Rust propose un système de macros (lui-même extensible) qui permet d'enrichir le langage de plusieurs manières qui sont inaccessibles aux seules fonctions. Nous en connaissons plusieurs comme **println!(), format!(), vec![]**, etc. Il existe la macro **assert_eq!()** qui permet de réaliser des tests en ligne comme dans l'exemple ci-dessus.

Test du vecteur en ligne

```
fn main() {
  let entiers = vec![1, 2, 3];
  assert_eq!(entiers, [1, 2, 3]); // Dans cet exemple, la compilation s'effectue correctement
}
```

Ce genre de test aurait pu être réalisée par une fonction générique, mais la macro réalise plusieurs actions qu'une fonction ne peut faire. Tout d'abord, lorsque l'insertion échoue, la macro génère un message d'erreur qui comporte le nom du fichier et le numéro de ligne du code source.

Une fonction n'a pas accès à ce genre d'information, mais une macro le peut, parce qu'elle fonctionne de manière totalement différente. Une macro est une sorte de « code lyophilisé ». lors de la compilation, avant le contrôle des types et bien avant la génération du code machine, chacune des macros est déployée (expanded), c'est-à-dire remplacée par du code source Rust.

Le déploiement et l'invocation de la macro précédente produit l'injection dans le fichier invoquant de tout le bloc de lignes suivantes (code source automatiquement modifié, d'où la notion de macro) :

Code source équivalent de la macro précédente

```
match (&entiers, &[1, 2, 3]) {
  (left_val, right_val) => {
    if !(*left_val == *right_val) {
      panic!("assertion failed: '(left == right)', \
            (left: '{:?}', right: '{:?}')", left_val, right_val);
    }
  }
}
```

Au cours de l'exécution, l'échec d'une assertion correspond à l'affichage suivant, ce qui indique alors un bogue dans l'interprétation du vecteur (mauvaise évaluation du programmeur sur le contenu du vecteur à ce moment précis) :

Test du vecteur en ligne

```
fn main() {
  let entiers = vec![1, 2, 3];
  assert_eq!(entiers, [1, 2]);
}
```

Résultat

```
thread 'main' panicked at 'assertion failed: `(left == right)`
   left: `[1, 2, 3]`,
 right: `[1, 2]`', src/main.rs:3:3
...
```

*Vous remarquez que notre macro **assert_eq!()** utilise une autre macro standard **panic!()** qui elle-même utilise deux macros **file!()** et **line!()**. Le compilateur déploie toutes les invocations de macros qu'il trouve dans la caisse concernée avant de passer à l'étape suivante de compilation (ressemble à la phase du préprocesseur du langage C++).*

Dans les invocations de macros, le nom est toujours suivi du signe « ! » afin de bien distinguer les macros du reste du code. Vous ne risquer ainsi pas d'invoquer une macro lorsque vous voulez appeler une fonction.

Les macros Rust n'ajoutent jamais de crochets ou de parenthèses en nombre impair. Enfin, elles supportent le mécanisme de motif, ce qui permet d'obtenir des macros faciles à écrire, à utiliser et maintenir.

Principe des macros

Pour définir une macro en **Rust**, vous commencez par le mot-clé **macro_rules !**. Nous devons ne pas mettre le signe **« ! »** après le nom de la macro que vous voulez définir. Ce signe ne doit être mentionné que lors de l'invocation de la macro. Voici par exemple comment la macro **assert_eq!()** est définie.

Définition de la macro assert_eq!()

```
macro_rules! assert_eq {
  ($left: expr, $right: exp) => ({   // expr → motif, le bloc d'instruction → modèle
    match (&$left, &$right) {
      (left_val, right_val) => {
        if !(*left_val == *right_val) {
          panic!("assertion failed: '(left == right)', \
              (left: '{:?}', right: '{:?}')", left_val, right_val);
        }
      }
    }
  });
}
```

*Notez que certaines macros ne sont pas définies de cette façon : les macros **file!**, **line!** et **macro_rules!** elles mêmes sont directement incorporées au compilateur, car ce sont des macros procédurales. Pour l'instant, restons concentrés sur les définitions avec **macro_rules!**, manière la plus simple de définir une macro.*

*Une macro définie grâce à **macro_rules!** fonctionne uniquement par comparaison des paramètres à un motif. Le corps de la macro est une succession de règles.*

Notez que vous pouvez utiliser des crochets droits ou des accolades à la place des parenthèses autour du motif ou du modèle, Rust acceptant les trois types de signe. Lors de l'invocation, les trois syntaxes suivantes sont équivalentes :

Syntaxe d'écriture des macros

```
assert_eq!(entiers, [1, 2]);
assert_eq![entiers, [1, 2]];
assert_eq!{entiers, [1, 2]};
```

*La seule différence est que les points-virgules sont normalement facultatifs après les accolades. Nous utiliserons dans la suite de l'invocation de **asset_eq !**, des crochets pour **vec !** et des accolades pour **macro_rules !**, mais ce n'est qu'une convention.*

Principe de déploiement d'une macro

Nous avons dit que **Rust** déployait les macros en tout début de compilation, le compilateur commence par parcourir tout le code source pour repérer les macros et les traiter en premier. Vous ne pouvez pas invoquer une macro avant que le compilateur ait trouvé sa définition.

*Les autres éléments du programme n'ont pas du tout cette contrainte, et une fonction peut être définie plus loin dans le code source que le premier appel à cette fonction. Vous pouvez appeler en ligne **50** une fonction dons le corps de définition ne commence qu'en ligne **130** dans la même caisse.*

*Le déploiement de notre invocation de macro ressemble à l'évaluation d'une expression **match**. Rust confronte les paramètres du motif. Les patrons de macro forment une sorte de mini-langage dans **Rust**. Ce sont en quelques sorte des expressions régulières appliquées à du code source.*

*Mais alors que les motifs d'expressions régulières traitent des caractères, les patrons de macros traitent des éléments du langage, c'est-à-dire des nombres, des noms, des signes de ponctuation et autres éléments qui constituent le code source d'un programme **Rust**.*

*Dans notre exemple, le patron comporte le membre **$left : expr**, qui invite **Rust** à trouver une expression entre parenthèse (dans l'exemple, il s'agit de **entiers**) afin de lui attribuer ce nom dans **$left**. **Rust** apparie ensuite la virgule dans le patron à la virgule qui suit l'appel à **entier**. **Rust** associe ensuite la seconde expression, la valeur **[1, 2]** et lui attribue le nom **$right**.*

*Dans ce patron,les deux membres sont du type **expr**, car ce sont des expressions. Nous verrons d'autres types de fragments de code un peu plus loin dans notre étude. Une fois que la patron a pu être associé à tous les paramètres fournis, Rust peut déployer le modèle (**template**) correspondant. **Rust** remplace les deux variables **$left** et **$right** par les deux fragments de code identifiés lors de la première étape d'appariement.*

Conséquences inattendues

Il existe deux subtiles différences entre des fragments de code source injectés dans les modèles et le code source injectés dans les modèles et le code source normal, qui s'appuie sur des valeurs. Ces différences ne sont pas évidentes au départ. La macro que nous étudions, **assert_eq !**, contient notamment des tournures étranges, mais qui en dise long sur la programmation des macros. Intéressons-nous à deux de ces tournures.

*Tout d'abord, pourquoi la macro prend-elle la peine de créer les deux variables **left_val** et **right_val**. Ne pourrions-nous pas simplifier le modèle en prenant en compte uniquement les paramètres **left** et **right**. Voici comment changer notre code :*

Définition de macro simplifiée

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

Pour répondre à cette question, essayer mentalement de déployer l'invocation de macro suivante :

assert_eq!(lettres.pop(), Some('z')) ;

*À votre avis, quel sera le résultat ? Rust va injecter les expressions trouvées dans le modèle à plusieurs endroits. Mais il semble illogique de faire évaluer la même expression plusieurs fois pour construire le message d'erreur. Non seulement, cela ralentit, mais surtout, puisque **lettres.pop()** enlève une valeur du vecteur, ce sera une autre valeur qui sera produite au second tour !*

*Voila pourquoi la macro n'évalue qu'une seule fois les expressions **$left** et **$right** puis stocke les valeurs trouvées. Passons à la deuxième question : pourquoi la macro emprunte des **références** à **$letf** et **$right** ? Nous pourrions directement stockées les valeurs dans les variables, comme l'exemple ci-après :*

Définition de la macro assert_eq!()

```
macro_rules! assert_eq {
  ($left: expr, $right: exp) => ({   // expr → motif, le bloc d'instruction → modèle
    match ($left, $right) {
      (left_val, right_val) => {
        if !(*left_val == *right_val) {
          panic!("assertion failed: '(left == right)', \
              (left: '{:?}', right: '{:?}')", left_val, right_val);
        }
      }
    }
  });
}
```

Dans l'exemple d'application que nous faisons ici, les deux paramètres sont des tableaux (dynamique et statique). Dans ce cas de figure, l'assertion dépossède ces deux types de variables, ce qui est lourd de conséquence.

Puisque nous ne voulons surtout pas qu'une assertion prenne possession d'une valeur, nous faisons en sorte que la macro emprunte des références.

*Nous pourrions nous demander pourquoi la macro utilise **match** au lieu de **let** pour définir des variables ; cela nous avait surpris aussi. Après vérification, il n'existe aucune raison particulière. Cela fonctionne très bien aussi avec **let**.*

Répétition d'expressions

Voici les deux formats dans lesquels se présente la macro standard **vec !**, et voici comment cette macro pourrait être implémentée :

Création de vecteurs avec les deux approches

```
fn main() {
  // Répète une valeur N fois
  let octets = vec![0_u8 ; 10];

  // Liste des valeurs, séparées par des virgules
  let premiers = vec![2, 3, 5, 7, 11, 13, 17];
}
```

macro équivalente

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

*Cette définition comporte trois règles. Voyons d'abord comment faire coopérer plusieurs règles avant d'aller dans les détails de chacune d'elles. Lorsque Rust analyse une invocation de macro telle que **vec![1, 2, 3]**, il essaie d'abord d'apparier les paramètres **1, 2, 3** avec le patron de la première règle qui est ici **$elem : expr, $n : expr**.*

*Cela ne fonctionne pas car bien que 1 soit une expression, la patron attend un signe **« ; »** juste après, et il n'y en a pas. Rust passe donc à la deuxième règle et ainsi de suite. Si aucune règle ne convient, cela déclenche une erreur.*

*La première règle convient tout à fait à une invocation telle que **vec![0_u8 ; 10]**. La fonction standard **std::vec::from_elem()** fait exactement ce qui est demandé ici ; la règle pose donc aucun problème.*

*La seconde règle permet de gérer l'invocation avec **vec![2, 3, 5, 7, 11, 13, 17]**. Le patron **$($x : expr), *** comporte une caractéristique nouvelle qui est la répétition et correspond à la paire **« ,* »**. Cela signifie que le patron pourra apparier entre zéro et plusieurs expressions séparées par des virgules. La syntaxe générique **$(PATRON), *** est utilisée pour trouver n'importe quelle liste à séparateur, chaque élément de la liste devant correspondre à **PATRON**.*

*Le symbole ***** garde le même sens que dans les expressions régulières **regex** (zéro ou plusieurs), même si ces expressions ne possèdent pas l'opérateur de répétition **,***. Pour faire trouver au moins une occurrence, vous disposez de l'opérateur **+**. Notez qu'il n'existe pas pour les macros l'opérateur **?**. Le tableau suivant présente tous les patrons de répétition.*

Patron	Description
$(…) *	Trouve **0** ou **plus** occurrences, **sans** séparateur.
$(…), *	Trouve **0** ou **plus** occurrences, avec séparateur **virgule**.
$(…) ; *	Trouve **0** ou **plus** occurrences, avec séparateur **point-virgule**.
$(…) +	Trouve **1** ou **plus** occurrences, **sans** séparateur.Trouve **0** ou **plus** occurrences, avec séparateur **virgule**.

$(…), +	Trouve **1** ou **plus** occurrences, avec séparateur **virgule**.
$(…); +	Trouve **1** ou **plus** occurrences, avec séparateur **point-virgule**.

Le fragment $x dans la définition n'est pas une expression isolée, mais une liste d'expressions. En effet le modèle de cette règle utilise la syntaxe de répétition : ***<[_]>::into_vec(Box::new([$($x), *]))****.*

Ici aussi, nous profitons de méthodes standard pour réaliser le traitement. Le code crée un tableau enveloppé dans une boîte, puis appelle la méthode ***[T]::into_vec()*** *pour convertir le tableau emboîté en un vecteur.*

La mention initiale, ***<[_]>****, est une façon originale de désigner le type « tranche de quelque chose » en laissant* ***Rust*** *déduire le type d'élément. Vous pouvez mentionner directement dans vos expressions les types dont les noms sont des identifiants normaux. En revanche, les types à écriture spéciales comme* ***fn()****,* ***&str*** *ou* ***[_]*** *doivent être entourés par des chevrons.*

La répétition entre en jeu à la fin du modèle, avec ***$($x), *****. Il s'agit de l'application de la formule* ***$(…), **** *utilisée dans le patron. La règle fait itérer dans la liste des expressions qui ont été appariées pour* ***$x*** *pour les insérer toutes dans le modèle en les séparant par des virgules.*

Dans cet exemple précis, le résultat répété a le même aspect que l'entrée, mais ce n'est pas toujours le cas. Nous aurions pu formuler la règle ainsi.

Expression répétitive simplifiée

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

La quatrième ligne de ce modèle, ***$(vecteur.push($x) ;) ****, provoque l'insertion d'un appel à* ***vecteur.push()*** *pour chacune des expressions dans* ***$x****.*

Contrairement à la règle générale Rust, les patrons de répétition dans le style $(…), * ne reconnaissent pas automatiquement la virgule finale facultative. Vous pouvez cependant ajouter ce support en ajoutant une règle, et c'est la troisième règle de notre macro vec ! :

Utilité de la troisième règle

```
($($x: expr), +, ) => {   // Si virgule finale présente
  vec![$($x), *]          // élimination de la virgule finale pour activer automatiquement la deuxième règle
};
```

Nous utilisons ici le patron ***$(…), +,*** *pour trouver une liste avec une virgule finale. Dans le modèle, nous appelons plusieurs fois* ***vec!*** *en ignorant cette virgule. Et cette fois-ci, la deuxième règle s'applique.*

Macros incorporées

Une dizaine de macros sont directement incorporées dans le compilateur **Rust** et servent à définir vos propres macros. Aucunes ne peut être implémentée uniquement en utilisant la macro **macro_rules!**. C'est pour cette raison qu'elles sont incorporées directement dans **rustc,** en voici les principales :

- ***file!()*** *se transforme en un littéral chaîne qui est le nom du fichier courant. Les deux macro* ***line!()*** *et* ***column!()*** *deviennent des littéraux de type* ***u32*** *pour indiquer le numéro de ligne à partir de* ***1*** *et le numéro de colonne à partir de* ***0****.*

  Dans le cas d'une imbrication d'invocation de macros dans différents fichiers, si la dernière macro invoque ***file!()****,* ***line!()*** *ou* ***column!()****, le résultat affichera la position de la première invocation de macros (celle qui vous intéresse).*

- ***stringify!(...elements...)*** *est déployée sous forme d'un littéral chaîne contenant les éléments mentionnés. Cette macro est utilisée par* ***assert!()*** *pour gérer le message d'erreur qui cite le code de l'assertion.*

  Notez que les invocations de macros qui sont enchâssées en paramètres ne sont heureusement pas déployées : en écrivant : ***stringify!(line!())****, vous obtenez la chaîne* ***line!()****.*

  Du fait que ***Rust*** *construit la chaîne à partir des éléments fournis, cette chaîne résultante ne contient aucun saut de ligne, ni commentaire.*

- ***concat!(str0, str1, ...)*** *est transformé en un littéral chaîne unique qui résulte de la concaténation des chaînes fournies.*

- ***cfg!(...)*** *devient une constante booléenne possédant la valeur* ***true*** *si la configuration de construction actuelle correspond aux conditions fournies entre parenthèses. Par exemple,* ***cfg!(debug_assertions)*** *vaut* ***true*** *si vous compilez en ayant les assertions de débogage.*

- ***include!(« nomfichier.rs »)*** *est remplacé par le contenu du fichier mentionné. Il doit s'agir de code source Rust valide, soit une expression, soit une séquence d'éléments.*

Ces macros sont bien sûr traitées au moment de la compilation. Autrement dit, si le fichier n'est pas trouvé, la compilation échoue ; l'exécution ne peut jamais échouer pour cette raison.

Quelques exemples

Je vous propose de réaliser un ensemble de macros qui permet de bien maîtriser tous les différents concepts que nous venons d'étudier. Pour cela, nous allons concevoir trois macros, la première très simple qui affiche juste un message de bienvenue, la deuxième qui permet d'afficher une ou plusieurs variables avec le nom de chaque variable associée.

La troisième macro permet de réaliser une fonction factorielle en utilisant les méthodes d'itérations sophistiquées avec du coup une écriture très raccourcie (d'où l'intérêt d'utiliser une macro plutôt qu'une fonction), mais en rajoutant la possibilité de proposer plusieurs calculs consécutifs, le résultat des différents valeur étant alors placé dans un vecteur.

Plusieurs macros utilitaires

```
macro_rules! bienvenue {
  () => { println!("Bienvenue!") };
}

macro_rules! affiche {
  ($valeur: expr) => {
    println!("{} = {:?}", stringify!($valeur), &$valeur);
  };
  ($x: expr, $($y: expr), +) => {
    print!("{} => {:?}, ", stringify!($x), &$x);
    affiche!($($y), +)
  }
}

macro_rules! fact {
  ($n: expr) => {
    (1..$n+1).product::<u32>()
  };
  ($($n: expr), +) => {
    {
      let mut vecteur = Vec::new();
      $(vecteur.push((1..$n+1).product::<u32>()); ) +
      vecteur
    }
  };
}

fn main() {
  bienvenue!();
  let x = fact!(5);
  affiche!(x);
  affiche!(fact!(6));
  let liste = fact!(5, 6, 7);
  affiche!(liste);
  affiche!(liste, x, fact!(3));
  affiche!(fact!(7, 4, 8));
}
```

Résultat

```
Bienvenue!
x = 120
fact!(6) = 720
liste = [120, 720, 5040]
liste => [120, 720, 5040], x => 120, fact!(3) = 6
fact!(7, 4, 8) = [5040, 24, 40320]
```

Types de fragments de macro

Pour concevoir une macro un tant soit peu complexe, il faut commencer par décider comment les paramètres d'entrée doivent être analysés. Par exemple, nous connaissons déjà le patron du style **$($element : expr), *** qui signifie « liste d'expressions Rust, séparées par des virgules ».

*Le problème est qu'un certain nombre de valeurs d'entrées, et notamment les objets, ne sont pas nécessairement des expressions **Rust** valides et ne seront donc pas trouvées.*

*Les concepteurs de **Rust**, voyant que toute valeur d'entrée n'était pas nécessairement une expression, ont défini d'autres types de fragments, qui sont représentés dans le tableau ci-dessous :*

Type de fragment	Description
expr	Une expression : **2+2, « chaîne », x.len()**, etc.
stmt	Une expression ou une déclaration, sans point-virgule final (emploi ardu ; préférez **expr** ou **block**)
ty	Un type : **String, Vec<u8>, (&str, bool)**, etc.
path	Un chemin : **::std::sync::mpsc**, etc.
pat	Un patron : **_ , Some(ref x)**, etc.
item	Un élément : **struct Point {x:f64, y : f64}**, etc.
block	Un bloc d'expressions : **{ s += « Ok\n » ; true }**

meta	Le corps d'un attribut : **inline, derive(Copy, Clone)**, etc.
ident	Un identifiant : **std, long_nom_variable**
tt	Un arbre d'éléments : **;, ≥, {}, [0 1 (+ 0 1)]**

*Les deux derniers types de fragments, **ident** et **tt**, permettent de trouver des paramètres qui ne ressemble pas à du code Rust. Le premier, **ident**, trouve n'importe quel identifiant et l'autre, **tt**, trouve un arbre d'éléments uniques. Il peut s'agir soit d'une paire de délimiteurs **(…)**, **[…]** ou **{…}**, avec tout le contenu, y compris les arbres imbriqués, soit un seul élément qui n'est pas un crochet délimiteur, comme **1926** ou **« Art of noise »**.*

L'exemple suivant nous montre comment créer de nouvelles fonctions automatiquement grâce aux macros et grâce à l'utilisation de ces deux derniers types de fragment.

Création automatique de fonctions

```
macro_rules! affiche {
  ($valeur: expr) => {
    println!("{} = {:?}", stringify!($valeur), &$valeur);
  }
}

macro_rules! fonctions {
  ($fonction: ident, $operation: tt) => {
    fn $fonction(n: u32) -> u32 {
      let mut resultat = 1;
      for x in 2..n+1 { resultat $operation x; }
      resultat
    }
  }
}

fonctions!(fact, *=);
fonctions!(triangle, +=);

fn main() {
  affiche!(fact(5));
  affiche!(triangle(5));
}
```

Résultat

```
fact(5) = 120
triangle(5) = 15
```