Rust Macros Tutorial

1 Introduction

Functions are the primary tool that programmers can use to build abstractions. Sometimes, however, programmers want to abstract over compile-time syntax rather than run-time values. Macros provide syntactic abstraction. For an example of how this can be useful, consider the following two code fragments, which both pattern-match on their input and both return early in one case, doing nothing otherwise:

```
match input_1 {
    special_a(x) => { return x; }
    _ => {}
}
// ...
match input_2 {
    special_b(x) => { return x; }
    _ => {}
}
```

This code could become tiresome if repeated many times. However, no function can capture its functionality to make it possible to abstract the repetition away. Rust's macro system, however, can eliminate the repetition. Macros are lightweight custom syntax extensions, themselves defined using the `macro_rules!` syntax extension. The following `early_return` macro captures the pattern in the above code:

```
macro_rules! early_return(
    ($inp:expr $sp:ident) => ( // invoke it like `(input_5 special_e)`
        match $inp {
            $sp(x) => { return x; }
            _ => {}
        }
    );
)
// ...
```

Rust 0.8-pre

fbeeeebf

```
early_return!(input_1 special_a);
// ...
early_return!(input_2 special_b);
```

Macros are defined in pattern-matching style: in the above example, the text `($inp:expr $sp:ident)` that appears on the left-hand side of the => is the *macro invocation syntax*, a pattern denoting how to write a call to the macro. The text on the right-hand side of the =>, beginning with `match $inp`, is the *macro transcription syntax*: what the macro expands to.

2 Invocation syntax

The macro invocation syntax specifies the syntax for the arguments to the macro. It appears on the left-hand side of the => in a macro definition. It conforms to the following rules:

1. It must be surrounded by parentheses.
2. `$` has special meaning (described below).
3. The `()`s, `[]`s, and `{}`s it contains must balance. For example, `([)` is forbidden.

Otherwise, the invocation syntax is free-form.

To take as an argument a fragment of Rust code, write `$` followed by a name (for use on the right-hand side), followed by a `:`, followed by a *fragment specifier*. The fragment specifier denotes the sort of fragment to match. The most common fragment specifiers are:

- `ident` (an identifier, referring to a variable or item. Examples: `f`, `x`, `foo`.)
- `expr` (an expression. Examples: `2 + 2`; `if true then { 1 } else { 2 }`; `f(42)`.)
- `ty` (a type. Examples: `int`, `~[(char, ~str)]`, `&T`.)
- `pat` (a pattern, usually appearing in a `match` or on the left-hand side of a declaration. Examples: `Some(t)`; `(17, 'a')`; `_`.)
- `block` (a sequence of actions. Example: `{ log(error, "hi"); return 12; }`)

The parser interprets any token that's not preceded by a `$` literally. Rust's usual rules of tokenization apply,

So `($x:ident -> (($e:expr)))`, though excessively fancy, would designate a macro that could be invoked like: `my_macro!(i->(( 2+2 )))`.

2.1 Invocation location

A macro invocation may take the place of (and therefore expand to) an expression, an item, or a statement. The Rust parser will parse the macro invocation as a "placeholder" for whichever of those three nonterminals is appropriate for the location.

At expansion time, the output of the macro will be parsed as whichever of the three nonterminals it stands in for. This means that a single macro might, for example, expand to an item or an expression, depending on its arguments (and cause a syntax error if it is called with the wrong argument for its location). Although this behavior sounds excessively dynamic, it is known to be useful under some

circumstances.

3 Transcription syntax

The right-hand side of the => follows the same rules as the left-hand side, except that a `$` need only be followed by the name of the syntactic fragment to transcribe into the macro expansion; its type need not be repeated.

The right-hand side must be enclosed by delimiters, which the transcriber ignores. Therefore `() => ((1,2,3))` is a macro that expands to a tuple expression, `() => (let $x=$val)` is a macro that expands to a statement, and `() => (1,2,3)` is a macro that expands to a syntax error (since the transcriber interprets the parentheses on the right-hand-size as delimiters, and `1,2,3` is not a valid Rust expression on its own).

Except for permissibility of `$name` (and `$(...)*`, discussed below), the right-hand side of a macro definition is ordinary Rust syntax. In particular, macro invocations (including invocations of the macro currently being defined) are permitted in expression, statement, and item locations. However, nothing else about the code is examined or executed by the macro system; execution still has to wait until run-time.

3.1 Interpolation location

The interpolation `$argument_name` may appear in any location consistent with its fragment specifier (i.e., if it is specified as `ident`, it may be used anywhere an identifier is permitted).

4 Multiplicity

4.1 Invocation

Going back to the motivating example, recall that `early_return` expanded into a `match` that would `return` if the `match`'s scrutinee matched the "special case" identifier provided as the second argument to `early_return`, and do nothing otherwise. Now suppose that we wanted to write a version of `early_return` that could handle a variable number of "special" cases.

The syntax `$(...)*` on the left-hand side of the => in a macro definition accepts zero or more occurrences of its contents. It works much like the * operator in regular expressions. It also supports a separator token (a comma-separated list could be written `$(...),*`), and + instead of * to mean "at least one".

```
macro_rules! early_return(
    ($inp:expr, [ $($sp:ident)|+ ]) => (
        match $inp {
            $(
```

Rust 0.8-pre

fbeeeebf

```
                $sp(x) => { return x; }
            )+
            _ => {}
        }
    );
)
// ...
early_return!(input_1, [special_a|special_c|special_d]);
// ...
early_return!(input_2, [special_b]);
```

4.1.1 Transcription

As the above example demonstrates, `$(...)*` is also valid on the right-hand side of a macro definition. The behavior of `*` in transcription, especially in cases where multiple `*`s are nested, and multiple different names are involved, can seem somewhat magical and intuitive at first. The system that interprets them is called "Macro By Example". The two rules to keep in mind are (1) the behavior of `$(...)*` is to walk through one "layer" of repetitions for all of the `$name`s it contains in lockstep, and (2) each `$name` must be under at least as many `$(...)*`s as it was matched against. If it is under more, it'll be repeated, as appropriate.

4.2 Parsing limitations

For technical reasons, there are two limitations to the treatment of syntax fragments by the macro parser:

1. The parser will always parse as much as possible of a Rust syntactic fragment. For example, if the comma were omitted from the syntax of `early_return!` above, `input_1 [` would've been interpreted as the beginning of an array index. In fact, invoking the macro would have been impossible.
2. The parser must have eliminated all ambiguity by the time it reaches a `$name:fragment_specifier` declaration. This limitation can result in parse errors when declarations occur at the beginning of, or immediately after, a `$(...)*`. For example, the grammar `$($t:ty)* $e:expr` will always fail to parse because the parser would be forced to choose between parsing `t` and parsing `e`. Changing the invocation syntax to require a distinctive token in front can solve the problem. In the above example, `$(T $t:ty)* E $e:exp` solves the problem.

5 Macro argument pattern matching

Now consider code like the following:

5.1 Motivation

Rust 0.8-pre

fbeeeebf

```
match x {
```

```
    good_1(g1, val) => {
        match g1.body {
            good_2(result) => {
                // complicated stuff goes here
                return result + val;
            },
            _ => fail!("Didn't get good_2")
        }
    }
    _ => return 0 // default value
}
```

All the complicated stuff is deeply indented, and the error-handling code is separated from matches that fail. We'd like to write a macro that performs a match, but with a syntax that suits the problem better. The following macro can solve the problem:

```
macro_rules! biased_match (
    // special case: `let (x) = ...` is illegal, so use `let x = ...` instead
    ( ($e:expr) ~ ($p:pat) else $err:stmt ;
      binds $bind_res:ident
    ) => (
        let $bind_res = match $e {
            $p => ( $bind_res ),
            _ => { $err }
        };
    );
    // more than one name; use a tuple
    ( ($e:expr) ~ ($p:pat) else $err:stmt ;
      binds $( $bind_res:ident ),*
    ) => (
        let ( $( $bind_res ),* ) = match $e {
            $p => ( $( $bind_res ),* ),
            _ => { $err }
        };
    )
)

biased_match!((x)       ~ (good_1(g1, val)) else { return 0 };
              binds g1, val )
biased_match!((g1.body) ~ (good_2(result) )
                  else { fail!("Didn't get good_2") };
              binds result )
// complicated stuff goes here
return result + val;
```

This solves the indentation problem. But if we have a lot of chained matches like this, we might prefer to write a single macro invocation. The input pattern we want is clear:

Rust 0.8-pre

fbeeeebf

```
    ( $( ($e:expr) ~ ($p:pat) else $err:stmt ; )*
      binds $( $bind_res:ident ),*
    )
```

However, it's not possible to directly expand to nested match statements. But there is a solution.

5.2 The recursive approach to macro writing

A macro may accept multiple different input grammars. The first one to successfully match the actual argument to a macro invocation is the one that "wins".

In the case of the example above, we want to write a recursive macro to process the semicolon-terminated lines, one-by-one. So, we want the following input patterns:

```
    ( binds $( $bind_res:ident ),* )
```

...and:

```
    (    ($e     :expr) ~ ($p     :pat) else $err     :stmt ;
      $( ($e_rest:expr) ~ ($p_rest:pat) else $err_rest:stmt ; )*
      binds  $( $bind_res:ident ),*
    )
```

The resulting macro looks like this. Note that the separation into `biased_match!` and `biased_match_rec!` occurs only because we have an outer piece of syntax (the `let`) which we only want to transcribe once.

```
macro_rules! biased_match_rec (
    // Handle the first layer
    (   ($e     :expr) ~ ($p     :pat) else $err     :stmt ;
     $( ($e_rest:expr) ~ ($p_rest:pat) else $err_rest:stmt ; )*
     binds $( $bind_res:ident ),*
    ) => (
        match $e {
            $p => {
                // Recursively handle the next layer
                biased_match_rec!($( ($e_rest) ~ ($p_rest) else $err_rest ; )*
                                  binds $( $bind_res ),*
                )
            }
            _ => { $err }
        }
    );
    ( binds $( $bind_res:ident ),* ) => ( ($( $bind_res ),*) )
)

// Wrap the whole thing in a `let`.
```

Rust 0.8-pre

fbeeeebf

```
macro_rules! biased_match (
    // special case: `let (x) = ...` is illegal, so use `let x = ...` instead
    ( $( ($e:expr) ~ ($p:pat) else $err:stmt ; )*
      binds $bind_res:ident
    ) => (
        let ( $( $bind_res ),* ) = biased_match_rec!(
            $( ($e) ~ ($p) else $err ; )*
            binds $bind_res
        );
    );
    // more than one name: use a tuple
    ( $( ($e:expr) ~ ($p:pat) else $err:stmt ; )*
      binds  $( $bind_res:ident ),*
    ) => (
        let ( $( $bind_res ),* ) = biased_match_rec!(
            $( ($e) ~ ($p) else $err ; )*
            binds $( $bind_res ),*
        );
    )
)



biased_match!(
    (x)       ~ (good_1(g1, val)) else { return 0 };
    (g1.body) ~ (good_2(result) ) else { fail!("Didn't get good_2") };
    binds val, result )
// complicated stuff goes here
return result + val;
```

This technique applies to many cases where transcribing a result all at once is not possible. The resulting code resembles ordinary functional programming in some respects, but has some important differences from functional programming.

The first difference is important, but also easy to forget: the transcription (right-hand) side of a `macro_rules!` rule is literal syntax, which can only be executed at run-time. If a piece of transcription syntax does not itself appear inside another macro invocation, it will become part of the final program. If it is inside a macro invocation (for example, the recursive invocation of `biased_match_rec!`), it does have the opportunity to affect transcription, but only through the process of attempted pattern matching.

The second, related, difference is that the evaluation order of macros feels "backwards" compared to ordinary programming. Given an invocation `m1!(m2!())`, the expander first expands `m1!`, giving it as input the literal syntax `m2!()`. If it transcribes its argument unchanged into an appropriate position (in particular, not as an argument to yet another macro invocation), the expander will then proceed to evaluate `m2!()` (along with any other macro invocations `m1!(m2!())` produced).

6 A final note

Macros, as currently implemented, are not for the faint of heart. Even ordinary syntax errors can be more difficult to debug when they occur inside a macro, and errors caused by parse problems in generated code can be very tricky. Invoking the `log_syntax!` macro can help elucidate intermediate states, invoking `trace_macros!(true)` will automatically print those intermediate states out, and passing the flag `--pretty expanded` as a command-line argument to the compiler will show the result of expansion.