知的システム構成論課題
堀先生担当分

航空宇宙工学専攻修士一年
荒居秀尚

2018 年 7 月 25 日

1　対象とするシステム

今回のレポートにおいては、人間を支援する人間-機械系と、機械が自動的に仕事をする自動システムの組み合わせのあり方に関する考察を「プログラミング」という分野に関して行う。議論の簡略化のため、「プログラミング」という言葉は本稿においては、「言語を用いて計算機に動作をさせること」と定義する。

本稿では以下の流れで議論を進める。

1. プログラミングと自動化の歴史
2. 現在のプログラミングおよびその活動支援の自動化の例
3. プログラミングにまつわる自動化の組み合わせ

2　プログラミングと自動化の歴史

2.1　プログラミング自体の自動化の歴史

プログラミングと自動化は切っても切り離せない関係にある。そもそも現状では、「自動化」と言った場合ほぼ確実にプログラミングという行為が介在することになる。もちろんハードウェア的に、すなわち機構的な工夫によって自動化を行う例も少なくないが、コンピュータの登場以後は産業構造を大きく変えるほどに自動化が進んだことを考えると、現代においては自動化においてはほぼ確実にコンピュータに人間の担っていた役割の何がしかを任せるという行為が発生しそれを行う手段こそがプログラミングである。

プログラミングという行為はそれ自体の自動化も進めながら発展してきたことも特徴的である。当初は 0 と 1 の羅列をコーディングする必要があったものを、ニーモニックを用いて人間により理解しやすいようにしたアセンブリ言語はそれまで人間の頭の中で行われていた、メモリ操作やループなどの基本的な演算操作を 0 と 1 に直す、という「翻訳作業」を自動化したと言える。
また、その後登場した FORTRAN などの高水準言語はコンパイラに、「低水準言語への翻訳、あるいは分解」を任せることで人間の「目的のために問題をより小さな記述単位まで落としこむ」という作業の一部を自動化

した。
また、「関数」「サブルーチン」「メソッド」などと呼ばれる言語仕様の開発は同じ記述を繰り返し書くという作業を減らすことで人間の負担を減らした効率化の例であると言えるし、型のシステムは本来行ってはいけない計算をしないように人間が気を付けなければいけなかった部分を計算機によるチェックで肩代わりした例と言える。
オブジェクト指向の登場は、プログラムの設計をより容易にし「工程の複雑性を取り除く」という形で自動化をしたとも言える。

このように、プログラミングは自動化とは切っても切り離せない関係にあり、それ自体も自動化されることを避けられない存在である [1]。

2.2 プログラミングを支援する自動化の歴史

プログラミングそれ自体がその一部を自動化され続けながら発展してきた一方でプログラミングという活動を支援する系も自動化され続けてきたといえる。

例えば、プログラミングにおいてはいまや必須といっても差し支えないテキストエディタであるが、プログラミング言語登場の頃は、パンチカードに穴を開ける穿孔機がその役割を担っていることを考えると多くの自動化がなされてきたというのは容易に理解できる [2]。今では、シンタックスハイライト機能によって人間の認知機能の一部を補完したり、コード補完機能によって処理を記憶する、という認知作業を自動化によって肩代わりしているとも言える。
また、git や subversion などのバージョン管理システムは人間による編集履歴の管理という部分を自動化したものと言える。
近年のクラウドの発達は、計算資源の確保という部分の一部を自動化した一方で、複数の計算機の協調の必要性を今まで以上に明確にし、結果として分散実行の自動化という自動化需要も産んだ。

プログラミング活動の支援と、プログラミングそのものはしばしば境界がかなり曖昧なためほとんど融合しているものとなることもある。たとえば、アプリケーションフレームワークは、アプリケーションの作成を支援するソフトウェアであって、アプリケーションの作成において決まりきった部分の自動化を行い、ユーザは自動化できない部分のみを記述すれば良いというものであるが、自動化自体はプログラミングそのものを自動化しているとも言える。

3 現在のプログラミングおよびその活動支援の自動化の例

3.1 フレームワーク

例として Web アプリケーションフレームワークの一つである。Ruby on Rails を挙げることにする [3]。Ruby on Rails は俗に「フルスタックな」フレームワークと呼ばれるように Web アプリケーション作成のあらゆる部分を包括して支援するようなフレームワークである。その名前にもあるように、ユーザが「まるでレールに乗っているかのように」アプリケーションを作成できるように設計されている。このフレームワークにおける

自動化についていくつか例を挙げる。

まず、テンプレート作成機能についてであるが、Ruby on Rails では 1 コマンドで設計に必要なファイル郡やフォルダの雛形が用意できるようになっている。これは、「Web アプリケーションでこのような機能を作りたかったらこのようなファイルが必要である」という経験値を結晶化した機能であると言える。
また、ユーザは HTML を直に全て 1 から書く必要はなく、部分的な HTML テンプレートに記述を行うだけで ERB と呼ばれるテンプレートエンジンがアセンブルした HTML を書き出す。また、よく使われる HTML スニペットは Rails の中で定義されたメソッドとして提供されているが、これは Rails のスニペットを書くことで HTML スニペットが生成されることに相当しプログラミング自体の自動化を行っているとも言える。
また、OR マッパーというソフトウェアにより、SQL コードを Ruby のコードでラップしてデータベース内のテーブルやレコードに対してオブジェクト指向的取り扱いができるようにされている。これは、Ruby のコードを書くことによって背後で SQL のコードが生成されていることに相当しプログラミングそのものの自動化と言っても差し支えない。

以上のような各種の自動化機能によって全体としてユーザのコード記述量を減らしているのが Web アプリケーションフレームワークであるが結果として、「ユーザがライブラリを使ってコードを書き、アプリケーションを作成する」という一連の流れを「フレームワークがコードを書き、人間が必要な部分を埋めてアプリケーションを作成する」という流れに変化させた点が大きな特徴であると言える。

3.2 グラフィカルプログラミング

グラフィカルプログラミング、あるいはビジュアルプログラミングはプログラミング支援技術の一種であるが、同時に人間の理解のしやすいデータ記述の方式から中間的なテキストベース言語ソースの出力、あるいは機械語までコンパイルする、などプログラミングそのものの自動化の例とも言える。

ビジュアルプログラミング自体はプログラミングという行為の参入障壁を引き下げていると考えられ、自動化によってプログラミング活動を支援している好例である。例としては、Mathworks 社の製品である Simulink[4] やゲームエンジン Unreal Engine に組み込まれている Blue Print[5] などがあげられる。
Simulink は、制御工学で用いられるブロック線図を作成することで C 言語のソースコードが生成されるようになっており科学技術計算においてよく用いられている。また、Blue Print は C++ 言語で行えるほぼすべての処理を記述することができるようになっており、型のエラーなどのミスを事前に防ぐという意味でも人間の活動を支援している自動化ツールの一例と言える。

3.3 テキストエディタ、IDE

テキストエディタ、または IDE などはプログラミング活動を支援する人間-機械系の代表的なものであると言える。現在ではテキストエディタはプログラミングにおいては必須の道具であり、プログラマの生産性を向上させる様々な仕組みが組み込まれていることが多い。

テキストエディタや IDE などに含まれる機能の例を挙げると、シンタックスハイライト、コード補完、コー

ド実行、スタイル提案、各種システムとの連携などが挙げられる。
シンタックスハイライトはコードの可読性を大幅に向上させるほか、コード補完はコーディング作業の効率化や人間の記憶の補助を行う。コードの実行は出力を確認しながらのコーディングができるように支援をし、スタイルの提案はコードの可読性向上や一貫性、複数人での開発の補助をしている。

近年では、IDE によるコードの分析機能なども発達しており変数名の提案など、よりプログラミング自体にも関わるような部分も自動化されつつあるのも注目に値する [6]。

3.4 Web フロントエンドの自動生成

近年の深層学習技術の流行に伴い、Web 制作の現場においてコーディングの自動化を試みた例が存在する [7]。この例では出力したい Web ページを学習済みニューラルネットワークモデルに入力として入れると、出力として HTML、CSS のソースコードが生成される。

この試みは静的なページでのみしか行われておらず、JavaScript による動きの表現や、バックエンドロジックの存在する Web ページに対しては適用できない。しかし、単純作業になりがちな HTML、CSS の記述を一部であっても自動化できることができれば、Web プログラマーの負担は大きく軽減することができるだろう。

3.5 プログラム合成

ソフトウェア工学の一分野であるプログラム合成の分野では古くから、形式言語で仕様を定めることでソースコードが生成される、いわゆる自動プログラミングの研究が行われてきた [8]。これらの研究の成果はアプリケーションフレームワークなどの設計に活かされているが、設計作業とコーディング作業を完全に切り離すことはできなかった。

近年では、先述のような大量の Web コーパスデータから学習を行い自動でプログラムを合成する研究が増えつつある [9][10]。プログラム合成の多くの例では、与えられた仕様に対してその仕様を満足するソースコードが出力されるというものが多く、工学的観点からも合理的である。

4 プログラミングにまつわる自動化の組み合わせ

以上のようにプログラミングにまつわる話においても自動化の取り組みが様々になされてきたことがわかる。一方で、これらの自動化を、人間を支援する人間-機械系と機械が自動的に仕事をする系の 2 つに分けようとした場合には、ほとんどの例が前者に分類されることがわかる。これに関して考察を加える。

そもそも、プログラミングという行為は多くが工学のものである。計算機科学はプログラミングとは非常に関連が深いが、計算というものの性質を扱うという点では必ずしも工学的視点に立つ必要が無く実際に計算機を用いてプログラムを書くことは必須ではない。一方でプログラミングという行為自体は、多くの場合人間の役に立つことを目的として行われるため、非常に工学的な性質が強い行為であると考えられる。この視点から見ると、プログラミングにまつわる自動化の殆どが人間を支援する話になることも頷ける。すなわち、多くのプログラムはあくまで人のために書かれるものであり、その背後には人間の欲求や需要に根ざしたロジックが

存在する。そしてプログラミングとはそのような欲求・需要をロジックに落とし込み、計算機に実行可能な形にする行為である。そのほとんどが、形式的で適切な変換規則を設定すれば自動的に記述することができるものだったとしても、あらゆるプログラムには必ずヒト、あるいはヒトと全く同じ考え方をしヒトと同じものの捉え方をするコンピュータ・プログラム (存在すればの話であるが) が生み出さなければいけない要素が存在する。すなわち、プログラミングと自動化という観点で機械が自動的に仕事をする系を考えた時、その機械はヒトと同等か、それ以上の内面的複雑さを持ち合わせた存在である必要があり、現在までにそのような機械は生まれていない。

以上の考察を踏まえると、機械が自動的に仕事をする系は今の所存在しないため組み合わせの議論自体が成り立たないが、今後生まれるとも限らないため、もう少し掘り下げて考察を行う。仮に、完全に自動的にプログラミングを行う機械が誕生したとしてそれは一体どのようなものになるだろうか？ この議論を行うときによく引き合いに出されるのは技術的特異点に関する議論である。技術的特異点に関しては様々な論点が存在するが、その中で機械が自動でプログラミングをできるようになった場合に関する考察も存在する。このような話でよくあるのは、機械が自動でプログラミングをできるようになると、自分自身のプログラムを書き換えてより性能を向上させたり、制限のない動作ができるようになり、一気に人間の知性を置き去りにする、というセンセーショナルなアイデアである。この話について考えるとき、人間に関しても似たような話が存在することがわかる。すなわち、遺伝子操作の技術である。

遺伝子操作の技術はこれまで様々な不可能を可能にしてきた。本来は作物が育たないような土地でも育つ植物や、虫害に強い作物、収量の多い穀物、クローンの動物など多くの成功しているように見える例を生み出してきた一方でその生態系への影響の不明さなどから様々な批判を浴びているのも確かである。遺伝子操作の技術が自然界に及ぼす影響が評価できないことは、自然界にあまねく存在している一方で人類が数百年かけても解けていない問題と強い結びつきがある。この問題は複雑系に関連した難題であるということだが、これは恐らく自動的に自分自身のプログラムを書き換えることができるようになった機械に関しても当てはまる。現在のところ複雑系の現象に対して人類が取りうる立場は計算機の能力を借りてその挙動を模式的に計算してみることしかできていないが、その計算機自体が自身のプログラムの書き換えに伴う影響を計算しながらプログラムを改変できるようになった場合にどのようなことが起こるのかは想像するのが非常に難しい問題である。人類にとっていい方向に転ぶのか、悪い方向に転ぶのかといった二値の計算を行えない以上、私は「プログラムを自動で改変する機械」が生まれないようにするのが最善の手だと考えている。

一方、人を支援する人間-機械系に関しては、現状の取り組みの延長線上にある収束点は人類として歓迎するべきものではないだろうか？ 人を支援する系は、確実に収束点が存在することが保証されている。つまり、機械にできる領域というものが予め決まっていると考えられる。今まで行われてきた全ての取り組みは、不要な取り組みの繰り返しや数あるパターンの繰り返しを機械が行うように置き換える形で発展してきた。こうして、不要な取り組みの一切を取り払った時に残るものは、先述したとおり人間本来の欲求からくるロジックの部分である。そのロジックの記述がより柔軟に行えるようになり、ついには自然言語によるプログラミングができるようになったとしても、欲求や需要を自然言語に直す部分は人間の行う仕事である。したがって、私の考える限りの人を支援する人間-機械系の終着点は自然言語でプログラムが記述できるようになることである。もちろん、自然言語ではなくその手段は絵かもしれないし音声かもしれないが、いずれにせよその前の部分には必ず人にしかできない部分が残っていることになるしそのあり方がプログラミングの自動化ということを考

えた時あるべき姿ではないだろうか？

参考文献

[1] 佐藤周行. プログラミング言語処理系論. http://www-sato.cc.u-tokyo.ac.jp/SATO.Hiroyuki/-PLDI2012/1.pdf.

[2] Margaret Rouse. History of punch card. https://whatis.techtarget.com/reference/History-of-the-punch-card.

[3] Rails. https://rubyonrails.org/.

[4] mathworks. simulink. https://jp.mathworks.com/products/simulink.html.

[5] Unreal Engine. ブループリントのベストプラクティス. http://api.unrealengine.com/latest/JPN/Engine/Blueprints/BestPractices/index.html.

[6] Microsoft. アーキテクチャを分析及びモデルする. https://docs.microsoft.com/ja-jp/visualstudio/modeling/analyze-and-model-your-architecture.

[7] Emil Wallner. Floydhub blog - turning design mockups into code with deep learning. https://blog.floydhub.com/turning-design-mockups-into-code-with-deep-learning/.

[8] 玉井哲雄. ソフトウェア工学から見たプログラム合成変換技術. http://tamai-lab.ws.hosei.ac.jp/pub/pgm-trans.pdf.

[9] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, and Michael D Ernst. Program synthesis from natural language using recurrent neural networks. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01*, 2017.

[10] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.