メタプロ班後期活動報告書:Rust

西見元希

目 次

1　背景

昨年 11 月頃に Rust に入門してみたところ、非常に高速で安全な言語機能と引き換えに多少冗長な書き方を要する部分があった。しかし Rust にはそれを補うための高級なマクロ機能があり、これを利用したメタプログラミングによって安全性を損なわずに快適にコードを書くことができる。そこで私は Rust のマクロ機能についての海外の文献をまとめ、実際に競技プログラミングにおいて広く利用されているマクロを解析した。

2　Rust の基礎知識

Rust は基本的には OCaml や Standard ML に似た式指向言語であり、それでいてシンタックスは C++などのシステムプログラミング言語に近いという特徴を持つ言語である。

2.1　let 文

let 文は名前を値に束縛する文である。 OCaml などと同様にパターンを用いた束縛が可能。

```
let x = 3;
let (a, b) = (1, 2);
```

また、let 文で宣言した変数は**不変値**となるため、値の変更をすることができない。可変な変数として宣言するには let の後に **mut キーワード**を指定する必要がある。

```
let mut x = 0;
x = 5;
```

let で束縛する名前が既にスコープ内に存在する場合、Rust は以前の束縛を消去し、新しく束縛し直す。これを**シャドーイング**と呼ぶ。シャドーイングは代入とは異なるのでイミュータブルな変数でも事実上その値を変えることができる。

```
let x = 3;
let x = 5;//x=>5
```

2.2　スコープと所有権

Rust は一般的なレキシカルスコープを採用していて、スコープ内に環境を持っている。

```
let x = 4;
{
    let x = 10;
    println!("x={}",x);//=> x=10
}
```

Rust ではメモリ安全性を保つため、スコープを抜けた変数の値は自動的に破棄される。

```
{
    let s = String::from("hello");
    //s はここでは有効
}
//s はここでは無効、String の"hello"は解放されている。
```

関数に引数を渡すと値の所有権は仮引数の変数に移動するので、関数が終了すると値は破棄されてしまう。また、関数が値を返すとその値の所有権は関数の呼び出し元に移動する。

```
fn  main() {
    let x = String::from("hello");
    let y = String::from("world");
    let s = plus_str(x, y);//=>x,y の所有権を plus_str に渡している
    println!("{}", s);
    println!("{}", x);//=>x は plus_str で破棄されているので使用できない
}

fn  plus_str(x: String, y: String) -> String {
    let s = format!("{} + {}", x, y);
    s
}
```

2.3 借用

所有権を渡す代わりに参照を引数として渡すことができる。これを**借用**という。例えば String の参照は&str 型である。

```
fn  main() {
    let x = String::from("hello");
    let y = String::from("world");
    let s = plus_str(&x, &y);
    println!("{}", s);
    println!("{}", x);
}

fn  plus_str(x: &str, y: &str) -> String {
    let s = format!("{} + {}", x, y);
    s
}
```

借用は変数として宣言することもできる。

```
let mut s = String::from("hello");
let t = &s;//s の不変借用の生成
println!("{}",t);//=>hello
```

上の例で t を宣言したが、t は不変借用であるため t を通して s を変更することは許されていない。可変借用を生成するには&ではなく&mut をつける必要がある。

```
let  mut s = String::from("hello");
let t = &mut s;//s の可変借用の生成
t.clear();//t の参照する値を""に変更する
```

2.4 ライフタイム

参照先の値がスコープを抜けて解放されているにも関わらず参照が生きていると、C 言語などでは未定義動作を行ってしまう。これを**ダングリング参照**と呼ぶ。以下のコードはダングリング参照の例である。

```
{
    let r;
    {
        let x = 5;
        r = &x;
    }
    println!("r: {}",r); //=>r はスコープの抜けた x への参照であるためコンパイルできない。
}
```

これを避けるために、Rust には参照の**ライフタイム** (生存期間) という概念があり、参照が指す値よりも早くスコープを抜けることを示すことができる。上の例では、Rust の借用精査機によって r のライフタイムは'a、x のライフタイムは'b で'a>'b であると推論される。コンパイル時にライフタイムが'a の変数 r が' b のライフタイムのメモリを参照しようとしていることを検知し、エラーを出すという仕組みになっている。場合によっては推論が上手くいかないためにライフタイムを明示しなければならないときもあるが、高度な内容となるため解説は省略する。

3 マクロの体系的な理論

ここでは、Rust におけるマクロプログラミングを解説した文献である The Little Book of Rust Macros の前半の内容を翻訳し、簡単にまとめている。

3.1 基礎知識

マクロが展開されるタイミングは言語によって異なるので、マクロについて知る前に Rust のコードがどのようにコンパイルしているかを知る必要がある。コンパイルの最初の段階は **tokenisation** と呼ばれ、ソースのテキストをトークンに切り分ける。 Rust のトークンには次のような種類がある。

- 識別子: foo, name, x, self, ...
- 整数:42, 0, ...
- キーワード: fn, _, self, match, ...
- ライフタイム指定子:'a, 'b, ...
- 文字列: “”, “abc”, ...
- 記号:[, :, ::, @, ->, ...

注意点

- self はキーワードでありながら識別子になりうることがある
- ::は独立した記号であり:が 2 つ並んだものではない

C や C++の define マクロはこの段階で作用するが Rust は違うことを留意しておく。

次の段階は**パース**であり、切り分けられたトークンを用いて**抽象構文木** (AST) を構築する。 Rust のマクロはトークンツリーと AST の違いを理解して書く必要がある。 Rust のマクロ処理はこの AST の構築後に行われる。

3.2 マクロ展開

AST の構築が終わると、コンパイラは意味解析を初める前にすべてのマクロを展開後の式に置き換える。マクロがマクロを含む定義になっていた場合でも全て展開される。ただし再帰的なマクロ展開はデフォルトだと 32 回までとなっており、それ以降の展開にマクロ呼び出しが含まれていた場合、コンパイルエラーとなる。この制限は`#![recursion_limit="..."]` 属性を利用して上げることができるが、クレート全体に作用してしまうため、デフォルトが推奨される。

展開後の式の結果は次のいずれかになる。

- expression
- パターン
- 0 個以上のアイテム
- 0 個以上の impl アイテム
- 0 個以上の文

(アイテムは関数や構造体、モジュールなどの比較的大きなスコープを指す。) ここに無いものを展開後の結果とするマクロを書くことはできない。

3.3 macro_rules!

macro_rules!でマクロ定義ができる。構文は次の通り。

```
macro_rules! $name {
    $rule0 ;
    $rule1 ;
    //...
    $ruleN ;
}
```

rule の項は最低 1 つは記述する必要があり、最後の項のセミコロンは省略することができる。 rule の項は次のように記述する。

```
    ($pattern) => {$expression}
```

実際には () や{}は別のグループを表す記号でも構わないが、慣用的にこの 2 つが用いられる。

macro_rules!呼び出しは実はマクロ展開されていない。 AST には何も表示されず、コンパイラにマクロを登録している。そのため、呼び出しが有効な位置ならどこでも macro_rules!を利用することができる。

3.4 Matching

複数の rule はパターンマッチと同じく上から順に見ていき、マッチした式に展開する。どの rule にもマッチしなかった場合エラーを返す。 empty パターンの単純な例

```
macro_rules! four {
    () => {1 + 3};
}
```

このパターンに合致するマクロ呼び出しは four!(), four[], four!{}のみになる。パターンはリテラルで表記されている、具体的なトークンツリーでもよい。

```
macro_rules! gibberish {
    (4 fn ['spang  "whammo"] @_@) => {...};
}
```

3.5 Captures

パターンは**変数捕捉**を含むことができる。これはマクロの引数を照合し、結果を変数に束縛して出力に代入できるというものである。変数捕捉は$の後に変数名, : , 変数捕捉の種類と並べることで記述できる。変数捕捉の種類は以下の通り。

- item:関数、構造体、モジュールなどのアイテム
- block:ブロック (中括弧で囲まれた文および式)
- stmt:文
- pat:パターン
- expr:式
- ty:型
- ident:識別子
- path:パス (foo, ::std::mem::replace, transmute::<_, int>…など)
- meta:`#[...]` や`#![]` などの属性
- tt:単一のトークンツリー

変数捕捉を行うマクロの簡単な例

```
macro_rules! time_five {
    ($e:expr) => { $e * 5 };
}

macro_rules! multiply_add {
    ($a:expr, $b:expr, $c:expr) => {$a * ($b + $c)};
}
```

3.6 反復

パターンには**反復**を含めることができる。基本的な形式は`$ ( ... ) sep rep` となっている。

- (…) には括弧でグループ化されたパターンが入る。括弧内のパターンが反復される。

- sep:複数の引数を受け取るときのセパレータ記号の指定。大体 , または ;。
- rep:*または+を指定して反復回数が 0 回以上か 1 回以上かを指示する。

繰り返しを用いたマクロの例

```
macro_rules! vec_strs {
    (
        $( $element:expr) , *
    ) => {{
        let mut v = Vec::new();
        $(v.push(format!("{}",$element));
        )*
        v
    }}
}
```

3.7 捕捉と展開の後退

パーサがトークンを用いて捕捉を始めると、それを中断したりバックトラックしたりすることはできない。ある rule にマッチした場合はたとえそのマッチングが失敗したとしても、それより後に定義した rule には一切マッチしないことに注意すべきである。

```
macro_rules! dead_rule {
    ($e:expr) => { ... };
    ($i:ident +) => { ... };
}

dead_rules(x +);//=>compile error!
```

x +は式として有効であるため 1 つ目の rule にマッチする。そして+演算子の右の項が存在しないためパーサは panic を返す。この問題を避けるため、rule は具体性の高いものから順に書いていくべきである。

また、将来のマクロ解釈の変更によるシンタックスの変化を避けるため、macro_rules!は捕捉の後に続くものを制限している。

- item:制限なし
- block:制限なし
- stmt: => , ; のみ
- pat: => , = if in のみ
- expr: => , ; のみ
- ty: , => : = > ; as のみ
- ident:制限なし
- path: , => : = > ; as のみ
- meta:制限なし
- tt:制限なし

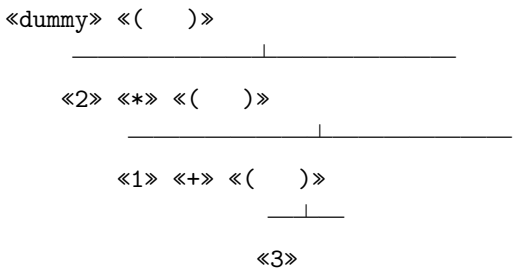
また、この制限に干渉していなくても反復の後に別の反復を付け加えることもできない。

さらによく間違える点として置換がトークンベースではないというものがある。次の stringify!マクロを用いた例を見よう。このマクロは入力したすべてのトークンを結合した文字列を返す。

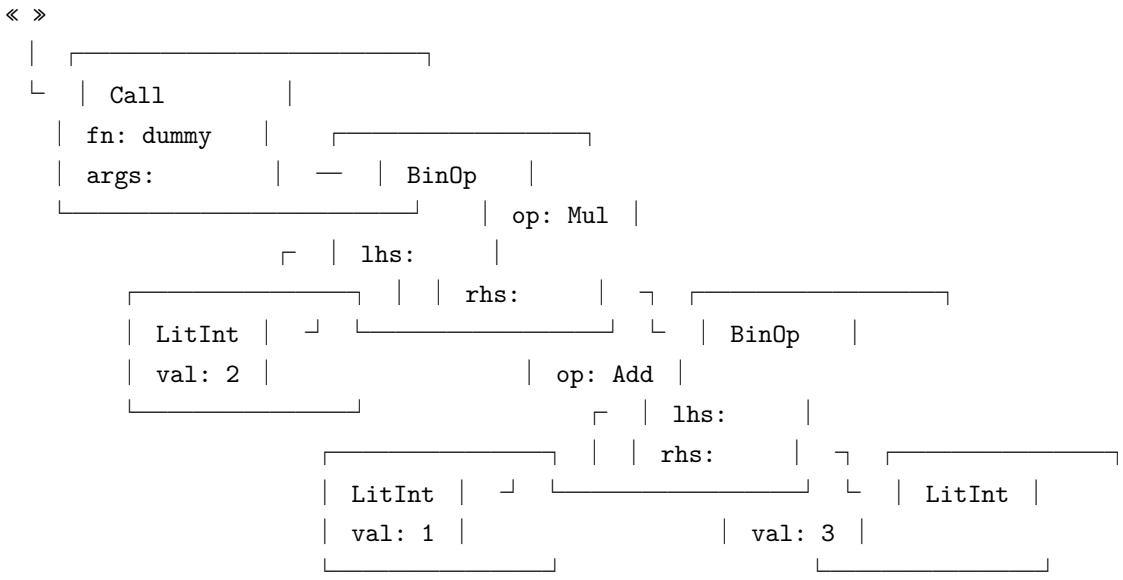
```
macro_rules! capture_expr_then_stringify {
    ($e:expr) => {
        stringify!($e)
    };
}

fn  main() {
    println!("{:?}", stringify!(dummy(2 * (1 + (3)))));
    println!("{:?}", capture_expr_then_stringify!(dummy(2 * (1 + (3)))));
}
//=> "dummy ( 2 * ( 1 + ( 3 ) ) )"
//=> "dummy(2 * (1 + (3)))"
```

1 つ目の stringify は dummy(2 * (1 + (3))) というトークンツリーに対して作用している。



対して、2 つ目の stringify は dummy(2 * (1 + (3))) という AST に対して作用している。



図は両者とも The Little Book of Rust Macros 2.3.1 Captures and Expansion Redux (https://danielkeep.github.io/tlborm/book/mbe-min-captures-and-expansion-redux.html) から引用。

この通り、capture_ expr_then_stringify に対して入力されるのは AST を含んだ 1 つのトークンツリーだけであり、出力は文字列化されたトークンではなく文字列化された AST のノードとなっている。

これを理解するための別の例を示す。

```
macro_rules! capture_then_match_tokens {
    ($e:expr) => {match_tokens!($e)};
}

macro_rules! match_tokens {
    ($a:tt + $b:tt) => {"got an addition"};
    (($i:ident)) => {"got an identifier"};
    ($($other:tt)*) => {"got something else"};
}

fn  main() {
    println!("{}\n{}\n{}\n",
        match_tokens!((caravan)), //=> got an identifier
        match_tokens!(3 + 6), //=> got an addition
        match_tokens!(5)); //=> got something else
    println!("{}\n{}\n{}",
        capture_then_match_tokens!((caravan)),//=> got something else
        capture_then_match_tokens!(3 + 6), //=> got something else
        capture_then_match_tokens!(5)); //=> got something else
}
```

入力を AST にパースすることで、置換結果は分解できなくなる。したがって、その内容をもう一度確認したりマッチングしようとすることはできない。

```
macro_rules! capture_then_what_is {
    (#[$m:meta]) => {what_is!(#[$m])};
}

macro_rules! what_is {
    (#[no_mangle]) => {"no_mangle attribute"};
    (#[inline]) => {"inline attribute"};
    ($($tts:tt)*) => {concat!("something else (", stringify!($($tts)*), ")")};
}

fn  main() {
    println!("{}\n{}\n{}\n{}",
        what_is!(#[no_mangle]), //=>no_mangle attribute
        what_is!(#[inline]), //=>inline attribute
        capture_then_what_is!(#[no_mangle]),//=>something else (# [ no_mangle ])
        capture_then_what_is!(#[inline]),//=>something else (# [ inline ])
    );
}
```

これを回避するためには `tt` または `ident` を用いて捕捉しなければならない。

3.8 健全なマクロ

多くの言語において、マクロは変数名が誤って捕捉されうる問題がある。これを**意図しない変数捕捉**と呼ぶ。例えば C 言語なら次のような事態が起こる可能性がある。

```
#define  incv_with_a(v) {int a=0; v++;}
int main(void){
    int a=10;
    incv_with_a(a);
    printf("%d\n",a);//=>10
}
```

これを避ける簡単なやり方はマクロ展開部の中で用いる変数名を絶対に干渉しない名前にすることである。

```
#define  incv_with_a(v) {int incv_with_a_var=0; v++;}
```

これで incv_with_a_var という変数が与えられない限り干渉することはなくなった。しかしこれは頑強とは言えないので、何らかの手段出す必要がある。 Common Lisp においては、gensym 関数がそれを担っている。 gensym はコンパイラに絶対に干渉しない変数名の生成を指示する。

```
(defmacro incv-with-a (v)
  (let ((a (gensym)))
    `(let ((,a 0))
      (incf ,v))))
```

Rust においては意図しない変数捕捉は発生しないようになっている。

```
macro_rules! using_a {
    ($e:expr) => {
        {
            let a = 42;
            $e
        }
    }
}
fn  main(){
    let a = 100;
    let four = using_a!(a / 10);
    println!("{}",a);//=> 10
}
```

この using_a マクロは`{let a = 42; a/10}`のように展開される。しかし Rust はすべての識別子に**構文コンテキスト値**を付加しており、変数名が一致しても文脈が異なると干渉しないようになっている。　ローカル変数を活用したい場合は次のようにする。

```
macro_rules! using_a {
    ($a:ident, $e:expr) => {
        {
            let $a = 42;
            $e
        }
```

```
    }
}
fn  main() {
    let four = using_a!(a, a / 10);
    println!("{}", four);//=> 4
}
```

3.9 特殊な識別子

一見識別子に見えても実際には異なる挙動を見せるもの 2 つがある。その一つは `self` である。 `self` は原則キーワードであるが、識別子の定義に当てはまる場合がある。

```
macro_rules! what_is {
    (self) => {"the keyword `self`"};
    ($i:ident) => {concat!("the identifier `", stringify!($i), "`")};
}

macro_rules! call_with_ident {
    ($c:ident($i:ident)) => {$c!($i)};
}

fn main() {
    println!("{}", what_is!(self));                  //=> the keyword `self`
    println!("{}", call_with_ident!(what_is(self)));//=> the keyword `self`
}
```

call_with_ident!マクロは識別子でないとマッチしないが、ここでは self が識別子であると認識してしまっている。つまり、self はキーワードでありながら識別子となるのである。

```
macro_rules! make_mutable {
    ($i:ident) => {let mut $i = $i;};
}

struct Dummy(i32);
impl Dummy {
    fn double(self) -> Dummy {
        make_mutable!(self);
        self.0 *= 2;
        self
    }
}
```

この例は make_mutable で変数を可変にシャドーイングすることを意図しているが、self は基本的にはキーワードで識別されるためにこのコードはコンパイルされない。

```
<anon>:2:28: 2:30 error: expected identifier, found keyword `self`
<anon>:2     ($i:ident) => {let mut $i = $i;};
                                    ^~
```

以下動かない例

```
//double 引数の self と body 内の self のコンテキストの違い
macro_rules! double_method {
    ($body:expr) => {
        fn double(mut self) -> Dummy {
            $body
        }
    };
}

struct Dummy(i32);
impl Dummy {
    double_method! {{
        self.0 *= 2;
        self
    }}
}
```

```
//_はキーワードであるが self と違い識別子と認識されることはない
//代わりに_は pat として認識され得る。
macro_rules! double_method {
    ($self_:ident, $body:expr) => {
        fn double($self_) -> Dummy {
            $body
        }
    };
}

struct Dummy(i32);
impl Dummy {
    double_method! {_, 0}
}
```

動く例

```
//double 内の self のコンテキストが一致するので動作する
macro_rules! double_method {
    ($self_:ident, $body:expr) => {
        fn double(mut $self_) -> Dummy {
            $body
        }
    };
}

struct Dummy(i32);
impl Dummy {
    double_method! {self, {
        self.0 *= 2;
```

```
        self
    }}
}

fn main(){
    let mut a = Dummy(3);
    a = a.double(); //a.double() は a の所有権を持っているので返す必要がある。
    //a.0=>6
}
```

3.10 デバッグ

trace_macros!マクロを使えばコンパイラにマクロ展開の結果を出力させることができる。

```
#![feature(trace_macros)]
macro_rules! each_tt {
    () => {};
    ($_tt:tt $($rest:tt)*) => {each_tt!($($rest)*);};
}

each_tt!(foo bar baz quux);
trace_macros!(true);
each_tt!(spim wak plee whum);
trace_macros!(false);
each_tt!(trom qlip winp xod);
```

出力は次の通り。

```
each_tt! { spim wak plee whum }
each_tt! { wak plee whum }
each_tt! { plee whum }
each_tt! { whum }
each_tt! {  }
```

`-Z trace-macros` オプションをコンパイラに指示することでコマンドラインからこれを有効にすることもできる。 (cargo でやろうとすると標準コンパイラを nightly にする必要があるので、rustup run nightly rustc …とやるのがよい？)

マクロプログラミングに便利なもう一つのマクロは **log_syntax!マクロ**である。このマクロは渡されたトークンを単純にすべて出力する。コンパイル時に必要なオプションは次のようになる。

```
rustc -Z unstable-options --pretty expanded main.rs
```

```
//コンパイラが歌を歌うコード
#![feature(log_syntax)]

macro_rules! sing {
    () => {};
    ($tt:tt $($rest:tt)*) => {log_syntax!($tt); sing!($($rest)*);};
```

```
}

sing! {
    ^ < @ < . @ *
    '\x08' '{' '"' _ # ' '
    - @ '$' && / _ %
    ! ( '\t' @ | = >
    ; '\x08' '\'' + '$' ? '\x7f'
    , # '"' ~ | ) '\x07'
}
```

3.11 スコープ

マクロのスコープは少し分かりづらくなっている。まず、マクロはそのスコープのサブモジュール内では使用することができる。

```
macro_rules! X { () => {}; }
mod a {
    X!(); //もしも関数ならば super::X と書かなければならない
}
```

コードの途中で定義された場合はそれ以降のコード内でないと利用できない。

```
mod a {
    // X!(); // undefined
}
macro_rules! X { () => {}; }
mod b {
    X!(); // defined
}
mod c {
    X!(); // defined
}
```

```
mod a {
    // X!(); // undefined
}
macro_rules! X { () => { Y!(); }; }
mod b {
    // X!(); // defined, but Y! is undefined
}
macro_rules! Y { () => {}; }
mod c {
    X!(); // defined, and so is Y!
}
```

ただし、`extern crate` した場合はこの通りではない。クレート内のマクロは呼び出したファイルの最上部で宣言されたように振る舞うため、どの場所でもマクロ呼び出しが可能となる。

```
mod a {
    // X!(); // defined, but Y! is undefined
}
macro_rules! Y { () => {}; }
mod b {
    X!(); // defined, and so is Y!
}
#[macro_use] extern crate foo;//foo 内には X が定義されている
mod c {
    X!(); // defined, and so is Y!
}
```

これらの挙動は#[macro_use] 属性を除いて関数においても同様である。

```
macro_rules! X {
    () => { Y!() };
}

fn a() {
    macro_rules! Y { () => {"Hi!"} }
    assert_eq!(X!(), "Hi!");
    {
        assert_eq!(X!(), "Hi!");
        macro_rules! Y { () => {"Bye!"} }
        assert_eq!(X!(), "Bye!");
    }
    assert_eq!(X!(), "Hi!");
}

fn b() {
    macro_rules! Y { () => {"One more"} }
    assert_eq!(X!(), "One more");
}
```

一般的には、モジュールを定義する前にルートの最上部にすべてのマクロを配置することが推奨される。こうすることでそのファイル内のあらゆる場所で一貫して定義したマクロを利用できる。

3.12 インポートとエクスポート

マクロを外部のスコープにエクスポートする方法は 2 種類ある。一つは#[macro_use] を使うことである。#[macro_use] はモジュールの場合でも外部クレートの場合でも使える。モジュールの場合は次のように使える。

```
#[macro_use]
mod macros {
    macro_rules! X { () => { Y!(); } }
    macro_rules! Y { () => {} }
```

```
}
X!();
```

#[macro_export] 属性はあらゆるスコープを無視してしまう。

```
//foo というクレートで定義しているとする
mod macros {
    #[macro_export] macro_rules! X { () => { Y!(); } }
    #[macro_export] macro_rules! Y { () => {} }
}
//macros が private のため X も Y もスコープ外だが、export されているためここでは利用できる。
```

外部クレートで利用するには次のようにする。

```
// X is defined
#[macro_use] extern crate foo;
X!();
```

ルートモジュールからは外部クレートしか#[macro_use] できないことに注意する。

特定のマクロだけをインポートするときは#[macro_use(foo)] などとする。$crate::X!などの記法は使えないので注意。

```
// Import *only* the `X!` macro.
#[macro_use(X)] extern crate macs;
// X!(); // X is defined, but Y! is undefined
macro_rules! Y { () => {} }
X!(); // X is defined, and so is Y!
```

4　マクロの実践的な利用

ここでは、Rust 競プロ界隈でよく使われている input!マクロを解読した。記事内で解説してあるマクロは通常のものと追記された遅延入力版のものがあり、後者は、EOF を入力に与える必要がある (一度に全ての入力を受取り、それから解析する手法のため) という前者の問題点を解決したものになっているが、クロージャなどのマクロとは異なる高度な概念を利用しているため、本項では前者のコードを読解していく。

主な使い方は以下の通り。

```
input!{
    n:usize,
    m:usize, //var:type と宣言すると入力値を受け取って let var:type = (入力値); に展開される
    v:[usize;n], //var:[type;size] と宣言すると Vec<type>型で size 個の入力を受け取る
    s:chars, //任意の長さの文字列を Vec<char>として受け取る
    vv:[[usize;n];m], //サイズが m*n の Vec<Vec<usize>>を入力として受け取る
}
```

input!マクロの定義は次のようになっている。 (解説のため一部改変している。)

```
macro_rules! input {
    ($($r:tt)*) => {
```

```
        let mut s = {
            use std::io::Read;
            let mut s = String::new();
            std::io::stdin().read_to_string(&mut s).unwrap();
            s
        };
        let mut iter = s.split_whitespace();
        input_inner!{iter, $($r)*}
    };
}
```

ここでは、まず入力を全て String s に渡し、s の空白区切りのイテレータを取得している。 input!マクロは入力を空白区切りでパースしてそれと引数のトークンツリーを input_inner!マクロに渡すという役割を行っている。

input_inner!マクロの定義は次のようになっている。

```
macro_rules! input_inner {
    ($iter:expr) => {};
    ($iter:expr, ) => {};

    ($iter:expr, $var:ident : $t:tt $($r:tt)*) => {
        let $var = read_value!($iter, $t);
        input_inner!{$iter $($r)*}
    };
}
```

1 つ目と 2 つ目のパターンは再帰的な展開の終端となる。最後のパターンで let 文への展開を行い、次の展開に移る。 let 文の右辺の値の部分は read_value!マクロが入力文字列 s のイテレータと値の型を受け取って展開する。

read_value!マクロの定義は次のようになっている。

```
macro_rules! read_value {
    ($iter:expr, ( $($t:tt),* )) => {
        ( $(read_value!($iter, $t)),* )
    };
    ($iter:expr, [ $t:tt ; $len:expr ]) => {
        (0..$len).map(|_| read_value!($iter, $t)).collect::<Vec<_>>()
    };
    ($iter:expr, chars) => {
        read_value!($iter, String).chars().collect::<Vec<char>>()
    };
    ($iter:expr, usize1) => {
        read_value!($iter, usize) - 1
    };
    ($iter:expr, $t:ty) => {
        $iter.next().unwrap().parse::<$t>().expect("Parse error")
    };
```

```
}
```

5 パターンそれぞれにおいて次のような処理を行っている。

1. 最初のパターンは型が (usize,usize) のようにタプルである場合で、このときはタプル内の各要素に対して read_value!を適用する。
2. 二番目のパターンは [usize; n] のように Vec 型と長さが渡されていた場合であり、これも n 個の各要素に対して read_value!を行う。
3. 三番目のパターンは chars というキーワードだった場合でこれは文字列を Vec 型として読み替える処理を行う。
4. 四番目は usize1 というキーワードだった場合で、usize 型で変数を読み取った後に 1 引いた値を返す。
5. 最後は単独の値のパターンであり、入力を与えられた型の値として返し、イテレータを次の変数の値に移す。

展開後の結果は次のようになる。

```
let mut s =
        {
            use std::io::Read;
            let mut s = String::new();
            std::io::stdin().read_to_string(&mut s).unwrap();
            s
        };
let mut iter = s.split_whitespace();

let n = iter.next().unwrap().parse::<usize>().expect("Parse error");
let m = iter.next().unwrap().parse::<usize>().expect("Parse error");
let v = (0..n).map(|_| iter.next().unwrap().parse::<usize>().expect("Parse error"))
              .collect::<Vec<_>>();
let s = iter.next().unwrap().parse::<String>().expect("Parse error")
            .chars().collect::<Vec<char>>();
let vv =
    (0..m).map(|_|
                    (0..n).map(|_|
                                    iter.next().unwrap().parse::<usize>()
                                        .expect("Parse error"))
                          .collect::<Vec<_>>())
         .collect::<Vec<_>>();
}
```

5 成果

成果として次の 2 つが挙げられる。

1. 体系的なマクロの理論を学習したことによる成果である。マクロの定義方法について体系的に学ぶことでマクロの定義コードをスムーズに読むことができるようになった。特に input!マクロなどは複雑なパターンで読みにくく、この活動でしっかり理論部分を学ばなければ読めなかったと思われる。また、同じパターンの関数定義などをマクロの自作によって短縮することができるようになった。

2. 実際に利用されているマクロを読んだことによる成果である。実際に読んだ input!マクロを利用して競技プログラミングを行った。マクロのコードを読む前は記事に書いてある内容を読むしかなく仕様を理解しきれていない面があったが、コードを読んだ後はそのマクロで展開されたコードが考えられるようになり、コンパイルエラーを減らすことができた。