，なかなか普及は進みませんでしたが，Netscape Navigator 4 以降，Internet Explorer 4 以降で標準サポートされるようになり，徐々に普及し始めました．

ちなみに携帯電話では，au と Vodafone（J フォン）は PNG を，DOCOMO は GIF を標準のフォーマットにしているようです．もう一つちなみに，米国では今年 6 月 20 日に LZW 法の特許期限が切れたようです．来年には，カナダ，日本，英国，ドイツ，フランス，イタリア各国で期限が切れるようです．

年	出来事
1983	LZW 法の米国特許出願
1984	LZW 法の日本国特許出願
1993	LZW 法の日本国特許成立
1993	UNISYS 社から CompuServe 社への交渉開始
1994	CompuServe 社からの特許使用料支払いにより和解 GIF89a として規格書に “UNISYS 社が LZW 法の 特許を持つこと” などの項目を追加
1999	UNISYS 社が 1993 年時点では請求対象としていなかった “無償のソフトウェアの開発者” への請求を開始 この時の騒ぎの大きさは，1993 年の事件に気付かなかった 人も騒ぎ出したほど

表 3.1: LZW に関する動き

[7]特許，特許権．
[8]Portable Network Graphics の略．「ピング」と読む．
仕様書：http://www.w3.org/TR/2003/PR-PNG-20030520/
[9]World Wide Web Consortium．WWW で利用される技術の標準化をすすめる団体．

3.3 JPEG について

JPEG は，画像符号化の標準化を行なう事を目的として設立された組織の名で，ISO[10] と ITU-T[11](旧 CCITT[12]) の共同組織．画像ファイルとして扱う手法を “JFIF（JPEG File Interchange Format）” と呼ぶ. が，一般には JFIF のことを JPEG という組織名で呼ぶことが多い.

3.3.1 仕様，特徴

JPEG は，主にフルカラー写真画像を対象としたフォーマットであり，前段に DCT(離散コサイン変換)（3.4.2 参照）と呼ばれる量子化を基本とした非可逆な圧縮を採用し，後段に Huffman 符号（3.4.3 参照）または算術圧縮（3.4.4 参照）を利用しています．また，DCT するとき，画像を８×８の大きさの画素に分割し，DCT により，各画素ブロックの輝度成分を DCT 係数に変換 します. 圧縮率は可変ですが，圧縮率を上げ過ぎると (特に色の差の激しい境界部分で) 独特のブロックノイズが発生しやすく，そのために，写真などの圧縮には向きますが，線画には不適です.

なお，JPEG の可逆圧縮 (ロスレス JPEG) や JPEG2000 というものも存在します．しかし，ロスレス JPEG は，圧縮率と普及率の低さから殆ど使われていません．また，JPEG2000 は，JPEG により作られた画像圧縮を改良し，新たな画像符号化の標準化を行なう事を目的として設立された協議会の名であり，ここで作られた技術自体も JPEG に倣い JPEG2000 と呼びます．ここでは，ウェーブレット変換を用いた圧縮率の向上等を実現しています．しかし，フォーマットにパテントの問題があり，なかなか普及しないようです.

国際標準でかつフリーなフォーマットとして愛用されてきましたが, 米国 Forgent Networks 社が 2002 年 7 月，突然画像圧縮の特許に関するライセンス料を要求しだし業界を (GIF に続きまたも) 騒然とさせました．特許のポイントは，周囲と平均を取ってデータを削減するという点です.

3.4 圧縮法

3.4.1 LZW (Lempel Ziv Welch)

旧 DEC 社[13]の Terry A.Welch が 1984 年[14]に LZ78 符号法を改良して考案した圧縮アルゴリズムで，スライド辞書法と呼ばれる種類のアルゴリズムです.

具体的には，入力情報のあるデータ列に対して付番した “辞書” を作成しておき，再度同じデータ列が出現した場合は，その番号を符号として用います．データ列の照合は最も長いものを優先させる最長一致法と呼ばれる方法を用いています．LZ78 との大きな違いは，LZ78 は符号化されたものの中に必ず元の入力データ列そのものが含まれるが，LZW は 1 文字で構成されるデータ列を前

[10] International Organization for Standardization．国際標準化機構.
[11] International Telecommunications Union - Telecommunications Standardization Sector.
国際電気通信連合 (ITU) の電気通信標準化部門.
[12] (仏)Comité Consultatif International Téléraphique et Téléphonique.
(英)International Telegraph and Telephone Consultative Committee．国際電信電話諮問委員会.
[13] Digital Equipment Corporation.
[14] 私の誕生年.

もって辞書に登録することで符号語中に入力データ列が含まれないよう工夫し，常に辞書の番号のみで符号化できるようにして効率を上げています.

符号化手順

入力文字列を “ABDAFABECDAE” とします.

1. まず（インデックス）0〜255 にはあらかじめ文字（キャラクタコード）をセット.
2. 未一致文字列になるまで 1 文字づつ入力．ここではまず “A” が入力されてこのコード 41h は 0〜255 の辞書の中にあるので，さらにもう 1 文字入力．これで入力文字列は “AB” となる．“AB” と一致する文字列は辞書にはないので “AB” を 256 番に登録.
3. 一致した文字列へのインデックスを圧縮文字として出力．今のところ最後に一致した文字は “A” だったのでインデックス 41h を出力.
4. 未一致だった文字（“B”）を次の入力文字として保存.

以下同様に 2 〜 4 を繰り返せば表 3.2 のような状態になります.
出力文字列は “ABDAF”,256,“EC”,258,“E” となります.

入力文字	辞書への登録状況	出力符号
“AB”	256 = “AB”	’A’(=41h)
“D”	257 = “BD”	’B’
“A”	258 = “DA”	’D’
“F”	259 = “AF”	’A’
“A”	260 = “FA”	’F’
“BE”	261 = “ABE”	256
“C”	262 = “EC”	’E’
“D”	263 = “CD”	’C’
“AE”	264 = “DAE”	258
EOF		’E’

表 3.2: LZW の圧縮状況

復号化手順

入力文字列を “ABCADF”,256,“A”,259,“E” とします.
復号は符号化のシミュレートをする事によって行われます.

1. 符号化のときと同じように０〜２５５へ文字コードを登録.

2. １文字ずつ入力して辞書にない文字列になるまで入力．今は，１文字入力すると “A” が入力されます．これはすでに辞書に登録済みなので，さらに１文字入力して “B” が得られます．“AB” は辞書にはないので，これを 256 番に登録．

3. 見つかった部分の文字（“A”）を出力．

4. 未一致だった部分の文字（“B”）を次の入力文字として保存．

以下同様に２～４を繰り返せば表 3.2 に似た状態（表 3.3）になって，復号が行えます．
出力文字列は “ABDAFABECDAE” となります．

入力符号	辞書への登録状況	出力文字
“AB”	256 = “AB”	“A”
“D”	257 = “BD”	“B”
“A”	258 = “DA”	“D”
“F”	259 = “AF”	“A”
“A”	260 = “FA”	“F”
“BE”	261 = “ABE”	“AB”
“C”	262 = “EC”	“E”
“D”	263 = “CD”	“C”
“AE”	264 = “DAE”	“DA”
EOF		“E”

表 3.3: LZW の解凍状況

3.4.2 DCT（Discrete Cosine Transform：離散コサイン変換）

入力ベクトルを x とした時の，離散コサイン変換は

$$X(k) = \sqrt{\frac{2}{N}} C(k) \sum_{n=0}^{N-1} x(n) \cos\left[\frac{(2n+1)k\pi}{2N}\right] \ , C(k) = \begin{cases} \frac{1}{2}(k=0) \\ 1(k \neq 0) \end{cases}, k = 0, 1, 2, ..., N-1$$

で定義されます．JPEG では２次元 DCT を使用します．例えば，ある８×８の画素ブロックの輝度成分を図 3.1 とすると，DCT 係数に変換したものは図 3.2 のようになります．JPEG では，この DCT 係数を量子化し、その後，ハフマン符合化をすることにより圧縮を行います．

62	65	65	68	72	84	42	16
48	57	60	60	66	66	51	24
45	54	58	60	65	60	48	26
37	48	53	56	53	48	40	33
58	50	52	64	60	51	42	31
64	58	64	72	75	74	49	50
82	72	65	76	76	72	53	39
79	71	60	67	55	58	48	40

図 3.1: 8×8 の画素ブロックの輝度成分

452	49	−60	34	−7	2	2	−8
−26	−9	−24	4	−19	−4	1	3
32	19	2	16	−8	0	3	−6
23	1	−9	11	5	−2	9	−3
−20	8	7	−4	1	2	0	−5
9	2	−6	3	−1	−4	2	−2
5	−4	−1	−1	0	−3	5	−2
4	10	−2	8	2	4	−3	1

図 3.2: DCT 係数

3.4.3 Huffman 符号

符号化手順

入力データを “abbbbccddd” とします．まず入力データ全ての文字のカウントをとって，それを出現確率とします．また，復号時にも必要となるため，先に各文字の出現率を出力しておきます．すると表 3.4 のような結果となります．

ここからハフマン木の構築を始めます．まず，最小の文字 “a” と次に小さい文字 “c” を探し α という 1 つの接点にまとめます．まとめるには α という接点を作り，“a” と “c” へのポインタまたはインデックスを記憶し，“c” と “a” の出現率を足し合わせたものを α の出現率にします．そして，小さい方（リストの上）に bit0 を、大きい方に bit1 を割り当てます.（図 3.3）これが後で符号語となります.

今度は最小の文字 α と “d” をくっつけ，符号ビットを割り当てます.（図 3.4）

さらに，最小のデータ 2 つをくっつけます.（図 3.5）

ようやくデータが 1 つ（根）になったので，木の構築は終わりです．そして各記号の符号語は木

を根からたどることによって得られます.（表 3.5）

これらはちゃんと一意に複合可能な符号語になっています．あとは頻度テーブルを初めに保存して，入力された記号に応じて符号語を出力するだけです.

故に入力データに対して次のような変換が行えます.

“abbbbccddd” → 111b,0b,0b,0b,0b,110b,110b,10b,10b,10b

記号	a	b	c	d
出現率	$\frac{1}{10}$	$\frac{4}{10}$	$\frac{2}{10}$	$\frac{3}{10}$

表 3.4: 出現率

記号	a	b	c	d
符号語	111	0	110	10

表 3.5: 得られた符号語

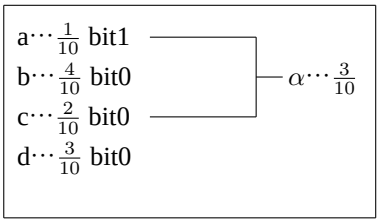

図 3.3:

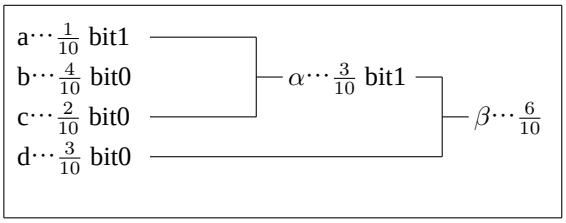

図 3.4:

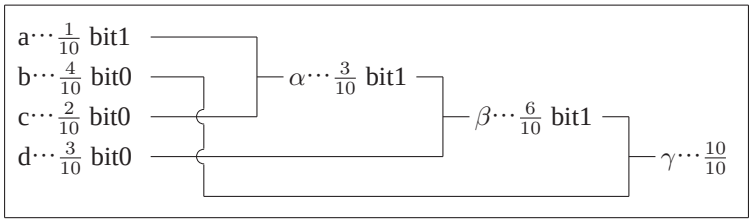

図 3.5:

復号化手順

復号時も同様に頻度テーブルから符号語を作り出し，入力ビットに応じて木を移動してたどり着いた文字を出力します．まずは各文字の頻度から，符号化のときと同じようにして木をつくり，符号語を得ます．すると，表 3.5 の符号語が得られます.

ここからデータの復号が始まり，まず 1 ビット読み込みます．すると，1b が得られるので図 3.5 より，β の方へ注目します．更に入力すると 0b が得られ，d に注目，このように元の記号にたどり着きます．同様に，111b が入力されたときは、$\gamma \to \beta \to \alpha \to$ a と注目すれば，復号出来ます．故に,

111b,0b,0b,0b,0b,110b,110b,10b,10b,10b → “abbbbccddd”

のように復元できます.

3.4.4 算術圧縮

符号化手順

入力文字列は “aadbbdcccc” とします.

まず Huffman 符号と同様にすべてのデータを先読みしてそれぞれのコードについて出現回数をカウントする．すると表 3.6 のような状態になります.

記号	生起確率	区間	区間幅
a	0.2	[0,0.2)	0.2
b	0.2	[0.2,0.4)	0.2
c	0.4	[0.4,0.8)	0.4
d	0.2	[0.8,1.0)	0.2

表 3.6: 各記号の生起確率と対応する区間

この出現回数表は復号の時にも必要になるため，出現確率を 8bit や 16bit などに丸め込んで，そのリストを出力しておきます．まず基本となる区間 [0,1) をとります．この場合区間幅は 1 - 0 = 1 となります．これを各記号の生起確率（a〜d の順）で分割しておき，入力文字列に対応する区間で繰り返し分割．つまり,

次の区間の下端 ＝（現在の区間の下端）＋（入力記号に対応する区間の下端）×（現在の区間幅）

次の区間の幅 ＝（現在の区間幅）×（入力記号の生起確率）

による分割をします．すると図 3.6，表 3.7 のように分割されます.

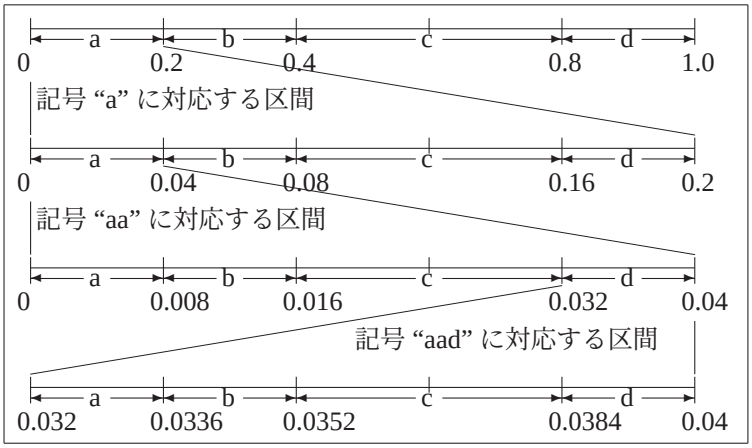

図 3.6: 区間の変化

以上のように分割を繰り返すと “aadbbdcccc” という文字列に対して [0.0342175744,0.0342192128) という区間が求まり，上下どちらを出力してもよいので，ここでの符

入力記号	区間の下限	区間の上限	区間幅
a	0	0.2	0.2
a	0	0.04	0.04
d	0.032	0.04	0.008
b	0.0336	0.0352	0.0016
b	0.03392	0.03424	0.00032
d	0.034176	0.03424	0.000064
c	0.0342016	0.0342272	0.0000256
c	0.03421184	0.03422208	0.00001024
c	0.034215936	0.034220032	0.000004096
c	0.0342175744	0.0342192128	0.0000016384

表 3.7: 符号化のようす

号語には区間の下限の 0.0342175744 が符号語として出力します.

復号化手順

復号は符号化のシュミレートをする事によって行われます.（単純な計算と比較だけで求まる）まず，先ほど出力した出現確率表を読込み，この表から最初の分割を同様にして作る.（この表は固定）そして，符号語の浮動小数点数 (0.0342175744) を読込み，その符号語がどの文字の分割域にあるか比較します．すると 0.0342175744 は 'a' の [0,0.2) にあることがわかるので 'a' を出力し，'a' の影響を符号語から除外します．具体的には,

$$\begin{aligned} \text{新しい符号語} &= \frac{\text{（符号語）} - \text{（'a' に対応する区間の下限）}}{\text{（'a' に対応する区間の幅）}} \\ &= \frac{0.0342175744 - 0}{0.2} \\ &= 0.171087872 \end{aligned}$$

が得られます．符号語 0.171087872 は 'a' の $[0, 0.2)$ にあることがわかるので 'a' を出力し，'a' の影響を符号語から除外.

$$\begin{aligned} \text{新しい符号語} &= \frac{\text{（符号語）} - \text{（'a' に対応する区間の下限）}}{\text{（'a' に対応する区間の幅）}} \\ &= \frac{0.171087872 - 0}{0.2} \\ &= 0.85543936 \end{aligned}$$

符号語 0.85543936 は 'd' の $[0.8, 1.0)$ にあることがわかるので 'd' を出力し，'d' の影響を符号語から除外.

$$\text{新しい符号語} = \frac{\text{（符号語）} - \text{（'d' に対応する区間の下限）}}{\text{（'d' に対応する区間の幅）}}$$

$$
\begin{aligned}
&= \frac{0.85543936 - 0.8}{0.2} \\
&= 0.2771968
\end{aligned}
$$

これらを繰り返せば文字列を復元することができます.

3.5 おわりに

普段何気なく使っている画像圧縮の仕組みがこのようなものだったとは，この原稿を書くときまで知りませんでした．これを読んで少しでも画像圧縮についてわかってもらえればと思います.

参考文献

[1] SAW's Junks 『GIF 形式とそれに含まれる LZW 法の問題』
http://www.zob.ne.jp/~saw/pc/gif.htm

[2] Furuichi's page 『圧縮法研究室』
http://www.ingnet.or.jp/~kojif/mu/index.htm

[3] 大魔界通信網 『通信用語の基礎知識プロジェクト』
http://www.wdic.org/

第Ⅴ部

趣味

パーソナルコンピュータの利用目的の１つに趣味があります。ここに掲載されているのは、コンピュータを趣味に利用したときの話です。

1 電車路線検索 ～快適な旅のために～

03240011 物質工学科 1 回生 臼木 由美子

1.1 初めに

Yahoo などの Web 検索サービスや、鉄道会社の WebSite[1]には乗換案内というサービスがあります。
出発地点と目的地を入力することで、その間の乗換駅や必要料金などを割り出してくれるものです。
中には、時刻表と連動しているものもあります。

今回はその乗換案内サービスの内から幾つかを抜き出し、何項目かの観点から使い易さを検証してみようと思います。
稚拙な文章ですが、どうかしばらくお付き合い下さい。

1.2 今回取り上げるサービス

Yahoo!路線情報
http://transit.yahoo.co.jp/
goo 路線
http://channel.goo.ne.jp/cgi-bin/tranavi/jrtrag.cgi
MSN 路線
http://transit.msn.co.jp/N1?an=-1
biztech 駅すぱあと
http://biznsold.nikkeibp.co.jp/cgi-bin/expwww/biztech_route/
えきねっと 時刻乗換案内
http://www.jnavi.eki-net.com/
ジョルダン 乗換案内
http://v203.jorudan.co.jp/norikae/cgi-bin/noricnt.cgi
らくらくおでかけネット
http://www.ecomo-rakuraku.jp/

[1] 日本では HomePage とも呼ばれる。

以上７つの WebSite を、今回は比較対象として取り上げます。

なお、対象としているのは全て無料のサービスです。
また、特定路線・特定地域のみのサービスは取り上げていません。

1.3 時刻表との連動

乗換案内サービスの中には、鉄道の時刻表と連動しているものがあります。
そのようなサービスでは、日にちと出発時刻・到着時刻から乗り継ぐべき電車を割り出すことができます。

時刻表と連動していないものに比べて効率的な乗換えが可能になる上、その場で時刻表を見る必要がなくなる便利な機能であると言えます。

今回調べた中では Yahoo!路線情報、MSN 路線、えきねっと 時刻乗換案内、ジョルダン 乗換案内の４つの WebSite がこの機能を持っていました。
中でも Yahoo!路線情報では出発・到着時刻のみならず、最終電車の検索も出来ました。

1.4 地図・出口情報との連動

先に挙げたものよりは数が少ないですが、検索結果から周辺地図や駅の出口周辺の情報を参照することのできるものもあります。
このようなサービスでは待ち合わせ場所の立案や目的地への道筋の下調べを路線の検索と同時に行うことが出来、よりスムーズに旅行、出張などの計画を立てることが出来ます。

今回調べた中では Yahoo!路線情報、goo 路線、えきねっと 時刻乗換案内、ジョルダン 乗換案内がこの機能を持っていました。
らくらくおでかけネットでは、地図を直接参照することは出来ないものの駅の WebSite が参照出来るようになっていました。
この条件が多様であるほど、立てるべき計画に応じた路線を素早く検索出来ることになります。

今回調べた中では、goo 路線、えきねっと 時刻乗換案内の２つの WebSite で空路、有料特急、新幹線の使用非使用を選択することが出来ました。
Yahoo!路線情報では、空路と新幹線以外の有料特急について選択可能でした。
また MSN 路線では、航空会社を選んで検索することが出来ました。

1.5 それ以外の機能

WebSite によっては、他のサービスに見られない独自の機能を持っていることがあります。
例えばらくらくおでかけネットは、交通機関のバリアフリーに関する情報を集めた WebSite というその特性から、駅のトイレ情報や車椅子での利用情報（リフト設備やスロープ、階段の有無）を参照する機能を持っています。
また、えきねっと 時刻乗換案内では、駅弁[2]の一覧という項目がありました。

他の WebSite と併用することで足りない情報を補強し、より役に立てることが出来るのではないでしょうか。

1.6 結論

時刻表や地図との連動、検索条件など一通りの機能を備え、駅構内情報や観光情報などの情報も豊富なのはえきねっと 時刻乗換案内でした。
Yahoo!路線情報は画面がコンパクトで見やすいのですが、大きな駅を経由する時は新幹線を利用した経路ばかりが検索結果として出てきてしまいました。

考慮できていない事柄も多くあり、とてもきちんとした検証とは言えないのですが今回は「単体で使いやすいのはえきねっと 時刻乗換案内」ということで、結論としたいと思います。

[2]駅の構内で売られている弁当。

編集後記

昨年は、後輩にやらせるようにしたいと書いた私でありましたが、結局今年も私が編集することになってしまいました。個人的に Lime の編集は嫌いでないというのと、TEX にどんどんのめりこんでしまった等いろいろな原因がありますが、なにより完成したときの達成感は忘れられないというのがあります。また、原稿がそろって編集しているとついつい記事を読んでしまっていて、編集そっちのけになることがあります。それはそれで有意義な時間を過ごすことができるわけで悪くはないと思っています。

しかし、やっぱりいつまでも私がやるわけにはいかないので、“来年こそは！” 後輩に任せたいと考えています。来年の Lime も御期待ください！

平成 15 年 10 月 20 日 編集担当 山本 大介

Lime Vol.28

平成 15 年 11 月 20 日発行 第 1 刷

発行 京都工芸繊維大学コンピュータ部
http://www.kitcc.org/