Sunaba の設計と実装

平山尚

2013 年 11 月 18 日 11 時 21 分 (GMT) バージョン

目次

第 1 章　まえがき 1
1.1　Sunaba の目的 1
1.1.1　最大の学習効果とは何か 1
1.1.2　挫折する原因は何か 2
1.1.3　難しい概念 2
1.1.4　Sunaba が満たさねばならない条件 4

第 2 章　設計 7
2.1　言語仕様 7
2.1.1　型 7
2.1.2　代入文の書式 7
2.1.3　制御構造 8
2.1.4　ブロック構造表現 8
2.1.5　return と戻り値の問題 9
2.1.6　名前付きメモリのスコープの問題 9
2.1.7　定数のスコープの問題 10
2.1.8　IO の問題 10
2.1.9　バーチャルマシン仕様 10
2.1.10　メモリ空間 12
2.2　ユーザインターフェイス 13
2.3　コンパイラ 13
2.4　実行プログラム 13
2.4.1　発音機構 13
2.5　デバグ機能 14

第 3 章　実装 15
3.1　ユーザインターフェイス 15
3.2　コンパイラ 15

3.3 実行プログラム 16
3.3.1 発音機構 16
3.4 デバグ機能 17

第 4 章 課題 **19**
4.1 言語仕様 19
4.2 ユーザインターフェイス 19
4.3 バーチャルマシン 20
4.3.1 命令の追加 20
4.3.2 関数ポインタ対応 20
4.4 コンパイラ 21
4.5 実行側 21
4.5.1 発音機構 21
4.5.2 機能追加 21
4.6 デバッグ機能 22

第 1 章

まえがき

1.1 Sunaba の目的

Sunaba の目的は、プログラマ人生の最初の 100 時間を極力実り多いものにすることである。

具体的に言えば、プログラミングを学びたい人に対して、最大の学習効果を提供するためのものである。

1.1.1 最大の学習効果とは何か

学習効果とは、ある人がこれによって学んだ結果得るスキルであり、また、プログラミングを楽しいあるいは面白いと思うような体験である。よって、この言語での経験がその後の人生において役立たねばならないし、また、この言語の経験が楽しいものでなければならない。

そしてもう一つ、挫折を防ぐことの重要性がある。

世の中の大半の人はプログラマに向いていない。ただしここで「向いていない」と言うのは、「放っておいてもプログラミングを生業として行けるわけではない」という程度の意味である。

向いている人は、放っておいても勝手にプログラマになるのであって、魅力的な課題を与えさえすれば、あとは放っておくだけで済む。仕事の中で自然に鍛えられるのは、このような恵まれた人間だけだ。

しかし、大半の人間は向いていない。この向いていない人間にいかにしてプログラミングを教えるかが極めて重要なのである。

向いていないなら放っておけば良いではないか、と思うかもしれない。しかし、それでは必要な数のプログラマを確保できないのが現実である。また、プログラマの肩書を得る人間であっても、必要な技術水準を満たさぬ者は多い。このような人間を作らぬようにす

るためにも、プログラミング教育はもっと周到に行われねばならない。

1.1.2 挫折する原因は何か

では、プログラミングは何故難しいのだろうか。

現在の私の結論は、一言で言えば、

「目標と現状の間に連続した線を引く能力が足りないから」

となる。学習時間や経験の問題ではない。そのような考え方をそもそも知らないのである。「目標を達成するにはどうすれば良いか」という問いを執拗に行う訓練を経ていない。

プログラミングは他の多数のスキルと同じく、パターン当てはめや暗記の能力だけでは一流には至れない。最終的に物を言うのは、問題解決力であり、さらに言えば問題設定力である。つまり、目的をどのように把握し、目標と言う名のチェックポイントをいかに設定し、具体的にそこへ到達する道をいかに引くかである。

このような能力を鍛えるには、少数の道具しかない状態に追い込むのが良い。覚えることが少なければ、記憶力の差は問題にならない。また、独自の言語であれば情報源はほとんど存在せず、情報を集めるという行為は無力となる。

したがって、他の言語の経験をそのままは使えないようにし、かつ、覚えるべきことを最小化することがこの目的では有効である。

1.1.3 難しい概念

では、問題解決力は覚えることを最小にし、適切な難易度設定で課題を出していけば良いと言えるだろう。これはゲームのレベルデザインと同様の考え方で良い。

ただ、具体的に考える際には、実際問題プログラミングのどのような概念が難しいのか、ということを知っておく必要がある。これは観察することによってしかわからないことであり、プログラミングが運良く出来てしまった者にはいくら想像してもわからないものである。既存の教材はそのような者が作っていたために、学習者にとって最適なものになっていなかったのではないか、というのが私の仮説だ。

変数の概念

変数はプログラマになってしまった者にとっては自明の概念だが、学習者にとってはそうではない。むしろ、極めて理解が困難な概念である。

これが理解困難な原因としては、まずは記号の問題がある。

```
a = 5
```

と書かれれば、普通は「a と 5 が等しい」と読む。これを「a が 5 と等しくなるようにせよ」と読めという段階で、すでに不自然なのだ。

「わかった、そういうものだとしよう」と納得してくれれば良いが、人は必ずしもそう物分りの良いものではない。「なんでだよ」と思えば、その不満がいつまでも心の底にくすぶり続け、意識的、あるいは無意識的に理解や努力を妨げる。よって、初心者にとって良い記法は極めて重要なのである。学習時の記法と、実用時の記法を変えるくらいのことはしても良い。

また、その「変数」という名称も、数学で習った概念が助けになれば良いが、必ずしもそうとは限らない。構造体やクラスが存在する言語で学習する場合、かなり序盤から単なる数でないものが変数に入ってくることになり、これによるギャップが理解を妨げるのである。

そして、最も問題なのが、その抽象性である。

「x なるものがある」と言われても、それはどこにあるのであろうか。コンピュータが何なのかもわかっていない状態で、「ルール上こうなっていて、こうなる」という地に足のついていない話をすれば、わからないのも当然である。

人は具体的なものをいくつか見て、そこから共通性をくくり出すことによって抽象的な概念に至る。断じて逆ではない。加えて、変数は全てがグローバルというような言語でもない限りスコープの概念がついて回ることになり、序盤から覚えることが増えてしまう。

この抽象度を下げるには、変数なる概念なしで始めるべきであろう。すなわち、「メモリと呼ばれるダイアルが何万個か入っていて、好きな番号のダイアルを好きな数にセットするのがプログラム」というレベルにまで話を単純化してしまえば良い。変数なる概念はどこにも出てこないし、「ダイアルみたいなのがコンピュータに入ってるんだよ」と言えば、これは極めて具体的である。「括弧の中でだけ有効な x なる何か」と比べてみれば良い。

また、変数を登場させる際にも、「とある番号のダイアルに番号じゃなくて名前をつけてみた」「でも番号は知らなくていい」というように説明すれば、具体性をさほど損わずに済む。

必然的にオブジェクト指向的な手法を最初から用いることはできないが、歴史上後から出てきたものを先に教えることが合理的であることはそれほどない。現れるものは、大抵の場合必要性を背景にして現れる。後になって現れたということは、必要性もまた後になって現れた可能性が高く、実際そのようなものがなくとも 100 行や 200 行のプログラムなら書けるのである。そもそも、単なる整数変数ですら抽象的すぎてわかりにくいのであるから、それ以上に抽象的な概念を最初から登場させるなど無茶である。

関数の概念

関数は理解が困難な概念である。変数に比べればはるかに容易いが、それでも最初からこれを前提とする必要はあるまい。

なお、関数は変数に比べれば教えやすい。まずは単なるマクロ展開として考えれば具体性は損ねなくて済むからだ。「foo() と書いてある所に、用意しておいた foo の中身を貼り

つけるんだよ」というわけである。この理解で困るのは相当に先のことであり、その意味で関数の概念はそう厄介なものではない。

ただし、「関数」という用語には問題がある。実用的な状況において、関数がまず必要とされるのは、プログラムの使い回しをしたいという欲求からである。したがって、「再利用可能なプログラム片」というイメージを持った言葉がふさわしい。「関数」という数学用語を持ちこむ必然性はどこにもない。サブルーチン、サブプログラムなどはその意味では優れている。ただし「ルーチン」は日本語として定着しているとは言い難いため、サブプログラムの方が良かろう。もちろん、ここで引数の有無、戻り値の有無などは問題ではない。それらがあってもなくても、「プログラム片」であることに違いはなく、それら入出力は単に便利な道具にすぎないからである。

1.1.4 Sunaba が満たさねばならない条件

以上のことから言って、Sunaba には満たさねばならない条件、あるいは満たすことが望ましい条件がいくつか設定される。

言語仕様

言語仕様は極限まで小さくなくてはならない。また、極限まで理解しやすくなければならない。プログラミングのお約束は一切を排除すべきである。

とりわけ、変数と関数に関しては、用語、書法に十分注意し、また、それらがなくともプログラムが書ける状況を用意すべきである。

加えて、書かねばならない文字数は小さいほど良い。初心者はキーボードにも熟練していないことが多く、キーストロークが多いことのコストを強く感じうる。また、書かねばならない文字数が少ないことは、自然と言語の書式上の自由度を奪うので、「同じ結果を得るプログラムは同じ書式を持つ」という理想に近づきやすくなる。

書式あるいはスタイルは所詮枝葉末節であって、そこに労力を割かせる必要はない。

機能性

達成感のあるプログラムを作れる程度の機能は持たねばならない。「達成感のあるプログラム」の定義が問題になるが、ここでは「4 マスづつ落ちてくるあの有名な落ちものゲームを作れる程度」とした。

もしサウンドを鳴らす機能が達成感を得る上で効果が大きいのであれば、それを入れても良いだろう。3D 描画、ネットワークなども同様だが、これらはさしあたっては後で良かろうと考えている。

プログラムを実行する手間が最小であること

最小の操作、最大のわかりやすさでプログラムを書いて実行できねばならない。

その意味で、「メモ帳で書いて、アプリのウィンドウにドラッグアンドドロップ」は悪くない。しかし、「メモ帳で書いて、ダブルクリック」で済めばなお良いし、そもそも「統合開発環境を用意して、そこで書いて F5」というのはそれに勝る。

しかし、凝ったものを用意すれば、その使い方を覚えるコストが発生することは忘れてはならない。ただし、エディタの統合によって保存し忘れと、メモ帳でどうしても不足する機能を補えるため、実装する時間さえあるなら最終的にはエディタを内蔵すべきであろう。

Sunaba を始める手間が最小であること

最小のインストール手順、時間であるべき。また、windows のオペレーションについても極力少ない知識と経験しか必要としないようにすべきである。

その意味で、仮に開発環境的なものを提供するとしても、その機能は最小に抑えるべきである。ある程度書けるようになったら別の言語に移行する、という前提から言って、「慣れてくると便利な機能」は基本的には不要である。

第 2 章

設計

2.1　言語仕様

言語要素を最小化することのみならず、段階的に学べるべきであることはすでに述べた。

変数と関数は必須ではなく、また書式の制約を強めるために、ブロックの識別は python にならうのが良い。

変数なしで済ますためにはメモリへの直接のアクセスが必要であり、これがあれば関数に引数や戻り値がなくとも関数が使える。以上から言って、番号を指定して直接メモリを扱う機能を基本に据えるべきである。

2.1.1　型

Sunaba に型の概念はない。メモリが丸見えであるから、メモリに格納できる整数型が唯一の型である。後述するようにこれは 32bit 値であり、「メモリ一個は-10 億から 10 億まで覚えられる」と単純に考えれば良い。正確な値を覚える必要はない。

また同様の理由で、ポインタと整数は区別しない。ポインタはメモリの番号が入った整数にすぎない。これは実用的な大きさのプログラムを書き始めると大変不便で、バグの温床となるわけだが、最初の 100 時間のうちにそれが問題になることはなかろう。その時期において、ポインタの書式を別途用意して学習者に負担を求めるのは得策ではない。

また、これによって、構造体もクラスも存在しえない。これらはオブジェクトなる概念につながる重要なものだが、Sunaba では扱わない。Sunaba 卒業後の課題として良かろう。

2.1.2　代入文の書式

代入文の書式は、

```
a -> 5
```

と右向きの矢印とした。C と混同する可能性はあるが、許容することとした。これは、「a が 5 を覚える」と、a を主語とした文章として読ませるためである。

```
a <- 5
```

の場合、「a に 5 を覚えてもらう」と主語が右辺に行き、若干不自然だと考えたためだ。ただし、こちらを採用している言語は多いので、Sunaba は少数派ということになる。

また、普通の、

```
a = 5
```

を用いなかったのは、比較演算子を==でなく=としたかったからである。また、これを「a が 5 と等しくなる」と読むのは、若干不自然である。

```
a := 5
```

のような独自の記号もありうるが、イメージが湧かないという欠点がある。いっそ、

```
a が 5 を覚える
```

とそのまま書けるようにするのも良いかもしれない。ただし記述量が多いので短縮記法は別に必要であろう。この記法だと、

```
a[( + 3) * 12 + 9] が 余り(a,7) * 100 + 3 を覚える
```

のように式が複雑化した時に不自然になるのも気になるが、すぐに短縮記法に移るだろうから問題はあるまい。

2.1.3 制御構造

制御構造は最小を旨とする。その意味では while だけあれば良い。しかし、while のみでプログラムを書くのは最初の 100 時間においてすら苦痛であり、if は構文糖として用意した。

しかしながら、構文糖はそれだけである。elseif や else は用意しない。for も foreach も switch も用意しない。break や continue も用意しない。これらなしで求める制御構造を作り上げる訓練は必要であり、一旦慣れればそれほどの苦痛にはならない。Sunaba において生産性は重要ではないからである。

2.1.4 ブロック構造表現

ブロックは python 形式で表現し、括弧は用いない。

これで嫌でもインデントの習慣がつき、インデントと実際の構造が乖離するようなことが起こり得なくなる。

また、プログラムも短くなり、記述量も減る。

2.1.5 return と戻り値の問題

Sunaba には return がない。このため、部分プログラムの途中で処理を終えることはできない。さらには、プログラムの途中で処理を終えたい、という欲求もかなえられない。無限ループで止めるか、メモリの範囲エラーや 0 除算を起こして無理矢理止めることしかできない。

しかし、途中 return はそれほど頻繁に使うものではなく、なければないで済む機能でもある。また文法要素も増えてしまう。

```
if a = 1
   return 1
return
```

値を返す return と返さない return が混在した時にエラーを出す必要もあるし、そもそもそのような区分があることが学習要素を増やしてしまう。

そこで、Sunaba では「特別な名前付きメモリをセットしておけばそれが出力」というだけの文法とし、制御構造に関する問題は出力から分離した。

```
def foo()
   out -> 1
```

やろうと思えば、これと別に値なしの return を作っても良い。しかし、今のところ Sunaba においては途中 return はないものとしている。

なお、部分プログラムの外であっても出力は可能である。この場合、メッセージウィンドウに表示される作りだ。しかし、出力をしていないプログラムにおいてはそのようなものが表示されるのは邪魔であり、明示的に出力したプログラムでのみ表示したい。とはいえ、スタックの底に内部的に存在する main の戻り値を書き変えたかどうかをストアの度にチェックすることはコストが大きすぎる。やむを得ず妥協として、0 以外の値が入っている時に表示する作りとした。

部分プログラム外では出力できない、としても良いが、中と外でできることが違えば、それを説明せねばならなくなる。だから、どこでも出力は可能ということにしておいた。ただし、この言語において main の戻り値を見られることにさしたる意味はない。せいぜいデバグに使える可能性がある、という程度である。

2.1.6 名前付きメモリのスコープの問題

Sunaba において、名前付きメモリは部分プログラムの境界をまたげない。つまり、外から中のものは見えないし、中から外のものは見えない。これに関して例外はない。

C/C++ ではグローバル変数のみは例外とされており、それとは異なる。しかし、C/C++ においては関数の外にコードはなく、Sunaba においては言うならばファイル全

体が main() なのであるから、そこに置かれている変数はグローバルではなく main() のローカルと解するのが妥当であろう。例外とすることはわかりにくい。

2.1.7 定数のスコープの問題

C において#define で作られる定数の類が全くないのは、あまりにも不便である。定数の類はグローバルが妥当であろう。でなくては、パラメータの類を変更してプログラム全体に影響を及ぼすようなことができなくなり、マクロ置換を上にかぶせたくなってしまう。それでは C の二の舞だ。

しかしその一方で、部分プログラムの中に限定された定数がないのもまた不便である。もし部分プログラムに限定された定数を認める場合、この作りは名前付きメモリと同一になるが、それなのに部分プログラム外の定数のスコープは異なる、ということになる。これは不整合である。以上のことから、定数は全てグローバルとした。スコープを限定した定数は存在しない。

さてここで問題になるのは、定数を部分プログラムの中で作ることを認めるか否かである。さらには繰り返しや条件実行も同様だ。つまり、インデントした行に定数定義を置くことを認めるかどうかである。

認める場合、一見ブロックスコープに見えてしまう定数がグローバルということになる。認めない場合、わざわざエラーメッセージを出さねばならない。

これに関しては、認めないこととした。ブロックスコープに見えてしまう、ということはやはり有害であろう。定数はインデント 0 の行でしか定義できないものとする。

2.1.8 IO の問題

IO はメモリ経由でのみ行う。画面に絵を描くのも、垂直同期を待つのも、キー入力を取るのも、全てメモリにマップされた IO を通す。近代的なコンピュータは API の覆いをはぎとればこうなっているわけで、それを直接見せる方が具体的でわかりやすかろう。「50000 番が 1 だったら上が押されてる」のようなコードを直接書く方が明らかに具体的である。「キーボードが 50000 番のダイアルを回しに来る」と言えば、API を覚える必要もなく極めて具体的である。

これに伴って、標準ライブラリの類は一切用意しない。ライブラリは用意するとしても、サンプル、あるいは慣れた者向けのおもちゃにすぎない。

2.1.9 バーチャルマシン仕様

Sunaba をどのように実行するかについては、インタープリタでもコンパイラでも良い。しかし、インタープリタでは速度を出しにくかろうし、言語仕様の変更が深いところまで

影響を与える懸念がある。

そこで、Sunaba ではバーチャルマシンの概念を採用し、このためのマシン語を吐いて、これを実行するようにした。これには、後に別の言語を作った時にコンパイラだけを作れば良いようにする、という意図もある。

バーチャルマシンはスタックマシンである。なぜレジスタマシンやアキュムレータマシンにしなかったかと言えば、それらになるとコンパイラの難度が高くなることが想像できるからである。実行速度の意味でもそれらの方が良いことは間違いないが、そう大したコンパイラを作れるわけでもないし、時間もない。そもそもそこは大して重要ではない。

スタックマシンで至れる程度の速度でも実用になるなら、それで良いのである。

演算ビット幅

演算のビット幅は 32bit である。

64bit としても良いのだが、まだ 64bit マシンの普及は完全とは言えないため、32bit としておいた。

レジスタ

レジスタは以下のものがある。

プログラムカウンタ 実行命令アドレスを指す
スタックポインタ スタック最上位アドレスを指す
フレームポインタ 現在実行中の部分プログラムが使うスタック開始位置を指す

3 本しかなく、状況を退避するのは容易である。デバグも容易い。

命令セット

また、命令セットは以下のようになっている。

i 即値プッシュ。31bit まで。
add 加算。スタックから 2 つ pop し、結果を push する。
sub 減算。スタックから 2 つ pop し、結果を push する。
mul 乗算。スタックから 2 つ pop し、結果を push する。
div 除算。スタックから 2 つ pop し、結果を push する。0 で割ると停止する。
lt 「より小さい」。スタックから 2 つ pop し、結果を push する。
le 「等しいか小さい」。スタックから 2 つ pop し、結果を push する。
eq 「等しい」。スタックから 2 つ pop し、結果を push する。
ne 「異なる」。スタックから 2 つ pop し、結果を push する。
ld pop して即値 (27bit) を加えたアドレスからロードして push する。

st pop して即値 (27bit) を加えたアドレスに、pop した値をストアする。
fld フレームポインタに即値 (27bit) を加えたアドレスからロードして push する。
fst フレームポインタに即値 (27bit) を加えたアドレスに pop した値をストアする。
j 即値 (26bit) にジャンプ
bz pop して 0 なら即値 (26bit) アドレスにジャンプ
call フレームポインタを push、プログラムカウンタを push、フレームポインタにスタックポインタを代入、即値 (26bit) アドレスにジャンプ。
ret 即値 (26bit) だけ pop。pop したアドレスにジャンプ、pop した値をフレームポインタに代入
pop 即値 (26bit) だけ pop。マイナスなら push。

命令セットは最小に近い。最小に「近い」、と歯切れが悪いのは、命令数を減らして高速化かつデバグを容易にするために、命令を加えた部分があるからである。

無条件ジャンプ、フレームポインタ相対ロードストアがそれだ。無条件ジャンプはなくとも良いし、フレームポインタ相対ロードストアは、フレームポインタをプッシュする命令があれば足りる。つまり、2 命令ほど無駄に多いことになる。

また、関数コールを全てコンパイル時にインライン展開してしまえば、call と ret が不要になり、フレームポインタも不要になる。全ての変数のアドレスがコンパイル時に明らかになるため、最適化も容易くなるし、速度も上がる。実際最初はそうだった。

しかし、それだと再帰ができない。また、そう大きくないプログラムでも書き方によってはプログラムサイズが莫大に膨れ上がることがある。

後述の理由からメモリ空間は小さく制限しているため、これはやむを得ない。

なお、ビット演算はない。コンピュータが 2 進で動いていることはコンピュータの本質ではないため、省いた。根本的に 2 進であることを意識することはなく、それを利用することもできない。

同様の理由で、除算も特別扱いしていない。プログラミング初心者が除算が遅いことを知る必要はない。それを知るべき時はもっと後である。

2.1.10 メモリ空間

メモリ空間の大きさは 60000+ ビデオメモリサイズで、100x100 であれば 70000 となる。

4 万までがヒープ領域 + プログラムロード領域、4 万から 5 万にスタック、5 万から 6 万までが IO、そして 6 万からがビデオメモリである。

なお、1 バイトは 32bit である。これもコンピュータの本質ではないし、そもそも Sunaba には型が単一サイズの整数しかないため、1 バイトを 8bit としても本質とは関係

ない不便をかけるだけとなる。

2.2 ユーザインターフェイス

使い勝手は極力良くなければならない。GUI の操作は標準的な windows アプリケーションから隔たるべきではない。

GUI の作成の手間から考えても、C#を用いるのが良かろう。

ただし、実行側は速度要求から C++ が求められるため、インターフェイスと実行側は TCP/IP あるいは C++/CLI などで間接的に通信すべきである。

2.3 コンパイラ

コンパイラは、一旦アセンブル命令の羅列を吐き出し、これをアセンブルしてバーチャルマシンのマシン語とする。こうすることで、言語の変更が及ぶ範囲を最小化している。

なお、pascal のように 1 パスでコンパイルできる言語仕様ではないため、プログラミングの難度を下げるために、パスを数多く分けている。

コメントの除去のみ、タブの置換のみ、文字置換のみ、字句解析のみ、のように多くの段階に分けることで、プログラミング難度を下げ、保守性を高めている。

ただし、多段にするとデバグ情報を保持する難度が上がる。これは今後の課題となる。

なお、言語は C++ である必要は必ずしもなく、C#でも良かったのだが、ここでは C++ で書いた。移植性を重視してのことである。

2.4 実行プログラム

実行速度は極力速い必要がある。そうなると、C++ で書かざるを得ない。

DirectX か OpenGL かについては、1.1 の範囲であれば OpenGL がコード量からも移植性からも良い。テクスチャの CPU 書き換えさえ可能であればそれで良く、描画は固定機能で良いからである。

2.4.1 発音機構

Sunaba には発音機構がある。波形データは読み込めず、正弦波発生器が 3 つあるだけとした。発音のタイミングでトリガーを引くだけのインターフェイスにし、アプリが処理落ちせずフレームカウントを時計として使える前提であれば、割り込みやスレッドは不要である。

なお、正弦波としたのは、矩形波や三角波では波形に関するパラメータが存在し得るからである。正弦波にパラメータはなく、後々思い悩む必要はない。

内部は XAudio2 を用いることとした。WASAPI にすると XP を捨てることになり、まだ時期尚早であろう。

2.5　デバグ機能

デバグ機能としては、メモリの状態を見たり、メモリを書き換えたり、一時停止ができたりはする。また、特定のメモリに Unicode を書き込むことで、メッセージウィンドウに文字を出すこともできる。

しかし、そこまでである。それ以上の機能は UI を汚すし、教育上有害でもある。なぜうまく動かないのかを考えるのも訓練のうちであり、デバグは一定以上支援してはならない。

本来で言えば、画面に点が描ける以上、それ以上の機能など全くなくても良いのである。その意味で、現状ある機能に関してもリリース時には隠す方が良いかもしれない。

第 3 章

実装

ここでは、各部の実装について細かいことを述べる。設計との線引きはさほど明らかではないため、「詳しいことはこちら」と考えて頂けば良い。

3.1 ユーザインターフェイス

3.2 コンパイラ

コンパイルは以下の段階を踏んで行われる。

- ファイル結合 (include の処理)
- タブの空白化
- 文字置換 (全角→半角等)
- コメント除去 (行番号不変)
- 字句解析
- ブロック構造決定
- 構文解析
- コード生成
- アセンブル

特徴的なのは文字置換で、半角に同じ文字があれば全角を半角に変換する。数字やアルファベット、プラスやマイナス、セミコロン、スペースなどは全角で書いてもここで半角に変換する。

コンピュータ初心者は全角と半角の区別がないことが多く、同じ変数であっても場所によって全角だったり半角だったりすることは容易に考えられる。そのようなものをエラーにしていては本質の学習に差し支えるだろう。フォントがプロポーショナルであったりするとなおさらこれは問題となるわけで、「固定幅のエディタで書く」という前提から捨て

てかかる必要がある。

3.3　実行プログラム

実行プログラムはバイナリのコードを受け取り、これを 1 命令づつ実行する。

実行を高速化するために、コードの実行は別スレッドにて行い、適宜メインスレッドの描画側と同期を取るものとする。

同期は頻繁にやるとオーバーヘッドが大きいため、規定の命令数を処理する単位でしか行わない。この命令数については調整が必要であろうが、マシンの速度によっても変わってくるため、多少小さ目に設定しておく必要がある。そうでないと十分なレスポンスが得られない。

3.3.1　発音機構

正弦波発生装置については、直接的に三角関数を計算せず、中に調和振動子シミュレータを用意することとした。

$$\ddot{x} = -kx$$

を、数値的に計算する。積分は forward-backward eular とした。すなわち、

```
a = (-k * x) + (-d * v)
v += a
x += v
```

として、更新した速度を即座に位置に足しこむ形式である。こうすることで安定化する。速度を優先して高次の積分法は用いていない。なお、k はバネ定数、d は減衰率である。

XAuido2 の SourceVoice は 1 つだけ用意し、3 つある調和振動子シミュレータの変位の和をクランプしてからデータを書きこんでいる。SourceVoice は起動直後から再生を開始し終了までそのままにしている。バッファ枯渇コールバックにてバッファ分の積分計算が行われ、このタイミングで発音要求が来ていれば、速度にパルスを加える。

この方法によって、三角関数の計算を要せず、ノイズもさして発生しない。SourceVoice を音の数だけ用意するよりも、XAudio2 のオーバーヘッドは小さい。

なお、1 チャネルあたりの調和振動子の数を増やせば、高次倍音まで入ってくるので豊かな音になるはずである。試してみると面白いかもしれない。

周波数→バネ定数変換

変換は以下の式で行う。

$$k = (\frac{2\pi f}{48000})^2$$

周波数に 2π を乗じれば角速度になるが、この周波数は秒におけるものなので、サンプル単位に変換する必要がある。Sunaba では 48kHz で決め打ちしているため、48000 で割れば良い。

調和振動子の質量は 1 としたため、この角速度を 2 乗したものが k となる。

振幅→初速変換

与える音の「強度」は振幅を指すが、これを初速に変換しないとパルスを与えられない。

単振動の式は、

$$x = A\sin(\omega t)$$

といった形式を取る。今時刻 0 で変位 0 としたため、サインのみとした。このような A を取るような初速 v0 が必要である。これを微分すると、

$$\dot{x} = A\omega\sin(\omega t)$$

となる。ここで、v0 は速度の最大値であり、すなわちこの振幅であるから、

$$v0 = A\omega$$

となる。角速度 ω は $2\pi f$ であるから、ユーザが与える秒における周波数を用いて、

$$v0 = \frac{2\pi F}{48000}$$

となる。これを初速として与えれば良い。

3.4 デバグ機能

内部的にはコールスタックの記録も行っているが、外には出していない。デバグ情報がないため、あるアドレスの命令が何と言う関数に属するのかがわからないためだ。これでは完全に役立たずである。

コンパイル時にデバグ情報を生成するようになるまでは、デバグに関係してできることはほとんどない。

第 4 章

課題

4.1 言語仕様

代入の書式はもっとわかりやすくできるのではないか。わかりやすい書式と短い書式の 2 つを用意すべきなのか、そうでないのか。

メモリマップはもっと小さくできる可能性がある。しかし、5 ケタ必要な段階で、今の作りでも悪くないかもしれない。現在のマップであれば 200x200 になっても VRAM の末尾が 5 桁で収まる。これくらいの解像度がおそらく限界であろう。また、自由領域を今より小さくすると、100x100 を単純に塗りつぶすようなプログラムが入らなくなる可能性がある。可能性があるとすれば自由領域を 10000 までに制限する程度だが、可能性を大きく制限する割に益が小さい。

50000 番台が広大すぎるきらいはあるが、3D 機能やネットワーク、サウンドなどを足す可能性はゼロではなく、空けておいて悪いことはない。

else と elseif は欲しいか。なければないで考えるので教育上は有益だが、あった方が楽だったり高速だったりする局面はそれなりにある。しかし、これがないことによって発生しやすいバグについては、教育的見地から言って必ずしも有害とは言えない。現在のところ、用意しない方向で良いかと思う。

4.2 ユーザインターフェイス

最大化への対応、iOS/Android/macOS への移植。

また、単体起動可能な exe を吐く機能もあって良い。ただしこれに関しては PC/mac のみとなるだろう。

インストーラがまだ出来ていない。しかし、そもそも作るべきかどうかも考えねばならない。ファイルをドラッグアンドドロップする、という操作がある段階で、サンプルのフォルダをエクスプローラで開かざるを得ないし、ファイルという存在を意識しない

ことはできないのである。下手にインストーラを用意するよりは、zip を展開して中にある exe を叩け、と言う方がわかりやすいかもしれない。インストーラを用意すればインストーラの操作説明が必要だし、アンインストールやバージョン更新の問題も出てくるからである。

多重起動を許していないのも場合によっては不便だ。これを許すには、通常のインターフェイスにおいては TCP/IP を使わないようにしないといけない。

4.3 バーチャルマシン

4.3.1 命令の追加

いくつか足すことで高速化すると思われる命令がある。

絶対アドレスロードストア 絶対アドレスが確定している場合に命令数を削減できる。IO などではこのケースはそれなりに多い。

直接メモリ即値演算 ロード、ロード、演算、ストア、の 4 命令を 1 命令に落とせるため、場合によっては効果が大きい。インクリメントのみ用意する、という対応でも良いが、美しさには欠ける。

演算付き分岐 各種演算に分岐フラグを持たせてしまえば、ロード、ロード、演算、分岐の 4 命令が、3 命令に削減できる。

即値付き演算 ロード、ロード、演算、ストア、の 4 命令を 3 命令に落とせるため、場合によっては有効である。しかし命令の種類が大きく増える。

メモリムーブ ロードとストアを統合して命令数を削る。しかし相対絶対で種類がかなり増える。

しかし、一番効果が高いのは、レジスタマシンにすることだろう。汎用レジスタを 16 本程度用意して適切に割り当てれば、理論的には劇的に命令数が減る。ただし、コンパイラは相当に複雑化するはずである。

4.3.2 関数ポインタ対応

関数コールでオペランドを取れるようにし、飛び先のアドレスを指定できるようにすれば、関数ポインタに対応できる。これをやっておかないと、将来オブジェクト指向言語を作った時に仮想関数を処理できない。

しかし、現段階では不要である。() さえつければどんなものでも関数コールになる、ということであり、それは初学者にとってはわかりにくすぎる。また、引数や戻り値のチェックも容易ではない。少なくとも型概念が入った言語でない限り、関数ポインタは導入すべきではない。

なお、入れる場合は、引数をとる call 命令を別に作るのが簡単であり、性能も落とさない。

4.4 コンパイラ

最適化が甘い。定数分岐最適化、死にコード消去、式変形、呼んでない関数の除去、定数をコピーしただけの変数の除去。

そのためには、構文木の段階での最適化を入れるべきだし、コード生成後にも最適化を入れるべきである。

それを言うなら、アキュムレータレジスタを足してアキュムレータマシンにすることで命令数を減らせるが、そこまでして高速化する必要はないだろう。即値入りの演算命令を足すのも同様の効果が見込める。

なお、エラーメッセージはわかりやすい方が良い。どのような間違い方があるかの具体例を多く試し、それぞれに対応して修正しやすいメッセージを出すべきである。

エラーが一つづつしか出ないのはこのままで良かろう。生産性は重要ではないため、一度にたくさんのエラーが出て混乱させるよりは一つづつ出す方が親切であろう。

4.5 実行側

また、命令のデコードを事前に行って即値を取り出しておくなどすればさらに高速化する。

また、10 進の色表現から 2 進の色表現に変換する部分が毎フレーム CPU にかかってくるため無駄が大きい。これは近代的なマシンであればシェーダでやるべきである。最低でもスレッドプールを立てて CPU 側でも分散した方が良い。

PC 版の標準アプリに関しては、TCP/IP を通さずに動く方が信頼性が上るし、レスポンスも良かろう。

メモリ空間サイズは設定可能にすべきか。すべきだとして、いかなる手段ですべきか。

4.5.1 発音機構

正弦波 3 チャネルでは表情が出ない。矩形波や三角波には高次倍音が含まれているが、正弦波にはそれがないからである。矩形波であれば 1 チャネルで良いような時でも、正弦波だと複数チャネル使わねばならなくなる。その意味で、チャネルは増やすべきかもしれない。

4.5.2　機能追加

2D アクセラレータ、3D アクセラレータ、ネットワークなどの機能を足すことは不可能ではない。

しかし、それはより高級な言語を用意してからの話だろう。

ただし、現状画面のクリアだけでも莫大な時間を要するため、おもちゃとして魅力的であるためには 2D アクセラレータくらいはあった方が良い可能性がある。

4.6　デバッグ機能

ローカル変数表示、コールスタック表示、ステップ実行、に関しては最低限欲しい機能である。メモリの値表示や書き換えについても UI として整備することが望ましい。ただし、現状これを可能にするためのデバグ情報の整備ができていない。設計から考え直す必要がある可能性がある。