佐賀大学全学教育機構

「情報科学の世界I」講義資料

担当者: 山下義行

2022 年度前期

目次（一部変更の可能性あり）

第 1 章　授業の概要 **1**

1.1　講義で学ぶ事項 1

1.2　講義の進め方、成績 2

第 2 章　Scratch プログラミングの基礎 **5**

2.1　Scratch とは 5

2.2　Scratch プログラミングのための準備 5

2.3　簡単なプログラムの編集と実行 13

2.3.1　猫が少し歩く 13

2.3.2　猫が少し歩いた後、方向を変える 14

2.3.3　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 15

2.3.4　緑の旗をクリックすると、猫が回転し続ける 15

2.3.5　緑の旗をクリックすると、猫が右往左往し続ける 16

2.3.6　ブロックの種類 16

2.4　処理の実行順序と変数の利用 18

2.4.1　変数 18

2.4.2　原点からの距離の計算 20

2.4.3　プログラムの保存 21

2.4.4　似ているプログラム 22

2.5　条件判定と実行の分岐 23

2.5.1　第一象限の判定 24

2.5.2　等価なプログラム 25

2.6　犬のスプライト 26

2.7　第 3 原理、第 4 原理 28

第 3 章　オートマトンへの準備 **32**

3.1　グラフの復習 32

3.2　記号列 33

3.3　言語 34

第 4 章　決定性有限状態オートマトン　38

4.1　例題：缶ジュースの自動販売機 38
4.2　状態と遷移 41
4.3　有限状態オートマトンの例 42
4.4　練習問題 46

第 5 章　決定性有限状態オートマトンのプログラミング　50

5.1　並行プログラム 50
5.2　自動販売機のプログラミング 54
5.2.1　全体の構造 54
5.2.2　四つのボタンのプログラム 54
5.2.3　猫のオートマトン制御部 55
5.2.4　グラスのプログラム 58
5.3　猫のオートマトン制御部のもうひとつの実装 60
5.4　その他のオートマトンのプログラミング例 62
5.5　練習問題 63

第 6 章　非決定性有限状態オートマトン　66

6.1　非決定性 66
6.1.1　非決定的な動作とは 66
6.1.2　缶ジュースの自動販売機 67
6.2　様々な NFA 69
6.3　練習問題 73

第 7 章　非決定性有限状態オートマトンのプログラミング　76

7.1　スプライトのクローン 76
7.2　自動販売機のプログラミング 77
7.2.1　改造のポイント 78
7.2.2　猫のオートマトン制御部 79
7.2.3　グラスのプログラム 81
7.3　その他のオートマトンのプログラミング例 86

第 8 章　オートマトンの等価変換　90

8.1　アイディア 90
8.2　変換方法 91
8.2.1　簡単な場合 91

8.2.2 自動販売機 95
8.3 DFA と NFA の等価性 97
8.4 DFA と NFA の違い 97
8.5 練習問題 98

第 9 章 正規表現 **101**
9.1 定義 101
9.2 正規表現の例 103
9.3 正規表現の応用例 105
9.4 有限状態オートマトンと正規表現の関係 109

第 10 章 出力付きオートマトン **113**
10.1 ムーア機械 113
10.2 ミーリー機械 115
10.3 改良型自動販売機のミーリー機械 116
10.4 ムーア機械とミーリー機械の等価性 116

第 11 章 出力付きオートマトンのプログラミング **120**
11.1 3 で割った余りを出力するムーア機械 120
11.1.1 二つのボタンのプログラム 120
11.1.2 猫のムーア機械のプログラム 120
11.2 改良型自動販売機のミーリー機械 123
11.2.1 三つのボタンのプログラム 123
11.2.2 猫のミーリー機械制御部 124
11.2.3 缶ジュースのプログラム 124
11.2.4 50 円硬貨のプログラム 125

第 12 章 プッシュダウンオートマトン **129**
12.1 スタックとは 129
12.2 スタックを用いた動作 130
12.3 決定性と非決定性 134

第 13 章 プッシュダウンオートマトンのプログラミング **140**
13.1 リスト 140
13.1.1 リスト型変数の作成 140
13.1.2 リストへのデータの追加 140

13.1.3 リストからのデータの削除 141
13.2 $a^n b^n$ を認識するプッシュダウンオートマトン 144

第 14 章 オートマトンの階層 **154**
14.1 四つの階層 154

第 15 章 チューリング・マシン **159**
15.1 定義 159
15.2 動作例 161
15.3 チューリング・マシンの表現力 161
15.3.1 四則演算 161

第 16 章 チューリング・マシンのプログラミング **166**
16.1 $a^m b^n \ \ (m > n)$ を認識するチューリング・マシン 166
16.2 もうひとつの $a^m b^n \ \ (m > n)$ を認識するチューリング・マシン 172
16.3 $3 + 2 = 5$ の計算を行うチューリング・マシン 176

第 17 章 チューリング・マシンについてのさらに踏み込んだ話題 **181**
17.1 万能チューリング・マシン 181
17.2 チューリング完全 183
17.3 計算可能と決定不能 185
17.4 停止性問題 186

第1章　授業の概要

1.1　講義で学ぶ事項

この講義で学ぶことは、コンピュータの動作やプログラミングに関する数学的な理論とそれに基づくコンピュータプログラミングです。

「コンピュータとは何か？」を知る早道は、コンピュータプログラミングを学ぶことです。しかし、プログラミングを学んだだけでは実はプログラミングの本質は分かりません。それを支える「計算の理論」を学ぶことも重要です。この授業ではコンピュータプログラミングの背後にある数学的な理論の概要を簡単なプログラミングと共に学び、コンピュータの本質に触れます。

授業は「コンピュータプログラミング」と「計算の理論」の両輪で進みます。

プログラミングでは、教育用プログラミング言語 Scratch を用います。小学生のプログラミング教育にも利用される言語ですが、多機能言語であり、大人のプログラミング教育にも耐えられるものになっています。実際、この授業ではメッセージパッシングを用いた高度なプログラミングを行います。

計算の理論にはいくつかの種類がありますが、ここでは最もポピュラーなオートマトンの理論を紹介します。具体的に以下の事項を学びます。

有限状態オートマトン オートマトン（automaton）は自動機械（automatic machine）と言い換えることができます。オートマトンは単に理論的な興味だけではなく、実用上はたとえばロボットの動作アルゴリズムの表現など様々な場面で応用されています。「有限状態」とは機械が有限の記憶領域しか持たないことを意味します。有限であることは大きな制約ですが、それによって興味深い数学的性質が成り立ちます。

プッシュダウンオートマトン 上の有限状態オートマトンに無限長のスタックを追加した自動機械です。機能が追加されているため、有限状態オートマトンよりも複雑な処理が可能になります。

チューリング・マシン [1] 現在私たちが使用しているコンピュータの数学的基礎とされるものです。コンピュータの限界を考える上で重要です。

[1] チューリング・マシンとも表記します。原語は Turing machine です。

1.2　講義の進め方、成績

まず、状況によっては以下に述べる内容を変更することもありうると了解してください。

講義は、原則、オンデマンドで進めます。講義資料（pdf ファイル、解説動画）は Microsoft Teams にアップロードします。

成績は以下の 3 項目で評価し、0 点〜100 点の成績を付ける予定です。

小テスト ... 達成度確認のための小テストをオンライン試験で実施します。配点は毎週 2 点、計 30 点を予定しています。小テストは (1) 確認テスト、(2) その週の授業に関する疑問、質問、感想、の 2 問からなり、それぞれの配点は 1 点です。重要な質問、おもしろい質問には毎週、動画で回答します。

レポート点 ... Scratch のプログラムを作成する課題を 2 回または 3 回予定しています。計 30 点の配点を予定しています。

期末試験点 ... 期末試験に 40 点の配点を予定します。小テストを拡充した内容のオンライン試験で実施する予定です。

1

1

第2章　Scratchプログラミングの基礎

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。オートマトンのプログラミングは 5 章以降で行います。

2.1　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。

様々なプログラミング言語の種類の中でもビジュアルプログラミング言語の範疇に入ります。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]。この授業では、Scratch を用いてオートマトンをプログラミングします。

小学生が使うから幼稚ということはありません。大学生のプログラミング学習に耐えうる内容を持っています[2]。Scratch では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ります。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。実際、この授業ではイベント駆動、並行プログラミング、メッセージ、クローン生成などの Scratch の機能の中でも高度な種類の機能を用いてオートマトンをプログラミングします。

2.2　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[3]。

令和 4 年 3 月現在、インデックスページは図 2.1 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で読むことができる訳ではありません。

[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
[2] が、さすがにプロフェッショナルが仕事に使う言語ではありません。
[3] 適当な検索エンジンで「scratch」を検索した場合、たいてい検索結果のトップに上の URL が表示されます。

図 2.1: Scratch のインデックスページ

図 2.2: Scratch のチュートリアルウインド：この授業では不要

注意：　現在の Scratch は、2019 年に更新されたバージョン 3.0 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。これらの内容は現在のバージョンとは**かなり異なります**ので注意してください。

実際に Scratch でプログラムを作るには、図 2.1 の、二つの赤い矢印の先のどちらかをクリックしてください。

図 2.3 がクリック後に移動するページです。移動直後にはチュートリアルウインドウ（図 2.2）がポップアップされているかもしれませんが、この授業では用いませんから、閉じてください。

図 2.3 が、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 2.4 はそのエリアの場所と名称です。赤枠で囲まれた 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（script）と呼びます。しかし無用な混乱を避けるために、この授業では可能な限り「プログラム」という呼び方を用います。

図 2.3: Scratch のエディタ・ページ

図 2.4: 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。手動でも変更可能です（たとえば猫をマウスでドラッグできます）。

スプライトペイン ... プログラムで動く全てのスプライトをリストアップしたエリアです[4]。 **スプライト** （sprite、妖精、小人の意味）はもともとビデオゲームの中で動き回る小さなキャラクタのことです。

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全ての部品 — 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。スプライトペインは出演する配役を並べた場所です。スクリプト＝プログラム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。

スプライトの実行のためのブロックはブロックパレットに用意されています。図 2.4 のブロックパレットの左側にブロックの種類が順に表示されています。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

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

変数 ... 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

その他 ... 手続きの抽象化を支援するブロック

[4]ペインの原語は pane です。pane は「領域」（area）というよりも「枠」というニュアンスらしい。

実際にエディタに表示されたブロックパレットは図 2.5、図 2.6、図 2.7 の通りです。ざっと眺めただけで多種多様な機能を使うことができると分かります。また図 2.3 の左下に「拡張機能」があります（図 2.4 左下矢印参照)。ここをクリックし、さらに便利な機能を追加することも可能です（この授業では拡張機能は用いません)。

なお、この授業は Scratch の講座ではありませんから、Scratch の全機能を解説するものではありません。この授業に必要な機能に限定して解説していきます。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。

また、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[5]。

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、基本的には日本語版 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。

注意：　この講義テキストに Scratch の様々な画像を紹介しますが、画像によって色味が異なって見えることがあります。たとえば、図 2.1 と図 2.5 の青いブロックでは発色が異なります。これはこの講義テキストの図を作る際に利用している編集ソフトウェアの色加工のせいで、現時点ではその解決方法を見つけることができないでいます。見づらく感じることがあるかもしれませんが容赦ください。

[5] Web ブラウザを利用しない実行方法もありますが、この授業では使用しません。

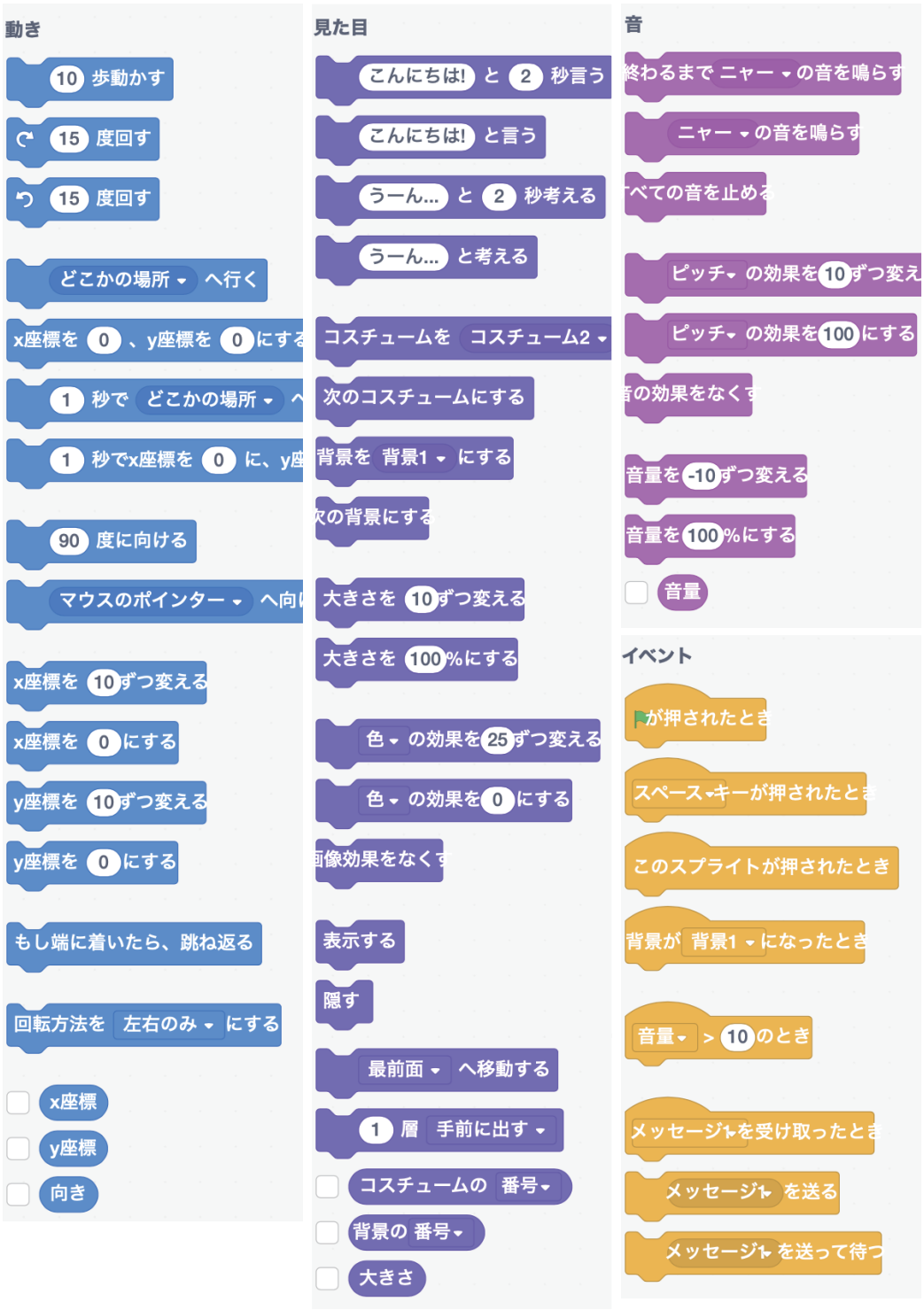


図 2.5: スプライト用の「動き」、「見た目」、「音」、「イベント」のブロックパレット

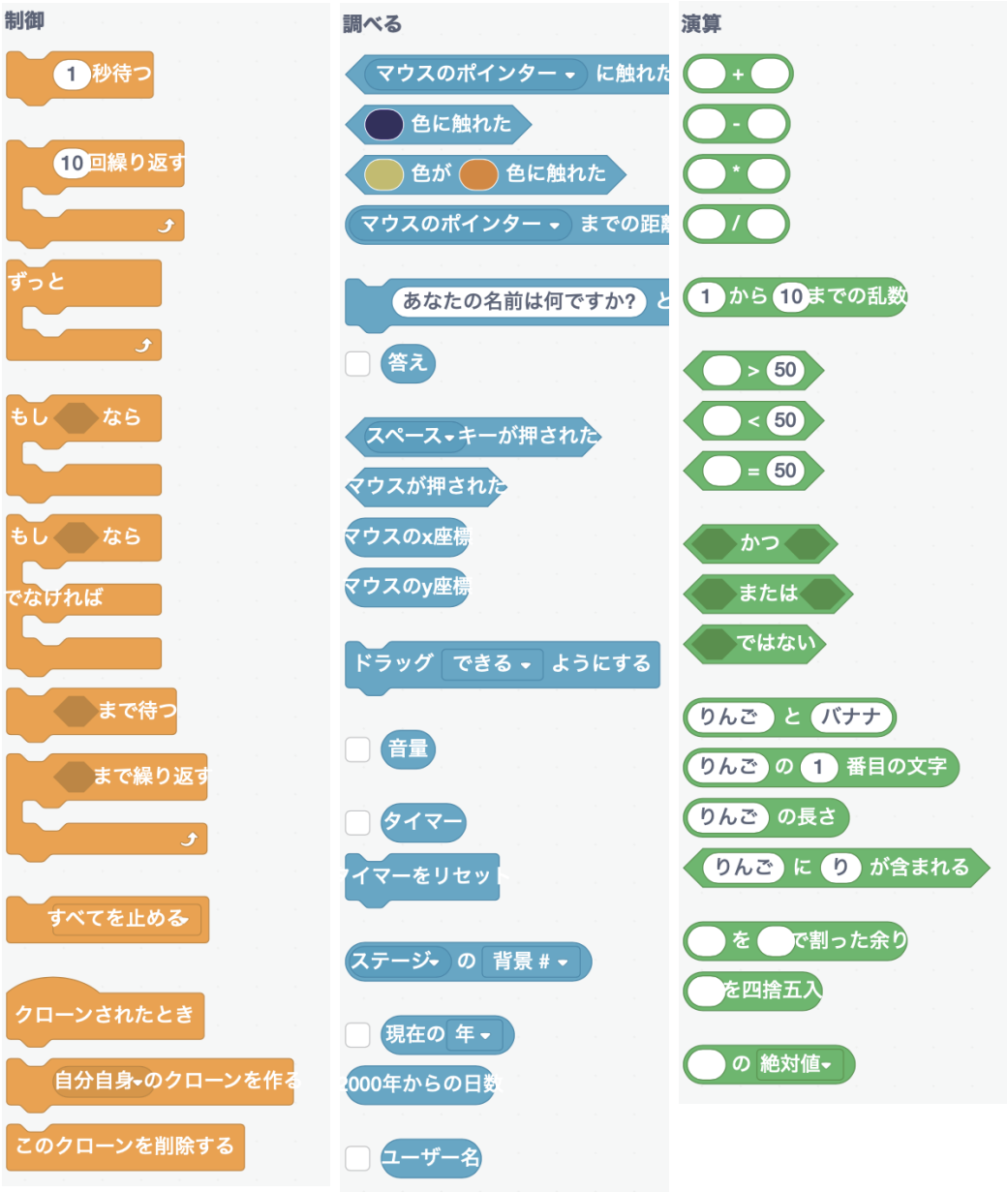


図 2.6: スプライト用の「制御」、「調べる」、「演算」のブロックパレット

図 2.7: スプライト用の「変数」、「ブロック定義」のブロックパレット：「リスト」を用いると、リストの操作に関するブロックが追加されます

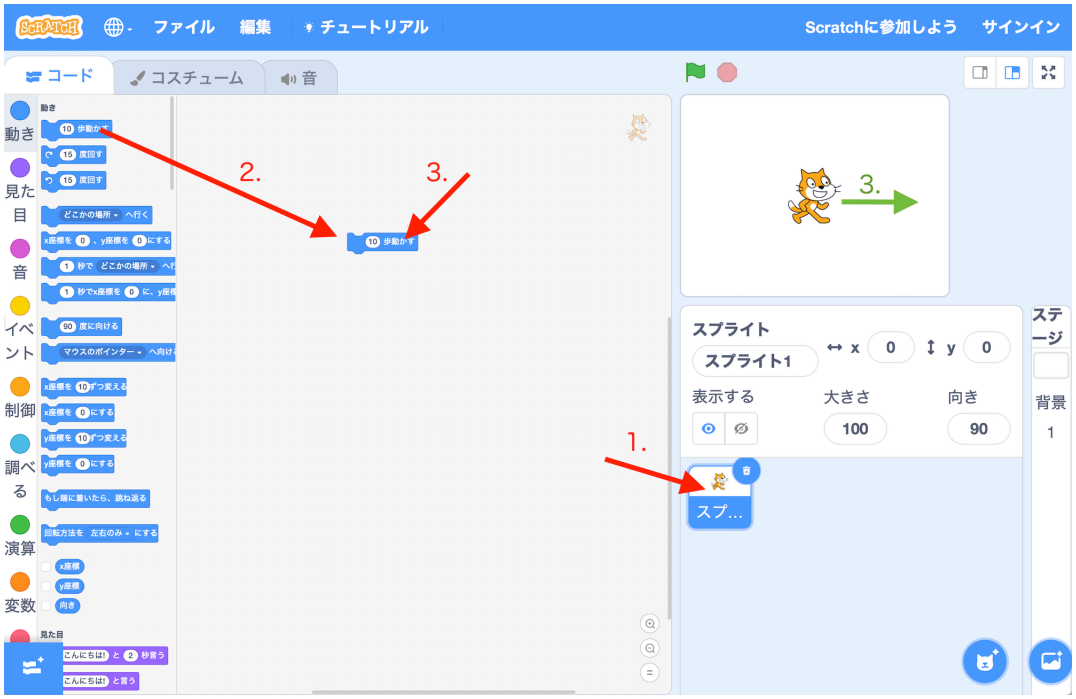


図 2.8: 猫が歩く

2.3　簡単なプログラムの編集と実行

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、様々なブロックをブロックパレットから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。

2.3.1　猫が少し歩く

図 2.8 を参考にしながら以下を試してください。

1. スプライトペインで猫（Scratch Cat という名前らしいです）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、実際にはクリックする必要はないかもしれません。

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ブロック

（以下、[(10) 歩動かす] ）をマウスでスクリプトエリアへドラッグします[6]。

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。

[6]「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) 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 2.9: 猫が少し歩いた後、方向を変える

以上が、最も単純な Scratch プログラムの実行です。

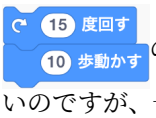
2.3.2　猫が少し歩いた後、方向を変える

プログラムを少し複雑にします。図 2.9 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ブロック

（以下、[(15) 度回す]）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 2.9 (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 2.9 (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。

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。

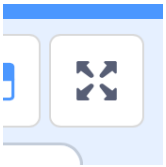
移動：　図 2.9 のように組み合わされた複数のブロックをまとめて移動させたいときには、**最上段**のブロックをマウスでポイントしてドラッグします。

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。

(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 2.10: 緑の旗をクリックすると...

(a) [ずっと] ブロックをドラッグ

(b) 図 2.9 (a) のプログラムを挟み込む

図 2.11: 無限ループを作ってみる

2.3.3　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台本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 2.10 (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 2.6 右列）から最上段のブロック

（以下、[緑の旗がクリックされたとき]）をドラッグし、図 2.10 (b) のようなプログラムを作ります。次に図 2.10 (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。

元のエディタ画面に戻したいときには右上の をクリックしてください。

2.3.4　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[7]。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

[7] 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。

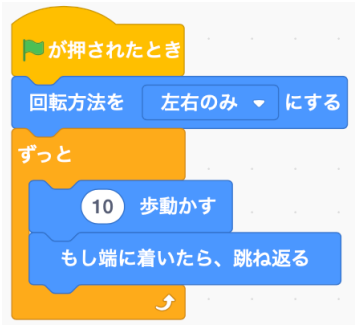


図 2.12: 猫が右往左往する

には強制終了ボタンが用意されており、それが図 2.10 (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 2.6 左列）の、上から 3 つ目のブロック

（以下、[ずっと]）です。まず、これを図 2.11 (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 2.9 (a) の 2 つのブロックを挟み込み、図 2.11 (b) のように完成させてください。図 2.11 (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。

2.3.5　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 2.12 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 2.12 のプログラムを動かしてみてください。

2.3.6　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。

　Scratch ブロックの形状は以下の 6 種類です[8]。

[8] ブロック形状の画像は日本語版 Scratch Wiki から転載しました。

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 2.5 の右列下の方）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 2.6 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 2.6 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 2.6 の中央列）、「演算」のブロックパレット（図 2.6 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 2.6 の中央列）、「演算」のブロックパレット（図 2.6 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。

ブロックの文章中での記述に関する注意：　文章中にブロックを

のように図示とすると、文章の流れが止まり、読みにくい場合が多々あります。既に上では

を [(10) 歩動かす] と言い換えたように、これ以降も混乱がない限り、ブロックを [〜]、⟨〜⟩、(〜) などの形で引用することがあります。

2.4　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。この講義テキストではそれを四つの原理に分けて解説します。

まず最初の原理は以下です。

第 1 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。

そのためにまず、変数の概念を解説します。

2.4.1　変数

コンピュータにはデータを記憶するメモリがあります。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） — には多くのプログラミング言語には一定の制限がありますが、Scratch の場合は任意の文字列も変数名として利用できます[9]。Scratch のプログラムでは多数の変数を使用できますが、この授業ではせいぜい 5 個程度の変数を用います。

Scratch では「変数」のブロックパレット（図 2.7 参照）から変数を利用できます。その準備として

1. まず、「変数」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す。そうすると、図 2.13 (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。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、「変数」のブロックパレットは図 2.13 (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 2.14 (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 2.14 (a) の青い囲みの中）から変数名を選んで行います。また、図 2.14 (b) のように、ステージの左上に変数名と現在の値が表示されます。ステージ上のこの表示は、ブロックパレット上の変数名の左隣の青いチェック（図 2.13 (b) を参照）を外すことで消えます。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。

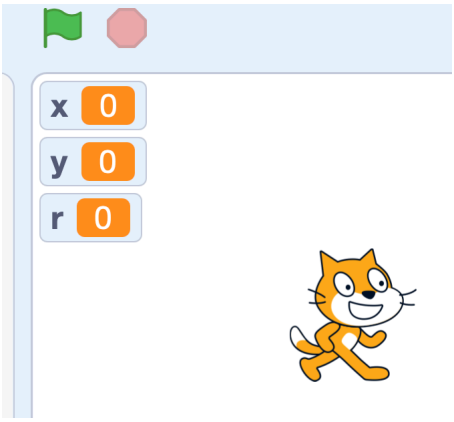
[9] 変数名を数字列（たとえば 777）にする、長大な名前を付けるなどは止めておいた方が無難です。多くのプログラミング言語では変数名は英文字の後に英文字または数字の並びが続くような名前に限定されています。

(a)「変数を作る」をクリックし、変数名 x を入力する

(b) 変数 x がブロックパレットで利用可能になる

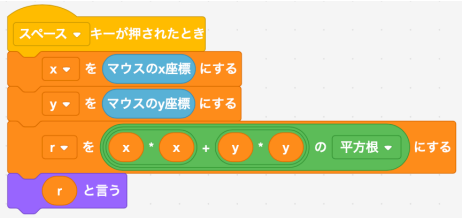
図 2.13: 変数 x の設定

(a) 変数 y、r を追加設定

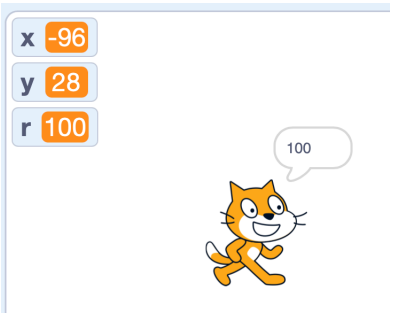


(b) 変数 x、y、r がステージに表示される

図 2.14: 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 2.15: 原点からの距離の計算

2.4.2　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。

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。

このプログラムの例が図 2.15 (a) です。

考察　図 2.15 (a) のプログラムを自ら組み上げ、実行しなさい。

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 2.15 (b) のように、猫が距離を答えてくれるはずです。

さて、図 2.15 (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します。これを **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、一連の動作を繰り返されることによって実行されており、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になっています。

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

単独の数値（たとえば図 2.15 (a) の「マウスの x 座標」や変数の値）はそれ自身で算術式です。演算子や関数を含む式（たとえば図 2.15 (a) のブロック ((((x)*(x))+((y)*(y))) の平方根)）も算術式です。Scratch で使用できる演算は、「演算」パレット（図 2.6 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。

(( )+( )) ... **加算** （addition）を表します。

(( )-( )) ... **減算** （subtraction）を表します。

(( )*( )) ... **乗算** （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乗算演算子は * で表します。Scratch もそれに従っています。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。

(( )/( )) ... **除算** （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 / 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[10]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。ただし、負数を用いた場合には演算結果に注意が必要です。

(( ) を四捨五入) ... 説明省略。ただし、負数を用いた場合には演算結果に注意が必要です。

(( ) の [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][11]、[log][12]、[e^][13]、[10^][14]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。

2.4.3　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 2.16 のようにアップロード（ファイルの読み込み）とダ

[10] 一様乱数とは、指定され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。
[11] ネイピア数 e を底とする自然対数関数
[12] 10 を底とする常用対数関数
[13] ネイピア数 e を底とする指数関数
[14] 10 を底とする指数関数

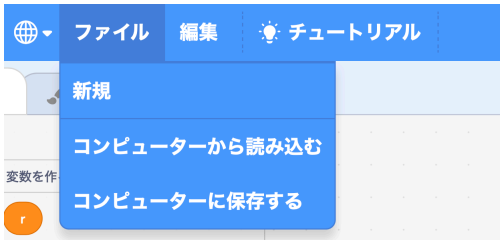
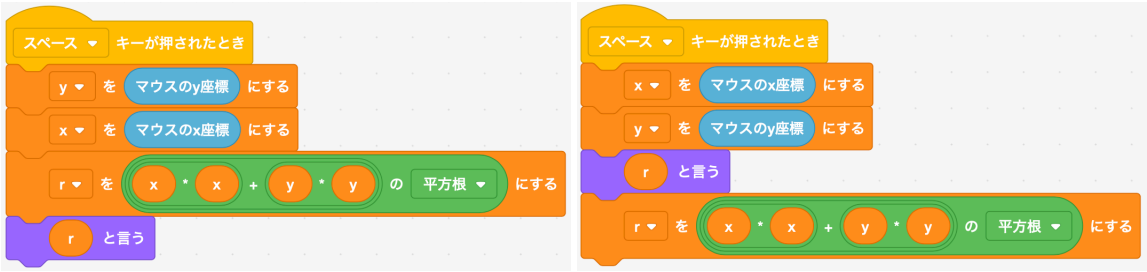


図 2.16: プログラムのアップロード/ダウンロード



(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 2.17: ブロックを入れ替えたプログラム

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。

　このアップロード／ダウンロード機能では Scratch のアカウントは不要です。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、ターボモード（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機能）を選択することも可能です。

2.4.4　似ているプログラム

　図 2.15 (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 2.15 (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 2.17 (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 2.17 (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 2.17 (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。

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。

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。

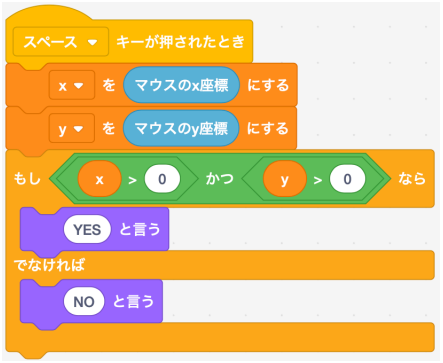
ヒント：スプライト（猫）がクリックされたことを感知するブロックは、「イベント」のブロックパレットの中の [このスプライトがクリックされたとき] ブロックです（図 2.5 の右列）。上のプログラム作成のポイントは、スプライト外の領域がクリックされた場合の処理をどうプログラム化するかという点です。

2.5　条件判定と実行の分岐

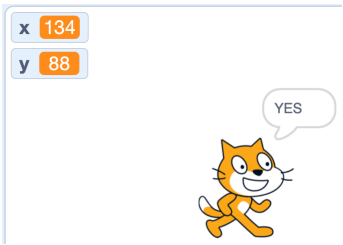
前節では、ブロックが順に実行されていくこと、および変数を利用できることを説明しました。これはプログラムの実行の大原則ですが、これだけでは複雑な処理はできません。そこで次の原理が必要となります。

第 2 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 には上の原理を実装する手段が用意されています。



(a) プログラム



(b) 実行結果

図 2.18: 第一象限の判定

2.5.1　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 軸上については適宜処理してよい。

このプログラムの一例が図 2.18 (a) です。このプログラムでは、「制御」のパレットのブロック [もし ⟨〜⟩ なら〜、でなければ〜]（図 2.6 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、

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、ブロック [YES と言う] を実行します。このとき、ブロック [NO と言う] 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] を実行します。[YES と言う] 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。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ブロック ⟨[〜] に触れた⟩ や ⟨ マウスが押された ⟩ 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には「以下」、「以上」を調べ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。

2.5.2　等価なプログラム

2.4.4 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。

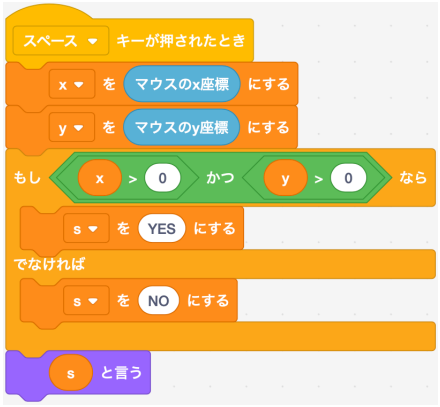
図 2.18 (a) の条件分岐を含むプログラムについても等価なプログラムは多数存在します。

まず、論理式を

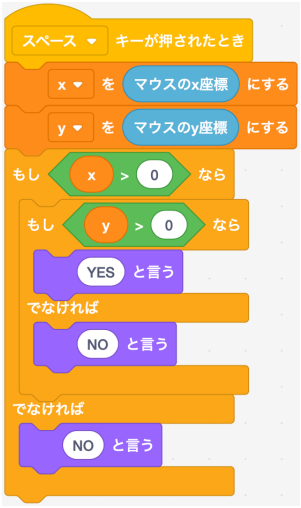
に置き換えても同じ真偽値と評価されるため、プログラムは等価です。言うまでもなく、論理式の置き換えには他にも様々なバリエーションがあります。

次に、図 2.19 (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 2.18 (a) と等価です。

図 2.19 (b) は、図 2.18 (a) とは異なるプログラム構造になっています。図 2.18 (a) では論理積ブロック ⟨⟨ $x > 0$⟩ 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 2.19 (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

図 2.19: 図 2.18 (a) のプログラムの様々な変形

図 2.20: 新しいスプライトを追加する（その 1）

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[15]。

2.6　犬のスプライト

ここまで猫のスプライトのみを用いてプログラムを作ってきましたが、Scratch では猫以外の様々な形状のスプライトが用意されています。その方法を解説します。

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

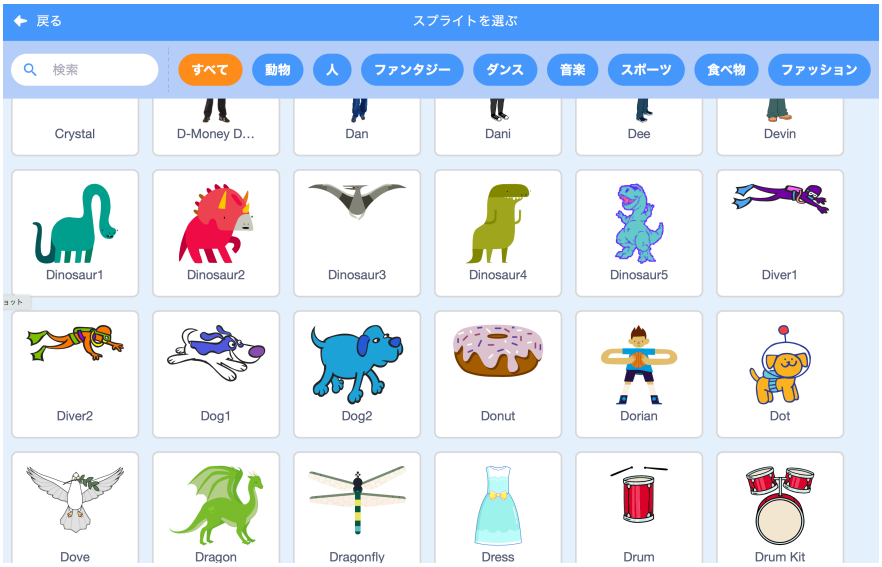


図 2.21: 新しいスプライトを追加する（その 2）

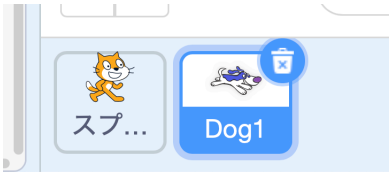


図 2.22: 新しいスプライトを追加する（その 3）

図 2.20 は、スプライトペインの部分です。赤矢印で指し示したボタンをクリックすると、図 2.21 のような様々なスプライトのリストが載ったウインドウが開きます。ここでたとえば中央からやや左下の Dog1 をクリックします。そうすると図 2.22 のように、スプライトペインに Dog1 が追加され、猫と同様にこの犬のスプライトが利用できるようになります。

次に、図 2.23 の、エディタ画面左上の矢印部分をクリックします。そうすると、図 2.24 の画面になります。これはスプライトの **コスチューム**（見た目）を種々、編集するための画面です。この画面左上の二つの緑の矢印の先を見てください。このスプライトには 2 種類のコスチュームが用意されています。

スプライトの見た目を変えるには、ブロックパレットの見た目の中のブロック [コスチュームを (...) にする]、または [次のコスチュームにする] を用います（図 2.5 の中央列の上の方）。

コスチュームは色や形を変えるなどの編集が可能です。たとえば図 2.24 の右上の赤い矢印をクリックすると、コスチュームが図 2.25 のように変化します。赤い矢印の先は画像を左右反転させるコマンドボタンです。この、左右反転した犬のスプライトは後に 5.1 節で用います。

図 2.23: コスチュームの編集（その 1）

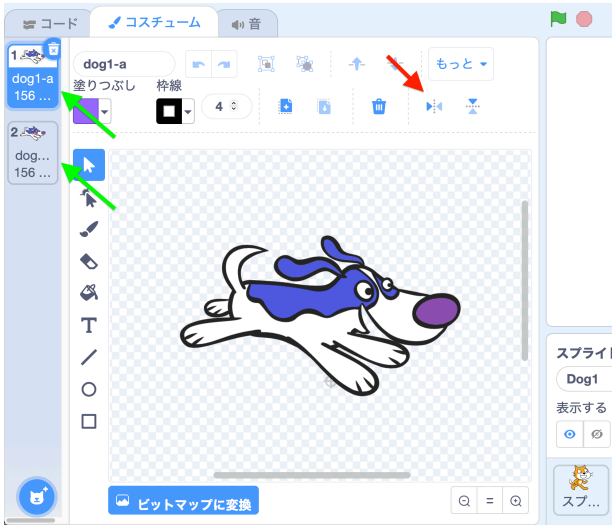


図 2.24: コスチュームの編集（その 2）

2.7　第 3 原理、第 4 原理

ここまでプログラム実行の第 1、第 2 原理を説明しましたが、残る二つの原理をここでまとめて紹介します。

第 3 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。

たとえば

は無限の繰り返しを可能にするブロックでした。それ以外にも制御ブロック（図 2.6 の左の列）には [(10) 回繰り返す] ブロックや [⟨ ⟩ まで繰り返す] ブロックなどが利用できます。この原理が無ければコンピュータの処理は分岐のみとなり、長大な処理は実現できません。

次の第 4 原理は第 3 原理と深く関連しています。

第 4 原理　メモリ上のデータの並びについて大量のデータを一括処理できる。

たとえば大学入試では多人数の受験生の成績を高速に処理せねばなりません。その場合、各受験生の成績をコンピュータのメモリに順に並べられておきます。そして各受験生について合計得点を計算する、あるいは全受験生の平均得点を計算するときには繰り返しを用いてデータ順に処理していきます。

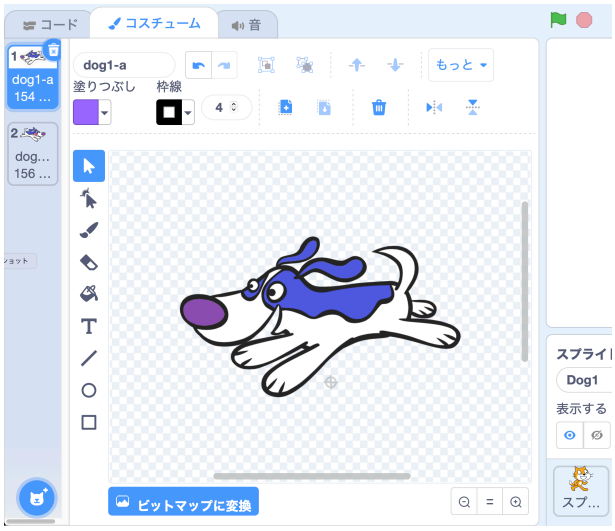


図 2.25: コスチュームの編集（その 3）

既に見てきたように第 3 原理までを用いるプログラムでも様々な処理が可能ですが、大量のデータを高速に繰り返し処理してこそコンピュータの本来の役割です。この講義では 13 章においてデータの並びを Scratch の **リスト** （list）を用いて取り扱います。

これら四つの原理は、コンピュータには何ができて何ができないのか、という問題と関連しています。最終章で述べますが、これら四つの原理が実装されているプログラミング言語では、コンピュータで理論上、可能な計算はすべてプログラム化できます。

1

1

第3章　オートマトンへの準備

プログラミングの話題は一旦、棚上げし、この章では今後必要となる基礎的な数学について解説します。

3.1　グラフの復習

この節の内容は次章において状態遷移図を理解するための準備です。

グラフ（graph）G は、有限個の **頂点**（vertex, node）の集合 V と、有限個の **枝**（edge, arc）の集合 E で定義されます。枝 e は二つの頂点 v, v' $(\in)$ を用いて $v \to v'$ と表します。v を枝 e の **始点**（start vertex）、v' を e の **終点**（end vertex）と呼びます。また v を v' の **先行頂点**（predecessor）、v' を v の **後続頂点**（successor）とも呼びます。たとえば以下はグラフの例です。

$$
\begin{aligned}
G_1 &= (V_1, E_1) \\
V_1 &= \{v_1, v_2, v_3, v_4\} \\
E_1 &= \{v_1 \to v_2,\ v_2 \to v_1,\ v_3 \to v_2,\ v_4 \to v_1\}
\end{aligned}
$$

このグラフを図示すると、図 3.1 の通りです。グラフの表現は図示による表現の方が直感的で一般的かもしれません。このテキストでもグラフはほとんどの場合に図示しますが、形式的な扱いを行う場合には上にような頂点と枝の表現を用います。

グラフ $G = (V, E)$ について、$v \to v' \in E$ ならば $v' \to v \in E$ であるとき、グラフは **無向グラフ**（undirected graph）であると言います。さもなくば **有向グラフ**（directed graph）と言います。このテキストで扱うほとんどのグラフは有向グラフです。

グラフ $G = (V, E)$ が枝 $v_1 \to v_2$, $v_2 \to v_3$, ..., $v_{k-1} \to v_k$ を持つとき、G には v_1 から v_k への **路、道**（path）があると言います。特に路の始点と終点が一致する（すわなち $v_1 = v_k$ となる）とき、その道（路）を **閉路、循環路**（closed path, cyclic path）と呼びます。

ある集合 L が存在し、グラフの頂点 $v \in V$ から $l \in L$ への写像 f が存在するとき、$f(v)$ を v の **ラベル**（label）と呼びます。同様に、グラフの枝 $e \in E$ から $l \in L$ への写像 g が存在するとき、$g(e)$ を e の **ラベル**（label）と呼びます。頂点や枝にラベルの付いたグラフを **ラベル付きグラフ**（labelled[1] graph）と呼びます。このテキストでは枝にラベルの付いたグラフが多数現れます。

[1] あるいは labeled。

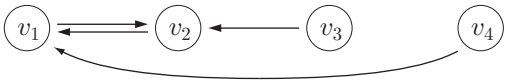


図 3.1: グラフの例

3.2　記号列

この節からは、形式言語理論に関する基本的な概念を定義します。

日本語のひらがな/カタカナ/漢字や英語のアルファベット 26 文字、数学における数記号（0〜9）のように、情報を表す不可分の基本単位としてしばしば **記号** （symbol）が用いられます。そして様々な情報は、ちょうど日本文がひらがな/カタカナ/漢字の並びで表されるように、あるいは英文がアルファベットの並びで表されるように、自然数が数記号の並びで表されるように、記号の並びとして表現されます。この記号の並びを一般に **記号列** （symbol string）あるいは単に **列**（string）と呼びます。記号列はこの講義で論じる内容の最も重要な操作対象です。

記号列を論じるときには、その記号列に含まれる記号の集合を、ある有限集合に固定して論じるのが一般的です。その記号の集合のことを **アルファベット** （alphabet）[2]と呼び、しばしばギリシャ文字 Σ を用いて表します。たとえばアルファベットを $\Sigma = \{a, b, c\}$ とするとき、$ababc$、$aaabbbccc$ などの、Σ に含まれる記号で構成される列をΣ の上の記号列と呼びます。

記号列 u について、$|u|$ は記号列の **長さ**（length）を表します。たとえば $|ababc| = 5$、$|aaabbbccc| = 9$ です。記号列の中には長さが 0 であるような記号列が存在します。そのような記号列を特に **空列**（empty string）と呼び、ギリシャ文字 ε で表します。もちろん $|\varepsilon| = 0$ が成り立つと考えます。

注意　空列を空集合と混同しないように注意してください。空列はれっきとした列です。空集合は列とは直接関係しません。

記号列 u の後に記号列 v を続けて書いた記号列 uv を u と v の **連接** （concatenation）と呼びます。$|u| = m$、$|v| = n$ ならば、$|uv| = m + n$ が成り立ちます。たとえば aaa の後に bbb を連接した記号列は $aaabbb$ です。

空列 ε は連接演算について単位元として振る舞います。すなわち、任意の記号列 u について

$$u\varepsilon = \varepsilon u = u$$

が成り立ちます。（これは加算の単位元 0、乗算の単位元 1 が

$$x + 0 = 0 + x = x, \quad x \times 1 = 1 \times x = x$$

[2]アルファベットと言えば英文字 26 文字を連想しがちですが、それはアルファベットの一例に過ぎません。

を満たすことと同じです。）単位元はその演算の数学的な性質を論じる際に様々な重要な役割を演じますが、空列もこの講義の様々な場面で重要な役割を演じることとなります。

同じ記号 a が n 回続けて現れるような列 $\underbrace{aaa\cdots aaa}_{n}$ は a^n と表すと約束します。ただし a^0 は空列とします。同様に、同じ記号列 u が n 回続けて現れるような列 $\underbrace{uuu\cdots uuu}_{n}$ は u^n と表します。これを再帰的に定義すれば、以下の通りです。

$$\begin{aligned} u^0 &= \varepsilon、\\ u^n &= uu^{n-1} \end{aligned}$$

必要ならば括弧を用いて $(u)^n$ と表してもよいとします[3]。

記号列 u、v、w について、$u = vw$ が成り立つとき、v は u の **接頭語** （prefix）であると言います。また w は u の **接尾語** （postfix）であると言います。たとえば $ababc$ の接頭語は、ε、a、ab、aba、$abab$、$ababc$ です。空列や記号列それ自身も接頭語であることに注意してください。接頭語の中から記号列それ自身を除いたものを **真の** （proper）接頭語と呼んで、記号列それ自身と区別する場合があります。真の接尾語も同様です。

注意　このテキストでは、誤解、曲解を避けるために、記号列を二重引用符 “ ” で囲み、たとえば “$ababc$”、“$aaabbbccc$” などと記述する場合があります。しかし、二重引用符の有無は記号列の内容に影響しません。$ababc$=“$ababc$” と考えてください。

3.3　言語

アルファベット Σ の上の記号列を要素とする集合を Σ の上の **形式言語** （formal language）または単に **言語** （language）と呼びます。たとえば $\Sigma = \{@\}$ とするとき、奇数個の@からなる列の集合:

$$L_1 = \{@^{2n+1} \mid n \geq 0\} = \{@, @@@, @@@@@, @@@@@@@, \cdots\}$$

は言語です。また素数個の@からなる列の集合:

$$L_2 = \{@^n \mid n \text{ は素数 }\} = \{@@, @@@, @@@@@, @@@@@@@, \cdots\}$$

も言語です。$\Sigma = \{0, 1, 2, \cdots, 8, 9\}$ とするとき、以下は 10 進数による素数の集合であって、やはり言語です。

$$L_3 = \{n \mid n \text{ は } 10 \text{ 進数の素数 }\} = \{2, 3, 5, 7, 11, 13, \cdots\}$$

空集合 $\emptyset$ も言語です。空列のみを要素とする集合 $\{\varepsilon\}$ も言語です。$\emptyset$ と $\{\varepsilon\}$ を混同しないように注意してください。

[3] このとき、括弧 () は記号列を構成する記号とは見なしません。

このように、形式言語とは特定の数学的条件を満たす記号列の集合のことです。そうすると、与えられた記号列 u が言語 L に属するか否かという判定問題 — これを **所属問題** （membership problem）とも呼びます — が興味深い問題として想起されてきます。

たとえば、記号列 @@ が上に示した言語 L_1 に所属するか否か（@@ の長さが奇数であるか否か）を知るにはどのようなアルゴリズムを用いればよいでしょうか。またそれは、@@ が言語 L_2 に所属するか否か（@@ の長さが素数であるか否か）という問題と比較して難しい問題でしょうか、あるいは易しい問題でしょうか。このテキストでは、このような所属問題をいくつかのカテゴリーに分類し、各カテゴリーの性質を明らかにしていきます。

所属問題は、一見、コンピュータとは無関係に思えるかもしれません。しかし、所属の是非を判定するプログラムをコンピュータに実装することを考えてみましょう。そのプログラムの複雑さは所属問題の性質によって分類できるのです。所属問題を知ることはコンピュータプログラムの複雑さを知ることに関係しており、プログラムの理論的な構造な性質について理解を深めてくれます。

ところで、これ以降の便利のために、Σ の上の全ての記号列の集合をこの分野の慣例に従い、Σ^* と表すと約束します。そうすると、Σ の上の任意の言語は Σ^* の部分集合ということになります。もちろん、Σ^* 自身も Σ^* の部分集合であり、Σ の上の言語です。

注意　「言語」というと、日本語、英語、中国語のような自然言語を想起する人がいるかもしれません。また、この授業では自然言語のコンピュータ処理（日英自動翻訳など）について学ぶものと期待する人もいるかもしれません。残念ながら、この授業で扱う「言語」はとても単純です。上に定義したように、記号列の集合のことをこの授業では言語と呼んでいます。この授業では、記号列の集合に関する所属問題を論じることが主なテーマであって、日英自動翻訳などについては全く触れません。というのも、自然言語のコンピュータ処理を行うには単に所属問題を解くだけでは足りず、構文論/意味論などの難解な問題に取り組まねばならないからです。

1

1

第4章　決定性有限状態オートマトン

この章では **決定性有限状態オートマトン**（deterministic finite state automaton）を解説します。このオートマトンには「決定性」、「有限状態」と冠されていますが、その大まかな意味は次の通りです。

「決定性」とはオートマトンの動作が全ての状況で一意に定まることを意味します。次々章では「非決定性」のオートマトンを学びます。

「有限状態」とは、オートマトンの **状態数** が有限であることを意味します。オートマトンは動作によってその状態を変えていきます。その状態を全て数え上げても高々有限個の状態しか持たないオートマトンを有限状態オートマトンと呼びます。世の中の多くの事象や機械は有限状態オートマトンと見なすことができます。一見すると有限状態と思えないような事象も、細部を捨象するならば有限状態と見なして問題のない場合が多々あります。このことから有限状態オートマトンは状態の遷移を伴う事象のモデル化に利用されています。特にオートマトン＝自動機械＝ロボットなどのモデル化に便利です。インターネットを飛び交うパケットのウイルスチェックなどにも利用されています。

4.1　例題：缶ジュースの自動販売機

この授業の前半で一貫して用いる例題をここで紹介します。自動機械の一種である自動販売機を考えます。

缶ジュースの自動販売機　ひなびた温泉旅館に缶ジュースの自動販売機が置かれているとします。この販売機は旧式のため、以下に述べる単純な仕組みで動作します。

1. 缶ジュースは 1 本 100 円です。
2. 販売機には、
   (a) 100 硬貨投入口、
   (b) 50 円硬貨投入口、
   (c) 返却ボタン、
   (d) 硬貨返却口、

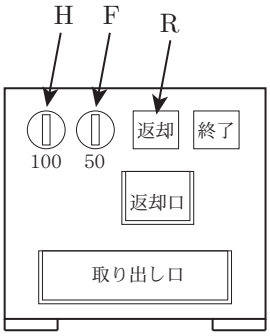


図 4.1: 缶ジュースの自動販売機

(e) 操作終了ボタン

が付いています（図 4.1 参照)。

3. 100 円硬貨、50 円硬貨を 100 円分以上投入した後に、操作終了ボタンを押すと缶ジュース 1 本が出てきて、その 1 回の販売処理が終了します。ただしお釣りは戻りません。お金の投入を間違えると利用者が一方的に損をしますが、文句は言えないことになっています。

4. 投入した硬貨総額が 100 円に満たないまま操作終了ボタンを押すと缶ジュースは出てこず、その回の販売処理がそのまま終了します。投入した硬貨は返却されません。この場合も利用者は損をしますが、やはり文句は言えないことになっています。

5. 操作終了ボタンを押す前に返却ボタンを押すと、それまでに投入した硬貨全てが返却口に戻り、それまでの投入金額が 0 円にリセットされます。幸いなことに、この場合は利用者は損をしません。

さて、利用者が行う自動販売機の操作を以下の三つの記号で表します。

H ... 100 円硬貨を投入する。

F ... 50 円硬貨を投入する。

R ... 返却ボタンを押す。

利用者が缶ジュースを 1 本得るためには、上の H、F、R の操作を何度か繰り返した後に操作終了ボタンを押すと想定されています。

さて、ここで問題です。

問題　操作終了ボタンを押したときに缶ジュースが自動販売機から出てくるような操作の列を全て示しなさい。ただし、利用者の損を関知する必要はありません。つまり、1000 円分の硬貨を投入

してお釣り 900 円が戻らなくとも気せず、そのような場合も缶ジュースが出てくる操作列と見なします。

たとえば HFRHHF はこの自動販売機の操作列の例です。この列は、

H 100 円硬貨を投入し、
F 50 円硬貨を投入し、
R 返却ボタンを押し、
H 100 円硬貨を投入し、
H 100 円硬貨を投入し、
F 50 円硬貨を投入する

操作の列を表しています。このとき、途中で硬貨が返却されますが、最終的には 250 円を投入していますから、ジュースは出てきます。

次に、操作列 HHFFRF ではジュースは出てきません。何故ならば、この列では、

H 100 円硬貨を投入し、
H 100 円硬貨を投入し、
F 50 円硬貨を投入し、
F 50 円硬貨を投入し、
R 返却ボタンを押し、
F 50 円硬貨を投入する

操作を行います。この場合、最終的には 50 円しか投入していませんから、ジュースは出てきません。

上の問題に答えるには、前者の HFRHHF のような、ジュースが出てくる操作列の性質を示せばよいのです。

次のように考えてみましょう。ジュースを得るには、操作を開始した後、返却ボタンを最後に押してから操作終了ボタンを押すまでの間に少なくとも 100 円硬貨 1 枚を投入するか、50 円硬貨 2 枚を投入する必要があります。このことから、ジュースを得るための操作列を以下の三つの操作列 u、v、w に分けて考えます。

1. u は H、F、R からなる任意の列、
2. v は H または FF、
3. w は H、F からなる任意の列

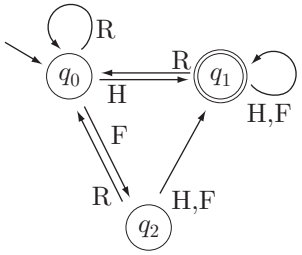


図 4.2: 缶ジュースの自動販売機の状態遷移図

ここに**「... からなる任意の列」と述べるときには空列も含む**ものと約束します。

列 u は 返却操作 R を含むため、ジュースを得るには無効かもしれない操作列を意味します。u は余分な硬貨を投入する操作列かもしれませんが、その余分な硬貨でジュースが出てくるとは確定できません。あるいは u は空列かもしれません。これに対して v はジュースを確実に得るための操作列です。w は余分な硬貨を投入する操作列です。w は空列かもしれません。

ジュースが出てくる操作列とは、u、v、w がこの順番に連結された列 uvw であれば必要十分です。よって、求める操作列の集合を L_J と表すとき、それは以下の通りです。

$$L_J = \{uvw \mid u \in \{\mathrm{H}, \mathrm{F}, \mathrm{R}\}^*,\ v \in \{\mathrm{H}, \mathrm{FF}\},\ w \in \{\mathrm{H}, \mathrm{F}\}^*\}$$

これが問題の答えです。

4.2　状態と遷移

引き続き、前節の缶ジュースの自動販売機を題材にします。次の問題を考えます。

問題　この自動販売機を有限状態オートマトンとして設計しなさい。

この問題の答えである有限状態オートマトンは **状態遷移図** （state transition graph、または state transition diagram）と呼ばれるグラフを用いて表すことができます。図 4.2 がそれです。

グラフの頂点 q_0、q_1、q_2 を **状態** （state）と呼びます。グラフの枝にはラベル H、F、R が付いていますが、これらはそのラベル（記号）を読んで状態を遷移することを表しています。3 個の状態の中で状態 q_1 のみ二重丸で表されていますが、このような二重丸で表される状態を **最終状態** （final state）と呼びます。図 4.2 には最終状態は 1 個だけです。必要ならば最終状態は複数個設けても構いません。オートマトンが動作を開始する際の状態を特に **初期状態** （initial state）と呼びます。このテキストでは、状態遷移図の中で矢印 ↘ で指された状態をひとつだけ設け、それを初期状態とします。図 4.2 では q_0 が初期状態です。

いま、ある人が列 HFRFHF の操作を行ったとします。このとき、オートマトンは与えられた操作列の記号を左からひとつずつ読みながら、初期状態から状態を変化（遷移）させていきます。以下は列 HFRFHF に対応する遷移の模式図です。

$$q_0 \xrightarrow{\mathrm{H}} q_1 \xrightarrow{\mathrm{F}} q_1 \xrightarrow{\mathrm{R}} q_0 \xrightarrow{\mathrm{F}} q_2 \xrightarrow{\mathrm{H}} q_1 \xrightarrow{\mathrm{F}} q_1$$

初期状態 q_0 から動作を始め、記号を読む毎に状態を遷移し、そして全ての記号を読み終えたところで動作を停止します。記号列を全て読み込んだときに、もしオートマトンが最終状態で停止したならば、読み込まれた記号列は **受理された** （accepted）と言います。上の動作例では状態 q_1 で動作を停止し、q_1 は最終状態ですから、記号列 HFRFHF は受理されました。この自動販売機の例では、受理は「自動販売機から缶ジュースが出てくる」ことに対応します。

逆に、もしオートマトンが最終状態で動作を停止しなかったならば、読み込まれた記号列は受理されなかった、あるいは拒否された（rejected）と言います。もうひとつの例題として、別の人が列 FFRFHRF の操作を行ったとします。このときのオートマトンの動作は以下の通りです。

$$q_0 \xrightarrow{\mathrm{F}} q_2 \xrightarrow{\mathrm{F}} q_1 \xrightarrow{\mathrm{R}} q_0 \xrightarrow{\mathrm{F}} q_2 \xrightarrow{\mathrm{H}} q_1 \xrightarrow{\mathrm{R}} q_0 \xrightarrow{\mathrm{F}} q_2$$

この場合、オートマトンの動作は状態 q_2 で終わっており、q_2 は最終状態ではないため、列 FFRFHRF は受理されません。「ジュースは出てこない」ということです。

さて、4.1 節の問題で求めた言語 L_J に含まれる記号列は図 4.2 のオートマトンで受理されることが確かめられます。逆に L_J に含まれない記号列は受理されないことも確かめられます。その意味で図 4.2 のオートマトンは言語 L_J の性質を表現したものになっています。

考察　缶ジュースの価格が 150 円の場合について、決定性有限状態オートマトンを求めなさい。

4.3　有限状態オートマトンの例

この節では様々な種類の有限状態オートマトンの例を挙げて、その特徴を解説します。ただし簡単のため、以下の全てのオートマトンについて $\Sigma = \{a, b\}$ とします。

例 1: Σ^*

受理言語が $\Sigma^* = \{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, ...\}$、すなわち Σ 上の全ての記号列の集合であるようなオートマトンの状態遷移図は図 4.3 (1) の通りです。これはどのような記号列を読んでも状態は常に最終状態 q_0 にいるような単純な構造のオートマトンです。

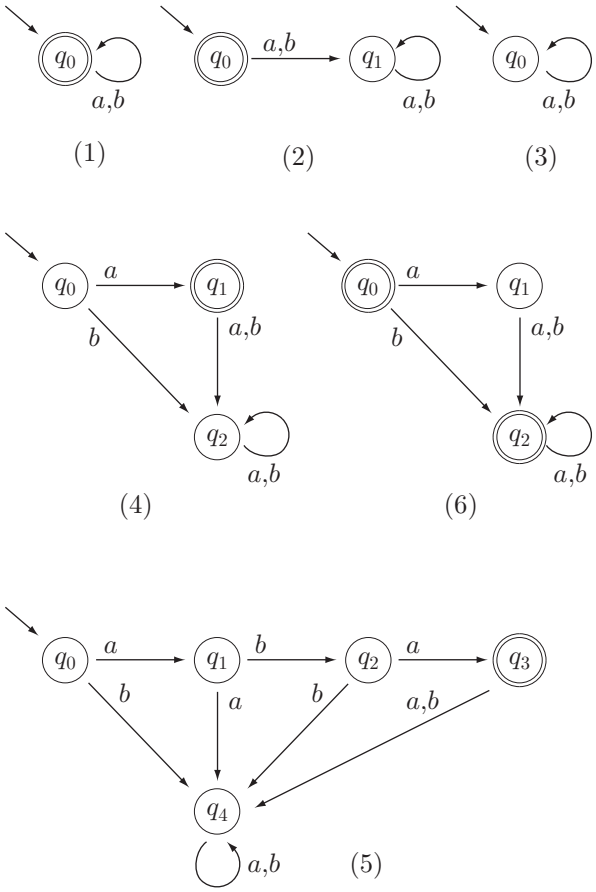


図 4.3: 様々な有限状態オートマトン（その 1）

例 2: $\{\varepsilon\}$

空列のみを受理するオートマトン（受理言語が空列のみからなる集合であるオートマトン）の状態遷移図は図 4.3 (2) の通りです。このオートマトンでは記号を全く読まないでオートマトンの実行が終了するときに空列を受理します。もし記号をひとつ読んだならば、状態は q_1 へ遷移します。q_1 ではどんな記号を読もうとも状態は遷移せず、q_1 のままです。q_1 は最終状態ではありませんから、記号列は受理されません。この q_1 のような状態を **死状態** （dead state）と呼びます。死状態は、記号列を受理しない機構をオートマトンに実現するためのテクニックのひとつです。

例 3: $\emptyset$

受理言語が空集合であるようなオートマトンの状態遷移図は図 4.3 (3) の通りです。このオートマトンは最終状態を持ちません。よって、どのような遷移を行っても記号列が受理されることはなく、受理言語は空です。このオートマトンの状態 q_0 も死状態であり、このオートマトンは死状態のみからなる特殊なオートマトンです。

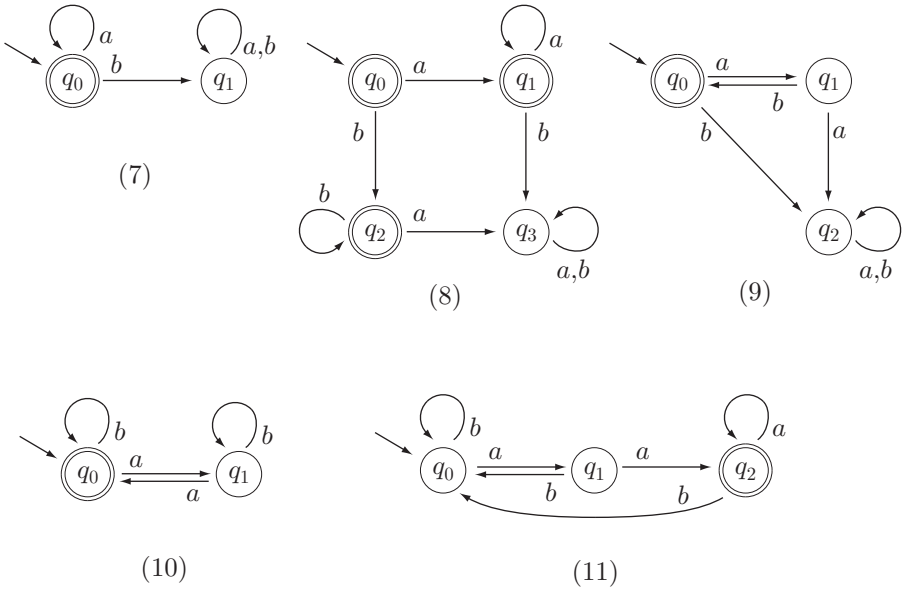


図 4.4: 様々な有限状態オートマトン（その 2）

例 4: $\{a\}$

長さが 1 の記号列 a のみを受理するオートマトンの状態遷移図は図 4.3 (4) の通りです。状態 q_2 が死状態であることに注意してください。

考察　受理言語が $\{a, b\}$ であるようなオートマトンを作りなさい。

例 5: $\{aba\}$

長さが 3 の記号列 aba のみが受理されるオートマトンの状態遷移図は図 4.3 (5) の通りです。状態 q_4 が死状態であることに注意してください。

例 6: $\Sigma^* - \{a\}$

受理言語が例 4 の **補集合** $\Sigma^* - \{a\}$ であるようなオートマトンの状態遷移図は図 4.3 (6) の通りです。このオートマトンの状態遷移図は例 4 のそれと全く同じ構造をしています。例 4 と異なるのは、最終状態が例 4 と例 6 では相補的になっている点です。つまり、例 4 で受理する記号列は例 6 では受理せず、例 4 で受理しない記号列は例 6 では受理するのです。一般に、ある受理言語の補集合を受理するオートマトンを作るには単に最終状態/非最終状態を反転させるだけです。実はオートマトン例 1 と例 3 もそのような関係になっています。

これ以降のオートマトンは状態遷移図が閉路を持つ場合です。

例 7: $\{a^n \mid n \geq 0\}$

0 個以上の a からなる列を受理するオートマトンの状態遷移図は図 4.4 (7) の通りです。受理する記号列が a^n（a が n 回繰り返す、n は 0 以上の任意の数）のような構造を持っているときには状態遷移図には **閉路** が存在します。図 4.4 (7) の場合は $q_0 \to q_0$ がその閉路です。なお、状態 q_1 は死状態です。

考察　受理言語が $\{a^n \mid n \geq 1\}$ であるようなオートマトンを作りなさい。この受理言語には空列 ε が含まれないことに注意しなさい。

例 8: $\{a^n \mid n \geq 0\} \cup \{b^n \mid n \geq 0\}$

0 個以上の a からなる列、または0 個以上の b からなる列を受理するオートマトンの状態遷移図は図 4.4 (8) の通りです。状態 q_1、q_2 はそれぞれ a^n、b^n（$n \geq 1$）を受理するための状態です。空列は初期状態 q_0 で受理されます。q_3 は死状態です。

例 9: $\{(ab)^n \mid n \geq 0\}$

a と b が交互に現れる列を受理するオートマトンの状態遷移図は図 4.4 (9) の通りです。このオートマトンでは、閉路 $q_0 \to q_1 \to q_0$ の遷移を繰り返すことで記号列 $(ab)^n$（$n \geq 0$）を受理します。

例 10: $\{u \in \Sigma^* \mid u$ に現れる a の数が偶数 $\}$

記号 a と b の出現順序は問わず、記号列の中に現れる a の数が偶数（0 は偶数とします）であるような列、たとえば ε、bbb、aab、$abba$、... を受理するオートマトンの状態遷移図は図 4.4 (10) の通りです。状態 q_0 は a を偶数個読んだ状態、q_1 は奇数個読んだ状態です。記号 b を読むことは受理と関係しないため、b を読んでも状態が変化しません。

考察　集合 $\{u \in \Sigma^* \mid u$ に現れる a の数が偶数かつ u に現れる b の数が偶数 $\}$ を受理集合とするようなオートマトンを作りなさい。

例 11: $\{uaa \mid u \in \Sigma^*\}$

u を Σ の上の任意の列とするとき、uaa は aa で終わる列を意味します。そのような列を受理するオートマトンの状態遷移図は図 4.4 (11) の通りです。q_0 は直前に読んだ記号が a ではなかった状態、q_1 は 2 個前の記号は a ではなく、直前の記号が a であった状態、q_2 は 2 個前と直前に読んだ記号が共に a であった状態を表します。途中で b を読んだ場合には、現在の状態に依らず、q_0 へ遷移します。この「q_0 へ遷移する」動作はある種の **リセット** 機構と考えることができます。

リセットボタンが押されたときに機械がそのときの状態に関わらず初期状態へ戻るという仕組みです。同様のリセット機構は、4.1 節の自動販売機の返却ボタンの動作、言い換えれば、図 4.2 の自動販売機のオートマトンの「記号 R を読んだときには、必ず q_0 へ遷移する」動作にも見ることができます。リセット機構はオートマトンを設計する上でのテクニックのひとつです。

考察　集合 $\{uaav \mid u, v \in \Sigma^*\}$ を受理集合とするようなオートマトンを作りなさい。すなわち、aa という部分列を含む列、たとえば aa、baa、$abaa$、$abbaabbb$ などを受理するオートマトンを作りなさい。

例 12: 有限状態オートマトンで表わすことができない言語

実は有限状態オートマトンの表現力は非常に限定的で、有限状態オートマトンでは決して表わすことができない言語が無限に存在します。

典型的な例として言語 $\{a^n b^n \mid n \geq 0\}$ を表すような有限状態オートマトンは存在しません。何故ならば、この言語に含まれる記号列では a の数と b の数が同じでなければなりません。数が同じか否か判断するには、a が現れた個数を一旦どこかに記憶しておき、その後に b の個数と比較せねばなりません。しかし、有限状態オートマトンは個数を記憶する装置を持たず、状態を記憶の代替にするしか方法がありません。その状態数は有限であるため、n が限りなく大きくなる場合に対応できません。逆に言えば以下の考察のオートマトンは作ることができます。

考察　適当な正数 N （たとえば $N = 2, 3, 4, ...$）について、集合 $\{a^n b^n \mid N \geq n \geq 0\}$ を受理集合とするようなオートマトンを作りなさい。そのオートマトンの状態数が $2N$ 程度であることを確認しなさい。

もし N という上限を設けないならば、状態数は無限に必要です。$\{a^n b^n \mid n \geq 0\}$ を受理する無限状態オートマトンは後章で学びます。

4.4　練習問題

1. $\Sigma = \{a, b\}$ として、以下の言語を受理する決定性有限状態オートマトン（の状態遷移図）をそれぞれ示しなさい。

   (a) 先頭と末尾にのみ b が現れる記号列の全体（bb, bab, $baaaab$ など）

   (b) 記号列 bb を含み、その他の箇所に b は現れない記号列の全体（bb, abb, $aaabba$ など）

   (c) 2 個の b を含む記号列の全体（b は連続して現れる必要はない。bb, $abab$, $aabaaabaa$, など）

(d) 記号列 bb を含む記号列の全体（b が 3 個以上現れてもよい。bb, $abbb$, $abaabbab$ など）

(e) a が偶数個（0 個を含む）、b が 3 の倍数個（0 個を含む）現れる記号列の全体（ε, aa, bbb, $ababb$, $abababa$ など）

(f) その記号列中の、長さが 3 の任意の部分記号列中に a が高々1 個しか現れないような記号列の全体（abb, $bbbabbbbabbabb$ など。$babbaba$ は末尾の aba に 2 個の a を含むから受理しない。）

1

1

第5章　決定性有限状態オートマトンのプログラミング

この章では前章のオートマトンを Scratch を用いてプログラム化します。Scratch の様々な機能を用いればそれはそう難しいことではありません。ポイントは、どんな有限状態オートマトンの状態遷移図も全く機械的にプログラム化できる点です。

この章で後ほど作るプログラムの実行の様子を図 5.1 に示す。これは図 4.1 の自動販売機を簡略化したもので、[H]、[F]、[R]、(end)、猫、グラスにはそれぞれスプライトを用いました。[H]、[F] は 100 円投入、50 円投入を表すボタンです。[R] は返却ボタンです。(end) は入力終了ボタンです。オートマトンの制御部は猫に担当させます。グラスはもし入力が受理されるならばジュースの注がれた表示となり（図 4.1 はジュースが入っているつもり）、不受理ならば空（から）の表示となります。

5.1　並行プログラム

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。ここではその簡単な例を紹介します。

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[1]。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実行が可能になります。この講義テキストでは Scratch の機能のひとつである **メッセージパッシング** （message passing）を用いてそのやり取りを実装します。

次のような問題を考えましょう。

猫と犬の掛け合い

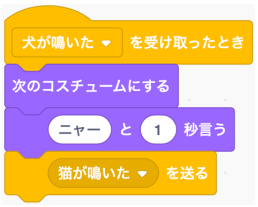
緑の旗をクリックすると、猫が「ニャー」と鳴き、次に犬が「ワン」と鳴き、次に猫が「ニャー」と鳴き、... という動作を 1 秒間隔で順に繰り返すプログラムを、猫と犬をそれぞれスプライトとして作りなさい。

ポイントは、猫と犬が順に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 2.6 の右列）の下の方のブロック

（以下、

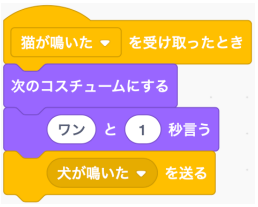
[1] 類似の術語に「並列実行」（parallel execution）があります。これら術語は区別されて使用されますが、その違いについてはこの授業の範囲を超えるので省略します。

図 5.1: 自動販売機の実行の様子

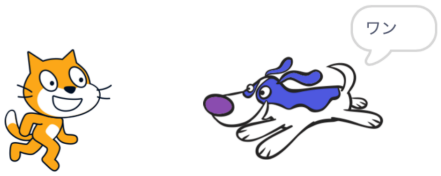


(a) 猫のプログラム

(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 5.2: メッセージパッシングによる猫と犬の掛け合い

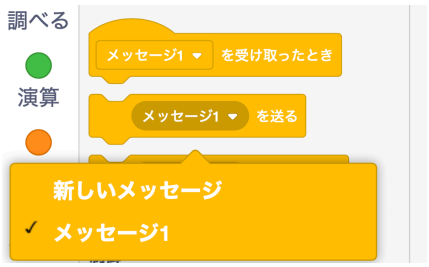
[[· · ·] を受け取ったとき]) と

（以下、[(· · ·) を送る]）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、メッセージはユーザが自由に新規登録することができます。新規登録はブロックの [· · ·] の中の「メッセージ 1▼」の▼をマウスでクリックして行います。そうすると、図 5.3 (a) のように、新しいメッセージを作るか、既存のメッセージ（この場合は「メッセージ 1」）を用いるか選択します。新しいメッセージを作る方を選んだ場合には図 5.3 (b) のようなダイアログウインドウが開きますから、新規のメッセージ名を入力すれば、これ以降、そのメッセージが利用可能となります。

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 5.2 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、



(a) 猫のプログラム

(c) 犬のプログラム

図 5.3: 新しいメッセージを作る方法

図 5.4: プログラム起動のための犬のプログラム

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、

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 5.2 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。それは犬に任せることにしましょう。図 5.4 のプログラムを追加します。これでプログラムは完成です。緑の旗がクリックされた後の実行は以下の通りです。

1. 緑の旗がクリックされると、図 5.4 のプログラムが実行されます。

2. 犬が鳴いたならば、図 5.2 (a) のプログラムが実行されます。

3. 猫が鳴いたならば、図 5.2 (c) のプログラムが実行されます。

4. 犬が鳴いたならば... 以下同様です。

上の実行は、 がクリックされるまで繰り返されます。

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 5.5 に示します。

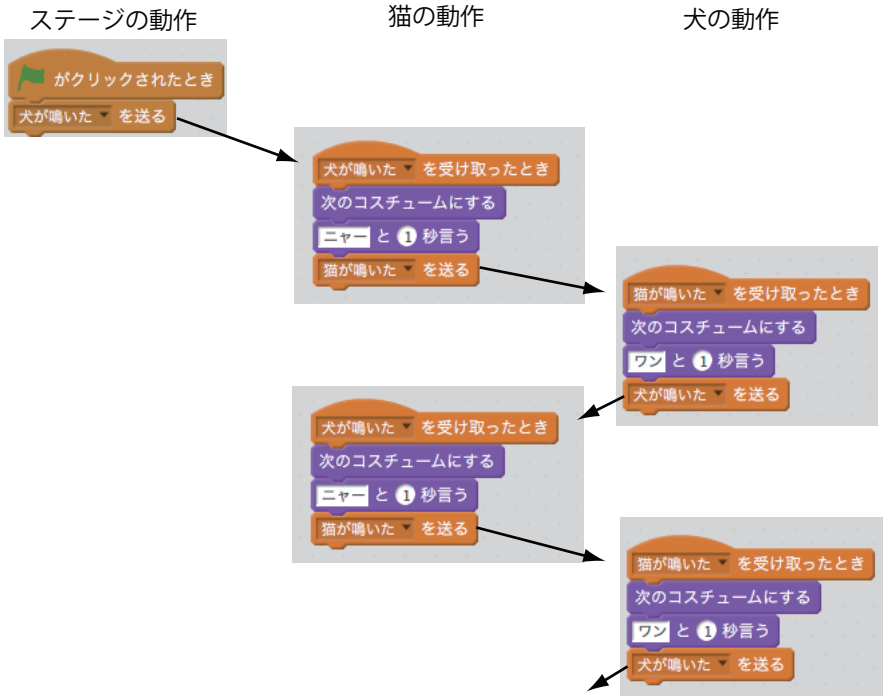


図 5.5: 猫と犬の掛け合いのタイミングチャート（矢印がメッセージパッシング）：ブロックの色合いが他所と違っていますが、ドローイングツールでの編集のせいです（詳しい原因は究明中）。

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。通常、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できないため、受信側プログラムでは送信元は不明です。2. については全員（自分を含めた全スプライト）です。メッセージは全てのスプライトに対して送信されます。これを**ブロードキャスト通信**（日本語訳では放送）とも言います。もしスプライトがそのメッセージについてブロック [[···] を受け取ったとき] 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。Scratch では 4. も付加できない仕様です。よってあまり複雑な送受信はできませんが、この章のオートマトンのプログラミングには Scratch の単純なメッセージで問題ありません。

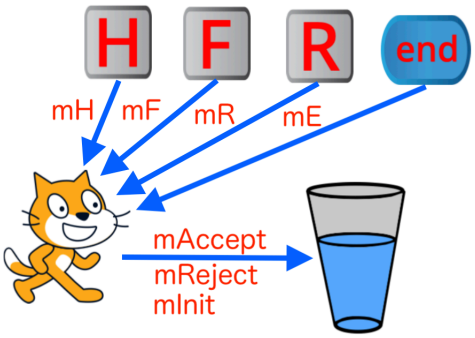


図 5.6: メッセージの送受信の流れ：赤字がメッセージ名

5.2　自動販売機のプログラミング

この節以降、メッセージパッシングを利用して決定性有限状態オートマトンをプログラミングします。まず考えるのはジュースの自動販売機です。その実行中の様子は図 5.1 です。

5.2.1　全体の構造

図 5.6 がこれから作るオートマトンのプログラムにおけるメッセージの流れです。赤字の mH、mF、mR、mE、mAccept、mReject、mInit が新規に用いるメッセージです。

1. 自動販売機を使うユーザは上部の四つのボタンを押します。
2. 押されたボタンは、自身が押されたことをオートマトン（猫）にメッセージ伝達します。
3. オートマトンはそのメッセージに基づいて動作します（状態遷移図での動作を思い出しましょう）。
4. オートマトンは最終的に受理（accept）、不受理（reject）の情報をグラスにメッセージ伝達します。
5. グラスは受け取った情報の種類に応じて適切な表示を行います。

以下では上の方針に基づいて各スプライトを具体的に設計していきます。

5.2.2　四つのボタンのプログラム

図 5.7 は四つのボタン（図 5.6 上部の [H]、[F]、[R]、(end) のスプライト）のそれぞれのプログラムです。スプライトがマウスでクリックされた（押された）ならば、それぞれのメッセージを全

(a) スプライト [H] のプログラム

(b) スプライト [F] のプログラム

(c) スプライト [R] のプログラム

(d) スプライト (end) のプログラム

図 5.7: 四つのボタンのプログラム

図 5.8: オートマトンの状態を保持する変数 state をローカル変数として作成

スプライトに向けてブロードキャストします。しかし、これら四つのボタンはメッセージを送信するもののメッセージを受信するプログラムを持たないため、四つのボタンは発せられたメッセージに反応しません。

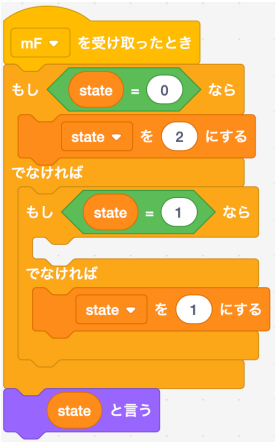
四つのボタンは単に「ボタンが押されたらメッセージをブロードキャストする」だけの機能しか持ちません。

5.2.3　猫のオートマトン制御部

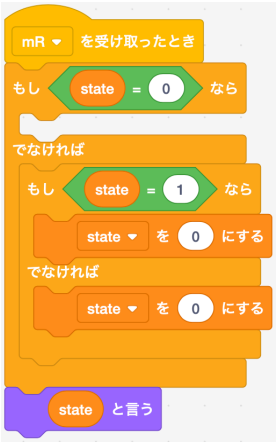
オートマトン（猫）はメッセージを受け取ったときに状態を遷移します。そのためにまず、オートマトンに状態を導入する必要があります。それを Scratch の変数で実装します。その変数を作る作業は図 5.8 の通りです。同様の図は、図 2.13 (a) ですでに紹介済みですが、両者の違いは図 5.8 では **ローカル変数** （local variable）を用いている点です。これに対して図 2.13 (a) で作成される変数を **グローバル変数** （global variable）と呼びます。それぞれの内容は以下の通りです。

グローバル変数 ... ひとつの Scratch のプログラムにこの名前の変数はひとつだけ存在し、全ての

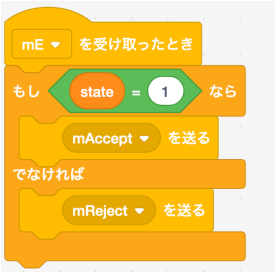
(a) メッセージ mH を受信したとき



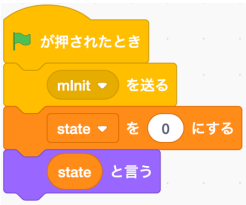
(b) メッセージ mF を受信したとき



(c) メッセージ mR を受信したとき



(d) メッセージ mE を受信したとき



(e) 緑の旗がクリックされたとき

図 5.9: それぞれのメッセージに反応するオートマトンのプログラム

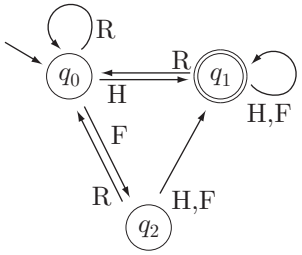


図 5.10: 缶ジュースの自動販売機の状態遷移図（再掲）

スプライトからその変数名でアクセスできます。

ローカル変数 ... この変数を定義するスプライト内にひとつだけ存在し、そのスプライト内ではグローバル変数と同じようにその変数名でアクセスできます。他のスプライトからもアクセスできますが、その場合には「どのスプライトのどの変数か」を指定する必要があります。

クラウド変数 ... これについては省略します。

オートマトンがプログラム内に唯一であるとき、その状態を表す変数はグローバル、ローカルのどちらでも問題ありません。しかし、後章ではひとつのプログラム内で同時に複数のオートマトンを実行させることとなるため、ここではその変数がオートマトン（猫）の中に存在するように、ローカル変数を用いることとします。

状態を表す変数をここでは state と名付けます。その state を用い、メッセージ毎に状態遷移を実装したものが図 5.9 です。

まず、図 5.9 (a) は [H] のボタンが押され、メッセージ mH がオートマトンに受信されたときの動作プログラムです。参考のため、図 5.10 にこのオートマトンの状態遷移図を再掲します。プログラムを説明すると以下の通りです。

1. もし メッセージ mH を受信したならば
   (a) もし state が 0 ならば、state を 1 に変える。
   (b) さもなくば
      i. もし state が 1 ならば、何もしない（state は 1 のまま）。
      ii. さもなくば（state は 2 のはず）、state を 1 に変える。
   (c) state を表示する。

最後の「state を表示する」は不要ですが、プログラムのデバッグ（プログラムの誤りを見つける作業）のために付けました。

メッセージ mF、mR を受信した場合も同様です。

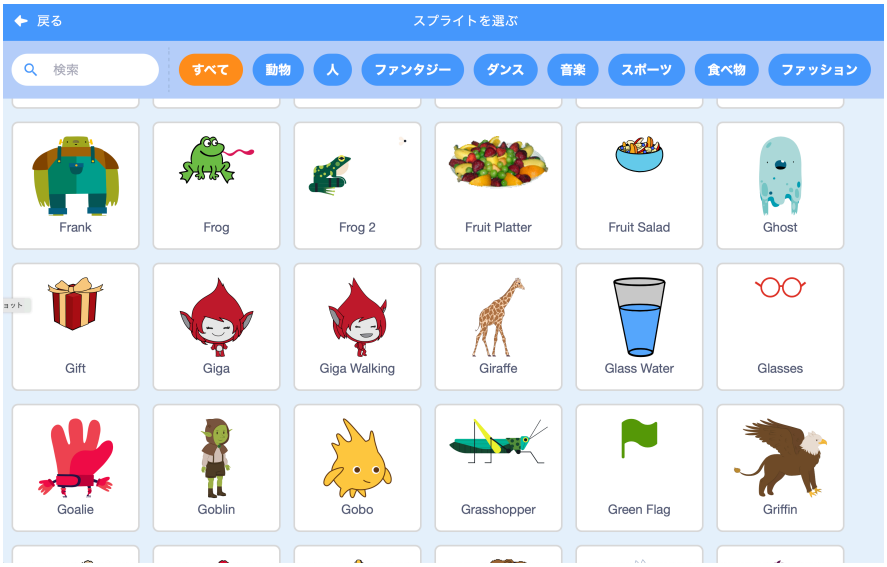
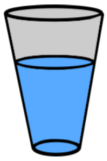


図 5.11: グラスのスプライトが表示されているスプライトリスト



(a) 受理のときの表示コスチューム

(b) 不受理のときの表示コスチューム

図 5.12: グラスの 2 種類のコスチューム

メッセージ mE を受信した場合の動作は上記とはやや異なります。mE が受信されたときには終了処理を行います。図 5.9 (d) のように、state が 1 ならば、状態 1 は最終状態であるため、ジュースを出す動作を行います。それ以外ならば、ジュースを出さない動作を行います。ジュースを出す/出さないはオートマトンの仕事ではなく、プログラムの中の結果表示部の仕事です。ここではそれはグラス（ジュースを注ぐ器）のスプライトに任せます。よって受理動作（ジュースを出す動作）ではメッセージ mAccept をブロードキャストし、不受理動作（ジュースを出さない動作）ではメッセージ mReject をブロードキャストします。

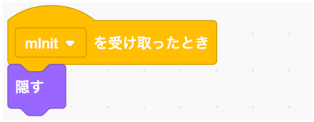
プログラムの初期化はオートマトンが行うことといます。図 5.9 (e) がその処理です。メッセージ mInit をブロードキャストし、state を初期状態番号 0 にし、state を表示します。

5.2.4　グラスのプログラム

グラスのスプライトは、Scratch に用意してあるスプライトリスト（図 5.11）の “Glass water” を用いることとし、受理される場合には図 5.12 (a) を用い、受理されない場合には図 5.12 (b) を

(a) メッセージ mAccept を受信したとき

(b) メッセージ mReject を受信したとき



(c) メッセージ mInit を受信したとき

図 5.13: グラスのプログラム

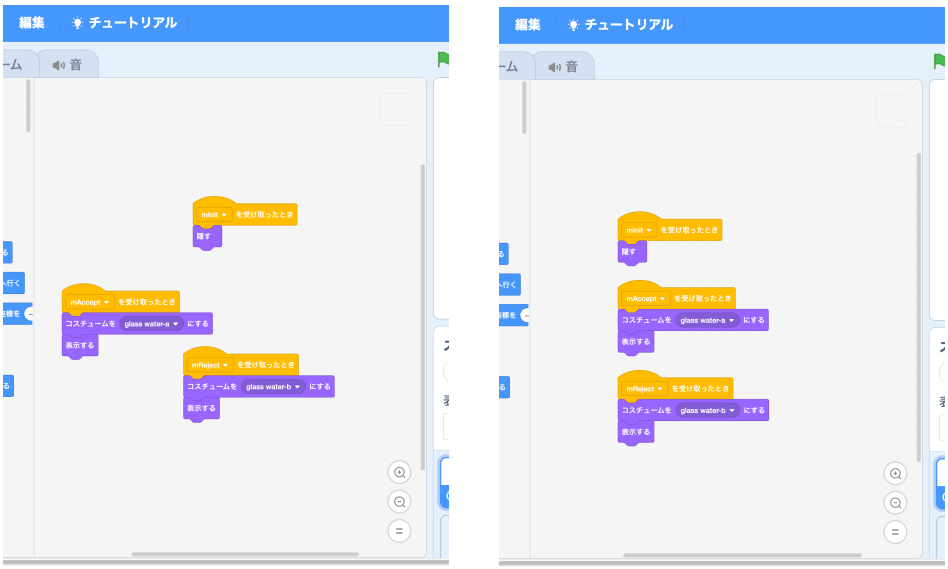
図 5.14: スクリプトエリア内で右ボタンを押したとき

用います。なお、図 5.12 (b) の “sorry” は “Glass water” のコスチュームに後付けしました。

　このスプライトのプログラムは図 5.13 の通りです。メッセージ mAccept を受信したときには “galss water-a” のコスチュームを、mReject を受信したときには “galss water-b” のコスチュームを表示します。初期化メッセージ mInit を受信したときにはスプライトをステージから隠します。

　以上で自動販売機のプログラムは完成です。

プログラムの見栄えをよくする機能：　Scratch のプログラムは、スクリプトエリア内のどこに置いても構いません。そのため、プログラムを作成する中で完成したプログラム、作りかけのプログラムがエリア内に無秩序に置かれた状態になることがよくあります。Scratch のオンラインエディタにはそれを整頓する便利な機能が用意されています。マウスカーソルがスクリプトエリア内にある状態で、マウスの右ボタンを教えてください。そうすると、図 5.14 のようなメニューが現れます。その中の「きれいにする」がその機能です。たとえばスクリプトエリアが図 5.15 (a) の状態のときに「きれいにする」を実行するとエリア内が図 5.15 (b) のように整頓されます。



(a) スクリプトエリア内に散乱するプログラム群　(b) スクリプトエリア内に整頓されたプログラム群

図 5.15: プログラムの見栄えをよくする

5.3　猫のオートマトン制御部のもうひとつの実装

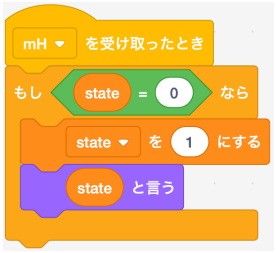
図 5.9 は、[もし ⟨ ⟩ なら〜、でなければ] のブロックを入れ子にして使い、少し読みにくいプログラムになっています。

これを図 5.16 のように変えると、独立したプログラムの個数は増えますが、それぞれは同形の単純なものになります。たとえば同図 (a) の意味は「メッセージ mH を受信したとき、もし state が 0 ならば state を 1 に変更し、state を表示する」となります。メッセージ mH を受信したときに (a) と (b) の二つのプログラムが動き出しますから処理が混乱しそうですが、条件式が異なりますから、実際にはどちらかひとつの処理のみが有効となり、混乱はなさそうです。

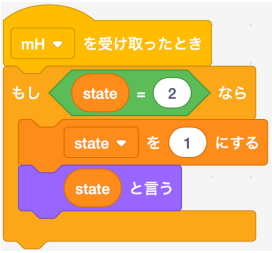
このプログラムは正しく動作するでしょうか。残念ながら、正しく動作しない場合があります。

考察　図 5.16 のプログラムが正しく動作しないかもしれない状況を示しなさい。

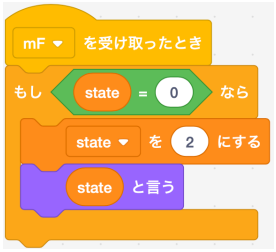
上のような難しい動作があるため、並行プログラムの開発はそう簡単ではありません。



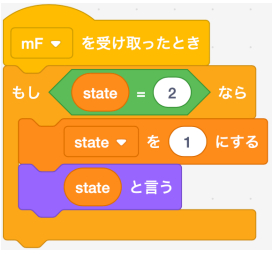
(a) メッセージ mH を受信したとき



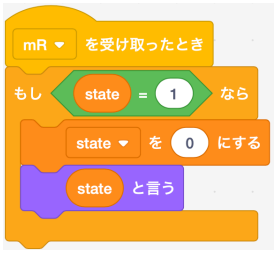
(b) メッセージ mH を受信したとき



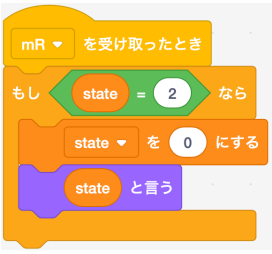
(c) メッセージ mF を受信したとき



(d) メッセージ mF を受信したとき

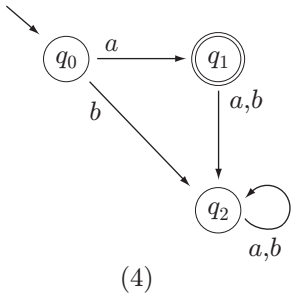


(e) メッセージ mR を受信したとき



(f) メッセージ mR を受信したとき

図 5.16: それぞれのメッセージに反応するオートマトンのプログラム：mE を受信したとき、緑の旗がクリックされたときのプログラム部分は図 5.9 (d)、(e) と同じのため、省略



(4)

図 5.17: 図 4.3 (4) の再掲

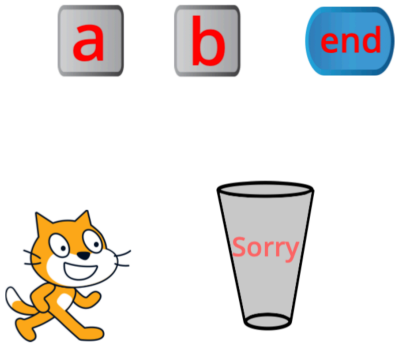


図 5.18: 自動販売機の実行の様子

5.4　その他のオートマトンのプログラミング例

自動販売機の例題プログラムはそのまま任意の有限状態オートマトンのプログラムに借用できます。ここでは、図 4.3 (4) のオートマトン（図 5.17 に再掲）を例として解説します。

まず、入力記号は a と b の二つですから、自動販売機のスプライト [H]、[F] をそのまま [a]、[b] に模様替えします。まず、コスチューム（スプライトの表示）を変え、メッセージ名を mA、mB に変えます。スプライト [R] は使用しないため削除します。スプライト (end) は自動販売機と全く同じとします。受理/不受理の場合の結果表示には自動販売機をそのまま流用しましょう。実行の様子は図 5.18 の通りです。

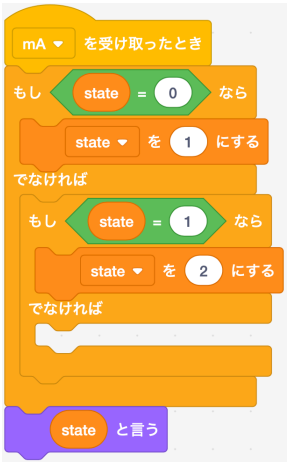
スプライト [a]、[b] のプログラムは図 5.19 の通りで、自動販売機の場合と同様です。オートマトンのプログラムは図 5.20 の通りです。ここに state が 2 のときには何もしないことに注意してください。このオートマトンでは状態 2 は死状態です。死状態ではどんな入力についても他の状態へ遷移しませんから、何もする必要はないのです。

このオートマトンでは a を 1 回押し、続けて (end) を押した場合のみ受理され、それ以外では全て不受理です。

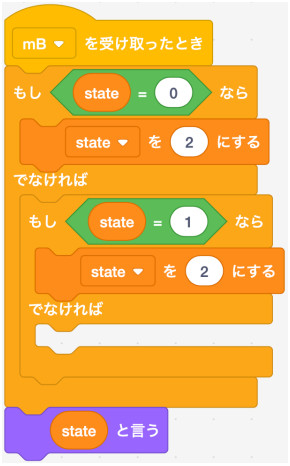
(a) スプライト [a] のプログラム

(b) スプライト [b] のプログラム

図 5.19: 二つのボタンのプログラム



(a) メッセージ mA を受信したとき



(b) メッセージ mB を受信したとき

図 5.20: それぞれのメッセージに反応するオートマトンのプログラム：mE を受信したとき、緑の旗がクリックされたときのプログラム部分は図 5.9 (d)、(e) と同じのため、省略

5.5　練習問題

前章の各 DFA をプログラミングしなさい。

1

1

第6章　非決定性有限状態オートマトン

この章では決定性有限状態オートマトンを機能拡張し、非決定性の動作を行う有限状態オートマトンを解説します。

「非決定性」（あるいは「非決定的」）とは、ある状態において次の動作が一意に定まらず、複数の動作が可能であることを言います。この拡張によって、オートマトンの表現力はさらに高まります。ただし、内容がやや分かりにくくなります。注意しながら読み進めてください。

6.1　非決定性

この節では非決定性の概念を説明します。

6.1.1　非決定的な動作とは

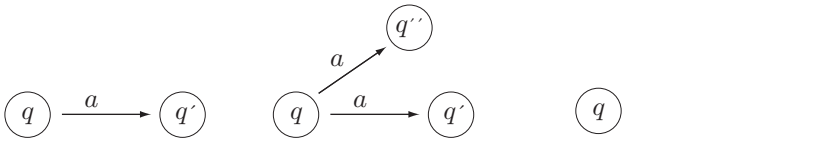
4章の決定性オートマトンでは、状態 q において記号 a を読んだときの遷移先状態 q' は必ず一意に決まりました（図 6.1 (1) 参照）。このようなオートマトンの動作は決定性であると言います。これに対して、状態 q において記号 a を読んだときの遷移先状態が q', q'', ... と複数あってもよい（図 6.1 (2) 参照）とするオートマトンの動作を **非決定的である**（nondeterministic）と言います。表現の自由度が上がるため、うまく利用すれば状態遷移図が書きやすくなります。

また、決定性オートマトンでは、状態 q において記号 a を読んだときの遷移先状態が存在しないということは決してありえませんが、非決定性オートマトンでは遷移先状態が無い（図 6.1 (3) 参照）場合も許されます。その場合、オートマトンの動作はそれ以上、先へ進めなくなります。

オートマトンの動作が非決定的ならば、様々な状態遷移がありえます。ある遷移先を選択して先へ進んだ場合、入力記号列を読み終えたときに最終状態で止まるかもしれません。別の遷移先を選択した場合には、記号列を読み終えたときに最終状態で止まらないかもしれません。では、遷移先をどのように選べばよいのでしょうか。

実は非決定性有限状態オートマトンでは**どの遷移先を選ぶかは重要な問題ではありません**。非決定性有限状態オートマトンでは、

- 初期状態から到達可能な **全て** の路を調べ、その中に最終状態で止まる路が **少なくともひとつある** ならば、その記号列は **受理** される



(1) 決定性　　(2) 非決定性（その1）　　(3) 非決定性（その 2, a による遷移がない）

図 6.1: 決定性と非決定性の違い

と考えます。逆に、

- 初期状態から到達可能な **全て** の路を調べ、その中に最終状態で止まる路が **全くない** ならば、記号列は受理されない

と考えます。重要なことは、特定の動作経路について受理/不受理を判断するのではなく、全ての経路を調べ尽くして判断するという点です。

慣れないうちは非決定性の便利さが理解しづらいかもしれませんが、しだいにその表現能力の高さに気付くはずです。

なお、これ以降は、記述の便利のために、決定性有限状態オートマトン（deterministic finite state automaton）を **DFA** と略記し、非決定性有限状態オートマトン (nondeterministic finite state automaton）を **NFA** と略記します。

6.1.2　缶ジュースの自動販売機

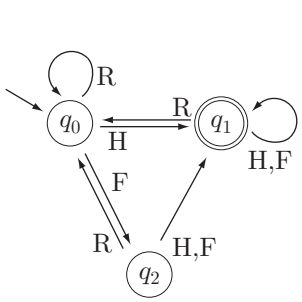
4.2 節の缶ジュースの自動販売機を再度検討します。

缶ジュースが出てくる記号列の集合 L_J を 4.2 節では以下のように求めました。

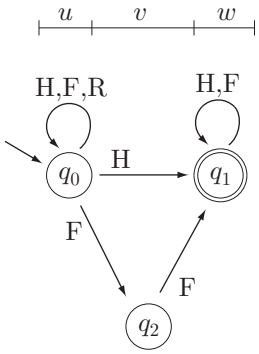
$$L_J = \{uvw \mid u \in \{\mathrm{H},\mathrm{F},\mathrm{R}\}^*,\ v \in \{\mathrm{H},\mathrm{FF}\},\ w \in \{\mathrm{H},\mathrm{F}\}^*\}$$

L_J に含まれる記号列は部分列 u、v、w を連接したものでした。しかし、この集合を受理する DFA（図 4.2、41）の形状は列 uvw の構造を反映する形にはなっておらず、集合とオートマトンの対応関係を読み取ることは容易ではありません。NFA を用いるならば、非決定的な状態遷移を用いた記述が可能であるため、記号列の構造を反映したオートマトンの設計が可能です。

図 6.2 (b) は缶ジュースの自動販売機を NFA（の状態遷移図）で表現したものです。図の上部には、部分列 u、v、w とオートマトンの各部位の対応関係を示しました。状態 q_0 から H、F、R を読んで q_0 へ戻る閉路が u $(\in \{H, F, R\}^*)$ に対応します。状態 q_0 から状態 q_1 へ至る二つの経路が v $(\in \{H, FF\})$ に対応します。状態 q_1 から H、F を読んで q_1 へ戻る閉路が w $(\in \{H, F\}^*)$ に対応します。



(a) 決定性オートマトン（図 4.2 の再掲）

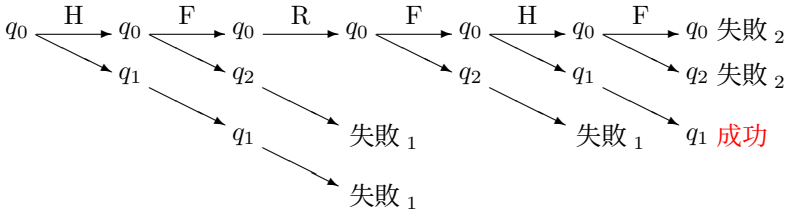


(b) 非決定性オートマトン

図 6.2: 缶ジュースの自動販売機の状態遷移図：決定性と非決定性の比較

4 章の DFA（図 6.2 (a) 参照）と異なるのは、状態 q_0 において H を読んだときに q_0 と q_1 の複数の状態へ遷移できることです。同様に q_0 において F を読んだときに q_0 と q_2 へ遷移できます。このような遷移は DFA ではありえません。また q_2 において H、R を読んだときにはどこへも遷移できません。このような「どこへも遷移できない」という状況も DFA ではありえません。DFA では、ある状態 q においてある記号 a を読んだ時にどこへ遷移すべきか、その遷移先は常にひとつ定まっていました。しかし、NFA では遷移先は **複数** あっても構いませんし、全く **無くても** 構いません。

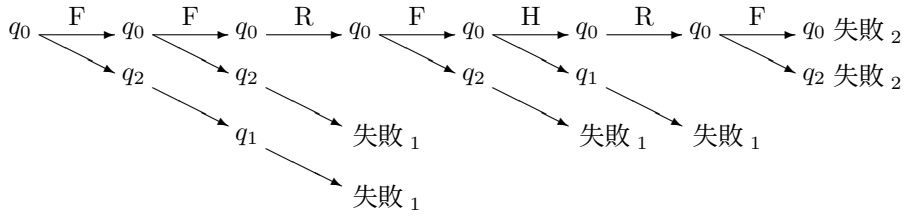
図 6.2 (b) のオートマトンで記号列 HFRFHF を読んだときの遷移は以下の図の通りです。



初期状態 q_0 において記号 H を読んだときの遷移可能な状態は q_0 と q_1 の二つです。このとき、どちらか一方の遷移が優先されるということは一切ありません。そこで、たとえば q_1 へ遷移してみます。そうすると、次に q_1 において記号 F を読むと q_1 へ遷移します。さらに q_1 において記号 R を読もうとしますが、このとき遷移する先がありません。よってこの路は行き止まりとなり、受理動作 **失敗** したこととなります。上の図には遷移先を誤って途中で行き止まりになる路が三つあります。上の図ではこのような路の終端に「失敗 $_1$」を付けました。また、記号列全体を読み終えたものの最終状態に至らなかった路が二つあります。上の図ではこのような路の終端に「失敗 $_2$」を付けました。しかし、記号列全体を読み終え、最終状態に至る路がひとつ存在します。この路を上の図では「 **成功** 」で表しました。ひとつだけとは言え、成功する路がありますから、記号列 HFRFHF はこのオートマトンによって受理されると考えます。成功する路が複数個あっても構い

ません。

もうひとつ例を示します。以下の図は、図 6.2 (b) のオートマトンで記号列 FFRFHRF を読んだときの全ての状態遷移です。



この場合、全ての路が失敗しています。よって記号列 FFRFHRF は受理されません。

NFA における受理/不受理は成功する路が少なくともひとつ存在するか否かで判断されます。受理される記号列による状態遷移であっても、失敗する路もあります。しかし、ひとつでも成功する路があれば、その記号列は受理されると判断します。

なお、 **任意** の DFA は NFA です。何故ならば、決定性の動作は非決定性の動作の特殊なケースと考えられるからです。

6.2　様々な NFA

4 章で示した DFA の言語の例（4.3 節参照）の中には、NFA を用いることで、より簡素に表現できるものがあります。以下、それを示し、4 章との比較を通して DFA と NFA の違いを明らかにします。もちろん、4 章と同様に $\Sigma = \{a, b\}$ と約束します。なお、4 章の例 1 から例 10 のうち、NFA を用いても DFA から簡単化できない例は以下に載せていません。

図 6.3、図 6.4 には、4 章の例 2、3、4、5、7、8、9、11 の決定性オートマトン（DFA）を左側に、対応する非決定性オートマトン（NFA）を右側に載せました。両者を比較しながら考えてください。

例 2: $\{\varepsilon\}$

空列のみを受理する NFA の状態遷移図は図 6.3 (2) の通りです。これに対応する DFA では死状態が必要でしたが、NFA では必要ありません。NFA では動作に行き詰まった場合には受理しないと約束されているからです。

例 3: $\emptyset$

受理言語が空集合であるような NFA の状態遷移図は図 6.3 (3) の通りです。DFA では自分自身へ戻る枝が必要でしたが、NFA ではそのような枝を付ける必要がありません。たとえ q_0 から q_0 へ遷移したところで受理しない事実に変わりないからです。

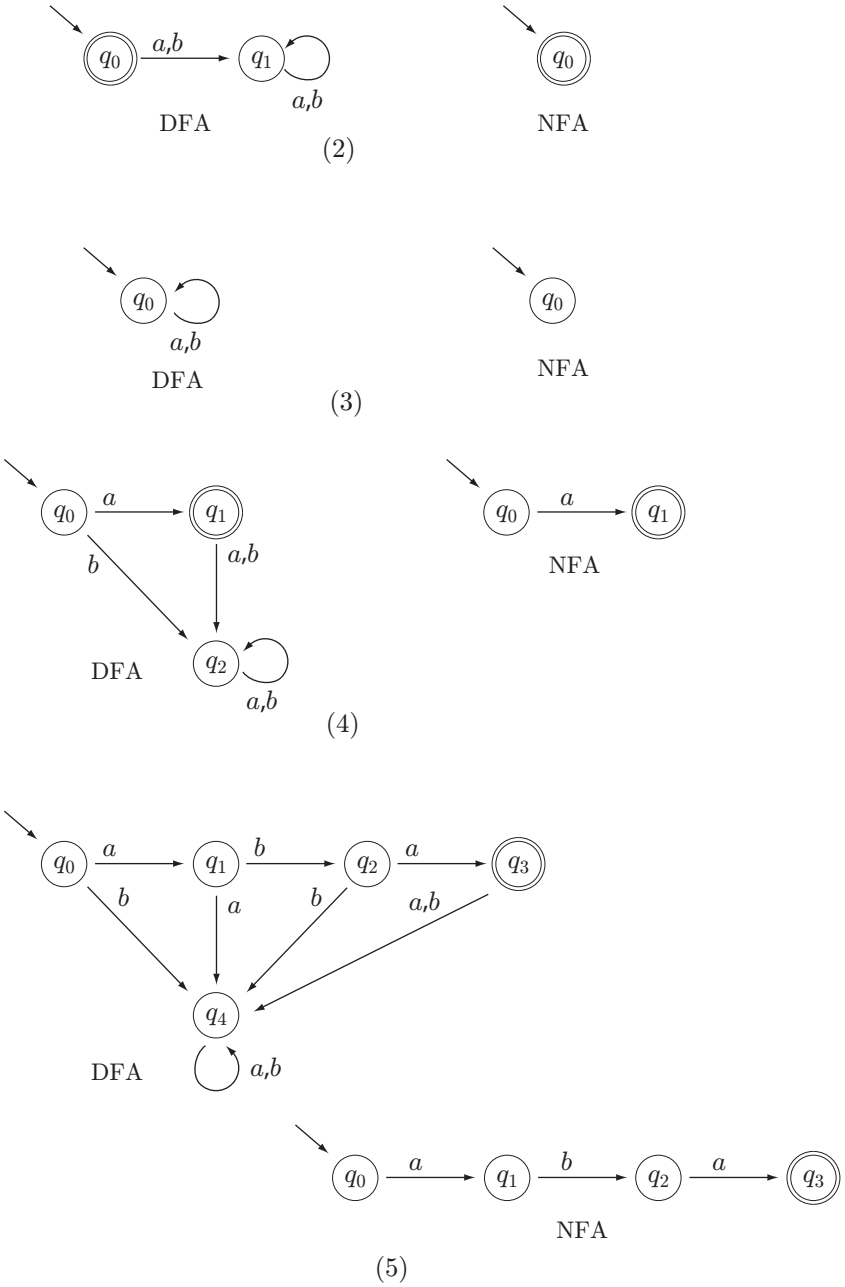


図 6.3: 様々な DFA と NFA の比較（その 1）

例 4: $\{a\}$

受理言語が記号列 a だけからなる集合 $\{a\}$ であるような NFA の状態遷移図は図 6.3 (4) の通りです。DFA では死状態を導入し、受理されないことが分かった記号列を途中から無駄読みしました。しかし NFA はそのような本質的でない部分の記述を必要としません。

例 5: $\{aba\}$

受理言語が集合 $\{aba\}$ であるような NFA の状態遷移図は図 6.3 (5) の通りです。DFA には死状態がありますが、NFA は死状態を必要としません。

例 7: $\{a^n \mid n \geq 0\}$

0 個以上の a からなる列を受理する NFA の状態遷移図は図 6.4 (7) の通りです。このオートマトンでも死状態を必要としませんから、DFA よりも状態数が減っています。

例 8: $\{a^n \mid n \geq 0\} \cup \{b^n \mid n \geq 0\}$

4 章の例 8 の DFA が例 7 の DFA の拡張であったように、 例 8 の NFA も例 7 の拡張として作ることができます。この NFA の状態遷移図は図 6.4 (8) の通りです。NFA の構造が a と b について対称的であることに注意してください。このオートマトンも死状態を必要としません。

例 9: $\{(ab)^n \mid n \geq 0\}$

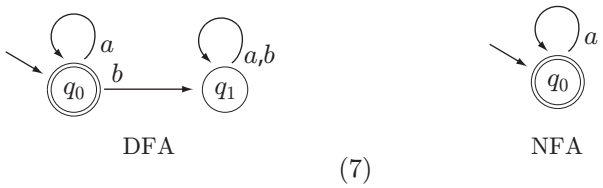
この NFA の状態遷移図は図 6.4 (9) の通りです。このオートマトンも死状態を必要としません。

実はこの節のここまで説明してきた NFA では死状態が必要ないという点が DFA との主な相違点でした。しかし、NFA と DFA の最も大きな違いは、NFA では複数の選択肢が可能という点です。その特長をうまく活かした例が以下です。

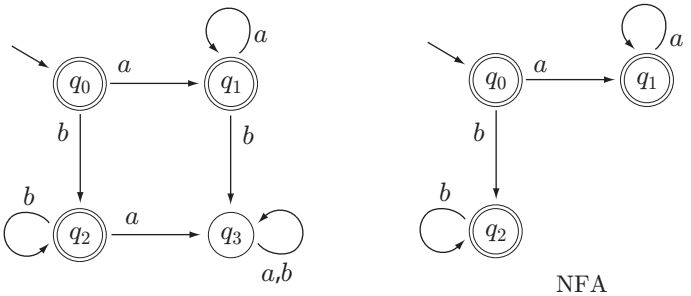
例 11: $\{uaa \mid u \in \Sigma^*\}$

記号列の末尾が必ず aa で終わるような列を受理する NFA の状態遷移図は図 6.4 (11) の通りです。DFA では b を読む度に q_0 へ遷移する構造になっていました。これはオートマトンの状態をリセットすることに相当します。しかし、NFA にはそのようなリセットの仕組みは必要ありません。

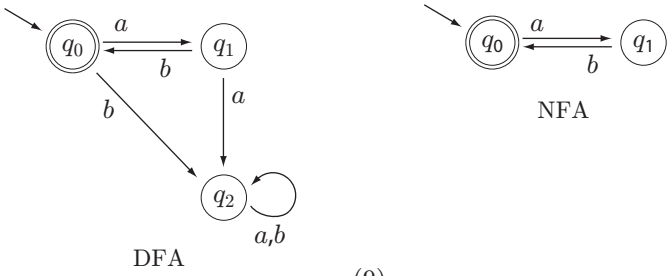
この NFA では状態 q_0 から a を読んで遷移する先に複数の選択肢が存在します。a を読んだときに q_0 に戻るか、q_1 へ移動するか、どちらを選択しても構いません。全ての記号を読み込んだときにちょうど最終状態 q_2 へ到達しているような遷移が存在すればよいのです。



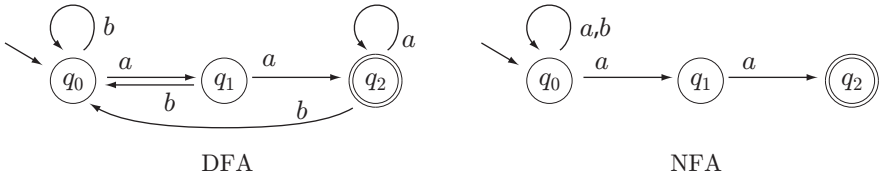
(7)



(8)



(9)



(11)

図 6.4: 様々な DFA と NFA の比較（その 2）

考察（4 章からのつづき）　集合 $\{uaav \mid u, v \in \Sigma^*\}$ を受理集合とするような NFA を作りなさい。すなわち、aa という部分列を含む列、たとえば aa、baa、$abaa$、$abbaabbb$ などを受理する NFA を作りなさい。

6.3　練習問題

1. $\Sigma = \{a, b\}$ として、以下の言語を受理する NFA（の状態遷移図）をそれぞれ示しなさい。ただし、いずれの NFA の状態数も 3 以下とし、枝の数はできるだけ少なくすること。

   (a) 先頭と末尾にのみ b が現れる記号列の全体（bb, bab, $baaaab$ など）。

   (b) 先頭と末尾に b が現れる記号列の全体（先頭と末尾以外に b が現れてもよい。bb, bab, bbb, $babaab$, $bbbab$ など）。

   (c) 記号列 bb を含む記号列の全体（b が 3 個以上現れてもよい。bb, $abbb$, $abaabbab$ など）。

1

1

第7章　非決定性有限状態オートマトンのプログラミング

まず前章の復習です。オートマトンの動作が非決定的ならば、様々な状態遷移がありえます。ある遷移先を選択したときには入力記号列を読み終えたときに最終状態で止まるかもしれず、別の遷移先を選択したときには最終状態で止まらないかもしれません。非決定性有限状態オートマトンではどの遷移先を選ぶかは重要な問題ではありません。初期状態から到達可能な **全て** の路を調べ、その中に最終状態へ到達する路が **少なくともひとつ** あるならば、その記号列は **受理** される、と考えます。ひとつもなければ受理しません。

非決定性の動作をそのままプログラム化するために、Scratch の **クローン**（clone ＝複製）の機能を用います。つまり、ある状態 q から記号 a を読んで、状態 q' と q'' へ遷移できる場合、オートマトンのクローンをひとつ作り、元のオートマトンは q' へ遷移させます。クローンのオートマトンには q'' へ遷移させます。この二つのオートマトンは、それ以降、同じ記号列を読みながらも独立して動作していきます。

この章では、まず、Scratch でクローンを作る方法を解説し、その後に決定性オートマトンのプログラムを非決定性へ改造します。

7.1　スプライトのクローン

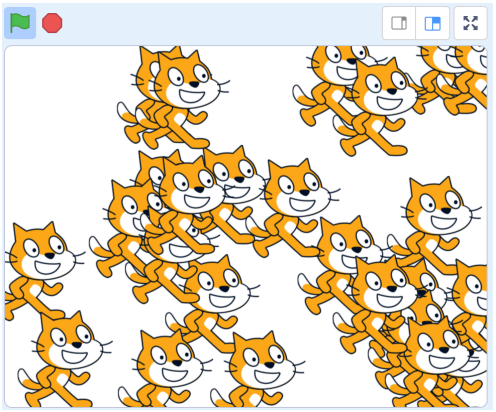
Scratch にはクローンに関する以下の三つのブロックがあります。

… スプライトがこのブロックを実行すると、このスプライトと全く同一のスプライトをステージ上にクローニング（複製）します。その直後にはクローンは元のスプライトと**全く同じ情報を持つ**ため、ステージ上ではピッタリと重なって表示されており、一見すると、クローニングが行われたように見えません。また、クローンのスプライトがスプライトペインにリストアップされることはありません。

… クローニングされたスプライトがプログラムの実行を開始する起点となるブロックです。

… このブロックを実行するスプライトがクローンであるならば、このクローンは消滅します。また、全てのプログラムを停止させるボタン を押したときには、このブロッ

(a) 自分自身のクローンを作り続ける猫のプログラム



(b) プログラムの実行の様子

図 7.1: 猫のクローンが 1 秒毎に作られていくプログラム

図 7.2: クローンの猫も自分のクローンを作る改造

クの実行如何に関わらず、全てのクローンが強制消滅します。このテキストではこのブロックは利用しません。

クローンを用いた例を示します。

図 7.1 は、緑の旗が押された後、猫が 1 秒毎に自分のクローンを作るプログラムです。クローンの猫は、生成された後すぐにステージ上のどこかへ（乱数で位置を求め）移動します。このプログラムを動かすとだんだんと猫がステージに増えていきます。

図 7.2 のプログラムは、図 7.1 の下方のプログラムを改造したものです。この改造によって、クローンもまたクローンを作るため、猫の数が爆発的に増えていきます。ただし Scratch の言語仕様ではクローンの数は 300 以上増えないため、それ以上のクローン生成命令は全て無視されます。

7.2　自動販売機のプログラミング

クローンの機能を用いれば、決定性オートマトンは容易に非決定性オートマトンへ改造できます。遷移先が複数になる局面でクローンを作り、それぞれの状態へ遷移させればよいのです。

7.2.1　改造のポイント

決定性オートマトンからの改造箇所を以下にまとめます。

プログラムの開始直後： プログラムは決定性オートマトンと同様に初期化処理を行い、同様に動き始めます。

入力ボタン： 四つのボタンは決定性オートマトンと同じ仕組みでメッセージをブロードキャストします。

個々の決定性オートマトン: この章では非決定性オートマトンの動作をクローニングされた複数の決定性オートマトンの動作として実現します。その個々の決定性オートマトンの動作は以下の通りとします。

遷移先が唯一： 遷移先が 1 個の場合、決定性オートマトンと同様に動きます。

遷移先が複数： 遷移先が n 個（$n > 1$）の場合、$n-1$ 個のクローンを作り、元のオートマトンと $n-1$ 個のオートマトンの計 n 個のオートマトンがそれぞれ異なる状態へ遷移します。

遷移先が無い： 遷移先が定義されていない場合、そのオートマトンの状態 state を -1 に設定します。この -1 を死状態に相当するものとし、これ以降、如何なる入力についてもこのオートマトンの状態を変更しません。もちろん最終状態へ遷移することもありません[1]。

終了： メッセージ mE を受け取った時、もしそのオートマトンの状態が最終状態であるならば、メッセージ mAccept をブロードキャストします。5 章とは異なり、最終状態ではないならば如何なるメッセージをブロードキャストしません。

グラス： このスプライトの動作はかなり変更します。というのも、複数のオートマトンが動作する場合、受理/不受理の結果も複数個ありえます。ここで重要なのは、記号列を受理するオートマトンが一つ以上あるか、あるいは全く無いかです。そこで、

mInit を受信したならば ... スプライトを隠します。これは 5 章と同じです。

mE を受信したならば ... 不受理の表示を行います。つまり、記号列を受理するオートマトンが居ないことを前提として、一旦、不受理の表示を行います。

mAccept を受信したならば ... 受理の表示を行います。なお、上の mE を受信したときに一旦、不受理の表示がなされています。よって表示を変更する（コスチュームを変える）ことになります。

[1] 決して最終状態にならないならば、このオートマトンを消滅させた方が分かりやすいし、無駄な実行を行わず効率的です。しかし、消滅できるのはクローンのみであり、クローンでないオリジナルのスプライトはプログラムで消滅させることはできませんから、結局、この方法はうまく働きません。

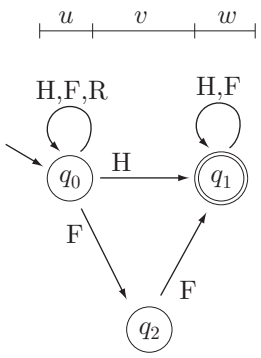


図 7.3: 缶ジュースの自動販売機（非決定性有限状態オートマトン）の状態遷移図の再掲

5 章で用いたメッセージ mReject はこの章では用いません。

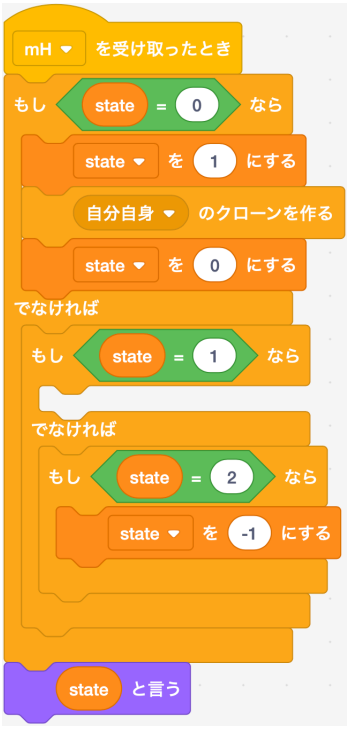
以下、この方針に沿って各スプライトを具体的に設計していきます。上記だけでは分かりにくいところがありますから、次項以降、実際のプログラムを示しながら解説します。

7.2.2　猫のオートマトン制御部

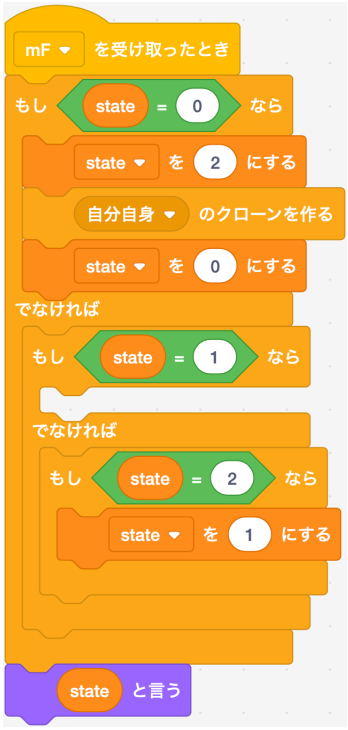
まず、図 7.3 に図 6.2 (b) の非決定性オートマトンを再掲します。これをプログラム化したものが図 7.4 です。

たとえば図 7.4 (a) は [H] ボタンが押され、メッセージ mH を受信したときの動作です。以下、その解説です。

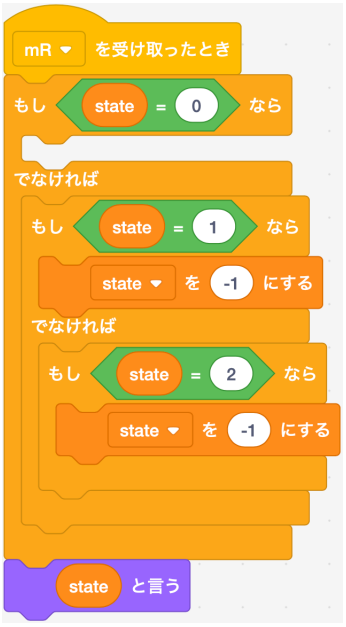
1. もし state が 0 ならば（図 7.3 で言えば q_0 ならば）、
   (a) state を 1 に変更し、
   (b) クローンを作ります。
   (c) クローン元のオートマトンは、 state を再度 0 に変更します。
2. さもなくば（state が 0 でないならば）
   (a) もし state が 1 ならば（図 7.3 で言えば q_1 ならば）、
      i. 何もしません。何故ならば遷移先も state は 1 だからです。
   (b) さもなくば（state が 1 でないならば）
      i. もし state が 2 ならば（図 7.3 で言えば q_2 ならば）、
         A. state を -1 に変更します。何故ならば、図 7.3 で言えば q_2 から H で遷移できる先はありません。そこで、state を死状態（-1）に変更し、これ以降、このオートマトンは如何なる入力にも状態遷移を行いません。



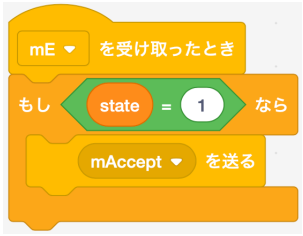
(a) メッセージ mH を受信したとき



(b) メッセージ mF を受信したとき



(c) メッセージ mR を受信したとき



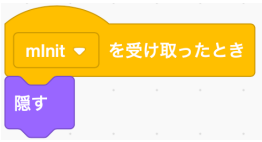
(d) メッセージ mE を受信したとき

図 7.4: それぞれのメッセージに反応するオートマトンのプログラム：mE を受信したとき、緑の旗がクリックされたときのプログラム部分は図 5.9 (d)、(e) と同じのため、省略

図 7.5: クローンの動作開始プログラム

(a) メッセージ mAccept を受信したとき

(b) メッセージ mE を受信したとき



(c) メッセージ mInit を受信したとき

図 7.6: グラスのプログラム

ii. さもなくば、何も行いません。

3. state を表示します。

上の動作でクローンが生成されたならば、そのクローンは図 7.5 のプログラムで動き始めます。図 7.1 で述べたように、クローンは生成された直後は元のスプライトと重なって表示され、生成されたことが目視で確認できないため、ここでは位置をステージの左下 $[-200, 0] \times [-200, 0]$ の範囲内で乱数で移動させています。

[F] ボタン、[R] ボタンが押された場合の動作も同様です。

メッセージ mE を受信した場合の動作は – 前項で述べたように 5 章とは異なり、state が 1 ならば、状態 1 は最終状態であるため、メッセージ mAccept をブロードキャストします（図 7.4 (d) 参照）。state が 1 以外の場合には何もしません。

7.2.3　グラスのプログラム

図 7.6 がグラスのプログラムです。これは 5 章の決定性オートマトンのプログラムとほとんど同じです。異なるのは、メッセージ mReject を受信したときの動作をメッセージ mE を受信したときの動作に変えている点だけです。

分かりにくいところなので、上に述べたオートマトン、グラスの動作を以下に実行例を通して見て行きましょう。

図 7.7 (a) は緑の旗をクリックし、プログラムが実行した直後のステージの様子です。オートマトンは初期状態（state = 0）です。

ここで [F] をクリックすると、ステージは図 7.7 (b) のように変化します。状態 0 から F で遷移できる先は 0 と 2 です。そこで、オートマトンのクローンが生成され、クローンの状態は 2、元のオートマトンの状態は 0 となります。

さらにもう一度 [F] をクリックすると、ステージは図 7.7 (c) のように変化します。状態 0 のオートマトンは再度、クローンを生成し、それぞれ状態 0 と状態 2 の二つのオートマトンになります。状態 2 のオートマトンは状態 1 へ遷移します。

次に [R] をクリックすると、ステージは図 7.7 (d) のように変化します。状態 1、状態 2 のオートマトンは R で遷移できませんから、その状態を −1 に変えます。−1 は死状態に相当し、これ以降、このオートマトンが最終状態に遷移することは決してありません。

次に [H] をクリックすると、ステージは図 7.7 (e) のように変化します。状態 0 のオートマトンはクローンを生成し、それぞれ状態 0 と状態 1 の二つのオートマトンになります。

そして (end) をクリックすると、ステージは図 7.7 (f) のように変化します。(end) のスプライトはメッセージ mE をブロードキャストします。グラスのスプライトはこのメッセージを受信し、グラスは一旦、不受理の表示を行います。全てのオートマトンも mE を受信します。そして状態 1 のオートマトンのみメッセージ mAccpet をブロードキャストし、それ以外のオートマトンは何も動作しません。結果、mAccpet はグラスのスプライトに受信され、ステージは図 7.8 (g) のように、グラスが受理の表示に上書きされます。グラスの表示が図 7.7 (f) の不受理から図 7.8 (g) の受理へ変更されますから、運が良ければ表示が切り替わる瞬間を見ることができるかもしれませんが、通常は表示は極めて短時間に切り替わるため、目視で確認することは困難です。

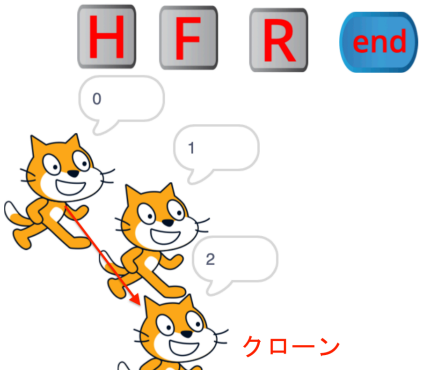
もしオートマトン、グラスの終了処理を 5 章のプログラムから変更しなかったならば、(end) のスプライトがブロードキャストしたメッセージ mE はオートマトンのみが受信し、グラスは受信しません（図 7.9 (a) 参照）。オートマトンはそれぞれの受理/不受理の結果を mAccept、mReject としてブロードキャストします。それらのメッセージはグラスによって受信され、受理、不受理の表示を行います。メッセージが受信される順番は不定です（この辺が並行プログラミングの難しさです）から最終的に受理の表示になるか、不受理の表示になるかは確定できません（図 7.9 (b) 参照）。これでは不都合です[2]。

図 7.10 は不受理の場合の実行例です。どのオートマトンも mAccept を送信しませんから、不受

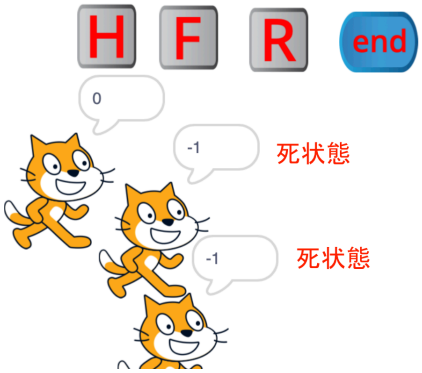
[2]厳密に言えば、図 7.7、図 7.8 も不都合がないとは言い切れません。メッセージ mE が (end) ボタンからグラスへ到着する時刻よりも、mE が (end) ボタンから記号列を受理するオートマトンへ到着し、メッセージ mAccept がそのオートマトンからグラスへ到着する時刻が早い場合があれば、表示は受理 → 不受理と変化し、誤った表示となります。1 回のメッセージ送受信が 2 回のメッセージ送受信よりも遅いことはほとんどありえませんが、0 とは言え切れません。

(a) 緑の旗をクリックした直後

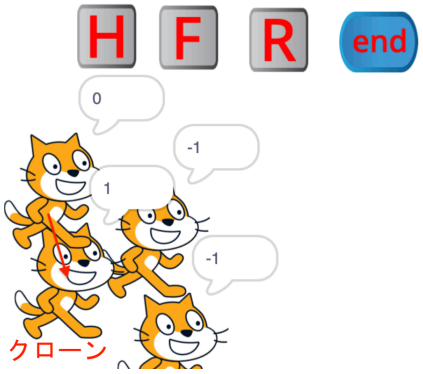
(b) [F] をクリックした直後



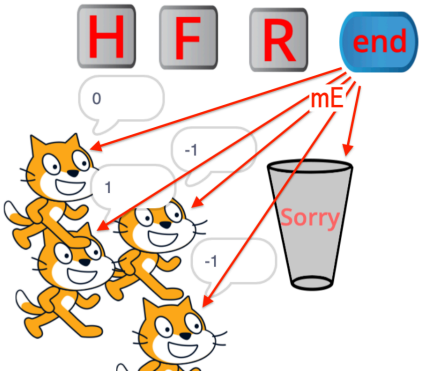
(c) [F] をクリックした直後



(d) [R] をクリックした直後

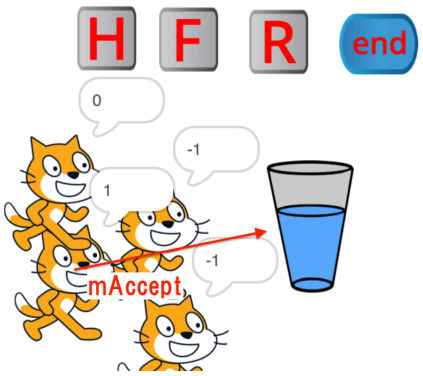


(e) [H] をクリックした直後



(f) (end) をクリックした直後

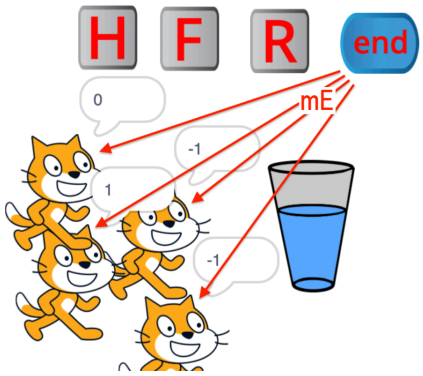
図 7.7: 非決定性オートマトンの動作例（その 1、その 2 へ続く）



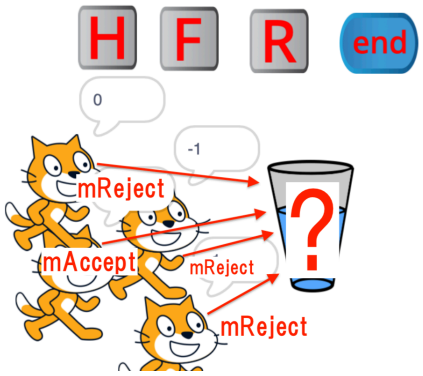
(g) 各オートマトンがメッセージ mE を受信した直後

図 7.8: 非決定性オートマトンの動作例（その 2）

1　理の表示は上書きされません。



(a) (end) をクリックした直後

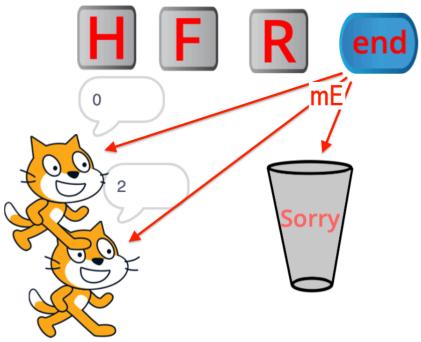


(b) 各オートマトンがメッセージ mE を受信した直後

図 7.9: オートマトン、グラスの終了処理を 5 章のままにした場合：複数のオートマトンからの mAccept と mReject が同時にグラスに届き、表示が確定しない

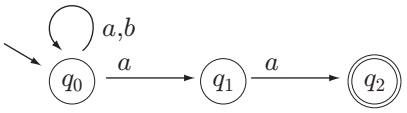
(a) 緑の旗をクリックした直後

(b) [F] をクリックした直後



(c) (end) をクリックした直後

図 7.10: 非決定性オートマトンの動作例（その 3）



(8)

図 7.11: 図 6.4 (11) の再掲

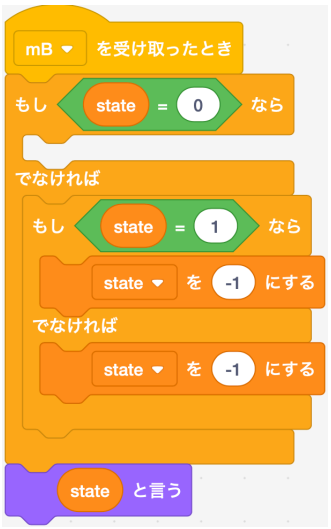
7.3　その他のオートマトンのプログラミング例

この章でも自動販売機の例題プログラムを任意の有限状態オートマトンのプログラムに借用します。図 6.4 (11) のオートマトン（図 7.11 に再掲）を例として解説します。

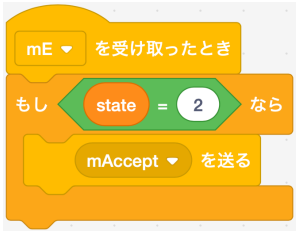
7.3 節と同様に自動販売機のスプライト [H]、[F] をそのまま [a]、[b] に模様替えします。コスチュームを変え、メッセージ名を mA、mB に変えます。スプライト [R] は削除します。スプライト (end) は自動販売機と全く同じです。受理/不受理の場合の結果表示には自動販売機をそのまま流用しましょう。スプライト [a]、[b] のプログラムは図 5.19 の通りです。グラスのスプライトのプログラムは前項、図 7.6 の通りです。

オートマトンのプログラムは図 7.12 の通りです。

(a) メッセージ mA を受信したとき



(b) メッセージ mB を受信したとき



(c) メッセージ mE を受信したとき

図 7.12: それぞれのメッセージに反応するオートマトンのプログラム：緑の旗がクリックされたときのプログラム部分は図 5.9 (e) と同じのため、省略

1

1

第8章　オートマトンの等価変換

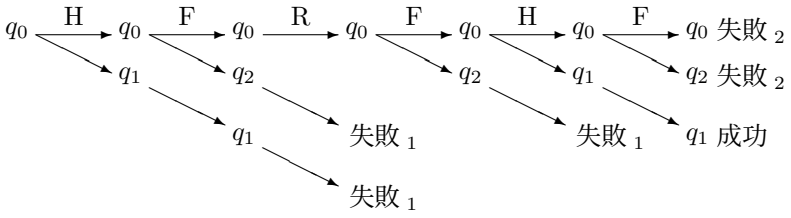
実は非決定性有限状態オートマトン（NFA）と決定性有限状態オートマトン（DFA）は**本質的に等価な**表現力を持ちます。ここでいう「本質的に等価」とは、ある形式言語（記号列の集合）を NFA で表現できるならば、必ず同じ言語を DFA で表現することができ、逆にある形式言語を DFA で表現できるならば、必ず同じ言語を NFA で表現することができることを言います。この章ではこの等価性を解説します。

一般に非決定的性質と決定的性質が等価になることはほとんどありません。たとえば現実世界の事象において「どちらを選択してもよい」（非決定性）という状況と「必ずこれを選択せよ」（決定性）という状況を比べれば、両者は全く様相を見せます。それが等価になるのは有限状態オートマトンの状態数が有限であるためです。

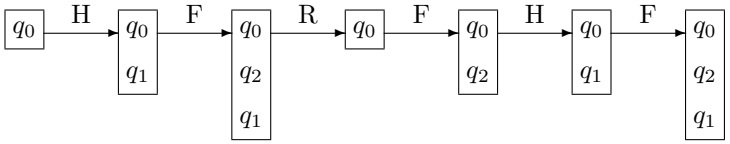
8.1　アイディア

NFA の最も興味深い性質は、 **任意** の NFA には **等価** な DFA が存在し、その NFA を DFA に変換する簡単な方法が知られていることです。

自動販売機の NFA（図 6.2 (b)）を思い出しましょう。記号列 HFRFHF による NFA の状態の遷移は以下の通りでした。



記号を読み進めるに従って、初期状態から到達可能な状態が増減を繰り返していきます。そして、記号を読み終えたときに、最終状態がひとつでも存在すれば、その記号列はこの NFA によって受理されたと考えました。この NFA の動作を DFA でシミュレートするには、NFA において初期状態から到達できる全ての **状態の集合** を DFA のひとつの状態と見なします。そうするとオートマトンの動作は読み込む記号に応じて常に一意に（決定的に）定まります。以下の図は、上の NFA の動作例について状態の集合を四角の枠で囲み、決定的な動作として表現したものです。



上の図の初期状態は $\boxed{q_0}$ です。ここで記号 H を読むと、NFA では状態 q_0 または q_1 に遷移しますが、それを DFA では状態 $\boxed{q_0\ q_1}$ に遷移すると解釈します。次に記号 F を読むと、NFA では q_0 から q_0 または q_2 へ遷移し、q_1 からは q_1 へ遷移します。それを DFA では状態 $\boxed{q_0\ q_1}$ から状態 $\boxed{q_0\ q_2\ q_1}$（あるいは $\boxed{q_0\ q_1\ q_2}$、集合なので同じものと見なします）へ遷移すると解釈します。これを繰り返し、結局、DFA は初期状態は $\boxed{q_0}$ から記号列 HFRFHF を読んで状態 $\boxed{q_0\ q_2\ q_1}$（あるいは $\boxed{q_0\ q_1\ q_2}$）へ遷移します。これを NFA に言い換えれば、q_0、q_1 または q_2 へ遷移することを意味し、この中に最終状態 q_1 が含まれていますから記号列は受理されたと考えます。DFA について言えば、$\boxed{q_0\ q_2\ q_1}$ の中には NFA の最終状態 q_1 が含まれますから、$\boxed{q_0\ q_2\ q_1}$ を DFA の最終状態と見なします。そうすると、DFA の最終状態に到達したこととなり、この DFA によって記号列は受理されます。

この、NFA の状態の集合を DFA のひとつの状態と見なし、NFA を DFA でシミュレートするアイディアはうまく行きそうです。実際うまく行きます。なぜならば、NFA の状態数は有限ですから状態の部分集合の数も高々有限です。今、NFA の状態数を n とするとき、変換後の DFA の状態の数は高々 2^n となります。2^n は n と共に非常に大きな数になります。$n = 6$ のときには、$2^n = 64$ となりますから、$n = 6$ の場合でさえ、人手で扱うことはもはや困難です。しかし有限数ですから、原理上は DFA で構成可能です。

8.2　変換方法

上に述べたアイディアは NFA から DFA への[1]変換を示唆しています。実際、**任意**（ここ大事！）の NFA は等価な（受理言語が等しい）DFA へ変換可能です。以下では、その変換方法を具体例で解説します。

8.2.1　簡単な場合

図 8.1 は、二つの記号列 00、01 のみを受理する単純な NFA です。状態 q_0 において 0 を読むときに非決定性の動作が起きます。これを前節の考え方に基づいて DFA に変換します。

図 8.2、図 8.3 がその DFA を作っていく過程を順に示したものです。

まず、全てのオートマトンの動作は初期状態から開始されますから、図 8.2 (a) のように、初期状態 q_0 をこれから求める DFA の状態遷移図に配置します。

[1]編集ミスで、解説動画を撮影した時点では「DFA から NFA への」になっていました。

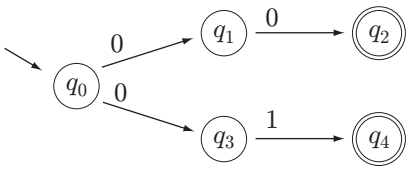


図 8.1: 単純な非決定性有限状態オートマトン

次に、q_0 から 0 を読んで遷移できる状態を元の NFA（図 8.1）から求めます。この場合、遷移できる先は q_1 と q_3 ですから、その両方へ遷移すると仮定し、図 8.2 (b) のように、q_1 と q_3 をラベルとする状態を状態遷移図に追加し、同時に遷移の枝も追加します。

次に、q_0 から 1 を読んで遷移できる状態を元の NFA から求めます。元の NFA には遷移する枝がありません。このような場合、図 8.2 (c) のように、新たに死状態 – これをここでは q_5 とします – を状態遷移図に追加し、q_0 から 1 を読んで q_5 へ遷移する枝を追加します。

次に、q_1 と q_3 の両方を持つ状態から 0 を読んで遷移できる状態を元の NFA から求めます。全ての遷移可能な状態を列挙したいため、この場合、q_1 から 0 を読んで遷移できる状態と q_3 から 0 を読んで遷移できる状態の集合和と考えます。そうすると、元の NFA では q_1 から 0 を読んで遷移できる先は q_2、q_3 から 0 を読んで遷移できる先はありません。よって、q_1 と q_3 の両方を持つ状態からは 0 を読んで q_2 へ遷移できるものとし、図 8.2 (d) のように、q_2 を遷移図へ追加し、同時に遷移の枝も追加します。なお、元の NFA では q_2 は最終状態ですから、DFA でも最終状態と設定します。

上と同様に、q_1 と q_3 の両方を持つ状態から 1 を読んで遷移できる状態を元の NFA から求めます。よって図 8.2 (e) のように、q_3 を遷移図へ追加し、同時に遷移の枝も追加します。

次に、q_2 から 0 を読んで遷移できる状態は元の NFA にはありませんから、死状態 q_5 へ遷移するものとし、図 8.3 (f) のように、q_2 から 0 を読んで q_5 へ遷移する枝を追加します。同様に、q_2 から 1 を読んで q_5 へ遷移する枝を追加します。

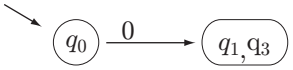
同様に、図 8.3 (g) のように、q_3 から 0, 1 を読んで q_5 へ遷移する枝を追加します。

同様に、図 8.3 (h) のように、q_5 から 0, 1 を読んで q_5 へ遷移する枝を追加します。

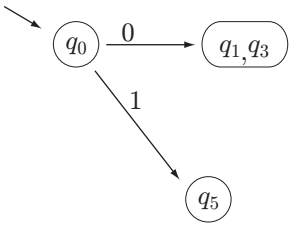
さて、もうこれ以上、状態遷移図へ追加できる要素はありません。よって、図 8.3 (h) の状態遷移図を持って DFA の作成は完了です。この DFA が元の NFA と同じ受理言語を持つことを確認してください。

図 8.2、図 8.3 では順次、状態、有向枝を追加していきましたが、追加の順番は任意です。初期状態から始めてどんな順番で記号を読んでいっても必ず同じ結果（状態遷移図）になります。

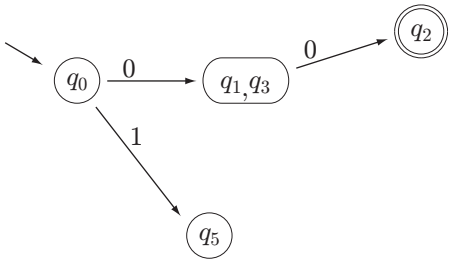
(a) 初期状態 q_0 を配置



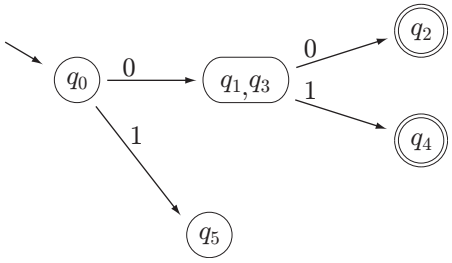
(b) q_0 から 0 を読んで遷移できる状態 q_1 と q_3 をひとつの状態として追加し、遷移の枝も追加



(c) q_0 から 1 を読んで遷移できる状態は元のオートマトンにないため、死状態として q_5 を追加し、遷移の枝も追加

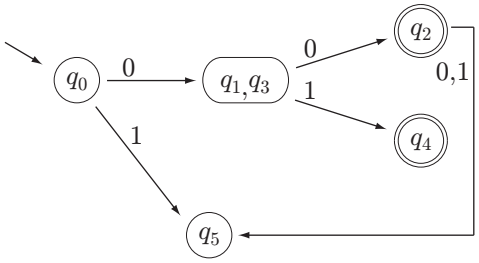


(d) q_1 と q_3 から 0 を読んで遷移できる状態 q_2 を追加し、遷移の枝も追加

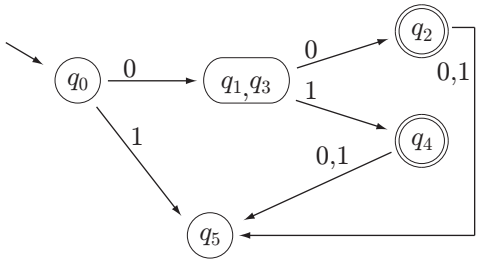


(e) q_1 と q_3 から 1 を読んで遷移できる状態 q_4 を追加し、遷移の枝も追加

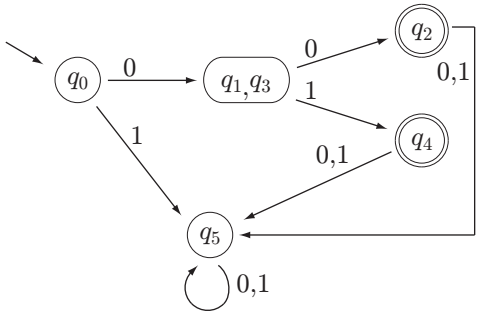
図 8.2: 図 8.1 の NFA から DFA への変換手順（その 1、その 2 へ続く）



(f) q_2 から 0 および 1 を読んで遷移できる状態は元のオートマトンにないため、死状態 q_5 へ遷移の枝を追加

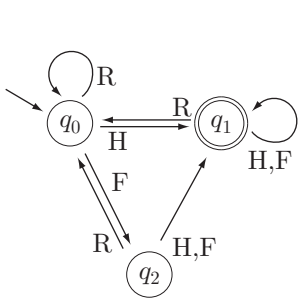


(g) q_4 から 0 および 1 を読んで遷移できる状態は元のオートマトンにないため、死状態 q_5 へ遷移の枝を追加

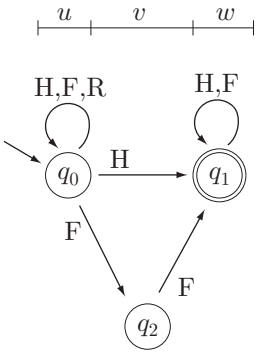


(h) 死状態 q_5 から 0 および 1 を読んで遷移できる元のオートマトンにないため、死状態 q_5 へ遷移の枝を追加

図 8.3: 図 8.1 の NFA から DFA への変換手順（その 1、その 2 へ続く）



(a) 決定性オートマトン（図 4.2 の再々掲）



(b) 非決定性オートマトン（図 6.2 (b) の再掲）

図 8.4: 缶ジュースの自動販売機の状態遷移図：決定性と非決定性の比較

8.2.2　自動販売機

缶ジュースの自動販売機についても NFA（図 8.4 (b)）を DFA に変換してみます。自動販売機はもともと DFA で定義しましたから、今さら変換するまでもないように思えますが、NFA を変換して作った DFA が元の DFA（図 8.4 (a)）と同じものになるのか確認しても面白いでしょう。

図 8.5 がその変換過程です。

前項の例と同様、まず初期状態 q_0 を状態遷移図に配置します（図 8.5 (a) 参照)。

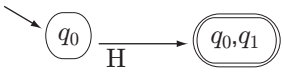
次に、q_0 から H を読んで遷移できる状態を元の NFA（図 8.4 (b)）から求めます。この場合、遷移できる先は q_0 と q_1 ですから、図 8.5 (b) のように、q_0 と q_1 をラベルとする状態を状態遷移図に追加し、同時に遷移の枝も追加します。なお、元の NFA では q_1 は最終状態ですから、q_0 と q_1 をまとめた状態も最終状態と設定します。

次に、q_0 から F を読んで遷移できる状態を元の NFA から求めます。この場合、遷移できる先は q_0 と q_2 ですから、図 8.5 (c) のように、q_0 と q_2 をラベルとする状態を状態遷移図に追加し、同時に遷移の枝も追加します。

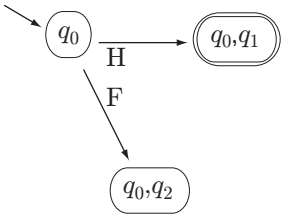
次に、q_0 と q_1 の両方を持つ状態から F を読んで遷移できる状態を元の NFA から求めます。このとき、元の NFA では q_0 から F を読んで遷移できる先は q_0 と q_2、 q_1 から F を読んで遷移できる先は q_1 ですから、q_0 と q_1 の両方を持つ状態からは F を読んで q_0、q_1、q_2 へ遷移できるものとし、図 8.5 (d) のように、 q_0、q_1、q_2 をひとつのまとめた状態を遷移図へ追加し、同時に遷移の枝も追加します。なお、元の NFA では q_1 は最終状態ですから、q_0、q_1、q_2 をまとめた状態も最終状態と設定します。

さらに状態を追加できないか調べます。しかしもうこれ以上新たに状態が追加されることはありません。四つの状態から H、F、R のそれぞれを読んで遷移する先を調べても必ず四つの状態のどれかに遷移します。そこでそれらの遷移の枝を全て追加し終えると、それが求める DFA（図

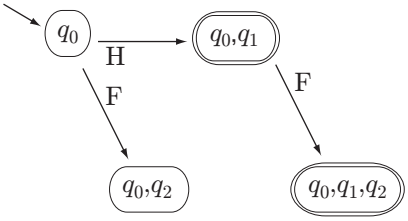
(a) 初期状態 q_0 を配置



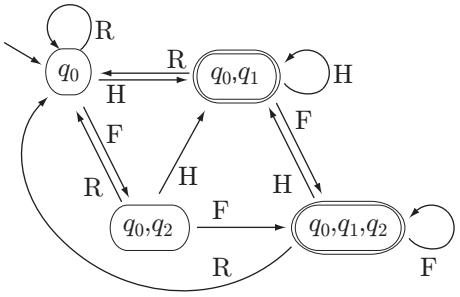
(b) q_0 から H を読んで遷移できる状態 q_0 と q_1 をひとつの状態として追加し、遷移の枝も追加



(c) q_0 から F を読んで遷移できる状態 q_0 と q_2 をひとつの状態として追加し、遷移の枝も追加



(d) q_0 と q_1 から F を読んで遷移できる状態 q_0、q_1、q_2 をひとつの状態として追加し、遷移の枝も追加



(e) 四つの状態間に遷移の枝を全て追加（このとき、新たな状態は追加されない）

図 8.5: 図 8.4 (b) の NFA から DFA への変換手順

8.5 (e)）となります。

さてここで求めた DFA（図 8.5 (e)）と、4 章で最初に示した DFA（図 4.2 ＝図 8.4 (a)）を比較しましょう。

すぐに気が付くのは図 8.5 (e) の状態数は 4 ですが、図 8.4 (a) の状態数は 3 です。状態数は異なりますが、しかし両者の受理言語（受理する記号列の集合）は同じであることが確認できます。常にそうとは言えませんが、上に述べた機械的な変換では変換後のオートマトンは状態数が多い **冗長な構成に** なりがちです。

冗長な状態数の多い DFA を状態数最小の DFA に変換する方法は別に知られていますが、かなり複雑な手順を必要とするため、この講義テキストでは省略します[2]。

8.3　DFA と NFA の等価性

6 章に述べたように、DFA（決定性オートマトン）は NFA（非決定性オートマトン）の特殊系（非決定的動作が全くない非決定性オートマトン）です。よって DFA の受理言語は全て NFA で表現できます。

この章で述べたように、NFA は DFA に変換できますから NFA の受理言語は全て DFA で表現できます。

以上の事実から、DFA と NFA は等価な表現力を持っていると言えます。すなわち、ある形式言語（記号列の集合）を NFA で表現できるならば、必ず同じ言語を DFA で表現することができ、逆にある形式言語を DFA で表現できるならば、必ず同じ言語を NFA で表現することができることを言います。

8.4　DFA と NFA の違い

ところで、6 章では NFA について「この拡張によって、オートマトンの表現力はさらに高まります」と述べました。そこでいう「表現力」は、上の**本質的な**表現力とは異なり、言語を表現する際の記述の容易さ、簡潔さ、分かりやすさのことです。これについては 6 章で述べたように、やはり NFA が DFA よりも優れています。

逆に DFA が NFA よりも優れている点は、実行方式の単純さです。NFA をそのまま実行する場合、クローンを作る必要があります[3]。しかしクローンを作る作業はコンピュータには大きな負荷であり、高速実行は望めません。DFA の場合には単に状態を遷移するだけですから、余計な負荷はなく、高速実行が可能です。そこで、NFA を高速実行させたい場合には、それを一旦、DFA に変換し、DFA として実行させる方式も用いられます。

[2] たとえばこの分野の古典的名著である、J. ホップクロフト (著), J. ウルマン (著), R. モトワニ (著), その他：オートマトン言語理論 計算論 (1)、サイエンス社などに載っています。

[3] あるいは（詳細は述べませんが）クローンに相当する機能で代替する必要があります。

8.5　練習問題

1. 6 章の章末問題の正解の NFA を DFA に変換しなさい。

1

1

第9章　正規表現

有限状態オートマトンの表現する言語（記号列の集合）は、ある種の数式として表現できることが知られています。それを **正規表現** [1]（regular expression）と呼びます。

「ジュースが出てくる記号列の集合」を思い出しましょう。これは集合記法で以下のように表現できました。

$$L_J = \{uvw \mid u \in \{\mathrm{H}, \mathrm{F}, \mathrm{R}\}^*,\ v \in \{\mathrm{H}, \mathrm{FF}\},\ w \in \{\mathrm{H}, \mathrm{F}\}^*\}$$

正規表現を用いるならば、これを

$$(\mathrm{H}|\mathrm{F}|\mathrm{R})^*(\mathrm{H}|\mathrm{FF})(\mathrm{H}|\mathrm{F})^*$$

と表わすことができます。前者に比べ後者の方が簡素な表現になっており、取り扱いに便利です。この便利さから正規表現は様々な分野に利用されています。最も実用的な例として、多くのワードプロセッサ、エディタの文字列検索処理では検索すべき文字列を正規表現で表わすことができます。理論的には、正規表現は有限状態オートマトンと等価な表現力を持ちます。

この章では正規表現を紹介し、正規表現を用いて記号列を表わす例をいくつか示します。また、有限状態オートマトンと正規表現の関係性を解説します。

9.1　定義

正規表現の記法にはいくつかの流儀がありますが、ここでは最も一般的に知られている記法を用いて正規表現を紹介します。

1. まず、アルファベット Σ から作られる記号列はそれ自体で正規表現であって、その記号列そのものを表すものとします。自動販売機の例で言えば、記号列 HFH は正規表現と見なすことができます。

2. 記号 ε は正規表現であって、空列を表すものとします。

3. 記号 $\emptyset$ は正規表現であって、どんな記号列にもマッチしないと仮定します。形式言語としては空集合に対応します。

[1] 正規表現を**正則表現**とも呼びます。いずれも regular expression の訳語です。

4. 新たに演算子 | を導入し、r_1、r_2 を正規表現とするとき、$r_1|r_2$ も正規表現であるとします。この表現は r_1 の表す記号列と r_2 の表す記号列の**和集合**を表します。直感的には「r_1 または r_2」などと読み下すことができます。たとえば、HF|RR は、「HF **または** RR」を表します。

5. 括弧 () を導入します。r を正規表現とするとき、r を括弧で囲んだ (r) も正規表現です。括弧 () は、演算子の結合順序を変更したり、明示するために用いるだけであり、新しい記号列を表す訳ではありません。ただし、後節で紹介する Microsoft Word の正規表現では、() の有無で表す記号列が変わる場合がありますので、注意してください。たとえば (HFH) は正規表現とあって、HFH を表します。

6. r_1、r_2 を正規表現とするとき、それらを連接した r_1r_2 も正規表現であって、r_1 の表す記号列と r_2 の表す記号列の連接とします。たとえば、HFH(HF|RR) は、「HFH の後に HF または RR が続く」記号列を表します。

7. 新たに後置演算子 * を導入し、r を正規表現とするとき、r^* も正規表現であって、「r の **0 回以上の繰り返し**」を表すとします。0 回も含まれることに注意してください。0 回の場合、空列を意味します。また、回数に上限がないことにも注意してください。たとえば、$(\mathrm{HF})^*$ は、ε、HF、HFHF、HFHFHF、HFHFHFHF、... などの無限個の記号列を表します。

なお、正規表現に現れる演算子の **優先順位** は以下のように仮定します。

$$(\ )\ >\ *\ >\ \text{連接}\ >\ |$$

たとえば、

$$(\mathrm{HF})^* = \varepsilon,\ \mathrm{HF},\ \mathrm{HFHF},\ \mathrm{HFHFHF},\ \mathrm{HFHFHFHF},\ ...$$

ですが、() がない場合、F のみが繰り返されるため、

$$\mathrm{HF}^* = \mathrm{H},\ \mathrm{HF},\ \mathrm{HFF},\ \mathrm{HFFF},\ \mathrm{HFFFF},\ ...$$

となります。また、以下：

$$(\mathrm{H}|\mathrm{F})^* = \varepsilon,\ \mathrm{H},\ \mathrm{F},\ \mathrm{HH},\ \mathrm{HF},\ \mathrm{FF}, \mathrm{HHH},\ ...$$

は、「H または F が 0 回以上繰り返す」ことを表しますが、() がない $\mathrm{H}|\mathrm{F}^*$ では「H または、F のみが 0 回以上繰り返す」ため、

$$\mathrm{H}|\mathrm{F}^* = \mathrm{H},\ \varepsilon,\ \mathrm{F},\ \mathrm{FF}, \mathrm{FFF},\ ...$$

となります。

ところで、演算子 * は 3.3 節（34 ページ）において Σ^* として既に登場しています。その * はここの * と同じ意味を持ちます。というのも、上に述べたように正規表現では * は 0 回以上の繰

り返しを表します。そうすると、Σ^* はアルファベット Σ が 0 回以上繰り返す記号列（の集合）を表すと考えることができます。その解釈は 3.3 節での Σ^* の定義 – 任意の記号列の集合 – と一致します。

9.2　正規表現の例

この節では、$\Sigma = \{a, b\}$ と仮定し、いくつかの正規表現の例を示します。これらの例は 4.3 節の 11 個の例題の言語にそれぞれ対応します。比較することで、オートマトンと正規表現の関係性が見えてきます。

例 1: $(a|b)^*$

これは Σ 上の **任意の記号列** を表す正規表現です。何故ならば、a または b が 0 回以上繰り返せば、どんな記号列を表現できるからです。

例 2: ε

定義から空列です。

例 3: $\emptyset$

定義から如何なる記号列も表しません。

例 4: a

アルファベットに属するひとつの記号 a も正規表現であり、長さ 1 の記号列 a そのものを表します。

例 5: aba

長さが 3 の記号列 aba を表します。

例 6: $\varepsilon|b|(a|b)(a|b)(a|b)^*$

a 以外の記号列を表します。例 4 の補集合に相当します。$(a|b)(a|b)(a|b)^*$ は長さが 2 以上の任意の記号列を表すことに注意してください。

例 7: a^*

特定のパターンが **繰り返す** ような記号列を表すことは正規表現の最も得意とするところです。これは a だけからなる記号列を表します。

例 8: $a^*|b^*$

a だけからなる記号列または b だけからなる記号列を表します。

例 9: $(ab)^*$

a と b が交互に現れる列 $(ab)^n = abab...ab$ を表します。

例 10: $(b^*ab^*a)^*b^*$

これは、記号 a と b の出現順序は問わず、記号列の中に現れる a の数が必ず偶数（0 は偶数とします）であるような列、たとえば ε、bbb、aab、$abba$、... を表わします。何故ならば、まず正規表現 $(aa)^*$ は a が偶数個現れる a のみからなる列を表します。記号列中に b は任意個任意の場所に現れて構わないので、$(aa)^*$ の中に b^*（0 個以上の b を表します）を配置すると、表題の正規表現になります。

例 11: $(a|b)^*aa$

この正規表現は末尾が記号列 aa で終わるような記号列を表します。何故ならば、$(a|b)^*$ は任意の記号列を表わし（例 1 参照)、その後ろに aa が連接されているからです。

考察　列の途中に aa を含むような記号列の集合を表す正規表現を示しなさい。

例 12: 正規表現で表すことができない言語

この講義テキストでは証明する余裕はないのですが、正規表現は有限状態オートマトンと同じ表現力を持ちます。つまり、正規表現で表すことができる言語は有限状態オートマトンで表すことができ、その逆も成り立ちます。よって言語 $\{a^nb^n \mid n \geq 0\}$ を表すような正規表現は存在しません。

あえて言えば、正規表現 a^*b^* がこの言語に近いのですが、 a^*b^* では a の個数と b の個数は異なっても構いません。正規表現では、部分記号列どうしの長さについて互いに制限を加えることはできないからです。

さて、正規表現 r_1、r_2 が同じ記号列を表すならば、それらは互いに **等価である** （equivalent）と言います。

図 9.1: Word の「高度な検索と置換」のウインドウ

たとえば任意の正規表現 r_1、r_2 について、$r_1|r_2$ は $r_2|r_1$ と等価です。

考察　$\Sigma = \{a, b\}$ とするとき、以下のそれぞれ二つの正規表現が等価であるか否か答えなさい。

(1) $a|b$ と $b|a$、　(2) $a|\varepsilon$ と $a|\emptyset$、　(3) $a^*|\varepsilon$ と $a^*|\emptyset$、　(4) aa^*b と a^*ab、

(5) $(aa)^*$ と a^*a^*、　(6) $(a|b)^*$ と $(a^*b^*)^*$

9.3　正規表現の応用例

記号列を文字列に置き換えて考えると、正規表現は文字列のパターンを表現するために便利です。実際、多くのワープロやエディタには正規表現を用いた文字列検索機能が用意されています。ここでは Microsoft Word（以下、Word）での正規表現の利用方法を紹介します。

Word で正規表現を利用するためには、まずメニューから「高度な検索と置換」を選びます。図 9.1 がその検索ウインドウです。このウインドウの左下（図中の赤い矢印）をクリックし、詳細設定を開いてください。次に、図 9.2 の通りに、「ワイルドカードを使用する」のみにチェックを入れます。これで正規表現を入力する準備が整いました。

以下は、図 9.2 の「検索対象：」の欄に入力できる正規表現のルールです。

1. まず、通常の、正規表現を用いない文字列は全て正規表現としても検索可能です。

   たとえば検索対象に 123 と入力すると、ファイル中から 文字列 123 を検索できます。

2. カギ括弧 [ ] は、前節の演算子 | に相当するもので、これによって「または」の検索が可能です。以下のようなヴァリエーションがあります。

   (a) $[a_1a_2...a_n]$ は、文字 a_1、a_2、...、または a_n を表します。

   たとえば [123] は、1、2、または 3 のいずれかの文字を表します。よって、0[123]4 は、014、024、または 034 のいずれかも文字列を表すことになります。

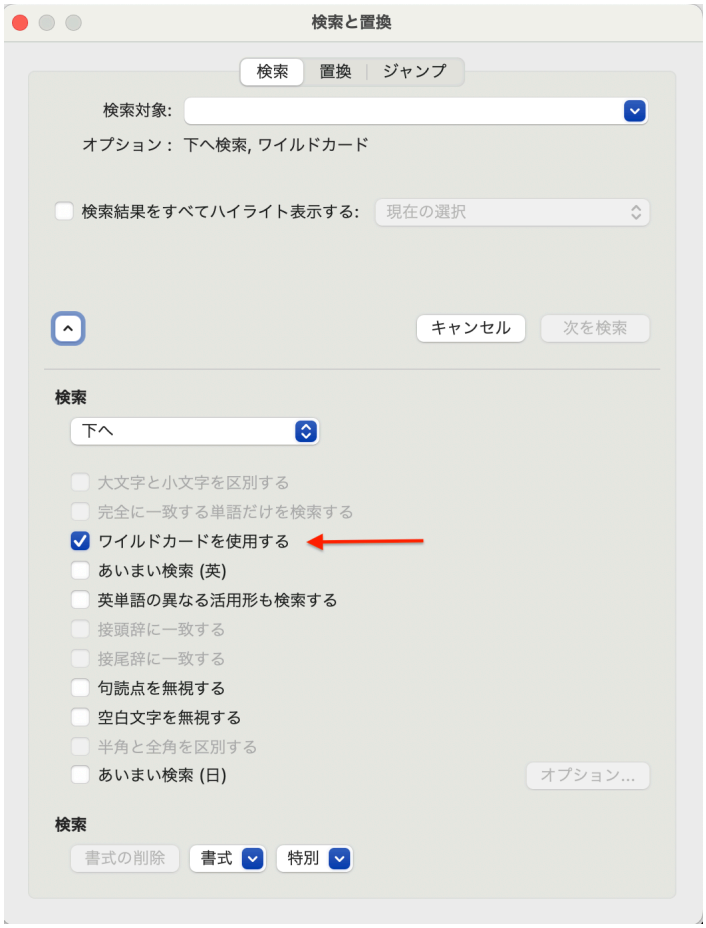


図 9.2: Word の検索ウインドウ

(b) [a_1-a_n] は、文字値が a_1 以上、a_n 以下のいずれかの文字を表します。ただし各文字の文字値および文字値の大小順序は別途定められています。文字値の大きさが $a_1 < a_2 < \ldots < a_n$ ならば、[a_1-a_n] は [$a_1a_2\ldots a_n$] の省略記法に相当します。

たとえば [1-5] は [12345] の省略記法ですから、1、2、...、または 5 のいずれかの文字を表します。よって 0[1-5]6 は、016、026、...、056 のいずれかの文字列を表すことになります。なお、1 の文字値は 5 の文字値よりも小さいため、意図した通りに動作しますが、0[5-1]6 は動作しません。[a-g] なども同様です。文字値の詳細について省略します。

Word の正規表現では「01 または 23」のような長さ 2 以上の文字列の列挙 – 前節の表現では 01|23 と表現できます – に対応する検索はできない制約があります。また、[ ] の入れ子、たとえば [0[12]]（0 または 1 または 2、前節の表現では 0|(1|2)）も Word の正規表現とは認められません。

3. 丸括弧（ ）は、正規表現を明示的に括（くく）るために利用できます。

4. r_1、r_2 を正規表現とするとき、それらを連接した r_1r_2 も正規表現です。

   たとえば `[01][23]` は、0 または 1 の後に 2 または 3 が続く文字列、つまり、02、03、12、13 の 4 種類の文字列を表します。

5. 演算子 $\{m,n\}$ と `@` は繰り返しを表す演算子です。以下のようなヴァリエーションがあります。

   (a) $r\{m\}$ は、r が m 回繰り返すような文字列を表します。

   たとえば `(01){3}` は、文字列 010101 を表します。`[01]{2}` は、0 または 1 が 2 回繰り返す文字列 00、01、10、11 を表します。また、`([01]){2}` は、0 が 2 回、または 1 が 2 回繰り返す文字列 00、11 を表します。Word の正規表現では（ ）の有無によって検索文字列の解釈が変わります（たぶん[2]）。

   (b) $r\{m,n\}$ は、r が m 回以上、n 回以下繰り返すような文字列を表します。

   たとえば `(01){3,5}` は、文字列 010101、01010101、0101010101 を表します。`[01]{2,3}` は、0 または 1 が 2 回以上 3 回以下繰り返す文字列 00、01、10、11、000、001、010、011、100、...、111 を表します。また、`([01]){2,3}` は、0 が 2 回以上 3 回以下繰り返す、または 1 が 2 回以上 3 回以下繰り返す文字列 00、11、000、111 を表します。

   (c) $r\{m,\}$ は、r が m 回以上繰り返すような文字列を表します。

   たとえば `(01){3,}` は、文字列 010101、01010101、0101010101、... を表します。`[01]{2,}` は、0 または 1 が 2 回以上繰り返す文字列 00、01、10、11、000、001、010、011、100、...、を表します。また、`([01]){2,}` は、0 が 2 回以上繰り返す、または 1 が 2 回以上繰り返す文字列 00、11、000、111、...、を表します。

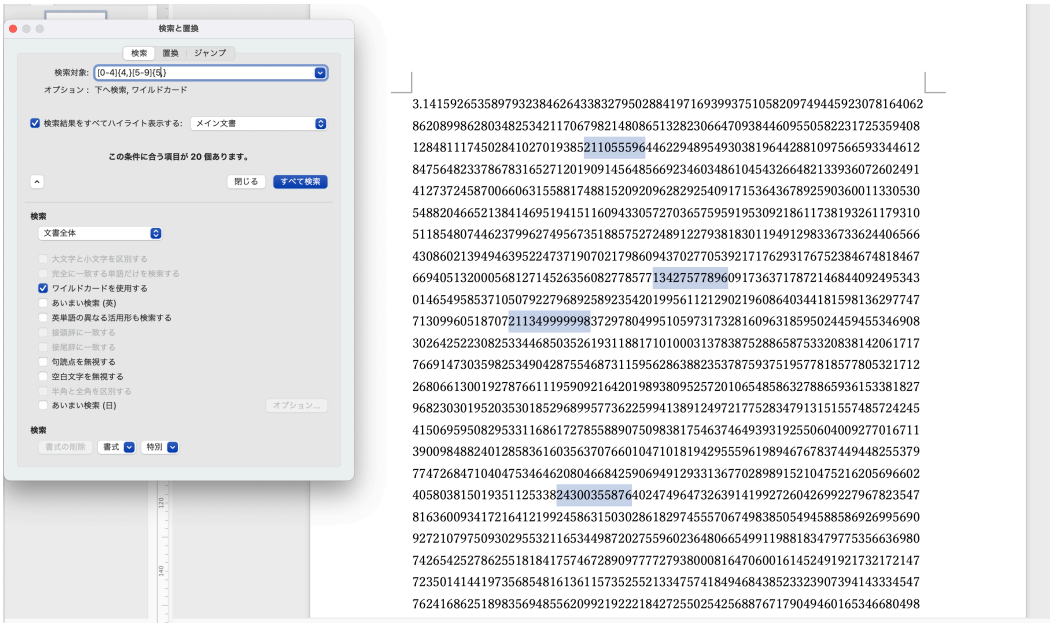
   (d) `r@` は、r が 1 回以上繰り返すような文字列を表します。これは $r\{1,\}$ の省略記法です（たぶん）。

6. 以下は Word の正規表現の特殊記号です。

   (a) `*` は、任意の文字列（空列を含む）を表します。

   たとえば `00*11` は、00 と 11 で囲まれた任意の文字列を表します。このとき、`*` 部分が空列である 0011 も対象文字列です。Word の正規表現での空列は、この `*` を用いてのみ利用可能です（たぶん）。

[2]本来は正確な解説を目指したいのですが、Word の正規表現に関する数学的に厳密な資料が手に入らないため、推定で解説するしかありませんでした。



(b) ? は、任意に 1 文字を表します。

たとえば 00??11 は、00 と 11 の間に任意の 2 文字が挟まれた文字列 00ab11、00bz11、...、002311、... などを表します。

(c) Word の正規表現の演算子に使用されている [ ]、( )、{ }、@、*、? などは正規表現中で特定の役割を演じ、単なる検索文字とは認識されません。これらの記号を文字として検索対象にするには、Windows の場合にはその文字の前に円記号 ¥、MacOS ならばバックスラッシュ記号 \ を付けます。

たとえば　¥*（または *）は、星印（アスタリスク）の文字そのものを表します。

図 9.3 は、Word で 1 万桁の円周率 3.1415...678 の数字列中から、[0-4]{4,}[5,9]{5,} =「0 から 4 のいずれかの数字が 4 回以上現れ、それに続いて 5 から 9 のいずれかの数字が 5 回以上現れる文字列」を全て検索している様子です。1 万桁の中に 20 箇所あることが分かります。正規表現を用いれば、このような少々複雑な文字列であっても検索可能です。最近の多くのワードプロセッサ、エディタには正規表現で検索する機能が装備されています。

考察　1 万桁の円周率の数字列には 777 のように同じ数字が連続して現れる箇所が多数見られます。最も長い数字列の長さとその数字を Word の正規表現の機能を用いて求めなさい。（ヒント：図 9.3 に見えています。）

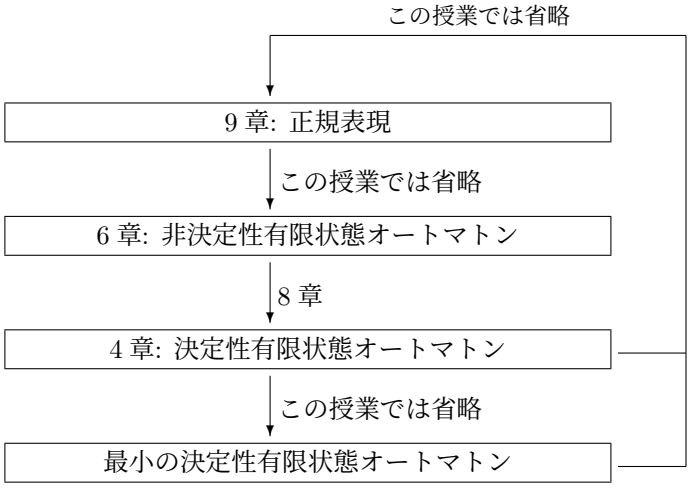


図 9.4: 有限状態オートマトンと正規表現の変換関係

9.4　有限状態オートマトンと正規表現の関係

図 9.4 は、ここまで学んできた有限状態オートマトンと正規表現の関係を相互変換という観点からまとめたものです。

この講義テキストで学んだ決定性有限状態オートマトン（DFA）、非決定性有限状態オートマトン（NFA）、正規表現の間には等価変換が知られています。

この講義テキストでは紹介していませんが、**任意の**正規表現は等価な（言語が等しい）NFA に変換できます。これには、正規表現の「または」の表現 ｜ は NFA の複数の状態への遷移で実装できます。正規表現の 0 回以上の繰り返し * は NFA の閉路として実装できます。結果として、任意の正規表現は NFA に**機械的に**等価変換可能です。

NFA から DFA への等価変換は 8 章で紹介した通りです。

冗長な DFA は、その状態の性質を調べることで、同じ働きをする複数の状態が見つかる場合があります。そのような状態はひとつに合併可能です。合併を繰り返すことで、最終的に最も状態数の少ない DFA に**機械的に**等価変換可能です。

有限状態オートマトンの理論では DFA を正規表現に変換する手法も知られています。

そうすると結局、DFA、NFA、正規表現が相互に変換可能になります。言い換えれば、あるひとつの表現形式で表された言語（記号列の集合）は必ず他の二つの表現形式でも表現可能なのです。その意味から「これらは同じ言語クラスを表す」と言い、特にこの言語クラスを **正規言語**（regular

language）と呼んでいます[3]。正規表現、有限状態オートマトンはいずれも正規言語を表すための数学的体系になっているのです。

　ここまで何度か述べたように、この正規言語には言語：$\{a^n b^n \mid n \geq 0\}$ は含まれません。この言語を表現するには、より複雑なオートマトンが必要となります。それについては 12 章以降で解説します。

[3]あるいは正則言語とも呼びます。

[1]

[1]

第10章　出力付きオートマトン

この章では有限状態オートマトンの応用例を紹介します。

有限状態オートマトンは入力された記号列の **受理／不受理** を調べる機械です。既に見てきたように、オートマトンの構造は決して単純なものではないのですが、しかし、動作結果が二者択一であるため、応用例が限られ、オートマトンが世の中の何の役に立つのか、あるいはコンピュータとどういう関係にあるのか、などという疑問が払拭できない人もいるかもしれません。そこで、オートマトンの **出力動作** を拡張したモデルを紹介します。

10.1　ムーア機械

ムーア機械 （Moore machine）[1]はオートマトンが状態を遷移する毎にあるひとつの記号を出力する機械です。

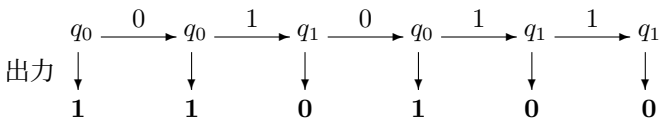
たとえば入力アルファベットを $\{0,1\}$ とし、入力記号列 01011 が与えられたと仮定します。このとき、入力記号列中の 0 と 1 を反転した記号列 10100 を出力するような機械（拡張された DFA）を図 10.1 のように作ることができます。

この機械は次の点で通常の DFA と異なります。

1. 最終状態が定義されません。また、入力記号列が終端することは仮定されておらず、記号が入力される限り、動作を続けます。

2. 各状態に出力記号が付加されており、状態遷移が起る度毎に、遷移先の状態に対応する出力記号が出力されます。また、機械が動作を始めた直後の最初の入力記号を読む前には初期状態の出力記号が出力されます。出力記号は、状態遷移図では状態の真上に記載すると約束します（図 10.1 では、q_0 に **0** が、q_1 に **1** が付加されています）。

よって、図 10.1 の機械に入力記号列 01011 が与えられた場合、状態遷移は以下のように起こり、結果、記号列 110100 が出力されます。



[1]「ムーア」とはこの機械を提案した数学者 エドワード・ムーア の名前です。

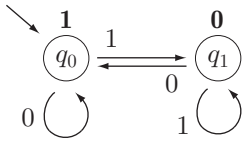


図 10.1: 入力記号を反転するムーア機械

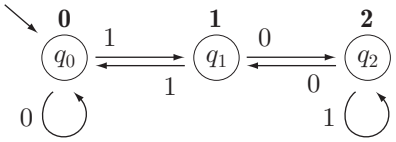
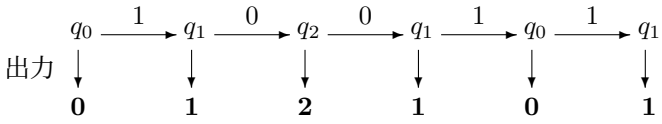


図 10.2: 入力 2 進数を 3 で割った余りを出力するムーア機械

出力記号列 110100 の先頭の 1 文字を無視するならば、これが入力記号列 01011 の bit 反転になっています。

やや複雑な例題として、その時点まで読み込んだ入力記号列を 2 進数 v と解釈したときに、v を 3 で割った余り $v \mod 3$ を出力するムーア機械を図 10.2 のように紹介します。この機械に入力列 10011 を与えた場合の動作は以下の通りです。



この出力が正しいことは、各時点までの入力記号列、その記号列の数値解釈、3 で割った余り、を以下のように列挙してみれば確認できます。

入力記号列	10 進数値	3 で割った余り	出力記号
ε	0	0	**0**
1	1	1	**1**
10	2	2	**2**
100	4	1	**1**
1001	9	0	**0**
10011	19	1	**1**

考察　図 10.2 のムーア機械が正しい答えを出力する理由を述べなさい。（ヒント：数値 v を 3 で割った余りを p（$p = 0, 1, 2$）とするとき、$2v$ を 3 で割った余りは $2p \mod 3$、$2v+1$ を 3 で割った余りは $2p+1 \mod 3$ になることを確認しなさい。）

なお、例題からも分かるように、ムーア機械の動作は決定性オートマトンを前提としており、非決定性オートマトンには馴染みません。

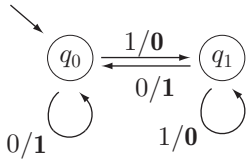


図 10.3: 入力記号を反転するミーリー機械

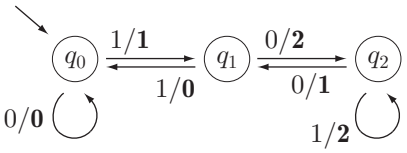


図 10.4: 入力 2 進数を 3 で割った余りを出力するミーリー機械

10.2　ミーリー機械

ミーリー機械（Mealy machine）[2]もオートマトンが状態を遷移するときにあるひとつの記号を出力する機械です。ムーア機械は遷移先の状態で記号を出力しますが、ミーリー機械では遷移の途中で出力するという点が異なります。

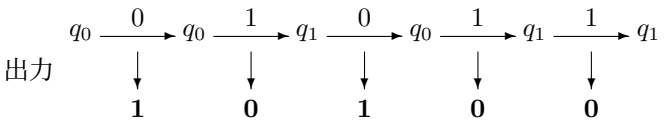
ムーア機械と同様に、入力記号列中の 0 と 1 を反転した記号列を出力するミーリー機械を図 10.3 に示します。

この機械は次の点で DFA や図 10.1 のムーア機械と異なります。

1. 最終状態が定義されません（この点はムーア機械と同様です。）。

2. 各有向枝に出力記号が付加されており、状態遷移が起る度毎に、遷移に用いる枝に付随する出力記号が出力されます。機械が動作を始めた直後の最初の遷移前には如何なる出力も起きません。出力記号 $\mathbf{b}$ は、状態遷移図では有向枝に入力記号 a と共に $a/\mathbf{b}$ と記載すると約束します。

よって、図 10.3 の機械に入力記号列 01011 が与えられた場合、状態遷移は以下のように起こり、結果、記号列 110100 が出力されます。



同様に、読み込んだ入力記号列を 3 で割った余りを出力するミーリー機械は図 10.4 の通りです。

[2]「ミーリー」とはこの機械を提案した数学者 ジョージ・H・ミーリー の名前です。なお、日本語では「ミーリー」ではなく「ミーリ」と記述している文献も多々あります。

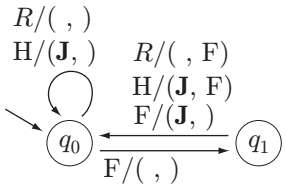


図 10.5: 改良型自動販売機

10.3　改良型自動販売機のミーリー機械

この講義テキストでは旧型の自動販売機を考えてきましたが、ミーリー機械を応用し、終了ボタンを押さなくとも 100 円投入すれば自動的に缶ジュースが出てくる改良された自動販売機を作ることができます。

図 10.5 がその状態遷移図です。出力記号を単なる記号から二つの記号のペア (**J**,F) に拡張しています。ここに、**J** は自動販売機から出てくる缶ジュースが出てくることを表し、F は五十円のお釣りが返ってくることを表します。ペアが (,F) の場合には、缶ジュースは出てきませんが、五十円のお釣りは返却されます。ペア (**J**,) の場合には、缶ジュースは出てきますが、お釣りはありません。ペア (,) の場合には、何も出てきません。

このミーリー機械の各状態遷移の意味はたとえば以下の通りです。

q_0 から H を読んで q_0 へ遷移するときに (**J**,) が出力される： 初期状態で 100 円投入すると、すぐに缶ジュースが出てきて、状態は初期状態に戻ります。

q_0 から F を読んで q_1 へ遷移するときに (,) が出力される： 初期状態で 50 円投入すると、q_1 の状態（自動販売機に 50 円が一時確保されている状態）に遷移します。

q_1 から H を読んで q_0 へ遷移するときに (**J**,F) が出力される： 既に 50 円が自動販売機に確保された状態で 100 円を投入すると、缶ジュースが出てきて、超過分の 50 円が返却されます。状態は初期状態に戻ります。

その他も同様である。

次章ではこのミーリー機械をプログラミングします。

10.4　ムーア機械とミーリー機械の等価性

図 10.1 と図 10.3、図 10.2 と図 10.4 を比べてみてください。ムーア機械とミーリー機械の類似性に気づくでしょう。

ムーア機械から出力される最初の 1 個の記号（機械が動作開始直後に初期状態で出力する記号）を無視するならば、図 10.1 と図 10.3 の機械は同じ入力記号列に対して同じ出力記号列を出力しま

す。図 10.2 と図 10.4 の機械も同様です。このように、最初の 1 個の出力を除いて入力記号列と出力記号列が厳密に一致するとき、それらムーア機械とミーリー機械は **等価** であると言います。このとき、次が成り立ちます。

1. 任意のムーア機械に等価なミーリー機械を作ることができる。

2. 任意のミーリー機械に等価なムーア機械を作ることができる。

1. の具体的な変換手法を与えることは比較的容易です。余裕のある人は図 10.1〜図 10.4 を参考に考えてみてください。2. の具体的な変換手法は少し複雑です。というのも、ムーア機械よりもミーリー機械の方が出力の自由度が高いからです。今、状態数 n のムーア機械、ミーリー機械を考えましょう。ムーア機械の出力は各状態で起きますから、n 種類のバリエーションを考えることができます。対して、ミーリー機械では遷移中に出力が起きますから、$n \times n$ 種類のバリエーションを考えることができます。よって、ムーア機械をミーリー機械へ変換することは容易です。ミーリー機械をムーア機械へ変換するには、$n \times n$ 種類のバリエーションをムーア機械で実現するため、状態数を増やす必要があるからです。

1

1

第11章　出力付きオートマトンのプログラミング

11.1　3で割った余りを出力するムーア機械

図 11.1 が「入力 2 進数を 3 で割った余りを出力するムーア機械」を 2 種類のボタン、2 種類の入力数表示部、猫の有限状態部から作り、実行している様子である。

プログラムは、緑の旗がクリックされてスタートし、直ちに猫のムーア機械は吹き出しに 0 を出力します。その後、[0]、[1] のボタンが押される毎に入力 2 進数にそれまで入力された 2 進数が表示され、同時にその 2 進数を 10 進数に変換した数値も表示され、ムーア機械はその数を 3 で割った余りを吹き出しに出力します。ムーア機械に実行終了はありませんから、ボタンが押される限り、この動作を繰り返します。なお、入力 2 進数の表示は 10 進数の表示を 2 進数に見せかけているだけです。[0]、[1] のボタンを 20 回以上押すと、正しい表示ができなくなります。

以下、各プログラム部分を解説する。

11.1.1　二つのボタンのプログラム

図 11.2 は、二つのボタンのプログラムです。単にメッセージ m0、m1 をブロードキャストするだけです。

11.1.2　猫のムーア機械のプログラム

図 11.4 がムーア機械本体のプログラムです。これは図 11.3（図 10.2 の再掲）の状態遷移図をプログラム化したものです。

たとえばメッセージ m0 を受信したならば、図 11.4 (a) が動作します。以下、その一連の処理内容です。

1. まず、2 進数表示のための変数「入力 2 進数：」を 10 倍します。変数「入力 2 進数：」は実際には 10 進数を保持する変数です[1]が、10 進数を用いて 2 進数に見せかけています。そこで桁をひとつ上げるために 10 倍します。

[1]厳密に言えば、コンピュータ内部では数値は 2 進数で保持されていますが、Scratch ではそれを 10 進数に変換して表示します。よって Scratch のプログラマは数値を 10 進数としてのみ取り扱うことができます。

図 11.1: 入力 2 進数を 3 で割った余りを出力するムーア機械の実行の様子：2 進数 110111101 （10 進数では 445）を入力し、猫が 1 と出力したところ

(a) スプライト [0] のプログラム

(b) スプライト [1] のプログラム

図 11.2: 二つのボタンのプログラム

2. 次に、10 進数表示のための変数「入力 10 進数：」を 2 倍します。変数「入力 10 進数：」は入力された 2 進数を 10 進数値として保持する変数です。そこで桁がひとつ上がる毎に 2 倍します。

3. もし状態（state）が 0 ならば、図 11.3 の通り、状態は変化しません。しかし、0 から 0 へ遷移したと見做し、ムーア機械は状態 0 の出力値 **0** を出力します。

4. もし状態が 1 ならば、状態 2 へ遷移します。ムーア機械は状態 2 の出力値 **2** を出力します。

5. さもなくば（状態が 2 ならば）、状態 1 へ遷移します。ムーア機械は状態 1 の出力値 **1** を出力します。

メッセージ m1 を受信したならば、図 11.4 (b) が動作します。以下、その一連の処理内容です。

1. まず、2 進数表示のための変数「入力 2 進数：」を 10 倍し、1 を足します。

2. 次に、10 進数表示のための変数「入力 10 進数：」を 2 倍し、1 を足します。

3. 以下、同様です。

緑の旗がクリックされたならば、図 11.4 (c) のように、プログラムの初期化を行います。

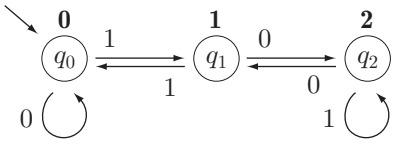


図 11.3: 入力 2 進数を 3 で割った余りを出力するムーア機械（再掲）

(a) メッセージ m0 を受信したとき

(b) メッセージ m1 を受信したとき

(c) 緑の旗がクリックされたとき

図 11.4: それぞれのメッセージに反応するムーア機械のプログラム：

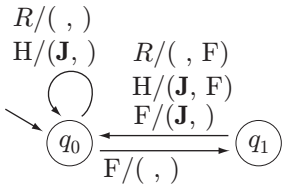


図 11.5: 改良型自動販売機の状態遷移図（図 11.5 の再掲）

図 11.6: 改良型自動販売機の実行の様子

11.2　改良型自動販売機のミーリー機械

Scratch でミーリー機械をプログラミングする例として、前章で紹介した「改良型自動販売機」（図 11.5）を検討します。ここでは、5.2 節（54 ページ）の DFA の自動販売機のプログラムを改造していきます。プログラムの実行の様子は図 11.6 の通りです。缶ジュースが出るときには缶ジュースのスプライトが 2 秒間表示[2]され、50 円のお釣りがあるときには 50 円硬貨が 2 秒間表示[3]されます。

11.2.1　三つのボタンのプログラム

図 11.7 は三つのボタン（図 11.6 上部の [H]、[F]、[R] のスプライト）のそれぞれのプログラムです。これまでと同様、そのスプライトが押されたならば、対応するメッセージをブロードキャストします。

[2]缶ジュースの画像は、フリーの素材サイト「パブリックドメイン Q：著作権フリー画像素材集」からダウンロードしました。
[3]50 円硬貨の画像は、フリーの素材サイト「GAHAG — 著作権フリー写真・イラスト素材集」からダウンロードしました。

(a) スプライト [H] のプログラム

(b) スプライト [F] のプログラム

(c) スプライト [R] のプログラム

図 11.7: 三つのボタンのプログラム

11.2.2　猫のミーリー機械制御部

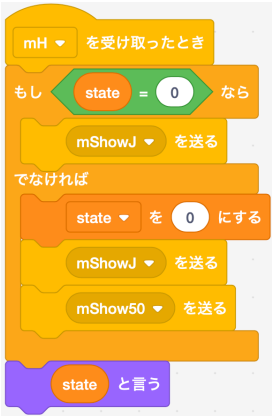
まず、図 11.8 (a) は [H] のボタンが押され、メッセージ mH がミーリー機械に受信されたときの動作プログラムです。図 11.5 を見ながらプログラムを説明すると以下の通りです。

1. もし メッセージ mH を受信したならば
   (a) もし state が 0 ならば、メッセージ mshowJ をブロードキャストします（state は 0 のまま）。
   (b) さもなくば（state は 1 なので）
      i. state を 0 にします。
      ii. メッセージ mshowJ をブロードキャストします
      iii. メッセージ mshow50 をブロードキャストします
   (c) state を表示する。

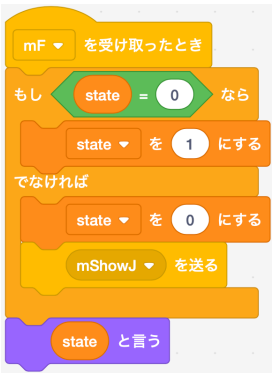
メッセージ mF、mR を受信した場合も同様です。初期化は図 11.8 (d) です。メッセージ mInit をブロードキャストし、state を初期状態番号 0 にし、state を表示します。

11.2.3　缶ジュースのプログラム

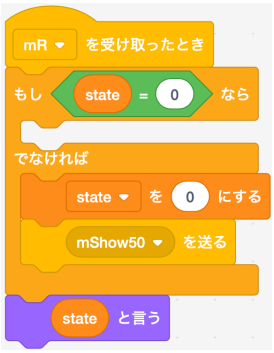
このスプライトのプログラムは図 11.9 の通りです。メッセージ mShowJ を受信したときには自分（缶ジュースの画像）を表示し、そのまま 2 秒間待ち、隠れます。
初期化メッセージ mInit を受信したときには隠れます。



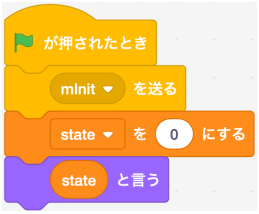
(a) メッセージ mH を受信したとき



(b) メッセージ mF を受信したとき



(c) メッセージ mR を受信したとき



(d) 緑の旗がクリックされたとき

図 11.8: それぞれのメッセージに反応するミーリー機械のプログラム

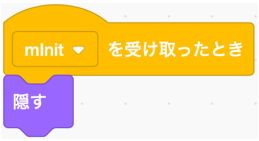
11.2.4　50 円硬貨のプログラム

このスプライトのプログラムは図 11.10 の通りです。受信するメッセージの種類を除いて、缶ジュースのプログラムと同様です。

ミーリー機械は一旦、実行を開始したならば、止まらないことに注意してください。[H]、[F]、[R] を押し続ける限り、動き続けます。

考察　缶ジュースの価格を 110 円とし、100 円硬貨、10 円硬貨の投入、硬貨の返却ができる場合のミーリー機械を設計しなさい。そして実際にプログラミングしなさい。

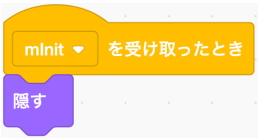
(a) メッセージ mShowJ を受信したとき



(b) メッセージ mInit を受信したとき

図 11.9: 缶ジュースのプログラム

(a) メッセージ mShow50 を受信したとき



(b) メッセージ mInit を受信したとき

図 11.10: 50 円硬貨のプログラム

1

[2]

第12章　プッシュダウンオートマトン

有限状態オートマトンに簡単な記憶装置を追加したオートマトンとして **プッシュダウンオートマトン**（pushdown automaton）がよく知られています。ここに「プッシュダウン」とは「最後に記憶された情報が最初に検索される」ことを意味します[1]。「プッシュダウン」は古い呼び方であり、現在では **スタック**（stack）という呼び方が一般的です。プッシュダウンオートマトンとはスタックを持つ有限状態オートマトンです。なお、以下ではプッシュダウンオートマトンを簡単にPDA と呼ぶこととします。

12.1　スタックとは

スタックを初めて聞いた人は、まず井戸を思い浮かべてください（図 12.1 参照）。

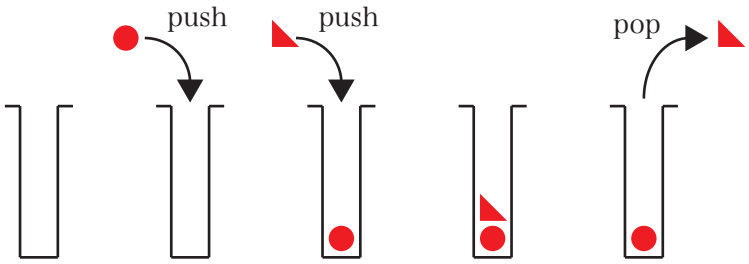


図 12.1: 井戸（スタック）に●と▲を順に push し、次に pop して▲を取り出す場合

その井戸（スタック）に●を入れます（push します）。次に▲を push します。大きさ的に●の上に▲がちょうど重なるものと仮定しましょう。ここで、スタックの一番上に見える物を取り出します。このとき、●は▲の下なので、●を取り出すことはできません。取り出すことができるのは▲です。スタックのことを last in, first out の構造とも言いますが、まさに最後に入れた▲が最初に取り出されています。

このスタックを紙面に図示するとき、縦型の図は無駄にスペースを取るため、以下では図 12.2 のように横に倒して図示するものとします。

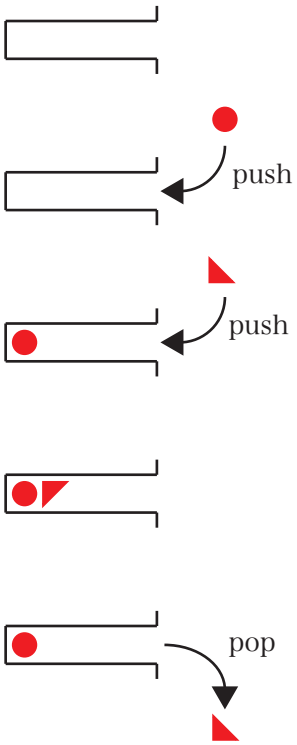


図 12.2: 井戸（スタック）を横にした場合

12.2　スタックを用いた動作

正規言語ではない言語 $\{a^n b^n \mid n \geq 1\}$ を受理する PDA を検討しましょう。

先に述べたように、PDA はスタックを持つ有限状態オートマトンです（図 12.3 参照）。PDA は、有限状態オートマトンと同じように入力記号を読んで状態を遷移しますが、同時にスタック上の頂上（図 12.3 では最右の記号）の記号をひとつ取り出して（pop）、読み、その記号の種類によって遷移先を選ぶことが可能です。また状態遷移時にスタックに記号を積む（push）こともできます。図 12.3 では、スタックの底を左、頂上を右に表しています（下図参照）。

底 ... 頂上

このオートマトンの動作は次の通りです。

1. 三つの状態 q_0、q_1、q_2 を持ちます。q_0 は初期状態です。この状態は記号 a を読み進めている状態です。q_1 は記号 b を読み進めている状態です。q_2 は最終状態です。

2. スタックの底（そこ、bottom）にはあらかじめ記号 # を置いておきます。このような記号を**スタック初期記号**（initial stack symbol）と呼びます。もしスタックから取り出した記号が # ならば、スタックの底が見えたことを意味します。

[1]同様の術語として、FILO（first in, last out）または LIFO（last in, first out）が知られています。

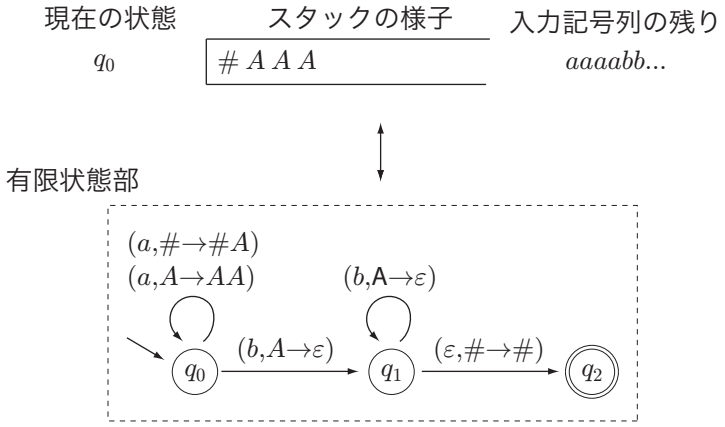


図 12.3: $a^n b^n$ を認識するプッシュダウンオートマトン

3. 状態 q_0 において

   (a) 記号 a を読んだときには、スタックの状態に関わらず、記号 A をスタックに積みます。

   (b) 記号 b を読んだときに、スタックの頂上の記号が A ならば、A をスタックから降ろします。そして q_1 へ遷移します。

   (c) 上の二つに当てはまらない場合には、オートマトンの動作は失敗と見なし、入力記号列は受理されません。

4. 状態 q_1 において

   (a) 記号 b を読んだときに、スタックの頂上の記号が A ならば、A をスタックから降ろします。

   (b) スタックの頂上の記号が $\#$ ならば、入力記号を読まないで、状態 q_2 へ遷移します。記号を読まないで遷移する動作は、空列 ε を読んで遷移する動作とも言い換えることができます。これは前章以前にはなかった新しい動作です。非常に特殊な動作のため、特にこれを ε **遷移**と呼びます。

   (c) 上の二つに当てはまらない場合には、オートマトンの動作は失敗と見なし、入力記号列は受理されません。

5. 入力記号を全て読み終えたときに最終状態 q_2 に遷移したならば、オートマトンの動作は成功です。記号列を受理します。さもなくば失敗です。

このオートマトンの動作のポイントは、読み込んだ入力記号 a と同じ個数の A をスタックに積むことです。そして次に入力記号 b を読むときには A をスタックからひとつずつ降ろしていきます。そして、a と同じ個数の b を読み終えたならば、スタックの頂上には $\#$ が見えているはずです。

つまり、スタックを用いて記号 a の個数を記憶するのです。このような記憶装置は DFA、NFA にはありませんでした。

ところで、ある状態 p において、入力記号が a であり、かつスタックの頂上の記号が Z であるときに、状態 q へ遷移し、スタックから Z を下ろし、新たにスタックに記号 Z_1、Z_2、...、Z_k（$k \geq 0$）をこの順に積むという動作を行うと仮定します。これをオートマトンの状態遷移図に表すために、状態遷移図の p から q への枝に以下のラベルを付けると約束します。

$$(a,\ Z \to Z_1 Z_2 ... Z_k)$$

記号を読まない ε 遷移の場合には以下の通りとします。

$$(\varepsilon,\ Z \to Z_1 Z_2 ... Z_k)$$

スタックに何も積まない場合には以下の通りとします。

$$(a,\ Z \to \varepsilon)$$

図 12.3 の状態遷移図には上のようなラベルを付けています。

図 12.3 に基づいて、実際に入力記号列 $aaaabbbb$ を読み込んでみましょう。

その準備として、オートマトンの実行の様子を的確に知るために、オートマトンの状態 q、スタックの内容 γ、（まだ読み込まれていない）入力記号列の残り u を以下のように並べて表すこととします。

状態	スタック	入力列の残り
q	γ	u

そうすると、例題の PDA で入力記号列 $aaaabbbb$ を読むときの初期のオートマトンの様子は以下の通りです。

状態	スタック	入力列の残り
q_0	$\#$	$aaaabbbb$

ここでオートマトンが記号 a を読み、上の規則 3.(a) によって様子は以下のように変化します。

状態	スタック	入力列の残り
q_0	$\#A$	$aaabbbb$

同じく、残りの三つの a を読むと、変化は以下の通りです。

状態	スタック	入力列の残り
q_0	$\#AA$	$aabbbb$
q_0	$\#AAA$	$abbbb$
q_0	$\#AAAA$	$bbbb$

状態	スタック	入力列の残り
q_0	$\#$	$aaaabbbb$
q_0	$\#A$	$aaabbbb$
q_0	$\#AA$	$aabbbb$
q_0	$\#AAA$	$abbbb$
q_0	$\#AAAA$	$bbbb$
q_1	$\#AAA$	bbb
q_1	$\#\mathrm{AA}$	bb
q_1	$\#\mathrm{A}$	b
q_1	$\#$	
q_2	$\#$	

図 12.4: 図 12.3 のプッシュダウンオートマトンの動作例（まとめ）

次に b を読むと、上の 3.(b) の規則によって、状態は q_1 に遷移し、スタック上の記号 A がひとつ減ります。

状態	スタック	入力列の残り
q_1	$\#AAA$	bbb

さらに 4.(a) の規則を 3 回適用し、以下のように変化します。

状態	スタック	入力列の残り
q_1	$\#\mathrm{AA}$	bb
q_1	$\#\mathrm{A}$	b
q_1	$\#$	

最後に 4.(b) の規則を適用すると、

状態	スタック	入力列の残り
q_2	$\#$	

となり、状態 q_2 において入力列は全て読み終えました。よって記号列 $aaaabbbb$ は受理されました。この一連の動作を図 12.4 にまとめました。

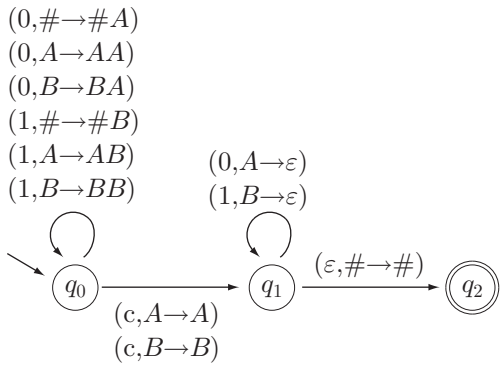


図 12.5: 言語 $\{ucu^R \mid u \in \{0,1\}^+\}$ を受理するプッシュダウンオートマトン

考察　入力記号列が ε、$aaabb$、$aabbb$ のときのオートマトンの動作をそれぞれ調べなさい。

12.3　決定性と非決定性

実は前節で紹介した PDA では非決定的な動作が起こりえます。というのは、ある状態 q において以下の二つの条件のいずれかを満たすとき、その PDA は非決定的な動作をします。

1. 入力記号が a、スタックの頂上の記号が Z であるときの動作が複数定義されている。
2. スタックの頂上の記号が Z であるとき、入力記号 a を読む動作と入力記号を読まない動作（ε 遷移）の両方が定義されている。

逆に言えば、上の二つの条件を共に満たさないとき、PDA は決定的な動作をします。

有限状態オートマトンでは決定性オートマトン（DFA）と非決定性オートマトン（NFA）の受理言語のクラスに **本質的な差** はありませんでした。すなわち、任意の NFA には、それと同じ受理言語を持つ DFA が常に存在します。しかし、PDA ではそのような性質は成り立ちません。非決定性 PDA の受理言語の中には、それと同じ受理言語を持つ決定性 PDA が存在しない場合があります。

これを知るための例題として二つの言語とその PDA を紹介します。

ひとつは、$\{0,1,\mathrm{c}\}$ 上の言語 $\{ucu^R \mid u$ は 0 と 1 からなる長さ 1 以上の任意の記号列 $\}$ です。ここに、u^R は記号列 u の中の記号の並びを左右逆転した記号列を表します。たとえば記号列 0c0, 1c1, 01c10, 10c01, 001c100, 011c110 などがこの言語に含まれます。

この記号列を受理する PDA は図 12.5 の通りです。この PDA の動作は、図 12.3 の PDA と似ています。記号列の前半部 u を読むときには、読んだ記号の情報をスタックに記録していきます。次に記号 c を読んだところで動作が切り替わります。記号列の後半部 u^R を読むときには、スタック上の情報を読み取りながら、読んだ記号とスタック上の情報を突き合わせていきます。

状態	スタック	入力列の残り
q_0	$\#$	011c110
q_0	$\#A$	11c110
q_0	$\#AB$	1c110
q_0	$\#ABB$	c110
q_1	$\#ABB$	110
q_1	$\#AB$	10
q_1	$\#A$	0
q_1	$\#$	
q_2	$\#$	

図 12.6: 図 12.5 のプッシュダウンオートマトンの動作例

図 12.6 に、記号列 011c110 を受理する動作を示します。最終状態 q_2 へ到達できましたから、この記号列は受理されました。

次に、$\{0,1\}$ 上の言語 $\{uu^R \mid u$ は 0 と 1 からなる長さ 1 以上の任意の記号列 $\}$ を検討します。この言語は、上の言語にとてもよく似ていますが、この言語では記号 c が使用されていません。記号列 uu^R は、上の例 ucu^R と同様に左右対称です。しかし、ucu^R では記号列のちょうど真ん中に記号 c がありますが、uu^R にはそのような記号はありません。たったこれだけの違いがオートマトンの性質を劇的に変えてしまいます。

この記号列を受理する PDA は図 12.7 の通りです。このオートマトンは非決定的です。何故ならば、状態 q_0 において、入力記号が 0 であり、スタックの頂上が A であるときの動作が 2 種類あります。また入力記号が 1 であり、スタックの頂上が B であるときの動作も 2 種類あるからです。

図 12.8 に、記号列 011110 を受理する動作（受理に成功する動作例）を示します。最終状態 q_2 へ到達できましたから、この記号列は受理されました。

しかし、図 12.9 のように動作することも可能です。これは失敗した例ですが、では、uu^R という形の記号列について成功するような動作を意図的に行うことは可能でしょうか。

答えは「否」です。入力記号列が uu^R という形ならば、u を読み終えた段階、つまり入力記号列のちょうど半分を読み終えたところで状態 q_0 から状態 q_1 へ遷移すればよいのです。しかし、ちょうど半分を読み終えたタイミングを知ることはできません。というのも、オートマトンは入力記号

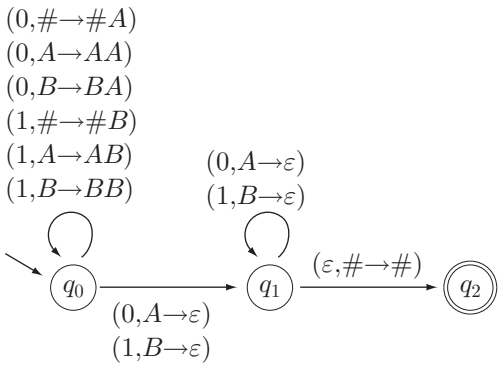


図 12.7: 言語 $\{uu^R \mid u \in \{0,1\}^+\}$ を受理するプッシュダウンオートマトン

を順に読み取っていき、入力記号が無くなったときに動作を終了します。オートマトンの動作が始まる前に入力記号列の全体の長さを知ることはできません。よって、入力列をちょうど半分を読み終えたタイミングを知ることはできないのです。原理的に不可能なのです。

8 章で NFA を DFA に等価変換したように、非決定性 PDA を決定性 PDA に変換できるでしょうか。

これも答えは「否」です。PDA の状態遷移部は有限状態ですが、無限長のスタックを持っていますから、スタックを含めた PDA 全体の状態数は無限です。8 章の方法は状態数が有限であることを巧妙に利用した方法ですから、適用できません。

以上のことは、非決定性 PDA と決定性 PDA が本質的に異なること、すなわち、変換によって非決定性 PDA を決定性 PDA に変換できない場合があることを意味しています。有限状態オートマトン（NFA と DFA）で成り立った非決定性と決定性の等価性（8.1 節を思い出しましょう）は実は異例なことなのです。

状態	スタック	入力列の残り
q_0	$\#$	011110
q_0	$\#A$	11110
q_0	$\#AB$	1110
q_0	$\#ABB$	110
q_1	$\#AB$	10
q_1	$\#A$	0
q_1	$\#$	
q_2	$\#$	

図 12.8: 図 12.7 のプッシュダウンオートマトンの動作例（成功例）

状態	スタック	入力列の残り
q_0	$\#$	011110
q_0	$\#A$	11110
q_0	$\#AB$	1110
q_0	$\#ABB$	110
q_0	$\#ABBB$	10
q_0	$\#ABBBB$	0
q_0	$\#ABBBBA$	

図 12.9: 図 12.7 のプッシュダウンオートマトンの動作例（失敗例）

1

[2]

第13章　プッシュダウンオートマトンのプログラミング

この章ではプッシュダウンオートマトンを Scratch でプログラム化します。

Scratch にはスタックはありませんが、データの並びを扱うための道具立てとして **リスト**（list）が用意されています。この章ではまず、リストについて概説し、それをスタックの代わりに用いて、プッシュダウンオートマトンをプログラミングします。

13.1　リスト

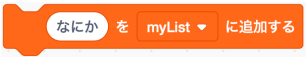
この節ではリストについて紹介します。

13.1.1　リスト型変数の作成

これまで用いてきた変数と同様に、リストは名前を付けて利用可能です。リストを作るには、図 13.1 (a) の矢印のボタンをクリックし、変数を作る場合と同様にリスト名を入力します。ここではリスト名として myList を入力します。そうすると、ブロックパレットに図 13.1 (b) のようなブロック群が新たに現れます。

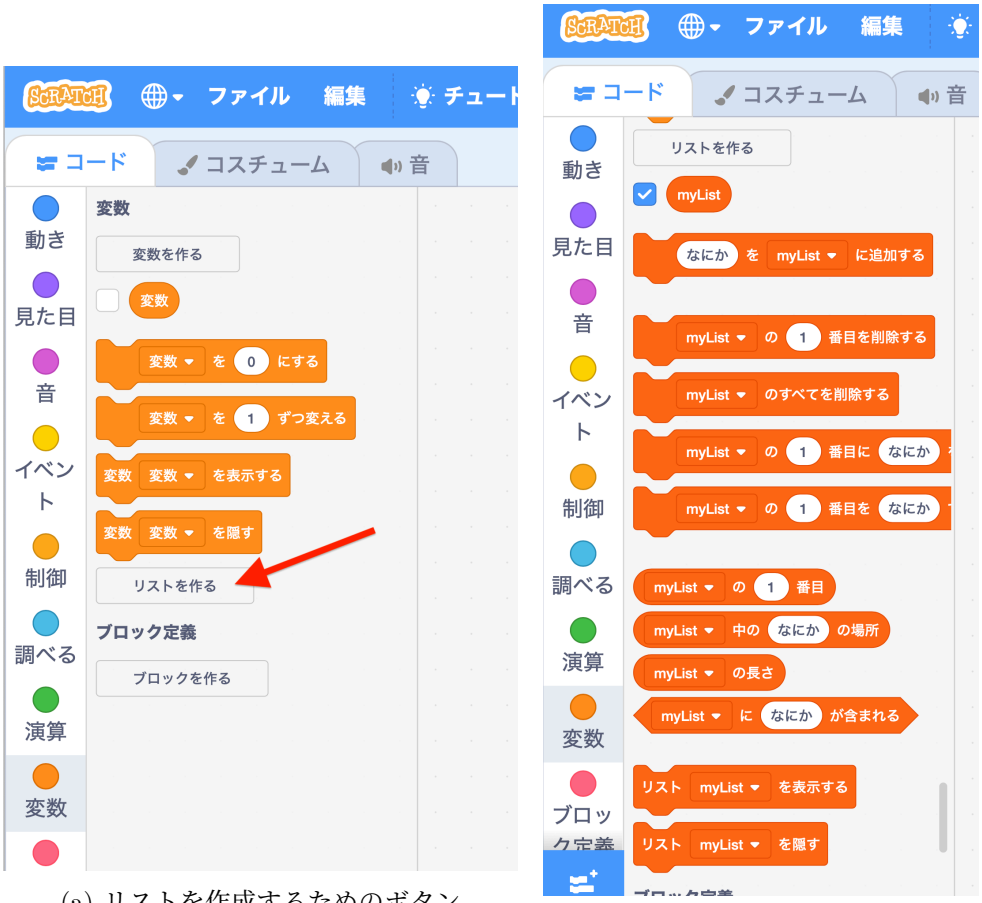
リスト作成直後にはそのリストがステージに図 13.2 のように表示されます。もしこの表示が不要ならば、変数の場合と同様に、ブロックパレット内のリスト名の左隣の青いチェック（図 13.1 (b) の上の方を参照）を外せば、表示は消えます。

13.1.2　リストへのデータの追加

リストはデータの並びです。作成直後にはデータが全く格納されていない空（から）の状態です。これにデータを追加するにはブロック 

（以下、文中では「(なにか) を [myList] に追加する」と表記） を用います。

たとえば図 13.3 (a) は、ブロック「(88) を [myList] に追加する」を実行した直後のリストの様子です。数値 88 がリストに格納されています。次に図 13.3 (b) は、ブロック「(23) を [myList] に追加する」を実行した直後のリストの様子です。リストの末尾（下側）に新しいデータを追加されていることに注意してください。図 13.3 (c) は、ブロック「(AAA) を [myList] に追加する」を実



(a) リストを作成するためのボタン

(b) リスト myList 作成後のブロックパレット

図 13.1: リストを作成する方法

行した直後のリストの様子です。文字列 AAA をリストに混在させることもできます。図 13.3 (d) は、ブロック「(??) を [myList] に追加する」を実行した直後のリストの様子です。?のような特殊記号を含む文字列を扱うこともできます[1]。

13.1.3　リストからのデータの削除

リストから特定のデータを削除するには、ブロック

（以下、「[myList] の (n) 番目を削除する」）を用います。文字通り、リストの先頭から n 番目のデータを削除します。

たとえば図 13.3 (e) は、ブロック「[myList] の (2) 番目を削除する」を実行し、同図 (d) のリストの 2 番目のデータ 23 を削除した直後の様子です。2 番目のデータが削除されたならば、直前の 3、4 番目のデータが新たに 2、3 番目のデータへ繰り上がります。次に図 13.3 (f) は、ブロック「[myList] の ([myList] の長さ) 番目を削除する」を実行し、リストの末尾のデータ（文字列） ??

[1]実はこれまで用いてきた変数にも同様の文字列を代入できます。Scratch では柔軟な処理が可能です。

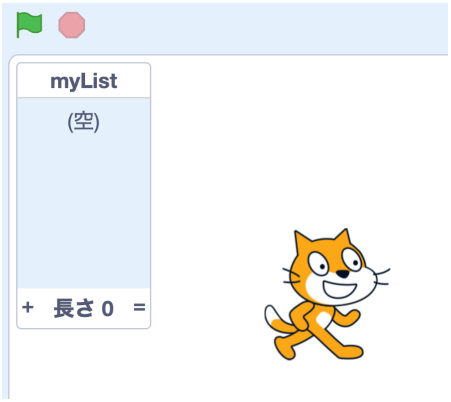
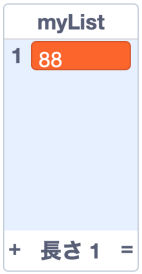
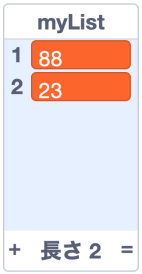


図 13.2: リスト作成直後のステージ

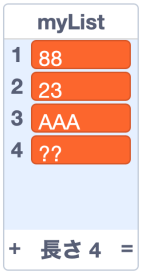


(a) ブロック「(88) を [myList] に追加する」を実行した直後

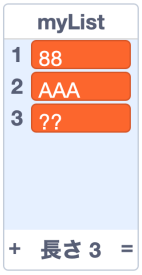


(b) ブロック「(23) を [myList] に追加する」を実行した直後

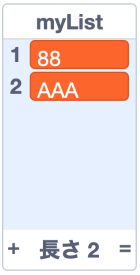
(c) ブロック「(AAA) を [myList] に追加する」を実行した直後



(d) ブロック「(??) を [myList] に追加する」を実行した直後



(e) ブロック「[myList] の (2) 番目を削除する」を実行した直後



(f) ブロック「[myList] の ([myList] の長さ) 番目を削除する」を実行した直後

図 13.3: リストの操作例

を削除した直後の様子です。ここに「[myList] の長さ」とは、図 13.3 (e) の状態では 3 です。よって、「[myList] の (3) 番目を削除する」動作となります。

　なお、リストに格納されている全てのデータを削除するには、

を用いることができます。

　図 13.1 (b) には上に説明した以外にもリストを操作するブロックがあります。まだ説明していないいくつかのブロックは後章で利用しますが、リストをスタックとして用いるには上に挙げたブロックのみで十分です。

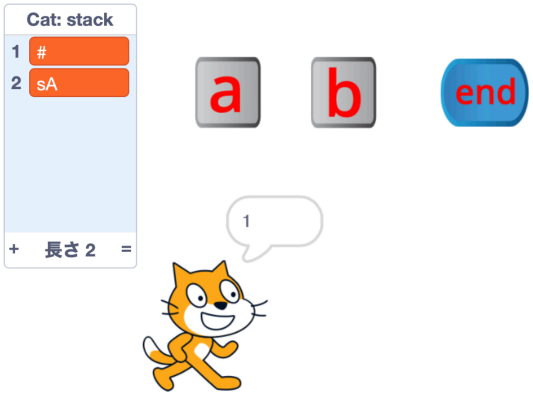


図 13.4: プッシュダウンオートマトンの実行の様子：ボタンを順に [a]、[a]、[b] とクリックした直後

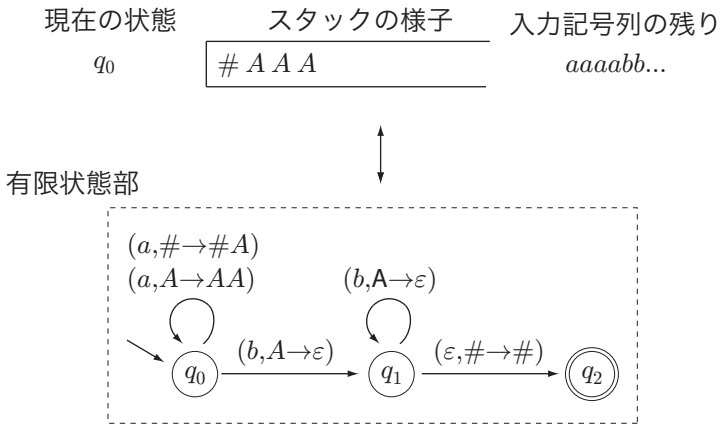


図 13.5: $a^n b^n$ を認識するプッシュダウンオートマトン（再掲）

13.2　$a^n b^n$ を認識するプッシュダウンオートマトン

12 章の最初の例題（図 12.3、図 13.5 に再掲）をプログラミングしてみましょう。

元となるプログラムとして 5.4 節の、a、b を入力記号とする決定性有限状態オートマトンのプログラムを用います。これにスタックを付加し、状態遷移を修正すればよいだけです。

図 13.4 はプログラムの実行の様子です。この図はボタン [a]、[a]、[b] を順に押した直後の様子です。さらに [b]、(end) をクリック入力すれば受理に成功します。

まず、ボタン [a]、[b] のプログラムは図 5.19（63 ページ）と同じです。ボタン (end) は図 5.7 (d)（55 ページ）と同じです。受理/不受理はこれまで同様にグラスで表現することにします。そのプログラムは図 5.13 （59 ページ）と同じです。

猫の有限状態部のスプライトにはあらかじめリスト stack を作成しておきます。stack にはスタック記号 #、sA を格納するものとします。

図 13.6 (a) は緑の旗がクリックされたときの動作です。メッセージ mInit をブロードキャストし（よってグラスのスプライトが初期化されます）、状態（state）を 0 に初期化し、スタック（stack）のデータを全て削除したのちに # を追加します。よってスタックの底に # が配置されます。

図 13.6 (b) は、メッセージ mA をボタンから受信したときの動作です。図 13.5 の状態遷移を参考にしつつ、解説します。

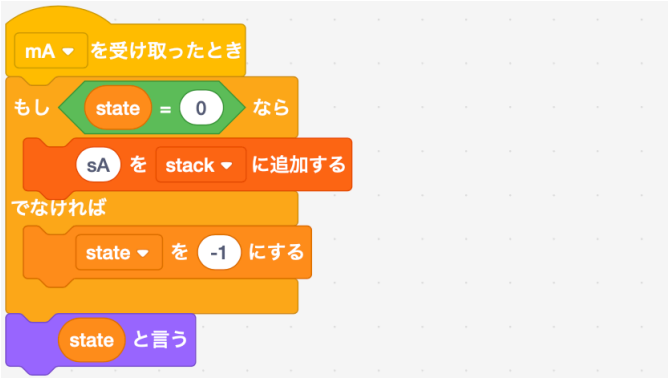
1. もし状態が 0 ならば、スタックの頂上の記号に関わらず、スタックに記号 sA を積みます。これは有限状態部の状態遷移図の図 13.7 の赤い色の部分です。
2. さもなくば（状態が 1 または 2 ならば）、死状態へ遷移します（状態を −1 にします）。
3. 状態（state）を表示します。

図 13.5 では状態 0 の場合、スタック記号が # の場合と sA の場合で分けて記述されていますが、スタック記号はこれ以外になく、かつ二つの場合で動作が同じであるため、これらをまとめて、「スタックの頂上の記号に関わらず、スタックに記号 sA を積む」ように処理を簡略化しました。

次に、メッセージ mB の処理はかなり複雑です。図 13.10 がメッセージ mB をボタンから受信したときの動作です。これも図 13.5 の状態遷移を参考にしつつ、解説します。

1. もし状態が 0 ならば、
   (a) もしスタックの頂上の記号が sA ならば、
      i. スタックからその記号を降ろします。次に状態を 1 へ遷移させます。
      ii. 上の i. の直後に、もしスタックの頂上の記号が # ならば、記号を読むことなく、さらに状態を 2 へ遷移させます。これら一連の動作は図 13.8 の赤い色の部分の動作です。
   (b) さもなくば（スタックの頂上の記号が sA でないならば）、死状態へ遷移します（状態を −1 にします）。
2. さもなくば（状態が 0 でないならば）、
   (a) もし状態が 1 ならば、
      i. もしスタックの頂上の記号が sA ならば、スタックからその記号を降ろします。（上の遷移直後に）もしスタックの頂上の記号が # ならば、記号を読むことなく、状態を 2 へ遷移させます。これら一連の動作は図 13.9 の赤い色の部分の動作です。

(a) 緑の旗がクリックされたとき



(b) メッセージ mA を受信したとき

図 13.6: プッシュダウンオートマトンの有限状態部のプログラム（その 1）

ii. さもなくば（スタックの頂上の記号が sA でないならば）、死状態へ遷移します（状態を -1 にします）。

(b) さもなくば（状態が 1 でないならば）、死状態へ遷移します（状態を -1 にします）。

3. 状態（state）を表示します。

最後にメッセージ mE の処理は図 13.10 の通りです。読み終えたときに状態 q_2 に入るならば受理、さもなくば不受理です。

プログラムを動かした結果は図 13.12、図 13.13 のようになります。プログラムと照らし合わせてください。

同様にして、図 12.5 の、受理言語が $\{ucu^R \mid u \in \{0,1\}^+\}$ であるような決定性 PDA をプログラミングすることも可能です。

さらに 7 章のクローン技術を用いれば、図 12.7 の、受理言語が $\{uu^R \mid u \in \{0,1\}^+\}$ であるような非決定性 PDA をプログラミングすることも可能です。しかしプログラムはかなり複雑になりますから省略します。

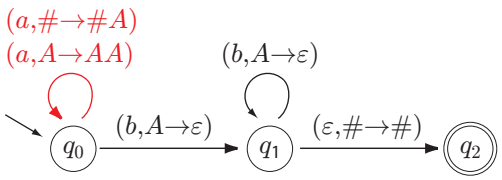


図 13.7: q_0 から a を呼んで遷移する部分

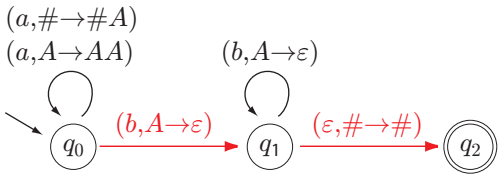


図 13.8: q_0 から b を呼んで遷移する部分

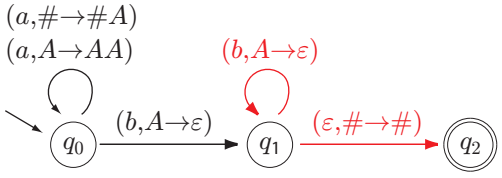
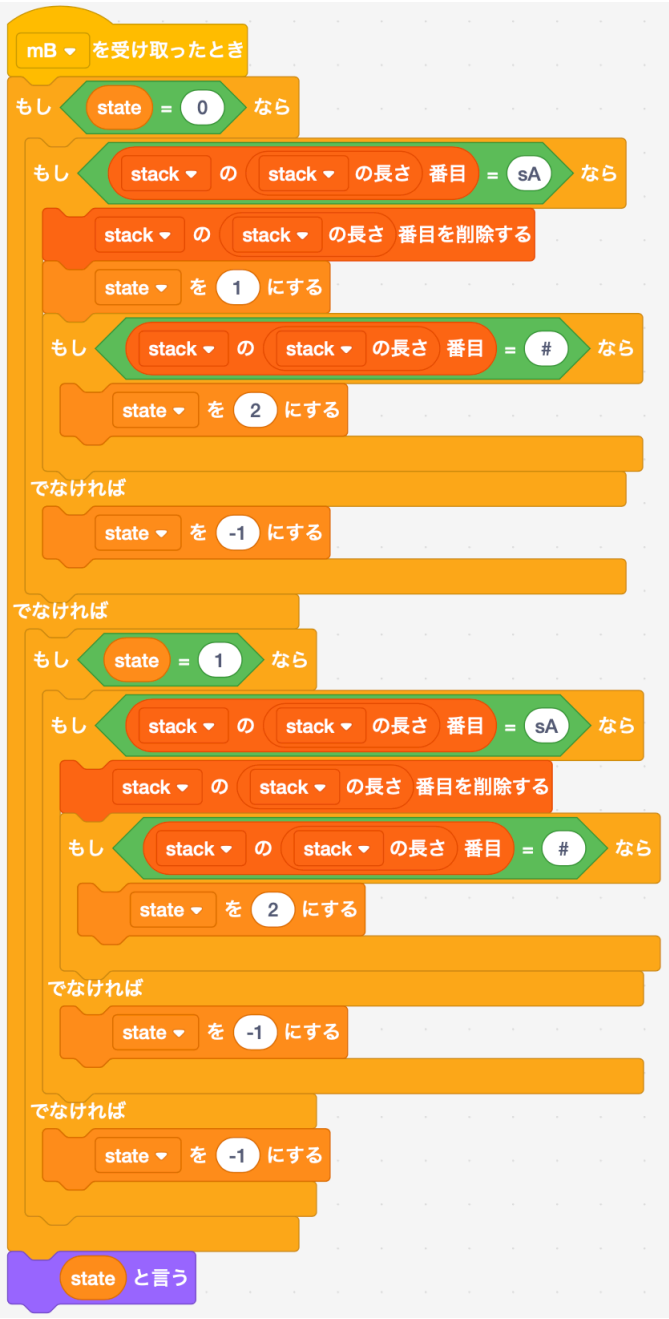


図 13.9: q_1 から b を呼んで遷移する部分

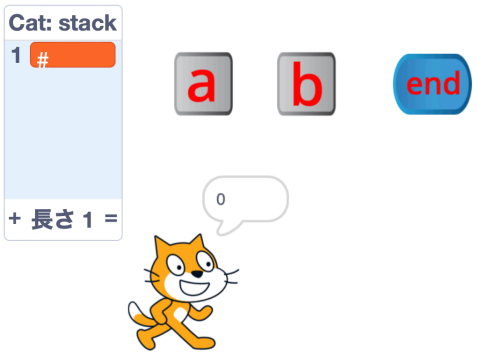


(c) メッセージ mB を受信したとき

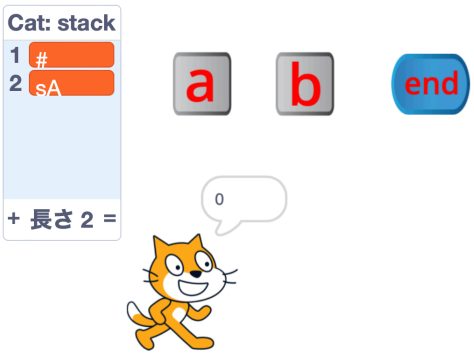
図 13.10: プッシュダウンオートマトンの有限状態部のプログラム（その 2）

(d) メッセージ mE を受信したとき

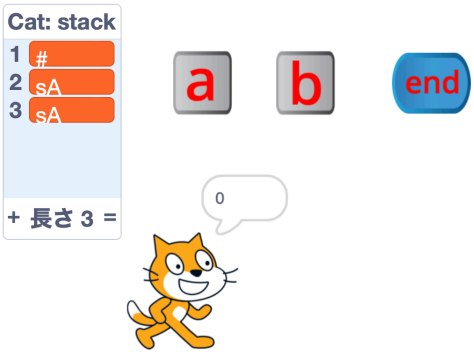
図 13.11: プッシュダウンオートマトンの有限状態部のプログラム（その 3）



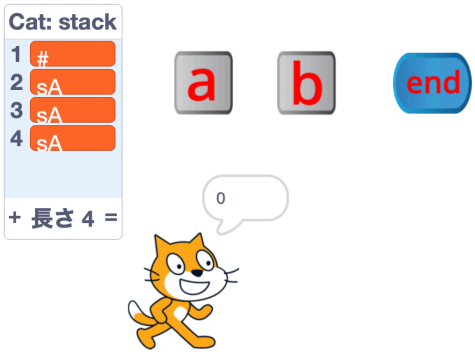
(a) 緑の旗をクリックした直後



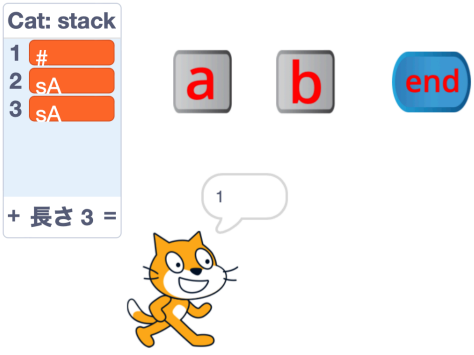
(b) [a] をクリックした直後



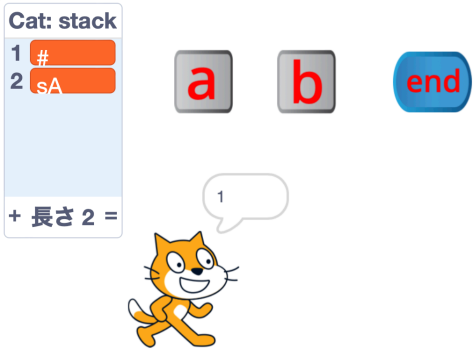
(c) [a] をクリックした直後



(d) [a] をクリックした直後

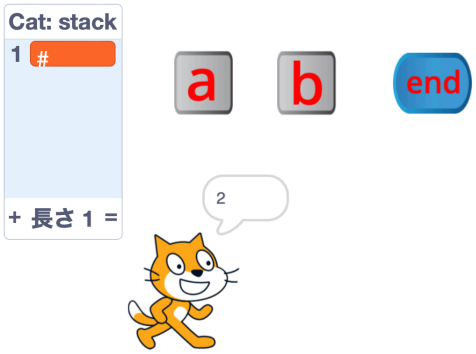


(e) [b] をクリックした直後

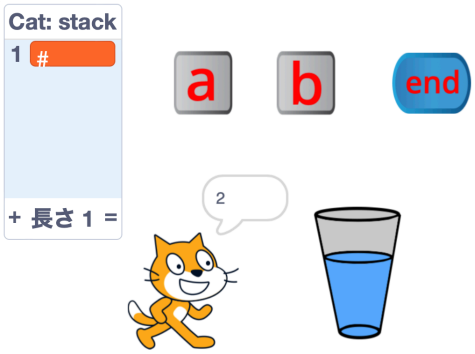


(f) [b] をクリックした直後

図 13.12: PDA プログラムの動作例（その 1、その 2 へ続く）



(g) [b] をクリックした直後



(h) (end) をクリックした直後

図 13.13: PDA プログラムの動作例（その 2）

[1]

[2]

第14章　オートマトンの階層

この講義テキストでは有限状態オートマトンを学び、さらにそれを超える表現力を持つプッシュダウンオートマトンを学びました。プッシュダウンオートマトンよりも表現力の大きなオートマトンも存在します。この章ではそれらを概観します。

14.1　四つの階層

講義テキスト前半で述べてきた有限状態オートマトンと正規表現は正規言語のクラスです。前章、前々章で述べたプッシュダウンオートマトンの言語クラスは、文脈自由言語（context-free language）と呼ばれています。ここに「文脈自由」は形式文法（formal grammar）の分野からの術語です。「文脈自由」の具体的な内容についてはこの講義テキストでは省略します。

この言語クラスの上にはさらに複雑な記号列の集合を表現する言語クラスがあり、文脈依存言語（context-sensitive grammar）と呼ばれています。このクラスのオートマトンは、線形拘束オートマトン（linear bounded automaton）と呼ばれています。さらにその上の言語クラスは句構造言語（phrase structure grammar）と名付けられており、対応するオートマトンはチューリング・マシン（Turing machine）です。これについては次章で解説します。

これらの関係をまとめたものが表 14.1 です。句構造言語よりも複雑な言語クラス、チューリング・マシンよりも表現力の高いオートマトンは見つかっていません。これについては最終章で少し詳しく紹介します。

表 14.1 の四つのクラスを包含関係で表したものが、図 14.1 です。正規言語が最も狭いクラスであり、句構造言語が最も広いクラスです。図にはそれぞれのクラスを特徴付ける記号列を追記しました。

表 14.1: 言語クラスとオートマトンの種類

言語クラス	オートマトン
句構造言語	チューリング・マシン
文脈依存言語	線形拘束オートマトン
文脈自由言語	プッシュダウンオートマトン
正規言語	有限状態オートマトン

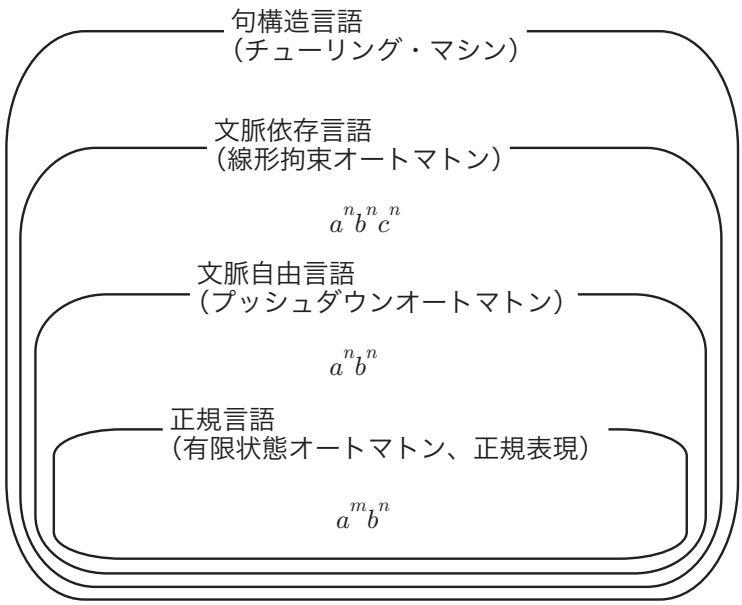


図 14.1: 四つの言語クラスとオートマトンの包含関係

まず、記号列 $a^m b^n$（ただし、m と n は異なってもよい）は、正規言語のクラスに属し、有限状態オートマトンでそれを受理する機械を作ることができました。また正規表現では $a^* b^*$ と表現しました。しかし、このクラスでは $a^n b^n$ となる記号列は表現できません。

記号列 $a^n b^n$ は文脈自由言語のクラスに属し、プッシュダウンオートマトンでそれを受理する機械を作ることができました。しかし、このクラスでは $a^n b^n c^n$ となる記号列は表現できないことが知られています。

記号列 $a^n b^n c^n$ は文脈依存言語のクラスに属し、線形拘束オートマトンでそれを受理する機械を作ることができます。このオートマトンは、チューリング・マシンに制限を加えたものです。

チューリング・マシンは様々なオートマトンの中で最も表現力の高いオートマトンであることが知られています。チューリング・マシンで受理/不受理を判定できる言語は句構造言語のクラスです。ところで、チューリング・マシンをもってしても受理/不受理を判定できない形式言語があることが知られています。

次章で述べるように、チューリング・マシンは現在、私たちが利用しているコンピュータの数学モデルとなっています。そのチューリング・マシンで判定できない言語は、どんなに技を尽くそうともコンピュータでも判定できません。これはコンピュータによる処理の限界を示すものと考えられます。

次章以降では、この講義テキストの最後の大項目であるチューリング・マシンについて、その仕組み、性質、限界について順に解説していきます。

2

3

第15章　チューリング・マシン

チューリング・マシンはイギリス人数学者アラン・チューリングが提案した計算機械です。構造が簡単であるにも関わらず、高い表現能力を持っており、コンピュータで計算可能な問題は全てチューリング・マシンで表現できることが知られています。チューリング・マシンは、現在のコンピュータが世に出てくる前に提案されたにも関わらず、現在のコンピュータとよく似た実行方式であるため、現在のコンピュータのモデルと見なされます。

この授業の最後の話題はこのチューリング・マシンとそれに関連する内容です。

15.1　定義

チューリング・マシンには様々なヴァリエーションがありますが、典型的なチューリング・マシンは **テープ** （tape）、 **テープヘッド** （tape head）、 **有限状態部** （finite control part）の三つからなります（図 15.1 参照）。

テープは **セル** （cell）の並びで構成されています。 左端から右方向に向かって無限個のセルが並んでおり、理論上、テープは半無限長[1]であると仮定します。テープ・ヘッドはテープのセルの上を左右へ 1 回に 1 セル分だけ移動できます。そして、ヘッドが指しているセルの記号を読み込み、かつそのセルに新たに記号を書き込むことができます。有限状態部は有限状態オートマトンと類似しています。有限個の状態を持ち、各状態ではテープ・ヘッドの左右への移動と記号の読み込み/書き込みの動作を制御します。

具体的なチューリング・マシンの動作は以下の通りです。

1. チューリング・マシンの動作開始直前の初期の状態は以下の通りです。
   (a) 長さ n の入力記号列 $a_1a_2...a_n$ が、テープの最左のセルから n 番目のセルまでにあらかじめ書き込まれています。それ以外の全てのセルには **空白文字** （blank symbol）B が書き込まれています。
   (b) テープ・ヘッドの初期位置は最左のセルです。
   (c) 有限状態部の状態は、あらかじめ定められた初期状態 q_0 です。

[1] テープが両側に無限に伸びるような定義（モデル化）もありますが、半無限長の場合と等価です。

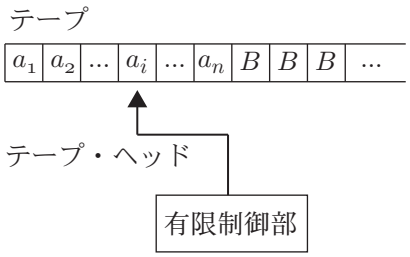


図 15.1: チューリング・マシンの構造

2. チューリング・マシンは、有限状態部の現在の状態 q と、ヘッドが現在、指しているセルの上の記号 a に基づいて以下の動作を行います。この動作は q と a について決定的とします。

   (a) 状態を q から q' へ遷移します。

   (b) ヘッドが指しているセルの内容を a から a' へ変更します。

   (c) ヘッドをひとつ左、またはひとつ右のセルへ移動します。

   これをチューリング・マシンの状態遷移図に表すために、状態遷移図の q から q' への枝に以下のラベルを付けると約束します。

$$(a \to a', L) \text{ または } (a \to a', R)$$

   ここに L はヘッドが左へひとつ移動することを、R はヘッドが右へひとつ移動することを表します。

3. 上の動作を次のいずれかを満たすまで繰り返します。

   (a) 次の動作が定義されていないときには入力列は受理されなかったと判断し、チューリング・マシンの動作を停止します。

   (b) q が最終状態であるときには直ちに入力列は受理されたと判断し、チューリング・マシンの動作を停止します。

チューリング・マシンには様々なヴァリエーションが考えられています。たとえば、非決定的な動作を行うもの、テープを複数個持つもの、1 本のテープに複数のテープ・ヘッドを持つもの、テープが左右に無限長のもの、などなど様々です。しかし、いずれも上の基本的な定義のチューリング・マシンと等価な表現力を持つことが知られています。

15.2　動作例

チューリング・マシンの例として $\Sigma = \{a, b\}$ 上の言語 $\{a^m b^n | \ m > n\}$ を受理する場合を図 15.2 のように設計します。この言語は文脈自由言語です[2]からわざわざチューリング・マシンを用いるまでもありませんが、分かりやすい簡単な例題として取り上げてみます。このチューリング・マシンの設計方針は以下の通りです。

1. この言語の記号列には a が必ず 1 個以上含まれます。そこで、まず最左のセルに書き込まれている a を補助記号 x に書き換えます。もしそのような a が存在しなければ記号列を受理しません。

2. 次にテープ・ヘッドを右へ移動し、最左の b を見つけます。もしそのような b が見つからないならば（すなわち空白記号 B が先に見つかったならば)、a の個数が b の個数よりも多いことになりますから、記号列を受理します。もしそのような b が見つかったならば、その b を補助記号 x に書き換えます。

3. 次にテープ・ヘッドを左へ移動し、最左の a を見つけます。以降の処理は 1. と同じです。

4. 最終状態に入ったならば直ちに受理するため、最終状態からの状態遷移を定義する必要はありません。

実際に動かしてみましょう。入力記号列 $aaabb$ について、チューリング・マシンの動作の様子を図示したものが図 15.3 です。チューリング・マシンは初期状態から動作を始め、テープ上を何度か行き来しながら、解析を進めています。有限状態オートマトンが入力記号列を 1 回しか解析できなかったのに対して、チューリング・マシンでは何度も解析できます。またテープ上の記号を書き換えることで、解析途中の結果をメモしておくこともできます。

15.3　チューリング・マシンの表現力

15.3.1　四則演算

この章の冒頭において、チューリング・マシンはコンピュータで計算可能な問題は全て表現可能と述べました。よって、たとえば正整数の四則演算もチューリング・マシンを用いて実行可能なのですが、ではどのような仕組みで演算を実現すればよいのでしょうか。

前節で見たように、チューリング・マシンの構造は非常に単純です。そのため、四則演算を実現するには少し工夫が必要です。

[2] $m > n$ という制約から、オートマトンは a の現れた個数 m を記憶しておき、b の個数 n はその m よりも小さくなければなりません。この記憶の必要性からこの言語は DFA などでは表現できず、PDA ならば表現できることが分かります。

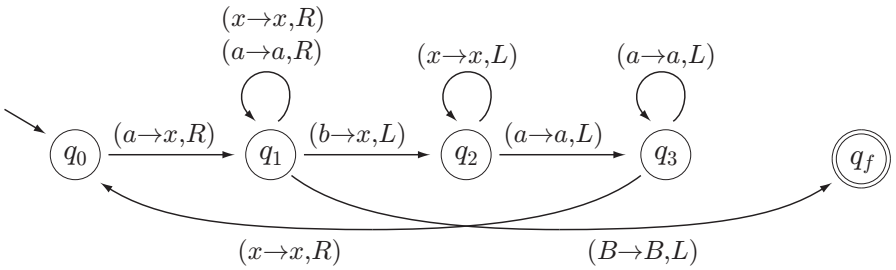


図 15.2: $a^n b^n$ を認識するチューリング・マシンの状態遷移図

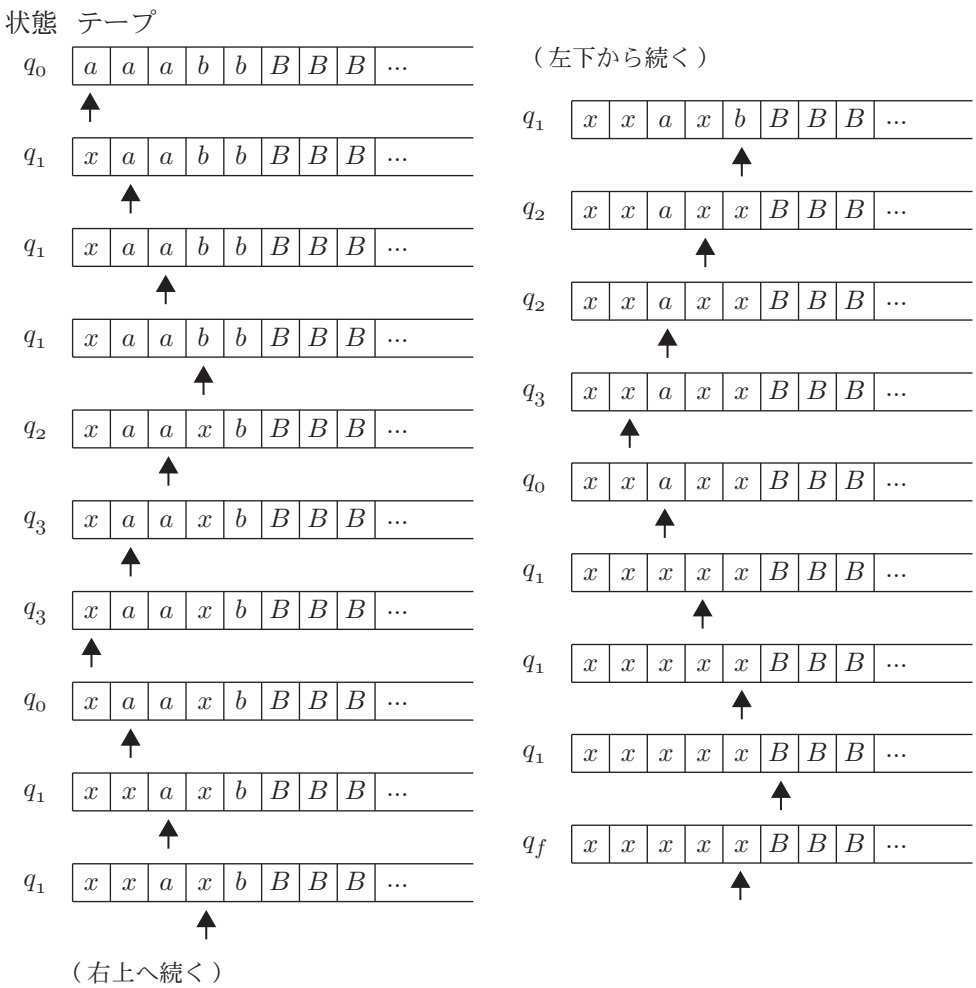


図 15.3: *aaabb* の受理過程

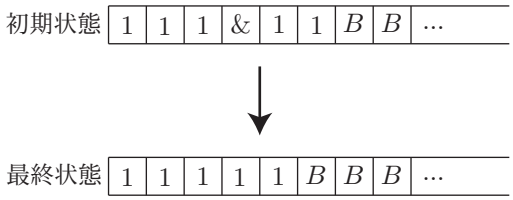


図 15.4: チューリング・マシンによる $3+2=5$ の計算

まず、正整数 n をチューリング・マシンに表現せねばなりません。ここではテープ上に記号 1 を n 個並べて表すと約束します[3]。

そして、m と n の加算のための入力記号列を、左から m 個の 1、区切り記号の &、n 個の 1 を並べて表すと約束します。となれば、加算の実行後には、左から $m+n$ 個の 1 が並ぶようにチューリング・マシンが動作すればよいのです。図 15.4 に、$3+2=5$ に対応するチューリング・マシンの初期状態と最終状態を示します。

考察　図 15.4 のような加算を行うチューリング・マシンを設計しなさい。

減算も同様です。これには、前節の言語 $\{a^m b^n |\ m > n\}$ を受理する例題が参考になります。

乗算 $m \times n$ では、m を n 回加えるように設計します。これには加算を行うチューリング・マシンの動作を利用します。

除算 m/n では、m から n を引いた回数をテープ上にメモしておけばよいでしょう。これには減算を行うチューリング・マシンの動作を利用します。

数の表現形式が私たちの通常の感覚からすれば単純過ぎるため違和感があるかもしれませんが、上のような方法でチューリング・マシンで四則演算を実現することができます。四則演算ができるならば、それを組み合わせることで指数や平方根の計算も可能です。さらに複雑な計算も可能であり、結果としてチューリング・マシンを用いて任意の算術計算を実行可能です。

[3]つまり、1 進法による数の表現を用いる訳です

2

3

第16章　チューリング・マシンのプログラミング

この章では前章のチューリング・マシンをプログラミングします。プッシュダウンオートマトンではスタックをリストで実装しましたが、ここではテープをリストで実装します。

また、プログラムがかなり複雑になるため、処理の**モジュール化**（modularization）を行います。これは、複数のブロック列による処理をひとつのブロックとして定義・利用し、プログラムの読みやすさを向上させる手法です。長大な処理を階層的に整理する際に有効です。

16.1　$a^m b^n \quad (m > n)$ を認識するチューリング・マシン

まず、復習します。タイトルのチューリング・マシンの有限状態部は図 16.1 の通りです。図中の状態 q から状態 q' への枝に

$$(a \to a', X)$$

がついている場合、

- もしテープヘッドの指しているセルの記号が a ならば、そのセルの記号を a' へ変更し、ヘッドを X 方向（$= L$（左）または R（右））へひとつだけ移動する

ことを表します。状態が最終状態 q_f に入ったならば、チューリング・マシンは即時停止して受理とします。動作が定義されていない場合、即時停止して不受理とします。

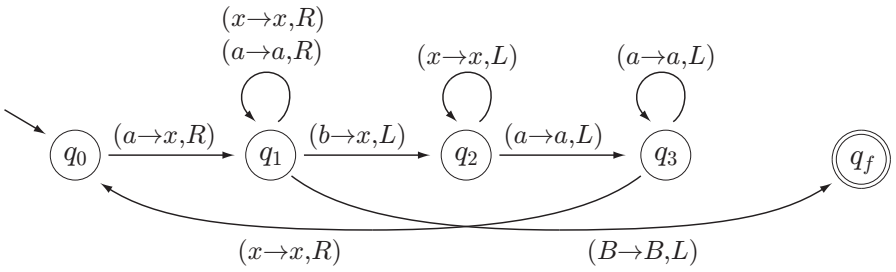


図 16.1: $a^n b^n$ を認識するチューリング・マシンの状態遷移図（図 15.2 の再掲）

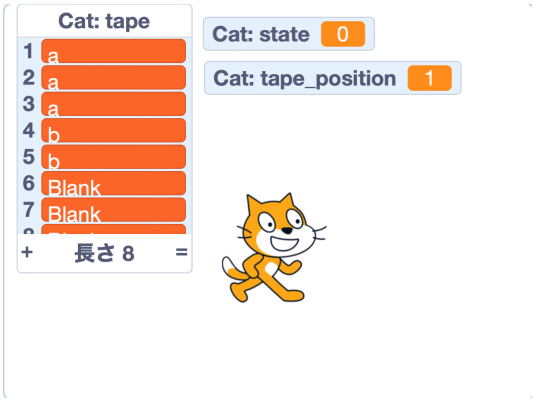


図 16.2: 猫のチューリング・マシンの実行の様子

このチューリング・マシンを Scratch でプログラム化し、実行する直前の様子が図 16.2 です。ステージ左側に表示されているのがテープです。実際にはリストを用いています。テープのセルには上から順に *aaabb* が書き込まれています。このセルを書き換えながらチューリング・マシンは動いていきます。

このチューリング・マシンのプログラムの、トップレベルの記述は 16.3 (a) です。ここにプログラム中のピンクのブロック

（以下、[[init]]）、

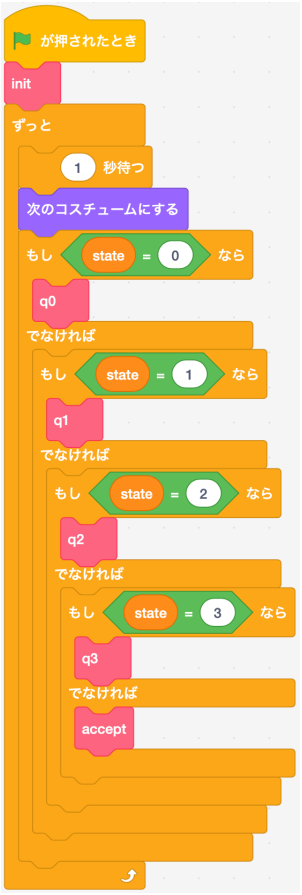
（以下、[[q0]]） などは新たにプログラマ自身が新たに定義したブロックです。これを以下では**ユーザ定義ブロック**（user-defined block）と呼び[1]、文中では [〜] ではなく、 [[〜]] で表すものとします。ユーザ定義ブロックは、既存のブロック（たいていの場合、複数のブロック）を組み合わせたプログラム片をひとつのブロックのように見せかけるものです。これによってプログラムが読みやすくなります。たとえば [[init]] の定義は図 16.3 (b) です。その図の中のブロック [[tape_init]] もユーザ定義ブロックです。その定義は図 16.3 (c) の通りです。

プログラマがブロックを新たに定義するにはブロックパレットの中の一番下（エディタの左下）の「ブロックを作る」ボタン：

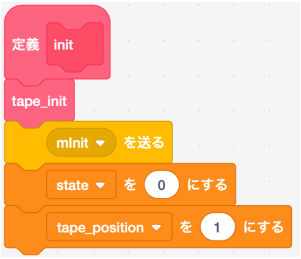
を押して行います。このボタンを押すと、図 16.4 のウインドウが開きます。ブロック名は、変数名、リスト名と同様に自由に決めて構いません。定義されたブロックは通常のブロックと同様に利用可能です。

ブロック [[init]] （図 16.3 (b) 参照）はその名前の通り、初期化を行うブロックとして定義しました。テープの初期化は [[tape_init]] で行い、続けてメッセージ mInit のブロードキャスト、state

[1] Scratch の正式名称は「定義ブロック」のようですが、これでは意味が分かりにくいため、この講義テキストでは「ユーザ」を付けます。



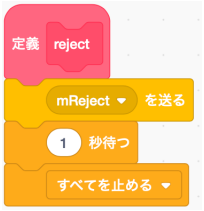
(a) トップレベルのプログラム



(b) 初期化のプログラム

(c) テープの初期化のプログラム

(d) 受理の処理のプログラム



(e) 不受理の処理のプログラム

図 16.3: チューリング・マシンのプログラム（その 1、その 2 へ続く）

図 16.4: 新しいブロックの定義ウインドウ

と tape_position の初期設定を行います。position はテープのヘッドの位置を表す変数です。初期値はテープの先頭を表す整数値 1 です。[[tape_init]] （図 16.3 (c) 参照）では、テープに事前に載せるデータを書き込みます。ここでは、*aaabb* の後に Blank を 3 個[2]載せました。なお、前章では空白記号には Blank ではなく B を用いていますが、他の記号との混乱を避けるため、この章では Blank を用います。

16.3 (a) では初期化の後は無限ループ [ずっと] です。無限ループでは実行がずっと続き、止まりませんが、後に述べるように、このプログラムでは受理/不受理が確定し、その処理を終えた時点で全てのプログラムを強制終了させます。ループの中では

1. まず、1 秒待って（待たないとプログラムの実行が一瞬で終わってしまいます）
2. 猫の表示を少し変え（プログラムが動いていることを示すためです）
3. 状態番号 n（$n = 0, 1, 2, 3$）毎にユーザ定義ブロック [[qn]] の処理を行い、
4. もし n が 0〜3 でないならば、即座に受理動作を行います（その場合、状態番号は f であり、f は最終状態を意味するから）。

受理/不受理の処理を担当するユーザ定義ブロック [[accept]]、[[reject]] はそれぞれ図 16.3 (d)、(e) の通りです。それぞれメッセージをブロードキャストし（そのメッセージはグラスが受信しま

[2] ここで考えるプログラムでは Blank は 1 個以上あれば正しく動作しますが、定義上は Blank はテープ上に無限個、記録されている仮定ですから、とりあえず 2 個を追加しました。

す)、1 秒待って（待たないとグラスの処理が追いつかない可能性があります)、チューリング・マシンのプログラムの全ての動作を停止させます。受理/不受理のメッセージに対するグラスのスプライトの反応は、以前に述べたものから変更していません。図 7.6（81 ページ）の通りです。

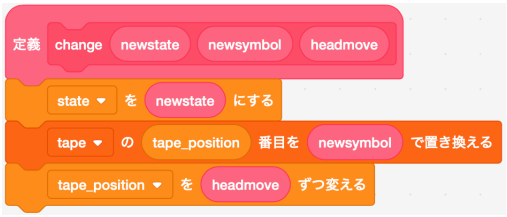
ユーザ定義ブロック [[q0]]、[[q1]]、...、[[q3]] の内容は図 16.5 (b)〜(e) の通りです。

なお、これらのプログラムではユーザ定義ブロック [[change()()()]] を用いています。このブロックの内容は図 16.5 (a) です。ユーザ定義ブロック [[change()()()]] は、他のユーザ定義ブロック [[init]]、[[q0]]、... などと異なり、**引数**（ひきすう[3]）を持ちます。引数は、そのユーザ定義ブロックの引き渡すパラメータのことです。ユーザ定義ブロックで全く同じ処理を繰り返すならば、引数＝パラメータは不要ですが、わずかに異なる処理を行わせたい場合に引数は便利です。ユーザ定義ブロックに引数を追加するには図 16.4 の時点で、左側の「引数を追加：数値またはテキスト」をクリックします。

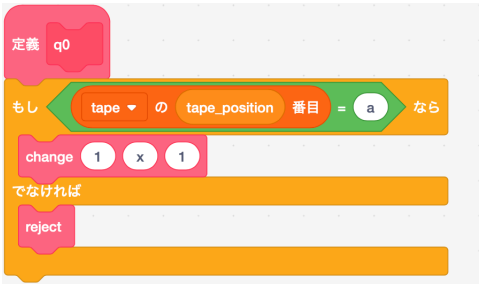
[[change()()()]] は三つの引数を用いてチューリング・マシンの状態を変化させるブロックです。図 16.5 (a) に定義されているように、状態 state を第 1 引数 newstate で更新し、現在のテープヘッド（リスト tape の現在の位置 tape_position）のセルの内容を第 2 引数 new symbol に置き換え、ヘッドの位置 tape_position を第 3 引数 headmove だけ ±1 増減します。

たとえば [[q0]] （図 16.5 (b)）では、もし tape の tape_position の位置のセルの内容が a ならば、 [change(1)(x)(1)] を実行します。さもなくば（tape の tape_position の位置のセルの内容が a でないならば)、次の状態が定義されておらず、すなわちこのチューリング・マシンは入力記号列を受理しないため、[[reject]] を実行します。[[q1]]、[[q2]]、[[q3]] も同様です。それぞれの状態においてテープヘッドの位置のセルの内容によって処理を選択します。

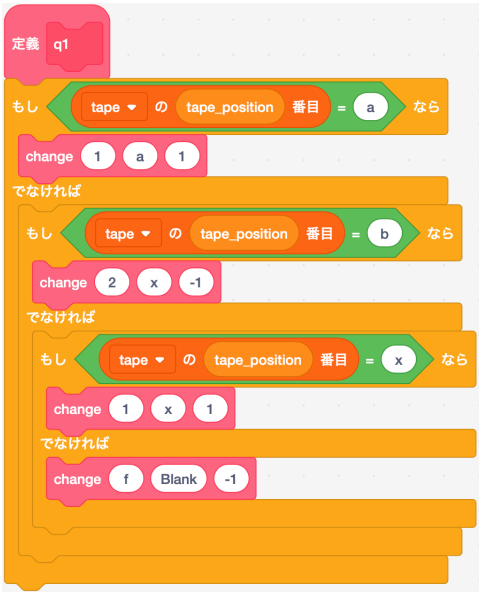
[3] これは「いんすう」とは読まない慣例になっています。特にプログラミングに興味のある人、将来、プログラマになるかもと思っている人は注意してください。



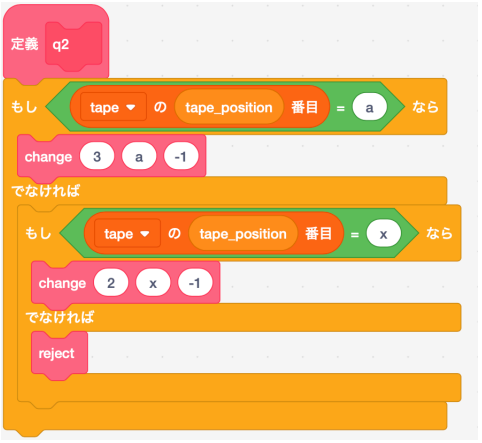
(a) マシンの状態変化を実施する汎用プログラム



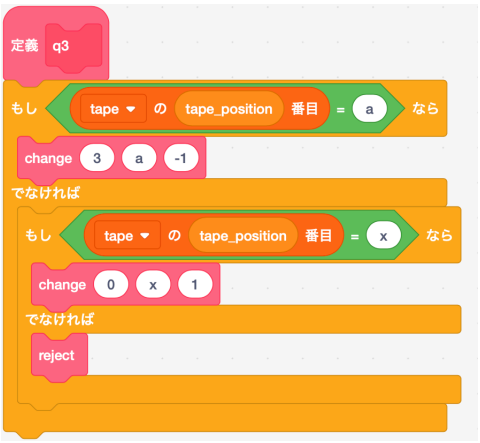
(b) 状態 q_0 での処理プログラム



(c) 状態 q_1 での処理プログラム



(d) 状態 q_2 での処理プログラム



(e) 状態 q_3 での処理プログラム

図 16.5: チューリング・マシンのプログラム（その 2）

表 16.1: 図 16.1 のチューリング・マシンの有限状態部の動作表：左から順に、現在の状態、現在のヘッドの位置のセルの内容、次の時点の状態、次の時点のセルの内容、ヘッドの移動方向

0	a	1	x	$+1$
1	a	1	a	$+1$
1	b	2	x	-1
1	x	1	x	$+1$
1	$Blank$	f	$Blank$	-1
2	a	3	a	-1
2	x	2	x	-1
3	a	3	a	-1
3	x	0	x	$+1$

16.2　もうひとつの $a^m b^n \quad (m > n)$ を認識するチューリング・マシン

ここで述べるプログラムがこの講義テキストの最後の例題プログラムです。

前節のプログラムでは、状態毎、セル記号毎に個別の処理を記述しました。しかし、チューリング・マシンの動作は画一的です。

- 現在の状態、現在のテープヘッドの位置のセルの内容から、
- 次の時点の状態、次の時点のセルの内容、ヘッドの移動方向が決まります。

そうであるならば、その動作ルールをテーブル形式にまとめて表現しておいて、必要に応じて、そのテーブルを参照してチューリング・マシンを動作させることができるはずです。

この節ではこの方針に基づき、前節のチューリング・マシンを前節とは別のプログラムで実装します。

まず、図 16.1 の状態遷移図は表 16.1 のようにまとめることができます。ただし状態遷移が未定義のものは除外しています。さらにこの表のデータを以下のように一列に並べましょう。

0	a	1	x	$+1$	1	a	1	a	$+1$	1	b	2	x	-1	...

表 16.1 の 1 行は 5 個のデータで出来ていますから、上のように 1 列に並べても配置は規則的でデータが混乱することはありません。この節のプログラムでは、上の 1 列のデータをリスト code に格納し、チューリング・マシンはこの code を読みながら動くようにプログラムを構築していくものとします。

もうひとつのチューリング・マシンが Scratch の上で実行を終了した直後の様子が図 16.6 です。ステージ左側にはリスト tape、ステージ中央にはリスト code が表示されています。下の方には変数 tape_position、code_position、state が表示されています。

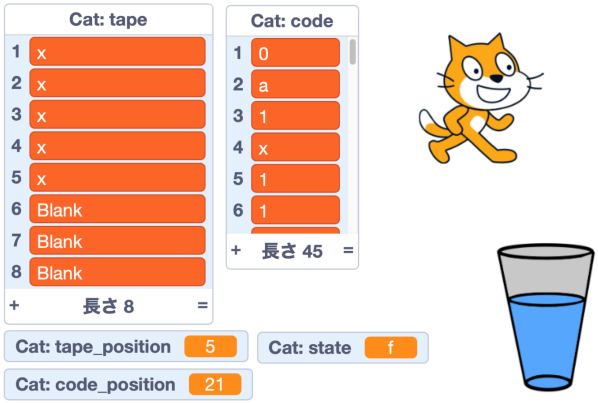


図 16.6: もうひとつの猫のチューリング・マシンの実行の様子

このチューリング・マシンのプログラムの、トップレベルの記述は図 16.7 です。ここにユーザ定義ブロック [[init]] は前節の初期化に加え、リスト code の初期化も行います。そのプログラムは図 16.8 (a) です。[[init]] の中で使用されているユーザ定義ブロック [[code_init]] は図 16.8 (b)、(c) です。45 個のデータをリスト code に格納しています。格納するデータは表 16.1 のものです。ユーザ定義ブロック [[tape_init]] は前節（図 16.3 (c)）と同じです。

もう一度、図 16.7 へ戻ります。初期化が終わった後のプログラムの挙動は以下の通りです。

1. まず、変数 code_position を 1 に初期化します。
2. code_position が code の長さを超えるまで以下を繰り返します。
   (a) もし code の code_position 番目の内容が state に等しく、かつ code の code_position+1 番目の内容がテープヘッドの位置のセルの内容（tape の tape_position 番目の内容）に等しいならば、状態遷移を行う条件に合致します。そこで
      i. state を code の code_position + 2 番目の内容に変更し、
      ii. テープヘッドの位置のセルの内容を code の code_position+3 番目の内容に変更し、
      iii. テープヘッドの位置を code の code_position + 4 番目の内容だけ変更します。
      iv. 上の状態遷移の結果、
         A. もし state が f（＝最終状態）になったならば、即座に受理動作を行い、プログラムの実行を終了します。
         B. さもなくば、次の状態遷移のために code_position を 1 に初期化します。
   (b) さもなくば（code の code_position 番目の内容が state と異なる、または code の code_position+1 番目の内容がテープヘッドの位置のセルの内容（tape の tape_position

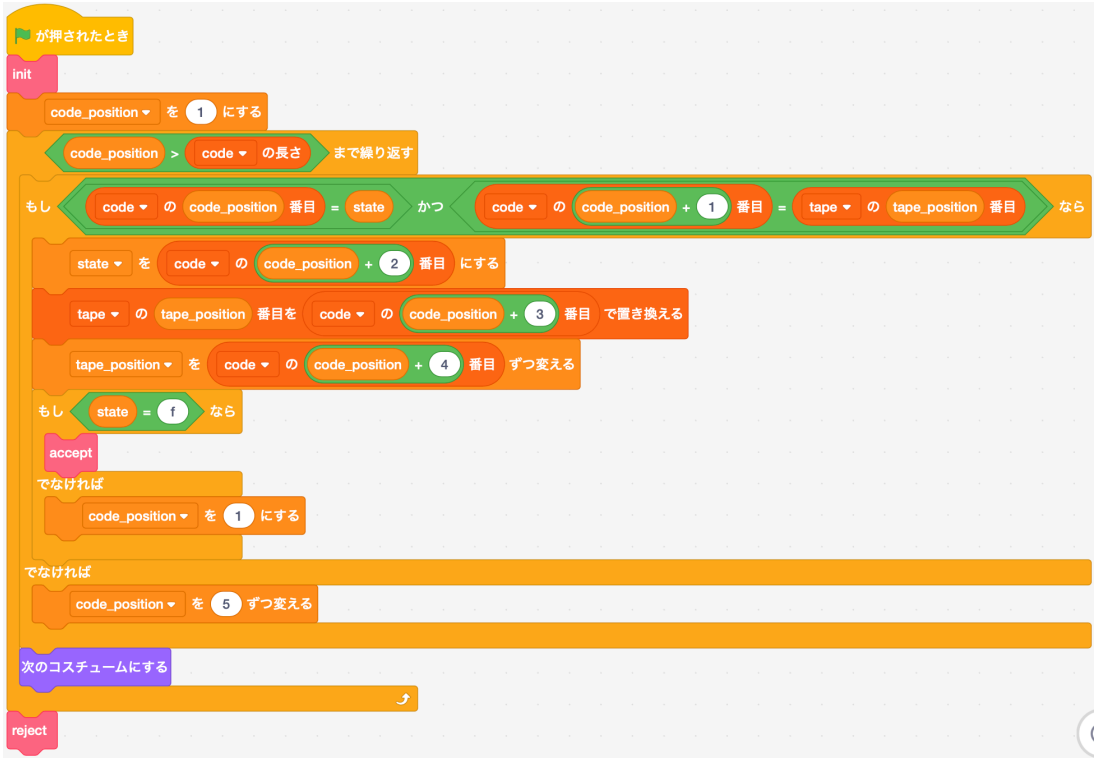


図 16.7: もうひとつのチューリング・マシンのプログラム（その 1、トップレベル、その 2 へ続く）

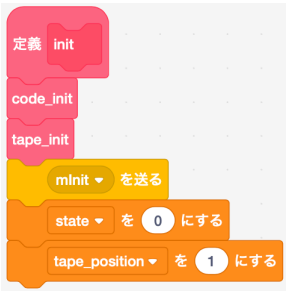
番目の内容）に異なるならば)、code_position を 5 だけ増やします。

(c) 猫のコスチュームを変更します。これのブロックはチューリング・マシンの実行に関係しませんが、これを削除するとチューリング・マシンの実行が一瞬で終了します。

3. もし code_position の値が code の長さよりも大きくなったならば、ループの実行が終了します。この場合、現在のチューリング・マシンの状態とヘッドの位置のセルの内容が、どの状態遷移のルールにもマッチしなかったことを意味しますから、不受理動作を行い、プログラムの実行を終了します。

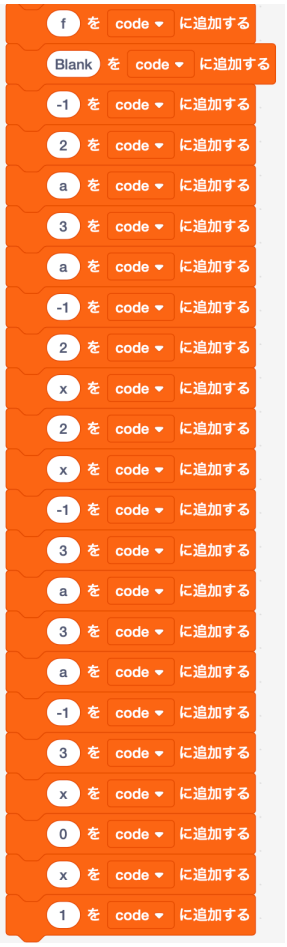
上のプログラムの実行は前節のプログラムの実行とは質的に異なることに注意してください。

前節のプログラムでは状態毎に動作をプログラム化しました。しかし、この節のプログラムでは全ての状態遷移はリスト code に記号化して載せています。ということは、code の内容を変更すれば別のチューリング・マシンの実行ができるのでしょうか。



(a) 初期化のプログラム

(b) code の初期化のプログラム（その 1、その 2 へ続く）



(c) code の初期化のプログラム（その 2）

図 16.8: もうひとつのチューリング・マシンのプログラム（その 2）

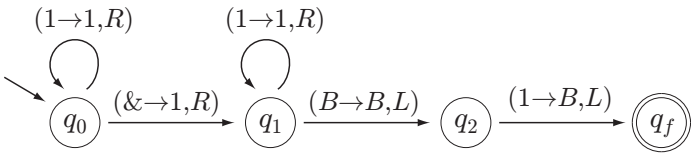


図 16.9: $3+2=5$ の計算を行うチューリング・マシンの状態遷移図

表 16.2: 図 16.9 のチューリング・マシンの有限状態部の動作表：左から順に、現在の状態、現在のヘッドの位置のセルの内容、次の時点の状態、次の時点のセルの内容、ヘッドの移動方向

0	1	0	1	+1
0	&	1	1	+1
1	1	1	1	+1
1	*Blank*	2	*Blank*	−1
2	1	f	*Blank*	−1

16.3　$3+2=5$ の計算を行うチューリング・マシン

前章、図 15.4 の $3+2=5$ の計算を行うチューリング・マシンの状態遷移図は図 16.9 のように作ることができます。（これは前章の考察の解答でもあります。）この状態遷移図の動作表は表 16.2 の通りです。そこで、この表の内容をリスト code に載せるプログラムを図 16.10 のように作り直します。

そして、tape には図 15.4 の内容を載せて、チューリング・マシンを実行してみます。図 16.11 (a) が実行開始直前の様子です。テープには、

$$1,\ 1,\ 1,\ \&,\ 1,\ 1,\ Blank,\ Blank,\ ...$$

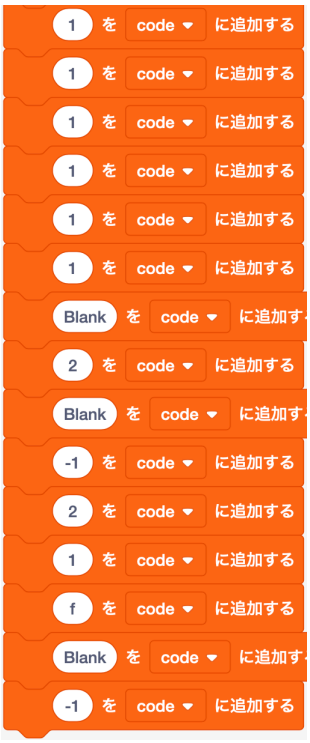
と入力記号を載せておきます。code と tape の内容以外は前節のプログラムをそのまま用いています。図 16.11 (b) が実行終了直後の様子です。テープに

$$1,\ 1,\ 1,\ 1,\ 1,\ Blank,\ Blank,\ ...$$

と記号が載っており、正常終了しています（ジュースの入ったグラスが表示されています）。$3+2=5$ が実行できていることに注意してください。このようにして、チューリング・マシンの中核部は変更せず、code と tape のみの変更で任意のチューリング・マシンの動作を実行できることが確認できます。

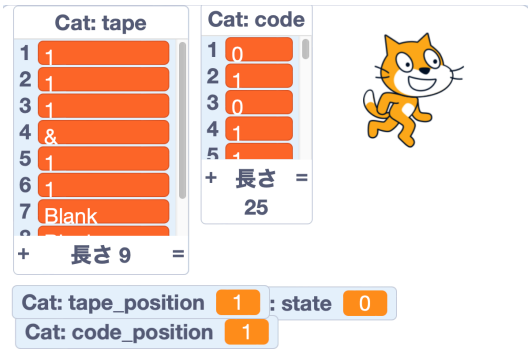
このチューリング・マシンは次章で紹介する万能チューリング・マシンに相当するものです。

(a) code の初期化のプログラム（その 1、その 2 へ続く）

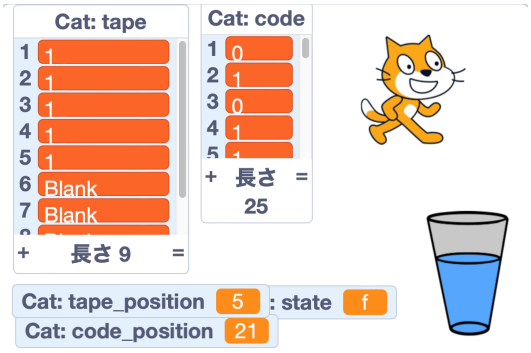


(b) code の初期化のプログラム（その 2）

図 16.10: 3 + 2 = 5 の計算を行うチューリング・マシンのプログラム（code の初期化）



(a) 開始直前



(b) 終了直後

図 16.11: $3 + 2 = 5$ の計算を行うチューリング・マシンの実行の様子

3

4

第17章　チューリング・マシンについてのさらに踏み込んだ話題

17.1　万能チューリング・マシン

チューリング・マシンでは多種多様な処理を記述することができます。それぞれの処理をそれぞれチューリング・マシンとして設計し、実行した場合、そのチューリング・マシンはその処理のための **専用** コンピュータ（special-purpose computer）として働きます。興味深いことに、チューリング・マシンの中にはそのような様々な専用チューリング・マシンをシミュレートできるものが存在し、特に任意のチューリング・マシンをシミュレートできるチューリング・マシンは **万能** チューリング・マシン（universal Turing machine）と呼ばれています。

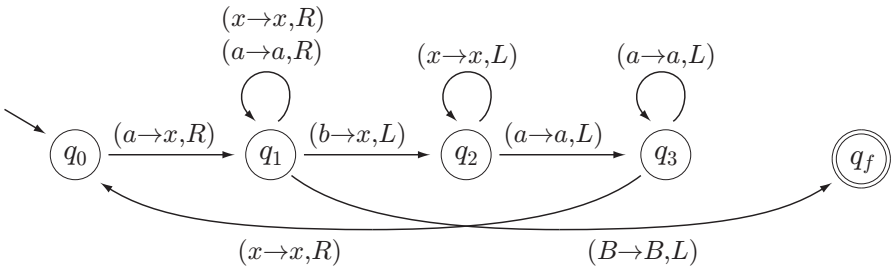
今、専用コンピュータであるチューリング・マシンを処理内容に応じ、番号を付けて TM_1、TM_2、... と区別することにします。たとえば TM_1 は加算を行うもの、TM_2 は減算を行うもの、... というように区別していきます。次に、その中のひとつのマシン TM_i へ与える入力記号列を S とします。このとき、S を入力とする TM_i の実行を $\mathrm{TM}_i(S)$ と表します。図 17.1 (a) がそれに対応します。

次に、万能チューリング・マシンを TM_U と表します。その入力は、TM_i の設計図 — これを $\mathbf{P}[\mathrm{TM}_i]$ と表します — とその TM_i への入力 S です。設計図である $\mathbf{P}[\mathrm{TM}_i]$ は入力ですから、テープの上に記号列として載っていなければなりません。前章のチューリング・マシンを思い出しましょう。たとえば、前章、前々章チューリング・マシンの状態遷移図：

(a) 専用マシーンによる実行
$\mathrm{TM}_i(S)$

(a) 万能マシーンによる実行
$\mathrm{TM}_U(\mathbf{P}[\mathrm{TM}_i],S)$

図 17.1: 専用チューリング・マシンと万能チューリング・マシン



は記号列として code 上に以下のように配置しました。

0	a	1	x	$+1$	1	a	1	a	$+1$	1	b	2	x	-1	...

このような記号列が $\mathbf{P}[\mathrm{TM}_i]$ です。これをチューリング・マシンのテープの先頭に載せておきます[1]。もちろん状態遷移図を記号列化する方法は一通りではなく、種々考えられますから、上はそのひとつの例に過ぎません。さて、図 17.1 (b) では、記号列 $\mathbf{P}[\mathrm{TM}_i]$ と S を空白記号 B を挟んでテープ上に初期配置し、そのテープ上をテープ・ヘッドが左右に動きながら動作する万能チューリング・マシン TM_U を図式化しています[2]。このような万能チューリング・マシンの実行を $\mathrm{TM}_U(\mathbf{P}[\mathrm{TM}_i], S)$ と表現しましょう。この実行では、TM_U は、S の上の記号をひとつ読み込んだ後に $\mathbf{P}[\mathrm{TM}_i]$ の中から次の状態遷移にマッチする規則を検索し、その規則に則った遷移を行い、S 上の該当箇所の記号を書き換え、さらに S の上の次の記号を読み込み ... と動作を行ないます。専用マシンの実行 $\mathrm{TM}_i(S)$ をひとつひとつシミュレートしていくため、万能マシンの実行 $\mathrm{TM}_U(\mathbf{P}[\mathrm{TM}_i], S)$ は $\mathrm{TM}_i(S)$ に比べ、非常に手間の掛かる遅い動作になりますが、等価な動作が可能です（機械が停止したときのテープの状態は $\mathrm{TM}_U(\mathbf{P}[\mathrm{TM}_i], S)$ と $\mathrm{TM}_i(S)$ で厳密に等しくなります)。逆に言えば、任意の TM_i、S について $\mathrm{TM}_i(S)$ と等価な動作が可能なチューリング・マシンを万能チューリング・マシンと呼びます。

もし万能マシンを用意できるならば、任意の専用マシンと等価な動作ができる訳ですから、いちいち特定用途の専用マシンを作る必要はありません。専用マシンに比べて実行ステップが長くなるでしょうが、万能マシン 1 台で任意の計算ができるのです。

今日、私たちがコンピュータと呼んでいるものは、万能チューリング・マシン TM_U に相当します。コンピュータのメモリあるいはハードディスクなどの記憶装置は、チューリング・マシンのテープに相当します。そして、私たちがプログラム（あるいはソフトウェア）と呼んでいるものは、$\mathbf{P}[\mathrm{TM}_i]$ に相当します。入力データは、S に相当します。コンピュータ上で入力データを与えてプログラムを実行することは、$\mathrm{TM}_U(\mathbf{P}[\mathrm{TM}_i], S)$ に相当します。現在のコンピュータは万能チューリング・マシンを数学的基礎とするものです。この方式の特徴は、プログラムをメモリ上にデータと

[1] 前章の Scratch のプログラムでは $\mathbf{P}[\mathrm{TM}_i]$ を code の上に載せましたが、ここでは tape の上の載せることを考えます。

[2] 前章では記号列 $\mathbf{P}[\mathrm{TM}_i]$ をリスト code に、記号列 S をリスト tape に、と二つのリストをテープに見立てて用いました。しかし、1 本のテープ上に code と tape の内容を載せることができます。

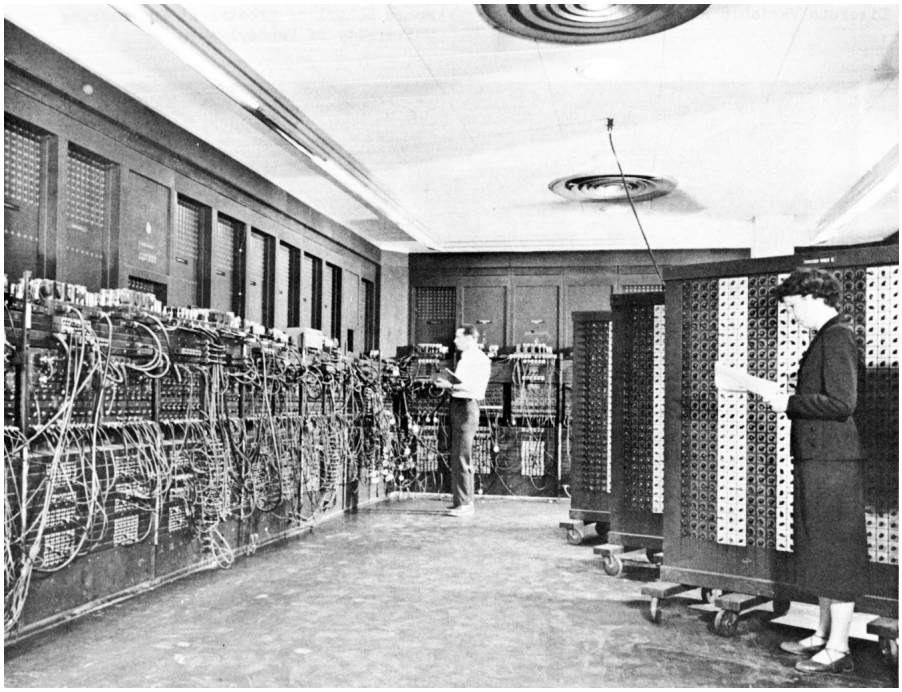


図 17.2: 世界初の電子式コンピュータ ENIAC の稼働風景（Wikipedia より）

記憶しておく点です。それゆえ、メモリ上のプログラムを書き換えるだけで、様々な計算を実行できます。この方式は **プログラム内蔵方式** （stored program method）と呼ばれ、コンピュータの重要な基本原理のひとつとなっています。

1943 年に世界で最初に開発された電子式コンピュータ ENIAC では、プログラムとデータは峻別されていました。入力データはメモリに格納されたものの、プログラムは多数のスイッチの ON/OFF と部品間のケーブルの接続によって実現されていました。そのため、プログラムを入れ替えるにはスイッチの切り替えとケーブルの張り替えが必要とされ、１回のプログラムの入れ替えに数日を要したようです。図 17.2 は、ENIAC の稼働風景です。左側に多数のスイッチとケーブルが見えます。これを設定し直すことがプログラミングに相当しました。その意味で ENIAC は専用コンピュータに近い構造でした。

プログラム内蔵方式は、ENIAC の次の世代のコンピュータ EDVAC によって実装されました。ドイツ系ユダヤ人研究者フォン・ノイマンがこの方式を提案したとされ、それに由来して、プログラム内蔵方式の今日のコンピュータをしばしば **ノイマン型コンピュータ** と呼んでいます。

17.2　チューリング完全

チューリング・マシンは有限状態オートマトン、プッシュダウンオートマトンよりも高い表現力を有し、多種多様な処理を記述することができますが、チューリング・マシンよりもさらに大きな

表現力、表現力を持つ機械は存在するのでしょうか。

20 世紀前半に多くの数学者によって様々な機械、数学体系が提案され、その表現力が研究されました。代表的なものとして以下のようなものがあります[3]。

チューリング・マシン ... 本章の主題です。

帰納的関数 ... 自然数の上の関数に対して、いくつかの計算上の仕組みを導入し、関数の表現力を論じるものです。

ラムダ計算 ... ラムダ式（λ 式）と呼ばれる表現形式に対して、いくつかの変形操作を定義し、様々な式の変形に関する性質を論じるものです。

タグシステム ... 文脈自由文法の生成規則に類似した規則を用いて記号列の変形を繰り返し、変形の性質を論じるものです。

興味深いことに、それぞれは独立して提案されたにも関わらず、これらはいずれも等価な表現力を持つことが分かっています。このことは、どのような手順や操作を用いようとも、機械的な手順や操作で記述できる内容[4]には共通の限界が存在することを示唆しています。ここで言う「表現力」は計算能力、計算可能性（computability）と換言することもできます。そこで、

計算可能である　＝　チューリング・マシンで計算できる

と見なそうという提案がなされました。この提案を、上記の等価性の研究に貢献した研究者の名前を冠して **チャーチ＝チューリングの提唱**（Church-Turing thesis）またはチャーチの提唱（Church's thesis）と呼んでいます。この提案は定理ではありませんから証明できません。「計算可能である」ことの定義を与えている（そう見なそうと提案しているに過ぎない）ことに注意してください。

チャーチ＝チューリングの提唱は、任意の計算可能な内容を実際に記述し、実行するにはチューリング・マシンを用いればよい、あるいはチューリング・マシンと等価なシステムを用いればよいことを意味しています。そのような等価なシステムのことを **チューリング完全**（Turing complete）[5]であると言います。

前節で述べたように、私たちが日頃使用しているノート PC はチューリング完全です。

プログラミング言語も計算を記述できるシステムと見なすことができます。まず、Scratch はチューリング完全です。何故ならば、前章で見たように、Scratch でチューリング・マシンをシミュレートするプログラムを作ることができるからです。この講義の 2.4 節から 2.7 節で、プログラム実行の四つの原理を紹介しました。これら四つの原理を実装したプログラミング言語はチューリング・マシンをシミュレートできるという意味でチューリング完全です。世の中に知られているプ

[3]個々について詳細な述べませんので、興味を持った人は自身で調べてみてください。
[4]「内容」とは、数学的には集合や関数を意味します。
[5]同じ意味で、チューリング計算可能（Turing computable）と呼ぶこともあります。

ログラミング言語のほとんどは四つの原理を実装しており、チューリング完全です。逆にチューリング完全でないものは一般にはプログラミング言語とは呼びません。ただしいくつかの反例として、Web 文書作成用の HTML（HyperText Markup Language）は一見するとプログラミング言語に似た構造を持っており、さらに名前に language が付いており、プログラミング言語風ですが、チューリング完全ではありません。

17.3　計算可能と決定不能

システムがチューリング完全であるならば、任意の計算可能な内容はそのシステムで実行できることが保証されます。ところで「計算可能」な内容とは具体的にどのようなものでしょうか。その答えは、「アルゴリズムが存在すること」です。その「アルゴリズム」は以下のように定義されます。

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.、2. は当然の条件です[6]。3. は重要です。手順の中にはこの条件を部分的にしか満たさないものも考えられ、それをアルゴリズムと区別して、 **セミアルゴリズム** （semi-algorithm）と呼ぶことがあります。ただし、4. の 条件は、このテキストの議論には馴染みません。ここでは代わりに「手順をチューリング完全な特定のプログラミング言語で記述できる」という条件を用いることとします。

アルゴリズムの例として、任意に与えられた二次方程式 $ax^2 + bx + c = 0$ が実数解を持つか否かを判定する問題を考えましょう。これは計算可能な問題です。何故ならば、判別式 $b^2 - ac$ が 0 以上ならば YES、さもなくば NO と回答すればよく、明らかに有限の手順で答えが出るからです。別の例として、任意に与えられた自然数 a と b が互いに素であるか否かを判定する問題を考えましょう。これも計算可能な問題です。何故ならば、互いに素か否かはユークリッドの互除法を用いて a と b の最大公約数を求め、それが 1 であるか否かを判定すればよいからです。もちろんユークリッドの互除法は有限の手順で終了します。このように、計算可能な問題とそれを解決するアルゴリズムは膨大に存在します。

[6] この授業では非決定性オートマトンを学びました。その動作はある意味で曖昧ですが、非決定的な動作を並列実行と解釈すれば、曖昧さは無くなります。

上に上げた二つの例はいずれも判定問題です。計算結果が YES/NO、1/0、真/偽などの 2 値で求められる問題 — 広い意味の判定問題 — のことを **決定問題**（decision problem）とも呼びます。ある決定問題に対するアルゴリズムが存在する場合にはその問題は **決定可能**（decidable）であると言い、さもなくば **決定不能**（undecidable）であると言います。世の中には膨大な数の決定問題が存在し、その多数は決定可能です。また、判定結果が YES、1、真の場合には停止するものの、NO、0、偽の場合には停止しないかもしれないセミアルゴリズムが存在する問題を半決定可能（semi-decidable）とも言います。

具体的な計算値 n を求める問題はそのままでは決定問題ではないのですが、それを「計算結果 n は定数 x 以下か否か」という問題へ置き換えることで決定問題へ読み替えることができます。そして、x を様々に変えて複数の決定問題を用意することで、n の値をいくらでも精密に求めることができます。たとえば $\sqrt{2}$ の値を計算する場合、まず「$\sqrt{2}$ は 2 以下か？」という問題を考えます。答えは YES です。そこで次に「$\sqrt{2}$ は 1 以下か？」という問題を考えます。答えは NO です。次に「$\sqrt{2}$ は 1.5 以下か？」という問題を考えます。答えは YES です。次に「$\sqrt{2}$ は 1.25 以下か？」という問題を考えます。答えは NO です。このように、$\sqrt{2}$ の計算精度はいくらでも小さくすることが可能です。計算可能性を論じることは決定問題を論じることと本質的に同じなのです。

ところで、決定不能な問題、チューリング・マシンを用いても計算できない問題は世の中に存在します。今日では様々な決定不能問題が存在することが知られていますが、このテキストでは決定不能問題の中で最も有名な **チューリング・マシンの停止性問題**（Turing Machine Halting Problem）[7]について解説し、この章およびこのテキスト全体を終えることにします。

17.4　停止性問題

プログラムのデバッグで難渋する場合のひとつが、開発したプログラムの実行が無反応になる場合です。正しい計算を行なっている途中なのか（よって、未だ答えが出ていないが、もう少し待てば計算結果が得られるか）、あるいは無限ループに陥っていて、今後、永遠にプログラムの実行が止まらないのか、判断が難しい場合があり、デバッグに苦労します。開発中のプログラムが無限ループを持つのか否かを自動的に発見してくれるソフトウェア・ツールがあると助かるのですが、計算可能性の理論はそのような便利なツールは理論上、存在しないことを明らかにしています。以下、そのことをチューリング・マシンをベースに背理法で証明します。

今、入力 S を受け取って、ある計算を行なうチューリング・マシン TM を考え、その実行を $TM(S)$ と表します。任意に与えられた S と TM について、$TM(S)$ が有限停止するか否かを判定できるチューリング・マシン H が存在すると仮定し、その実行を、ちょうど万能チューリング・

[7]厳密に言えば、チューリング・マシンの停止性問題は半決定問題です。

マシンの場合と同様に、H($\mathbf{P}[TM], S$) と表すことにします。なお、$\mathbf{P}[TM]$ は、TM を記号列に変換したものです（17.1 節参照）。ここで、H($\mathbf{P}[TM], S$) の実行は有限停止し、

1. もし $TM(S)$ が停止するならば（そのときに限り） ... H($\mathbf{P}[TM], S$) は YES を出力する
2. もし $TM(S)$ が停止しないならば（そのときに限り） ... H($\mathbf{P}[TM], S$) は NO を出力する

と仮定します。上に述べたように、そのような H があると便利なのですが、残念ながらそのような H は理論上、存在しえないことを証明していきます。

まず、上の入力記号列 S は任意ですから、S の代わりに $\mathbf{P}[TM]$ を用いることもできるはずです。その場合、 H の実行は H($\mathbf{P}[TM], \mathbf{P}[TM]$) という奇妙な実行になりますが、それが現実世界でどのような意味を持つのかなどということは当面、気に掛けないことにします。

次に、 H を改造した以下のチューリング・マシン M を作ります。この改造はわずかなので、もし H を作ることができる（H が存在する）ならば、必ず M も作ることができます。

1. もし H($\mathbf{P}[TM], \mathbf{P}[TM]$) が YES を出力するならば（そのときに限り） ... M($\mathbf{P}[TM]$) は停止しない[8,9]。
2. もし H($\mathbf{P}[TM], \mathbf{P}[TM]$) が NO を出力するならば（そのときに限り） ... M($\mathbf{P}[TM]$) は停止する[10]。

次に、M への入力として M を記号列化した $\mathbf{P}$[M] を与える実行 M($\mathbf{P}$[M]) を考えてみます。これも現実世界では奇妙な実行ですが、そこに気を取られること無く、数学的な意味のみを追求して行きます。

さて、私たちは、M($\mathbf{P}$[M]) が停止するか否かに注目します。これについて M の定義から

1. もし M($\mathbf{P}$[M]) が停止するならば ... H($\mathbf{P}$[M], $\mathbf{P}$[M]) が NO を出力します。
2. もし M($\mathbf{P}$[M]) が停止しないならば ... H($\mathbf{P}$[M], $\mathbf{P}$[M]) が YES を出力します。

次に、もとの H の定義から

1. もし H($\mathbf{P}$[M], $\mathbf{P}$[M]) が NO を出力するならば ... M($\mathbf{P}$[M]) は停止しません。（参考：もし H($\mathbf{P}[TM], S$) が NO を出力するならば ... $TM(S)$ は停止しません。）
2. もし H($\mathbf{P}$[M], $\mathbf{P}$[M]) が YES を出力するならば ... M($\mathbf{P}$[M]) は停止します。（参考：もし H($\mathbf{P}[TM], S$) が YES を出力するならば ... $TM(S)$ は停止します。

[8]第一の改造は引数の数に関するものです。H の引数は二つですが、M の引数はひとつです。しかし、これは問題ではありません。H($\mathbf{P}[TM], \mathbf{P}[TM]$) の実行では二つの引数が同一であることに注意してください。M は実行直後に受け取った引数をテープ上にコピーすればよいのです。

[9]第二の改造では、H が YES を出力する動作部分を M では無限ループ（同じ状態を無限に遷移し続けるような有限状態部）に置き換えます。

[10]第三の改造では、H が NO を出力するときには M は停止するように改造します。

上の 2 段階の論理を縮めると

1. もし M(**P**[M]) が停止するならば ... M(**P**[M]) は停止しません。
2. もし M(**P**[M]) が停止しないならば ... M(**P**[M]) は停止します。

これは矛盾です。よって、任意に与えられた S と TM について、$TM(S)$ が有限停止するか否かを判定できるチューリング・マシン H が存在すると仮定したことが間違いであると結論されます。

停止性問題においてチューリング・マシンであることは問題の本質ではありません。チューリング完全なプログラミング言語についても — もちろん、Scratch についても — 同様の性質が成り立ちます。

コンピュータは非常に便利な道具であって素人目線では何でもできそうに思えます。しかし実際にはできないこともある – しかもそれが数学的に厳密に定義されているにもかかわらず – ということです。これがこの講義テキストの最後の話題でした。

注意：　蛇足です。停止性問題が決定不能であるからと言って、停止性について全く知ることができない訳ではありません。以下のプログラムは明らかに有限停止します。

この程度の単純なプログラムについてはコンピュータを用いた停止性の自動判定が可能です。

（以上）