

# Chapitre 21. Macros

*A cento (du latin pour « patchwork ») est un poème entièrement composé de vers cités d'un autre poète.*

—Matt Madden

Rust prend en charge les *macros*, un moyen d'étendre le langage d'une manière qui va au-delà de ce que vous pouvez faire avec les seules fonctions. Par exemple, nous avons vu la macro, ce qui est pratique pour les tests: `assert_eq!`

```
assert_eq!(gcd(6, 10), 2);
```

Cela aurait pu être écrit comme une fonction générique, mais la macro fait plusieurs choses que les fonctions ne peuvent pas faire. La première est que lorsqu'une assertion échoue, génère un message d'erreur contenant le nom de fichier et le numéro de ligne de l'assertion. Les fonctions n'ont aucun moyen d'obtenir cette information. Les macros le peuvent, car leur façon de travailler est complètement différente. `assert_eq!` `assert_eq!`

Les macros sont une sorte de raccourci. Pendant la compilation, avant que les types ne soient vérifiés et bien avant qu'un code machine ne soit généré, chaque appel de macro est *développé*, c'est-à-dire remplacé par du code Rust. L'appel de macro précédent s'étend à quelque chose d'à peu près comme ceci :

```
match (&gcd(6, 10), &2) {
    (left_val, right_val) => {
        if !(*left_val == *right_val) {
            panic!("assertion failed: `(left == right)`,\n\
                (left: `{:?}`, right: `{:?}`)", left_val, right_val);
        }
    }
}
```

`panic!` est également une macro, qui elle-même s'étend à encore plus de code Rust (non montré ici). Ce code utilise deux autres macros et `.`. Une fois que chaque appel macro dans la caisse est complètement étendu, Rust passe à la phase suivante de compilation. `file!()` `line!()`

Au moment de l'exécution, un échec d'assertion ressemblerait à ceci (et indiquerait un bogue dans la fonction, car c'est la bonne réponse)

```
: gcd() 2
```

```
thread 'main' panicked at 'assertion failed: `(left == right)`', (left: `1`  
right: `2`)', gcd.rs:7
```

Si vous venez de C++, vous avez peut-être eu de mauvaises expériences avec les macros. Les macros Rust adoptent une approche différente, similaire à celle de Scheme. Par rapport aux macros C++, les macros Rust sont mieux intégrées au reste du langage et donc moins sujettes aux erreurs. Les appels de macro sont toujours marqués d'un point d'exclamation, de sorte qu'ils se démarquent lorsque vous lisez du code, et ils ne peuvent pas être appelés accidentellement lorsque vous vouliez appeler une fonction. Les macros Rust n'insèrent jamais de crochets ou de parenthèses inégaux. Et les macros Rust sont livrées avec une correspondance de motifs, ce qui facilite l'écriture de macros à la fois maintenables et attrayantes à utiliser. `syntax-rules`

Dans ce chapitre, nous allons montrer comment écrire des macros à l'aide de plusieurs exemples simples. Mais comme beaucoup de Rust, les macros récompensent une compréhension profonde, nous allons donc parcourir la conception d'une macro plus compliquée qui nous permet d'intégrer des littéraux JSON directement dans nos programmes. Mais il y a plus dans les macros que ce que nous pouvons couvrir dans ce livre, nous terminerons donc avec quelques conseils pour une étude plus approfondie, à la fois des techniques avancées pour les outils que nous vous avons montrés ici, et pour une installation encore plus puissante appelée *macros procédurales*.

## Principes de base des macros

[La figure 21-1](#) montre une partie du code source de la macro `assert_eq!`

`macro_rules!` est le principal moyen de définir des macros dans Rust. Notez qu'il n'y a pas d'après dans cette définition de macro : le `!` n'est inclus que lors de l'appel d'une macro, pas lors de sa définition. `! assert_eq !`

Toutes les macros ne sont pas définies de cette façon : quelques-unes, comme `println!`, et elle-même, sont intégrées dans le compilateur, et nous parlerons d'une autre approche, appelée macros procédurales, à la fin de ce chapitre. Mais pour la plupart, nous nous concentrerons sur `macro_rules!`, qui est

(jusqu'à présent) le moyen le plus simple d'écrire le  
votre.file! line! macro\_rules! macro\_rules!

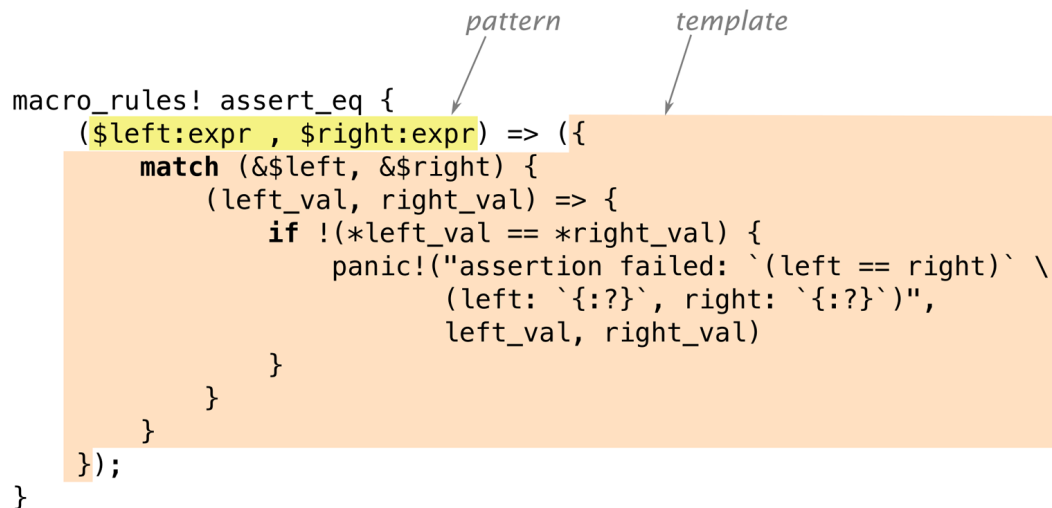
Une macro définie avec fonctionne entièrement par correspondance de motifs. Le corps d'une macro n'est qu'une série de règles : macro\_rules!

```
( pattern1 ) => ( template1 );

( pattern2 ) => ( template2 );

...

macro_rules! assert_eq {
    ($left:expr , $right:expr) => ({
        match (&$left, &$right) {
            (left_val, right_val) => {
                if !(*left_val == *right_val) {
                    panic!("assertion failed: `(left == right)` \
                        (left: `{:?}`, right: `{:?}`)",
                        left_val, right_val)
                }
            }
        }
    });
}
```



Graphique 21-1. La macro assert\_eq!

La version de la [figure 21-1](#) n'a qu'un seul modèle et un seul modèle. assert\_eq!

Incidemment, vous pouvez utiliser des crochets ou des accolades au lieu de parenthèses autour du motif ou du modèle; cela ne fait aucune différence pour Rust. De même, lorsque vous appelez une macro, ceux-ci sont tous équivalents :

```
assert_eq!(gcd(6, 10), 2);
assert_eq![gcd(6, 10), 2];
assert_eq!{gcd(6, 10), 2}
```

La seule différence est que les points-virgules sont généralement facultatifs après les accolades. Par convention, nous utilisons des parenthèses lors de l'appel, des crochets pour , et des accolades pour . assert\_eq! vec! macro\_rules!

Maintenant que nous avons montré un exemple simple de l'expansion d'une macro et la définition qui l'a générée, nous pouvons entrer dans les détails nécessaires pour le mettre en œuvre:

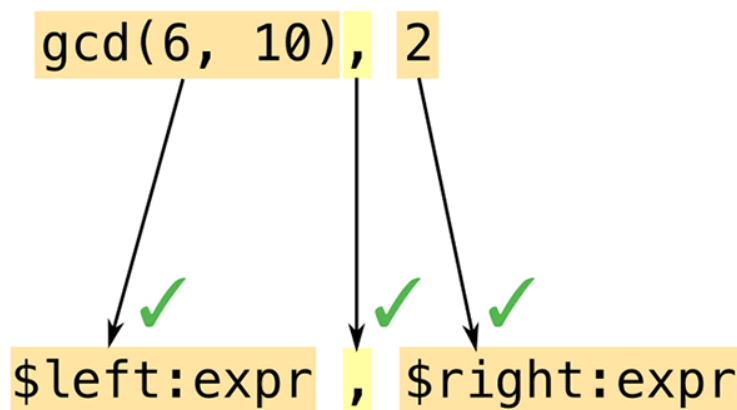
- Nous vous expliquerons exactement comment Rust s’y prend pour trouver et développer des définitions de macro dans votre programme.
- Nous soulignerons quelques subtilités inhérentes au processus de génération de code à partir de modèles de macro.
- Enfin, nous montrerons comment les modèles gèrent la structure répétitive.

## Principes de base de l’expansion macro

Rust développe les macros très tôt pendant la compilation. Le compilateur lit votre code source du début à la fin, en définissant et en développant les macros au fur et à mesure. Vous ne pouvez pas appeler une macro avant qu’elle ne soit définie, car Rust développe chaque appel de macro avant même d’examiner le reste du programme. (En revanche, les fonctions et autres [éléments](#) n’ont pas besoin d’être dans un ordre particulier. Vous pouvez appeler une fonction qui ne sera définie que plus tard dans la caisse.)

Lorsque Rust développe un appel macro, ce qui se passe ressemble beaucoup à l’évaluation d’une expression. Rust correspond d’abord aux arguments contre le modèle, comme le montre [la figure 21-](#)

[2](#). `assert_eq! match`



Graphique 21-2. Développement d’une macro, partie 1 : correspondance des modèles avec les arguments

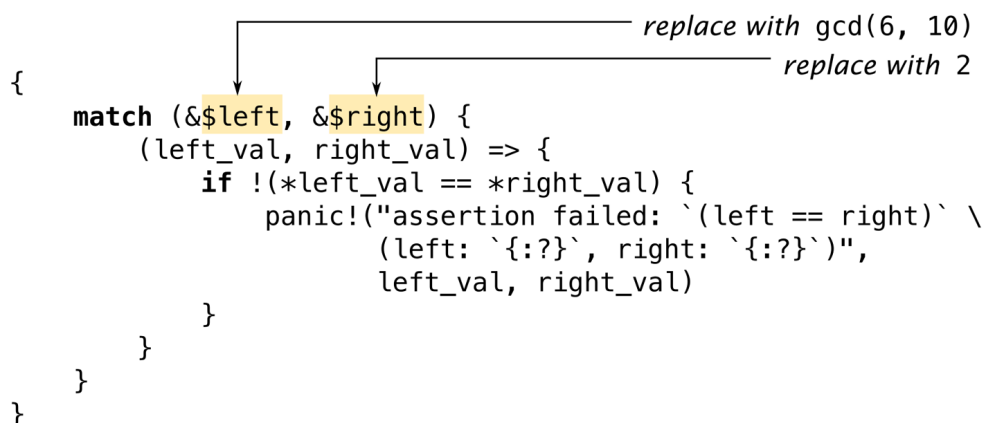
Les modèles de macro sont un mini-langage dans Rust. Ce sont essentiellement des expressions régulières pour faire correspondre le code. Mais là où les expressions régulières fonctionnent sur des caractères, les modèles fonctionnent sur des *jetons* – les *nombre*s, les noms, les signes de ponctuation, etc. qui sont les éléments constitutifs des programmes Rust. Cela signifie que vous pouvez utiliser librement les commentaires et les espaces blancs dans les modèles de macro pour les rendre aussi lisibles que possible. Les commentaires et les espaces blancs ne sont pas des jetons, ils n’affectent donc pas la correspondance.

Une autre différence importante entre les expressions régulières et les modèles de macro est que les parenthèses, les crochets et les accolades se produisent toujours par paires correspondantes dans Rust. Ceci est vérifié avant que les macros ne soient développées, non seulement dans les modèles de macro, mais dans toute la langue.

Dans cet exemple, notre motif contient le *fragment* , qui indique à Rust de correspondre à une expression (dans ce cas, ) et de lui attribuer le nom . Rust fait ensuite correspondre la virgule dans le motif avec la virgule suivant les arguments de . Tout comme les expressions régulières, les motifs n'ont que quelques caractères spéciaux qui déclenchent un comportement de correspondance intéressant; tout le reste, comme cette virgule, doit correspondre textuellement, sinon la correspondance échoue. Enfin, Rust correspond à l'expression et lui donne le nom . `$left:expr gcd(6, 10) $left gcd 2 $right`

Les deux fragments de code de ce modèle sont de type : ils attendent des expressions. Nous verrons d'autres types de fragments de code dans [« Types de fragments »](#). `expr`

Étant donné que ce modèle correspond à tous les arguments, Rust développe le *modèle* correspondant ([Figure 21-3](#)).



Graphique 21-3. Développement d'une macro, partie 2 : remplissage du modèle

Rust remplace et avec les fragments de code qu'il a trouvés lors de la correspondance. `$left $right`

C'est une erreur courante d'inclure le type de fragment dans le modèle de sortie : écrire plutôt que simplement . Rust ne détecte pas immédiatement ce genre d'erreur. Il voit comme une substitution, puis il traite comme tout le reste dans le modèle: les jetons à inclure dans la sortie de la macro. Ainsi, les erreurs ne se produiront pas tant que vous *n'aurez pas appelé* la macro ; ensuite, il générera une fausse sortie qui ne se compilera pas. Si vous recevez des messages d'erreur comme et lors de l'utilisation d'une nouvelle macro, vérifiez-la pour cette erreur. ([« Débogage des](#)

[macros »](#) offre des conseils plus généraux pour des situations comme celle-ci.) `$left:expr $left $left :expr cannot find type `expr` in this scope help: maybe you meant to use a path separator here`

Les modèles de macro ne sont pas très différents de l'un des douze langages de modèles couramment utilisés dans la programmation Web. La seule différence – et c'est une différence significative – est que la sortie est du code Rust.

## Conséquences imprévues

Le branchement de fragments de code dans des modèles est subtilement différent du code normal qui fonctionne avec des valeurs. Ces différences ne sont pas toujours évidentes au début. La macro que nous avons examinée, `assert_eq!`, contient des morceaux de code légèrement étranges pour des raisons qui en disent long sur la programmation macro. Regardons deux morceaux amusants en particulier. `assert_eq!`

Tout d'abord, pourquoi cette macro crée-t-elle les variables `left` et `right`? Y a-t-il une raison pour laquelle nous ne pouvons pas simplifier le modèle pour ressembler à ceci?

```
left_val right_val
```

```
if !($left == $right) {
    panic!("assertion failed: `(left == right)` \
          (left: `{:?}`, right: `{:?}`)", $left, $right)
}
```

Pour répondre à cette question, essayez d'étendre mentalement l'appel macro `assert_eq!`. Quel serait le résultat? Naturellement, Rust brancherait les expressions correspondantes dans le modèle à plusieurs endroits. Cela semble être une mauvaise idée d'évaluer à nouveau les expressions lors de la création du message d'erreur, et pas seulement parce que cela prendrait deux fois plus de temps: puisque `pop()` supprime une valeur d'un vecteur, cela produira une valeur différente la deuxième fois que nous l'appellerons! C'est pourquoi la macro réelle calcule et ne calcule qu'une seule fois et stocke ses valeurs. `assert_eq!(letters.pop(), Some('z'))` `letters.pop()` `$left $right`

Passons à la deuxième question : pourquoi cette macro emprunte-t-elle des références aux valeurs de `left` et `right`? Pourquoi ne pas simplement stocker les valeurs dans des variables, comme celle-ci?

```
$left $right
```

```
macro_rules! bad_assert_eq {
    ($left:expr, $right:expr) => ({
```

```

        match ($left, $right) {
            (left_val, right_val) => {
                if !(left_val == right_val) {
                    panic!("assertion failed" /* ... */);
                }
            }
        }
    });
}

```

Pour le cas particulier que nous avons examiné, où les arguments macro sont des entiers, cela fonctionnerait bien. Mais si l'appelant passait, disons, une variable comme `ou` , ce code déplacerait la valeur hors de la variable! `String $left $right`

```

fn main() {
    let s = "a rose".to_string();
    bad_assert_eq!(s, "a rose");
    println!("confirmed: {} is a rose", s); // error: use of moved value
}

```

Comme nous ne voulons pas que les assertions déplacent des valeurs, la macro emprunte plutôt des références.

(Vous vous êtes peut-être demandé pourquoi la macro utilise plutôt que de définir les variables. Nous nous sommes demandés aussi. Il s'avère qu'il n'y a pas de raison particulière à cela. aurait été équivalent.) `match let let`

En bref, les macros peuvent faire des choses surprenantes. Si des choses étranges se produisent autour d'une macro que vous avez écrite, il y a fort à parier que la macro est à blâmer.

Un bogue que vous *ne verrez pas* est ce bogue de macro C++ classique :

```

// buggy C++ macro to add 1 to a number
#define ADD_ONE(n)  n + 1

```

Pour des raisons familières à la plupart des programmeurs C++, et qui ne valent pas la peine d'être expliquées en détail ici, un code banal, comme `ou` produit des résultats très surprenants avec cette macro. Pour le résoudre, vous devez ajouter d'autres parenthèses à la définition de macro. Ce n'est pas nécessaire dans Rust, car les macros Rust sont mieux intégrées au langage. Rust sait quand il manipule des expressions, il ajoute

donc efficacement des parenthèses chaque fois qu'il colle une expression dans une autre. `ADD_ONE(1) * 10` `ADD_ONE(1 << 4)`

## Répétition

La macro standard se présente sous deux formes : `vec!`

```
// Repeat a value N times
let buffer = vec![0_u8; 1000];

// A list of values, separated by commas
let numbers = vec!["udon", "ramen", "soba"];
```

Il peut être mis en œuvre comme ceci:

```
macro_rules! vec {
    ($elem:expr ; $n:expr) => {
        ::std::vec::from_elem($elem, $n)
    };
    ( $( $x:expr ),* ) => {
        <[_]>::into_vec(Box::new([ $( $x ),* ]))
    };
    ( $( $x:expr ),+ , ) => {
        vec![ $( $x ),* ]
    };
}
```

Il y a trois règles ici. Nous expliquerons le fonctionnement de plusieurs règles, puis examinerons chaque règle à tour de rôle.

Lorsque Rust développe un appel de macro comme `vec![1, 2, 3]`, il commence par essayer de faire correspondre les arguments avec le modèle de la première règle, dans ce cas `$elem:expr ; $n:expr`. Cela ne correspond pas: `1, 2, 3` est une expression, mais le motif nécessite un point-virgule après cela, et nous n'en avons pas. Rust passe donc à la deuxième règle, et ainsi de suite. Si aucune règle ne correspond, c'est une erreur. `vec![1, 2, 3]` `1, 2, 3` `$elem:expr ; $n:expr` `1`

La première règle gère des utilisations telles que `vec![0_u8; 1000]`. Il arrive qu'il existe une fonction standard (mais non documentée), qui fait exactement ce qui est nécessaire ici, donc cette règle est simple. `std::vec::from_elem`

La deuxième règle gère `vec!["udon", "ramen", "soba"]`. Le modèle `$( $x:expr ),*` utilise une fonctionnalité que nous n'avons jamais vue auparavant: la répétition. Il correspond à 0 expression ou plus, séparées par des virgules. Plus généralement, la syntaxe est



utilisée pour faire correspondre n'importe quelle liste séparée par des virgules, où chaque élément de la liste correspond à . `vec! [ "udon" , "ramen" , "soba" ] $( $x:expr ) , * $( PATTERN ) , * PATTERN`

Le `ici` a la même signification que dans les expressions régulières (« 0 ou plus ») bien qu'il soit vrai que les regexps n'ont pas de répéteur spécial. Vous pouvez également utiliser pour exiger au moins une correspondance, ou pour zéro ou une correspondance. [Le tableau 21-1](#) présente l'ensemble des modèles de répétition. `*` , `* +` ?

Tableau 21-1. Modèles de répétition

Modèle	Signification
<code>\$( ... ) *</code>	Faire correspondre 0 fois ou plus sans séparateur
<code>\$( ... ) , *</code>	Faire correspondre 0 fois ou plus, séparées par des virgules
<code>\$( ... ) ; *</code>	Faire correspondre 0 fois ou plus, séparés par des points-virgules
<code>\$( ... ) +</code>	Faire correspondre 1 fois ou plus sans séparateur
<code>\$( ... ) , +</code>	Faire correspondre 1 fois ou plus, séparées par des virgules
<code>\$( ... ) ; +</code>	Faire correspondre 1 fois ou plus, séparés par des points-virgules
<code>\$( ... ) ?</code>	Faire correspondre 0 ou 1 fois sans séparateur
<code>\$( ... ) , ?</code>	Faire correspondre 0 ou 1 fois, séparés par des virgules
<code>\$( ... ) ; ?</code>	Faire correspondre 0 ou 1 fois, séparés par des points-virgules

Le fragment de code n'est pas seulement une expression unique, mais une liste d'expressions. Le modèle de cette règle utilise également la syn-

taxe de répétition : `$x`

```
<[_]>::into_vec(Box::new([ $( $x ),* ]))
```

Encore une fois, il existe des méthodes standard qui font exactement ce dont nous avons besoin. Ce code crée un tableau en boîte, puis utilise la méthode pour convertir le tableau en boîte en vecteur. `[T]::into_vec`

Le premier bit, `,`, est une façon inhabituelle d'écrire le type « tranche de quelque chose », tout en s'attendant à ce que Rust déduise le type d'élément. Les types dont les noms sont des identificateurs simples peuvent être utilisés dans des expressions sans tracas, mais des types tels que `,`, ou doivent être enveloppés dans des crochets. `<[_]> fn() &str [_]`

La répétition entre à la fin du modèle, où nous avons `.`. C'est la même syntaxe que nous avons vue dans le modèle. Il parcourt la liste des expressions que nous avons appariées et les insère toutes dans le modèle, séparées par des virgules. `$( $x ),* $( ... ),* $x`

Dans ce cas, la sortie répétée ressemble à l'entrée. Mais cela ne doit pas nécessairement être le cas. Nous aurions pu écrire la règle comme ceci :

```
( $( $x:expr ),* ) => {  
    {  
        let mut v = Vec::new();  
        $( v.push($x); )*  
        v  
    }  
};
```

Ici, la partie du modèle qui lit insère un appel à `push` pour chaque expression dans `v`. Un bras macro peut s'étendre à une séquence d'expressions, mais ici nous n'avons besoin que d'une seule expression, nous enveloppons donc l'assemblage du vecteur dans un bloc. `$( v.push($x); )* v.push() $x`

Contrairement au reste de Rust, les motifs utilisés ne prennent pas automatiquement en charge une virgule de fin facultative. Cependant, il existe une astuce standard pour prendre en charge les virgules de fin en ajoutant une règle supplémentaire. C'est ce que fait la troisième règle de notre macro : `$( ... ),* vec!`

```
( $( $x:expr ),+ , ) => { // if trailing comma is present,  
    vec![ $( $x ),* ]    // retry without it  
};
```

Nous avons l'habitude de faire correspondre une liste avec une virgule supplémentaire. Ensuite, dans le modèle, nous appelons récursivement, en laissant la virgule supplémentaire de côté. Cette fois, la deuxième règle correspondra. `$ ( ... ) , + , vec !`

## Macros intégrées

Le compilateur Rust fournit plusieurs macros qui sont utiles lorsque vous définissez vos propres macros. Aucun d'entre eux n'a pu être mis en œuvre en utilisant seul. Ils sont codés en dur en `: macro_rules ! rustc`

`file ! ( ) , , line ! ( ) column ! ( )`

`file ! ( )` se développe en un littéral de chaîne : le nom de fichier actuel. et développez jusqu'aux littéraux donnant la ligne et la colonne actuelles (en comptant à partir de 1). `line ! ( ) column ! ( ) u32`

Si une macro en appelle une autre, qui en appelle une autre, le tout dans des fichiers différents, et que la dernière macro appelle `, ,` ou `,` elle se développera pour indiquer l'emplacement du *premier* appel de macro. `file ! ( ) line ! ( ) column ! ( )`

`stringify ! ( ... tokens ... )`

Se développe en un littéral de chaîne contenant les jetons donnés. La macro l'utilise pour générer un message d'erreur qui inclut le code de l'assertion. `assert !`

Les appels de macro dans l'argument *ne sont pas* développés : se développe à la chaîne `. stringify ! ( line ! ( ) ) "line ! ( )"`

Rust construit la chaîne à partir des jetons, de sorte qu'il n'y a pas de sauts de ligne ou de commentaires dans la chaîne.

`concat ! ( str0 , str1 , ... )`

Se développe à un littéral de chaîne unique créé en concaténant ses arguments.

Rust définit également ces macros pour interroger l'environnement de génération :

`cfg ! ( ... )`

Se développe à une constante booléenne, si la configuration de build actuelle correspond à la condition entre parenthèses. Par exemple, est vrai si vous compilez avec des assertions de débogage activées. `true cfg ! ( debug_assertions )`

Cette macro prend en charge exactement la même syntaxe que l'attribut décrit dans [« Attributs »](#), mais au lieu de la compilation conditionnelle, vous obtenez une réponse vraie ou fausse. `#[cfg(...)]`

```
env!( "VAR_NAME" )
```

Se développe en une chaîne : valeur de la variable d'environnement spécifiée au moment de la compilation. Si la variable n'existe pas, il s'agit d'une erreur de compilation.

Cela ne vaudrait pas grand-chose, sauf que Cargo définit plusieurs variables d'environnement intéressantes lorsqu'il compile une caisse. Par exemple, pour obtenir la chaîne de version actuelle de votre caisse, vous pouvez écrire :

```
let version = env!( "CARGO_PKG_VERSION" );
```

Une liste complète de ces variables d'environnement est incluse dans la [documentation Cargo](#).

```
option_env!( "VAR_NAME" )
```

C'est la même chose que sauf qu'il renvoie un `Option` qui est si la variable spécifiée n'est pas définie. `env! Option<'static str> None`

Trois autres macros intégrées vous permettent d'importer du code ou des données à partir d'un autre fichier :

```
include!( "file.rs" )
```

Développe le contenu du fichier spécifié, qui doit être du code Rust valide, soit une expression, soit une séquence [d'éléments](#).

```
include_str!( "file.txt" )
```

Se développe jusqu'à un contenant le texte du fichier spécifié. Vous pouvez l'utiliser comme ceci: `&'static str`

```
const COMPOSITOR_SHADER: &str =  
    include_str!( "../resources/compositor.glsl" );
```

Si le fichier n'existe pas ou n'est pas valide UTF-8, vous obtiendrez une erreur de compilation.

```
include_bytes!( "file.dat" )
```

C'est la même chose, sauf que le fichier est traité comme des données binaires, et non comme du texte UTF-8. Le résultat est un fichier `&'static [u8]`

Comme toutes les macros, celles-ci sont traitées au moment de la compilation. Si le fichier n'existe pas ou ne peut pas être lu, la compilation échoue. Ils ne peuvent pas échouer au moment de l'exécution. Dans tous les cas, si le nom de fichier est un chemin d'accès relatif, il est résolu par rapport au répertoire qui contient le fichier actif.

Rust fournit également plusieurs macros pratiques que nous n'avons pas couvertes auparavant:

```
todo!(), unimplemented!()
```

Ceux-ci sont équivalents à `panic!`, mais transmettent une intention différente. `panic!` va dans les clauses, les armes et d'autres cas qui ne sont pas encore traités. Il panique toujours. `todo!` est à peu près la même chose, mais transmet l'idée que ce code n'a tout simplement pas encore été écrit; certains IDE le signalent pour avis. `unimplemented!` est à peu près la même chose, mais transmet l'idée que ce code n'a tout simplement pas encore été écrit; certains IDE le signalent pour avis.

```
matches!(value, pattern)
```

Compare une valeur à un modèle et renvoie si elle correspond ou non. C'est l'équivalent d'écrire `value == pattern`.

```
match value {  
    pattern => true,  
    _ => false  
}
```

Si vous recherchez un exercice d'écriture de macro de base, c'est une bonne macro à répliquer, d'autant plus que l'implémentation réelle, que vous pouvez voir dans la documentation standard de la bibliothèque, est assez simple.

## Débogage des macros

Le débogage d'une macro capricieuse peut être difficile. Le plus gros problème est le manque de visibilité sur le processus d'expansion macro. Rust va souvent développer toutes les macros, trouver une sorte d'erreur, puis imprimer un message d'erreur qui n'affiche pas le code entièrement développé qui contient l'erreur!

Voici trois outils pour vous aider à dépanner les macros. (Ces fonctionnalités sont toutes instables, mais comme elles sont vraiment conçues pour être utilisées pendant le développement, pas dans le code que vous archiveriez, ce n'est pas un gros problème dans la pratique.)

Tout d'abord, et le plus simple, vous pouvez demander à montrer à quoi ressemble votre code après avoir développé toutes les macros. Permet de

voir comment Cargo appelle . Copiez la ligne de commande et ajoutez-la en tant qu'options. Le code entièrement étendu est vidé sur votre terminal. Malheureusement, cela ne fonctionne que si votre code est exempt d'erreurs de syntaxe. `rustc cargo build --verbose rustc rustc -Z unstable-options --pretty expanded`

Deuxièmement, Rust fournit une macro qui imprime simplement ses arguments sur le terminal au moment de la compilation. Vous pouvez l'utiliser pour le débogage de style -. Cette macro nécessite l'indicateur de fonctionnalité. `log_syntax!() println! #[feature(log_syntax)]`

Troisièmement, vous pouvez demander au compilateur Rust d'enregistrer tous les appels de macro sur le terminal. Insérez quelque part dans votre code. À partir de ce moment, chaque fois que Rust développe une macro, il imprime le nom et les arguments de la macro. Par exemple, considérez ce programme : `trace_macros!(true);`

```
#[feature(trace_macros)]

fn main() {
    trace_macros!(true);
    let numbers = vec![1, 2, 3];
    trace_macros!(false);
    println!("total: {}", numbers.iter().sum::<u64>());
}
```

Il produit cette sortie:

```
$ rustup override set nightly
...
$ rustc trace_example.rs
note: trace_macro
--> trace_example.rs:5:19
|
5 |         let numbers = vec![1, 2, 3];
|                               ^^^^^^^^^^^^^^^
|
= note: expanding `vec! { 1 , 2 , 3 }`
= note: to `< [ _ ] > :: into_vec ( box [ 1 , 2 , 3 ] )`
```

Le compilateur affiche le code de chaque appel de macro, avant et après l'expansion. La ligne désactive à nouveau le traçage, de sorte que l'appel à n'est pas tracé. `trace_macros!(false); println!()`

## Construire le json! Macro

Nous avons maintenant discuté des fonctionnalités de base de `serde_json`. Dans cette section, nous allons développer de manière incrémentielle une macro pour créer des données JSON. Nous utiliserons cet exemple pour montrer ce que c'est que de développer une macro, présenter les quelques éléments restants de `serde`, et offrir quelques conseils sur la façon de s'assurer que vos macros se comportent comme souhaité.

Au [chapitre 10](#), nous avons présenté cet énumérateur pour représenter les données JSON :

```
#[derive(Clone, PartialEq, Debug)]
enum Json {
    Null,
    Boolean(bool),
    Number(f64),
    String(String),
    Array(Vec<Json>),
    Object(Box<HashMap<String, Json>>)
}
```

La syntaxe d'écriture des valeurs est malheureusement plutôt verbeuse :

```
let students = Json::Array(vec![
    Json::Object(Box::new(vec![
        ("name".to_string(), Json::String("Jim Blandy".to_string())),
        ("class_of".to_string(), Json::Number(1926.0)),
        ("major".to_string(), Json::String("Tibetan throat singing".to_string()))
    ].into_iter().collect()),
    Json::Object(Box::new(vec![
        ("name".to_string(), Json::String("Jason Orendorff".to_string())),
        ("class_of".to_string(), Json::Number(1702.0)),
        ("major".to_string(), Json::String("Knots".to_string()))
    ].into_iter().collect())
]);
```

Nous aimerions pouvoir écrire ceci en utilisant une syntaxe plus JSON :

```
let students = json!([
    {
        "name": "Jim Blandy",
        "class_of": 1926,
        "major": "Tibetan throat singing"
    },
    {
```

```

        "name": "Jason Orendorff",
        "class_of": 1702,
        "major": "Knots"
    }
}
);

```

Ce que nous voulons, c'est une macro qui prend une valeur JSON comme argument et se développe à une expression Rust comme celle de l'exemple précédent. `json!`

## Types de fragments

Le premier travail d'écriture d'une macro complexe consiste à déterminer comment faire *correspondre ou analyser* l'entrée souhaitée.

Nous pouvons déjà voir que la macro aura plusieurs règles, car il y a plusieurs sortes de choses différentes dans les données JSON : objets, tableaux, nombres, etc. En fait, nous pourrions deviner que nous aurons une règle pour chaque type JSON :

```

macro_rules! json {
    (null)      => { Json::Null };
    ([ ... ]) => { Json::Array(...) };
    ({ ... }) => { Json::Object(...) };
    (???)      => { Json::Boolean(...) };
    (???)      => { Json::Number(...) };
    (???)      => { Json::String(...) };
}

```

Ce n'est pas tout à fait correct, car les modèles macro n'offrent aucun moyen de séparer les trois derniers cas, mais nous verrons comment y faire face plus tard. Les trois premiers cas, au moins, commencent clairement par des jetons différents, alors commençons par ceux-ci.

La première règle fonctionne déjà :

```

macro_rules! json {
    (null) => {
        Json::Null
    }
}

#[test]
fn json_null() {
    assert_eq!(json!(null), Json::Null); // passes!
}

```



Pour ajouter la prise en charge des tableaux JSON, nous pouvons essayer de faire correspondre les éléments en tant que `s : expr`

```
macro_rules! json {
    (null) => {
        Json::Null
    };
    ([ $( $element:expr ),* ]) => {
        Json::Array(vec![ $( $element ),* ])
    };
}
```

Malheureusement, cela ne correspond pas à tous les tableaux JSON. Voici un test qui illustre le problème :

```
#[test]
fn json_array_with_json_element() {
    let macro_generated_value = json!(
        [
            // valid JSON that doesn't match `$(element:expr)`
            {
                "pitch": 440.0
            }
        ]
    );
    let hand_coded_value =
        Json::Array(vec![
            Json::Object(Box::new(vec![
                ("pitch".to_string(), Json::Number(440.0))
            ]).into_iter().collect())
        ]);
    assert_eq!(macro_generated_value, hand_coded_value);
}
```

Le motif signifie « une liste d’expressions Rust séparées par des virgules ». Mais de nombreuses valeurs JSON, en particulier les objets, ne sont pas des expressions Rust valides. Ils ne correspondront pas. `$( $element:expr ),*`

Étant donné que tous les bits de code que vous souhaitez faire correspondre ne sont pas une expression, Rust prend en charge plusieurs autres types de fragments, répertoriés dans [le tableau 21-2](#).

Tableau 21-2. Types de fragments pris en charge par `macro_rules!`

Type de fragment	Correspondances (avec exemples)	Peut être suivi de...
<code>expr</code>	Une expression : <code>2 + 2</code> , <code>"udon"</code> <code>x.len()</code>	<code>=&gt;</code> <code>,</code> <code>;</code>
<code>stmt</code>	Expression ou déclaration, à l'exclusion d'un point-virgule de fin (difficile à utiliser; essayez ou à la place) <code>expr block</code>	<code>=&gt;</code> <code>,</code> <code>;</code>
<code>ty</code>	Un type : <code>String</code> , <code>Vec&lt;u8&gt; (&amp;str, bool)</code> <code>dyn Read + Send</code>	<code>=&gt;</code> <code>,</code> <code>;</code> <code>=</code> <code> </code> <code>{</code> <code>[</code> <code>:</code> <code>&gt;</code> <code>as</code> <code>where</code>
<code>path</code>	Un chemin (discuté) : <code>ferns</code> , <code>::std::sync::mpsc</code>	<code>=&gt;</code> <code>,</code> <code>;</code> <code>=</code> <code> </code> <code>{</code> <code>[</code> <code>:</code> <code>&gt;</code> <code>as</code> <code>where</code>
<code>pat</code>	Un modèle (discuté) : <code>_</code> , <code>Some(ref x)</code>	<code>=&gt;</code> <code>,</code> <code>=</code> <code> </code> <code>if</code> <code>i</code> <code>n</code>
<code>item</code>	Un point (discuté) : <code>struct Point { x: f64, y: f64 }, mod ferns;</code>	Rien
<code>block</code>	Un bloc (discuté) : <code>{ s += "ok\n"; true }</code>	Rien
<code>meta</code>	Le corps d'un attribut (discuté) : <code>inline</code> , <code>derive(Copy, Clone)</code> <code>doc="3D models."</code>	Rien
<code>literal</code>	Valeur littérale : <code>1024</code> , <code>"Hello, world!"</code> <code>1_000_000f64</code>	Rien
<code>lifetime</code>	Une vie : <code>'a</code> , <code>'item</code> <code>'static</code>	Rien

Type de fragment	Correspondances (avec exemples)	Peut être suivi de...
<code>vis</code>	Un spécificateur de visibilité : <code>pub,, pub(crate) pub(in module::submodule)</code>	Rien
<code>ident</code>	Un identifiant : <code>std,, Json longish_variable_name</code>	Rien
<code>tt</code>	Une arborescence de jetons (voir texte) : <code>;,,, &gt;= {} [0 1 (+ 0 1)]</code>	Rien

La plupart des options de ce tableau appliquent strictement la syntaxe Rust. Le type correspond uniquement aux expressions Rust (pas aux valeurs JSON), ne correspond qu'aux types Rust, etc. Ils ne sont pas extensibles : il n'y a aucun moyen de définir de nouveaux opérateurs arithmétiques ou de nouveaux mots-clés qui reconnaîtraient. Nous ne serons pas en mesure de faire correspondre l'une de ces données JSON arbitraires. `expr ty expr`

Les deux derniers, `et` , prennent en charge les arguments de macro correspondants qui ne ressemblent pas au code Rust. `correspond` à n'importe quel identificateur. `correspond` à un seul *arbre de jetons* : soit une paire de crochets correctement assortie, `,` ou `,` et tout ce qui se trouve entre les deux, y compris les arbres de jetons imbriqués, soit un seul jeton qui n'est pas un crochet, comme `ou`. `ident tt ident tt (...)` `[...]` `{...}` `1926 "Knots"`

Les arbres à jetons sont exactement ce dont nous avons besoin pour notre macro. Chaque valeur JSON est une arborescence de jeton unique : nombres, chaînes, valeurs booléennes et sont tous des jetons uniques ; les objets et les tableaux sont entre crochets. Nous pouvons donc écrire les modèles comme ceci: `json! null`

```
macro_rules! json {
  (null) => {
    Json::Null
  };
  ([ $( $element:tt ),* ]) => {
    Json::Array(...)
  }
}
```

```

};
({ $( $key:tt : $value:tt ),* }) => {
    Json::Object(...)
};
($other:tt) => {
    ... // TODO: Return Number, String, or Boolean
};
}

```

Cette version de la macro peut correspondre à toutes les données JSON. Maintenant, il ne nous reste plus qu'à produire le code Rust correct. `json!`

Pour s'assurer que Rust peut acquérir de nouvelles fonctionnalités syntaxiques à l'avenir sans casser les macros que vous écrivez aujourd'hui, Rust restreint les jetons qui apparaissent dans des modèles juste après un fragment. Le « Peut être suivi de... » du [tableau 21-2](#) indique quels jetons sont autorisés. Par exemple, le modèle est une erreur, car il n'est pas autorisé après un fichier. Le motif est OK, car il est suivi de la flèche, l'un des jetons autorisés pour un `,` et est suivi de rien, ce qui est toujours autorisé. `$x:expr ~ $y:expr ~ expr $vars:pat =>`  
`$handler:expr $vars:pat => pat $handler:expr`

## Récurtivité dans les macros

Vous avez déjà vu un cas trivial d'une macro s'appelant elle-même : notre implémentation utilise la récursivité pour prendre en charge les virgules de fin. Ici, nous pouvons montrer un exemple plus significatif: doit s'appeler de manière récursive. `vec! json!`

Nous pourrions essayer de prendre en charge les baies JSON sans utiliser la récursivité, comme ceci :

```

([ $( $element:tt ),* ]) => {
    Json::Array(vec![ $( $element ),* ])
};

```

Mais cela ne fonctionnerait pas. Nous collerions des données JSON (les arbres de jetons) directement dans une expression Rust. Ce sont deux langues différentes. `$element`

Nous devons convertir chaque élément du tableau de la forme JSON en Rust. Heureusement, il y a une macro qui fait ça : celle que nous écrivons !

```
([ $( $element:tt ),* ]) => {
    Json::Array(vec![ $( json!($element) ),* ])
};
```

Les objets peuvent être pris en charge de la même manière :

```
({ $( $key:tt : $value:tt ),* }) => {
    Json::Object(Box::new(vec![
        $( ( $key.to_string(), json!($value) ) ),*
    ].into_iter().collect()))
};
```

Le compilateur impose une limite de récursivité aux macros : 64 appels, par défaut. C'est plus que suffisant pour les utilisations normales de , mais les macros récursives complexes atteignent parfois la limite. Vous pouvez l'ajuster en ajoutant cet attribut en haut de la caisse où la macro est utilisée : json!

```
#![recursion_limit = "256"]
```

Notre macro est presque terminée. Il ne reste plus qu'à prendre en charge les valeurs booléennes, numériques et de chaîne. json!

## Utilisation de traits avec des macros

Écrire des macros complexes pose toujours des énigmes. Il est important de se rappeler que les macros elles-mêmes ne sont pas le seul outil de résolution d'énigmes à votre disposition.

Ici, nous devons soutenir , et , convertir la valeur, quelle qu'elle soit, en un type de valeur approprié. Mais les macros ne sont pas bonnes pour distinguer les types. On peut imaginer écrire : json!(true) json!(1.0) json!("yes") Json

```
macro_rules! json {
    (true) => {
        Json::Boolean(true)
    };
    (false) => {
        Json::Boolean(false)
    };
    ...
}
```

Cette approche s'effondre tout de suite. Il n'y a que deux valeurs booléennes, mais plutôt plus de nombres que cela, et encore plus de chaînes.

Heureusement, il existe un moyen standard de convertir des valeurs de différents types en un type spécifié: le trait, couvert . Nous devons simplement implémenter ce trait pour quelques types: From

```
impl From<bool> for Json {
    fn from(b: bool) -> Json {
        Json::Boolean(b)
    }
}

impl From<i32> for Json {
    fn from(i: i32) -> Json {
        Json::Number(i as f64)
    }
}

impl From<String> for Json {
    fn from(s: String) -> Json {
        Json::String(s)
    }
}

impl<'a> From<&'a str> for Json {
    fn from(s: &'a str) -> Json {
        Json::String(s.to_string())
    }
}

...
```

En fait, les 12 types numériques devraient avoir des implémentations très similaires, il pourrait donc être logique d'écrire une macro, juste pour éviter le copier-coller:

```
macro_rules! impl_from_num_for_json {
    ( $( $t:ident ) * ) => {
        $(
            impl From<$t> for Json {
                fn from(n: $t) -> Json {
                    Json::Number(n as f64)
                }
            }
        ) *
    };
}
```

```

}

impl_from_num_for_json!(u8 i8 u16 i16 u32 i32 u64 i64 u128 i128
                        usize isize f32 f64);

```

Maintenant, nous pouvons utiliser pour convertir un de n'importe quel type pris en charge en `Json`. Dans notre macro, cela ressemblera à ceci:

```

Json::from(value) value Json

```

```

( $other:tt ) => {
    Json::from($other) // Handle Boolean/number/string
};

```

L'ajout de cette règle à notre macro lui permet de réussir tous les tests que nous avons écrits jusqu'à présent. En rassemblant toutes les pièces, il ressemble actuellement à ceci:

```

json!

```

```

macro_rules! json {
    (null) => {
        Json::Null
    };
    ([ $( $element:tt ),* ]) => {
        Json::Array(vec![ $( json!($element) ),* ])
    };
    ({ $( $key:tt : $value:tt ),* }) => {
        Json::Object(Box::new(vec![
            $( ($key.to_string(), json!($value)) ),*
        ]).into_iter().collect())
    };
    ( $other:tt ) => {
        Json::from($other) // Handle Boolean/number/string
    };
}

```

Il s'avère que la macro prend en charge de manière inattendue l'utilisation de variables et même d'expressions Rust arbitraires dans les données JSON, une fonctionnalité supplémentaire pratique:

```

let width = 4.0;
let desc =
    json!({
        "width": width,
        "height": (width * 9.0 / 4.0)
    });

```

Parce qu'il s'agit d'une arborescence de jetons unique, la macro lui correspond donc avec succès lors de l'analyse de l'objet. `(width * 9.0 / 4.0) $value:tt`

## Cadrage et hygiène

Un aspect étonnamment délicat de l'écriture de macros est qu'elles impliquent de coller du code de différentes étendues ensemble. Ainsi, les pages suivantes couvrent les deux façons dont Rust gère la portée: une façon pour les variables et les arguments locaux, et une autre façon pour tout le reste.

Pour montrer pourquoi cela est important, réécrivons notre règle d'analyse des objets JSON (la troisième règle de la macro montrée précédemment) pour éliminer le vecteur temporaire. Nous pouvons l'écrire comme ceci: `json!`

```
({ $($key:tt : $value:tt),* }) => {  
  {  
    let mut fields = Box::new(HashMap::new());  
    $( fields.insert($key.to_string(), json!($value)); )*  
    Json::Object(fields)  
  }  
};
```

Maintenant, nous remplissons le non pas en utilisant mais en appelant à plusieurs reprises la méthode. Cela signifie que nous devons stocker la carte dans une variable temporaire, que nous avons appelée `.HashMap collect() .insert() fields`

Mais alors que se passe-t-il si le code qui appelle utilise une variable qui lui est propre, également nommée ? `json! fields`

```
let fields = "Fields, W.C.";  
let role = json!({  
  "name": "Larson E. Whipsnade",  
  "actor": fields  
});
```

Développer la macro collerait ensemble deux bits de code, les deux en utilisant le nom pour des choses différentes! `fields`

```
let fields = "Fields, W.C.";  
let role = {  
  let mut fields = Box::new(HashMap::new());  
  fields.insert("name".to_string(), Json::from("Larson E. Whipsnade"));  
};
```



```

        fields.insert("actor".to_string(), Json::from(fields));
        Json::Object(fields)
    };

```

Cela peut sembler un piège inévitable chaque fois que les macros utilisent des variables temporaires, et vous réfléchissez peut-être déjà aux correctifs possibles. Peut-être devrions-nous renommer la variable que la macro définit en quelque chose que ses appelants ne sont pas susceptibles de transmettre: au lieu de `fields`, nous pourrions l'appeler

```
__json! fields __json$fields
```

La surprise ici est que *la macro fonctionne telle quelle*. Rust renomme la variable pour vous! Cette fonctionnalité, d'abord implémentée dans les macros Scheme, s'appelle *hygiène*, et on dit donc que Rust a des *macros hygiéniques*.

La façon la plus simple de comprendre l'hygiène macro est d'imaginer que chaque fois qu'une macro est développée, les parties de l'expansion qui proviennent de la macro elle-même sont peintes d'une couleur différente.

Les variables de différentes couleurs sont alors traitées comme si elles avaient des noms différents :

```

let fields = "Fields, W.C.";
let role = {
    let mut fields = Box::new(HashMap::new());
    fields.insert("name".to_string(), Json::from("Larson E. Whipsnade"));
    fields.insert("actor".to_string(), Json::from(fields));
    Json::Object(fields)
};

```

Notez que les bits de code qui ont été transmis par l'appelant de macro et collés dans la sortie, tels que `et`, conservent leur couleur d'origine (noir). Seuls les jetons provenant du modèle de macro sont peints. `"name"` `"actor"`

Maintenant, il y a une variable nommée (déclarée dans l'appelant) et une variable distincte nommée (introduite par la macro). Comme les noms sont de couleurs différentes, les deux variables ne sont pas confondues. `fields` `fields`

Si une macro a vraiment besoin de faire référence à une variable dans la portée de l'appelant, l'appelant doit transmettre le nom de la variable à la macro.

(La métaphore de la peinture n'est pas censée être une description exacte du fonctionnement de l'hygiène. Le mécanisme réel est même un peu plus intelligent que cela, reconnaissant deux identificateurs comme identiques, indépendamment de la « peinture », s'ils se réfèrent à une variable commune qui est à la fois dans la portée de la macro et de son appelant. Mais des cas comme celui-ci sont rares à Rust. Si vous comprenez l'exemple précédent, vous en savez assez pour utiliser des macros hygiéniques.)

Vous avez peut-être remarqué que de nombreux autres identificateurs ont été peints d'une ou plusieurs couleurs au fur et à mesure que les macros étaient développées : `!`, `et`, `,` par exemple. Malgré la peinture, Rust n'a eu aucun mal à reconnaître ces noms de type. C'est parce que l'hygiène dans Rust est limitée aux variables et arguments locaux. En ce qui concerne les constantes, les types, les méthodes, les modules, la statique et les noms de macros, Rust est « daltonien ». `Box HashMap Json`

Cela signifie que si notre macro est utilisée dans un module où `!`, `et`, ou n'est pas dans la portée, la macro ne fonctionnera pas. Nous montrerons comment éviter ce problème dans la section suivante. `json! Box HashMap Json`

Tout d'abord, nous examinerons un cas où l'hygiène stricte de Rust nous gêne, et nous devons le contourner. Supposons que nous ayons de nombreuses fonctions qui contiennent cette ligne de code :

```
let req = ServerRequest::new(server_socket.session());
```

Copier et coller cette ligne est pénible. Peut-on utiliser une macro à la place ?

```
macro_rules! setup_req {
    () => {
        let req = ServerRequest::new(server_socket.session());
    }
}

fn handle_http_request(server_socket: &ServerSocket) {
    setup_req!(); // declares `req`, uses `server_socket`
    ... // code that uses `req`
}
```

Tel qu'il est écrit, cela ne fonctionne pas. Il faudrait que le nom dans la macro fasse référence au local déclaré dans la fonction, et vice versa pour la variable `!`. Mais l'hygiène empêche les noms dans les macros de « se heurter » avec des noms dans d'autres étendues, même dans des cas

comme celui-ci, où c'est ce que vous voulez.

```
server_socket server_socket req
```

La solution consiste à transmettre à la macro tous les identificateurs que vous prévoyez d'utiliser à l'intérieur et à l'extérieur du code de macro :

```
macro_rules! setup_req {
    ($req:ident, $server_socket:ident) => {
        let $req = ServerRequest::new($server_socket.session());
    }
}

fn handle_http_request(server_socket: &ServerSocket) {
    setup_req!(req, server_socket);
    ... // code that uses `req`
}
```

Depuis et sont maintenant fournis par la fonction, ils sont la bonne « couleur » pour cette portée.

```
req server_socket
```

L'hygiène rend cette macro un peu plus verbeuse à utiliser, mais c'est une fonctionnalité, pas un bug: il est plus facile de raisonner sur les macros hygiéniques en sachant qu'elles ne peuvent pas jouer avec les variables locales derrière votre dos. Si vous recherchez un identifiant comme dans une fonction, vous trouverez tous les endroits où il est utilisé, y compris les appels de macro.

```
server_socket
```

## Importation et exportation de macros

Étant donné que les macros sont développées au début de la compilation, avant que Rust ne connaisse la structure complète du module de votre projet, le compilateur dispose d'affordances spéciales pour les exporter et les importer.

Les macros visibles dans un module sont automatiquement visibles dans ses modules enfants. Pour exporter des macros d'un module « vers le haut » vers son module parent, utilisez l'attribut. Par exemple, supposons que notre *lib.rs* ressemble à ceci : `#[macro_use]`

```
#[macro_use] mod macros;
mod client;
mod server;
```

Toutes les macros définies dans le module sont importées dans *lib.rs* et donc visibles dans le reste de la caisse, y compris dans et

```
.macros client server
```

Les macros marquées par `pub` sont automatiquement et peuvent être référencées par chemin, comme d'autres éléments. `#`

```
[macro_export] pub
```

Par exemple, la caisse fournit une macro appelée `lazy_static`, qui est marquée par `pub`.

Pour utiliser cette macro dans votre propre caisse, vous devez écrire

```
:lazy_static lazy_static #[macro_export]
```

```
use lazy_static::lazy_static;
lazy_static!{ }
```

Une fois qu'une macro est importée, elle peut être utilisée comme n'importe quel autre élément :

```
use lazy_static::lazy_static;

mod m {
    crate::lazy_static!{ }
```

Bien sûr, faire l'une de ces choses signifie que votre macro peut être appelée dans d'autres modules. Une macro exportée ne doit donc pas dépendre de quoi que ce soit dans la portée - il est impossible de dire ce qui sera dans la portée où elle est utilisée. Même les caractéristiques du prélude standard peuvent être ombragées.

Au lieu de cela, la macro doit utiliser des chemins absolus vers tous les noms qu'elle utilise. fournit le fragment spécial pour aider à cela. Ce n'est pas la même chose que `pub`, qui est un mot-clé qui peut être utilisé dans les chemins n'importe où, pas seulement dans les macros. agit comme un chemin absolu vers le module racine de la caisse où la macro a été définie. Au lieu de dire `pub`, on peut écrire `pub use`, ce qui fonctionne même si `Json` n'a pas été importé. peut être remplacé par `pub use`. Dans ce dernier cas, nous devons réexporter `pub use`, car `pub` ne peut pas être utilisé pour accéder aux fonctionnalités privées d'une caisse. Il s'étend vraiment à quelque chose comme `pub use`, un chemin ordinaire. Les règles de visibilité ne sont pas affectées.

```
macro_rules! $crate crate $crate Json $crate::Json Hash
Map ::std::collections::HashMap $crate::macros::HashMap Has
hMap $crate ::jsonlib
```

Après avoir déplacé la macro vers son propre module et l'avoir modifiée pour l'utiliser, elle ressemble à ceci. Voici la version finale

```
:macros $crate
```

```

// macros.rs
pub use std::collections::HashMap;
pub use std::boxed::Box;
pub use std::string::ToString;

#[macro_export]
macro_rules! json {
    (null) => {
        $crate::Json::Null
    };
    ([ $( $element:tt ),* ]) => {
        $crate::Json::Array(vec![ $( json!($element) ),* ])
    };
    ({ $( $key:tt : $value:tt ),* }) => {
        {
            let mut fields = $crate::macros::Box::new(
                $crate::macros::HashMap::new() );
            $(
                fields.insert($crate::macros::ToString::to_string($key),
                               json!($value));
            )*
            $crate::Json::Object(fields)
        }
    };
    ($other:tt) => {
        $crate::Json::from($other)
    };
}

```

Étant donné que la méthode fait partie du trait standard, nous utilisons également pour nous y référer, en utilisant la syntaxe que nous avons introduite dans [« Appels de méthode complets »](#). Dans notre cas, ce n'est pas strictement nécessaire pour que la macro fonctionne, car c'est dans le prélude standard. Mais si vous appelez des méthodes d'un trait qui n'est peut-être pas dans la portée au point où la macro est appelée, un appel de méthode complet est le meilleur moyen de le

```

faire.to_string() ToString $crate $crate::macros::ToString:
:to_string($key) ToString

```

## Éviter les erreurs de syntaxe lors de la correspondance

La macro suivante semble raisonnable, mais elle pose quelques problèmes à Rust :

```
macro_rules! complain {
    ($msg:expr) => {
        println!("Complaint filed: {}", $msg)
    };
    (user : $userid:tt , $msg:expr) => {
        println!("Complaint from user {}: {}", $userid, $msg)
    };
}
```

Supposons que nous l'appelions comme ceci:

```
complain!(user: "jimb", "the AI lab's chatbots keep picking on me");
```

Pour les yeux humains, cela correspond évidemment au deuxième modèle. Mais Rust essaie d'abord la première règle, en essayant de faire correspondre toutes les entrées avec . C'est là que les choses commencent à mal tourner pour nous. n'est pas une expression, bien sûr, donc nous obtenons une erreur de syntaxe. Rust refuse de balayer une erreur de syntaxe sous le tapis - les macros sont déjà assez difficiles à déboguer. Au lieu de cela, il est signalé immédiatement et la compilation s'arrête. \$msg:expr user: "jimb"

Si un autre jeton d'un modèle ne correspond pas, Rust passe à la règle suivante. Seules les erreurs de syntaxe sont fatales, et elles ne se produisent que lorsque vous essayez de faire correspondre des fragments.

Le problème ici n'est pas si difficile à comprendre: nous essayons de faire correspondre un fragment, , dans la mauvaise règle. Ça ne va pas correspondre parce que nous ne sommes même pas censés être ici. L'appelant voulait l'autre règle. Il existe deux façons simples d'éviter cela. \$msg:expr

Tout d'abord, évitez les règles confuses. Nous pourrions, par exemple, modifier la macro afin que chaque modèle commence par un identificateur différent :

```
macro_rules! complain {
    (msg : $msg:expr) => {
        println!("Complaint filed: {}", $msg);
    };
    (user : $userid:tt , msg : $msg:expr) => {
        println!("Complaint from user {}: {}", $userid, $msg);
    };
}
```

Lorsque les arguments de macro commencent par `,` nous obtenons la règle 1. Quand ils commenceront par `,` nous obtiendrons la règle 2. Quoi qu'il en soit, nous savons que nous avons la bonne règle avant d'essayer de faire correspondre un fragment. `msg user`

L'autre façon d'éviter les erreurs de syntaxe fallacieuses est de mettre en premier des règles plus spécifiques. Mettre la règle en premier résout le problème avec `,` car la règle qui provoque l'erreur de syntaxe n'est jamais atteinte. `user: complain!`

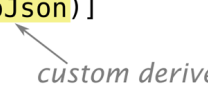
## Au-delà de `macro_rules!`

Les modèles de macro peuvent analyser des entrées encore plus complexes que JSON, mais nous avons constaté que la complexité devient rapidement incontrôlable.

[The Little Book of Rust Macros](#), par Daniel Keep et al., est un excellent manuel de programmation avancée. Le livre est clair et intelligent, et il décrit tous les aspects de l'expansion macro plus en détail que nous ne l'avons ici. Il présente également plusieurs techniques très intelligentes pour presser des motifs en service comme une sorte de langage de programmation ésotérique, pour analyser des entrées complexes. Nous sommes moins enthousiastes à ce sujet. Utiliser avec précaution. `macro_rules! macro_rules!`

Rust 1.15 a introduit un mécanisme distinct appelé *macros procédurales*. Les macros procédurales prennent en charge l'extension de l'attribut pour gérer les dérivations personnalisées, comme illustré à [la figure 21-4](#), ainsi que la création d'attributs personnalisés et de nouvelles macros appelées comme les macros décrites précédemment. `#[derive] macro_rules!`

```
#[derive(Copy, Clone, PartialEq, Eq, IntoJson)]
struct Money {
    dollars: u32,
    cents: u16,
}
```



Graphique 21-4. Appel d'une macro procédurale hypothétique via un attribut `IntoJson` `#[derive]`

Il n'y a pas de trait, mais ce n'est pas grave : une macro procédurale peut utiliser ce hook pour insérer le code qu'elle veut (dans ce cas, probablement). `IntoJson impl From<Money> for Json { ... }`

Ce qui rend une macro procédurale « procédurale », c'est qu'elle est implémentée en tant que fonction Rust, et non en tant qu'ensemble de rè-

gles déclaratives. Cette fonction interagit avec le compilateur à travers une fine couche d'abstraction et peut être arbitrairement complexe. Par exemple, la bibliothèque de base de données utilise des macros procédurales pour se connecter à une base de données et générer du code basé sur le schéma de cette base de données au moment de la compilation. `diesel`

Étant donné que les macros procédurales interagissent avec les internes du compilateur, l'écriture de macros efficaces nécessite une compréhension du fonctionnement du compilateur qui est hors de la portée de ce livre. Il est cependant largement couvert dans la [documentation en ligne](#).

Peut-être, après avoir lu tout cela, avez-vous décidé que vous détestez les macros. Et alors ? Une alternative consiste à générer du code Rust à l'aide d'un script de build. La [documentation Cargo](#) montre comment le faire étape par étape. Cela implique d'écrire un programme qui génère le code Rust que vous voulez, d'ajouter une ligne à *Cargo.toml* pour exécuter ce programme dans le cadre du processus de génération et d'utiliser pour obtenir le code généré dans votre caisse. `include!`

[Soutien](#)   [Se déconnecter](#)

©2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)