

部長挨拶

「ついにこの文章を書く年になってしまいました。」

二年前に部活を卒業した先輩が遺した言葉です。この頃はこんな言葉とも無縁で、今まで通りの日常、そして部活動がずっと続くとばかり思っていました。

ですがどうでしょう。この有様です。これで部活動が今まで通りにできるわけでもなく、特に夏休みの活動は活動時間も、活動日も半分以下になってしまいました。一年を通して大きな比重を占める夏休みの時間が削られるわけですから、文化祭の準備もおそらく…いや、それでも最善は尽くせたと考えましょう。今ある時間でできることを着実にやるしかありません。普段何かと理由を付けて作業をしない後輩にも聞かせてやりたいぐらいです。

さて、今年の文化祭ですが、いくつか変えてみたところがあります。一番目に付くのは部屋の配置でしょう。今までPC班と電工班で一部屋ずつ分けて展示をしていたのですが、今年はあえてPC班と電工班を混ぜて展示してみました。というのも、電工班での滞在時間がPC班と比べ短いので、そこを平均化してより多くの展示に触れて欲しいもありますし、電工班員があまりに少なく一部屋に展示できるだけの作品を用意できないというのもあります。後者が大きいのでしょうか、電工班をずっと引っ張ってきた身としては悲しい話です。また、数年前から始まった壁新聞も今年は大判プリンターで印刷することになりました。今まで模造紙に油性ペンで手書きしていて、これがかなり時間を喰うのです。そこで去年は家庭用のプリンターで、A4の紙に分割して印刷することにしてみました。とはいえるでもA4の紙を繋ぎ合わせる手間が掛かります。そこで実行委員から大判プリンターが使えるという話を聞いたのです。これはもうやるしかないでしょう。

あとは参加団体名ですかね。「物理部展#2021」ではなく「物理部展丶」です。丶です。きっかけは2018年の「物理部展#2018」でした。当時の部長がふざけて（本人は大真面目だったのかも？） "#" の真ん中に点を打って丶にしてしまったのです。しかしその時は文化祭の実行委員に気づかれずそのまま "#" になったのですが、以降 "#" の真ん中に点を打って#か丶か紛らわしいようにして、その年の実行委員を試すということをずっとやっていたのです。

そうしたら今年、なんと実行委員が初めてこれを丶とみなしてしまって、晴れて今年の展示名が「物理部展丶2021」となったのです。こんなことってあるんですね、という感想しかありませんが。そして部長としての二年間を振り返ってみて、思えば先輩からの「続投は部活が崩壊するからやめとけ」という制止も振り切って二年目も部長のポストに居座ったわけですが、やはり自分の裁量で物事を進められるのは気が楽です。何をやるにも報告、連絡、相談なのは面倒で仕方がないです。そういう性格なんです。（後輩がそうしないのは困りますけど）

ただ学校側に提出する書類などは全部管理しないといけませんし、そういう面倒さはあったのでしょうか。ただ面倒なら後輩に任せればいいので、特に気になりませんでしたね。

あとは後輩はまだしも、一部は先生や友達からも「部長」と呼ばれるようになりました。部長に就任したての頃はある種の高揚感すら感じたのですが、今は何とも思わなくなってしまいまし

た。文化祭が終わって部活を引退したらどうなるんでしょうか。「元部長」なんでしょうか。そろそろ名前で呼んで欲しいんですけど...。

関係ないことまで色々書いてしまいましたが、最後に挨拶だけ。物理部展両2021にお越しいただきありがとうございました。今年は（今年も？）準備期間も短くお見苦しい点もあるかもしれません、そこは後輩に期待しましょう。ぜひ来年もお越しください。

目次

1. バーゼル級数を利用した円周率の計算	2
2. ゲームのBGMについて	7
3. 深層強化学習によるリバーシAI	16
4. アーキテクチャへの扉	28
5. AMDのZen3について考える	38
6. 芸術の重要性について	41
7. 理想と現実の界面で	47
8. コイルガン 四年間のすべて	56
9. 自作コンパイラを実装してみた	73

バーゼル級数を利用した円周率の計算

バーゼル級数とは

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = 1 + \frac{1}{4} + \frac{1}{9} + \cdots = \frac{\pi^2}{6}$$

これがバーゼル級数です。今回はこの級数を利用して円周率を計算していこうと思います。また、プログラミングに疎い人でも楽しめるように表現を曖昧にしてたり多少ずれています。

Pythonでの計算

今回はプログラミング言語にPythonという言語を選択しました。

Pythonは文法が比較的簡単でわかりやすく、優れたツールの豊富さが魅力の言語です(もちろん文法が比較的に簡単といえどもしっかり理解しようと思うととても時間がかかります)。

デメリットとしては、仕様上比較的実行に時間が掛かり、用途によっては向かないことなどが挙げられます。

Pythonの実行環境

Python 3.8.2 64bit

ソースコード

方法としては左辺を計算しそれを6倍して $\sqrt{}$ を取るのですが、 $\frac{1}{k^2}$ を $k = 1$ から無限に足し続けることは不可能なので、今回は $k = 100,000,000$ まで足し続けたいと思います。

```
#数学関連のためのツールを利用する
from math import *
#桁の多い小数を正確に扱うためのツールを利用する
from decimal import *
if __name__ == "__main__":
```

```
#変数sumを定義
sum=Decimal(0)
#k=1からk=100000000まで
for i in range(100000000):
    #1/k^2を計算してsumに加える
    sum += Decimal(1)/((Decimal(i)+Decimal(1))**2)
#級数の左辺の結果を6倍し✓をとる
pi = Decimal(sum*6).sqrt()
#結果を表示
print(pi)
```

実行結果

```
3.141592644040496686182218894
```

これだけではどれだけ正確に計算できているのか分かりづらいので、合っている桁に色を付けてみたいと思います。

```
#数学関連のためのツールを利用する
from math import *
#桁の多い小数を正確に扱うためのツールを利用する
from decimal import *
if __name__=="__main__":
    #変数sumを定義
    sum=Decimal(0)
    #k=1からk=100000000まで
    for i in range(100000000):
        #1/k^2を計算してsumに加える
        sum += Decimal(1)/((Decimal(i)+Decimal(1))**2)
    #級数の左辺の結果を6倍し✓をとる
    pi = Decimal(sum*6).sqrt()
    pi_str = str(pi).replace(".", "")
    length = len(pi_str)
    #実際の円周率の定数PIを定義
    PI = str(31415926535897932384626433832795)
    #計算した数字と実際の円周率を一桁ずつ比較し合致している場合は色をつけて表示
    for i in range(length):
        if pi_str[i] == PI[i]:
            print('\033[31m'+pi_str[i]+\033[0m',end="")
        else:
            print(pi_str[i],end="")
    print()
```

実行結果2

3141592644040496686182218894

小数点以下7桁まで合っているのがわかります。

バーゼル級数の証明

ここからは先程のバーゼル級数の証明をしていくのですが、正直このあたりはネット等で情報が山ほどあるのでそちらを見ていただいたほうがわかりやすいと思います(そもそもこの部誌の証明はネットで調べたものとほぼ変わりませんし)。

バーゼル級数が収束することの証明

$$\begin{aligned} 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{n^2} &< 1 + \frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \cdots + \frac{1}{(n-1)n} \\ &= 1 + \left(1 - \frac{1}{2}\right) + \left(\frac{1}{2} - \frac{1}{3}\right) + \cdots + \left(\frac{1}{n-1} - \frac{1}{n}\right) \\ &= 2 - \frac{1}{n} \end{aligned}$$

よりこの級数は2以下に収束する

$0 \leq x \leq \frac{\pi}{2}$ において $\sin x \leq x \leq \tan x$ となることの証明

図において三角形ABC < 扇型OAB < 三角形OBCが成り立つ

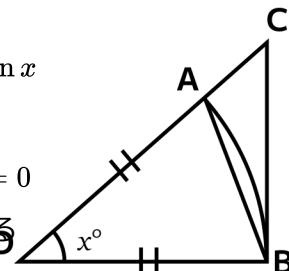
$$\text{すなわち } \frac{1}{2} \sin x < \frac{1}{2}x < \frac{1}{2} \tan x$$

よって $\sin x < x < \tan x$

$x = 0$ のとき $\sin x = 0, \tan x = 0$

$x = \frac{\pi}{2}$ のとき $\sin x = 1$ である

よって命題は示された



ド・モアブルの定理の証明

$(\cos \theta + i \sin \theta)^n = \cos n\theta + i \sin n\theta$ を示す

$n = 1$ のときは自明である

$n = k$ のとき成り立つと仮定すると

$$\begin{aligned} (\cos \theta + i \sin \theta)^{k+1} &= (\cos \theta + i \sin \theta)^k(\cos \theta + i \sin \theta) \\ &= (\cos k\theta + i \sin k\theta)(\cos \theta + i \sin \theta) \\ &= (\cos k\theta \cos \theta - \sin k\theta \sin \theta) + i(\sin k\theta \cos \theta + \cos k\theta \sin \theta) \\ &\quad (\text{加法定理より}) \\ &= \cos(k+1)\theta + i \sin(k+1)\theta \end{aligned}$$

であるから $n = k$ が成り立てば $n = k+1$ でもなりたつ

よって示された

バーゼル級数の証明

$k = 1, 2, \dots, n$ に対し $\theta_k = \frac{k\pi}{2n+1}$ とする

すると $0 \leq \theta_k \leq \frac{\pi}{2}$ より $\sin \theta_k \leq \theta_k \leq \tan \theta_k$ となる

各辺の逆数をとって二乗すると

$$\frac{1}{\tan^2 \theta_k} \leq \frac{(2n+1)^2}{k^2 \pi^2} \leq \frac{1}{\sin^2 \theta_k}$$

これを $k = 1$ から n まで足し合わせる

$$\frac{\pi^2}{(2 + \frac{1}{n})^2 n^2} \sum_{k=1}^n \frac{1}{\tan^2 \theta_k} \leq \sum_{k=1}^n \frac{1}{k^2} \leq \frac{\pi^2}{(2 + \frac{1}{n})^2} \left(\frac{1}{n} + \frac{1}{n^2} \sum_{k=1}^n \frac{1}{\tan^2 \theta_k} \right) \cdots (1)$$

$$S_n = \sum_{k=1}^n \frac{1}{\tan^2 \theta_k} \text{ とする}$$

$\sin(2n+1)\theta_k = 0$ より

$z = (\cos \theta_k + i \sin \theta_k)^{2n+1}$ の虚部は 0 であり

また、 $\sin \theta_k \neq 0$ であるから z を $\sin^{2n+1} \theta_k$ で割ると

$$z' = \left(\frac{1}{\tan \theta_k} + i \right)^{2n+1} \text{ の虚部は 0 である}$$

バーゼル級数の証明(続き)

この z' の虚部は $\frac{1}{\tan^2 \theta_k}$ の n 次多項式とみなせる

この n 次多項式を $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0 = 0$ とする

$k = 1, 2, \dots, n$ に対して $f\left(\frac{1}{\tan^2 \theta_k}\right) = 0$ である

n 次方程式の解と係数の関係より $S_n = -\frac{a_{n-1}}{a_n}$

二項定理を用いて $a_n = 2n + 1, a_{n-1} = -\frac{(2n+1)(2n)(2n-1)}{6}$ であるから

$$S_n = \frac{n(2n-1)}{3}$$

よって $\lim_{n \rightarrow \infty} \frac{S_n}{n^2} = \frac{2}{3}$ であるから(1)式においてはさみうちの原理より

$$\sum_{k=1}^n \frac{1}{k^2} = \frac{\pi^2}{2^2} \cdot \frac{2}{3} = \frac{\pi^2}{6}$$

参考

<https://manabitimes.jp/math/878>

<https://manabitimes.jp/math/689>

<https://manabitimes.jp/math/1315>

ゲームのBGMについて

1.はじめに

皆さんこんにちは、中学3年生の明松です。今年初めてポジトロンの記事を書かせていただくことになりました!!!中2のときにこの記事を書いたのですが、新型コロナウイルスによる文化祭の規模縮小により物理部が部誌を出しませんでした。

さて、本題へ移っていきましょう!僕はゲームをするとき、BGMを気にしてしまうような人です。また、僕が作るゲームのBGMは基本自分で作ります。そんな僕が思ったことをまとめます。

ゲームのBGMを作つてみたい!と思われている方はこの記事を参考にして、僕なんかよりもっとBGMを作られている方やこれを職業とされている方は、へえ、こんなかわいい考え方があるんだなと思いながら読んでください。くれぐれも「あいつ幼稚な考え方だな」と思ったりするのはおやめください。傷つきます。

2.それぞれの種類の音符はどのようなところに使われるか

まずはそもそも音符にはどういう種類があるの?という方のために、種類の説明をします。こちらをご覧ください。音符の名前とその記号の対応表です。

音符の種類	全音符	2分音符	4分音符	8分音符
-------	-----	------	------	------



その他にも譜点付きや16分音符、32分音符などがありますがそれでは細かい話になってしまふので省きます。気になるという方は調べてみてください。

さて、それぞれの音符についての説明をします。まず、全音符とは、4拍分音を鳴らしてしまう音符です。あまり主旋律で使われることはありますか、使われるとしたらどのような場面で使われるのでしょうか？

2分音符とは、2拍分音を鳴らす音符です。こちらもあまり主旋律で使われることはありません。こちらの使いみちも気になりますよね？

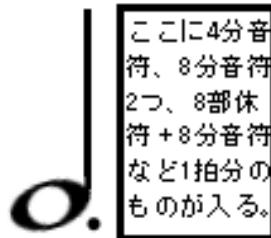
4分音符とは、1拍分音を鳴らす音符です。これはよく使われます。しかしBGMのジャンルによっても使われる頻度が異なります。

全音符、2分音符、4分音符だけではのんびりしたBGMしか作れません。そこで活躍するのは8分音符です。8分音符とは、2分の1拍分音を鳴らす音符です。この音符があることでBGMも大きく変わっていきます。

さて、ここまで音符の種類について解説していきました。では、それぞれの種類の音符はどのような種類のBGMを作るのか、見ていきましょう。

まず、全音符が主旋律に使われる場合としては、悲しい曲の偶数小説目や、神秘的な曲の冒頭などがあります。また、低音に使われる場合としては...とはいっても **僕の場合は** 低音の部分を作っているときに全音符は乱発するのでまあ「たまに出てくるだろう」と思つとけばよいでしょう...
(個人の意見です)

しかし、全音符そのものが使われることは少ないのです。次の図のように、付点2分音符で3拍分演奏しつつ、後ろに1拍分のメロディーを入れる、というもののはうが多いと思いますね。



付点2分音符

次に、2分音符が主旋律に使われる場合では、先ほど挙げたように、付点をつけて3拍分演奏されることがあります。2拍分演奏させる場合としては、悲しい曲や、神秘的な曲の冒頭などがあります。うん。2分音符はほぼ全音符と同じ役割を果たすのではないでしょうか？ いえ、多くのジャンルの曲の最後や8の倍数小説目の部分に使われることも多くあります。低音のパートにおいては、もうスターと言ってしまえるくらいたくさん出てきます。

また、一番よく使われるであろう4分音符が主旋律に使われる場合としては、かっこいい系のBGMが挙げられます。また、3拍子のものの場合にも使われます。しかしながら4分音符だけで使われる場合は少なく、8分音符と組み合わせて使われることが多いです。4分音符は本当に8分音符との組み合わせが良いのです！例えば、4分音符→8分音符→4分音符→8分音符→…や、8分音符→4分音符→8分音符→4分音符→…などという組み合わせがあります。また、4分音符に付点をつけて1.5拍分音を鳴らす場合もあります。低音に使われる場合では、4分音符がそのまま4拍分並べて使われたり、ほかの音符に交じって出てきたりします。

8分音符が主旋律に使われる場合では、4分音符や16分音符などと並べられて使われることが多いです。あるいは付点のある音符の前や後ろに入ったりしています。8分音符は、低音においても隠れたスターとして大活躍します。8分音符だけで並べられたり、他の音符に混じって使われたり…まあとにかくたくさん使われます。

ここまで僕はめちゃめちゃな説明をしてきたのですが、いかがでしたか？言いたいことは、音符の並べ方としては、単独で使われるときとほかの種類の音符と並べて使われるときとがあるということですね。さて、それでは次の章へ行きましょう。

3.テンポの違いはBGMにどのような影響をもたらすのか

さて、テンポの違いとBGMの関係についてみていきましょう。一つだけ皆さんにこの章では覚えていただきたい言葉があります。「**BPM**」です。これは"Beats Per Minute"の略です。つまり何かというと、1分間に4分音符を休まずにいくつ演奏することができるのか、ということです。例えば、皆さんは楽譜の最初に「♩ = 83」みたいなものは見たことがありますか？これは、1分間に4分音符を休まずに83回演奏できる、ということなのです。この単位は心拍数にも使われています。

では、テンポ数の具体例を挙げていきましょう。ざっと説明していくと、まず、80BPM前後では「遅い」テンポであり、ゆっくりしているので、落ち着いたような曲、神秘的な曲、悲しい曲などの例が挙げられます。

100BPMは、いわゆる「とても遅くはないが速いとか普通のテンポではない」といった中途半端なリズムであるため、緊迫した場面や、何かを考えさせるような場面のBGMで使うことができます。

120BPM前後の曲の用途はいろいろあります。例えば、かっこいい系の曲、少し落ち着いた曲、少し元気な曲、少し緊迫感のある曲などがあります。なぜ「少し」という表現をたくさん使用したかというと、あまり強い表現ができないからです。120BPM以外のテンポで強い表現をすることができます。でも一番使い道が多いのはこの120BPMなんですね。

そして150BPM前後は「速い」リズムであるため、とても激しい音楽に使われます。ここに低音などをたくさん入れると工夫次第では120BPMくらい、またはそれよりもかっこいい音楽を作ることができます。

また170BPMは「更に速い」ことから、更に激しい雰囲気になります。

使い道は結構少ないので応用として、ゲームのワンシーン内でも雰囲気の起伏が激しいときは、テンポを途中で変えるものもあります。これを見て「ボス戦のBGMとかのことじゃない?」などと思った方もいらっしゃると思います。確かにボス戦が「雰囲気の起伏が激しい」の典型的な例と言えるでしょう。

このように、テンポの違いもBGMの雰囲気を変え、様々な使い方ができるのです。

むしろ、ゲームの雰囲気に合わせたBGMを作る際、まずはそのシーンにあったテンポとは何かを考えることが重要となってくると僕は考えます。

4.BGMに声は入れるべきなのか

さて、ここからはレベルを少し上げます。基本的な音楽の構成要素について見ていったのですが、ここでBGMに歌詞は要るのかということを考察します。本来BGMには歌詞がないのが普通なのですが、歌詞を入れるか否かでだいぶ奥が深くなります。

声の入れ方にはどのようなパターンがあるのか見ていきます。

- よくある普通の歌みたいに歌詞のようなものを入れる
- 「アー」「オー」などの単母音の声を入れる
- 意味不明な(文字起こし不可能な)歌詞を入れる
- 1単語や短い文章など、1つのフレーズをある場所に入れる
- サビだけに歌詞を入れる
- サビにだけ歌詞を入れない
- ほとんど声だけ

などいろいろあります。パターンを挙げたらきりがありませんね。このように、歌詞に声を入れるというのは奥が深いものなのです。

しかし、本題はここから。BGMに歌詞を入れるのには次のような目的があると(少なくとも僕は)考えます。

- ゲームをする上で、BGMの存在感を一層強める
- 今のゲームのシーンをより鮮明に反映する
- 何かしらの演出をより豪華にする
- 声のない単調なBGMからより複雑に、より長くし、ゲームの奥深さを増す

などいろいろあります。しかしながら、BGMの声には短所もあります。

- ゲームのクオリティがある程度高くないとBGMとのギャップが際立ってしまう
- 音量を調整しないと声が際立ちすぎたり、逆にまったく意味がなかつたりしてしまう
- 特に日本語など、歌詞が聞き手にとってとても良くわかる言語だとゲームよりも歌詞のほうが印象に残りやすくなってしまう

声にはこのような短所もあり扱いが難しく、曲制作に長けているような人でもなかなか難しかったりします。

上の短所にもありました。BGMの声の音量調整は特に複雑です。そこで、音量と声との印象の関係を表にまとめてみました。表の横の列は、声パートの音量が他の楽器の音量の何%なのかを示し、縦の列は全体音量を示します。なお、特にこの表は、作曲の経験があまりなく、ゲームもほとんどしない僕の独断と偏見のみに基づいています。くれぐれも参考にしそうないようになります。

	～50%	～100%	100%～
静か	声はほとんど聞こえない、BGMの存在感は薄い	ある程度バランスが取れている、存在感は小さい	ほとんど声だけが印象に残る
普通	ゲームに集中しやすい	特にバランスの取れたBGM、ゲームの雰囲気を感じやすい	何かしらの演出に使われるような独特な印象
うるさきめ	声よりはメロディを気にしやすい	BGMの存在感そのものは特に大きい	声の印象が大変高い

このように、声を扱うのはとても難しいのですが、使いこなせるようになると曲の幅がとても広がります。みなさんがゲームをするときも、特に声を気にしてはいかがでしょうか。

また、どんなに声を工夫しようとも声が似合わないような演出やシーンもたくさんあります。

「こんなシーンは果たして声を入れるべきなのか」と考えてみるのもとても楽しいものです。

しかし、更に難しいポイントが次の章にあります。みなさんはなんだと思いますか？

5.BGMの楽器の種類、数はどのような印象の違いをもたらすのか

さて、答えは「楽器」。BGMを構成するにあたって特に重要なポイントです。もちろん楽器なしではBGMは少しも完成しません。

ということで、まずは楽器の種類について書いていきます。楽器の種類と特徴を羅列していきます。なお、これはWikipediaより「楽器分類別一覧」というページを参考にしていますが、説明は自分で書いたものです。

1. 擦弦楽器

音を細かくすれば激しい印象に、長い音を繰り返せば落ち着いた印象にしてくれ、隠れたところで使いみちが多様な楽器。ゲームの印象をより美しくする。バイオリン、ビオラ、チェロなど

2. 撥弦楽器

琴などの古の楽器から、ギターなどの現代的な楽器まで様々な種類がある。弦を単独で弾いても美しい音が出るが、特にこの楽器から出る和音が美しく印象的である。メインパートに使われるも多いが、サブパート、伴奏に使われることが多い。琴、ハープ、ギターなど

3. 打弦楽器

ピアノがその代表例。このような楽器は単独でもBGMを成り立たせることができるほか、いわゆる「右手パート」「左手パート」の両方を同時に同じBGMに使うことも可能。この楽器も使いみちは多様。ピアノなど

4. 管楽器

木管楽器と金管楽器がある。他の楽器との共通点が少ない印象的な音色をしており、メインパートになりやすい。BGMの聞き手に、元気な印象や激しい印象をもたらす。また、他の楽器との組み合わせ次第で落ち着いた印象にしたり、感動的にしたりといろいろ工夫できる。クラリネット、サクソフォーン、オーボエ、トランペットなど

5. シンセサイザー

一概にシンセサイザー、といっても設定次第で100通りの設定があれば100通りの楽器に、10000通りの設定をすれば10000通りの楽器と化すという奥深い楽器。基本的なシンセサイザーは、波の動きが一定のものが多く、他の楽器をベースに作られているものもあれば独特的な電子音を奏でるものもある。ただ、シンセサイザーがメインパートで上記の楽器のみがサブパート、というものは少ない。ほとんどがシンセサイザーのパートで、隠し味としてピアノなどの楽器が入っていることはある。また特に低音パートなどのサブパートにはシンセサイザーを使うと曲のカッコよさやバリエーションが一段と上がる。

ここからは打楽器を紹介していきます。

6. 木製体鳴楽器

打楽器の中で、やわらかい印象のある音がする。和風なシーンや落ち着いたシーン等に使われることが多い。 カスタネット、ギロ、ウッドブロック など

7. 金属製体鳴楽器

先ほどの6と同じ体鳴楽器だが、材料が金属である、というだけでも違いはある。聞き手に落ち着いた印象を与えるという特徴は変わりないのだが、木製よりもインパクトが強く、やわらかいと固いの中間地点をカバーするような楽器である。鈴のように少し落ちていた音であるものもあれば、シンバルのように激しいものもある。 シンバル、トライアングル、ベル など

8. シェイカー

楽器の中ではマイナーなジャンル。マラカスなど、粒上のものを振って音を出す、独特な楽器。音量はそこまで大きくないが、他の打楽器だけではところどころ空いてしまう隙間を埋めるためには不可欠だったりする。また独特な音であることから、サブパートには特に役に立つ。マラカス、カバサ など

9. 木製鍵盤打楽器

これとその次に紹介するものは、打楽器なのに音階がある、特殊なものである。木製鍵盤打楽器には木琴などがあり、落ち着いているが明るい音であるため、メインパートに使われることは少ない。が、例えばメインパートと全く違う音を奏でさせるなど、サブパートの中では独立したパートとして使われることが多い。シロフォン、マリンバ など

10. 金属製鍵盤打楽器

9と同じ鍵盤打楽器だが、9とは違いがある。6と7の違いでもあったように、金属製であることで落ち着いているような音色を保つつインパクトを強くしている。だから、サブパートに使われることが多いが、それだけでなくメインパートや、立ち位置がメインパートくらい大きいパートに使われることもある。鉄琴 など

11. 人体を使うもの

ここで紹介するのは楽器、というよりかは人間の体を使って出せる音のことである。ピンとこない人もいるかもしれないが、ここではハンドクラップやスナップなどのボディーパーカッションがあげられる。第4章で挙げた声も含まれる。これらを使うことで人間がBGMに溶け込んでいるように表現することができる。ハンドクラップ、フィンガースナップ など

12. 膜鳴楽器

この打楽器はジャンルが広い。そもそも膜鳴楽器とは、膜をたたくような楽器のことであり、太鼓などが挙げられる。打楽器としては不可欠な楽器で、リズムを刻ませたり聞き手にインパクトを与えたりと、数多の使い道がある。ドラム、太鼓、ティンパニ など

13. シンセサイザー

ここでのシンセサイザーとは、シンセサイザーの打楽器としての役割のことである。またシンセサイザー?と思うかもしれないが、その通りであって、結局5で挙げたシンセサイザーと設定する内容はほとんど同じなのである。ただシンセサイザーを打楽器として使うことで、5のシンセサイザーと併用することでエレクトリックな感じが増したり、BGMのカッコよさを挙げたりすることができるるのである。

ここまで楽器の種類を羅列しましたがいかがでしたでしょうか。ただ僕の拙い日本語と文章力のなさのせいでなかなか理解できなかったという方もいらっしゃると思われます。ですがまだ中3ということでお許しください。

さて、BGMにおいては楽器の種類も重要なのですが、楽器の数も重要になります。たとえかっこいいBGMを作ろうとしたって楽器の数が少なければ難しくなりますし、落ち着いた雰囲気の曲を作ろうとしたとしても楽器が無駄に多ければできません。また打楽器の支配率も重要となります。ということで、楽器の数と打楽器の支配率とBGMの印象をまとめてみました。横の列が楽器の数で、縦の列が打楽器の支配率です。なお、ここでは楽器の種類によっても曲調が左右されるため、あまり具体的な数値は書いていません。第4章と同じことを言いますが、作曲の経験があまりなく、ゲームもほとんどしない僕の独断と偏見のみに基づいています。くれぐれも参考にしそぎないように。

	楽器少なめ	やや少なめ～やや多め	楽器多め
少ない	落ち着いた、冷たい印象、感動的な印象	落ち着いてい印象	神秘的な印象
50%前後	これから曲が始まるような印象	曲の中盤で、元気な印象	愉快な印象
多い	緊迫した印象	上よりも元気な印象	激しい印象

以上です。表現が抽象的すぎて何を言っているのかわからなかったという方もいると思います。ただこれは僕が勝手に表現しているだけのものなので、なんとなく分かっていただければいいと思います。

このように、楽器の種類がBGMの制作において特に重要なのです。ただ、楽器にはいろいろあり、名前を聞いただけではどの楽器がどのような印象を与えるのかがわからない、という方もいらっしゃると思います。しかし、そんなときはとにかく楽器を触り、音を聞いて理解するのが一番無難だと僕は考えます。また、いろいろなBGMを聞いてその中になんの楽器が入っているのかを想像するのも良いでしょう。ただ聞いているだけのBGMも楽器を意識すれば見方が変わると思います。

6.さいごに

ここまでいかがでしたでしょうか。

改めていいますが、この記事は一般的な音楽の常識ではなく、僕の考えだけで構成されています。これはBGM制作において重要な知識まとめ、というものではなく、あくまで僕が考えた、BGM制作をする上で重要なポイントであるということを改めて言っておきます。

BGMというものの制作には、もちろん音楽の知識も大変重要なのですが、それ以上にゲームにおいてBGMを意識して聞いているのか、またそれによる勘があるのかも重要になってきます。あくまでここで僕が挙げたのは参考程度で、ゲームのBGM制作は結局大半は勘であり、その勘をフルに働かせた人こそよいBGMを作ることができます。

と偉そうに言いつつ、僕はこのおわりにを書いている中3現在も音楽の知識とBGMの勘には自信を持っていません。今になってようやくコード(CとかCmとかC7とかそういうやつ)を勉強しています。ただ未だにその知識は身についていなくて、やはりBGMの制作は実践の経験のみによってクオリティが増すんだなと実感しました。

この記事は8000文字を超える大ボリュームの記事となったのですが、音楽の専門知識をあまり持っていない中での執筆だったのでどうも納得がいっていません。数年間かけある程度音楽の知識を身に着けたらまたPositronに音楽関係の記事を出そうかなと思います。それまでに楽しみにしておいてください。

最後に、この部誌の執筆をサポートしてくださった物理部員の皆様、部誌を検閲・発行してくださった顧問の先生方、そしてこのPositronを読んでくださっている読者の皆様に感謝申し上げます。

また、僕は他の記事も書いていますのでぜひちらもご覧ください!

こちらの記事を読んでくださりありがとうございました!

参考文献

- <https://ja.wikipedia.org/wiki/楽器分類別一覧>

深層強化学習によるリバーシAI

はじめに

いわゆる**AI**（人工知能、**artificial intelligence**）というものは実はかなり定義があいまいで、比較的しっかりした説明をしそうなWikipediaでさえ、「～ともされる」「～もこう呼ばれることがある」と、なんとも歯切れの悪い説明をしています。要約して説明するなら、「『コンピューターで人間の知能を再現してみよう』という分野（またはその研究対象）のこと」といったところでしょうか。『AIがこれから社会で～』『我々の仕事はAIが～』といった言葉はよく聞きますが、多くの人はぼんやりとしたイメージしか持っていないと思います。注目されていると言われている「AI」とはどういった仕組みで、何ができる、何ができないのでしょうか。そういう疑問から自分の手で実際に簡易的なものを作ってみて理解を深めたいと思い、この記事を書きました。この記事を読んでそういったことが伝われば幸いです。

この記事は自分で理解を深め、知識を確かめるために書いたものもあるので、何も知らない状態では理解しづらい文、数式が多く含まれると思われます。また、内容に誤りも含まれるかもしれませんので、ご了承ください。

目次

1. 強化学習とは
 - 1-1. 概要
 - 1-2. Q学習の仕組み
2. 深層学習とは
 - 2-1. 概要
 - 2-2. ニューラルネットワークの構造
 - 2-3. 最急降下法
 - 2-4. 誤差逆伝搬法
3. 深層強化学習とは
 - 3-1. 強化学習の課題
 - 3-2. ニューラルネットワークによる解決

4. 深層強化学習によるゲームAI

- 4-1. リバーシについて
- 4-2. 学習方法について
- 4-3. 結果と考察

1. 強化学習とは

1-1 概要

強化学習（reinforcement learning） は AI が学習をして知識を獲得する**機械学習（machine learning）** の手法の1つです。強化学習の特徴はその学習方法にあります。別の機械学習の手法であり、画像認識などで用いられる**教師あり学習（supervised learning）** と比べてみましょう。教師あり学習では、前もって用意された問題（入力）とそれに対する解答（出力）をセットにした**教師データ**を使って学習を進めます。人間が何かを学ぶ時のように、自分の出した答えと解答を見比べて、誤差を修正して学習するわけです。しかし実際の場合には、学習すべき問題が多すぎる、もとになるデータベースがない、そもそもこれといった圧倒的な正解がない、といった理由で十分に教師データを用意できない場合があります。このような場合に対応しやすいのが先ほどの強化学習という手法です。強化学習では教師データをまったく必要としません。代わりに学習をする対象である**環境（environment）** に対してランダムな行動を繰り返すなどして試行錯誤を重ねます。すると環境から良い行動には**報酬（reward）** が、悪い行動にはマイナスの報酬である**罰（punishment）** が与えられて、それをもとに教師データの代わりとなる知識を自ら作り出すことができます。なので、なにかのスコアといった数値的な報酬に変換しやすい要素を最大化させるなどの行動を学習したいときに有効であると言えます。また、ランダムに試した行動がうまくいったかを学習するため、人間が思ってもみなかった答えを編み出すこともあります。今回はその強化学習の手法のうち問題解決のためにどういった行動パターンをとればよいか学習をする**Q学習（Q-learning）** というものについて考えてみます。

1-2 Q学習の仕組み

Q学習ではある**状態（situation）** における**行動（action）** の評価値となる**Q値（Q-Value）**（統計学の用語）というものを学習します。学習を通して、ある状態でのある行動が良いかどうかを正確に判断できるようになれば、すなわち評価値であるQ値を適当に出力できるようになれば学習成功ということになります。良い行動かどうかというのは、環境から与えられる報酬（マイナス値の報酬である罰を含む）のみによって判断できるので状態 s における行動 a の Q 値を $Q(s, a)$ と表すと、環境から与えられる報酬 r を用いて以下のように $Q(s, a)$ を更新すればよさそうです。

$$Q(s, a) = Q(s, a) + r$$

しかし実際にすべての行動に報酬や罰を与えられることは少ないです（もし与えられるなら教師あり学習のほうが良いかもしれません）。サッカーやバスケットボールなどの球技を環境の例として考えてみましょう。これらのスポーツでは「シュートを成功させる」という行為には点（=報酬）が与えられます。しかし、「ゴールの近くでパスを受け取る」という行動はそれ自体に対して点は与えられません。ですが、一連の流れで見ると「ゴールの近くでパスを受け取る」という行為は点に強く結びつく行動であり、高い評価を与えるべき行動だと考えられます。なのでこういった行動を「ほぼ点が入ったようなもの」とみなして時刻 t におけるQ値を次のように更新してみることにします。

$$Q(s_t, a) = Q(s_t, a) + (r + \max Q(s_{t+1}, a') - Q(s_t, a))$$

なにやらごちゃごちゃと式が増えましたが、やっていることはここまで複雑ではありません。 $\max Q(s_{t+1}, a')$ というのは時刻 t に状態 s で行動 a をとった後に、時刻 $t+1$ においてとれる行動のQ値の最大値を表します。つまりこれまでの報酬に加えて、「高いQ値のある状態に移行すること」そのものを報酬として与えるようにしています。しかしそういった状態に移行しても必ずしもその高いQ値の行動をとれるとは限りませんし、何よりたくさん更新を繰り返していくとQ値が発散してしまうので、割引率 γ ($0 < \gamma < 1$) を用いて更新式を次のように変更します。

$$Q(s_t, a) = Q(s_t, a) + (r + \gamma \max Q(s_{t+1}, a') - Q(s_t, a))$$

割引率 γ は時系列的なつながりの強い場合には1に近く、弱い場合には0に近く設定されます。これによってその後の行動との関係性の強さを設定できます。また、実際に学習を進める場合にはすべてのQ値をランダムな値で初期化してから、「もっともQ値の高い行動をとる」などのように設定し、偶発的な報酬や罰に左右されないように何回も更新を繰り返して少しづつQ値を修正します。なので1回の更新が与える影響を減らすために学習係数 α を用いて更新式を次のように変更します。

$$Q(s_t, a) = Q(s_t, a) + \alpha(r + \gamma \max Q(s_{t+1}, a') - Q(s_t, a))$$

これで一般的に用いられている更新式の完成です。上記の内容を大雑把にまとめます。

- 環境によって報酬や罰が与えられる場合はその行動のQ値がそのまま決まる
- そうでない場合は移行先の最高Q値を割り引いた値を、その行動のQ値にする
- 学習開始時はすべてのQ値はランダムに初期化され、だんだんと修正をして学習をする

2. 深層学習とは

2-1 概要

昨今のAIブームは第三次と言われていますが、その火種となったのが深層学習（ディープラーニング、**deep learning**）という技術です。これは主にもともとあったニューラルネットワーク（神経網、**neural network**）のペーセプトロン（**Perceptron**）という手法を発展させたものを指します。ニューラルネットワークは人間の脳にある神経細胞のネットワークをコンピューター上で数学的に再現を試みる、というものです。ニューラルネットワークは特定の入力に対して特定の出力をするという条件を複数同時に満たすことのできる関数を作成できたりします。なのでニューラルネットワークは知能が関数で表現できることを前提にした上で、その知能を神経細胞を模した関数の集合によって再現してみるという試みとも考えられると思います。もし実在する複雑な問題に対しても関数化が完全に可能になれば、いろいろなことができるようになると考えられます。最近になってから注目を浴びているような印象がありますが、実はニューラルネットワークの歴史自体はとても長く、人工知能研究の歴史の中でも比較的初期段階から研究されていた手法です。しかし近年のコンピューター技術の発展によりニューロン（神経細胞、**neuron**）の数を容易に増やすことができるようになり、より複雑なネットワークを形成することができるようになったため、ニューラルネットワークの発展的な分野である深層学習等の研究が加速したといえるでしょう。ニューロンの数を増やすだけでなく、そのネットワークの形を工夫する畳み込みニューラルネット（**convolutional neural network**）などの研究も進んでおり、その最たる例であるGoogleなどの画像認識等の技術には目を見張ります。

2-2 ニューラルネットワークの構造

ニューラルネットワークはニューロンが集合して形成するものなので、まずはニューロンとは何かというところから説明します。ニューロンはそれぞれの入力に対応する重み（**weight**）という係数と、しきい値（**threshold**）という係数をもち、複数の入力に対して計算をした結果を伝達関数（活性化関数、**transfer function**）に代入して主に单一の出力をします。これを数学的に表すと、

$$o = f(\sum_i x_i w_i - v)$$

(o : 出力 x : 入力 w : 重み v : しきい値 f : 伝達関数)

となります。すこし分かりにくいですが強化学習の式よりもやっていることは単純です。それぞれの入力の値がどれほど出力に影響されるべきかを決定する重みと入力を掛け合わせたものが xw です。なので $\sum_i x_i w_i$ というのは入力に重みをかけたものをすべて足し合わせるということなので、入力から取得できる情報を、何が重要か判断して取捨選択するという側面があります。その

合計値からしきい値を引き、伝達関数に代入したものが出力となります。しきい値には誤差などの比較的小さな情報を無視するようとする効果があります。伝達関数にはいろいろな関数が使用され、それぞれで効果は違いますが、極端に小さな値や極端に大きな値の影響を小さくするといった効果のものがあります。また、誤差逆伝搬法（後述）に、この伝達関数の微分が深く関わっているため学習の精度や速度に直結する部分であると言えます。一般的には

$$y = \frac{1}{1 + e^{-x}} \quad (\text{シグモイド関数})$$

$$y = \begin{cases} x & (0 < x) \\ 0 & (x \leq 0) \end{cases} \quad (\text{ReLU関数})$$

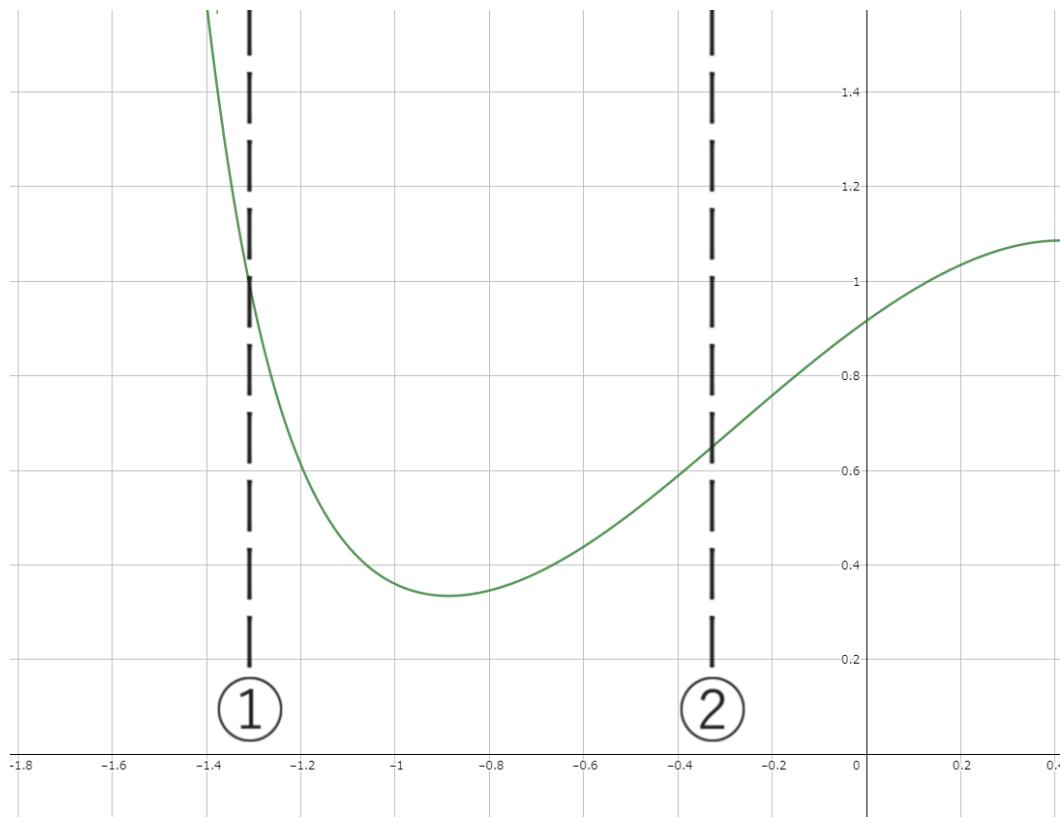
などが用いられることが多いです。こうして計算を行う複数個のニューロンを層状に構成し、それぞれのニューロンの出力結果を次の層のニューロンの入力にするというものが一般的にニューラルネットワークと呼ばれるものです。深層学習では、入力を受ける入力層、情報を伝える中間層、出力をする出力層などの層が3~4以上組み合わさったものを使った学習がもっとも多く見られます。また、入力層や出力層のニューロンを増やすことで、複数の入力に対して複数の出力することも可能です。

2-3 最急降下法

ニューロンの組み合わせで任意の関数を再現するには最急降下法（**steepest descent**）と誤差逆伝搬法（**backpropagation**）によってそれぞれのニューロンの重みとしきい値を正しく修正することが必要です。詳しくやろうとすると必要な数学的知識が多く、少しややこしいので今回は簡単に説明しようと思います。まず、 x についての関数 $f(x)$ で、出力結果をある値に近づける方法について考えてみます。調整する前の値として $x = n$ を代入して、出力結果を近づけたい値を t とおきます。「 $f(n)$ が t に近づくように n を変更する」ということは「 $f(n) - t$ が 0 に近づくように n を変更する」ということと同じなので、

$$E(x) = \frac{1}{2} \{f(x) - t\}^2$$

とおいて、ここで目標を「 $E(n)$ が 0 に近づくように n を変更する」に置き換えてみます。ここで n の変更具合を決定するのに使うのが微分です。 $y = E(x)$ のグラフが以下のようになっている場合を例に考えてみます。



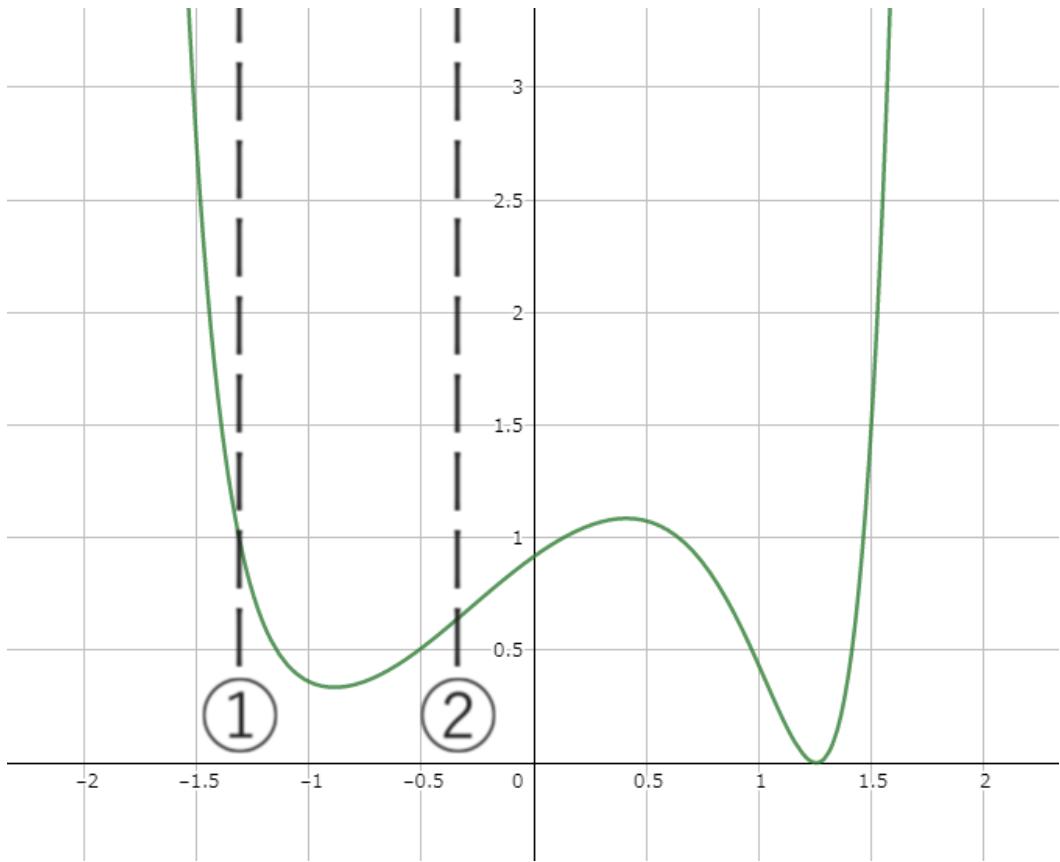
$x = n$ が①である場合、 $E(n)$ の減る方向に変更すればよいので n を増やせばいいことがわかります。これは $x = n$ で傾きが負であるからです。逆に②の場合は傾きが正であるので n は減らせばいいとわかります。傾きは

$$\frac{dE}{dx} = \frac{dE}{df} \frac{df}{dx} = \{f(x) - t\}f'(x)$$

で求められるので定数 α ($0 < \alpha$)を用いて

$$n' = n - \alpha \{f(n) - t\}f'(n)$$

とすればよさそうです。この更新を何回も繰り返していくというのが最急降下法の方法です。ちなみに、この式に含まれる定数 α は学習速度、精度ともに深く関わっている定数です。もし小さすぎれば、変化量が少なすぎて何回も調整し直さなければなりません。



また、先ほどの範囲で最小であった $n = -0.9$ 付近は局所的なもので、全体でみると $n = 1.3$ あたりが最小である、という **局所解 (local solution)** についての問題も変化量が小さい場合には発生しやすくなります。逆に、定数が大きすぎる場合には変化量が大きすぎて最適解を通り過ぎてしまい収束が遅くなります。これは $\alpha > 1$ の場合に何が起こるか、単純な $y = x^2$ で考えるとわかりやすいです。

2-4 誤差逆伝搬法

前項で説明した最急降下法を、関数であるニューロンが複数個組み合わさったニューラルネットワークに組み込むのが誤差逆伝搬法です。説明の前に少しニューロンの式を以下のように変形してみます。

$$o = f(\sum_i x_i w_i)$$

(o : 出力 x : 入力 w : 重み f : 伝達関数)

2-3の式からしきい値がなくなりました。これはしきい値もある一定の入力（定数）に重みをかけたものとみなすことができるからです。重みひとつだけを変数とみなしたとき、ニューロンは $o = f(\sum_i x_i w_i)$ で表され、入力 x_i やほかの重み付き入力は定数とみなせるので（偏微分）さきほどの例に当てはめることができます。伝達関数をシグモイド関数にした場合の伝達関数の $f(x)$ の微分は

$$\begin{aligned} f'(x) &= \left(\frac{1}{1 + e^{-x}} \right)' = -\frac{(1 + e^{-x})'}{(1 + e^{-x})^2} = \frac{e^{-x}}{(1 + e^{-x})^2} \\ &= e^{-x} \left(\frac{1}{1 + e^{-x}} \right)^2 = e^{-x} f(x)^2 = \left\{ \frac{1}{f(x)} - 1 \right\} f(x)^2 = f(x)\{1 - f(x)\} \end{aligned}$$

となるので出力層の重みの更新式は、伝達関数の微分などを2-3の式に当てはめ、重みの係数である x_i を考慮して、

$$w'_i = w_i - \alpha(o - t)o(1 - o)x_i$$

と考えられます。これが出力層のニューロンの最急降下法です。ここから出力層のひとつ前の中間層のニューロンの重みを調整することを考えます。出力層とは違い、ニューロンの出力が直接誤差の式に組み込まれていないので、少し追加する必要があります。しかし基本的な考え方方は先ほどとまったく同じです。誤差である E を 0 に近づければよいのでまずは E を中間層の重み w_a で偏微分してみます。

$$\frac{\partial E}{\partial w_a} = \frac{\partial E}{\partial o_b} \frac{\partial o_b}{\partial o_a} \frac{\partial o_a}{\partial w_a} = (y - t)(1 - o_b)o_b w_b(1 - o_a)o_a x_a$$

(x_a, w_a, o_a : 中間層の入力、重み、出力 x_b, w_b, o_b : 出力層の入力、重み、出力

E : 出力層の誤差 t : 目標値)

なので中間層の重みの更新式は次のようにすればいいとわかります。

$$w'_a = w_a - \alpha(y - t)(1 - o_b)o_b w_b(1 - o_a)o_a x_a$$

また、さらに層が重なっていく場合にはさらにそれぞれの微分を加えればよいです。こういった式がまさに誤差逆伝搬法の中身であり、この方法では、微分によって出力層の誤差がそれぞれのニューロンに逆行して伝搬されていると言えます。

3. 深層強化学習とは

3-1 強化学習の課題

強化学習はさまざまな課題に対処できる手法ではありますが、実際の問題にすんなりと適応できるかというとそうではありません。大きな課題のうちの1つに「多すぎる状態を保存しきれない」というものがあります。Q学習において、すべてのQ値は状態と行動を対応させた状態で保存されます。単純な問題であれば、保存すべき状態も行動の選択肢も少ないために簡単に保存できます。しかしボードゲームなどの比較的単純な問題でさえ、数手進めるだけで何億通りもの盤面があり、現在のメモリでは保存しきることができません。（将来的に技術が進歩すれば解決できる問題というわけでもありません）実際の問題だったらなおさらです。Q値の保存ができないければもちろんQ学習を進めることはできません。この問題を解決するのに役立つのがニューラルネットワークです。

3-2 ニューラルネットワークによる解決

ニューラルネットワークは複数の入力に対して特定の出力をするように学習でき、これは複数の変数をもつ任意の関数の疑似的な模倣ができるといいかえられます。なので先ほどの問題は、状態を複数の変数で表現し、Q値を出力する関数をニューラルネットワークによって学習することで解決できると考えられます。具体的な例で考えてみましょう。先ほどと同じようにサッカーやバスケットボールといったスポーツで一プレイヤーについての行動を例にしてみましょう。こういった球技で変数として表すのがよさそうな要素をリストアップしてみます。

- 自分の位置、速度
- 味方と敵の位置、速度
- ポールの位置、速度
- 残り時間

これらは数値で表現可能な情報なので変数として扱うことができます。これらの入力に対して、「前に進む」などについての評価値を算出し、それに基づき行動を行います。そして行動をとったから報酬や罰によってQ値を修正します。この場合、Q値はニューラルネットワークによって構成されるひとつの関数の計算結果として表現されるので、さきほどのニューロンの重みを修正することによって任意の出力に修正できます。つまり、知識をニューロンの重みとして蓄積することができます。

4.深層強化学習によるゲームAI

4-1 リバーシについて

リバーシは白と黒が一面ずつある石と、 8×8 マスに区切られている盤を使って2人でプレイされる完全情報ゲームです。とても有名ですが主なルールを明文化しておきます。

- プレイヤーは基本的に交互に1つずつ石を置いていく
- 自分の番に置く石は、置いてある自分の石と相手の石を直線的に挟めるマスにのみ置くことができる
- 置いた石と自分の石で挟まれた相手の石はすべて自分の石になり、裏返る
- おける場所がない場合にはパスとなる
- おける場所がある場合には必ず石をおかなくてはならない
- 相手も自分もおける場所がない場合にはゲーム終了となる
- ゲーム終了時に石の多いプレイヤーが勝利する
- ゲーム開始時には中央4マスにそれぞれの石が2個ずつ互い違いに置かれる

今回リバーシを学習対象に選んだ理由は以下の通りです。

- 8×8 マスなので入力となる情報が単純すぎず複雑すぎない
- 手順通りに進めれば毎回同じ結果になる
- ルールが単純で広く知られているゲームである
- ほかにも人工知能でやってる方がいる
- 実際にすぐに対戦できるので強さがわかりやすい
- ゲームの実装が複雑でない

4-2 学習方法について

もちろん学習方法は多層パーセプトロンによる深層学習を採用しました。 詳細は以下の通りです。

- 第1層～第3層からなりそれぞれの層は128、64、64個のパーセプトロンから構成される
- 練習相手となる簡易的なコンピュータープレイヤーの強さが適切になるように、そのふるまいを直近100試合の勝率を用いて以下のように設定

勝率(0~1)の確率でランダムに石の置き場所を決定する
そうでない場合は半分の確率で、2手先の時点で相手の石より自分の石がなるべく多くなる
ようなマスに置く
そうでない場合は同様に3手先の時点で多くなるようなマスに置く

- 入力はそれぞれのマスに白が置いてあるか、黒が置いてあるか、空白であるかという状態をそれぞれ0と1で表したものに設定。つまり0または1で表される $64 \times 3 = 192$ 個の要素から構成される
- 出力層はそれぞれのマスに置くことのQ値 ($0 \leq Q \leq 1$) に設定。つまり0~1で表されるそれぞれのマスの評価値が計64個出力される
- ニューロンとQ学習に関する定数等は以下のように設定

 Q 学習の学習係数 $\alpha = 0.1$ 割引率 $\gamma = 0.9$ ニューロンの学習定数 $\alpha = 0.05$ 出力の誤差の許容値 $T = 19.2(0.1 \times 192)$ 伝達関数

シグモイド関数 学習対象となる

試合数 500

また、Pythonのモジュール等で手軽に実装することは可能でしたが、理解を深めるため、今回は何もない状態からC++でコーディングしました。

4-3 結果

ある程度の客観的な指標になると思い、石を置ける場所からランダムに選択するコンピュータープレイヤーとの1000試合の結果をここに示しておきます。

試行	勝ち	引き分け	負け
1回目	937	7	56
2回目	812	4	184
3回目	847	25	128
平均	865.3	12	122.7

結果、ランダムな手に対しての勝率は9割ぐらいを予想してましたが約8割5分でした。

ちなみに人間との対戦結果は、人がたまに負けるくらいの印象でした。

おわりに

今回の制作を通して、その仕組みがわかり今まで朦朧としていた印象が固りました。また、大企業などの膨大なデータベースを用いたAIの強力を強く実感しました。次はより複雑な問題や実生活に密接な問題などに挑戦したいと思います。

参考文献

- ・ 小高 知宏、『強化学習と深層学習 -C言語によるシミュレーション-』、オーム社、2017
- ・ 『誤差逆伝播法を宇宙一わかりやすく解説してみる』、ロボット・IT雑食日記、
2018/6/14、閲覧日2021/9/30、<https://www.yukisako.xyz/entry/backpropagation>

アーキテクチャへの扉

はじめに（免責）

この記事はネットや本などで知識だけを身に着けた経験の浅い人が書いています。この人はこんな考え方をしているんだなあ位にこの記事を読み流してきちんとした本を読むことをお勧めします。本記事の考えのもとになっている本はを最後に紹介します。

本記事の目標

本記事ではプログラムとはどんなものか、またオブジェクト指向とは何かを説明しながらプログラムの大まかな構造を考えていきます。最終的に、良いと言われているようなプログラムの構造を考えられればなと思います。

プログラムの評価は使う人が直接触るわけではないので、いわゆる絵や音楽などの受け手からの評価とは違ってきます。プログラムに触るのは仕様を変更したり追加したりする同業者または自分です。つまり、プログラムはそれを作り替える人から評価を受けるわけです。よって良いプログラムとは仕様を変更したり追加したりしやすいプログラムということになります。

プログラムを評価する指標は大まかに三つほどあります。一つ目は読みやすさです。読みづらければプログラムを理解することが大変になってしまいます。二つ目は再利用のしやすさです。同じようなものを何度も書いているとプログラムが無駄に大きくなってしまいますし、それだけ労力がかかってしまいます。三つめは正しさです。当然ですがバグが多いプログラムはプログラマに対してだけでなくユーザーに対しても被害が出ます。

プログラム

コンピュータプログラム（英：computer program）とは、コンピュータに対する命令（処理）を記述したものである。

Wikipediaからプログラムの説明を引用してきました。プログラムは仕事の手順を示した静的なデータです。



<https://cookpad.com/recipe/2477629>

これは生チョコのレシピの一部です。プログラムはこのような料理のレシピと同じように「最初にこれをやる、次にこれをやる……」ということが書かれたデータです。

料理をするときには材料が必要となり、その材料をもとに食べ物を作ります。プログラムも同じく、あるデータを使って決まった仕事を行い、得られたデータを出力します。つまり、**プログラムは入力、処理、出力で構成されています**。（このうちのどれかが欠けることはあります）関数について考えてみても、この構成要素がわかります。関数は引数が入力され、中である決まった処理をして、戻り値を出力します。関数はそれ自体が一つのプログラムになります。

入力の段階で処理しやすいデータになっていると関数などで中に書くコードが少なくなるのでプログラムが単純になります。料理でいうところの下ごしらえです。3分クッキングでは下ごしらえを完璧にすましてあるので実際の処理内容（レシピ）がとても分かりやすくなっています。関数の中でデータを動的に取得して（現在時刻の取得など）から、そのデータと引数を使って処理をしたいことがあると思いますが、やってはいけません。これは処理の部分でデータを用意していることになります。また、ある引数を入れたときに常に同じ戻り値が返ってこないことになります。これは、入力と出力を見てプログラムが正しく動いているかどうか判断することができなくなるのでやってはいけません。現在時刻なども外で取得してから関数の引数などに渡しましょう。

プログラムの依存関係

プログラムはプログラムの中で使うことができます。関数の場合関数の呼び出しと言ったりします。依存関係は使うものと使われるものの関係です。使うものは使われるものに依存しているといいます。車で考えてみましょう。車はタイヤを使います。タイヤは車に使われます。車がなくてもタイヤの機能は成り立ちますが、タイヤがないと車の機能は成り立ちません。つまり、タイヤは車に依存しておらず、車はタイヤに依存していることがわかります。AプログラムがBプログラ

ムを使っているとすると、BプログラムがないとAプログラムを使うことができませんが、AプログラムがなくてもBプログラムを使うことができます。なので、AプログラムはBプログラムに依存していることとなります。

さて、車はタイヤに依存していると話しましたが一般的に物が製品を使う、つまり製品が部品に依存するという関係が成り立っています。しかし、プログラムの中では部品と製品との違いがあいまいなことがあります。ではどのようにプログラムの依存関係を考えるかというと、より根本的かつ汎用的なプログラムのほうに依存します。製品と部品では部品のほうが汎用的です。汎用的な部品は一つの製品だけでなく、様々なところで使うことができます。タイヤはネジに依存していますが、ネジを使っている製品はとてもたくさんあります。また、AプログラムがBプログラムに依存しているとBプログラムの仕様が変わったときにAプログラムの仕様を変えなければならない可能性があります。しかし、その逆はありません。ネジの太さが変わってしまうとそのネジを使っているタイヤは使えなくなってしまいます、タイヤのネジ穴が太くなってしまってもネジが使えなくなることはありません。

アプリケーション

アプリケーションはプログラムその物ではありません。入力、処理、出力では説明できないからです。ただ、アプリケーションの仕様を分解するとそれぞれを入力、処理、出力で説明できるようになります。ここからアプリケーションは複数のプログラムが集まって出来ていることがわかります。

アプリケーションのユーザはパソコンやスマホなどのデバイスを操作して画面に何かが表示されたり音が鳴ったりすることを期待します。ユーザの操作がプログラムの手順を始めるトリガーとなり、処理に必要なデータを作ることがあります。

ユーザの操作から得られるデータ以外にアプリケーションやサーバ上にあるデータが必要となるときがあります。これらのデータを得るためにデータアクセサが必要になります。ユーザの操作によって得られるデータや、データベース、ファイルなどのデータはそのままでは処理に向いていないことがよくあります。よって、**処理に必要なデータを処理しやすいデータに直す**プログラムが必要になります。

ロジックに必要なデータは様々なところから持ってくるので、それらを一括にまとめてロジックに渡せると便利です。様々なデータとプログラムを繋ぐことからこのプログラムをロジックと名付けます。

出力されるデータもただのデータでしかないので、出力されたデータを解釈して画面に表示したり音を鳴らしたりなどのユーザの体験を提供するプログラムが必要です。

これらを踏まえてアプリケーションの中のプログラムの基本的な構造を考えてみます。

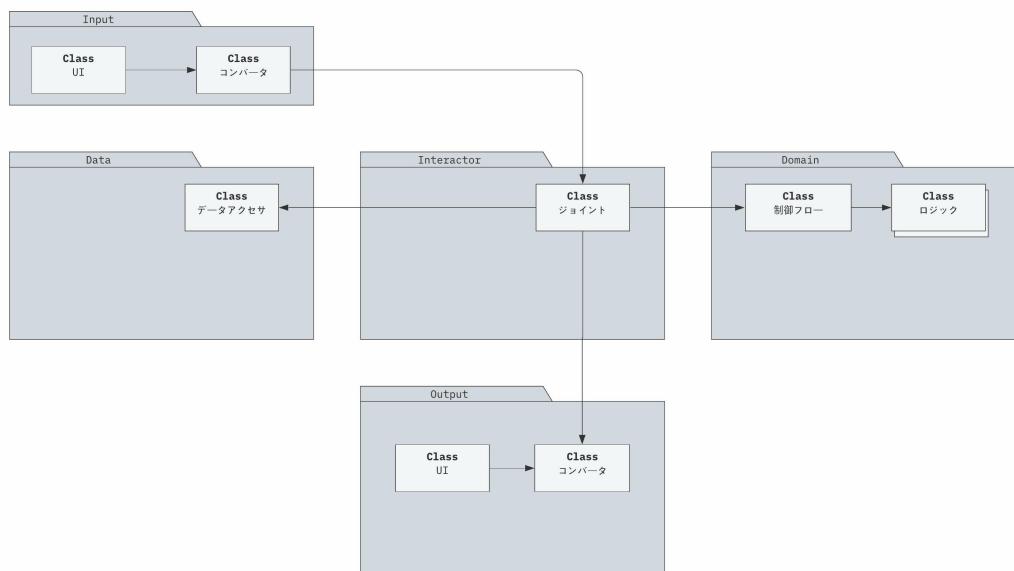
- ユーザの操作で処理がスタートします
- ユーザの操作で得た入力データを処理しやすいデータに変換し、「ジョイント」に入力します。
- 「ジョイント」で「データアクセサ」からファイルやデータベースのデータを持ってきて「ロジック」に入力します
- 「ロジック」で入力されたデータに対してある決まった処理を行い、得たデータを「ジョイント」に返します
- 「ジョイント」で帰ってきたデータを複数の「出力」に渡します。
- 「出力」で渡されたデータを解釈して画面に表示したり、音を鳴らしたりと様々なことをします

これら一つ一つのプログラムも入力、処理、出力で出来ていることがわかります。

では、依存関係の話をアプリケーションのプログラムに応用してみましょう。アプリケーションのプログラムは基本的に入力、ロジック、出力、データアクセサ、ジョイントでできています。アプリケーションにとって根本的なのはロジックです。ロジックには仕様その物が書かれています。また、ロジックは同じ仕様の別アプリケーションでも使うことができます。

一方、入力、出力はアプリケーションごとにUIが異なり、UIが変わってもアプリケーションが成り立つ（パズドラは何回かUIが変わっています）ことから根本的ではないことがわかります。また、データアクセサもファイルの拡張子やデータベースの種類などによってアプリケーションの根本的な仕様は変わらないので根本的ではありません。

そして、ジョイントは入力、出力、データアクセサとロジックの間を取り持つので根本度も中間になります。よって、ジョイントがロジックに依存して入力、出力、データアクセサがジョイントに依存することになります。しかし、ジョイントはデータアクセサや出力を参照する必要があるのでこれらの依存関係を守ることは難しくなってしまいます。



アプリケーション構造_手続き

オブジェクト指向

これまで話した構造がBasicやC++などの手続き型言語で出来る構造です。この構造をC#やJavaなどのオブジェクト指向言語を使うことでさらに強化していきます。（PythonやC++などのオブジェクト指向言語では対応していない機能がありますが、代替手段があります。多分）ただ、その前にオブジェクト指向の機能はどんなものがあるか確認していきましょう。

クラス

クラスは型と呼ばれることもあります。型、つまり物を作るときに元となるものです。クラス自体は使うことができません。クラスを使って物を作るのがですが、その物をインスタンスと呼びます。クラスには主にフィールドと呼ばれるデータ（変数）とメソッドと呼ばれるフィールドを使ったプログラムを定義することができます。クラスには二つの使い道があり、一つ目はインスタンスを生成することで、二つ目はインスタンスを入れる変数にすることです。

インスタンス

インスタンスの中にはクラスで定義されたデータとプログラムが入っています。このデータやプログラムは公開するか非公開にするかクラスで決めることができます。公開するとインスタンスの外で使うことができて、非公開にするとインスタンスの内側でしか使うことができません。

インスタンスには二種類の使い方があります。データのまとまりと共通のデータに対するプログラム群です。これは二種類の使い方どちらでも使えるわけではなく、どちらか一つの使い方に絞る必要があります。データのまとまりとして使うインスタンスをデータ構造、共通のデータに対するプログラム群として使うインスタンスをオブジェクトといいます。データ構造もオブジェクトもデータが主となっていることがわかります。よって、クラスはデータのまとまりとして名前を付けるべきです。

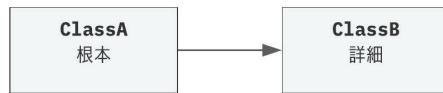
データ構造はデータのまとまりです。それ以上でもそれ以下でもありません。プログラムにデータを渡したりデータを出力したりするときに使います。データ構造はただのデータなのでクラス内には公開フィールドしか定義してはいけません。

プログラムは入力、処理、出力で出来ていると話しました。当然メソッドもそれに従わなければなりません。メソッドの入力は引数のほかにフィールドがあります。（メソッドにフィールドを入力しない場合、そのプログラムはメソッドではなく関数で実装した方がよいことになります）引数はメソッド内の処理を実行するときに入力するデータなのに対し、フィールドはあらかじめ決めておくデータになります。メソッドを使うときには引数しか設定しないのでフィールドをころころ変えてしまうとある引数に対して常に同じ戻り値が返ってこないことになります。よって、オブジェクトのフィールドは外からも（できれば中からも）変更できないようにするのが好ましいです。ここでフィールドを非公開にする必要が出てきます。オブジェクト内のデータが欲しいときには公開したいデータだけをまとめた専用のデータ構造のクラスを作り、そのインスタンスを返すメソッド（またはプロパティ）を返します。

インターフェース

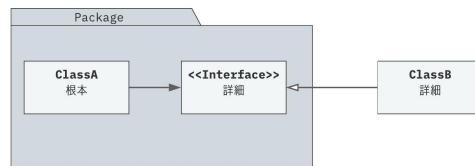
インターフェースは各プログラムのつなぎ目です。インターフェースには実装されていないメソッド（またはプロパティ）を定義することができ、それらをクラスに実装させることで間接的にクラスのメソッド（またはプロパティ）を使うことができます。インターフェースは複数のクラスを「同じ機能を持つもの」として同じように扱えます。インターフェースを変数にすることによってその変数にはインターフェースが実装されているクラスだったらなんでも入れることができます。

また、インターフェースを介してクラスをつなぐことによって依存関係を逆転することができます。例えば、根本的なロジックを持つクラスAが詳細的なロジックを持つクラスBを参照したいとします。



依存関係逆転の問題

しかし、これだと根本的なクラスが詳細的なクラスに依存していることになってしまいます。そこでクラスA専用のインターフェースBを用意してクラスBがそれを実装します。



依存関係逆転の解決

インターフェースBはクラスA専用なのでクラスAが欲しい機能だけが定義されています。クラスAの仕様が変わりインターフェースBに対して必要な機能が変わるとインターフェースBは中身の処理を定義していないので簡単に機能を変えることができます。インターフェースBの機能を変えるとクラスBを変更するかインターフェースBの機能を持った新しいクラスを作る必要があります。これで詳細的なクラスBが根本的なクラスAに実質依存することができます。

抽象クラス

抽象クラスはインスタンスを生成することができず、型として使います。その代わり抽象クラスを基に派生クラスをつくることができ、派生クラスは抽象クラスとして扱うこともできます。派生クラスはその名の通り抽象クラスから派生したものです。よって派生クラスは抽象クラスとして扱える必要があります。イメージとしては抽象クラスが種類、派生クラスが物となります。抽象クラスは厄介な機能なのであまり使う頻度は高くありません。というか安易に使ってはいけません。機能をまとめたいだけならインターフェースを使いましょう。

オブジェクト指向を使ったプログラムの構造

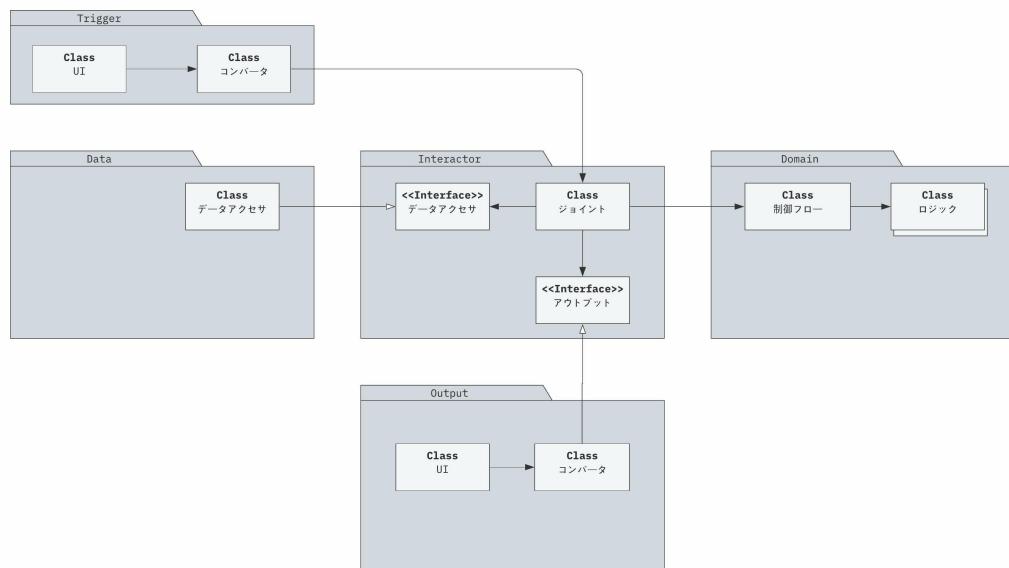
オブジェクト指向の主な機能を確認したところでこれをアプリケーションのプログラムに適用してみましょう。オブジェクト指向は再利用のしやすさに特化しています。

ソフトウェアのプログラムは入力、出力、ロジック、ジョイント、データアクセサで出来ていること、そして入力と出力がジョイントに依存していてジョイントがロジックに依存しているべきだということをプログラムの章で話しました。関数だけではこの依存関係にすることが難しいですが、オブジェクト指向の機能であるインターフェースによって依存関係を逆転させることで可能になります。

インスタンスを使うことによってデータとプログラムが一体となり、処理プログラムを再利用できるようになりました。そこで、様々なデータに対する処理をそれぞれクラスで定義し、それを使って一つの大きな処理をこなすプログラムを作ります。このような小さなロジックを複数フィールドに持ち、それらをつかって大きな処理をするメソッドを持つオブジェクトを制御フロー（またはトランザクション）と名付けます。

ジョイントがデータアクセサのインターフェースに依存することでファイルの拡張子やデータベースのツールが変わったときもジョイント部分を一切変更せずにデータアクセサを付け替えるだけでよくなります。

これらの変更を加えたプログラムの構造を図にしてみます。



アプリケーション構造

この構造がアプリケーションの中に複数できることになります。ファイルやデータベース、画面表示やサーバ通信はそれぞれ例です。実際はこの構造からクラスを付け足したり減らしたりして仕様にあった構造を作ることになりますが基本的にこの構造を維持することでそれぞれの機能を

再利用しやすくなると思います。また、使うツールによっては変換が必要ないデータが出てくる可能性があります。しかし、必ず別の処理専用のデータ構造にデータを移す必要があります。なぜなら、処理で扱うデータ構造と入力、出力で扱うデータ構造と一緒にすると入力、出力の都合でデータ構造を変更しなければいけなくなったりしたときに処理もその影響を受けてしまうからです。

先人の知恵

今回のプログラムの構造に使われている知恵

- DRY(Don't repeat yourself)
- OAOO(Once and only once)
- SRP(Single Responsible Principle, 単一責務の原則) 注：名前に惑わされて間違った説明をしていることがあります
- OCP(Open Closed Principle, オープンクローズドの原則)
- ISP(Interface segregation principle, インターフェース分離の原則)
- DIP(Dependency inversion principle, 依存性逆転の法則)
- SDP(Stable-dependencies principle, 安定依存の原則)

プログラムの構造を考えるうえでの知恵

- GOFのデザインパターン 注：目的ではなく手段です
- DDD(Domain-driven design, ドメイン駆動設計)
- LSP(Liskov substitution principle, リスコフの置換原則)
- 求めるな、命じよ
- デルメルの法則

読みやすいコードにするための知恵

- PLS(Principle of least surprise, 驚き最小の原則)
- ループバックチェック

正しいプログラムを作る上での知恵

- テスト駆動設計
- ハンブルオブジェクトパターン

おすすめの本

- リーダブルコード（清和書林にあります！！）
- Clean Architecture
- オブジェクト指向でなぜ作るのか（清和書林にあります！！）

さいごに

これまでつらつらと書いてきましたが、小さいアプリケーションではこの構造を完璧に守らなくてもあまり問題にならずにかけちゃいます。しかし、プログラムの構造を適切に作ることによって機能を拡張しやすくなるのは事実です。この記事でプログラムの構造を作る上で様々な考え方があることを知ってもらえたなら幸いです。

AMDのZen3について考える

高1 山路開

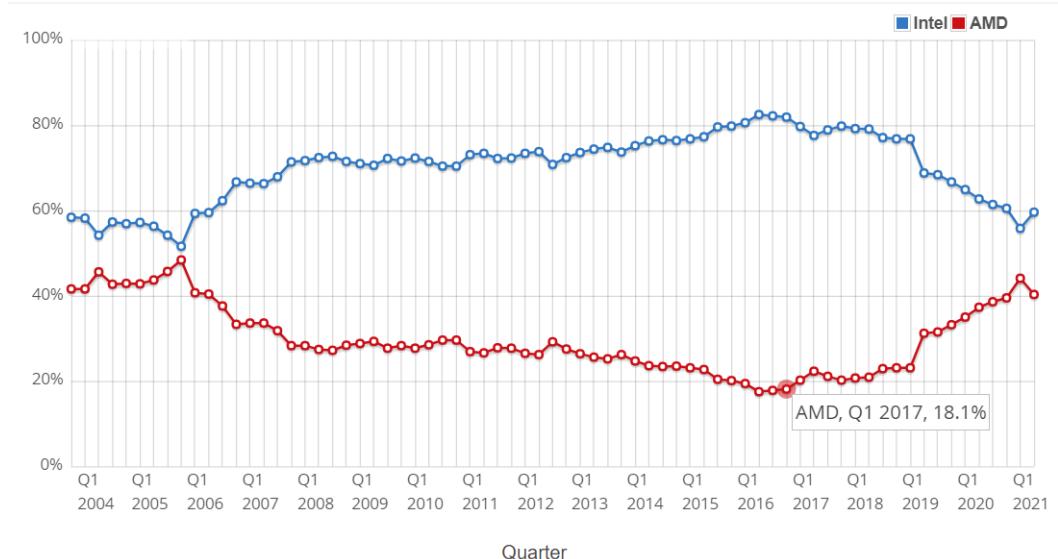
はじめに

昨年の10月、AMDがRyzen5000シリーズを発売し、CPU業界は大きく盛り上りました。この部誌ではどうしてRyzen5000シリーズがあれほど大きな話題を呼んだのか考えたいと思います。RyzenThreadRipperシリーズに関しては大きく省いている点があること、また筆者の個人的な意見を多く含んでる場合がありますのでご了承ください。

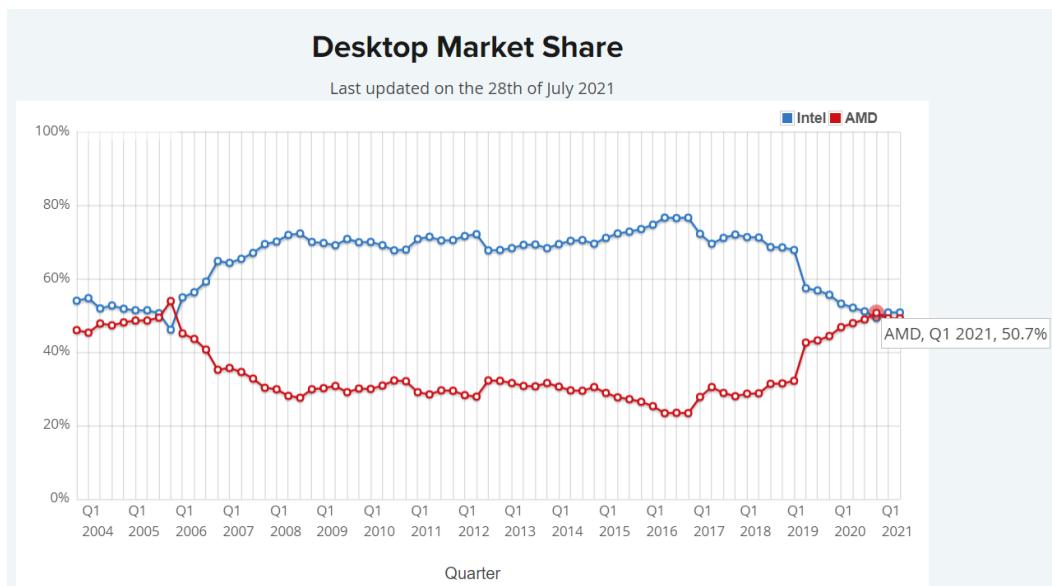
そもそもRyzen5000シリーズとは

そもそもRyzenシリーズはAMDが2016年12月13日に発表したCPUのブランドです。これにより当時はIntelがCPU市場を独占していた状況に大きく風穴を開けました。

Ryzen1000シリーズはIntel製CPUに比べ、安価でクロック数が低い一方で、コア数、スレッド数が多いという点で人気を集めました。同時にIntelはCPUの開発に苦戦しており、多少性能を向上させた程度ものを別の世代として販売していたことから、Intelを見捨てた消費者も多かったことでしょう。



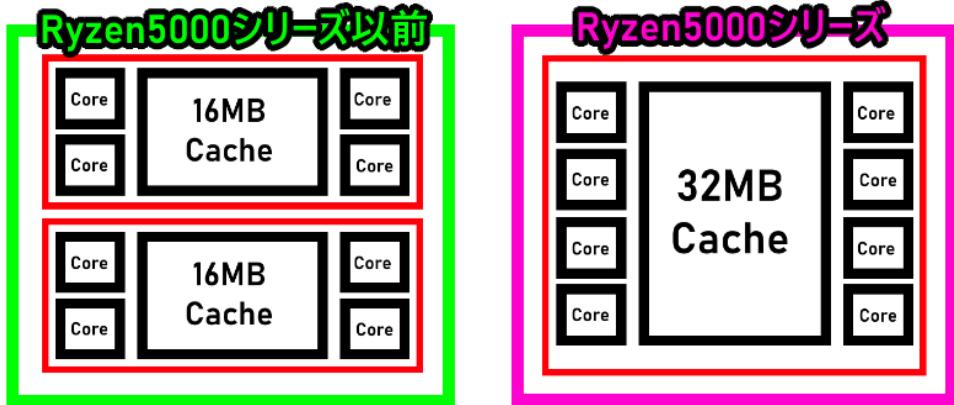
グラフを見ればわかるように、Ryzenの発売直後の2017Q1期(1月～3月)に、今まで下がり気味であったAMD製CPUのマーケットシェア率は上昇に転じています。



またデスクトップCPUに関しては2021,Q1(1月～3月)時点ではIntel製CPUのマーケットシェアを追い抜いています。これはおそらくRyzen5000シリーズ発売の影響でしょう。いろいろなWEBメディアでも大きく取り上げられていましたからね。ちなみに混乱した方もいらっしゃったかもしれません、Ryzen5000シリーズはRyzenの第4世代の製品です。(Ryzen4000シリーズは第3世代のセキュリティ強化モデル)

Ryzen5000シリーズの特徴

今までのRyzen7,Ryzen9シリーズでは、4コア+L3キャッシュ16MBを1グループとしてそれを複数搭載することによって多コアを実現していましたが、Ryzen5000シリーズからは8コア+L3キャッシュ32MBを1グループとすることでコア間のデータ転送速度を大幅にアップさせ、また1つのCPUが大量のL3キャッシュに瞬時にアクセスできるようになりました。これによりCPUの性能は19%アップしたとされています。参考画像ボ^ソッ



Ryzen5000シリーズのRyzen9ではこのグループを2基、Ryzen7では1基搭載しています。こうして性能が飛躍的に向上したにもかかわらず、Ryzenのコストパフォーマンスは維持され続けています。まあ実際すごいのはAMDではなくこのCPUの設計と製造を担当しているTSMCなんだけれどね。

この先のRyzenについて

すでにRyzen6000シリーズ(Zen4)の情報はリークされていますが、最大の違いはなんといってもソケットの変更でしょう。今までのRyzenは、デスクトップ向け製品に関してはずっとAM4という規格のCPUソケットを採用し続けてきました。そのおかげでユーザーはマザーボードメーカーが対応してさえいれば「Ryzen1000シリーズのCPUを使ってきたけれどそろそろ性能に不満が...」となった際に簡単に世代を超えてCPUをアップグレードすることができたのです。しかしソケットの形状が変わってしまうとそれもかないません。~~10年ほど前からIntelは2世代ごとにソケット形状を少しずつ変化させるという暴挙に出ていますが~~今までこのアップグレードの簡単さがRyzenシリーズの売りであったような気もするので、これからパソコンを買う人の中ではRyzenシリーズを避ける人も出てくるかもしれませんね。まあまだリーク段階なのでソケット形状が変わると確定したわけではありません。今後の最新情報に期待ですね！

終わりに

参考文献

- CPUシェアグラフ https://www.cpubenchmark.net/market_share.html
- AMD CCXに関する情報 <https://www.gdm.or.jp/review/2020/1105/368230>

芸術の重要性について

1.はじめに

皆さんこんにちは、またまた登場の中学生明松です。

さて突然ですが、皆さんは音楽、美術、書道といった芸術について興味を持っていますでしょうか？もしかしたら「芸術を学んでも意味がない」「芸術は役に立たない」と考えている方もいらっしゃると思いますが、僕は、そう考えるのは甘いと思っています。

皆さんは芸術というと何を思い浮かべるでしょうか？例えば昔の芸術家の絵画とか、有名音楽家の音楽などといったものを鑑賞したり、絵や彫刻を作らされたり、音楽を演奏したりすることを思い浮かべる人もいらっしゃると思います。しかし芸術とはそんなに幅の狭いものではありません。古い芸術を見ながら新しい芸術を切り拓いていく、それが芸術だと僕は考えます。具体のことについてはこの章以降で説明しますが、学校で習うような国語、数学、社会、理科などの教科同様、芸術もまたこの世界には欠かせません。

物理部でもプログラミングや電子工作にとどまらず、3DCG制作や映像編集、作曲といったこともしている部員もいたり、そうでなくともそれらを趣味にしている部員もたくさんいます。僕がこの記事を書こうと思った理由は、芸術に未知の可能性を秘めている(?)物理部の部員である僕から芸術の存在意義をまだ見いだされていない方に芸術の重要性を伝えたいと思った他、僕自身に対しても「この記事を書いて改めて芸術と向き合ってみよう」と思ったからです。

もちろん僕は芸術関係の活動がまだまだ他の人に比べては浅いため、芸術の本質からはかけ離れている事があるかもしれませんし、これはあくまで僕の考えなので、他の人が思っていることとは異なるかもしれません。あくまで1人の人間の考えと思ってください。また、物理部の活動内容とはズレている事があるかもしれませんのが了承ください。僕の拙い日本語もご愛嬌ということです。

物理部員の方々を含め、ごゆっくりお読みください。

2.なぜわたしたちは芸術を学ぶのか

そもそも私達はなぜ学校の授業として芸術を学ぶのでしょうか。僕が勝手に考えたことを説明します。

もちろん、芸術を学ぶのは学校側が芸術を使って金を稼ぐことを促しているからである、ということはありません。

まず理由の1つとして、「芸術とはなにか」を学ぶためということがあります。もちろん人間は生まれてから何かしらの方法で教えられるまでは「芸術」の概念はわかりません。しかも、「芸術

とはなにか」というものが芸術についての知識が何もない状態で端的に言葉で伝えられても分かる人は少ないでしょう。鑑賞・実践などを繰り返すことでようやく「芸術とはなにか」がわかるようになります。

2つ目は、芸術的なセンスを発揮するためです。人間は生まれながらにして芸術的センスを持っているのですが、唯持っているだけでは発揮することはできません。芸術を鑑賞したり実践したりして芸術的センスを磨き、発揮できるようにするということです。

3つ目には、芸術作品を芸術作品として捉える能力を育むためということもあります。もし学校で芸術について学んでいなければ芸術作品をただの絵、ただの物質の塊、ただの音としか捉えることができません。しかし授業などによってただの絵、ただの物質の塊、ただの音にも芸術性を見出すことができるようになります。先程述べた「芸術的なセンスの発揮」にもつながっていきます。

3.芸術とはなにか

さて、「芸術とは何か」ということを説明しようかなと思います。

世の中には色々な芸術作品がありますが、皆さんはどうのようなものを芸術作品と考えていますか？ある人は芸術作品と考えているものであっても、別の人には芸術作品と考えていないかもしれません。このように、世の中に存在するモノの中で芸術作品とみなすかみなさないかは個人次第です。

また、芸術作品というとどのようなイメージがあるでしょうか？あまり日常においては使われない、または身近ではないと考える人も多いと思います。しかし芸術作品の判断基準が曖昧であることから考えれば、役に立つようなものだって芸術作品とカウントされるかもしれないのです。

したがって、僕は芸術とは世の中に存在するモノすべてと考えます。

その芸術の中にも「役に立つ芸術」と「役に立たない芸術」の2つに分かれています。

世の中に存在するモノすべてというのは、音楽などといった形のないものも含まれています。また、役に立つ芸術であるかどうかというのは一般的な分類が存在しているわけではなく、個人によって分かれます。ある人が、これは役に立つと考えているものでも他の人にとっては役に立たないと考えることもよくあるからです。

俗に「芸術」と言われている音楽、美術、書道などは「役に立たない芸術」です。しかし「役に立たない」とことと「存在してもしなくても良い」ということは混同してはいけません。ただ私達が日常生活を送るうえで役に立たないというだけです。詳しくは第5章で説明します。

4.芸術の重要性

いよいよ本題の、芸術の重要性についてです。

まず、この世の中には「芸術作品」と呼ぶことができるものは無限にあり、世の中に存在するすべてのモノから学ぶことができることがあるのです。

芸術は「世の中に存在するモノすべて」である以上、何かしらモノを作る時には常に、今まで取り入れてきた芸術の知識が必要となります。ということは、今まで取り入れてきた芸術の知識と勘すべてが新たな芸術作品の制作に役に立ちます。

世の中に存在する芸術作品にも独創的なものが多くあります。その独創的なものは、何も芸術の知識や勘がない状態から生まれると思われるがちですが、何も知識がない状態では芸術作品を生み出すことさえもできません。独創的な芸術作品は数多ある芸術作品を知ることで、そこから生み出されるのです。

つまり何が言いたいかというと、「芸術」、つまり色々なものを吸収することで日常においてものを独創的な観点から見ることができるようになり、また独創的なものを作り出すことができるようになります。その独創性は、芸術の情報を吸収すれば吸収するだけ大きくなります。

もちろん、役に立たない芸術も含みます。

5.役に立たない芸術の重要性

では、俗に言う「芸術」、つまりは役に立たない芸術の重要性についても説明していきます。

先程述べたとおり、役に立たない芸術はあくまで日常生活において直接的に役に立たないだけであり、これらの存在も世の中において欠かせません。

役に立たない芸術も役に立つ芸術同様、ものを独創的な観点から見るように役に立ちます。しかし、役に立たない芸術の重要性はそれだけではありません。

まず、役に立つ芸術は役に立つが故に、そこから感動するということはあまりないでしょう。なぜなら役に立つ芸術の作品が作られるうえで、芸術的な意図はないからです。

皆さんは絵画を見たり音楽を聞いたりして、心を動かされたことはありますか？役に立たない芸術は、人の心を動かすためにあるのです。

むしろこちらの重要性のほうが納得することができるでしょうか。

もちろん役に立つ芸術も同じくらい人の心を動かしているのですが、それに気づくことはできません。役に立たない芸術は人の心を直接的に動かしています。

しかし、それは役に立たない芸術としての存在意義であって、日常において役に立たない芸術に存在意義があるということにはまだ繋がりません。役に立たない芸術が、役に立つ芸術との間の垣根を超える瞬間が何処かにあるはずです。

僕は先程「役に立つ芸術から直接的に心を動かされる事は少ない」と言いましたが、「見えないところで心が動かされている」、つまり心が動かされていないように思われていても、実は心の

底ではその役に立つ芸術の作品に夢中になっているということはあります。そのようなことは、あるものが役に立たない芸術の要素を取り入れているか次第でありえるかもしれません。以上より、僕はこう考えます。

役に立たない芸術を通して人の心を直接的に動かすことを学び、その知見を利用して、役に立つ芸術作品をつくる上で、役に立たない芸術から得た独創性を活かすことができるというだけでなく、人の心を見えないところで動かすという点においても役立てることができるということです。

これが本当の、役に立たない芸術が役に立つ芸術との垣根を超える瞬間です。

6.物理部と芸術

さて、ここまで全く物理部に関係ないことを書いてきたのですが、これはこの部活である物理部の部誌であり、少しは物理部のことについても書こうと思います。物理部と芸術はどのような関係性について説明します。「そもそも物理部とはなにか」という根本的なところまで説明してしまうかもしれませんね。

物理部はプログラミングや電子工作をしている部活であり、一見芸術に関係ない部活であると思われると思いますが、物理部、いや、部活というもののすべて芸術団体であると僕は考えます。この物理部の場合、「プログラム」「機械」という名の「芸術作品」を創り上げています。

その「プログラム」「機械」にも種類は部員が作ってきた数だけあり、同じものはありません。特に物理部は「モノ」を作る部活であり、つまりははっきりとした「芸術作品」を作る部活であるということです。

物理部員は、先輩が作った芸術作品の仕組みを知り、それに似たものを作ろうとしているがその過程で道がずれ、結果的にオリジナリティのある芸術作品を作ることができるようになると思います。

その中にも「役に立つ芸術」作品、「役に立たない芸術」作品を作っている部員がそれぞれいて、その役に立つか立たないかの垣根を超えてものづくりに励んでいます。

「はじめに」で、「3DCG制作や映像編集、作曲といったこともしている部員がいる」と言及したように、それはつまりたとえ特定のことであっても、芸術の面白さと可能性を見出し、役に立たない芸術に足を踏み入れる部員も増えてきています。

だからこそ、ものづくりの楽しさを知っている部員がいる物理部は、芸術団体の頂点を名乗るのにふさわしいと言えるのではないでしょうか。

7.芸術を学ぶにあたって

さて、芸術を学ぶにあたって、私達は何を意識すればいいのかを考えてみます。この章で説明する芸術は、主に役に立たない芸術を示しています。

そもそも、芸術を教わるがまま学ぶというのも1つの芸術の学習で、それだけでも十分な芸術の知識を得ることができるとはいえ、それだけでは勿体ありません。与えられた少ないものから、その更に上を学ぶこともできるはずです。

例えばこのようなことができるでしょう。

芸術の情報が1つ与えられたら、自分が得た役に立つ芸術または役に立たない芸術の情報を1つ引っ張り出します。次に、与えられた芸術の情報をその情報と対比します。そこから共通点を見つけ出します。共通点があろうとなかろうと、それを数回繰り返します。

もちろんそこから見つけ出したことを何かしらの紙に書き記すなどする必要はありませんし、頭に留めておくくらいで十分だと思います。

このようなことをすることで、得た芸術の情報を独立させず、すべての情報を互いにつなげる事ができ、芸術をひとかたまりにすることができます。そうすることで、第5章や第6章で書いたようなことをすることが容易になるのです。

もちろん今挙げた方法は一例で、情報をひとかたまりにする方法は他にもたくさんあります。その方法を考えるところから始め、芸術を学ぶときに活かしてみるのも良いかもしれません。

8.結局何が言いたいのか

ここまで長ったらしい文章を書いてきたのですが、結局僕が言いたいのはこれだけです。

- ・ 芸術とは、世の中に存在するモノすべてである
- ・ 芸術には役に立つ芸術、役に立たない芸術がある
- ・ 芸術を学ぶことで日常においてものを独創的な視点から見ることができる
- ・ 役に立たない芸術は人の心を動かすことに繋がる
- ・ 物理部は芸術団体の頂点を名乗るのにふさわしいのではないか
- ・ 芸術をただ学ぶだけでは勿体ない

という僕の芸術に関しての独断と偏見でした。改めて言いますが、これは一般的な考えではなく1人の人間の考えです。もちろん反対意見もあるだろうし、この人何を言っているんだと思う方もいらっしゃると思います。あくまで参考程度にしていただければ幸いです。

9.おわりに

という感じでこの記事を書かせていただきましたがいかがでしたでしょうか。

この記事を書いて見て、感想や反省や後悔、驚きなど色々あります。

まずこの記事を書いている僕が一番勉強になりました。物理部の部誌の執筆という名の、芸術と向き合う機会を作れたのはとてもいい経験になったなと思いました。また、中3で文章力もない僕がこのような部誌を(しかも2記事も)書いてしまったのは自分でも驚いています。

文化祭が始まるまで後何週間かのときにもう1記事書くなどといって、(他の部活も兼部しているため)ただでさえ忙しい文化祭の準備の労働量が何倍にもなるとわかっていましたながら、結局は締め切りギリギリまでかかってしまい、編集の先輩方にも迷惑をかけてしまったことを反省していますし、後悔もしています。そんな中でも学校の宿題を遅れなし未提出なしでなんとかやることができたのが一番の自慢でしょう。

僕の、部誌を2記事も書くという奇行を成し遂げることができたのも、部誌の執筆をサポートしてくださった先輩方のおかげです。改めて先輩方に感謝しています。

最後に、僕の拙い文章でも最後までこの記事を読んでくださっている読者の皆様にも感謝を申し上げます。ぜひ他の記事もお楽しみください。

読んでくださりありがとうございました!

理想と現実の界面で

1.はじめに

こんにちは。物理部ポジトロンを手に取っていただきありがとうございます。展示は楽しめていただけましたか。今年の文化祭は、限定的ではありますが、浅野生とその家族、以外の来場者もいらっしゃるということと、自分がこの部活に携わる最後の文化祭になるということが重なり、僕としても、「文化祭」に対する意欲というものが生まれたので、部誌という形でも関わらせていただきました。技術的に興味深いことは他の部員がたくさん書いてくれているはずなので、僕はあえてそういう専門的、ではないことについて書きたいと思います。

僕は中1から物理部に入っておりますが、気が付いてみたらもう高2。これといった大会やら、コンテストやらに全力で取り組んだり、さらなる技術的な高みを目指して毎日努力奮闘する、といったことはなく、作りたいものが見つかったら、活動日にノートパソコンを持っていって地下一階の物理教室でのほほんと作業する、といった感じで、良い意味で緊張感のない活動をしてきました。ですが、決して社交的とは言えない僕でも、それなりには部活のことに参加したり、わずかな貢献もできたのかなあ、と思うばかりです(できてなかったら悲しいですねw)。

小さくてゆるいコミュニティを形成しながら、しばしば友人や先輩後輩とのおしゃべりを交えつつ作業に打ち込める環境はとても快適な場所でございました。そういった居心地の良い場所だったからこそ、あっという間だったのかもしれませんね。しかしまあ、時間がたつのは速くとも、ここで出会ったものはとてつもなく大きいものです。ここでは、そんなこんなを少しばかり話したいと思います。

2.理想

自作ゲームのあるべき姿

今年の物理部は、電子工作班とPC班で同じ教室を使って展示するというスタイルをとっており、例年の文化祭のような電工とPCを分けた展示とはひと味違った雰囲気が出ていることでしょう。そのなかで今年もPC班員たちが作ったゲームが変わらず人気なのでしょうか。

僕は今年、ゲームを自作しようとは全く思いませんでした。中3の文化祭以来、僕はゲームを自作するのを断念したからです。というのも、中2から中3にかけて、僕は寿司打（※フリーのタイピングゲーム）みたいなゲームを作りたいということで、タイピングゲームを自作したのですが、夏休みの中盤くらいになってから、タイピングゲームは子ども受けがよくないゲームだろうか

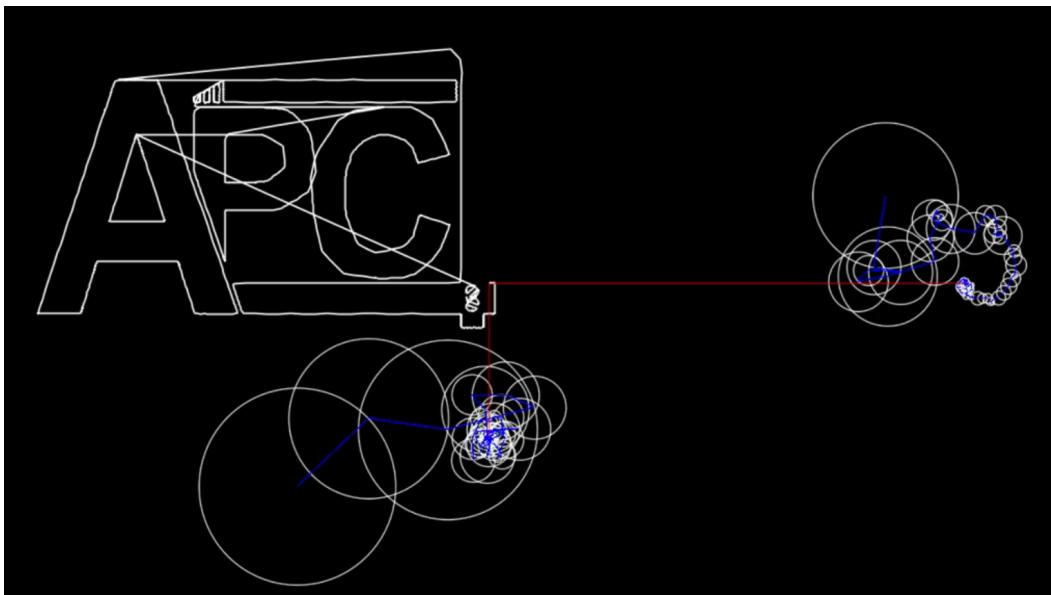
ら、子どもたちが遊べるようなゲームをひとつ作るよう言われ、僕はとても困惑したことがあるのです。結局、それから文化祭までの短い時間になんとかもう1つのゲームを間に合わせたのですが、それ以来、遊んでもらうために、「文化祭」のためにゲームを自作する、ということに対しているいろいろ考え、疑問を抱き始めました。というのもどうも僕は「文化祭」のために、子どもたちに人気ではあるけれど、自分では作りたいとも思わないゲームを作らされるのを苦痛に感じ、どうしてもそれを強制されることに耐えられなかったようです。

中1中2がプログラミングに慣れる練習として、とりあえずゲームを自作するという考えに関しては、まだ的を射ていると思います。僕も今振り返ってみると、実際にある程度の量のコードを書いて一つのゲームを完成させたあとには、なんとなくプログラムの全体を見渡す能力が少しついたのかなあ、と思わなくもないです。しかし、中3、高1、高2にまでなって作りたくもないゲームを作らされる、作らなければいけないような雰囲気になるのはもったいない気がするんですね。パソコンでできる技術的な物と言えば自作ゲームのプログラムだけでなく、様々な種類があるにもかかわらず。例えば、何か特定のことをしてくれるプログラムの制作や、人工知能の開発、広く言えば動画編集や競技プログラミングなども含まれているでしょう。ひとそれぞれ興味のあることはもちろん違うでしょうし、彼らの興味に割かれるべき時間が、「文化祭」のためのゲーム制作によって榨取されていくなんてことがあっていいのだろうかと、疑わざるを得ません。「そもそもゲームを作るのが好きだ」という人にとっては何ともないでしょうが、それ以外の人にとっては厄介極まりないことでしょう。その時以降、「文化祭」のために強制的にゲームを自作させられるというようなことがなかったので良かったのですが、今後とも、このような犠牲者が出ないように気を付けていただきたいものですね。

文句ばっかり書き連ねていたら読み心地もよろしくないと思いますし、決して改善に向かわないでの、とりあえずほどほどにしておきますね。

フーリエ級数展開によるお絵描き

僕が今年作ったのは、もちろん子どもたちに遊ばせることのできるようなゲームではなく、ただ、時間についての関数を実フーリエ級数展開して三角関数(角度についての関数)の式に変換した後に、円を回転させて図形を描くというプログラムです。



ふーりえAPC

フーリエ級数展開を使って図形を描画するという試みは、世界中でたくさんの人が行ってきているので、インターネットで調べていただければ概要はわかると思います。僕は、おととしに、当時高校2年生だった先輩が作ったプログラムをみせてもらってこれを初めて知りました。円をくるくるまわしながら描いていくのこぎり波だったり矩形波、「なんだかよくわからないけどすごいな」という感じで見ていました。その光景はとても複雑に見えましたが、実際は、大きさの異なる円を回すことで図形を描くという、意外と単純(?)な方法がありました。その後先輩からサインやらコサインやら(←全然わからなくてほとんど覚えていない...)説明を賜りましたが、中3だった僕は残念ながら理解するには到底及びませんでした。

しかしながら、そこで見たものが頭の中に焼き付いていたのか、高校二年生になって数学B(文系)の授業で三角関数を復習したときに、そういえば2年前にフーリエなんちゃらしてた先輩いたなあ、となり、高校二年生になった僕ならできるかもしれないと思いながら、制作に取り掛かりました。覚えていたことは、その先輩がフーリエうんぬんでおえかきをしていたということぐらいだったので、そもそもの数式やそれらの導き方をネットで調べるところから始めなければなりませんでしたが、周りの人からの協力を受けて、楽しみながら完成させることができました。これはsvg形式の画像ファイルを読み込ませると、その画像を一筆書きしてくれるので、完成した後は、直ちに好きな画像を突っ込み、喜びにふけっていました。

こういう風に、僕の「作りたい！」という単なる欲求を満たすために作ったものを「作品」として展示しているだけに過ぎないのですが、そうはいってもやはり、僕にそのように、作りたいという「情熱」を思い起こさせるだけの魅力が存在するわけで、その情熱を少しでも感じ取ってい

ただきたいという所存でございます。2年も前に見たのが記憶の片隅に残っていたというのは、僕がその妙な動きに多少なりとも惹かれていたからでしょう。好奇心とはすこし違うようなこの妙な心の揺れ動き、皆さんは感じたことはないでしょうか。

大きな「情熱」との出会い

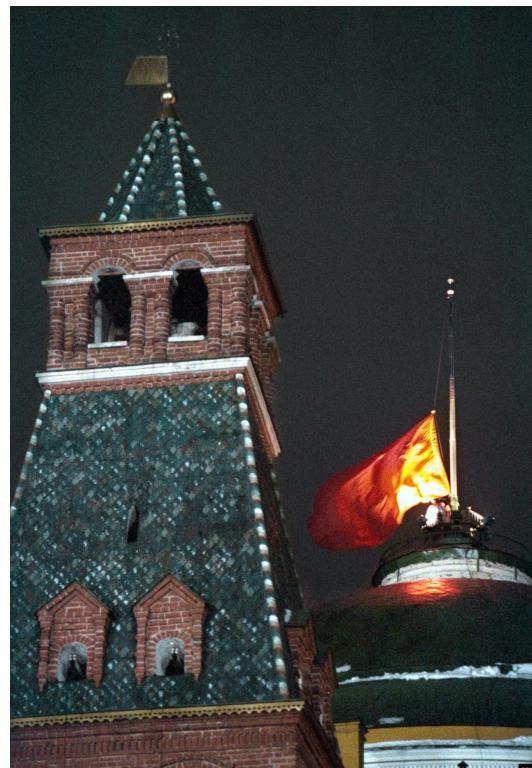
中学1年生の時に物理教室で、僕は、今でも自分に絶えることなく「情熱」を与え続けるモノと偶然にも出会いました。そしてそのモノというのがなんと、ソビエト社会主义共和国連邦国歌なのです。皆さんのがこの告白についてどう思おうとも、構いません。しかし、僕のソ連国歌に対する「情熱」は疑いようのない事実です。僕は非常に困ったことに、いわゆるJ-POPでも洋楽でもなく、ソ連国歌を好きになってしまったのです(最近はみんなが聞くようなものも少しあるようになりましたが)。僕はこの歌を初めて聴いたとき、もちろん流れてくる歌詞の意味は全く知りませんでしたが、「感銘」という言葉で表しきれないようなナニカが僕を満たしていくような、そんな不思議な衝動を感じました。

それからというもの僕は、ソ連国歌の歌詞についていろいろ調べたり、ソ連に関連した情報を集めたりするにつれて、近現代の歴史的な流れや思想や知識を得ていったりと、ソ連の国歌が原動力になって、いろいろな物に首を突っ込んでいくことになりました。そしてソ連国歌とソ連の関係を知れば知るほど、賛成できない点も複数ある中、ソ連に対する魅力はそれらをはるかに上回っていました。今持っている音楽に対する興味も、この国歌がなければ持たなかつたかもしれません。どうして僕がこんなにソ連に魅了されているのか、その謎は僕でさえもいまだ探求中、整理中でありますが、なんとかしてソビエト連邦を僕の生活におけるコンテクストに位置づけ、ある程度言語化したいという個人的な思いで、「情熱」の一例として紹介させていただきます。

さて、当初僕はというと、ソビエト連邦に一種の理想像、理想の共同体像を重ね掛けていたことがあり、今となっては少々過激だったかなあと想い返すのですが、このことがなければそんなにソ連のことを好きにならなかったと思われます。ソビエト連邦は、すごくざっくりいうと、指導者層の荒廃、政治的、外交的な欠陥などにより、1991年12月25日の真夜中にとうとうソ連の最初で最後の大統領ゴルバチョフが自身の大統領辞任を表明し、解体するという形で、第一次世界大戦期のロシア革命やそれに続く内戦を経て1922年に高い理想を掲げて成立したその国は、その69年にもわたる激動の歴史に終止符を打ちました。僕は当時生まれておりませんが、とても衝撃的なことだっただろうことは容易に想像できます。

ここで僕が注目したいのは、その時に立ち会わせていた当時のソ連の国民です。人々がソビエト連邦という国に対する帰属意識を強く持っていたかどうかは定かではないですが、生まれ育った祖国が、かつては世界を席巻した超大国から徐々に衰退の一途をたどっていき、しまいには国が瓦解していくのを目の当たりにしたソビエトの国民はいったい何を思ったのでしょうか。当時の写真や動画に映る人々は祖国の衰退に対して逆らおうと奮闘していたけれど、無力にもそれは成功しませんでした。彼らがソ連崩壊の時に受けたショックというのはいったいどれほどのものだ

ったのでしょうか。壮大な曲調でもあるけれど、どこか寂しげな感じもするその国歌によって、僕はこういった無力感に思いを馳せ、同情せざるを得ませんでした。そして、とあることで思い悩み、よりどころを求めた僕は、最近かつ最大の「共同体」の崩壊を自分のにおけるものと重ね、親近感を伴って想起したのでした。なんだか、歴史が、僕なんかよりもはるかに大きいショックを受けただろうソビエト連邦の国民一人一人が、僕の側に立って、応援し、励ましてくれているようなそのような妄念を感じてしまうのです。



ソ連国旗降納

かつてのソビエト連邦が、望ましい共同体であったかどうかは疑わしいですが、それでも、その応援してくれているような感覚は僕の辛い気持ちをいつも癒してくれました。それ以来ソ連国歌は基本の原動力として、僕に大きな影響を与えています。しかし、自分の考えが COMMUNISM というよりむしろ COMMUNITARIANISM のほうに近そうですし、決して COMMUNISM を称賛しているわけではないことは了承していただきたいです。

個々の「情熱」を咀嚼する

物理部にいると、様々な人からこういった「情熱」を感じ取ることができます(もちろんですが、ソ連国歌ではございません)。部員のそれぞれが、僕に、彼らの趣味だったり、興味のある分野の話や、取り組んでいることについての話をしてくれるのですが、彼らが話しているときのその表情や口調、それから身振り手振りなどから、意図せずとも「情熱」がにじみ出てくるんですね。そして、その「情熱」の形というのは十人十色で、それぞれの人の持っている性格、視点、価値観、奥深さ等等を本当によく表現していることでしょう。また、同じ人によっても、時と場合によってその「情熱」がポジティブになったりネガティブになったりすることもありますし、そういう点でも、バラエティーに富んでいるといえるでしょう。自分にとってそれらは未知なものばかりであるので、それらを見たり、触ったり、聴いたりしてようやくそのわずかを味わうに至ります。ですが、そのわずかな経験を通して、「那人」の人となりをより深く知ることになるでしょうし、あるいは、それによって自分が新しく触発されるかもしれません。他人の「情熱」や、その「情熱」から経験した事柄のような、未知との遭遇の可能性にありふれている場というのが、この物理部であり、もっと広く言えばこの浅野の文化祭なんじゃないかなと個人的には思っております。



物理部風景

こんなにも時間のかかる「作品」、しかも、作らなくても生活に支障をきたさないだろう「作品」を莫大な労力をかけてわざわざ作り上げるというある種の狂気を持つためには、間違いなく、とてつもなく大きい原動力が必要でしょう。そして、その原動力がどこから来ているのか、僕たちは、それらの「作品」の制作者でありますから、もちろんその「作品」に対する説明や思い入れを、展示に来てもらった皆さんに紹介すると思います。が、その説明を受けただけで終わらせてほしくないです。なぜかというと、僕たちの、「作品」に対する説明というのはあくまでも僕たち自身の一元的な見方による説明でしかないからです。その説明を受けるだけでは、必然的にその「作品」のもつ可能性は限定されてしまうでしょう。僕ら作品制作者の「情熱」と、「鑑賞者」の感じたものの交わる点、その二つを折衷したところにこそ、無限の可能性が秘められているのではないでしょうか。作る側→見る側という一方的な主体客体の関係を超えて、「鑑賞者」自らが主体意識を持って「作品」と触れ合い、その界面で二つが融合することによって、唯一無二性が生じるのではないかでしょうか。そしてそれは文字通りかけがえのないものとなるのではないでしょうか。ぜひとも、この文化祭で何かしらそういうものを得てもらえたならあと願うばかりです。

3.現実

僕はいまこうやって自分の文化祭の理想像を語らせてもらっているわけなんですけども、なかなか簡単にそういう形にできるわけでもありません。すなわち、自由に、自分の興味のあることをして、事が上手く運ぶわけではないということです。その大きな理由の一つは、文化祭での集客効果という観点から、そちらへ方向転換しづらいということにあります。やはり、文化祭において、部員たちの自作ゲームは子どもたちにとても人気で、PC班の集客に大きな貢献をしてきているのはこれまでの4年間にこの目で見てきました。かつて、ライフゲームを展示していました。(今年もだと思いますが、)壁新聞も一応展示していましたが、自作ゲームを展示している机のほうより人が集中するということなどほとんどなく、もしもゲームの展示がなくなったらPC班の展示はおそらく閑静とした場になってしまふことでしょう。たとえその物寂しさに耐えられたとしても、やがてお金の面で、持ちこたえられなくなるはずです。

物理部は他の部活と同様に、部費という形で学校から援助を受けて成り立っており、あくまで浅野の経済力ありきでの物理部なのです。そしてその部費は、学校に対する貢献や、部活の実績等の要素で上下する可能性が高いのです(実際に部費を大幅に減らされたことはないので完全なる真実とは言い難いです)。部員からの部費を一切徴収していない上、部員数が多く、このお金が少ないと、パソコンが足りなくなったり、電工の備品を購入できなかったり、いろいろと不自由が生まれてしまいます。ゆえに、文化祭で、AsanoTheBestなるものでなるべく上位に入り学校への貢献を可視化し、なるべく多くの部費をいただかないと物理部そのものが崩壊しかねません。で

すが、自作ゲームの力を借りずに文化祭での人気を保ち続けるのは考えづらいですし、部費が少ない状態で部活動を続けていくということも困難でしょう。すなわち、この成果主義的なシステムから逃れる方法は今の状態では到底考えられないのが現実だということです。

ここまで言っておいて何も手だてがないのか、とお思いになるかもしれません、まあ、実際現時点で生徒にできることはほとんどないでしょう。このような場で、あまり理想をかましすぎても結局空回りして終わりという、歴史の二の舞三の舞を演じてしてしまうだけなので、ここはある程度譲歩するのが好手と言えるでしょう。しかし、完全に理想を諦めてはいけません。理想と現実の狭間で、理想を捨てずに、現実に抗い続け、止揚した先にある自由の可能性を信じて奮闘するのです。理想から出発した「情熱」を現実とうまく融け合わせ、このシステムの中で運用可能な形にしていくことこそが、最善の解決方法ではないでしょうか。

4.まとめ

時と場合によって、自身の心を揺さぶったものが受け入れられないことがあります。例えば、僕が先ほど例に挙げた「情熱」のことを他人に話したとき、はねつけられるということもまあ何回かありました。やはり現実は、自己の理解をすり抜け、自己の思い通りにならず、無限に連鎖する否定性を持っていると僕は考えます。ゆえに、「現実」というのは往々にして、「理想」そして「情熱」をも否定してくるものです。そこで僕たちはこのアンチテーゼとしての「現実」に屈するべきではないでしょう。自身のもつ「情熱」を、そしてそれが存在するという搖るぎない事実を強く根拠にもって、新たな「総合」を生み出していくべきではないでしょうか。おそらくその作業には少なからぬ苦痛や困難が立ちはだかっていると思います。ただ、その先には、自身と他人が融合した、ユニークなものが待っているはずです。ぜひそれを求めていただきたいですし、また、自分もそれを探求していきたいと思っております。きっと、歴史上で同じように奮闘してきた人々も寄り添ってくれていることでしょう。

5.おわりに＋スペシャルサンクス

最後まで読んでいただき本当にありがとうございます。内容のほうはいかがでしたか。物理部のことについて多少触れながら、僕の個人的な考えをエッセイ風に書くのは僕にとってあまりにも難しうぎで、これを書きながら何度も、無謀な挑戦をしたもんだなあ、と思いました。それぞれで、言いたいことが似通っている部分もあるはずなのですが、僕が思うに、なんだか趣旨が若干曖昧で、一貫性のない文章になってしまいました。ともあれ、物理部の文化祭展示に来ていただいて、かしこくもこの部誌をとっていただいた方々に、これを読んで何かご自身の中で少しでも変わったことがあれば、僕は非常に嬉しい限りです。

ちなみにこの部誌は「自分自身」を一番の対象読者として想定しながら書きました。上にも少々書きましたが、僕のソ連に対するこの「情熱」を渾沌としたものから少しでも秩序づけたくて書いたというのが大きいです。なので、内容が個人的すぎたり、わかりにくかったり(わざと濁している箇所もございますが)、論理的根拠に欠ける点があったと思いますがお許しください。

最後に、多くの時間を一緒に過ごした物理部員のみなさん、本当にありがとうございました。特に、同学年の方々には、特別に感謝し申し上げます。そして、これからもぜひよろしくお願ひします。

6. 参考文献

Wikipedia-エマニュエル・レビィナス <https://ja.wikipedia.org/wiki/エマニュエル・レビィナス> 10月6日アクセス

ソ連国旗降納の画像元 <https://www.rferl.org/a/1830507.html> 10月6日アクセス

コイルガン 四年間のすべて

お久しぶりです。そうでない方もお久しぶりです。高二になってしまった部長です。前回部誌を寄稿してから早二年、今でもバカの一つ覚えみたいにコイルガンを作り続けているわけですが、その間いろいろありましたね。「いろいろ」の中身はあえて触れませんが。

1.ただのコイルガン？

コイルガン…なんて漠然と言ったところで、人によって色々なやり方があります。電源、昇圧回路、コンデンサ、スイッチング素子、コイルの段数、コイルの巻き。どんな理由で何を選んだか、そこに焦点を当てて解説します。

1-1.電源

コイルガンはEML(Electro Magnetic Launcher)の一種で、その名の通り電気を使って金属製の弾を発射するものです。電気を使うということは当然電源が必要で、その電源をどうするかも人によって様々です。

一番簡単(に思える)のは、コンセントのAC100V電源をブリッジダイオードで整流して、コンデンサで平滑した後にそのままコイルに流す方法でしょうか。ただこれはコンセントからコンデンサの間にトランスを挟まないといけないという欠点があります。(なぜかはググってください)いや、少し言い方が雑でした。実を言うとACアダプタにもトランスは入っていますし、トランスを挟むことそのものが欠点になるわけではないのです。そう、**50~60Hz用の大きくて重いトランスを挟むことが欠点なのです。**今のACアダプタはスイッチング回路を使って相当小さく、軽くなっています。半導体技術の賜物ですね。このスイッチング回路を一から自作するのは大変ですし、コイルガンの機能の肝になるわけではないので市販のACアダプタで済ませることにしました。また電池駆動も考えたのですが、乾電池は一瞬で無くなってしまってランニングコストが相当高く付きそうなのと、eneloopなどの乾電池型の充電池も容量が少ないので交換の手間がありますし、かといってニッケル水素電池やリチウムイオン電池を組み込んで、充電制御回路まで作るのも面倒です。幸い僕が製作で使う物理教室にはあちこちにコンセントがありますし、文化祭の展示の時はコイルガン用に電源タップを用意すればいいですからね。

さてACアダプタと一口に言っても電圧、電流、電源容量と色々な種類のものがあります。これをおざなりにすると痛い目に遭います。何て言ったって中2の文化祭で充電が異様に遅かったのはそのせいでしたから。例えば電源容量(平たく言うとW数です)の小さいものを選ぶと、電力量は電圧と電流の積ですから、高電圧を扱うコイルガンでは引き出せる電流量が極端に少なくなってしまいます。

まいます。5V2AのACアダプタで100Vのコンデンサを充電する、ぐらいなら待ち時間も気にならないぐらいだとは思いますが、今回のコイルガンは400V2720μF(後述)を10秒ぐらいで充電したいので、あんまり電源容量の少ないやつでは困ります。売ってる中で一番大きいやつがいいですね。

電源容量が大きければ何でもいいわけではありません。極端な話0.1V500Aだったりすると困るわけです。例えば半導体の耐電流は電圧に関わらず絶対的な電流で決まります。今回はなるべく大きくして電流を小さくする方針で行きます。仮に電圧が高すぎるなら抵抗で分圧すればいいのに比べて(電流消費が小さい場合に限りますが)、低すぎるから昇圧するのは色々面倒ですし。結果電圧は15V前後にすることにしました。これは昇圧回路で使うNE555(後述)というICの耐圧が16Vだからです。16VのACアダプタもあるのですが、電源電圧でギリギリを攻めるのはさすがに怖すぎます。

また取り回しも重要です。先述の通り物理教室はあちこちにコンセントがありますが、かといってコンセントから全く動けないのは困ります。ですからできればPC用のACアダプタぐらいの大きさで、ATX電源のような大きいものは使わないことにしました。

この「なるべく電源容量の大きなもの、でもPC用のACアダプタぐらいの大きさで」という条件を元に、秋葉原の秋月電子で探してみたところ、15V 3.34Aがちょうどよさそうですね。何でよりもよって3.34Aなのかは知りませんが。

(<https://akizukidenshi.com/catalog/g/gM-08432/>) これで電源については解決です。

1-2.昇圧チョッパ

コイルガンの肝となる部分の一つが昇圧チョッパです。「高速充電」や「連射」など、使いやすいコイルガンを作るにはこの改良が不可欠です。

これだけ言っておきながら、昇圧チョッパに関してはほとんど改良点がありません。本当はネットで色々調べたり、どこを改良するか構想も描いていたのですが、何しろ時間がなくて...。基本も大事だということでお許しください。

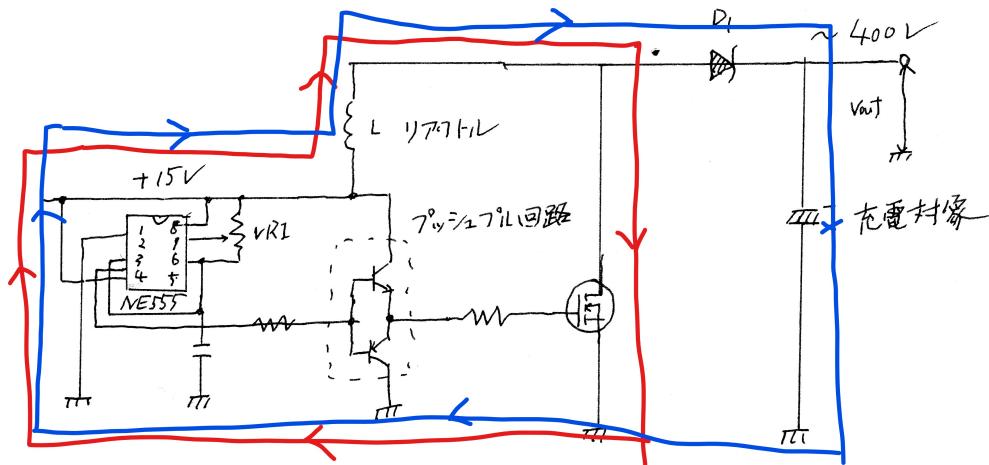
さて、一番簡単にやろうとするなら使い捨てカメラの(ストロボの)昇圧回路にコンデンサを繋げるのがいいのでしょうか。確かに簡単ではあるのですが、当然欠点も多くあります。

まず一つは性能が低いことです。ストロボの消費電力はコイルガンと比べてかなり少い(はず)なので、回路もかなり簡素です。これでは充電に相当時間が掛かるでしょう。

二つ目、これが一番大きいのですが、僕にわざわざ写真店に行って「回収した使い捨てカメラ、いくつか頂けませんか」なんて言う度胸がないことです。「コイルガンに...」なんて言ったところで相手にとっては意味不明ですし、そこまでして性能の低いものを集めに行く必要はありません。

また昇圧チョッパ単体で市販品があればいいのですが、コイルガン用なんてものは当然なく、そもそも昇圧チョッパ自体がマイナーなんです。降圧はPCやスマホ用にICもたくさんあるのですが...。 ICレベルなら昇圧チョッパもあるのでしょうか。だからといって「400Vまで耐えられて、数A出せる」ものは調達も難しいでしょう。ここに関しては使い捨てカメラを諦めるなら自作しかありません。

自作するにはまず昇圧チョッパの原理を理解しないといけません。昇圧チョッパは次のような回路構成になっています。



スイッチON時

まずスイッチQ1がONのとき、電流は赤線の経路で流れます。このとき、コイルに電流が流れるエネルギーが次第に蓄えられていきます。電流I[A]が流れるリアクトルL[H]に蓄えられるエネルギーP[J]は次式で示されます。

$$W = \frac{1}{2} LI^2$$

スイッチOFF時

ここでスイッチQ1をOFFにしてみましょう。スイッチQ1がOFFのとき、電流は青線の経路で流れます。このとき、Q1がON状態のうちにインダクタに貯まったエネルギーが矢印の通りにコンデンサに蓄えられます。コンデンサの両端に電流が流れればエネルギーもその分蓄えられ、両端の電圧も上昇していきます。また出力電圧(=コンデンサの電圧) V_{out} が入力電圧 V_{in} より大きくなつたとしても、ダイオードD1の働きで V_{out} 側から電流が流れ出ることはないので、負荷抵抗を繋がない限りコンデンサに蓄えられたエネルギーは保持されます。

こうしてコイルガンに必要な数百V、今回は400Vの電圧が得られるわけです。ちなみに、今回は主に執筆時間の関係で説明が相当雑ですが、数式を交えた詳細な解説はネット上にたくさんあるので、気になった方はぜひ調べてみてください。

「スイッチ」を「切り替える」

さて、問題はこの先にあります。先ほどの解説ではスイッチを「理想的な」、つまり電圧電流の制限もなく、ON時の電気抵抗はゼロで、一瞬でON-OFFの切り替えができる、なおかつ制御が不要なものとして扱っていました。現実的にそんなものがあるかどうかなど言うまでもないでしょう。ありません。そこでこの「理想的な」スイッチの代わりに何か使えそうなものを用意しないといけません。

「スイッチ」と漠然というと多くの方はこの「トグルスイッチ」を思い浮かべるでしょう。確かにこれもスイッチの役割を果たせますし、扱う電流が小さければ全く問題はありません。射撃用のトリガーなどはこれを使っていますし。

ただし、昇圧チョッパの「スイッチ」となると話は違います。まずは電圧です。回路構成からもわかるとおり、このスイッチには出力電圧がそのまま掛かります。今回は400Vですね。一応スイッチそのものの耐圧の条件は満たせるはずですが、問題はそこではありません。

出力電圧がそのまま掛かって、なおかつスイッチON時の電流を見てみると、このとき電源-インダクタ-スイッチで閉回路ができるわけですが、この回路かなり大きな電流が流れます。具体的には…これはインダクタの定数なども考慮して色々計算してやる必要があるのですが、結局のところACアダプタの最大電流程度になりそうです。勘ですけど。

また、「スイッチ」を「切り替える」速度も問題です。先ほど「スイッチON時インダクタにエネルギーが貯まり、スイッチOFF時それが放出される」ことを述べましたが、これはすなわち「インダクタにどれほどエネルギーが貯められればいいか」はスイッチがONの時間、すなわちスイッチのON-OFFが切り替わる周期に依存することになります。

「インダクタにどれほどエネルギーが貯められればいいか」というのはインダクタの定格電流のことです。当然ながら定格電流の大きなインダクタは大きく、重くなります。なるべく回路は小型化したいので、インダクタも例外ではありません。スイッチング周期を短くしてインダクタを小型化しましょう。

(今後は数値として扱いやすい一秒間の振動数、つまり周波数 $f[\text{Hz}]$ で解説していきます。周波数 f は次式で求められます。)

$$f[\text{Hz}] = \frac{1}{T[\text{s}]}$$

さて、スイッチング周期を短く、つまり周波数を高くするにあたって大きな制約が立ちはだかります。それは「スイッチ」です。具体的に今回はスイッチング周期を10kHz程度にしたいのですが、周期にすると $\frac{1}{10000}$ 秒です。こんな早さでトグルスイッチのON-OFFを切り替えられる人はいるのでしょうか。もし我こそは!という方がいらっしゃいましたらお近くの物理部員までお声かけください。詳しくお話を伺います。

それはさておき、トグルスイッチではない、少なくとも人間が $\frac{1}{10000}$ 秒の周期で切り替えずに済むようなスイッチを選びたいところです。

どんな「スイッチ」?

答えから言ってしまうと、この用途にはトランジスタなどの半導体スイッチが最適です。

半導体スイッチはモノによっては数百V、数十Aぐらいなら軽く耐えられますし、ON-OFFの切り替えにあたって機械接点に由来するアークなども出ません。そして何よりも、電子回路によってON-OFFの制御ができるのです。これが最大の利点でしょう。

半導体スイッチにもいくつか種類があり、代表的なものはバイポーラトランジスタ、接合型電界効果トランジスタ(J-FET)、金属酸化膜半導体電界効果トランジスタ(MOSFET)、絶縁ゲートバイポーラトランジスタ(IGBT)、サイリスタ、トライアック(サイリスタの亜種)などがあります。このうちサイリスタやトライアックは今回の用途には使えないほか、J-FETは大電力用途のものがあまりないので、必然的に選択肢は残りの三つに絞られます。

先ほど挙げた、いわば「トランジスタ系」の三つを比較するうえで重要なのは「耐電圧」「ON抵抗」「ゲート電圧(MOSFET, IGBTのみ)」「ゲート容量(MOSFET, IGBTのみ)」の四つです。耐電圧は読んで字のごとく素子が耐えられる最大電圧、ON抵抗とは素子がON状態の時の電気抵抗で、これは電力損失や発熱に関わります。またゲート電圧は...平たくいえばONにするために必要な電圧です。ここではそれだけで十分です。最後にゲート容量。これもここでは「素子のONしにくさ」を表すものだと考えてください。これらのトランジスタは一長一短で、ありきたりな言い方ですが用途に合ったものを選ぶのが大切です。

これを踏まえた上で各素子の特徴を以下の表に示します。

素子	耐電圧	ON抵抗	ゲート電圧	ゲート容量	個人的な扱いやすさ
BJT	~300V	高め	0.6V	考慮せず	◎
MOSFET	~600V	低め	5V前後	小さめ	○
IGBT	~1000V	中程度?	15V前後	大きめ	△

このうち、今回の出力電圧は400Vを予定しているのでBJTは選択肢から外れます。またON抵抗も大きいのでその分熱問題も厄介です。

そうなるとMOSFETとIGBTの二択となるのですが、今回はMOSFETで困る用途ではないのでMOSFETを選択しました。当然IGBTでも問題はありません。

...というのは嘘です。

何を隠そう、最初はMOSFETを使うつもりだったのですが、実際に製作するときにIGBTを間違えて実装してしまいました。MOSFETとIGBT、役割も性質もそれなりに似ているが故の間違いでした。

ただ回路自体は問題なく動作したのでそのままにすることにしました。もちろん全ての回路で互換性があることを保証しているわけではありませんよ。今回たまたま動いただけです。

ゲート駆動

先ほど「半導体スイッチは電子回路でON-OFFが制御できる」と述べました。逆に言えば電子回路なしでは何もできません。手で動かしたりももちろんできません。そこで必要になるのが発振回路です。

今回はNE555でPWM信号を生成して、それをプッシュプル回路で電流増幅したのちゲートを駆動することにしました。意味不明ですね。もう少し詳しく解説します。

PWMとは？

あまり長々と説明しても話が逸れてしまうだけなので、ざっくりと言うと「昇圧ショッパを駆動するには、PWMがいい」のです。そのPWMもざっくりと言うと「周波数は一定のまま、ONとOFFの比率を調整する」ものです。PWMはトランジスタや抵抗などから、一から作ることもできますし、オペアンプなども選択肢の一つではありますが今回はNE555という有名なICを使うことにしました。NE555でPWMを生成する回路は下図の通りです。（回路図）NE555の内部回路の説明はNE555のデータシートに譲りますが、外部については少し解説します。

図中の6,7番ピンに接続されている可変抵抗と2番ピンに接続されている(フィルム)コンデンサで発振周波数を決定します。以下に発振周波数との関係を示します。(データシートから引用)

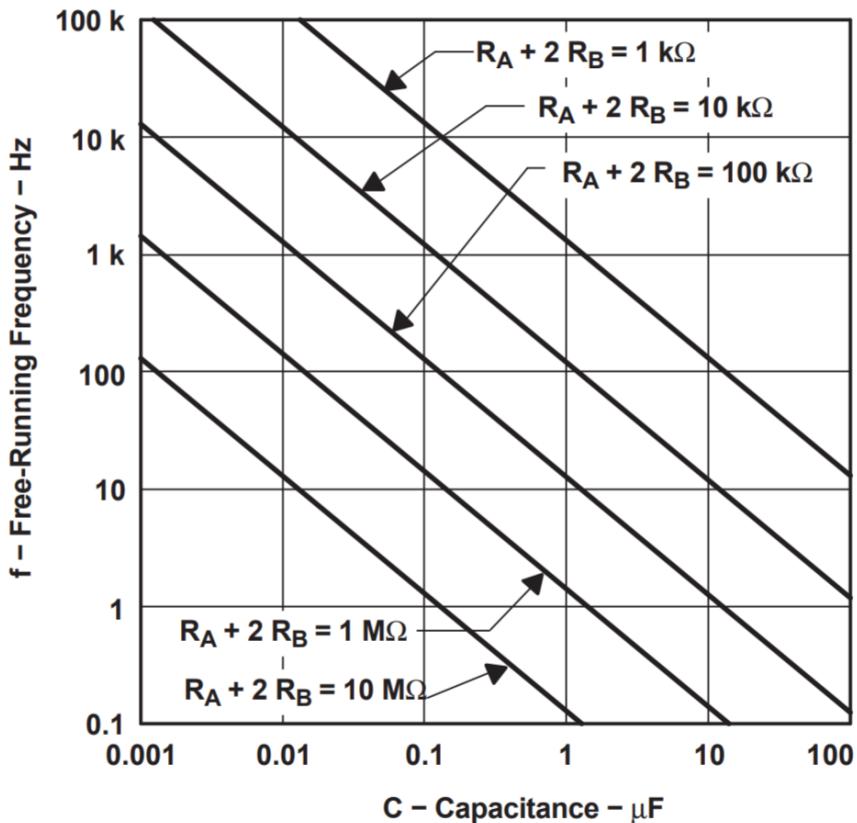
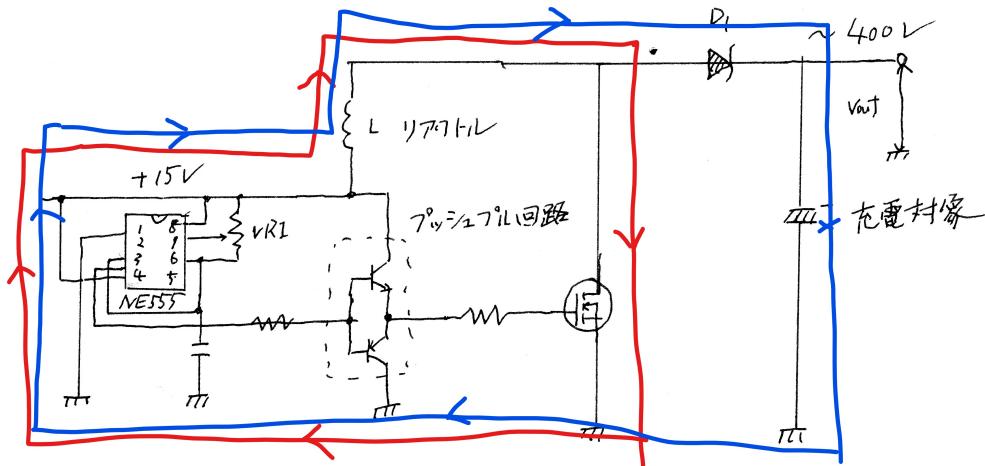


Figure 14. Free-Running Frequency

発振周波数は今回可聴域の範囲で決定しました。発振周波数が可聴域の範囲にあると、ゲート駆動回路で半導体スイッチを駆動して、実際にインダクタに流れる電流が変化すると発振周波数で音が出ます。いや、どの周波数でも音は出ているのですが、それが聞こえるのが可聴域だけなのです。(定義からして当たり前なのですが...)

よってこうすることで「コイルから音が出ていれば、昇圧回路が正常に動いている」ということがわかるようになります。

プッシュプル回路



プッシュプル回路は上の回路図の点線の部分です。

NE555は実際に見てみるとわかるのですが、かなり小ぶりなICです。このICで大電力用のMOSFETを駆動するには少し不安があります。何となくの予想ですが。

そこでこのプッシュプル回路の出番です。プッシュプル回路はpnpトランジスタとnpnトランジスタをそれぞれ電源側とGND側に、出力にエミッタが繋がるように配置します。言葉で説明しても意味不明なので、昇圧回路全体の回路図を参照してください。

プッシュプル回路の(電圧)増幅率は**1**で、電流の吸い込みも吐き出しもインピーダンスの低いトランジスタのエミッタ側なので結果的に電流増幅になって…

自分で説明しても眠くなってきたので、細かい説明は僕の2019年の部誌「基礎からはじめるトランジスタ」に少しだけあります。(宣伝)

(<https://asanobuturi.github.io/document/2019/transistor/index.html>)

とにかく「電流を供給する能力を上げる」とだけ理解しておいてください。

これで昇圧回路の解説は終わりです。

2. 単段 or not 単段

単段だのnot 単段だの、一体何の話でしょう？ とその前に、なぜ「単段」ではいけないのかについてお話しします。

コイルガンは、その動作原理からもわかるように電力を継続的に、平均的に消費し続けるのではなく、一瞬のうちに大電力を消費し、消費しきった後は一切何も起こらないというよくある電子機器とは全く違う動作をします。(当然外部から見れば昇圧回路で電力を継続的に消費しているのですが、ここではあくまで内部のコンデンサの電荷の消費の話です)

当然威力を増そうとするならコイルを太く、巻き数を多く、コンデンサ電圧を高く、弾を大きく、など何もかも重厚長大にせざるを得ません。本当に「何もかも」なのです。そして部品の価格や扱いやすさは必ずしも性能に比例しません。いや、逆にきれいに比例することなんてほとんどありません。ならば最も価格と性能のバランスがとれた部品、つまりコスパの良い部品を使うのが最も賢いやり方でしょう。

コイルガンの威力を増すのには、部品単体の性能を上げる他にもう一つ方法があります。部品を複数使うのです。そのために生まれた工夫こそが多段式なのです。

多段式コイルガンは、その名の通り複数のコイルを用意して弾が通過した順番に一段目から電流を流していくものです。この多段式コイルガンは単段式より比較的簡単に威力と効率を上げることができます。

多段式の射撃ではコイルへ通電するタイミングが最も重要になります。一段目は自分の裁量で打つか打たないか決めるとして、二段目以降はどのタイミングで電流を流せばいいのでしょうか。もちろん感覚でやるわけではありません。これこそが次の射撃制御です。

3.射撃制御

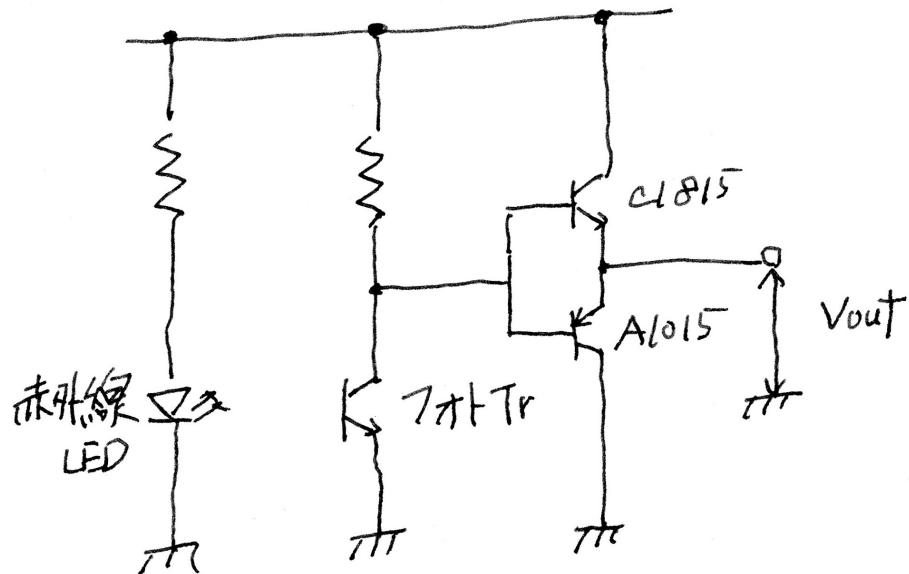
先ほど述べた通り、多段式コイルガンに付きものの問題は「射撃タイミングをどうとるか」です。これに関しては僕の知る限り二通りの方法があります。

一つはマイコンで一段目のコイルに電流を流してから、一定の間隔をおいて二段目以降にも電流を流す方法です。

僕はマイコンに関する知識が皆無で、なおかつ次の方程式がマイコンよりも確実だと思ったので、この方法は採用しませんでした。もう一つは赤外線センサを使って弾の位置を検知し、二段目以降のコイルに弾が差し掛かったところで通電する方法です。

赤外線センサというのは、つまり赤外線を使って弾などを検知すればいいわけですが、これは赤外線LEDと赤外線用のフォトトランジスタを使って実現しています。赤外線LEDは通電すると赤外線を発します。そしてフォトトランジスタは光を感知すると電流を流す部品です。トランジスタ、というのはトランジスタのゲートが光で反応するようになった振る舞いをするからです。

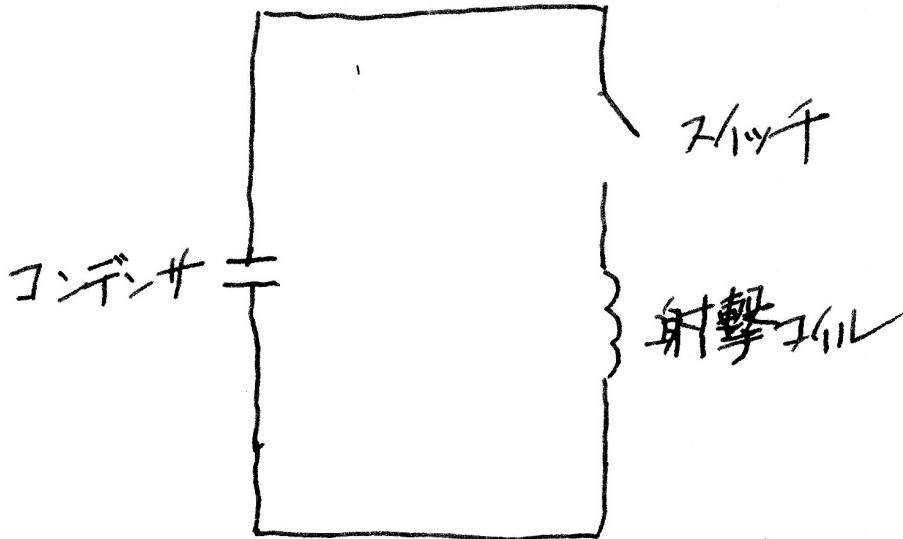
検知回路の回路図を下に示します。



この検知回路の出力をそのまま主回路のIGBTに繋げるわけではありません。ゲートドライバ回路が必要になります。ゲートドライバ回路の解説の前にまずは射撃回路の説明から。(名前紛らわしいですね)

4.回生...というより

射撃回路もコイルガンの肝になる部分の一つです。射撃回路の半導体スイッチをONにして、コンデンサとコイルを接続して通電します。ここで、一番単純な回路を下に示します。



これにも問題点がいくつかあります。

まず、スイッチをOFFにした時にコイルに流れている電流の逃げ場がないことです。コイルは通電するとエネルギーを蓄えますが、逆に言うと電源から電流の供給がなくなってしまってもしばらくは蓄えたエネルギーで電流を流し続けようとなります。細かい説明は省きますが電流の変化を持っているエネルギーで邪魔しようとするのです。

コイルの両端電圧を求める式を以下に示します。

$$V(t) = L \frac{di(t)}{Dt}$$

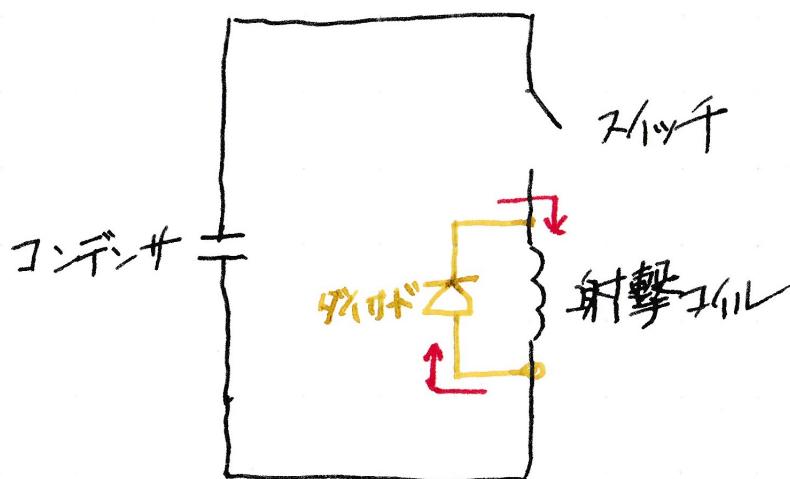
これ、つまるところは電流の微分なんです。

そして考えてみましょう。コイルに電流が流れていって、エネルギーもそれなりに蓄えた状態で急に電流を止めたらどうなるか。

正解は「負方向に高電圧が発生する」です。高電圧といっても、普段コイルガンで扱うレベルの

ものではないかもしれません。(実際にやろうとしたことはないのでわかりませんが...) 大抵の半導体素子は正方向にはそれなりに電圧に耐えられても、負方向はかなり脆弱であることがほとんどです。LEDが最たる例ですね。ですからこのような事態は望ましくありません。ではどうするか。

こうしてやればいいでしょう。

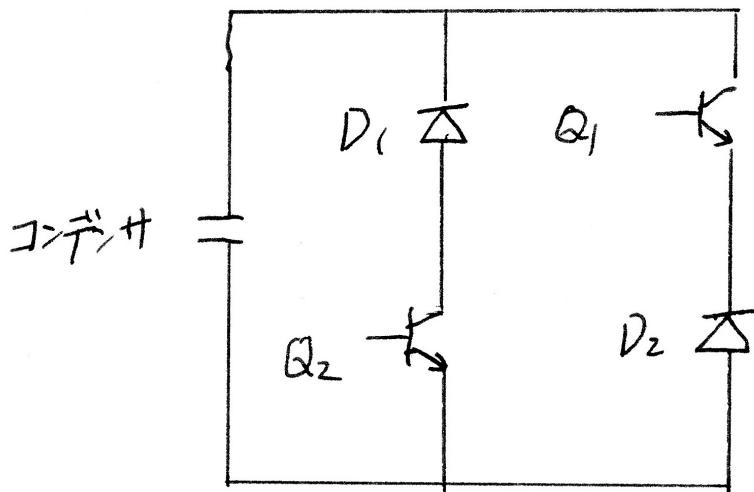


コイルの両端を短絡させるようにダイオードを接続します。ここで重要なのが、このダイオードが電源とは逆方向に接続していること。この回路に通電してからスイッチをOFFにすると、コイルとダイオードだけで回路ができる、そこにコイルの余ったエネルギーから来た電流が流れ続けます。コイルといっても理想的な「コイル」ではなく電気抵抗などもありますから、そのうち電流はゼロになります。ここまで過程で高電圧は発生しません。

ただしこの回路にも問題点があります。それは「電流がゼロになるまでが遅い」ことです。コイルガンにおいて、スイッチをOFFにした後の電流は邪魔でしかありません。コイルに通電した時、金属製の弾に対しては中央に引っ張る方向に力が働きます。ここで弾がコイルの真ん中に差し掛かったところでスイッチをOFFにするとして、この回路だと電流がゼロになるまで時間が掛かるので、かなりの時間弾の進行方向とは逆向き、つまり進んでいる弾を引っ張る向きに力が働くことになります。これは結構まずいことです。ただでさえ低いコイルガンの効率を更に下げてしまう大きな要因になります。

そこで考え出されたのがこの回生回路です。原理は簡単。回路図中の二つのスイッチは同時に操作します。

スイッチがONの時、他の射撃回路と同様にコイルに電流が流れます。続いてスイッチをOFFにすると、コイルは電流の変化を妨げようとしていますから、電流は今まで流れていた方向に流れます。すると、図中のダイオードD1,D2を介して電流がコンデンサに流れていきます。コンデンサには電流が流れると電荷が貯まります。純粋なコイルの電力ロスで電流を減らしていた時と比べて、すぐ電流がゼロになります。



ここまで偉そうに解説しておいて、実はこれネットで見つけた回路図なんです。

このぼんずさんという方です。よくこんな回路思いつくなあとつくづく思います。ありがとうございました。

(<https://pons.blog.jp/archives/82618265.html>)

5.安全装置

コイルガンの製作において、こんな状況に遭遇することは一度や二度ではないはずです。「コンデンサに充電したはいいけど、射撃回路が動かないから放電できない」ことです。手っ取り早いのは、ノコギリやドライバーなど、持ち手が絶縁されている(ように見える)金属製の工具でショートさせてやることでしょうか。確かに簡単ですし、何ならSNS映えはするのですがやる側はたまたまもんじゃありません。ショートした時に出る音と光、またやり方が悪いと感電する可能性もありますし、ただただ単純に怖いです。周りも音と光で驚かせてしまいます。余談ですが、以前この手順で放電したら、この音に驚いて後輩が悲鳴を上げた動画が半ばミーム化しているぐらいですから。

そこで、こんなちょっとした回路を追加してみることにしました。

「大きめのスイッチと抵抗を繋いで、スイッチをONになるとコンデンサの電荷が抵抗を介して放電される」回路です。放電に際して音や火花が散るのは困るので、抵抗値をそこそろ大きめにして放電時の電流を小さくする必要があります。また電力量自体は電圧が大きいので必然的に大きくなりますから、それも考慮して選ばないといけません。

今回はまたま昔使っていた $10k\Omega$ 50Wのメタルクラッド抵抗(セメントで巻き線を固めた後、金属の放熱板を取り付けた抵抗)が部活にあったので、それとこれまた部活にあった大きめのスイッチを組み合わせてみました。

結果は大成功。仮に射撃回路が不調でも安全に電荷を放電させることが出来るようになりました。

6.コイルと弾丸

コイルは今回一番工夫したところです。コイルガンと言いつつもここの改良をしている人は中々見かけません。

コイルガンのコイルといえば、一番メジャーなやり方は外径10mmぐらいのアクリルパイプに、直径0.5mmぐらいのUEW(ポリウレタン導線、エナメル線の一種ではんだごてで被覆が融ける)をきつく巻き付けることでしょう。確かにそれでも相当な威力が出ますし、僕も去年まではそのやり方だったのですが、アクリルパイプって意外と折れやすいんですよね。力を掛けた巻こうとするとポキッと折れてしまいます。コイルを数段分、数時間かけて巻いたところで中のパイプが折れたら一からやり直しです。それからアクリルパイプの外径が10mmなら、大抵アクリルの厚みが1mmなので内径は8mmです。その8mmにぴったり合う弾丸が用意しづらいのです。例えばホームセンターで売っている釘で一番太いのが直径5mm前後。(記憶が曖昧ですが...) 内径8mmのアクリルパイプと比べても3mmすき間があることになります。それにコイルを巻いてある面は外径10mmですから5mmもすき間があります。当然弾を中で動かさないといけませんから、若干の遊びが必要だとしても3mmは多すぎます。それにコイルを巻いてある面は外径10mmですから

5mmもすき間があります。これではエネルギー効率も悲惨なことになってしまいます。

そこではまずは弾丸を選んでからコイルの内径を決めるにしました。弾丸として最も代表的なのは釘でしょう。ホームセンターで釘を買ってきてから頭を落とすと、先端の尖った立派な弾丸のできあがりです。ただ先述の通り太さに限界があるのと、頭を落とす作業が意外と面倒なのが欠点です。弾数が少ないと撃つ度に弾を拾いに行かないといけませんし、文化祭の展示でそれはやりたくないですね。

次に考えたのが鉄の丸棒です。鉄製で中身の詰まった棒です。ただ切断が面倒なのには変わりありませんし、何しろホームセンターに取り扱いがないのです。おそらく錆びやすいからで、通販では買えそうなのですがこれは最終手段ということにしておきました。

他に何かないかとホームセンターの釘が置いてあるあたりをずっと回っていたら…ありました。

「平行ピン」です。鉄の丸棒を短く切断して切り口の角を落とした形をしていて、本来は穴同士の位置合わせなどに使うそうです。ホームセンターに取り扱いのある平行ピンは外径5mmと6mmでした。従ってコイルの内径も5mm~6mm程度にしないといけません。色々考えた結果、弾丸の平行ピンを5mm、コイルの内径を6mmとして1mm遊びを持たせることにしました。後はコイルについて決めるだけです。

7.アクリルかアルミか

先述の通り最もメジャーなのはアクリルパイプに導線を巻いていく方法です。今回は内径が6mmですから、これにアクリルの板厚を考慮して外径を決めることになります。ただアクリルパイプの板厚は薄くとも1mmのものしか見かけません。では素材を変えればいいのでは？そう思って色々調べてみたのですが、例えばアルミパイプでやると強度的には安心ですが、渦電流の影響が気になります。渦電流とは…ここでは割愛します。結局は電力ロスです。

アクリルもだめ、アルミもだめ、(他の金属も同じ理由でなし)じゃあどうする？僕が出した答えは「芯をなくす」ことでした。

芯をなくすとしても、じゃあ何に巻けばいいのでしょうか。そう、最初は強い素材の芯に巻いて、そのあと中身の芯を抜けばいいのです。…と簡単には言いましたが、後から考えればこれが地獄の始まりだったような気もします。

8.接着剤

空心コイルを作るうえで不可欠なのが接着剤などでコイルを固定すること。通常のコイルは芯にきつく導線を巻くことで形を保っていますが、空心コイルにはそれがあいません。そしてその接着剤の使い方や種類でも色々と試行錯誤しました。

まず試したのは、「コイルをきつく巻いてから接着剤で固める」ことを一層ずつ繰り返すことでした。ただし、すぐに致命的な欠点も見つかりました。なんとせっかく作ったコイルが引き抜け

ないのです！これではいくら頑張って巻いても使い物になりません。まず疑ったのはコイルを成型するための接着剤がコイルを巻いたステンレスパイプに付いてしまったのではないか、ということでした。対策として、ステンレスパイプの上に紙を巻いて、仮にその紙と導線が固着しても紙とパイプの間が滑って引き抜けるようにしました。ただこの対策をしたところでコイルが抜けないのは変わらず、結局結論は「コイルをきつく巻きすぎている」ことでした。その後まず一層目を特にゆるめに巻いて、一層目の段階でそれなりに自由に動かせることを確認してから二層目以降も少しうるめに巻いていく…という方法にしました。結果はそれなりに成功と言えるのでしょうか。目標の全長8cm15層まで巻くとさすがに少しきつくなってしまいますが、それでも全く抜けないわけじゃありません。

9.筐体

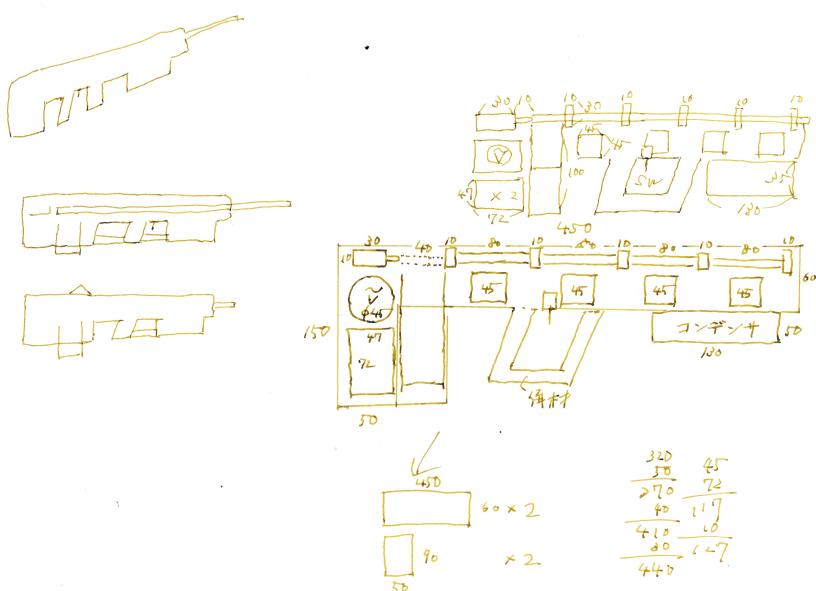
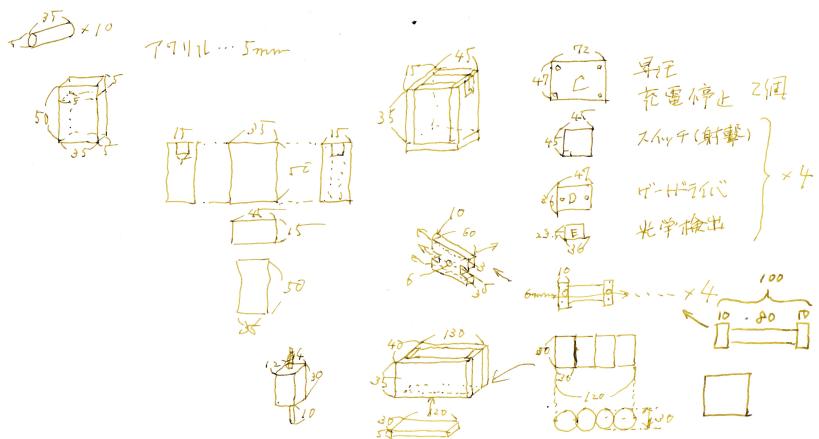
おととしや去年のコイルガンは筐体と呼ぶこと自体が疑わしくなるほど手を抜いてしまいました。動作そのものには関係ないですからね。ただ筐体がないと自分でも感電しそうで怖いので、今年はさすがに少しだけ手を掛けることにしました。

まずは大まかな構造と材質からです。去年の筐体は二枚の1mmのアクリル板をスペーサーで繋ぎ合わせた構造だったのですが、まずアクリル板が薄すぎてたわんでしまったのが反省点でした。さらに基板の取り付けをグルーガンで済ませたり配線もその場で適当に考えたりと…結果、よくわからない理由で動きませんでした。

その反省を活かし、今年はアクリル板の厚さを5mmに、また基板などの取り付けもネジができるようにしました。去年5mmにしなかったのはPカッターで切れるかどうか心配だったからなのですが、試しにやってみたらいけたので問題ありません。

10.総評

ページ数や時間の制約上、画像が少なくて少々雑な説明になったり射撃回路のゲート駆動回路の説明を飛ばしていたりしたのですが、もう締め切りなのでこれで一旦終わりとします。また気が向いたら加筆するので乞うご期待。そして色々語っておきながら、執筆時点(10/7)ではまだ組み立て終わっていないんですね。まあさすがにできていると信じましょう。それではみなさん、物理部展~~2021~~をお楽しみください！



自作コンパイラを実装してみた

高2 中野太輔

1.はじめに

こんにちは、高二の中野です。今回の部誌では、コンパイラについて紹介してみたいと思います！自分自身昔からコンパイラに対して「難しそう」みたいな偏見を抱えていて全く触れてこなかったのですが、ふとしたきっかけで自作Cコンパイラについて体系的にまとめているサイト(参考文献に載せておきます)を見かけて覗いたところ、「意外といけそう」と思ったので軽く始めてみたらハマってしまったという次第です。ところどころ至らない点があると思うので、もし不明な点や疑問があればこのメールアドレス(s2017197@asano.ed.jp)に連絡してください。

2.対象読者

C++, Pythonなどのメジャー言語で基礎的なコードが書ける人。ある程度人のコードが読める人。今回の部誌は扱う内容がかなり学問的というか高度なので、細かいプログラミング言語の文法などは説明しないのでご了承ください。同様に小学生がこれを読むのもあまりおすすめしません。受験勉強してください。逆にこの条件から外れた方々にとっては割と楽しめる内容かと思います。わからない用語・関数などが出てきたら適宜自分で調べてみてください。コーディングにおいて自分のほしい情報を限られた時間でインターネット等で見つけるのは大事な能力です。ちなみに私はC++のリファレンスを参照する際には [cpprefjp\(<https://cpprefjp.github.io>\)](https://cpprefjp.github.io) を主に使っています。

3.コンパイラとはなんぞや

ではここから実際にコンパイラとは何かを説明しましょう。

Wikipediaにはこのように書いてあります。

コンパイラ（英: compiler）は、コンピュータ・プログラミング言語の処理系（言語処理系）の一種で、高水準言語によるソースコードから、機械語あるいは元のプログラムよりも低い水準のコードに変換（コンパイル）するプログラムである。

有名な話かもしれません、コンピュータは0と1の羅列(2進数)をデータとして扱い、それをもとに計算などを実行しています。要は、コンピュータは0010100110011001010101000などのような我々人間にとって一見摩訶不思議に見える数字の羅列を用いています。一方、我々は普段日本語や英語などの自然言語を用いています。この人間とコンピュータの通訳係となってくれるのがコンパイラです。大半のプログラミング言語は人間にとってある程度の可読性が担保されています。もちろん、我々はコンピュータを利用するためにはさっきの0と1の羅列を自分でキーボードで打つわけにはいけませんよね？その、割と人間寄りいわば自然言語寄りのプログラミング言語をコンピュータが理解できるように0と1の羅列に変換するのがまさにコンパイラの仕事なのです。コンパイラの仕組みを理解することできっとあなたはコンピュータと、より親しくなるでしょう！

4.コンパイルの手順

前章ではコンパイラの概要について説明しました。ここから詳しい話に移りたいと思います。まず簡潔に言うと、コンパイラは以下の4つのフェーズを踏んで実行されます。

1.字句解析

↓

2.構文解析

↓

3.アセンブリコードの生成

↓

4.生成したアセンブリコードをバイナリに変換する

これだけだと分かりづらいと思うので具体例を出して説明します。今 int main() {return 0; } というC言語のソースコードをコンパイルしたいとします。当然コンパイラへの入力形式は文字列です(C++で扱うとしたら const char* や std::string のような型を用いるでしょう)。しかしコンパイラからしてみると、文字列の状態だと大変扱いづらいのです。プログラムによっては int main () { return 0; /*hogehoge*/ } のような本来コンパイルする際には不要な空白やコメントがついている場合もあります。それを除去してよりソースコードの本質を抽出するような処理が1番目の字句解析です。このプログラムの場合、ソースファイルは [int, main, (,), {, return, 0, ;, }] という一つづつが意味を持った字句(トークン)の配列に変換され、コメントや空白なども全て無視されます。こうすることで、2番目以降の処理が格段にやりやすくなるのです。次にこのトークン配列を **構文解析木(AST)** という木構造に変換します(なぜこのようにする必要があるのかは後ほど説明します)。この工程を文字通り構文解析と呼びます。そして、その構文解析木を元にアセンブリコードを生成して、そのコード

を機械語(バイナリ)に変換(アセンブルといいます)すればコンパイラの仕事は終わりです。字句解析の過程は比較的に簡単なので、ページ数的にここでは省略したいと思います。では次章からさっそく構文解析から実装していきましょう！ちなみに今回はすべてC++で実装しています。

5.開発環境

今回私はVisual Studioでコーディング&ビルドし、生成したアセンブリコードは WSL (Windows Subsystem Linux) というWindowsマシン上でLinux(Ubuntu)が動く的な仮想環境上で実行ファイルにアセンブルしました。ちなみにビルドする際は x86 ではなく x64 でビルドしてください。 x86 だとなぜか WSL の呼び出しに失敗します。

6.構文解析

gcc などの我々が普段書くようなコードをコンパイルしてくれるコンパイラの全てをここで実装するのは到底不可能なので、まず最初に、四則演算をしてくれるコンパイラを作成したいと思います。例を上げると、 1 + 2 * (3 + 4) という文字列を入力すると 15 を返してくれるアセンブリコードを出力するプログラムを作ります。そして、今章では前章で扱ったフローのうち、構文解析について説明します。ちなみにmain関数にコンパイラのすべての要素を実装するのは可読性的にあまりよろしくないので、構文解析は Parse という専用の関数に実装し、それを main 関数で呼ぶ形式を取りたいと思います。また構文解析の段階では字句解析は終了している (Tokenize 関数で実装している)ので、以下に示す Token クラスのベクターである std::vector<Token> tokens; が既に存在しているという前提で話を進めます。ちなみにこれからはソースコードは全体ではなく変更部分だけを載せたいと思います。

ソース1

```
//予め必要なファイルはここでインクルードする
#include <iostream>
#include <string>
#include <vector>
#include <memory>
#include <functional>
#include <fstream>
#include <algorithm>
#include <cassert>

//トークンの種類を表す列挙型
enum class TOKENTYPE {
    NUMBER, //数字リテラルトークンを表す
    CHARACTER, //文字リテラルトークンを表す
```

```

STRING, //文字列リテラルトークンを表す
SYMBOL, //(), {}, {}などの記号トークンを表す
KEYWORD, //intやreturnなどの予約語トークンを表す
IDENTIFIER, //変数名などの識別子トークンを表す
NONE //トークンが初期化されてない状態を表す
};

//トークンクラス
class Token {
public:
    TOKENTYPE type;
    std::string string;
};

std::vector<Token> tokens;

int Tokenize(){ ... } //ここで字句解析を行う。字句解析の結果はtokensに保存される。本誌では扱わない。

int Parse(){ ... } //ここで構文解析を行う。本章はこの関数に焦点を当てる。

int GenerateAssembly(){ ... } //ここでアセンブリコードを生成する。8章ではこの関数に焦点を当てる

int main(){
    Tokenize();
    Parse();
    GenerateAssembly();
}

```

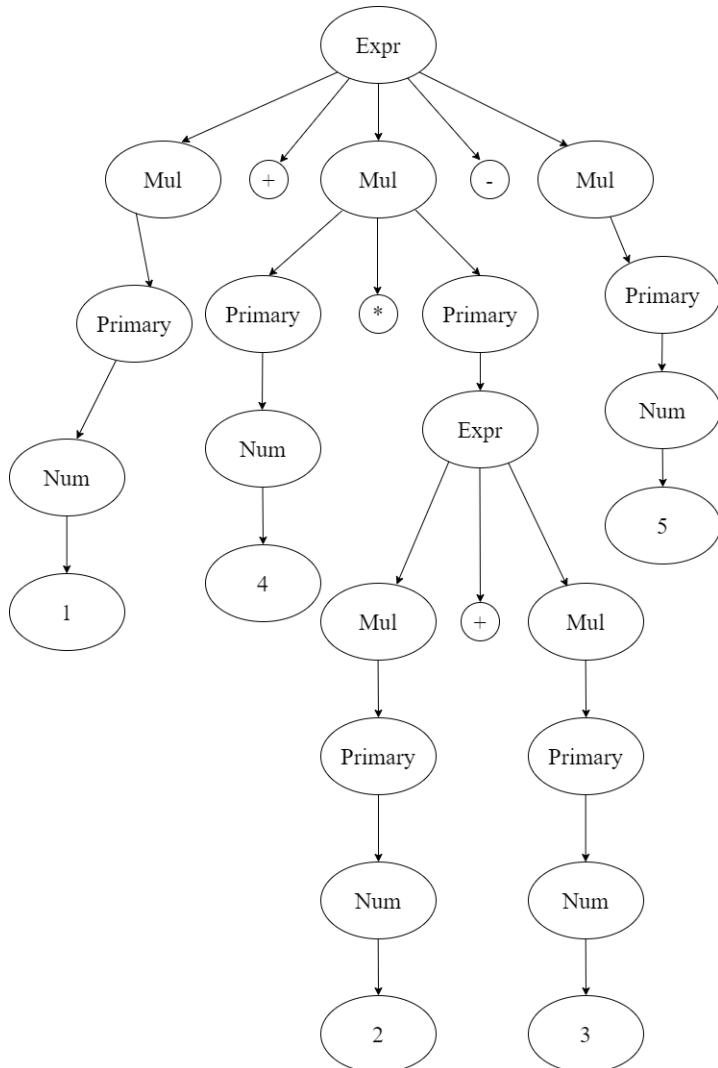
さて、本格的にコーディングを始める準備が整いました。ここで先程提示した $1 + 2 * (3 + 4)$ という数式に考えてみましょう。もし、この世の数式が数字と `+` と `-` のみで表されていたらその値を計算するのはとても簡単です。前から数字を読んでいって `+` で繋がっていたら前の結果にその値を足す、`-` であればその値を引く、というふうにすることで楽に実装できます。問題を複雑にしているのは `+` や `-` ではなく `*` や `/` であるということになります。そして、ここで大事なのは数式には文法があるということです(コンパイル実装においてさらに広義的に考えると、プログラムのソースコードにも文法があるということになります)。ルールがなかったら数式をどの人が見ても普遍的に認識することができません。ロシア語圏では数式はこのように理解されて、フランス語圏ではまた別にこのように理解されて～～ということはありえないのです(自然言語にはそういうのは多々あります)。また、もちろんルールが存在するということはそのルールを記述する専用のフォーマットが存在するということです。有名どころでいうとBNF記法(バッカス・ナウア記法)などがあります。Wikipediaの記事(<https://ja.wikipedia.org/wiki/バッカス・ナウア記法>)を見ると、アメリカでの住所表記のルールがBNF記法で記述されていて面白かったです。このように、BNF記法は広範に及んで

適用可能な記法なので昔からずっと使われているのだと思われます。そして、数式というのもまたBNF記法の適用範囲内なのです。今回はBNF記法を拡張したEBNF記法で数式の文法のルールを書いてみます。それが ソース2 です。

ソース2

```
Expr = Mul ("+" Mul | "-" Mul)*  
Mul = Primary ("*" Primary | "/" Primary)*  
Primary = Num | "(" Expr ")"  
Num = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

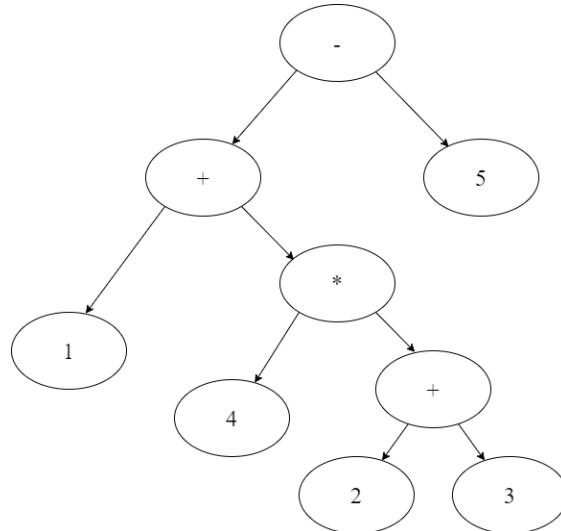
このEBNF記法ではnumが数字リテラル(0~9)を表しており、どの数式もこのような入れ子構造(木構造)で表現することができ、我々が小学校で習った四則演算の順序としっかり一致しています。以下に示す木構造は、式 1 + 4(2 + 3) - 5 を ソース2 のEBNF記法に基づいて分解したものになります。BNFなどの生成規則に基づいて生成された木構造を **構文解析木** といいます。



構文解析木1

一見なんだか難しそうな気がしますが、この木構造もちゃんと上の規則に従っています。そして、プログラミング言語もこれらの規則が複雑になっているだけでちゃんとコーディング時の文法ルールもBNFで記述することができ、ソースコード自体も構文解析木に落とし込むことができます。4章で私はトークン列は構文解析木に変換する必要があると言いました。木構造の特徴として再帰的処理(深さ優先探索など)が行いやすいということがあります。このことこそがトークン列を構文解析木に変換する意義の一つです。何千行もある膨大なソースファイルを正確にアセンブリに変換するには再帰的処理を行うことで明快に実装することができるのです。

ただしExpr, Mulなどのノードは実際のコンパイラ実装には不要なのでそれらのノードを削った木構造のことを **抽象構文木** といいます。以下に示す木構造は上の構文解析木を抽象構文木に加工したものです。



抽象構文木1

では、これからいよいよ実際に抽象構文木を実装していきます。木構造のノードを表すのには基本ポインタ型を用いますが、実装言語がC++なのでその特徴を生かしてメモリ解放などをいちいち気にしなくて済むスマートポインタである `std::shared_ptr` を使いたいと思います。いちいちこの長い型名をタイプするのは面倒なので、グローバル空間で最初にエイリアス宣言をします。

ソース3

```
template<class T>
using Ptr = std::shared_ptr<T>;
```

すると、それぞれのノードの型は次のように表すことができます。

```
//ノードの種類を表す列挙型
enum class NODETYPE{
    ADD, // +ノードを表す
    SUB, // -ノードを表す
    MUL, // *ノードを表す
```

```

    DIV, // ノードを表す
    NUMBER, // 数字リテラルの末端ノードを表す
    NONE //無効なトークンを表す
};

class Node{
public:
    Ptr<Node> lhs, rhs;
    NODETYPE type;
    int num; //数字ノードの時に使う
    Node(){
        type = NODETYPE::NONE;
        num = -1;
    }
};

```

四則演算の場合抽象構文木は完全二分木なので、`lhs` と `rhs` で自分のノードの子ノードを表します。次にノードの追加などのユーティリティ関数を実装しようと思います。また、グローバル空間が汚染されないように構文解析で用いる関数群は全てラムダ式として `Parse` 関数のスコープ内で宣言します。またラムダ式の場合、宣言と初期化を分けられないので前方宣言の必要がある場合は `std::function` を使います。細かい仕様は各自リファレンスで調べてください。ユーティリティ関数を実装したのが [ソース4](#) です。一応述べておくと、以下のコードは全て `Parse` 関数のスコープ内に記述されます。

ソース4

```

size_t token_pos = 0; //今見ているトークンの番号

//もし今見ているトークンが引数の文字列と一致するならtoken_posをインクリメントしてtrueを返す
auto ConsumeByString = [&](std::string str)->bool {
    if (token_pos >= tokens.size())return false;
    if (tokens[token_pos].string == str) {
        ++token_pos;
        return true;
    }
    return false;
};

//lhsとrhsを子ノードに持つノードを返す
auto MakeBinaryNode = [](NODETYPE type, Ptr<Node> lhs, Ptr<Node> rhs)-
>Ptr<Node> {
    Ptr<Node> node = std::make_shared<Node>();
    node->type = type;
    node->lhs = lhs;
    node->rhs = rhs;
};

```

```

    return node;
};

//末端ノード(数字リテラルのノード)を返す関数
auto MakeNum = [](int num)->Ptr<Node> {
    Ptr<Node> node = std::make_shared<Node>();
    node->num = num;
    node->type = NODETYPE::NUMBER;
    return node;
};

```

ソース4 はやることをただ実装しているだけなのでコードを読み解くのはそこまで難しくないと思います。では、いよいよ構文木の真髄を実装します。まず、与えられたトークン列全体は ソース2 における Expr に分類可能です。逆に与えられたトークン列が式(Expr)として解釈できなかったらおかしいですよね？だって、今は式がコンパイラの入力として与えられているのですから。また、 Expr は式全体を評価する以外にも Primary = Num | "(" Expr ")" という生成規則がある以上、別のところで例えば、より木が深くなった部分でも評価する必要があります。最も有名かつ手軽な手法が Expr, Mul, Primary, Num などのそれぞれのEBNFの要素を関数として実装する手法です。こうすることで再帰的な処理ができ、比較的楽に実装することができます。こうすることでそれぞれの生成規則の要素の相互関係を関数呼び出しで対応することができます。この手法を 再帰下降構文解析 といいます。式全体を Expr と決めつけてそのあとによりミクロな視座で部分を評価して木を伸ばすという性質を考えればこの名前の由来も合点がいきます。次の ソース4 がそれぞれのEBNFの要素を関数として実装したコードになります。ひと目見ただけじゃ本当にそれが機能するかどうかわからないと思いますが、この後に実際の例を用いて説明するので安心してください。また、これらの関数の実装の本質は ソース2 の生成規則なので適宜 ソース2 を参照してください。

ソース5

```

std::function<Ptr<Node>(void)> Expr, Mul, Primary; //これらはお互いに他の
関数を呼び出し合うこともあるので前方宣言をしておきます

Expr = [&]()>Ptr<Node>{
    //Exprは1個以上のMulが+もしくは-で結合されている

    Ptr<Node> node = Mul(); //この段階で1つのMulを読み込んだので後は+もしく
    //は-が存在すればそのトークンを消費してMulを読み込む(5-1)
    for(;;){
        if(ConsumeByString("+"))node = MakeBinaryNode(NODETYPE::ADD,
node, Mul());//(5-3)
        else if(ConsumeByString("-"))node =
MakeBinaryNode(NODETYPE::SUB, node, Mul());
    }
}

```

```

        else break;
    }
    return node;
};

Mul = [&]()>>Ptr<Node>{
    //Mulは1個以上のPrimaryが*もしくは/で結合されている
    Ptr<Node> node = Primary();//(5-2)
    for(;;){
        if(ConsumeByString("*"))node = MakeBinaryNode(NODETYPE::MUL,
node, Primary());
        else if(ConsumeByString("/"))node =
MakeBinaryNode(NODETYPE::DIV, node, Primary());
        else break;
    }
    return node;
};

Primary = [&]()>>Ptr<Node>{
    //Primaryは()でくくられたExprか、数字リテラルである
    Ptr<Node> node;
    if(ConsumeByString("(")){
        //もし(があったらそれはExprであるということ
        node = Expr();
        ConsumeByString(")");
    }else{
        //もし違ったら数字リテラルである
        assert(tokens[token_pos].type == TOKENTYPE::NUMBER); //(注)このようなアサーションを入れといたほうがデバッグなどがしやすいです
        node = MakeNum(std::stoi(tokens[token_pos].string));
        ++token_pos;
    }
    return node;
};

```

これらがそれぞれのEBNFの要素を関数として実装した結果です。また式全体すなわち与えられたトークン列は `Expr` としてみなすことができるので、最初に `Expr` 関数を呼び出します。それが `ソース6` です。

ソース6

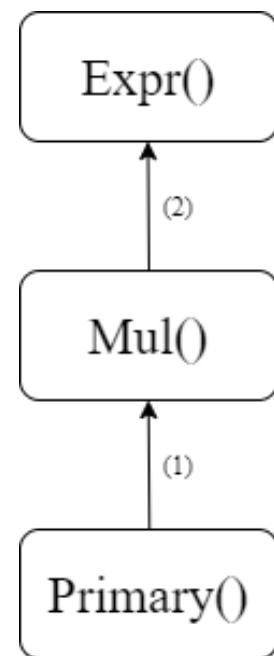
```
Ptr<Node> program = Expr(); //全体の抽象構文木はprogramに格納される
```

これを提示されただけではよくわからないと思うので具体例を出して一緒に考えてみましょう。今回は `4 * 2 + 3 * 7` という式について考えてみます。まずソース6より `Expr()` が呼び出されます。そして今度は `Expr()` 内で `Mul()` が呼ばれます(ソース5の5-1のところ)。次に `Mul()` 内で `Primary()` が呼ばれます(ソース5の5-2のところ)。これらの呼び出し関係を図で表すと ソース7 のようになります。

画像内の矢印は戻り値を表しています。それぞれの関数は戻り値が `Ptr<Node>` のポインタ型なので木もしくはノード単体を表していることになります。(1)では `Primary` の実装を見ると数字リテラルのノードが帰ることになります。それをビジュアル化すると ソース8 になります。

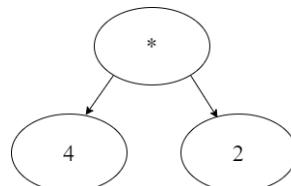


ソース8



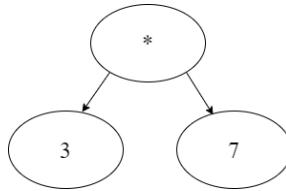
戻り値の関係(ソース7)

また式を見ると `4` と `2` が `*` で結合されているので(2)では以下のような木構造が返されることがあります。



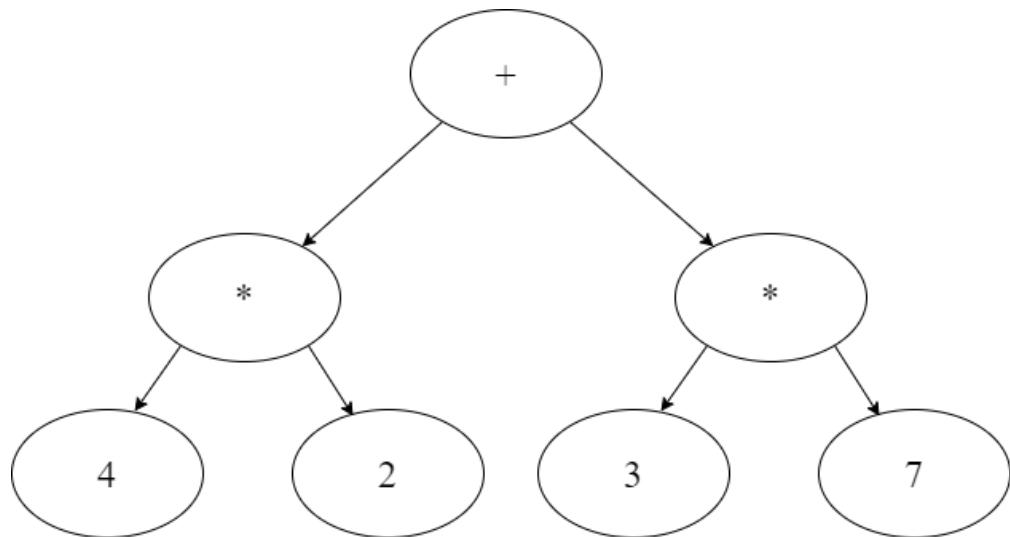
ソース9

この段階でようやく最初に呼ばれていた `Expr` 関数に戻りました。この時点での式全体 `4 * 2 + 3 * 7` のうちの `4 * 2` までが処理されたことになります。またその後には `+` が続いているので、ソース5の5-3よりもう一度 `Mul()` が呼ばれる事になります。またそこで以下のようないくつかの木構造が返されます。



ソース10

そして、`Expr` 関数内の `node = MakeBinary(NODETYPE::ADD, node, Mul());` という処理によりこれらの 2 つの木が `+` 演算子のノードで結合されることになります。その結果が以下になります。



ソース11

これで式全体 `4 * 2 + 3 * 7` の読み込みが完了しました。これ以降はもう `+` も `-` も存在しないのでここで `Expr` の処理は打ち切られ上の木構造がそのまま `program` 変数に格納されることになります。これによってちゃんと式全体がEBNFの文法規則に従って抽象構文木に変換されたことになります。これが構文解析の全てです。後はEBNFの文法規則が複雑になるだけで、「それぞれの要素を関数として実装して再帰的な処理をする」という今までやってきた大まかな流れは変わりません。現に私が今実装しているコンパイラもこの流れを変えていません。みなさん、ここまでお疲れ様でした。次章ではこの章で作られた抽象構文木をもとに実際にアセンブリコードを出力してみましょう！

7.アセンブリで四則演算

本章では前章で構築した構文解析木をもとにアセンブリコードの出力を目標に実装していきます。

まずはスタックという概念について説明します。おそらくプログラマーの大半はスタックやキューといった言葉は一度くらい聞いたことがあると思います。Wikipediaにはスタックについてこのように書かれています。

スタックは、コンピュータで用いられる基本的なデータ構造の1つで、データを後入れ先出し (LIFO: Last In First Out; FILO: First In Last Out) の構造で保持するものである。抽象データ型としてのそれを指すこともあるれば、その具象を指すこともある。

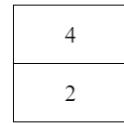
スタックとは誤解を恐れずに言うと、値を追加(Push)したり、コンテナ上の要素を削除(Pop)することができるデータ構造です。ここで重要なのは、要素が削除される場所が一意に定まるということです。以下のような $\{2, 3, 5, 7\}$ と表されるスタックを考えてみてください。(ソース12)

スタックでは、値を追加するときは図の(1)方向でしか追加することができず、値を削除するときも基本的に(1)側にある要素しか削除しかできません。またキューの場合には値を削除するときは(2)側の要素しか削除できません。そしてコンテナの途中に値を割り込んで追加することもできません。これを不便に思うかもしれません、このことこそがスタックそしてキューの最大の特徴なのです。

次になぜコンパイラ作成の過程でスタックの概念が登場するのかを説明します。まず、前提としてどんなに複雑な式でも2項の計算の組み合わせとして計算することができます。前章で扱った $4 * 2 + 3 * 7$ という式で考えてみましょう。まず最初に $4 * 2$ を計算して 8 という結果を得ます。次に $3 * 7$ を計算して 21 という結果を得ます。そして最後に計算結果である $8 + 21$ を計算して 29 という式全体の値を計算することができました。これが私が先程言った「どんなに複雑な式でも2項の計算の組み合わせとして計算」の意味です。

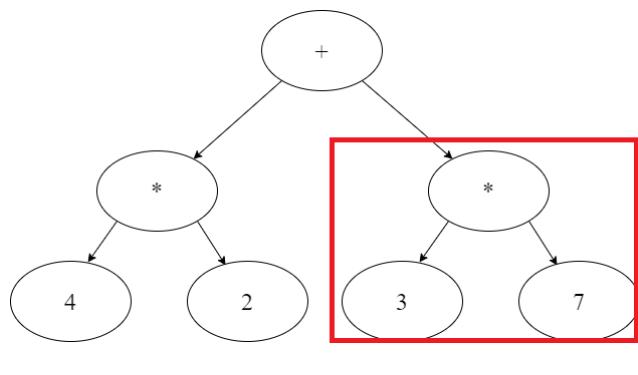
今、前章で扱った抽象構文木を思い出してみてください。あれらも、もし自分のノードが演算子のノードだったら必ず2つのノードを子として保持していますよね？必ずしもその子ノードが数字リテラルであるとは限りませんが。このような背景から式全体を計算するには、「数字リテラルすなわち末端ノードが現れるまで走査し、2つの子ノードを親ノードの演算子ノードに従って計算しそれを上に伝播すればよい」という方針が立ちます。ここでこの伝播を表現するのにスタックを用いるのです。では先程と同様に $4 * 2 + 3 * 7$ の例で考えてみましょう。まず、構文

木の頂点を起点として深さ優先探索(Depth-First Search)をします(深さ優先探索を知らない人は適当に各自で調べてください)。そして、子ノードがもし数字リテラルであればスタックにpushします。そうすることで * ノードを探索した後、スタックは次のようにになります。



ソース13

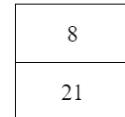
そしてこの時、自分自身のノードは * であるので、スタックから2つの値をポップしてそれらに * 演算を施した結果を再度スタックにpushします。そして次は ソース14 で赤で囲った部分木にも同様の処理を施してあげるとスタックは ソース15 のようになります。



そして、赤で囲った部分木の親は `*` のでスタックから2つの値を同様にポップして計算した結果をスタックにpushするとスタックは以下になります。



ソース15



ソース16

最後に、抽象木全体の頂点は `+` ノードなのでスタックに残っている `8` と `21` に `+` 演算を施した結果をスタックにpushして次のような結果を得られます。



ソース17

これが式全体の演算結果を抽象構文木に基づいて計算するアルゴリズムです。そしてこのアルゴリズムをアセンブリで表現する必要、すなわちその演算をするアセンブリコードを生成する必要があるのですが、それを実装するには多少のアセンブリの知識が必要なのでそれについてまず説明します。

そもそもx86-64のCPUにはスタック上で直接値の計算ができません。基本x86-64のCPUはレジスタと呼ばれるメモリよりも高速にアクセスが可能である記憶領域においての演算しか行いません。このコンピュータのことを **レジスタマシン** と言います。なので、私達は先程紹介したスタックの概念とレジスタの演算を結びつける必要があります。また、レジスタにはそれぞれ名前がついており、それぞれの役割が暗黙的に割り当てられています。以下に示す表が今回使うレジスタたちです。

レジスタ	役割
<code>rax</code>	汎用レジスタ

レジスタ	役割
rdi	汎用レジスタ
rsp	スタックトップのアドレスを保持しているレジスタ

汎用レジスタとは、用途が特に定まっておらずユーザーが様々な演算する際に値を記憶しておけるレジスタのことです。

では次にレジスタに対して演算を施すCPUの命令を紹介します。アセンブリのコードはこれらの命令の連続であると考えることができます。

命令	命令の意味
pop r1	スタックから要素を一つポップしてその値をレジスタr1に保存する
push r1 1 や 15 などの即値でもよい)	レジスタr1に格納されている値をスタックにプッシュする(レジスタではなく r1 や 15 などの即値でもよい)
add r1, r2	r1とr2にそれぞれ格納されている値を足してその結果をr1に保存する
sub r1, r2	r1に格納されている値からr2に格納されている値を引いてその結果をr1に保存する
imul r1, r2	r1とr2にそれぞれ格納されている値を掛け合わせてその結果をr1に保存する
ret	raxレジスタに保存されている値を自分の関数の戻り値としてリターンする

割り算の命令だけは書かなかったのですが、これはなぜかというと除算命令は他の四則演算と仕様が違うので同列に書けなかったからです。除算命令を実装したいときは代わりに以下のコードを使ってください。

```
cqo
idiv rdi
```

この詳細を知りたい人はググってください。

これで準備が整いました。実際に `4 * 2 + 3 * 7` を計算するアセンブリコードを書いてみましょう。以下に示すのはあくまでアセンブリのコードであり、今まで紹介してきたC++のコードとは全く別物なので勘違いしないでください。

ソース18

```
.intel_syntax noprefix
.global main
main:
    #アセンブリの場合エントリポイントはmainなのでその中で処理を書く
    push 4
    push 2

    pop rdi
    pop rax
    #この段階でrdiには2, raxには4が格納されている

    imul rax, rdi #ここで4 * 2の演算を実行して8がraxに新たに保存される
    push rax #そして8をスタックにプッシュ

    push 3
    push 7
    #この段階でスタックは{ 8, 3, 7 }という並びになっている

    pop rdi
    pop rax
    #この段階でrdiには7, raxには3が格納されています

    imul rax, rdi #ここで3 * 7の演算を実行して21がraxに新たに保存される
    push rax #21をスタックにプッシュ
    #この時点でスタックは{8, 21}という並びになっている

    pop rdi
    pop rax
    add rax, rdi #ここで8 + 21 = 29が計算され、結果がraxに保存される
    ret
```

はい、ここまでで計算結果がraxレジスタに保存されるという状態を実現できました。そしてこのコードでは `ret` 命令を追加してmain関数の戻り値として計算結果を返すようにしました。では実際にWSLのターミナルで本当に計算結果が帰ってくるのか確認してみましょう。では、上記のコードを `test.s` として保存してください。そしてこのアセンブリコードを `cc -o test test.s` というコマンドでアセンブルして、`test` という実行ファイルを生成します。その後、`./test` でアセンブルした実行ファイルを実行します。最後に `echo $?` を実行します。このコマンドは直前に実行したプログラムの戻り値を表示するコマンドです。

これらをWSL上で実行すると、なんと、実際に29が表示されました！

自分のプログラムしたものがこうやって形として現れると嬉しいですよね。

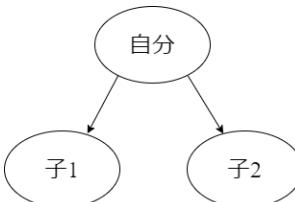
```
29
:~/mnt/c/Users/.../Documents/2021年度部誌/test$ cc -o test test.s
:~/mnt/c/Users/.../Documents/2021年度部誌/test$ ./test
:~/mnt/c/Users/.../Documents/2021年度部誌/test$ echo $?
```

ソース19

8.アセンブリコードの自動生成

いよいよ、この章では6章と7章で学んだことを総動員してコンパイラ最後の砦、アセンブリコードの生成を実装していきます。これを実装すれば4章で説明したコンパイルのフローがすべて実装されたことになりました。後は構文などのプログラミング言語の要素を追加していくたびにこれらのフローのそれぞれの要素に追加で実装すればいいので、今章で大まかな全体としてのプログラムは完成することになります。まあこの後も様々困難が待ち構えているのですが。。。

ここではC++で前章で取り扱ったようなアセンブリコードを自動生成できるように実装をするのですが、大まかな方針は既に前章で紹介しています。「数字リテラルすなわち末端ノードが現れるまで走査し、2つの子ノードを親ノードの演算子ノードに従って計算しそれを上に伝播すればよい」ということです。これだけ言われてもやはりピンとこないと思うので丁寧に説明していきます。まず抽象構文木の特徴でもあるのですが、末端ノード以外、すなわち数字リテラル以外のノードは以下の構造をとっています。



ソース20

この場合、「自分」は子ノードを2つ持っているので必ず演算子ノードです。また「子1」、「子2」は数字ノードであるとは限りません。しかし、抽象構文木の頂点から深さ優先探索をして末端に行けば必ず「子1」と「子2」が数字ノードである部分木が1つ以上は存在します。なぜならどんなに複雑な式であっても必ず $\triangle + \triangle$ のような2項からなる式の組み合わせにすぎないからです。これらのことから、もし子ノードが数字だったらその値をpush、でなければ走査を続けるといったようなコードを実装すれば良いことになります。C++でこれを実装すると以下のようにになります。

ソース21

```

int GenerateAssembly(){
    //生成するアセンブリコードはここに格納する
    std::string res = ".intel_syntax noprefix\n.global main\nmain:\n";
    //構文木を再帰的に走査する関数。再帰呼び出しをするのでこの命名をしたけどあまり気に入ってない
    std::function<void(Ptr<Node>)> Recursive;
    Recursive = [&](Ptr<Node> node){
        //もし今走査しているノードが数字ノードだったらその値をプッシュ
        if(node->type == NODETYPE::NUMBER){
            res += ("\tpush " + std::to_string(node->num) + "\n");
            return;
        }

        //子ノードについても同様の走査をする
        Recursive(node->data[0]);
        Recursive(node->data[1]);

        //この段階で2つの値がスタックトップに存在しているはず。
        res += "\tpop rdi\n";
        res += "\tpop rax\n";

        switch(node->type){
            case NODETYPE::ADD:
                res += "\tadd rax, rdi\n";
                break;
            case NODETYPE::SUB:
                res += "\tsub rax, rdi\n";
                break;
            case NODETYPE::MUL:
                res += "\timul rax, rdi\n";
                break;
            case NODETYPE::DIV:
                res += "\tcqo\n";
                res += "\tidiv rdi\n";
                break;
        }
        //最後に計算結果をプッシュ
        res += "push rax\n";
    };
    //構文木の頂点から走査を始める
    Recursive(program);

    //計算結果をraxにポップする
    res += "\tpop rax\n";
}

```

```
//最後に計算結果をプログラムの戻り値として返す
res += "\tret\n";

return 0;

}
```

これで実際にプログラムに `4 * 2 + 1` という式を入力として与えて `std::cout << res;` で生成されたアセンブリコードを実際に出力させると以下が得られます。

```
.intel_syntax noprefix
.globl main
main:
    push 4
    push 2
    pop rdi
    pop rax
    imul rax, rdi
    cqo
    idiv rdi
    push rax
    push 1
    pop rdi
    pop rax
    add rax, rdi
    push rax
    ret
```

ソース22

これでようやくコンパイラの体裁をなしてきました。後は得られたアセンブリコードをコンパイラ内部でアセンブルして実行ファイルを生成するところまでやっていきましょう。以下のようないくつかの関数を実装してください。

ソース23

```
int Assemble() {
    //出力されたアセンブリコードをresult.sに保存する
    std::ofstream assembly_file{ "result.s" };
    assembly_file << assembly_code;
    assembly_file.close();

    //result.sをバイナリファイルに変換する
    int res = system("wsl cc -o result result.s");

    //もしccコマンドが正常終了しなかったらエラーログを出す
}
```

```

if (res != 0) std::cout << "Assembling didn't go well.\n";
return 0;
}

int Run() {
//resに式の演算結果が保存されるのでそれを出力
int res = system("wsl ./result");
std::cout << res << "\n";
return 0;
}

```

このコードはコンパイラ実装の本流とは関係ないので説明は省略します。

よって、`main` 関数内の関数呼び出しの順番は

```

Tokenize
↓
Parse
↓
GenerateAssembly
↓
Assemble
↓
Run

```

となります。

ここまで、コンパイラ作成のおおまかな流れを解説しましたが、いかがでしたでしょうか？
C++を使った実際の実装はここまでとしたいと思います。これから変数などのいわゆる「プログラミング言語」らしい機能を紹介したいと思いますが、これらの実装を詳しく説明するのは流石に骨が折れるので、概要だけお話したいと思います。

9.比較演算子

ここでは比較演算子を説明して `12 > 2` などの真の条件式が与えられたら1を、`1 == 2` などの偽の条件式が与えられたら0を返せるようにします。構文解析木の実装方法は前章で丁寧に扱ったのでその部分はEBNFの変更点だけを説明します。まず、最上の階層で `expr` を使いたいので既存のEBNFを以下のように書き換えます。

ソース24

```
add = mul ( "+" mul | "-" mul )*
mul = primary ( "*" primary | "/" primary )*
primary
```

また、`==` よりも `>` などの不等号のほうが先に結合します(例: `1 == 4 > 1` の場合、`4 > 1` が先に評価され、その次に `1 == 1` が評価されます)。MSVCなどのコンパイラで実際にソース25 のようなコードをコンパイルすると `false` が帰るのでやはり `==` より `>` が先に結合されることがわかります(もし `2 == 2` が先に評価されるのであれば、`true` が帰るはずです)。

```
#include <iostream>

int main() {
    std::cout << std::boolalpha << (2 == 2 > -1) << std::endl;
}
```

ソース25

このことを考慮すると以下のようないくつかのEBNFを新たに追加すればいいとわかります。

ソース26

```
expr = equality
equality = relational ( "==" relational | "!=" relational )*
relational = add ( "<" add | "<=" add | ">" add | ">=" add )*
```

このようなEBNFを設計することで `12` などの数字単体からなる式や、`1 + 1 > 0` などの不等式や、`2(3 + 4) > 10 == 1` などのような不等号と等号を混ぜた式まで構文解析できるようになります。ここまで段階で比較演算子に対応した `Parse()` を実装することができます。次に比較演算子に対応した `GenerateAssembly()` すなはちアセンブリコードの生成の実装を説明します。基本は四則演算のアセンブリ生成と同じ流れです。木全体を走査して、`+` ノードであれば `add` 命令を生成するのと同様に `>` や `==` ノードであればそれに対応したアセンブリの命令を生成すればよいです。以下にその命令を示します。

命令	命令の意味
cmp r1, r2	整数レジスタr1とr2の値を比較して比較結果をフラグレジスタに格納する
sete al	フラグレジスタを読み取って、 cmp 命令の比較結果が <code>r1 == r2</code> となったら al レジスタに 1 を格納する
setne al	フラグレジスタを読み取って、 cmp 命令の比較結果が <code>r1 != r2</code> となった al レジスタに 1 を格納する
setl al	フラグレジスタを読み取って、 cmp 命令の比較結果が <code>r1 < r2</code> となったら al レジスタに 1 を格納する
setle al	フラグレジスタを読み取って、 cmp 命令の比較結果が <code>r1 <= r2</code> となったら al レジスタに 1 を格納する

ただしここで一つ注意があります。上の表で `al` というレジスタを用いましたが、実はこれは頻繁に使う整数レジスタ `rax` の下位8bitに割り当てられた別名レジスタに過ぎないです。従って比較結果を `rax` に格納したい場合は残りの56bitを0で埋める必要があります。それを行ってくれるのが `movzb` 命令です。なお、ここでは `>` と `>=` に対応する命令を紹介していませんが、単に構文解析時に右辺と左辺を入れ替えれば `>` や `>=` は `<` や `<=` に置換可能です。`A > B` は `B < A` と同値であるからです。

命令	命令の意味
<code>movzb rax al</code>	<code>rax</code> レジスタの <code>al</code> 部分はそのままで残りの56bitをゼロクリアす る

以上を踏まえて実装すると以下のようになります(掲載しているのは `GenerateAssembly` 関数の `switch` 文の新規ラベルです)。ちなみに構文解析木では、 `==`, `!=`, `<`, `<=` がそれぞれ `NODETYPE::EQUAL`, `NODETYPE::NOT_EQUAL`, `NODETYPE::LESS`, `NODETYPE::LESS_OR_EQUAL` に対応しています。

ソース27

```
case NODETYPE::EQUAL:
    //ここが処理されている段階で `A == B` のAがraxレジスタに、Bがrdiレジスタに
    //格納されている
    res += "\tcmp rax, rdi\n";
```

```

res += "\tsete al\n";
res += "\tmovzb rax, al\n";
break;
case NODETYPE::NOT_EQUAL:
//以下同様に実装していく
res += "\tcmp rax, rdi\n";
res += "\tsetne al\n";
res += "\tmovzb rax, al\n";
break;
case NODETYPE::LESS:
res += "\tcmp rax, rdi\n";
res += "\tsetl al\n";
res += "\tmovzb rax, al\n";
break;
case NODETYPE::LESS_OR_EQUAL:
res += "\tcmp rax, rdi\n";
res += "\tsetle al\n";
res += "\tmovzb rax, al\n";
break;

```

今まで長々と説明してきましたがいざ実装すると意外とあっけなかったですね。ここまで以下の
ようなコードがコンパイルできるようになります。

ソース28

```
1 == 1 <= 0
```

10.おわりに

さあ、ここまで比較演算子の実装まで終わりました。私はC言語の主要な機能(ローカル変数、関数、ポインタなど)を実装したのですが、私が今これを執筆しているのが2021年9月25日で、時間がキツキツになっているのでここで終わりとしたいと思います。今回の部誌でコンパイラについて興味を持ってくれる人がいたら幸いです。もし、私のコードの全体が見たい人がいたら1章に載せたメールアドレスに連絡してください。読んでいただきありがとうございました！！

編集後記

高2 永田雅典

この度は、私達物理部の部誌「Positron 2021」を手にとっていただき、どうもありがとうございます。今年の部誌の編集を担当しました、高校2年の永田です。今年の部誌は2万字級の記事がひとつ、1万字級の記事がひとつ、そして一人で2つの記事を執筆した部員が合計1万字超の記事を書き上げ、全体で100ページもの超大ボリュームの部誌を完成させる事ができました。記事の内容はいかがでしたでしょうか。

さて、今年の部誌では新たな試みとして、記事の下書きをMarkdownにて執筆してもらい、編集と組版(レイアウト)の作業をVivlioStyleというオープンソースのCSS組版ソフトウェアでこなすということを行いました。従来のWordファイルでの原稿提出で課題であった、レイアウト管理の煩雑さを解消し、CSSで部誌全体のレイアウトを一括変更できたことはとても良かったと思います。ウェブ上に公開されているVivlioStyleのテーマをベースに様々な調整を加えて、そこそこ雑誌らしい整った体裁になりました。

また、Wordファイルと異なりMarkdownファイルはシンプルなテキストファイルであるため、校正時にはMarkdownファイルをGitでバージョン管理することで、校正前後の差分を閲覧しやすくするなど細かい事務作業の効率化を図ることもできました。私は今年で部活動を引退することになりますが、MarkdownとCSS組版を駆使した編集のシステムは、ぜひ来年の編集担当にも継承してもらいたいと思います。

ただし、VivlioStyleのCSS組版は非常に細かく高度な編集ができる一方で、組版の難易度は極めて高く、CSSに慣れていない私には厳しいものがありました。そのことは編集の作業時間に現れしており、この部誌のすべてが完成したのは文化祭2日前の木曜日となってしまいました。その木曜日の早朝に印刷するという約束で待機してもらっていた顧問の先生には本当に申し訳ありません。部員の皆さんにも多くの作業を手伝ってもらいました。文化祭直前の忙しい中お手数をおかけしたなと思っています。ちなみに、ここで2割ほど責任転嫁をしておくと、木曜の未明になって部誌の原稿を完成させた某部員にはちょっとだけキレております。

こちらの部誌は、私達物理部のウェブサイトにて電子版を公開する予定ですので、ぜひどちらも合わせてご覧いただければと思います。裏表紙にウェブサイトのQRコードを記載しておりますのでご利用ください。2018年の部誌ではPDFを直接ウェブに記載し、2019年の部誌ではHTMLを直接記述してウェブ記事のような形にしてきたPositron電子版ですが、今年はVivlioStyleの昨日を用いてついにキ○ドルのような手軽に読める電子書籍が実現できるかもしれません。(実際にどうなるかはこの文章の執筆時点ではわかりませんが・・・)

それでは最後に、この冊子を手にとってくださった皆さんに改めてお礼を申し上げます。そして、この後も引き続き「物理部展丼2021」と「第42回打越祭」をお楽しみください！