MACROS 1.1 + SYN + QUOTE

@dtolnay <David Tolnay>

github.com/dtolnay/talks

Servo

```
$ ./mach run --memory-profile
  2.66 MiB -- url(https://servo.org/)
     0.42 MiB -- layout-thread
        0.37 MiB -- stylist
        0.04 MiB -- display-list
        0.00 MiB -- local-context
     0.09 MiB -- dom-tree
```

```rust
pub trait HeapSizeOf {
  fn heap_size_of_children(&self) -> usiz
}
```

```rust
pub trait HeapSizeOf {
  fn heap_size_of_childre(&self) -> usize
}
```

```rust
pub trait HeapSizeOf {
  fn heap_size_of_childr(&self) -> usize;
}
```

```rust
pub trait HeapSizeOf {
  fn heap_size_of_child(&self) -> usize;
}
```

```rust
pub trait HeapSizeOf {
  fn heap_size_of_chil(&self) -> usize;
}
```

```rust
pub trait HeapSizeOf {
  fn heap_size_of_chi(&self) -> usize;
}
```

```rust
pub trait HeapSizeOf {
  fn heap_size_of_ch(&self) -> usize;
}
```

```rust
pub trait HeapSizeOf {
  fn heap_size_of_c(&self) -> usize;
}
```

```rust
pub trait HeapSizeOf {
  fn heap_size_of_(&self) -> usize;
}
```

```rust
pub trait HeapSizeOf {
  fn heap_size_of(&self) -> usize;
}
```

```rust
pub trait HeapSizeO {
  fn heap_size_o(&self) -> usize;
}
```

```rust
pub trait HeapSize {
   fn heap_size_(&self) -> usize;
}
```

```rust
pub trait HeapSize {
    fn heap_size(&self) -> usize;
}
```

```rust
impl HeapSize for u8 {
    fn heap_size(&self) -> usize {
        0
    }
}
```

```rust
impl HeapSize for u8 {
    fn heap_size(&self) -> usize {
        0
    }
}
```

```rust
impl HeapSize for u8 {
    fn heap_size(&self) -> usize {
        0
    }
}
```

```rust
impl HeapSize for u8 {
    fn heap_size(&self) -> usize {
        0
    }
}
```

```rust
impl<T> HeapSize for Box<T>
  where T: HeapSize + ?Sized
{
  fn heap_size(&self) -> usize {
    size_of_alloc(&**self as *const T)
     + (**self).heap_size()
  }
}
```

```rust
impl<T> HeapSize for Box<T>
  where T: HeapSize + ?Sized
{
  fn heap_size(&self) -> usize {
    size_of_alloc(&**self as *const T)
     + (**self).heap_size()
  }
}
```

```rust
impl<T> HeapSize for Box<T>
  where T: HeapSize + ?Sized
{
  fn heap_size(&self) -> usize {
    size_of_alloc(&**self as *const T)
      + (**self).heap_size()
  }
}
```

```rust
impl<T> HeapSize for Box<T>
  where T: HeapSize + ?Sized
{
  fn heap_size(&self) -> usize {
    size_of_alloc(&**self as *const T)
     + (**self).heap_size()
  }
}
```

```rust
impl<T> HeapSize for Box<T>
  where T: HeapSize + ?Sized
{
  fn heap_size(&self) -> usize {
    size_of_alloc(&**self as *const T)
     + (**self).heap_size()
  }
}
```

```rust
impl<T> HeapSize for Box<T>
  where T: HeapSize + ?Sized
{
  fn heap_size(&self) -> usize {
    size_of_alloc(&**self as *const T)
     + (**self).heap_size()
  }
}
```

```rust
impl<T> HeapSize for [T]
  where T: HeapSize
{
  fn heap_size(&self) -> usize {
    self.iter()
        .map(HeapSize::heap_size)
        .sum()
  }
}
```

```
impl<T> HeapSize for [T]
  where T: HeapSize
{
  fn heap_size(&self) -> usize {
    self.iter()
        .map(HeapSize::heap_size)
        .sum()
  }
}
```

```
impl<T> HeapSize for [T]
  where T: HeapSize
{
  fn heap_size(&self) -> usize {
    self.iter()
        .map(HeapSize::heap_size)
        .sum()
  }
}
```

```rust
impl<T> HeapSize for [T]
  where T: HeapSize
{
  fn heap_size(&self) -> usize {
    self.iter()
        .map(HeapSize::heap_size)
        .sum()
  }
}
```

```rust
impl<T> HeapSize for [T]
  where T: HeapSize
{
  fn heap_size(&self) -> usize {
    self.iter()
        .map(HeapSize::heap_size)
        .sum()
  }
}
```

```rust
impl<T> HeapSize for [T]
  where T: HeapSize
{
  fn heap_size(&self) -> usize {
    self.iter()
        .map(HeapSize::heap_size)
        .sum()
  }
}
```

```rust
impl<T> HeapSize for [T]
  where T: HeapSize
{
  fn heap_size(&self) -> usize {
    self.iter()
        .map(HeapSize::heap_size)
        .sum()
  }
}
```

```rust
pub struct Gradient {
    pub stops: Vec<ColorStop>,
    pub repeating: bool,
    pub gradient_kind: GradientKind,
}
```

```rust
pub struct Gradient {
    pub stops: Vec<ColorStop>,
    pub repeating: bool,
    pub gradient_kind: GradientKind,
}

impl HeapSize for Gradient {
    fn heap_size(&self) -> usize {
        self.stops.heap_size()
         + self.repeating.heap_size()
         + self.gradient_kind.heap_size()
    }
}
```

```rust
pub struct Gradient {
  pub stops: Vec<ColorStop>,
  pub repeating: bool,
  pub gradient_kind: GradientKind,
}

impl HeapSize for Gradient {
  fn heap_size(&self) -> usize {
    self.stops.heap_size()
     + self.repeating.heap_size()
     + self.gradient_kind.heap_size()
  }
}
```

```rust
pub struct Gradient {
    pub stops: Vec<ColorStop>,
    pub repeating: bool,
    pub gradient_kind: GradientKind,
}

impl HeapSize for Gradient {
    fn heap_size(&self) -> usize {
        self.stops.heap_size()
            + self.repeating.heap_size()
            + self.gradient_kind.heap_size()
    }
}
```

```rust
pub struct Gradient {
    pub stops: Vec<ColorStop>,
    pub repeating: bool,
    pub gradient_kind: GradientKind,
}

impl HeapSize for Gradient {
    fn heap_size(&self) -> usize {
        self.stops.heap_size()
            + self.repeating.heap_size()
            + self.gradient_kind.heap_size()
    }
}
```

```rust
pub struct Gradient {
    pub stops: Vec<ColorStop>,
    pub repeating: bool,
    pub gradient_kind: GradientKind,
}

impl HeapSize for Gradient {
    fn heap_size(&self) -> usize {
        self.stops.heap_size()
            + self.repeating.heap_size()
            + self.gradient_kind.heap_size()
    }
}
```

```rust
pub struct Gradient {
  pub stops: Vec<ColorStop>,
  pub repeating: bool,
  pub gradient_kind: GradientKind,
}

impl HeapSize for Gradient {
  fn heap_size(&self) -> usize {
    self.stops.heap_size()
     + self.repeating.heap_size()
     + self.gradient_kind.heap_size()
  }
}
```

```
println!("Hello, {}", person)
```

```
println!("Hello, {}", person)

assert!(x.len() > 0);
```

```
println!("Hello, {}", person)

assert!(x.len() > 0);
```

```
::std::io::_print(::std::fmt::Arguments::new_v1(
  {
    static __STATIC_FMTSTR: &'static [&'static str] = &["Hello, ", "\n"];
    __STATIC_FMTSTR
  },
  &match (&person,) {
    (__arg0,) => [
      ::std::fmt::ArgumentV1::new(__arg0, ::std::fmt::Display::fmt),
    ],
  }
));
```

```
assert!(x.len() > 0);
```

```
::std::io::_print(::std::fmt::Arguments::new_v1(
  {
    static __STATIC_FMTSTR: &'static [&'static str] = &["Hello, ", "\n"];
    __STATIC_FMTSTR
  },
  &match (&person,) {
    (__arg0,) => [
      ::std::fmt::ArgumentV1::new(__arg0, ::std::fmt::Display::fmt),
    ],
  }
));
```

```
assert!(x.len() > 0);
```

```
::std::io::_print(::std::fmt::Arguments::new_v1(
  {
    static __STATIC_FMTSTR: &'static [&'static str] = &["Hello, ", "\n"];
    __STATIC_FMTSTR
  },
  &match (&person,) {
    (__arg0,) => [
      ::std::fmt::ArgumentV1::new(__arg0, ::std::fmt::Display::fmt),
    ],
  }
));

if !(x.len() > 0) {
  ::std::rt::begin_panic(
    "assertion failed: x.len() > 0",
    {
      static _FILE_LINE: (&'static str, u32) = ("src/main.rs", 3u32);
      &_FILE_LINE
    }
  )
}
```

```rust
#[derive(Clone, PartialEq)]
pub struct Gradient {
    pub stops: Vec<ColorStop>,
    pub repeating: bool,
    pub gradient_kind: GradientKind,
}
```

```rust
#[derive(Clone, PartialEq)]
pub struct Gradient {
    pub stops: Vec<ColorStop>,
    pub repeating: bool,
    pub gradient_kind: GradientKind,
}
```

```rust
#[derive(Clone, PartialEq, HeapSize)]
pub struct Gradient {
    pub stops: Vec<ColorStop>,
    pub repeating: bool,
    pub gradient_kind: GradientKind,
}
```

```rust
#[derive(Clone, PartialEq, HeapSize)]
pub struct Gradient {
    pub stops: Vec<ColorStop>,
    pub repeating: bool,
    pub gradient_kind: GradientKind,
}
```

```
#![feature(proc_macro)]
```

```rust
#[proc_macro_derive(HeapSize)]
pub fn heap_size(input: TokenStream)
                    -> TokenStream
{
  /* ... */
}
```

```
#[proc_macro_derive(HeapSize)]
pub fn heap_size(input: TokenStream)
                 -> TokenStream
{
  /* ... */
}
```

```
#[proc_macro_derive(HeapSize)]
pub fn heap_size(input: TokenStream)
                    -> TokenStream
{
  /* ... */
}
```

```
#[proc_macro_derive(HeapSize)]
pub fn heap_size(input: TokenStream)
                        -> TokenStream
{
    /* ... */
}
```

```rust
#[proc_macro_derive(HeapSize)]
pub fn heap_size(input: TokenStream)
                    -> TokenStream
{
    /* ... */
}
```

```rust
#[proc_macro_derive(HeapSize)]
pub fn heap_size(input: TokenStream)
                    -> TokenStream
{
  /* ... */
}
```

```
#![feature(proc_macro)]
```

```
#[proc_macro_derive(HeapSize)]
pub fn heap_size(input: TokenStream)
                    -> TokenStream
{
  /* ... */
}
```

```
#![feature(proc_macro, proc_macro_lib)]

extern crate proc_macro;


#[proc_macro_derive(HeapSize)]
pub fn heap_size(input: TokenStream)
                    -> TokenStream
{
  /* ... */
}
```

```
#![feature(proc_macro, proc_macro_lib)]

extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro_derive(HeapSize)]
pub fn heap_size(input: TokenStream)
                 -> TokenStream
{
  /* ... */
}
```

```
#![feature(proc_macro, proc_macro_lib)]

extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro_derive(HeapSize)]
pub fn heap_size(input: TokenStream)
                 -> TokenStream
{
  /* ... */
}
```

```
#![feature(proc_macro, proc_macro_lib)]

extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro_derive(HeapSize)]
pub fn heap_size(input: TokenStream)
                 -> TokenStream
{
    /* ... */
}
```

```rust
let s = input.to_string();

// Parse the string representation
let ast =
  syn::parse_macro_input(&s).unwrap();

// Build the impl
let gen = impl_heap_size(&ast);

// Return the generated impl
gen.parse().unwrap()
```

```rust
let s = input.to_string();

// Parse the string representation
let ast =
   syn::parse_macro_input(&s).unwrap();

// Build the impl
let gen = impl_heap_size(&ast);

// Return the generated impl
gen.parse().unwrap()
```

```rust
let s = input.to_string();

// Parse the string representation
let ast =
   syn::parse_macro_input(&s).unwrap();

// Build the impl
let gen = impl_heap_size(&ast);

// Return the generated impl
gen.parse().unwrap()
```

```rust
let s = input.to_string();

// Parse the string representation
let ast =
  syn::parse_macro_input(&s).unwrap();

// Build the impl
let gen = impl_heap_size(&ast);

// Return the generated impl
gen.parse().unwrap()
```

```rust
let s = input.to_string();

// Parse the string representation
let ast =
  syn::parse_macro_input(&s).unwrap();

// Build the impl
let gen = impl_heap_size(&ast);

// Return the generated impl
gen.parse().unwrap()
```

```rust
let s = input.to_string();

// Parse the string representation
let ast =
  syn::parse_macro_input(&s).unwrap();

// Build the impl
let gen = impl_heap_size(&ast);

// Return the generated impl
gen.parse().unwrap()
```

```rust
let s = input.to_string();

// Parse the string representation
let ast =
  syn::parse_macro_input(&s).unwrap();

// Build the impl
let gen = impl_heap_size(&ast);

// Return the generated impl
gen.parse().unwrap()
```

```rust
fn impl_heap_size(ast: &syn::MacroInput)
                  -> quote::Tokens
{
    let name = &ast.ident;
    let sum = heap_size_sum(&ast.body);
    quote! {
        impl HeapSize for #name {
            fn heap_size(&self) -> usize {
                #sum
            }
        }
    }
}
```

```
fn impl_heap_size(ast: &syn::MacroInput)
                  -> quote::Tokens
{
  let name = &ast.ident;
  let sum = heap_size_sum(&ast.body);
  quote! {
    impl HeapSize for #name {
      fn heap_size(&self) -> usize {
        #sum
      }
    }
  }
}
```

```
fn impl_heap_size(ast: &syn::MacroInput)
                      -> quote::Tokens
{
  let name = &ast.ident;
  let sum = heap_size_sum(&ast.body);
  quote! {
    impl HeapSize for #name {
      fn heap_size(&self) -> usize {
        #sum
      }
    }
  }
}
```

```
fn impl_heap_size(ast: &syn::MacroInput)
                      -> quote::Tokens
{
    let name = &ast.ident;
    let sum = heap_size_sum(&ast.body);
    quote! {
        impl HeapSize for #name {
            fn heap_size(&self) -> usize {
                #sum
            }
        }
    }
}
```

```
fn impl_heap_size(ast: &syn::MacroInput)
                      -> quote::Tokens
{
  let name = &ast.ident;
  let sum = heap_size_sum(&ast.body);
  quote! {
    impl HeapSize for #name {
      fn heap_size(&self) -> usize {
        #sum
      }
    }
  }
}
```

```
fn impl_heap_size(ast: &syn::MacroInput)
                      -> quote::Tokens
{
  let name = &ast.ident;
  let sum = heap_size_sum(&ast.body);
  quote! {
    impl HeapSize for #name {
      fn heap_size(&self) -> usize {
        #sum
      }
    }
  }
}
```

```
fn impl_heap_size(ast: &syn::MacroInput)
                      -> quote::Tokens
{
  let name = &ast.ident;
  let sum = heap_size_sum(&ast.body);
  quote! {
    impl HeapSize for #name {
      fn heap_size(&self) -> usize {
        #sum
      }
    }
  }
}
```

```
fn impl_heap_size(ast: &syn::MacroInput)
                      -> quote::Tokens
{
  let name = &ast.ident;
  let sum = heap_size_sum(&ast.body);
  quote! {
    impl HeapSize for #name {
      fn heap_size(&self) -> usize {
        #sum
      }
    }
  }
}
```

```
fn heap_size_sum(body: &syn::Body)
                 -> quote::Tokens
{
    match *body {
        syn::Body::Struct(ref fields) => {
            struct_sum(fields)
        }
        syn::Body::Enum(_) => {
            unimplemented!()
        }
    }
}
```

```
fn heap_size_sum(body: &syn::Body)
                 -> quote::Tokens
{
  match *body {
    syn::Body::Struct(ref fields) => {
      struct_sum(fields)
    }
    syn::Body::Enum(_) => {
      unimplemented!()
    }
  }
}
```

```
fn heap_size_sum(body: &syn::Body)
                 -> quote::Tokens
{
  match *body {
    syn::Body::Struct(ref fields) => {
      struct_sum(fields)
    }
    syn::Body::Enum(_) => {
      unimplemented!()
    }
  }
}
```

```
fn heap_size_sum(body: &syn::Body)
                 -> quote::Tokens
{
    match *body {
        syn::Body::Struct(ref fields) => {
            struct_sum(fields)
        }
        syn::Body::Enum(_) => {
            unimplemented!()
        }
    }
}
```

```
fn heap_size_sum(body: &syn::Body)
                 -> quote::Tokens
{
  match *body {
    syn::Body::Struct(ref fields) => {
      struct_sum(fields)
    }
    syn::Body::Enum(_) => {
      unimplemented!()
    }
  }
}
```

```
fn struct_sum(fields: &syn::VariantData)
                -> quote::Tokens
{
  match *fields {
    syn::VariantData::Struct(ref s) => {
      braced_struct_sum(s)
    }
    syn::VariantData::Tuple(ref t) => {
      tuple_struct_sum(t)
    }
    syn::VariantData::Unit => quote!(0)
  }
}
```

```
fn struct_sum(fields: &syn::VariantData)
                -> quote::Tokens
{
  match *fields {
    syn::VariantData::Struct(ref s) => {
      braced_struct_sum(s)
    }
    syn::VariantData::Tuple(ref t) => {
      tuple_struct_sum(t)
    }
    syn::VariantData::Unit => quote!(0)
  }
}
```

```
fn struct_sum(fields: &syn::VariantData)
                  -> quote::Tokens
{
  match *fields {
    syn::VariantData::Struct(ref s) => {
      braced_struct_sum(s)
    }
    syn::VariantData::Tuple(ref t) => {
      tuple_struct_sum(t)
    }
    syn::VariantData::Unit => quote!(0)
  }
}
```

```
fn struct_sum(fields: &syn::VariantData)
                -> quote::Tokens
{
  match *fields {
    syn::VariantData::Struct(ref s) => {
      braced_struct_sum(s)
    }
    syn::VariantData::Tuple(ref t) => {
      tuple_struct_sum(t)
    }
    syn::VariantData::Unit => quote!(0)
  }
}
```

```
fn struct_sum(fields: &syn::VariantData)
                -> quote::Tokens
{
  match *fields {
    syn::VariantData::Struct(ref s) => {
      braced_struct_sum(s)
    }
    syn::VariantData::Tuple(ref t) => {
      tuple_struct_sum(t)
    }
    syn::VariantData::Unit => quote!(0)
  }
}
```

```
fn struct_sum(fields: &syn::VariantData)
                -> quote::Tokens
{
  match *fields {
    syn::VariantData::Struct(ref s) => {
      braced_struct_sum(s)
    }
    syn::VariantData::Tuple(ref t) => {
      tuple_struct_sum(t)
    }
    syn::VariantData::Unit => quote!(0)
  }
}
```

```rust
fn braced_struct_sum(fs: &[syn::Field])
                     -> quote::Tokens
{
    let fnames = fs.iter()
                   .map(|f| &f.ident);
    quote! {
        0 #(
            + self.#fnames.heap_size()
        )*
    }
}
```

```
fn braced_struct_sum(fs: &[syn::Field])
                        -> quote::Tokens
{
    let fnames = fs.iter()
                   .map(|f| &f.ident);
    quote! {
        0 #(
            + self.#fnames.heap_size()
        )*
    }
}
```

```
fn braced_struct_sum(fs: &[syn::Field])
                        -> quote::Tokens
{
    let fnames = fs.iter()
                   .map(|f| &f.ident);
    quote! {
        0 #(
            + self.#fnames.heap_size()
        )*
    }
}
```

```
fn braced_struct_sum(fs: &[syn::Field])
                        -> quote::Tokens
{
    let fnames = fs.iter()
                   .map(|f| &f.ident);
    quote! {
        0 #(
            + self.#fnames.heap_size()
        )*
    }
}
```

```
fn braced_struct_sum(fs: &[syn::Field])
                        -> quote::Tokens
{
    let fnames = fs.iter()
                   .map(|f| &f.ident);
    quote! {
        0 #(
            + self.#fnames.heap_size()
        )*
    }
}
```

```rust
fn braced_struct_sum(fs: &[syn::Field])
                        -> quote::Tokens
{
    let fnames = fs.iter()
                   .map(|f| &f.ident);
    quote! {
        0 #(
            + self.#fnames.heap_size()
        )*
    }
}
```

```
fn braced_struct_sum(fs: &[syn::Field])
                        -> quote::Tokens
{
    let fnames = fs.iter()
                   .map(|f| &f.ident);
    quote! {
        0 #(
            + self.#fnames.heap_size()
        )*
    }
}
```

```
fn braced_struct_sum(fs: &[syn::Field])
                        -> quote::Tokens
{
    let fnames = fs.iter()
                   .map(|f| &f.ident);
    quote! {
        0 #(
            + self.#fnames.heap_size()
        )*
    }
}
```

```
fn braced_struct_sum(fs: &[syn::Field])
                        -> quote::Tokens
{
    let fnames = fs.iter()
                   .map(|f| &f.ident);
    quote! {
        0 #(
            + self.#fnames.heap_size()
        )*
    }
}
```

```
fn tuple_struct_sum(fs: &[syn::Field])
                    -> quote::Tokens
{
    let indices = 0..fs.len();

    quote! {
        0 #(
            + self.#indices.heap_size()
        )*
    }
}
```

```
fn tuple_struct_sum(fs: &[syn::Field])
                    -> quote::Tokens
{
    let indices = 0..fs.len();

    quote! {
        0 #(
            + self.#indices.heap_size()
        )*
    }
}
```

```
fn tuple_struct_sum(fs: &[syn::Field])
                    -> quote::Tokens
{
    let indices = 0..fs.len();

    quote! {
        0 #(
            + self.#indices.heap_size()
        )*
    }
}
```

```
fn tuple_struct_sum(fs: &[syn::Field])
                    -> quote::Tokens
{
    let indices = 0..fs.len();

    quote! {
        0 #(
            + self.#indices.heap_size()
        )*
    }
}
```

```
fn tuple_struct_sum(fs: &[syn::Field])
                    -> quote::Tokens
{
  let indices = 0..fs.len();

  quote! {
    0 #(
      + self.#indices.heap_size()
    )*
  }
}
```

```
fn tuple_struct_sum(fs: &[syn::Field])
                    -> quote::Tokens
{
    let indices = 0..fs.len();

    quote! {
        0 #(
            + self.#indices.heap_size()
        )*
    }
}
```

```
fn tuple_struct_sum(fs: &[syn::Field])
                    -> quote::Tokens
{
    let indices = 0..fs.len();

    quote! {
        0 #(
            + self.#indices.heap_size()
        )*
    }
}
```

```
fn tuple_struct_sum(fs: &[syn::Field])
                    -> quote::Tokens
{
    let indices = 0..fs.len();

    quote! {
        0 #(
            + self.#indices.heap_size()
        )*
    }
}
```

```rust
#![feature(proc_macro, proc_macro_lib)]

extern crate proc_macro;
extern crate syn;
#[macro_use] extern crate quote;

use proc_macro::TokenStream;
use syn::{Body, VariantData};

#[proc_macro_derive(HeapSize)]
pub fn heap_size(input: TokenStream)
                 -> TokenStream {
  let s = input.to_string();
  let ast = syn::parse_macro_input(&s)
                .unwrap();

  let name = &ast.ident;
  let sum = heap_size_sum(&ast.body);
  let gen = quote! {
    impl HeapSize for #name {
      fn num_fields() -> usize {
        #sum
      }
    }
  };

  gen.parse().unwrap()
}
```

```rust
fn heap_size_sum(body: &Body) -> quote::Tokens {
  match *body {
    Body::Struct(VariantData::Struct(ref fs)) => {
      let fnames = fs.iter().map(|f| &f.ident);
      quote! {
        0 #(
          + self.#fnames.heap_size()
        )*
      }
    }
    Body::Struct(VariantData::Tuple(ref fs)) => {
      let indices = 0..fs.len();
      quote! {
        0 #(
          + self.#indices.heap_size()
        )*
      }
    }
    Body::Struct(VariantData::Unit) => quote!(0),
    Body::Enum(_) => unimplemented!(),
  }
}
```

```rust
#![feature(proc_macro)]

#[macro_use]
extern crate heapsize_derive;

#[derive(HeapSize)]
pub struct Gradient {
    pub stops: Vec<ColorStop>,
    pub repeating: bool,
    pub gradient_kind: GradientKind,
}
```

```
#![feature(proc_macro)]

#[macro_use]
extern crate heapsize_derive;

#[derive(HeapSize)]
pub struct Gradient {
    pub stops: Vec<ColorStop>,
    pub repeating: bool,
    pub gradient_kind: GradientKind,
}
```

```rust
#![feature(proc_macro)]

#[macro_use]
extern crate heapsize_derive;

#[derive(HeapSize)]
pub struct Gradient {
    pub stops: Vec<ColorStop>,
    pub repeating: bool,
    pub gradient_kind: GradientKind,
}
```

```
#![feature(proc_macro)]

#[macro_use]
extern crate heapsize_derive;

#[derive(HeapSize)]
pub struct Gradient {
    pub stops: Vec<ColorStop>,
    pub repeating: bool,
    pub gradient_kind: GradientKind,
}
```

```rust
#![feature(proc_macro)]

#[macro_use]
extern crate heapsize_derive;

#[derive(HeapSize)]
pub struct Gradient {
    pub stops: Vec<ColorStop>,
    pub repeating: bool,
    pub gradient_kind: GradientKind,
}
```

Syn Internals

```rust
/// e.g. 'std::collections::HashMap'
#[derive(Debug, Clone, Eq, PartialEq)]
pub struct Path {
    pub global: bool,
    pub segments: Vec<PathSegment>,
}
```

```rust
/// e.g. 'std::collections::HashMap'
#[derive(Debug, Clone, Eq, PartialEq)]
pub struct Path {
    pub global: bool,
    pub segments: Vec<PathSegment>,
}
```

```rust
/// e.g. 'std::collections::HashMap'
#[derive(Debug, Clone, Eq, PartialEq)]
pub struct Path {
    pub global: bool,
    pub segments: Vec<PathSegment>,
}
```

```rust
/// e.g. 'std::collections::HashMap'
#[derive(Debug, Clone, Eq, PartialEq)]
pub struct Path {
    pub global: bool,
    pub segments: Vec<PathSegment>,
}
```

```rust
/// e.g. 'std::collections::HashMap'
#[derive(Debug, Clone, Eq, PartialEq)]
pub struct Path {
    pub global: bool,
    pub segments: Vec<PathSegment>,
}
```

```
named!(pub path -> Path, do_parse!(
  global: option!(punct!("::")) >>
  segments: separated_nonempty_list!(
    punct!("::"), path_segment) >>
  (Path {
    global: global.is_some(),
    segments: segments,
  })
));
```

```
named!(pub path -> Path, do_parse!(
  global: option!(punct!("::")) >>
  segments: separated_nonempty_list!(
    punct!("::"), path_segment) >>
  (Path {
    global: global.is_some(),
    segments: segments,
  })
));
```

```
named!(pub path -> Path, do_parse!(
  global: option!(punct!("::")) >>
  segments: separated_nonempty_list!(
    punct!("::"), path_segment) >>
  (Path {
    global: global.is_some(),
    segments: segments,
  })
));
```

```
named!(pub path -> Path, do_parse!(
  global: option!(punct!("::")) >>
  segments: separated_nonempty_list!(
    punct!("::"), path_segment) >>
  (Path {
    global: global.is_some(),
    segments: segments,
  })
));
```

```
named!(pub path -> Path, do_parse!(
  global: option!(punct!("::")) >>
  segments: separated_nonempty_list!(
    punct!("::"), path_segment) >>
  (Path {
    global: global.is_some(),
    segments: segments,
  })
));
```

```
named!(pub path -> Path, do_parse!(
  global: option!(punct!("::")) >>
  segments: separated_nonempty_list!(
    punct!("::"), path_segment) >>
  (Path {
    global: global.is_some(),
    segments: segments,
  })
));
```

```
named!(pub path -> Path, do_parse!(
  global: option!(punct!("::")) >>
  segments: separated_nonempty_list!(
    punct!("::"), path_segment) >>
  (Path {
    global: global.is_some(),
    segments: segments,
  })
));
```

```rust
impl ToTokens for Path {
  fn to_tokens(&self, t: &mut Tokens) {
    for (i, seg) in self.segments
                        .iter()
                        .enumerate() {
      if i > 0 || self.global {
        t.append("::");
      }
      seg.to_tokens(t);
    }
  }
}
```

```rust
impl ToTokens for Path {
    fn to_tokens(&self, t: &mut Tokens) {
        for (i, seg) in self.segments
                                .iter()
                                .enumerate() {
            if i > 0 || self.global {
                t.append("::");
            }
            seg.to_tokens(t);
        }
    }
}
```

```
impl ToTokens for Path {
    fn to_tokens(&self, t: &mut Tokens) {
        for (i, seg) in self.segments
                                .iter()
                                .enumerate() {
            if i > 0 || self.global {
                t.append("::");
            }
            seg.to_tokens(t);
        }
    }
}
```

```rust
impl ToTokens for Path {
  fn to_tokens(&self, t: &mut Tokens) {
    for (i, seg) in self.segments
                        .iter()
                        .enumerate() {
      if i > 0 || self.global {
        t.append("::");
      }
      seg.to_tokens(t);
    }
  }
}
```

```rust
impl ToTokens for Path {
  fn to_tokens(&self, t: &mut Tokens) {
    for (i, seg) in self.segments
                        .iter()
                        .enumerate() {
      if i > 0 || self.global {
        t.append("::");
      }
      seg.to_tokens(t);
    }
  }
}
```

```rust
impl ToTokens for Path {
  fn to_tokens(&self, t: &mut Tokens) {
    for (i, seg) in self.segments
                        .iter()
                        .enumerate() {
      if i > 0 || self.global {
        t.append("::");
      }
      seg.to_tokens(t);
    }
  }
}
```

Quote Internals

```
quote! { return f(x); }
```

```
quote! { return f(x); }

{
    let mut s = quote::Tokens::new();
    s.append("return");
    s.append("f");
    s.append("(");
    s.append("x");
    s.append(")");
    s.append(";");
    s
}
```

```
quote! { return f(x); }

{
    let mut s = quote::Tokens::new();
    s.append("return");
    s.append("f");
    s.append("(");
    s.append("x");
    s.append(")");
    s.append(";");
    s
}
```

```
quote! { return f(x); }

{
    let mut s = quote::Tokens::new();
    s.append("return");
    s.append("f");
    s.append("(");
    s.append("x");
    s.append(")");
    s.append(";");
    s
}
```

```
quote! { return f(x); }

{
    let mut s = quote::Tokens::new();
    s.append("return");
    s.append("f");
    s.append("(");
    s.append("x");
    s.append(")");
    s.append(";");
    s
}
```

```
quote! { return f(x); }

{
    let mut s = quote::Tokens::new();
    s.append("return");
    s.append("f");
    s.append("(");
    s.append("x");
    s.append(")");
    s.append(";");
    s
}
```

```
quote! { return f(x); }

{
    let mut s = quote::Tokens::new();
    s.append("return");
    s.append("f");
    s.append("(");
    s.append("x");
    s.append(")");
    s.append(";");
    s
}
```

```
quote! { return f(x); }

{
    let mut s = quote::Tokens::new();
    s.append("return");
    s.append("f");
    s.append("(");
    s.append("x");
    s.append(")");
    s.append(";");
    s
}
```

```
quote! { return f(x); }

{
    let mut s = quote::Tokens::new();
    s.append("return");
    s.append("f");
    s.append("(");
    s.append("x");
    s.append(")");
    s.append(";");
    s
}
```

```
quote! { return f(x); }

{
    let mut s = quote::Tokens::new();
    s.append("return");
    s.append("f");
    s.append("(");
    s.append("x");
    s.append(")");
    s.append(";");
    s
}
```

```
quote! { return f(#x); }

{
    let mut s = quote::Tokens::new();
    s.append("return");
    s.append("f");
    s.append("(");
    x.to_tokens(&mut s);
    s.append(")");
    s.append(";");
    s
}
```

```
quote! { return f(#x); }

{
    let mut s = quote::Tokens::new();
    s.append("return");
    s.append("f");
    s.append("(");
    x.to_tokens(&mut s);
    s.append(")");
    s.append(";");
    s
}
```

```
quote! { sum = 0 #( + #x )* }
```

```
quote! { sum = 0 #( + #x )* }
```

```
quote! { sum = 0 #( + #x )* }
```

```
quote! { sum = 0 #( + #x )* }

{
    let mut s = quote::Tokens::new();
    s.append("sum");
    s.append("=");
    s.append("0");
    for x in x {
        s.append("+");
        x.to_tokens(&mut s);
    }
    s
}
```

```
quote! { sum = 0 #( + #x )* }

{
    let mut s = quote::Tokens::new();
    s.append("sum");
    s.append("=");
    s.append("0");
    for x in x {
        s.append("+");
        x.to_tokens(&mut s);
    }
    s
}
```

```
quote! { sum = 0 #( + #x )* }

{
    let mut s = quote::Tokens::new();
    s.append("sum");
    s.append("=");
    s.append("0");
    for x in x {
        s.append("+");
        x.to_tokens(&mut s);
    }
    s
}
```

```
quote! { sum = 0 #( + #x )* }

{
    let mut s = quote::Tokens::new();
    s.append("sum");
    s.append("=");
    s.append("0");
    for x in x {
        s.append("+");
        x.to_tokens(&mut s);
    }
    s
}
```

```
quote! { sum = 0 #( + #x )* }

{
    let mut s = quote::Tokens::new();
    s.append("sum");
    s.append("=");
    s.append("0");
    for x in x {
        s.append("+");
        x.to_tokens(&mut s);
    }
    s
}
```

```
quote! { sum = 0 #( + #x )* }

{
    let mut s = quote::Tokens::new();
    s.append("sum");
    s.append("=");
    s.append("0");
    for x in x {
        s.append("+");
        x.to_tokens(&mut s);
    }
    s
}
```

```
quote! { sum = 0 #( + #x )* }

{
    let mut s = quote::Tokens::new();
    s.append("sum");
    s.append("=");
    s.append("0");
    for x in x {
        s.append("+");
        x.to_tokens(&mut s);
    }
    s
}
```

```
quote! { sum = 0 #( + #x )* }

{
    let mut s = quote::Tokens::new();
    s.append("sum");
    s.append("=");
    s.append("0");
    for x in x {
        s.append("+");
        x.to_tokens(&mut s);
    }
    s
}
```

```
quote! { #( #x ),* }
```

```
quote! { #( #x ),* }
```

```
quote! { #( #x ),* }

{
  let mut s = quote::Tokens::new();
  for (i, x) in x.into_iter()
                 .enumerate() {
    if i > 0 {
      s.append(",");
    }
    x.to_tokens(&mut s);
  }
  s
}
```

```
quote! { #( #x ),* }

{
    let mut s = quote::Tokens::new();
    for (i, x) in x.into_iter()
                   .enumerate() {
        if i > 0 {
            s.append(",");
        }
        x.to_tokens(&mut s);
    }
    s
}
```

```
quote! { #( #x ),* }

{
    let mut s = quote::Tokens::new();
    for (i, x) in x.into_iter()
                   .enumerate() {
        if i > 0 {
            s.append(",");
        }
        x.to_tokens(&mut s);
    }
    s
}
```

```
quote! { #( #x ),* }

{
  let mut s = quote::Tokens::new();
  for (i, x) in x.into_iter()
                 .enumerate() {
    if i > 0 {
      s.append(",");
    }
    x.to_tokens(&mut s);
  }
  s
}
```

```
quote! { #( #x ),* }

{
  let mut s = quote::Tokens::new();
  for (i, x) in x.into_iter()
                 .enumerate() {
    if i > 0 {
      s.append(",");
    }
    x.to_tokens(&mut s);
  }
  s
}
```

```
quote! { #( #x => #y, )* }
```

```
quote! { #( #x => #y, )* }
```

```
quote! { #( #x => #y, )* }

{
    let mut s = quote::Tokens::new();
    for (x, y) in x.into_iter().zip(y) {
        x.to_tokens(&mut s);
        s.append("=>");
        y.to_tokens(&mut s);
        s.append(",");
    }
    s
}
```

```
quote! { #( #x => #y, )* }

{
    let mut s = quote::Tokens::new();
    for (x, y) in x.into_iter().zip(y) {
        x.to_tokens(&mut s);
        s.append("=>");
        y.to_tokens(&mut s);
        s.append(",");
    }
    s
}
```

```
quote! { #( #x => #y, )* }

{
    let mut s = quote::Tokens::new();
    for (x, y) in x.into_iter().zip(y) {
        x.to_tokens(&mut s);
        s.append("=>");
        y.to_tokens(&mut s);
        s.append(",");
    }
    s
}
```

```
quote! { #( #x => #y, )* }

{
    let mut s = quote::Tokens::new();
    for (x, y) in x.into_iter().zip(y) {
        x.to_tokens(&mut s);
        s.append("=>");
        y.to_tokens(&mut s);
        s.append(",");
    }
    s
}
```

```
quote! { #( #x => #y, )* }

{
    let mut s = quote::Tokens::new();
    for (x, y) in x.into_iter().zip(y) {
        x.to_tokens(&mut s);
        s.append("=>");
        y.to_tokens(&mut s);
        s.append(",");
    }
    s
}
```

```
quote! { #( #x => #y, )* }

{
    let mut s = quote::Tokens::new();
    for (x, y) in x.into_iter().zip(y) {
        x.to_tokens(&mut s);
        s.append("=>");
        y.to_tokens(&mut s);
        s.append(",");
    }
    s
}
```

MACROS 1.1 + SYN + QUOTE

@dtolnay <David Tolnay>

Questions?