Макросы Rust

Команда Rust. Перевод сообщества rustycrate.ru

27.01.2016

Оглавление

1

Введение

Замечание: работа еще в процессе.

Целью данной книги является попытка объединения коллективных знаний Rust сообщества. Таким образом, как дополнения (в форме pull request), так и запросы (в форме issue) приветствуются.

Если хочешь помочь, смотри the GitHub repository[1]

Спасибо

Спасибо за предложения и исправления: IcyFoxy, Rym, TheMicroWorm, Yurume, akavel, cmr, eddyb, ogham, и snake_case.

Лицензия

[1] https://github.com/DanielKeep/tlborm/
[2] http://creativecommons.org/licenses/by-sa/4.0/
[3] http://opensource.org/licenses/MIT

2

Макросы, методическое введение

Эта глава расскажет о системе Rust Макрос-По-Примеру (Macro-By-Example): `macro_rules!`. Вместо того, чтобы попытаться раскрыть, как она работает на примерах, она попытается дать вам полное доскональное объяснение того, *как* вся система работает. Поэтому она предназначена для людей, которые хотят получить полное понимание работы системы, а не поверхностное руководство по ней.

Есть также глава по макросам в Rust Book[1], которая более доступна, то есть объяснения даются на более высоком уровне, также есть практическое введение[2] - глава в этой книге, в которой представлено руководство по реализации макроса.

Расширения синтаксиса

Перед тем, как разговаривать о *макросах*, необходимо обсудить общий механизм, на котором они построены: *расширение синтаксиса*. Чтобы сделать *это*, мы должны обсудить, как компилятор обрабатывает исходный код Rust и общие принципы, по которым строятся макросы, определяемые пользователем.

Анализ исходного кода

Первым этапом компиляции для программ на Rust является токенизация. На этом этапе исходный текст преобразуется в набор токенов (*т.е.* неразделимых лексических блоков; эквиваленты “словам” на программном языке). Rust поддерживает различные типы токенов. Среди них, такие как:

- Идентификаторы: `foo`, `Bambous`, `self`, `we_can_dance`, `LaCaravane`, …

[1] http://doc.rust-%20lang.org/book/macros.html
[2] https://github.com/ruRust/tlborm/blob/master/text/pim-README.md

- Целые числа: `42`, `72u32`, `0_______0`, ...
- Ключевые слова: `_`, `fn`, `self`, `match`, `yield`, `macro`, ...
- Время жизни: `'a`, `'b`, `'a_rare_long_lifetime_name`, ...
- Строки: `""`, `"Leicester"`, `r##"venezuelan beaver"##`, ...
- Символы: `[`, `:`, `::`, `->`, `@`, `<-`, ...

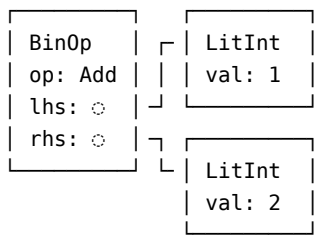
Необходимо выделить из приведенного списка следующее: первое, `self` является как идентификатором, *так и* ключевым словом. Почти во всех случаях `self` - это ключевое слово, но оно может также *трактоваться* как идентификатор, который придет позже (вместе с проклятьями). Во-вторых, в список ключевых слов входят разные подозрительные фразы, такие как `yield` и `macro`, которые *на самом деле* не входят в язык, но разбираются компилятором - они зарезервированы на будущее. Третье, в список символов *также* входят элементы, которые не используются языком. Если взять `<-`, то это рудимент: он был удален из грамматики, но не из словаря. Наконец, помните, что `::` - это выдающийся токен; это не просто два токена `:`. То же самое справедливо для всех составных символьных токенов в Rust, начиная с Rust 1.2. [3]

Сравнивая с другими языками, на этом этапе у некоторых из них есть макро уровень, а у Rust *нет*. Например, макросы C/C++ *эффективно* выполняются на этом этапе. [4] Вот почему работает следующий код: [5]

```
#define SUB void
#define BEGIN {
#define END }

SUB main() BEGIN
    printf("Oh, the horror!\n");
END
```

Следующий этап - разбор, где поток токенов превращается в Абстрактное Синтаксическое Дерево (AST). Здесь строится синтаксическая структура программы в памяти. Например, сочетание токенов `1 + 2` преобразуется соответственно в:



AST содержит структуру *всей* программы, хотя основывается она исключительно *на лексической* информации. Например, компилятор может знать, что часть выражения относится к переменной “`a`” на этом этапе, хотя он понятия не имеет, что такое “`a`”, или *откуда* ее взять.

[3] у `@` есть назначение, о котором большинство людей забывают: он используется в паттернах для того, чтобы связать нетерминальную часть паттерна с именем. Даже член команды ядра Rust - тот, кто разрабатывал *конкретно* эту главу, читая ее перед одобрением, не вспомнил, что у `@` есть это назначение. Позор, какой позор.

[4] На самом деле, препроцессор C использует другие лексические структуры по отношению к самому C, хотя различия *очень* незначительны.

[5] *Будет* ли это работать - это совершенно *другой* вопрос.

После того, как было сконструировано AST, обрабатываются макросы. Однако, прежде чем мы это обсудим, мы должны поговорить о деревьях токенов.

Деревья токенов

Деревья токенов - это нечто среднее между токенами и AST. Для начала, *почти* все токены являются деревьями токенов; если более конкретно, они являются *листьями*. Есть еще одна вещь, которая тоже может быть листом дерева, но о ней дальше.

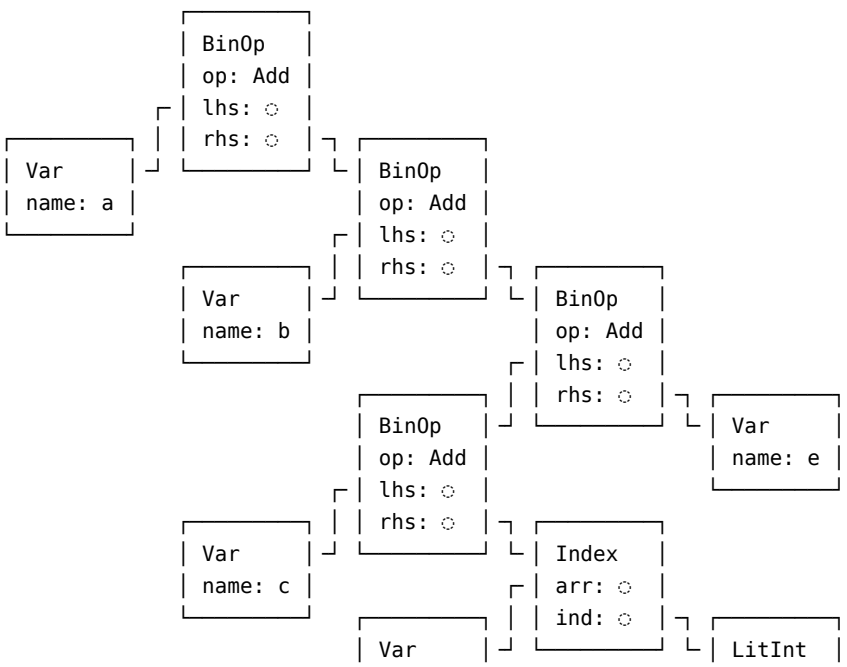
Единственные базовые токены, которые *не* являются листьями, это токены “группировки”: `(...)`, `[...]` и `{...}`. Эти три - *внутренние узлы* деревьев токенов, или то, что и конструирует структуру дерева. Для конкретного примера, эта связка токенов:

```
a + b + (c + d[0]) + e
```

будет разобрана в следующие деревья токенов:

```
«a» «+» «b» «+» «(   )» «+» «e»
          ┌──────┴──────┐
          «c» «+» «d» «[   ]»
                      ┌─┴─┐
                      «0»
```

Помните, что это не имеет *никакого отношения* к выражениям, производимым AST; вместо одного корневого узла, здесь *девять* деревьев токенов на корневом уровне. Для справки, AST будет следующим:



Важно понимать различие между AST и деревьями токенов. При написании макросов вам придется иметь дело с *обеими вещами* по отдельности.

Другой важный аспект: *нельзя* указать *непарную* обычную, фигурную или квадратную скобку; также невозможно указать неправильно вложенные группы в дереве токенов.

Макросы в AST

Как раньше отмечалось, обработка макросов в Rust происходит *после* построения AST. Поэтому синтаксис, используемый для вызова макроса, *должен быть* собственной частью синтаксиса языка. На самом деле, есть несколько форм “расширений синтаксиса”, которые являются частью синтаксиса Rust. Конкретно, следующие формы (с примерами):

- `# [ $arg ]`; *например* `#[derive(Clone)]`, `#[no_mangle]`, ...
- `# ! [ $arg ]`; *например* `#![allow(dead_code)]`, `#![crate_name="blang"]`, ...
- `$name ! $arg`; *например* `println!("Hi!")`, `concat!("a", "b")`, ...
- `$name ! $arg0 $arg1`; *например* `macro_rules! dummy { () => {}; }`.

Первые две - “атрибуты” и относятся как к специальными конструкциями языка (такие как `#[repr(C)]`, которая используется при запросе C-совместимого ABI для пользовательского типа) так и синтаксическим расширениями (такие как `#[derive(Clone)]`). В данный момент нет возможности определить макрос, который бы использовал эти формы.

Именно третья форма представляет для нас интерес: это форма, доступная для использования макросов. Помните, что она не *ограничена* только для макросов: это общая синтаксическая форма расширения. Например, если `format!` - это макрос, `format_args!` (которая используется для *реализации* `format!`) - *нет*.

Четвертая форма - это по существу вариация, которая *недоступна* для макросов. На самом деле, *единственным* местом, где она используется является `macro_rules!`, к которому мы еще вернемся.

Игнорируя все кроме третьей формы (`$name ! $arg`), встает вопрос: откуда парсер Rust знает, как выглядит `$arg` для всех возможных расширения синтаксиса? Ответ заключается в том, что ему и *не нужно* это. Вместо этого, аргументом вызова расширения синтаксиса является *одиночное* дерево токенов. Говоря более конкретно, это одиночное, *безлистное* дерево токенов; `(...)`, `[...]`, или `{...}`. Обладая этими знаниями, должно быть ясно, как парсер может разобрать все эти формы вызова:

```
bitflags! {
    flags Color: u8 {
        const RED    = 0b0001,
        const GREEN  = 0b0010,
        const BLUE   = 0b0100,
        const BRIGHT = 0b1000,
    }
```

```
}

lazy_static! {
    static ref FIB_100: u32 = {
        fn fib(a: u32) -> u32 {
            match a {
                0 => 0,
                1 => 1,
                a => fib(a-1) + fib(a-2)
            }
        }

        fib(100)
    };
}

fn main() {
    let colors = vec![RED, GREEN, BLUE];
    println!("Hello, World!");
}
```

Хотя эти вызовы *выглядят* так, как-будто содержат разный код Rust, парсер просто видит коллекцию бессмысленных деревьев токенов. Для ясности, мы можем заменить все эти синтаксические “черные ящики” на ⬚ и получим:

```
bitflags! ⬚

lazy_static! ⬚

fn main() {
    let colors = vec! ⬚;
    println! ⬚;
}
```

Для повторения: парсер *ничего* не делает с ⬚; он запоминает токены, которые они содержат, но не пытается их *понять*.

Важные выводы из этого следующие:

- Есть несколько расширений синтаксиса в Rust. Мы будем говорить *только* о макросах, определяемых конструкцией `macro_rules!`.
- Только из-за того, что вы видите что-то в форме `$name! $arg`, не означает, что это на самом деле макрос; это может быть другой тип расширения синтаксиса.
- Входом в каждый макрос является одиночное безлистное дерево токенов.
- Макросы (на самом деле расширения синтаксиса) разбираются как часть абстрактного синтаксического дерева (AST).

> **В сторону**: Из-за первого вывода некоторое описываемое ниже (включая следующий параграф) будет применяться к расширениям синтаксиса *в общем*. [6]

Последний вывод особенно важен, у него *значительные* последствия. Из-за того, что макросы разбираются в AST, они могут появляться **только** на том месте, в котором они явно поддерживаются. Конкретно макросы могут появляться в следующих местах:

- Паттерны
- Утверждения
- Выражения
- Элементы
- Элементы `impl`

Некоторые вещи, *не* указанные в списке:

- Идентификаторы
- Варианты при поиске совпадения с образцом
- Поля структур
- Типы [7]

Нет абсолютно, определенно *никакой* возможности использовать макросы в том месте, которе *не* указано в первом списке.

Развертывание

Развертывание - это относительно несложна штука. В какой-то момент *после* создания AST, но перед тем, как начать создавать семантическое представление программы, компилятор развернет все макросы.

Этот процесс включает в себя проход AST, определение мест вызовов макросов и замены их на развертывание. В случае расширений синтаксиса, не связанных с макросами, то,*как* компилятор их разворачивает, зависит от самих синтаксических расширений. Таким образом, расширения синтаксиса проходят через *точно такой же* процесс развертывания, как и макросы.

Когда компилятор запускает расширение синтаксиса, он ожидает, что результат можно будет разобрать в синтаксический элемент из известного ему списка, основываясь на контексте. Например, если вы вызываете макрос в границах видимости модуля, компилятор разберет результат в узел AST, представляющий элемент. Если вызвать макрос на позиции выражения, компилятор разберет результат в узел AST, представляющий выражение.

На самом деле, он может разобрать результат расширения синтаксиса в одно из следующих:

- выражение,
- паттерн,
- ноль или больше элементов,

[6] Гораздо удобнее и быстрее писать “макрос”, чем “расширение синтаксиса”.

[7] Макросы типа доступны в нестабильном Rust с `#![feature(type_macros)]`; см. Issue #27336[8].

- ноль или больше элементов `impl` , или
- ноль или больше утверждений.

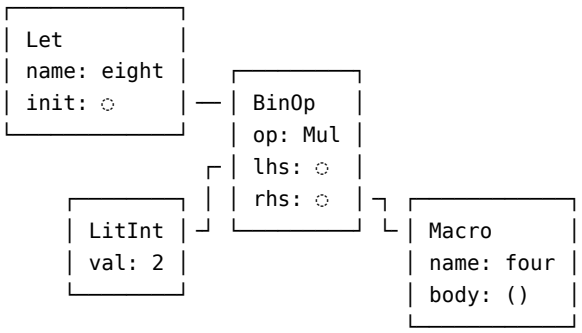
Другими словами, то, *где* вы можете вызвать макрос, определятся результатом, в который он разворачивается.

Компилятор берет узел AST и полностью заменяет узел вызова макроса, на то, во что он разворачивается. *Это структурная операция*, а не текстовая!

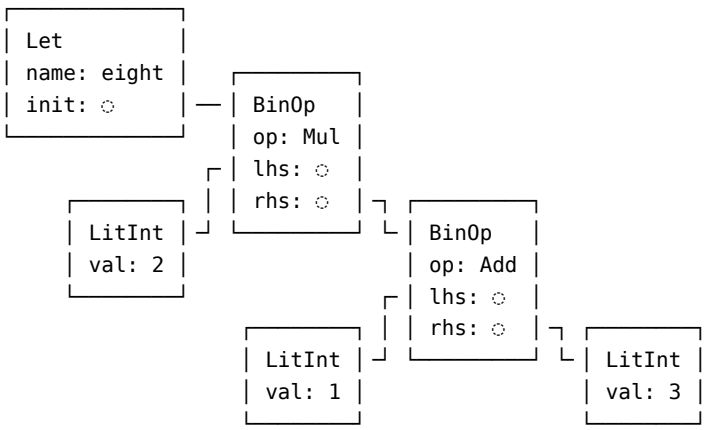
Например у нас есть следующее:

```
let eight = 2 * four!();
```

Мы можем изобразить часть AST таким образом:



Из контекста, `four!()` *должен* развернуться в выражение (инициализаторы могут быть *только* выражением). Поэтому, чем бы ни было реальное развертывание, оно будет интерпретироваться как законченное выражение. В данном случае мы подразумеваем, что `four!` определено так, что разворачивается в выражение `1 + 3`. В результате, после развертывания вызов можно изобразить так:



Можно также написать это так:

```
let eight = 2 * (1 + 3);
```

Заметьте, что мы добавили скобки, *несмотря на то, что* их не было в развертывании. Помните, что компилятор всегда трактует развертывание макроса, как законченный узел AST, **не** как простую последовательность токенов. По- другому, даже если вы не обернули сложное выражение в скобки, компилятор все равно “правильно” интерпретирует результат или поменяет порядок оценки.

Важно понимать, что развертывание макроса считается узлами AST. У такого дизайна следующие следствия:

- В дополнении к тому, что есть ограниченный набор *мест вызовов*, макросы могут разворачиваться *только в тот тип узла AST, который* ожидает парсер в этом месте.
- В завершении вышесказанного - макросы *никогда* не могут развернуться в незаконченное выражение или синтаксически неправильную конструкцию.

Нужно знать еще одну вещь про развертывание: что происходит, если расширение синтаксиса разворачивается во что-то, содержащее *другой* вызов расширения синтаксиса. Например, определим другое описание `four!`; что произойдет, если он развернется в `1 + three!()`?

```
let x = four!();
```

Разворачивается в:

```
let x = 1 + three!();
```

Компилятор ищет в получившемся результате развертывания дополнительные вызовы макросов и разворачивает их. Поэтому после второго развертывания получится следующее:

```
let x = 1 + 3;
```

Вынести из этого можно следующее - развертывание осуществляется “проходами”; столько, сколько нужно, чтобы полностью развернуть все вызовы.

Хотя, *немного не так*. На самом деле компилятор лимитирует количество таких рекурсивных проходов, которые он выполнит перед тем, как сдастся. Это называется лимит рекурсии макросов и, по умолчанию, равен 32. Если 32 развертывание содержит вызов макроса, компилятор прервет свою работу с ошибкой, сообщающей, что лимит рекурсии исчерпан.

Этот лимит можно увеличить через атрибут `#![recursion_limit="…"]`, однако это *необходимо* делать по всему контейнеру. Обычно, рекомендуется пробовать и пытаться удерживать макросы внутри этого лимита, где это только возможно.

macro_rules!

Держа все это в голове, мы можем представить сам `macro_rules!`. Как раньше было замечено, `macro_rules!` это *само по себе* расширение синтаксиса, имея ввиду, что это *технически* не часть синтаксиса Rust. Он использует следующую форму:

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

Должно быть *по крайней мере* одно правило. Вы можете опустить точку с запятой после последнего правила.

Каждое “`rule`” выглядит так:

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

Вообще, круглые и фигурные скобки могут обозначать любой тип групп, но принято, что круглые скобки ставят вокруг паттерна (pattern), а фигурные - вокруг развертывания (expansion).

Если вам интересно, сам `macro_rules!` разворачивается в… *ничто*. По крайней мере, хоть что-то не появляется в AST; скорее, он манипулирует внутренними структурами компилятора для регистрации макроса. Таким образом, *формально* вы можете использовать `macro_rules!` в любой позиции, в которой пустое развертывание является валидным.

Совпадения с образцом

Когда макрос вызывается, интерпретатор `macro_rules!` проходит все правила, одно за другим в лексическом порядке. Для каждого правила он пытается определить, совпадает ли содержание на входе с `паттерном` правила. Паттерн должен совпасть *целиком*, чтобы вход считался совпавшим.

Если вход совпадает с паттерном, вызов заменяется на `развертывание`; иначе, пробуется следующее правило. Если ни одно из правил не совпало, развертывание макроса вызывает ошибку.

Самый простой пример пустого паттерна:

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

Этот паттерн считается совпавшим с образцом тогда и только тогда, когда вход тоже будет пустым (*например* `four!()`, `four![]` or `four!{}`).

Заметьте, что символы для группировки (скобки) при вызове макроса *не* сопоставляются с образцом. Это означает, что вы можете вызывать макрос как `four![]`, и он все равно совпадет. Только *содержимое* на входе рассматривается.

Паттерны также могут содержать деревья литеральных токенов, которые должны полностью совпасть с образцом. Это делается просто написанием деревьев токенов в обычной форме. Например, для того, чтобы совпадение с выражением `4 fn ['spang "whammo"] @_@` выполнилось, вам нужно написать:

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

Вы можете использовать любые деревья токенов, какие сможете написать.

Метапеременные

Патерны также могут содержать метапеременные (captures). Это позволяет проверять вход на совпадение с образцом, основываясь на некоторой общей категории грамматики, получая преобразованный в переменную результат, который в дальнейшем может быть подставлен на выход.

Метапеременные обозначаются долларом (`$`), за которым следуют идентификатор, двоеточие (`:`), и наконец тип метапеременной, который должен быть одним из следующих:

- `item`: объект, например, функция, структура, модуль, и т.д.
- `block`: блок (например, блок утверждений или выражений, окруженный фигурными скобками)
- `stmt`: утверждение
- `pat`: паттерн
- `expr`: выражение
- `ty`: тип
- `ident`: идентификатор
- `path`: путь (например, `foo`, `::std::mem::replace`, `transmute::<_, int>`, ...)
- `meta`: мета объект; вещи которые идут внутри атрибутов `#[...]` и `#![...]`
- `tt`: одиночное дерево токенов

Например, вот макрос, который захватывает метапеременную на входе как выражение:

```
macro_rules! one_expression {
    ($e:expr) => {...};
}
```

Эти метапеременные по максимуму используют парсер компилятора Rust, всегда обеспечивая “корректность”. Метапеременная типа `expr` будет *всегда* захватывать полное, валидное выражение для той версии Rust, под которой оно компилируется.

Вы можете смешивать деревья литеральных токенов и метапеременные в определенных пределах (объясняются ниже).

Метапеременная `$name:kind` может быть заменена в развертывании написанием `$name`. Как пример:

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

В большинстве своем, как и развертывания макроса, метапеременные заменяются на полные узлы AST. Это означает, что неважно, какие выражения из токенов захвачены в метапеременной `$e`, она будет интерпретироваться как одно, законченное выражение.

У вас также может быть несколько метапеременных в одном паттерне:

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

Повторения

Паттерны могут содержать повторения. Это позволяет определить совпадения для связки токенов. Общая форма для этого - `$ ( ... ) sep rep`.

- `$` - знак доллара.
- `( ... )` - повторяющийся паттерн в скобках .
- `sep` - *дополнительный* разделитель. Например: `,` и `;`.
- `rep` - *требуемый* контроль повторений. В данный момент это может быть *или* `*` (обозначающая ноль или более повторений) или `+` (обозначающий одно или более повторений). Вы не можете написать “ноль или одно” или любое другое более конкретное число повторений.

Повторения могут содержать любые валидные паттерны, включая деревья литеральных токенов, метапеременные и другие повторения.

Повторения используют такой же синтаксис и в развертывании.

Например, внизу макрос, который форматирует каждый элемент как строку. Он ищет совпадения с нулем или более выражений, разделенных запятой, и разворачивает их в выражение, которое конструирует вектор.

```
macro_rules! vec_strs {
    (
        // Начало повторения:
        $(
            // Каждое повторение должно содержать выражение...
            $element:expr
        )
        // ...разделенное запятыми...
        ,
        // ...ноль или более раз.
        *
    ) => {
        // Заключаем разложение в скобки (блок) таким образом, что
        // мы можем использовать несколько утверждений.
        {
            let mut v = Vec::new();
```

```
            // Начало повторения:
            $(
                // Каждое повторение будет содержать следующее утверждение, в котором
                // $element будет заменен на соответствующее выражение.
                v.push(format!("{}", $element));
            )*

            v
        }
    };
}
```

Мелочи

Эта секция описывает некоторые мелкие детали системы макросов. Как минимум, вам стоит хотя бы узнать о существовании этих проблем.

Вернемся к Метапеременным и Развертыванию

Когда парсер начинает захватывать токены в метапеременные, *он уже не может остановиться или откатиться*. Это означает, что паттерн во втором правиле макроса, приведенного ниже, *никогда не совпадет с образцом*, что бы ни подавали на вход:

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

Представим, что случится, если этот макрос вызвать как `dead_rule!(x+)`. Интерпретатор начнет с первого правила и попытается распарсить вход как выражение. Первый токен (`x`) подходит как выражение. Второй *тоже* подходит как выражние, представляя собой узел двоичного сложения.

В таком случае, выдав это выражение на вход, без правого оператора сложения, вы ожидаете, что парсер пропустит первое правило и перейдет ко второму. Вместо этого, парсер вызовет панику и отменет компиляцию, ссылаясь на ошибку в синтаксисе.

Таким образом, важно понимать, что вы должны при написании правил в патернах указывать наиболее общую форму, а не какую-либо конкретную.

Для защиты от дальнейших изменений синтаксиса, изменяющих интерпретацию входа макроса, `macro_rules!` ограничивает то, что может следовать за различными метапеременными. Полный список для Rust 1.3 следующий:

- `item`: что угодно.
- `block`: что угодно.

- `stmt`: `=>` `,` `;`
- `pat`: `=>` `,` `=` `if` `in`
- `expr`: `=>` `,` `;`
- `ty`: `,` `=>` `:` `=` `>` `;` `as`
- `ident`: что угодно.
- `path`: `,` `=>` `:` `=` `>` `;` `as`
- `meta`: что угодно.
- `tt`: что угодно.

В дополнение, `macro_rules!` обычно запрещает повторению идти за другим повторением, даже если по содержанию они не конфликтуют.

Еще одна особенность подстановки, которая обычно удивляет людей, это то, что подстановка *не* основана на токенах, несмотря на то, что *выглядит* именно так. Вот простая демонстрация этого:

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

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

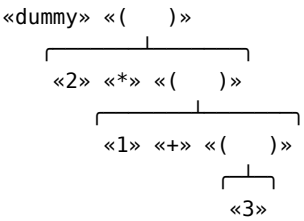
Помните, что `stringify!` - это встроенное расширение синтаксиса, которое берет все токены, которые ей дали на вход, и объединяет их в одну большую строку.

Вывод будет следующий:

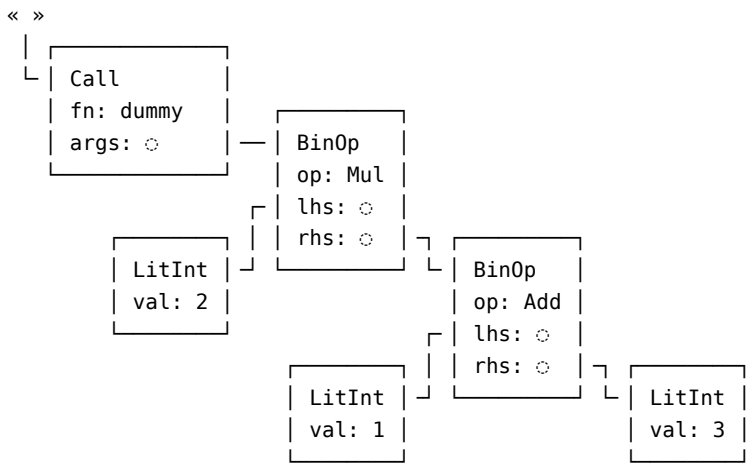
```
"dummy ( 2 * ( 1 + ( 3 ) ) )"
"dummy(2 * (1 + (3)))"
```

Заметьте, что, *несмотря на* одинаковые входные данные, выводы разные. Это происходит из-за того, что первый вызов преобразует в строку выражение из деревьев токенов, в то время как второй преобразует в строку *узел выражения AST*.

Для визуализации разницы - вот, с чем на входе, макрос `stringify!` вызывается в первом случае:



...а вот, с чем, он вызывается во втором:



Как вы можете заметить, здесь именно *одно* дерево токенов, которое содержит AST, который был распарсен из входа вызова `capture_expr_then_stringify!`. Отсюда, то, что вы здесь видите в выводе - не преобразованные в строку токены, а преобразованные в строку *узлы AST*.

Эта особенность имеет и другие последствия. Представьте следующее:

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

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

fn main() {
    println!("{}\n{}\n{}\n",
        match_tokens!((caravan)),
        match_tokens!(3 + 6),
        match_tokens!(5));
    println!("{}\n{}\n{}",
        capture_then_match_tokens!((caravan)),
        capture_then_match_tokens!(3 + 6),
        capture_then_match_tokens!(5));
}
```

Вывод:

```
got an identifier
got an addition
```

```
got something else

got something else
got something else
got something else
```

Распарсив вход в узлы AST, подстановочный результат становиться *неразрывным*; *т.е.* вы не можете проверить содержимое или найти совпадение с образцом внутри него никогда больше.

Вот *еще* пример, который может особенно смутить:

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

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

fn main() {
    println!(
        "{}\n{}\n{}\n{}",
        what_is!(#[no_mangle]),
        what_is!(#[inline]),
        capture_then_what_is!(#[no_mangle]),
        capture_then_what_is!(#[inline]),
    );
}
```

Вывод:

```
no_mangle attribute
inline attribute
something else (# [ no_mangle ])
something else (# [ inline ])
```

Единственный способ избежать этого - использовать захват в метапеременные, используя `tt` или `ident` типы. Если вы будете использовать захват в метапеременные друго типа, единственное, что вы сможете сделать с результатом - пробросить его прямо на вывод.

Гигиена

Макросы в Rust *частично* гигиеничны. Конкретно, они гигиеничны, когда дело доходит до большинства идентификаторов, но *негигиеничны*, когда используются обощенные типы или время жизни.

Гигиена проявляется в виде добавления всем идентификаторам невидимого значения “контекста синтаксиса”. Если два идентификатора сравниваются, то сравниваются *как* текстовые имена идентификаторов, *так и* контексты синтаксиса, чтобы сказать, что идентификаторы идентичны.

Для того, чтобы это проиллюстрировать, возьмем следующий код:

Будем использовать цвет фона, чтобы отметить контекст синтаксиса. Теперь давайте развернем вызов макроса:

Первое, напомним, что вызовы `macro_rules!` эффективно *исчезают* после развертывания.

Второе, если вы попытаетесь скомпилировать этот код, компилятор ответит что-то навроде следующего:

```
<anon>:11:21: 11:22 error: unresolved name `a`
<anon>:11 let four = using_a!(a / 10);
```

Заметьте, что цвет фона (*т.е.* контекст синтаксиса) для развернутого макроса *меняется* как часть развертывания. Каждое развертывание макроса дает новый уникальный контекст синтаксиса для своего содержимого. Как результат, здесь *два различных `a`* в развернутом коде: один в первом контексте синтаксиса, а второй во втором. Другими словами, a - это не тот же идентификатор, что и a, какими бы похожими они не казались.

Тем не менее, токены, которые были подставлены *в* развертывание, *сохраняют* их оригинальный контекст синтаксиса (в силу того, что были предоставлены макросу, а не являлись частью самого макроса). Таким образом, решением будет - исправить макрос следующим образом:

Что, после развертывание станет:

Компилятор примет этот код, потому что используется теперь только одна `a`.

Идентификаторы, не являющиеся идентификаторами

Есть два токена, с которыми вы, вероятно, сталкивались когда-либо, которые *выглядят* как идентификаторы, но ими не являются. Кроме случаев, когда они выступают именно в роли идентификаторов.

Первый - `self`. Это *совершенно точно* ключевое слово. Однако, ему также случается быть определением идентификатора. В обычном коде на Rust `self` не может быть интерпретирован как идентификатор, но в макросе такая возможность *есть*:

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

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

```
fn main() {
    println!("{}", what_is!(self));
    println!("{}", call_with_ident!(what_is(self)));
}
```

Вывод следующий:

```
the keyword `self`
the keyword `self`
```

Но в этом нет смысла; `call_with_ident!` требует идентификатор, находит совпадение с образцом, и заменяет его! Получается `self` в одно и то же время как является ключевым словом, так им и не является. Возможно, вам непонятно, в каких случаях это может быть важно. Рассмотрим такой пример:

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

struct Dummy(i32);

impl Dummy {
    fn double(self) -> Dummy {
        make_mutable!(self);
        self.0 *= 2;
        self
    }
}
#
# fn main() {
#     println!("{:?}", Dummy(4).double().0);
# }
```

Возникнет ошибка при компиляции:

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

То есть макрос спокойно и успешно сопоставил `self` с идентификатором, позволив вам использовать его в случае, когда вам на самом деле так его использовать нельзя. Но, постойте; он каким-то образом помнит еще и что `self` является ключевым словом, даже если это идентификатор. Поэтому, пример ниже будет работать, так ведь?

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

```
struct Dummy(i32);

impl Dummy {
    fn double(self) -> Dummy {
        make_self_mutable!(mut_self);
        mut_self.0 *= 2;
        mut_self
    }
}
#
# fn main() {
#     println!("{:?}", Dummy(4).double().0);
# }
```

Здесь выдаётся ошибка:

```
<anon>:2:33: 2:37 error: `self` is not available in a static method. Maybe a `self` ar
↳ gument is missing? [E0424]
<anon>:2     ($i:ident) => {let mut $i = self;};
                                         ^~~~
```

Это тоже не имеет никакого смысла. `self` и не находится в статическом методе. Очень похоже, что компилятор жалуется, что `self`, который он пытается использовать - это не тот же самый `self`... получается у ключевого слова `self` гигиена, как у... идентификатора.

```
macro_rules! double_method {
    ($body:expr) => {
        fn double(mut self) -> Dummy {
            $body
        }
    };
}

struct Dummy(i32);

impl Dummy {
    double_method! {{
        self.0 *= 2;
        self
    }}
}
#
# fn main() {
#     println!("{:?}", Dummy(4).double().0);
# }
```

Та же ошибка. Что насчет...

```
macro_rules! double_method {
    ($self_:ident, $body:expr) => {
        fn double(mut $self_) -> Dummy {
            $body
        }
    };
}

struct Dummy(i32);

impl Dummy {
    double_method! {self, {
        self.0 *= 2;
        self
    }}
}
```

Наконец-то, *заработало*. Получается `self` является как ключевым словом, *так и* идентификатором - как ему захочется. Конечно, это сработает и для других, похожих конструкций, не так ли?

```
macro_rules! double_method {
    ($self_:ident, $body:expr) => {
        fn double($self_) -> Dummy {
            $body
        }
    };
}

struct Dummy(i32);

impl Dummy {
    double_method! {_, 0}
}
#
# fn main() {
#     println!("{:?}", Dummy(4).double().0);
# }
```

```
<anon>:12:21: 12:22 error: expected ident, found _
<anon>:12     double_method! {_, 0}
                             ^
```

Нет, конечно нет. `_` это ключевое слово, которое правильно в образцах и выражениях, но при этом *не является* идентификатором в том смысле, как им является `self`, несмотря на то, что описание идентификатора абсолютно такое же.

Вам может показаться, что это можно обойти, используя `$self_:pat`; и в таком случае, `_` подойдет! За исключением того, что... нет, не подойдет, потому что `self` не является образцом. Улыбаемся и машем.

Как-то обойти это в данном случае (в случае если вам нужна комбинация этих токенов) можно, используя совпадения с образцом по `tt` вместо этого.

Отладка

`rustc` предоставляет набор инструментов для отладки макросов. Один из самых полезных - `trace_macros!`, который представляет собой директиву компилятору, требующую от него делать дамп каждого вызова макроса до развертывания. Например, имеем следующее:

```
#![feature(trace_macros)]

macro_rules! each_tt {
    () => {};
    ($_tt:tt $($rest:tt)*) => {each_tt!($($rest)*);};
}

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

Вывод:

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

Это *особенно* неоценимо при отладке макросов с большой глубиной рекурсии. Вы можете также включить эту директиву из командной строки добавив `-Z trace-macros` к команде.

Во вторых, есть `log_syntax!`, который заставляет компилятор выводить все токены, которые подаются на вход. Например, это заставит компилятор петь песню:

```
#![feature(log_syntax)]

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

sing! {
    ^ < @ < . @ *
    '\x08' '{' '"' _ # ' '
    - @ '$' && / _ %
```

```
    ! ( '\t' @ | = >
    ; '\x08' '\'' + '$' ? '\x7f'
    , # '"' ~ | ) '\x07'
}
```

Эту команду можно использовать, чтобы сделать более точную отладку, чем `trace_macros!`.

Иногда, то, во что макрос *разворачивается*, представляет проблему. Для отладки можно использовать аргумент компилятора `--pretty`. Ниже пример:

```
// Короткая инициализация  `String`.
macro_rules! S {
    ($e:expr) => {String::from($e)};
}

fn main() {
    let world = S!("World");
    println!("Hello, {}!", world);
}
```

скомпилировано со следующей командой :

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

выдает следующий вывод (исправлен с форматированием):

```
#![feature(no_std, prelude_import)]
#![no_std]
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
extern crate std as std;
// Shorthand for initialising a `String`.
fn main() {
    let world = String::from("World");
    ::std::io::_print(::std::fmt::Arguments::new_v1(
        {
            static __STATIC_FMTSTR: &'static [&'static str]
                = &["Hello, ", "!\n"];
            __STATIC_FMTSTR
        },
        &match (&world,) {
             (__arg0,) => [
                ::std::fmt::ArgumentV1::new(__arg0, ::std::fmt::Display::fmt)
            ],
        }
    ));
}
```

Другие опции `--pretty` можно посмотреть так - `rustc -Z unstable-options --help -v`; полный список опций не приводится из-за того, чтобы, как понятно из имени, такой список может поменяться в любое время.

Области видимости

Способ, которым макросам задается область видимости, интуитивно понятен. Во- первых, в отличие от всего остального в языке, макросы остаются видимыми в под- модулях.

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

> **Внимание**: Для данных примеров, помните, что все они *ведут себя* одинаково*, даже если модули располагаются в разных файлах.

Во-вторых, *также* будучи непохожим на все остальное в языке, макросы доступны *только* после их определения. Обратите внимание на этот пример, который демонстрирует, что макросы не “выплывают” из своей области видимости:

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

Дабы прояснить этот момент, эта зависимость от лексического порядка также проявляется, если вынести макрос вовне:

```
mod a {
    // X!(); // не определен
}
macro_rules! X { () => {}; }
```

```
mod b {
    X!(); // определен
}
mod c {
    X!(); // определен
}
```

Однако, эта зависимость *не* проявляется внутри самих макросов:

```
mod a {
    // X!(); // не определен
}
macro_rules! X { () => { Y!(); }; }
mod b {
    // X!(); // определен, но Y! - не определен
}
macro_rules! Y { () => {}; }
mod c {
    X!(); // определен, как и Y!
}
```

Макросы могут быть экспортированы из модуля через атрибут `#[macro_use]` .

```
mod a {
    // X!(); // не определен
}
#[macro_use]
mod b {
    macro_rules! X { () => {}; }
    X!(); // определен
}
mod c {
    X!(); // определен
}
```

Помните, что это может работать несколько странным образом из-за того, что идентификаторы в макросе (включая и другие макросы внутри) разрешаются только после развёртывания:

```
mod a {
    // X!(); // не определен
}
#[macro_use]
mod b {
    macro_rules! X { () => { Y!(); }; }
    // X!(); // определен, но Y! - не определен
}
macro_rules! Y { () => {}; }
mod c {
```

```
    X!(); // определен, как и Y!
}
```

Еще одной сложностью является то, что `#[macro_use]`, применяемый к `внешним контейнерам`, *не* ведет себя таким образом: такие объявления фактически *поднимаются* наверх в модуле. Поэтому, считая, что `X!` определен во внешнем контейнере `mac`:

```
mod a {
    // X!(); // определен,но Y! - не определен
}
macro_rules! Y { () => {}; }
mod b {
    X!(); // определен, как и Y!
}
#[macro_use] extern crate macs;
mod c {
    X!(); // определен, как и Y!
}
# fn main() {}
```

Наконец, помните, что поведение внутри областей видимости определяется также как и для *функций*, за исключением `#[macro_use]` (для которого определяется по- другому):

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

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

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

Учитывая эти правила для областей видимости, общим советом будет, ставить все макросы наверх в вашем корневом модуле, перед всеми остальными. Это гарантирует, что они будут доступны *постоянно*.

Импорт/Экспорт

Существует два способа выставить макрос наружу, в более широкую область видимости. Первый - использовать атрибут `#[macro_use]`. Его можно применять *как* к модулям, так и к внешним контейнерам. Например:

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

X!();
```

Макрос можно экспортировать из текущего контейнера, используя атрибут `#[macro_export]`. Помните, что такой способ *игнорирует* все области видимости.

Если у нас есть следующее описание библиотечного пакета `macs`:

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

// X! и Y! *не* определены здесь, а *экспортированы*,
// несмотря на то, что `macros` является приватным.
```

Следующий код работает, как и ожидается:

```
X!(); // X определен
#[macro_use] extern crate macs;
X!();
#
# fn main() {}
```

Помните, что вы можете использовать `#[macro_use]`, который ссылается на внешний контейнер, *только* из корневого модуля.

Наконец, если макросы импортируются из внешнего контейнера, можно контролировать *какой* именно макрос импортируется. Можете использовать эту возможность, чтобы избежать раздутия пространства имен, или чтобы переопределить какой-либо конкретный макрос, например, так:

```
// Импортируем  *только* макрос `X!` .
#[macro_use(X)] extern crate macs;

// X!(); // X определен, а Y! не определен
```

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

X!(); // X определен, и Y! определен

fn main() {}
```

При экспортировании макросов, часто полезно ссылаться на имена, не связанные с макросами, в содержащем макросы контейнере. Из-за того, что контейнеры могут переименовываться, есть специальная переменная замены: `$crate`. Она будет *всегда* разворачиваться в префикс абсолютного пути к содержащему макрос контейнеру (*например*, `:: macs`).

Помните, что такой подход *не* работает для самих макросов, из-за того, что макросы не используют стандартные преобразования имен каким бы то ни было образом. Поэтому вы не можете использовать, что-то вроде `$crate::Y!` для того, чтобы сослаться на указанный макрос внутри вашего контейнера. Следствием этой особенности и особенности подхода к выборочному импорту через `#[macro_use]`, является то, что на настоящей момент *нет способа* гарантировать, что любой ваш макрос будет доступен при импорте в другом контейнере.

Рекомендуется *всегда* использовать абсолютные пути к именам, не связанным с макросами, чтобы избежать конфликтов, *включая* конфликты с именами в стандартной библиотеке.

3

Макросы, Практическое введение

Эта глава расскажет о системе Rust макрос-по-примеру, используя относительно простой, практичный пример. Мы *не* будем пытаться объяснить всю сложность системы; нашей целью является сделать так, чтобы вы чувствовали себя комфортно с макросами и понимали как и почему они пишутся.

Есть также глава по макросам в Rust Book[1], в которой объяснения даются на более высоком уровне и методическое введение - глава в этой книге, которая объясняет систему макросов подробно.

Маленький контекст

> **Заметка**: не паникуйте! Дальше пойдет разговор о математике. Вы можете спокойно пропустить этот раздел, если хотите добраться до самого мяса этой статьи.

Если вы не знали, рекуррентное соотношение - это последовательность, в которой каждое значение определяется в терминах одного или нескольких *предыдущих*, с одним или несколькими начальными значениями. Например, последовательность Фибоначчи[2] можно описать такой связью:


```
<span class="katex"><span class="katex-inner"><span style="height: 0.68333em;" class="strut"></span><span style
```


Итак, первые два числа в последовательности - 0 и 1, а третье - F0 + F1 = 0 + 1 = 1, четвертое - F1 + F2 = 1 + 1 = 2, и так далее.

Итак, *из-за того, что* такая последовательность может продолжаться бесконечно, описание функции `fibonacci` получилось довольно хитрым, ведь вам же не нужно возвращать полный список элементов. Все, что вы *хотите* - это вернуть что-то, лениво просчитав столько элементов, сколько надо.

---

[1] http://doc.rust-lang.org/book/macros.html

[2] https://en.wikipedia.org/wiki/Fibonacci_number

В Rust, это называется создать `Iterator`. Это не особо *трудно*, но тут потребуется довольно много рутинных действий: нужно определить свой тип, понять, какое состояние хранить в нем, и затем реализовать типаж `Iterator` для него.

Получается, рекуррентное соотношение - это настолько просто, что почти от всех конкретных деталей можно абстрагироваться и создать маленький генератор кода на базе макроса.

Итак, сказав все, что нужно, давайте начнем.

Создание

Обычно, если я работаю над новым макросом, первое, что я решаю - это то, как будет выглядеть вызов макроса. В данном конкретном случае в первом приближении получится следующее:

```
let fib = recurrence![a[n] = 0, 1, ..., a[n-1] + a[n-2]];

for e in fib.take(10) { println!("{}", e) }
```

После этого мы можем обратить внимание на определении макроса, даже если мы не уверены во что, он должен разворачиваться. Это полезно, потому что если вам пока непонятно, как разбирать входящий синтаксис, то, *возможно*, вам придется поменять его.

```
macro_rules! recurrence {
    ( a[n] = $($inits:expr),+ , ... , $recur:expr ) => { /* ... */ };
}
```

Подразумевая, что вы не знакомы с синтаксисом, позвольте мне объяснить это определение. Здесь представлено определение макроса, выполняемому через систему `macro_rules!`, названное `recurrence!`. У этого макроса одно правило разбор. Это правило говорит, что вход данного макроса должен совпадать с:

- последовательностью литеральных токенов `a [ n ] =`,
- повторяющейся последовательностью (`$( ... )`), используя `,` как разделитель, и одно или больше (`+`) повторений:
  - валидного *выражения*, захваченного в переменную `inits` (`$inits:expr`)
- последовательностью литеральных токенов `, ... ,`,
- вылидным *выражением*, захваченным в переменную `recur` (`$recur:expr`).

Наконец, правило говорит, *если* вход совпадает с образцом, то вызов макроса нужно заменить на последовательность токенов `/* ... */`.

Стоит отметить, что `inits`, как понятно из названия, на самом деле содержит *все* выражения, которые совпадают на этой позиции, а не только первое или последнее. Более того, он захватывает их *как последовательность*, а не склеивая их всех вместе в одно выражение. Также помните, что вы можете изменить повторение на “ноль и больше”, используя * вместо +. Поддержки “одного или нескольких” или еще более конкретного числа повторений тут нет.

В качестве упражнения, давайте возьмем прелагаемый вход и пропустим его через правило, чтобы посмотреть, как оно выполняется. Колонка “позиция”, показывающая, какая часть паттерна должна совпасть следующей, отмечается “⌂”. Помните, что в некоторых случаях, может быть больше одного возможного “следующего” элемента, с которым найдется совпадение. “Вход” будет содержать все токены, которые еще *не* были обработаны. `inits` и `recur` будут содержать содержимое их связей.

```
<thead>
    <tr>
        <th>Позиция</th>
        <th>Вход</th>
        <th><code>inits</code></th>
        <th><code>recur</code></th>
    </tr>
</thead>
<tbody class="small-code">
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>⌂</code></td>
        <td><code>a[n] = 0, 1, ..., a[n-1] + a[n-2]</code></td>
        <td></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code> ⌂</code></td>
        <td><code>[n] = 0, 1, ..., a[n-1] + a[n-2]</code></td>
        <td></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>  ⌂</code></td>
        <td><code>n] = 0, 1, ..., a[n-1] + a[n-2]</code></td>
        <td></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>   ⌂</code></td>
        <td><code>] = 0, 1, ..., a[n-1] + a[n-2]</code></td>
        <td></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>     ⌂</code></td>
        <td><code>= 0, 1, ..., a[n-1] + a[n-2]</code></td>
        <td></td>
```

```
    <td></td>
</tr>
<tr>
    <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
        <code>       ⌂</code></td>
    <td><code>0, 1, ..., a[n-1] + a[n-2]</code></td>
    <td></td>
    <td></td>
</tr>
<tr>
    <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
        <code>         ⌂</code></td>
    <td><code>0, 1, ..., a[n-1] + a[n-2]</code></td>
    <td></td>
    <td></td>
</tr>
<tr>
    <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
        <code>                     ⌂  ⌂</code></td>
    <td><code>, 1, ..., a[n-1] + a[n-2]</code></td>
    <td><code>0</code></td>
    <td></td>
</tr>
<tr>
    <td colspan="4" style="font-size:.7em;">
```

Внимание/em>: здесь два ⌂, потому что следующий входной токен можно с/em> запятой-разделителем em>между/em>em> элементами в и с/em> запятой em>после/em> повторения. Система макроса будет <иметь ввиду обе возможности, до тех пор пока не сможет определить, какую <выбрать.

```
    </td>
</tr>
<tr>
    <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
        <code>         ⌂                ⌂</code></td>
    <td><code>1, ..., a[n-1] + a[n-2]</code></td>
    <td><code>0</code></td>
    <td></td>
</tr>
<tr>
    <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
        <code>                     ⌂  ⌂ <s>⌂</s></code></td>
    <td><code>, ..., a[n-1] + a[n-2]</code></td>
    <td><code>0</code>, <code>1</code></td>
    <td></td>
</tr>
<tr>
    <td colspan="4" style="font-size:.7em;">
```

Внимание/em>: третий, подчеркнутый маркер показывает, что система макроса <после обработки последнего токена удалила одну из предыдущих возможных веток.

```
        </td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>         ⌂               ⌂</code></td>
        <td><code>..., a[n-1] + a[n-2]</code></td>
        <td><code>0</code>, <code>1</code></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>         <s>⌂</s>                    ⌂</code></td>
        <td><code>, a[n-1] + a[n-2]</code></td>
        <td><code>0</code>, <code>1</code></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>                              ⌂</code></td>
        <td><code>a[n-1] + a[n-2]</code></td>
        <td><code>0</code>, <code>1</code></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>                                          ⌂</code></td>
        <td></td>
        <td><code>0</code>, <code>1</code></td>
        <td><code>a[n-1] + a[n-2]</code></td>
    </tr>
    <tr>
        <td colspan="4" style="font-size:.7em;">

            <em>Внимание</em>: этот конкретный шаг должен объяснить, что
            <такая связь, как <tt>$recur:expr</tt>, заменит <em>все
            <выражение</em>, используя знания компилятора о том, что считать
            <правильным выражением. Как будет рассказано дальше, вы можете
            <так делать и с другими конструкциями языка.


        </td>
    </tr>
</tbody>
```

Что нужно понять из этого всего, это то, что система макроса будет *пытаться* последовательно сопоставить предложенные на входе токены с представленным правилом. Мы еще вернемся

к “попыткам”.

Теперь, давайте напишем конечную, полностью развернутую форму. Для этого развертывания, я определил что-то вроде этого:

```
let fib = {
    struct Recurrence {
        mem: [u64; 2],
        pos: usize,
    }
```

Это будет тип итератора. `mem` будет буфером в памяти, который будет содержать несколько последних значений, достаточных для продолжения рекуррентных вычислений. `pos` должен следить за значением `n`.

> **В сторону**: Я выбрал `u64` как “достаточно большой” тип для элементов этой последовательности. Не волнуйтесь о том, как это будет работать для *других* последовательностей; мы еще вернемся к этому.

```
    impl Iterator for Recurrence {
        type Item = u64;

        #[inline]
        fn next(&mut self) -> Option<u64> {
            if self.pos < 2 {
                let next_val = self.mem[self.pos];
                self.pos += 1;
                Some(next_val)
```

Нам нужна ветка, которая будет заполнять начальные значения последовательности; ничего необычного.

```
            } else {
                let a = /* something */;
                let n = self.pos;
                let next_val = (a[n-1] + a[n-2]);

                self.mem.TODO_shuffle_down_and_append(next_val);

                self.pos += 1;
                Some(next_val)
            }
        }
    }
```

Тут все немножко посложнее; мы еще вернемся и посмотрим, *как* именно определить `a`. Также, `TODO_shuffle_down_and_append` это еще один временный заполнитель; Мне нужно что-то, что заменит `next_val` в конце массива, сдвинув все остальное вниз на одну позицию и удалив 0й элемент.

```
    Recurrence { mem: [0, 1], pos: 0 }
};

for e in fib.take(10) { println!("{}", e) }
```

Наконец, вернем экземпляр нашей новой структуры, по которому затем можно выполнять итерации. Объединяя все, полное развертывание будет таким:

```
let fib = {
    struct Recurrence {
        mem: [u64; 2],
        pos: usize,
    }

    impl Iterator for Recurrence {
        type Item = u64;

        #[inline]
        fn next(&mut self) -> Option<u64> {
            if self.pos < 2 {
                let next_val = self.mem[self.pos];
                self.pos += 1;
                Some(next_val)
            } else {
                let a = /* something */;
                let n = self.pos;
                let next_val = (a[n-1] + a[n-2]);

                self.mem.TODO_shuffle_down_and_append(next_val.clone());

                self.pos += 1;
                Some(next_val)
            }
        }
    }

    Recurrence { mem: [0, 1], pos: 0 }
};

for e in fib.take(10) { println!("{}", e) }
```

> **В сторону**: Да, это *действительно* означает, что мы определяем разные структуры `Recurrence` и их реализации при каждом вызове макроса. Большинство из этого мы оптимизируем в конце разумным использованием атрибута `#[inline]`.

Очень полезно проверять развертывание во время его написания. Если вы заметите что-то в развертывании, что должно различаться при вызове, но на текущий момент не входит в синтаксис макроса, вы должны решить, где ввести это. В данном случае мы добавили `u64`, но пользователь может захотеть что-то другое. Поэтому давайте исправим это.

```
macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ , ... , $recur:expr ) => { /* ... */ };
}

/*
let fib = recurrence![a[n]: u64 = 0, 1, ..., a[n-1] + a[n-2]];

for e in fib.take(10) { println!("{}", e) }
*/
```

Здесь я добавил новый захват в метапеременную: `sty`, которая обозначает тип.

> **В сторону**: Если вам интересно, кусок, идущий после двоеточия в захвате, может быть одним из определенных типов сопоставления синтаксиса. Самые распространенные - это `item`, `expr` и `ty`. Полное объяснение дается в Макросы, Методическое введение; `macro_rules!` (Захват).
>
> Есть тут еще одна интересная вещь: в интересах будущего улучшения языка компилятор ограничивает то, какие токены вы можете ставить *после* сопоставления, основываясь на его типе. Обычно, с этим сталкиваешься, когда выполняешь сопоставление с выражениями и утверждениями; за ними может идти *только* `=>`, `,` и `;`.
>
> Полный список можно найти в Макросы, Методическое введение; Мелочи; Вернемся к Метапеременным и Развертыванию.

Индексирование и сдвиг

Я не буду уделять много внимания этой части, потому что она не затрагивает напрямую макросы. Мы хотим сделать так, чтобы пользователь смог обращаться к предыдущим значениям в последовательности, индексируя `a`; мы хотим, чтобы это работало как двигающееся окно, в котором видны только последние (в данном случае, 2) элементы последовательности.

Можем сделать это довольно просто, используя тип оболочку:

```
struct IndexOffset<'a> {
    slice: &'a [u64; 2],
    offset: usize,
}

impl<'a> Index<usize> for IndexOffset<'a> {
    type Output = u64;

    #[inline(always)]
    fn index<'b>(&'b self, index: usize) -> &'b u64 {
        use std::num::Wrapping;

        let index = Wrapping(index);
```

```
        let offset = Wrapping(self.offset);
        let window = Wrapping(2);

        let real_index = index - offset + window;
        &self.slice[real_index.0]
    }
}
```

> **В сторону**: из-за того, что время жизни *вгоняет в ступор* новичков в Rust, побыстрому объясню: `'a` и `'b` - это параметры времени жизни, которые используются для слежения за ссылками (*т.е.* захваченными указателями на какие-то данные). В этом случае, `IndexOffset` захватывает ссылку на наши данные итератора, поэтому нам надо следить, как долго ей можно удерживать их в себе, и для этого используется `'a`.
>
> `'b` используется, потому что функция `Index::index` (то, как на самом деле реализуется синтаксис под-скрипта) *также* параметризирована временем жизни, в расчете на возврат захваченной ссылки. `'a` и `'b` не обязательно совпадают во всех возможных случаях. Анализатор заимствований должен убедиться, что, даже если мы явно не сопоставим `'a` и `'b` друг с другом, мы на самом деле по неосторожности не нарушим целостность памяти.

Меняем определение `a` на:

```
let a = IndexOffset { slice: &self.mem, offset: n };
```

Единственный оставшийся вопрос - что насчет `TODO_shuffle_down_and_append`? Я не смог найти метод в стандартной библиотеке, совпадающий по семантике с тем, что мне нужно, но его и нетрудно сделать ручками самому.

```
{
    use std::mem::swap;

    let mut swap_tmp = next_val;
    for i in (0..2).rev() {
        swap(&mut swap_tmp, &mut self.mem[i]);
    }
}
```

Здесь новое значение перемещается в конец массива, сдвигая остальные элементы на один вниз.

> **В сторону**: делая так, знайте, что этот код будет работать также и для типов, не поддерживающих копирование.

Рабочий код теперь выглядит так:

```
macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ , ... , $recur:expr ) => { /* ... */ };
}

fn main() {
    /*
    let fib = recurrence![a[n]: u64 = 0, 1, ..., a[n-1] + a[n-2]];

    for e in fib.take(10) { println!("{}", e) }
    */
    let fib = {
        use std::ops::Index;

        struct Recurrence {
            mem: [u64; 2],
            pos: usize,
        }

        struct IndexOffset<'a> {
            slice: &'a [u64; 2],
            offset: usize,
        }

        impl<'a> Index<usize> for IndexOffset<'a> {
            type Output = u64;

            #[inline(always)]
            fn index<'b>(&'b self, index: usize) -> &'b u64 {
                use std::num::Wrapping;

                let index = Wrapping(index);
                let offset = Wrapping(self.offset);
                let window = Wrapping(2);

                let real_index = index - offset + window;
                &self.slice[real_index.0]
            }
        }

        impl Iterator for Recurrence {
            type Item = u64;

            #[inline]
            fn next(&mut self) -> Option<u64> {
                if self.pos < 2 {
                    let next_val = self.mem[self.pos];
                    self.pos += 1;
                    Some(next_val)
```

```
                } else {
                    let next_val = {
                        let n = self.pos;
                        let a = IndexOffset { slice: &self.mem, offset: n };
                        (a[n-1] + a[n-2])
                    };

                    {
                        use std::mem::swap;

                        let mut swap_tmp = next_val;
                        for i in (0..2).rev() {
                            swap(&mut swap_tmp, &mut self.mem[i]);
                        }
                    }

                    self.pos += 1;
                    Some(next_val)
                }
            }
        }

        Recurrence { mem: [0, 1], pos: 0 }
    };

    for e in fib.take(10) { println!("{}", e) }
}
```

Заметьте, что я поменял порядок объявления `n` и `a`, а также обернул их (вместе с рекуррентным выражением) в блок. Причина для первого довольна тривиальна (`n` должна быть определена раньше, чтобы я мог использовать его для `a`). Причина второго - это то, что заимствованная ссылка `&self.mem` не дает произойти дальнейшим сдвигам (вы не можете изменить то, что связано в другом месте). Оборачивание в блок гарантирует, что заимствование `&self.mem` заканчивается до него.

Между прочим, единственной причиной, по которой код, делающий сдвиг `mem`, находится внутри блока, является желание приблизить область видимости, в которой доступен `std::mem::swap`, просто ради аккуратности.

Если мы выполним этот код, мы получим:

```
0
1
2
3
5
8
13
21
34
```

Это успех! Теперь, давайте скопируем и вставим код в развертывание макроса, и заменим развернутый код на его вызов. Получаем:

```
macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ , ... , $recur:expr ) => {
        {
            /*
                Идущий дальше код, это *буквально* код выше,
                вырезанный и вставленный на новую позицию. Никаких изменений
                больше не выполнялось.
            */

            use std::ops::Index;

            struct Recurrence {
                mem: [u64; 2],
                pos: usize,
            }

            struct IndexOffset<'a> {
                slice: &'a [u64; 2],
                offset: usize,
            }

            impl<'a> Index<usize> for IndexOffset<'a> {
                type Output = u64;

                #[inline(always)]
                fn index<'b>(&'b self, index: usize) -> &'b u64 {
                    use std::num::Wrapping;

                    let index = Wrapping(index);
                    let offset = Wrapping(self.offset);
                    let window = Wrapping(2);

                    let real_index = index - offset + window;
                    &self.slice[real_index.0]
                }
            }

            impl Iterator for Recurrence {
                type Item = u64;

                #[inline]
                fn next(&mut self) -> Option<u64> {
                    if self.pos < 2 {
                        let next_val = self.mem[self.pos];
                        self.pos += 1;
                        Some(next_val)
```

```
                    } else {
                        let next_val = {
                            let n = self.pos;
                            let a = IndexOffset { slice: &self.mem, offset: n };
                            (a[n-1] + a[n-2])
                        };

                        {
                            use std::mem::swap;

                            let mut swap_tmp = next_val;
                            for i in (0..2).rev() {
                                swap(&mut swap_tmp, &mut self.mem[i]);
                            }
                        }

                        self.pos += 1;
                        Some(next_val)
                    }
                }
            }

            Recurrence { mem: [0, 1], pos: 0 }
        }
    };
}

fn main() {
    let fib = recurrence![a[n]: u64 = 0, 1, ..., a[n-1] + a[n-2]];

    for e in fib.take(10) { println!("{}", e) }
}
```

Очевидно, что мы не *используем* еще захват в метапеременные, но мы можем добавить его довольно просто. Однако, если мы попытаемся скомпилировать код, `rustc` вернет ошибку, говорящую нам:

```
recurrence.rs:69:45: 69:48 error: local ambiguity: multiple parsing options: built-in
↳ NTs expr ('inits') or 1 other options.
recurrence.rs:69     let fib = recurrence![a[n]: u64 = 0, 1, ..., a[n-1] + a[n-2]];
                                                             ^~~
```

Вот тут мы и попались в ограничение `macro_rules`. Проблемой является вторая запятая. Когда `macro_rules` видит ее во время развертывания, он не может решить разобрать ему *следующее* выражение как `inits` или как `...`. К сожалению, он не такой умный, чтобы понять, что `...` не является правильным выражением, и поэтому он сдается. Теоретически, все должно работать, но на самом деле нет.

> **В сторону**: Я *сделал* этот раздел таким, каким он должен быть. В общем, он *должен был бы* работать как описано, но не в этом случае. Устройство `macro_rules`, как оно

> сейчас есть, имеет свои слабости, и всегда стоит помнить, что в некоторых случаях вам придется исказить код немного, чтобы заставить его работать.
>
> В этом *конкретном* случае две проблемы. Первая - система макроса не знает, что составляет некоторые грамматические элементы, а что нет (*например*, выражения); это работа парсера. Поэтому, она не знает, что `...` не является выражением. Вторая - она не может попытаться захватить составной грамматический элемент (такой, как выражение), без его 100% совпадения.
>
> Другими словами, она может попросить парсер попытаться разобрать вход как выражение, а парсер ответит на любую проблему отменой с ошибкой. Единственным способом, как система макроса сможет работать - просто попытаться избежать ситуаций, в которых это может стать проблем.
>
> К положительному моменту можно отнести то, что *никто* не в восторге от этого. Ключевое слово `macro` уже зарезервировано для будущего более строго определения системы макроса. А пока, страдаем..

К счастью, решение очень простое: мы удаляем запятую из синтаксиса. Чтобы удержать равновесие, мы удалим *обе* запятые вокруг `...`:

```
macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ ... $recur:expr ) => {
//                                     ^~~ изменено
        /* ... */
    };
}

fn main() {
    let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-2]];
//                                           ^~~ изменено

    for e in fib.take(10) { println!("{}", e) }
}
```

Это успех! Теперь можем заменить вещи из *развертывания* на *захваченные метапеременные*.

Замена

Замена того, что вы захватили в метапеременную, очень проста; вы заменяете содержимое захваченного в `$sty:ty` на `$sty`. Итак, давайте пройдемся и исправим все `u64`:

```
macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ ... $recur:expr ) => {
        {
            use std::ops::Index;

            struct Recurrence {
                mem: [$sty; 2],
//                    ^~~~ изменено
```

```
                pos: usize,
            }

            struct IndexOffset<'a> {
                slice: &'a [$sty; 2],
//                          ^~~~ изменено
                offset: usize,
            }

            impl<'a> Index<usize> for IndexOffset<'a> {
                type Output = $sty;
//                            ^~~~ изменено

                #[inline(always)]
                fn index<'b>(&'b self, index: usize) -> &'b $sty {
//                                                      ^~~~ изменено
                    use std::num::Wrapping;

                    let index = Wrapping(index);
                    let offset = Wrapping(self.offset);
                    let window = Wrapping(2);

                    let real_index = index - offset + window;
                    &self.slice[real_index.0]
                }
            }

            impl Iterator for Recurrence {
                type Item = $sty;
//                          ^~~~ изменено

                #[inline]
                fn next(&mut self) -> Option<$sty> {
//                                       ^~~~ изменено
                    /* ... */
                }
            }

            Recurrence { mem: [1, 1], pos: 0 }
        }
    };
}

fn main() {
    let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-2]];

    for e in fib.take(10) { println!("{}", e) }
}
```

Давайте решим вопрос посложнее: как превратить `inits` в массив литералов `[0,1]` *и* в массив типов, `[$sty; 2]`. Для первого мы можем сделать так:

```
            Recurrence { mem: [$($inits),+], pos: 0 }
//                             ^~~~~~~~~~ изменено
```

Это полная противоположность захвату в метапеременные: повтор `inits` один или несколько раз, каждый из которых отделяется запятой. Это развернется в ожидаемую последовательность токенов: `0, 1`.

Почему-то превратить `inits` в литерал `2` немного сложнее. Оказывается, нет прямого способа это сделать, но мы *можем* исправить это, написав второй макрос. Сделаем оба преобразованием за один ход.

```
macro_rules! count_exprs {
    /* ??? */
}
```

Очевидно: получив ноль выражений, вы ожидаете, что `count_exprs` развернется в литерал `0`.

```
macro_rules! count_exprs {
    () => (0);
//  ^~~~~~~~~~ добавлено
}
```

> **В сторону**: Вы должно быть заметили, что я использую круглые скоби вместо фигурных для развертывания. `macro_rules` все равно, *какие* скобки вы используете, если это любые из пар: `( )`, `{ }` или `[ ]`. На самом деле, вы можете использовать любые скобки и у самого макроса (*т.е.* скобки сразу после имени макроса), и скобки вокруг синтаксического правила или скобки вокруг соответствующего развертывания.
>
> Вы можете также использовать любые скобки при *вызове* макроса, но с ограничениями: макрос, вызываемый как `{ ... }` или `( ... );` будет *всегда* разбираться как *элемент* (*т.е.*, как объявление `struct` или `fn`). Это важно учитывать при использовании макросов внутри тела функций; это помогает устранить неоднозначность между тем, что делать - “разбирать как выражение” и “разбирать как утверждение”.

Что если у вас *одно* выражение? Этому должен соответствовать литерал `1`.

```
macro_rules! count_exprs {
    () => (0);
    ($e:expr) => (1);
//  ^~~~~~~~~~~~~~~~~ добавлено
}
```

Два?

```
macro_rules! count_exprs {
    () => (0);
    ($e:expr) => (1);
    ($e0:expr, $e1:expr) => (2);
//  ^~~~~~~~~~~~~~~~~~~~~~~~~~~ добавлено
}
```

Мы можем “упростить” это, по-другому выразив случай с двумя выражениями через рекурсию.

```
macro_rules! count_exprs {
    () => (0);
    ($e:expr) => (1);
    ($e0:expr, $e1:expr) => (1 + count_exprs!($e1));
//                            ^~~~~~~~~~~~~~~~~~~ изменено
}
```

Это работает, Rust сможет преобразовать `1 + 1` в константное значение. Что если у нас три выражения?

```
macro_rules! count_exprs {
    () => (0);
    ($e:expr) => (1);
    ($e0:expr, $e1:expr) => (1 + count_exprs!($e1));
    ($e0:expr, $e1:expr, $e2:expr) => (1 + count_exprs!($e1, $e2));
//  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ добавлено
}
```

> **В сторону**: Вам должно быть интересно, можем ли мы поменять порядок следования правил. В данном конкретном случае, *да*, но система макроса может быть иногда требовательна к этому и не пожелает работать. Если вы напишите макрос с несколькими правилами, который, вы клянетесь, должен работать, но выдает ошибки на неожиданных токенах, попробуйте поменять порядок следования правил.

Получившийся паттерн можете посмотреть ниже. Мы всегда можем уменьшить список выражений, выполняя совпадение с одним выражением, за которым следуют ноль и более выражений, разворачивая это в `1 +` оставшееся количество выражений.

```
macro_rules! count_exprs {
    () => (0);
    ($head:expr) => (1);
    ($head:expr, $($tail:expr),*) => (1 + count_exprs!($($tail),*));
//  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ изменено
}
```

> **ТВЭП**: это не *единственный*, и даже не *лучший* способ выполнить подсчет. Лучше использовать Подсчет из следующих глав.

Наконец, теперь мы можем изменить `recurrence` так, чтобы определить необходимый размер `mem`.

```
// добавлено:
macro_rules! count_exprs {
    () => (0);
    ($head:expr) => (1);
    ($head:expr, $($tail:expr),*) => (1 + count_exprs!($($tail),*));
}

macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ ... $recur:expr ) => {
        {
            use std::ops::Index;

            const MEM_SIZE: usize = count_exprs!($($inits),+);
//          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ добавлено

            struct Recurrence {
                mem: [$sty; MEM_SIZE],
//                        ^~~~~~~~ изменено
                pos: usize,
            }

            struct IndexOffset<'a> {
                slice: &'a [$sty; MEM_SIZE],
//                              ^~~~~~~~ изменено
                offset: usize,
            }

            impl<'a> Index<usize> for IndexOffset<'a> {
                type Output = $sty;

                #[inline(always)]
                fn index<'b>(&'b self, index: usize) -> &'b $sty {
                    use std::num::Wrapping;

                    let index = Wrapping(index);
                    let offset = Wrapping(self.offset);
                    let window = Wrapping(MEM_SIZE);
//                                        ^~~~~~~~ изменено

                    let real_index = index - offset + window;
                    &self.slice[real_index.0]
                }
            }

            impl Iterator for Recurrence {
                type Item = $sty;
```

```
                #[inline]
                fn next(&mut self) -> Option<$sty> {
                    if self.pos < MEM_SIZE {
//                                  ^~~~~~~ изменено
                        let next_val = self.mem[self.pos];
                        self.pos += 1;
                        Some(next_val)
                    } else {
                        let next_val = {
                            let n = self.pos;
                            let a = IndexOffset { slice: &self.mem, offset: n };
                            (a[n-1] + a[n-2])
                        };

                        {
                            use std::mem::swap;

                            let mut swap_tmp = next_val;
                            for i in (0..MEM_SIZE).rev() {
//                                      ^~~~~~~ изменено
                                swap(&mut swap_tmp, &mut self.mem[i]);
                            }
                        }

                        self.pos += 1;
                        Some(next_val)
                    }
                }
            }

            Recurrence { mem: [$($inits),+], pos: 0 }
        }
    };
}
/* ... */
```

Сделав это, мы можем заменить последнюю вещь: выражение `recur`.

```
# macro_rules! count_exprs {
#     () => (0);
#     ($head:expr $(, $tail:expr)*) => (1 + count_exprs!($($tail),*));
# }
# macro_rules! recurrence {
#     ( a[n]: $sty:ty = $($inits:expr),+ ... $recur:expr ) => {
#         {
#             const MEMORY: uint = count_exprs!($($inits),+);
#             struct Recurrence {
#                 mem: [$sty; MEMORY],
```

```
#                 pos: uint,
#             }
#             struct IndexOffset<'a> {
#                 slice: &'a [$sty; MEMORY],
#                 offset: uint,
#             }
#             impl<'a> Index<uint, $sty> for IndexOffset<'a> {
#                 #[inline(always)]
#                 fn index<'b>(&'b self, index: &uint) -> &'b $sty {
#                     let real_index = *index - self.offset + MEMORY;
#                     &self.slice[real_index]
#                 }
#             }
#             impl Iterator<u64> for Recurrence {
/* ... */
               #[inline]
               fn next(&mut self) -> Option<u64> {
                   if self.pos < MEMORY {
                       let next_val = self.mem[self.pos];
                       self.pos += 1;
                       Some(next_val)
                   } else {
                       let next_val = {
                           let n = self.pos;
                           let a = IndexOffset { slice: &self.mem, offset: n };
                           $recur
//                         ^~~~~ изменено
                       };
                       {
                           use std::mem::swap;
                           let mut swap_tmp = next_val;
                           for i in range(0, MEMORY).rev() {
                               swap(&mut swap_tmp, &mut self.mem[i]);
                           }
                       }
                       self.pos += 1;
                       Some(next_val)
                   }
               }
/* ... */
#             }
#             Recurrence { mem: [$($inits),+], pos: 0 }
#         }
#     };
# }
# fn main() {
#     let fib = recurrence![a[n]: u64 = 1, 1 ... a[n-1] + a[n-2]];
#     for e in fib.take(10) { println!("{}", e) }
# }
```

И, когда мы скомпилируем наш законченный макрос...

```
recurrence.rs:77:48: 77:49 error: unresolved name `a`
recurrence.rs:77     let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-2]];
                                                               ^
recurrence.rs:7:1: 74:2 note: in expansion of recurrence!
recurrence.rs:77:15: 77:64 note: expansion site
recurrence.rs:77:50: 77:51 error: unresolved name `n`
recurrence.rs:77     let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-2]];
                                                                 ^
recurrence.rs:7:1: 74:2 note: in expansion of recurrence!
recurrence.rs:77:15: 77:64 note: expansion site
recurrence.rs:77:57: 77:58 error: unresolved name `a`
recurrence.rs:77     let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-2]];
                                                                       ^
recurrence.rs:7:1: 74:2 note: in expansion of recurrence!
recurrence.rs:77:15: 77:64 note: expansion site
recurrence.rs:77:59: 77:60 error: unresolved name `n`
recurrence.rs:77     let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-2]];
                                                                         ^
recurrence.rs:7:1: 74:2 note: in expansion of recurrence!
recurrence.rs:77:15: 77:64 note: expansion site
```

... постойте, что? Так быть не должно... проверим, во что разворачивается макрос.

```
$ rustc -Z unstable-options --pretty expanded recurrence.rs
```

Аргумент `--pretty expanded` говорит `rustc` выполнить развертывание макроса, и затем вернуть получившееся AST обратно в исходный код. Эта опция не считается стабильной, поэтому надо указать `-Z unstable-options`. Вывод (после форматирования) показан ниже; в частности, обратите внимание на то место в коде, где `$recur` был заменен:

```
#![feature(no_std)]
#![no_std]
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
extern crate std as std;
fn main() {
    let fib = {
        use std::ops::Index;
        const MEM_SIZE: usize = 1 + 1;
        struct Recurrence {
            mem: [u64; MEM_SIZE],
            pos: usize,
        }
        struct IndexOffset<'a> {
            slice: &'a [u64; MEM_SIZE],
```

```
            offset: usize,
        }
        impl <'a> Index<usize> for IndexOffset<'a> {
            type Output = u64;
            #[inline(always)]
            fn index<'b>(&'b self, index: usize) -> &'b u64 {
                use std::num::Wrapping;
                let index = Wrapping(index);
                let offset = Wrapping(self.offset);
                let window = Wrapping(MEM_SIZE);
                let real_index = index - offset + window;
                &self.slice[real_index.0]
            }
        }
        impl Iterator for Recurrence {
            type Item = u64;
            #[inline]
            fn next(&mut self) -> Option<u64> {
                if self.pos < MEM_SIZE {
                    let next_val = self.mem[self.pos];
                    self.pos += 1;
                    Some(next_val)
                } else {
                    let next_val = {
                        let n = self.pos;
                        let a = IndexOffset{slice: &self.mem, offset: n,};
                        a[n - 1] + a[n - 2]
                    };
                    {
                        use std::mem::swap;
                        let mut swap_tmp = next_val;
                        {
                            let result =
                                match ::std::iter::IntoIterator::into_iter((0..MEM_SIZ
↳ E).rev()) {
                                    mut iter => loop {
                                        match ::std::iter::Iterator::next(&mut iter) {
                                            ::std::option::Option::Some(i) => {
                                                swap(&mut swap_tmp, &mut self.mem[i]);
                                            }
                                            ::std::option::Option::None => break,
                                        }
                                    },
                                };
                            result
                        }
                    }
                    self.pos += 1;
                    Some(next_val)
```

```
                }
            }
        }
        Recurrence{mem: [0, 1], pos: 0,}
    };
    {
        let result =
            match ::std::iter::IntoIterator::into_iter(fib.take(10)) {
                mut iter => loop {
                    match ::std::iter::Iterator::next(&mut iter) {
                        ::std::option::Option::Some(e) => {
                            ::std::io::_print(::std::fmt::Arguments::new_v1(
                                {
                                    static __STATIC_FMTSTR: &'static [&'static str] =
↳ &["", "\n"];
                                    __STATIC_FMTSTR
                                },
                                &match (&e,) {
                                    (__arg0,) => [::std::fmt::ArgumentV1::new(__arg0,
↳ ::std::fmt::Display::fmt)],
                                }
                            ))
                        }
                        ::std::option::Option::None => break,
                    }
                },
            };
        result
    }
}
```

Но все вроде в порядке! Если мы добавим несколько нехватающих атрибутов `#![feature(...)]` и запустим под ночной сборкой `rustc`, он даже компилируется! ... *как?!*

> **В сторону**: У вас не получится скомпилировать это на не-ночной сборке `rustc`. Происходит это из-за того, что развертывания макроса `println!` зависит от внутренних деталей компилятора, которые еще публично *не* стабилизированы.

Соблюдая гигиену

Проблема здесь в том, что идентификаторы в макросах Rust обладают *гигиеной*. Поэтому идентификаторы из двух разных контекстов *не могут* сталкиваться. Чтобы объяснить разницу, возьмем простой пример.

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

```
            $e
        }
    }
}

let four = using_a!(a / 10);
```

Макрос просто принимает выражение, оборачивает его в блок и определяет переменную a внутри него. Мы используем его как окольный путь вычисления 4. На самом деле здесь *два* контекста синтаксиса, но они невидимы. Поэтому, для помощи вам, дадим каждому контексту свой цвет. Начнем с неразвернутого кода, в котором только один контекст:

Теперь развернем вызов.

Как видно, a, определяемая макросом находится в другом контексте по отношению к a, которую мы подсунули в наш вызов. Поэтому компилятор считает их абсолютно разными идентификаторами, *не принимая во внимания, что у них одинаковое лексическое представление*.

Это то, с чем надо быть *особенно* осторожным при работе с макросами: макросы могут сформировать AST, которые не будут компилироваться, но, которые, если написать от руки или с использованием `--pretty expanded`, все же *компилируются*.

Решением здесь является захватить идентификатор *с подходящим контекстом синтаксиса*. Чтобы сделать это, надо снова улучшить синтаксис нашего макроса. Продолжая с нашим простым примером:

Это развернется в:

Сейчас контексты совпадают, и код компилируется. Можем аналогичным образом изменить наш `recurrence!`, явно захватывая `a` и `n`. После изменений, получим:

```
macro_rules! count_exprs {
    () => (0);
    ($head:expr) => (1);
    ($head:expr, $($tail:expr),*) => (1 + count_exprs!($($tail),*));
}

macro_rules! recurrence {
    ( $seq:ident [ $ind:ident ]: $sty:ty = $($inits:expr),+ ... $recur:expr ) => {
//    ^~~~~~~~~~   ^~~~~~~~~~ изменено
        {
            use std::ops::Index;

            const MEM_SIZE: usize = count_exprs!($($inits),+);

            struct Recurrence {
                mem: [$sty; MEM_SIZE],
                pos: usize,
            }

            struct IndexOffset<'a> {
```

```
                slice: &'a [$sty; MEM_SIZE],
                offset: usize,
            }

            impl<'a> Index<usize> for IndexOffset<'a> {
                type Output = $sty;

                #[inline(always)]
                fn index<'b>(&'b self, index: usize) -> &'b $sty {
                    use std::num::Wrapping;

                    let index = Wrapping(index);
                    let offset = Wrapping(self.offset);
                    let window = Wrapping(MEM_SIZE);

                    let real_index = index - offset + window;
                    &self.slice[real_index.0]
                }
            }

            impl Iterator for Recurrence {
                type Item = $sty;

                #[inline]
                fn next(&mut self) -> Option<$sty> {
                    if self.pos < MEM_SIZE {
                        let next_val = self.mem[self.pos];
                        self.pos += 1;
                        Some(next_val)
                    } else {
                        let next_val = {
                            let $ind = self.pos;
//                                  ^~~~ изменено
                            let $seq = IndexOffset { slice: &self.mem, offset: $ind };
//                                  ^~~~ изменено
                            $recur
                        };

                        {
                            use std::mem::swap;

                            let mut swap_tmp = next_val;
                            for i in (0..MEM_SIZE).rev() {
                                swap(&mut swap_tmp, &mut self.mem[i]);
                            }
                        }

                        self.pos += 1;
```

```
                        Some(next_val)
                    }
                }
            }

            Recurrence { mem: [$($inits),+], pos: 0 }
        }
    };
}

fn main() {
    let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-2]];

    for e in fib.take(10) { println!("{}", e) }
}
```

И он компилируется! Теперь попробуем другую последовательность.

```
↳ };
for e in recurrence!(f[i]: f64 = 1.0 ... f[i-1] * i as f64).take(10) {
    println!("{}", e)
}
```

Получим:

```
1
1
2
6
24
120
720
5040
40320
362880
```

Это победа!

4

Паттерны

Паттерны разбора и развертывания.

Обратные вызовы

```
macro_rules! call_with_larch {
    ($callback:ident) => { $callback!(larch) };
}

macro_rules! expand_to_larch {
    () => { larch };
}

macro_rules! recognise_tree {
    (larch) => { println!("#1, the Larch.") };
    (redwood) => { println!("#2, the Mighty Redwood.") };
    (fir) => { println!("#3, the Fir.") };
    (chestnut) => { println!("#4, the Horse Chestnut.") };
    (pine) => { println!("#5, the Scots Pine.") };
    ($($other:tt)*) => { println!("I don't know; some kind of birch maybe?") };
}

fn main() {
    recognise_tree!(expand_to_larch!());
    call_with_larch!(recognise_tree);
}
```

Следуя порядку, по которому макросы разворачиваются, невозможно (по крайне мере для Rust 1.2) передать информацию макросу из развертывания *другого* макроса. Это могло бы сделать

модуляризацию макросов очень сложной.

Альтернативой является использование рекурсии и передача обратного вызова. Вот как будет выглядеть пример в таком случае:

```
recognise_tree! { expand_to_larch ! (  ) }
println! { "I don't know; some kind of birch maybe?" }
// ...

call_with_larch! { recognise_tree }
recognise_tree! { larch }
println! { "#1, the Larch." }
// ...
```

Используя повторяющиеся `tt`, можно также пересылать произвольные аргументы обратному вызову.

```
macro_rules! callback {
    ($callback:ident($($args:tt)*)) => {
        $callback!($($args)*)
    };
}

fn main() {
    callback!(callback(println("Yes, this *was* unnecessary.")));
}
```

Вы можете, конечно, вставлять дополнительные токены в аргументы, по необходимости.

Последовательный потребитель TT

```
macro_rules! mixed_rules {
    () => {};
    (trace $name:ident; $($tail:tt)*) => {
        {
            println!(concat!(stringify!($name), " = {:?}"), $name);
            mixed_rules!($($tail)*);
        }
    };
    (trace $name:ident = $init:expr; $($tail:tt)*) => {
        {
            let $name = $init;
            println!(concat!(stringify!($name), " = {:?}"), $name);
            mixed_rules!($($tail)*);
        }
    };
}
```

Этот паттерн - возможно *самая мощная* доступная техника разбора макросов, позволяющая разбирать грамматику любой сложности.

“Потребитель TT” - это рекурсивный макрос, который работает, последовательно обрабатывая шаг за шагом то, что ему подали на вход. На каждом шаге он ищет совпадение и удаляет (потребляет) сочетание токенов из головы входа, создает какой-то промежуточный результат, затем вызывает сам себя с оставшейся частью входа в качестве аргумента.

Причина по которой “TT” указано в имени - необработанная часть входа *всегда* захватывается как `$($tail:tt)*`. Делается это, потому что повторение `tt` - это единственный способ *без потерь* захватить часть входа макроса.

Единственными жесткими ограничениями, которые накладываются на потребитель TT, являются те же, что и на всю систему макросов в целом:

- Вы можете использовать совпадение только с литералами или грамматическими конструкциями, которые могут захватываться `macro_rules!`.
- Вы не можете использовать совпадение с несбалансированной группой.

Важно, однако, помнить о лимите рекурсии макросов. `macro_rules!` не обладает *никакой* формой устранения или оптимизации хвостовой рекурсии. При написании потребителя TT рекомендуется предпринимать все усилия, чтобы удержаться в лимите рекурсии. Можно сделать это, добавляя дополнительные правила для учета вариантов входа (что противоположно использованию рекурсии в промежуточном слое), или, идя на компромиссы, и подгоняя вход под использования стандартных повторений.

Внутренние правила

```
#[macro_export]
macro_rules! foo {
    (@as_expr $e:expr) => {$e};

    ($($tts:tt)*) => {
        foo!(@as_expr $($tts)*)
    };
}
```

Из-за того, что макросы не подчиняются стандартным правилам приватности или подстановки, любой публичный макрос *должен* поставляться вместе со всеми макросами, от которых он зависит. Это можем привести к разрастанию пространства имен макроса, или даже к конфликтам с макросами из других контейнеров. Это может также приводить в замешательство пользователей, которые попытаются *выборочно* импортировать макросы: они должны последовательно импортировать *все* макросы, включая те, которые могут быть не задокументированны публично.

Хорошим решением является следующее - скрывать публичные макросы *внутри* экспортируемого макроса. Пример выше показывает как обычный макрос `as_expr!` может быть перемещен *внутрь* публично экспортируемого макроса, который использует его.

Причина, по которой используется `@` - начиная с Rust 1.2, токен `@` *не* используется в префиксной позиции; таким образом, он не может ни с чем конфликтовать. Другие символы или уникальные префиксы могут использоваться по желанию, но использования `@` получило широкое распространение, поэтому использование его может помочь читателям в понимании кода.

> **Помните**: токен `@` раньше использовался в префиксной позиции для обозначения указателя для сборщика мусора, а еще раньше, когда язык использовал символы для обозначения указателей. *Сейчас* его единственная цель - связать имя с паттерном. Для этого, однако, он используется как *инфиксный* оператор, что не конфликтует с его использованием здесь.

В дополнение, внутренние правила часто идут *до* любых “основных” правил, чтобы избежать проблем с тем, что `macro_rules!` может попытаться некорректно разобрать внутренний вызов как что-то, чем оно не может быть, например, выражением.

Если при экспорте по крайней мере один внутренний макрос невозможно не экспортировать (*например*, у вас много макросов, которые зависят от общего набора правил), вы можете использовать паттерн, чтобы объединить *все* внутренние макросы в один супер-макрос.

```
macro_rules! crate_name_util {
    (@as_expr $e:expr) => {$e};
    (@as_item $i:item) => {$i};
    (@count_tts) => {0usize};
    // ...
}
```

Спихиваемые накопления

```
macro_rules! init_array {
    (@accum (0, $_e:expr) -> ($($body:tt)*))
        => {init_array!(@as_expr [$($body)*])};
    (@accum (1, $e:expr) -> ($($body:tt)*))
        => {init_array!(@accum (0, $e) -> ($($body)* $e,))};
    (@accum (2, $e:expr) -> ($($body:tt)*))
        => {init_array!(@accum (1, $e) -> ($($body)* $e,))};
    (@accum (3, $e:expr) -> ($($body:tt)*))
        => {init_array!(@accum (2, $e) -> ($($body)* $e,))};
    (@as_expr $e:expr) => {$e};
    [$e:expr; $n:tt] => {
        {
            let e = $e;
            init_array!(@accum ($n, e.clone()) -> ())
        }
    };
}

let strings: [String; 3] = init_array![String::from("hi!"); 3];
```

Все макросы в Rust **должны** в результате разворачиваться в законченный, подходящий по синтаксису элемент (такой как выражение, элемент, *и т.д.*). Это означает, что нельзя сделать макрос, разворачивающийся в частичную конструкцию.

Можно надеяться, что приведенный выше пример выразится так:

```
macro_rules! init_array {
    (@accum 0, $_e:expr) => {/* empty */};
    (@accum 1, $e:expr) => {$e};
    (@accum 2, $e:expr) => {$e, init_array!(@accum 1, $e)};
    (@accum 3, $e:expr) => {$e, init_array!(@accum 2, $e)};
    [$e:expr; $n:tt] => {
        {
            let e = $e;
            [init_array!(@accum $n, e)]
        }
    };
}
```

Ожидаем, что развертывание литералов из массива выполнится следующим образом:

```
            [init_array!(@accum 3, e)]
            [e, init_array!(@accum 2, e)]
            [e, e, init_array!(@accum 1, e)]
            [e, e, e]
```

Однако, для это потребуется, чтобы каждый промежуточный шаг разворачивался в незаконченное выражение. Даже при том, что промежуточные результаты никогда не будут использоваться *снаружи* контекста макроса, это все равно запрещено.

Спихивание, однако, позволяет нам последовательно строить связку токенов, без необходимости иметь законченную структуру на каждом шаге до окончания. В примере на самом верху, связка вызовов макроса выполнится так:

```
init_array! { String:: from ( "hi!" ) ; 3 }
init_array! { @ accum ( 3 , e . clone (  ) ) -> (  ) }
init_array! { @ accum ( 2 , e.clone() ) -> ( e.clone() , ) }
init_array! { @ accum ( 1 , e.clone() ) -> ( e.clone() , e.clone() , ) }
init_array! { @ accum ( 0 , e.clone() ) -> ( e.clone() , e.clone() , e.clone() , ) }
init_array! { @ as_expr [ e.clone() , e.clone() , e.clone() , ] }
```

Как вы видите, каждый слой добавляет накопление к выходу до тех пор пока последнее правило не выделяет все это в законченную структуру.

Единственным плохим моментов в приведенной редакции является использование `$($body:tt)*` для того, чтобы сохранить значение на выходе, не запуская его разбор. `($input) -> ($output)` - это просто соглашение, призванное упростить понимание таких макросов.

Спихиваемые накопления - часто используемая часть последовательного потребителяTT, так как она позволяет конструировать промежуточные результаты произвольной сложности.

Замена на повторение

```
macro_rules! replace_expr {
    ($_t:tt $sub:expr) => {$sub};
}
```

Этот паттерн представляет случай, когда совпавшая последовательность отбрасывается, а переменная, представлявшая ее, заменяется на какой-то повторяющийся паттерн, который подходит по длине.

Например, представим создание экземпляра кортежа, в котором не больше 12 элементов (ограничение для Rust 1.2), и каждому присвоено значение по умолчанию.

```
macro_rules! tuple_default {
    ($($tup_tys:ty),*) => {
        (
            $(
                replace_expr!(
                    ($tup_tys)
                    Default::default()
                ),
            )*
        )
    };
}
```

ТДЭП: мы *могли бы* просто использовать `$tup_tys::default()`.

Здесь, мы на самом деле не *используем* совпадение по типам. Вместо этого, мы выбрасываем их и заменяем одним, повторяющимся выражением. Говоря по-другому, нам все равно, *какие* типы используются, нам важно только *сколько* их.

Разделители

```
macro_rules! match_exprs {
    ($($exprs:expr),* $(,)*) => {...};
}
```

Есть много мест в грамматике Rust, где нельзя использовать запятую в качестве разделителя. Два основных способа найти совпадение (например) со списком выражений (`$($exprs:expr),*` и `$($exprs:expr,)*`) могут работать как *без*, *так и* с разделяющей запятой, но *не оба случая одновременно*.

Поставив повторение `$(,)*` *после* основного списка, можно захватить любое количество (включая ноль или одну) разделяющих запятых, или других разделителей, которые вы используете.

Помните только, что так получится сделать не всегда. Если компилятор отвергнет это, вам, вероятно, следует использовать несколько сопоставлений и/или инкрементальное сопоставление.

Связывание TT

```
macro_rules! call_a_or_b_on_tail {
    ((a: $a:expr, b: $b:expr), call a: $($tail:tt)*) => {
        $a(stringify!($($tail)*))
    };

    ((a: $a:expr, b: $b:expr), call b: $($tail:tt)*) => {
        $b(stringify!($($tail)*))
    };

    ($ab:tt, $_skip:tt $($tail:tt)*) => {
        call_a_or_b_on_tail!($ab, $($tail)*)
    };
}

fn compute_len(s: &str) -> Option<usize> {
    Some(s.len())
}

fn show_tail(s: &str) -> Option<usize> {
    println!("tail: {:?}", s);
    None
}

fn main() {
    assert_eq!(
        call_a_or_b_on_tail!(
            (a: compute_len, b: show_tail),
            the recursive part that skips over all these
            tokens doesn't much care whether we will call a
            or call b: only the terminal rules care.
        ),
        None
    );
    assert_eq!(
        call_a_or_b_on_tail!(
            (a: compute_len, b: show_tail),
            and now, to justify the existence of two paths
            we will also call a: its input should somehow
            be self-referential, so let's make it return
            some ninety one!
        ),
        Some(91)
    );
}
```

В особенно сложных рекурсивных макросах может понадобится большое число аргументов

для того, чтобы передать идентификаторы и выражения последующим слоям. Однако, в зависимости от реализации, может присутствовать много промежуточных слоев, которые должны пробрасывать эти аргументы, не *используя* их.

Таким образом, может быть очень полезно связать такие аргументы вместе в одно TT и положить его в группу. Это позволит слоям, не использующим их, просто захватывать и заменять одно `tt`, вместо того, чтобы захватывать и заменять каждый аргумент из группы.

Пример выше связывает выражения `$a` и `$b` в группу, которая дальше может передаваться как одно `tt` по рекурсивным правилам. Эта группа потом разбирается терминальными правилами для доступа к каждому выражению в отдельности.

Видимость

Сопоставление и замена видимости может быть достаточна хитра в Rust, из-за отсутствия типа `vis` в сопоставлении.

Сопоставление и игнорирование

В зависимости от контекста, это можно сделать повторениями:

```
macro_rules! struct_name {
    ($(pub)* struct $name:ident $($rest:tt)*) => { stringify!($name) };
}
```

Пример сопоставит элементы `struct`, которые и приватны и публичны. Или `pub pub` (очень публичны), или даже `pub pub pub pub` (очень очень публичны). Лучшей защитой от этого является надежда. Надежда на то, что люди не будут использовать его так глупо.

Сопоставление и замена

Из-за того, что нельзя связать повторение само по себе с переменной, нет и возможности захватить `$(pub)*` так, чтоб его можно было заменить. В результате нужно несколько правил.

```
macro_rules! newtype_new {
    (struct $name:ident($t:ty);) => { newtype_new! { () struct $name($t); } };
    (pub struct $name:ident($t:ty);) => { newtype_new! { (pub) struct $name($t); } };

    (($($vis:tt)*) struct $name:ident($t:ty);) => {
        as_item! {
            impl $name {
                $($vis)* fn new(value: $t) -> Self {
                    $name(value)
                }
            }
        }
    };
```

```
}

macro_rules! as_item { ($i:item) => {$i} }
```

> **Смотри также**: AST Преобразования.

В этом случае мы используем возможности сопоставления произвольной последовательности токенов внутри группы с `()` или `(pub)`, затем заменяем содержимое на выходе. Из-за того, что парсер не ожидает на этой позиции развертывания повторения `tt`, нам придется использовать AST преобразования, чтобы развертывание правильно разбиралось.

Предварительно

Этот раздел предназначен для паттернов или техник, которые имеют сомнительную ценность, или которые могут быть *слишком* хороши для включения в основной список.

Счеты

> **Предварительно**: нужны более привлекательные примеры. Несмотря на то что важная часть макроса `0ok!`, относящаяся к поиску совпадений с вложенными группами, *не* входящим в группы Rust, очень необычна, она может не заслуживать включения в этот раздел.

> **Заметьте**: в этой секции подразумевается понимание спихиваемых накоплений и [последовательных потребителей TT (#incremental-tt-munchers).

```
macro_rules! abacus {
    ((- $($moves:tt)*) -> (+ $($count:tt)*)) => {
        abacus!(($($moves)*) -> ($($count)*))
    };
    ((- $($moves:tt)*) -> ($($count:tt)*)) => {
        abacus!(($($moves)*) -> (- $($count)*))
    };
    ((+ $($moves:tt)*) -> (- $($count:tt)*)) => {
        abacus!(($($moves)*) -> ($($count)*))
    };
    ((+ $($moves:tt)*) -> ($($count:tt)*)) => {
        abacus!(($($moves)*) -> (+ $($count)*))
    };

    // Проверка получившегося результата на ноль.
    (() -> ()) => { true };
    (() -> ($($count:tt)+)) => { false };
}
```

```
fn main() {
    let equals_zero = abacus!((++-+-+++--++---++----+) -> ());
    assert_eq!(equals_zero, true);
}
```

Эта техника может использоваться в тех случаях, когда вам нужно отслеживать изменения счетчика, начинающегося с чего-то близкого к нулю, и нужно поддерживать следующие операции:

- Инкремент на единицу.
- Декремент на единицу.
- Сравнение с нулем (или другим фиксированным, конечным числом).

Значение *n* обозначает *n* экземпляров конкретного токена, входящего в группу. Изменение выполняется рекурсией и спихиваемых накоплений. Определяя используемый токен за `x`, операции выше реализуются по-следующему:

- Инкремент на единицу: совпадение с `($($count:tt)*)` заменяется на `(x $($count)*)`.
- Декремент на единицу: совпадение с `(x $($count:tt)*)` заменяется на `($($count)*)`.
- Сравнение с нулем: совпадение с `()`.
- Сравнение с единицей: совпадение с `(x)`.
- Сравнение с двойкой: совпадение с `(x x)`.
- *(и так далее...)*

Таким образом, операции с счетчиком похожи на щелканье токенов туда и обратно, как на счётах. [1]

По правде говоря, эти операции *также* можно было назвать “унарный подсчет”[3].

Если вам нужны отрицательные значения, то -*n* можно заменить на *n* экземпляров *другого* токена. В примере выше, +*n* представлено *n* + токенами, а -*m* - *m* - токенами.

В данном случае операции немного осложняются; инкремент и декремент эффективно меняют свое значение на противоположное, если счетчик отрицательный. Считая, что + и - представляют положительный и отрицательный токены соответственно, операции меняются следующим образом:

- Инкремент на единицу:
- совпадение с `()`, заменяется на `(+)`.
- совпадение с `(- $($count:tt)*)`, заменяется на `($($count)*)`.
- совпадение с `($($count:tt)+)`, заменяется на `(+ $($count)+)`.
- Декремент на единицу:
- совпадение с `()`, заменяется на `(-)`.
- совпадение с `(+ $($count:tt)*)`, заменяется на `($($count)*)`.
- совпадение с `($($count:tt)+)`, заменяется на `(- $($count)+)`.

[1] Это крайне тонкое сравнение на самом деле скрывает *настоящую* причину такого названия: избежать *еще одной* штуки со словом “токен” в названии. Поговори со своим писателем о семантическом насыщении[2] сегодня!

[3] https://en.wikipedia.org/wiki/Unary_numeral_system

- Сравнение с 0: совпадение с `()`.
- Сравнение с +1: совпадение с `(+)`.
- Сравнение с -1: совпадение с `(-)`.
- Сравнение с +2: совпадение с `(++)`.
- Сравнение с -2: совпадение с `(--)`.
- *(и так далее...)*

Заметьте, что пример выше объединяет некоторые из этих правил вместе (например, он объединяет инкремент `()` и `($($count:tt)+)` в инкремент `($($count:tt)*)`).

Если вы хотите достать текущее *значение* счетчика, можно использовать обычный подсчет[4]. Для примера выше терминальные правила можно заменить следующими:

```
macro_rules! abacus {
    // ...

    // This extracts the counter as an integer expression.
    (() -> ()) => {0};
    (() -> (- $($count:tt)*)) => {
        {(-1i32) $(- replace_expr!($count 1i32))*}
    };
    (() -> (+ $($count:tt)*)) => {
        {(1i32) $(+ replace_expr!($count 1i32))*}
    };
}

macro_rules! replace_expr {
    ($_t:tt $sub:expr) => {$sub};
}
```

> **ТДЭП**: грубо говоря, приведенная редакция `abacus!` невообразимо сложна. Ее можно заменить на гораздо более эффективную, использующую повторения, только в том случае, если вам *не* надо нигде сопоставлять паттерн со значением счетчика в макросе:
>
> ```
> macro_rules! abacus {
>     (-) => {-1};
>     (+) => {1};
>     ($($moves:tt)*) => {
>         0 $(+ abacus!($moves))*
>     }
> }
> ```

[4] ../blk/README.html#counting

5

Строительные блоки

Многоразовые сниппеты кода макросов.

AST Преобразования

Парсер Rust не очень силен в подстановках `tt`. Проблемы могут возникнуть, когда парсер ожидает особенную грамматическую конструкцию, а *вместо этого* находит кучу подставленных `tt`. Вместо того, чтобы попытаться их распарсить, он очень часто просто *сдается*. В этих случаях необходимо применить AST преобразование.

```
macro_rules! as_expr { ($e:expr) => {$e} }
macro_rules! as_item { ($i:item) => {$i} }
macro_rules! as_pat  { ($p:pat) =>  {$p} }
macro_rules! as_stmt { ($s:stmt) => {$s} }
```

Эти преобразования часто используются с [push-down accumulation] макросами для того, чтобы парсер обработал последнее выражение `tt`, как особый вид грамматической конструкции.

Помните, что этот особый набор макросов определяется тем, во что они могут разворачиваться, а *не* тем, какие метапеременные они могут захватывать. То есть, потому что макрос не может появится на позиции типа [1], вы не можете иметь `as_ty!` макрос.

Подсчет

Повторение с заменой

Подсчет чего-либо в макросе - удивительно хитрая задача. Самое простое решение - использовать замену и повторяющееся сопоставление с образцом.

[1] Смотри Issue #27245[2].

```
macro_rules! replace_expr {
    ($_t:tt $sub:expr) => {$sub};
}

macro_rules! count_tts {
    ($($tts:tt)*) => {0usize $(+ replace_expr!($tts 1usize))*};
}
```

Это прекрасно подходит для небольшого количества переменных, но скорее всего *сломает компилятор*, когда на входе их будет около 500 или больше. Представьте, что вывод будет выглядеть как-то так:

```
0usize + 1usize + /* ~500 `+ 1usize`s */ + 1usize
```

Компилятор должен распарсить это в AST, который произведет, фактически, абсолютно несбалансированное бинарное дерево с 500+ уровнями глубины.

Рекурсия

Старый подход - использовать рекурсию.

```
macro_rules! count_tts {
    () => {0usize};
    ($_head:tt $($tail:tt)*) => {1usize + count_tts!($($tail)*)};
}
```

> **Заметьте**: По состоянию на `rustc` 1.2, у компилятора появляются *серьезные* проблемы с производительностью, если большое число целочисленных переменных неизвестного типа должны пройти через выведение. Мы используем здесь явно переменные типа `usize` для избежания этого.
>
> Если такой вариант не подходит (например, когда тип должен подставляться), вы можете помочь компилятору, используя `as` (*например* `0 as $ty`, `1 as $ty`, *и т.д.*).

Этот подход *работает*, но довольно быстро достигнется лимит рекурсии. В отличие от подхода с повторениями, вы можете расширить количество переменных на входе сопоставлением множества токенов за один раз.

```
macro_rules! count_tts {
    ($_a:tt $_b:tt $_c:tt $_d:tt $_e:tt
     $_f:tt $_g:tt $_h:tt $_i:tt $_j:tt
     $_k:tt $_l:tt $_m:tt $_n:tt $_o:tt
     $_p:tt $_q:tt $_r:tt $_s:tt $_t:tt
     $($tail:tt)*)
        => {20usize + count_tts!($($tail)*)};
    ($_a:tt $_b:tt $_c:tt $_d:tt $_e:tt
     $_f:tt $_g:tt $_h:tt $_i:tt $_j:tt
```

```
     $($tail:tt)*)
        => {10usize + count_tts!($($tail)*)};
    ($_a:tt $_b:tt $_c:tt $_d:tt $_e:tt
     $($tail:tt)*)
        => {5usize + count_tts!($($tail)*)};
    ($_a:tt
     $($tail:tt)*)
        => {1usize + count_tts!($($tail)*)};
    () => {0usize};
}

fn main() {
    assert_eq!(700, count_tts!(
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        // Повторение ломается где-то после этого места
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
    ));
}
```

Такая формулировка будет работать вплоть до ~1,200 значений.

Длина среза

Третий подход [к решению проблемы] — помочь компилятору сконструировать неглубокий AST, который не приведет к переполнению стека. Это можно сделать, сконструировав переменную массива и вызвав метод `len`.

```
macro_rules! replace_expr {
    ($_t:tt $sub:expr) => {$sub};
}
```

```
macro_rules! count_tts {
    ($($tts:tt)*) => {<[()]>::len(&[$(replace_expr!($tts ())),*])};
}
```

Этот подход протестирован на коде длиной до 10000 токенов и, возможно, будет работать для гораздо более длинного кода. *Обратной стороной* является то, что в Rust 1.2 он *не может* быть использован для представления константного выражения. Хотя результат может быть оптимизирован до простой константы (при билде в режиме дебага он компилируется вниз для загрузки из памяти), он все еще не может использоваться в качестве константы (значение `const` или точный размер массива).

Однако, если необходим неконстантный подсчет, это один из самых предпочтительных методов.

Подсчет через Enum

Этот подход может использоваться, если вам нужно посчитать набор взаимно различных идентификаторов. Необходимо добавить, что результат этого подхода используется как константа.

```
macro_rules! count_idents {
    ($($idents:ident),* $(,)*) => {
        {
            #[allow(dead_code, non_camel_case_types)]
            enum Idents { $($idents,)* __CountIdentsLast }
            const COUNT: u32 = Idents::__CountIdentsLast as u32;
            COUNT
        }
    };
}
```

У этого метода есть два недостатка. Первый, как приведено выше, - он может считать *только* валидные идентификаторы (которые, к тому же, не являются ключевыми словами), и он не позволяет идентификаторам повторяться.

Второй недостаток - этот подход *не совсем* чист, имея ввиду, что если вместо одного из идентификаторов на входе вы используете `__CountIdentsLast`, макрос сломается из-за дублирования вариантов в `enum`.

Разбор Enum

```
macro_rules! parse_unitary_variants {
    (@as_expr $e:expr) => {$e};
    (@as_item $($i:item)+) => {$($i)+};

    // Правила выхода.
```

```
    (
        @collect_unitary_variants ($callback:ident ( $($args:tt)* )),
        ($(,)*) -> ($($var_names:ident,)*)
    ) => {
        parse_unitary_variants! {
            @as_expr
            $callback!{ $($args)* ($($var_names),*) }
        }
    };

    (
        @collect_unitary_variants ($callback:ident { $($args:tt)* }),
        ($(,)*) -> ($($var_names:ident,)*)
    ) => {
        parse_unitary_variants! {
            @as_item
            $callback!{ $($args)* ($($var_names),*) }
        }
    };

    // Поглощение атрибута.
    (
        @collect_unitary_variants $fixed:tt,
        (#[$_attr:meta] $($tail:tt)*) -> ($($var_names:tt)*)
    ) => {
        parse_unitary_variants! {
            @collect_unitary_variants $fixed,
            ($($tail)*) -> ($($var_names)*)
        }
    };

    // Обработка варианта, дополнительно: с инициализацией
    (
        @collect_unitary_variants $fixed:tt,
        ($var:ident $(= $_val:expr)*, $($tail:tt)*) -> ($($var_names:tt)*)
    ) => {
        parse_unitary_variants! {
            @collect_unitary_variants $fixed,
            ($($tail)*) -> ($($var_names)* $var,)
        }
    };

    // Отмена варианта с полезной нагрузкой (payload)
    (
        @collect_unitary_variants $fixed:tt,
        ($var:ident $_struct:tt, $($tail:tt)*) -> ($($var_names:tt)*)
    ) => {
        const _error: () = "cannot parse unitary variants from enum with non-unitary v
```

```
↳ ariants";
    };

    // Правило входа.
    (enum $name:ident {$($body:tt)*} => $callback:ident $arg:tt) => {
        parse_unitary_variants! {
            @collect_unitary_variants
            ($callback $arg), ($($body)*,) -> ()
        }
    };
}
```

Этот макрос показывает, как вы можете использовать [incremental tt muncher] и [push-down accumulation] для парсинга вариантов `enum` , в котором все варианты унитарны (*m.e.* не имеют полезной нагрузки (payload)). После завершения, `parse_unitary_variants!` вызывает макрос [callback] со списком вариантов (плюс поддерживает любые другие произвольные аргументы).

Его можно изменить, чтобы также парсить поля `struct` , вычислять дополнительные значения для вариантов, или даже выделять имена *всех* вариантов в произвольный `enum`.

6

Примеры с аннотацией

Этот раздел содержит макросы из реальной жизни[1], снабженные примечаниями, в которых описаны их дизайн и конструкция.

Ook!

Этот макрос представляет собой реализацию эзотерического языка Ook![2], которая изоморфна эзотерическому языку Brainfuck[3].

Модель выполнения языка очень проста: память представляет собой массив “ячеек” (обычно, как минимум, 8 бит) переменной длины (обычно, как минимум, 30.000 “ячеек”). В памяти есть указатель, который начинается с 0. Наконец, есть стек выполнения (используемый для реализации циклов) и указатель на инструкцию (команду) в программе, хотя эти два последних не принадлежат выполняющейся программе; они являются свойствами среды выполнения, как таковой.

Язык состоит всего из трех служебных слов: `Ook.`, `Ook?` и `Ook!`. Они объединяются в пары, формируя 8 различных операций:

- `Ook. Ook?` - инкрементировать указатель.
- `Ook? Ook.` - декрементировать указатель.
- `Ook. Ook.` - инкрементировать значение в ячейке памяти по указателю.
- `Ook! Ook!` - декрементировать значение в ячейке памяти по указателю.
- `Ook! Ook.` - переписать значение из ячейки памяти по указателю в стандартный output.
- `Ook. Ook!` - переписать значение из стандартного input в ячейку памяти по указателю.
- `Ook! Ook?` - начать цикл.
- `Ook? Ook!` - прыгнуть в начало цикла, если значение в ячейке памяти по указателю не равно 0; иначе, продолжить.

[1] По большей части.

[2] http://www.dangermouse.net/esoteric/ook.html

[3] http://www.muppetlabs.com/~breadbox/bf/

Ook! интересен тем, что он является тьюринг-полным, что означает, что среда, в которой вы можете реализовать его, должна быть *тоже* тьюринг-полной.

Реализация

```
#![recursion_limit = "158"]
```

Это, на самом деле, минимиальный лимит рекурсии, для которого скомпилируется пример программы, приведенный в конце. Если вы хотите узнать, что может быть таким фантастически сложным, что сможет *оправдать* изменение лимита рекурсии в пять раз от дефолтного значения ... угадайте[4].

```
type CellType = u8;
const MEM_SIZE: usize = 30_000;
```

Это здесь, чтобы убедиться, что они видимы для расширения макроса.[5]

```
macro_rules! Ook {
```

Имя *по-хорошему* должно было быть `ook!` по правилам именования, но случай уж был слишком удобен, чтобы пройти мимо них.

Правила для этого макроса разбиты на секции используя паттерн internal rules[6].

Первым будет правило `@start`, обрабатывающее настройки блока, в котором будет происходить остальной разворот. Здесь нет ничего интересного: мы определяем некоторые переменные и вспомогательные функции, и затем выполняем основную часть разворота.

Несколько небольших замечаний:

- Мы разворачиваем в функцию, поэтому можем использовать `try!` для упрощения перехвата ошибок.
- Использование имен с подчеркиванием вначале нужно, чтобы компилятор не ругался на неиспользованные функции и переменные если, например, пользователь пишет Ook! для программы, у которой нет I/O.

```
    (@start $($Ooks:tt)*) => {
        {
            fn ook() -> ::std::io::Result<Vec<CellType>> {
                use ::std::io;
                use ::std::io::prelude::*;

                fn _re() -> io::Error {
```

[4] https://en.wikipedia.org/wiki/Hello_world_program

[5] Эти поля *могли бы* быть определены внутри макроса, но тогда они бы явно передавались в каждом вызове (из-за чистоты макроса). Чтобы быть честным, к тому временя как я понял, что мне *нужно* их определить, макрос был уже почти написан и ... ну, *вы* бы хотели заново пройти весь путь и исправить это, если на самому деле *огромной нужды* в этом нет?

[6] ../pat/README.html#internal-rules

```
                    io::Error::new(
                        io::ErrorKind::Other,
                        String::from("ran out of input"))
                }

                fn _inc(a: &mut [u8], i: usize) {
                    let c = &mut a[i];
                    *c = c.wrapping_add(1);
                }

                fn _dec(a: &mut [u8], i: usize) {
                    let c = &mut a[i];
                    *c = c.wrapping_sub(1);
                }

                let _r = &mut io::stdin();
                let _w = &mut io::stdout();

                let mut _a: Vec<CellType> = Vec::with_capacity(MEM_SIZE);
                _a.extend(::std::iter::repeat(0).take(MEM_SIZE));
                let mut _i = 0;
                {
                    let _a = &mut *_a;
                    Ook!(@e (_a, _i, _inc, _dec, _r, _w, _re); ($($Ooks)*));
                }
                Ok(_a)
            }
            ook()
        }
    };
```

Парсинг кода операции

Далее следуют правила “выполнения”, которые используется для парсинга кода операции на входе.

Обычная форма этих правил - `(@e $syms; ($input))`. Как можно заметить из правила `@start`, `$syms` - это коллекция символов, необходимая для того, чтобы реализовать программу: вход, выход, массив памяти, *и т.д.*. Мы используем TT bundling[7] для простой передачи этих символов дальше, промежуточным правилам.

Первое, это правило прерывает нашу рекурсию: если на входе у нас ничего нет - мы останавливаемся.

```
    (@e $syms:tt; ()) => {};
```

Следующее, у нас есть одно правило для *почти* каждого кода операции. Для этого мы берем код операции, подставляем соответствующй код на Rust, затем рекурсивно перемещаемся в

[7] ../pat/README.html#tt-bundling

хвост входа: a textbook TT muncher[8].

```
    // Increment pointer.
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook. Ook? $($tail:tt)*))
    => {
        $i = ($i + 1) % MEM_SIZE;
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };

    // Decrement pointer.
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook? Ook. $($tail:tt)*))
    => {
        $i = if $i == 0 { MEM_SIZE } else { $i } - 1;
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };

    // Increment pointee.
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook. Ook. $($tail:tt)*))
    => {
        $inc($a, $i);
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };

    // Decrement pointee.
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook! Ook! $($tail:tt)*))
    => {
        $dec($a, $i);
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };

    // Write to stdout.
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook! Ook. $($tail:tt)*))
    => {
        try!($w.write_all(&$a[$i .. $i+1]));
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };

    // Read from stdin.
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook. Ook! $($tail:tt)*))
    => {
        try!(
            match $r.read(&mut $a[$i .. $i+1]) {
```

[8] ../pat/README.html#incremental-tt-munchers

```
                Ok(0) => Err($re()),
                ok @ Ok(..) => ok,
                err @ Err(..) => err
            }
        );
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };
```

Вот здесь вещи становятся более запутанными. Этот код, `Ook! Ook?`, означает начало цикла. Циклы Ook! переводятся в следующий код на Rust:

> **Замечание**: это *не* рабочая часть кода.
>
> ```
> while memory[ptr] != 0 {
>     // Содержимое цикла
> }
> ```

Конечно, мы не можем *на самом деле* выделить незаконченный цикл. Это *можно* решить, используя pushdown[9], что ведет к более фундаментальной проблеме: мы не можем *написать* `while memory[ptr] != {`, в принципе, *где бы то ни было*. Это происходит потому, что в противном случае у нас появится лишняя, “висячая”, скобка.

Для того, чтобы решить это, мы разделяем вход на две части: все *внутри* цикла, и все, что *после* него. Правила `@x` берут на себя первое, `@s` - второе.

```
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook! Ook? $($tail:tt)*))
    => {
        while $a[$i] != 0 {
            Ook!(@x ($a, $i, $inc, $dec, $r, $w, $re); (); (); ($($tail)*));
        }
        Ook!(@s ($a, $i, $inc, $dec, $r, $w, $re); (); ($($tail)*));
    };
```

Извлечение цикла

Следующее - `@x`, или правило “извлечения”. Оно отвечает за следующее - взять хвост со входа и извлечь содержимое из цикла. Общая форма этого правила: `(@x $sym; $depth; $buf; $tail)`.

Назначение `$sym` такое же, как и выше. `$tail` - это вход, который нужно распарсить, в то время как `$buf` - это push-down accumulation buffer[10], в который мы будем собирать коды операций, находящиеся внутри цикла. Но что на счет `$depth`?

Сложности добавляет то, что циклы могут быть *вложенными*. Итак, мы должны как-то отслеживать, сколько уровней вложенности у нас в данный момент. Мы должны делать это очень аккуратно, чтобы не остановить парсинг слишком рано или слишком поздно, а остановить именно тогда *когда нужно*.[11]

[9] ../pat/README.html#push-down-accumulation

[10] ../pat/README.html#push-down-accumulation

[11] Это известный факт[ˆaeg-ook.md--fact], что сказка “Маша и трих медведя” - это на самом деле аллегория на технику аккуратного лексического парсинга.

Из-за того, что мы не можем выполнять арифметические действия в макросах, а написание собственных правил явного совпадения с целым числом является неосуществимым (представьте очень много копи-паста таких правил для всех положительных целых чисел), мы вместо этого вернемся к самому древнему и самому почтенному методу счета в истории: счету на пальцах.

Но из-за того, что у макроса *нет* пальцев, мы используем token abacus counter[12] вместо них. Для специфики, мы будем использовать много@, где каждое `@` представляет собой один дополнительный уровень. Если мы объеденим эти `@` в группу, мы сможем реализовать три операции, которые нам нужны:

- Инкремент: `($($depth:tt)*)` заменяем на `(@ $($depth)*)`.
- Декремент: `(@ $($depth:tt)*)` заменяем на `($($depth)*)`.
- Сравнение с нулем: сравнение с `()`.

Первое - это правило, необходимое, чтобы найти совпадение с выражением `Ook? Ook!`, заканчивающее цикл, который мы парсим. В этом случае мы скармливаем накопленное содержание цикла правилам `@e`, определенным до этого.

Заметьте, что нам *не нужно* делать что-либо с оставшимся хвостом на входе (он будет обрабатываться правилами `@s`).

```
(@x $syms:tt; (); ($($buf:tt)*);
    (Ook? Ook! $($tail:tt)*))
=> {
    // Внешний цикл заканчивается. Обрабатываем значения из буфера.
    Ook!(@e $syms; ($($buf)*));
};
```

Следующее, у нас есть правила для входа и выхода из вложенных циклов. Это увеличивает счетчик и добавляет коды операций в буфер.

```
(@x $syms:tt; ($($depth:tt)*); ($($buf:tt)*);
    (Ook! Ook? $($tail:tt)*))
=> {
    // На один уровень глубже.
    Ook!(@x $syms; (@ $($depth)*); ($($buf)* Ook! Ook?); ($($tail)*));
};

(@x $syms:tt; (@ $($depth:tt)*); ($($buf:tt)*);
    (Ook? Ook! $($tail:tt)*))
=> {
    // На один уровень выше.
    Ook!(@x $syms; ($($depth)*); ($($buf)* Ook? Ook!); ($($tail)*));
};
```

Наконец, у нас есть правило для “всего остального”. Обратите внимание на метапеременные `$op0` и `$op1`: как подразумевается в Rust, наш токен Ook! - это всегда *два* токена в Rust: идентификатор `Ook` и другой токен. Таким образом, мы можем найти все нецикличные коды операций по совпадению с `!`, `?`, и `.` вместо `tt`.

[12] ../pat/README.html#abacus-counters

Здесь мы оставляем `$depth` нетронутым и просто добавляем коды операций в буфер.

```
(@x $syms:tt; $depth:tt; ($($buf:tt)*);
    (Ook $op0:tt Ook $op1:tt $($tail:tt)*))
=> {
    Ook!(@x $syms; $depth; ($($buf)* Ook $op0 Ook $op1); ($($tail)*));
};
```

Пропуск цикла

Это *в целом* то же самое, что и извлечение цикла, кроме того, что мы не заботимся о *содержании* цикла (и, в связи с этим, нам не нужен буфер накопления). Все, что нам нужно - это знать когда мы *прошли* мимо цикла. В этом случае, мы продолжаем обрабатывать вход, используя правила `@e`.

В связи с вышесказанным, эти правила представлены без дальнейших объяснений.

```
// Конец цикла
(@s $syms:tt; ();
    (Ook? Ook! $($tail:tt)*))
=> {
    Ook!(@e $syms; ($($tail)*));
};

// Вход во вложенный цикл.
(@s $syms:tt; ($($depth:tt)*);
    (Ook! Ook? $($tail:tt)*))
=> {
    Ook!(@s $syms; (@ $($depth)*); ($($tail)*));
};

// Выход из вложенного цикла.
(@s $syms:tt; (@ $($depth:tt)*);
    (Ook? Ook! $($tail:tt)*))
=> {
    Ook!(@s $syms; ($($depth)*); ($($tail)*));
};

// Не связанный с циклами код операции.
(@s $syms:tt; ($($depth:tt)*);
    (Ook $op0:tt Ook $op1:tt $($tail:tt)*))
=> {
    Ook!(@s $syms; ($($depth)*); ($($tail)*));
};
```

Точка входа

Это единственное не-внутреннее правило.

Стоит отметить, что этот шаблон просто ищет совпадения по *всем* токенам, переданным ему, и поэтому он *чрезвычайно опасен*. Любая ошибка может привести к неправильному распознаванию всех правил выше, что может в свою очередь привести к бесконечной рекурсии.

Когда вы пишете, изменяете, или занимаетесь отладкой макроса как этот, будет уместно временно подставить в начало что-либо, навроде `@entry`. Это позволит избежать случая бесконечной рекурсии, и, с большей вероятностью, приведет к ошибкам распознавания в правильных местах.

```
    ($($Ooks:tt)*) => {
        Ook!(@start $($Ooks)*)
    };
}
```

Использование

Вот, наконец, и наша тестовая программа.

```
fn main() {
    let _ = Ook!(
        Ook. Ook?  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook! Ook?  Ook? Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook?  Ook! Ook!  Ook? Ook!  Ook? Ook.
        Ook! Ook.  Ook. Ook?  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook! Ook?  Ook? Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook?
        Ook! Ook!  Ook? Ook!  Ook? Ook.  Ook. Ook.
        Ook! Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook! Ook.  Ook! Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook! Ook.  Ook. Ook?  Ook. Ook?
        Ook. Ook?  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook! Ook?  Ook? Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook?
        Ook! Ook!  Ook? Ook!  Ook? Ook.  Ook! Ook.
        Ook. Ook?  Ook. Ook?  Ook. Ook?  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook! Ook?  Ook? Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook?  Ook! Ook!  Ook? Ook!  Ook? Ook.
        Ook! Ook!  Ook! Ook!  Ook! Ook!  Ook! Ook.
        Ook? Ook.  Ook? Ook.  Ook? Ook.  Ook? Ook.
```

```
        Ook! Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook! Ook.  Ook! Ook!  Ook! Ook!  Ook! Ook!
        Ook! Ook!  Ook! Ook!  Ook! Ook!  Ook! Ook.
        Ook! Ook!  Ook! Ook!  Ook! Ook!  Ook! Ook!
        Ook! Ook!  Ook! Ook!  Ook! Ook!  Ook! Ook!
        Ook! Ook.  Ook. Ook?  Ook. Ook?  Ook. Ook.
        Ook! Ook.  Ook! Ook?  Ook! Ook!  Ook? Ook!
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook! Ook.
    );
}
```

На выходе после запуска (после продолжительной паузы, во время которой компилятор выполнит сотни рекурсивных разворотов макроса) получается следующее:

```
Hello World!
```

С этим, мы показали ужасающую правду, что `macro_rules!` является тьюринг-полным!

В дополнение

Представленное выше основано на макро-реализации изоморфного языка “Hodor!”. Manish Goregaokar затем реализовал интерпретатор Brainfuck, используя макрос Hodor![13]. Таким образом, это интерпретатор Brainfuck, написанный на Hodor!, который, в свою очередь, написан на `macro_rules!`.

Легенда гласит, что после увеличения лимита рекурсии к *трем миллионам* и запуска на *четыре дня*, он наконец выполнился.

…с переполнением стека и падением. И по настоящий день, изоязык-как-макрос остается решительно *нежизнеспособным* методом разработки на Rust.

[13] https://www.reddit.com/r/rust/comments/39wvrm/hodor_esolang_as_a_rust_macro/cs76rqk?context=10000