グラフとアルゴリズムとプログラムのやさしいおはなし

渡邉敏正

2021 年 10 月 27 日

第 7 回

1　コンピュータとプログラム

第 6 回で、グラフにおいて基点 s から連結な点をすべて求めるアルゴリズム $\mathbf{compo}(G, s)$ を作成しました。次のステップは、プログラム作成を意識したアルゴリズムの詳細化および実際にプログラム作成に取り組むことです。しかし、そのステップに進む前にいくつかの準備をしておくことが、プログラミングに不慣れな方々がスムーズにこのステップに入って行くことにつながると判断しました。少なくとも準備として取り組むべきことは、

- コンピュータの大まかな仕組み、特にメモリと演算装置について基礎的知識を得ること
- アルゴリズム、プログラムおよびコンピュータの関係を理解すること
- ベクトルや行列をコンピュータのメモリに配列として格納すること
- これらに関連して繰り返し処理を C 言語の for 文や while 文で記述すること
- 条件によって操作を選択して実行する処理を C 言語の条件判断文 (if〜else 文) で記述すること

などです。当初はこれらを一括して説明することを考えていました。相当長くなるので分割を考えるべきでは、しかし分割するとわかりにくくなるのでは、と悩みましたが、分割して説明することに決めました。

アルゴリズムからプログラムを作成するステップ (ここでは仮に**アルゴリズムのプログラム化**とよんでおきましょう) はいくつかの分野にまたがる知識が要求されるのでわかりにくいかもしれません。しかし、アルゴリズムをプログラムとして実現するためには避けて通ることができない重要なステップですので、できるだけ例を使いながらわかりやすく説明していこうと思います。

その第一歩として、まずコンピュータの大まかな仕組み、およびプログラミングの基礎についての説明が必要と考え、今回は以下の内容に焦点を当てました。

- コンピュータの大まかな仕組み、特にメモリと演算装置の関係を簡潔に説明すること
- アルゴリズム、プログラムおよびコンピュータの関係をわかりやすく説明すること

内容として抽象的な説明が多くなりますので、できるだけ図と具体例を使いながらイメージを伝えることを意識しました。細かいことは飛ばしていいですので、大まかなところだけでもわかっていただくことを狙っています。気楽に読んでみてください。

なお、皆さんのわかりやすさを優先してイメージを伝えることに力点を置きましたので、厳密さは若干犠牲

にした箇所があることはお断りしておきます。コンピュータの仕組みやプログラムの処理過程は複雑で多岐にわたるため、厳密で詳細な記述はかえってわかり難くなる可能性があります。正確さを失わない範囲で簡略化してイメージを伝えることを選択しました。なお、説明はコンピュータから見た場合と、プログラムから見た場合があり、重複する箇所があります。

1.1　コンピューターの大まかな仕組み ー CPU とメモリを中心にー

プログラミングに必要と思われる事柄に限定して、コンピュータの仕組みや動作原理の大まかな説明をします。対象は**プログラム格納方式コンピュータ** (stored-program computer) です。現在皆さんが使うパソコンはほぼすべてこの方式です。図 53 を見て下さい。なお、CPU は**中央処理装置** (Central Processing Unit) の略称です。

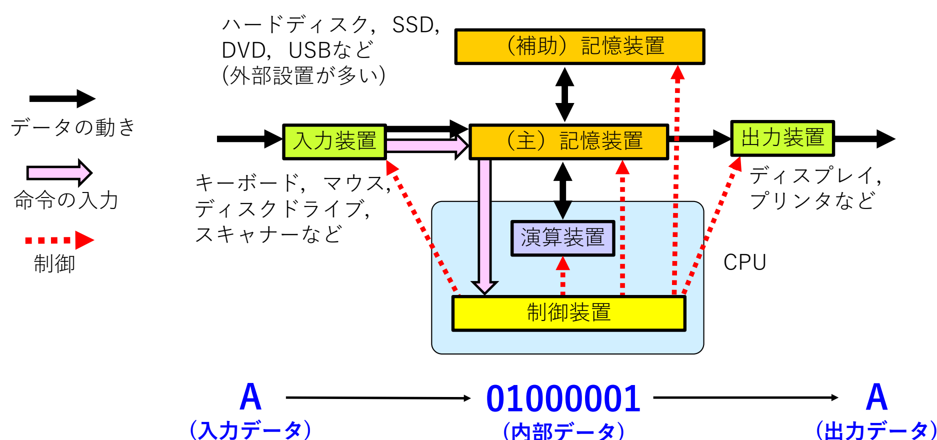

図 53　CPU(中央処理装置) と記憶装置 (メモリ) を中心としたコンピュータの大まかなイメージ

1. **コンピュータの全体像：**
   - コンピュータは大きく分けて入力装置、出力装置、記憶装置、演算装置、制御装置の 5 つの装置から構成されます。演算装置と制御装置を合わせて中央処理装置 (CPU) とよんで、4 つの装置からなる、と表現する場合もあります。
   - 情報は入力装置を通ってコンピュータ内部に入り、出力装置を通って外部に出ます。
   - 記憶装置は情報を保存する場所で、主記憶装置 (内部に存在) と補助記憶装置 (外部または内部に存在) がありますが、ここでは主記憶装置に着目します。なお、CPU の内部にも**レジスタ** (register) と呼ばれる小さな記憶装置がいくつかあります。以後、記憶機能を持った装置のことを「メモリ」と表現します (主記憶装置の意味で使うことが多いです)。
   - 演算装置はメモリにある情報をレジスタに取り出して、そこで様々な処理を行います。通常、その結果を再びメモリに保存します。
   - 制御装置は全体の状況を見ながら各装置にいろいろな指示を出して全体の処理を円滑に進行させます。(オーケストラの指揮者のイメージです。)
2. **コンピュータを動作させるには：**
   - コンピュータに対する動作の指示は外部から与える場合と、内部の装置間でやり取りする場合があ

ります。

- 外部からの指示については、キーボードから直接に指示を与える場合、あるいはプログラムとして記述してそれに従って動作させる場合などですが、何れにしても指示は入力装置を通して一度コンピュータ内部に蓄えられます。
- 蓄える場所は基本的には主記憶装置あるいはレジスタです。蓄えた指示を一つずつ解読し、その内容に従って動作します。

3. **プログラムとは：プログラム** (program) は、正確に言えばコンピュータプログラム (computer program) ですが、単にプログラムと言う場合が大半です。
   - プログラムはコンピュータに実行させる動作や処理を、具体的な操作の系列として詳細に記述した (コンピュータに対する) 指令書です。
   - 記述には専用の特殊な言語 (用語と書き方の規約を集めたもの) を使用します。それを**プログラミング言語** (programming language) といい、例えば C(言語)、Java、Python、Ruby、PHP など色々あります[*1]。
   - プログラムには色々なデータ処理も含まれ、必要なデータをプログラム内に記述する場合もあれば外部から読み込む場合もあります。また、生成したデータを内部に保管する場合もあれば外部に書き出すこともあります。
   - (機械語以外の) プログラミング言語で記述したプログラムを**ソースプログラム** (source program) と言います。
4. **オペレーティングシステム** (Operating System: OS)：
   - コンピュータの入出力機器やコンピュータシステム自体の管理を行うプログラムです[*2]。基本ソフトウェアという場合もあります。(**ソフトウェア** software はとりあえずプログラムやその集まりと考えておいてください。)
   - 上述の 5 つの装置それ自体は電子部品あるいはそれらから構成されている電子機器にすぎません[*3]。各装置がそれぞれの持つべき機能を発揮するのは OS がそのように管理運用しているからです。これらの多数の電子部品から構成されている精巧な電子機器 (**ハードウェア**：hardware) を OS が管理運用することで上で述べたようなコンピュータとして機能させている訳です。言ってみれば

     ハードウェア + オペレーティングシステム=コンピュータ

     ということです。
   - 代表的な OS は Microsoft の Windows、Apple の macOS(Mac の OS の名称)、UNIX や Linux などです。通常 OS はアセンブリ言語や機械語、あるいは特殊な言語で書かれることが多いのですが、UNIX や Linux は C 言語をベースに書かれています。
5. **プログラミング** (programming)：
   - これから皆さんに説明しようとしているプログラミングとは、コンピュータすなわちオペレーティングシステムの上で動作する (C 言語や Java などの) プログラミング言語の処理システムを利用し

[*1] これらは**高級言語** (high-level language) と呼ばれ、人間に分かりやすい文章形式の記述です。その他、**アセンブリ言語** (assembly language) や**機械語** (machine language) という処理効率はいいのですが人間には分かりにくい記述のプログラミング言語もあります。

[*2] 例えば、マウスを動かすとマウスカーソルが動く、キーボードで文字を打つと画面にその文字が表示される、キーボードから打ち込んだ命令を実行する、イヤホンから音を鳴らす、なども OS がやっている仕事です。

[*3] 現在では、集積回路基板の集まりです。

て、ソースプログラムを作ってコンピュータに意図する動作をさせようとすることです。

- このようなプログラムを**応用プログラム** (application progaram) と言います。「アプリ」という用語が製品のプログラムを意味する場合によく使われますが、ここでの応用プログラムあるいはプログラムという用語は製品よりは我々が自作するソースプログラムなどの意味で使います。

6. **オペレーティングシステムと応用プログラムの関係**：OS と我々が作る応用プログラムの関係をもう少し説明しておきます。図 54 を見てください。
   - OS は主記憶に置かれて、そこで各種装置に指示を出して情報交換をしながら全体を管理・運用します。青字の矢印がそれを示しています。
   - 一方、我々が作る応用プログラムはソースプログラムとして主記憶に格納されますが、それでコンピュータが動作するのではありません。
   - 後述しますが、ソースプログラムは実行可能プログラムというものに変換され、これも主記憶に格納されます。
   - その後 OS を介して、実行可能プログラムに記載の命令に沿った形で種々の装置が動作し、ソースプログラムで意図した操作が実行されます。

   図 55 に示しますように図 54 の「応用プログラム」はソースプログラムから実行可能プログラムへの変換過程、および実行可能プログラムの実行過程を合わせて象徴的に示したものです。赤字の実線矢印は実行可能プログラムへの変換およびその命令に従った動作指示のやり取りを表し、破線矢印は動作指示が OS を介して装置に届いて動作が起こるというイメージを表しています。変換過程の詳細、実行過程の詳細はそれぞれ図 56、図 57 を参照してください。

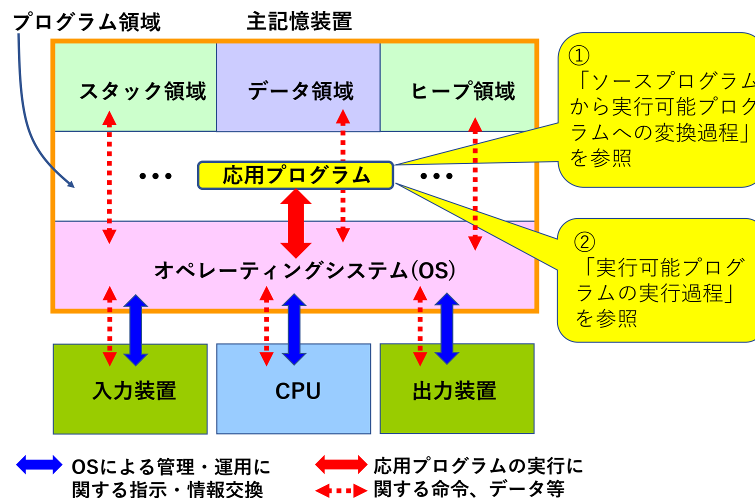

図 54　オペレーティングシステムと応用プログラムの関係 (イメージ図)：「応用プログラム」はソースプログラムから実行可能プログラムへの変換過程、および実行可能プログラムの実行過程を含めた表記です。これらの詳細は図 56、図 57 を参照してください。なお、領域名の説明は省略します

7. **プログラムの処理過程** (コンパイル言語を中心に)：
   (a) **コンパイル言語とインタプリタ言語**：
      - プログラミング言語は、実行に際してコンパイルという操作が必要な**コンパイル言語** (compile language) と、コンパイルなしでソースプログラムから直接的に実行できる**インタプリタ言語**

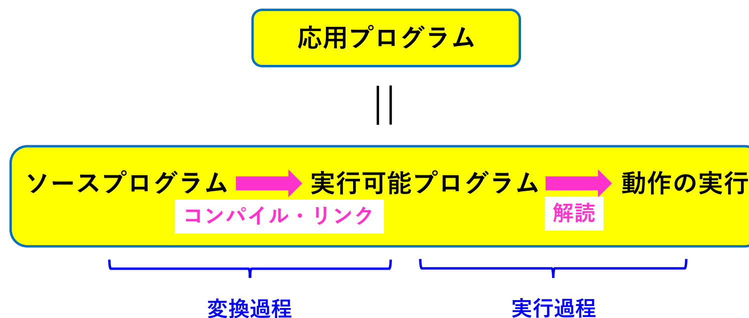

図 55　図 54 の応用プログラムが表しているプログラム処理の階層構造 (変換過程の詳細は図 56 を、実行過程の詳細は図 57 をそれぞれ参照して下さい)

(interpreter language) に大別されます。

- 前者は少し手間がかかりますが処理が高速という利点があり、後者は手間はかかりませんが一般的に処理が遅い傾向にあります。

以下では、特に断らない限り、コンパイル言語を扱います。今回扱う C 言語がコンパイル言語であることが一つのの理由です。なお、Java もコンパイル言語、一方で Python, Ruby, PHP はインタプリタ言語です。

(b) **コンパイルとリンク** (図 54「応用プログラム」のソースプログラムから実行可能プログラムへの変換過程についての詳細説明です)：図 56 を見てください。コンピュータが扱うことができる情報は 0 と 1 だけです。ソースプログラムやその中で使うデータあるいは途中で発生するデータなどはすべて主記憶に 0,1 の情報として格納されます。ソースプログラム自体は 0,1 の系列として格納されますが、それに基づいてコンピュータが動作する訳ではありません。

- **コンパイル** (compile) という操作によってソースプログラムは非常に細かい単純な動作を表す 0,1 系列 (**機械語命令** (machine language instruction) とよんでおきます) の集まりに分解されます。
- これを**再配置可能プログラム** (relocatable program) と言います。これは必ずしもコンピュータにとって実行に適した形とは限りません。また、例えば画面への表示やキーボードからの文字入力、あるいは三角関数といった頻繁に使うプログラム部品 (**ライブラリ** (library) といいます) は再配置可能プログラムとしてあらかじめ用意されており、プログラムの中ではこれらを引用する形で記述して利用します。
- **リンク** (link) という操作で、再配置可能プログラムの機械語命令群をコンピュータの実行に適した順序に並び替え、同時にライブラリなども含めて実行に必要な様々な情報と組み合わせて結合し、ファイルとして保存します。
- これを**実行可能プログラム** (executable program) とよびます。これも主記憶に格納されます。(この辺りのことは次の 1.2 でもプログラムの処理という視点から再度説明しますので、そちらも参照してください。)

なお、コンパイルあるいはリンクを実行するプログラムをそれぞれ**コンパイラ** (compiler) あるいは**リンカ** (linker) と言います。図にはそれらを記載しています。リンカは**リンケージエディタ** (linkage editor) ということもあります。 前処理系の説明は省略します。

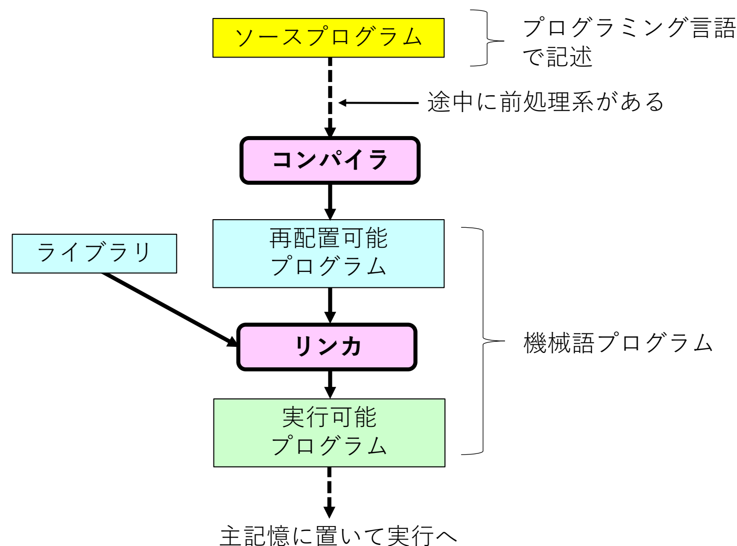

図 56　CPU 内部でのソースプログラムから実行可能プログラムへの変換過程 (コンパイル言語の場合)

(c) **実行可能プログラムの解読と実行** (図 54「応用プログラム」の実行過程、具体的には図 56 における実行可能プログラムの実行過程の詳細説明です)：図 57 を見てください。

- CPU の制御装置内部では、主記憶装置内にある実行可能プログラムから順次機械語命令を一つずつ読み出して**命令キュー** (instruction queue) というメモリに蓄える作業が行われます。これはこの後に述べるデコーダとは連動せず、独自で判断しながら取り出しを反復します。
- 一方、**デコーダ** (decoder) は命令キューから命令を 1 つずつ取り出してその命令内容や使用データの存在場所を解読し、その解読情報に基づいて操作の実行指示を出していきます。これでソースプログラムで意図した動作をコンピューターに実行させることになります。

(d) **データの読み込みと処理結果の扱い**：

- 処理に必要な指令やデータは制御装置の指示に従ってキーボード、外部記憶装置などの入力装置からスタック領域、ヒープ領域、データー領域などのメモリに読み込まれます。
- 一方、演算結果や外部への指令などは制御装置の指示に従ってディスプレイやプリンタなどの出力装置を通して外部に出ていきます。

8. **文字コードについて**：

(a) **コンピュータの中では**：コンピュータの外部と内部では、扱うデータの形式は大きく違います。図 53 の下の方を見てください。例えば、皆さんが半角文字のアルファベット A をキーボードから打ち込んだとしますと、入力装置は 01000001 という 0 と 1 の系列に変換します。(半角文字については注意 7.1 を参照してください。) 内部では A は 01000001 という系列で扱われます。一方、これを外部に表示する場合にこの 0 と 1 の系列をそのまま表示したのでは人間には意味不明ですので、ディスプレイやプリンタなどの出力装置が A と変換して表示します。

(b) **アスキーコード**：アルファベット、数字、記号など (半角文字) は **ASCII コード** (アスキーコード ASCII: American Standard Code for Information Interchange) とよばれる世界共通の 0 と 1

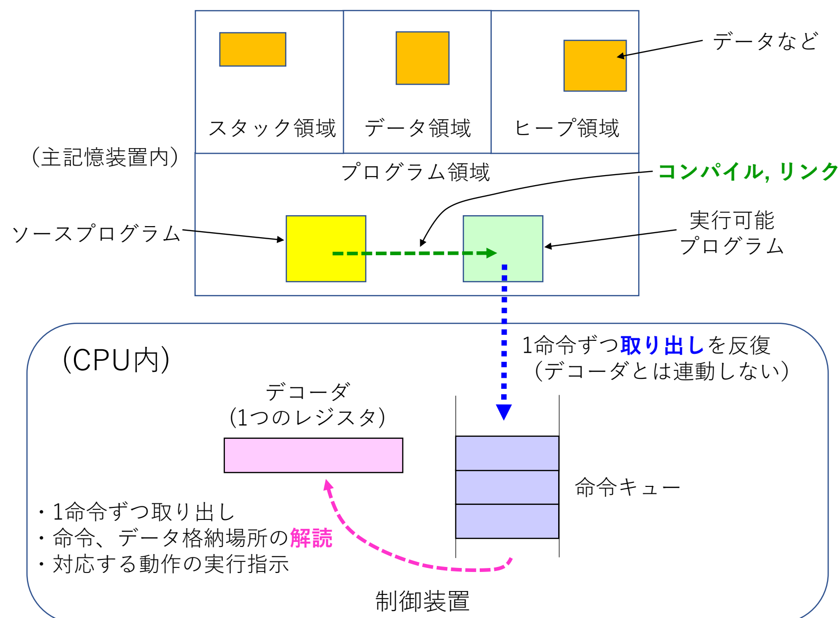

図 57　実行可能プログラムの実行処理：読み出しと解読のイメージ図

が 8 個並んだ系列が決まっています。例えば、

文字など (半角)	アスキーコード	文字など (半角)	アスキーコード
a	01100001	1	00110001
%	00100101	=	00111101

などです。

注意 7.1 縦と横の長さの比率が 1:1 の文字は**全角文字**、1: 0.5 の文字は**半角文字**と言います。例えば **愛** や **Ａ** はそれぞれ全角 1 文字、ai や AI はそれぞれ半角 2 文字です。**Ａ** と A の横幅の違いが見えるかと思います。0 または 1 それぞれ 1 つを**ビット** (bit) といい、8 ビットを 1 **バイト** (byte) とよびます。コンピュータのすべての情報は 1 バイトコード (0 または 1 が 8 個並んだ系列) を基本にして扱われます。半角は 1 バイトコードで、全角は 2 バイトコードでそれぞれ表されます。通常の漢字やひらがなは 2 バイトコードです。データ量の呼び方として、1000B (1000 byte)=1KB (1 kilobyte)、1000KB=1MB (1 megabyte)、1000MB=1GB (1 gigabyte)、1000GB=1TB (1 terabyte) などがあります。

注意 7.2 アスキーコードは元々は 7 ビットでしたが、先頭に 0 を追加して 1 バイトコードとし、先頭を 1 とするコードは各国の固有の文字に割り当てました。日本ではカタカナが割り当てられており、通常両者を合わせて **JIS コード**[*4] とよばれ、このカタカナは **1 バイト半角カナ**とよばれます。したがって、JIS コードはアスキーコードを含みます。

(c) **MLB は何処に**：いま、$\cdots$ MLB $\cdots$ という部分を JIS コードで見ると

[*4] 正式名称は JIS X 0201 です。

$$\cdots 0\,\overbrace{01001101}\,\overbrace{01001100}\,\overbrace{01000010}\cdots$$

となります。(左端の 0 は無視して) 左から順に 0,1 の 8 個ずつが M L B それぞれのアスキーコードです。コンピュータは左から 8 個ずつ読んでこれをディスプレイなどに ··· MLB ··· と表示します。ところが、すぐ左側が 0 であったとして、(どこかで 0 か 1 が抜けてしまったなどの) 何かの理由でこの系列をすぐ左の 0 から順に 8 個ずつ JIS コードとして読んで表示したとすると

$$\cdots\,\underbrace{00100110}\,\underbrace{10100110}\,\underbrace{00100001}\,0\cdots$$

を読んだことになり、··· &ｦ! ··· が表示されて文字化けが起こります。

(d) **愛が消える：**一方、漢字、ひらがな、カタカナといった日本語文字など (全角文字) もすべて内部で扱う 0,1 が 16 個並んだ系列が決まっています。ただ、いくつかの方式が存在しており、方式が異なれば対応する 0, 1 の系列が違います。このことも「文字化け」の一つの要因となります。例えば日本語 EUC(EUC-JP) では**愛**という漢字は内部データとしては

1011 0000 1010 0110

という系列です (分かりやすいように 4 ビットずつスペースを入れています)。これを A さんが B さんに送ったとします。B さんがこれを Shift JIS のコードとして読んだとすると表示は**ｰｦ**となり、文字化けが起こって愛は消えてしまいます。

(e) **見えないけれど：**

- コンピュータ内部でこのような文字などを表す 0 と 1 の系列を**文字コード** (character code) とよびます。また、文字自身と文字コードの対応の仕方を**文字符号化方式** (character encoding scheme) といいます。
- パソコン OS は MS-DOS, Windows 系、Mac いずれも以前は Shift-JIS ベースが支配的でしたが、現在は世界共通的な文字符号化方式であるユニコード系の UTF-8/UTF-16 に移行しています。以前は国内のパソコン OS は MS-DOS, Windows 系が多数を占めていましたので、パソコンは Windows 系でデータは Shift-JIS という印象があったのも事実です。
- **ウエッブサイト** (Web site) の表示のベースとなる HTML (Hyper Text Markup Language) ファイルは文字コードの影響を受けやすいのですが、上述の経緯からかなり前に Shift-JIS コードで作られたウエッブサイトの HTML ファイルが残っていることがよくあります。
- 最近のパソコンやアプリケーションソフトウェアは自動識別・自動変換などの文字化け表示防止機能を持つものが多く、ユニコード系 OS のパソコンで Shift-JIS データを読む場合でも文字化けは起こりにくくなっています。しかしながら、ユニコード系 OS のパソコン、タブレットなどでそのような機能がない場合に、Shift JIS コードの HTML ファイルを表示して文字化けが発生した、という現象はまだ見ることがあります。

ちょと一言 ウエッブサイトは一般にマークアップ言語 HTML で基本的な枠組みを記述します。これに、スタイルシート言語 CSS (Cascading Style Sheets) で様々なスタイル (写真のサイズや表示位置の指示、色の変化、動きの追加、など) を加えるというのが通常の姿です。

我々**マスターズ広島のウエッブサイト**を構成している HTML+CSS の記述データは最近まで文字コードがすべて Shift JIS で、ヘッダーとよばれる領域に使用文字コードの記述がありませんでした。

文字コードは表に出ませんし、通常は殆ど気にかけることはないのですが、パソコン OS が扱う文字コードとデータの文字コードが異なると文字化けが発生します。データの中に 1 ヶ所でも異なる文字コードが混在し

ていると文字化けが発生しその原因を突き止めるために手間がかかります。ただ、最近のパソコンやアプリケーションソフトウェアは自動識別・自動変換の文字化け表示防止機能を持つものが多く、このことで困ることはほとんどありません。ヘッダーに使用文字コードの記述があれば文字化けは発生しません。

ただ、使用文字コード (Shift JIS) の記述がない HTML ファイルで構成された当会のウェッブサイトをこのような機能を持たない機種 (例えば OS がユニコード系) で表示させたこともあったようで、文字化けが発生した、ということを聞いていました。

文字化けの一つの解決策は、各 HTML ファイルで使用している文字コードをヘッダーに記述することです。このことは広島大学の情報メディア教育研究センター (IMC) のヘルプデスク担当者からも示唆を受けていましたが、ウエッブサイトの HTML ファイル更新を実行するには少し勇気が必要です。ウエッブページは常に外部に向けて表示されていますので、書き換えにいくらかでも不具合があると文字化けで読めないウエッブページが現実に表示されてしまいます。当初 HTML ファイルの扱いに不慣れなため、実際何回か書き換えにトライしている間に文字化けしたページを表示してしまう失敗もしました。

2020 年夏から 2021 年初頭にかけて、IMC の機種更新に伴うウエッブサイトの移行がありました。それに関連して IMC が性能の良いウエッブサイト管理システムを導入しました。IMC のヘルプデスクに相談しながら当会のウエッブサイトの移行作業をやりましたが、その過程でこの管理システムに少しずつ慣れてきて、やっとヘッダーに文字コードを記述する作業を実行することができました。その結果、これまで文字化けが発生していたケースは解消されています。

広報担当幹事の下、ウエッブサイトの情報更新は意外と頻繁に行いますので、この文字コードの問題は重要です。現在は、HTML ファイルの文字コードを気にすることなく編集作業ができるエディタがあります。また、IMC にも性能の良いウエッブサイト管理システムがありますので、両者を適宜利用して本会のウエッブサイトを維持管理しています。なお、これらのエディタは必要であれば更新時に適宜文字コードを変換して作業を行い、最終的に所望の文字コードに再変換することができます。

1.2 アルゴリズム、プログラムおよびコンピュータ

アルゴリズム、プログラムおよびコンピュータ三者の関係を説明します。まずは大まかな関係を説明し、次に実例でもう一度これらの関係を説明することにします。これにより、プログラミングの意味やイメージが少しでも伝わることを期待しています。なお、「1.1 コンピューターの大まかな仕組み」と重複する説明があることは了解してください。

1.2.1 全体的な説明

何か実行したい処理をコンピュータに行わせるには、キーボードから直接に指示を出す場合もありますが、少し長い処理や複雑な処理、繰り返し実行する処理などはソースプログラムとして記述して、これを実行させることが一般的な姿です。言ってみれば「プログラムによってコンピュータを操る」という感覚でしょうか。以下の説明でその雰囲気が少しでも伝われば、と思っています。

コンパイル言語では、プログラムによる処理の実行は通常は

ソースプログラム作成 と **コンパイル + リンク + 実行 (エラー修正を含む)**

という 2 つのステップから構成されます。実際にはこれらの 2 ステップの反復がかなり長い間続くのが普通です。

1. **プログラミング言語とその処理系**：
   - ソースプログラムの記述には、プログラミング言語を一つ決める必要があります。プログラミング言語としては、例えばC(言語)、Java、Python、Ruby、PHP など色々あります。
   - プログラミング言語は、記述に使用する用語と文法 (用語の使用法、記述の規約など) などで構成されます。これに従ってソースプログラムを記述します。このソースプログラムの指示通りにコンピュータを操作させるためには、コンピュータ内部に当該の言語処理システムが必要です。
   - コンパイル言語の処理システムは大きく**コンパイラ** (compiler) と**リンカ** (linker) から構成されます[*5]。リンカは**リンケージエディタ** (linkage editor) ともいいます。

   通常使っている Mac の OS には C 言語処理システムが含まれていますので、今回は C 言語をプログラミング言語として採用しています。コンパイラは処理システムに含まれている gcc (リンカの機能も合わせ持つ) を使用します。
2. **操作対象の具体化**：
   - まず実行したい処理をアルゴリズムなどの具体的な形にします。
   - 通常はそれをプログラミング言語を用いてソースプログラムとして記述します。こちらのやりたい処理内容をコンピューターに伝えるには高級言語で記述したソースプログラムを使うことが一般的です[*6]。
   - なお、対象とする処理の種類や分野とプログラミング言語には相性のようなものがあります。ただ、プログラミング言語の選択がソースプログラム作成に大きな影響を与える場合もあれば、どのプログラミング言語でもソースプログラム作成にそれほど影響がない場合もあります。
   - 今回、準備段階の例題として実行しようとしている処理はどのプログラミング言語でも差し支えのない内容です。
3. **エディタによるソースプログラムの作成**：
   - アルゴリズムなどをプログラミング言語の用語、記述規則に従ってソースプログラムとして書いていきます。
   - ソースプログラムを記述する際には**エディタ** (editor) を使いますが、処理システムにエディタが含まれている場合とそうでない場合があります。
   - エディタは Windows や Mac のワードとか Windows のワードパッドなどと同様の文章作成に必要な機能を持ちますがプログラム作成用に単純化されており、代わりに自動の字下げ機能やコンパイルとの連動などプログラムの作成・実行に必要となる機能が強化されています。
   - 今回は Mac 上のエディタ X-code と Visual-Studio Code を使いました。
4. **コンパイル、リンクおよび実行可能ファイル**：図 58 を見てください。これはコンパイル言語についての図 56 の再掲です。
   - ソースプログラムを電子データとしてコンピュータに読み込ませるのですが、ソースプログラムそのものがコンピュータに伝わるのではありません。
   - ソースプログラムは人間には理解しやすい文字列ですが、コンピュータはそれは読めません。コンピュータが読めるデータは 1 と 0 です (実際には電圧のオンとオフです)。
   - そこで、**コンパイラ** (compiler) はソースプログラムをコンピュータが読める 0,1 のデータに (内容

[*5] ここではコンパイラ言語のみを扱います。「**コンパイル、リンクおよび実行可能ファイル**」の項も参照してください。
[*6] アセンブリ言語や機械語でプログラムを書く場合もありますが特殊なケースが大半ですので、通常は高級言語を使うと考えています。

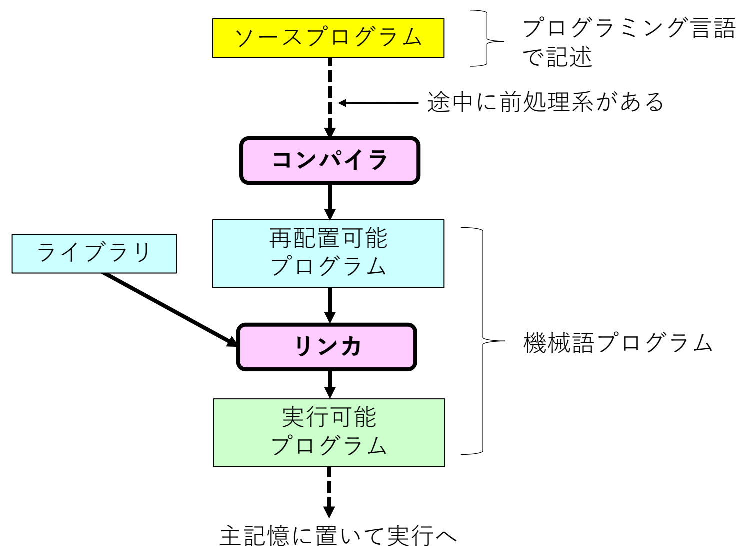

図 58　コンパイル言語のソースプログラムから実行可能プログラムへの変換過程 (図 56 の再掲)

を変えることなく) 変換する操作**コンパイル** (compile) を実行します。

- ソースプログラムは非常に細かい単純な命令 (0,1 系列で**機械語命令** (machine language instruction) とよぶことにします) の集まりに分割、変換されます。コンパイルの結果を**再配置可能プログラム** (relocatable program) といいます。
- **リンカ** (linker) は、これらの細かい命令をコンピュータの実行に適した順序に並び替え、かつライブラリなどの実行に必要な情報を付け加えて連結 (**リンク** (link)) して、コンピュータが実行可能な $0,1$ の長い系列、**実行可能プログラム** (executable program)、を作成します。
- 「この実行可能プログラムに従って操作を実行しなさい」という指示を処理システムを通して与えるとコンピュータがプログラムの指示に従って動作をする、ということです。(この辺りのソースプログラムから実行可能プログラムに至る過程はイメージが皆さんに伝わり難いと思います。具体例を使った補足説明を後述しますので、そちらも参照してください。)

5. **エラーチェック：**
   - ソースプログラムの文法上のエラーはコンパイラがチェックしてエラーがあれば画面に表示されます。まずこの文法のエラーチェックを通過しないと実行には進めません。
   - 一方、内容のエラーは人間のチェックが必要です。例えば、加算 + を減算 − と記述していても、文法上正しく記述されていればコンパイルは通過しますが、処理結果にエラーが出る可能性が高くなります。アルゴリズム自体の考え方が間違っていれば意図する結果が出てこないのは当然です。このようなエラーはコンパイラではチェックできません。
   - エラーのことをプログラミングでは**バグ** (bug) といい、このようなエラー修正作業のことを**デバッギング** (debugging) といいます。バグが「虫」で、デバッギングは「虫を取ること」という意味です。実は、プログラミングにおいてはこのデバッギングが**通過に忍耐力が必要となる大きな関所**なのです。

注意 7.3 プログラミング言語は分類の仕方がいくつかあります。一応、以下に簡単な説明はしておきますが、分かりにくいと思うときはスキップして差し支えありません。

- まずは、1.1 の 7 (a) で言及したコンパイル言語とインタプリタ言語です。次は、**手続き型言語** (procedural language) と**宣言型言語** (declarative language) です。手続き型言語は**命令型言語** (imperative language) と言う場合もあります。
- **手続き** (procedure) は入力データに対して処理をして結果を出力する仕組みの総称で、命令という場合もありますしいくつかの命令の集まりの意味で使うこともあります。手続き型はこれを基本の記述形式とする言語を指します。何かを処理する方法、つまりアルゴリズムを記述することが基本になります。
- 一方、**宣言** (declaration) は「何を得たいか」という対象 (出力) の定義です。宣言型はアルゴリズム (どうやって得るかの方法) ではなく、処理の結果として何を得たいか (結果の性質や状態) を宣言として記述する言語です。目指す結果を得るための方法（手続き）は特に示さずその言語処理系が持っているものを利用します。ターゲットを示し、それを求めるのことは処理系に任せるというやり方です。
- 別な分類方法として、**オブジェクト指向言語** (object-oriented language) か否かがあります。互いに関連するデータの集まりとそれぞれのデータに対する手続き (処理方法) の集合をまとめて**オブジェクト** (object) と呼びます。オブジェクト指向言語はオブジェクトをプログラムの基本的な構成単位として、オブジェクト間での指示やデータのやり取りに基づいてプログラムを組み立てる形式の言語です。これに当てはまらない言語を非オブジェクト指向言語あるいは在来型言語といいます。
- C 言語は在来型の手続き型言語であり、Java, Python, Ruby, PHP などはオブジェクト指向型の手続き型言語です。(C 言語と Java はコンパイル言語、Python, Ruby, PHP はインタープリタ言語です。)

ちょっと一言 確か工学部の 2 年次で「プログラミング序説」という講義での FORTRAN (フォートランと読みます) でした。私が最初に触れたプログラミング言語です。在来型の手続き型言語で今も使われており、現在あるプログラミング言語の殆どはこれから育っていきました。処理の基本となる操作をコーディング用紙に筆記し、それが文法にあっているか否かをチェックする作業が講義の中心でした。コンピュータ (専ら電子計算機と言っていました) の仕組みや動作、プログラムを作ることの意味もほとんど理解せず、それで単位が取れたことを喜んだ程度の関わりでした。

4 年次の卒業研究で、1000 時間当たりの誤り率が最小となるような、4 個の 2 値電子素子を使った 10 進カウンタの論理設計を求める、という課題にようやく辿り着き、やるべきことは $0, 1$ からなる 4 ビットコード 16 個の中からどのような 10 個を選んで 0〜9 に対応づければ全体の誤り率が最小となるか、を調べることに帰着できると考えました。(実は、これで得られる結果は $0, 1$ の変化の総数最小化で必ずしも誤り率最小化とは限らないこと、つまりターゲットが少しずれていたことも後日わかりました。) 簡単に見えるかもしれませんが、回路論理式の構成とその簡単化に加えて多くの数値計算があり、電卓と筆算ではとても無理と思われる作業量になります。否応無しに、自分でプログラムを組んでコンピュータで誤り率を求める必要性が迫ってきました。「プログラミング序説」のテキストをひっぱり出して読み直し、自らプログラムを書いて計算するという作業を通して、自分の無知を実感しながら先生が講義で話されたことが「あー、そういうことだったのか」とやっと (2 年遅れで) 理解できた、という経験をしました。今から考えれば、自分の意志で動くことの大切さを教えてもらった気がします。

しかし、初学者同然の者が書いた言わば一夜漬けのプログラムでは「コンパイルとデバッグ」というサイクルが延々と続くことは必然でした。まるで「計算して欲しければちゃんと書きなさい、はいもう一度！」とコ

ンピュータに突き返され続けているような苦闘の日々でした。

この苦労は一方で私に「プログラミングとは何か」を教えてくれました。曲り形にも計算法 (アルゴリズム) を考え、FORTRAN でプログラムを組み、デバッグを繰り返して妥当な結果にたどり着く、というステップを経験したことになります。この経験は私にとっては大きな財産となりました。これ以後、Basic, Pascal, C, Java など色々なプログラミング言語に出会いましたが、アルゴリズムはプログラミング言語の影響を殆ど受けませんので、一度作ったという経験がこれらの言語でアルゴリズムを作る場合の助けになりました。また、どの言語でも「えーと、繰り返しはどう書くのかな」など文法の違いをチェックしながら容易にプログラミングに入っていくことができました。一つの言語に慣れれば他の言語も大丈夫、ということを実感した次第です。

1.2.2 ソースプログラムと実行可能プログラムの例

ソースプログラムからコンパイルを経由して実行可能プログラムに至る過程は、プログラミングに馴染みのない方にはイメージが伝わり難いと思います。以下に、具体例と図を使って補足説明をします。

以下のシンプルな C 言語ソースプログラムを例とします。エディタ Visual Studio Code で書きました。これは整数型変数 a(整数だけ格納できる容器のイメージ) に 45 を、整数型変数 b に 27 を設定し、これらの加算結果 72 を整数型変数 c に格納します。ただし printf 文で結果を表示させていないので、実行しても結果は表示されません。なお、できるだけ短くするため 1 行に詰めたり色々と省略もあり、推薦される標準的な記述ではありませんが、正しく動作することは確認しています。左端の行番号は説明用に入れています。実際のプログラムに書くとエラーになります。

```
1  /* Sum of two integers: sum.c */
2  #include<stdio.h>
3  main()
4  {
5      int a=45, b=27, c;
6      c=a+b;
7  }
```

注意 7.4 説明の前に重要な注意をしておきます。`int a=45;` や `c=a+b;` などの C 言語プログラム中の `=` は等号ではありません。`=` の右側の式などの値を左側の変数に代入することを意味します。文章などでは、代入の意味を明示するためしばしば `a` ← `45;` や `c` ← `a+b;` と矢印で書くことがあります。

上記のプログラムを gcc でコンパイルすると、プログラム部分は 10 個の機械語命令に変換されます。0, 1 のデータとして見れば合計で 248 個の 0, 1 から構成された一つの系列です。この単純なプログラムでは再配置可能プログラムがそのまま実行可能プログラムになっています。以下に

(5 行目) `int a=45, b=27, c;`　(6 行目) `c=a+b;`

に対応する部分の機械語部分のみを示します。(実際には、前後にこれら以外の機械語命令があります。) 左側が得られる機械語で、これらはつながった系列ですがわかりやすいように命令ごとに分けました。右側は対応する**アセンブリ言語** (assembly language) の記述です[*7]。参考のために記載しています。0, 1 の系列では人間

[*7] 負の整数は実際には**補数** (complement) という特別な数値表現で扱いますが、ここでは通常の負の数の表示としています。

にとっては動作内容の把握が難しいので、少しだけ動作内容を表す記号で記述したのがアセンブリ言語です。**アセンブラ言語** (assembler language) ともいいます。

(機械語)	(対応するアセンブリ言語)
10000011 11101100 00001100	`subl $12, %esp`
11000111 01000101 11111100 00101101 00000000 00000000 00000000	`movl $45, -4(%ebp)`
11000111 01000101 11111000 00011011 00000000 00000000 00000000	`movl $27, -8(%ebp)`
10001011 01000101 11111100	`movl -4(%ebp), %eax`
00000011 01000101 11111000	`addl -8(%ebp), %eax`
10001001 01000101 11110100	`movl %eax, -12(%ebp)`

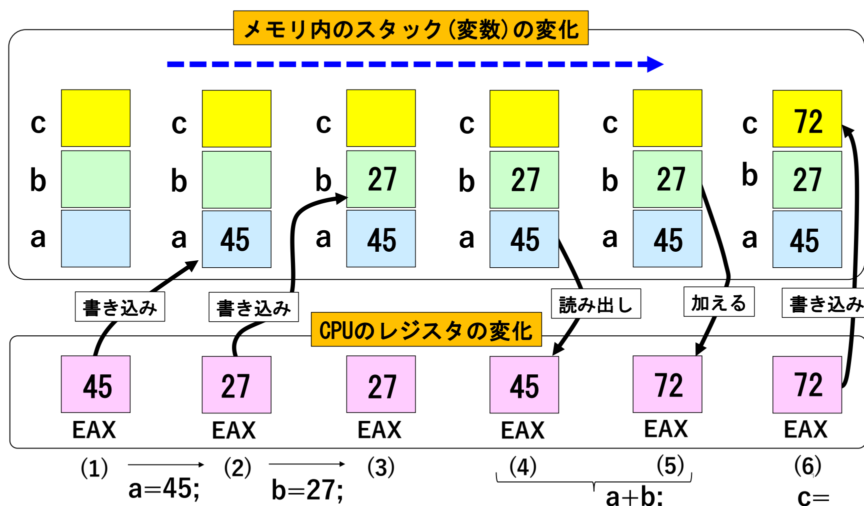

図 59　a+b の計算を実行するコンピュータの内部動作 (左から右へ時系列で表示)

アセンブリ言語の内容説明は省略し、図 59 でコンピュータ内部での対応する動作を示します。各々の命令は図からわかりますように単純な動作になっています。機械語と図の対応なども含めて説明します。

- 宣言 + 代入文 (混合形式) `int a=45, b=27, c;` の宣言部分 `int a, b, c` によって、図の (1) のように、まず主記憶内のスタックという領域に整数用の変数が 3 つ用意され、`a, b, c` と名前が付けられます。ぴったり対応という訳ではありませんが、役割としては 1 番目の機械語が対応します。
- さらに、代入部分 a=45 によって、図の (1) から (2) のように CPU 内部のレジスタ EAX に 45 が入り、それがスタック内の変数 a に格納されます。(スタックは一列に容器が並んだイメージで、データの読み書きに特別な機能を備えています。) 2 番目の機械語の内容です。
- 同様に、代入部分 b=27 によって、図の (2) から (3) のようにレジスタ EAX に 27 が入り、それが変数 b に格納されます。3 番目の機械語の内容です。
- 次に、代入文 c=a+b; についてです。図の (4) のように、まず変数 a の値 45 がレジスタ EAX に読み出されます。4 番目の機械語の内容です。
- 次に、図の (5) のように変数 b の値 27 がレジスタ EAX に加えられます。その結果、レジスタ EAX の値は 72 となります。5 番目の機械語の内容です。

- 図の (6) のように、レジスタ EAX の値 72 が変数 c に書き込まれて、加算の実行が完了します。6 番目の機械語の内容です。

この動作からコンピュータが細かな操作を積み重ねて一つの処理を実行している様子が見えると思います。

1.3 まとめ

今回は、第 6 回の "グラフにおいて基点 s から連結な点をすべて求めるアルゴリズム $\mathbf{compo}(G, s)$" のプログラム作成に向けた準備として、以下の 2 点に焦点を当てました。

- コンピュータの大まかな仕組み、特にメモリと演算装置の関係を簡潔に説明すること
- アルゴリズム、プログラムおよびコンピュータの関係をわかりやすく説明すること

おそらく多くの方にとって初めて聞く内容だと予想します。しかも抽象的な説明が多くなりますので分かりにくいと思います。図や実例を使って少しでも分かりやすくすることを意識して説明しましたが、どうでしょうか。

皆さんに「何やらコンピュータは難しそうだし、プログラミングもよくわからん」というネガティブな印象を与えてしまったかもしれません。しかしながら、例えば「ふーん、そんな感じか」というようなイメージがぼんやりとでもつかめた、ということで差し支えありません。文法に違反しないようにプログラムを書けば、内部での処理がどうなっているか、など気にしなくてもコンピュータは正確に迅速に処理をしてくれます。

では、なぜ今回のような概要を説明したのでしょうか。プログラムによってコンピュータを操る、とまでいかなくても、プログラムを書いて自分の意図する処理をコンピュータで実行するためには、相手すなわちコンピュータがどのようなものか、をある程度知っておくことが色々な場面で役に立ちます。コンピュータの仕組みは広範囲に亘りかつ複雑で、全体像を理解することは至難の業です。最初から広く深くということではなく、プログラミングの過程で出会う事柄について、不明な点や関連することを必要に応じて少しずつ理解しておく程度でもプログラミングには十分役立ちます。プログラミングに親しみながら少しずつ慣れていき、同時に必要な知識を徐々に取り込んでいけばいいと思います。このようなプログラミングとの付き合い方をしていけば、

- プログラミングを円滑に進めること
- 誤りがなくて処理が速く、メモリ使用効率のよいプログラムを書くこと
- 外からの影響を受けにくい頑健なプログラムを作成すること
- 読みやすく、以後のメンテナンスが容易になるプログラムを作ること

などにつながっていきます。これらはプログラミングに際して常に心に留めておくべきこと、言ってみれば

プログラミングの心

です。少し大げさかもしれませんが、これからプログラミングに関わっていただきたい若い人達に最小限の基礎知識を提供したい、という私の思いもあります。

次回は、プログラム作成に直接的に関係する以下の事柄を中心に、実例を使いながら説明する予定です。

- ベクトルや行列をコンピュータのメモリに配列として格納すること
- これらに関連して繰り返し処理を C 言語の for 文や while 文で記述すること

- 条件によって操作を選択して実行する処理を C 言語の条件判断文 (if〜else 文) で記述すること