第1章　はじめに

1.1　授業の進め方

毎回の授業は、基本的に以下の三部構成で行います。

- 前回の質問票への回答 (約 20〜30 分程度)
- 講義
- 今回の講義に関する質問票の記入 (約 5〜10 分)

講義ではこのテキストを配ります。テキストは可能な限り早めに配る予定です。なお、テキストの pdf ファイルは講義担当者のホームページ：

`http://www.fu.is.saga-u.ac.jp/~yaman/yamane.html`

に置いてあります[1] から、欠席した人はインターネットでダウンロードできます。ダウンロードの方法が分からない人や、ダウンロードしたファイルの印刷方法が分からない場合には山下に「○ページから○ページまでコピーをほしい」とメイルで連絡してください[2]。再履修者へ： テキストの内容は、一部の修正を除き、昨年度から大幅な変更はしない予定です。

1.2　成績

質問点、レポート点、期末試験点で決定します。

質問点 質問点は質問票を提出した回数で決めます。質問票には、毎回の講義の最後にその回の講義で分かりにくかったところや、関連する質問を記入ししてください。1 回当たり 1 点とし、質問点の上限を 10 点とします。授業は予定では全部で 15 回ありますから、質問点の合計値は 15 点になりますが、レポート点、試験の点とのバランスを考慮し、上限を 10 点に設定しています。ただし、講義のまったく関連のない質問、決意表明[3]、名前だけの白紙の質問票は 0 点です。

レポート点 講義に関するレポートを 1 回だけ書いてもらうことを予定しています。約 20 点程度の予定です。レポート課題は適切な時期に告知します。

[1] 学科のホームページのスタッフのページ (山下義行) を経由して辿り着けます。
[2] 口頭では忘れる場合があるのでメイルで連絡してください
[3] 質問票に「復習をがんばります!」などと書いている学生がよくいますが、0 点です。

期末試験点 テキスト、ノート等の持ち込みは不可です 。約 70 点の配分を予定しています。

質問点、レポート点、期末試験点の合計が 60 点以上で、この科目の合格です。なお、この科目はコンピュータの基礎を学ぶことを目的としており、その内容は基本情報技術者試験 (旧・情報処理二種）の内容と類似しています。そこで、この試験の合格者に限り、質問票の提出を免除し、毎回、質問点 1 点を与えます。該当者は、その旨を証明できる書類等を持って申し出てください。ただしレポート、期末試験は免除しません。

1.3 講義の概要

この科目は、コンピュータについて全く知識のない学生を対象にしています。

初心者が抱くであろう素朴な疑問: コンピュータはどのような構造をしており、どのような仕組みで動作するのか、プログラミングとはどのようなものか、過去/現在の代表的なコンピュータシステムにはどのようなものがあるか等の素朴な疑問に答えることを目標しています。そのために、初心者に理解しやすい内容を段階を追って説明していきます。情報表現の基本である **2 進数/16 進数とその補数表現**については、特にがっちりと学習する予定です。

授業中に紙面で配布する講義テキストでは、テキスト中に枠囲みの空白 [　　] があります。この空白に入る語句は、授業中に説明します。山下@講義担当者のホームページの pdf ファイルにはその語句が [**語句**] のように埋めてありますから、授業中に聞き逃した場合にはダウンロードしてチェックしてください。

第2章　様々なコンピュータ・システム

講義の始まりとして、この章ではコンピュータの種類と用途について概説し、コンピュータとは何かという疑問に実例に通して答えることを目的としています。

コンピュータのことを初めて学ぶ人には内容や言葉が難しいと感じられるかもしれませんが、気にする必要はありません。慣れるに従い、自然と理解できるようになっていきます。

2.1　システムの規模による分類

コンピュータを規模（占有面積、体積、価格）で分類することは最も簡単な分類法です。大きい方から見ていきましょう。

スーパーコンピュータ：高い演算能力を持つコンピュータの総称です。最近では100PFLOPS[1]に届く能力のものも開発されています。実行方式は歴史と共に変遷しており、現在は超並列コンピュータ[2]が主流です。TOP500[3]と呼ばれるスーパーコンピューターのランキングが有名です。図2.1は日本最速のスーパーコンピュータ：京の筐体群を側面から見た写真です。800個を超える筐体が計算機室のフロアに敷き詰められています。

国内外の離れた場所に点在するスーパーコンピュータを高速通信網でつないで仮想的に1台の大きなコンピュータとみなす方式のコンピュータも考えられており、これを**グリッドコンピュータ**と呼ばれています。また、インターネットでつながれた多数のパソコン（パーソナルコンピュータ、PCと略します）を1台のコンピュータとみなす場合もグリッドコンピュータと呼ばれます。後者の場合、公益性の高い課題の解決に全世界からボランティアを募り、家庭用PCが稼働していない夜間にそのPCの計算パワーを提供してもらうような仕組みで運用されています。インターネット上に仮想的な巨大コンピュータが存在するイメージです。

グリッドコンピュータに似た形態のコンピュータとして、最近ではクラウドコンピュータがしばしば話題になります。グリッドコンピュータは高速計算を主目的としたシステムですが、

[1] FLOPSはFLoating point operations Per Secondの略であり、加減乗算を1秒間に何回実行できるかを表す単位です。P（ペタ）は10の15乗を表します（2.3節参照）。1 PFLOPSは加減乗算を1秒間に10の15乗回（=1000兆回）実行できることになります。なお、除算は加減乗算に比べ複雑な処理のため、FLOPSの対象からは外されています。

[2] 複数台のコンピュータを同時に実行させる型のコンピュータを並列コンピュータと呼びます。超並列コンピュータでは数千台、数万台のコンピュータを同時に実行させます。

[3] https://www.top500.org

図 2.1: スーパーコンピュータの例：京（日本、理化学研究所）（Wikipedia“京 (スーパーコンピュータ)” から転載）

図 2.2: 大型コンピュータの例：IBM System/360（Wikipedia“System/360” から転載）

クラウドコンピュータは多様な処理をインターネットを介して不特定多数の PC で高速に処理することを主目的としています。次節でも触れます。

大型コンピュータ：汎用大型コンピュータ、メインフレームとも呼ばれます。コンピュータが高価であった時代には今のパソコン以下の能力しか持たないコンピュータがビル 1 棟を占有するほどの大きさでした。図 2.2 は、歴史的名機として名高い IBM System/360 の写真です。

ワークステーション、

デスクトップ型ハイエンドパソコン：

ワークステーション[4]は、大型コンピュータほどではないものの高い演算能力、美しいディスプレイ画面、マウス等の入力装置を備え、GUI が利用可能な個人使用コンピュータの総称です。ハイエンドパソコンはワークステーションよりも性能、機能が劣るものという立て分けでしたが、現在では両者の区別がほとんどなく、ワークステーションという言葉は使われなくなっています。

ノートパソコン：ノートほどには薄くも軽くもないので、ブック型パソコンとも呼ばれますが、今や高価な文房具のひとつとして仕事になくてはならないものです。低価格化、高性能化、加えて可搬性[5]という特徴からデスクトップ型はノート型にシェアを奪われています。近い将来は（あるいは、もうすでに）、一部の高信頼かつ高性能なハイエンドパソコンを除いて、パソコンは全てノートパソコンで占められるでしょう。

家庭用ゲーム：様々なプログラム（ゲーム）を実行できると[6]いう意味でゲーム機はコンピュータです。

携帯電話：単なる電話ではなく、様々なプログラムを実行できるという意味でやはりコンピュータです。むしろ携帯電話を電話として使用している頻度は，今や極めて低いのではないでしょうか。スマートフォンは小さなパソコンと言ってもよいでしょう。

組み込み型コンピュータ：いまやほとんどの電気製品（洗濯機、電子レンジなど）の中には極小規模のコンピュータが組み込まれています。今はまだ限定的ですが、今後これらのコンピュータがインターネットと接続されるようになれば、生活に劇的な変化をもたらすかもしれません。

2.2 システムの用途による分類

今やコンピュータの利用は社会の隅々にまで浸透しています。その利用の内容を列挙してみましょう。

大規模数値計算：コンピュータはまさに高速計算機械としての要求から開発されました。コンピュータは単に高速計算を行う道具ではありませんが、計算が主要な用途であることには変わりはありません。コンピュータによる演算（$1+2=3$、$0.3\times0.4=0.12$ などの計算）を特に数値計算（numerical calculation）と呼びます。数値計算は今やほとんど全ての科学/工学に必須の技法です。素粒子質量の計算、銀河系の生成シミュレーション、

[4]直訳すると「仕事の場所」ということですが、まさにコンピュータを使った仕事の場所を意図した名称です。しばしば WS と略されます。

[5]蛇足ですが、ノートパソコンは衝撃に弱いので、自転車のかごに直接入れて運搬などの扱いをすると確実に壊れます。

[6]プログラムによって、様々な処理が可能なことを、万能性（universality）と呼びます。コンピュータの重要な性質のひとつです。歴史的には von Neuman のプログラム内蔵方式がそれを可能にしました。

建物の耐震解析、気象予測、核兵器の設計、自動車の車体の設計、タンパク質の性質の解析など無数の応用が考えられます。また計算物理、計算化学などの、数値計算に特価した科学分野も確立されています。

事務処理：会社における給与計算などが典型です。かつては大型計算機を用いていましたが、今や同等の仕事はパソコンで十分にこなせます。

銀行オンラインシステム：コンピュータの最も大きな用途は、数値計算よりも銀行のオンラインシステムかもしれません。第一に顧客データ（預金高、氏名、住所など）を正しく管理する仕組みが必要です。また銀行端末からの預け入れ、引き出し、振り替えを正確に遂行する仕組みも必要です。さらには各銀行システム間でのお金の移動もあります。これらを 1 回のミスもなく何年も実行し続ける完成度の高いプログラムを作ることが要求されます。商用コンピュータの販売と同時に始まった銀行オンラインシステムの開発は既に 30 年以上に渡って継続されており、システム全体はもはや個人の管理能力をはるかに越えるほど巨大化していると言われています。

また、ソフトウェア会社の主要な顧客は銀行を始めとする金融関係です。もしあなたがソフトウェア会社にプログラマとして入社したならば、かなりの頻度で金融業務に関連するプログラム開発を担当することになるでしょう。

情報検索、**データベース**：コンピュータは単に計算をするための道具ではありません。コンピュータを用いて、膨大なデータを管理することも可能です。そのような目的のためのシステムはデータベース（database、データの蓄積基地）と呼ばれています。データベースでは単に大量のデータを蓄積するだけではなく、その中から必要なデータを適切かつ高速に検索できる仕組みが必要です。

たとえば図書館の蔵書データベースは今や図書館利用には欠かせないものです。かつてそれらは紙製の蔵書カードで管理されていました。しかし今はコンピュータ端末から瞬時に蔵書を検索可能です。最近では日本全国の大学図書館の蔵書を同時に検索することも可能です。

多くの会社は顧客データベースを持っています。最近ではビッグデータと呼ばれる様々な統計情報がビジネスに利用されています。それらを有効に活用することは今や非常に重要です。

最近、特に忘れてならないことに、インターネット上での情報検索があります。利用者が必要とする情報を簡単なキーワードから的確に検索するために、優秀な検索手法や検索ソフトウェアの重要性が増しています。逆にユーザの立場からは、必要な情報を的確に検索する技術を身につける必要があります。

ビデオゲーム：家庭用 TV を映像の出力先とするビデオゲーム（TV ゲーム）は、今やコンピュータ利用の重要な分野のひとつになっています。最近ではインターネッ

トを利用したネットワークゲームも流行しています。

デジタル家電：組み込み型コンピュータの話は 2.1 節で取り上げました。それらを組み込んだ家電品を特に「デジタル家電」と呼ぶようです。ひと昔前の制御用マイコンと異なり、最近の組み込み型コンピュータは通信機能を持っていることが特徴です。

クラウドコンピューティング：これは、ネットワークを介したコンピュータの新しい利用方法です。コンピュータの形態や規模を表す言葉ではなく、むしろ利用形態のひとつです。種類、数、規模が雲（クラウド）のように不確定で膨大なコンピュータの集合体にネットワークを介して可能な様々なサービスを総称してクラウドコンピューティングと呼んでいます（amazon でできることを思い起こしましょう）。コンピュータ・ネットワーク、インターネットが進歩したことによるコンピュータの新しい利用方法です。現在も急速な拡大を続けています。技術的にはグリッドコンピュータを用いる場合がありますが、コンピュータの形態自体は雲の向こうに隠れて見えない、見えなくてもよいという位置付けです。

ネットワーク管理：たとえば日本からアメリカへコンピュータネットワークを介してデータを送信した場合、そのデータは日本とアメリカをつなぐネットワーク上の複数のコンピュータをバケツ・リレーしてアメリカへ届くことになります。この際のバケツリレーを実行することもコンピュータの大きな役割になります。また日本からアメリカへの経路は複数存在しますから、どの経路を辿るか、それを決定することもコンピュータの仕事です。ネットワーク上をウイルスなどの悪意を持ったソフトウェアが流れないように防衛する仕組みも重要です。コンピュータ・ネットワークはそれ自身で（数値計算のような）特定の仕事を行う訳ではありませんが、ちょうど高速道路が我々の社会生活に必要であるのと同じように、社会基盤として重要です。今やコンピュータ・ネットワークが止まれば、社会活動のほとんど全てが機能不全に陥ります。

2.3 休憩: 数量を表す修飾子

コンピュータでは様々な数量単位 — bit（ビット）, byte（バイト）, Hz（ヘルツ）, bps, FLOPS（フロップス）など — が使用されますが、大きな数を用いるときには特に K（キロ）、M（メガ）、G（ギガ）などの修飾子を付けて数字の数を減らし、見やすくすることがあります。これらは国際単位系における接頭語として認定されているものです。いずれもギリシャ語を語源としています。

K（キロ）：$10^3 = 1000$。ただしメモリ容量を表すときには、1000 ではなく $2^{10} = 1024$ を K としますから混乱がないように注意が必要です。国際単位系においては、

1024 は K ではなく Ki（キビ）と表すことが 2006 年に決定していますが、未だ Ki は十分に普及しておらず、混乱が見られます。

M（メガ）：$10^6 = 1000000$。ただし K と同様にメモリ容量などで使用されるときには、10^6 ではなく、$2^{20} = 1048576$ を M とします。同上、2^{20} には Mi（メビ）が正しい修飾語ですが、普及していません。

G（ギガ）：10^9。メモリ容量については同様に、Gi（ギビ）が正しい修飾語です。

T（テラ）：10^{12}。メモリ容量については同様に、Ti（テビ）が正しい修飾語です。

P（ペタ）：10^{15}。メモリ容量については同様に、Pi（ペビ）が正しい修飾語です。

E（エクサ）：10^{18}。現時点ではまだこの修飾子が使用されることはほとんどありませんが、10 年後、20 年後にはかなり頻繁に用いられるようになるかもしれません。メモリ容量については同様に、Ei（エクスビ）が正しい修飾語です。

コンピュータの性能を知る上で最も基本的な数値はコンピュータの動作周波数（Hz）、演算能力（FLOPS）、メモリ容量（byte）です。現在、最高水準のコンピュータでは、動作周波数は G Hz、演算能力は P FLOPS、メモリ容量は P byte のオーダーに達しています。

第3章　情報科学/工学の歴史

ここではコンピュータの歴史について学びます。いくつかの情報科学年表 から重要（と講義担当者が判断した）項目を抜き出し、各項目について簡単なコメントを付けました。

初めて見聞きする言葉の羅列にとまどうかもしれませんが、暗記する必要はありません。言葉の意味を完全に理解する必要もありません。コンピュータの歴史の大きな流れを把握できれば、それで十分です。

3.1　ENIAC 以前

コンピュータが、現在の我々の知っている形で登場するのは第二次世界大戦前後です。3.2 節で紹介する ENIAC はそのような最初のコンピュータですが、しかし、それ以前に幾つもの萌芽的な事項が歴史に登場します。ここではまずそれらを概観します。

なお、☆マークがある項目 は、この学科で後にみなさんが受ける授業と密接に関連しています。

紀元前 3 世紀

1. アルキメデスが円周率を 3 桁まで計算。以後、円周率の計算は各時代のコンピュータ（や人間）の演算能力を表すシンボルになります。

12 世紀

2. **アルゴリズム** の語源となる「Liger Algorismi（アルフワリズム[1] の書）」がペルシャ語からラテン語へ翻訳される。アルゴリズム（algorithm）は情報科学/工学にとってきわめて重要な概念です（7.2 節で学びます）。

17 世紀

3. 円周率が 32 桁まで計算される。

[1] アルフワリズムは 9 世紀初めのアッバス朝の数学者とされています。

図 3.1: タイガー計算機（Wikipedia“機械的計算機 ”から転載）

19 世紀

4. イギリス人バベッジ（Charles Babbage）が、エイダ（Augusta Ada Byron）夫人[2] の支援のもと、解析機械（analytical engine）を構想、試作する。イギリスはちょうど産業革命の頃（日本は江戸時代）であり、蒸気機関を動力源にする機械式コンピュータが構想されました。この解析機械は歴史的に忘れ去れていましたが、最近になって歴史上最初のコンピュータとしての評価が固まってきています。エイダ夫人は、歴史上最初のプログラマ[3] と呼ばれています。

5. 数学者ド・モルガン（de Morgen）が著書「Formal Logic（形式論理）」を発表（1847）。論理学は、数学のみならず、コンピュータの理論としても重要です。

1910 年代

6. さまざまな機械式計算機[4] が発明される。日本でも大本寅二郎が手回し式計算機「タイガー計算機」（図 3.1）を国産開発、発売（1924）し、好評を博します。

[2]詩人バイロンの娘でもあります。
[3]このことに敬意を表して、アメリカ国防省によって設計主導されたプログラミング言語は Ada と名付けられました。
[4]正確には、計算機（コンピュータ）ではなく、四則演算を高速に行う卓上計算機に近いものです。「高速」とは言え、手回し式なので 1 回の演算に数秒掛るという速度です。当時の理系大学生は、教授の研究補助で、朝から晩までタイガー計算機の取っ手を回していたそうです。

1920 年代

7. タイプライタなどの事務機器メーカだった CTR 社が IBM（International Business Machine）社と改名[5]。後に世界最大のコンピュータ製造会社となり、パソコンが現れる 1980 年代まで世界の頂点に君臨します。

1930 年代

8. ドイツ系ユダヤ人ゲーデル（Kurt Gödel）が不完全性定理を証明（1931）。ある種の数学的体系では真とも偽とも証明できない命題が存在することを示しました。数学のみならず思想/哲学にも衝撃を与えました。コンピュータ処理の限界を示したことにもなっています。

9. イギリス人チューリング（Alan Turing）が **チューリング機械** を発表（1937）。これは最も有名なコンピュータの数学理論です。

10. ☆　アルゴリズムの数学的基礎が固まる。語源は上に述べた通りですが、現代的な意味でのアルゴリズム（とプログラム）の定義がこの時代に確立されます。

3.2　ENIAC 以後

電子計算機（現代的意味でのコンピュータ）の発明は情報科学/工学の発展を加速させていきます。ENIAC 以降、本格的なコンピュータ時代がやってきます。

1940 年代

11. 真空管式コンピュータ ENIAC の開発がアメリカで始まる（1943）。1946 年に完成。17468 本の真空管で動作し、消費電力 150kW、幅 24m× 奥行き 0.9m× 高さ 2.5m、重量 30 トンという規模でした。同様の開発プロジェクトとして、同時期に EDVAC、EDSAC、Mark I などのコンピュータが相次いで開発されます[6]。図 3.2 に当時の ENIAC の写真を載せます。

12. ☆　ドイツ系ユダヤ人フォン・ノイマン（von Neumann）が **プログラム内蔵方式**（stored program method）を提案した[7]（1945）。この方式を採用した世界初のコンピュータはイギリスの EDSAC（1949）とされています。以後、現在までほとんど全てのコンピュータはこの方式に基づいています。これをノイマン型コンピュータとも呼びます。

[5]映画「2001 年宇宙の旅」に出てくるコンピュータの名前 HAL は IBM の 3 文字のアルファベットを 1 文字ずらして（I→H, B→A, M→L）作った名前として有名です。

[6]歴史上最初の電子計算機を ENIAC とする見方は広く受け入れられていますが、特定用途に限れば、1939 年に試作された ABC（Atanasoff-Berry Computer）とする考えもあります。

[7]実際の発案者はノイマンではなく、その辺、少しゴタゴタした事情があったらしい。興味のある人はインターネットで検索してみてください。いろいろ出てきます。

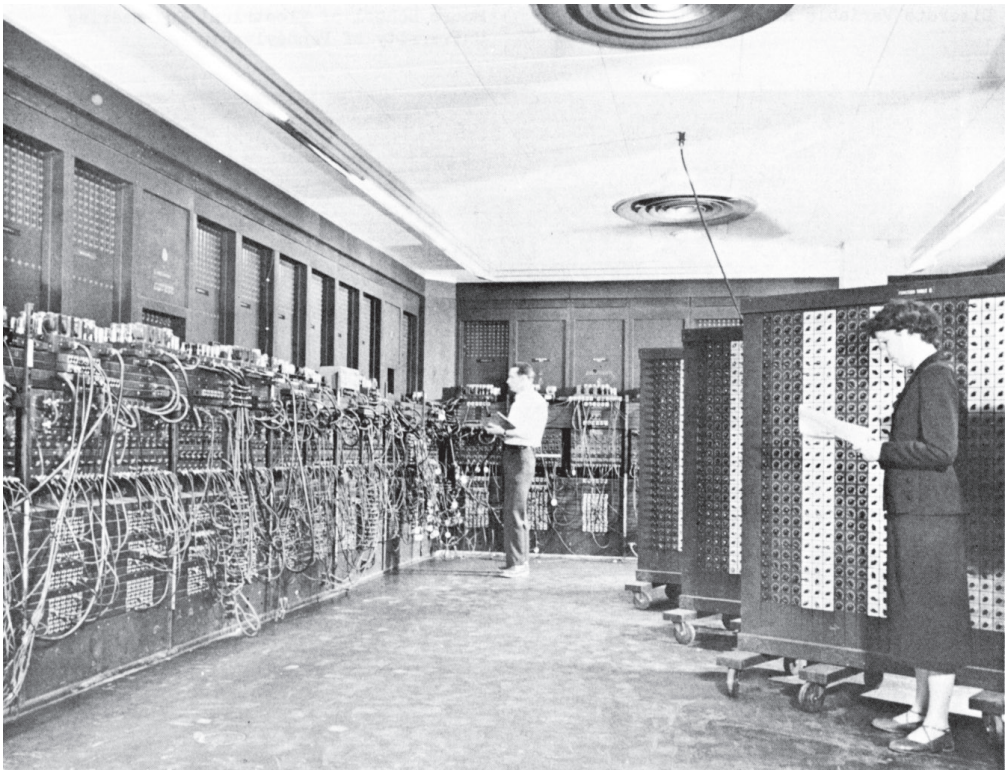


図 3.2: ENIAC コンピュータ（”U.S. Army Photo”, Wikipedia“ENIAC ”から転載）

13. ☆　オートマトン（自動機械）の理論の研究が始まる。これは情報科学の基礎理論のひとつです。

1950 年代

14. プログラミング言語 FORTRAN の開発が始まる（1954）。歴史上最初の**高水準プログラミング言語**（high-level programming language）です。

15. 人工知能という言葉が発明される（1955）。

16. 日本電信電話公社がパラメトロン計算機 武蔵野 I を開発する（1957）。これは世界史的な意義は薄いのですが、当時の日本で独創的なコンピュータが開発された事例として有名です。

17. ☆　MIT のマッカーシー（MacCarthy）が関数型プログラミング言語 Lisp を開発（1958）。この言語は、人工知能などの記述言語として現在でも使用されています。

18. チェス・ゲームのプログラムが作られ始める。

図 3.3: PDP-11（Wikipedia“PDP-11 ”から転載）

1960 年代

19. ☆　時分割処理システムが MIT で開発される（1961）。これによって、ひとつのコンピュータで同時に複数の仕事を行うことが可能になります。

20. ☆　アメリカで ASCIIコード（ASCII code）が制定される（1963）。英文字、数字、基本記号のコード体系として今でも利用されています。この講義でも解説します。

21. IBM 社システム/360 シリーズを発表（1964）。汎用大型コンピュータの歴史的名機として有名です。その後、UNIX が広まるまでのほとんど全ての汎用大型コンピュータのひな形となりました [8]。

22. チューリング賞が創設される（1966）。残念ながら、未だ日本人の受賞者はいません（数名の候補者は挙がりましたが）。

23. アメリカ国防総省の推進で大学間ネットワーク ARPANET が構築される（1967）。これがインターネットの第一歩になりました。

1970 年代

24. DEC 社がミニコンピュータ PDP-11 を発表（1970）。小型コンピュータ（といっても一部屋分くらいの大きさ）の名機として有名です（図 3.3 参照）。

[8] 日本のコンピュータメーカーは競って IBM 互換機を開発しましたが、それが IBM 産業スパイ事件（1982）につながっていきます。

25. Intel 社が 4bit マイクロプロセッサチップ 4004 を開発（1971）。後のパソコン時代はここからスタートすると言っても過言ではないでしょう。この開発に日本人[9]が関わっていたことは有名です。

26. ☆　プログラミング言語 C が開発される（1972）。その後、UNIX の記述言語として世の中に浸透していきます。

27. ☆　通信規約（プロトコル）TCP/IP が設計される（1974）。現在のインターネットで使用されているプロトコルです。

28. アメリカ人ビル・ゲイツ（Bill Gates）がパソコン用言語 BASIC を開発し、マイクロソフト社を設立（1975）。ゲイツはこのマイクロソフト社の経営で世界一の大富豪になりますが、その秘訣は経営手法[10]がそれまでのソフトウェア開発会社と異なっていたためです。

29. ☆　ローカルエリアネットワーク Ethernet が設計される（1975）。現在のインターネット技術（特にハードウェア面で）の出発点と考えてよいでしょう。

30. クレイ（Cray）社がスーパーコンピュータ Cray1 を発表（1975）。ベクトル・コンピュータと呼ばれる第一世代のスーパーコンピュータです。

31. 未解決問題だった地図の四色塗り分け問題がコンピュータを用いて肯定的に証明される（1976）。コンピュータを数学の重要な証明に用いた最初の例として有名です。

32. ☆　バークレー版 **UNIX** が開発される（1976）。UNIX OS が世界に広がるきっかけとなりました。

33. 東芝が日本語ワードプロセッサを開発（1978）。コンピュータの入力装置はいわゆるタイプライタ（現在のキーボード）を基盤とするのですが、その意味で日本語の扱いには障壁となっていました。それを突破する技術として日本のコンピュータ史上特筆すべき発明です。

34. インテル社が 16bit プロセッサ 8086 を発表（1978）。現在ほとんど全てのパソコンに組み込まれているプロセッサのひな形です。

35. X 線 CT 装置の開発に対してノーベル賞が贈られる（1979）。これは、コンピュータを用いた機器開発に対する最初のノーベル賞です。

36. ☆ **構造化プログラミング**（structured programming）が提唱される。これは現在最も普及しているプログラミング技術の基本です。

[9]嶋正利氏（ビジコン社、当時）。
[10]製品を販売会社などに売る際に、開発した製品の全ての権利を売却するのではなく、使用権のみを売却するという手法。

37. ビデオゲームが開発、販売され始める。コンピュータゲームは以前から存在しましたが、いずれも文字端末を用いたものでした。しかし 70 年代の後半から少しずつ画像出力装置が安価になり、その結果、ビデオゲームが商業ベースに乗ってきます。

38. ☆　公開鍵暗号法の研究が進む。通信に暗号はつきものですが、その理論の中でも最も有名かつ最もよく利用されている方法です。

1980 年代

39. マイクロソフト社が MS-DOS を開発（1981）。

40. Sun 社がワークステーションを発表（1981）。このコンピュータ以降「ワークステーション（WS）」という言葉が高性能デスクトップ型コンピュータの名称として定着しますが、最近では WS とパソコンの境界があいまいになり、死語になりつつあります。

41. IBM 社が IBM PC を発表（1981）。現在のパソコンの原型です。それまで IBM 社は汎用大型コンピュータの開発、販売で世界の頂点に君臨していましたが、自社の開発したこのパソコンによって皮肉にもハードウェア・メーカーとしての頂点の地位を手放すことになります。

42. ☆　プログラミング言語 C++ が設計される（1983）。本学科でみなさんが習得する言語です。オブジェクト指向言語でありながら、伝統的なプログラミングスタイルを踏襲しており、広く受け入れられています。

43. 任天堂ファミリーコンピュータを発売（1983）。任天堂がトランプなどの卓上娯楽品販売の会社から大転換、飛躍するきっかけとなりました。これまでに約 6000 万台が出荷されました。

44. アップル（Apple）社が Macintosh シリーズを発表（1984）。

45. 円周率が 10 億桁まで計算される（1989）。10 億桁の数値そのものは重要ではなく [11]、コンピュータの計算能力の象徴として重要です。

46. 論理型プログラミング言語 Prolog が普及する。本学科でみなさんが習得する言語のひとつです。

1990 年代

47. ☆ UNIX ライクなオペレーティングシステムのカーネル（中核プログラム） Linux[12] が発表される（1991）。無償（フリー）かつクリーン（ソースプログラムが公開されていてプログラムの内容が第三者にも把握できる）な基幹ソフトウェアとして現在、広く使用されています。

[11] SF 小説の中には、円周率の膨大な数字列の中に宇宙の真理が秘匿されている、というモチーフを持つものがありますが、はたして...

[12] 正式な発音は規定されていませんが、多くの人がリナックスと発音するようです。

48. ☆ World Wide Web（WWW）プロジェクトが発表される（1991）。当時普及しつつあったインターネット上の応用手法の主流となり、インターネットを一般家庭まで普及させる原動力となりました。インターネットは当初は研究者間のデータのやり取りなどに使用されていましたが、1990 年半ばの商用化によって個人の趣味的な homepage などが作られるようになり、現在の状況を迎えます。インターネットの普及・発展は、人類史上の最重要事項のひとつになると思われます。

49. ソニー・コンピュータエンタテインメントがプレイステーションを発売（1994）。

50. ☆マイクロソフトウェアが Windows 95 を開発、発売（1993）。Windows 自体は'80 年代から開発、販売されていましたが、CPU のパワー不足からユーザを満足させるものではありませんでした[13]。しかし'90 年代に入ってやっと CPU のパワーが追いつき、急速に広まっていきます。特に機種非依存[14] を実現した Windows95 からはほとんど全ての PC マシンに実装されるようなり、パソコンの標準 OS の地位を得ています。

51. ☆サン・マイクロシステムズにより、プログラミング言語 Java の開発環境が公式にリリースされる（1996）。

52. IBM のチェス専用スーパーコンピュータ、ディープ・ブルー（Deep Blue）がチェス世界チャンピオンガルリ・カスパロフに勝利した（1997）。

2000 年代

53. 超並列コンピュータが開発される。Cray1 のような一台で高速演算を行う方式は 90 年代初めに性能向上が鈍化し、代わって安価なコンピュータ数千台を同時に実行させて高速演算を行う並列方式へ移行しています。特に最近ではインターネットを使った並列方式の研究が盛んです（2 章 GRID コンピュータを参照）。

54. ノートパソコンが普及する。1980 年代末に商用化されたノートパソコンは携帯性の良さから急速に普及しました。部品の小型化、低電力化、バッテリ性能の向上、液晶技術の進歩などで家庭用、個人用にはノートパソコンの性能で事足りる状況です。ちなみに 2015 年 1 年間の世界全体でのパソコン（というべきか PC）の出荷台数は 3 億 1,300 万台であり、その内にノートパソコン（ノート PC）の占める割合は約 68%に当たります。

55. ☆ **オブジェクト指向**（object-oriented）の実用化。1980 年から始まったオブジェクト指向プログラミングは 2000 年代に入り、実用期に入ります。本学科でも必修科目のひとつです。

[13] マウスをクリックしてからウインドウが開いていく様子がスローモーションのように見えた、というくらいに CPU のパワー不足は深刻でした。

[14] かつては同じ PC という規格のパソコンであっても、A 社製では動くが、B 社製では動かないという状況が普通でした。

表 3.1: 家庭用ゲーム機の累計販売台数

発売開始年	ゲーム機名	販売台数（万）
1977	Atari 2600	3,000
1983	ファミリーコンピュータ	6,200
1988	メガドライブ	4,000
1989	ゲームボーイ	11,900
1990	スーパーファミコン	4,900
1994	PlayStation	12,500
	セガサターン	920
1996	Nintendo64	3,300
1998	ドリームキャスト	910
1999	PlayStation 2	15,500
2002	Xbox	2,400
2004	ニンテンドー DS シリーズ	15,400
2005	Xbox360	7,800
	PlayStation 3	7,900
2006	Wii	10,000
2011	ニンテンドー 3DS	6,400
2013	PlayStation 4	5,300

販売台数は主に Wikipedia による

56. 家庭用ゲーム機が普及する。表 3.1 に過去の代表的なゲーム機の発売開始年と累計販売数をまとめました。ゲーム機は同一製品が大量に売れるため、開発に膨大なコストを掛けて安価に販売してもビジネスとして成り立つビジネスモデルになっています。PlayStation 3 のように、ひと昔前のスーパーコンピューター並みの計算能力を持つものも開発されています。

57. ☆ **コンピュータウイルス** が発生する。ネットワークや記録メディアの進歩と共に、コンピュータシステムに障害を起こさせるソフトウェア（ウイルス）が大量発生し、ときとして社会全体に深刻なダメージを与えるようになってきました。

58. Apple 社が iPhone を発売する（2007）。スマートフォンの歴史は実質的にここから始まります。スマートフォンは携帯電話の発展形と見なされますが、実際にはスマートフォンはコンピュータとしての機能が主であり、膨大な種類のアプリケーション（アプリ）が提供されています。それによって私たちの生活様式は大きく変化しています。ちなみに iPhone シリーズは 2014 年末までに 6 億 6300 万台を出荷しています。

59. Web サービス会社 Amazon（1994 年設立）、Google（1998 年設立）、ソーシャル・ネットワーキング・サービス（SNS）Facebook などが急成長する。

第4章　コンピュータの基本構成

前章まではコンピュータの分類、歴史を述べてきました。コンピュータを遠くから眺めることはここまでとして、この章からはコンピュータの動く仕組みを近くから詳細に見ていきます。

まずこの章ではコンピュータの基本構造を概説します。

4.1　基本構成要素

どんなコンピュータも、**演算装置**（CPU、プロセッサとも呼びます）、**記憶装置**、**入力装置**、**出力装置**の四つの基本構成要素から成っています（図 4.1 参照）。

通常、入力装置で取得したデータは演算装置へ転送され、それが記憶装置へ書き込まれます。入力装置から直接、記憶装置へデータが転送されることは通常は行われません[1]。記憶装置と演算装置の間では何度もデータをやりとりし、データの加工を繰り返します。そして加工されたデータは出力装置へ転送されます。コンピュータの利用者は、出力装置への出力結果を見て、次の入力を行います。その入力から次の出力が作り出され、... というサイクルを通して、利用者はコンピュータを操作していきます。

上の四つの装置のどのひとつが欠けて、それをコンピュータとは呼びません。以下の節ではそれぞれの構成要素について述べます。

4.2　入力装置

コンピュータに限らず、何らかの仕事をする機械は、外界の情報を内部へ取り込む入力部を持っていなければなりません。コンピュータはその入力に応じて自身の内部状態を更新して行きます。コンピュータ利用者の立場から見れば、利用者は入力装置を通してコンピュータを操作します。

キーボード、**マウス**は現在の標準的な入力装置です。最近ではタッチパネルが普及しつつあります。それ以外にもトラックボール、イメージスキャナなども入力装置です。ゲーム機のコントロール・パッドも立派な入力装置です。ネットワーク接続用ソケットや USB ソケットは一般には入力装置とは呼びませんが、広い意味では入力装置になり得ます。

[1] 演算装置を介さないデータ転送処理を DMA 転送（Direct Memory Access）などと呼び、高速処理が要求される箇所で使用されます。

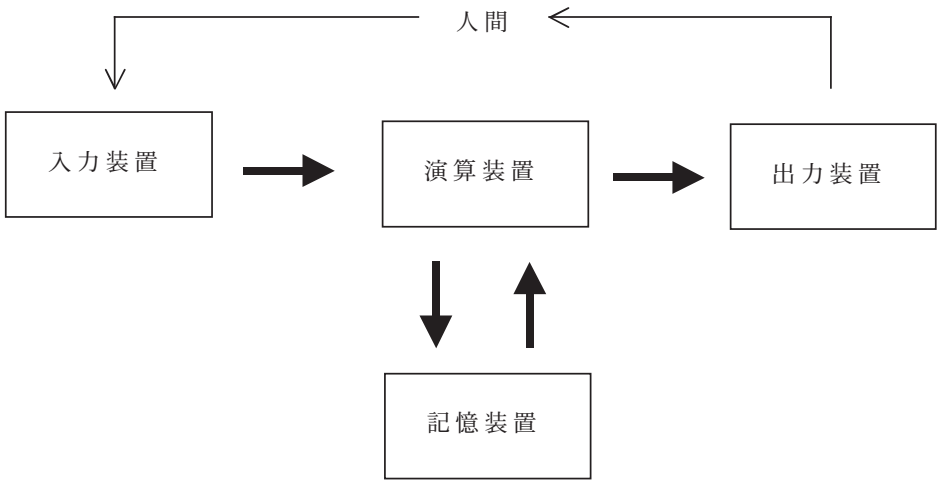


図 4.1: コンピュータの基本構成要素

キーボードやマウスを持たず、単にネットワークに接続されただけのコンピュータもありえます。そのような場合、ネットワーク経由で別のコンピュータのキーボードやマウス情報を取り込むことも可能です。

今や全く利用されませんが、20 年以上前には紙テープや厚紙カードが入力に利用されていました [2]。さらに古い ENIAC の時代には入力はパネル上の数百個のスイッチや、ケーブル・ジャックの接続によって行っていました。

4.3 記憶装置

入力された情報や演算装置による計算結果を一時的/長期的に蓄える装置は全て記憶装置です。単に記憶装置あるいは **メモリ** (memory) と言えば、一般には演算装置から直接読み書き可能な **メイン・メモリ** (主記憶装置) を指しますが、広い意味ではハードディスクなども記憶装置です。

最近のコンピュータでは、記憶装置は演算装置からの距離によって階層化されています。演算装

[2] 昭和 30 年代、40 年代の SF 映画などで見ることができます。

置に最も近い層にはキャッシュ・メモリ[3]があり、次にメイン・メモリがあります。ここまでを通常、**内部記憶装置**と呼びます。内部記憶装置の外には**外部記憶装置**があり、具体的にはハードディスクなどの周辺装置、さらにネットワークを介してアクセス可能なネットワーク共有型ハードディスク、さらに遠くにはインターネットを介してアクセス可能な記憶装置等が考えられます。

記憶装置には二つの役割があります。ひとつは**高速**にデータを読み書きすることです。もうひとつは**大量**のデータを記憶することです。通常、演算装置に近いほど高速、小容量、高価であり、演算装置から離れるに従い、低速、大容量、安価です。演算装置の近くでは少量の同じデータが何度も演算装置によってアクセスされる傾向にありますから、高速性が要求されます。これに対して、演算装置から遠いデータは頻繁にアクセスされることは稀（まれ）です[4]から、高速アクセスよりも大容量であることが要求されます。

記憶装置は、理論的に言えば、コンピュータに状態を導入します。コンピュータの動作とは、ある状態から別の状態への状態変化と見なすことができます。通常、記憶装置を持たないコンピュータは考えられません。

4.4 演算装置

CPU（Central Processing Unit）あるいは単に**プロセッサ**とも言います。このテキストでは「プロセッサ」という呼び方を用います。

コンピュータにおける代表的な演算は数値計算です。しかしコンピュータは単に計算を行う機械ではなく、様々な論理演算も可能です。またそのような演算をある規則に従って順次自動的に実行していくことも可能です。実行の具体例は 5 章以降で述べますが、このテキストで述べる演算装置はいずれもノイマン型と呼ばれる構造を持っています。

現在、世の中で最も広く利用されている商用プロセッサは、Intel 社の**x86 系**プロセッサです。最近ではそれを 64bit 化した x86-64（または簡単に x64）プロセッサ が主流です。

x86 系とは、Intel 社が 1978 年に開発した 16bit プロセッサ 8086 の後継プロセッサの総称です。もちろん、8086 は現在は発売/使用されておらず、Xeon や Core Duo、Core i-3/5/7 などの x86 系を拡張したものが使用されています。

その他にもサン・マイクロシステムズ社の UltraSPARC プロセッサ、IBM 社の POWER プロセッサ、PlayStation 3 に搭載された Cell プロセッサ[5]などが有名です。また、最近では 1 台のコ

[3] メインメモリの低速性を補うために、演算装置とメインメモリの間に存在し、最も重要なデータのみを格納する高速かつ小容量のメモリのことをキャッシュメモリと呼びます。

[4] 通常コンピュータでは、頻繁にアクセスされるデータは演算装置の近いところへ（半）自動的に移動されるような設計がなされます。

[5] ソニー・コンピュータエンタテインメント、ソニー、IBM、東芝の共同開発です。

ンピュータに複数のプロセッサを搭載するマルチコア・プロセッサが普及しつつあります[6][7]。

4.5 出力装置

出力装置は、コンピュータの実行結果を外部へ通知するために必須です。

最も代表的な出力装置は、**液晶ディスプレイ装置**（liquid crystal display、LCD）です。以前は CRT（Cathode-Ray Tube）ディスプレイ装置 — いわゆるブラウン管ディスプレイ装置 — が主流でした。20 年以上前の CRT ディスプレイ装置では横 80 文字、縦 24 文字の英数字と簡単な記号しか表示できませんでした。しかしメモリの価格が下がるに従い、（本来は計算結果の記憶などに用いる）メモリの一部分を画面情報の記憶[8]に充てる余裕が生じ、その結果ディスプレイは単なる文字を表示するだけではなく、画像を表示できるようになってきました。そして'80 年代後半からは白黒の高精細 CRT ディスプレイ装置が商用化され、さらに'90 年代半ばからは現在のような**フルカラー**高精細ディスプレイ装置が一般的に利用されるようになりました。しかし、最近では CRT は急速を数を減じ、代わりにカラー液晶パネルが用いられています。

プリンターも重要な出力装置です。かつて'70 年代には CRT ディスプレイ装置すら存在せず、英数字のみを印字可能なプリンタがほとんど唯一の出力装置でした。3 章で見たようにレーザープリンターが広く使用されるようになるのは、'80 年代の半ばからです。また'90 年代に入ると、レーザープリンターよりも廉価なインクジェット方式のプリンターが実用化されます。

その他の出力装置として、ネットワーク接続用ソケットや USB ソケットも出力部と考えてよいでしょう。

やや横道にそれますが、最近では**生体**とコンピュータをつなぐ研究が盛んです。人工視力や人工聴力の研究は 20 年以上前から行われています。また表面筋電位を使って人体の動作をサポートする装置も実用化に近づいています。人間の脳波をコンピュータに取り込み、キーボードを介せず、直接、頭で考えたことをコンピュータに入力する研究も始まっています。これらの装置は、身体に障害者を持つ人をコンピュータでサポートする上で非常に重要です。また、生体とコンピュータを直接つなぐ装置[9]として、今後の社会の在り方に大きな影響を与えるかもしれません。

---

[6] プロセッサ内の特に演算部、実行制御部のことをプロセッサ・コア（processor core）と呼びます。マルチコアとは、プロセッサコアが複数個（マルチ）搭載されていることを意味します。

[7] PLAYSTATION 3 の Cell プロセッサには計 9 基のプロセッサが搭載されており、これらを並列に実行し、高速なゲーム画面の描画を行っています。

[8] ディスプレイ装置の情報の記憶に特化したメモリを特にビデオ・メモリと呼びます。

[9] 映画「マトリックス」で主人公らが後頭部にコンピュータの端子を差し込んでマトリックスの仮想世界に接続していました。あのような接続装置は絵空事ではありません。

第5章　Scratchプログラミング（基本編）

これからの2章はプログラミングの入門です。この章ではプログラミング言語 Scratch について簡単に紹介します。そして次章で様々な種類の問題を Scratch でプログラミングします。

5.1　Scratchとは

Scratch は、マサチューセッツ工科大学（Massachusetts Institute of Technology、MIT）のメディアラボにおいて開発されたプログラミング言語およびその実行環境です。アイコン（コンピュータディスプレイ上の小さな図形の部品）を画面上で組み合わせることでプログラムを作成します。

プログラミング言語の種類としてはビジュアルプログラミング言語の範疇に入ります。プログラムの実行が、様々なイベント（マウスのクリック、キーインなどのユーザの入力など）によって開始されるため、イベント駆動型言語という特徴もある。加えて複数のプログラムが同時に実行できるため、並行プログラミング言語でもあります。このように、様々な機能を持つ言語ですが、直感的な操作でプログラミングできるため、小学生のプログラミング学習にも用いられています[1]。小学生が使うから幼稚ということはありません。大学生の利用にも耐えうる内容を持っています。

本学科では１年生後期からプログラミング言語 C++を本格的に学びます。C++と Scratch はかなり異なる外面の言語ですが、しかし、プログラミングの本質において両言語は同一です。この授業では、今後の C++学習の助けになること、プログラミングの本質を理解するきっかけになることを目的として、あえて Scratch を選びました。

5.2　Scratchプログラミングのための準備

Scratch のインデックスページ（トップページ）は以下です。

`https://scratch.mit.edu`

適当な web ブラウザで上の URL を訪問してください。この URL をいちいちキーボードからキーイン入力するのは面倒かもしれません。その場合、適当な検索エンジンで「scratch」を検索してください。ほとんどの場合に上の URL が検索結果のトップに表示されますから、それをクリックすれば上のページを訪問できるはずです。あるいは、もしこのテキストを pdf 表示ソフトで読んでいるならば、上の URL 部分をクリックすれば web ブラウザが開くかもしれません。

[1] もともと MIT は様々な教育用言語の開発を行ってきており、Scratch はそのひとつです

図 5.1: Scratch のインデックスページ

平成 30 年 4 月現在、インデックスページは図 5.1 のようなものです。

もしページが英語で表示されたならば、ページの最下部の言語選択メニューで日本語を選択してください。ただし、Scratch のサイトはもともと英語がメインですから全てを日本語化できる訳ではありません。

このページの上部に三つの入り口が大きく表示されています。

やってみる ... 実際にプログラムを作り、実行するページへ飛びます。ここがメインの作業場になります。

例を見る ... MIT の Scratch チームが用意した様々な Scratch プログラムの例が紹介されています。

Scratch に参加 ... Scratch のアカウントを作成します。アカウント作成後、作成したプログラムを Scratch のサーバーにそのまま保存できるようになります。

また、ページ最上部のメニューには以下のボタンがあります。

Scratch ... Scratch の全てのページの左上に SCRATCH ボタンがあります。これをクリックすると常に図 5.1 のインデックスページへ戻りますから、自分の場所が分からなくなった場合にはこのボタンをクリックしてください。

作る ... 「やってみる」とほとんど同じですが、右側に Scratch のチュートリアルが表示されます。

見る ... 世界中の人が作成した様々な Scratch プログラムが紹介されています。

話す ... Scratch の掲示板です。使用言語は英語です。

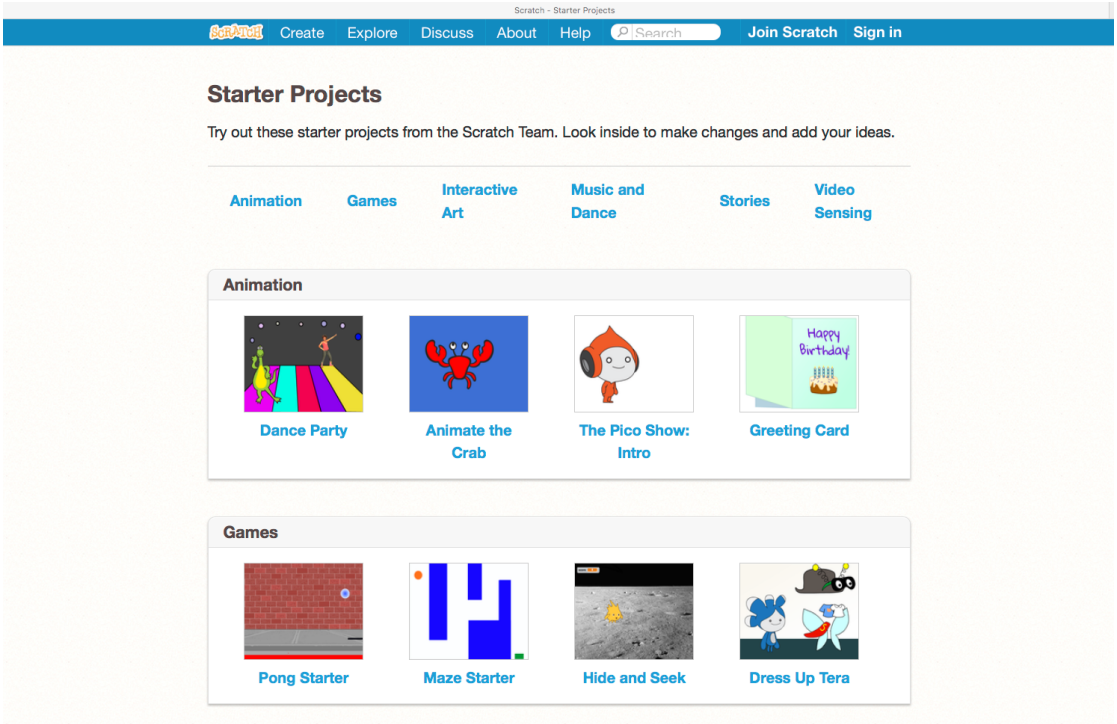


図 5.2: Scratch の「例を見る」のページ

Scratch でできそうなことを手早く知るには、「例を見る」のページへ行ってみましょう。それが図 5.2 です。図 5.2 だけでも 8 個のプログラムが示されています。興味のあるものをクリックすると、そのプログラムを実際に動かしてみることができます。

注意： 現在の Scratch は、2013 年 5 月に更新されたバージョン 2.0 です。インターネット等の解説、書籍の中にはこれよりも古いバージョンに関するものもありますが、内容がかなり異なりますので注意してください。

Scratch は Web ブラウザの上で動作しますから、ソフトウェアを PC へインストールする必要はありません [2]。ただし、ブラウザはアドビ Flash Player を使用します。よって、Flash Player を実行できない iPhone/iPad の Safari ブラウザでは Scratch は実行できません [3]。

実際に Scratch でプログラムを作るには、図 5.1 の「やってみる」をクリックします。図 5.3 がクリック後に移動したページです。これは Scratch のオンラインエディタです。このエディタは大きく分けて 4 つのエリアから成ります。図 5.4 はそのエリアの場所と名称です。実線で囲まれた 4 つのエリアの概要は以下の通りです。なお、Scratch ではプログラムのことをスクリプトと呼びますが、この授業では無用な混乱を避けるために、可能な限り「プログラム」という呼び方を持ちます。

[2] Web ブラウザを利用しない実行方法もありますが、煩雑なので説明を省略します。
[3] iPhone/iPad で Scratch を動かすには、Flash Player が動作する別のブラウザを利用せねばなりません。

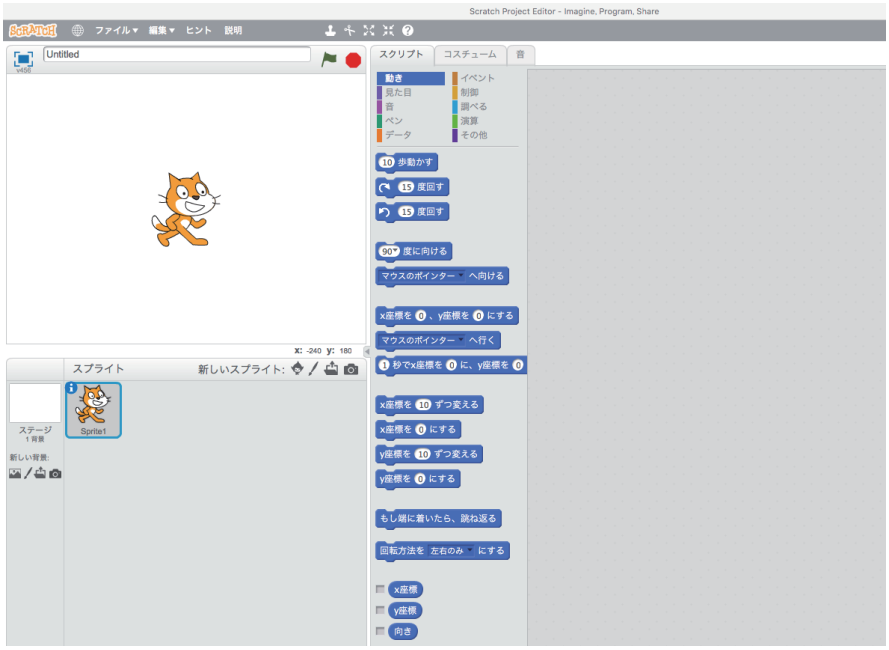


図 5.3: Scratch の「やってみる」のページ

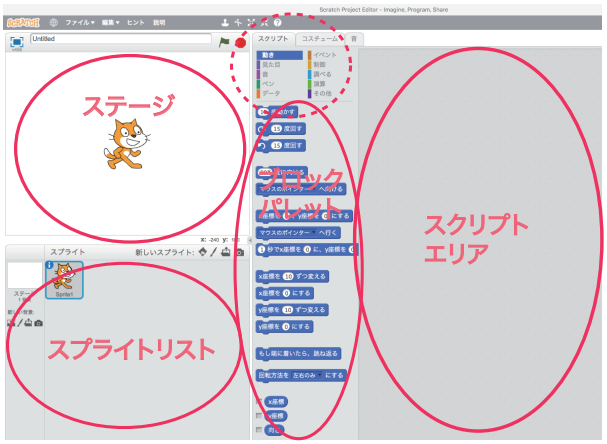


図 5.4: エリア名称

ステージ ... 実際にプログラムの実行の様子が表示されるエリアです。

スプライトリスト ... プログラムで動く全てのスプライトのリストです。

スクリプトエリア ... 各スプライトに付随するプログラムの編集エリアです。

ブロックパレット ... プログラム中で使用できる部品 — Scratch ではこれを**ブロック**と呼びます — のリストです。

スプライト（sprite=妖精、小人）とは一般にビデオゲームの中で動き回る小さなキャラクタのことを言います[4]。

スプライトは劇における配役、俳優です。ステージは舞台です。スプライトリストは出演する配役リストです。スクリプト（script）は各配役のための台本です。ブロックパレットは、台本の中で使用できる最小単位の部品の一覧です。この一覧から適切な動作を適切な順番で組み上げて台本を仕上げます。プログラム完成後にはそれぞれの台本に従って俳優が舞台の上で自律的に動きます。

スプライトのための多数のブロックはブロックパレットに用意されています。図 5.4 の点線の囲みにブロックの種類がメニューで表示されています。ブロックの種類には以下のものがあります。

動き ... ステージ上でのキャラクタの位置、傾き角などを変更するブロック

見た目 ... ステージ上でのキャラクタの表示方法を変更するブロック

音 ... 様々な音を任意の大きさ、テンポでコンピュータから鳴らすブロック

ペン ... キャラクタに付随するペンでステージに線を引いたり、線の書類を変えるブロック

データ ... 変数（数値を格納するメモリエリア）とリスト（数値の並び）に関するブロック

イベント ... プログラム実行途中の様々なイベント（事象、出来事）を表すブロック

制御 ... 条件分岐、繰り返しなどの、プログラム処理の流れを制御するブロック

調べる ... プログラム中の様々な状態を真偽値で判定するブロック

演算 ... 四則演算、初等関数、論理演算を行うブロック

その他 ... 手続き抽象化などの拡張機能を支援するブロック（本授業では取り扱わない予定）

実際にエディタに表示されたブロックパレットは図 5.5、図 5.6、図 5.7 の通りです。多種多様な機能を使うことができると分かります。なお、この授業では Scratch をプログラミングの体験用に用いているだけであり、Scratch の講座ではありませんから、全てのブロックを解説することは行いません。興味のある人は、以下の Wiki ページなどを自分で調べてください。

[4] ゲーム中で動く被写体を 3D CG などの技術でまじめに描画すると CPU の計算負荷が無駄に大きくなり過ぎるため、コンピュータ内には通常、小さなキャラクタを簡易に描画するハードウェアが組み込まれています。このハードウェアで描画されるキャラクタを特にスプライトと呼びます。

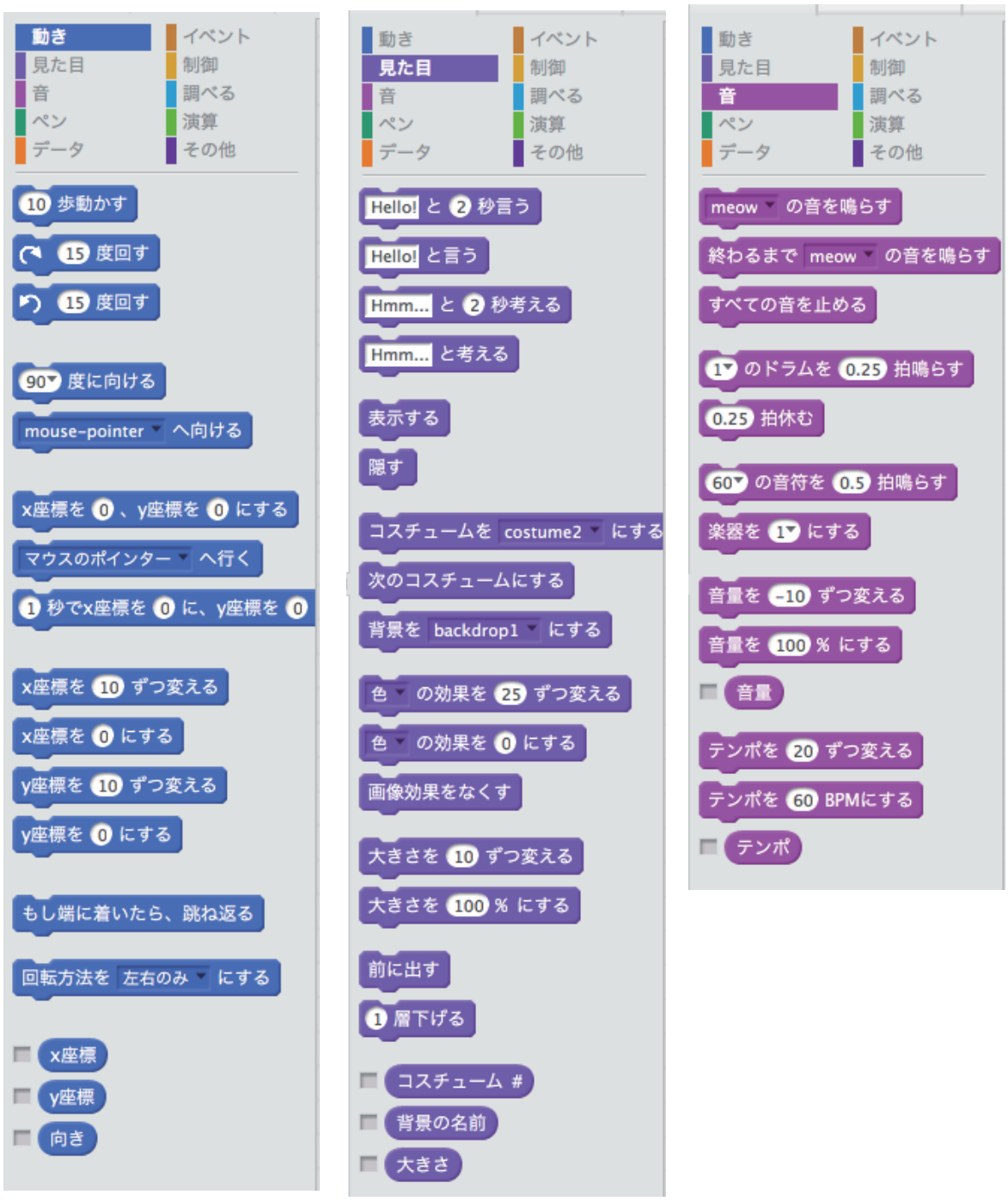


図 5.5: スプライト用の「動き」、「見た目」、「音」のブロックパレット

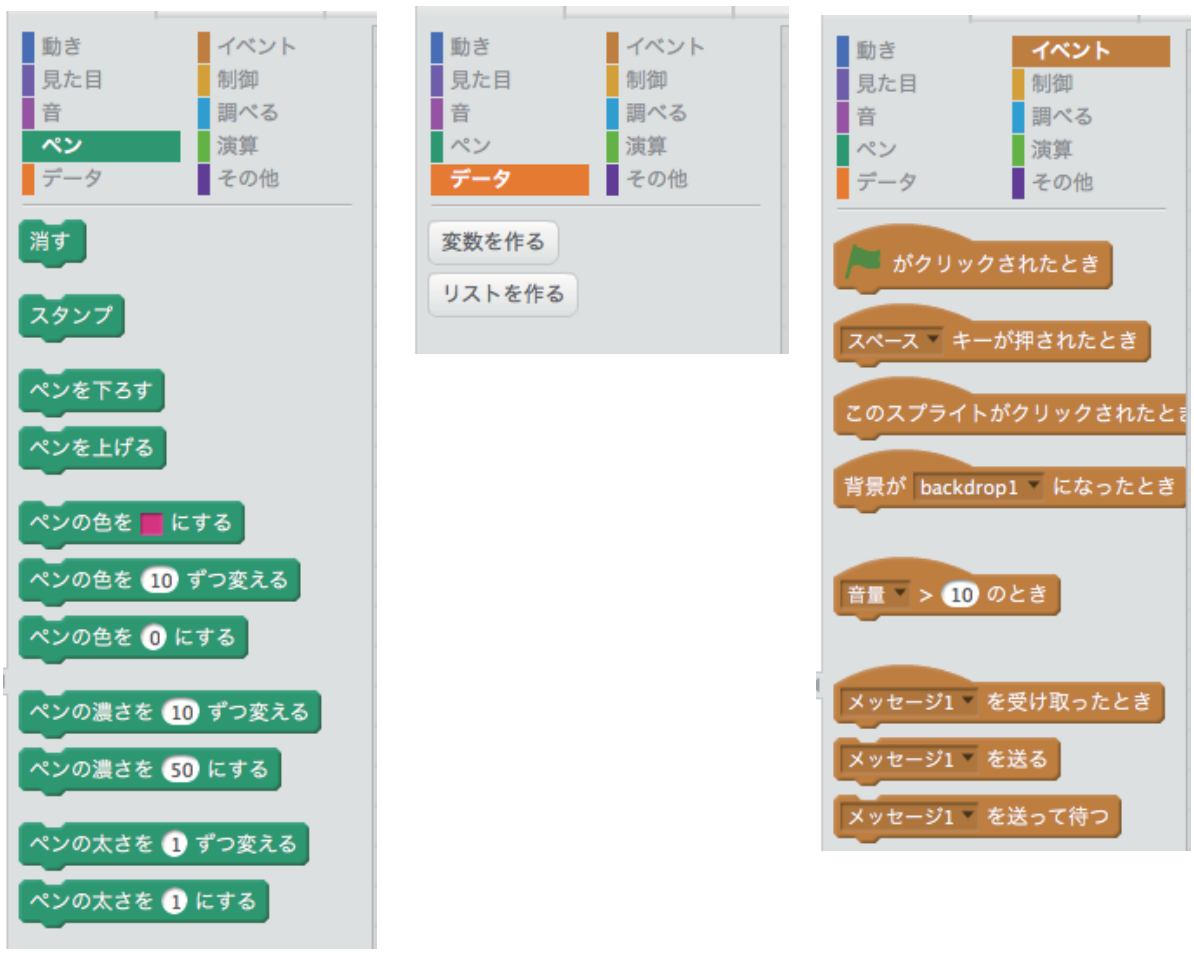


図 5.6: スプライト用の「ペン」、「データ」、「イベント」のブロックパレット

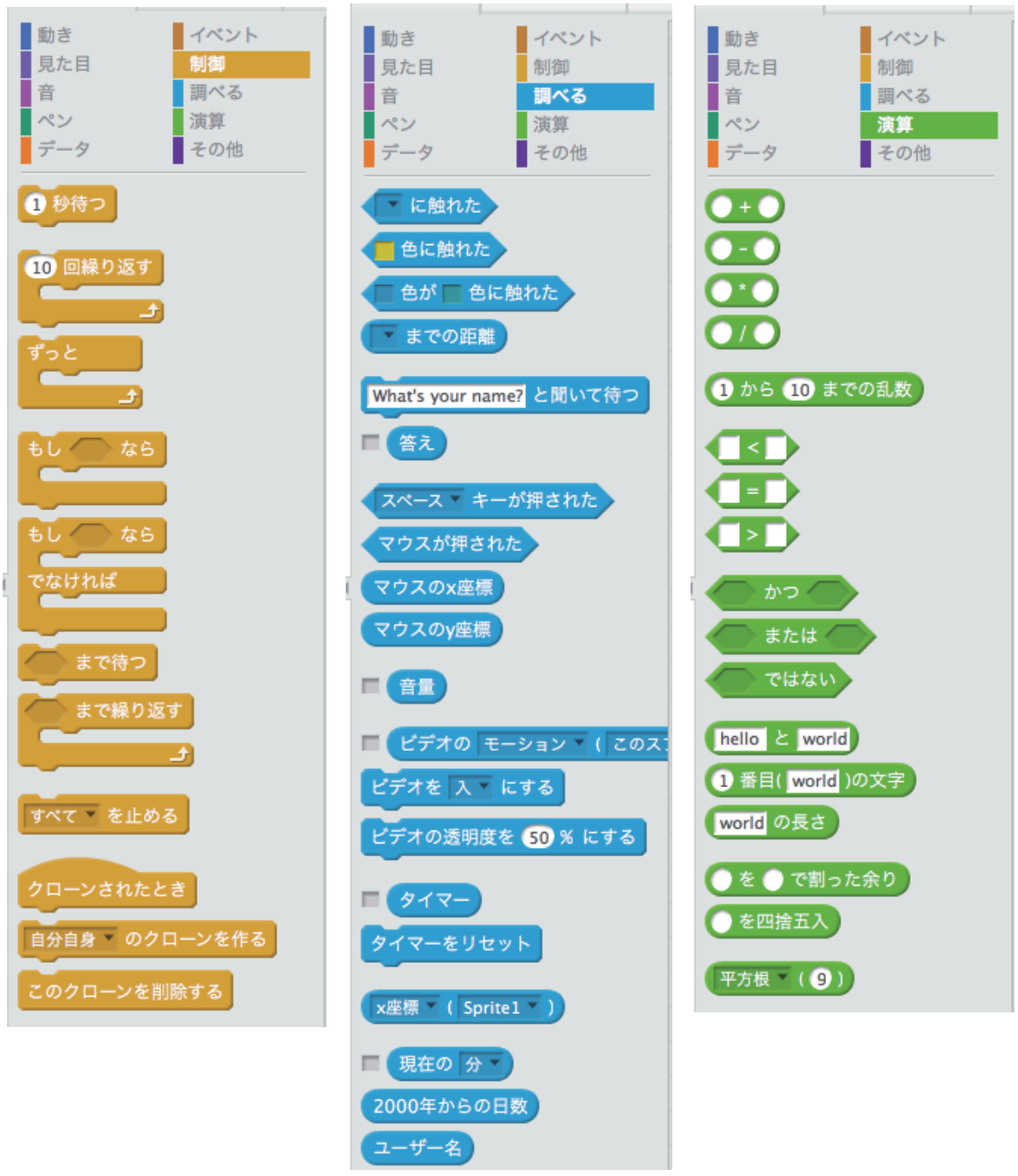


図 5.7: スプライト用の「制御」、「調べる」、「演算」のブロックパレット

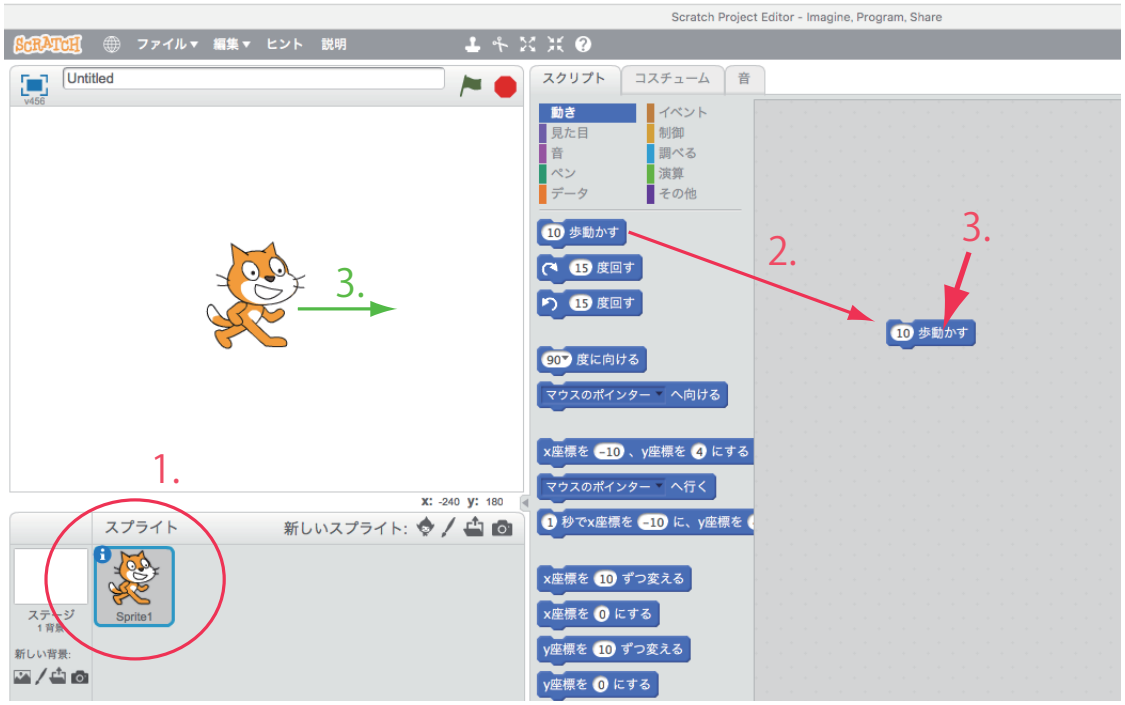


図 5.8: 猫が歩く

講義テキストの用語および掲載画像について　Scratch に関するこのテキストの用語は、日本語版 Scratch Wiki：

`http://jp.scratch-wiki.info/wiki/`

に従っています。講義テキストでは Scratch に関する詳細を述べる余裕はありませんから、不明な点、疑問などは上のページを参考にしてください。

　講義テキストに載せた Scratch の画像は全て Scratch オンラインエディタの画像をデジタルコピーしたもの、または日本語版 Scratch Wiki の画像を転載したものです。

5.3　簡単なプログラムの編集と実行

　さて、実際に簡単なプログラムを作ってみましょう。基本的な作業は、ブロックパレットから選んだブロックをスクリプトエリアにドラッグし、ブロックを組み上げていくことです。マウスの操作は直感的ですから、あまり悩まず、ともかく手を動かしてみてください。

5.3.1　猫が少し歩く

　図 5.8 を参考にしながら以下を試してください。

(a) 2つのブロックを組み合わせる

(b) 猫が 15° だけ下を向く

図 5.9: 猫が少し歩いた後、方向を変える

1. スプライトリストで「猫」(Scratch Cat と呼ぶらしい）を選択します（マウスでクリックします)。たぶん、初めてこのページを訪問した状態ではすでに猫が選ばれているはずですから、実際には何もする必要はありません。

2. 次に、「動き」のブロックパレットの一番上に表示されている「(10) 歩動かす」のブロックをマウスでスクリプトエリアへドラッグします[5]。

3. そして、ドラッグしたブロックをクリックしてみましょう。猫が右へ少し移動したことに気づいたでしょうか。必要ならば、何度でもクリックしてください。

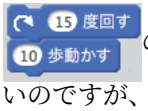
以上が、最も単純な Scratch プログラムの実行です。

5.3.2 猫が少し歩いた後、方向を変える

プログラムを少し複雑にします。図 5.9 のように、

1. 「動き」のブロックパレットの、上から 2 番目に表示されている「(15) 度回す」ブロックを先ほどの「(10) 歩進む」ブロックの真下へ、2つのブロックの凹凸がピッタリと一致するようにドラッグします。そうすると、2つのブロックの辺どうしが張り付くはずです（図 5.9 (a) 参照)。

2. それらブロックのどちらかをクリックしてみましょう。猫が右へ少し移動し、下方向（時計回り）へ 15 度だけ傾きます（図 5.9 (b) 参照)。

各ブロックはスプライトの基本動作を表しています。ブロックが張り合わされた（組み合わされた）場合には、上方のブロックから順に下へ向かって実行されていきます。実行の順序は重要です。たとえば、 

の場合では回転と前進が逆の順序になります。この例の場合には逆であっても大差ないのですが、一般にはブロックの動作順序を入れ替えるとプログラムの実行結果が全く異なることがありますから、細心の注意が必要です。

[5]「ブロックをドラッグ（drag）する」とは、マウスボタンをそのブロックの上で押し下げ、押し下げた状態を維持したまま、マウスポインタを目的の位置へ移動させることです。

(a) 緑の旗と
赤いストップボタン

(b) 緑の旗で実行を開始する
プログラム

(c) ステージのみを表示す
るボタン

図 5.10: 緑の旗をクリックすると...

ブロックの操作に関する注意　ブロックを張り合わせる操作は実際にマウスで触っていると自然と体得できるはずですが、迷いそうなところを以下、簡単に注記します。

移動：　図 5.9 のように組み合わされた複数のブロックをまとめて移動させたいときには、**最上段**のブロックをマウスでポイントしてドラッグします。

切り離し：　組み合わされた複数のブロックを切り離したいときには、最上段**以外**のブロックをマウスでポイントしてドラッグします。

削除：　（単体、複数に限らず）ブロックをスクリプトエリアから削除したいときには、ブロックをブロックパレットへドラッグします。

5.3.3　緑の旗をクリックすると、猫が少し歩いた後、方向を変える

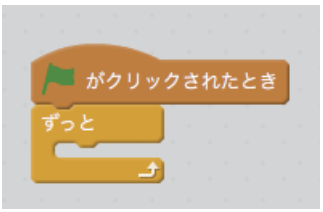
スクリプトエリアのブロックをクリックしてプログラムを実行する方法は、プログラムを開発している途中では便利です。しかし、通常、プログラムの実行中にそのプログラムをわざわざユーザに見せることはしません（舞台で劇が上演されているときに舞台裏を見せることはしません）。

そのための仕組みとして Scratch には図 5.10(a) の緑の旗をクリックするとプログラムの実行が開始される機能が用意されています。この機能を利用するには、「イベント」のブロックパレット（図 5.6 右列）から最上段のブロック「がクリックされたとき」をドラッグし、図 5.10(b) のようなプログラムを作ります。図 5.10(c) のボタンをクリックすると、Web ブラウザ全体にステージだけが表示され、スクリプトエリアは表示されません。このときに画面右上の緑の旗をクリックすればプログラムの実行が開始されます。

5.3.4　緑の旗をクリックすると、猫が回転し続ける

同じ処理を永遠に繰り返すことを**無限ループ**（infinite loop）と呼びます[6]。永遠に繰り返すとコンピュータの実行が止まらない＝コンピュータが壊れる、などと恐る必要はありません。Scratch

[6] コンピュータの専門用語と思いますが、すでに日常用語として認知されているかもしれません。



(a) 「ずっと」ブロックをドラッグ

(b) 図 5.9(a) のプログラムを挟み込む

図 5.11: 無限ループを作ってみる

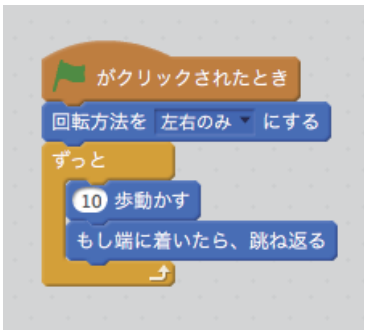


図 5.12: 猫が右往左往する

には強制終了ボタンが用意されており、それが図 5.10(a) の赤い 8 角形のボタンです。必要に応じてこのボタンを押しましょう。

無限ループの機能は、「制御」のブロックパレット（図 5.7 左列）の、上から 3 つ目の「ずっと」ブロックです。まず、これを図 5.11(a) のように組み上げましょう。

「ずっと」ブロックは、その右側に隙間があり、他のブロックをそこへ挟み込めるような形になっています。そこで、その隙間に図 5.9(a) の 2 つのブロックを挟み込み、図 5.11(b) のように完成させてください。図 5.11(a) の「ずっと」ブロックには 2 つのブロックを挟み込むだけの十分な隙間がないように見えますが、挟み込みたいブロックを隙間へドラッグしていくと自動的に隙間が拡がる仕掛けになっています。

プログラム完成後、緑の旗をクリックしてプログラムを開始すると、猫がステージでくるくると回転し続けます。プログラムを停めたいときには、赤いストップボタンをクリックします。面白くなりそうな予感がしてきました。

5.3.5 緑の旗をクリックすると、猫が右往左往し続ける

Scratch プログラムの紹介として最後にもうひとつ、よく取り上げられる例を紹介します。

図 5.12 を動かしてみましょう。猫がステージの左端から右端の間を行ったり来たりします。実際に何が起きるかは、自ら図 5.12 のプログラムを動かしてみてください。

5.3.6 ブロックの種類

ここまでの説明ですでに気づいていることでしょうが、Scratch のブロックは幾つかの特徴的な形状をしており、形状によってはブロック間で組み合わせができない場合があります。実は Scratch ブロックの形状は以下の 6 種類です[7]。

ハット（hat）ブロック  ... プログラムの開始処理に相当するブロックです。「イベント」のブロックパレット（図 5.6 の右列）に多く載っています。その形状から、このブロックの上にブロックを張り合わせることができません。

スタック（stack）ブロック  ... プログラムの途中の処理に相当するブロックです。多くのブロックがこの形状であって、その上下にブロックを張り合わせることができます。

キャップ（cap）ブロック  ... プログラムの終了処理に相当するブロックです。「制御」のブロックパレット（図 5.7 の左列）にいくつか載っている程度です。その形状から、このブロックの下にブロックを張り合わせることができません。

C 型（C）ブロック  ... このブロックの中にブロックを挟み込むことのできるブロックです。「制御」のブロックパレット（図 5.7 の左列）に載っています。複雑なプログラム実行を行うために必要です。

値（reporter）ブロック  ... 数値や文字列を表すブロックです。「調べる」のブロックパレット（図 5.7 の中央列）、「演算」のブロックパレット（図 5.7 の右列）に多く載っています。単独では動作せず、上の 4 種類のブロックの中に組み込んで計算などを行います。

真偽値（boolean）ブロック  ... 条件判定に関わるブロックです。「調べる」のブロックパレット（図 5.7 の中央列）、「演算」のブロックパレット（図 5.7 の右列）に多く載っています。値ブロック同様に単独では動作せず、上の 4 種類のブロックの中に組み込んで複雑な条件判定を行います。

Scratch の入門は以上です。

5.4 処理の実行順序と変数の利用

プログラム実行の基本原理は、多くのプログラミング言語で共通しています[8]。もちろん 1 年生後期から学ぶ C++においても共通しています。基本原理の分類法にはいくつかありますが、ここでは４つの原理で解説していきます。まず最初の原理は以下のものです。

[7] ブロック形状の画像は日本語版 Scratch Wiki から転載しました。
[8] 全く異なる原理で動作するプログラミング言語もありますが、それについては 3 年正で学ぶことができます。

第一原理 プログラムは上から順番にひとつずつ実行されていく。

すでに前節でその簡単な例を見てきましたし、ほとんど自明のことに思えるかもしれません。ここでは、もう少し複雑な例を考えていきます。

5.4.1 変数

コンピュータにメイン・メモリと呼ばれる記憶装置があることは前章で述べた通りです。多くのプログラミング言語では、そのメモリの上の、ある一定の大きさのメモリ領域に自由な名前を付けてデータの記憶に利用できます。その領域のことを **変数**（variable）と呼びます。変数の領域は、コンピュータが自動的に割り当てるため、ユーザがメモリの内部構造を知る必要はありません。変数の名前 — **変数名**（variable name） — は自由とはいえ、名付けには一定の制限があり、一般には英文字の後に英文字または数記号が 0 個以上続くような文字列と考えてください。プログラムで使用できる変数の個数に制限はありませんが、人（プログラマ）がひとつのプログラム中で管理できる変数の数はせいぜい数百個程度が限界です。

Scratch では「データ」のブロックパレット（図 5.6 の中央列）から変数を利用できます。その準備として

1. まず、「データ」のブロックパレット中の「変数を作る」ボタンをクリックしましょう。そうすると、図 5.13(a) のように、「新しい変数」というダイアログボックスが現れますから、ここではキーボードから変数名として「x」を入力し、さらに「OK」ボタンをクリックします。結果、「データ」のブロックパレットは図 5.13(b) のように変化し、変数 x が利用可能になります。

2. 同様にして、変数 y、r を作りましょう。結果、図 5.14(a) のように、変数 x、y、r がパレットから利用可能になります。なお、ブロックでの x、y、r の使い分けは変数名の右側のプルダウンメニュー（図 5.14(a) の青い囲みの中）から変数名を選んで行います。また、5.14(b) のように、ステージの左上に変数名と現在の値が現れます。変数を新規設定した時点では変数の値は 0 です。

以上で三つの変数を利用する準備が整いました。

5.4.2 原点からの距離の計算

Scratch の矩形のステージには x-y 座標が張られており、x の範囲は $[-240, 240]$、y の範囲は $[-180, 180]$ です。中央が座標原点 $(0, 0)$ です。

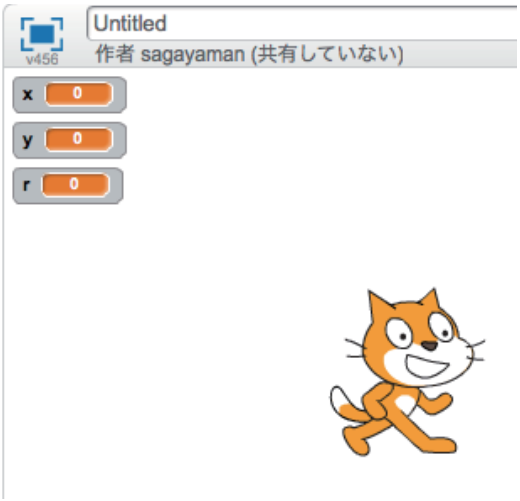
そのことを踏まえ、ここで次のような動作をするプログラムを作ることが考えます。

(a)「変数を作る」をクリックし、変数名 x を入力

(b) 変数 x がブロックパレットで利用可能

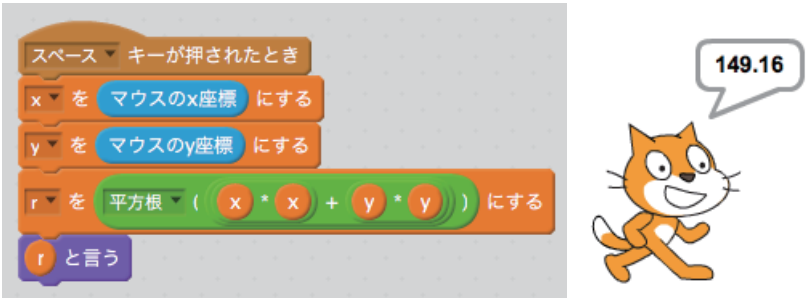
図 5.13: 変数 x の設定

(a) 変数 y、r を追加設定



(b) 変数 x、y、r がステージに表示される

図 5.14: 変数 x、y、r の設定



(a) 原点からの距離の計算プログラム　　(b) 実行結果

図 5.15: 原点からの距離の計算

原点からの距離の計算プログラム

キーボードのスペースキーを押した時のマウスの位置 (x, y) と座標原点 $(0, 0)$ との距離 $\sqrt{x^2 + y^2}$ を猫が教えてくれるプログラムを作りなさい。

そのプログラムのひとつの例が図 5.15(a) です。

考察　「見た目」、「データ」、「イベント」、「調べる」、「演算」のブロックパレットから適切なブロックを選び、図 5.15(a) のプログラムを自ら作ってみなさい。

このプログラムを組み上げた後、マウスをステージ上の適当な位置へ置き、スペースキーをクリックすると、図 5.15(b) のように、猫が距離を答えてくれるはずです。

さて、図 5.15(a) の 2、3、4 段目のブロック「[x] を ( ) にする」、「[y] を ( ) にする」、「[r] を ( ) にする」は、変数に特定の値を格納することを表すブロックです。詳しく述べれば、その変数の表すメモリ中の特定の記憶領域に値を格納する動作です。この動作を多くのプログラミング言語では（もちろん C++においても）**代入**（assignment）と呼んでいます。コンピュータでの主な計算は、(1) 代入する値を計算し、(2) その計算結果を変数へ代入する、という動作を繰り返されることによって実行しており、その意味で代入はプログラムの実行において最も重要な動作のひとつです。

代入される値は**算術式**（arithmetic expression）によって表され、実行時にコンピュータによって自動的に計算されます。

たとえば、単独の数値（たとえば図 5.15(a) の「マウスの x 座標」や変数の値）はそれ自身で算術式とみなされます。

演算子や関数を含む式（たとえば図 5.15(a) の「平方根 (((x)*(x))+((y)*(y)))」）も算術式です。Scratch で使用できる演算は、「演算」パレット（図 5.7 右列）に載っている以下の値ブロック（丸い形状のブロック）です。

( )+( ) ... 

（addition）を表します。

( )-( ) ... 

（subtraction）を表します。

( )*( ) ... 

（multiplication）を表します。乗算演算子は通常の算数、数学では × や • で表されたり、あるいは演算子が省略されることもありますが、Scratch に限らずほとんど全てのプログラミング言語では（もちろん C++においても）乗算演算子は専ら * で表します。Scratch もそれを踏襲しています。記号 * をアスタリスク（asterisk）またはスターと呼びます。日本語訳では星印と呼びます。記号 * を乗算記号に用いるのは、初期のコンピュータではキーボードから × や • を入力できなかったためです。

( )/( ) ... 

（division）を表します。除算演算子は通常の算数、数学では ÷ や分数形式 $\frac{a}{b}$ で表されますが、Scratch に限らずほとんど全てのプログラミング言語では（もちろん C++においても） / を用います。記号 / をスラッシュ（slash）と呼びます。

( ) から ( ) までの乱数 ... 指定された範囲の一様乱数[9] を求めます。範囲が整数値で指定されている場合には整数の乱数を、小数点数で指定されている場合には小数点数の乱数を求めることができます。

( ) を ( ) で割った余り ... 説明省略

( ) を四捨五入 ... 説明省略

[平方根 ( )] ... このブロックの [平方根] の部分は関数名になっており、プルダウンメニューで [絶対値]、[切り下げ]、[切り上げ]、[sin]、[cos]、[tan]、[asin]、[acos]、[atan]、[ln][10]、[log][11]、[e^][12]、[10^][13] に変更できます。これら関数の計算プログラムはあらかじめシステムに組み込まれています。

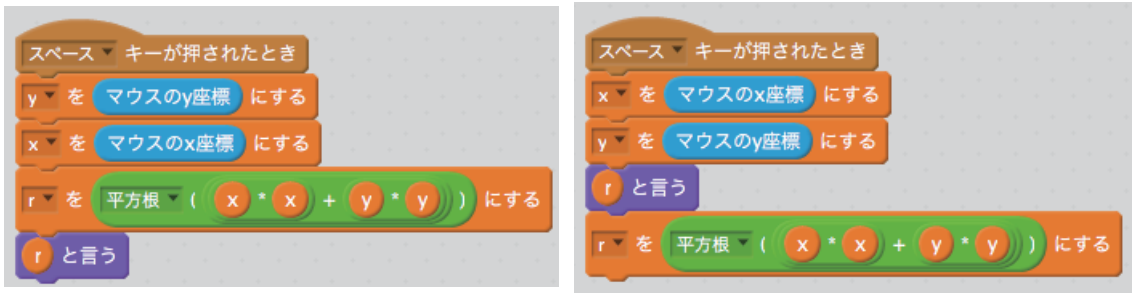
これらの演算を用いることで、様々な複雑な計算が可能です。後章では円周率の計算、2 次元グラフィックスに挑戦します。

5.4.3 プログラムの保存

オンラインエディタで作成したプログラム／作成途中のプログラムをハードディスクにファイルとして保存できます。逆に、保存したプログラムをオンラインエディタに読み込み、再度実行させ

[9] 一様乱数とは、指定さえた範囲の全域で数値が均等な確率で出現する乱数のことです。
[10] ネイピア数 e を底とする自然対数関数
[11] 10 を底とする常用対数関数
[12] ネイピア数 e を底とする指数関数
[13] 10 を底とする指数関数

図 5.16: プログラムのアップロード/ダウンロード



(a) 2 行目と 3 行めの入れ替え

(b) 4 行目と 5 行目の入れ替え

図 5.17: ブロックを入れ替えたプログラム

ることも、プログラムの作成を再開することもできます。オンラインエディタ左上のメニューバーから「ファイル」メニューを選ぶと、図 5.16 のようにアップロード（ファイルの読み込み）とダウンロード（ファイルの保存）のコマンドが現れます。少し大き目のプログラムを開発するときには重宝する機能です。

なお、このアップロード／ダウンロード機能は Scratch のアカウントを必要としません。あくまでも手元のコンピュータとのローカルなアップロード／ダウンロードに過ぎません。アカウントを登録すると、Scratch のサーバーにプログラムを保存でき、開発したプログラムを他のユーザに公開することもできるようになりますが、この授業の範囲内ではアカウントの登録は必須ではありません。

メニューバーには他にも便利な機能があります。地球儀マーク のメニューではオンラインエディタの言語（日本語、英語など）を選ぶことができます。「編集」メニューではエディタのステージのサイズを小さくしたり、Scratch プログラムの画像描画の質を落とし、その代わりに計算の速度を向上させる（ターボモード）ことも可能です。

5.4.4 似ているプログラム

図 5.15(a) のプログラムは、原点からの距離を正しく計算するプログラムですが、ひとつの計算を行うプログラムは一種類とは限りません。また、一見すると正しいそうなプログラムが大きな間違いを犯していることもよくあります。そのような性質がプログラミングという作業を厄介で骨の折れるものにしています。

たとえば、図 5.15(a) のプログラムの上から 2 つ目と 3 つ目のブロックを入れ替えたプログラムが図 5.17(a) のプログラムです。同様に、上から 4 つ目と 5 つ目のブロックを入れ替えたプログラムが図 5.17(b) のプログラムです。どちらのプログラムもスペースキーを押すと猫が何か数値を教えてくれますから、プログラムの挙動だけからは、一見、正しく動いているようにも見えます。実際、(a) のプログラムは正しいプログラムです。しかし、(b) のプログラムは間違ったプログラムです。その違いを理解してください。

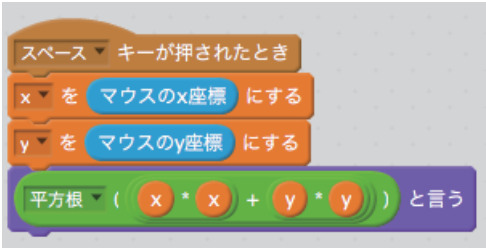
プログラムは上から下に向かって順に実行されていきます。プログラマは、その実行順序に沿って個々の動作が意図した通りに進んでいるか、正確に理解する必要があります。図 5.17(b) のように、間違った実行順序は重大な問題を引き起こします。

さらに別の Scratch プログラムでも図 5.15(a) と同等の計算は可能です。図 5.18 に 4 つの例を挙げます。いずれも猫は正しい結果を教えてくれます。図 5.18(a) では、距離の計算結果を変数 r に格納せず、直接、猫に言わせています。結果、ブロック数がひとつ減っています。図 5.18(b) では、二乗の計算のための代入のブロックを追加しています。結果、ブロック数は増えています。図 5.18(c) では、(b) のプログラムのブロックを一部、入れ替えています。この場合の入れ替えは距離の計算結果に影響を与えません。図 5.18(d) は変数を全く使用しないプログラムです。この場合、ブロック数は 2 個と激減し、ここまで見てきた距離を計算するプログラムの中では最少のブロック数です。ブロック数が減った代わりに、2 番目のブロックが異様に長くなっており、やや読みにくいプログラムに感じられます。

プログラムは、コンピュータへの指示を書き下した文章です。文章という意味では日本語や英語などの自然言語の文章と類似した性質を持ちます。ある意味を表す文章の書き方は唯一ではなく、表現上の幅があり、同一の文章でなくとも同じ意味を表すことが可能です。同じ意味を表す場合でも、どの表現が分かりやすいのか、どの表現が誤解を与えやすいか、という違いがあります。どのような表現方法が優れているのか、その基準はそう単純ではなく、一筋縄では行かない難しい問題です。

ところで、図 5.18(b) の上から 3 行目のブロック

では、代入する算術式 (x)*(x) に変数 x が使用されており、かつ、計算結果の代入先が同じ変数 x である点で注意が必要です。このブロックが実行される直前に変数 x に 8 が格納されていたと仮



(a) 変数 r を使用しない場合

(b) 算術式をより細かく分解した場合

(c) 左記 (b) のブロックを入れ替えた場合

(c) 変数を全く使用しない場合

図 5.18: 図 5.15(a) のプログラムの様々な変形

定しましょう。代入の実行では、まず式の値が計算されます。よって 8 が二乗されて、64 が求められます。そして、その 64 が変数 x に代入されます。結果として、この行の実行後には x には 64 が入っています。元の値 8 はどこにも残っていません。**上書き**（overwrite）されたのです。上書きされた値は決して復元できません。もし元の x の値 8 がプログラムの他の場所で必要とされないならば、上書きは何の問題も生じません。しかし他の場所で必要ならば、x を上書きすべきではなく、別の変数に代入すべきです。

　上書きはプログラムの読み易さを低下させます。上書きを行うと、変数に格納されている値が実行の途中で変化することになります。そうすると、プログラムのその部分を理解する上で、前後の文脈、順序関係を精密に理解する必要が生じ、プログラマの精神的なストレスが増えるからです。

ここの例でも、3 行目のブロックの実行前には x にはマウスの X 座標値が格納されていますが、実行後には二乗の値が入っており、格納されている値の解釈が変化しています。

かつてメモリが高価であった時代には、上書きは多用されました。メモリが安価になった現在は、上書きはできれば避けた方がよいでしょう。その意味で図 5.18(b) および (c) のプログラムは悪い（＝読みにくい）プログラムの例になっています。しかし、後に示すように、上書きを本質的に避けられないプログラムも存在します。

考察（ちょっと難しい） 以下のような動作をするプログラムを Scratch で作りなさい。

原点からの距離の計算プログラム（その 2）

マウスをクリックした時 のマウスの位置 (x, y) と座標原点 $(0, 0)$ との距離 $\sqrt{x^2 + y^2}$ を猫が教えてくれるプログラムを作りなさい。

ヒント：スプライト（猫）がクリックされたことを感知するブロックは、「イベント」のブロックパレットの中の「このスプライトがクリックされたとき」ブロックです（図 5.6 の右列の上から 3 つ目のブロック）。上のプログラム作成のポイントは、スプライト外の領域をマウスでクリックした場合の処理をどうプログラム化するかという点です。

5.5 条件判定と実行の分岐

前節では、ブロックが順に実行されていくこと、および変数を利用できることを説明しました。これはプログラムの実行の大原則ですが、これだけでは複雑な処理はできません。そこで次の原理が必要となります。

第二原理 プログラム中では、条件判定を行い、その判定結果の真偽に従い、次に実行する動作を取捨選択できる。

Scratch には（もちろん C++にも）上の原理を実装する手段が用意されています。

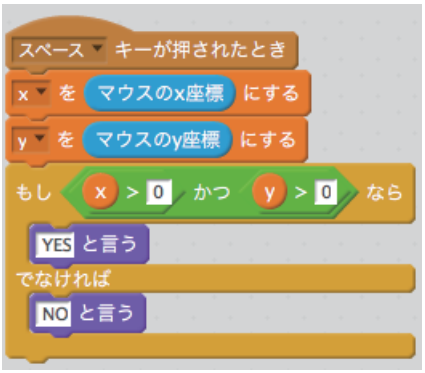
5.5.1 第一象限の判定

次のような動作をするプログラムを考えます。

第一象限を判定するプログラム

キーボードのスペースキーを押した時のマウスの位置 (x, y) について、その位置が x-y 座標系の第一象限ならば（$x > 0$ かつ $y > 0$ ならば）、猫が “YES” と言い、さもなくば “NO” というプログラムを作りなさい。x 軸、y 軸上については考慮する必要はない（どの象限としてもよい）。

このプログラムの一例が図 5.19(a) です。このプログラムでは、「制御」のパレットの「もし ⟨〜⟩ なら〜でなければ〜」ブロック（図 5.7 の左列の上から 5 つ目）を新たに使用しています。このブロックは、

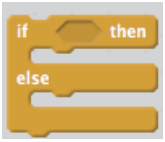


(a) プログラム

(b) 実行結果

図 5.19: 第一象限の判定

1. まず、条件判定式 $\langle\langle x > 0\rangle$ かつ $\langle y > 0\rangle\rangle$ の値を求めます。式の値を求めることを**評価**（evaluate）と呼びます。
2. もし判定式の値が**真**（true）であるならば、「YES と言う」ブロックを実行します。このとき、「NO と言う」ブロックは実行しません。
3. さもなくば（判定式の値が**偽**（false）であるならば）、「NO と言う」ブロックを実行します。「YES と言う」ブロックは実行しません。

条件判定の結果によって処理内容を取捨選択する動作を一般に**条件分岐**（conditional branch）と呼びます。多くのプログラミング言語では（もちろん C++でも）、条件式が真の場合に実行されるプログラム部分をしばしば**then 部**（then part）と呼び、偽の場合に実行されるプログラム部分をしばしば**else 部**（else part）と呼びます。というのも、多くのプログラミング言語では条件分岐のアイディアを、if、then、else の 3 つのキーワードで表すため、then 部、else 部という呼び方が広く認知されています。実際、Scratch の英語版でも「もし $\langle\sim\rangle$ なら〜でなければ〜」ブロックを 

と表現しており、if、then、else が用いられています（なお、C++では if、else はキーワードとして用いていますが、then は用いません）。

プログラムの実行において条件分岐は頻繁に行われています。条件分岐のないストレートな処理の方がまれです。たとえばロールプレイングゲームにおいて「HP が 0 になったらゲームオーバー。さもなくばゲームを継続」という条件分岐はゲームのプレイ中、常に実行されており、これ無しではゲームは成立しません。

Scratch で使用できる条件判定式はブロックパレット中の真偽値ブロック  に関係するものであり、それらは以下の 3 種類に分類できます。

- **組み込み述語** **（built-in predicate）** ... 実行時の様々な状態を真偽値で示すブロックで、もともと Scratch によって用意されたものです。たとえば「調べる」のブロックパレットの「⟨[ ] に触れた⟩」ブロックや「⟨ マウスが押された ⟩」ブロックなどです。
- **関係演算子** **（relational operator）** ... 数値の大小関係を判定するブロックです。「演算」のブロックパレットの中の「⟨[ ] < [ ]⟩」、「⟨[ ] = [ ]⟩」、「⟨[ ] > [ ]⟩」のブロックです。本格的なプログラミング言語（C++を含む）には「⟨[ ] ≤ [ ]⟩」や「⟨[ ] ≥ [ ]⟩」に相当する演算子も利用できますが、残念ながらこの二つは Scratch には定義されていません。同等のことを表現するには 2 つの論理演算子ブロックを組み合わせる必要があります。
- **論理演算子** **（logical operator）** ... いわゆる AND、OR、NOT を表す真偽値の上の演算子です。「演算」のブロックパレットの中の「⟨⟨ ⟩ かつ ⟨ ⟩⟩」、「⟨⟨ ⟩ または ⟨ ⟩⟩」、「⟨⟨ ⟩ ではない」のブロックです。

5.5.2 等価なプログラム

5.4.4 節において「似ているプログラム」を紹介しました。「似ている」ことを**等価**（equivalent）とも言います。より正確には、同じ入力に対して同じ出力を返す二つのプログラムを互いに等価であると言います。また別の言い方では「同じ意味を持つプログラム」とも言います。これ以降はこれらの言い方を用いることにします。

図 5.19(a) の条件分岐を含むプログラムについても等価なプログラムは多数存在します。

まず、論理式を 

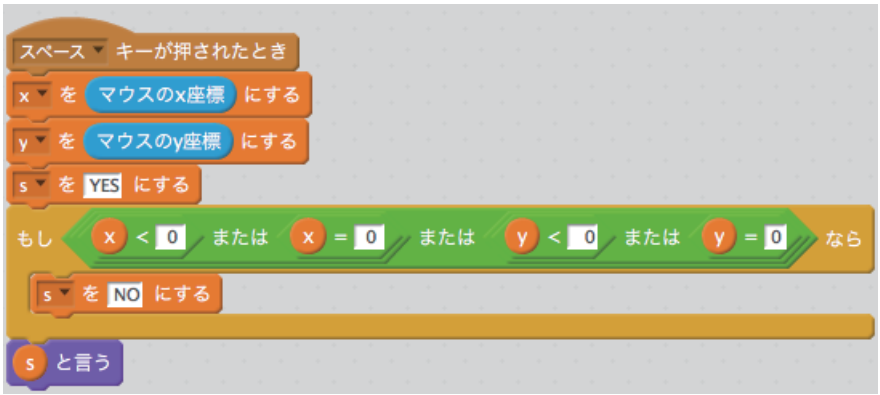
に置き換えても同じ真偽値と評価されるため、プログラムは等価です。言うまでもなく、論理式の置き換えには他にも様々なバリエーションがあります。

次に、図 5.20(a) は、猫に言わせる言葉を一旦、変数 s に代入するプログラムです。条件分岐の箇所では、真偽値に依って変数へ代入する言葉をそれぞれ “YES”、“NO” にします。そして、猫の発話はプログラムの制御が合流した後に行います。これはやはり図 5.19(a) と等価です。

図 5.20(b) は、図 5.19(a) とは異なるプログラム構造になっています。図 5.19(a) では論理積「⟨⟨ $x > 0$⟩ かつ ⟨ $y > 0$⟩⟩」を用いて第一象限の条件を表しました。しかし、図 5.20(b) ではまず「⟨ $x > 0$⟩」を判定し、それが真ならば次に「⟨ $y > 0$⟩」を判定し、それも真ならば第一象限であると判断しています。このプログラムでは条件分岐の中に条件分岐が含まれる**入れ子**（nest）構造になっていることに注意してください。複雑な条件を判定するときには入れ子構造を用いた方がプログラムが簡潔に表現できる場合があります。

(a) 変数 s へのデータの格納

(b) 条件分岐の入れ子



(c) 「もし ⟨〜⟩ なら」ブロックの利用

図 5.20: 図 5.19(a) のプログラムの様々な変形

Scratch には「もし ⟨〜⟩ なら〜でなければ〜」ブロックの「でなければ〜」の部分がないブロック — 「もし ⟨〜⟩ なら」ブロック — も用意されています。このブロックでは、条件が成り立つ場合にはそれが内蔵する処理を行いますが、成り立たない場合には何も実行しません。図 5.20(c) は、敢えてこのブロックを用いてプログラムを作成してみた例です。変数 s にあらかじめ “YES” を代入しておき、マウスの位置が第一象限でないならば s の値を “NO” で上書き変更します。このプログラムは図 5.19(a) のプログラムに比べて読みにくいプログラムかもしれません。こういうプログラミングを推奨する訳ではありませんが、等価なプログラムの一例として紹介しました。

考察　次のような動作をするプログラムを Scratch で作りなさい。

象限を判定するプログラム

キーボードのスペースキーを押した時のマウスの位置 (x, y) について、猫がその位置の象限（第 1〜第 4 象限）を “1”、“2”、“3”、“4” のいずれかで教えるプログラムを作りなさい。x 軸、y 軸上については考慮する必要はない（どの象限としてもよい）。

なお、第 1〜4 象限は原点の回りに反時計周りに定義されています [14]。

5.6 繰り返し実行とループ

コンピュータの実行に関する第三の原理は以下の通りです。

第三原理 プログラム中に繰り返し実行部を作ることで、多数回の類似の処理を簡素に表現できる。

以下、繰り返し処理の例を描画プログラムで紹介します。

5.6.1 ペンによる多角形の描画

Scratch の「ペン」のブロックパレット（図 5.6 左列参照）には線画を描くための機能が充実しています。ステージ上のスプライトの中心にペンが付いていますと考えてください。ただし、そのペンは見えません。ペン先はステージから離れている（上がっている）、またはステージに付いている（下ろされている）のいずれかの状態を取ります。そのペン先をステージに下ろす動作が「ペンを下ろす」ブロックです。逆は「ペンを上げる」ブロックです。ペン先が下りた状態でスプライトがステージ上を移動すると、スプライトの軌跡がステージ上に描かれます。ペン先が上がった状態では軌跡は描かれません。

手始めに正方形を描くプログラムを考えます。

正方形を描画するプログラム

緑の旗をクリックすると猫が 1 辺の長さが 100 の正方形を描くプログラムを作りなさい。

図 5.21 がその描画プログラムと実行結果です。このプログラムでは、緑の旗がクリックされたならば、まず猫に付随するペンのペン先を下げ、次に猫が 100 歩進み、進行方向を時計回りに 90 度を変える動作を 4 回繰り返し、最後にペンを上げて実行終了です。

次に正 6 角形を描画しましょう。

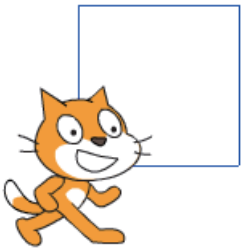
正 6 角形を描画するプログラム

緑の旗をクリックすると猫が 1 辺の長さが 100 の正 6 角形を描くプログラムを作りなさい。

図 5.21 と同じ発想でプログラミングしたものが図 5.22(a) です。100 歩進み、方向を反時計回りに

[14] $x > 0 \wedge y > 0$ が第一象限、$x < 0 \wedge y > 0$ が第二象限、以下同様。

(a) プログラム



(b) 実行結果

図 5.21: 正方形の描画

60 度方向を変えることを 6 回繰り返しています。しかし、多数の類似するブロックを張り合わせて作ったプログラムは決して読みやすいものではありません。

これを解決するために繰り返し処理を用います。

Scratch の「制御」のブロックパレットには「( ) 回繰り返す」ブロックが用意されています。それを用いたプログラムが図 5.22(b) です。(a) のプログラムに比べてブロック数が半減しており、明らかに読みやすいプログラムになっています。

このプログラムのもうひとつの優れた点は他の正多角形の描画プログラムへ直ちに応用できることです。正 n 角形を描きたいときには、図 5.22(b) の繰り返し回数を n 回とし、回転角度を $360/n$ 度に設定すればうまくいきます。ただし、猫の歩数が 100 のままでは多角形が大きくなり過ぎるかもしれません。猫の動く歩数 ℓ と多角形が内接する円の半径 r は

$$\ell = 2r\sin(180/n)$$

という関係が成り立ちます（高校数学で証明できます）から、ステージ（$-240 \leq x \leq 240$、$-180 \leq y \leq 180$）にちょうど収まるような多角形として $r = 100$ としたいならば、$\ell = 200\sin(180/n)$ と設定します。もし正 360 角形を描きたいならば、繰り返し回数は 360 回、回転角度は 1 度、歩数は $\ell = 200\sin(180/360) \approx 1.745$ です。

翻って、図 5.22(a) のように繰り返しを用いずに正 360 角形を描画するならば、723 個のブロックを張り合わせる必要があります。もはや人手でのプログラミングは不可能です。しかし図 5.22(b) のプログラムを用いるならば、プログラムの大きさは n に依らず一定です。大量の計算を行うには繰り返し処理は不可欠なのです。

考察　次のような動作をするプログラムを Scratch で作りなさい。

(a) ブロックを単純に積み重ねたプログラム

(b) 繰り返しブロックを用いたプログラム

図 5.22: 正 6 角形の描画

星形図形の描画プログラム

緑の旗をクリックすると猫が図 5.23(a)、(b) のような星型図形を描画するプログラムを作りなさい。

なお、正 m 角形を想定し、その頂点を $n-1$ 個飛ばしで結ぶときにできる星型を正 m/n 角形と呼びます。たとえば 5 芒星は、正 5 角形を想定し、その頂点を 1 個飛ばしに結ぶと描画できる図形であるため、これを正 5/2 角形と呼ぶことができます。

5.6.2 内接多角形

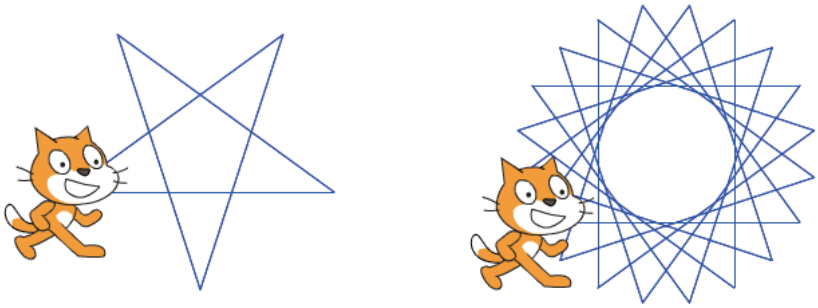
正多角形の描画方法には、円に内接する多角形を用いて描画する方法もあります。

内接正多角形を描画するプログラム

緑の旗をクリックすると猫が半径 r の円に内接する正 n 角形を描画するようなプログラムを作りなさい。

今、原点を中心とした半径が r の円を考えます。その円周上の n 個の均等に配置された点の座標値 (x_i, y_i) は正 n 角形の頂点ですが、たとえばそれは以下の式で表すことができます。

$$\begin{aligned} (x_i, y_i) &= (r\cos\theta_i,\ r\sin\theta_i)\,, \\ \text{ただし}\quad \theta_i &= \left(\frac{360}{n}\right) i \end{aligned} \tag{5.1}$$



(a) 正 5/2 角形（5 芒星）　(b) 正 20/7 角形

図 5.23: 星形図形（正 m/n 角形）

なお、(x_i, y_i) と (x_{i+n}, y_{i+n}) は任意の i について同一点であることに注意してください。

多角形を描くには点列 $(x_0, y_0) \to (x_1, y_1) \to (x_2, y_2) \to \ldots \to (x_{n-1}, y_{n-1}) \to (x_n, y_n) = (x_0, y_0)$ を順に線分で結べばよく、図 5.24 はそのプログラムと実行結果です。この図では $r = 100$、$n = 11$ としました。このプログラムでは、sin、cos を計算するブロックおよび「x 座標を ( )、y 座標を ( ) にする」ブロックを用いました。また 6 個の変数 r、n、i、theta（$= \theta$）、x、y を用いました。

このプログラムは以下のブロック構造を有する点が、図 5.22 のプログラムと異なります。

この構造では繰り返し処理が n 回実行されますが、繰り返し処理の度毎に変数 i の値が、1, 2, ..., n と増えていきます。その i の値を参照することで、繰り返し毎に異なる処理を実現しているのです。

考察　次のような動作をするプログラムを Scratch で作りなさい。

星形図形の描画プログラム

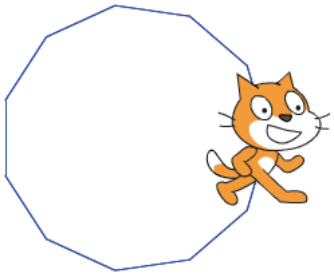
緑の旗をクリックすると猫が図 5.23(a)、(b) のような星型図形を描画するプログラムを、図 5.24(a) のプログラムを改造して作りなさい。

5.6.3　渦巻き

内接多角形の発展形として渦巻きを作ることができます。

渦巻きは、ペンが原点の周りを回るときにその回転半径が変化して行く図形です。半径が回転に比例して増えるならば、式 (5.1) を以下のように修正すればよいでしょう。ここに α が比例定数

(a) 半径 100 の正 11 角形の描画プログラム



(b) 実行結果

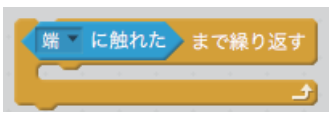
図 5.24: 内接多角形の描画

です。

$$
\begin{aligned}
(x_i, y_i) &= (r_i \cos\theta_i,\ r_i \sin\theta_i)\,, \qquad (5.2)\\
\text{ただし}\quad r_i &= \alpha \cdot i,\\
\theta_i &= \left(\frac{360}{n}\right) i
\end{aligned}
$$

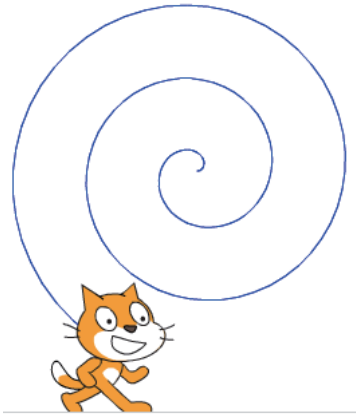
この式に基づく Scratch のプログラムと実行結果は図 5.25 の通りです。ただし、$\alpha = 1.1$ としました。また、$n = 50$ と大きな数に設定したため、多角形の角はほとんど目立たず、滑らかな曲線に見えます。

　多角形の場合には 1 回転分だけペンを動かせば十分でしたが、渦巻きの場合には原点の回りに何周分もペンを動かす必要があります。そこで、可能な限り広い範囲に描画することを意図し、図 5.25 のプログラムでは以下のブロックを用いました。



この制御ブロックは、処理を固定回数だけ繰り返すのではなく、スプライトが ⟨[端] に触れた⟩ という条件が成り立つまで繰り返すブロックです。条件 ⟨...⟩ の部分は他の真偽値ブロックに置き換えることができますから、繰り返しの終了を柔軟に制御できるようになっています。

(a) 渦巻きの描画プログラム



(b) 実行結果

図 5.25: 渦巻きのプログラム

ほとんど全てのプログラミング言語では（もちろん C++でも）、固定回数繰り返す機能とある条件が成り立つまで繰り返す機能（あるいはある条件が成り立っている間、繰り返す機能）が装備されており、プログラムの内容に応じて使い分けられています。

さらに改造を加えます。複数の渦巻きを重ねて多重渦巻きを作りましょう。

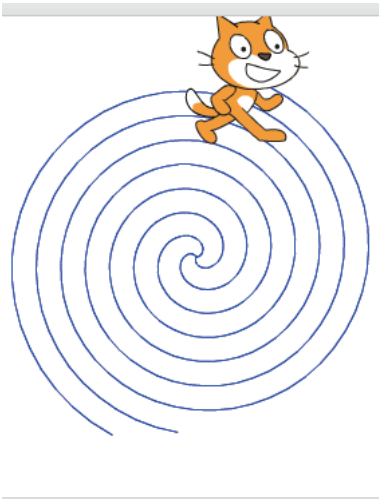
多重度を m とするとき、式 (5.2) に $360/m$ 度毎に異なる位相 ϕ_j（$j = 0, 1, ..., m-1$）を追加します。

$$
\begin{aligned}
(x_{i,j}, y_{i,j}) &= (r_i \cos(\theta_i + \phi_j),\ r_i \sin(\theta_i + \phi_j)\,, \qquad (5.3)\\
\text{ただし}\quad r_i &= \alpha \cdot i,\\
\theta_i &= \left(\frac{360}{n}\right) i,\\
\phi_j &= \left(\frac{360}{m}\right) j
\end{aligned}
$$

この式に基づく Scratch のプログラムは図 5.26 の通りです。ただし、$m = 3$ とし、3 重化しました。繰り返し処理が **入れ子** になっていることに注意してください。

考察　次のような動作をするプログラムを Scratch で作りなさい。

(a) 3 重渦巻きの描画プログラム



(b) 実行結果

図 5.26: 3 重渦巻きのプログラム

螺旋（らせん）の描画プログラム

緑の旗をクリックすると猫が図 5.27 のような 2 重螺旋を描画するプログラムを作りなさい。

なお、螺旋はその中心位置が少しずつ移動していく円と考えることができます。図 5.27 の場合、螺旋は x 軸方向へ移動していますから、以下のような式で記述できます。ここに β は定数です。

$$
\begin{aligned}
(x_{i,j}, y_{i,j}) &= (r_i \cos(\theta_i) + \beta \cdot i,\ r_i \sin(\theta_i)), \qquad (5.4)\\
\text{ただし}\quad r_i &= \alpha \cdot i,\\
\theta_i &= \left(\frac{360}{n}\right) i
\end{aligned}
$$

1 2 重螺旋を描くにはさらに位相の異なる螺旋を重ねる必要があります。

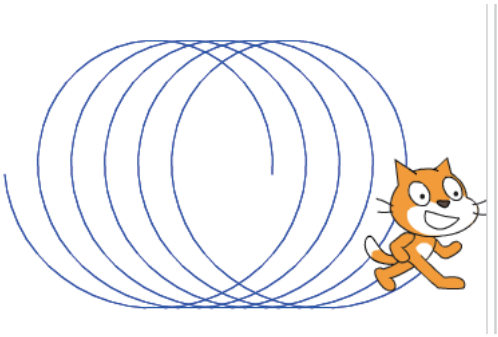


図 5.27: 2 重螺旋

5.6.4 ループ

ここまで様々な例で見てきたように、繰り返し処理ではプログラム中の同じ位置で類似の動作を多数回繰り返します。その様子はプログラムの実行が一定の範囲をクルクルと回っているように理解できることから、繰り返し処理のプログラム部分のことを簡単に **ループ**（loop）とも呼びます。プログラム中に繰り返し部があることを「ループがある」、繰り返し実行している様子を「ループしている」、繰り返し処理が終了し、直下の処理へ実行が移ることを「ループから脱出する」「ループから抜け出す」などと言います。

図 5.26(a) のプログラムでは繰り返し処理の中に繰り返し処理が含まれています。いわばループの中にループがあることから、このような入れ子構造を **2 重ループ**（doubly nested loop, double loop）と呼びます。

条件分岐の入れ子や繰り返し処理の入れ子構造は、少し複雑な処理にはしばしば必要になるものです。

5.6.5 補足：渦巻きと螺旋

図 5.25(b)、5.26(b) は渦巻きです。図 5.27 は螺旋（らせん）です。しかし、一般に「渦巻き」と「螺旋」（spiral と helix）はしばしば混乱して用いられています。たとえば図 5.25(b) の渦巻きはアルキメデスの螺旋と呼ばれています。「渦巻き」と「螺旋」という名称の付いた図形については、それが何を表しているのか、しっかり確認する必要があります。

考察 対数螺旋（logarithmic spiral）について調べ、次のような動作をするプログラムを Scratch で作りなさい。

対数螺旋の描画プログラム

緑の旗をクリックすると猫が対数螺旋を描画するプログラムを作りなさい。

第6章　Scratchプログラミング（応用編）

プログラミングの三つの原理（逐次実行、条件判定と分岐、繰り返し）を説明するために、前章では簡単なプログラムの例を示してきました。この章ではプログラミングの奥深さを感じさせる例題を取り上げます。

6.1　平方根の計算

まず最初の例題は平方根の計算です。

> 平方根の計算
>
> 緑の旗をクリックすると猫が $\sqrt{2}$ の値を教えてくれるプログラムを作りなさい。ただし、「演算」のパレットの関数ブロック（「[平方根]( )」、「[ln]、「[^]( )」など）を用いないこと。

上の「但し書」は重要です。Scratch の「[平方根]( )」ブロックを使えば $\sqrt{2}$ の計算は簡単に終わります。しかしここでは私たち自身が「[平方根]( )」ブロックをプログラミングするつもりで、平方根のプログラムを四則演算の組み合わせで計算することに挑戦します。

平方根には様々な計算方法が知られています。筆算では開平法と呼ばれる計算法が知られていますが、コンピュータでは開平法は用いません。コンピュータは人手に比べ計算が圧倒的に高速なため、その特徴を活かした力技で計算する方が有利なのです。開平法のような、計算回数は少ないものの計算手順が複雑な方法は人手での計算では有効ですが、コンピュータには適しません。

ここでは三つの計算方法を紹介します。

6.1.1　二分法

図 6.1 の数直線を用いて説明します。

$\sqrt{2}$ が区間 [1, 2] にあることは明らかです。$\sqrt{2}$ がこの区間中のどの辺りにあるかをより詳しく知るために、この区間の中間点 1.5 $(= (1+2)/2)$ についてその二乗を計算してみます。そうすると、$1.5^2 = 2.25 > 2$ ですから、平方根の単調性（$x > y$ ならば $\sqrt{x} > \sqrt{y}$）により $1.5 > \sqrt{2}$ が成り立ちます。よって区間 [1, 2] を前半部 [1, 1.5] と後半部 [1.5, 2] の二つの区間に分けるならば、求める平方根は区間 [1, 1.5] に含まれます。次に区間 [1, 1.5] の中間点 1.25 $(= (1+1.5)/2)$ についてその二乗を計算します。$1.25^2 = 1.5625 < 2$ ですから $1.25 < \sqrt{2}$ が成り立ち、$\sqrt{2}$ は区間 [1.25, 1.5] にあることが分かります。そこで、次に区間 [1.25, 1.5] の中間点 1.375 $(= (1.25+1.5)/2)$ に

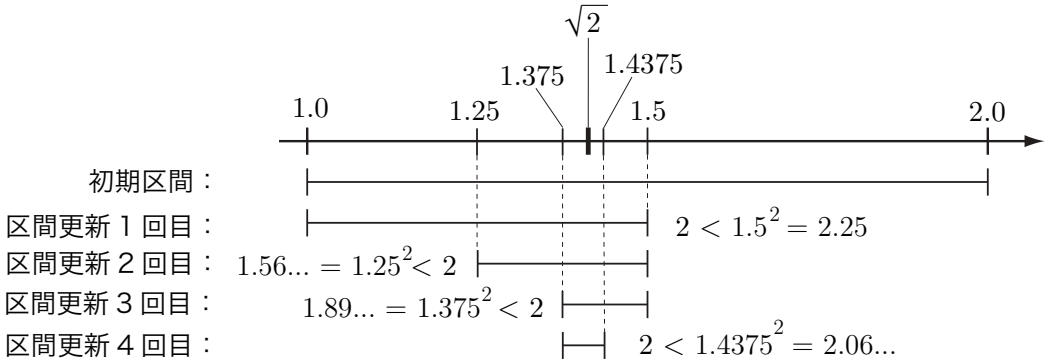


図 6.1: 二分法によって $\sqrt{2}$ を含む区間が狭められていく過程

ついてその二乗を計算し... と続けて行けば、$\sqrt{2}$ が存在する区間の幅は 1 回の計算で **半分** になり、求める値 $\sqrt{2} = 1.41421356...$ の含まれる区間の幅を限りなく狭めていくことができます。

この方法を手順として述べると以下の通りです。

1. 初期の区間を定め、
2. 区間の中間値と中間値の二乗を求め、
3. 区間を更新します。
4. 上の 1. から 4. を一定回数だけ繰り返し、最終的に得られた区間の中間値を $\sqrt{2}$ の近似値とみなします。

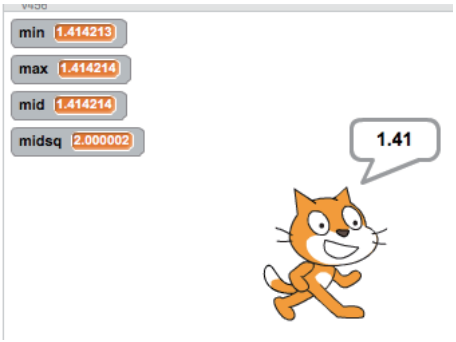
上の方法を Scratch でプログラミングするために、手順をさらに詳細に設計します。

1. 二つの変数 `min` と `max` を用いて区間 [`min`, `max`] を表すと約束します。初期値は `min`=1、`max`=2 とします。
2. 区間の中間値と中間値の二乗を計算し、変数 `mid` と `midsq` に格納します。
3. もし `midsq` の値が 2 よりも小さいならば、中間値 `mid` を新たな `min` とします。さもなくば `mid` を新たな `max` とします [1]。
4. 上の処理を 20 回繰り返し、最終的に得られた

プログラムは図 6.2(a) の通りです。前章で述べたように、プログラムの作り方には大きな自由度がありますから、図 6.2(a) が二分法の唯一のプログラムではありません。様々なプログラムの書き方がありえます。なお、プログラムの実行途中の様子を知るために、図 6.2(a) のプログラムでは

[1] 平方根 $\sqrt{2}$ は無限小数なので、`midsq` が 2 に等しくなることはありえません。よって「`midsq` が 2 に等しいときには...」という判定は必要ありません。

(a) プログラム



(b) 実行結果

図 6.2: 二分法による $\sqrt{2}$ の計算プログラム

ループ中に「[mid] と (0.5) 秒言う」ブロックを追加しています。このブロックは猫が mid の値を 0.5 秒間だけ表示します。表示中はプログラムの実行は一時停止状態になります。

図 6.2(b) の猫が教えてくれる $\sqrt{2}$ の値は 3 桁しかありません。より詳細な $\sqrt{2}$ の値を知るには図 6.2(b) 左上の変数の箱の中を見た方がよいでしょう。以下は 0.5 秒毎に変わる mid の箱の中の数値を順にリストアップしたものです。

```
1.5
1.25
1.375
1.4375
1.40625
1.421875
1.414062
1.417968
1.416015
1.415039
1.414550
1.414306
1.414184
1.414245
1.414215
1.414199
1.414207
1.414211
1.414213
```

```
#include <iostream>
#include <cmath>
#include <unistd.h>
using namespace std;

int main(){
    double min = 1.0, max = 2.0, mid;

    for(int i = 0; i < 20; i++){
        mid = (min+max)/2.0;
        cout << mid << endl; sleep(1);
        double midsq = mid*mid;
        if(midsq < 2.0){
            min = mid;
        }else{
            max = mid;
        }
    }
    mid = (min+max)/2.0;
    cout << mid << endl;
}
```

図 6.3: C++を用いた二分法による $\sqrt{2}$ の計算プログラム（試験範囲外）

```
1.414214
```

中間値 mid の値が真値 $\sqrt{2} = 1.41421356...$ に近付いている様子が分かります。二分法では繰り返し毎に区間が 1/2 に狭まります。よって n 回繰り返すことで中間値と真値の誤差は $(1/2)^n$ 以下になります。

ところで、同じ計算を「プログラミング概論/演習 1,2」で学ぶ C++でプログラミングした場合、そのプログラムは図 6.3 のようになります[2]。多くのプログラミング言語では（C++でも）プログラムは英文字、数字、記号の並びでできています。プログラムを図的に表す Scratch がむしろ特殊です。図 6.3 ではキーワード `for` が繰り返しを表し、`if else` が条件分岐を表します。このプログラムを図 6.2(a) を見比べてください。何となく同じように読めるのではないでしょうか。

なお、図 6.3 は今後の学習のための参考であって、この科目の成績評価（試験、レポート）とは無関係です。

考察　図 6.2(a) のプログラムを改造し、次のような動作をするプログラムを作りなさい。

任意の数の平方根の計算プログラム

変数 x (> 1) にあらかじめ任意の正数を代入しておくとする。緑の旗をクリックすると猫が x の平方根を教えてくれるプログラムを作りなさい。x< 1 の場合についても考察しなさい。

[2]ただし、C++の仕様の都合により、計算途中の一時停止時間は 0.5 秒ではなく、1 秒としました。

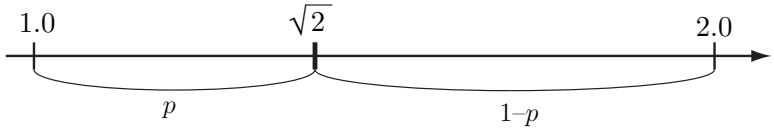


図 6.4: 区間 $[1,2]$ の中のランダムな数 x が $x^2 < 2$ となる確率 p

考察　図 6.2(a) のプログラムを改造し、次のような動作をするプログラムを作りなさい。

任意の数の立方根の計算プログラム

変数 x (> 1) にあらかじめ任意の正数を代入しておくとする。緑の旗をクリックすると猫が x の立方根を教えてくれるプログラムを作りなさい。

考察　「調べる」のブロックパレットの中の 

を実行すると、猫が「What's your name?」と質問すると同時にステージ下方に入力フィールドが現れます。そのフィールドに数値や文字列を入力すると、その値が 

に自動的に代入されます。図 6.2(a) のプログラムを改造し、この二つのブロックを使って次のような動作をするプログラムを作りなさい。

任意の数の立方根の計算プログラム

緑の旗をクリックすると猫が「正数値を入力してください」と入力を催促する。そこでユーザがキーボードから数値を入力すると、猫が入力した値の平方根（または立方根）を教えてくれる、そのようなプログラムを作りなさい。

6.1.2　モンテカルロ法

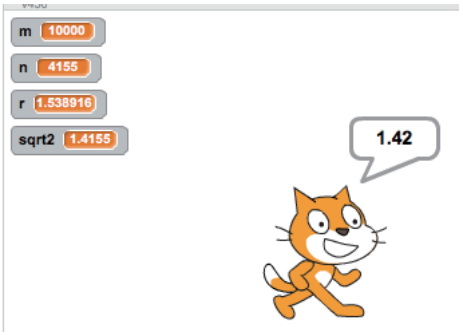
モンテカルロはカジノで有名な街ですが、カジノでの必勝法の研究からモンテカルロ法と呼ばれている興味深い計算手法が考え出されました。

今、区間 [1,2] の中の適当な実数 x をひとつランダム（random）に選びます。たとえば 1.2498、1.342、1.9834553、.... などの任意の数です。ランダムに選ぶ数が区間から一様な確率で選ばれるとき、そのような数を **一様乱数**（uniform random number）と呼びます。より正確に言えば、それを形作る数列についてその区間内の全ての実数が無秩序に法則性なく同じ確率で現れる [3] ような数のことです。コンピュータはプログラム通りにしか動きませんから、コンピュータで無秩序な数を生成できるはずがないのですが、**あたかも無秩序に見えるような数列をプログラムで擬似的に生成する**という手法を用いるのが通例です。これを擬似乱数と呼び、詳細は述べませんがすでに様々な計算方法が知られています。

次に、その数の二乗 x^2 を計算します。このとき、x^2 の値が 2 よりも小さい確率は $p = \sqrt{2} - 1$ です。何故ならば、二乗しても 2 以下の数は区間 $[1, \sqrt{2}]$ に含まれ、その幅 $\sqrt{2} - 1$ は全区間 $[1, 2]$ の幅 $2 - 1$ に対して、$\left(\sqrt{2} - 1\right) / (2 - 1)$ の比率になるからです（図 6.4 参照）。

[3] 説明に当たり、Wikipedia「乱数列」を参考にしました。

(a) プログラム



(b) 実行結果

図 6.5: モンテカルロ法による $\sqrt{2}$ の計算プログラム

さて、$p = \sqrt{2} - 1$ を変形すると $\sqrt{2} = p + 1$ を得ます。つまり、x^2 の値が 2 よりも小さい確率 p が分かれば $\sqrt{2}$ を知ることができるのです。このことに注目し、実際に区間 [1,2] の中から M 個の数をランダムに求め、その二乗が 2 よりも小さかった数の個数 N を求めます。そうすると確率 p は N/M です。そして $\sqrt{2} = p + 1 = N/M + 1$ と平方根が計算できます。

ランダムな数を用いて **確率的** な計算を行う方法を一般にモンテカルロ法と呼んでいます。この方法は正確な数値を高速に求めることには不向きですが、考え方が非常に単純かつ素朴であるため、効果的な計算方法が見つかっていない複雑な数値の計算や様々な事象のシミュレーションに有効です。

この方法の Scratch によるプログラムと実行結果は図 6.5 の通りです。なお、このプログラムでは「(1.0) から (2.0) までの乱数」ブロックを用いて、一様乱数を生成しています[4]。図 6.5(a) のプログラムでは 1 万回の繰り返し実行を行っています。結果、1.4155 という値が計算されています（猫は、1.4155 を 3 桁に四捨五入した値 1.42 を表示します）。なお、乱数を用いる実行のため、得られる数値は実行の度に異なるかもしれないことを注意しておきます。

さて、1 万回繰り返したにも関わらず、真値 $\sqrt{2} = 1.41421356...$ から大きな誤差があります。参考のために、同じプログラムを繰り返し回数を変えて実行した結果は以下の通りです。

```
1.415500   ... 1 万回の場合
1.417480   ... 10 万回の場合
1.413691   ... 100 万回の場合
```

[4] ブロック中の数値を整数で入力し、「(1) から (2) までの乱数」ブロックを用いると、整数値 1 または 2 をランダムに生成します。これを用いると、正しい計算はできません。ところで、Scratch のエディタでハードディスクにダウンロードしたプログラムをアップロードすると、小数点以下が 0 の数（たとえば 1.0）は整数に置換されるという問題があります。注意してください。

```
#include <iostream>
#include <stdlib.h>
using namespace std;

int main(){
    int m = 10000, n = 0;
    for(int i = 0; i < m; i++){
        double r = double(random()%1000000)/1e6+1.0;
        if(r*r < 2.0){
            n++;
        }
    }
    double sqrt2 = double(n)/double(m)+1.0;
    cout << sqrt2 << endl;
}
```

図 6.6: C++を用いたモンテカルロ法による $\sqrt{2}$ の計算プログラム（試験範囲外）

```
1.414437  ... 1000 万回の場合
```

モンテカルロ法では繰り返し回数 n に対して誤差が $1/\sqrt{n}$ になることが知られており、繰り返し回数を 100 倍しても誤差は $1/\sqrt{100} = 1/10$ にしかなりません。二分法では 1 回の繰り返しで誤差が 1/2 になり、たかだか 20 回で 6 桁以上の精度 [5] を得られましたが、それと比較すると明らかに非効率です。そのため、確率的な手法であるモンテカルロ法の応用範囲は限られています。しかし効果的な計算方法が見つかっていない難問を解くためにはしばしば同様の確率的な計算方法が用いられます。

参考のため、図 6.5(a) に対応する C++のプログラムを図 6.6 に載せておきます [6]。

考察 図 6.5(a) のプログラムを改造し、次のような動作をするプログラムを作りなさい。

任意の数の立方根の計算プログラム

変数 x (> 1) にあらかじめ任意の正数を代入しておくとする。緑の旗をクリックすると猫が x の立方根を教えてくれるプログラムを作りなさい。

6.1.3 ニュートン・ラフソン法

二分法はモンテカルロ法よりも少ない繰り返し回数で高い精度が得られる計算方法でした。しかし、ここで紹介するニュートン・ラフソン法（あるいは単にニュートン法）は、二分法よりもさら

[5] $(1/2)^{20} < 1/1048576 < 0.000001$

[6] 図 6.6 のプログラムは手抜きをしており、区間 [1,2] の乱数を高々100 万分の 1 の精度までに限定しているのでそれ以上の精度の計算はできません。

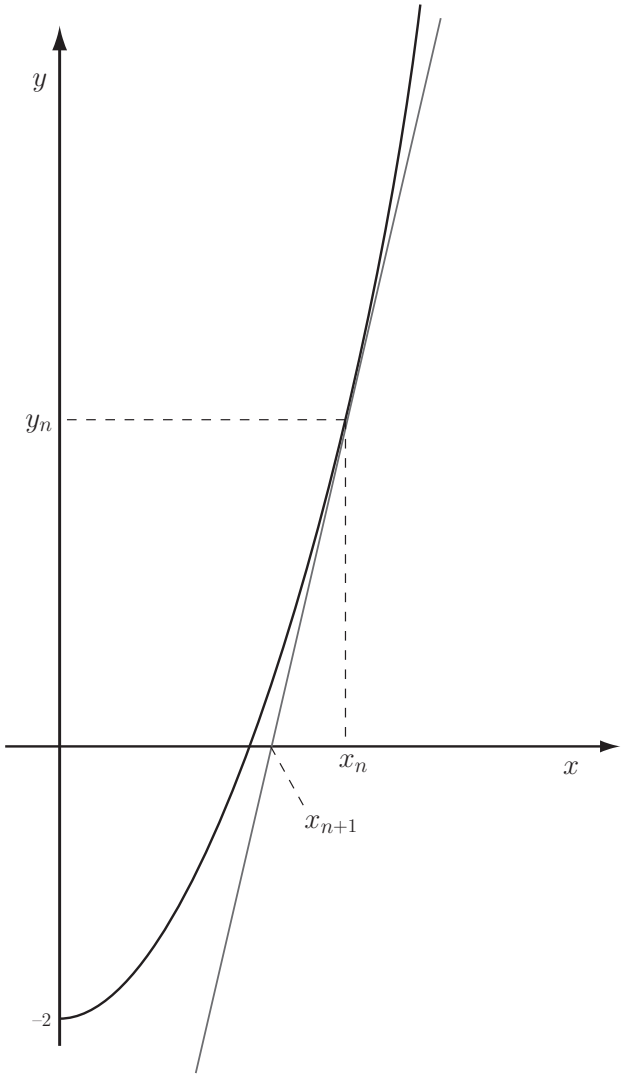


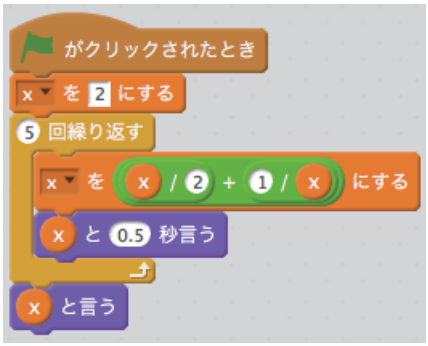
図 6.7: ニュートン・ラフソン法の 1 ステップ

に高速に平方根を求めることのできる方法です。現在、この方法は平方根を計算する最も実用的な方法として知られ、実際に広く使用されています。

ニュートン・ラフソン法では、二次方程式 $y = x^2 - 2$ の $y = 0$ のときの解（つまり $x = \sqrt{2}$）を求めることを考えます。図 6.7 は $y = x^2 - 2$ のグラフです。グラフと x 軸の交点座標は $(\sqrt{2}, 0)$ ですから、この交点を計算で求めることができれば $\sqrt{2}$ の値を知ることができます。

今、x 軸上の点 $(x_n, 0)$ を任意にひとつ与えます。ただし、$\sqrt{2} < x_n$ となる点、たとえば $x_n = 2$ とします。この点から垂直に直線を伸ばし、グラフ $y = x^2 - 2$ と交わる点 (x_n, y_n) を求めます。次に、(x_n, y_n) を通り、グラフ $y = x^2 - 2$ と接するような直線を求めます。この直線を $y = ax + b$ と表わすとき、$a = 2x_n$、$b = -x_n^2 - 2$ が成り立つことが容易に証明できます [7]。次に、この直線

[7] $y = ax + b$ は点 (x_n, y_n) を通過し、かつ $y = ax + b$ の傾き a は $x = x_n$ において $dy/dx = d(x^2 - 2)/dx = 2x$ に等しく、かつ $y_n = x_n^2 - 2$ であるという条件から a と b を具体的に求めることができます。



(a) プログラム

(b) 実行結果

図 6.8: ニュートン・ラフソン法による $\sqrt{2}$ の計算プログラム

$y = ax + b$ が x 軸と交わる点を $(x_{n+1}, 0)$ とするとき、$x_{n+1} = -b/a$ が成り立ちます。よって、$x_{n+1} = x_n/2 + 1/x_n$ という漸化式を得ます。$y = x^2 - 2$ のグラフの性質（図 6.7 参照）から、明らかに $\sqrt{2} < x_{n+1} < x_n$ が成り立ちます。よって、この計算を繰り返すことで、

$$\sqrt{2} < \ldots < x_{n+3} < x_{n+2} < x_{n+1} < x_n$$

と、漸化式の値は限りなく $\sqrt{2}$ に近づいていきます。

まとめます。漸化式：

$$\begin{aligned} x_0 &= 2, \\ x_{n+1} &= \frac{x_n}{2} + \frac{1}{x_n} \quad (n > 0) \end{aligned}$$

を用いて、数列の値を x_0, x_1, x_2, x_3, ... と計算していくならば、その値は $\sqrt{2}$ に近づいていきます。なお、初期値 x_0 の値は $\sqrt{2}$ よりも大きい任意の数値で構いません。そのような数として 2 を与えるのが単純です。

この方法による Scratch のプログラムは図 6.8 の通りです。繰り返しによる変数 x の値の変化は以下の通りになります。

```
1.5
1.416666
1.414215
1.414213
1.414213
```

4 回の繰り返しで 6 桁の計算精度を得ており、二分法よりも高速です。実際、コンピュータにおける平方根の計算はニュートン・ラフソン法が定番です。

後期の授業の参考までに、C++を用いて記述したニュートン・ラフソン法のプログラムを図 6.9 に載せます。

考察　図 6.8(a) のプログラムを改造し、次のような動作をするプログラムを作りなさい。

```
#include <iostream>
#include <unistd.h>
using namespace std;

int main(){
    double x = 2.0;
    for(int i = 0; i < 5; i++){
        x = x/2.0+1.0/x;
        cout << x << endl; sleep(1);
    }
    cout << x << endl;
}
```

図 6.9: C++を用いたニュートン・ラフソン法による $\sqrt{2}$ の計算プログラム（試験範囲外）

任意の数の立方根の計算プログラム

変数 x (> 1) にあらかじめ任意の正数を代入しておくとする。緑の旗をクリックすると猫が x の立方根を教えてくれるプログラムを作りなさい。

この節では平方根の 3 種類の計算方法を紹介しました。ひとつの問題が与えられたとき、それを解く方法はひとつとは限りません。プログラマは様々な方法の特徴を理解し、それらを状況に応じて使い分けねばなりません。

6.2 円周率の計算

二つ目の例題として円周率の計算方法を紹介します。

円周率とは、任意の円の **円周** の長さをその円の **直径** の長さで割った数です。

円周率の計算

緑の旗をクリックすると猫が円周率の値を教えてくれるプログラムを作りなさい。

平方根同様に様々な計算方法が知られています。ここでは代表的な方法を概説します。

6.2.1 正多角形に関する漸化式

まず、古典的な計算方法として円に内接する正 n 角形を考えましょう。正 n 角形の外周の長さを円の直径で割った数は、n が大きくなるに従い、円周率に近づいて行きます。

正 n 角形の外周の長さ L_n が分かっているならば、正 $2n$ 角形の外周の長さ L_{2n} は L_n から比較的容易に計算できることが知られています。まず、この関係式を導いてみましょう。

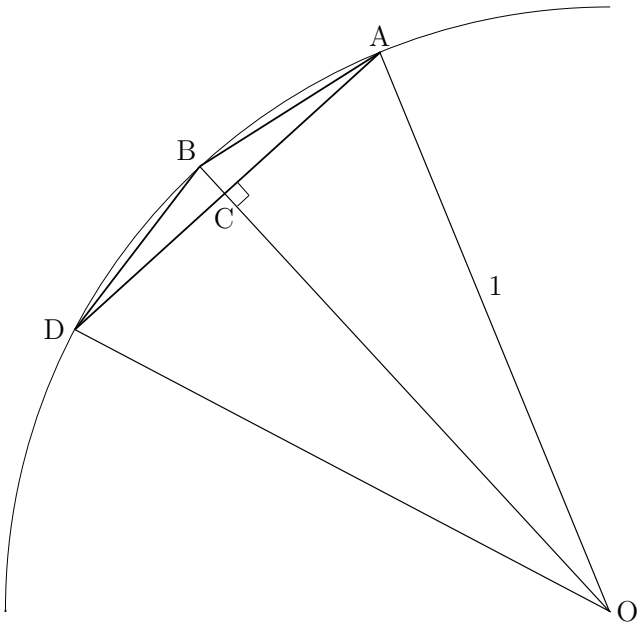


図 6.10: 正 n 角形と正 $2n$ 角形の関係

今、半径 1 の円に内接する正 n 角形のひとつの辺が図 6.10 の線分 AD であると仮定します。そうすると、AD の中間点 C を通る直線は辺を正確に二分しますから、線分 AB、BD は正 $2n$ 角形の二つの辺です。ここで $\angle$ ACO は直角ですから、三平方の定理から以下の式が成り立ちます。

$$\begin{aligned} |\mathrm{AC}|^2 + |\mathrm{CO}|^2 &= 1 \\ |\mathrm{AC}|^2 + |\mathrm{BC}|^2 &= |\mathrm{AB}|^2 \end{aligned}$$

この 2 式を以下のように変形します。

$$|\mathrm{CO}| = \sqrt{1 - |\mathrm{AC}|^2} \tag{6.1}$$

$$|\mathrm{BC}| = \sqrt{|\mathrm{AB}|^2 - |\mathrm{AC}|^2} \tag{6.2}$$

以下は明らかに成り立ちます。

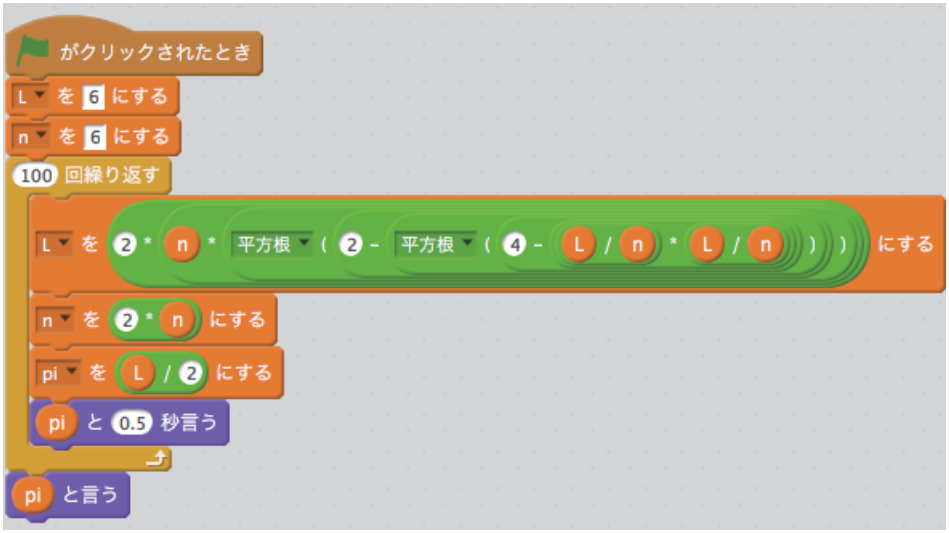
$$|\mathrm{BC}| + |\mathrm{CO}| = 1$$

この |BC|、|CO| に式 (6.1)、式 (6.2) を代入すると以下の通りです。

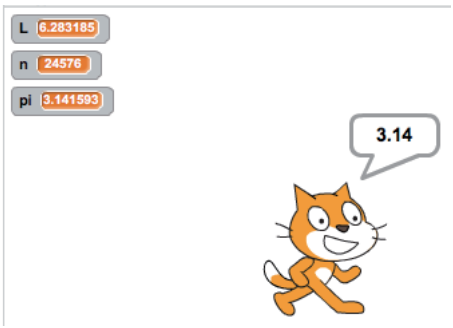
$$\sqrt{|\mathrm{AB}|^2 - |\mathrm{AC}|^2} + \sqrt{1 - |\mathrm{AC}|^2} = 1$$

この式を |AB| を左辺に置くように変形すると以下の式を得ます。

$$|\mathrm{AB}| = \sqrt{2 - 2\sqrt{1 - |\mathrm{AC}|^2}}$$



(a) プログラム



(b) 実行途中結果

図 6.11: 正多角形による円周率の計算プログラム

この式に、$|\mathrm{AC}| = |\mathrm{AD}|/2$ を代入すると以下の式を得ます。

$$|\mathrm{AB}| = \sqrt{2 - \sqrt{4 - |\mathrm{AD}|^2}}$$

正 n 角形の外周の長さ L_n は $n|\mathrm{AD}|$、正 $2n$ 角形の外周の長さ L_{2n} は $2n|\mathrm{AB}|$ です。これを上の式へ代入すると、最終的に以下の式を得ます。

$$L_{2n} = 2n\sqrt{2 - \sqrt{4 - (L_n/n)^2}} \tag{6.3}$$

さて、半径 1 の円に内接する正 6 角形について $L_6 = 6$ が成り立ちます。そこで、上の式を用いて $L_6 = 6$ から出発し、L_{12}、L_{24}、... を求めて行けば、次第に 2π に近づいて行くはずです。そして最後にその値を 2 で割れば、それが円周率の近似値です。

上の方法で円周率を計算するプログラムが図 6.11 です。このプログラムでは、途中結果も表示するようにしています。とは言え、猫は 3 桁しか教えてくれませんから、より精密な値を知るためにステージ左上の変数の値（図 6.11 の左上）を順に見ていくことにします。

```
正 12 角形          3.105829
正 24 角形          3.132629
正 48 角形          3.139350
正 96 角形          3.141032
正 192 角形         3.141452
正 384 角形         3.141558
正 768 角形         3.141584
正 1536 角形        3.141590
正 3072 角形        3.141592
正 6144 角形        3.141593
正 12288 角形       3.141593
正 24576 角形       3.141593
正 49152 角形       3.141593
正 98304 角形       3.141593
正 196608 角形      3.141593
正 393216 角形      3.141594
正 786432 角形      3.141592
正 1572864 角形     3.141609
正 3145728 角形     3.141587
正 6291456 角形     3.141674
正 12582912 角形    3.141674
正 25165824 角形    3.143073
正 50331648 角形    3.159806
正 100663296 角形   3.181981
正 201326592 角形   3.354102
正 402653184 角形   4.242641
正 805306368 角形   6.000000
正 1610612736 角形  0.000000
正 3221225472 角形  0.000000
```

円周率の真値 `3.14159265...` と比較してみてください。正 6144 角形から正 196608 角形の範囲では真値に最も近い値が求められています。しかし、さらに辺の数を増やすと、逆に真値から離れていきます。これは計算誤差によるものです。結果として、この古典的な計算方法では精度の高い計算は期待できません。なお、計算誤差については 8 章で学びます。

6.2.2 モンテカルロ法

平方根の場合と同様に円周率の計算にもモンテカルロ法が適用できます。基本的なアイディアは、図 6.12 の正方形の面積と 4 分円（4 分割した円）の面積の比を確率的に計算する方法です。

今、図 6.12 の正方形の 1 辺の長さを 1 とします。このとき、その正方形の面積は 1 です。次に 4 分円の面積は半径 1 の円の面積の 1/4 ですから、$\pi/4$ です。今、$0 \leq x \leq 1$、$0 \leq y \leq 1$ の矩形領域の中の点 (x, y) をランダムに求めたと仮定します。そうすると、その点が 4 分円の中に含まれる確率（すなわち $x^2 + y^2 \leq 1$ が成り立つ確率）は

$$\frac{\text{4 分円の面積}}{\text{正方形の面積}} = \frac{\pi/4}{1} = \frac{\pi}{4}$$

となるはずです。よって、その確率を 4 倍すれば円周率が求められます。

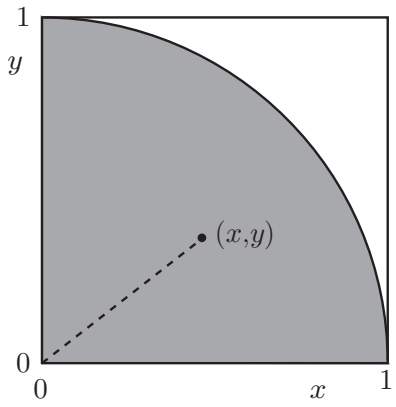
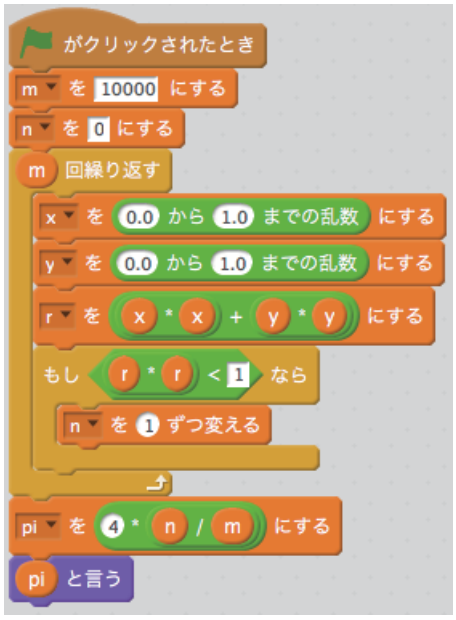
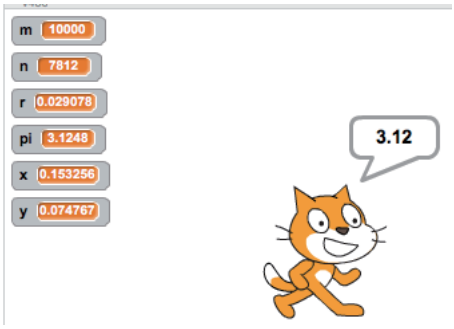


図 6.12: 正方形の面積と 4 分円の面積



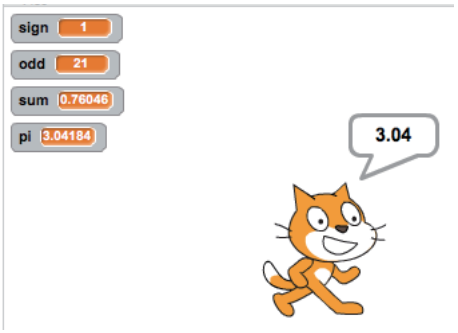
(a) プログラム



(b) 実行途中結果

図 6.13: モンテカルロ法による円周率の計算プログラム

(a) プログラム



(b) 実行途中結果

図 6.14: 逆三角関数による円周率の計算プログラム

この方法によるプログラムは図 6.13 の通りです。以下は実際に計算した結果です [8]。

3.124800 ... 1 万回の場合
3.149440 ... 10 万回の場合
3.141964 ... 100 万回の場合
3.141280 ... 1000 万回の場合
3.141208 ... 1 億回の場合

モンテカルロ法が実用に堪えないことは既に平方根の計算で見た通りです。しかし、素朴に円周率を計算できる点で興味深い手法です。

6.2.3 逆三角関数の公式

円周率を多数桁計算する方法として逆三角関数の性質を用いる方法が知られており、近年の多くの円周率計算はこの方法を基礎にしています。

まず、以下の式が成り立つことを思い出しましょう。

$$\arctan(1) = \frac{\pi}{4}$$

よって $\arctan(1)$ を何らかの方法で計算し、その値を 4 倍すれば円周率が求めることができます。

[8] 確率計算ですから、実行の度に求められる値は異なります。

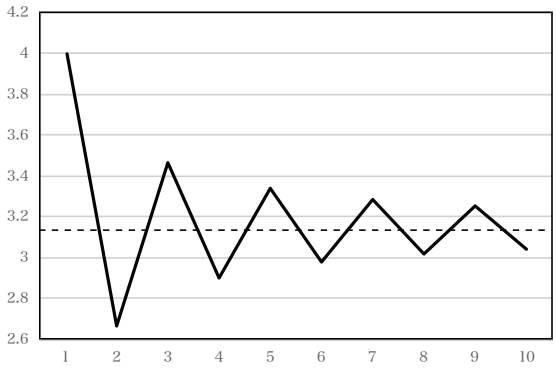


図 6.15: モンテカルロ法による円周率の計算プログラム

テイラー展開（Taylor expansion）を用いると、逆正弦関数 $\arctan(x)$ は以下のように多項式近似できることが知られています。

$$\arctan(x) = \sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1} x^{2i+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \cdots$$

特に $x = 1$ のときには、

$$\arctan(1) = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

となります。この式はライプニッツの級数またはグレゴリーの級数と呼ばれており、式自体は 15 世紀から知られていたようです。この式を愚直に計算すれば円周率に辿り着けるはずです。上の式は $i = 0, ..., \infty$ の無限級数和ですが、もちろん実際には有限の計算しかできません。そこで、$i = 0, ..., N$ までで計算を打ち切ることにし、和 $\sum_{i=0}^{N} (-1)^i/(2i+1)$ を求めてみます。

図 6.14 は $N = 10$ までの和を求めるプログラムです。このプログラムでは計算の途中結果 $\sum_{i=0}^{k} (-1)^i/(2i+1)$（$k = 0, 1, ..., 9$）も表示します。以下がその途中結果です。

```
4.0
2.666667
3.466667
2.895238
3.339683
2.976046
3.283738
3.017072
3.252366
3.041840
```

この 10 個の数値の系列をグラフ化したものが図 6.15 です。数値が円周率を挟んで上下に振れながら収束していきます。しかし収束は速いとは言えません。というのも、級数和 $\sum_{i=0}^{N} (-1)^i/(2i+1)$ の各項の絶対値 $\left|(-1)^i/(2i+1)\right| = 1/(2i+1)$ は i に反比例する値であるため、ゆっくりとした速度で

しか減少しません。求める円周率の値が 6 桁の精度に達するには、少なくとも $1/(2i+1) \leq 0.000001$、つまり $i \geq 500,000$ でなければならず、膨大な繰り返し計算を要します。

上の問題点を改良すべく、現在ではより計算に適した公式を用いています。たとえば以下は 18 世紀に発見されたマチンの公式です。

$$4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right) = \frac{\pi}{4}$$

この公式の左辺各項をテイラー展開で置き換えると以下の式を得ます。

$$4\sum_{i=0}^{N}\frac{(-1)^i}{2i+1}\left(\frac{1}{5}\right)^{2i+1} - \sum_{i=0}^{N}\frac{(-1)^i}{2i+1}\left(\frac{1}{239}\right)^{2i+1} \approx \frac{\pi}{4}$$

第 1 項は $(1/5)^{2i+1}$ に比例しますから、i の増加と共に急速に小さくなります。第 2 項 $(1/239)^{2i+1}$ は第 1 項よりもさらに急速に小さくなる数です。よってライプニッツの級数よりも少ない繰り返し回数で高い精度の値を求めることができます。

また、以下は高野 [9] の公式です。マチンの公式よりもさらに高速な計算が可能です。

$$12\arctan(\frac{1}{49}) + 32\arctan(\frac{1}{57}) - 5\arctan(\frac{1}{239}) + 12\arctan(\frac{1}{110443}) = \frac{\pi}{4}$$

その他にも類似の公式が知られていますが、詳細は省略します。

6.2.4 ガウス＝ルジャンドルの反復計算

4 番目に述べる円周率の計算方法は、特に最近の円周率の計算に用いられているものです。

計算方法は、四つの数列 a_n、b_n、t_n、p_n について以下の初期値：

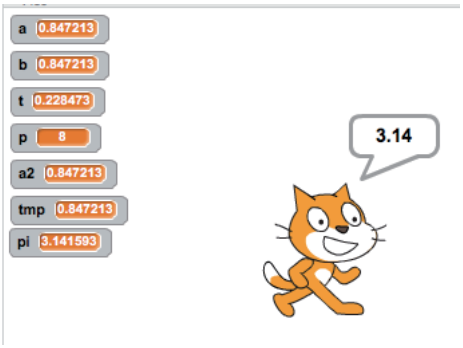
$$\begin{aligned} a_1 &= 1, \\ b_1 &= \frac{1}{\sqrt{2}}, \\ t_1 &= \frac{1}{4}, \\ p_1 &= 1 \end{aligned}$$

と漸化式（$n \geq 1$）：

$$\begin{aligned} a_{n+1} &= \frac{a_n + b_n}{2}, \\ b_{n+1} &= \sqrt{a_n \cdot b_n}, \\ t_{n+1} &= t_n - p_n \cdot (a_n - a_{n+1})^2, \\ p_{n+1} &= 2 \cdot p_n \end{aligned}$$

[9] 高野喜久雄。日本の詩人、数学者。

(a) プログラム



(b) 実行途中結果

図 6.16: ガウス＝ルジャンドルの反復計算法による円周率の計算プログラム

で反復計算します。そして円周率の値を以下の式で求めます。

$$\pi \approx \frac{(a_n + b_n)^2}{4 \cdot t_n}$$

この計算方法の理論的背景は複雑であるため、ここでは述べません。余力のある人は自ら調べてください。この計算式は計算手順が単純であるにも関わらず、計算効率がきわめて良い（真値への収束が速い）ことが知られています。

図 6.16 に Scratch のプログラムを載せます。以下は $n = 1, 2, 3$ の場合の計算結果です。

```
3.140579
3.141593
3.141593
```

急速に真値へ近づくことが分かります。

6.3 並行プログラム

このテキストのここまでの説明では、1 匹の猫のスプライトのみを取り扱い、その猫の動作を様々にプログラミングしてきました。しかし、Scratch では同時に複数のスプライトをステージ上で配置し、それぞれをそれぞれのプログラムで動かすことができます。また「ステージ」にもプログラムを与えることができます。この説ではそのような二つの例を紹介します。前節までのプログラミングとは様子が異なることに気づくでしょうか。

複数のプログラムが同時に動くことを一般に**並行実行**（concurrent execution）と呼びます[10]。しかし複数のプログラムがそれぞれ独立に動くだけでは並行実行は意味を持ちません。プログラム間でデータのやり取りが行われることで、単独のプログラムとは異なる様相の複雑なコンピュータ処理が可能になります。

データのやり取りを行う方法にはいくつかの方法があります。最も代表的な方法は**メッセージパッシング**（message passing）および**共有メモリ**（shared memory）です。この節ではそれぞれの事例を紹介します。

6.3.1 猫と犬の掛け合い

次のようなプログラムを考えましょう。

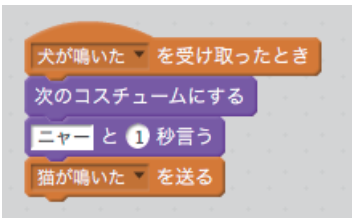
> 猫と犬の掛け合い
>
> 緑の旗をクリックすると、猫が「ニャー」と鳴き、次に犬が「ワン」と鳴き、次に猫が「ニャー」と鳴き、... という動作を 1 秒間隔で繰り返すプログラムを、猫と犬をそれぞれスプライトとして作りなさい。

猫と犬のそれぞれの動作は簡単です。問題は、鳴くタイミングをうまくコントロールすることです。ここではそれを、「イベント」のブロックパレット（図 5.6 の右列）の下の方の「[ ] を受け取ったとき」ブロックと「[ ] を送る」ブロックを用いて実現します。[ ] の中にはメッセージの種類を指定します。Scratch のオンラインエディタではあらかじめ「メッセージ 1」が登録されていますが、メッセージの種類はユーザが自由に登録することができます。ブロックの [ ] の中の「メッセージ 1▼」の▼をマウスでクリックして新規登録してください。

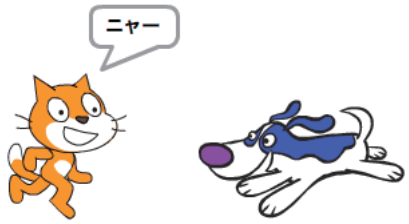
さて、猫と犬のスプライトに作成したプログラムの例を図 6.17 に示します。猫のプログラムでは、

1. 「犬が鳴いた」というメッセージを受信したときに猫のプログラムが起動し、
2. 猫のコスチュームを変更（スプライトの画像を少し変更）し、
3. 「ニャー」と 1 秒間言い、

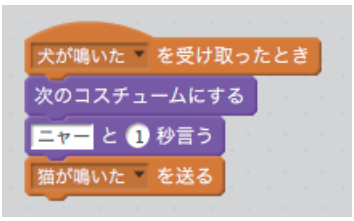
[10] 類似の術語に「並列実行」（parallel execution）があります。これら術語は区別されて使用されますが、その違いについてはこの授業の範囲を超えるので省略します。



(a) 猫のプログラム



(b) 猫が鳴いている様子



(c) 犬のプログラム

(b) 犬が鳴いている様子

図 6.17: メッセージパッシングによる猫と犬の掛け合い

4. 「猫が鳴いた」というメッセージを発信

します。同様に、犬のプログラムでは、「猫が鳴いた」というメッセージを受信したときに...（以下、同様）、最後に「犬が鳴いた」というメッセージを発信します。よって、一旦どちらかのプログラムが起動すると、この二つのプログラムは連鎖的に実行され続けます。

ところで図 6.17 では「緑の旗をクリックすると、」に相当するブロックがありません。それは「ステージ」に任せることにしましょう。図 6.18(a) のようにオンラインエディタのステージを選択し、そのスクリプトエリアに図 6.18(b) のプログラムを作ればよいでしょう。これでプログラムは完成です。まとめとして、緑の旗がクリックされた後の実行の様子（タイミングチャート）を図 6.19 に示します。

上のように、複数のプログラムがメッセージを媒介として並行動作する仕組みを「メッセージパッシング」と呼んでいます。一般的なメッセージには (1) メッセージの送信元、(2) メッセージの受信先、(3) メッセージの種類、(4) メッセージに付随するデータを指定できるのですが、Scratch では (3) のみを指定できるように単純化されています。(1) の情報はメッセージに付加されないため、受信側プログラムでは送信元は不明です（メッセージの種類をうまく設計することで判断できますが）。(2) については全員（自分を含めた全スプライト）です。もしスプライトがそのメッセージについて「[ ] を受け取ったとき」ブロックを持っていないならば、そのスプライトはそのメッセージを無視します。(4) も付加できない仕様ですから、Scratch ではあまり複雑な送受信はできません [11]。Scratch でデータを送受信したい場合には次で述べる「共有メモリ」を用います。

[11] 多くの本格的なメッセージパッシングシステムでは (1)〜(4) の全てが可能です。

(a) ステージを選択

(b) プログラム

図 6.18: ステージのプログラム

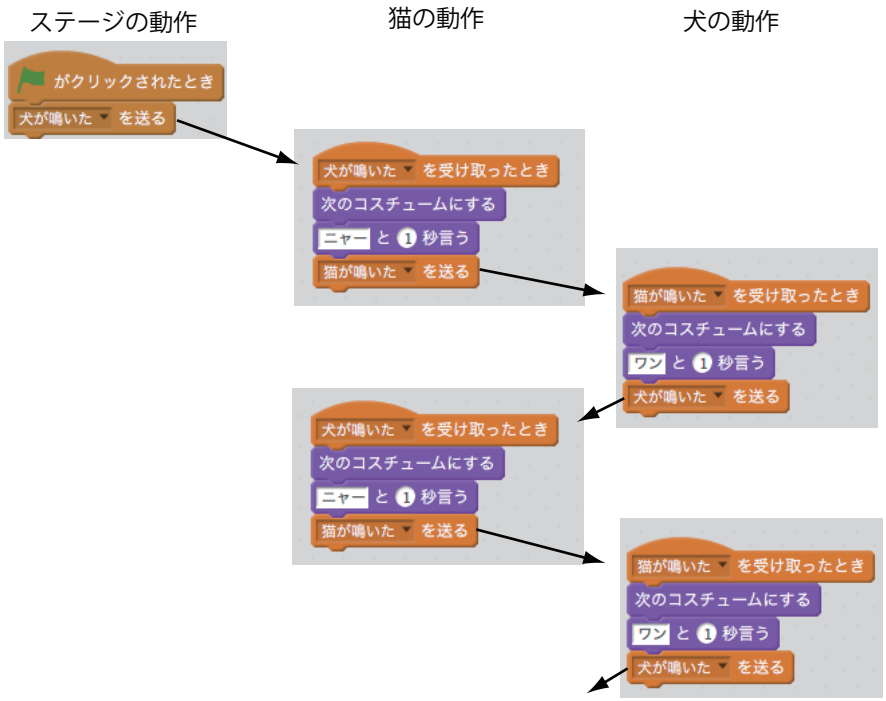


図 6.19: 猫と犬の掛け合いのタイミングチャート（矢印がメッセージパッシング）

6.3.2 4匹の猫の食事

次のようなプログラムを考えます。これは、並行プログラムの分野で知られている「食事する哲学者の問題」（Dining Philosophers' Problem）を猫に書き直したものです。「猫は箸を使って食事しない」などと野暮なことは言わないことにします。

4匹の猫の食事

緑の旗をクリックすると、4匹の猫（＝4個のスプライト）それぞれが以下のような行動を繰り返すプログラムを作りなさい。

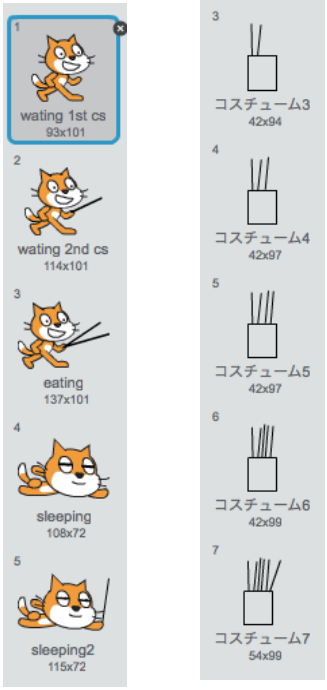
1. (a) 寝る。
   (b) 目を覚まし、食事のために1本目の箸を箸立てから取る。
   (c) 箸を持ったまま寝る（実はこの動作が重要）。
   (d) 2本目の箸を箸立てから取る。
   (e) 食事をする。
   (f) 2本の箸を洗って、箸立てに戻す。
   (g) (a) へ戻る。
2. 箸は箸立てに M 本がまとめて用意されている。M は2以上6以下の数とする。
3. 箸立てに箸がない（他の猫たちが全ての箸を使っている）ときには、猫は、少し待って、再度、箸を取る動作を行う。

4匹の猫が同時に食事をするには8本の箸が必要ですが、時間をずらして食事をするならば、それよりも少ない本数の箸で間に合うはずです。その点を考慮しつつ、上の要求事項の通りにプログラムを作成し、実行してみましょう。

今回のプログラム作成では少しだけ画面表示に凝ってみます。まず、猫と箸立ての表示のために、図6.20(a)、(b) のように複数種類のコスチュームを用意します。このコスチュームを利用したプログラムの実行のイメージは図6.20(c) のようなものです。この図では2匹が寝ており、その内の1匹は箸を1本持ったまま寝ています。他の1匹は2本目の箸を待っています（箸立てには2本の箸が残っていますが、この猫が箸立てをチェックしたときには運悪く箸がなかったのでしょう）。最後の1匹は食事中です。計6本の箸が図中にありますが、この総本数を減らすと、猫の行動に不都合が生じてきます。そこがこの問題の面白いところなのですが、それについては後述します。

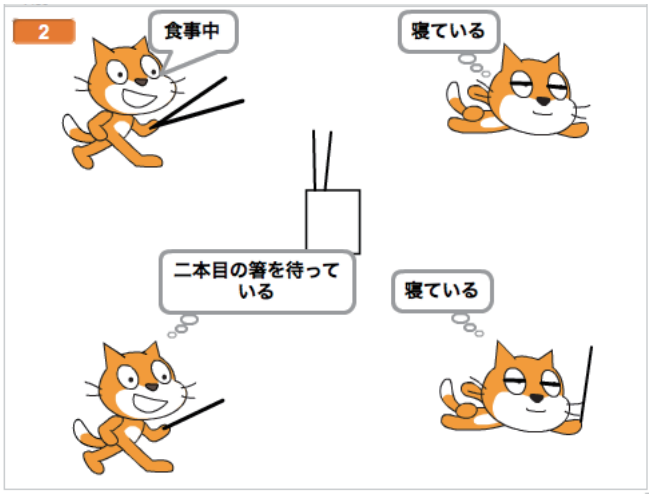
プログラムを作るに当たり、まず変数 chopsticks を用意し[12]、この変数で箸立てに残っている箸の本数を表すことにします。この変数を用いる箸立てのプログラムは図6.21(a) の通りです。二つのプログラムからなり、ひとつは、緑の旗がクリックされたときに起動し、chopsticks に初期値（図6.21(a) では6を代入し、「箸の数が更新された」というメッセージを送信するプログラムです。もうひとつは、「箸の数が更新された」というメッセージを受け取ったとき、箸立てのコスチュー

[12] ここまで、変数を作る際に現れるダイアログボックスにおいて「すべてのスプライト用」であることを前提としてきました。この例題においてはそうであることが必須です。



(a) 猫のコスチューム一覧

(b) 箸立てのコスチューム一覧



(c) 睡眠と食事を繰り返す 4 匹の猫の実行イメージ（箸が 6 本の場合）

図 6.20: コスチュームと実行のイメージ

ムを箸の数 chopsticks に合わせて変更するプログラムです。なお、「コスチュームを ( ) にする」ブロックにおいて ( ) 内に整数値を指定した場合、その整数値をコスチューム番号と解釈し、番号に該当する図 6.20(b) のコスチュームを表示します。

それぞれの猫のプログラムはプログラムは図 6.21(b) ですが、これを元の仕様に合わせて解説すると以下の通りです。

1. 緑の旗がクリックされるとプログラムが起動します。
2. 以下を無限に繰り返します。
   (a) コスチュームを「寝ている」ものに変更し、0.5 秒〜1.9 秒の間の乱数で決まる時間（以下、乱数で決まる時間）だけ待機します。

(a) 箸立てのプログラム

(b) 猫のプログラム

図 6.21: 4 匹の猫の食事のプログラム

(b) chopsticks の値が 0 である限り（＝値が 0 よりも大きくなるまで）何度でも、コスチュームを「1 本目の箸を待っている」ものに変更し、乱数で決まる時間だけ待機します。

(c) 上のループを脱出したならば、chopsticks の値を 1 だけ減らし（＝箸を箸立てから 1 本取り出し)、「箸の数が更新された」というメッセージを送信し（そうすると、箸立てのプログラムが起動し、箸立てのコスチュームが更新されます)、コスチュームを「箸を 1 本持って寝ている」ものに変更し、乱数で決まる時間だけ待機します。

(d) chopsticks の値が 0 である限り（＝値が 0 よりも大きくなるまで）何度でも、コスチュームを「2 本目の箸を待っている」ものに変更し、乱数で決まる時間だけ待機します。

(e) 上のループを脱出したならば、chopsticks の値を 1 だけ減らし、「箸の数が更新された」というメッセージを送信し、コスチュームを「食事中」ものに変更し、乱数で決まる時間だけ待機します。

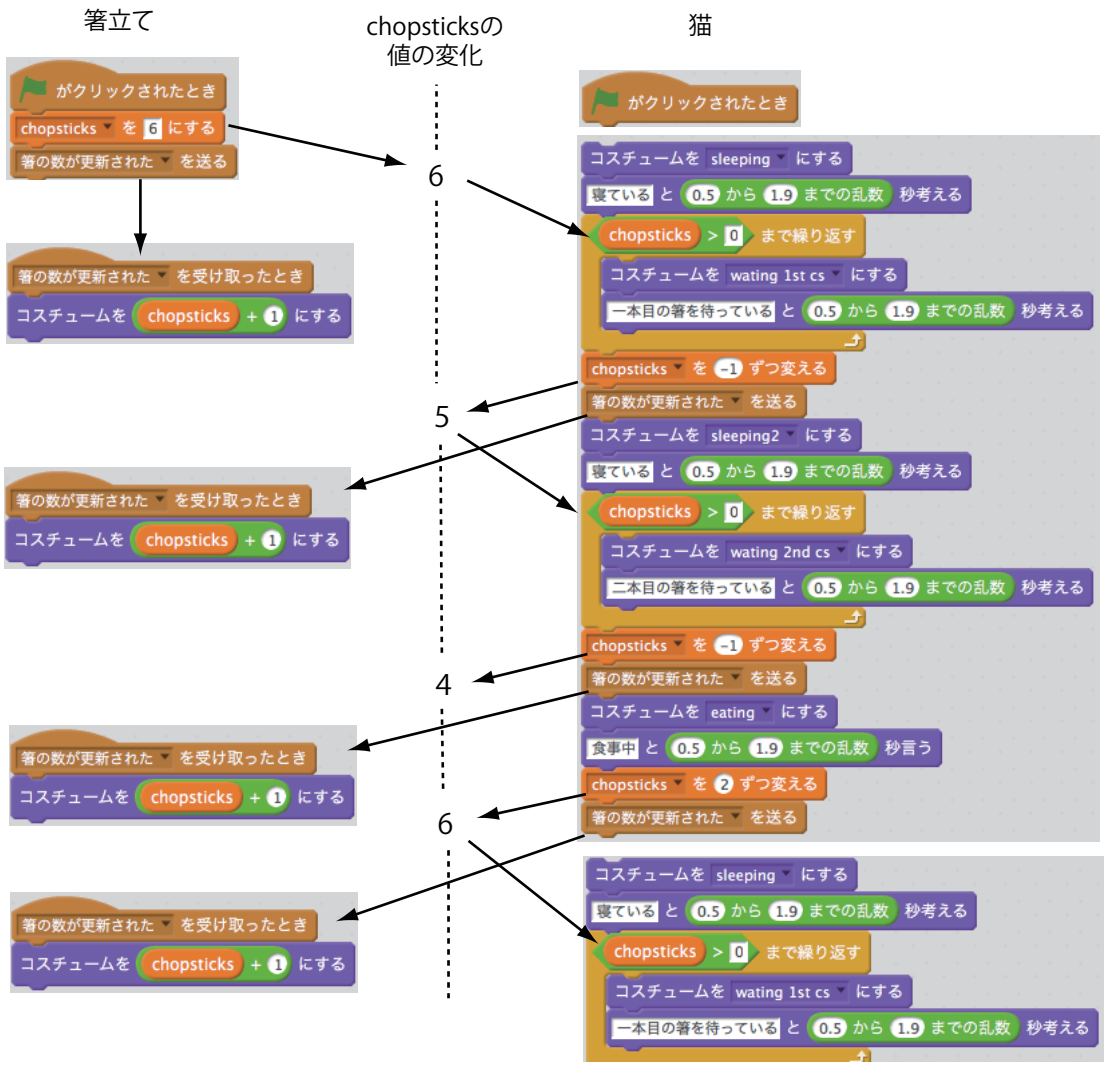


図 6.22: 箸立てと猫の動作、変数 chopsticks の値の変化のタイミングチャート

(f) chopsticks の値を 2 だけ増やし、「箸の数が更新された」というメッセージを送信します。

プログラム理解のために、図 6.22 に箸立てと猫 1 匹が動く場合の実行の様子（メッセージパッシングと変数の値の変化）を示します。

猫のプログラムが完成したならば、その猫のスプライトを複製し、4 匹に増やします。スプライトを複製するには、複製元のスプライトをマウスの右ボタンでクリックします。そうすると、図 6.23 のようにメニューが現れます。「複製」コマンドを選ぶと、スプライトリストに複製が現れます。

さっそく実行してみましょう。4 匹の猫がそれぞれ寝て食べてを繰り返す様子が見られるはずです。このプログラムでは猫どうしの直接のメッセージのやり取りはありません。しかしその代りにプログラム間で共有された変数 chopsticks（＝共有メモリ）を介して間接的なデータのやり取りを行っています。そのため、この方式を一般に共有メモリ方式による並行プログラム実行と呼んでい

図 6.23: スプライトの複製の方法（右ボタンクリックでメニューを表示し、複製コマンドを選ぶ）

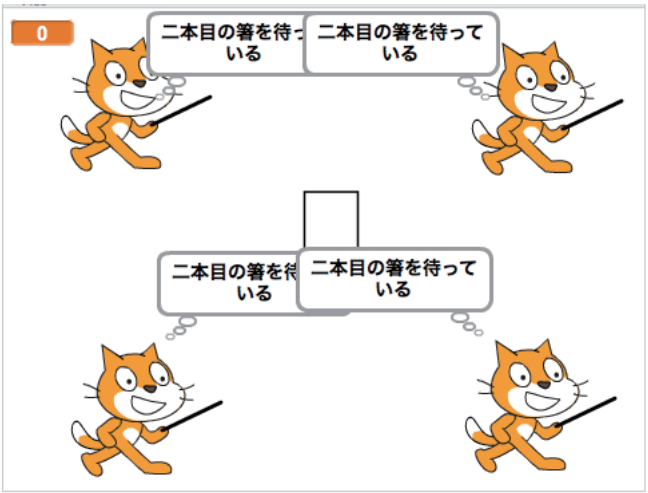


図 6.24: デッドロックに陥った状況

ます。

さてここからが問題です。箸の総数を 5 本に減らすと実行の様子はどう変化するでしょうか。猫が箸を手にするチャンスは箸の総数が 6 本の場合に比べて減るのは明らかです。しかし劇的な変化はありません。

次に箸の総数を 4 本に減らすとどうでしょうか。この場合、プログラム実行開始後に程なくしてステージは図 6.24 のような状況になります。4 匹の猫全てが 2 本目の箸を待ち続けて、しかもこの状況はその後、全く変化しません。何故ならば、残存する箸は箸立てに残っておらず、4 匹の猫はそれぞれ 1 本目の箸を取得したまま、2 本目を待ち続けるからです。

並行実行のこのような状況を **デッドロック**（deadlock）状態と呼びます。日本語訳では「行き詰まり」ですが、デッドロックという呼称が一般的です。デッドロックの原

因は、**共有資源**（shared resource）に関する**排他制御**（mutual exclusion）の失敗です。コンピュータプログラムでは様々な機器やデータを用いて計算を行います。それら機器やデータを抽象的に**計算資源**（computational resource）と呼びますが、その中でも特に複数のプログラムによって同時に利用される資源を共有資源と呼びます。この例題では箸が共有資源です。その箸の管理方法に問題であったためにデッドロックが生じた訳です。

デッドロックの回避方法にはいくつかの常套手段が知られています。この例題に関する最も簡単な解決法は、猫が箸を 1 本ずつ取得するのではなく、2 本を一度に取得するように変更することです。あるいは、猫が 1 本目の箸を取得したならば、その猫が 2 本目を取得するまでは他の猫は箸を全く取得できないように制限を掛けることです。これらの操作が上に紹介した「排他制御」の内容になります。しかし、より詳細な排他制御の方法はこの講義の範囲を超えますため、これ以上の解説は省略します。

考察　「猫と犬の掛け合い」や「4 匹の猫の食事」のように、複数の処理が協調しながら同時実行する種類の問題を自ら考案し、プログラミングしなさい。

第7章　アルゴリズムとプログラム

プログラムというものがどのようなものか、この講義テキストではScratchのプログラミングを通して簡単に紹介してきました。これまでコンピュータやプログラミングのことを全く知らなかった人も、前章、前々章を通して、それらがぼんやりと見えてきたのではないでしょうか。

7.1　プログラム開発工程

一般に、プログラムを作る作業は次のように進みます。

(A) プログラム化すべき問題が与えられる。

(B) 問題を分析し、処理内容の大きな流れ、基本構造を決める。

(C) プログラミングと直接関係しない、比較的抽象度の高いレベルで処理内容を詰める。

(D) 特定のプログラミング言語、プロセッサ向きにプログラムを作るための前準備/設計を行う。

(E) 実際にプログラムを書き下す。

(F) プログラムの間違いを修正する。

たとえば平方根を求めるという問題が与えられたとしましょう。これが上の (A) の段階です。

通常、いきなりプログラムを作成することはありません。まず、どのような方法で問題を解決するのか、基本となる数式やアイディアを出し、それをまとめていきます。前節では平方根の解法として3種類を紹介しました。そのいずれを利用するか、あるいは別の方法を用いるか、各解法の特徴を考慮して決定します。これが (B) の段階です。

そして、アイディアに致命的な欠陥はないか等を注意深く調べながら、さらに条件判定式やプログラムの分岐構造、ループ構造を少しずつ詳細に固めていきます。これが (C) の段階です。この段階ではまだ特定のコンピュータを想定している訳ではなく、前章で述べたプログラムの三大原理:

- プログラム中の文は上から順番に逐一実行されていく。
- プログラム中では、条件判定を行い、その判定結果の真偽に従い、次に実行する文を取捨選択できる。
- プログラム中に繰り返し実行部を作ることで、多数回の類似の処理を簡素に表現できる。

の下に一般的な処理手順を完成させます。この処理手順のことを一般に **アルゴリズム**（algorithm）と呼びます。問題が簡単な場合には、プログラマは無意識にアルゴリズムを作り上げるかもしれません。少し複雑な場合には、簡単なメモを取るかもしれません。しかし、より複雑な場合には図式を用いたり[1]、データの変化などを表にまとめたりしながら、計算手順を精密な形にまとめていくのが一般的です。

アルゴリズムが定まったならば、それを特定のプログラミング言語やプロセッサ上に実装（実現) するために、その言語やプロセッサに固有の設計を行います。たとえば 6.1 節では、プログラミングの前に変数名、変数の使い方を決めました。これが (D) に相当します。

そして最後にプログラミングを行います。これが (E) です。プログラムを書き下しても、それで終わりではありません。ほとんどの場合、最初に書いたプログラムには間違い — これを **バグ**（bug）[2] と呼びます — が含まれています。そこで、プログラマはプログラムを実際に実行し、実行結果を確認しながら、バグをひとつひとつ取り除いていかねばなりません。これが (F) です。

(A) から (F) へ向かう途中で不都合を発見したら、逆戻りして設計し直すことも必要です。そのような作業を繰り返して、プログラムは形になっていきます。

通常、(A) に近い段階をプログラム開発の **上流工程**（upstream process）、逆に (F) に近い段階を **下流工程**（downstream process）と呼びます。一般に、開発するプログラムの規模が大きくなればなるほど、上流工程での問題分析・設計は重要度が増し、難易度が上がり、それを担当するソフトウェア技術者には高いスキルが要求されます。それに対して下流工程は上流での設計に基づくプログラミングが中心の作業ですから、設計さえ正しく終えていれば個々のプログラミングはそれほど難しいものではありません。この意味から、下流工程よりも上流工程がより重要度が高い訳ですが、しかし上流工程のスキルは一朝一夕に身につくものではありませんし、上流工程の仕事には下流工程の管理も含まれますから、上流工程の技術者は下流工程についても知悉している必要があります。結果として、ソフトウェア技術者は当初はプログラマとして下流工程で開発経験を積みながら、上流工程のスキルを磨いていく場合がほとんどです。

本学科のカリキュラムでは、ソフトウェア技術者に必要な上流工程、下流工程の知識と技術をともに深く学ぶことができるように科目を配置しています。1 年後期からはプログラミング（すなわち下流工程の技術）を学び、2 年生後期からは **ソフトウェア工学**（これが上流工程の技術）の講義も始まります。

---

[1] この講義では省略していますが、プログラムを図式化する方法としてフローチャート（flow chart）などがよく知られています。

[2] 英単語 bug とは虫のことです。特にゴキブリやシラミなどのありがたくない虫を指します。プログラムのバグとは、プログラムに潜む、虫酸の走るような間違いのことなのです。

7.2 アルゴリズム

既に述べたように、アルゴリズムとは **処理手順** のことです。「計算法」「算法」と言ってもよいでしょう。この術後は情報科学/工学では非常に重要な言葉です。みなさん自身、本学科に在籍する 4 年間（あるいは 6 年間、9 年間）でこの単語を何度となく使うことになるはずです。

アルゴリズム（=処理手順）を可能な限り厳密に定義するならば以下の 4 条件になります。

1. 手順通りに実行すれば、正しい答えを求めることができる。
2. 手順にあいまいさがない。また個々人の経験や感性に応じて手順の解釈が異なるということがない。
3. 手順の実行は、必ず有限回の操作で終了/停止する。無限に実行し続けることはない。
4. 手順は、特定のコンピュータ、プロセッサ、プログラミング言語を想定して作られていない。

1. は当然成り立つべき事項です。

2. では、たとえば「どちらの手順を実行してもよい」というあいまいさは許されません。また「スマートと思う手順を実行しなさい」「おもしろい方の手順を実行しなさい」というのも許されません。何がスマートか、何がおもしろいかは個人の趣味・趣向に依るからです。次に実行すべき事項は必ず客観的に明確に定まっていなければなりません。

3. の条件は重要です[3]。計算が永遠に終わらないかもしれないということは非常に不都合です。アルゴリズムを作るときには、必ず有限停止することを保証すべきです。

4. の条件は、アルゴリズムとプログラムを区別する重要な条件です。プログラムはコンピュータ上で実際に実行可能な手順であり、その意味で重要です。これに対してアルゴリズムは上記 (B)、(C) の段階で作られる中間生産物に過ぎません。しかし実は、中間生産物であるアルゴリズムがプログラムの本質を表しており、多くの場合、アルゴリズムはプログラムよりも重要です。6.1 節を振り返ってみましょう。「平方根を計算せよ」という問題が与えられた場合、どの手法（二分法、モンテカルト法、ニュートン・ラフソン法のいずれ）を用いるかが、プログラムの性質、性能を決める要因となります。もしモンテカルロ法を選んだならば、どんなにプログラムを工夫したとしても実行時間の割に計算精度はほとんど向上しません。しかしニュートン・ラフソン法では、特別にプログラムの作り方を工夫しなくとも短時間で高精度の計算が可能です。この計算手法がアルゴリズムに他なりません。そしてプログラムは、そのアルゴリズムを特定の記述法を用いて実体化し、コンピュータで実行可能にしたものになります。

[3] 1.〜4. の中で 3. のみ成り立たない手順を「セミ・アルゴリズム」と呼ぶことがあります。たとえば、論理学のある種の命題をコンピュータで自動的に証明する場合に、正しい命題は有限回の手順で自動的に証明できるが、間違った命題は永遠に証明を終えることができない場合があります。世の中には 3. が成り立たない手順が多数存在することが分かっています。

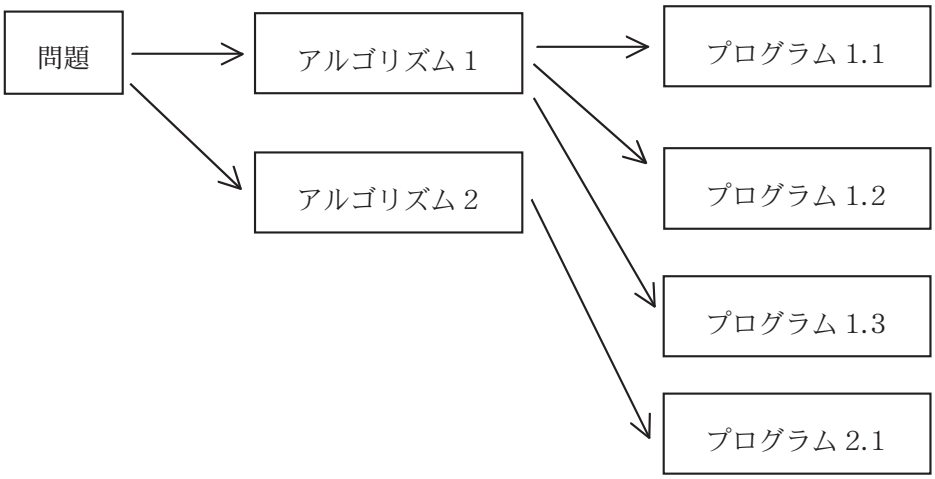


図 7.1: 問題、アルゴリズム、プログラムの関係

7.3 ふたたびプログラム開発工程

7.1 節で紹介した 6 段階のプログラム開発工程は、おおまかには

1. 問題があたえられる。
2. アルゴリズムを決定する。
3. プログラムを開発する。

にまとめることができます。この 3 段階を図 7.1 に図示します。

既に述べたように、ごく単純な問題を除けば、ひとつの問題には複数のアルゴリズムが存在します。アルゴリズムの選択は、その後の開発工程全体を大きな影響を与えますから、非常に重要です。ソフトウェア技術者は、プログラム開発の容易さ、プログラムの実行速度、保守性など様々な条件を予想し、考慮した上でひとつのアルゴリズムを選択せねばなりません。間違ったアルゴリズムを選んでしまった場合、最悪、1 から開発し直しになるかもしれません。

アルゴリズムが決まったならば、それに基づいて実際にプログラムを作成します。既に述べたように、プログラムにはちょうど日本語の作文がそうであるように様々な書き方がありうるため、プログラマは、他のプログラマにも読みやすい（可読性）、プログラムのデバッグ、改造が容易（保守性）なプログラムを作ることが重要です。プログラミングの技術は日進月歩ですから、最新の技術を駆使し、プログラム開発効率を高めることにも注力すべきです。

ここまで、コンピュータとはどのようなものか、プログラミングとはどのような作業か、初学者のコンピュータの理解を促すことを目的にプログラミング言語 Scratch を通して概観してきました。

次章からは、プログラミングの背後に見え隠れしていたコンピュータの基礎知識の解説に入っていきます。

第８章　小数点数の不思議

様々な数値がコンピュータの中にどのように格納されているかを知ることはコンピュータを深く理解していく上で不可欠の事柄です。この章からはプログラミングから一旦離れ、コンピュータ内のデータの話に移ります。

コンピュータのメモリには様々な情報を格納できます。それらの情報の代表的なひとつが数値です。コンピュータの数値の表現は二種類に大別されます（下図参照）。ひとつは **整数**、もうひとつは **小数点数** です。小数点数には固定小数点数と **浮動小数点数** の二種類があります。

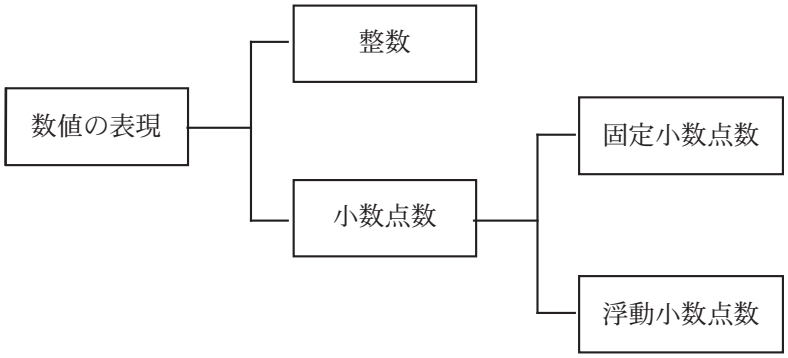


図 8.1: 数値の種類

Scratch プログラミングではその辺りを曖昧にしたままプログラミングについて述べてきましたが、そろそろ数値について詳細に知るべき段階になりました。と言うのも、コンピュータの中の数値はしばしば奇妙な挙動を示すからです。

整数とは、..., -3, -2, -1, 0, 1, 2, 3, ... などの、負の自然数、0、正の自然数のことです。最近のコンピュータでは、-2^{31}（$=-2,147,483,648$）から $2^{31}-1$（$=2,147,483,647$）の範囲の整数を主に扱います。しかし整数だけでは 0.5 のような、絶対値が 1 よりも小さい数値を扱うことができません。また、これよりも大きな数の扱いができません。このような数の表現のために、コンピュータでは浮動小数点数が考え出されました。

この章では、まず小数点数の話から始めます。

8.1 浮動小数点数

小数点数とは、いわゆる小数点を使って表される数値のことです。これには固定小数点数（fixed-point number）と浮動小数点数（floating-point number）の二種類の形式があります。しかし固定小数点数が用いられることは皆無に等しく [1][2]、コンピュータで扱う小数点数のほとんど全ては浮動小数点数という形の数値です。

浮動小数点数とは、読んで字の如く、小数点が数値の上を浮動 [3] する形式の数値です。これだけの説明ではチンプンカンプンですから、たとえば 123.45 という数値を考えましょう。これは別の表現では 1.2345×10^2 と表すことができます。また 77777 は 7.7777×10^4、0.034567 は 3.4567×10^{-2}、-3.1415 は -3.1415×10^0 と表すことができます。いま示した四つの数値:

$$1.2345 \times 10^2、7.7755 \times 10^4、3.4567 \times 10^{-2}、 -3.1415 \times 10^0$$

の共通点は、いずれも 0 でない数字が 1 の位（くらい）から始まるように小数点の位置が調整されている点です。その調整のために 10^n という倍数を用いています。この調整によって、如何なる数も必ず

$$a_1 \ . \ a_2 a_3 \cdots a_k \times 10^n$$

という形式で表すことができます。この形式を浮動小数点数の**正規形**（normal form）と呼びます。あるいはこの表現を単に浮動小数点数と呼びます。また $a_1.a_2a_3\cdots a_k$ の部分を **仮数部**（significand）、n の部分を **指数部**（exponent）と呼びます。仮数部の長さ k は有限かつ固定です。ここでは $k = 5$ とします [4]。

さて、ここで浮動小数点数の（正規形の）四則演算例をいくつか示します。まず以下の加算を考えましょう。

$$\begin{aligned} & 4.0000 \times 10^2 + 7.0000 \times 10^2 \\ \rightarrow \quad & 11.0000 \times 10^2 \\ \rightarrow \quad & 1.1000 \times 10^3 \end{aligned}$$

上の計算では、和 11.0000×10^2 を正規形 1.1000×10^3 へ変換しています。次の加算:

$$4.0000 \times 10^1 + 9.7000 \times 10^2$$

[1] 固定小数点数が用いられない理由は、実用上はそれが整数で代用できるためです。簡単に説明すると、たとえば「小数点以下を 3 桁まで扱う」と約束した場合、それらの数を 1000 倍すると必ず整数値になります。よって整数値さえ扱うことができれば、わざわざ固定小数点数を特別に扱う必要はないのです。

[2] この授業でも固定小数点数を講義しません。それを講義すると混乱する学生が出てくるからです。

[3] 「浮動（floating）」という語は、フラフラと動き、落ち着きのない様子を想起させますが、浮動小数点数について、その連想はあながち間違いではありません。

[4] 実際のコンピュータの k の長さは、10、20、40 などの大きな数です。後に述べるように、浮動小数点数には誤差が付きまといますから、$k = 5$ くらいの長さでは実用に耐えません。ここでは説明の便宜上、小さな k にしました。

$$\rightarrow \quad 0.4000 \times 10^2 + 9.7000 \times 10^2$$
$$\rightarrow \quad 10.1000 \times 10^2$$
$$\rightarrow \quad 1.0100 \times 10^3$$

では、加算する二つの数の指数部が異なります。そこで、まず、大きい方の数に指数部を揃えます。そして仮数部どうしを加算し、最後に正規形へ変換しています。乗算:

$$2.0000 \times 10^2 * 8.0000 \times 10^4$$
$$\rightarrow \quad (2.0000 * 8.0000) \times 10^{(2+4)}$$
$$\rightarrow \quad 16.0000 \times 10^6$$
$$\rightarrow \quad 1.6000 \times 10^7$$

では、仮数部どうし、指数部どうしで乗算し、その後に正規形へ変換します。

このように、任意の正規形の浮動小数点数どうしで四則演算を行い、正規形の浮動小数点数を求めることが可能です。

8.2 浮動小数点演算の誤差

既に述べたように、浮動小数点数の仮数部の長さは有限であり、固定長です。このため、数値の表現には誤差が生じます。そして誤差はしばしば奇妙な現象を引き起こします。ここでは代表的な 3 種類の現象を紹介します。これらは、コンピュータの計算が必ずしも常に正しい結果を導く訳ではないことを示唆しています。

8.2.1 丸め誤差

仮数部の大きさが 5 桁であることに注意し [5]、以下の式を計算してみます。

$$(1.0000 \times 10^0 / 3.0000 \times 10^0) * 3.0000 \times 10^0$$

計算は以下のように進みます [6]。

$$\rightarrow \quad (0.3333 \times 10^0) * 3.0000 \times 10^0$$
$$\rightarrow \quad (3.3330 \times 10^{-1}) * 3.0000 \times 10^0$$
$$\rightarrow \quad 9.9990 \times 10^{-1}$$

本来は 1 を 3 で割って 3 を掛けるのですから、答えは 1 になるべきです。しかし、仮数部が有限の大きさを持つために、1 よりも小さな数値が求められました。コンピュータで浮動小数点計算を行

[5] 指数部はここで解説する奇妙な現象とは直接関係しないので、2 桁程度としておきましょう。
[6] これは浮動小数点演算の一例に過ぎません。演算の方法はプロセッサによって種々異なります。誤差を小さくする様々な方法が現在も研究されています。

うと、仮数部の最下桁にこのような小さな誤差が発生します。これを **丸め誤差** を呼んでいます。丸め誤差は浮動小数点計算では避けて通れない誤差です。演算を繰り返し行うと丸め誤差が堆積していきます。

8.2.2 情報落ち

以下の式には加算と減算が含まれます。

$$3.0000 \times 10^{-3} + 2.0000 \times 10^{2} - 2.0000 \times 10^{2}$$

この式の加算と減算の優先順位を、左側優先として

$$(3.0000 \times 10^{-3} + 2.0000 \times 10^{2}) - 2.0000 \times 10^{2}$$

と解釈すると、演算は以下のように進みます。

$$\begin{aligned}
&\rightarrow \quad (0.0000 \times 10^{2} + 2.0000 \times 10^{2}) - 2.0000 \times 10^{2} \\
&\rightarrow \quad (2.0000 \times 10^{2}) - 2.0000 \times 10^{2} \\
&\rightarrow \quad 2.0000 \times 10^{2} - 2.0000 \times 10^{2} \\
&\rightarrow \quad 0.0000 \times 10^{2} \\
&\rightarrow \quad 0.0000 \times 10^{0}
\end{aligned}$$

すなわち答えは 0 です。元の式は普通に書くと $0.003+200-200$ を意味しますから、普通に計算すれば答えは 0.003 のはずです。しかし上の浮動小数点演算では、演算の途中で 0.003 が消えてしまい、答えは 0 になりました。0.003 が消えた理由は、200（$=2.0000\times10^{2}$）に比べ 0.003（$=3.0000\times10^{-3}$）が 5 桁以上小さ過ぎたためです。このような現象を **情報落ち** と呼んでいます。

8.2.3 桁落ち

今、測定精度が 3 桁しかない測定器からコンピュータへ二つの浮動小数点数 2.0005×10^{2} と 2.0002×10^{2} を入力し、その差を求めるとします。

$$\begin{aligned}
&2.0005 \times 10^{2} - 2.0002 \times 10^{2} \\
\rightarrow \quad &0.0003 \times 10^{2} \\
\rightarrow \quad &3.0000 \times 10^{-2}
\end{aligned}$$

このとき、求められた数値 3.0000×10^{-2} はどのような意味を持つでしょうか。元の数値の精度が 3 桁ですから、実は 3.0000×10^{-2} という数値の全体が測定誤差の範囲内であって、如何なる有用

な情報も持ち得ません。このように、大きさが近い二つの数値の間の減算によって、信頼できない桁の数値がもっともらしく浮動小数点数の最前面に現れて出てくる現象を **桁落ち** と呼んでいます。

ここでは測定器から入力された数値の差を求めましたが、この例に限らず、桁落ちは数値計算では頻繁に起こりえます。上に紹介した丸め誤差は演算の度に発生し、浮動小数点数の中に少しずつ堆積していきます。そのようにして作られた浮動小数点数どうしの減算で求められた数は、やはり桁落ちしているかもしれないのです。

上に示したように、浮動小数点数の演算には常に誤差が付きまといます。特に情報落ちや桁落ちは浮動小数点数の特異な現象であり、この特徴を認識しないままにプログラムを作ると、全く想像しなかったような変な計算結果に出会うことがあります[7]。誤差を押さえる最も一般的な方法は、仮数部の桁数を十分大きくすることです。仮数部を約 20 桁くらいの大きさに設定すると、たいていの科学計算において十分と言われています。それでも心配な場合（たとえば原子炉のコンピュータ・シミュレーションのような、決して間違いの許されない場合）には仮数部を 40 桁以上に設定して計算することもあります。

8.3 誤差の例

前章 6.2.1 項において多角形近似での円周率の計算はうまく行かないことを見ました。その理由を考察してみましょう。

以下が多角形による計算式でした。

$$L_{2n} = 2n\sqrt{2 - \sqrt{4 - (L_n/n)^2}}$$

まず、この式の中の $4-(L_n/n)^2$ が情報落ちを起こします。と言うのは、通常、私たちのコンピュータで扱う浮動小数点数の仮数部は 10 進数換算[8] で約 16 桁の精度を持ちます。また簡単のため、$L_n = 2\pi \approx 6$ と仮定しましょう。そうすると、式 $4-(L_n/n)^2$ の第 1 項 4 と第 2 項 $(L_n/n)^2$ の数値の大きさの比について

$$\frac{(L_n/n)^2}{4} < 10^{-16}$$

が成り立つと仮定するとき、$(L_n/n)^2$ が完全に無視される事態が生じるため、与式は

$$\begin{aligned} L_{2n} &= 2n\sqrt{2 - \sqrt{4 - (L_n/n)^2}} \\ &= 2n\sqrt{2 - \sqrt{4}} \\ &= 2n\sqrt{2 - 2} \\ &= 0 \end{aligned}$$

[7] コンピュータには全くの素人の、数学や物理の大先生が「コンピュータは便利だ」と聞いて、試しにコンピュータを使ってみたところ、理論値とかけ離れた計算値が出て「コンピュータは使い物にならん！」と怒ったという笑い話がかつては幾つも聞かれました。誤差の話を知らないでコンピュータを使った失敗談の典型です。

[8] 実際のコンピュータ内部では全ての数値は 2 進数で保持されているため、正確には誤差は 2 進数で議論すべきなのですが、ここでは 10 進数に換算して概算します。

となり、円周率は 0 と計算されてしまいます。簡単な式の変形から、上の条件は

$$n > 3 \times 10^8$$

と変形できます。実際、前章 6.2.1 項を見ると、正 1610612736 角形のときに円周率は 0 と計算されています。このとき $2n = 1610612736 \approx 1.6 \times 10^9$、すなわち $n \approx 8 \times 10^8$ であることから、上記の予想値とおおむね合うことが分かります。これが円周率の計算がうまく行かなかった大きな理由です。

仮に $n < 3 \times 10^8$ であったとしても、部分的な情報落ちが生じます。部分的な情報落ちが起きると $4 - (L_n/n)^2$ の下位の桁に計算誤差が混入します。結果、$\sqrt{4 - (L_n/n)^2}$ にも誤算が混入することとなります。そこで今、$\sqrt{4 - (L_n/n)^2} = 2 - \epsilon$ と置きましょう。ここに ϵ は誤差の混入した小さな数です。そうすると、$2 - \sqrt{4 - (L_n/n)^2} = 2 - 2 + \epsilon = \epsilon$ となり、誤差の混入した数が全面で出てきます。つまり、桁落ちが生じていることとなります。

以上のようにして、多角形近似での円周率の計算がうまく進まなかった理由を理解することができます。

考察　和:

$$\sum_{n=1}^{1000000} \frac{1}{n^2} = \frac{1}{1} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{999999^2} + \frac{1}{1000000^2}$$

を計算する際に、できるだけ誤差（情報落ち）を小さくする計算手順を示しなさい。また、そのプログラムを作りなさい。

第9章　2進数、10進数、16進数

コンピュータの記憶装置に格納されるデータは全て **2進数** の形式で格納されています。そしてコンピュータが四則演算を行うときには 2 進数の形式で計算しています。この章では、その 2 進数に関連する話題を取り上げます。

9.1　2進数とビット、バイト、ワード

まず 2 進数と 10 進数を対比しながら復習します。たぶん簡単すぎる内容ですが、万が一にも以下を理解できていないと先へ進めませんから、あえて復習しておきます。

1. 2 進数は、数記号 0 と 1 を用いた数の表現体系です。それに対して 10 進数は、数記号 0、1、...、9 を用いた数の表現体系です。

2. 2 進数のひとつの桁には 0 または 1 のいずれかの数字を置くことができます。よって、数の表現について言えば 1 桁当たり 2 通りの表現が可能です。それに対して 10 進数ではひとつの桁には 0、...、9 の数字を置くことができますから、1 桁当たり 10 通りの場合があります。

3. 2 桁の 2 進数は、00、01、10、11 の 4 通りの表現があります [1]。1 桁で 2 通りなので、2 桁では 2^2 通りとなるからです。それに対して 2 桁の 10 進数には、00、01、02、...、99 の 100 通りの表現があります。これは $10^2 = 100$ だからです。

4. n 桁の 2 進数には 2^n 通りの表現があります。それに対して n 桁の 10 進数では 10^n 通りの表現があります。

5. 2 進整数を小さい方から列挙すると、0、1、10、11、100、101、110、111、1000、... となります。1 桁当たりに使用できる数字が 2 種類しかないため、桁数が **急速** に伸びていきます。それに対して 10 進整数を小さい方から列挙すると、0、1、...、9、10、11、...、99、100、101、... となります。

6. ある数 m が等式:

$$m = b_p \times 2^p + b_{p-1} \times 2^{p-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

[1] 最左に 0 が現れる数値 00、01 では、0 を省略し、それぞれ 0、1 と書いても構いません。しかし、桁幅が明示されているときには 0 を先頭に付けた方が間違いが生じません。

（ただし各 b_i は 0 または 1）を満たすとき、m の 2 進数表現は $b_p\ b_{p-1}\ \ldots\ b_1\ b_0$ です。たとえば 10 進数の数 101 は

$$101 = 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

を満たすので、101 は 2 進数表現では 1100101 です。この考え方を基に 2 進数と 10 進数の間の数の変換が可能です [2]。

考察　10 進整数の 277 を 2 進整数に変換しなさい。任意の 10 進整数 n を 2 進整数に変換する方法（特に筆算で変換する手順）を示しなさい。

以上、基本事項を復習できたところで、さっそく**情報量（記憶容量）**の単位を定義します。

bit（ビット） ...　最小の情報単位 [3]。2 進数の 1 桁の大きさに等しく、0 と 1 の 2 通りの情報（表現）しかありません。

byte（バイト） ... **1文字** を表すための情報単位 [4]。通常、2 進数の 8 桁の大きさを持ち、すなわち 8 bit が 1 byte です。00000000 から 11111111 までの 256 通りの情報（表現）があります。

この二つの単位はコンピュータの知識の基本中の基本ですから、暗記してください（これを知らないと、ものすごく恥ずかしい）。bit、byte よりも大きな情報の単位も存在します。

word（ワード） ... コンピュータが **一度** に扱うことのできる（計算できる）データの大きさ。これはコンピュータの種類に依存します。かつての 16bit コンピュータの時代には主に 16 bit（=2 byte ）を 1 word と呼んでいました。現在は 32 bit または 64 bit コンピュータが主流のため、32bit（= 4 byte ）または 64bit（= 8 byte）を 1 word と呼ぶことが多いようです。bit、byte と異なり、word の大きさは環境や **時代** と共に変化します。そのため、この単位の使用には少し注意が必要ですが、情報のひとまとまりの大きさを表す便利な単位のためによく用いられます。

これ以外にも nibble（ニブル）=4 bit、double-word = 2 word、quad-word = 4 word などの情報の大きさの単位が知られていますが、特殊ですから覚える必要はありません。

注意　bit、byte の先頭文字 B だけを用いて記憶容量を表すことがしばしば行われます。たとえば「記憶容量 16MB」などです。この場合、16MB が 16 Mbit なのか 16 Mbyte なのか、判断できないという問題があります。しかしほとんどの場合に B は byte の意味で使われています。まれ

[2] 必修科目「情報基礎数学 I」で習ったはずです。この授業では軽く触れる程度にします。
[3] bit の語源は bite（噛む）です。すなわち、噛まれてバラバラになった小さな物が bit です。
[4] byte は純然たるコンピュータ用の新造語です。語源は bit と bite です。

に bit の意味で使われている場合があります。特に記憶容量の大きさを強調したいときに意図的に bit の意味で使うことが多いようです。bit と byte では 8 倍も差がありますから勘違いすると大変です。B に出会ったときには注意してください。

補足 1　上に byte は 1 文字を表す情報単位と述べました。では何故 8 bit を 1 byte に設定したのでしょうか。7 bit や 9 bit ではないのでしょうか。かつては 6 bit （$= 2^6 = 64$ 通りの情報）や 7 bit （$=2^7=128$ 通りの情報）を 1 byte と設定したこともあったようです。しかし、一般に英文字、数字、句読点、特殊記号など [5] がほぼ 8 bit 以内（256 通りの情報）に収まることが最も大きな理由とされています。1 byte は（英文の）1 文字に相当するため、 byte 数はそのまま文字数と見なすことができ、たとえば 2K byte の（英語の）文書と言えば、約 2 千文字からなる文書とすぐに判断できます。また 7 や 9 ではなく 8 を選んだ理由として、8 が 2 のベキ乗数 2^3 であることも重要と思われます。ベキ乗数はコンピュータ・ハードウェアを構築する際に何かと好都合なのです。

補足 2　「何故コンピュータは 2 進数を使うのか」、「10 進数を使った方が素直ではないか」、これは毎年必ず、みなさんから出る質問です。以下のような、いくつかの理由が考えられています。

1. 過去の電子技術の発展が、たまたま 2 値（電流が流れている/流れていない、電圧が一定以上/一定未満）を扱うことに適している。
2. 真偽を扱う論理学が 2 値に適している [6]。
3. 電子回路で整数を表現するとき、10 進数の電子回路よりも 2 進数の電子回路が素子数が少ない。[7]。

しかし、これらの理由が明確に認識され、2 進数の利用がコンピュータの常識になるには数十年の年月が必要でした。実際、世界初の電子計算機 ENIAC は 10 進数に基づいて作られたことが知られています [8]。

9.2　16 進数

2 進数は人間には扱いにくい数です。10 進数の整数 999 を 2 進数で表現すると 1111100111 となります。同じく 9999 は 10011100001111 となります。これらの 2 進数が何桁からなるか [9]、即

[5] 後に述べますが、制御文字も 1 byte で表現されます。制御文字とは、その 1 文字でコンピュータに特定の指示を与えることのできる、特別に考案された文字のことです。

[6] コンピュータは、数値演算を自動的に高速に行う機械（計算機）であると同時に、論理操作を自動的に行う機械でもあります。

[7] 実際には 3 進数が最も素子数を小さくできることが知られています。この理由は Wikipedia の「三進法」の解説部分が簡潔で分かりやすいのでそのまま引用すると、「コンピュータなどの計算機械で、N 進記法でひとケタを表現・記憶するコストが N に比例すると仮定する。すると、最大値 M までを表現・記憶できるようにするためのコストは、ひとケタぶんのコストに必要な桁数を掛けたものとなり、具体的には $N \times \log_N M$ である。この値が極小になるのは N がネイピア数 e の時であるが、e 進法は通常の数の表現には全く適さない。（引用ここまで）」なお、引用文中の「コスト」とは回路の素子数と解釈できます。実際の N は自然数ですから、$N = e = 2.71...$ とはできません。N が自然数の場合、$N \times \log_N M$ の値は $N = 3$ のとき最小値、$N = 2$ のとき 2 番目の最小値になります。

[8] 講義担当者は、2004 年夏に上野の科学博物館で ENIAC の部品を見ましたが、真空管 10 本で 10 進数 1 桁を記憶していました。

[9] 10 桁と 14 桁が正解です。

答できる人はいないでしょう。また 10011100001111 と 10010101001101 のどちらの方が大きな数か[10]、即答できる人もいないでしょう。人間が認識するには 2 進数はあまりに桁が多く、目がチラチラしてしまいます。しかし、だからと言って、2 進数を使わない訳にはいきません。コンピュータのデータ表現が 2 進数を基にしている以上、2 進数を避けては通れません。そこで、解決策として「桁の多さに人の目がちらつかず、かつ 2 進数の性質を残した」進数を用いることが行われています。

16 進数はそのような目的で利用されています。16 進数とは以下のような性質の数体系です。9.1 節の 2 進数と同じ考察を 16 進数について述べてみましょう。

1. 16 進数は、16 個の数記号 0、1、...、9、A、B、C、D、E、F（または a〜f）を用いた数の表現体系です。A〜F は英文字ですが、それを 16 進数では数記号として代用しています。

2. 16 進数の 1 桁には 0、...、F のいずれかの記号を置くことができますから、数値の表現には 1 桁当たり 16 通りの場合があります。

3. 2 桁の 16 進数には、00、01、...、0F、10、11、...、9F、A0、A1、...、FF の 256 通りの表現があります。これは $16^2 = 256$ だからです。

4. n 桁の 16 進数は 16^n 通りの表現があります。

5. 16 進整数を小さい方から列挙すると、0、1、...、F、10、...、FF、100、...、FFF、1000、... となります。1 桁当たりに使用できる数字が 16 通りもあるため、桁数の伸びは 10 進数に比べて **緩やか** です。

6. ある数 m が等式:

$$m = h_p \times 16^p + h_{p-1} \times 16^{p-1} + ... + h_1 \times 16^1 + h_0 \times 16^0$$

（ただし各 h_i は 0〜F）を満たすとき、m の 16 進数表現は $h_p\ h_{p-1}\ \cdots\ h_1\ h_0$ です。たとえば 10 進数の 101 は

$$101 = 6 \times 16^1 + 5 \times 16^0$$

を満たすので、16 進数表現では 65 です。

16 進数は 2 進数と高い親和性を持っています。$16 = 2^4$ であるため、16 進数の 1 桁は 2 進数の 4 桁 = 4 bit に厳密に対応します。つまり 2 進数の 4 桁が与えられれば、それを 16 進数の 1 桁に簡単に変換できます。逆に 16 進数の 1 桁が与えられれば、それを 2 進数の 4 桁に変換できます。これによってどんなに桁数の多い数であっても、2 進数と 16 進数の間の数の変換は簡単に行うことができます。たとえば先に出てきた 2 進数 10011100001111 は、

10 0111 0000 1111

[10]前者が大きい数です。上から 5 桁目が異なります。

と下位の桁から 4 桁ずつに分割できます。さらに上位 2 桁の 10 には 00 を追加し、それぞれが 4 桁の 2 進数になるように調整します。

$$0010 \quad 0111 \quad 0000 \quad 1111$$

そして、各 4 桁を以下の変換表:

$$0000 \to 0、\quad 0001 \to 1、\quad 0010 \to 2、\quad 0011 \to 3、$$
$$0100 \to 4、\quad 0101 \to 5、\quad 0110 \to 6、\quad 0111 \to 7、$$
$$1000 \to 8、\quad 1001 \to 9、\quad 1010 \to A、\quad 1011 \to B、$$
$$1100 \to C、\quad 1101 \to D、\quad 1110 \to E、\quad 1111 \to F、$$

で 16 進数の 1 桁に変換すれば、

$$2 \quad 7 \quad 0 \quad F$$

という 16 進数に変換できます。逆に任意の 16 進数は、各桁を 4 桁の 2 進数に変換すれば、2 進数への変換が完成します。

16 進数は、人が 2 進数を簡単に読むための表現に過ぎません。しかし、2 進数の性質が損なわれず、かつ人にとって読みやすいため、慣れてしまうと 2 進数よりもはるかに便利です。

補足　9.1 節の補足 1、2 もそうですが、この辺りの講義では素朴かつ根本的な質問が多く寄せられます。16 進数について毎年必ず出る質問に「何故 8 進数や 32 進数は用いないのか」というものがあります。

実は 8 進数は以前はよく用いられました。しかし 8 進数の 1 桁は 3 bit であり（$8 = 2^3$）、数 3 は 1 byte の bit 数である 8 を割り切る数ではありません。1 byte が 8 bit である単位系では、8 進数を用いることに居心地の悪さが付きまといます。これに対して 16 進数の 1 桁は 4 bit であり、4 は 8 を割り切ります。1 byte は 16 進数の 2 桁に対応するという単純さは歓迎される性質です。

32 進数は、2 進数とは逆の意味で人の手に負えません。16 進数において 0、1、...、9、A、B、...、F を数記号として用いたのと同様に、32 個の数字を表す記号として 0、1、...、9、A、B、...、V を用いたとします。このとき、たとえば SU という 32 進数の 2 桁の数値が 10 進数のどの位の大きさの数であるか、想像できるでしょうか [11]。32 進数 TA と SU の差がどれくらいか即答できるでしょうか [12]。ほとんどの人にとって困難な計算であり、そういう計算に慣れそうにもありません。また 32 進数の 1 桁は 5 bit であり、5 が 8 を割り切らないのも問題です。

結論を言えば、人にとって扱い易い進数は 10 に近い進数です。小さすぎても、大きすぎても人の手に追えません。そして 2 進数の性質を残す進数は 2 のベキ乗数です。さらに 8 bit を割り切る進数が好都合です。となると、結局 16 進数が残ります。

[11] 答えは 926 です。S は 28 番目の数記号、U は 30 番目の数記号なので、$28 \times 32^1 + 30 \times 32^0$ です。
[12] SU、SV、T0、T1、...、T9、TA と数えると、TA は SU から数えて 12 番目の数です。

以上で 2/10/16 進数についての基本事項の解説を終えます。これ以降もこれらの数を種々取り上げていきますが、その際にいちいち「2 進数の...」、「10 進数の...」、「16 進数の...」などと付け加えたのではとても不便です。そこで 2 進数を表す場合には添字「2」を数値の後に付け、10 進数を表す場合には添字「10」を数値の後に付け、16 進数を表す場合には添字「16」を数値の後に付けると約束します。たとえば 1010_2、1010_{10}、1010_{16} などと表すことにします。

9.3 2 進数、10 進数、16 進数の対応関係

この章のまとめとして、2/10/16 進数の対応関係を見てみましょう（表 9.1 参考）。

まず、2 進数 0_2、...、1111_2 と 10 進数 0_{10}、...、15_{10} と 16 進数 0_{16}、...、F_{16} の間の対応関係は即答できるようになってください。特に 10_{10}、...、15_{10} と A_{16}、...、F_{16} の関係は確実に覚えてください。コンピュータの内部構造を考えたり、コンピュータ内のデータを直接操作するようなプログラムを作成する際に、この辺りの数を暗記しておくと直感が働きやすくなります。byte の区切りの数の大きさも覚えてください。1 byte の大きさは 256_{10} です。その直前の数 255_{10} とペアにして覚えてください [13]。2 byte の大きさ（約 6 万 5 千）、4 byte の大きさ（約 43 億）を憶えておくのも良いでしょう。

寄り道　表 9.1 に出てくる最大数は $FFFFFFFF_{16}$ = 約 43 億です。コンピュータ・ネットワークでは、インターネットにつながれた世界中のコンピュータを区別するために、これまでは 32 bit の IP アドレス [14]（IPv4）が使用されてきました。しかし次世代の IP アドレス（IPv6）は 128 bit の大きさを持ちます。128 bit 整数の最大値は $FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF_{16}$ ですが、これは約 10^{38} に相当します。つまり IPv6 では 10^{38} 個のコンピュータをインターネットに接続可能です。地球の人口を仮に 100 億=10^{10} とすると、一人当たり 10^{28} 個のコンピュータをインターネットに接続可能となります [15]。一人の人間にとって 10^{28} という数は無限に等しく、2 章で触れたユビキタス・コンピューティングは IPv6 によって実現可能と考えられています。

[13] コンピュータ（ゲーム）関連書籍の中に「○○を 256 倍楽しむ」という名称のものがあります。何故 256 なのか、一般人にはピンと来ませんが、コンピュータ関係者ならばみんな知っています。

[14] IP アドレスとは、世界中のコンピュータに一意に割り振られた番号です。「アドレス」とあるように、国、地域、組織毎にまとまって割り振られます。詳細は、2 年生以降の授業で学びます。

[15] 同様のことを IPv4 について考えると、32 bit =約 43 億のため、地球の人口を 43 億として、一人当たり 1 個の IP アドレスしかありません。事実、現在の IPv4 仕様の IP アドレスは枯渇しかかっています。しかも全 IP アドレスの 70%をアメリカが使用しており、残りの 30%を残りの国で分け合って使用しています。そのため、国によって IP アドレスの割当格差に関する問題意識が大きく異なり、このことが問題を深刻化させているとも言われています。なお、70%の IP アドレスを有するアメリカが IP アドレスを他の国に譲渡できるかと言えば、問題はそう簡単ではありません。ネットワーク上のパケット配信の仕組みの理由から IP アドレスは連続した数値でなければ有効利用できないためです。たとえば佐賀大学の IP アドレスは 85310000 から 8531FFFF の範囲が割り当てられています。仮にこの範囲に未使用の IP アドレスがあっても、それを個別に佐賀大学外に譲渡することは不可能なのです。

表 9.1: 2/10/16 進数の対応

2 進数	10 進数	16 進数	コメント
0	0	0	
1	1	1	
10	2	2	
...	...	...	
111	7	7	
1000	8	8	2 進数がここから 4 桁になる。
1001	9	9	
1010	10	A	ここから 16 進数の数記号に英文字を使用。
1011	11	B	
1100	12	C	
1101	13	D	
1110	14	E	
1111	15	F	ここまでの対応関係は暗記しておきたい。
10000	16	10	ここから 16 進数は 2 桁になる。
10001	17	11	
...	...	...	
11111111	255	FF	1byte の大きさの数の最大の数。
100000000	256	100	1byte を越えた最初の数。256 は暗記したい。
100000001	257	101	
...	...	...	
$1\underbrace{000\cdots000}_{10\text{ 個}}$	1024	400	10bit の最大数はほぼ 1000 である。
...	...	...	
$\underbrace{1111\cdots1111}_{15\text{ 個}}$	32767	7FFF	2byte の正数（=15bit の数）の最大。
...	...	...	
$\underbrace{1111\cdots1111}_{16\text{ 個}}$	65535	FFFF	2byte の数の最大。
$1\underbrace{0000\cdots0000}_{16\text{ 個}}$	65536	10000	2byte を越えた最初の数
...	...	...	
$\underbrace{11111\cdots11111}_{31\text{ 個}}$	2147483647	7FFFFFFF	4byte 正数の最大。
...	...	...	
$\underbrace{11111\cdots11111}_{32\text{ 個}}$	4294967295	FFFFFFFF	4byte の大きさの数の最大の数。

第10章　2進数の四則演算と補数表現

この章の目的は、2 進数の四則演算が 10 進数と同様にできることの確認、そしてコンピュータ内での負数の表現方法の解説です。

10.1　2進数の四則演算

まず 2 進数の四則演算を筆算を用いて検討します。

10.1.1　2進数の加算

2 進数どうしの加算は、同じ桁どうしの加算、桁上がりの二つの原則に従い、10 進数の場合と同様に可能です。たとえば 2 進数 11111111_2 と 1100_2 の加算を筆算で行う場合、

$$\begin{array}{cccccccccc} & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ + & & & & & & 1 & 1 & 0 & 0 \\ \hline & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{array}$$

となります。2 進数の場合には数記号は 0 と 1 の二種類しかないため、慣れてしまえば計算は 10 進数よりも簡単です。

10.1.2　2進数の減算

2 進数どうしの減算も加算同様です。たとえば 2 進数 10000111_2 からの 1100_2 の減算を筆算で行う場合、

$$\begin{array}{cccccccccc} & & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ - & & & & & & 1 & 1 & 0 & 0 \\ \hline & & & 1 & 1 & 1 & 1 & 0 & 1 & 1 \end{array}$$

となります。

減算では小さな数から大きな数を引く場合が問題になります。その場合、答えが負数になります。そうすると負数をコンピュータ内に表現せねばなりませんが、それについては後に説明します。

10.1.3 2 進数の乗算

2 進数どうしの乗算も 10 進数の場合と同様です。ただし 10 進数と異なり、数字が 0 と 1 の二種類しかありませんから、乗算が劇的に簡単になります。

まず、10 進数の乗算 $34_{10} \times 23_{10}$ の筆算を確認しましょう。

$$\begin{array}{cccc} & & 3 & 4 \\ \times & & 2 & 3 \\ \hline & 1 & 0 & 2 \\ & 6 & 8 & \\ \hline & 7 & 8 & 2 \end{array}$$

ここでは、「$34_{10} \times 3_{10}$」、「$34_{10} \times 2_{10}$」という、2 桁の数と 1 桁の数の乗算を 2 回行っていることに注意してください。

同様に 2 進数の乗算 $1011_2 \times 101_2$ の筆算を考えましょう。

$$\begin{array}{cccccccc} & & & & 1 & 0 & 1 & 1 \\ \times & & & & & 1 & 0 & 1 \\ \hline & & & & 1 & 0 & 1 & 1 \\ & & & & & & 0 & \\ & 1 & 0 & 1 & 1 & & & \\ \hline & 1 & 1 & 0 & 1 & 1 & 1 & \end{array}$$

この場合も 10 進数と同様に、まず「$1011_2 \times 0_2$」、「$1011_2 \times 1_2$」という、4 桁の数と 1 桁の数の乗算を行っています。ただし 10 進数と異なるのは、この乗算が非常に簡単であることです。$1011_2 \times 0_2$ の結果は乗数が 0 ですから 0 です。次に、$1011_2 \times 1_2$ の結果は乗数が 1 ですから被乗数の 1011_2 そのものです。つまり、乗算とは言っても、結果は **0** または **被乗数** そのもののどちらかです。よって、この部分を電子回路で実現するならば、AND 回路だけで簡単に実現できます。

2 進数の除算も 10 進数の除算と同様に筆算が可能です。そして 2 進数の除算は 10 進数の除算よりも簡単な仕組みで計算できますが、詳細は省略します。

10.2 メモリへの整数の格納（補数表現への導入）

コンピュータのメモリに以下のような 2 byte = 16 bit の記憶領域があると仮定します。

もしここに整数 101_2（$= 5_{10}$）を格納するならば、数値を右詰めにして以下のように格納します。

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

これに異論を唱える人はいないでしょう。

では、負数 -101_2（$= -5_{10}$）を格納するときは、いったいどのように格納すればよいでしょうか。たとえば以下のように格納するのでしょうか。

0	0	0	0	0	0	0	0	0	0	0	0	−	1	0	1

あるいは以下でしょうか。

−	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

いずれも間違いです。コンピュータの各 bit には 0 または 1 しか格納できません。よって bit にマイナス記号「−」を格納することはできません。**0 と 1** だけを用いて負の数を表現せねばならないのです。

そこで、最も安直に思い付くのは、「負数を格納するときには最上位 bit を 1 にする」と約束する方法です。この場合、-101_2 は以下のようになります。

1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

確かにこの方法は分かりやすく、かつてはこの方法が実際に採用されたこともあったようです。しかし、現在のコンピュータでは採用されていません。現在、世の中の全てのコンピュータで採用されている負数 -101_2 の格納方式は以下の通りです。この表現方法は、正負の 2 進整数の四則演算を行う際に最も合理的な数値の表現方法とされています。

1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1

何故これが -101_2 なのでしょうか。

10.3 補数表現（アイディア）

減算の原点に立ち帰り、0_2 から 101_2 を引くことを考えてみましょう。

$$\begin{array}{rrrr} & & & 0 \\ - & 1 & 0 & 1 \\ \hline & & & ? \end{array}$$

ただし単に 0_2 から 101_2 を引くのではなく、0_2 の代わりに、**下位 16 bit**[1] が全て 0 であり、**17 bit** 目が 1 である 2 進数 10000000000000000_2 から 101_2 を引くことを考えます。下の計算を見て下さい。後の説明のために 16 bit 目と 17 bit 目の境界を線で示しました。

[1]以下、この章では bit を単位として用いますが、ほとんどの場合「桁」に置き換えて言うことが可能です。

	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
−														1	0	1
	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1

このようにして求められた 16 bit の内容 1111111111111011_2 を $-5_{10} = -101_2$ と見なせば、それが 10.2 節の最後に示した 16 bit の内容に他なりません。

ところで、この数を -5_{10} と見なすに足る正当な根拠はあるのでしょうか。もし上の数が -5_{10} であるならば、たとえばこれに $12_{10} = 1100_2$ を加えた数は $-5_{10} + 12_{10} = 7_{10} = 111_2$ になるはずです。実際に計算してみましょう。

	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
+													1	1	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

17 bit 目に 1 が繰り上がっていますが、それを **無視** し、下位 16 bit のみに注目するならば、得られた数は 0000000000000111_2 すなわち 111_2 であり、正しい計算結果です。

負数を表すための、この特別な方法は加減算について辻褄が合い、うまく行きそうです。実際うまく行きます。しかも演算において、その数が負数なのか正数なのかをいちいち意識する必要がなく、既に知っている整数の演算と見なして計算できるという利点を持ちます。

上では 16 bit （16 桁）の整数について検討しましたが、同じ計算法は任意の bit 数についても可能です。たとえば 8 bit の $-5_{10} = -101_2$ の表現を求めるには、以下のように 9 bit 目が 1、下位 8 bit が全て 0 であるような数から 101_2 を引きます。

	1	0	0	0	0	0	0	0
−						1	0	1
	1	1	1	1	1	0	1	1

結果、11111011_2 が -5_{10} を表すことになります。16 bit の場合は 1111111111111011_2 が -5_{10} の表現でしたが、計算の前提となる bit 幅が異なると、-5_{10} の表現も異なるものになるのです。次にこの 11111011_2 に $5_{10} = 101_2$ を足してみます。答えは $-5_{10} + 5_{10} = 0_{10} = 0_2$ になるはずです。

	1	1	1	1	1	0	1	1
+						1	0	1
1	0	0	0	0	0	0	0	0

9 bit 目に 1 が繰り上がりますが、それを無視するならば、確かに $00000000_2 = 0_{10}$ が求められました。

以上の事柄を、数値の bit 幅が N の場合についてまとめます。

- ある数 n について、その符号を反転した数 $-n$ を求めるには、減算 $2^N - n$ を行い、結果の下位 N bit を $-n$ の表現とします。この表現のことを特に n の **補数** （complement）または **補数表現** と呼びます。また逆に n は $-n$ の補数、補数表現でもあります。正負の整数をこのように表現する方式自体も補数表現と呼ばれます。

- 補数表現で表された数の間の加算では、通常の演算後、演算結果の下位 N bit までに注目し、$N+1$ bit 目以上は無視します。
- 補数表現で表された数の間の減算では、被減算数の $N+1$ bit 目は 1 であると仮定して演算を行います。演算結果では下位 N bit までに注目し、$N+1$ bit 目以上は無視します。

考察　8 bit 、16 bit 、32 bit の大きさについてそれぞれ -1_2 の補数表現を求めなさい。

10.4　補数表現（bit 数と範囲）

補数表現で表すことのできる数値の範囲は限られています。

簡単のため、8 bit の補数表現について述べます。10.3 節に従えば、様々な正負の 10 進数に対応する 8 bit の 2 進数は以下の表 10.1 のように順に求められます。注意したいのは、2 進数 10000000_2 には二つの 10 進数 128_{10} と -128_{10} が対応することです。また、10000001_2 には 129_{10}、-127_{10} が対応し、01111111_2 には 127_{10}、-129_{10} が対応しています。このように、ひとつの 2 進数に複数の 10 進数が対応する状況は不都合です。

これは 8 bit の 2 進数にその表現力を越えて 10 進数を割り当てようとしたことに原因があります。もともと 8 bit の 2 進数は 256 通りの数値しか表現できません。そこで、8 bit に高々 256 個の 10 進数のみを割り当てるように限定すればよいのです。そして、それ以上の個数の数を扱うときには 16 bit の 2 進数を用います。

通常、補数表現では、

- 最上位 bit が $\boxed{0}$ である数を正の数または 0 に対応づけ、
- 最上位 bit が $\boxed{1}$ である数を負の数に対応づけます。

つまり 8 bit の場合には、00000000_2 から 01111111_2 までの 128 個の数を 0_{10} から 127_{10} に対応づけ、11111111_2 から 10000000_2 までの 128 個の数を -1_{10} から -128_{10} と対応付けます。そう約束すると、表 10.1 は表 10.2 のように修正すべきです。このとき 128_{10} 以上の数や-129_{10} 以下の数は表現できません。

表 10.2 の扱える数の範囲がとても狭いことに驚くかもしれませんが、8 bit の数ではもともと 256 通りの情報しか扱えませんから、仕方ありません。もしこの範囲を越えて正負の数を扱いたいならば、16 bit （16 桁）の 2 進数を扱うのがよいでしょう。16 bit では 65536 個の範囲の数値を扱うことができます（表 10.3 参照）。しかし 16 bit で扱うことのできる範囲も高々 ± 3 万 2 千の大きさですから、少し大きな計算を行えば、あっという間にこの範囲を越えてしまいます。そこで通常、正負の整数を扱うときには 32 bit を用いるのが一般的です。この場合の範囲は表 10.4 の通りです。±約 20 億の範囲を表現できますから、たいていの整数計算はこれでまかなえます。

表 10.1: 8 bit の 2 進数補数と 10 進数の対応（重複に注意）

2 進数	10 進数	コメント
...	...	
10000001_2	129_{10}	-127_{10} と重複
10000000_2	128_{10}	-128_{10} と重複
01111111_2	127_{10}	
01111110_2	126_{10}	
...	...	0_{10} から 127_{10} までの全ての補数表現において 8bit 目が 0 になることに注目。
00000010_2	2_{10}	
00000001_2	1_{10}	
00000000_2	0_{10}	
11111111_2	-1_{10}	
11111110_2	-2_{10}	-1_{10} から -128_{10} までの全ての補数表現において 8bit 目が 1 になることに注目。
...	...	
10000001_2	-127_{10}	
10000000_2	-128_{10}	
01111111_2	-129_{10}	127_{10} と重複
01111110_2	-130_{10}	126_{10} と重複
...	...	

理論的に言えば、補数表現に用いることのできる bit 数は任意です。しかし通常、コンピュータが一括して扱うことのできる bit 数（桁数）は 8、16、32 などに限定されています[2] から、それ以外の bit 数を扱うことは実用的ではありません。このテキストでも 8、16、32 以外は用いません。

注意　補数表現では何 bit の数値を扱っているのかを明確にすることが重要です。たとえば 2 進数 11111111_2 は、もしこれが 8 bit の補数表現ならば、-1_{10} を表します。しかし 16 bit の補数表現として考えるならば、255_{10} を表していると見なされます。何故ならば、11111111_2 の bit 長は 8 bit しかなく、16 bit に足りません。このような場合、上位 8 bit に 0 を充当し、16 bit の 2 進数 0000000011111111_2 と見なすのが一般的です。そうすると、これは 255_{10} です。32 bit の表現においても上位 24 bit に 0 を充当し、255_{10} と見なします。このように、ある 2 進数を補数表現で表された数と見なすとき、前提とする bit 幅によって解釈値が変わる場合があります。無用の混乱を避けるためには、事前に「8/16/32 bit の...」あるいは「8/16/32 桁の...」などと bit 幅を明示することが重要です。

[2] 64 bit コンピュータでは 64 bit も一括して扱うことができます。しかし 32 bit コンピュータで 64 bit を扱うときには、32 bit に二分割して取り扱いますから 2 倍の手間が掛かり、演算時間は 2 倍になります。

表 10.2: 8 bit の 2 進数補数と 10 進数の対応（1 対 1 対応）

2 進数	10 進数	コメント
01111111_2	127_{10}	
...	...	8 bit 目が 0 になることに注目。
00000000_2	0_{10}	
11111111_2	-1_{10}	
...	...	8 bit 目が 1 になることに注目。
10000000_2	-128_{10}	

表 10.3: 16 bit の 2 進数（16 進数）補数と 10 進数の対応（1 対 1 対応）

2 進数（16 進数）	10 進数	コメント
$0111111111111111_2 (= 7FFF_{16})$	32767_{10}	
...	...	16 bit 目が 0 になることに注目。
$0000000000000000_2 (= 0000_{16})$	0_{10}	
$1111111111111111_2 (= FFFF_{16})$	-1_{10}	
...	...	16 bit 目が 1 になることに注目。
$1000000000000000_2 (= 8000_{16})$	-32768_{10}	

10.5 不思議な計算

10.5.1 突然、負の数が...

表 10.2 によると、8 bit の補数表現で扱うことのできる最大数は 127_{10}=01111111_2 です。さて、この数に 1 を加えると何が起こるでしょうか。「最大数に 1 を加えてはいけない」と禁止するのもひとつの考え方ですが、敢えてこの加算を遂行すれば以下の通りです。

$$\begin{array}{cccccccccc} & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ + & & & & & & & & 1 \\ \hline & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

10000000_2 は表 10.2 によると、-128_{10} を表しています。つまり、以下の等式が成り立ってしまいます。

$$127_{10} + 1_{10} = -128_{10}$$

最大数に 1 を加えると負の最小数が出てくるのです。同様のことは 16 bit /32 bit の補数表現でも起こります。

$$\begin{aligned} 32767_{10} + 1_{10} &= -32768_{10} \\ 2147483647_{10} + 1_{10} &= -2147483648_{10} \end{aligned}$$

表 10.4: 32 bit の 2 進数（16 進数）補数と 10 進数の対応（1 対 1 対応）

2 進数（16 進数）	10 進数	コメント
$0\underbrace{1111...1111}_{31\text{ 個}}{}_2 (= 7\text{FFFFFFF}_{16})$	2147483647_{10}	
...	...	32 bit 目が 0 になることに注目。
$\underbrace{0000...0000}_{32\text{ 個}}{}_2 (= 00000000_{16})$	0_{10}	
$\underbrace{1111...1111}_{32\text{ 個}}{}_2 (= \text{FFFFFFFF}_{16})$	-1_{10}	
...	...	32 bit 目が 1 になることに注目。
$1\underbrace{0000...0000}_{31\text{ 個}}{}_2 (= 80000000_{16})$	-2147483648_{10}	

このように、もともと扱える数値の最大値を越えて演算を行う現象を

オーバーフロー（overflow、桁溢れ）と呼んでいます[3]。

数をだんだん増やしていくと突然、負の数が現れる訳ですから、これは日常の感覚からすると大変奇妙なことです。もちろん通常の計算でこのようなことが起きては困りますし、大変危険です。プログラマはオーバーフローが起きないように細心の注意を払うべきです[4]。またオーバーフローが予想されるときには、 bit 幅が大きいデータ表現に変更すべきです。

10.5.2 負数どうしの乗算

補数表現の加減算では演算数の符号を気にすることなく、あたかも正数どうしのように演算しても構わないことを 10.3 節において示しました。加減算と同様に、$(-3_{10}) \times (-4_{10}) = 12_{10}$ などの乗算も、あたかも正数どうしの乗算のように簡単に実現できます。この例を示します。

簡単のため 8 bit の補数表現を用いるとします。-3_{10} の補数表現は 11111101_2 です（表 10.1 参照）。-4_{10} の補数表現は 11111100_2 です。そこでこの二つの数について、10.1 節 (c) で述べた方法と同様に、正数どうしの乗算を行ってみます。

[3]逆に-128_{10} から 1 を引くと 127_{10} になりますが、これもオーバーフローです。類語にアンダーフロー（underflow、下位桁あふれ）があります。興味のある人は自ら調べてください。

[4]銀行にコツコツと積み立て預金をしていて、あまりに預金額が大きくなったために、突然、預金が借金に変わってしまう... ようなことはあってはなりません。コンピュータの中には、オーバーフローをエラーとして自動的に検知できる機能を持つものもあります。

									1	1	1	1	1	1	0	1
×									1	1	1	1	1	1	0	0
									0	0	0	0	0	0	0	0
								0	0	0	0	0	0	0	0	
							1	1	1	1	1	1	0	1		
						1	1	1	1	1	1	0	1			
					1	1	1	1	1	1	1	0	1			
				1	1	1	1	1	1	0	1					
			1	1	1	1	1	1	0	1						
+		1	1	1	1	1	1	0	1							
	1	1	1	1	1	0	0	1	0	0	0	0	1	1	0	0

なお、8 bit 目と 9 bit 目の境には線を引きました。9 bit 目以上を無視するならば、答えは 00001100_2 です。これは 12_{10} に他なりません。負数どうしの乗算から正数が出てきた訳です。何故このような計算が可能なのか、10.7 節で理屈を述べます。

考察 $(-3_{10})*(4_{10})$、$(3_{10})*(-4_{10})$ を補数表現を用いて計算し、正しく -12_{10} が求められることを確認しなさい。

10.5.3 補数の簡単な求め方

補数を簡単に求める方法を示します。

10.3 節では 101_2 の 8 bit の補数表現を求めるために、2 進数 100000000_2 から 101_2（厳密には 00000101_2）を引きました。この操作は、次のようにも変形できます。

$$\begin{aligned} 100000000_2 - 00000101_2 &= (1_2 + 11111111_2) - 00000101_2 \\ &= 11111111_2 - 00000101_2 + 1_2 \\ &= (11111111_2 - 00000101_2) + 1_2 \end{aligned}$$

上式は、まず 11111111_2 から 00000101_2 を引き、その後に 1_2 を加える、という 2 段階の操作で補数が求められることを意味します。

そこで、前半の引き算を行うと以下の通りです。

	1	1	1	1	1	1	1	1	... 被減算数
−	0	0	0	0	0	1	0	1	... 減算数
	1	1	1	1	1	0	1	0	... 減算結果

この計算の減算数と減算結果の各 bit に着目すると、0 は 1 に、1 は 0 に、と数字の反転（ bit の反転）の関係にあることに気付きます。

0	0	0	0	0	1	0	1
↓	↓	↓	↓	↓	↓	↓	↓
1	1	1	1	1	0	1	0

1 から 0 を引けば 1、1 から 1 を引けば 0 ですから、11111111_2 からどんな数を引いても、減算数と減算結果の間では必ずこの反転が起きます。つまり実際には減算を行う必要はなく、単に bit 反転を行えばよいのです。

このことから、00000101_2 の補数を求める操作とは、00000101_2 の各 bit を反転し、その後に 1_2 を加える、という操作と考えることができます。単純な bit の反転と 1_2 の加算のみです。そのため、補数は非常に簡単に求めることができるのです。

一般化して述べます。n bit の 2 進数 $b_nb_{n-1}...b_2b_1$ の補数を求めるには、以下の 2 段階の操作を行います。

- $b_nb_{n-1}...b_2b_1$ の各 bit の 0/1 を反転し、2 進数 $c_nc_{n-1}...c_2c_1$ を求めます。この $c_nc_{n-1}...c_2c_1$ を $b_nb_{n-1}...b_2b_1$ の **1 の補数**（1's complement）と呼びます。
- $c_nc_{n-1}...c_2c_1$ に 1_2 を加えます。求められた補数を、1 の補数と対比して、特に **2 の補数**（2's complement）と呼びます。

考察　(1) 8 bit の 2 進数 00000000_2 の補数が 00000000_2 であることを確認しなさい。(2) 8 bit の 2 進数 00000001_2 の補数が 11111111_2 であることを確認しなさい。

10.6　10 進数の補数表現

ここまで 2 進数の補数表現について検討してきましたが、すっきりと理解できていないかもしれません。実は補数表現は 2 進数独自のものではなく、任意の進数について定義することができます。ここでは特に 10 進数について概説します。それを知ることで、補数表現の本質について理解が深まるのではないでしょうか。

4 桁の 10 進数を考え、それを以下のような空白で図式化します。

もしここに整数 5_{10} を格納するならば、数値を右詰めにして以下のように格納します。

0	0	0	5

この 10 進数 4 桁で負数 -5_{10} を格納するときはは以下の通りです。

9	9	9	5

これが -5_{10} であることは、下 4 桁が 0 であるような大きな数から 5_{10} を引くことで理解できます。

	1 0	0	0	0
−				5
	9	9	9	5

このようにして求められた 9995_{10} を -5_{10} と見なすことができることを二つの事実から確認しましょう。

ひとつは $-5_{10} + 6_{10} = 1_{10}$ となる事実です。

		9	9	9	5
+					6
	1	0	0	0	1

5 桁目への繰り上がりを無視すれば、1 が求められています。

もうひとつは、$(-5_{10}) * (-5_{10}) = 25_{10}$ となる事実です。

					9	9	9	5
×					9	9	9	5
				4	9	9	7	5
			8	9	9	5	5	
		8	9	9	5	5		
+	8	9	9	5	5			
	9	9	9	0	0	0	2	5

5 桁目以上を無視すれば、25 が求められています。

10 進数の補数を簡単に求めるには、まず各桁が 9 であるような数から対象とする数を引きます。たとえば -123_{10} の場合、以下の通りです。

	9	9	9	9	... 被減算数
−	0	1	2	3	... 減算数
	9	8	7	6	... 減算結果

このようにして求められた数を **9 の補数** と呼びます。次に、これに 1 を加えると、9877 を得ます。これが -123_{10} を表す数です。この数を **10 の補数** と呼びます。

このようにして私たちがよく知る 10 進数においても補数の概念は成り立つことが理解できます。

10.7 補数表現を用いた計算の理屈

（ここの内容は期末試験対象外です。参考程度に理解してください）

何故、補数表現は加減乗算に正しい結果を与えるのでしょうか。

簡単のため、8 bit の場合を考えます。-128_{10} から 127_{10} の中の任意の数 m、n が与えられた時、その補数表現 $C(m)$、$C(n)$ を以下の式で定義します。

$$\begin{aligned} C(m) &= (256M + m)\ mod\ 256, \\ C(n) &= (256N + n)\ mod\ 256 \end{aligned}$$

ただし M、N は 1 以上の任意の正数とします。すなわち $C(m)$ の値は、m に 256 の倍数 （256、512、768、... などの数） を加え、256 の剰余を求めたものであると定義します。

次に補数どうしの加算 $+_C$、乗算 $\times_C$ を

$$\begin{aligned} C(m) +_C C(n) &= (C(m) + C(n)) \ mod \ 256, \\ C(m) \times_C C(n) &= (C(m) \times C(n)) \ mod \ 256 \end{aligned}$$

と定義します。そうすると以下が成り立ちます。

$$\begin{aligned} C(m) +_C C(n) &= (C(m) + C(n)) \ mod \ 256 \\ &= ((256M + m) \ mod \ 256 + (256N + n) \ mod \ 256) \ mod \ 256 \\ &= (256M + m + 256N + n) \ mod \ 256 \\ &= (256(M + N) + (m + n)) \ mod \ 256 \\ &= (256M' + (m + n)) \ mod \ 256 \qquad (M' = M + N) \\ &= C(m + n) \end{aligned}$$

すなわち、m と n のそれぞれの補数表現の和は、m と n の和の補数表現に等しいのです。減算は、減算数を符号反転することで加算に置換できます。積についても同様の式の変形が可能です。

$$\begin{aligned} C(m) \times_C C(n) &= (C(m) \times C(n)) \ mod \ 256 \\ &= (((256M + m) \ mod \ 256) \times ((256N + n) \ mod \ 256)) \ mod \ 256 \\ &= ((256M + m) \times (256N + n)) \ mod \ 256 \\ &= (256^2MN + 256(Mn + Nm) + mn) \ mod \ 256 \\ &= (256N' + mn) \ mod \ 256 \qquad (N' = 256MN + (Mn + Nm)) \\ &= C(mn) \end{aligned}$$

これらのことから、負数を含む整数の加減乗算を行うには、演算する正負の数をそれぞれ補数で表現しておき、補数間の加減乗算を行い、最終的に得られた演算結果の補数表現を元の正負の数の表現に変換することで実現可能です。もちろん、その際にはオーバーフローが起きないことが前提です。なお、除算は取り扱いが複雑になるため、この講義では考えません。

第11章　浮動小数点数の2進数表現

浮動小数点数は8章で取り上げました。しかし8章は10進数に基づく議論であったため、実際のコンピュータの表現形式と少し異なります。実際には浮動小数点数も2進数に基づいて表されています。私たちは9章、10章を通して、既に2進数には十分に慣れ親しみましたから、この章では8章を補足説明します。

11.1　2進数の浮動小数点数

8章を思い出してください。$a_1.a_2a_3...a_k \times 10^n$ という形式の数値の表現を（10進数の）浮動小数点数の正規形と呼びました。ただし a_1 は0ではありません。同様にして、$b_1.b_2b_3...b_k \times 2^n$ という形式の数値表現を2進数の浮動小数点数の正規形と呼びます。ただし b_1 は0ではありません。

たとえば 14.5_{10} という数の10進数の浮動小数点数（の正規形）は、$1.45_{10} \times 10^1$ です。この数の2進数の浮動小数点数（の正規形）は、$1.1101_2 \times 2^3$ です。何故ならば、14.5_{10} の2進数表現は 1110.1_2 ですが、この数の小数点を左へ三つ移動すると、正規形 $1.1101_2 \times 2^3$ になるからです。

11.2　IEEEモデル

（ここの内容は期末試験対象外です。参考程度に理解してください）

上のような浮動小数点数の表現法を厳密に定めた規格のひとつとして、IEEE[1] モデル（またはIEEE表現、IEEE形式）があります。これは現在、世界中のほとんどのコンピュータで採用されている規格です。

IEEEモデルには32 bit長（単精度）と64 bit長（倍精度）の表現があり、それぞれ図11.1のようなbit形式をしています。この図の三つの部分、符号、指数部、仮数部を以下に説明します。

11.2.1　符号部

その数が正の数か、負の数かを示すbitです。符号bitとも呼びます。0ならば正数、1ならば負数を表します。多くの場合、浮動小数点数の仮数部には補数表現を用いない[2]ため、その代わりとして、このような明示的な符号bitを用います。

[1] IEEEは、The Institute of Electrical and Electronics Engineers（電子電気技術者学会）の略語です。IEEEは（IE^{3}とも書かれますが）「アイトリプルイー」と発音します。ACM（エーシーエム、Association for Computer Machinary、計算機械学会）と双璧をなすコンピュータ研究の国際的な学術団体です。共にアメリカに本拠を置いています。

[2] 整数の場合は補数表現が便利でしたが、浮動小数点数は整数ほど単純ではないため、補数表現を用いるメリットが少ないのです。

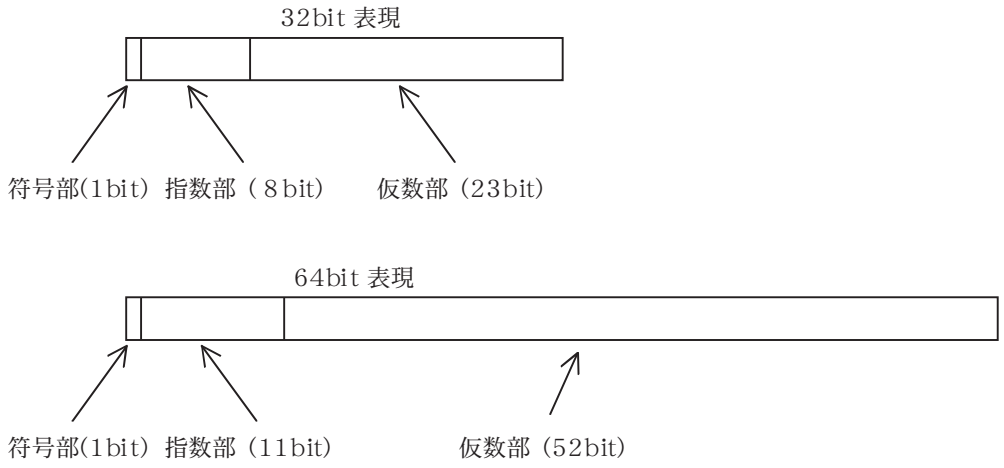


図 11.1: コンピュータの基本構成要素

11.2.2 指数部

32 bit 表現の場合、8 bit の大きさを持っています。IEEE モデルでは、指数部に補数表現は用いず[3]、指数 n に 127_{10}（=01111111$_2$）を加えた数 $n+127_{10}$ を指数部に格納します[4]。逆に言えば、たとえば指数部の 8 bit パターンが 1_{10} であるならば、指数部の表す数値は -126_{10}（$=1_{10}-127_{10}$）であることを意味します。指数部の 8 bit が 254_{10} ならば実際の数値は 127_{10}（$=254_{10}-127_{10}$）を意味します。後に述べますが、IEEE モデルでは指数部の全ての bit が 0 または 1 の場合は特殊な数値を意味するため、指数部の有効範囲は $[-127, 128]$ ではなく、$[-126, 127]$ です。

64 bit 表現では、表したい数 n に 1023_{10}（=01111111111$_2$）を加えた数 $n+1023_{10}$ を指数部に格納します。そのため、指数部の範囲は $[-1022, 1023]$ です。2^{-1023}　10^{-102}、2^{1023}　10^{102} ですから、64 bit 表現では、非常に小さな数から非常に大きな数値までを扱えます。通常のほとんどの数値計算は 64 bit 表現で足りると考えられています。

11.2.3 仮数部

仮数部は、11.1 節で示した $b_1.b_2b_3...b_k$ の部分です。ここに「b_1 は 0 ではない」という条件は、2 進数の場合には「b_1 が必ず 1 である」ことを意味します。よって仮数部は必ず $1.b_2b_3...b_k$ という形になります。つまり仮数部として記憶せねばならないデータは b_1、b_2、b_3、...、b_k ではなく、b_2、b_3、...、b_k の範囲です。この部分の大きさが、32 bit 表現では 23 bit 、64 bit 表現では 52 bit です。

11.1 節で例に挙げた数値 14.5_{10} を 32 bit の IEEE 表現で表してみましょう[5]。

[3]IEEE 以外の浮動小数点数の規格の中には、指数部の表現に補数表現を用いる規格も存在します。
[4]これをバイアス（bias）表現とも呼びます。補数表現とは異なる正負の数の表現方法です。
[5]このような問題はいかにも基本情報技術者試験に出そうな問題です。

まず、この数は正数ですから、符号 bit は 0 です。次に $14.5_{10} = 1.1101_2 \times 2^3$ でしたから、指数部は、$3_{10} + 127_{10} = 130_{10} = 10000010_2$ です。「1.1101」の「1.」を省略し、さらに右側に 0 を詰めて 23 bit にすると、仮数部は $11010000000000000000000_2$ となります。結局、14.5_{10} の 32 bit IEEE 表現は、いま求めた三つの部分をつなぎあわせた

$$01000001011010000000000000000000_2$$

となります。負数 -14.5_{10} では符号 bit のみを反転し、

$$11000001011010000000000000000000_2$$

となります。

10 進数の数値 m を 32 bit の IEEE 表現に変換する方法をまとめます。

1. m の絶対値を正規形の 2 進数 $1.b_2b_3...b_k \times 2^n$ に変換します。ただし、n が $[-126, 127]$ の範囲に含まれないならば、32 bit の IEEE 表現で表すことができません。
2. もし m が正数ならば符号 bit を 0、負数ならば 1 とします。
3. n に 127_{10} （$= 01111111_2$）を加えた 2 進数を指数部の bit パターンとします。
4. もし bit パターン $b_2b_3...b_k$ が 23 bit 以下ならば（k が 24 以下ならば）、$b_2b_3...b_k$ の後ろに適当数の 0 を付加して 23 bit のパターンを作り、それを仮数部の bit パターンとします。もし bit パターン $b_2b_3...b_k$ が 23 bit を越えているならば、$b_2b_3...b_{24}$ の 23 bit パターンを仮数部の bit パターンとします[6] （この操作は 7 章でいう丸め処理に相当し、これによって丸め誤差が発生しますが仕方のないことです）。
5. 符号部、指数部、仮数部をこの順に連結して作られる 32 bit が IEEE 表現の浮動小数点データです。
6. 上に当てはまらない特殊な数値の表現については、以下の補足を参照にしてください。

考察　10 進数 1.0_{10}、0.5_{10}、0.3_{10} の 32 bit 表現を求めなさい。

補足　IEEE モデルでは 0 や無限大などの特殊な数値を表現できるように工夫されています。これは、「0 で割る」などの演算に対応できるようにするためです[7]。特殊な数値として、以下のようなものが定義されています。

+0 ... 符号 bit が 0、指数部、仮数部の全 bit が 0。正の無限小を表します。

[6] ただし $b_2b_3...b_k$ を $b_2b_3...b_{24}$ に端折る際に、b_{25} 以下の値を用いて、10 進数の四捨五入に相当する処理を行い、b_{24} に 1 を加える場合もあります。

[7] IEEE モデルよりも以前の、このような特殊な数値の表現を持たない浮動小数点数の表現法では、「0 で割った」結果が有限値になるなどの現象が起きていました。これではプログラムの間違いを見落とす可能性があり、大問題です。

-0 ... 符号 bit が 1、指数部、仮数部の全 bit が 0。負の無限小を表します。

$+\infty$ ... 符号 bit が 0、指数部の全 bit が 1、仮数部の全 bit が 0。正の無限大を表します。$1.0/0$ などの演算によって生じます。

∞ ... 符号 bit が 1、指数部の全 bit が 1、仮数部の全 bit が 0。負の無限大を表します。$-1.0/0$ などの演算によって生じます。

NaN ... 指数部の全 bit が 1、仮数部の 1 箇所以上の bit が 1。$\log(-1)$ などの演算によって生じる未定義数を意味します。

第12章　文字列の2進数表現

8章以降、コンピュータ内に格納されるデータの表現について解説しています。その一連の内容の最後として、コンピュータ内に格納される英語の文章のデータ表現について紹介します。

12.1　メモリの構造

Scratch プログラミングではデータの格納場所として変数を用いました。5.4 節では「変数の領域はコンピュータが自動的に割り当てるため、ユーザがメモリの内部構造を知る必要はありません」と述べましたが、高度なコンピュータ処理を行うにはメモリの構造の理解は不可欠です。

メモリはデータの **1次元** 的な並びです。メモリ内の個々のデータは **1 byte** の大きさを持ち、それにアクセスするためには **アドレス** [1]（address）を用います。アドレスは、個々のデータのメモリの先頭からの位置（先頭から数えて何番目のデータか）を示す整数値であり、日本語ではしばしば「**番地**」という単位で指定されます [2]。よって、たとえばメモリの 10 番地のデータを演算に用いたいときには、プロセッサはメモリの 10 番地のデータをプロセッサ内に転送します。逆に、演算結果をメモリの 15 番地に保存したいときには、プロセッサはプロセッサ内に一時保存されている演算結果をメモリの 15 番地へ転送します。なお、先頭の番地は 1 番地ではなく **0番地** と約束されているため、たとえば先頭から数えて 3 番目のデータは「2 番地のデータ」となります。アドレスの最大値はコンピュータ毎に異なりますが、16bit コンピュータならば 65535 番地（=FFFF_{16} 番地）、32bit コンピュータならば 4294967295（=$\mathrm{FFFFFFFF}_{16}$ 番地）です（表 9.1 を思い出しましょう）。

5 章では述べませんでしたが、Scratch プログラムで用いた変数はメモリ内の特定の番地に対応付けられています。今、Scratch プログラムが 4 byte の大きさを持つ変数 a を用いると仮定します。その場合、Scratch のプログラムは変数 a を特定の番地（たとえば 101 番地〜104 番地の 4 byte）に自動的に割り当てます。そして a の数値を使用するときには 101 番地〜104 番地からデータを転送し、a に数値を代入するときには数値を 101 番地〜104 番地に転送するという処理を行っていたのです。

---

[1]「住所」という日本語訳になりますが、「住所」という言い回しが用いられることはありません。

[2]「番地」という語は日本語特有の言い回しです。英語で「... 番地」に直接対応する言い回しは無く、たとえば「10 番地に...」に対応する語句は “... at address 10”です。

アドレス	0	1	2	3	4	5	6	7	8	...
データ	49	20	61	6D	20	6F	6C	64	2E	...

図 12.1: メモリ内のアドレスとデータの例（データは 16 進数表示）

図 12.1 は、メモリのアドレスとデータの一例を図式化したものです。それぞれのアドレスには個別の 1 byte の大きさの数値データが入っています。各データは 1 byte の大きさですから、0〜255_{10}（0〜FF_{16}）の数値を格納できます。実はこの図の 0 番地から 8 番地までの 9byte のデータには英文の “I am old.” という文字列が格納されています。何故、このメモリデータが英文に対応するのか、その理由を次節で述べます。

12.2 ASCII 文字コード

コンピュータは数値（整数、浮動小数点数）だけではなく、文字も扱うことができます（ワープロ、エディタは文字を扱うソフトウェアの代表格です）。しかし、コンピュータ内のデータは全て 2 進数値として保持せねばなりませんから、文字も数値に変換して取り扱わねばなりません。文字を数値に変換することを **コード化**（coding）と呼ぶます。特定の文字の数値をその文字の**文字コード値**（あるいは単にコード値、コード）と呼び、文字と数値の対応関係表を**コード表**（code table）と呼びます。

英数字のコード表として現在、最も一般的な **ASCII コード**は、1960 年代の初めにアメリカで制定されました。ASCII とは、American Standard Code for Information Interchange（情報交換用アメリカ標準符号）に由来する言葉です [3]。現在、ほとんど全てのコンピュータは英数字にこのコードを使用しています [4]。

このコードは 7bit からなり、文字と 7bit の数値の対応は表 12.1 のように定められています。縦方向（行）の 0 から 7 が 7bit 中の上位 3bit を表し、横方向（列）の 0 から F が 7bit 中の下位 4bit を表します。たとえば $4B_{16}$（$= 1001011_2 = 75_{10}$）という ASCII コード値は、表 12.1 の 4 行 11 列目（$= B_{16}$ の列）の文字 ‘K’ が対応します。また、ASCII コード値 20_{16}（$= 0100000_2 = 32_{10}$）は 2 行 0 列目の 空白記号 ‘ ’ に対応します。

表 12.1 はおおまかに以下のような分類になっています。

00_{16}〜$1F_{16}$（0_{10}〜31_{10}） ... **制御文字**（通信などに利用されます）

30_{16}〜39_{16}（48_{10}〜57_{10}） ... 0 から 9 までの数字

41_{16}〜$5A_{16}$（65_{10}〜90_{10}） ... A から Z の英大文字

[3] パソコン雑誌の「ASCII」、「アスキー」の名称は、これに由来します。
[4] かつて IBM 社が大型コンピュータの覇者として君臨してきた頃には IBM 社の定めた EBCDIC コードも利用されていましたが、今はほとんど使用されていません。

表 12.1: ASCII コード表

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	NL	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

61_{16}〜$7A_{16}$（97_{10}〜122_{10}） ... a から z の英大文字

その他 ... 句読点などの記号

ASCII コードの主要部分を暗記しておくと、コンピュータの生データを読むときに便利です。特に 20_{16}（$= 32_{10}$）が空白記号に対応し、30_{16}（$= 48_{10}$）が数字のゼロ ‘0’ に対応し、41_{16}（$= 65_{10}$）が ‘A’ に対応すること（あるいは 40_{16}（$= 64_{10}$）が ‘@’ に対応すること）は覚えておいてもよいでしょう。

制御文字とは、ディスプレイの状態を変更するなどの特別の役割を受け持つコード値のことで、文字には対応しません。制御文字の多くは歴史的使命を終え、現在はあまり使用されていません。以下には、現在でも使用されている代表的な制御文字を説明します。

BEL ... Beep（ビープ）音を鳴らしたりするときに使用します。このコードを普通の文字のように画面に表示させようとすると「ピッ」、「ビー」というような音が鳴ります。一般に注意を喚起するときに使用します。

BS ... Back Space の略。カーソルの一つ前の文字を削除する意味を持ちます。この文字を画面に出力すると、カーソルの直前の 1 文字を削除し、カーソルがひとつ分だけ左に戻ります。

NL ... New Line の略。改行を表す文字です。NL は LF（Line Feed）と呼ばれることもあります。

CR ... Carriage Return の略。行の復帰を表すものです。カーソルが現在の行の先頭に移動します。NL と CR は共に文章の改行に利用されています。厳密に言えば、改行は NL であり、CR は次の行へ移動しないため改行ではありません。しかし類似した意味を持つため混乱が生じ、システム毎に改行を表す文字が異なるという事態を招いています。UNIX 系では NL が、Macintosh では CR がそれぞれ改行を表す文字です。Windows では CR+NL の 2 文字が改行を表しています。そのためテキスト・データをコンピュータ間で転送したり [5]、テキスト処理のプログラムを自分で作成するときには注意が必要です。

[5]転送時に自動的に改行文字を変換するソフトウェアもありますが、常にそうとは限りません。

ESC ... Escape の略。このコードは様々な場面で様々な意味を持ちます。多くのキーボードでは左上に ESC キーが存在します。

DEL ... Delete の略です。カーソル位置の文字を削除するという意味を持ちます。BS と類似しており、NL と CR の関係同様に、混乱を招いています[6]。

さて、図 12.1 のメモリの 0 番地から 8 番地の数値は以下の通りでした。

アドレス	0	1	2	3	4	5	6	7	8	...
データ	49	20	61	6D	20	6F	6C	64	2E	...

よって、これらの数値に対応する文字を表 12.1 から見つけると、以下の通りです。

‘I’, ‘ ’, ‘a’, ‘m’, ‘ ’, ‘o’, ‘l’, ‘d’, ‘.’

コンピュータ内の英文データは全てこのような方法で記憶されています。ワープロ、エディタは、ASCII コード表の対応に基づいて、ユーザがキーボードから入力した英文を数値データの列に変換するソフトウェアです。ワープロ、エディタでは文書の修正も可能ですが、それは記憶されている数値データの列を別の数値データに置き換える作業に相当します。たとえば、上記英文 “I am old.” の “old” を “sad” に変更するには、メモリの 5 番地、6 番地の内容を以下のように変更すればよいのです。

アドレス	0	1	2	3	4	5	6	7	8	...
データ	49	20	61	6D	20	73	61	64	2E	...

“I am sad.” を “He is sad.” に修正するには少し手間が掛かります。“I am” が空白を含めて 4 文字であったのに対して “He is” は空白を含めて 5 文字です。そこで、先頭に 1 文字分の記憶領域を増やすために “sad.” の内容を 1 番地分だけ後ろへ移動し、その後に先頭に “He is” を格納します。結果は以下の通りです。

アドレス	0	1	2	3	4	5	6	7	8	9	...
データ	48	65	20	69	73	20	73	61	64	2E	...

さらに修正を続けます。もし「英文中の全ての英子文字を英大文字に変換したい」と考えたならば、どのような処理を行えばいいでしょうか。表 12.1 によれば、メモリ中の 61_{16}〜$7A_{16}$ の数値をそれぞれ 41_{16}〜$5A_{16}$ へ変換すればよいことが分かります。すなわち、メモリ中の 61_{16}〜$7A_{16}$ の数値から 20_{16} を引けばよいのです。よって、上の “He is sad.” を “HE IS SAD.” にするにはメモリの内容を以下のように変更します。

アドレス	0	1	2	3	4	5	6	7	8	9	...
データ	48	45	20	49	53	20	53	41	44	2E	...

残念ながらプログラミング言語 Scratch にはこれらの文字列処理をプログラミングする機能は用意されていません。「プログラミング概論 1/2」で学ぶ C++ではコンピュータのメモリに直接アクセスすることができますから、文字列処理について実際にプログラムを作成する演習も行います。

[6] DEL、BS のいずれかしか持たないキーボード、両方持つキーボードなどがあります。どれを削除キーにするか、環境設定できるソフトウェアもあったりします。

補足　ASCII は英文字のための文字コード表です。英語のアルファベットの数は限られているため、各文字とも 7bit の大きさに収まります。しかし、日本語では漢字/ひらがな/カタカナなど数千に及ぶ文字を用いるため、各文字のコード値は 1 byte に収まりません。そこで日本語文字では 2 byte の大きさに 1 文字を格納しています。現在の日本語のデジタル文書では、主に 3 種類のコード体系：JIS 漢字コード [7]、シフト JIS 漢字コード、EUC 漢字コード [8] が用いられています。また最近では世界中の全ての言語の文字を 2 byte（または 4 byte）に収める Unicode[9] も検討されています。

8 章からこの章まで、様々なデータのコンピュータ内部での構造を学んできました。ここで学んだ事柄の多くは 1 年生後期に始まる科目群、特に「プログラミング概論 1」の学習に役立ちますので、しっかりと理解しておきましょう。

[7] JIS は Japanese Industrial Standards の略です。
[8] Extended Unix Code の略。
[9] コンピュータが生活に浸透している現在、各言語の文字をどのようにコード化するか、国益を左右するため、国際的な論争も起きているようです。

第13章　まとめ

この講義では、次の事柄を学びました。

- コンピュータの歴史、種類、用途
- Scratch を用いたプログラミングの入門
- 2 進数、16 進数、補数表現、浮動小数点、ASCII コードなどのデータ表現の基礎

ここで学んだ事柄はいずれもコンピュータの基礎の基礎です。幾つかの言葉については講義中に「覚えていないと恥ずかしい」と指摘しました。そのようなところは特によく復習しておいてください。また、後期から始まる授業ではこれらを発展させた内容が出てきます。

この講義でほとんど触れることができなかった事項として以下があります。

- コンピュータネットワーク
- データベース
- 暗号
- 人工知能
- 信号処理

これらは今後の授業で学ぶことになります。