2014
秋号

百萬石
Hyakumangoku Vol.42

ドメインで遊ぼう！
Tachibana Waita

~~ヤマ~~iPXE ノススメ
imadahiroshi

pecoで遊ぶ
masawada

府中刑務所文化祭レポート
zico

電通大Objet紀行 2014
f_taka6

簡単なコンパイラの作成
nomeaning

ACM-ICPC アジア地区予選東京大会に参加したお話
darui1113

Published by MAMA

部長挨拶

中川涼太 (zico)
zico@mma.club.uec.ac.jp

『百萬石』をお手に取って頂き、ありがとうございます。電気通信大学 MMA(Microcomputer Making Association) は、1975 年に創立されたコンピュータに関する技術系サークルです。創立当時は名前の通りマイクロコンピュータのシステムを自作することから始まったサークルですが、現在はプログラミングに関する研究・開発や、ネットワーク・サーバ管理が主な活動となっています。IC カードを鍵として利用する電子錠システム “kagisys” の開発、部室のサーバやサークル棟のネットワーク管理、プログラミングコンテストへの参加、プログラミング言語の講習会の開催などがその例です。

弊学の学祭「調布祭」では毎年、弊部の部員や OB・OG などが持ち寄ったジャンクパーツなどを販売する「ぢゃんく市」を開店しています。半ば骨董的な商品や、意外な掘り出し物がみられます。そして今回はぢゃんく市に加えてもう 1 つイベントを開催します。Dentoo.LT と言います。

> Dentoo.LT is “Lightning Talks in The University of Electro-Communications.”
>
> LT（ライトニングトーク）とは、５分程度で行われる簡易なプレゼンテーションのことです。
> 発表テーマは自由。おいしい麦茶の入れ方からちょっとマニアックな Eject コマンドの使い方まで何でも発表してください。
> 学外の方の参加や発表もお待ちしております。
>
> Dentoo.LT：ATND
> https://atnd.org/events/32172

Dentoo.LT は弊部員である masawada の主催で 2012 年 10 月に開かれてから不定期に開催が続いています。今や部をあげたイベントとなり、多くの方々にご参加頂いております。Dentoo.LT の特徴は、LT の内容のジャンルを一切問わないことです。登壇、観覧にも学内外・学生・社会人を問わずどなたでもご参加いただけます。今年は大学本部からのサポートもあり、7 月のオープンキャンパスや、今回の調布祭での開催に至りました。“Dentoo.LT #8.5” と題して、実質的に第 9 回目の開催となります。

本誌にも Dentoo.LT に負けず劣らずのバリエーション豊かな記事が集まっています。どうぞお楽しみください。

2014 年 11 月 21 日

目次

第 1 章　ドメインで遊ぼう！ Tachibana Waita 1
1.1 ドメインとは 1
1.2 ドメインの読み方 1
1.3 ドメインで遊ぶ 2

第 2 章　ヤマ iPXE ノススメ hiro1357 / @eserver_dip_jp ＞ワ＜ 4
一合目　PXE って使った事ありますか？ 4
二合目　では、iPXE とは？ 5
三合目　どうやったら使えるの？ 5
四合目　iPXE を触ってみる 6
五合目　インターネット上にあるサーバーから CentOS をインストールしてみる 8
六合目　スクリプト化してみる 9
七合目　スクリプトを自動生成してみる 9
八合目　SAN につないでみる 9
九合目　スクリプトを埋め込んでみる 10
頂上　ROM に焼いてみる 10
まとめ 10
参考文献 11

第 3 章　peco で遊ぼう masawada 12
3.1 概要 12
3.2 便利なスクリプトたち 13

第 4 章　府中刑務所文化祭レポート 中川 涼太 (zico) 14
4.1 府中刑務所文化祭とは 14
4.2 プリズンアドベンチャーツアー（所内見学） 16
4.3 庁舎 17
4.4 刑務所作業製品の販売 18
4.5 パネル展示 20

4.6 ライブステージ 21
4.7 おわりに 22
参考文献 22

第 5 章 電通大 Objet 紀行 2014 f_taka6 23
5.1 はじめに 23
5.2 事前調査 23
5.3 実地調査 25
5.4 まとめ 27
5.5 おわりに 37
参考文献 38

第 6 章 コンパイラの作成 nomeaning 40
6.1 どんな言語を作るか 40
6.2 コンパイルの流れ 41
6.3 スキャン (字句解析) 41
6.4 パース (構文解析) 43
6.5 アセンブリの生成 45
6.6 アセンブルと実行ファイルの作成 55
6.7 いろいろ動かしてみる 55
6.8 終わりに 57

第 7 章 ACM-ICPC アジア地区予選東京大会に参加したお話 darui1113 58
7.1 はじめに 58
7.2 チームに加わるまで 58
7.3 チーム結成〜国内予選 59
7.4 東京大会まで 59
7.5 東京大会 59
7.6 今後 60
7.7 終わりに 60
参考文献 61

第 1 章

ドメインで遊ぼう！

Tachibana Waita

概要
昨今、「ドメイン」を知らない若者が増えている。ドメインとは一部の人間にとって非常に有用な物であり、その可能性は計り知れない。そんなドメインの導入から簡単な利用までを本章では追っていきたい。

1.1 ドメインとは

ドメインとは、「IP ネットワークにおいて個々のコンピュータを識別する名称の一部」である。...などと小難しい事を言っても恐らく本記事の想定する読者に対しては何を言っているかさっぱり分からないと思うので、例を挙げてみる。

- google.com
- uec.ac.jp
- dentoo.lt

これらが「ドメイン」と呼ばれる物である。ドメインとは IP アドレスと呼ばれるコンピュータを識別する為の数字群と紐付けて利用する物であり、これを使えば、わざわざ IP アドレスなどという覚えづらい数字を覚える必要はない。昨今ようやく現実的な利用が行われてきている IPv6 アドレスを正確に覚えきれる人間は稀であるだろう。

1.2 ドメインの読み方

基本的にドメインはドットを使って区切られており、末端の方が大枠として捉えれば問題はない。例えば「uec.ac.jp」なら、「日本」の「学術的」な「uec」という組織である事を示している。ここから更に「サブドメイン」なる概念を加えることで、ドメインの利用が広がる。
弊サークル MMA のウェブサイトは、「www.mma.club.uec.ac.jp」である。このドメインを読むことで、「日本の大学の電気通信大学のサークルの MMA」という意味合いに取ることが出来る。

ここで「この会社は良い」「彼処はダメだ、信用できねぇ」とかを述べるとサークルの皆様に迷惑がかかってしまうかもしれないので、言及は避ける。詳細に聞きたい方は直接筆者に聞くか、筆者のブログに

書いてある気がする。
無料でドメインを取得したい方々に対して、サブドメインを無償で出しているサービスがある。

- dot.tk
- dip.jp

等が有名だろうか。この辺りのドメインを取得すれば良いと思う。

1.3　ドメインで遊ぶ

前章を読めば、一応ドメインを取得する事は可能であるだろう。ここから遊んでいく事が可能になる。筆者が持っているドメインを使って実際に遊んでみよう。この文章は 2014 年に生成された物で、実際にアクセス出来るかは分からないので注意して頂きたい。

筆者が所有しているドメインに「susibo.com」というドメインが存在する。このドメインと紐付いている IP アドレスを変更したりして遊ぶ、というくだらない遊びを本記事では取り扱う。

最初は筆者のハンドルネームである「whywaita」を付けてみよう。つまり「whywaita.susibo.com」という訳である。
ドメインから紐付いている IP アドレスを取得する事を「名前解決を行う」と呼ぶ。この時、DNS と呼ばれるサービスが動いた権威サーバに問い合わせを行っている。(詳細は各自で検索して欲しい)
ドメインを取得した場所からネームサーバを指定して DNS 設定を記入しよう。この辺りに関しては詳細に記事を書く気が無いので、Google 先生などに聞けば良いのでは無いだろうか。Google 先生のドメインは「google.com」である。「google.co.jp」というドメインも所有しているが。(余談ではあるが、Google もドメイン管理サービスを始める気らしい)

この後、ネームサーバによって指定したサーバで Web サーバを立ち上げて、バーチャルドメインの設定などを行う。またルート部分に、「.htaccess」というファイルを置いて、以下のようなおまじないを書いてみる。

```
RedirectMatch 302 .* http://twitter.com/whywaita
```

これで http://whywaita.susibo.com にアクセスすれば、ちゃんと筆者の Twitter にリダイレクトされる。良い事である。

ドメインは、色々な文字列を取得する事が出来る。更に取得後は、自由な文字列でサブドメインを設定出来る。
これらを組み合わせれば、非常にメッセージ性を持たせた URL を作る事が可能である。貴方も是非メッセージを持たせてみては如何だろうか？

1.3.1　余談

最後に、筆者の持っているドメインの中で公開されている物を列挙してみる。ちなみに、ドメイン自体は 1 つ持っていればサブドメインを作成すれば実用性に取っては十分なので、完全に趣味の領域である

事を付け加えておく。

- uec.tokyo
- littletrial.info
- painfuldance.asia
- susibo.com
- uec.xyz
- uectokyo.org
- whywaita.info
- whywrite.it
- whywriteit.org

少し前までもっとドメインを持っていたのだが、維持費の観点から幾つか放出した。放出したドメインは基本的に買収業者に取得されるので再取得は絶望的だろう。その辺は皆様の裁量で取り扱って頂きたい。

第 2 章

~~ヤマ~~ iPXE ノススメ

hiro1357 / @eserver_dip_jp >ワ<

概要
みなさん！ iPXE って知ってますか！？今回は「ヤマノススメ」・・・じゃなくて「iPXE のススメ」というタイトルで、皆さんと一緒に iPXE に登ってみたいと思います！

一合目　PXE って使った事ありますか？

iPXE を知るには、まずそもそも PXE というのが何なのかを知ってないといけません。というわけで、PXE とはなんぞや？という話から始めたいと思います。

PXE というのは、オペレーティングシステム (OS) をネットワーク経由で運び、起動させるための仕組みです。Intel 社が策定した規格のようです。
(`http://ja.wikipedia.org/wiki/Preboot_Execution_Environment`)
どのような仕組みかを簡単に説明するとこのような感じです。

1. PC の電源を投入
2. ネットワークカード内にあるファームウェアが読みこまれます。
3. BIOS の POST 後、読みこまれたファームウェアが動作します。
4. ファームウェアはまず、DHCP によりネットワーク設定の取得を試みます。
5. DHCP サーバはネットワークカードからの要求により、ネットワーク設定と、tftp サーバ情報、ブートストラップローダのファイル名を送信します。
6. 次に、受信した情報をもとに、tftp サーバからブートストラップローダの取得を試みます。
7. ブートストラップローダの受信が完了したら、受信したブートストラップローダの実行を試みます。
8. 実行されたブートストラップローダは、その他の、起動に関するファイルをメモリ上に読みこんだり、NFS や iSCSI などを使用してオペレーティングシステムを起動します。

この PXE という仕組みには、それによって起動する端末にハードディスクや SSD などを必要としないという特徴があり、端末自体に機能を持たないシンクライアントやオペレーティングシステムのインストール作業などの用途で使用されています。

二合目　では、iPXE とは？

それだけでも便利な PXE ですが、いくつかの制約があります。たとえば、通信プロトコルに TFTP しか使えないことや、条件によって読み込むファイルを変更するなどの細かな設定が難しいことなどがあげられます。

iPXE とは、そのような問題点を解消した、新しいネットワークブートのためのシステムです。

三合目　どうやったら使えるの？

まずは、iPXE を入手しましょう。

`http://ipxe.org/start`

公式サイトでは、git でソースコードを入手し、コンパイルする方法が紹介されていますが、今回はジェネレータがあるのでそれを使うことにしましょう。

`https://rom-o-matic.eu/`

最初のラジオボタンは ``Standard, for most common use''、次の Choose an output format:は ``UNDI only(.kpxe)'' を選択し、後は何も変更せずに ``Proceed'' をクリックしてください。undionly.kpxe というバイナリがダウンロードできるはずです。

iPXE 自体はネットワークカードなどのデバイスの ROM に書き込んで使用するファームウェアですが、通常の PXE を経由してロードすることも可能です。(kpxe は iPXE を PXE でロード可能なようにコンパイルしたものです。)

まずは、DHCP サーバが PXE に関連する情報を送信するように設定します。isc-dhcp を使用する場合は通常の設定に加えて次のような設定を追加してみてください。

```
next-server 192.168.0.2;
if exists user-class and option user-class = "iPXE" {
    filename "http://webserver/script-generator.php";
} else {
    filename "undionly.kpxe";
}
```

next-server は、次にアクセスする tftp サーバの IP アドレスを指定するためのオプションです。

isc-dhcp では、if 文が使用できます。これによって、iPXE からの要求なのか、そうでないのかを判別し、異なる応答を返すことができます。

filename には、tftp サーバからダウンロードするファイル名を指定します。

なぜこのような設定になるかというと、以下のような順にロードされるためです。

1. PXE により、DHCP の要求が送信されます。
2. iPXE がダウンロードされ、実行されます。
3. iPXE により、DHCP の要求が送信されます。
4. filename に指定されたブートストラップローダまたは iPXE スクリプトがダウンロードされ、実行されます。

iPXE スクリプトについては後述します。

四合目　iPXE を触ってみる

iPXE が実行されると、このような画面になります。

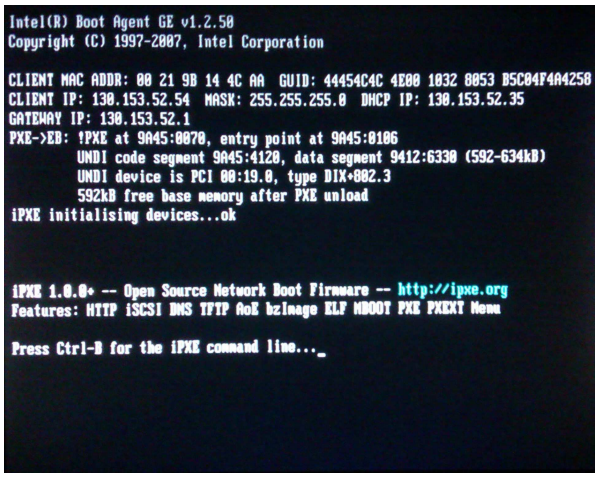


図 2.1　iPXE の画面

ここで、‘‘Press Ctrl-B for the iPXE command line...’’ と表示されたら、Ctrl と B を同時に押してください。このようなプロンプトが表示されます。

```
iPXE>
```

このプロンプトでは、iPXE が対応しているコマンドを実行することができます。
よく使われるコマンドの例としては、以下のようなものがあります。

コマンド名	説明
dhcp	dhcp の要求を送信し、ネットワーク設定を行います。 `dhcp`
chain	スクリプトまたは次のブートストラッププログラムをロードし、実行します。 `chain http://myserver/ipxe-script-generator.cgi`
imgfetch	起動に必要なイメージファイルやプログラムなどをダウンロードします。 `imgfetch http://myserver/initrd.img`
kernel	カーネルをロードします。 `kernel http://myserver/vmlinuz`
initrd	起動イメージをロードします。imgfetch の別名です。 `initrd http://myserver/initrd.img`
boot	スクリプトまたは次のブートストラッププログラムをロードし、実行します。chain の別名です。 `boot`
sanhook	SAN デバイスをアタッチします。 `sanhook iscsi:10.0.4.1:::1:iqn.2010-04.org.ipxe.dolphin:storage`
sanboot	SAN デバイスを使用して起動します。 `sanboot iscsi:10.0.4.1:::1:iqn.2010-04.org.ipxe.dolphin:storage`
reboot	コンピュータを再起動します `reboot`
poweroff	コンピュータの電源を切ります。 `poweroff`
ifstat	コンピュータのネットワークインタフェースを確認します。 `ifstat`
ping	疎通確認をします。 `ping machine.domain`
set	変数をセットします。 `set baseurl http://myserver/`
echo	文字列や変数を表示します。 `echo ${baseurl}`

その他のコマンドに関しては公式サイト (`http://ipxe.org/cmd`) で確認することができます。

五合目　インターネット上にあるサーバーから CentOS をインストールしてみる

コマンドを把握したところで、これを実際に使って、CentOS のインストーラを起動してみましょう。まずはじめに、DHCP サーバから IP アドレスをもらいましょう。

```
iPXE> dhcp
DHCP (net0 00:00:00:00:00:00)........ ok
iPXE>
```

これで IP アドレスが取得できました。
さっそく jaist から CentOS の起動イメージをダウンロードしましょう。

```
iPXE> set baseurl http://ftp.jaist.ac.jp/pub/Linux/CentOS/6/os/x86_64
iPXE> initrd ${baseurl}/images/pxeboot/initrd.img
http://ftp.jaist.ac.jp/pub/Linux/CentOS/6/os/x86_64/images/pxeboot/initrd.img.... ok
iPXE> kernel ${baseurl}/images/pxeboot/vmlinuz stage2=${baseurl}/images/install.img
http://ftp.jaist.ac.jp/pub/Linux/CentOS/6/os/x86_64/images/pxeboot/vmlinuz.... ok
iPXE>
```

kernel で指定されている stage2 は vmlinuz の引数です。
これで準備は整いました。起動してみましょう！

```
iPXE> boot
```

起動してくれました！

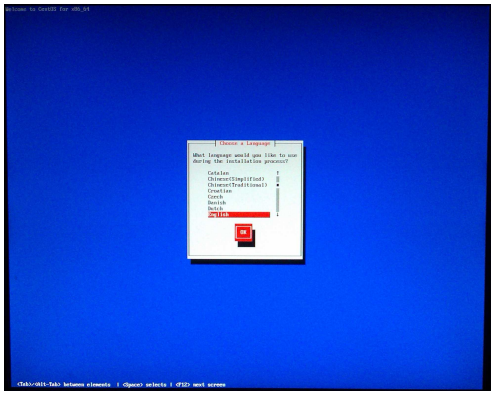


図 2.2　iPXE の画面

あとは、起動したインストーラにしたがって、いつも通りインストール作業をすすめるだけです。

六合目　スクリプト化してみる

いままで手打ちで作業してきましたが、これはスクリプトファイルにして自動化することができます。

スクリプト化は非常に簡単です。今まで入力してきた内容を並べるだけです。iPXE スクリプトの先頭には ‘‘`#!ipxe`’’ という shebang をつけます。

上の作業をスクリプト化するとこのようになります。

```
#!ipxe
dhcp
set baseurl http://ftp.jaist.ac.jp/pub/Linux/CentOS/6/os/x86_64
initrd ${baseurl}/images/pxeboot/initrd.img
kernel ${baseurl}/images/pxeboot/vmlinuz stage2=${baseurl}/images/install.img
boot
```

これは、どうやって呼び出すかというと、‘‘`chain`’’ を使います。
まず、スクリプトファイルを保存し、適当な web サーバにアップロードします。
そして、`chain` コマンドをこのように実行してみてください。

```
iPXE> chain http://myserver/script.ipxe
```

以降、スクリプトに書かれた内容が自動的に実行されます。

七合目　スクリプトを自動生成してみる

このスクリプトは http でロードできるため、スクリプトファイルの代わりにスクリプトを生成するサーバサイドのスクリプト（ややこしい）を使うこともできます。たとえば、IP アドレスごとに、異なる Linux カーネルをロードしたいとか、後述する SAN への接続先を変更したいといった場合でも対応することができます。

八合目　SAN につないでみる

iPXE は SAN(Storage Area Network) にも対応しているため、iSCSI などのネットワークストレージからの起動もすることができます。
接続方法は簡単です。

```
#!pxe
dhcp
sanboot iscsi:10.0.4.1::3260::iqn.2010-04.org.ipxe.dolphin:storage
```

sanboot コマンドを使用するだけです。sanboot コマンドには、iSCSI 特有の書式に従って接続先を指定します。

```
iscsi::[ip アドレス]::[ポート番号]::[イニシエータ名]:[ターゲット名]
```

sanhook を使えば、iSCSI ターゲットに Linux をネットワークインストールすることもできます。
CentOS をインストールしたい場合は、このようにします。

```
#!ipxe
dhcp
sanhook iscsi:10.0.4.1::3260::iqn.2010-04.org.ipxe.dolphin:storage
set baseurl http://ftp.jaist.ac.jp/pub/Linux/CentOS/6/os/x86_64
initrd ${baseurl}/images/pxeboot/initrd.img
kernel ${baseurl}/images/pxeboot/vmlinuz stage2=${baseurl}/images/install.img
boot
```

九合目　スクリプトを埋め込んでみる

iPXE は、コンパイル時に指定することにより、iPXE スクリプトを kpxe バイナリ内に埋め込むことができます。
ジェネレータを使うと、簡単に埋め込むことができます。
https://rom-o-matic.eu/

埋め込んだスクリプトは起動時に自動的に実行されます。

頂上　ROM に焼いてみる

じつはまだやったことが無いのですが、コンパイルした iPXE をインタフェースの ROM に直接焼いてしまうこともできるようです。
http://ipxe.org/howto/romburning

まとめ

ここまで読んでいただいた皆さん、おつかれさまでした。眺めはいかがでしょうか。

iPXE は高機能なので、いろいろな登り方があると思います。もし、ネットワークブートを使う機会があれば挑戦してみてはいかがでしょうか。

さて、もう少ししたら下ることにしましょう。ゆっくり温泉に入りたいですね。

参考文献

[1] iPXE - open source boot firmware [start]
http://ipxe.org/

[2] ROM-o-matic.eu | Generate iPXE images | open source network boot firmware
https://rom-o-matic.eu/

[3] Preboot Execution Environment - Wikipedia
http://ja.wikipedia.org/wiki/Preboot_Execution_Environment

第 3 章

peco で遊ぼう

masawada

3.1　概要

3.1.1　peco とはなにか

peco は入力をインタラクティブに検索、フィルタリングすることができる便利なツールだ。全て Golang で書かれているため、追加でライブラリ等をインストールする必要がなく多くのプラットフォーム上で動作する。peco 単体では全く使い道がないが、awk などと組み合わせることでとても強力なツールとなり得る。

3.1.2　peco のインストール

さっそく peco をインストールしてみよう。Golang の処理系をインストールし peco をビルドすることもできるが GitHub でバイナリが配布されている[*1]ため、これを`/usr/local/bin`などに配置することで簡単に使用することができる。

Mac OS X で Homebrew を使っている場合は以下のコマンドでインストールすることもできる。

ソースコード 3.1　Mac でのインストール

```
$ brew tap peco/peco
$ brew install peco
```

3.1.3　挙動を理解する

試しに以下のコマンドを発行してみよう。

ソースコード 3.2　Mac でのインストール

```
$ peco < /usr/include/stdio.h
```

発行したらカーソルキーで適当な行を選択して Enter キーを押そう。すると、選択した行が標準出力に表示されるはずだ (画像 3.1)。

[*1] `https://github.com/peco/peco/releases`

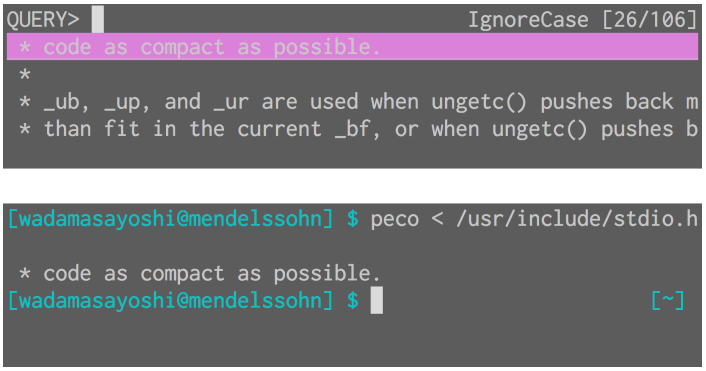


図 3.1　peco の挙動

3.2　便利なスクリプトたち

最後に、peco を使った便利なスクリプトを紹介する。

3.2.1　git-find

```
cd $(git rev-parse --show-toplevel)
DST_PATH=$(git grep -n $1 | grep -v "[0-9]:\s*//" | peco | awk -F ":" '{print "-c "$2" "$1}
    ');
if [ ${#DST_PATH} -ne 0 ]; then
  vim $DST_PATH;
fi
```

`git-find`は git リポジトリ内を検索し、Vim でその行を開く git サブコマンドだ。実際に検索するには `git find [word]`のように使う。行を開くためのオプションがエディタによって異なるため、Emacs で使いたい場合は多少の修正が必要となる。

3.2.2　pk

```
COMMAND=$1
PID=$(ps ax | grep $COMMAND | peco | awk '{print $1}')
if [ ${#PID} -ne 0 ]; then
  /bin/echo -n "Are you sure to kill process $[PID]? [y/n] "
  read ans

  if [ $ans = "y" ]; then
    kill -9 $PID
  fi
fi
```

よく ssh をつないだまま端末を落としてセッションが残ることがある。`pk`コマンドを用いることで、`ps ax | grep ssh`で PID を調べて kill せずとも `pk ssh`とするだけで ssh のプロセスを終了させることができる。

第 4 章

府中刑務所文化祭レポート

中川 涼太 (zico)

概要
受刑者を収容し、社会復帰のための更生を促す刑務所。府中刑務所文化祭では受刑者の生活や刑務所のお仕事について知ることができます。ネットにはあまり情報がないこのイベントについてお伝えします。

4.1　府中刑務所文化祭とは

秋の季節には各地で文化的なイベントが開催されます。とくに文化祭というと一般には学校や大学の行事ですが、調布市のおとなりの府中市には刑務所の文化祭「府中刑務所文化祭」があります。

府中刑務所文化祭は、日本最大の刑務所である府中刑務所（東京都府中市晴見町４の１０）が毎年 11 月 3 日（月・祝）文化の日に開催する文化祭です。入場は無料。今年で 39 回目の開催で、主な内容は以下のとおりです。

- 一日刑務所長によるテープカット
- 刑務所作業製品の販売
- プリズンアドベンチャーツアー
- 府中麦飯食堂
- 多数の模擬店

図 4.1　府中刑務所文化祭入場ゲート

私は午前 11 時頃に「第 39 回府中刑務所文化祭」と書かれた黄色いアーチから入場し、そのまま真っすぐ進んでプリズンアドベンチャーツアーの受付所へ向かいました。今年の一日刑務所長はモデル・女優の篠田麻里子さんでしたが、残念ながらテープカットは見逃してしまったのでそれ以降の時間のレポートとなります。

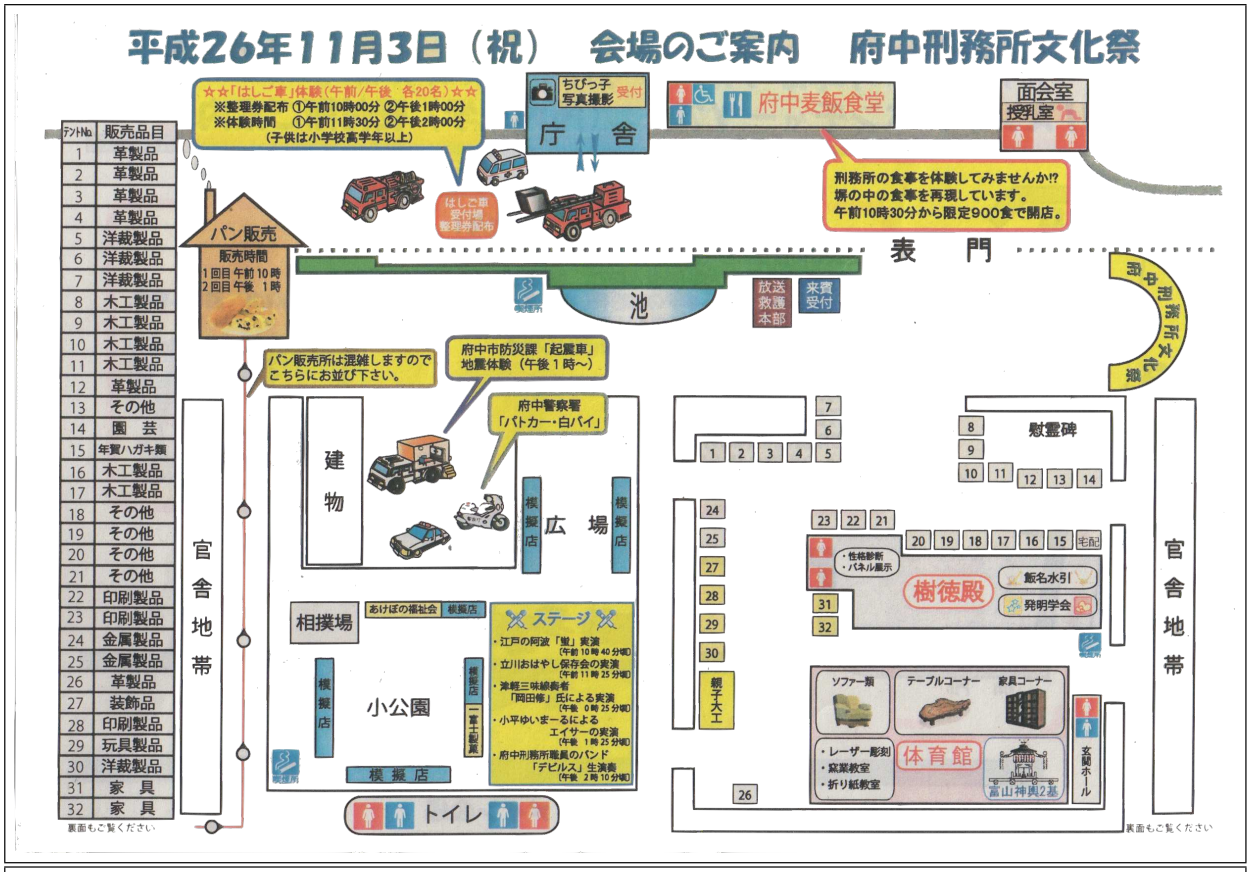


図 4.2　会場の案内 (上:オモテ面, 下:ウラ面)。A5 のコピー用紙でした。

4.2　プリズンアドベンチャーツアー（所内見学）

プリズンアドベンチャーツアーでは所内を歩いて見学できます。もちろん受刑者の方と会うことはありません。去年までは人数制限のあるバスツアーだったそうですが、今年はウォークラリーなので気軽に入れました。図 4.2 のように回りこんだところにある受付所から南門の手前まで行列ができていました。

30-40 分ほどで受付にたどり着きました。所内では携帯電話やカメラは使用禁止で、配布された袋に入れておきます。入場券を受け取り、入り口の前で一旦少し待ちました。入り口の扉はトラックが入れるくらいの大きさでした。

図 4.3　府中街道沿いの行列と高い壁。

図 4.4　ウォークラリーの入り口。テレビの取材があったようです。

扉が開くと何故か小さな歓声が。中に進むとすぐにまた同じような扉が続き、一定の人数が入ったら最初の扉は閉まりました。横は壁で囲まれているのでまるで閉じ込められたような雰囲気に。職員の方がウォークラリーの説明を始め、「入るときに受け取った入場券がないと出所できません」「無事に出所してきてください」とユーモアを見せます。2 つ目の扉が開いたらやっと広い所内が見えました。建物と建物の間は立入禁止のテープとぼろぼろのすずらんテープをパイプ椅子に付けて作った通路に沿って移動しました。

4.2.1　工場

いくつかの工場がありました。革製品の製作所では、靴や小銭居入れ、ブックカバーなどが展示されていました。整然と並んだ作業台と、安全意識を換気するポスターのある様子は普通の作業場と変わらない感じでした。ただしトイレの壁や個室の上部にはガラス窓があり、中が確認できるようになっていました。また、他にメモ帳やノートなどの印刷製品、椅子やすのこなどの木工製品の工場もありました。

4.2.2　入浴施設

入浴施設も見学しました。入り口が 2 つあり、それぞれ別の脱衣所につながっていますが、どちらも同じ浴場につながっています。2 つの脱衣所の間には設備の制御室がありました。見学ではまず片方の脱衣所から浴場を覗いて、大きめの浴槽が 2 つ隣接しているのを確認しました。そのまわりに洗い場が並んでいました。職員の方の話だと受刑者の方の入浴は週 2 回だそうです（少ない！！）。そこから制御室を通り、他方の脱衣所から外へ出ました。制御室は監視の場所でもあり、窓から浴場全体を見渡すことが出来ます。

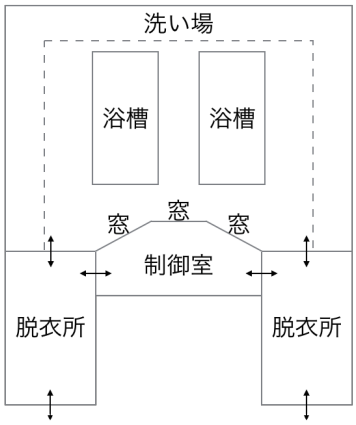


図 4.5　入浴施設の間取り図

4.2.3　体育館

体育館もありました。展示内容は古い舎房の鍵や反則物品、受刑者の食事のメニューでした。反則物品については、所内でやりが作られた事例があるそうです。展示スペースが小さかったのであまり見えませんでしたが、食事は朝昼晩とあまり変わらないメニューで、白米が麦飯だったことは確認できました。

30-40 分程度で最初の 2 つの扉まで戻ってきました。扉の横の通用口から出て入場券を渡し、無事、出所しました。

4.3　庁舎

会場へ戻って庁舎に向かいました。府中麦飯食堂では刑務所の食事を体験できますが、限定 900 食なので、来た時 (12 時頃) には既に終了していました。庁舎内では公益社団法人矯正協会による性格テスト

図 4.6　府中刑務所庁舎

図 4.7　麦飯完売のお知らせ

が行われていました。60 の質問に「はい」か「いいえ」で答え、性格を判断するというものです。更生へ向けて性格を矯正するため、ということでしょうか。私のテスト結果は、繊細で、意志が弱い性格ということでした。「まわりからはかたくるしいと思われることがあるかもしれません。もっと肩の力を抜いてゆったりやってはいかがでしょうか。」なるほど、堅苦しい……。当たっています。

図 4.8　性格テストの質問用紙

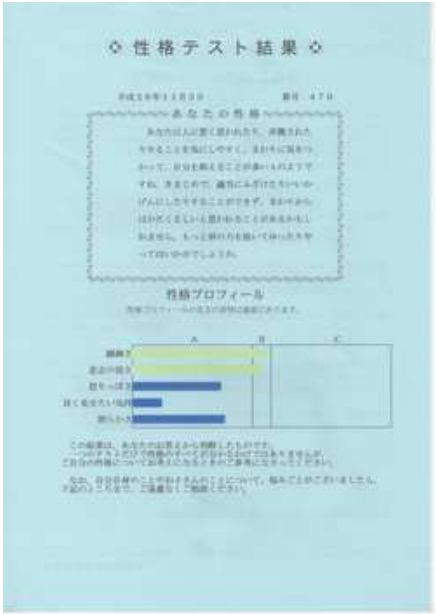


図 4.9　性格テストの結果

4.4　刑務所作業製品の販売

刑務所で作られた製品がたくさん販売されていました。所内見学で展示されていたようなものが全国の刑務所から集まっていました。他に縫製製品や金属製品、神輿などもあり、とくに函館少年刑務所のマル獄のマークが印象的でした。前掛けが有名だそうです。

図 4.10　刑務所作業製品の販売エリア。人気です。

図 4.11　函館少年刑務所の作業製品（前掛け）[*1]

[*1] 公益社団法人矯正協会刑務作業協力事業部 (CAPIC)e-shop より (http://www.e-capic.com/SHOP/707-03-00-01.html)

布製のブックカバー (税込 220 円) を買ってみました。和柄で落ち着くデザインです。

図 4.12　富山刑務所謹製 子供神輿。一応売り物のようです。

図 4.13　ブックカバー (文庫本サイズ)

後にも説明しますが、刑務作業は刑法で定められている懲役刑の内容であり、受刑者の改善更生と社会復帰のための重要な処遇方策の一つです。刑務作業の紹介、作業製品の販売を行う公益社団法人矯正協会刑務作業協力事業部 (CAPIC) は、刑務作業のメリットを次のように説明しています。

> 刑務作業は国の事業であり、安心して仕事を任せられます。
> - 常時一定の労働力が確保できます。
> - 敷地の購入費、工場、倉庫等の建設費がいりません。
> - 経常費が節約できます。
> - 情報管理が徹底できます。（施設の特殊性から）
> - 労務管理の心配がありません。(刑務所の責任で行うため)
> - 福利厚生関係費が要りません。(一般労働者ではないため)
> - 技術指導者が配置されています。
>
> CAPIC ウェブサイトより
> `http://www.e-capic.jp/keimusagyo`

刑務所特有の利点により、安くて品質の良い商品が実現するようです。皆さまも刑務所作業製品を是非お買い求め下さい。

4.5　パネル展示

4.5.1　受刑者の処遇について

会場の一部となっていた樹徳殿（体育館）では受刑者の処遇についてパネル展示が行われていました。

受刑者の処遇は、刑事収容施設及び被収容者等の処遇に関する法律において、その者の資質及び環境に応じ、その自覚に訴え、改善更生の意欲の喚起及び社会生活に適応する能力の育成を図ることを旨として行うものと規定されています。受刑者の入所から出所までの流れは次のようになります。

1. 入所
2. 入所時調査
3. 刑執行開始時の指導
4. 矯正処遇
5. 満期釈放前指導 (前期)
6. 矯正処遇
7. 釈放前指導
8. 出所

工程の主である矯正処遇は、作業、改善指導及び教科指導の 3 つの柱で構成されます。作業は先に述べた工場での製作、教科指導は義務教育の内容に準ずる指導のことです。

改善指導は、受刑者に犯罪の責任を自覚させ、社会生活に適応するのに必要な知識や生活態度を習得させるために必要な指導を行うものです。一般改善指導と特別改善指導に分かれ、特別改善指導はさらに 6 つに分かれます。

一般改善指導　全ての受刑者が受けます。講話、体育、行事、面接、相談助言その他の方法により、

1. 被害者感情を理解させ、罪の意識を培わせること
2. 規則正しい生活習慣や健全な考え方を付与し、心身の健康の増進を図ること
3. 生活設計や社会復帰への心構えを持たせ、社会適応に必要なスキルを身に付けさせること

などを目的として行われます。

特別改善指導　特定の受刑者が受けます。

- 被害者の視点を取り入れた教育: 殺人・傷害致死等の人の生命・身体を害する罪による受刑者のうち、被害者やその遺族等に対する謝罪の意識が低い者に対して行われます。場合によっては、被害者等に苦しみや悲しみ等を伝えてもらう講話を実施することもあるようです。
- 薬物依存離脱指導
- 暴力団離脱指導
- 性犯罪再犯防止指導
- 交通安全指導
- 就労支援指導

4.5.2　余暇活動の援助

刑事収容施設及び被収容者等の処遇に関する法律では、被収容者に対して、余暇時間帯等における自己契約作業、知的、教育的及び娯楽的活動、運動競技などの活動について、援助を与えることとされています。

クラブ活動　余暇活動の援助の一環として、被収容者が参加する短歌、俳句、書道、絵画など、主として文化系のクラブがあります。作品は別のコーナーで見ることが出来ました。

通信教育　学校教育法に規定する中学校、高等学校、大学等の通信制課程及び文部科学大臣が認定した通信教育を受講することができます。

ラジオ・テレビの視聴　被収容者が時事の報道に接する機会を付与するとともに、余暇時間における教養の向上と娯楽のため、一定の条件の下に、ラジオ・テレビを視聴させています。調布 FM は月 1 回、所内で DJ 番組を放送しているそうです。

4.6　ライブステージ

文化祭といえばバンドのライブ！ (？) ステージでは市民団体によるお囃子や三味線、エイサーなどが披露されていました。そして最後の演目は、毎年行っているという府中刑務所職員バンド「デビルス」の生演奏です。

図 4.14　府中刑務所職員バンド「デビルス」

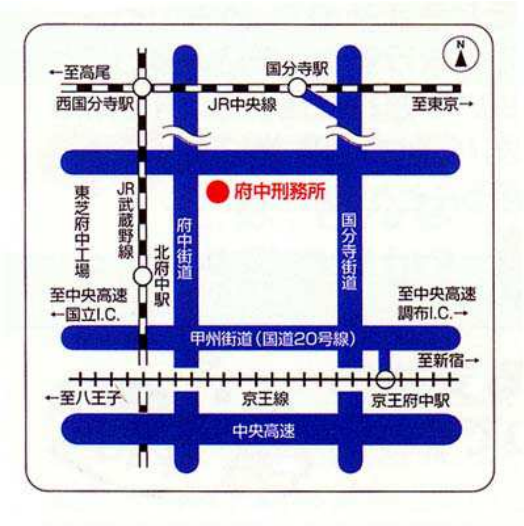
デビルスはインストバンドで、メンバーは若い方からおじさままで 6 名いらっしゃいました。曲目は昭和歌謡のギターアレンジやベンチャーズ、ディズニー映画『アナと雪の女王』の”Let It Go” などでした。クールな演奏と和やかな MC で会場は一体となりました！！ライブ終了後はすぐに文化祭終了時刻の午後 3 時となりました。

4.7　おわりに

麦飯食堂に行けなかったのは残念でしたが、刑務所内に立ち入ったことは貴重な体験でした。受刑者を常時監視することについてイメージはあったものの、実際に見るといささか生々しく感じました。しかし入浴が週 2 回というのは少ないですね。とくに夏は大変そうです。増やしてあげて欲しいものです。

パネル展示では、刑務所内でクラブ活動が行われていることが意外でした。とはいえ、文化的な生活を営む権利は尊重されなければなりませんから、これくらいは当然のことです。

「塀の中」を見学できる貴重な機会、第 40 回の開催も 11 月 3 日文化の日と予定されています。以上、中川がお伝えしました。



府中刑務所 [4]

- JR 武蔵野線「北府中駅」から徒歩 10 分
- JR 中央線「国分寺駅」から府中駅行きバス「晴見町」下車徒歩 5 分
- 京王線「京王府中駅」から国分寺駅行きバス「晴見町」下車徒歩 5 分

参考文献

[1] 公益社団法人矯正協会刑務作業協力事業部（CAPIC）ウェブサイト http://www.e-capic.jp/

[2] 法務省 刑事施設 (刑務所・少年刑務所・拘置所)
http://www.moj.go.jp/kyousei1/kyousei_kyouse03.html

[3] 法務省 平成 22 年版 犯罪白書 ～ 重大事犯者の実態と処遇 ～
http://hakusyo1.moj.go.jp/jp/57/nfm/mokuji.html

[4] 法務省 施設のご案内（府中刑務所）
http://www.moj.go.jp/KYOUSEI/KEIMUSAGYO/sagyo/sisetu_fuchu.html

第 5 章

電通大 Objet 紀行 2014

f_taka6

概要
ある調査によると、この電通大のキャンパスで「スゴイ」というところは「オブジェ (仏:objet)」だそうですよ。この記事はオブジェクト指向 (英:object-oriented) とは関係ないです。
皆さん、オブジェは好きですよね？

5.1　はじめに

5.1.1　2014 年 9 月 20 日

夏休みも終わりに近づいた 9 月のある日、Twitter の TL に流れてきた「学生ウォーカー」という WEB サイトに、この電通大に関するとある記載があった。[1] その内容は次ページに引用した。

この WEB サイトのアンケート結果に拠ると、電通大キャンパス内で最もスゴイというところは「オブジェ」らしい。1 位にも関わらず得票率が 1 割に満たないことは気にしてはいけない。

電気通信大学情報理工学部の学生数 3,418 名 [2] と同程度の数字である 3,478 名というアンケートを採った一人暮らしの学生数は、どの大学のアンケート結果ページにも書いており、電通大では何人にアンケートと採ったのかは書かれていない。また、電通大で一人暮らしをしていると思われるのは、電通大パンフレット記載の住居についてのアンケート結果 [3] から「下宿・アパート 51 %、五思寮・国際交流会館 3 %」の合わせて 53 %であり、半分ぐらいの学生のみを対象にしている。その様なことも気にしない。

同一項目において他の大学は、東京農工大学 (工学部) では「自然豊か」、東京工業大学 (大岡山) では「ウッドデッキから見える桜がとてもキレイ」、東京理科大学 (葛飾) では「新築できれい」、一橋大学 (国立) では「建物に趣がある」、首都大学東京では「広さ」、明治大学 (和泉) では「図書館が素晴らしい」、上智大学 (四谷) では「敷地が狭いから友達に会える」といったものが並ぶ中、電通大は「オブジェ」である。

ということで、この記事は電通大の学内を探索して、オブジェを見て回ったことについて記す。

5.2　事前調査

5.2.1　2014 年 10 月 13 日-30 日

まず、Google 検索で「電通大　オブジェ」と検索すると、学友会群青編集委員会が発行している学友会誌のバックナンバー情報*1より「UEC Times 2011」に学内のオブジェについての記事「学内謎オブジェ事情」[4] が存在することや、アンサイクロペディア*2の電気通信大学の記事 [5] の「学

*1 バックナンバー (電気通信大学 学友会 群青編集委員会) http://www.megrokai.or.jp/gunjyo/publication/index.html

*2 フリー八百科辞典とも呼ばれる真実性よりもユーモアを重視しているらしい wiki 形式の WEB サイト http://ja.uncyclopedia.info/wiki/

“電気通信大学 アンケート (大学について) — 学生ウォーカー”
http://www.gakusei-walker.jp/s/43/survey/univ.html

首都圏へ進学した一人暮らしの大学生 3,478 名が答える！

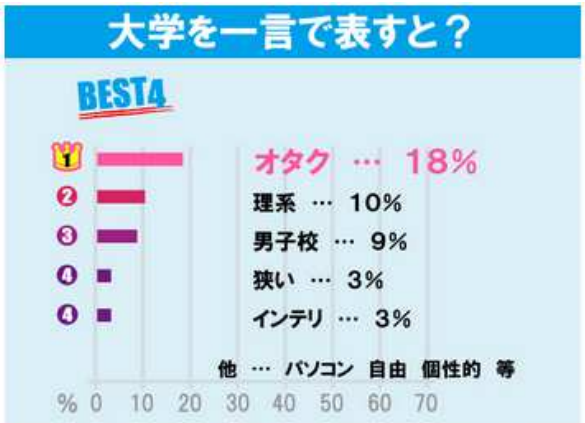
電気通信大学編

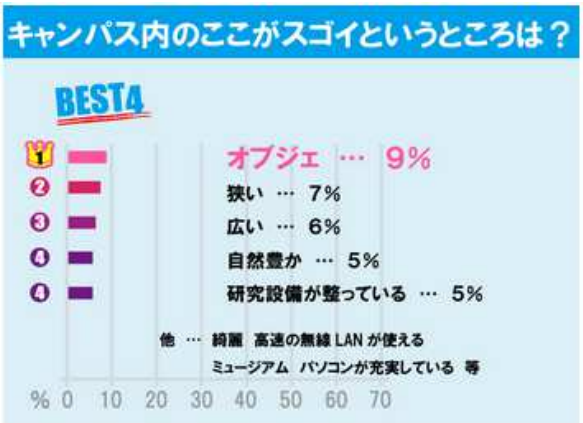
学生ウォーカーでは、電気通信大学の在学生で一人暮らしをしている方を対象に「お部屋探しについて」「大学について」「学生生活について」「アルバイトについて」アンケートを行いました。

首都圏へ進学した一人暮らしの大学生 3,478 人が答えた！

【電気通信大学について】

電気通信大学について聞いてみたところ『オタク』『理系』『男子校』など、『キャンパス内は無線 LAN が整備されている』や『研究設備が整っている』など理系の大学らしい意見もありました。学食で一番人気のメニューは僅差で『チキン竜田』が人気となりました。





※調査方法：電気通信大学の学生を対象に、2014 年 4 月〜6 月に WEB アンケートを実施

内のオブジェ」についての記載が主に出てきた。また、学友会群青編集委員会の群青かわら版[*3]では、新たなオブジェが電通大入りしたときには記事 [6][7] となっていた。「UEC Times 2011」については、MMA 部室の蔵書の中に存在していたためこれを閲覧し、他の文献については、WEB で閲覧可能なものを閲覧した。

これらによると、電通大には以下のオブジェがあるらしい。

- 輪郭の断片
- 泉
- 橋を渡す
- far beyond the
- 打ち棄てられたパラボラアンテナ
- 意図不明の巨大コンクリスピーカー
- 国際交流会館前のリサージュ
- 東 1 号館尖塔の皿
- ベンゼン池
- 西 2 号館のパラボラアンテナ群

これらの中で、「西 2 号館のパラボラアンテナ群」以外のものは東地区に存在し、10 個中 9 個が東地区という事態になっている。「学内謎オブジェ事情」の末尾では「本稿で紹介した謎オブジェ以外に、学内に変なものがあれば、遠慮なく群青編集委員会へ (奥付の連絡先へ) 知らせてほしい。空白地帯の西地区での発見の一報であれば、なおさら歓迎する。」と記載されており、西地区でのオブジェ情報が求められている様子が読み取れる。

その他にも、2010 年 4 月のコミュニケーションパークの開設時に「輪郭の断片」と「泉」が設置され、[8] それに続いて明確な名前のあるオブジェ「橋をわたす」[6]「far beyond the」[7][9] と置かれていった様子も知ることができた。

*3 群青かわら版 (電気通信大学 学友会 群青編集委員会) http://www.megrokai.or.jp/gunjyo/kawara/index.html

5.3　実地調査

今回の調査の対象は、大学概要に電気通信大学の所在地として載っている土地 [10] のうち、「調布キャンパス」(調布市調布ヶ丘) と「学生寮」(調布市富士見町) の合わせて $115,433\mathrm{m}^2$ を対象とし、「多摩川運動場」(調布市多摩川), 「調布宿舎」(調布市小島町), 「藤沢浜見寮」(神奈川県藤沢市辻堂東海岸), 「菅平宇宙電波観測所・菅平セミナーハウス」(長野県上田市菅平高原) は対象としない。以下「調布キャンパス」と「学生寮」をあわせて調布キャンパスとする。

事前調査で出てきたオブジェの中には、地面の模様や建物の意匠、池、アンテナといったものが含まれていたので、最低限の要件を「その位置に固定されている」とした上で、「説明文がある」や「目立つ」など、ある程度広くオブジェと捉えていくことにした。

5.3.1　2014 年 10 月 31 日

1 日目。スマートフォンを手に持ち、調布キャンパスの外側の公道に接している部分を回り、何か無いかを見た。存在していたのは「東京青年師範学校跡の碑」だけであった。他に正門付近を回り、東京農工大学から贈られた「ニュートンのりんごの木」[11] を撮影したりした。

5.3.2　2014 年 11 月 1 日

2 日目。スマートフォンとビニール傘を手に持ち、雨が降ったり止んだりする調布キャンパス東地区を回った。

道を外れれば、膝上の高さまで草が生えているところや、藪が背の高さまであるところがあったりする、「自然豊か」な風情が存在する。草木に溜められた雨水に濡れながら掻き分けて進み、事前

図 5.1　自然豊かな電通大 (東 35 号館西側付近)

調査で挙げられたものや設置してあった幾つかの碑などを撮影した。藪の中には、打ち棄てられてよく分からなくなった物など、不法投棄されているようなものはあったが、固定して設置されているようなものは無かった。雨に濡れた草木と触れ合い、服が水浸しになったことは、無駄であった気がしなくもない。

5.3.3　2014 年 11 月 4 日

3 日目。同様にスマートフォンを手に持ち、調布キャンパス西地区を回った。

西地区は 2014 年現在、体育館の取り壊し及び新築工事が進行中であり、その部分には入れなかった。体育館工事に併せたのか、西地区東側の公道に面している部分では歩道整備工事も進行しており、その部分にも近寄れなかった。

西地区は東地区と比べて、道を外れても草の高さが低く、背の高さほどもある藪というのも殆どなく歩きやすかったが、碑も無く説明文があるものも「情報数理工学科記念樹」しかなく、西地区のオブジェ困窮具合が窺うことができた。また、事前調査であった「西 2 号館のパラボラアンテナ群」は、下から見て確認できなかった。

図 5.2　西 31 号館玄関前から撮影した西 2 号館

5.3.4　2014 年 11 月 7 日

4 日目。同様にスマートフォンを手に持ち、東地区正門前にある「ニュートンのりんごの木」[11] の贈り元である東京農工大学小金井キャンパスへ、元の木を見に行くために向かった。

電通大から東京農工大学小金井キャンパスへは、調布駅北口・調布銀座・火の見下の何れかの停留所で京王バス武 91 武蔵小金井駅南口行きに乗車し、終点の武蔵小金井駅南口停留所で下車し、農工大通りを東の方へ道なりに進めば、辿り着くことができる。

東京農工大学小金井キャンパスでは、「ニュートンのりんごの木」は 4 号館の南側に植えられており、[12] 立派に枝を広げている。この木の穂木が植えられたのは 1999 年 4 月のことであるそうなので、15 年ほどしたら、電通大のあの木も立派に枝を伸ばしているのだろうか。とりあえずは、りんごが生っているか、葉が生い茂っているときに見にきたかった思いがある。

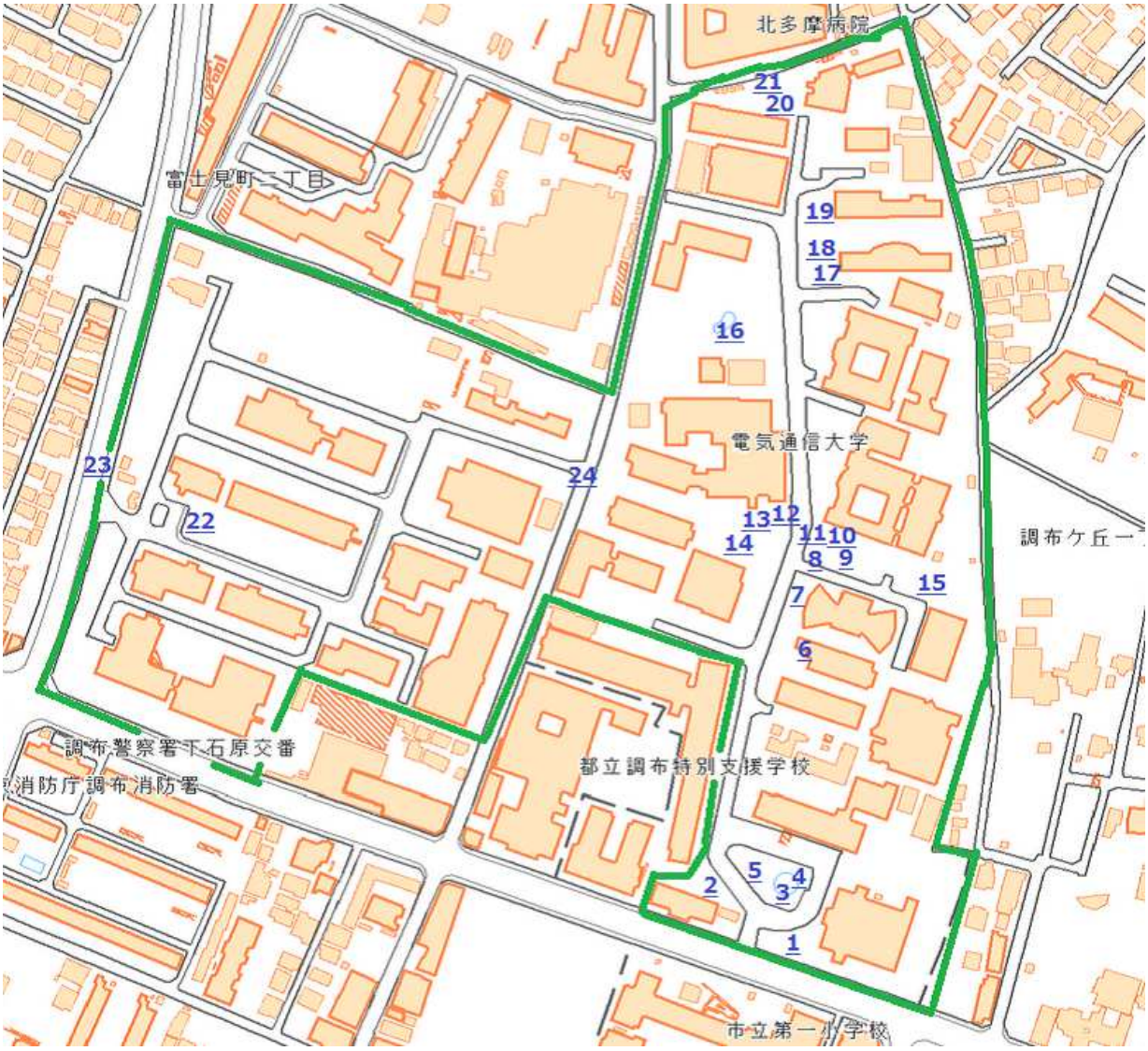


図 5.4　電気通信大学オブジェ地図

図 5.3　東京農工大学小金井キャンパスの「ニュートンのりんごの木」

5.4　まとめ

上記地図は、国土地理院の地理院地図[*4]に、筆者が調布キャンパスのだいたいの外縁を緑色に塗り、オブジェ存在箇所に青数字を書き加えたものになる。外縁などはうろ覚えで塗ったので、多少誤りがあっても気にしてはいけない。C 棟の形状が古いことなども気にしてはいけない。

[*4] http://portal.cyberjapan.jp/site/mapuse4/index.html?lat=35.657306&lon=139.542868&z=18&did=std

以下が、数字と名称 (仮称含む) の対応となる。

1. 叙勲記念植樹
2. 無線通信士教育発祥之府
3. 噴水
4. 時計
5. ニュートンのりんごの木
6. 東 1 号館尖塔の皿
7. far beyond the
8. 電気通信大学短期大学部 閉学記念植樹
9. 泉
10. 角田稔学長 退官記念植樹
11. 電通大ちょうふエリア放送 送信所
12. 「人と人との出会いの場」
13. 記念植樹 ハナミズキ
14. 輪郭の断片
15. 橋を渡す
16. 池
17. コンクリート製巨大スピーカー
18. 使われていないパラボラアンテナ
19. ベンゼン池
20. リサージュマーク
21. 石灯籠
22. 情報数理工学科記念樹
23. 東京青年師範学校跡
24. 押ボタン式信号機

これらを上から順に紹介する。

5.4.1　叙勲記念植樹

丘の小道 (正門側から南門へ向かう道) の入口の左手、写真中央に写る木がその木である。木に括り付けられた説明文には「叙勲記念植樹 名誉教授 平島 正喜 先生 昭和 57 年 11 月 ウメ」とある。平島正喜氏とは、電気通信大学第 6 代学長を昭和 51 年 (1976 年)5 月 1 日から昭和 57 年 (1982 年)4 月 30 日まで務めた方だと思われる。

図 5.5　叙勲記念植樹

5.4.2　無線通信士教育発祥之府

図 5.6　無線通信士教育発祥之府

創立 80 周年記念会館「リサージュ」の入口の手前にある碑である。碑には「無線通信士教育発祥之府 1918 年電気通信大学の前身校である無線電信講習所において民間の無線通信士育成が始められた以来 1 世紀に亘りその教育が継承されてきた

事を記念してこの碑を建てる 2005 年 5 月吉日 船舶通信士労働組合」という文章が刻まれている。

目黒会報 17-2 号の記事 [13] に拠ると、太平洋戦争中に電通大の前身校の多くの卒業生が船舶通信士として戦没しているほか、学生も実習生として乗り込み亡くなっており、昭和 30 年 (1950 年) 頃に学内に慰霊碑を建てられないか大学当局に問い合わせたが、宗教上の問題などがあるということで設置できなかった。平成 11 年 (1999 年) に船舶のアナログの無線電信が廃止され、それと同時に無線通信士も消滅することになり、船舶通信士労働組合が残務処理として処分資産の大半を電気通信大学歴史資料館に寄付することになった。それと同時に、無線通信士という職の名を残すことと慰霊の意を含めた記念碑を学内に建てられないか大学に問い合わせ、「無線通信士教育発祥之府」なら良いという許しを得て、無線通信士という職名を後世に残すことは先に逝った方々にも供養にもなるとして、建立されたものだという。

5.4.3　噴水

図 5.7　平時の噴水

何かイベントごとがあるときのみに稼働することで知られている、正門と本館の間にある噴水。どう考えても、元気に稼働して水を吹きだしている日よりも、配管剥き出しで少しの澱んだ水が

図 5.8　有事の噴水 (2014 年 4 月 4 日)

残っている姿を晒している日の方が多い。

5.4.4　時計

図 5.9　時計

噴水近くにある時計。正門と南門から入ってきた人たちに今の時刻を知らせてくれている。入構したときに少し目を上に向ければ、遅刻かどうかがだいたい分かる。調布キャンパス内で外に時計がある場所はここだけだと思われる。昼休みなどに多く人が集まるコミュニケーションパークにも

時計は存在した方が有用だとは思う。

5.4.5　ニュートンのりんごの木

図 5.10　ニュートンのりんごの木 (4 月 15 日)

図 5.11　ニュートンのりんごの木 (10 月 31 日)

正門から入って本館方向に進んだ左手側にある、2014 年 3 月 26 日に東京農工大学より贈られ植樹された木。[11] ニュートンが万有引力を発見したときの伝承で出てくるりんごの木を、継ぎ木で増やしていったものの一つになる。6 ヶ月程度でも成長は分かる。

図 5.12　東 1 号館尖塔の皿

5.4.6　東 1 号館尖塔の皿

東 1 号館の上の方。皿らしい。特徴的な形はしている。こちら側は凹状となっているが、反対側は凸状になっている。「学内謎オブジェ事情」[4] に拠れば、ライトアップされることもあるらしい。

5.4.7　far beyond the

メインストリート沿い旧 C 棟前にある、2013 年 4 月 5 日に除幕された [7][9] 黒御影石でできた灯台を形取った彫刻。梶谷誠学長 (当時)(2014 年度現在・学長顧問) が平成 24 年度東京芸術大学卒業作品展に足を運び、石倉未那美氏のこの作品を選定したらしい。除幕式が挙行され、「さわやかな風の吹く中、創立 95 周年の幕開けを飾るにふさわしいイベントとなりました。」[9] とのこと。古い地図と見比べると、設置場所は、2012 年頃に C 棟の一部を取り壊して [14] できた余地にあたると思われる。

図 5.13　far beyond the

図 5.14　電気通信大学短期大学部 閉学記念植樹

far
beyond
the

説明板では上の様に記載されており、「目の前にただ漠然と広がる海と、それを前にただ茫然と立ち尽くすあなたへ」という言葉がそえられている。

5.4.8　電気通信大学短期大学部 閉学記念植樹

碑には寄贈主として「短期大学部卒業生一同 平成 2 年 3 月」とある。電気通信大学短期大学部は、昭和 28 年 (1953 年)8 月 1 日に大学に併設され、昭和 62 年 (1987 年)10 月 1 日に電気通信学部に夜間主コースが設置されたことに伴いそれに統合され、平成 4 年 (1992 年)3 月 31 日に廃止されている。[15]

5.4.9　泉

2010 年 4 月に、施設跡の空き地を憩いの場として整備したというコミュニケーションパーク。そこにふさわしい「人と人とのコミュニケーション」

図 5.15　泉

をテーマにしたという、東京藝術大学の学生制作によるもののひとつ。[8] 名倉達了氏による制作。御影石製。

説明板には「静寂な森の中でこんこんと水が湧き出るように、私たちは命を受け継ぎ、思考し創造行為を続ける。」という言葉がそえられている。

5.4.10　角田稔学長 退官記念植樹

1994 年 4 月に六八会というところから寄贈されたというシダレザクラ。六八会という名称は、世

図 5.16　角田稔学長 退官記念植樹

図 5.17　電通大ちょうふエリア放送 送信所

間一般で第 68 回や 68 年などの卒業生の団体などに使われている。角田稔氏とは、電気通信大学第 8 代学長を昭和 63 年 (1988 年)5 月 1 日から平成 6 年 (1994 年)4 月 30 日まで務めた方で、平成 6 年 (1994 年)5 月 1 日に名誉教授の称号が授与されている。専門は電子工学（電子物性、電子計測）。

5.4.11　電通大ちょうふエリア放送 送信所

「この施設では、当地域では使用されていない放送用周波数「ホワイトスペース」を活用して、狭小なエリアでのワンセグ放送を行う無線局です。」とある、U.E.Cast -調布ワンセグ-の放送を発信している地上一般放送局。

U.E.Cast の WEB ページより、「U.E.Cast は、地域向けワンセグ放送の技術検証・実証実験を通じて、地域の情報発信支援を行うプロジェクトチームです。電気通信大学のエンジニアリングデザイン科目の PBL（課題実践型教育）から誕生したもので、関係教員の指導の下現在学生で運営しています。」[16] とのこと。

5.4.12　「人と人との出会いの場」

図 5.18　「人と人との出会いの場」

東 3 号館 (旧称：総合研究棟) の前にある、平成 14 年 (2002 年)3 月設置の碑。電通大の沿革には「2002 年 3 月 31 日 総合研究棟（コミュニケーションパーク）竣工 新図書館完成」[15] と記載があり、それに伴って設置されたものと推測される。

過去には「コミュニケーションパーク」という言葉で東 3 号館を指していたことがあるのかもしれない。

5.4.13　記念植樹 ハナミズキ

図 5.19　記念植樹 ハナミズキ

「人と人との出会いの場」の横にある、同様に平成 14 年 (2002 年)3 月に植えられたハナミズキ。説明板の樹木名と日付の間には、「附属図書館長 矢部 初男 附属図書館事務長 八角 晃弘」とある。

5.4.14　輪郭の断片

図 5.20　輪郭の断片

2010 年 4 月に、施設跡の空き地を憩いの場として整備したというコミュニケーションパーク。そこにふさわしい「人と人とのコミュニケーション」をテーマにしたという、東京藝術大学の学生制作によるもののひとつ。[8] 片村信氏による制作。赤御影石製。大学会館から東 3 号館へ向かう線上のど真ん中、他と違い地面に直に置いてあるような形なので、目に入る機会も多い。

説明板には「記憶の中の輪郭、未来へ想い描く輪郭、今ここに在る輪郭。何気ない存在の断片に触れながら、また新しく出会っていく。」という言葉がそえられている。

5.4.15　橋を渡す

図 5.21　橋を渡す

コミュニケーションパークの東端にある 2011 年の春休み中に設置されたらしい、石井琢人氏が製作したブロンズ製の作品。[6] 説明板によると、正しい名称は「橋を渡す -川の流れのように、人と人が繋がるように-」らしい。

説明板には「本作品は、制作途上に起こる表情や色の変化、その時々の些細な出来事や気持ちの揺らぎを刻み残そうと試みた。 完成が近づくにつれ、それら全てが Link されていった。」という言葉がそえられている。片方の橋の下しか舗装していないのは、意味があるのだろう。

5.4.16　池

図 5.22　池

「噴水とベンゼン池がオブジェなら、保健管理センターと東 35 号館の間にある普通っぽい池も、オブジェであろう。」という気持ちで加えられたもの。

図 5.23　ベンチと藤棚

ベンチと藤棚も用意されていて、憩いの場みたいな雰囲気を醸し出している。しかし、ここに人が滞在して懇談しているような場面は、私は見たことがない。

図 5.24　コンクリート製巨大スピーカー前面

図 5.25　コンクリート製巨大スピーカー背面

5.4.17　コンクリート製巨大スピーカー

東 9 号館とメインストリートの間にある、青・赤・緑と 3 色のコンクリート製のあれ。

とある技術系の電気通信大学公認団体の構成員筋の話では、50 年ぐらい前の工学研究部[*5]が製作したものらしい。40 年ぐらい前には、音楽系団体が調布祭での演奏時に使用していたとか。今でも新歓時期には小音量で鳴らしているという話もある。[*6]

[*5] 電気通信大学工学研究部 `http://www.koken.club.uec.ac.jp/`

[*6] 他にも、とある先輩が 2014 年 4 月に「これが電通大工学研究部名物、巨大スピーカーである！」などと写真とともにツイートしてたり、「工学研究部 オーディオ班」

5.4.18 使われていないパラボラアンテナ

図 5.26 使われていないパラボラアンテナ

上記のスピーカーの隣にあるパラボラアンテナ。「國興電業株式会社 昭和 54 年 3 月」という札がついている。國興電業株式会社は、現在の國興システムズ株式会社にあたる。

「学内謎オブジェ事情」[4] に拠ると、かつて電通大が気象衛星ひまわりからの電波を受信し日本各所に配信するときに利用していたもので、2011 年時点では無線部*7が不定期に掃除する他は野に晒されたまま放置されている状態だという。

5.4.19 ベンゼン池

東 10 号館とメインストリートの間にある、六角形であのベンゼンの記号の形に似た池 (?)。「学内謎オブジェ事情」[4] に拠れば、2011 年時点で 20 年前からベンゼン池と呼ばれていたらしい。鯉がいるらしいが、見えなかった。

図 5.27 ベンゼン池

5.4.20 リサージュマーク

図 5.28 リサージュマーク

国際交流会館の敷地に入って左手の、小さな公園みたいな空間の地面の模様。他では見ない地面の模様。

5.4.21 石灯籠

リサージュマークのある空間にある石灯籠。とても日本的雰囲気。やはり他では見ない。

が「巨大スピーカー・コンクリートホーンに象徴される班」と 2000 年度の班長名とともに書かれている WEB ページがあったりする。

*7 電気通信大学無線部 (電通大クラブ) `http://www.megrokai.or.jp/ja1zgp/`

図 5.29　石灯籠

図 5.30　情報数理工学科記念樹

5.4.22　情報数理工学科記念樹

西門から入ってすぐの西 4 号館の前にある碑と木。情報数理工学科は 1973 年 4 月 1 日に設置され (情報数理工学専攻は 1977 年に設置)、1987 年 10 月 1 日の改組で計算機科学科と併せて情報工学科に再編されている。

碑の裏には「1990 年 7 月 1 日 山崎吾郎氏寄贈」とあり、その日付から見て、情報数理工学科というものが存在していたことを後世に残すことが目的の記念樹なのかもしれない。

しかし、この碑の最も近くにある、この碑が指していると思われる木は、枝のほぼすべてを取られて幹だけを晒しているような状態になってしまっている。何が起きたのだろうか。

5.4.23　東京青年師範学校跡

電気通信大学を新制大学とするときに土地の広さが問題になり、個人所有であった現在の東地区を比較的安値で譲ってもらい、東京青年師範学校 (後継校:東京学芸大学) の跡地であった現在の西地区と併せて、新制大学の基準の 3 ～ 5 万坪以上としたそうだ。[17] ということで、この碑は西門の外、西門を出て右手に存在する。

図 5.31　東京青年師範学校跡

同地は 1949 年 5 月の東京学芸大学発足時には東京学芸大学調布分教場となったが、1951 年 3 月の東京青年師範学校廃止と同時に調布分教場も廃止されている。[19] なお、電気通信大学の調布校舎開校は 1952 年 4 月 [15] なので、教場が隣り

合って存在したことは無い。東京学芸大学の寮は1956 年頃まで存在していたようである。[18]

5.4.24　押ボタン式信号機

図 5.32　押しボタン式信号機 (2 月 12 日 (起動日))

「far beyond the のとき [9] も、ニュートンのりんごの木のとき [11] も、式典を行っていた」&「中門の横断歩道に押しボタン式信号機ができたとき [20] に、式典を行っていた」⇒「中門の信号機もオブジェ」 という思考で後になって付け足したもの。

火入れ式という式典では、調布警察署の署長さんの挨拶があったり、初渡りなどをしていた。

「平成 22 年度「キャリアデザイン C」(現「エンジニアリングデザイン」) の学生プロジェクト「電通大ウォーカー」(学内交通環境改善チーム) の活動の中で行った調布警察署への提案要望が認められ、設置の運びとなりました。」[20] とのこと。同じプロジェクト内で提議されている「横断歩道の位置が門の位置とはずれている件」や「西地区の南側に新西門を作って交通を分散させたい」といったものは、実行できていことも同時に知らしめている。

中門に信号ができたことにより、横断歩道に歩行者が居ようが停まろうとしない多くの自動車を塞き止めて、安全に渡ることができるようになった。信号無視して横断歩道へ飛び出していくような人達もいれば、信号無視して横断歩道に突入する自動車もいるので気を付ける必要はあるが。

5.5　おわりに

5.5.1　2014 年 11 月 17 日

オブジェは思ったより多かったというか、少なかったというか、妥当というか。西地区で「情報数理工学科記念樹」の存在を見かけていて、「他にも似たような植樹はあるのかな？」とわくわくしていたので、見つけられなかったことは残念に思っている。

私が伺ったとき、東京農工大学は学園祭が行われており、東京農工大学 MMC[*8]さんのブースに立ち寄って、展示品の説明などを受け、部誌購入をし、配っていた風船犬を頂いた。案内をしてくださった方、ありがとうございました。

書き始めたときには「7,8 ページで終わるかな？」などと考えていたのにも関わらず、倍程度になってしまった。百萬石で他の方がどのくらい書くのか、そもそも誰が書くのか何を書くのかを知らないので、浮きすぎないか不安ではある。

また、締切 (11/14) を超過してしまい、百萬石編集担当の masn19 副部長には本当に迷惑をかけました。(締切日にどのくらい記事が集まっていたかは知らないが、) 締切を伸ばしてくださりありがとうございました。

[*8] Micro Computer Club in TUAT `http://www.tuat.ac.jp/~tuatmcc/`

参考文献

[1] “電気通信大学 アンケート (大学について) — 学生ウォーカー” http://www.gakusei-walker.jp/s/43/survey/univ.html (2014 年 11 月 1 日閲覧)

[2] “学生数｜電気通信大学” http://www.uec.ac.jp/about/basicinfo/number_student/ (2014 年 11 月 12 日閲覧)

[3]「電通大生の住宅アンケート」,『国立大学法人 電気通信大学 受験生のための 大学案内 2015』, p.52, 国立大学法人 電気通信大学. http://www.uec.ac.jp/about/profile/pamph/pdf/annai2015.pdf (2014 年 11 月 12 日閲覧)

[4]「学内謎オブジェ事情」,『UEC Times 2011』, pp.64-65, 電気通信大学 学友会 群青編集委員会.

[5] “電気通信大学 - アンサイクロペディア”(学内の建造物-学内のオブジェ) http://ja.uncyclopedia.info/wiki/電気通信大学 (2014 年 11 月 1 日閲覧)

[6] 草場 (2011)「謎オブジェに新顔」,『群青かわら版』2011 年 5 月 17 日号 (2011 年第 1 号), p.2, 電気通信大学 学友会 群青編集委員会. http://uecgunjyo.web.fc2.com/kawara/1101.pdf (2014 年 11 月 1 日閲覧)

[7] 篠原 (2013)「新オブジェ誕生」,『群青かわら版』2013 年 5 月 17 日号 (2013 年第 1 号), p.2, 電気通信大学 学友会 群青編集委員会. http://uecgunjyo.web.fc2.com/kawara/1301.pdf (2014 年 11 月 1 日閲覧)

[8]「コミュニケーションパークの紹介」,『調布電通大通り No.24』2010 年夏号, 国立大学法人 電気通信大学 広報センター ・ 調布市 行政経営部 広報課. http://www.uec.ac.jp/about/publicity/pamphlet/pdf/dooori24.pdf (2014 年 11 月 1 日閲覧)

[9]「【報告】新モニュメント「far beyond the」の除幕式を挙行」(電気通信大学) http://www.uec.ac.jp/news/announcement/2013/20130422-4.html (2014 年 11 月 1 日閲覧)

[10]「土地・建物面積及び所在地一覧」,『概要 2014-2015』(2014 年 8 月), p.30, 電気通信大学広報センター. http://www.uec.ac.jp/about/profile/pamph/pdf/gaiyou2014-2015.pdf (2014 年 11 月 12 日閲覧)

[11]「ニュートンのりんごの木 寄贈記念植樹式【3 月 26 日開催】」(電気通信大学) http://www.uec.ac.jp/news/event/2013/20140311-1.html. (2014 年 11 月 1 日閲覧)

[12]「香取研の写真集　おまけ ニュートンのリンゴ」(香取研 / フォトギャラリー) http://www.tuat.ac.jp/~katori/newtonapple.html　(2014 年 11 月 5 日閲覧)

[13] 小田原敏介 (2005),「無線通信士記念碑建立」,『目黒会報 Chofu network 17-2 号』(2005 年 8 月) http://www.megrokai.or.jp/megurokai/kaihou/kiji/kiji172/kinenhi.html (2014 年 11 月 14 日閲覧)

[14] 増田 (2012),「C 棟工事始まる」,『群青かわら版』2012 年 1 月 25 日号 (2011 年第 6 号), p.2, 電気通信大学 学友会 群青編集委員会. http://uecgunjyo.web.fc2.com/kawara/1106.pdf(2014 年 11 月 15 日閲覧)

[15] “沿革・歴史｜電気通信大学” http://www.uec.ac.jp/about/profile/history.html (2014 年 11 月 15 日閲覧)

[16]「U.E.Cast とは?」,“電気通信大学 U.E.Cast -調布ワンセグ-” http://www.ce.uec.ac.jp/uecast/Action/U.E.Cast.html (2014 年 11 月 16 日閲覧)

[17] 電気通信大学六十周年史編集実行委員会編 (1980), 「1-1 用地探しの苦労」, 『電気通信大学六十年史』(第 4 章第 1 節 1-1), 電気通信大学同窓会 社団法人目黒会 http://warp.ndl.go.jp/info:ndljp/pid/261763/ssro.ee.uec.ac.jp/lab_tomi/uec/uec-80/uec-60/kouhen/4-1-1.htm（国立国会図書館によるアーカイブ）(2014 年 11 月 16 日閲覧)

[18] 電気通信大学六十周年史編集実行委員会編 (1980), 「4-2 寮」, 『電気通信大学六十年史』(第 3 章第 4 節 4-2), 電気通信大学同窓会 社団法人目黒会 http://warp.ndl.go.jp/info:ndljp/pid/261763/ssro.ee.uec.ac.jp/lab_tomi/uec/uec-80/uec-60/kouhen/3-4-2.htm（国立国会図書館によるアーカイブ）(2014 年 11 月 16 日閲覧)

[19] “沿革表｜国立大学法人 東京学芸大学” http://www.u-gakugei.ac.jp/01gaiyo/enkaku.html (2014 年 11 月 17 日閲覧)

[20]「中門信号機設置記念 灯入れ式【2 月 12 日開催】」(電気通信大学) http://www.uec.ac.jp/news/event/2013/20140207-2.html (2014 年 11 月 17 日閲覧)

第 6 章

コンパイラの作成

nomeaning

概要
コンピューターを使っている時、プログラミング言語には良くお世話になっています。それなのに、どのようにしてソースコードから実行ファイルを作り出しているのかをあまり知らなかったので、自分で作ってみることにしました。

6.1　どんな言語を作るか

6.1.1　作成する言語の仕様

次のような C っぽい言語を作ることにしました。

- 出力するアセンブリは x86 のみで、単一のソースから ELF 形式の実行ファイルを作ることだけを考える。
- アセンブリを NASM でアセンブルし、ld を用いて実行ファイルを生成する。
- 最適化とか一切考慮しない。
- 実装する文は、変数宣言、条件分岐 (if,else)、繰り返し (for, while, break, continue)、直接アセンブリ出力 (asm)、値の返却 (return) のみ。
- 演算子として実装するのは、四則演算 (`+`,`-`,`/`,`*`,`%`)、比較 (`>`, `>=`, `<`, `<=`, `==`, `!=`)、代入 (`=`)、ポインタ演算 (`*`, `&`, `[]`) のみ。
- 型は整数型 (char, short, int, unsigned) とそのポインタ型、配列型のみ。
- 変数はローカル変数のみ。変数宣言時のポインタ指定は全ての変数に有効。
- 関数について引数のチェックをしない。

6.1.2　開発に利用するツール

言語を開発する時に利用する言語としては Ruby、パーサジェネレータとしては Racc を用いることにしました。[*1]

6.2　コンパイルの流れ

ソースコードを実行ファイルに変換する時は、図 6.1 のような順番で変換されます[*2]。まず、ソースコードはスキャナ (字句解析器) によりトークン (字句) に分解されます。次に構文解析器がプログラムの文法にしたがって解析を行い、構文木を生成します。最後に構文木を元にアセンブリを生成 (コンパイル) します。そのアセンブリをアセンブラでアセンブルし、オブジェクトファイルを作り、リンカにより実行ファイルを作り出します。今回作成したコンパイラでは、アセンブルと実行ファイルの生成は外部プログラムである NASM と GNU ld に任せました。

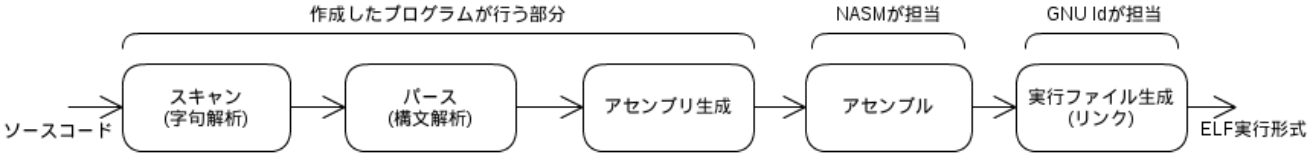


図 6.1　プログラムの動作の流れ

6.3　スキャン (字句解析)

スキャンでは、プログラムをトークンに分割します。トークンとはソースコードにおいて意味のある最小の文字列のことです。例えば次のようなソースコードの場合、

```
int main() {
  int x = 3;
  return x * 2;
}
```

予約語 “`int`”, 識別子 “`main`”, 記号 “`(`”, 記号 “`)`”, 記号 “`{`”, 予約語 “`int`”, 識別子 “`x`”, 記号 “`=`”, 数値 “`3`”, 記号 “`;`”, 予約語 “`return`”, 識別子 “`x`”, 記号 “`*`”, 数値 “`2`”, 記号 “`;`” のようにトークンに分割されます。

[*1] 時間がないので筆者が慣れている言語、パーサジェネレータを選びました。パーサジェネレータでは後でちょっとだけ説明しています。

[*2] 実用的な言語では事前処理や中間コード生成、最適化などが入るかと思います。

制作したコンパイラにおいて、スキャナは Ruby の strscan ライブラリ[*3]を用いてソースコード 6.1 のように実装しました。まず、parse メソッドが呼ばれると引数の str を StringScanner を用いて分割され、クラス変数 queue にトークンの種類とその中身の配列として加えて行きます。Racc において、スキャナはパーサの `next_token` メソッドで次のトークンの種類と中身を渡す必要があるので、クラス変数 queue から 1 つ取り出して返しています。トークンの終了はトークンの種類に false を指定することで表すことになっているので従います。

ソースコード 6.1　スキャナの実装

```
1  require 'strscan'
2  require_relative 'parser_require'
3
4  class Parser
5    RESERVED = %w(void asm char short int long unsigned
6      if else for while do break continue
7      return
8    )
9
10   def parse(str)
11     @queue = []
12
13     s = StringScanner.new(str)
14     reserved = Regexp.new(RESERVED.join('|'))
15
16     until s.eos?
17       case
18       when s.scan(/\s+/)
19         # 空白は読み飛ばす
20       when s.scan(/\/\/.+?\n/)
21         nil
22       when s.scan(reserved)
23         @queue << [s[0].upcase.to_sym, s[0]]
24       when s.scan(/"([^\\"]|\\[^x]|\\x[a-f0-9]{2})*"/) # 文字列
25         str = s[0][1..-2]
26         str = str.gsub(/(\\[^x]|\\x[a-f0-9]{2})/) do
27           eval('"%s"'% $1)
28         end
29         @queue << [:STRING, str]
30       when s.scan(/'([^\\']|\\[^x]|\\x[a-f0-9]{2})*'/) # 文字
31         str = s[0][1..-2]
32         str = str.gsub(/(\\[^x]|\\x[a-f0-9]{2})/) do
33           eval('"%s"'% $1)
34         end
35         @queue << [:NUMBER, str[0].ord]
36       when s.scan(/0x[0-9a-f]+/) # 16進整数
37         @queue << [:NUMBER, s[0][2..-1].to_i(16)]
38       when s.scan(/([1-9][0-9]*|0)/) # 10進整数
39         @queue << [:NUMBER, s[0].to_i]
40       when s.scan(/[a-zA-Z_][a-zA-Z_0-9]*/) # 識別子
41         @queue << [:IDENT, s[0]]
42       when s.scan(/[\(\)\[\]\{\}]/) # 括弧
43         @queue << [s[0], nil]
44       when s.scan(/>>=|<<=/) # 3文字の演算子
45         @queue << [s[0], s[0]]
46       when s.scan(/>>|<<|==|>=|<=|\+\+|\-\-|\->|!=|\|\||&&|\+=|-=|\*=|\/=|%=|&=|\|=|^=/) # 二文字の演算子
47         @queue << [s[0], s[0]]
48       when s.scan(/[<>.&*+\-~!*\/%&^|?:=,]/) # 1文字の演算子
49         @queue << [s[0], s[0]]
50       when s.scan(/;/) # 文の終了
51         @queue << [s[0], s[0]]
52       else
53         p s
54         raise 'ScanError'
55       end
56     end
57     @queue << [false, '$end']
58     do_parse
59   end
60
61   def next_token
62     @queue.shift
63   end
64 end
```

[*3] 正規表現を用いて高速に文字列を分けるためのライブラリです。

6.4　パース (構文解析)

パースではスキャンによりトークンに分割されたソースコードを構文木に変換する作業を行います。先のプログラムの例の場合は、図 6.2 のような構文木が生成されます。

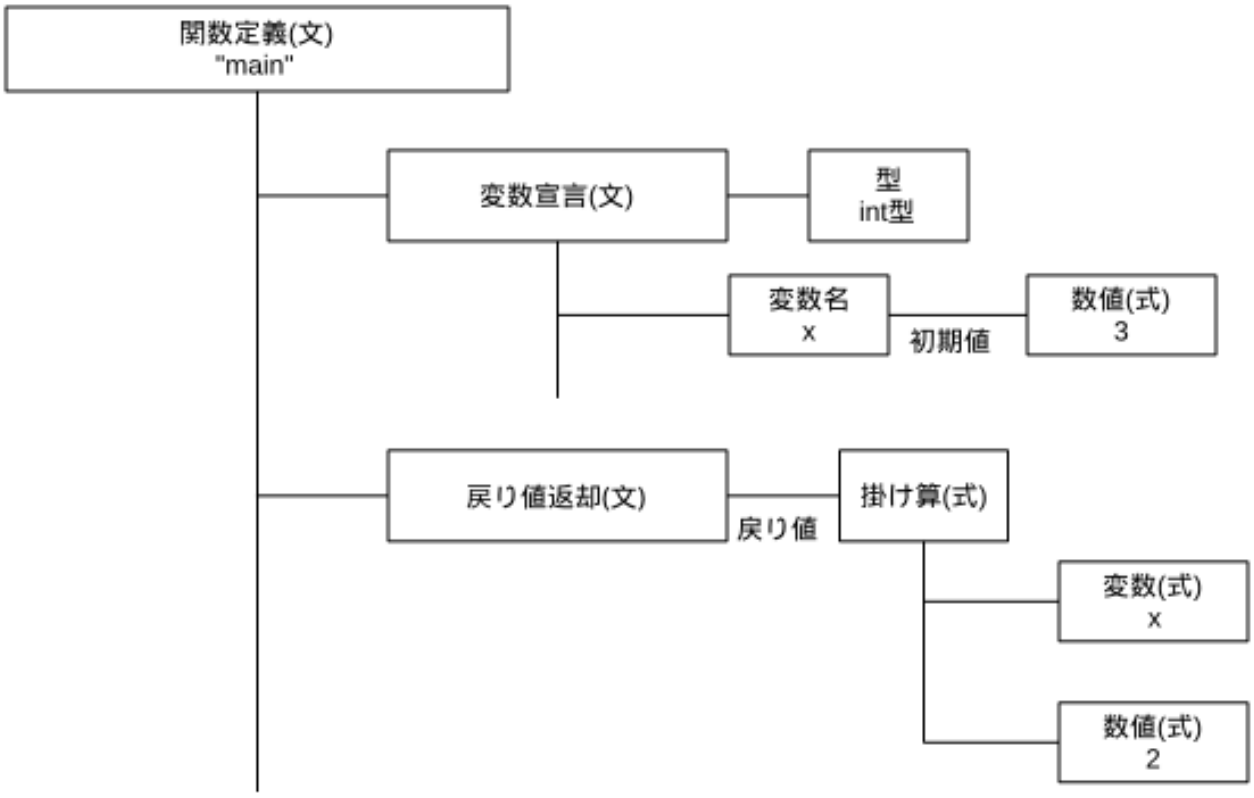


図 6.2　構文木の例

パーサを自分で一から作るのは割と面倒[*4]なので、普通はパーサジェネレータを利用します。パーサジェネレータで最も有名なのは yacc というソフトウェアです。今回用いる Racc も yacc のソースコードも元にして開発されています。

制作したコンパイラのパーサ部はソースコード 6.2 のようになってます。まず演算子の優先順序を `prechigh` と `preclow` に囲まれた部分に優先度が高い順に記述します。`nonassoc` は単項演算子を表し、`left`, `right` は 2 項演算子の結合の方向を表します。最初の `IF`、`ELSE` は演算子ではないのですが、if 文を右結合にするために利用しています。ルールは BNF 記法に似た、次のような形式で記述されています。

[*4] 再帰下降型で作れるならそこまででも無いと思います。

ルール名: 他のルールや Token を空白で並べたもの { マッチした時の Ruby コード }
| あるいは他のルール {マッチした時の Ruby コード}

ルール名は小文字文字列で表し、他の文字列はトークンの種類を表します。例えば、次のルール expr は、2+3、5、(2*5)+3、((((5)))) などにマッチします。マッチした時の Ruby コードは構文木の生成に主に用いられています[*5]。

```
expr: expr + expr
    | expr * expr
    | NUMBER
    | '(' expr ')'
```

このプログラムでは、statement で終わるルールが文を、expression や expr で終わるルールが式を表わします。各 Ruby コードでそれぞれに対応した構文木のノードを作成します。

ソースコード 6.2　パーサの実装

```
1  class Parser
2  prechigh
3    nonassoc ELSE
4    nonassoc IF
5    nonassoc SUFFIX_INCREMENT SUFFIX_DECREMENT
6    nonassoc UPLUS UMINUS PREFIX_INCREMENT PREFIX_DECREMENT POINTER ADDRESSOF
7    left '['
8    left '*' '/' '%'
9    left '+' '-'
10   left '<' '<=' '>=' '>'
11   left '==' '!='
12   right '='
13   left ','
14 preclow
15 rule
16   program: program function {result << val[1] }
17            | {result = [] }
18
19   function: type IDENT '(' arg_list ')' '{' statement_list '}' {
20     result = FunctionNode.new(val[1], val[0], val[3], val[6])
21   }
22
23   statement_list: statement_list statement {result << val[1] if val[1] }
24                 | {result = []}
25
26   statement: expression ';' {result = ExpressionStatement.new(val[0])}
27            | declare ';'
28            | ifstatement
29            | forstatement
30            | whilestatement
31            | asmstatement
32            | RETURN expression ';' {result = ReturnStatement.new(val[1])}
33            | BREAK ';' {result = BreakStatement.new}
34            | CONTINUE ';' {result = ContinueStatement.new}
35            | ';' {result = ExpressionStatement.new(NullExpression.new)}
36
37   asmstatement: ASM '(' STRING ')' ';' {result = AsmStatement.new(val[2])}
38
39   ifstatement: IF '(' expression ')' block = IF {result = IfStatement.new(val[2], val[4], [])}
40              | IF '(' expression ')' block ELSE block = ELSE { result = IfStatement.new(val[2], val[4], val[6])
       }
41
42   forstatement: FOR '(' forexpr ';' forexpr ';' forexpr ')' block {result = ForStatement.new(val[2],val[4],val
       [6],val[8])}
43
44   forexpr: expression | {result = NullExpression.new}
45
46   whilestatement: WHILE '(' expression ')' block {result = ForStatement.new(NullExpression.new, val[2],
       NullExpression.new, val[4])}
47
48   block: statement {result = [val[0]] }
49        | '{' statement_list '}' { result = val[1] }
50
```

[*5] 簡単な計算機の作成とかでは直接値を計算することもあると思います。

```
51  |  arg_list: arg_list_not_empty | {result = []}
52  |
53  |  arg_list_not_empty: func_arg{result = [val[0]]}
54  |                    | arg_list_not_empty ',' func_arg {result << val[2]}
55  |
56  |  func_arg: type IDENT { result = FuncArg.new(val[1], val[0]) }
57  |
58  |  declare: type declare_list { result = DeclareStatement.new(val[0], val[1]) }
59  |
60  |  declare_list: IDENT {result = [[val[0], nil]]}
61  |              | IDENT '=' expression {result = [[val[0],val[2]]] }
62  |              | declare_list ',' declare_list {result += val[2]}
63  |
64  |  type: INT {result = IntegerType.new(32, :signed)}
65  |      | SHORT {result = IntegerType.new(16, :signed)}
66  |      | CHAR {result = IntegerType.new(8, :signed)}
67  |      | UNSIGNED INT {result = IntegerType.new(32, :unsigned)}
68  |      | UNSIGNED SHORT {result = IntegerType.new(16, :signed)}
69  |      | UNSIGNED CHAR {result = IntegerType.new(8, :unsigned)}
70  |      | type '*' {result = PointerType.new(val[0])}
71  |      | type '[' NUMBER ']' {result = ArrayType.new(val[0], val[2]) }
72  |
73  |  expression:
74  |        NUMBER{result = NumberExpression.new(val[0])}
75  |        | IDENT {result = VariableExpression.new(val[0])}
76  |        | function_call
77  |        | '++' expression = PREFIX_INCREMENT {result = AssignmentExpression.new(val[1], AddExpression.new(val
    |      [1], NumberExpression.new(1)))}
78  |        | '--' expression = PREFIX_DECREMENT {result = AssignmentExpression.new(val[1], SubExpression.new(val
    |      [1], NumberExpression.new(1)))}
79  |        | '+' expression = UPLUS {result = val[1]}
80  |        | '-' expression = UMINUS {result = UminusExpression.new(val[1])}
81  |        | '*' expression = POINTER {result = PointerExpression.new(val[1])}
82  |        | '&' expression = ADDRESSOF {result = AddressExpression.new(val[1])}
83  |        | expression '[' expression ']' {result = ArrayExpression.new(val[0], val[2])}
84  |        | expression '*' expression {result = MulExpression.new(val[0], val[2])}
85  |        | expression '/' expression {result = DivExpression.new(val[0], val[2])}
86  |        | expression '%' expression {result = ModExpression.new(val[0], val[2])}
87  |        | expression '+' expression {result = AddExpression.new(val[0], val[2])}
88  |        | expression '-' expression {result = SubExpression.new(val[0], val[2])}
89  |        | expression '>' expression {result = CmpExpression.new(val[0], val[2], val[1])}
90  |        | expression '<' expression {result = CmpExpression.new(val[0], val[2], val[1])}
91  |        | expression '>=' expression {result = CmpExpression.new(val[0], val[2], val[1])}
92  |        | expression '<=' expression {result = CmpExpression.new(val[0], val[2], val[1])}
93  |        | expression '==' expression {result = CmpExpression.new(val[0], val[2], val[1])}
94  |        | expression '!=' expression {result = CmpExpression.new(val[0], val[2], val[1])}
95  |        | expression '=' expression  {result = AssignmentExpression.new(val[0], val[2])}
96  |        | '(' expression ')' { result = val[1] }
97  |        | STRING { result = ConstStringExpression.new(val[0]) }
98  |
99  |  function_call:
100 |        IDENT '(' expression_list ')' { result = FuncallExpression.new(val[0], val[2]) }
101 |
102 |  expression_list: expression_list_not_empty | {result = []}
103 |
104 |  expression_list_not_empty: expression { result = [val[0]]}
105 |                           | expression_list_not_empty ',' expression {result << val[2] }
106 | end
```

このファイル (parsar.y) は次のコマンドによって、Ruby のクラスを含んだソースコード (parsar.rb) に変換出来ます。

```
$ racc parser.y -o parser.rb
```

6.5 アセンブリの生成

パーサにより生成された木を元に x86 アセンブリを生成します。

6.5.1 関数のアセンブリへの変換

関数はソースコード 6.2 の 20 行目周辺を見ると分かる通り、戻り値の型、関数名、引数のリスト (型と名前のリスト)、関数内の文のリストの 4 つの情報を持ちます。戻り値の型は関数のテーブルを作らな

い都合上[*6]、符号付き 32bit 整数のみとしました。

関数をアセンブリに変換する時には、呼び出し規約を定める必要があります。これは、例えば引数をどのように渡すか[*7]、スタックの処理はどのようにするか[*8]などの関数呼び出しに必要なルールのことです。今回は x86 では主に使われている[*9]`cdecl` という呼び出し規約を改変したものを用いました[*10]。

関数のアセンブリの変換の実装はソースコード 6.3 のようになります。これは次のような流れで変換を行っています。

1. 関数内の文が必要とするスタック領域を求め、それぞれにベースレジスタからの相対位置を割り当てます。
2. 必要とされるスタック領域の大きさを元に関数の最初の部分を作り出します。
3. 関数内の変数の情報や return する時のジャンプ先のラベルを保持した context クラスのインスタンスを作成します。
4. context クラスに引数の情報 (名前、型情報とベースレジスタからの相対位置) を代入します。
5. 関数内の各文を順にコンパイルします。
6. 関数の終わりの部分を作り出します。

ソースコード 6.3　関数の実装

```
1  require_relative 'context.rb'
2
3  class FunctionNode
4    attr :name, :type, :args, :statement
5    def initialize(name, type, args, statements)
6      @name = name
7      @type = type
8      @args = args
9      @statements = statements
10   end
11
12   def compile
13     stack_size = 0
14     @statements.each do |statement| # 各文にスタック領域を割り当てる
15       statement.stack = statement.require_stack.map do |req|
16         stack_size += req.size
17         stack_size = (stack_size + req.align - 1) / req.align * req.align
18         -stack_size # ebp からの pos
19       end
20     end
21     stack_size = (stack_size + 3) / 4 * 4 # alignment
22     context = Context.new # 変数､ラベルなどの情報を保持する
23     context.return_label = "label_return_#{@name}"
24     # 引数の処理
25     args.each.with_index do |declare, i|
26       context.add_variable(declare.type, declare.name, +8 + i*4)
27     end
28     # 関数の最初の部分の生成
29     output = ""
30     output << "func_#{@name}:\n"
31     output << "  push ebp\n"
32     output << "  mov ebp, esp\n"
33     output << "  sub esp, #{stack_size}\n" if stack_size > 0
34     ## 各文をコンパイル
35     @statements.each do |statement|
36       output << statement.compile(context)
37     end
38     # 関数の終わりの部分の作成
39     output << "label_return_#{@name}:\n"
40     output << "  mov esp, ebp\n"
41     output << "  pop ebp\n"
```

[*6] 実装を楽にするためです。

[*7] 例: スタックに載せるかレジスタに載せるかなど

[*8] 例: 引数の分スタックをポップするか、呼び出された側が引数のスタックをポップするか

[*9] gcc や MSVC が採用しているはずです

[*10] 呼び出した関数が ebx レジスタを変更しても良いというルールにしました。

```
    output << "  ret\n"
    return output
  end
end
```

6.5.2　文のアセンブリへの変換

各種文のアセンブリへの変換はソースコード 6.4 のような基底クラスを作成し、それを継承する形で行います。`require_stack` メソッドは必要とするスタック領域の大きさとアラインメントのペアの配列を返し、`stack=`メソッドにより関数の変換時にスタック領域が割り当てられる時に呼び出されます。`compile` メソッドは context を引数に取り、変換したアセンブリの文字列を返します。

ソースコード 6.4　文の基底クラス

```
class StatementBase
  def require_stack
    return []
  end

  def stack=(stacks)
    @stack = stacks
  end

  def compile
    raise NotImplementedException.new
  end
end
```

変数宣言文のアセンブリへの変換

変数宣言文は、`require_stack` メソッドに変数の個数だけその型が要求するサイズとアラインメントを返すようにします。要求するサイズは整数型ならその大きさ、ポインタ型は 4byte、配列型なら要素数と配列にする型が要求するサイズの積だけの大きさが必要になります[*11]。`compile` メソッドではまず、context クラスに変数名、型情報やスタック位置情報などを追加し、初期値がある変数について初期値の式を評価し代入します。式の評価と代入については後で詳しく説明します。

ソースコード 6.5　変数宣言文の実装

```
require_relative 'statement_base'

class DeclareStatement < StatementBase
  attr :variables, :type
  def initialize(type, variables)
    @variables = variables
    @type = type
  end

  def compile(context)
    output = ''
    variables.zip(@stack).each do |(name, expr), stack|
      context.add_variable(type, name, stack)
      if expr
        output << expr.compile(context)
        output << "  mov [ebp%+d], eax\n" % stack
      end
    end
    output
  end
end
```

[*11] 実際には型の要求するメモリ容量は `Type` クラスにて計算しています。

if 文のアセンブリへの変換

if 文の実装はソースコード 6.6 のようになります。if 文自身ではスタックは利用しませんが、条件式と条件の真偽により実行されるブロック (文の集合) で必要になるスタックを `require_stack` メソッドで返すようにします。アセンブリへの変換はまず条件式を評価し、その値が 0 以外なら最初のブロック (文の集合) を、そうでないなら次のブロックを実行するようなアセンブリを生成することで行います。

ソースコード 6.6　if 文の実装

```
1  require_relative 'statement_base'
2
3  class IfStatement < StatementBase
4    attr :cond, :then_block, :else_block
5    def initialize(cond_expr, then_block, else_block)
6      @cond = cond_expr
7      @then_block = then_block
8      @else_block = else_block
9    end
10
11   def require_stack
12     req_stacks = @cond.require_stack_all
13     @then_block.map(&:require_stack).each{|s| req_stacks += s}
14     @else_block.map(&:require_stack).each{|s| req_stacks += s}
15     req_stacks
16   end
17
18   def stack=(stack)
19     @cond.stack, stack = stack.partition.with_index{|a,i| i < @cond.require_stack_all.size}
20     (@then_block + @else_block).each do |b|
21       b.stack, stack = stack.partition.with_index{|a,i| i < b.require_stack.size}
22     end
23     raise 'StackSizeError' if stack.size > 0
24   end
25
26   def compile(context)
27     output = @cond.compile(context)
28     rnd = rand(1 << 64).to_s
29     label_else = "label_else_" + rnd
30     label_endif = "label_endif_" + rnd
31     output << "  test eax, eax\n"
32     output << "  je #{label_else}\n"
33     @then_block.each do |b|
34       output << b.compile(context)
35     end
36     output << "  jmp #{label_endif}\n"  if @else_block.size > 0
37     output << "#{label_else}:\n"
38     @else_block.each do |b|
39       output << b.compile(context)
40     end
41     output << "#{label_endif}:\n"
42   end
43 end
```

for 文のアセンブリへの変換

for 文は基本的には if 文と同様に実装します。for 文は初期式、条件式、更新式、繰り返しブロックを持ちますが、まず初期式をアセンブリに変換し、次に繰り返しブロック、更新式、条件式の順に変換します。この時、初期式の直後に条件式まで無条件ジャンプするようにし、条件式で条件を満たした場合に繰り返しブロックの先頭にジャンプするようにします。また、break 文と continue 文を実装するために、関数から渡される context をコピーし、break や continue する時に飛ぶべきラベルを設定して、ブロック内のコンパイルメソッドを呼び出すようにします。

ソースコード 6.7　return 文の実装

```
1  require_relative 'statement_base'
2  require_relative 'null_expression'
3
4  class ForStatement
5    attr :init_expr, :cond_expr, :update_expr, :block
6    def initialize(init_expr, cond_expr, update_expr, block)
7      @init_expr, @cond_expr, @update_expr, @block = init_expr, cond_expr, update_expr, block
8    end
```

```

  def require_stack
    req_stacks = @init_expr.require_stack_all + @cond_expr.require_stack_all + @update_expr.require_stack_all
    @block.map(&:require_stack).each{|s| req_stacks += s}
    req_stacks
  end

  def stack=(stack)
    @init_expr.stack, stack = stack.partition.with_index{|a,i| i < @init_expr.require_stack_all.size}
    @cond_expr.stack, stack = stack.partition.with_index{|a,i| i < @cond_expr.require_stack_all.size}
    @update_expr.stack, stack = stack.partition.with_index{|a,i| i < @update_expr.require_stack_all.size}
    (@block).each do |b|
      b.stack, stack = stack.partition.with_index{|a,i| i < b.require_stack.size}
    end
    raise 'StackSizeError' if stack.size > 0
  end

  def compile(context)
    rnd = rand(1 << 64).to_s
    label_for = "label_for_" + rnd
    label_cond = "label_cond_" + rnd
    label_endfor = "label_endfor_" + rnd
    label_update = "label_update_" + rnd
    next_ctx = Marshal.load(Marshal.dump(context))
    next_ctx.continue_label = label_update
    next_ctx.break_label = label_endfor
    output = ''
    output << @init_expr.compile(context)
    output << "  jmp #{label_cond}\n"
    output << "#{label_for}:\n"
    @block.each do |b|
      output << b.compile(next_ctx)
    end
    output << "#{label_update}:\n"
    output << @update_expr.compile(next_ctx)
    output << "#{label_cond}:\n"
    output << @cond_expr.compile(next_ctx) unless @cond_expr.is_a?(NullExpression)
    output << "  mov eax, 1\n" if @cond_expr.is_a?(NullExpression)
    output << "  test eax, eax\n"
    output << "  jne #{label_for}\n"
    output << "#{label_endfor}:\n"
    output
  end
end
```

while 文のアセンブリへの変換

while 文は for 文の初期式と更新式に何もしない式を代入して実現します。

return 文のアセンブリへの変換

return 文のコンパイルは戻り値となる式を評価した後、context に含まれる関数終了ラベルにジャンプするようなアセンブリを生成するものになります。

ソースコード 6.8　return 文の実装

```
require_relative 'statement_base'

class ReturnStatement < ExpressionStatement
  def compile(context)
    output = @expression.compile(context)
    output << "  jmp #{context.return_label}\n"
    return output
  end
end
```

continue, break の実装

return と同様に context に含まれるループ関連のラベルに無条件ジャンプするようにします。

式文の実装

式文とは、単一の式のみからなる文のことです。例えば `a=3;` や `printf("Hello World");` などが式文となります。単純に式文は式をアセンブリに変換する形で実装します。

ソースコード 6.9　式文の実装

```
require_relative 'statement_base'

class ExpressionStatement < StatementBase
  attr :expression
  def initialize(expression)
    @expression = expression
  end

  def compile(context)
    @expression.compile(context)
  end

  def require_stack
    @expression.require_stack_all
  end

  def stack=(stacks)
    @expression.stack = stacks
  end
end
```

6.5.3　式の実装

式のアセンブリへの変換は、eax レジスタに評価した値が入るようにするというルールで行います。ただし、左辺値については ebx,ecx,edx,ebp を用いて mov の第一引数に渡せるような形 (`[ebp - 10]` など) に変換することにします。また、式には評価した時の型があるため、それを `type` メソッドで返すようにします。

式の基底クラスはソースコード 6.11 のようになります。式は子を持つこと[*12]がとても多いため、子の要求するスタック領域の制御についても基底クラス側で行います。

ソースコード 6.10　式の基底クラス

```
class ExpressionBase
  attr :child
  # 子ノードも含めた全ての必要スタック領域
  def require_stack_all
    stacks = require_stack
    child.each do |c|
      stacks += c.require_stack_all
    end
    return stacks
  end

  def require_stack
    []
  end

  def stack=(stacks)
    size = require_stack.size
    @stack = stacks[0, size]
    stacks = stacks[size..-1]
    child.each do |c|
      size = c.require_stack_all.size
      c.stack = stacks[0, size]
      stacks = stacks[size..-1]
    end
    raise 'Incorrect stacks' unless stacks.size == 0
  end

  def compile(context)
    raise NotImplementedException.new('compile')
  end

  def type(context)
    raise NotImplementedException.new('type')
  end
end
```

[*12] 演算子は全て子を持ちます

数値 (定数) 式の実装

数値式の実装はとても簡単です。eax レジスタにその数字を代入するようなアセンブリを生成します。型はとりあえず無条件に 32bit 符号付き整数とします。

ソースコード 6.11　数値式の実装

```
1  require_relative 'expression_base'
2  class NumberExpression < ExpressionBase
3    def initialize(number)
4      @number = number
5      @child = []
6    end
7
8    def compile(context)
9      "  mov eax, #{@number}\n"
10   end
11
12   def type(context)
13     return IntegerType.new(32, :signed)
14   end
15 end
```

算術演算・比較演算の式の実装

算術演算・比較演算の式のアセンブリへの変換は、まず右辺の式をコンパイルしその eax の値をスタックに保存してから左辺式をコンパイルするということを行います。そのためスタック領域が 4byte[*13]必要になります。

算術演算の型は左辺、右辺とも 32bit 整数であることを前提とし、どちらかが符号無し整数だったら、その計算後の型も符号無し整数ということにします。

比較演算も同様に左辺式、右辺式が 32bit 整数であることを前提にしますが、結果はかならず符号有り整数ということにします。

例えばかけ算の場合ソースコード 6.12 のような実装です。

ソースコード 6.12　かけ算式の実装

```
1  require_relative 'expression_base'
2  require_relative 'require_stack'
3
4  class ArithmeticExpressionBase < ExpressionBase
5    def initialize(child1, child2)
6      @child = [child1, child2]
7    end
8
9    def require_stack
10     [RequireStackSize.new(4,4)]
11   end
12
13   def type(context)
14     raise 'not 32bit integer' unless @child.all?{|a|a.type(context).bits == 32}
15     signed = @child.all?{|a|a.type(context).signed == true}
16     return IntegerType.new(32, signed ? :signed : :unsigned)
17   end
18 end
19
20 class MulExpression < ArithmeticExpressionBase
21   def compile(context)
22     mul = type(context).signed ? 'imul' : 'mul'
23     s = @stack[0]
24     output = ''
25     output = @child[1].compile(context)
26     output << "  mov [ebp%d], eax\n" % s
27     output << @child[0].compile(context)
28     output << "  mov ebx, [ebp%d]\n" % s
29     output << "  #{mul} ebx\n"
30     output
31   end
32 end
```

[*13] 全ての算術演算は 32bit 整数で行われます。

変数式の実装

変数式では、まず context からその名前の変数を探し、mov 命令を用いて変数の位置のメモリの情報を eax に代入するようなアセンブリを生成します。変数が 8bit 整数や 16bit 整数の場合、movsx 命令や movzx 命令を用いて符号拡張して 32bit 整数にします。

また、変数は左辺値なので、`lvalue` メソッドにて代入する時のアクセス方法 (ebp - 相対位置) を返すようにします。

ソースコード 6.13　変数式の実装

```
1  require_relative 'expression_base'
2  require_relative 'type'
3
4  class VariableExpression < ExpressionBase
5    attr :name
6    def initialize(name)
7      @name = name
8      @child = []
9    end
10   def compile(context)
11     variable = context.variable[name]
12     output = ''
13     if variable.type.is_a?(IntegerType)
14       mov = variable.type.signed ? 'movsx' : 'movzx'
15       if variable.type.bits == 32
16         output << "  mov eax, [ebp%+d]\n" % variable.pos
17       elsif variable.type.bits == 16
18         output << "  #{mov} eax, WORD [ebp%+d]\n" % variable.pos
19       elsif variable.type.bits == 8
20         output << "  #{mov} eax, BYTE [ebp%+d]\n" % variable.pos
21       end
22     elsif variable.type.is_a?(PointerType)
23       output << "  mov eax, [ebp%+d]\n" % variable.pos
24     elsif variable.type.is_a?(ArrayType)
25       output << "  lea eax, [ebp%+d]\n" % variable.pos
26     end
27     output
28   end
29
30   def lvalue(context)
31     variable = context.variable[name]
32     if variable.type.is_a?(IntegerType) || variable.type.is_a?(PointerType)
33       output = ''
34       return output, 'ebp%+d' % variable.pos
35     end
36   end
37
38   def type(context)
39     # 整数は暗黙の型変換で 32bit に変更する
40     variable = context.variable[name]
41     if variable.type.is_a?(IntegerType)
42       return IntegerType.new(32, context.variable[name].type.signed ? :signed : :unsigned)
43     elsif variable.type.is_a?(PointerType)
44       return variable.type
45     elsif variable.type.is_a?(ArrayType)
46       return PointerType.new(variable.type.type)
47     end
48   end
49 end
```

代入式の実装

代入式では、まず代入する式を `compile` メソッドでアセンブリに変換し、eax の値をスタック領域に保存します。次に代入される式を `lvalue` メソッドでアセンブリに変換し、得られたアクセサの表すメモリにスタック領域に保存された値を eax に戻して保存します。この時、代入先の型が 8bit、16bit、32bit のいずれかに応じて、al,ax,eax レジスタの内容をメモリに保存するようにします。

ソースコード 6.14　代入式の実装

```
1  require_relative 'expression_base'
2  require_relative 'require_stack'
3
4  class AssignmentExpression < ExpressionBase
```

```
  def initialize(left, right)
    @child = [left, right]
  end
  def compile(context)
    output = @child[1].compile(context)
    output << "  mov [ebp%d], eax\n" % @stack[0]
    o, accessor = @child[0].lvalue(context)
    output << o
    output << "  mov eax, [ebp%d]\n" % @stack[0]
    if @child[0].type(context).is_a?(IntegerType)
      case @child[0].type(context).bits
      when 8
        output << "  mov [#{accessor}], al\n"
      when 16
        output << "  mov [#{accessor}], ax\n"
      when 32
        output << "  mov [#{accessor}], eax\n"
      end
    else
      raise 'Not IntegerType'
    end
    output
  end
  def type(context)
    if @child[0].type(context).is_a?(IntegerType)
      signed = @child[0].type(context).signed == true
      IntegerType.new(32, signed ? :signed : :unsigned)
    else
      @child[0].type(context).is_a?(IntegerType)
    end
  end

  def require_stack
    [RequireStackSize.new(4,4)]
  end
end
```

アドレス演算子式の実装

アドレス演算子は&**式（左辺値）**の形を取り、式のメモリ上の位置をその式の型のポインター型で返却する演算子です。式の `lvalue` メソッドによりアクセサが取得出来るので、lea 命令を用いてアクセサの表すアドレスを eax に代入します。

ソースコード 6.15　アドレス演算子式の実装

```
require_relative 'expression_base'

class AddressExpression < ExpressionBase
  def initialize(child)
    @child = [child]
  end

  def compile(context)
    output, accessor = @child[0].lvalue(context)
    output << "  lea eax, [#{accessor}]\n"
    return output
  end

  def type(context)
    return PointerType.new(child[0].type(context))
  end
end
```

ポインタ演算子、添字演算子の実装

ポインタ演算子`*expr` は添字演算子 `expr[0]` と等価なので、添字演算子のみを作成します[*14]。

添字演算子はポインタ (または配列) がアドレスにそのポインタが表す型の大きさと添字の式の値の積を加えたアドレスのメモリの値を返却するようにして実装します。左辺値にもなり、その時は、アドレス自身を返却するようにします。また、添字演算子は変数の式と同様にメモリ内の値を取得する式なの

[*14] ポインタ型に加減算が出来ない仕様なので

で、型の大きさによって movzx や movsx などの符号拡張命令を用いてメモリの内容を取得するようにします。

ソースコード 6.16　添字演算子式の実装

```
require_relative 'expression_base'

class ArrayExpression < ExpressionBase
  def initialize(expr, index)
    @child = [expr, index]
  end

  def require_stack
    [RequireStackSize.new(4,4)]
  end

  def compile(context)
    p = child[0].type(context).type.size.size
    output = @child[0].compile(context)
    output << "  mov [ebp%+d], eax\n" % @stack[0]
    output << @child[1].compile(context)
    output << "  mov ebx ,[ebp%+d]\n" % @stack[0]
    type = child[0].type(context).type
    if [1,2,4].include?(p)
      accessor = "ebx + #{p} * eax"
    else
      output << "  mov edx, #{p}\n"
      output << "  mul edx\n"
      accessor = "ebx + eax"
    end
    if type.is_a?(IntegerType)
      mov = type.signed ? 'movsx' : 'movzx'
      if type.bits == 32
        output << "  mov eax, [#{accessor}]\n"
      elsif type.bits == 16
        output << "  #{mov} eax, WORD [#{accessor}]\n"
      elsif type.bits == 8
        output << "  #{mov} eax, BYTE [#{accessor}]\n"
      end
    elsif type.is_a?(PointerType)
      output << "  mov eax, [#{accessor}]\n"
    elsif type.is_a?(ArrayType)
      output << "  lea eax, [#{accessor}]\n"
    else
      raise 'NotImplemented'
    end
    return output
  end

  def lvalue(context)
    p = child[0].type(context).type.size.size
    output = @child[0].compile(context)
    output << "  mov [ebp%+d], eax\n" % @stack[0]
    output << @child[1].compile(context)
    output << "  mov ebx ,[ebp%+d]\n" % @stack[0]
    output << "  lea ebx, [ebx + #{p} * eax]\n"
    return output, "ebx"
  end

  def type(context)
    return child[0].type(context).type
  end
end
```

関数呼び出し式の実装

関数呼び出しは、右側の引数から順にスタックにプッシュして行き、呼び出し先関数のラベルを call し、プッシュした分のスタックポインターを調整することで実現されます。今回の言語では関数のラベルは **func_関数名**となっています。また、引数は例え 8bit 整数や 16bit 整数だったとしても、32bit 分プッシュします。

ソースコード 6.17　関数呼び出し式の実装

```
require_relative 'expression_base'
require_relative 'util'

class FuncallExpression < ExpressionBase
  attr name
  def initialize(name, child)
    @name = name
    @child = child
```

```
  end

  def compile(context)
    output = ''
    size = 0
    @child.each.reverse_each do |expr|
      output << expr.compile(context)
      if expr.type(context).is_a?(IntegerType)
        output << Util.convert_to_32(expr.type(context))
      end
      output << "  push eax\n"
      size += 4
    end
    output << "  call func_#{@name}\n"
    output << "  add esp, #{size}\n" if size > 0
    return output
  end
end
```

6.6　アセンブルと実行ファイルの作成

アセンブリの生成により作られたアセンブリを NASM(アセンブラ) によって機械語に変換します。

リンク時に`_start` ラベルがある場所がエントリーポイントとなります。エントリーポイントでは、ソースコード 6.18 のように main 関数を呼び出し、その戻り値で exit する処理を行います。

ソースコード 6.18　エントリーポイントの処理

```
  SECTION .text
global _start
_start:
  pop eax ; eax = argc
  mov ebx ,esp ; ebx = argv
  mov ecx, eax
  shl ecx, 2
  add esp, ecx
  pop ecx
  mov ecx, esp ; ecx = env

  push ecx
  push ebx
  push eax
  call func_main

  mov ebx, eax
  mov eax,1
  int 0x80 ; exit(func_main(argc,argv,env));
```

NASM によって、ELF のオブジェクトを出力するには次のようなコマンドを用います。

```
$ nasm -f elf -o オブジェクトファイル名 生成したアセンブリファイル
```

最後に ELF のオブジェクトファイルから ELF 実行形式を GNU ld を用いて作成します。

```
$ nasm -o 生成する実行ファイル名 -m elf_i386 オブジェクトファイル
```

6.7　いろいろ動かしてみる

6.7.1　標準入出力

この言語、標準ライブラリと呼ばれるようなものがないため自分で入出力を記述する必要があります。Unix のシステムコールを利用して標準入出力するようなプログラムを書きました。

ソースコード 6.19　標準入出力

```
int write(int fd, char *buf, int count) {
  asm("mov eax, 4");
  asm("mov ebx, [ebp + 8]");
  asm("mov ecx, [ebp + 12]");
  asm("mov edx, [ebp + 16]");
  asm("int 0x80");
}
int read(int fd, char *buf, int count) {
  asm("mov eax, 3");
  asm("mov ebx, [ebp + 8]");
  asm("mov ecx, [ebp + 12]");
  asm("mov edx, [ebp + 16]");
  asm("int 0x80");
}
int strlen(char *a){
  int len = 0;
  for(len = 0; a[len] != 0; ++len){}
  return len;
}

int putchar(char a) {
  write(1, &a, 1);
}

int puts(char *a) {
  write(1, a, strlen(a));
  putchar(10);
}

// 改行が来るまでbuf に書き出す
int gets(char *buf, int size) {
  char s;
  int read = 0;
  for(;read<size;++read) {
    if(read(0, &s, 1) == 0) return read;
    if(s == '\0') return read;
    buf[read] = s;
  }
  return read;
}

// 整数を一つ出力する
int print(int x) {
  if(x == 0) putchar('0');
  else if(x < 0){
    putchar('-');
    print(-x);
  }else{
    if(x < 10) {
      putchar('0' + x);
    }else{
      print(x / 10);
      putchar('0' + x % 10);
    }
  }
}
```

6.7.2 Hello World

ソースコード 6.20　helloworld

```
int main(int argc, char** argv, char** env) {
  puts("Hello, World!!");
}
```

```
$ ruby bin/generate-elf.rb program/hello.c
$ ./a.out
Hello, World!!
```

ちなみに、作られた HelloWorld プログラムは余計なライブラリがリンクされていないため、strip すると 760bytes と結構小さいプログラムになります。

6.7.3　FizzBuzz

ソースコード 6.21　FizzBuzz

```
1  int main(int argc, char** argv, char** env) {
2    int i;
3    for(i = 1; i <= 100; ++i) {
4      if(i % 15 == 0) {
5        puts("FizzBuzz");
6      }else if(i % 3 == 0) {
7        puts("Fizz");
8      }else if(i % 5 == 0) {
9        puts("Buzz");
10     }else {
11       print(i);putchar(10);
12     }
13   }
14 }
```

これも正しく動作して、実行ファイルは 1280bytes でした。

6.8　終わりに

プログラミング言語を作ると難しいという印象を持っているかもしれませんが、意外と簡単です。なので、言語作りたいなーと思っている人はまず作ってみてはいかがでしょうか。楽しいバグとの戦いが待ってます。

第 7 章

ACM-ICPC アジア地区予選東京大会に参加したお話

darui1113

概要
国内外にはプログラミングコンテストが数多く存在する。その中でも歴史の古い「ACM 国際大学対抗プログラミングコンテスト (ACM International Collegiate Programming Contest) のアジア地区予選東京大会に参加した話。

7.1　はじめに

ACM 国際大学対抗プログラミングコンテスト (以下略称で ACM-ICPC と呼ぶ) とは ACM(Association for Computer Machinery) という計算機学会が主催するプログラミングコンテストで、同じ大学の 3 人一組でチームを作り、プログラミングの能力を競う。ACM-ICPC の世界大会では、各地で行われた地区大会を勝ち抜いたチームが集まり、競い合う。また、世界大会には一つの大学から一つのチームしか参加できないため、大学対抗プログラミングコンテストと位置づけられている。

日本では、地区大会に出場するために、まずは国内予選と呼ばれるインターネット上で行われる予選を突破しなければいけない。国内予選では、3 時間で 6 問程度の問題が出題され、正答数を競う。その中から 30 チーム程が選ばれ、地区大会に参加することになる。選抜方法については細かい規定があるが、ここでは省略させていただく。

地区大会では 5 時間で 10 問程度の問題が出題され、これらの問題をどれだけ多く、早く解けるかを競う。

7.2　チームに加わるまで

弊大学には情報工学工房という講義がある。この講義では、情報工学に関するいくつかの題材について取り組むグループに別れ、そのグループごとに課題を進めていく。この題材の一つに「競技プログラミング」というものがある。このグループで、ACM-ICPC の国内予選に参加するためのチームを組むことになったらしいのだが、3 人 1 組でのチームを組むとなると、人が足りなくなったようなのである。その埋

め合わせとして、情報工学工房の別のグループへ、ACM-ICPC の国内予選への参加に興味がある人の募集がかかったところに、私が釣られてしまったわけである。その後、紆余曲折あったが、一緒に埋め合わせで参加した先輩と相談した結果、私の方が Java の適正があると判断され、Java で問題を解いていくチームに参加することになった。正直なところ、Java なんて 2 年ほど前に触ってそれっきりだったのでチームでやっていけるのか非常に不安だった。

7.3　チーム結成〜国内予選

Java を使うので、チーム名は「You and JAVA」になった[*1]。一度、チーム練習を行って、それから国内予選に臨んだ。

しかし、チームメイトのお二人が私よりも断然プログラミングができる方達だったため、私の国内予選中の仕事といえば、あらかじめ問題を読んでおいて簡単な説明をするくらいであった（しかし、国内予選の問題は日本語のためあまり意味が無かった）。チームメイトがコーディングしてる様を眺めていることしかできなかったと言ってもよい。

7.3.1　国内予選突破

国内予選終了直後の順位を確認すると、なんと選抜確定圏内に居た模様。これによりアジア地区大会に参加することになったのだが、私は国内予選でこの有様だったため、アジア地区大会で何も出来ずに惚けている様が想像できた。

7.4　東京大会まで

東京大会に参加することになったので、それまで出来る限りアルゴリズムを勉強しなければならなくなった訳です。とりあえず、プログラミングコンテスト チャレンジブック (通称：蟻本) を開いて勉強を始めた。しかし、動的計画法に取り組み始めたところで辛くなってきた。特に問題に対してどうやって動的計画法を適用したら良いのかということを主に悩んで苦労していた（なお、これを執筆している現在も解決には至っていない）。

こうして満足いくレベルまで練習できずに東京大会を迎えたわけであった。

7.5　東京大会

アジア地区である東京大会は 2 日に分けて行われた。また、3 日目にはご褒美的なイベントがあったのですが、都合により私は欠席した（主に講義と実験のせい）。

[*1] あなたと JAVA,
今すぐダウンロード

7.5.1　一日目

一日目は実際にコンテスト当日に使う環境に慣れることと、JavaChallenge なるイベントが行われた。その前に会場内では全部英語でやり取りされるのが一番辛かった (二日目も同様)。

環境に慣れることは適当にプログラム書いたり、設定調整の手順を確認したりして終了。翌日には端末が初期化されるので、翌日に再設定することになる。JavaChallenge では何か良くわからないゲームの AI のプログラミングをした。サンプルプログラムをたった 5 行程度書き換えたプログラムで上位決定戦 (?) に参加できたことは秘密。AI のプログラミングは調布祭で開催される UECda というイベントでも行われるので興味のある読者は行ってみてはどうだろうか (ステマ)。私も参加したかったが、諸事情により出場させるプログラムを準備できなかったため、今回は参加することを見送った。

7.5.2　二日目

二日目は朝 9 時から会場に集まり、5 時間の間、問題を解くことに集中した・・・わけではない。実際に問題の解放を考え、そのコードを書くことは、私以外のメンバー 2 人が行い、私はひたすら問題文 (英語) を読み、他の二人がすぐに解法を考えることに取りかかれるように、その問題文を日本語で要約していた。英語辛い (どうせ今後も使うので勉強しましょう)。

7.5.3　その後

無事にコンテストを終えて、パーティーのようなものを終え、即帰宅。何やら翌日にイベントがあったらしいが、実験と重なるのでそちらは欠席。

7.6　今後

問題を解くことは楽しかったので、これから腰を据えて勉強していきたいものです。こういったプログラミングコンテストの問題を解く為の Web サイト (オンラインジャッジ) はたくさんあるので、少し紹介だけしておきたい。

- AIZU ONLINE JUDGE 日本語対応で ACM-ICPC 以外のコンテスト問題もある
- PKU JudgeOnline 全部英語だけど問題がたくさんある
- TopCoder SRM なんだかんだで手を付けれてないので今後にでも

7.7　終わりに

こんなチラ裏記事になってしまったが、コンテストの方は今後も続けて行きたい。

がんばってアルゴリズム勉強していこうな。

参考文献

[1] http://icpc.iisf.or.jp/

百萬石 2014 -秋- © 電気通信大学 MMA

2014 年 11 月 20 日　初版第一刷発行
2014 年 11 月 23 日　第二版第一刷発行　【本書の無断転載を禁ず】

著　者　Tachibana Waita, zico, f_taka6, masawada
darui1113, nomeaning, hiro1357 / @eserver_dip_jp >ワ<
表　紙　g_nootoko
編集者　masn19
発行者　電気通信大学 MMA
発行所　電気通信大学 MMA 部室
〒182−8585 東京都調布市調布ヶ丘 1−5−1 サークル会館 208 号室
http://www.mma.club.uec.ac.jp/
印刷所　電気通信大学 MMA 部室
製本所　電気通信大学 MMA 部室