Whitespace 超入門

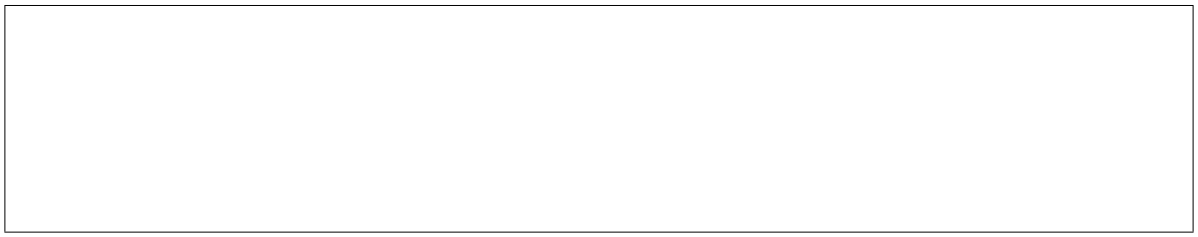
すしす (Twitter: @susisu2413 / GitHub: susisu)

* 万が一これを読んでも入門できなかった場合は `https://susisu.github.io/wspace/wspace.pdf` を読んでください.

1 Whitespace とは

Whitespace [*1] [*2] は難解プログラミング言語 (esolang) と呼ばれる言語のひとつで, 半角スペース ␣, タブ文字 ⇥, 改行文字 (LF) のみでプログラムを記述します. プログラム中の ␣, ⇥, 以外の文字はすべてコメントとして扱われます.

雰囲気を知ってもらうためには, 実際にプログラムを見てもらうのが早いでしょう. 以下は定番の “Hello, world!” を出力するプログラムです. **改行文字も意味をもっています**ので, 末尾の改行なども忘れないように注意してください.



どうでしょうか? このような (一見奇妙な) 記法で何が書けるのだろうと思われるかもしれませんが, Whitespace は**チューリング完全**な, つまり, C++ や Brainf*ck, Lazy K などと同等の計算能力を持っている言語であり, かつ (難解プログラミング言語の中では) 言語自体の機能もかなり豊富なので, 様々なプログラムを Whitespace でも記述することが可能です.

また, Whitespace ならではの利点として, 文芸的プログラミングが容易である (単にプログラム中に直接コメントを書けば良い) ことや, 逆にわざと間違ったコメントを書くことで難読化したり, 別のプログラム中に Whitespace のプログラムを隠蔽したり [*3], プログラムを印刷して元のファイルを消してしまえば解読が困難になるなどがあります. 中でも私が特に Whitespace をおすすめする理由として, 例えばあなたが真に驚くべきプログラムを思いついたときに, もしそれを書き留めておく余白が無ければ, 何を思いついたのか忘れてしまうかもしれません. Whitespace ならば, 上の Hello, wolrd! の例 (実は 48 行ある) を見ても分かるように紙上のあらゆる余白に好きなだけ書くことが出来るので, こういった場合に非常に役に立ちます.

猫が一見何もない方向を見つめる (フェレンゲル-シュターデン現象) のも, 実は壁に書かれた Whitespace のプログラムを読んでいるのだという説もあります [要出典].

この記事では, Whitespace でプログラムを書いたり, プログラムを読んだりするために必要な言語仕様を, 例を交えながら解説します. 実際にプログラムを試してみたい場合は, 本家サイトなどにあるインタプリタを用いてください. もしくは, この記事の執筆に当たり, Node.js でインタプリタを作成しました[*4] ので, よければ使ってやってください. Whitespace のソースファイルの拡張子は `.ws` にするのが良さそうです.

[*1] 本家サイト (記事執筆時点では閲覧不可) `http://compsoc.dur.ac.uk/whitespace/`

[*2] `https://esolangs.org/wiki/Whitespace`

[*3] `https://github.com/uhyo/i-challe-presentation/`

[*4] `https://github.com/susisu/Whitespace-JS`

2 Whitespace の言語仕様

Whitespace はスタック指向の手続き型の言語で, プログラムは命令の列からなっており, スタック上の値を操作したり, ヒープの値を読み書きする命令と, フロー制御の命令の組み合わせによってプログラムを記述します.

各命令は IMP コマンド (引数) の形をしています. IMP (instruction manipulation parameter) は命令の種類を表すタグのようなもので, 「スタック操作」「数値演算」「ヒープアクセス」「入出力」「フロー制御」の五種類があり, 例えば, スタック操作の命令は, すべて同じ IMP　で始まり, その後に具体的な操作を表す部分 (コマンド) が続きます.

2.1 スタック操作

スタック操作の IMP は　です. 新たに値を積む, 値を交換するなど, スタックに関する操作を行います.

2.1.1 <INTEGER>

スタックに新たに整数値 (本家の実装では任意精度) を積みます. <INTEGER> の部分には引数として整数を, 符号 (　= 正,　　= 負) の後に, 二進数で絶対値 (　= 0,　　= 1), 最後に終端を表す
という順番で記述します.

以下はスタックに値を積むプログラムの例です. [] は空のスタック, あるいはスタックの底を表していて, 命令によって新たに積まれた値は左から順にコロン : 区切りで書いています. 命令の前後のコメントは, その命令が実行される前後のスタックの状態を表しています. 改行が意味を持っているので, コメントとコードが入り乱れていて少々見にくいですが, これは回避できないので許してください.

```
[]
6:[]
-10:6:[]
```

2.1.2

スタックの一番上にある値を複製します. ほとんどの命令はスタックから値を読むときに上の値を取り除くので, 後で他の命令でも値を使う場合などに用います.

```
1:[]
 1:1:[]
```

2.1.3

スタックの上 2 つの値を交換します. 後ほど説明する数値演算で, 例えば既にスタック上にある値を, 新たにスタックに積んだ値から引きたい, などの際に便利です.

```
1:2:[]
        2:1:[]
```

2.1.4

スタックの一番上にある値を捨てます. それだけです.

```
1:2:[]

2:[]
```

2.1.5 <INTEGER>

整数 n を指定して, スタックの上から n 番目にある値をコピーして, スタックの一番上に積みます. なお, スタックの一番上は 0 番目ですのでご注意ください.

```
1:2:3:[]
2:1:2:3:[]
```

2.1.6 <INTEGER>

整数 n を指定して, スタックの一番上の値はそのまま残しつつ, 二番目以下を上から n 個削除します.

```
1:2:3:4:[]

1:4:[]
```

2.2 数値演算

数値演算の IMP は です. スタック上の値に関する計算を行います.

2.2.1

スタックの上から二つの値を取り出して足し算をし, 結果をスタックの一番上に積みます.
以下は実際にスタック上に値を二つ積んでから, それらの和を計算する例です.

```


2:7:[]              7+2=9:[]
```

2.2.2

スタックの上から二つの値を取り出して引き算をし, 結果をスタックに積みます.

以下は足し算の場合と同じように, 実際にスタック上に値を二つ積んでから, それらの差を計算する例です. **先にスタックに積んだ方が左の項**となることに注意してください.

```
2:7:[]                    7-2=5:[]
```

順番を逆にしたい場合は適宜
を使いましょう.

```
2:7:[]
    7:2:[]                    2-7=-5:[]
```

2.2.3

掛け算です.

```
2:7:[]
7*2=14:[]
```

2.2.4

割り算です. 結果は実数ではなく整数です.

```
2:7:[]                    7/2=3:[]
```

2.2.5

剰余 (割った余り) を計算します.

```
2:7:[]                        7mod2=1:[]
```

2.3 ヒープアクセス

ヒープアクセスの IMP は　　です. ヒープ領域 (永続的なメモリ) に値を書きこんだり, 値を読み取ったりします.

2.3.1

スタックから値とアドレスを取り出し, ヒープ上のアドレスの位置に値を書き込みます. **先にスタックに積んだ方がアドレス**ですので, これまたご注意ください.

以下の例ではヒープのアドレス 10 の位置に 2 を書き込んでいます. ヒープは {...,address:value,...} のようにアドレスの後に値を書いて表しています.

```
2:10:[],{}                    [],{10:2}
```

2.3.2

スタックからアドレスを取り出し, ヒープ上のそのアドレスにある値を読み取り, スタックの一番上に積みます.

今度は逆に, 先ほどヒープに書きこんだ値を読み取ってみます. ヒープ上の値は, スタックとは違い, 読み取った後もそのまま残ります.

```
10:[],{10:2}                          2:[],{10:2}
```

2.4 入出力

入出力の IMP は
です.

2.4.1

スタックから値を取り出し, その値 = 文字コードに対応する文字を出力します.

```
64:[]
  [], 出力->"@"
```

2.4.2

スタックから値を取り出し, その整数値を文字列に変換したものを出力します.

```
64:[]
        [], 出力->"64"
```

2.4.3

スタックからアドレスを取り出し, 入力から一文字読み取り, ヒープ上のアドレスの位置に読み取った文字の文字コードを書き込みます.

```
10:[]
        入力->"@",[],{10:64}
```

2.4.4

スタックからアドレスを取り出し, 入力から**一行**読み取り, それを整数に変換して, ヒープ上のアドレスの位置にその整数を書き込みます.

```
10:[]
                    入力->"64\n",[],{10:64}
```

2.5 フロー制御

フロー制御の IMP は
です. プログラム中の別の位置にジャンプする, プログラムを終了するなど, プログラムのフローを制御します.

2.5.1 <LABEL>

プログラム中の現在の位置にラベルを設定します. <LABEL> の部分にはラベルを, 二進数 (= 0, = 1) の後に終端を表す
という順番で記述します. このラベルは**静的**なもので, 現在位置より後にあるラベルも参照することも可能です.

2.5.2

<LABEL>

指定したラベルの位置にジャンプします. ループを作成したりする場合に役に立ちます.

```
2:6:[]

ラベル 1 へジャンプ (                    ) 内の引き算は実行されない

ラベル 1
        6:[], 出力->"2"
```

2.5.3 <LABEL>

スタックから値を取り出し, 0 であったとき, 指定したラベルにジャンプします. 0 でない場合は何も起こりません.
は条件分岐を行うための命令の一つです. 以下の例は数値の入力を受け取り, 0 であれば後続の処理にジャンプし, それ以外であれば対応する文字を出力して再度入力を待つプログラムです.

```
ラベル 0
0:[]
 0:0:[]
                    入力->n,0:[],{0:n}                          n:[],{0:n}
 n:n:[],{0:n}
```

```
n:[],{0:n}, 入力が 0 ならばラベル 1 へジャンプ
          [],{0:n}, 出力->n
'\n':[],{0:n}
  [],{0:n}, 出力->'\n'


ラベル 0 へジャンプ

ラベル 1
```

2.5.4 <LABEL>

スタックから値を取り出し, それが負であったとき, 指定したラベルにジャンプします. 0 または正であった場合は何も起こりません.

はジャンプする条件以外は と同じですので, 例は省略します.

2.5.5 <LABEL>

現在の位置をコールスタックに積んでから, 指定したラベルの位置にジャンプします. 後に説明する を用いることで, 再度元の場所に戻ってくることが可能です.

2.5.6

コールスタックの一番上の位置に処理を戻します.

と組み合わせて使用することで, サブルーチンの呼び出しといった処理を行うことが可能です. 以下の例では三つの数の足し算を行うサブルーチンを作成しています. スタックに引数となる値を積んでから でサブルーチンを呼び出し, サブルーチン内では処理を行って結果をスタックに積んでから, で元の位置に処理を戻します.

```
1:[]
2:1:[]
3:2:1:[]

ラベル 0 のサブルーチンを呼び出す,6:[]
          [], 出力->"6"


ラベル 1 へジャンプ

ラベル 0(サブルーチン),a:b:c:[]          b+a:c:[]          c+b+a:[]

処理を戻す

ラベル 1
```

2.5.7

プログラムをその場で終了します. すべてのプログラムは (少なくとも本家の実装では) 必ず

で終了する必要があります.

```
終了, これ以降は実行されない
1:[] とはならない
        [], 出力->1 もない
```

3 “Hello, world!” の解説

ここでは一番最初に挙げた “Hello, world!” のコードを解説します. このコードはヒープへのアクセス, スタックの操作, ループ, 出力など, 一般的なプログラムでもよく使うような処理を一通り含んでいます.

先頭から順番に見ていきましょう.

```
{0:'H'}

{0:'H',1:'e'}

{0:'H',...,2:'l'}

{0:'H',...,3:'l'}

{0:'H',...,4:'o'}

 {0:'H',...,5:',}

{0:'H',...,6:'\s'}

{0:'H',...,7:'w'}

{0:'H',...,8:'o'}

{0:'H',...,9:'r'}

{0:'H',...,10:'l'}
```

```
{0:'H',...,11:'d'}

{0:'H',...,12:'!'}

{0:'H',...,13:'\n'}
```

この部分では, ヒープのアドレス 0 から 13 までの領域に “Hello, world!“n” という文字列を配置しています. (コメントの \s と \n はそれぞれ半角スペースと改行 (LF) を表しています.)

次の部分では繰り返し処理 (C 言語などの for 文に近いこと) を行っています. まずは以下のようにカウンタをでスタックに積んで初期化します. for 文で言うと () 内の一番目の部分です.

```
i=0:[]
```

次に

でループの先頭にラベルをつけます.

```
ラベル 0
```

そしてその次に, ループの終了条件の判定を行います. for 文なら () 内の二番目の部分です.

や

を用いて, カウンタが 13 より大きければ, ループを抜けるように後ろのラベルへジャンプします.

```
 i:i:[]
13:i:i:[]
        i:13:i:[]                    13-i:i:[]

i:[], もし 13-i<0 ならラベル 1 へジャンプ
```

続いてループ処理の本体, for 文では {} 内の部分です. カウンタの位置の文字をヒープから

で読み取り,

で出力しています.

```
 i:i:[],{...,i:c,...}                          c:i:[],{...,i:c,...}
  i:[], 出力->c
```

最後にカウンタを 1 増やした後,

でループの先頭のラベル 0 にジャンプして, ループ部分は終了です. カウンタを増やすのは for 文の () 内の三番目の部分ですね.

```
1:i:[]              i+1:[]
```

ラベル 0 **へジャンプ**

ループの後には脱出のためのラベル 1 を書いておきます.

ラベル 1

最後に,

でプログラムを終了します.

終了

いかがでしょう. 最初はその記法に驚いたかもしれませんが, よくよく中身を見てみると今度はその普通さに驚いたのではないでしょうか.

4 まとめ

この記事を読んだ皆さんは無事 Whitespace に入門できたことと思います. これからは文芸的プログラミングを楽しんだり, プログラムを暗号化したり, 狭い余白を有効活用するなど, どんどん便利に Whitespace を使っていきましょう.

おや, こんなところにちょうどいい余白が...