2023/9/20 ver. 1.2

Yet 言語のマニュアル

目次

Yet 言語のバッカスナウア記法（BNF）..... 2
拡張子..... 2
コンパイルの流れ..... 2
インストール方法..... 3
コンパイル方法..... 3
実行方法..... 3
コメント..... 3
組み込み関数..... 4
Yet 言語のマクロ機能とインクルード機能、演算子定義..... 5
領域と変数..... 5
配列..... 6
Lisp 言語への変換機能..... 6
サンプルコード..... 7
演算子の定義..... 11
Yet 仮想環境..... 12
アセンブリ・機械語..... 13
Yet ヴィジュアル環境..... 14
Yet 言語での画像生成..... 15
参考..... 16

Yet 言語のバッカスナウア記法(BNF)

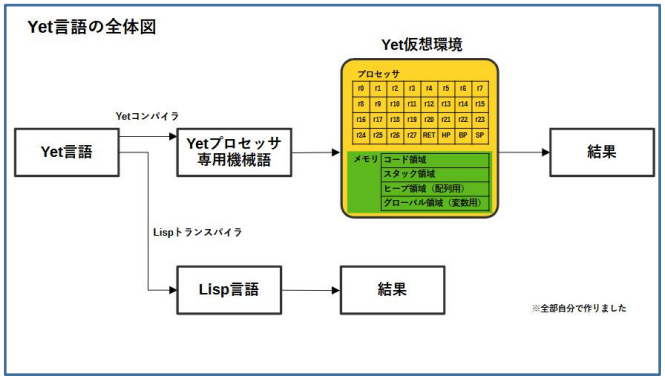
```
  digit       ::=”0” | ”1” | ”2” | ”3” | ... | ”8” | ”9”
  number      ::=digit | number digit
  letter      ::=”a” | ”b” | ... | ”z” | ”A” | ”B” | ... | ”Z”
  identifier  ::=letter | identifier letter | identifier digit |
identifier ”_”
  factor      ::=number | identifier {”(” parameter ”)”} | ”(” expression
”)” | block | array
  array_expr  ::=factor {(”@” | ”$”) factor}
  power       ::=array_expr {”^” array_expr}
  term        ::=power {(”*” | ”/” | ”%”) power}
  add_expr    ::=term {(”+” | ”-”) term}
  eq_expr     ::=add_expr {(”==” | ”!=” | ”<” | ”>” | ”<=” | ”>=”) add_expr}
  expression  ::=eq_expr {”=” eq_expr}
  suite       ::={expression ”;” {suite}}
  parameter   ::={expression {”,” parameter}}
  block       ::=”{” suite ”}”
  array       ::=”[” parameter ”]”
```

拡張子

Yet 言語のプログラムファイル　.yet
Yet 仮想機械語ファイル　.ye

コンパイルの流れ

コンパイラが受け取ったYetプログラムは字句解析器によって字句解析され、トークンに分割される。トークンは構文解析器に渡され、構文木になる。その構文木が仮想機械語へ変換される。



インストール方法

任意のディレクトリで zip ファイルを展開する。

```
$ unzip yet.zip
```

yet ディレクトリに移動する。

```
$ cd yet
```

mybin ディレクトリにパスを通す。

```
$ export PATH="$PATH:$(pwd)/mybin"
```

demo ディレクトリに移動する。

```
$ cd demo
```

コンパイルや実行を行う。

コンパイル方法

コンパイルは yetc コマンドで行うことができる。
-o オプションで出力ファイルを指定することができる。 指定しないと a.ye に出力される
-p オプションでアセンブリを画面出力
-L オプションで Lisp 言語に変換(-p オプションがついていた場合は Lisp 言語を画面出力)

```
$ yetc -o sample.ye sample.yet
$ yetc -p -o sample.ye sample.yet
$ yetc -pL -o sample.lisp sample.yet
```

実行方法

仮想機械語ファイルの実行は yet コマンドで行う。

```
$ yet sample.ye
```

コメント

'と'で囲まれた部分はコメントとなる。

組み込み関数

組み込み関数は write, read, return, if, declare, makearray の 6 つのみである。

write(n)	n の値を標準出力。
read()	標準入力を返す。
return(n)	関数内で使い、 n を戻り値として返す。
if(e, t, f)	条件式 e が 0 以外だったら t を実行、0 だったら f を実行する。 fは省略可。 tとfには{と} で囲われたブロックを渡さなければいけない。 (今後、式も使えるように改良する予定)
declare(v1, v2...)	関数内でのみ使用できるローカル変数v1、v2...を宣言する。 各関数の先頭の行で1度だけ使うことができる。
makearray(n)	要素数が n の配列を作成。

```
'sample code'
abs(x) = {
    declare(n);
    if(x < 0, {n = -1 * x;}, {n = x;});
    return(n);
};
main() = {
    write(abs(read()));
};
```

変数名や関数名を組み込み関数名にすることはできない。

Yet 言語のマクロ機能とインクルード機能、演算子定義

マクロ機能は文字列を数字に置き換える。プログラム中に

```
#define N 10
```

のように書く。行末のセミコロンは必要ない。

インクルード機能は指定した yet プログラムファイルをそのままその場所に埋め込む。

```
#include math
```

のように書く。行末のセミコロンは必要なく、yet 拡張子はつけない。

演算子定義は新しい演算子を定義する。

```
func(x, y) = x + y * 2;
#defoperator +* func 6 1
```

のように書く。演算子定義をするときは、
#defoperator [演算子名] [関数名] [演算子の優先順位] [左から計算する場合は 1，右から計算する場合は 0]
という構文で書く。詳しくは 11 ページを参照。

領域と変数

Yet 言語の関数以外の変数はすべて64bit 浮動小数点数である。 関数は変数であるが64bit 整数である。

変数はローカル変数とグローバル変数に大別される。 関数内でのみ有効なローカル変数はスタック領域に保存され、関数の外に出ると参照・代入ができなくなる。
グローバル変数はグローバル領域に保存され、プログラム中のどこでも使うことができる。
同じ名前のローカル変数とグローバル変数があった場合は、ローカル変数が優先される。

```
a = 3;
main() = {
    declare(b, c, d);
    b = a;
    c = a + 3;
    d = c + 5;
};
```

配列

Yet 言語の配列は次のように宣言する。

```
a = [0, 1, 2, 3];
b = makearray(5);
```

この場合、配列 a の指定したインデックスへの参照・代入は以下のように行う。(bの場合も同様である)

```
a@0;  '参照は@演算子を用いる'
a$1 = 5;  '代入は$演算子を用いる'
a@(n + 1); 'a の n+1 番目を参照'
```

配列の個々のデータはヒープ領域に保存され、 変数そのもの(上の例では a)はヒープ領域での配列の先頭の場所が格納されている。

Lisp 言語への変換機能

コンパイル時に Lisp トランスパイラによって構文木を Lisp 言語に変換(トランスパイル)することができる。 出力される Lisp 言語は Common Lisp の実行環境である roswell でテストを行っている。

Yet 言語と Lisp 言語の仕様の違いにより、Lisp 言語への変換に対応していない機能がある。例えば、高階関数の場合、Yet 言語では関数と数値の扱いが同じだが、Lisp 言語では別々に扱われているため、変換ができない。(現在対策中)

サンプルコード

n の階乗を求める関数 fact(n)を定義

```
'fact.yet'
fact(n) = n == 1? 1 : n * fact(n - 1);
```

x の n 乗を求める関数 pow(x, n)を定義

```
'pow.yet'
pow(x, n) = n == 0? 1 : x * pow(x, n - 1);
```

三角関数の sin x と cos x をテイラー展開で求める関数 taylor_sin(x, n)と taylor_cos(x, n)を定義する。nにはテイラー展開の項の数を指定する。

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \ldots$$

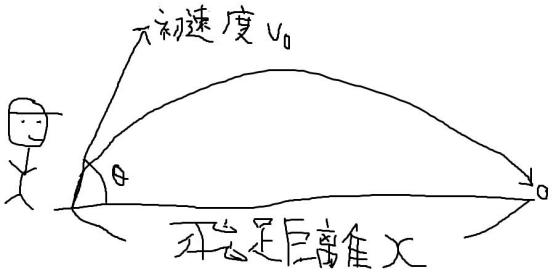
$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \ldots$$

```
'sankakukansu.yet'
#include fact
#include pow

item_sin(x, n) = {
    declare(m, s);
    m = 2 * n + 1;
    s = n % 2 == 0? 1 : -1;
    return(s * pow(x, m) / fact(m));
};
taylor_sin(x, n) = n == 0? item_sin(x, 0) : (item_sin(x, n) +
taylor_sin(x, n - 1));

item_cos(x, n) = {
    declare(m, s);
    m = 2 * n;
    s = n % 2 == 0? 1 : -1;
    return(s * pow(x, m) / fact(m));
};
taylor_cos(x, n) = n == 0? 1 : (item_cos(x, n) + taylor_cos(x, n - 1));
```

大谷選手が 0～90 度で時速 165km のボールを投げた時の飛距離を計算するプログラム



$$x = \frac{v_0^{\,2} \sin 2\theta}{g}$$

```
'throw.yet'
#include sankakukansu
#define g 9.80665 '重力加速度'
#define v0 45.83 '165km / 3600s'
#define notch 1
#define pi 3.1415926535

position_x(theta) = pow(v0, 2) * taylor_sin(2 * theta, 30) / g;
angle_range(start, end) = {
    write(start);
    write(position_x(start/180*pi));
    if(start > end - notch, {return(0);}, {angle_range(start + notch,
end);});
};
main() = {
    angle_range(0, 90);
};
```

フィボナッチ数列の n 番目の数を求めるプログラム

$$F_n = \begin{cases} n & (n < 2) \\ F_{n-1} + F_{n-2} & (n \geq 2) \end{cases}$$

通常の再帰

```
'fib.yet'
fib(n) = n < 2? n : fib(n - 1) + fib(n - 2);
main() = {
    write(fib(read()));
};
```

メモ化再帰

```
'fibmemo.yet'
fibarray = makearray(40);
init_array(n, x) = {
    fibarray$(n - 1) = x;
    if(n == 1, {return(0);}, {init_array(n - 1, x);});
};
fib(n) = {
    declare(f);
    if(fibarray@n == -1, {f = fib(n - 1) + fib(n - 2); fibarray$n = f;
return(f);}, {return(fibarray@n);});
};
main() = {
    init_array(40, -1);
    fibarray$0 = 0;
    fibarray$1 = 1;
    write(fib(read()));
};
```

高階関数の使用例

```
'higher_order_demo.yet'
add(a, b) = a + b;
print(f) = {
    write(f(3, 5));
};
main() = {
    print(add);
};
```

関数ともととなる配列、結果が代入される配列、配列の長さを受け取りもととなる配列の各値を計算して結果が代入される配列に代入する map 関数を定義

```
'map.yet'
map(f, a1, a2, len) = {
    a2$(len - 1) = f(a1@(len - 1));
    if(len == 1, {return(0);}, {map(f, a1, a2, len - 1);});
};
double(a) = a * 2;
main() = {
    array1 = [3, 5, 6];
    array2 = makearray(3);
    map(double, array1, array2, 3);
    write(array2@0);
    write(array2@1);
    write(array2@2);
};
```

演算子の定義

演算子の役割と優先順位、どちらから計算するか（a は左辺, b は右辺を表す）

優先順位が高い演算子が先に計算される。例えば、

3 + 4 * 5 → 3 + (4 * 5)

左から計算する場合は

1 + 2 + 3 → (1 + 2) + 3

となり、右から計算する場合は

1 ^ 2 ^ 3 → 1 ^ (2 ^ 3)

となる。

演算子	役割	優先順位	どちらから計算するか
=	a に b を代入	1	右
?	三項演算子	2	左
==	a と b が等しかったら 1，違ったら 0	3	左
!=	a と b が等しくなかったら 1,等しかったら 0	3	左
<	a<b だったら1，違ったら 0	3	左
>	a>b だったら1，違ったら 0	3	左
<=	b が a 以上だったら1，違ったら 0	3	左
>=	b が a 以下だったら1，違ったら 0	3	左
+	a + b	4	左
-	a − b	4	左
*	a × b	5	左
/	a ÷ b	5	左
%	a を b で割った余り	5	左
^	a の b 乗（小数も可）	6	右
@	配列 a の b 番目	7	左
$	配列 a の b 番目 （要素に代入する場合）	7	左

Yet 仮想環境

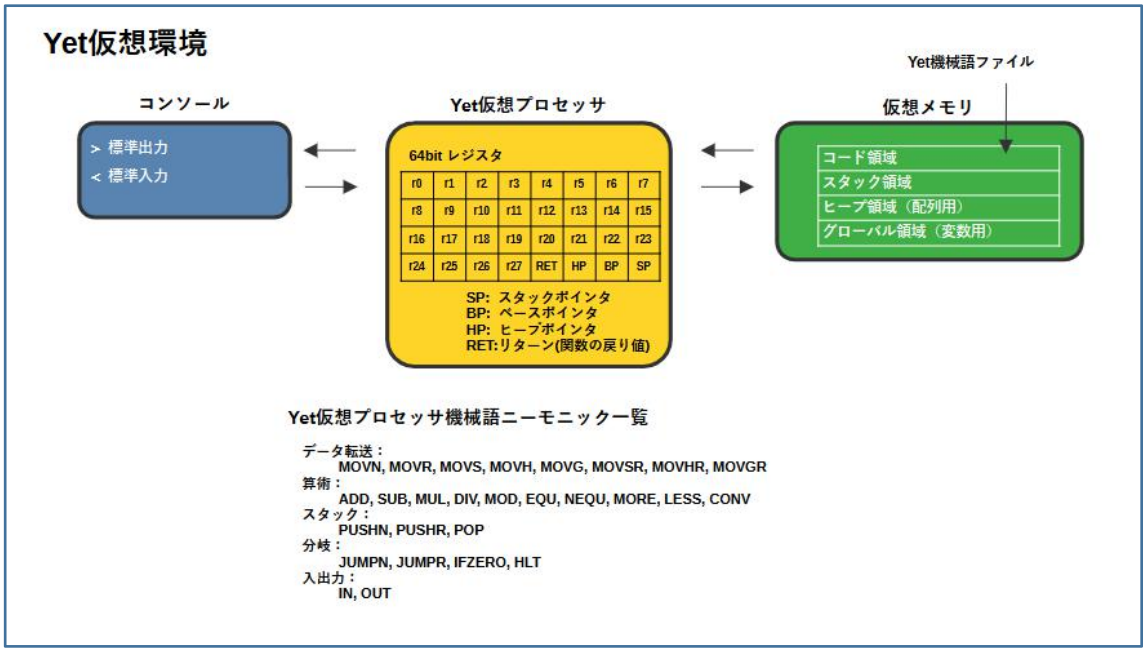
Yet 仮想環境は、 32 個の64bit レジスタを備えた仮想プロセッサと4つのメモリから構成される Yet 機械語の実行環境である。

レジスタの役割一覧（レジスタは「r レジスタの番号」として表記する。 例：r0）

レジスタ名	別名	役割
r0～r27	なし	汎用レジスタ：主に計算など
r28	RET	リターンレジスタ：関数の戻り値を保存
r29	HP	ヒープポインタ：動的領域の管理（未実装）
r30	BP	ベースポインタ：SPの保存、ローカル変数、引数の管理
r31	SP	スタックポインタ：スタックの管理

メモリの役割一覧

メモリ名	役割
コード領域	機械語を保存
ヒープ領域	配列を保存
グローバル領域	グローバル変数を保存
スタック領域	引数やローカル変数などの保存



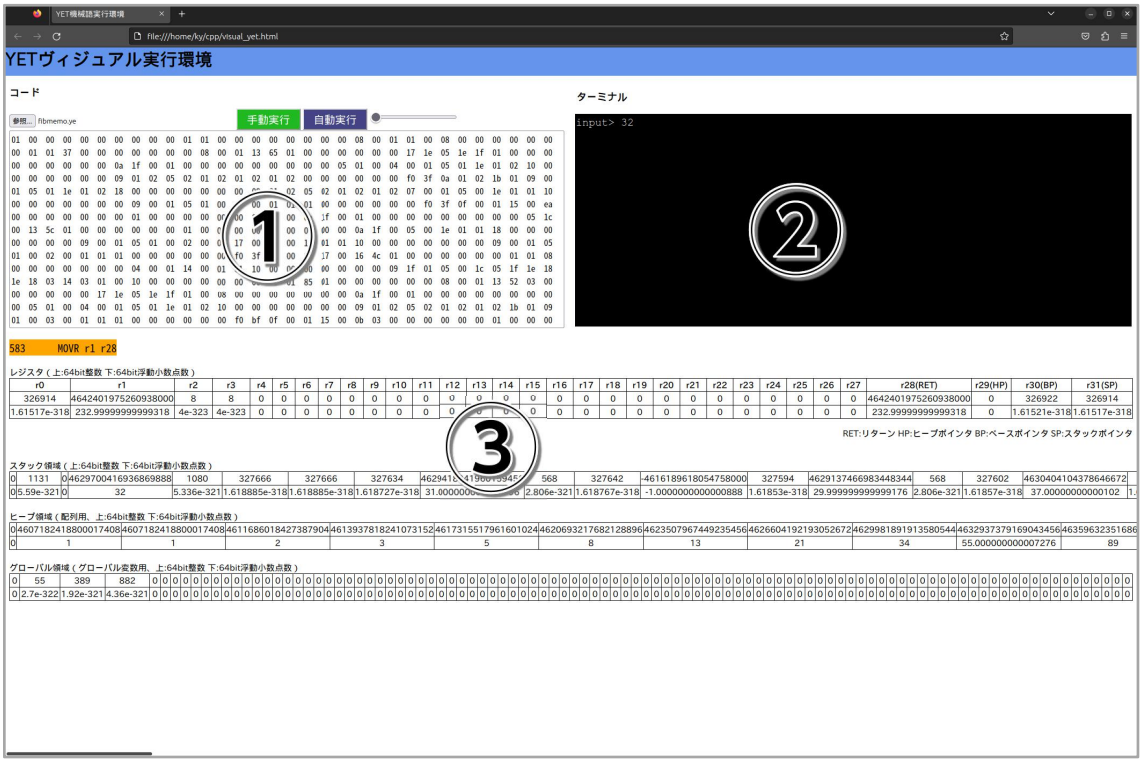
アセンブリ・機械語

命令一覧

機械語	アセンブリ	機能
1	MOVN r1 num	r1 に数値 num をコピー
2	MOVS r1 r2	r1 にスタック領域のr2番目の値をコピー
3	MOVH r1 r2	r1 にヒープ領域のr2番目の値をコピー
4	MOVG r1 r2	r1 にグローバル領域のr2番目の値をコピー
8	MOVR r1 r2	r1にr2をコピー
5	MOVSR r1 r2	スタック領域のr1 番目にr2をコピー
6	MOVHR r1 r2	ヒープ領域のr1 番目にr2をコピー
7	MOVGR r1 r2	グローバル領域のr1 番目にr2をコピー
9	ADD r1 r2	r1 ＋ r2 → r1
10	SUB r1 r2	r1 － r2 → r1
11	MUL r1 r2	r1 × r2 → r1
12	DIV r1 r2	r1 ÷ r2 → r1
13	MOD r1 r2	r1 を r2 で割った余り → r1
14	EXP r1 r2	r1 のr2乗 → r1　　（使えますがずるいので非推奨としておきます）
15	EQU r1 r2	r1＝r2なら1, それ以外は0 → r1
16	NEQU r1 r2	r1＝r2なら0, それ以外は1 → r1
17	MORE r1 r2	r1＞r2なら1, それ以外は0 → r1
18	LESS r1 r2	r1＜r2なら1, それ以外は0 → r1
19	JUMPN num	コードポインタを 64bit 整数 num に移動
20	JUMPR r1	コードポインタを r1 に移動
21	IFZERO r1 num	r1=0 だったらコードポインタを64bit 整数 num に移動
22	PUSHN num	スタックに数値 num をプッシュする
23	PUSHR r1	スタックに r1 をプッシュする
24	POP r1	スタックをポップし r1 にコピー
25	OUT r1	r1を 64bit 浮動小数点数として標準出力
26	IN r1	標準入力を 64bit 浮動小数点数としてr1にコピー
27	COMV r1	r1を64bit 浮動小数点数から 64bit 整数に変換
28	HLT	終了させる

Yet ヴィジュアル環境

機械語ファイルを実行することができ、レジスタと仮想メモリの状態がリアルタイムで分かるようになっている Web 上で動く実行環境である。



① 参照ボタンでファイルを指定できる。コード領域をバイト単位で見ることができる。
手動実行ボタンを押すと1命令ずつ実行する。
自動実行ボタンを押すと指定した間隔ずつ命令を実行する。
スライダーを動かすことで自動実行の間隔を指定できる。
今実行されている命令のアドレスと内容がオレンジ色で分かるようになっている。

② 標準出力された実行結果を表示する。

③ レジスタと仮想メモリの状態をリアルタイムで表示する。
表の上の行は 64bit 整数、下の行は64bit 浮動小数点数が表示されている。

Yet 言語での画像生成

画像生成は yetd コマンドを使うことでできる。svg 画像を生成する。
-o オプションで出力ファイルを指定できる。指定しない場合は a.svg となる。

```
$ yetd -o aaa.svg testshape.ye
```

yet 実行ファイルを指定するだけで生成できる。

画像を生成するには shape.yet をインクルードしなければいけない。描くことができるのは、直線、円、四角形、三角形のみである。
関数とその引数はそれぞれ、
直線：line(x1, y1, x2, y2,c1,2,c3)
円：circle(x,y,r,c1,c2,c3)
四角形：rect(x,y,width,height,c1,c2,c3)
三角形:tri(x1,y1,x2,y2,x3,y3,c1,c2,c3)
である。x1,x2,y1 はそれぞれ1つ目のｘ座標、2つ目のｘ座標、1つ目のｙ座標を表している。
c1,c2,c3は RGB 値の1番目（赤）、2番目（緑）、3番目（青）である。
また、set_size 関数で画像の大きさを指定できる。指定しなかった場合は 100x100 ますとなる。
set_size(x,y)

```
’testshape.yet’
#include shape

main() = {
    set_size(500, 500);
    circle(250, 200, 100, 0, 0, 0);
};
```

参考

書籍

照井一成 『コンピュータは数学者になれるのか？』

千葉滋 『スクリプト言語の作り方』

上原周 『作ろう！CPU』

Peter Farrell『Python ではじめる数学の冒険』

Toby Segaran 『集合知プログラミング』

増井敏克 『プログラミング言語図鑑』

株式会社アンク 『HTML/CSS の絵本』

株式会社アンク 『JavaScript の絵本』

WEB サイト

x86 アセンブリ言語での関数コール
https://vanya.jp.net/os/x86call/

C 言語で Taylor 展開で sin x 求めたい
https://note.com/electrical_cat/n/n5c5cc6ab2df9

高校数学の美しい物語 斜方投射の公式の導出と飛距離を伸ばす方法
https://manabitimes.jp/math/1097

フィボナッチ数を扱う - 再帰呼出し
https://blog.y-yuki.net/entry/2018/08/20/094000