

Chapitre 21. Macros

Un cento (du latin pour "patchwork") est un poème composé entièrement de lignes citées 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 des fonctions seules. Par exemple, nous avons vu la `assert_eq!` macro, ce qui est pratique pour les tests :

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

Cela aurait pu être écrit comme une fonction générique, mais la `assert_eq!` macro fait plusieurs choses que les fonctions ne peuvent pas faire. La première est que lorsqu'une assertion échoue, `assert_eq!` 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 ces informations. Les macros le peuvent, car leur mode de fonctionnement est complètement différent.

Les macros sont une sorte de raccourci. Lors de 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 qu'il est remplacé par du code Rust. L'appel de macro précédent se développe 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 aussi une macro, qui lui-même s'étend à encore plus de code Rust (non montré ici). Ce code utilise deux autres macros, `file!()` et `line!()`. Une fois que chaque appel de macro dans la caisse est complètement développé, Rust passe à la phase suivante de compilation.

Au moment de l'exécution, un échec d'assertion ressemblerait à ceci (et indiquerait un bogue dans la `gcd()` fonction, puisque `2` c'est la bonne

réponse) :

```
thread 'main' panicked at 'assertion failed: `(left == right)`, (left: `17`,  
right: `2`)', gcd.rs:7
```

Si tu viens à partir 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 `syntax-rules`. 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 qu'ils ne peuvent pas être appelés accidentellement lorsque vous vouliez appeler une fonction. Les macros Rust n'insèrent jamais de crochets ou de parenthèses sans correspondance. Et les macros Rust sont livrées avec une correspondance de modèle, ce qui facilite l'écriture de macros à la fois maintenables et attrayantes à utiliser.

Dans ce chapitre, nous montrerons comment écrire des macros à l'aide de plusieurs exemples simples. Mais comme beaucoup de Rust, les macros récompensent une compréhension approfondie, 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 de 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 fonction encore plus puissante appelée *macros procédurales*.

Principes de base des macros

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

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

Toutes les macros ne sont pas définies de cette façon : quelques-unes, comme `file!`, `line!`, et elle-même `macro_rules!`, sont intégrées au 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 vôtre.

Une macro définie avec `macro_rules!` fonctionne entièrement par correspondance de modèle. Le corps d'une macro n'est qu'une série de règles :

```
( modèle1 ) => ( modèle1 );

( modèle2 ) => ( modèle2 );

...

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)
                }
            }
        }
    });
}
```

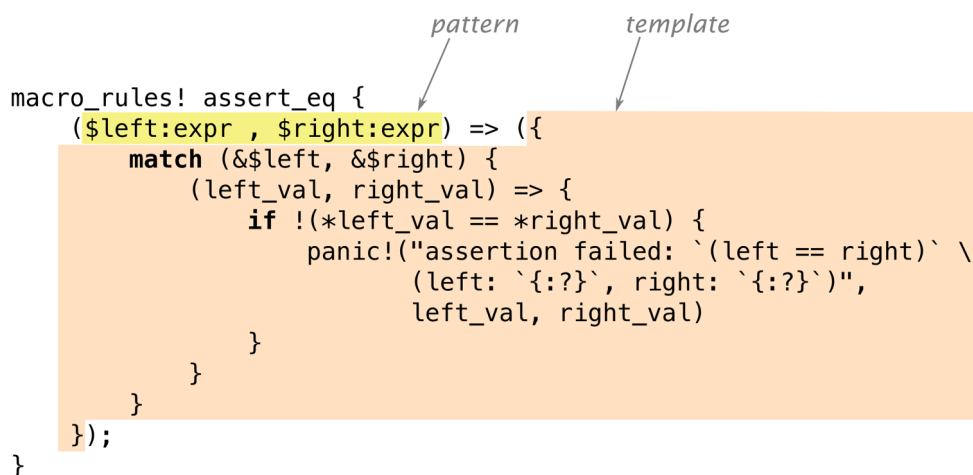


Illustration 21-1. La `assert_eq!` macro

La version de `assert_eq!` de la [Figure 21-1](#) n'a qu'un motif et un modèle.

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, elles sont toutes équivalentes :

```
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 pour appeler `assert_eq!`, des crochets pour `vec!` et des accolades pour `macro_rules!`.

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

- Nous expliquerons exactement comment Rust s'y prend pour trouver et étendre les définitions de macros dans votre programme.

- Nous soulignerons certaines 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'extension de macro

Rust développe les macros très tôt lors de la compilation. Le compilateur lit votre code source du début à la fin, définissant et 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 de regarder le reste du programme. (En revanche, les fonctions et autres [éléments](#) n'ont pas besoin d'être dans un ordre particulier. Il est normal d'appeler une fonction qui ne sera définie que plus tard dans la caisse.)

Lorsque Rust développe un `assert_eq!` appel de macro, ce qui se passe ressemble beaucoup à l'évaluation d'une `match` expression. Rust compare d'abord les arguments avec le motif, comme le montre la [figure 21-2](#).

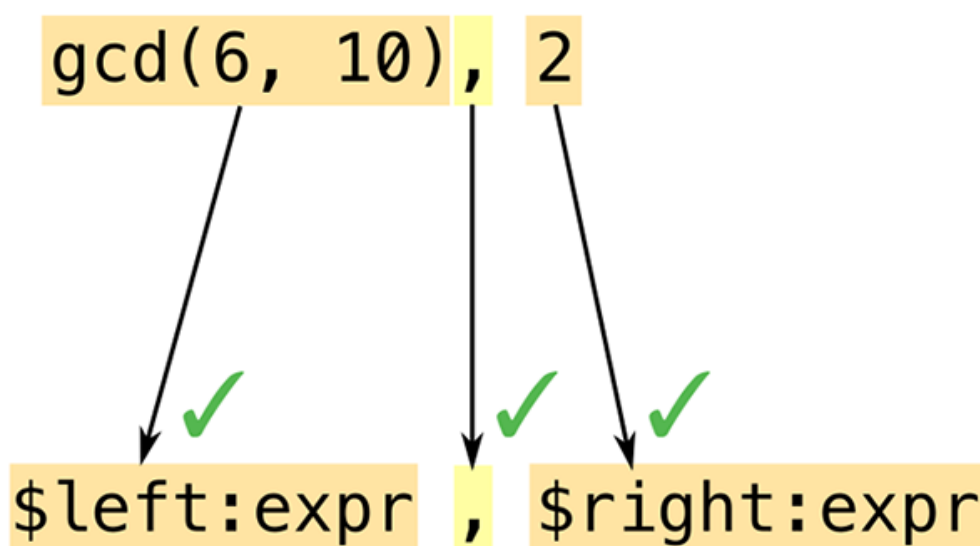


Illustration 21-2. Développer une macro, partie 1 : faire correspondre les arguments aux motifs

Les modèles de macro sont un mini-langage dans Rust. Ce sont essentiellement des expressions régulières pour le code correspondant. Mais là où les expressions régulières opèrent sur des caractères, les modèles opèrent sur des *jetons*— les chiffres, 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 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 apparaissent toujours par paires dans Rust. Ceci est vérifié avant que les macros ne soient développées, non seulement dans les modèles de macro mais dans tout le langage.

Dans cet exemple, notre modèle contient le *fragment* `$left:expr`, qui indique à Rust de faire correspondre une expression (dans ce cas, `gcd(6, 10)`) et de lui attribuer le nom `$left`. Rust fait alors correspondre la virgule du motif avec la virgule qui suit `gcd` les arguments de `.` Tout comme les expressions régulières, les modèles 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 `2` et lui donne le nom `$right`.

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

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

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

Illustration 21-3. Développer une macro, partie 2 : remplir le modèle

Rust remplace `$left` et `$right` par les fragments de code trouvés lors de la correspondance.

C'est une erreur courante d'inclure le type de fragment dans le modèle de sortie : écrire `$left:expr` plutôt que simplement `$left`. Rust ne détecte pas immédiatement ce genre d'erreur. Il considère `$left` comme une substitution, puis il traite `:expr` comme tout le reste du 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. alors il générera une fausse sortie qui ne sera pas compilée. Si vous obtenez des messages d'erreur comme `cannot find type `expr` in this scope` et `help: maybe you meant to use a path separator here` lors de l'utilisa-

tion d'une nouvelle macro, vérifiez-la pour cette erreur. (["Debugging Macros"](#) offre des conseils plus généraux pour des situations comme celle-ci.)

Les modèles de macros ne sont pas très différents des douze langages de modèles couramment utilisés dans la programmation Web. La seule différence - et elle est significative - est que la sortie est du code Rust.

Conséquences inattendues

Bouclage des 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 premier abord. La macro que nous avons examinée, `assert_eq!`, contient des morceaux de code un peu étranges pour des raisons qui en disent long sur la programmation de macros. Regardons deux morceaux amusants en particulier.

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

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

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

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 ceci ?

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

```

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 "s"
}

```

Puisque nous ne voulons pas que les assertions déplacent les valeurs, la macro emprunte des références à la place.

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

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 connues pour la plupart des programmeurs C++, et cela ne vaut pas la peine d'être expliqué en détail ici, un code banal comme `ADD_ONE(1) * 10` ou `ADD_ONE(1 << 4)` produit des résultats très surprenants avec cette macro. Pour résoudre ce problème, vous ajouteriez plus de parenthèses à la définition de la macro. Ce n'est pas nécessaire dans Rust, car les macros Rust sont mieux intégrées au langage. Rust sait quand il gère les expressions, donc il ajoute efficacement des parenthèses chaque fois qu'il colle une expression dans une autre.

Répétition

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

```
// 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 implémenté 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 `1`, `2`, `3` avec le modèle de la première règle, dans ce cas `$elem: expr ; $n: expr`. Cela ne correspond pas : `1` est une expression, mais le modèle nécessite un point-virgule après cela, et nous n'en avons pas. Alors Rust passe ensuite à la deuxième règle, et ainsi de suite. Si aucune règle ne correspond, c'est une erreur.

La première règle gère les utilisations comme `vec![0u8; 1000]`. Il se trouve qu'il existe une fonction standard (mais non documentée) `std::vec::from_elem`, qui fait exactement ce qui est nécessaire ici, donc cette règle est simple.

La deuxième règle gère `vec!["udon", "ramen", "soba"]`. Le motif, `$($x: expr),*`, utilise une fonctionnalité inédite : la répétition. Il correspond à 0 ou plusieurs expressions, séparées par des virgules. Plus généralement, la syntaxe `$(PATTERN),*` est utilisée pour faire correspondre n'importe quelle liste séparée par des virgules, où chaque élément de la liste correspond à `PATTERN`.

Le `*` ici a la même signification que dans les expressions régulières ("0 ou plus") bien que les regexps n'aient pas de `,*` répéteur spécial. Vous pouvez également utiliser `+` pour exiger au moins une correspondance, ou `?`

pour zéro ou une correspondance. [Le tableau 21-1](#) donne la suite complète des modèles de répétition.

Tableau 21-1. Motifs de répétition

Motif	Sens
<code>\$ (...) *</code>	Correspondance 0 fois ou plus sans séparateur
<code>\$ (...) , *</code>	Match 0 ou plusieurs fois, séparés par des virgules
<code>\$ (...) ; *</code>	Correspondance 0 fois ou plus, séparées par des points-virgules
<code>\$ (...) +</code>	Match 1 ou plusieurs fois sans séparateur
<code>\$ (...) , +</code>	Match 1 ou plusieurs fois, séparés par des virgules
<code>\$ (...) ; +</code>	Match 1 ou plusieurs fois, séparés par des points-virgules
<code>\$ (...) ?</code>	Match 0 ou 1 fois sans séparateur
<code>\$ (...) , ?</code>	Match 0 ou 1 fois, séparés par des virgules
<code>\$ (...) ; ?</code>	Match 0 ou 1 fois, séparés par des points-virgules

Le fragment de code `$x` n'est pas simplement une expression unique mais une liste d'expressions. Le modèle de cette règle utilise également la syntaxe de répétition :

```
<[_]>:: 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 encadré, puis utilise la `[T]::into_vec` méthode pour convertir le tableau encadré en vecteur.

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

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

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

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

Ici, la partie du modèle qui lit `$(v.push($x);) *` insère un appel à `v.push()` pour chaque expression dans `$x`. Un bras de macro peut se développer en 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.

Contrairement au reste de Rust, les modèles utilisant `$(...), *` ne prennent pas automatiquement en charge une virgule finale 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 `vec!` macro :

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

Nous utilisons `$(...), + ,` pour faire correspondre une liste avec une virgule supplémentaire. Ensuite, dans le modèle, nous appelons `vec!` de manière récursive, en omettant la virgule supplémentaire. Cette fois, la deuxième règle correspondra.

Macros intégrées

La rouillecompiler fournit plusieurs macros utiles lorsque vous définissez vos propres macros. Aucun de ceux-ci ne pourrait être mis en œuvre en utilisant `macro_rules!` seul. Ils sont codés en dur dans `rustc` :

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

`file!()` se développe à un littéral de chaîne : le nom de fichier actuel. `line!()` et `column!()` développer en `u32` littéraux donnant la ligne et la colonne actuelles (en partant de 1).

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

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

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

Les appels de macro dans l'argument ne sont *pas* développés :

`stringify!(line!())` se développe jusqu'à la chaîne `"line!()"`.

Rust construit la chaîne à partir des jetons, il n'y a donc pas de saut de ligne ni de commentaire 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 construction :

`cfg!(...)`

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

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

`env!("VAR_NAME")`

Se développe à une chaîne : la valeur de la variable d'environnement spécifiée au moment de la compilation. Si la variable n'existe pas, c'est une erreur de compilation.

Cela ne servirait à rien si ce n'est que Cargo définit plusieurs variables d'environnement intéressantes lorsqu'il compile une caisse. Par exemple, pour obtenir la chaîne de version actuelle de votre crate, 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")
```

Cette est identique à `env!` sauf qu'il renvoie un `Option<'static str>` c'est-à-dire `None` si la variable spécifiée n'est pas définie.

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

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

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

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

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

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

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

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

Cette est la même sauf que le fichier est traité comme des données binaires, et non comme du texte UTF-8. Le résultat est un `&'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 relatif, il est résolu par rapport au répertoire qui contient le fichier courant.

Rust fournit également plusieurs macros pratiques que nous n'avons pas abordées précédemment :

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

Ces deux sont équivalents à `panic!()`, mais transmettent une intention différente. `unimplemented!()` va dans `if` les clauses, `match` les armes et autres cas qui ne sont pas encore traités. Ça 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.

`matches!(value, pattern)`

Compare une valeur à un modèle, et retourne `true` si elle correspond, ou `false` autrement. C'est équivalent à écrire :

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

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

Macros de débogage

Débogage une macro capricieuse peut être difficile. Le plus gros problème est le manque de visibilité sur le processus d'expansion macro. Rust développera souvent toutes les macros, trouvera une sorte d'erreur, puis imprimera un message d'erreur qui n'affichera pas le code entièrement développé contenant l'erreur !

Voici trois outils pour 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 voudriez enregistrer, ce n'est pas un gros problème en pratique.)

Tout d'abord et le plus simple, vous pouvez demander `rustc` de montrer à quoi ressemble votre code après avoir développé toutes les macros. Utilisez `cargo build --verbose` pour voir comment Cargo appelle `rustc`. Copiez la `rustc` ligne de commande et ajoutez `-Z unstable-options --pretty expanded` la en tant qu'options. Le code entièrement développé est vidé sur votre terminal. Malheureusement, cela ne fonctionne que si votre code est exempt d'erreurs de syntaxe.

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

Troisièmement, vous pouvez demander au compilateur Rust de consigner tous les appels de macro au terminal. Insérer `trace_macros!(true);` 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 :

```
#![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, à la fois avant et après l'expansion. La ligne `trace_macros!(false);` désactive à nouveau le traçage, de sorte que l'appel à `println!()` n'est pas tracé.

Construire le json! Macro

Nous avons maintenant discutées principales fonctionnalités de `macro_rules!`. Dans cette section, nous allons développer progressivement une macro pour créer des données JSON. Nous allons utiliser cet exemple pour montrer à quoi ressemble le développement d'une macro, présenter les quelques éléments restants de `macro_rules!` et offrir

quelques conseils sur la façon de s'assurer que vos macros se comportent comme vous le souhaitez.

De retour au [chapitre 10](#), nous avons présenté cette énumération 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 `Json` valeurs est malheureusement assez 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 proche de 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 `json!` macro qui prend une valeur JSON comme argument et se développe en une expression Rust comme celle de l'exemple précédent.

Types de fragments

La premièreLe 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 existe différentes sortes de choses dans les données JSON : objets, tableaux, nombres, etc. En fait, nous pourrions supposer que nous aurons une règle pour chaque type JSON :

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

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

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 pourrions essayer de faire correspondre les éléments en tant que `expr s` :

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

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

Type de fragment	Correspondances (avec exemples)	Peut être suivi de...
<code>expr</code>	Une expression: <code>2 + 2, "udon", x.len()</code>	<code>=> , ;</code>
<code>stmt</code>	Une expression ou une déclaration, n'incluant aucun point-virgule final (difficile à utiliser ; essayez <code>expr</code> ou à la <code>block</code> place)	<code>=> , ;</code>
<code>ty</code>	Un type: <code>String, Vec<u8>, (&str, bool), dyn Read + Send</code>	<code>=> , ; = { [: > as where</code>
<code>path</code>	Un chemin (discuté): <code>ferns, ::std::sync::mpsc</code>	<code>=> , ; = { [: > as where</code>
<code>pat</code>	Un modèle (discuté): <code>_, Some(ref x)</code>	<code>=> , = if in</code>
<code>item</code>	Un élément (discuté): <code>struct Point { x: f64, y: f64 }, mod ferns;</code>	N'importe quoi
<code>block</code>	Un bloc (discuté): <code>{ s += "ok\n"; true }</code>	N'importe quoi
<code>meta</code>	Le corps d'un attribut (discuté): <code>inline, derive(Copy, Clone), doc="3D models."</code>	N'importe quoi
<code>literal</code>	Une valeur littérale : <code>1024, "Hello, world!", 1_000_000f64</code>	N'importe quoi
<code>lifetime</code>	Une durée de vie: <code>'a, 'item, 'static</code>	N'importe quoi
<code>vis</code>	Un spécificateur de visibilité : <code>pub, pub(crate), pub(in module::submodule)</code>	N'importe quoi

Type de fragment	Correspondances (avec exemples)	Peut être suivi de...
<code>ident</code>	Un identifiant : <code>std</code> , <code>Json</code> , <code>longish_variable_name</code>	N'importe quoi
<code>tt</code>	Un arbre à jetons (voir texte) : <code>;</code> , <code>>=</code> , <code>{}</code> , <code>[0 1 (+ 0 1)]</code>	N'importe quoi

La plupart des options de ce tableau appliquent strictement la syntaxe Rust. Le `expr` type correspond uniquement aux expressions Rust (et non aux valeurs JSON), `ty` correspond uniquement 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 `expr` les reconnaîtraient. Nous ne pourrions pas faire en sorte qu'aucune de ces données corresponde à des données JSON arbitraires.

Les deux derniers, `ident` et `tt`, prennent en charge les arguments de macro correspondants qui ne ressemblent pas au code Rust. `ident` correspond à n'importe quel identifiant. `tt` correspond à un seul *arbre à jetons*: soit une paire de parenthèses correctement appariées, `(...)`, `[...]`, 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 une parenthèse, comme `1926` ou `"Knots"`.

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

```
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 `json!` macro peut correspondre à toutes les données JSON. Il ne nous reste plus qu'à produire le code Rust correct.

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

Récursivité dans les macros

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

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

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

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

Nous devons convertir chaque élément du tableau du formulaire JSON en Rust. Heureusement, il existe une macro qui fait cela : 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 `json!`, 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 :

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

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

Utiliser des caractéristiques avec des macros

Complexe d'écriture les macros posent 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 prendre en charge `json!(true)`, `json!(1.0)` et `json!("yes")`, en convertissant la valeur, quelle qu'elle soit, en le type de `Json` valeur approprié. Mais les macros ne sont pas bonnes pour distinguer les types. On peut imaginer écrire :

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

Cette approche tombe en panne 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 `From` trait, couvert. Nous devons simplement implémenter ce trait pour quelques types :

```
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 peut donc être judicieux 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);

```

Nous pouvons maintenant utiliser `Json::from(value)` pour convertir `value` n'importe quel type pris en charge en `Json`. Dans notre macro, cela ressemblera à ceci :

```

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

```

L'ajout de cette règle à notre `json!` macro lui permet de passer tous les tests que nous avons écrits jusqu'à présent. En rassemblant toutes les

pièces, cela ressemble actuellement à ceci :

```
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 `(width * 9.0 / 4.0)` qu'il est entre parenthèses, il s'agit d'un arbre à jetons unique, de sorte que la macro le correspond avec succès `$value:tt` lors de l'analyse de l'objet.

Cadrage et hygiène

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

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

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

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

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

L'expansion de la macro collerait ensemble deux morceaux de code, les deux utilisant le nom `fields` pour des choses différentes !

```
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 être 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 `json!` définie par la macro en quelque chose que ses appelants ne sont pas susceptibles de transmettre : au lieu de `fields`, nous pourrions l'appeler `__json$fields`.

La surprisevoici que *la macro fonctionne telle quelle*. Rust renomme la variable pour vous ! Cette fonctionnalité, implémentée pour la première fois dans les macros Scheme, est appelée *hygiène*, et on dit donc que Rust a *des macros hygiéniques*.

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

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

```
let fields = "Champs, WC" ;
laisser rôle = {
    let champs mut = Box::new(HashMap::new());
    fields.insert( "nom" .to_string(), Json::from( "Larson E. Whipsnade" ) );
    fields.insert( "actor" .to_string(), Json::from( fields ) );
    Json::Objet(champs)
} ;
```

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

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

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 véritable mécanisme est même un peu plus intelligent que cela, reconnaissant deux identifiants comme identiques, indépendamment de la « peinture », s'ils se réfèrent à une variable commune qui est à la fois pour la macro et son appelant. Mais des cas comme celui-ci sont rares dans 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 étaient peints d'une ou plusieurs couleurs au fur et à mesure que les macros étaient développées : `Box`, `HashMap` et `Json`, par exemple. Malgré la peinture, Rust n'a eu aucun mal à reconnaître ces noms de types. 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, les statiques et les noms de macros, Rust est "daltonien".

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

Tout d'abord, nous allons considérer un cas où l'hygiène stricte de Rust gêne, et nous devons le contourner. Supposons que nous ayons plusieurs fonctions contenant cette ligne de code :

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

Copier et coller cette ligne est une douleur. Pouvons-nous 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`
}
```

Comme écrit, cela ne fonctionne pas. Il faudrait que le nom `server_socket` dans la macro fasse référence au local `server_socket` déclaré dans la fonction, et vice versa pour la variable `req`. Mais l'hygiène empêche les noms dans les macros de "entrer en collision" avec des noms dans d'autres portées, même dans des cas comme celui-ci, où c'est ce que vous voulez.

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

Puisque `req` et `server_socket` sont maintenant fournis par la fonction, ils sont la bonne "couleur" pour cette portée.

L'hygiène rend cette macro un peu plus longue à utiliser, mais c'est une fonctionnalité, pas un bogue : 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 `server_socket` dans une fonction, vous trouverez tous les endroits où il est utilisé, y compris les appels de macro.

Importation et exportation de macros

Depuis les macros sont développés au début de la compilation, avant que Rust ne connaisse la structure complète des modules de votre projet, le compilateur dispose de moyens spéciaux 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' `#[macro_use]` attribut. Par exemple, supposons que notre *lib.rs* ressemble à ceci :

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

Toutes les macros définies dans le `macros` module sont importées dans *lib.rs* et donc visibles dans le reste du crate, y compris dans `client` et `server`.

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

Par exemple, le `lazy_static` crate fournit une macro appelée `lazy_static`, qui est marquée par `#[macro_export]`. Pour utiliser cette macro dans votre propre caisse, vous écririez :

```
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 devrait donc pas dépendre de quoi que ce soit dans la portée - on ne sait pas ce qui sera dans la portée où elle est utilisée. Même les caractéristiques du prélude standard peuvent être masquées.

Au lieu de cela, la macro doit utiliser des chemins absolus vers tous les noms qu'elle utilise. `macro_rules!` fournit le spécialfragment `$crate` pour aider à cela. Ce n'est pas la même chose que `crate`, qui est un mot-clé qui peut être utilisé n'importe où dans les chemins, pas seulement dans les macros. `$crate` agit comme un chemin absolu vers le module racine du crate où la macro a été définie. Au lieu de dire `Json`, nous pouvons écrire `$crate::Json`, ce qui fonctionne même s'il `Json` n'a pas été importé. `HashMap` peut être remplacé par

```
::std::collections::HashMap OU $crate::macros::HashMap.
```

Dans ce dernier cas, nous devons réexporter `HashMap`, car `$crate` ne peuvent pas être utilisés pour accéder aux fonctionnalités privées d'une caisse. Cela s'étend vraiment à quelque chose comme `::jsonlib`, un chemin ordinaire. Les règles de visibilité ne sont pas affectées.

Après avoir déplacé la macro vers son propre module `macros` et l'avoir modifiée pour utiliser `$crate`, elle ressemble à ceci. C'est la version finale:

```
// 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 `.to_string()` méthode fait partie du `ToString` trait standard, nous l'utilisons `$crate` également pour nous y référer, en utilisant la syntaxe que nous avons introduite dans [« Appels de méthode entièrement qualifiés »](#) :

`$crate::macros::ToString::to_string($key)` . Dans notre cas, ce n'est pas strictement nécessaire pour faire fonctionner la macro, car `ToString` 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 moment où la macro est appelée, un appel de méthode entièrement qualifié est la meilleure façon de le faire.

Éviter les erreurs de syntaxe lors de la correspondance

La macro suivante semble raisonnable, mais cela 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 ainsi :

```

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

```

Aux yeux de l'homme, cela correspond évidemment au deuxième modèle. Mais Rust essaie d'abord la première règle, en essayant de faire correspondre toutes les entrées avec `$msg: expr` . C'est là que les choses commencent à mal tourner pour nous. `user: "jimb"` 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.

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, `$msg:expr`, 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.

Tout d'abord, évitez les règles confuses. Nous pourrions, par exemple, modifier la macro afin que chaque motif commence par un identifiant 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 la macro commencent par `msg`, nous obtenons la règle 1. Lorsqu'ils commencent par `user`, nous obtenons la règle 2. Dans tous les cas, nous savons que nous avons la bonne règle avant d'essayer de faire correspondre un fragment.

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

Au-delà des macro_règles !

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](#), de Daniel Keep et al., est un excellent manuel de `macro_rules!` programmation avancée. Le livre est clair et intelligent, et il décrit chaque aspect de l'expansion macro plus en détail que nous l'avons ici. Il présente également plusieurs techniques très astucieuses pour mettre `macro_rules!` des modèles en service comme une

sorte de langage de programmation ésotérique, pour analyser des entrées complexes. Nous sommes moins enthousiastes. Utiliser avec précaution.

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

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

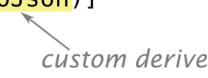


Image 21-4. Invoquer une `IntoJson` macro procédurale hypothétique via un `#[derive]` attribut

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

Ce qui rend une macro procédurale "procédurale", c'est qu'elle est implémentée comme une fonction Rust, et non comme un ensemble de règles déclaratives. Cette fonction interagit avec le compilateur à travers une fine couche d'abstraction et peut être arbitrairement complexe. Par exemple, la `diesel` 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.

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

Après avoir lu tout cela, vous avez peut-être décidé que vous détestez les macros. Quoi alors ? Une alternative consiste à générer du code Rust à l'aide d'un script de construction. La [documentation Cargo](#) montre comment procéder 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 construction et d'utiliser `include!` pour obtenir le code généré dans votre caisse.

