バイナリ生成環境「daisy-tools」実験報告

— 培養・進化で簡単な ELF バイナリを生成！ —

［著］大神祐真

技術書典 8（2020 年春）新刊
2023 年 5 月 20 日　ver 1.1

■免責

本書は情報の提供のみを目的としています。
本書の内容を実行・適用・運用したことで何が起きようとも、それは実行・適用・運用した人自身の責任であり、著者や関係者はいかなる責任も負いません。

■商標

本書に登場するシステム名や製品名は、関係各社の商標または登録商標です。
また本書では、™、®、© などのマークは省略しています。

はじめに

本書をお手にとっていただきありがとうございます！[*1]

本書では、前著「バイナリ生物学入門」で紹介した「実行バイナリを生物に見立てる設計」へ「進化」の仕組みを追加することで、実行バイナリを進化により生み出せるようにした「daisy-tools」というツールを紹介する本です。

プログラマの三大美徳の一つに「怠惰」がありますが、その究極は「寝て、起きたらできあがっている」だと思います。本書で紹介する daisy-tools は、「コードを書くことでバイナリを生成」する従来の「プログラミング」による方法とは違い、「バイナリを進化させる」事で任意のバイナリを生成します。それによりエンジニアは、進化を評価するスクリプトを書くだけで、実行バイナリ本体に関するコードは書かずに目的のバイナリを生成できます。(と言っても、現実的に生成できるのは簡単なもののみですが。)

本書の構成

本書では daisy-tools について以下の 3 章構成で紹介します。

- 第 1 章 daisy-tools のコンセプトとアーキテクチャ
  - daisy-tools の基本コンセプトと、バイナリを進化させる仕組みを紹介します。
- 第 2 章 daisy-tools の使い方
  - 使い方を実行例と共に紹介します。
- 第 3 章 実験結果紹介
  - 「終了ステータス 0 で正常終了する」・「'A' を出力して正常終了する」という 2 つの ELF バイナリの生成について実験結果を紹介します。

PDF/HTML 版や本書の更新情報について

本書の PDF/HTML 版は筆者のウェブページで公開しています。

- `http://yuma.ohgami.jp`

本書の内容について訂正や更新があった場合もこちらのページに記載しますので、何かおかしな点などあった場合はまずはこちらのページをご覧ください。

[*1] こんな何だかよく分からない本を、本当にありがとうございます。。

目次

第 1 章　daisy-tools のコンセプトとアーキテクチャ

この章では「daisy-tools」の基本コンセプトとアーキテクチャを紹介します。

1.1　コンセプト

普段、実行バイナリを作る際はには、ソースコードを書いてそれをコンパイルする、という流れが一般的です。スクリプト言語等の場合も、結局は「ソースコードを書く」事は変わらないと思います。

「daisy-tools」(というより前著で紹介した「バイナリ生物学」) のコンセプトは、バイナリを生物のように生成することです。生物学で理解されている生物が増殖・進化する仕組みを、バイナリの生成に応用します。

1.2　本書でやること

前著では、独自 OS のメモリ上で、生物と見立てた関数が、運動・代謝・成長・増殖・死の 5 つの振る舞いをするデータ構造と実装例を説明しました。

本書では、細胞分裂時に DNA が突然変異する仕組みを導入します。そして、代謝・運動の結果に応じて各個体へ「適応度」を設定することで、目的の方向へ進化できるようにします。

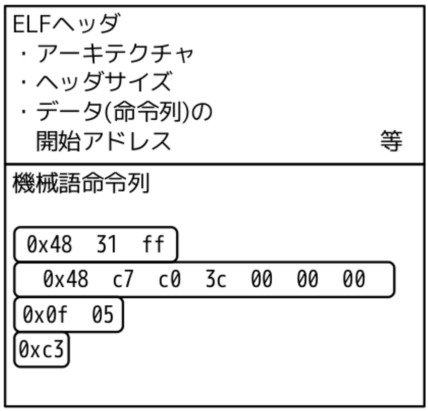
なお、前著では独自 OS 上のアプリの関数を対象としていましたが、本書では、Linux 上で実行可能な ELF バイナリを生成します。

♣ ELF バイナリについて

本書では、実行バイナリを培養するシステムである daisy-tools を使って ELF バイナリを生成します。

まずは、生成する ELF バイナリがどのようなものであるかをざっくりと紹介します。

本書で扱う ELF フォーマットの要素は「EFL ヘッダ」と「機械語命令列」のみで、簡単に説明すると図 1.1 の通りです。



▲図 1.1: 本書で使う ELF の要素

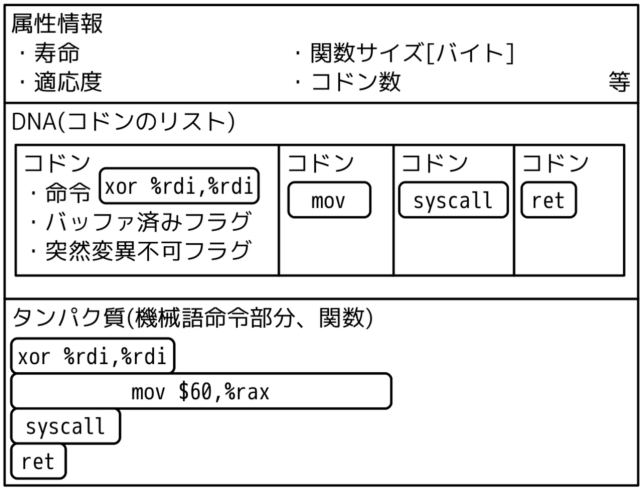
重要なのは、ELF バイナリには実行される機械語命令列自体と、それが ELF バイナリ中のどこにあるかを示す情報が入っているということです。実行させたい機械語命令を並べ、そこへのアドレスを適切に示すようにヘッダを用意すれば ELF バイナリは作れるわけです。

図 1.1 の機械語命令箇所をアセンブラで示すと図 1.2 の通りです。図示の際の分かりやすさのため、以降はアセンブラで図示するようにします。ただ、実際に扱うのはそれに対応する機械語です。

▲図 1.2: 機械語部分をアセンブラで図示

♣ 実行バイナリを細胞に見立てる

前著でも紹介した通り、実行バイナリを細胞に見立てることで、生物としての振る舞いをさせます。構造は図 1.3 の通りです。



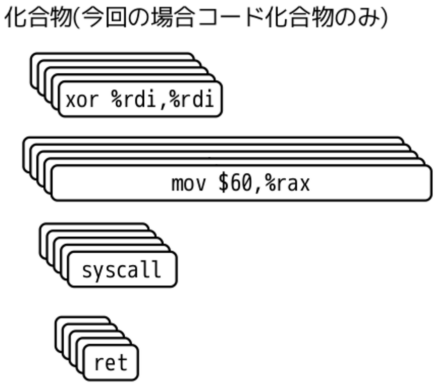
▲図 1.3: 細胞の構造

構成要素については前著でも紹介した通りですが、突然変異と進化の仕組みを入れるために以下の要素を追加しています。

- 属性情報: 適応度
- DNA: 突然変異不可フラグ

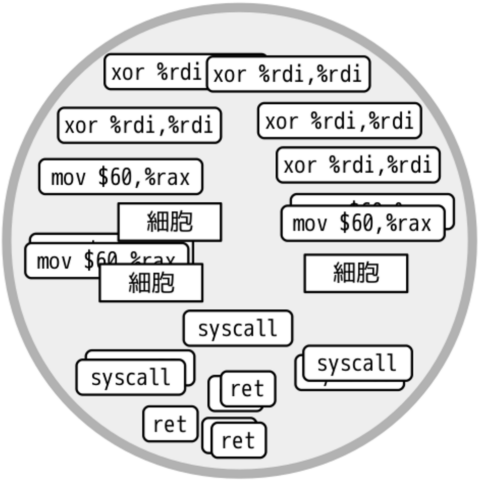
なお、前著では 1 命令をさらに分解したバイト列を単位としていましたが、本書では 1 命令を最小単位とします。そのため、本書でのコード化合物は 1 命令そのものとなります。また、DNA を構成するコドンも 1 命令単位となります。

今回対象とする実行バイナリは入力データを取らないため、細胞の他に存在する物質はコード化合物のみです (図 1.4)。



▲図 1.4: コード化合物

daisy-tools における実行バイナリ培養は、これらの細胞とコード化合物をシャーレに入れて培養させるイメージです (図 1.5)。



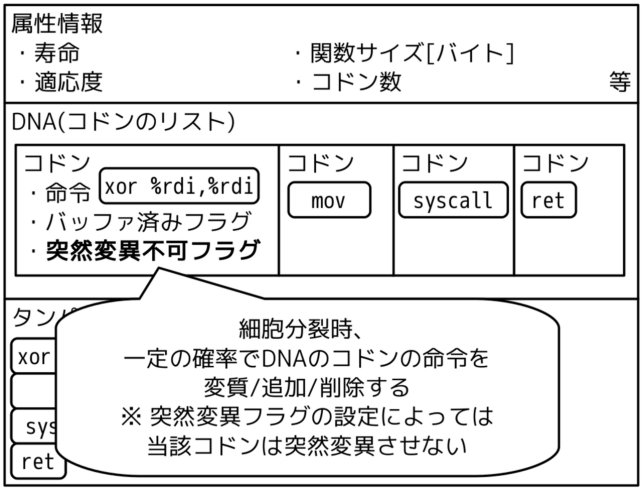
▲図 1.5: 実行バイナリを培養するイメージ

♣ 突然変異の仕組み

突然変異は、細胞分裂時にあらかじめ設定された確率で発生します。

突然変異の単位はコドンですので「1 命令」の単位です。「対象とするコドン」・「突然変異の仕方 (追加/変更/削除)」・「変異後の命令」をそれぞれランダムに選び、例えば「DNA 内の 2 番目のコドン」を対象に「mov 命令」へ「変更」する、といったように突然変異を行います。

「突然変異不可フラグ」を設定しておくと、そのコドンは突然変異の対象にしないようにすることができます (図 1.6)。これにより、「最後の ret 命令だけは突然変異で変更/削除されないようする」等といった設定が可能です。



▲図 1.6: 突然変異不可フラグ

【コラム】体が大きくなると寿命も延びる

daisy-tools では、DNA へコドンが追加される突然変異が起きた場合、ある一定の比率で寿命を延ばすようにしています。

それは、DNA を長くする方向の突然変異が細胞の生存戦略においてデメリットにならないようにするためです。

前著でも紹介した通り、細胞が分裂により増殖する流れは以下の通りです。

1. DNA の各コドンが示すコード化合物を取り込む (各サイクルで最大 1 個ずつ)
2. DNA の全コドンのコード化合物が揃ったら細胞分裂

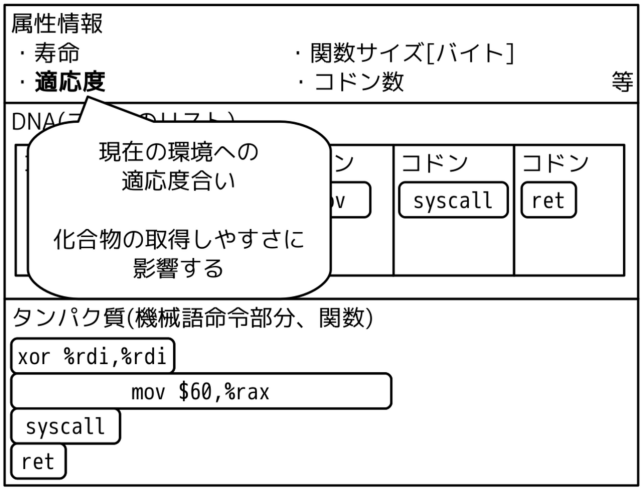
この時、各サイクルで寿命も 1 ずつ減るため、DNA が長くなった際に寿命が短い頃のままだと、長い DNA は各サイクルで 1 つずつ化合物を取り込んでも、短い DNA より残せる子の数は減ってしまいます。

このような不平等が起こらないように、DNA が長くなった場合 (体が大きくなった場合) は、寿命も延ばすようにしています。

♣ 意図した方向へ進化を導く仕組み

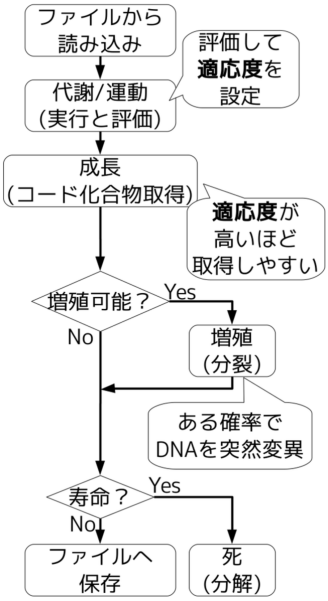
ただ突然変異をさせ続けるだけでは、目的の実行バイナリへ近づけていくのは難しいです。

そのためには、正に「海に餌が少なくなったから陸へ上がった」というような進化の方向性を付ける仕組みが必要です。そのために細胞の構造に追加した「適応度」というパラメータを使います (図 1.7)。



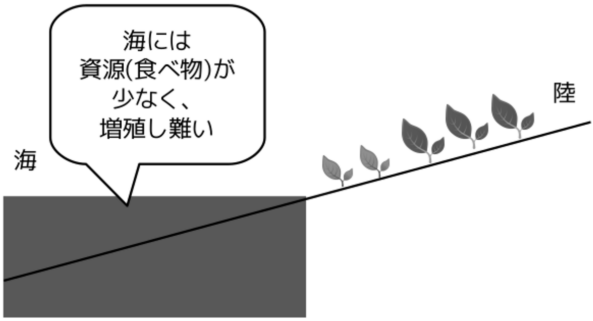
▲図 1.7: 細胞の適応度パラメータ

適応度は各周期で細胞を評価する際に設定されるパラメータです。そしてこの適応度を、各周期で細胞が化合物を取り込める確率に使います (図 1.8)。自身の DNA に合致するコード化合物を揃えれば分裂が行えるため、適応度が高いほど化合物を多く取り込め、増殖も行いやすくなる訳です。

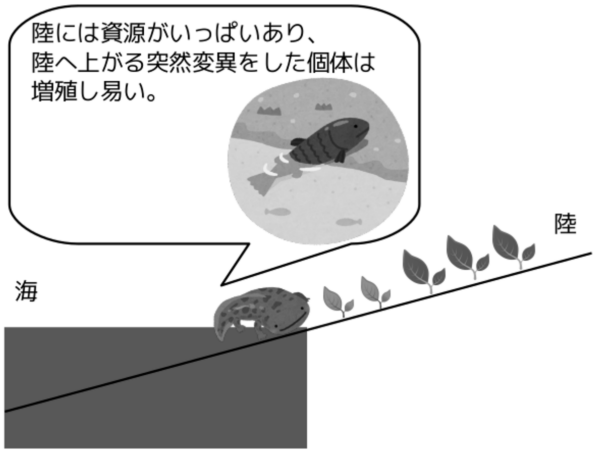


▲図 1.8: 細胞周期での適応度の使われ方

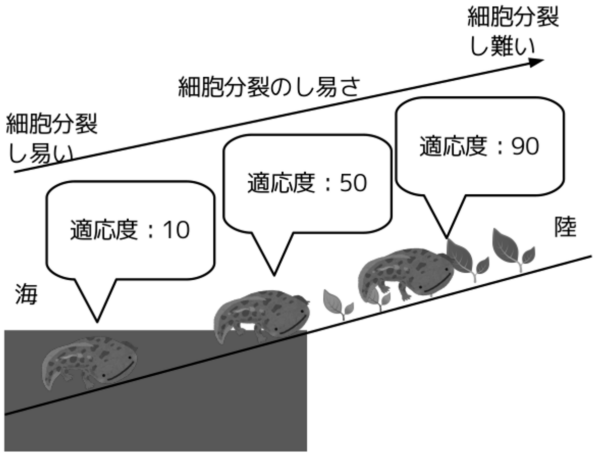
イメージとしては図 1.9、図 1.10、図 1.11 の通りです。



▲図 1.9: 適応度による進化 (1)



▲図 1.10: 適応度による進化 (2)



▲図 1.11: 適応度による進化 (3)

本書で紹介する「daisy-tools」では、このようにして実行バイナリの培養・進化を行います。

第 2 章　daisy-tools の使い方

この章では daisy-tools の基本的な使い方を説明します。

2.1　主に使用するツール類

daisy-tools は様々なツールで構成されています。主に使用するツールは表 2.1 の通りです。なお、これらのファイルは daisy-tools の GitHub リポジトリの Releases ページからダウンロードできます (後述)。

▼表 2.1: daisy-tools の基本的なツール

ツール名	機能/役割
dsy-sysenv	周期的な細胞の実行を行う。 daisy-tools のシステムを担うメインの実行バイナリ。
dsy-cell2elf	daisy-tools の細胞形式のバイナリを ELF バイナリへ変換する。
dsy-dump-cell	細胞形式バイナリの内容を独自のリスト形式 あるいは JSON 形式でダンプする。
dsy-name	生成される細胞バイナリ・化合物バイナリのファイルの 名前付けを行うスクリプト。(ユーザー定義)
dsy-eval	各周期で dsy-sysenv から呼び出され、指定された細胞を評価し 適応度を返すスクリプト。(ユーザー定義)

`dsy-sysenv` はカレントディレクトリにリスト 2.1 の構成で配置された細胞ファイル・化合物ファイルを認識します。

▼リスト 2.1: dsy-sysenv が認識するカレントディレクトリ構成

```
.
|-- bin
|   |-- "dsy-"で始まる各ツールを配置
```

```
|   |-- dsy-sysenv
|   |-- dsy-cell2elf
|   |-- dsy-dump-cell
|   |-- dsy-name  <- ユーザー定義スクリプト
|   |-- dsy-eval  <- ユーザー定義スクリプト
|   `-- ...etc
|-- cell
|   `-- 細胞ファイルを配置
|-- code
|   `-- コード化合物ファイルを配置
|-- data
|   `-- データ化合物ファイルを配置(現バージョンでは未使用)
`-- samples
    `-- 細胞ファイル・コード化合物ファイルを生成するサンプルツール群
```

リスト 2.1 のディレクトリ構成のカレントディレクトリ上で `dsy-sysenv` を実行すると、存在する全ての細胞ファイルに対して「代謝/運動」・「成長」・「増殖」・「死」といった生物の振る舞いを繰り返し実行します。

`dsy-name` と `dsy-eval` は、 `dsy-sysenv` から必要に応じて呼び出されるユーザー定義のスクリプトで、 `dsy-sysenv` 実行前に予め用意しておく必要があります。

以降ではそれぞれのファイルについて説明します。

♣ dsy-name について

`dsy-sysenv` が新たに細胞ファイルやコード化合物ファイルを生成する際に呼び出すスクリプトで、それらのファイル名を決めるために使用します。

- 第 1 引数に"cell"あるいは"code"を指定して呼び出される
  - 名付ける対象が細胞ファイル/コード化合物ファイルのどちらなのかを指定
- `dsy-sysenv` は標準出力に出されたものをファイル名として扱う
- 指定されたファイル名が既に存在する場合、 `dsy-sysenv` は `dsy-name` を繰り返し呼び出す
  - 現時点 (2020 年 2 月現在) では、存在しないファイル名が返されるまでリトライし続ける実装

生成される細胞や化合物には固有の名前が付けられると面白いかなと思い、ファイル名の名付けの部分は別スクリプトへ切り出しました。

ただ、動作上、ファイル名は何でも良いので、例えばリスト 2.2 の様にランダムな文字列を返すようにしてしまっても構いません。

▼リスト 2.2: dsy-name の実装例

```
#!/bin/bash

echo $RANDOM | md5sum | cut -c-10
```

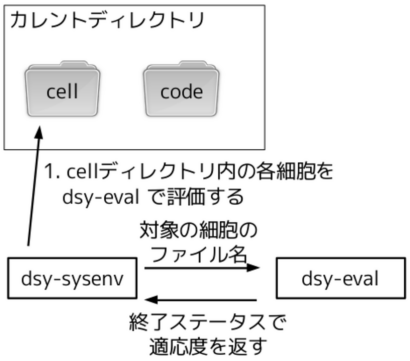
なお、リスト 2.2 は `samples/dsy-name` に用意してあります。これをそのまま使う場合は、 `bin` ディレクトリに `dsy-name` という名前でこのファイルのシンボリックリンクを作成すれば良いです。(もちろん Windows 等の環境で実行する場合はファイルコピーでも構いません。)

♣ dsy-eval について

`dsy-eval` は、各周期で各細胞の適応度を決めるために細胞毎に呼び出されるスクリプトです。ユーザーはこのファイルで細胞をどのように評価するかを定義します。

- 第 1 引数に対象の細胞ファイル名を指定して呼び出される
  - "cell"から始まるパス指定ではなく、細胞ファイル名のみの指定
- `dsy-sysenv` は `dsy-eval` の終了ステータスの値を適応度として扱う
  - 適応度の範囲は 0 から 100
  - 100 の適応度が返された場合、 `dsy-sysenv` は目的のバイナリが生成できたとして実行終了する
  - 0 から 100 以外の終了ステータスを返された場合、 `dsy-sysenv` はエラー終了する

評価の流れを図示すると図 2.1 の通りです。



▲図 2.1: dsy-eval を使用した評価の流れ

例えば「細胞がエラーなく終了できるか (終了ステータスが 0 であるか)」のみを評価するようなものはシェルスクリプトでリスト 2.3 のように記述します。

▼リスト 2.3: 終了ステータスの成否のみで評価

```
#!/bin/sh

set -uex

if [ $# -ne 1 ]; then
        echo "Usage: $0 CELL_FILE" 1>&2
        exit 200
fi

cell_file=$1
bin/dsy-cell2elf $cell_file ${cell_file}.elf
chmod +x ${cell_file}.elf
./${cell_file}.elf && fitness=100 || fitness=50
rm ${cell_file}.elf
exit $fitness
```

`dsy-cell2elf` は第 1 引数で指定された細胞ファイルを ELF バイナリへ変換し、第 2 引数で指定されたパスへ保存します。

リスト 2.3 では、 `dsy-cell2elf` で変換した ELF バイナリを試しに実行し、正常終了したら適応度として終了ステータス 100 を、異常終了したら適応度として終了ステータス 50 を返すようにしています。

適応度は、細胞が化合物を取り込める確率として使用されるものなので、0 を返してしまうと、その細胞は今後一切化合物を取り込めなくなり、細胞分裂も行えなくなってしまう点にご注意ください。また、0 で無くとも、あまりにも低い適応度ばかりを与えると適応度 100 の細胞が生まれる前に環境の細胞が全滅してしまう点も注意が必要です。経験的には最低適応度は 50 くらいにしておくと良い感じです。

リスト 2.3 のスクリプトは、 `samples/dsy-eval-exit_0` にあります。これをそのまま使う場合は、このファイルのシンボリックリンクかコピーを `bin` ディレクトリに `dsy-eval` という名前で作成してください。

本書で `dsy-eval` として使用する評価スクリプトは全て `samples` ディレクトリに置いてあります。

2.2　使用例: 簡単な評価スクリプトで試す

ここでは、使い方の紹介として、前節の評価スクリプトを使用して、次章でも実験

結果として紹介する「正常終了するだけの ELF バイナリ生成」へ向けた環境を作ってみます。

♣ ツールをダウンロード

daisy-tools リポジトリの Releases ページでビルド済みバイナリを ZIP アーカイブで配布しています。

- daisy-tools リポジトリの Releases ページ
  - `https://github.com/cupnes/daisy-tools/releases`

ダウンロードしたらお好きな場所へ展開してください。`dsy-work` というディレクトリができあがります。このディレクトリ名は変更しても構いません。

なお、現行のビルド済みバイナリの動作環境は表 2.2 の通りです。

▼表 2.2: 動作環境

ライブラリ/ソフトウェア	バージョン
Linux	特になし。ELF が実行できれば OK。
GNU C ライブラリ (libc6)	2.24
bash	特になし。bash のシェルスクリプトが実行できれば OK。

標準的な Linux 環境であればディストリビューション問わず動作すると思います[*1]。

ただ、必須要件ではないのですが、ログのほとんどは syslog に出力しますので、syslog が使えない環境 (WSL 等) では syslog 出力のログは見ることができません。

♣ 細胞ファイルを配置

進化のベースとなる最初の状態の細胞ファイルを `cell` ディレクトリへ配置します。

本書で進化のベースとして使用する細胞ファイルは以下のいずれかです。

- `cell/initial`
  - `ret` 命令のみの細胞
  - 自身を構成する命令には突然変異不可フラグ設定済み
  - `samples/create-init-cell` で生成可能
- `cell/exit`
  - 終了ステータス 0 で終了するための機械語命令群と `ret` 命令の細胞

[*1] 筆者が使用しているのは Debian GNU/Linux 9 (stretch) です。

    - 自身を構成する命令には突然変異不可フラグ設定済み
    - `samples/create-exit-cell` で生成可能

それぞれのツールを実行すると、`cell` ディレクトリにそれぞれのファイル名で細胞ファイルを生成します。なお、daisy-tools では今の所、ファイル名で何らかの判断は行っていないので、ファイル名は変更しても構いません。

例えば、`initial` の細胞ファイルを `cell` ディレクトリへ配置する場合は、以下のように `samples/create-init-cell` を実行します。

```
$ samples/create-init-cell
$ ls cell/
initial
```

♣ コード化合物ファイルを配置

コード化合物ファイルは、細胞が成長や増殖する際に使用するコード化合物に当たるファイルです。表 2.3 の構成の 16 バイトのファイルです。

▼表 2.3: コード化合物ファイルの構成

オフセット	内容	長さ [バイト]
0	命令の長さ [バイト]	8
8	機械語命令	8

なお、`dsy-sysenv` ではコード化合物ファイル 1 つを 1 命令として扱います。そのため、分割してほしくない複数の命令がある場合、1 つのコード化合物ファイルに並べておけば、1 命令として扱われます。

本書で生成する「正常終了するだけの実行バイナリ」と「文字'A' を出力する実行バイナリ」については、生成するために必要なコード化合物ファイルは、それぞれ以下のサンプルツールで生成できます。

- 正常終了するだけの実行バイナリ
    - 必要なコード化合物ファイルは `samples/create-exit-codes` で生成できる
- 文字'A' を出力する実行バイナリ
    - 必要なコード化合物ファイルは `samples/create-puta-codes` で生成できる

それぞれのツールは共に、実行時に「各命令で何個ずつコード化合物ファイルを生成するか」をコマンドライン引数で指定します。そして実行すると、指定した個数ず

つ、各命令のコード化合物ファイルが `code` ディレクトリへ生成されます。

例えば、「正常終了するだけの実行バイナリ」に必要なコード化合物ファイルを生成するには以下の通りです。(各命令 100 個ずつ生成)

```
$ samples/create-exit-codes 100
+ '[' 1 -ne 1 ']'
+ num_codes=100
+ mkdir -p code
++ seq -w 100
+ for i in $(seq -w ${num_codes})
...
$ ls -1 code
ret_001
..
ret_100
syscall_60_001
...
syscall_60_100
xor_rdi_rdi_001
...
xor_rdi_rdi_100
```

このサンプルでは各コード化合物ファイルのファイル名を各命令になぞらえて付けていますが、細胞ファイル同様にファイル名は何でも構いません。

なお、実行ログからも分かる通り、 `samples/create-exit-codes` のサンプルツールはシェルスクリプトです。簡単なコマンドでコード化合物ファイルを生成する際に参考にしてみてください。

♣ ユーザー定義スクリプトを配置

`bin` ディレクトリに `dsy-name` と `dsy-eval` のユーザー定義スクリプトを配置してください。

前述の通り、 `dsy-name` は `samples/dsy-name` のシンボリックリンクかコピーで構いません。

```
$ ln -s ../samples/dsy-name bin/dsy-name
$ ls -l bin/dsy-name
lrwxrwxrwx 1 yohgami yohgami 19  2月 12 14:59 bin/dsy-name -> ../samples/dsy›
›-name
```

`dsy-eval` は生成したいバイナリによって変える必要があります。前節で説明した評価スクリプトで試す場合は以下のようにシンボリックリンクを作成します。

```
$ ln -s ../samples/dsy-eval-exit_0 bin/dsy-eval
$ ls -l bin/dsy-eval
lrwxrwxrwx 1 yohgami yohgami 26  2月 12 15:02 bin/dsy-eval -> ../samples/dsy›
›-eval-exit_0
```

♣ dsy-sysenv を実行

以上で準備は完了です、 `dsy-sysenv` を実行すると環境の動作が始まり、細胞の生命活動が始まります。

```
$ bin/dsy-sysenv
```

`dsy-sysenv` を実行すると `running` というファイルがカレントディレクトリに作成されます。中断したい場合はこのファイルを削除してください。すると、その周期の全ての動作を終えたところで `dsy-sysenv` は終了します。

```
$ ls running
running
$ rm running
```

`dsy-sysenv` は現在の環境の状態を `cell` と `code` ディレクトリに保存しています。再開したい場合は単に `dsy-sysenv` をもう一度実行すれば良いです。

そして、適応度が 100 の細胞が生まれると、 `dsy-sysenv` はその細胞ファイルをカレントディレクトリに `out.cell` という名前で保存し、その周期を実行し終えた後、プログラム自体を終了します。

第 3 章　実験結果紹介

この章では daisy-tools を使用して「終了ステータス 0 で正常終了するだけの実行バイナリ」と「文字'A' を出力して正常終了する実行バイナリ」のそれぞれを生成する実験結果を紹介します。

3.1　実験 1: 正常終了するだけのバイナリ生成

前章で紹介した簡単な評価スクリプトで、「何もせず正常終了 (ステータス 0 で exit) するだけの ELF バイナリ」を生成させてみます。

♣ 実験条件

実験条件として、`cell`・`code` の各ディレクトリと評価スクリプト `dsy-eval` の状態を説明します。

まず、`cell` と `code` の各ディレクトリ内のファイルは、それぞれ `samples/create-init-cell` と `samples/create-exit-codes` で生成しました。各命令に対するコード化合物の数は 100 個ずつです。

そして、`dsy-eval` は前章で紹介したもので、`samples` ディレクトリのファイルとしては `dsy-eval-exit_0` です。

♣ 補足: code ディレクトリ内の各命令について

今回の実験で `code` ディレクトリに配置したコード化合物ファイルについて補足します。

`code` ディレクトリに配置したファイルが、「正常終了するだけのバイナリ」を構成する機械語命令列の部分です。ここに配置した命令が突然変異により細胞の中で組み合わされ、「正常終了するだけのバイナリ」が生成される事を期待します。

`code` ディレクトリに配置した各命令について、対応するアセンブラとともに列挙すると表 3.1 の通りです。

▼表 3.1: 実験 1:code ディレクトリに配置する命令一覧

	機械語	**アセンブラ**
	[終了ステータス 0 で exit]	
(1)	48 31 ff	xor %rdi,%rdi
(2)	48 31 c0 b0 3c 0f 05	xor %rax,%rax; mov $60,%al; syscall
	[return]	
(3)	c3	ret

表 3.1 の (1) から (3) の各行が各コード化合物ファイルに相当し、それぞれ 100 個ずつ `code` ディレクトリに配置しています。

内容について簡単に説明すると、(1) と (2) が終了ステータス 0 で `exit` システムコールを呼び出すための命令です。プログラムが終了する際には `exit` システムコールを呼ぶ必要があり、このシステムコールは第 1 引数 (RDI) に終了ステータスをとります。そのため (1) で RDI を `xor` 命令でゼロクリアし終了ステータスに 0 を設定しています。

そして、(2) で `exit` システムコールを呼び出しています。ここで、前章で紹介した「1 つのコード化合物ファイルに複数命令を列挙すれば 1 命令として扱われる」機能を使っています。システムコールを呼び出す `syscall` 命令を実行すると、RAX に格納された番号のシステムコールがカーネル内で呼び出されるのですが、ここでは「RAX のゼロクリア」・「RAX へ 60 を加算 (60 を設定)」・「システムコール呼び出し」までを 1 命令としています。

「システムコール番号の設定」と「システムコール呼び出し」を 1 命令として扱われるようにしているのは、 `dsy-sysenv` を実行するホスト側で意図しないシステムコールが呼び出されないようにするためです。細胞の突然変異では、機械語命令列のどこにどの命令が挿入/削除/変更されるかが分かりません。どの細胞も評価の過程で一度実行することになるので、RAX が未設定の状態でシステムコール呼び出しが行われることで、例えば意図しないファイルアクセスなどが行われると困るため、明示的に指定したシステムコールのみが使われるように、「システムコール番後設定」と「システムコール呼び出し」を 1 命令としています。

最後に (3) の `ret` 命令は単に関数の `return` です。初期状態の細胞は `ret` 命令のみの細胞であるため、自分自身の細胞分裂のために `ret` 命令のコード化合物も置いています。

♣ 実験結果

ここでは 10 回の実験を行い、結果は表 3.2 の通りでした。

▼表 3.2: 実験 1:実験結果

	総時間	総サイクル	結果
1	42 秒	23	生成成功
2	59 秒	23	生成成功
3	57 秒	21	生成成功
4	59 秒	33	生成成功
5	12 秒	10	生成成功
6	1 分 6 秒	42	生成成功
7	33 秒	23	生成成功
8	1 分 17 秒	32	生成成功
9	51 秒	43	生成成功
10	1 分 3 秒	29	生成成功

10 回の実験全てで終了ステータス 0 で正常終了する実行バイナリを生成することができました。

では、生成された実行バイナリの機械語命令の並びは期待値通りだったのでしょうか？ 確認してみると、全てのケースで、生成された実行バイナリの機械語命令部分がリスト 3.1 の通りでした。

▼リスト 3.1: 実験 1:生成されたバイナリ

```
0:   48 31 c0                xor    %rax,%rax
3:   b0 3c                   mov    $0x3c,%al
5:   0f 05                   syscall
7:   c3                      retq
```

第 1 引数である RDI をゼロクリアする `xor` 命令が無いです。実は RDI は明示的にゼロクリアしなくとも、動作上はゼロになっているのですね。

♣ 考察

1 分前後でできあがるというのは、体感としてはあっという間です。「仕掛けて、寝て、起きたらできあがっている」の理想に対しては、全く申し分ないです。

ただ、今回の場合、「60 番のシステムコール (exit) を呼び出す」という 1 つ分のコドンが追加されれば良く、1 回の突然変異で目的のバイナリへ至ることができるものでした。

より複雑な実行バイナリではどうでしょうか。それは次の実験で確認してみます。

3.2　実験 2: 'A' を出力する実行バイナリ生成

続いて、前節よりは少し複雑な「文字'A' を画面に出力して正常終了する実行バイナリ」の生成を行ってみます。

♣ 実験条件

`cell` と `code` の各ディレクトリは、それぞれ `samples/create-exit-cell` と `samples/create-puta-codes` で生成したものを配置しました。各命令に対するコード化合物の数も実験 1 同様に 100 個ずつです。

評価スクリプト `dsy-eval` は、`samples/dsy-eval-puta_0` を使用していて、内容はリスト 3.2 の通りです。

▼リスト 3.2: 実験 2:評価スクリプト

```
#!/bin/bash

set -uex

TMP_NAME=.test

if [ $# -ne 1 ]; then
        echo "Usage: $0 CELL_FILE" 1>&2
        exit 200
fi

cell_file=$1
bin/dsy-cell2elf ${cell_file} ${TMP_NAME}.elf
chmod +x ${TMP_NAME}.elf
./${TMP_NAME}.elf >${TMP_NAME}.out || true
if grep -q 'A' ${TMP_NAME}.out; then
        exit 100
fi
exit 50
```

評価の仕方としては実験 1 と同様に「成功 (適応度 100) か失敗 (適応度 50) か」のみです。今回の場合は「実行してみて文字'A' を出力できたら適応度 100」そして「それ以外は 50」としました。

♣ 補足: code ディレクトリ内の各命令について

`code` ディレクトリに配置した各命令について、対応するアセンブラとともに列挙

すると表 3.3 の通りです。

▼表 3.3: 実験 2:code ディレクトリに配置する命令一覧

	機械語	アセンブラ
	['A' を標準出力へ出力]	
(1)	48 c7 c7 01 00 00 00	mov $1,%rdi
(2)	48 c7 c6 78 00 40 00	mov $0x400078,%rsi
(3)	48 83 c6 22	add $34,%rsi
(4)	48 c7 c2 01 00 00 00	mov $1,%rdx
(5)	48 31 c0 b0 01 0f 05	xor %rax,%rax; mov $1,%al; syscall
	[終了ステータス 0 で exit]	
(6)	48 31 ff	xor %rdi,%rdi
(7)	48 31 c0 b0 3c 0f 05	xor %rax,%rax; mov $60,%al; syscall
	[return]	
(8)	c3	ret

「['A' を標準出力へ出力]」の部分では、`write` システムコール (システムコール番号:1) を使用して標準出力へ'A' を出力しています。

`write` システムコールは、第 1 引数 (RDI) にファイルディスクリプタ、第 2 引数 (RSI) に書き込むデータのアドレス、第 3 引数 (RDX) にデータのバイト数をとります。

そこで、(1) で標準出力のファイルディスクリプタである 1 を RDI へ設定し、(2) と (3) で文字'A' の先頭アドレスを RSI へ設定 (後述)、(4) でバイト数 1 を RDX へ設定した上で、(5) でシステムコール番号 1 のシステムコールを呼び出します。

(3) で RSI へ格納しているアドレスは、全 ASCII 文字が並んでいる領域のアドレスです。write システムコールが出力するデータをアドレスで渡す必要があるため、ASCII のキャラクターマップを ELF バイナリに埋め込むことで対応しています。(4) で RSI に足し合わせている値はこのキャラクターマップの中の文字'A' へのオフセットです。

ELF バイナリに埋め込んでいる ASCII マップについては daisy-tools リポジトリ内の `dsy-cell2elf.c` を見てみてください。

(6)・(7)・(8) は実験 1 と同じものです。

♣ 実験結果

実験結果は表 3.4 の通りです。

延べ 3 日以上の時間をかけ、9 万回以上のサイクルを実施しましたが、'A' を出力するバイナリへ進化することはありませんでした。

▼表 3.4: 実験 2:実験結果

	総時間	総サイクル	結果
1	3 日 16 時間	96,685	適応度 100 に至らず中止

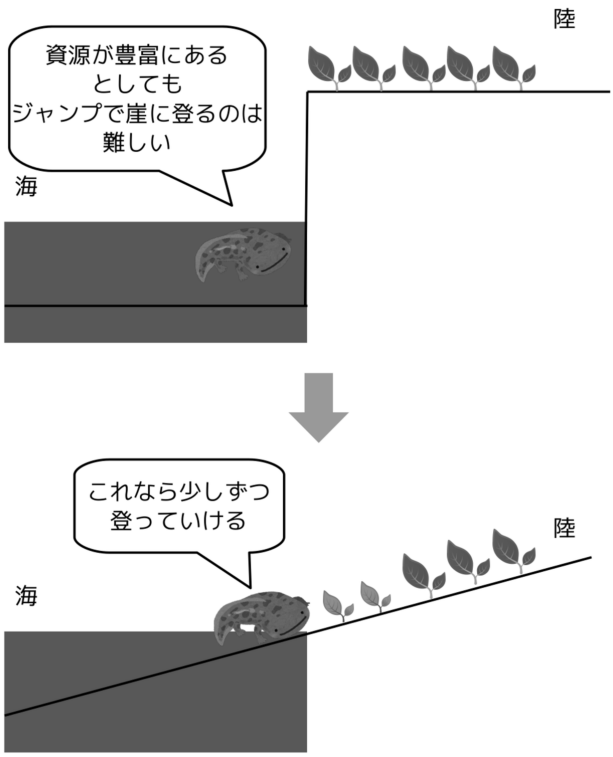
♣ 考察

どれだけ時間をかけても期待した進化に至らない要因は、期待する進化の幅が高すぎる事にあると考えられます。

実験 1 の場合、「ret 命令の前に exit システムコールの命令列が追加される」という 1 度の突然変異で期待する状態に至ることができたため、「期待する結果か否か」という「1 か 0 か」の評価でも進化を導くことはできました。

しかし、実験 2 の場合、2 つ以上の命令 (列) をある程度決まったルールで並べる必要があるため、「1 か 0 か」という評価では、評価に至るまでに必要な進化の幅が高すぎるのです。これではランダム性に頼るばかりで、進化を導くことはできません。

イメージとしては図 3.1 のような感じです。例え海に食料が少なかったとしても、切り立った崖の上にジャンプするような進化は難しいです。そうではなく、できる限りなだらかな斜面で徐々に食料を得る機会が増えるような環境だと、小さなステップの進化を繰り返すことで目的地へたどり着くことができます。



▲図 3.1: 急な崖上へジャンプするような進化は難しい

3.3　実験 3: 'A' を出力する実行バイナリ生成 (改良版)

徐々に適応度が上がるような評価スクリプトを用意して実験を行ってみます。

♣ 実験条件

`cell` と `code` の各ディレクトリの初期状態は実験 2 と同じです。

評価スクリプト `dsy-eval` は、徐々に適応度が上がるように工夫してリスト 3.3 のようにしてみました。(同じものが `samples/dsy-eval-puta` にあります)

▼リスト 3.3: 実験 3:評価スクリプト

```
#!/bin/bash

set -uex

TMP_NAME=.test

if [ $# -ne 1 ]; then
        echo "Usage: $0 CELL_FILENAME" 1>&2
        exit 200
fi

cell_filename=$1

bin/dsy-cell2elf ${cell_filename} ${TMP_NAME}.elf
chmod +x ${TMP_NAME}.elf
exit_status=0
./${TMP_NAME}.elf >${TMP_NAME}.out || exit_status=$?

if [ ${exit_status} -ne 0 ]; then
        # 突然変異でret命令が変なところに入る等、
        # 終了ステータスが0ではない状態で終了するようになってしまった場合
        # 適応度は0とする
        exit 0
fi

bin/dsy-dump-cell -j ${cell_filename} >${TMP_NAME}.json
num_codns=$(jq .attr.num_codns ${TMP_NAME}.json)

if [ ${num_codns} -eq 0 ]; then
        # コドン数=0 は何かがおかしい
        exit 200
fi

# コドン数に応じて適応度を上げる
# 適応度 = 6 * コドン数 + 32
fitness=$(((6 * num_codns) + 32))
if [ ${fitness} -gt 90 ]; then
        # コドン数に応じた適応度では90以上にはならない
        fitness=90
fi

if grep -q 'A' ${TMP_NAME}.out; then
        # 'A'を出力できた場合、適応度100
        fitness=100
elif [ $(wc -c ${TMP_NAME}.out | cut -d' ' -f1) -gt 0 ]; then
        # 'A'でなくとも何かを出力したなら適応度に9点加点する
```

```
        fitness=$((fitness + 9))
fi

exit ${fitness}
```

主にコドン数 (命令数) に応じて、最大 90 まで適応度を上げるようにしてみました。また、'A' でなくとも何かを出力できた場合はそれも評価するようにしています。

♣ 実験結果

実験結果は表 3.5 の通りです。

▼表 3.5: 実験 3:実験結果

	総時間	総サイクル	結果
1	3 時間 38 分	1,805	生成成功
2	5 時間 56 分	2,743	生成成功
3	6 時間 15 分	2,988	生成成功

期待する状態へ進化を導くことができました。
生成された実行バイナリは、例えば 1 回目の実験ではリスト 3.4 の通りです。

▼リスト 3.4: 実験 3:1 回目の実験結果

```
  0:   48 83 c6 22             add    $0x22,%rsi
  4:   48 c7 c7 01 00 00 00    mov    $0x1,%rdi
  b:   48 c7 c6 78 00 40 00    mov    $0x400078,%rsi
 12:   48 31 ff                xor    %rdi,%rdi
 15:   48 c7 c7 01 00 00 00    mov    $0x1,%rdi
 1c:   48 83 c6 22             add    $0x22,%rsi
 20:   48 c7 c2 01 00 00 00    mov    $0x1,%rdx
 27:   48 31 c0                xor    %rax,%rax
 2a:   b0 01                   mov    $0x1,%al
 2c:   0f 05                   syscall
 2e:   48 31 ff                xor    %rdi,%rdi
 31:   48 31 c0                xor    %rax,%rax
 34:   b0 3c                   mov    $0x3c,%al
 36:   0f 05                   syscall
 38:   c3                      retq
```

♣ 考察

3 日以上かけても到達できなかった状態へ、最短で 3 時間半で到達できたことは大きな進歩です。やはり進化を導くには評価を適切に行う必要があると考えられます。

ただ、生成されたバイナリは目的の振る舞いは達成しますが、無駄な命令をいくつも蓄えています。1 回目の実験結果のみ掲載しましたが、無駄に蓄えている命令が違うだけで 2 回目と 3 回目の実験結果も同様でした。

今回設定した評価スクリプトの場合、命令数が多い、すなわち体が大きい個体ほど栄養を取り入れる機会を多く得る世界である事に加え、前述の通り、命令数の多さに比例して寿命を延ばすようにしているため、このような結果になるのは仕方ないと言えます。対策としては、例えば、適応度 100 の個体が現れた時点で環境を止めるのではなく継続するようにして、適応度 100 の個体のみ「コドン数 (命令数) が少ないほど適応度をより上げる[*1]」というように評価の仕方を切り替える事が考えられます。

[*1] 100 を超える適応度を化合物を取得する確率としてどう扱うかが問題ですが、例えば「1 つは確実に取得できるとした上で、100 を超えている分の値を、さらにもう一つ取得できる確率として扱う」事が考えられます。

おわりに

ここまで本書をお読みいただきありがとうございました！ 本書で前著の設計に「進化」の仕組みを導入し、ついに実行バイナリを生成できるようになりました。その方法は生物学にアイディアを得たオリジナルのものですが、ジャンルとしては「遺伝的アルゴリズム」の一種になるのかなと思います。本来の遺伝的アルゴリズムは生物の遺伝の仕組みのみをモデル化しており、わざわざ生物が生きる振る舞いを模倣したりはしませんが、「バイナリ生物学」では「生きていると言い張れる」だけの振る舞いをバイナリにさせた上で進化の仕組みを導入します。そのため、より生々しく生物の振る舞いを模倣している手法であるかと思います。

3 つの実験を通して、評価スクリプトを適切に用意すれば実行バイナリを生成できる事を示しました。生成された実行バイナリは、評価スクリプトの評価は満たしつつも、機械語のコード列としては「想定よりも短いコードで実現可能」だったり、「余計なコードがいくつも付いていた」り、といったものが生成されていた点は興味深いポイントです。些細なことでもありますが、前者は、設計者が想定していた以上の最適化が行われていたとも言えます。後者は、実行上は問題ないですが、余計なゴミがくっついているという意味ではバグ (?) かもしれません。バグに対しては評価方法を改善すれば良いので、「人間の想定以上の解が産まれ得る」という点が「進化」によるバイナリ生成の重要ポイントじゃないかなと思います。

また、このバイナリ生成方法では、エンジニアは評価スクリプトしか書かないため、それをテストコードと捉えるなら、「エンジニアがテストコード書かない問題」を解決しているとも (一応) 言えます。加えて評価の仕方はアーキテクチャ非依存にできるので、「一つの評価スクリプトで複数のアーキテクチャ向けの実行バイナリを生成する」といったことも可能です。

課題は「実用的な実行バイナリは生成できるのか」あるいは「そういうものではない」としても「何に使えるのか？」という所ですね。実用的な実行バイナリは今回実験したようなバイナリよりももっと規模が大きく多機能であるため、例えば「別々の環境で単機能毎に進化した細胞を組み合わせて多細胞生物にする」といったような新しい進化の仕組みが必要そうです。なお、daisy-tools は「バイナリ」であれば生成対象は「実行バイナリ」以外も可能な方法であるため、現状の実装でも何か解決できる問題はあるかもしれません。

なお、「daisy-tools」という名前の元ネタはガイア理論の「デイジーワールド」です。「バイナリを生きている様に産み出す」という発想元には小さい頃に遊んだ「シムアース」があったりします。

バイナリ生成環境「daisy-tools」実験報告

培養・進化で簡単な ELF バイナリを生成！

2020 年 2 月 29 日　ver 1.0 （技術書典 8 新刊）
2023 年 5 月 20 日　ver 1.1 （技書博指定の奥付を追加）

著　者　大神祐真
発行者　大神祐真
連絡先　yuma@ohgami.jp
　　http://yuma.ohgami.jp
　　@yohgami (https://twitter.com/yohgami)
印刷所　日光企画

(powered by Re:VIEW Starter)