The Rust Reference Manual

0.10 (46867cc 2014-04-02 16:59:39 -0700)

Contents

1 Introduction

This document is the reference manual for the Rust programming language. It provides three kinds of material:

- Chapters that formally define the language grammar and, for each construct, informally describe its semantics and give examples of its use.
- Chapters that informally describe the memory model, concurrency model, runtime services, linkage model and debugging facilities.
- Appendix chapters providing rationale and references to languages that influenced the design.

This document does not serve as a tutorial introduction to the language. Background familiarity with the language is assumed. A separate tutorial document is available to help acquire such background familiarity.

This document also does not serve as a reference to the standard or extra libraries included in the language distribution. Those libraries are documented separately by extracting documentation attributes from their source code.

1.1 Disclaimer

Rust is a work in progress. The language continues to evolve as the design shifts and is fleshed out in working code. Certain parts work, certain parts do not, certain parts will be removed or changed.

This manual is a snapshot written in the present tense. All features described exist in working code unless otherwise noted, but some are quite primitive or remain to be further modified by planned work. Some may be temporary. It is a *draft*, and we ask that you not take anything you read here as final.

If you have suggestions to make, please try to focus them on *reductions* to the language: possible features that can be combined or omitted. We aim to keep the size and complexity of the language under control.

> **Note:** The grammar for Rust given in this document is rough and very incomplete; only a modest number of sections have accompanying grammar rules. Formalizing the grammar accepted by the Rust parser is ongoing work, but future versions of this document will contain a complete grammar. Moreover, we hope that this grammar will be extracted and verified as LL(1) by an automated grammar-analysis tool, and further tested against the Rust sources. Preliminary versions of this automation exist, but are not yet complete.

2 Notation

Rust's grammar is defined over Unicode codepoints, each conventionally denoted `U+XXXX`, for 4 or more hexadecimal digits `X`. *Most* of Rust's grammar is confined to the ASCII range of Unicode, and is described in this document by a dialect of Extended Backus-Naur Form (EBNF), specifically a dialect of EBNF supported by common automated LL(k) parsing tools such as `llgen`, rather than the dialect given in ISO 14977. The dialect can be defined self-referentially as follows:

```
grammar : rule + ;
rule    : nonterminal ':' productionrule ';' ;
productionrule : production [ '|' production ] * ;
production : term * ;
term : element repeats ;
element : LITERAL | IDENTIFIER | '[' productionrule ']' ;
repeats : [ '*' | '+' ] NUMBER ? | NUMBER ? | '?' ;
```

Where:

- Whitespace in the grammar is ignored.

- Square brackets are used to group rules.
- `LITERAL` is a single printable ASCII character, or an escaped hexadecimal ASCII code of the form `\xQQ`, in single quotes, denoting the corresponding Unicode codepoint `U+00QQ`.
- `IDENTIFIER` is a nonempty string of ASCII letters and underscores.
- The `repeat` forms apply to the adjacent `element`, and are as follows:
  - `?` means zero or one repetition
  - `*` means zero or more repetitions
  - `+` means one or more repetitions
  - NUMBER trailing a repeat symbol gives a maximum repetition count
  - NUMBER on its own gives an exact repetition count

This EBNF dialect should hopefully be familiar to many readers.

2.1 Unicode productions

A few productions in Rust's grammar permit Unicode codepoints outside the ASCII range. We define these productions in terms of character properties specified in the Unicode standard, rather than in terms of ASCII-range codepoints. The section Special Unicode Productions lists these productions.

2.2 String table productions

Some rules in the grammar – notably unary operators, binary operators, and keywords – are given in a simplified form: as a listing of a table of unquoted, printable whitespace-separated strings. These cases form a subset of the rules regarding the token rule, and are assumed to be the result of a lexical-analysis phase feeding the parser, driven by a DFA, operating over the disjunction of all such string table entries.

When such a string enclosed in double-quotes (`"`) occurs inside the grammar, it is an implicit reference to a single member of such a string table production. See tokens for more information.

3 Lexical structure

3.1 Input format

Rust input is interpreted as a sequence of Unicode codepoints encoded in UTF-8, normalized to Unicode normalization form NFKC. Most Rust grammar rules are defined in terms of printable ASCII-range codepoints, but a small number are defined in terms of Unicode properties or explicit codepoint lists. [1]

3.2 Special Unicode Productions

The following productions in the Rust grammar are defined in terms of Unicode properties: `ident`, `non_null`, `non_star`, `non_eol`, `non_slash_or_star`, `non_single_quote` and `non_double_quote`.

3.2.1 Identifiers

The `ident` production is any nonempty Unicode string of the following form:

- The first character has property `XID_start`
- The remaining characters have property `XID_continue`

that does *not* occur in the set of keywords.

Note: `XID_start` and `XID_continue` as character properties cover the character ranges used to form the more familiar C and Java language-family identifiers.

3.2.2 Delimiter-restricted productions

Some productions are defined by exclusion of particular Unicode characters:

- `non_null` is any single Unicode character aside from `U+0000` (null)
- `non_eol` is `non_null` restricted to exclude `U+000A` (`'\n'`)
- `non_star` is `non_null` restricted to exclude `U+002A` (`*`)
- `non_slash_or_star` is `non_null` restricted to exclude `U+002F` (`/`) and `U+002A` (`*`)
- `non_single_quote` is `non_null` restricted to exclude `U+0027` (`'`)
- `non_double_quote` is `non_null` restricted to exclude `U+0022` (`"`)

[1] Substitute definitions for the special Unicode productions are provided to the grammar verifier, restricted to ASCII range, when verifying the grammar in this document.

3.3 Comments

```
comment : block_comment | line_comment ;
block_comment : "/*" block_comment_body * '*' + '/' ;
block_comment_body : [block_comment | character] * ;
line_comment : "//" non_eol * ;
```

Comments in Rust code follow the general C++ style of line and block-comment forms, with no nesting of block-comment delimiters.

Line comments beginning with exactly *three* slashes (`///`), and block comments beginning with a exactly one repeated asterisk in the block-open sequence (`/**`), are interpreted as a special syntax for `doc` attributes. That is, they are equivalent to writing `#[doc="..."]` around the body of the comment (this includes the comment characters themselves, ie `/// Foo` turns into `#[doc="/// Foo"]`).

Non-doc comments are interpreted as a form of whitespace.

3.4 Whitespace

```
whitespace_char : '\x20' | '\x09' | '\x0a' | '\x0d' ;
whitespace : [ whitespace_char | comment ] + ;
```

The `whitespace_char` production is any nonempty Unicode string consisting of any of the following Unicode characters: `U+0020` (space, `' '`), `U+0009` (tab, `'\t'`), `U+000A` (LF, `'\n'`), `U+000D` (CR, `'\r'`).

Rust is a "free-form" language, meaning that all forms of whitespace serve only to separate *tokens* in the grammar, and have no semantic significance.

A Rust program has identical meaning if each whitespace element is replaced with any other legal whitespace element, such as a single space character.

3.5 Tokens

```
simple_token : keyword | unop | binop ;
token : simple_token | ident | literal | symbol | whitespace token ;
```

Tokens are primitive productions in the grammar defined by regular (non-recursive) languages. "Simple" tokens are given in string table production form, and occur in the rest of the grammar as double-quoted strings. Other tokens have exact rules given.

3.5.1 Keywords

The keywords are the following strings:

```
as
break
crate
do
else enum extern
false fn for
if impl in
let loop
match mod mut
priv pub
ref return
self static struct super
true trait type
unsafe use
while
```

Each of these keywords has special meaning in its grammar, and all of them are excluded from the `ident` rule.

3.5.2 Literals

A literal is an expression consisting of a single token, rather than a sequence of tokens, that immediately and directly denotes the value it evaluates to, rather than referring to it by name or some other evaluation rule. A literal is a form of constant expression, so is evaluated (primarily) at compile time.

```
literal : string_lit | char_lit | num_lit ;
```

Character and string literals

```
char_lit : '\x27' char_body '\x27' ;
string_lit : '"' string_body * '"' | 'r' raw_string ;

char_body : non_single_quote
          | '\x5c' [ '\x27' | common_escape ] ;

string_body : non_double_quote
            | '\x5c' [ '\x22' | common_escape ] ;
raw_string : '"' raw_string_body '"' | '#' raw_string '#' ;
```

```
common_escape : '\x5c'
              | 'n' | 'r' | 't' | '0'
              | 'x' hex_digit 2
              | 'u' hex_digit 4
              | 'U' hex_digit 8 ;

hex_digit : 'a' | 'b' | 'c' | 'd' | 'e' | 'f'
          | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
          | dec_digit ;
oct_digit : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' ;
dec_digit : '0' | nonzero_dec ;
nonzero_dec: '1' | '2' | '3' | '4'
           | '5' | '6' | '7' | '8' | '9' ;
```

A *character literal* is a single Unicode character enclosed within two `U+0027` (single-quote) characters, with the exception of `U+0027` itself, which must be *escaped* by a preceding U+005C character (\).

A *string literal* is a sequence of any Unicode characters enclosed within two `U+0022` (double-quote) characters, with the exception of `U+0022` itself, which must be *escaped* by a preceding `U+005C` character (\), or a *raw string literal*.

Some additional *escapes* are available in either character or non-raw string literals. An escape starts with a `U+005C` (\) and continues with one of the following forms:

- An *8-bit codepoint escape* escape starts with `U+0078` (`x`) and is followed by exactly two *hex digits*. It denotes the Unicode codepoint equal to the provided hex value.
- A *16-bit codepoint escape* starts with `U+0075` (`u`) and is followed by exactly four *hex digits*. It denotes the Unicode codepoint equal to the provided hex value.
- A *32-bit codepoint escape* starts with `U+0055` (`U`) and is followed by exactly eight *hex digits*. It denotes the Unicode codepoint equal to the provided hex value.
- A *whitespace escape* is one of the characters `U+006E` (`n`), `U+0072` (`r`), or `U+0074` (`t`), denoting the unicode values `U+000A` (LF), `U+000D` (CR) or `U+0009` (HT) respectively.
- The *backslash escape* is the character `U+005C` (\) which must be escaped in order to denote *itself*.

Raw string literals do not process any escapes. They start with the character `U+0072` (`r`), followed zero or more of the character `U+0023` (`#`) and a `U+0022` (double-quote) character. The *raw string body* is not defined in the EBNF grammar above: it can contain any sequence of Unicode characters and is terminated only by another `U+0022` (double-quote) character, followed by the same number of `U+0023` (`#`) characters that preceeded the opening `U+0022` (double-quote) character.

All Unicode characters contained in the raw string body represent themselves, the characters `U+0022` (double-quote) (except when followed by at least as many `U+0023` (`#`) characters as were used to start the raw string literal) or `U+005C` (\) do not have any special meaning.

Examples for string literals:

```
"foo"; r"foo";                     // foo
"\"foo\""; r#""foo""#;             // "foo"

"foo #\"# bar";
r##"foo #"# bar"##;                // foo #"# bar

"\x52"; "R"; r"R";                 // R
"\\x52"; r"\x52";                  // \x52
```

Number literals

```
num_lit : nonzero_dec [ dec_digit | '_' ] * num_suffix ?
        | '0' [       [ dec_digit | '_' ] * num_suffix ?
              | 'b'   [ '1' | '0' | '_' ] + int_suffix ?
              | 'o'   [ oct_digit | '_' ] + int_suffix ?
              | 'x'   [ hex_digit | '_' ] + int_suffix ? ] ;

num_suffix : int_suffix | float_suffix ;

int_suffix : 'u' int_suffix_size ?
           | 'i' int_suffix_size ? ;
int_suffix_size : [ '8' | '1' '6' | '3' '2' | '6' '4' ] ;

float_suffix : [ exponent | '.' dec_lit exponent ? ] ? float_suffix_ty ? ;
float_suffix_ty : 'f' [ '3' '2' | '6' '4' ] ;
exponent : ['E' | 'e'] ['-' | '+' ] ? dec_lit ;
dec_lit : [ dec_digit | '_' ] + ;
```

A *number literal* is either an *integer literal* or a *floating-point literal.* The grammar for recognizing the two kinds of literals is mixed, as they are differentiated by suffixes.

Integer literals An *integer literal* has one of four forms:

- A *decimal literal* starts with a *decimal digit* and continues with any mixture of *decimal digits* and *underscores*.
- A *hex literal* starts with the character sequence `U+0030 U+0078` (`0x`) and continues as any mixture hex digits and underscores.
- An *octal literal* starts with the character sequence `U+0030 U+006F` (`0o`) and continues as any mixture octal digits and underscores.
- A *binary literal* starts with the character sequence `U+0030 U+0062` (`0b`) and continues as any mixture binary digits and underscores.

An integer literal may be followed (immediately, without any spaces) by an *integer suffix*, which changes the type of the literal. There are two kinds of integer literal suffix:

- The `i` and `u` suffixes give the literal type `int` or `uint`, respectively.
- Each of the signed and unsigned machine types `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64` and `i64` give the literal the corresponding machine type.

The type of an *unsuffixed* integer literal is determined by type inference. If a integer type can be *uniquely* determined from the surrounding program context, the unsuffixed integer literal has that type. If the program context underconstrains the type, the unsuffixed integer literal's type is `int`; if the program context overconstrains the type, it is considered a static type error.

Examples of integer literals of various forms:

```
123; 0xff00;                       // type determined by program context
                                   // defaults to int in absence of type
                                   // information

123u;                              // type uint
123_u;                             // type uint
0xff_u8;                           // type u8
0o70_i16;                          // type i16
0b1111_1111_1001_0000_i32;         // type i32
```

Floating-point literals A *floating-point literal* has one of two forms:

- Two *decimal literals* separated by a period character `U+002E` (`.`), with an optional *exponent* trailing after the second decimal literal.

- A single *decimal literal* followed by an *exponent*.

By default, a floating-point literal has a generic type, but will fall back to `f64`. A floating-point literal may be followed (immediately, without any spaces) by a *floating-point suffix*, which changes the type of the literal. There are two floating-point suffixes: `f32`, and `f64` (the 32-bit and 64-bit floating point types).

Examples of floating-point literals of various forms:

```
123.0;                              // type f64
0.1;                                // type f64
0.1f32;                             // type f32
12E+99_f64;                         // type f64
```

Unit and boolean literals The *unit value*, the only value of the type that has the same name, is written as `()`. The two values of the boolean type are written `true` and `false`.

3.5.3 Symbols

```
symbol : "::" "->"
       | '#' | '[' | ']' | '(' | ')' | '{' | '}'
       | ',' | ';' ;
```

Symbols are a general class of printable token that play structural roles in a variety of grammar productions. They are catalogued here for completeness as the set of remaining miscellaneous printable tokens that do not otherwise appear as unary operators, binary operators, or keywords.

3.6 Paths

```
expr_path : ident [ "::" expr_path_tail ] + ;
expr_path_tail : '<' type_expr [ ',' type_expr ] + '>'
               | expr_path ;

type_path : ident [ type_path_tail ] + ;
type_path_tail : '<' type_expr [ ',' type_expr ] + '>'
               | "::" type_path ;
```

A *path* is a sequence of one or more path components *logically* separated by a namespace qualifier (`::`). If a path consists of only one component, it may refer to either an item or a slot in a local control scope. If a path has multiple components, it refers to an item.

Every item has a *canonical path* within its crate, but the path naming an item is only meaningful within a given crate. There is no global namespace across crates; an item's canonical path merely identifies it within the crate.

Two examples of simple paths consisting of only identifier components:

```
x;
x::y::z;
```

Path components are usually identifiers, but the trailing component of a path may be an angle-bracket-enclosed list of type arguments. In expression context, the type argument list is given after a final (`::`) namespace qualifier in order to disambiguate it from a relational expression involving the less-than symbol (`<`). In type expression context, the final namespace qualifier is omitted.

Two examples of paths with type arguments:

```
# struct HashMap<K, V>;
# fn f() {
# fn id<T>(t: T) -> T { t }
type T = HashMap<int,~str>;  // Type arguments used in a type expression
let x = id::<int>(10);         // Type arguments used in a call expression
# }
```

4 Syntax extensions

A number of minor features of Rust are not central enough to have their own syntax, and yet are not implementable as functions. Instead, they are given names, and invoked through a consistent syntax: `name!(...)`. Examples include:

- `format!` : format data into a string
- `env!` : look up an environment variable's value at compile time
- `file!`: return the path to the file being compiled
- `stringify!` : pretty-print the Rust expression given as an argument
- `include!` : include the Rust expression in the given file
- `include_str!` : include the contents of the given file as a string
- `include_bin!` : include the contents of the given file as a binary blob
- `error!`, `warn!`, `info!`, `debug!` : provide diagnostic information.

All of the above extensions are expressions with values.

4.1 Macros

```
expr_macro_rules : "macro_rules" '!' ident '(' macro_rule * ')' ;
macro_rule : '(' matcher * ')' "=>" '(' transcriber * ')' ';' ;
matcher : '(' matcher * ')' | '[' matcher * ']'
        | '{' matcher * '}' | '$' ident ':' ident
        | '$' '(' matcher * ')' sep_token? [ '*' | '+' ]
        | non_special_token ;
transcriber : '(' transcriber * ')' | '[' transcriber * ']'
            | '{' transcriber * '}' | '$' ident
            | '$' '(' transcriber * ')' sep_token? [ '*' | '+' ]
            | non_special_token ;
```

User-defined syntax extensions are called "macros", and the `macro_rules` syntax extension defines them. Currently, user-defined macros can expand to expressions, statements, or items.

(A `sep_token` is any token other than `*` and `+`. A `non_special_token` is any token other than a delimiter or `$`.)

The macro expander looks up macro invocations by name, and tries each macro rule in turn. It transcribes the first successful match. Matching and transcription are closely related to each other, and we will describe them together.

4.1.1 Macro By Example

The macro expander matches and transcribes every token that does not begin with a `$` literally, including delimiters. For parsing reasons, delimiters must be balanced, but they are otherwise not special.

In the matcher, `$` *name* `:` *designator* matches the nonterminal in the Rust syntax named by *designator*. Valid designators are `item`, `block`, `stmt`, `pat`, `expr`, `ty` (type), `ident`, `path`, `matchers` (lhs of the `=>` in macro rules), `tt` (rhs of the `=>` in macro rules). In the transcriber, the designator is already known, and so only the name of a matched nonterminal comes after the dollar sign.

In both the matcher and transcriber, the Kleene star-like operator indicates repetition. The Kleene star operator consists of `$` and parens, optionally followed by a separator token, followed by `*` or `+`. `*` means zero or more repetitions, `+` means at least one repetition. The parens are not matched or transcribed. On the matcher side, a name is bound to *all* of the names it matches, in a structure that mimics the structure of the repetition encountered on a successful match. The job of the transcriber is to sort that structure out.

The rules for transcription of these repetitions are called "Macro By Example". Essentially, one "layer" of repetition is discharged at a time, and all of them must be discharged by the time a name is transcribed. Therefore, `( $(`

`$i:ident ),* ) => ( $i )` is an invalid macro, but `( $( $i:ident ),* ) => ( $( $i:ident ),* )` is acceptable (if trivial).

When Macro By Example encounters a repetition, it examines all of the `$` *name* s that occur in its body. At the "current layer", they all must repeat the same number of times, so `( $( $i:ident ),* ; $( $j:ident ),* ) => ( $( ($i,$j) ),* )` is valid if given the argument `(a,b,c ; d,e,f)`, but not `(a,b,c ; d,e)`. The repetition walks through the choices at that layer in lockstep, so the former input transcribes to `( (a,d), (b,e), (c,f) )`.

Nested repetitions are allowed.

4.1.2 Parsing limitations

The parser used by the macro system is reasonably powerful, but the parsing of Rust syntax is restricted in two ways:

1. The parser will always parse as much as possible. If it attempts to match `$i:expr [ , ]` against `8 [ , ]`, it will attempt to parse `i` as an array index operation and fail. Adding a separator can solve this problem.
2. The parser must have eliminated all ambiguity by the time it reaches a `$` *name* `:` *designator*. This requirement most often affects name-designator pairs when they occur at the beginning of, or immediately after, a `$(...)*`; requiring a distinctive token in front can solve the problem.

4.2 Syntax extensions useful for the macro author

- `log_syntax!` : print out the arguments at compile time
- `trace_macros!` : supply `true` or `false` to enable or disable macro expansion logging
- `stringify!` : turn the identifier argument into a string literal
- `concat!` : concatenates a comma-separated list of literals
- `concat_idents!` : create a new identifier by concatenating the arguments

5 Crates and source files

Rust is a *compiled* language. Its semantics obey a *phase distinction* between compile-time and run-time. Those semantic rules that have a *static interpretation* govern the success or failure of compilation. We refer to these rules as "static semantics". Semantic rules called "dynamic semantics" govern the behavior of programs at run-time. A program that fails to compile due to violation of a compile-time rule has no defined dynamic semantics; the compiler should halt with an error report, and produce no executable artifact.

The compilation model centres on artifacts called *crates*. Each compilation processes a single crate in source form, and if successful, produces a single crate in binary form: either an executable or a library.[2]

A *crate* is a unit of compilation and linking, as well as versioning, distribution and runtime loading. A crate contains a *tree* of nested module scopes. The top level of this tree is a module that is anonymous (from the point of view of paths within the module) and any item within a crate has a canonical module path denoting its location within the crate's module tree.

The Rust compiler is always invoked with a single source file as input, and always produces a single output crate. The processing of that source file may result in other source files being loaded as modules. Source files have the extension `.rs`.

A Rust source file describes a module, the name and location of which – in the module tree of the current crate – are defined from outside the source file: either by an explicit `mod_item` in a referencing source file, or by the name of the crate itself.

Each source file contains a sequence of zero or more `item` definitions, and may optionally begin with any number of `attributes` that apply to the containing module. Attributes on the anonymous crate module define important metadata that influences the behavior of the compiler.

```
// Package ID
#[ crate_id = "projx#2.5" ];

// Additional metadata attributes
#[ desc = "Project X" ];
#[ license = "BSD" ];
#[ comment = "This is a comment on Project X." ];

// Specify the output type
#[ crate_type = "lib" ];

// Turn on a warning
#[ warn(non_camel_case_types) ];
```

A crate that contains a `main` function can be compiled to an executable. If a `main` function is present, its return type must be `unit` and it must take no arguments.

[2]A crate is somewhat analogous to an *assembly* in the ECMA-335 CLI model, a *library* in the SML/NJ Compilation Manager, a *unit* in the Owens and Flatt module system, or a *configuration* in Mesa.

6 Items and attributes

Crates contain items, each of which may have some number of attributes attached to it.

6.1 Items

```
item : mod_item | fn_item | type_item | struct_item | enum_item
     | static_item | trait_item | impl_item | extern_block ;
```

An *item* is a component of a crate; some module items can be defined in crate files, but most are defined in source files. Items are organized within a crate by a nested set of modules. Every crate has a single "outermost" anonymous module; all further items within the crate have paths within the module tree of the crate.

Items are entirely determined at compile-time, generally remain fixed during execution, and may reside in read-only memory.

There are several kinds of item:

- modules
- functions
- type definitions
- structures
- enumerations
- static items
- traits
- implementations

Some items form an implicit scope for the declaration of sub-items. In other words, within a function or module, declarations of items can (in many cases) be mixed with the statements, control blocks, and similar artifacts that otherwise compose the item body. The meaning of these scoped items is the same as if the item was declared outside the scope – it is still a static item – except that the item's *path name* within the module namespace is qualified by the name of the enclosing item, or is private to the enclosing item (in the case of functions). The grammar specifies the exact locations in which sub-item declarations may appear.

6.1.1 Type Parameters

All items except modules may be *parameterized* by type. Type parameters are given as a comma-separated list of identifiers enclosed in angle brackets (`<...>`), after the name of the item and before its definition. The type parameters of an item are considered "part of the name", not part of the type of the item. A referencing path must (in principle) provide type arguments as a list of comma-separated types enclosed within angle brackets, in order to refer to the type-parameterized item. In practice, the type-inference system can usually infer such argument types from context. There are no general type-parametric types, only type-parametric items. That is, Rust has no notion of type abstraction: there are no first-class "forall" types.

6.1.2 Modules

```
mod_item : "mod" ident ( ';' | '{' mod '}' );
mod : [ view_item | item ] * ;
```

A module is a container for zero or more view items and zero or more items. The view items manage the visibility of the items defined within the module, as well as the visibility of names from outside the module when referenced from inside the module.

A *module item* is a module, surrounded in braces, named, and prefixed with the keyword `mod`. A module item introduces a new, named module into the tree of modules making up a crate. Modules can nest arbitrarily.

An example of a module:

```
mod math {
    type Complex = (f64, f64);
    fn sin(f: f64) -> f64 {
        /* ... */
# fail!();
    }
    fn cos(f: f64) -> f64 {
        /* ... */
# fail!();
    }
    fn tan(f: f64) -> f64 {
        /* ... */
# fail!();
    }
}
```

Modules and types share the same namespace. Declaring a named type that has the same name as a module in scope is forbidden: that is, a type definition, trait, struct, enumeration, or type parameter can't shadow the name of a module in scope, or vice versa.

A module without a body is loaded from an external file, by default with the same name as the module, plus the `.rs` extension. When a nested submodule is loaded from an external file, it is loaded from a subdirectory path that mirrors the module hierarchy.

```
// Load the ‘vec‘ module from ‘vec.rs‘
mod vec;

mod task {
    // Load the ‘local_data‘ module from ‘task/local_data.rs‘
    mod local_data;
}
```

The directories and files used for loading external file modules can be influenced with the `path` attribute.

```
#[path = "task_files"]
mod task {
    // Load the ‘local_data‘ module from ‘task_files/tls.rs‘
    #[path = "tls.rs"]
    mod local_data;
}
```

View items

```
view_item : extern_crate_decl | use_decl ;
```

A view item manages the namespace of a module. View items do not define new items, but rather, simply change other items' visibility. There are several kinds of view item:

- `extern crate` declarations
- `use` declarations

Extern crate declarations

```
extern_crate_decl : "extern" "crate" ident [ '(' link_attrs ')' ] ? [ '=' string_lit ] ? ;
link_attrs : link_attr [ ',' link_attrs ] + ;
link_attr : ident '=' literal ;
```

An *`extern crate` declaration* specifies a dependency on an external crate. The external crate is then bound into the declaring scope as the `ident` provided in the `extern_crate_decl`.

The external crate is resolved to a specific `soname` at compile time, and a runtime linkage requirement to that `soname` is passed to the linker for loading at runtime. The `soname` is resolved at compile time by scanning the compiler's library path and matching the optional `crateid` provided as a string literal against the `crateid` attributes that were declared on the external crate when it was compiled. If no `crateid` is provided, a default `name` attribute is assumed, equal to the `ident` given in the `extern_crate_decl`.

Four examples of `extern crate` declarations:

```
extern crate pcre;

extern crate std; // equivalent to: extern crate std = "std";

extern crate ruststd = "std"; // linking to 'std' under another name

extern crate foo = "some/where/rust-foo#foo:1.0"; // a full package ID for external tools
```

Use declarations

```
use_decl : "pub" ? "use" ident [ '=' path
                          | "::" path_glob ] ;

path_glob : ident [ "::" path_glob ] ?
          | '*'
          | '{' ident [ ',' ident ] * '}' ;
```

A *use declaration* creates one or more local name bindings synonymous with some other path. Usually a `use` declaration is used to shorten the path required to refer to a module item. These declarations may appear at the top of modules and blocks.

Note: Unlike in many languages, `use` declarations in Rust do *not* declare linkage dependency with external crates. Rather, `extern crate` declarations declare linkage dependencies.

Use declarations support a number of convenient shortcuts:

- Rebinding the target name as a new local name, using the syntax `use x = p::q::r;`.
- Simultaneously binding a list of paths differing only in their final element, using the glob-like brace syntax `use a::b::{c,d,e,f};`

- Binding all paths matching a given prefix, using the asterisk wildcard syntax `use a::b::*;`

An example of `use` declarations:

```
use std::num::sin;
use std::option::{Some, None};

# fn foo<T>(_: T){}

fn main() {
    // Equivalent to 'std::num::sin(1.0);'
    sin(1.0);

    // Equivalent to 'foo(~[std::option::Some(1.0), std::option::None]);'
    foo(~[Some(1.0), None]);
}
```

Like items, `use` declarations are private to the containing module, by default. Also like items, a `use` declaration can be public, if qualified by the `pub` keyword. Such a `use` declaration serves to *re-export* a name. A public `use` declaration can therefore *redirect* some public name to a different target definition: even a definition with a private canonical path, inside a different module. If a sequence of such redirections form a cycle or cannot be resolved unambiguously, they represent a compile-time error.

An example of re-exporting:

```
# fn main() { }
mod quux {
    pub use quux::foo::*;

    pub mod foo {
        pub fn bar() { }
        pub fn baz() { }
    }
}
```

In this example, the module `quux` re-exports all of the public names defined in `foo`.

Also note that the paths contained in `use` items are relative to the crate root. So, in the previous example, the `use` refers to `quux::foo::*`, and not simply to `foo::*`. This also means that top-level module declarations should be at the crate root if direct usage of the declared modules within `use` items is desired. It is also possible to use `self` and `super` at the beginning of a `use` item to refer to the current and direct parent modules respectively. All rules regarding accessing declared modules in `use` declarations applies to both module declarations and `extern crate` declarations.

An example of what will and will not work for `use` items:

```
# #[allow(unused_imports)];
use foo::native::start;  // good: foo is at the root of the crate
use foo::baz::foobaz;    // good: foo is at the root of the crate

mod foo {
    extern crate native;

    use foo::native::start; // good: foo is at crate root
//  use native::start;      // bad:  native is not at the crate root
    use self::baz::foobaz;  // good: self refers to module 'foo'
    use foo::bar::foobar;   // good: foo is at crate root

    pub mod bar {
        pub fn foobar() { }
    }

    pub mod baz {
        use super::bar::foobar; // good: super refers to module 'foo'
        pub fn foobaz() { }
    }
}

fn main() {}
```

6.1.3 Functions

A *function item* defines a sequence of statements and an optional final expression, along with a name and a set of parameters. Functions are declared with the keyword `fn`. Functions declare a set of *input slots* as parameters, through which the caller passes arguments into the function, and an *output slot* through which the function passes results back to the caller.

A function may also be copied into a first class *value*, in which case the value has the corresponding *function type*, and can be used otherwise exactly as a function item (with a minor additional cost of calling the function indirectly).

Every control path in a function logically ends with a `return` expression or a diverging expression. If the outermost block of a function has a value-producing expression in its final-expression position, that expression is interpreted as an implicit `return` expression applied to the final-expression.

An example of a function:

```
fn add(x: int, y: int) -> int {
    return x + y;
}
```

As with `let` bindings, function arguments are irrefutable patterns, so any pattern that is valid in a let binding is also valid as an argument.

```
fn first((value, _): (int, int)) -> int { value }
```

Generic functions A *generic function* allows one or more *parameterized types* to appear in its signature. Each type parameter must be explicitly declared, in an angle-bracket-enclosed, comma-separated list following the function name.

```
fn iter<T>(seq: &[T], f: |T|) {
    for elt in seq.iter() { f(elt); }
}
fn map<T, U>(seq: &[T], f: |T| -> U) -> ~[U] {
    let mut acc = ~[];
    for elt in seq.iter() { acc.push(f(elt)); }
    acc
}
```

Inside the function signature and body, the name of the type parameter can be used as a type name.

When a generic function is referenced, its type is instantiated based on the context of the reference. For example, calling the `iter` function defined above on `[1, 2]` will instantiate type parameter `T` with `int`, and require the closure parameter to have type `fn(int)`.

The type parameters can also be explicitly supplied in a trailing path component after the function name. This might be necessary if there is not sufficient context to determine the type parameters. For example, `mem::size_of::<u32>() == 4`.

Since a parameter type is opaque to the generic function, the set of operations that can be performed on it is limited. Values of parameter type can only be moved, not copied.

```
fn id<T>(x: T) -> T { x }
```

Similarly, trait bounds can be specified for type parameters to allow methods with that trait to be called on values of that type.

Unsafety Unsafe operations are those that potentially violate the memory-safety guarantees of Rust's static semantics.

The following language level features cannot be used in the safe subset of Rust:

- Dereferencing a raw pointer.
- Calling an unsafe function (including an intrinsic or foreign function).

Unsafe functions Unsafe functions are functions that are not safe in all contexts and/or for all possible inputs. Such a function must be prefixed with the keyword `unsafe`.

Unsafe blocks A block of code can also be prefixed with the `unsafe` keyword, to permit calling `unsafe` functions or dereferencing raw pointers within a safe function.

When a programmer has sufficient conviction that a sequence of potentially unsafe operations is actually safe, they can encapsulate that sequence (taken as a whole) within an `unsafe` block. The compiler will consider uses of such code safe, in the surrounding context.

Unsafe blocks are used to wrap foreign libraries, make direct use of hardware or implement features not directly present in the language. For example, Rust provides the language features necessary to implement memory-safe concurrency in the language but the implementation of tasks and message passing is in the standard library.

Rust's type system is a conservative approximation of the dynamic safety requirements, so in some cases there is a performance cost to using safe code. For example, a doubly-linked list is not a tree structure and can only be represented with managed or reference-counted pointers in safe code. By using `unsafe` blocks to represent the reverse links as raw pointers, it can be implemented with only owned pointers.

Behavior considered unsafe This is a list of behavior which is forbidden in all Rust code. Type checking provides the guarantee that these issues are never caused by safe code. An `unsafe` block or function is responsible for never invoking this behaviour or exposing an API making it possible for it to occur in safe code.

- Data races
- Dereferencing a null/dangling raw pointer
- Mutating an immutable value/reference

- Reads of undef (uninitialized) memory
- Breaking the pointer aliasing rules with raw pointers (a subset of the rules used by C)
- Invoking undefined behavior via compiler intrinsics:
  - Indexing outside of the bounds of an object with `std::ptr::offset` (`offset` intrinsic), with the exception of one byte past the end which is permitted.
  - Using `std::ptr::copy_nonoverlapping_memory` (`memcpy32`/`memcpy64` instrinsics) on overlapping buffers
- Invalid values in primitive types, even in private fields/locals:
  - Dangling/null pointers in non-raw pointers, or slices
  - A value other than `false` (0) or `true` (1) in a `bool`
  - A discriminant in an `enum` not included in the type definition
  - A value in a `char` which is a surrogate or above `char::MAX`
  - non-UTF-8 byte sequences in a `str`

Behaviour not considered unsafe This is a list of behaviour not considered *unsafe* in Rust terms, but that may be undesired.

- Deadlocks
- Reading data from private fields (`std::repr`, `format!("{:?}", x)`)
- Leaks due to reference count cycles, even in the global heap
- Exiting without calling destructors
- Sending signals
- Accessing/modifying the file system
- Unsigned integer overflow (well-defined as wrapping)
- Signed integer overflow (well-defined as two's complement representation wrapping)

Diverging functions A special kind of function can be declared with a `!` character where the output slot type would normally be. For example:

```
fn my_err(s: &str) -> ! {
    println!("{}", s);
    fail!();
}
```

We call such functions "diverging" because they never return a value to the caller. Every control path in a diverging function must end with a `fail!()` or a call to another diverging function on every control path. The `!` annotation does *not* denote a type. Rather, the result type of a diverging function is a special type called ⊥ ("bottom") that unifies with any type. Rust has no syntax for ⊥.

It might be necessary to declare a diverging function because as mentioned previously, the typechecker checks that every control path in a function ends with a `return` or diverging expression. So, if `my_err` were declared without the `!` annotation, the following code would not typecheck:

```
# fn my_err(s: &str) -> ! { fail!() }

fn f(i: int) -> int {
   if i == 42 {
     return 42;
   }
   else {
     my_err("Bad number!");
   }
}
```

This will not compile without the `!` annotation on `my_err`, since the `else` branch of the conditional in `f` does not return an `int`, as required by the signature of `f`. Adding the `!` annotation to `my_err` informs the typechecker that, should control ever enter `my_err`, no further type judgments about `f` need to hold, since control will never resume in any context that relies on those judgments. Thus the return type on `f` only needs to reflect the `if` branch of the conditional.

Extern functions Extern functions are part of Rust's foreign function interface, providing the opposite functionality to external blocks. Whereas external blocks allow Rust code to call foreign code, extern functions with bodies defined in Rust code *can be called by foreign code*. They are defined in the same way as any other Rust function, except that they have the `extern` modifier.

```
// Declares an extern fn, the ABI defaults to "C"
```

```
extern fn new_vec() -> ~[int] { ~[] }

// Declares an extern fn with "stdcall" ABI
extern "stdcall" fn new_vec_stdcall() -> ~[int] { ~[] }
```

Unlike normal functions, extern fns have an `extern "ABI" fn()`. This is the same type as the functions declared in an extern block.

```
# extern fn new_vec() -> ~[int] { ~[] }
let fptr: extern "C" fn() -> ~[int] = new_vec;
```

Extern functions may be called directly from Rust code as Rust uses large, contiguous stack segments like C.

6.1.4 Type definitions

A *type definition* defines a new name for an existing type. Type definitions are declared with the keyword `type`. Every value has a single, specific type; the type-specified aspects of a value include:

- Whether the value is composed of sub-values or is indivisible.
- Whether the value represents textual or numerical information.
- Whether the value represents integral or floating-point information.
- The sequence of memory operations required to access the value.
- The kind of the type.

For example, the type `(u8, u8)` defines the set of immutable values that are composite pairs, each containing two unsigned 8-bit integers accessed by pattern-matching and laid out in memory with the `x` component preceding the `y` component.

6.1.5 Structures

A *structure* is a nominal structure type defined with the keyword `struct`.

An example of a `struct` item and its use:

```
struct Point {x: int, y: int}
let p = Point {x: 10, y: 11};
let px: int = p.x;
```

A *tuple structure* is a nominal tuple type, also defined with the keyword `struct`. For example:

```
struct Point(int, int);
let p = Point(10, 11);
let px: int = match p { Point(x, _) => x };
```

A *unit-like struct* is a structure without any fields, defined by leaving off the list of fields entirely. Such types will have a single value, just like the unit value `()` of the unit type. For example:

```
struct Cookie;
let c = [Cookie, Cookie, Cookie, Cookie];
```

6.1.6 Enumerations

An *enumeration* is a simultaneous definition of a nominal enumerated type as well as a set of *constructors*, that can be used to create or pattern-match values of the corresponding enumerated type.

Enumerations are declared with the keyword `enum`.

An example of an `enum` item and its use:

```
enum Animal {
  Dog,
  Cat
}

let mut a: Animal = Dog;
a = Cat;
```

Enumeration constructors can have either named or unnamed fields:

```
enum Animal {
    Dog (~str, f64),
    Cat { name: ~str, weight: f64 }
}

let mut a: Animal = Dog(~"Cocoa", 37.2);
a = Cat{ name: ~"Spotty", weight: 2.7 };
```

In this example, `Cat` is a *struct-like enum variant*, whereas `Dog` is simply called an enum variant.

6.1.7 Static items

```
static_item : "static" ident ':' type '=' expr ';' ;
```

A *static item* is a named *constant value* stored in the global data section of a crate. Immutable static items are stored in the read-only data section. The constant value bound to a static item is, like all constant values, evaluated at compile time. Static items have the `static` lifetime, which outlives all other lifetimes in a Rust program. Static items are declared with the `static` keyword. A static item must have a *constant expression* giving its definition.

Static items must be explicitly typed. The type may be `bool`, `char`, a number, or a type derived from those primitive types. The derived types are references with the `static` lifetime, fixed-size arrays, tuples, and structs.

```
static BIT1: uint = 1 << 0;
static BIT2: uint = 1 << 1;

static BITS: [uint, ..2] = [BIT1, BIT2];
static STRING: &'static str = "bitstring";

struct BitsNStrings<'a> {
    mybits: [uint, ..2],
    mystring: &'a str
}

static bits_n_strings: BitsNStrings<'static> = BitsNStrings {
    mybits: BITS,
    mystring: STRING
};
```

Mutable statics If a static item is declared with the `mut` keyword, then it is allowed to be modified by the program. One of Rust's goals is to make concurrency bugs hard to run into, and this is obviously a very large source of race conditions or other bugs. For this reason, an `unsafe` block is required when either reading or writing a mutable static variable. Care should be taken to ensure that modifications to a mutable static are safe with respect to other tasks running in the same process.

Mutable statics are still very useful, however. They can be used with C libraries and can also be bound from C libraries (in an `extern` block).

```
# fn atomic_add(_: &mut uint, _: uint) -> uint { 2 }

static mut LEVELS: uint = 0;
```

```
// This violates the idea of no shared state, and this doesn't internally
// protect against races, so this function is `unsafe`
unsafe fn bump_levels_unsafe1() -> uint {
    let ret = LEVELS;
    LEVELS += 1;
    return ret;
}

// Assuming that we have an atomic_add function which returns the old value,
// this function is "safe" but the meaning of the return value may not be what
// callers expect, so it's still marked as `unsafe`
unsafe fn bump_levels_unsafe2() -> uint {
    return atomic_add(&mut LEVELS, 1);
}
```

6.1.8 Traits

A *trait* describes a set of method types.

Traits can include default implementations of methods, written in terms of some unknown `self` type; the `self` type may either be completely unspecified, or constrained by some other trait.

Traits are implemented for specific types through separate implementations.

```
# type Surface = int;
# type BoundingBox = int;

trait Shape {
    fn draw(&self, Surface);
    fn bounding_box(&self) -> BoundingBox;
}
```

This defines a trait with two methods. All values that have implementations of this trait in scope can have their `draw` and `bounding_box` methods called, using `value.bounding_box()` syntax.

Type parameters can be specified for a trait to make it generic. These appear after the trait name, using the same syntax used in generic functions.

```
trait Seq<T> {
   fn len(&self) -> uint;
   fn elt_at(&self, n: uint) -> T;
   fn iter(&self, |T|);
}
```

Generic functions may use traits as *bounds* on their type parameters. This will have two effects: only types that have the trait may instantiate the parameter, and within the generic function, the methods of the trait can be called on values that have the parameter's type. For example:

```
# type Surface = int;
# trait Shape { fn draw(&self, Surface); }

fn draw_twice<T: Shape>(surface: Surface, sh: T) {
    sh.draw(surface);
    sh.draw(surface);
}
```

Traits also define an object type with the same name as the trait. Values of this type are created by casting pointer values (pointing to a type for which an implementation of the given trait is in scope) to pointers to the trait name, used as a type.

```
# trait Shape { }
# impl Shape for int { }
# let mycircle = 0;

let myshape: ~Shape = ~mycircle as ~Shape;
```

The resulting value is a managed box containing the value that was cast, along with information that identifies the methods of the implementation that was used. Values with a trait type can have methods called on them, for any method in the trait, and can be used to instantiate type parameters that are bounded by the trait.

Trait methods may be static, which means that they lack a `self` argument. This means that they can only be called with function call syntax (`f(x)`) and not method call syntax (`obj.f()`). The way to refer to the name of a static method is to qualify it with the trait name, treating the trait name like a module. For example:

```
trait Num {
    fn from_int(n: int) -> Self;
}
impl Num for f64 {
    fn from_int(n: int) -> f64 { n as f64 }
}
let x: f64 = Num::from_int(42);
```

Traits may inherit from other traits. For example, in

```
trait Shape { fn area() -> f64; }
trait Circle : Shape { fn radius() -> f64; }
```

the syntax `Circle : Shape` means that types that implement `Circle` must also have an implementation for `Shape`. Multiple supertraits are separated by spaces, `trait Circle : Shape Eq { }`. In an implementation of `Circle` for a given type `T`, methods can refer to `Shape` methods, since the typechecker checks that any type with an implementation of `Circle` also has an implementation of `Shape`.

In type-parameterized functions, methods of the supertrait may be called on values of subtrait-bound type parameters. Referring to the previous example of `trait Circle : Shape`:

```
# trait Shape { fn area(&self) -> f64; }
# trait Circle : Shape { fn radius(&self) -> f64; }
fn radius_times_area<T: Circle>(c: T) -> f64 {
    // ‘c‘ is both a Circle and a Shape
    c.radius() * c.area()
}
```

Likewise, supertrait methods may also be called on trait objects.

```
# trait Shape { fn area(&self) -> f64; }
# trait Circle : Shape { fn radius(&self) -> f64; }
# impl Shape for int { fn area(&self) -> f64 { 0.0 } }
# impl Circle for int { fn radius(&self) -> f64 { 0.0 } }
# let mycircle = 0;

let mycircle: Circle = ~mycircle as ~Circle;
let nonsense = mycircle.radius() * mycircle.area();
```

6.1.9 Implementations

An *implementation* is an item that implements a trait for a specific type.

Implementations are defined with the keyword `impl`.

```
# struct Point {x: f64, y: f64};
# type Surface = int;
# struct BoundingBox {x: f64, y: f64, width: f64, height: f64};
# trait Shape { fn draw(&self, Surface); fn bounding_box(&self) -> BoundingBox; }
# fn do_draw_circle(s: Surface, c: Circle) { }

struct Circle {
    radius: f64,
    center: Point,
}

impl Shape for Circle {
    fn draw(&self, s: Surface) { do_draw_circle(s, *self); }
    fn bounding_box(&self) -> BoundingBox {
        let r = self.radius;
        BoundingBox{x: self.center.x - r, y: self.center.y - r,
         width: 2.0 * r, height: 2.0 * r}
    }
}
```

It is possible to define an implementation without referring to a trait. The methods in such an implementation can only be used as direct calls on the values of the type that the implementation targets. In such an implementation, the trait type and `for` after `impl` are omitted. Such implementations are limited to nominal types (enums, structs), and the implementation must appear in the same module or a sub-module as the `self` type.

When a trait *is* specified in an `impl`, all methods declared as part of the trait must be implemented, with matching types and type parameter counts.

An implementation can take type parameters, which can be different from the type parameters taken by the trait it implements. Implementation parameters are written after the `impl` keyword.

```
# trait Seq<T> { }

impl<T> Seq<T> for ~[T] {
   /* ... */
}
impl Seq<bool> for u32 {
   /* Treat the integer as a sequence of bits */
}
```

6.1.10 External blocks

```
extern_block_item : "extern" '{' extern_block '}' ;
extern_block : [ foreign_fn ] * ;
```

External blocks form the basis for Rust's foreign function interface. Declarations in an external block describe symbols in external, non-Rust libraries.

Functions within external blocks are declared in the same way as other Rust functions, with the exception that they may not have a body and are instead terminated by a semicolon.

```
# use std::libc::{c_char, FILE};

extern {
    fn fopen(filename: *c_char, mode: *c_char) -> *FILE;
}
```

Functions within external blocks may be called by Rust code, just like functions defined in Rust. The Rust compiler automatically translates between the Rust ABI and the foreign ABI.

A number of attributes control the behavior of external blocks.

By default external blocks assume that the library they are calling uses the standard C "cdecl" ABI. Other ABIs may be specified using an `abi` string, as shown here:

```
// Interface to the Windows API
extern "stdcall" { }
```

The `link` attribute allows the name of the library to be specified. When specified the compiler will attempt to link against the native library of the specified name.

```
#[link(name = "crypto")]
extern { }
```

The type of a function declared in an extern block is `extern "abi" fn(A1, ..., An) -> R`, where `A1...An` are the declared types of its arguments and `R` is the decalred return type.

6.2 Visibility and Privacy

These two terms are often used interchangeably, and what they are attempting to convey is the answer to the question "Can this item be used at this location?"

Rust's name resolution operates on a global hierarchy of namespaces. Each level in the hierarchy can be thought of as some item. The items are one of those mentioned above, but also include external crates. Declaring or defining a new module can be thought of as inserting a new tree into the hierarchy at the location of the definition.

To control whether interfaces can be used across modules, Rust checks each use of an item to see whether it should be allowed or not. This is where privacy warnings are generated, or otherwise "you used a private item of another module and weren't allowed to."

By default, everything in rust is *private*, with two exceptions. The first exception is that struct fields are public by default (but the struct itself is still private by default), and the remaining exception is that enum variants in a `pub` enum are the default visibility of the enum container itself.. You are allowed to alter this default visibility with the `pub` keyword (or `priv` keyword for struct fields and enum variants). When an item is declared as `pub`, it can be thought of as being accessible to the outside world. For example:

```
# fn main() {}
// Declare a private struct
struct Foo;

// Declare a public struct with a private field
pub struct Bar {
    priv field: int
}

// Declare a public enum with public and private variants
pub enum State {
    PubliclyAccessibleState,
    priv PrivatelyAccessibleState
}
```

With the notion of an item being either public or private, Rust allows item accesses in two cases:

1. If an item is public, then it can be used externally through any of its public ancestors.
2. If an item is private, it may be accessed by the current module and its descendants.

These two cases are surprisingly powerful for creating module hierarchies exposing public APIs while hiding internal implementation details. To help explain, here's a few use cases and what they would entail.

- A library developer needs to expose functionality to crates which link against their library. As a consequence of the first case, this means that anything which is usable externally must be `pub` from the root down to the destination item. Any private item in the chain will disallow external accesses.
- A crate needs a global available "helper module" to itself, but it doesn't want to expose the helper module as a public API. To accomplish this, the root of the crate's hierarchy would have a private module which then internally has a "public api". Because the entire crate is a descendant of the root, then the entire local crate can access this private module through the second case.

- When writing unit tests for a module, it's often a common idiom to have an immediate child of the module to-be-tested named `mod test`. This module could access any items of the parent module through the second case, meaning that internal implementation details could also be seamlessly tested from the child module.

In the second case, it mentions that a private item "can be accessed" by the current module and its descendants, but the exact meaning of accessing an item depends on what the item is. Accessing a module, for example, would mean looking inside of it (to import more items). On the other hand, accessing a function would mean that it is invoked. Additionally, path expressions and import statements are considered to access an item in the sense that the import/expression is only valid if the destination is in the current visibility scope.

Here's an example of a program which exemplifies the three cases outlined above.

```
// This module is private, meaning that no external crate can access this
// module. Because it is private at the root of this current crate, however, any
// module in the crate may access any publicly visible item in this module.
mod crate_helper_module {

    // This function can be used by anything in the current crate
    pub fn crate_helper() {}

    // This function *cannot* be used by anything else in the crate. It is not
    // publicly visible outside of the ‘crate_helper_module‘, so only this
    // current module and its descendants may access it.
    fn implementation_detail() {}
}

// This function is "public to the root" meaning that it’s available to external
// crates linking against this one.
pub fn public_api() {}

// Similarly to ’public_api’, this module is public so external crates may look
// inside of it.
pub mod submodule {
    use crate_helper_module;

    pub fn my_method() {
        // Any item in the local crate may invoke the helper module’s public
        // interface through a combination of the two rules above.
        crate_helper_module::crate_helper();
    }

    // This function is hidden to any module which is not a descendant of
    // ‘submodule‘
    fn my_implementation() {}

    #[cfg(test)]
    mod test {

        #[test]
        fn test_my_implementation() {
            // Because this module is a descendant of ‘submodule‘, it’s allowed
            // to access private items inside of ‘submodule‘ without a privacy
            // violation.
            super::my_implementation();
        }
    }
}

# fn main() {}
```

For a rust program to pass the privacy checking pass, all paths must be valid accesses given the two rules above. This includes all use statements, expressions, types, etc.

6.2.1 Re-exporting and Visibility

Rust allows publicly re-exporting items through a `pub use` directive. Because this is a public directive, this allows the item to be used in the current module through the rules above. It essentially allows public access into the re-exported item. For example, this program is valid:

```
pub use api = self::implementation;

mod implementation {
    pub fn f() {}
}

# fn main() {}
```

This means that any external crate referencing `implementation::f` would receive a privacy violation, while the path `api::f` would be allowed.

When re-exporting a private item, it can be thought of as allowing the “privacy chain” being short-circuited through the reexport instead of passing through the namespace hierarchy as it normally would.

6.2.2 Glob imports and Visibility

Currently glob imports are considered an "experimental" language feature. For sanity purpose along with helping the implementation, glob imports will only import public items from their destination, not private items.

> **Note:** This is subject to change, glob exports may be removed entirely or they could possibly import private items for a privacy error to later be issued if the item is used.

6.3 Attributes

```
attribute : '#' '[' attr_list ']' ;
attr_list : attr [ ',' attr_list ]* ;
attr : ident [ '=' literal
             | '(' attr_list ')' ] ? ;
```

Static entities in Rust – crates, modules and items – may have *attributes* applied to them. [3] An attribute is a general, free-form metadatum that is interpreted according to name, convention, and language and compiler version. Attributes may appear as any of

- A single identifier, the attribute name
- An identifier followed by the equals sign '=' and a literal, providing a key/value pair
- An identifier followed by a parenthesized list of sub-attribute arguments

Attributes terminated by a semi-colon apply to the entity that the attribute is declared within. Attributes that are not terminated by a semi-colon apply to the next entity.

An example of attributes:

```
// General metadata applied to the enclosing module or crate.
#[license = "BSD"];

// A function marked as a unit test
#[test]
fn test_foo() {
  ...
}
```

[3] Attributes in Rust are modeled on Attributes in ECMA-335, C#

```
// A conditionally-compiled module
#[cfg(target_os="linux")]
mod bar {
  ...
}

// A lint attribute used to suppress a warning/error
#[allow(non_camel_case_types)]
pub type int8_t = i8;
```

> **Note:** In future versions of Rust, user-provided extensions to the compiler will be able to interpret attributes. When this facility is provided, the compiler will distinguish between language-reserved and user-available attributes.

At present, only the Rust compiler interprets attributes, so all attribute names are effectively reserved. Some significant attributes include:

- The `doc` attribute, for documenting code in-place.
- The `cfg` attribute, for conditional-compilation by build-configuration (see Conditional compilation).
- The `crate_id` attribute, for describing the package ID of a crate.
- The `lang` attribute, for custom definitions of traits and functions that are known to the Rust compiler (see Language items).
- The `link` attribute, for describing linkage metadata for a extern blocks.
- The `test` attribute, for marking functions as unit tests.
- The `allow`, `warn`, `forbid`, and `deny` attributes, for controlling lint checks (see Lint check attributes).
- The `deriving` attribute, for automatically generating implementations of certain traits.
- The `inline` attribute, for expanding functions at caller location (see Inline attributes).
- The `static_assert` attribute, for asserting that a static bool is true at compiletime.
- The `thread_local` attribute, for defining a `static mut` as a thread-local. Note that this is only a low-level building block, and is not local to a *task*, nor does it provide safety.

Other attributes may be added or removed during development of the language.

6.3.1 Conditional compilation

Sometimes one wants to have different compiler outputs from the same code, depending on build target, such as targeted operating system, or to enable release builds.

There are two kinds of configuration options, one that is either defined or not (`#[cfg(foo)]`), and the other that contains a string that can be checked against (`#[cfg(bar = "baz")]` (currently only compiler-defined configuration options can have the latter form).

```
// The function is only included in the build when compiling for OSX
#[cfg(target_os = "macos")]
fn macos_only() {
  // ...
}

// This function is only included when either foo or bar is defined
#[cfg(foo)]
#[cfg(bar)]
fn needs_foo_or_bar() {
  // ...
}

// This function is only included when compiling for a unixish OS with a 32-bit
// architecture
#[cfg(unix, target_word_size = "32")]
fn on_32bit_unix() {
  // ...
}
```

This illustrates some conditional compilation can be achieved using the `#[cfg(...)]` attribute. Note that `#[cfg(foo, bar)]` is a condition that needs both `foo` and `bar` to be defined while `#[cfg(foo)] #[cfg(bar)]` only needs one of `foo` and `bar` to be defined (this resembles in the disjunctive normal form). Additionally, one can reverse a condition by enclosing it in a `not(...)`, like e. g. `#[cfg(not(target_os = "win32"))]`.

To pass a configuration option which triggers a `#[cfg(identifier)]` one can use `rustc --cfg identifier`. In addition to that, the following configurations are pre-defined by the compiler:

- `target_arch = "..."`. Target CPU architecture, such as `"x86"`, `"x86_64"` `"mips"`, or `"arm"`.
- `target_endian = "..."`. Endianness of the target CPU, either `"little"` or `"big"`.

- `target_family = "..."`. Operating system family of the target, e. g. `"unix"` or `"windows"`. The value of this configuration option is defined as a configuration itself, like `unix` or `windows`.
- `target_os = "..."`. Operating system of the target, examples include `"win32"`, `"macos"`, `"linux"`, `"android"` or `"freebsd"`.
- `target_word_size = "..."`. Target word size in bits. This is set to `"32"` for 32-bit CPU targets, and likewise set to `"64"` for 64-bit CPU targets.
- `test`. Only set in test builds (`rustc --test`).
- `unix`. See `target_family`.
- `windows`. See `target_family`.

6.3.2 Lint check attributes

A lint check names a potentially undesirable coding pattern, such as unreachable code or omitted documentation, for the static entity to which the attribute applies.

For any lint check `C`:

- `warn(C)` warns about violations of `C` but continues compilation,
- `deny(C)` signals an error after encountering a violation of `C`,
- `allow(C)` overrides the check for `C` so that violations will go unreported,
- `forbid(C)` is the same as `deny(C)`, but also forbids uses of `allow(C)` within the entity.

The lint checks supported by the compiler can be found via `rustc -W help`, along with their default settings.

```
mod m1 {
    // Missing documentation is ignored here
    #[allow(missing_doc)]
    pub fn undocumented_one() -> int { 1 }

    // Missing documentation signals a warning here
    #[warn(missing_doc)]
    pub fn undocumented_too() -> int { 2 }

    // Missing documentation signals an error here
    #[deny(missing_doc)]
    pub fn undocumented_end() -> int { 3 }
}
```

This example shows how one can use `allow` and `warn` to toggle a particular check on and off.

```
#[warn(missing_doc)]
mod m2{
    #[allow(missing_doc)]
    mod nested {
        // Missing documentation is ignored here
        pub fn undocumented_one() -> int { 1 }

        // Missing documentation signals a warning here,
        // despite the allow above.
        #[warn(missing_doc)]
        pub fn undocumented_two() -> int { 2 }
    }

    // Missing documentation signals a warning here
    pub fn undocumented_too() -> int { 3 }
}
```

This example shows how one can use `forbid` to disallow uses of `allow` for that lint check.

```
#[forbid(missing_doc)]
mod m3 {
    // Attempting to toggle warning signals an error here
    #[allow(missing_doc)]
    /// Returns 2.
    pub fn undocumented_too() -> int { 2 }
}
```

6.3.3 Language items

Some primitive Rust operations are defined in Rust code, rather than being implemented directly in C or assembly language. The definitions of these operations have to be easy for the compiler to find. The `lang` attribute makes it possible to declare these operations. For example, the `str` module in the Rust standard library defines the string equality function:

```
#[lang="str_eq"]
pub fn eq_slice(a: &str, b: &str) -> bool {
    // details elided
}
```

The name `str_eq` has a special meaning to the Rust compiler, and the presence of this definition means that it will use this definition when generating calls to the string equality function.

A complete list of the built-in language items follows:

Traits

`const` Cannot be mutated.

`owned` Are uniquely owned.

`durable` Contain references.

`drop` Have finalizers.

`add` Elements can be added (for example, integers and floats).

`sub` Elements can be subtracted.

`mul` Elements can be multiplied.

`div` Elements have a division operation.

`rem` Elements have a remainder operation.

`neg` Elements can be negated arithmetically.

`not` Elements can be negated logically.

`bitxor` Elements have an exclusive-or operation.

`bitand` Elements have a bitwise `and` operation.

`bitor` Elements have a bitwise `or` operation.

`shl` Elements have a left shift operation.

`shr` Elements have a right shift operation.

`index` Elements can be indexed.

`eq` Elements can be compared for equality.

`ord` Elements have a partial ordering.

Operations

`str_eq` Compare two strings for equality.

`uniq_str_eq` Compare two owned strings for equality.

`annihilate` Destroy a box before freeing it.

`log_type` Generically print a string representation of any type.

`fail_` Abort the program with an error.

`fail_bounds_check` Abort the program with a bounds check error.

`exchange_malloc` Allocate memory on the exchange heap.

`exchange_free` Free memory that was allocated on the exchange heap.

`malloc` Allocate memory on the managed heap.

`free` Free memory that was allocated on the managed heap.

`borrow_as_imm` Create an immutable reference to a mutable value.

`return_to_mut` Release a reference created with `return_to_mut`

`check_not_borrowed` Fail if a value has existing references to it.

`strdup_uniq` Return a new unique string containing a copy of the contents of a unique string.

> **Note:** This list is likely to become out of date. We should auto-generate it from `librustc/middle/lang_items.rs`.

6.3.4 Inline attributes

The inline attribute is used to suggest to the compiler to perform an inline expansion and place a copy of the function in the caller rather than generating code to call the function where it is defined.

The compiler automatically inlines functions based on internal heuristics. Incorrectly inlining functions can actually making the program slower, so it should be used with care.

`#[inline]` and `#[inline(always)]` always causes the function to be serialized into crate metadata to allow cross-crate inlining.

There are three different types of inline attributes:

- `#[inline]` hints the compiler to perform an inline expansion.
- `#[inline(always)]` asks the compiler to always perform an inline expansion.
- `#[inline(never)]` asks the compiler to never perform an inline expansion.

6.3.5 Deriving

The `deriving` attribute allows certain traits to be automatically implemented for data structures. For example, the following will create an `impl` for the `Eq` and `Clone` traits for `Foo`, the type parameter `T` will be given the `Eq` or `Clone` constraints for the appropriate `impl`:

```
#[deriving(Eq, Clone)]
struct Foo<T> {
    a: int,
    b: T
}
```

The generated `impl` for `Eq` is equivalent to

```
# struct Foo<T> { a: int, b: T }
impl<T: Eq> Eq for Foo<T> {
    fn eq(&self, other: &Foo<T>) -> bool {
        self.a == other.a && self.b == other.b
    }

    fn ne(&self, other: &Foo<T>) -> bool {
        self.a != other.a || self.b != other.b
    }
}
```

Supported traits for `deriving` are:

- Comparison traits: `Eq`, `TotalEq`, `Ord`, `TotalOrd`.
- Serialization: `Encodable`, `Decodable`. These require `serialize`.
- `Clone`, to create `T` from `&T` via a copy.
- `Hash`, to iterate over the bytes in a data type.
- `Rand`, to create a random instance of a data type.
- `Default`, to create an empty instance of a data type.
- `Zero`, to create an zero instance of a numeric data type.
- `FromPrimitive`, to create an instance from a numeric primitive.
- `Show`, to format a value using the `{}` formatter.

6.3.6 Stability

One can indicate the stability of an API using the following attributes:

- `deprecated`: This item should no longer be used, e.g. it has been replaced. No guarantee of backwards-compatibility.
- `experimental`: This item was only recently introduced or is otherwise in a state of flux. It may change significantly, or even be removed. No guarantee of backwards-compatibility.
- `unstable`: This item is still under development, but requires more testing to be considered stable. No guarantee of backwards-compatibility.
- `stable`: This item is considered stable, and will not change significantly. Guarantee of backwards-compatibility.
- `frozen`: This item is very stable, and is unlikely to change. Guarantee of backwards-compatibility.
- `locked`: This item will never change unless a serious bug is found. Guarantee of backwards-compatibility.

These levels are directly inspired by Node.js' "stability index".

There are lints for disallowing items marked with certain levels: `deprecated`, `experimental` and `unstable`; the first two will warn by default. Items with not marked with a stability are considered to be unstable for the purposes of the lint. One can give an optional string that will be displayed when the lint flags the use of an item.

```
#[warn(unstable)];

#[deprecated="replaced by ‘best‘"]
fn bad() {
    // delete everything
}

fn better() {
    // delete fewer things
}

#[stable]
fn best() {
    // delete nothing
}
```

```
fn main() {
    bad(); // "warning: use of deprecated item: replaced by ‘best‘"

    better(); // "warning: use of unmarked item"

    best(); // no warning
}
```

> **Note:** Currently these are only checked when applied to individual functions, structs, methods and enum variants, *not* to entire modules, traits, impls or enums themselves.

6.3.7 Compiler Features

Certain aspects of Rust may be implemented in the compiler, but they're not necessarily ready for every-day use. These features are often of "prototype quality" or "almost production ready", but may not be stable enough to be considered a full-fleged language feature.

For this reason, rust recognizes a special crate-level attribute of the form:

```
#[feature(feature1, feature2, feature3)]
```

This directive informs the compiler that the feature list: `feature1`, `feature2`, and `feature3` should all be enabled. This is only recognized at a crate-level, not at a module-level. Without this directive, all features are considered off, and using the features will result in a compiler error.

The currently implemented features of the compiler are:

- `macro_rules` - The definition of new macros. This does not encompass macro-invocation, that is always enabled by default, this only covers the definition of new macros. There are currently various problems with invoking macros, how they interact with their environment, and possibly how they are used outside of location in which they are defined. Macro definitions are likely to change slightly in the future, so they are currently hidden behind this feature.
- `globs` - Importing everything in a module through `*`. This is currently a large source of bugs in name resolution for Rust, and it's not clear whether this will continue as a feature or not. For these reasons, the glob import statement has been hidden behind this feature flag.
- `struct_variant` - Structural enum variants (those with named fields). It is currently unknown whether this style of enum variant is as fully supported as the tuple-forms, and it's not certain that this style of variant

should remain in the language. For now this style of variant is hidden behind a feature flag.

- `once_fns` - Onceness guarantees a closure is only executed once. Defining a closure as `once` is unlikely to be supported going forward. So they are hidden behind this feature until they are to be removed.
- `managed_boxes` - Usage of `@` pointers is gated due to many planned changes to this feature. In the past, this has meant "a GC pointer", but the current implementation uses reference counting and will likely change drastically over time. Additionally, the `@` syntax will no longer be used to create GC boxes.
- `asm` - The `asm!` macro provides a means for inline assembly. This is often useful, but the exact syntax for this feature along with its semantics are likely to change, so this macro usage must be opted into.
- `non_ascii_idents` - The compiler supports the use of non-ascii identifiers, but the implementation is a little rough around the edges, so this can be seen as an experimental feature for now until the specification of identifiers is fully fleshed out.
- `thread_local` - The usage of the `#[thread_local]` attribute is experimental and should be seen as unstable. This attribute is used to declare a `static` as being unique per-thread leveraging LLVM's implementation which works in concert with the kernel loader and dynamic linker. This is not necessarily available on all platforms, and usage of it is discouraged (rust focuses more on task-local data instead of thread-local data).
- `link_args` - This attribute is used to specify custom flags to the linker, but usage is strongly discouraged. The compiler's usage of the system linker is not guaranteed to continue in the future, and if the system linker is not used then specifying custom flags doesn't have much meaning.

If a feature is promoted to a language feature, then all existing programs will start to receive compilation warnings about #[feature] directives which enabled the new feature (because the directive is no longer necessary). However, if a feature is decided to be removed from the language, errors will be issued (if there isn't a parser error first). The directive in this case is no longer necessary, and it's likely that existing code will break if the feature isn't removed.

If a unknown feature is found in a directive, it results in a compiler error. An unknown feature is one which has never been recognized by the compiler.

7 Statements and expressions

Rust is *primarily* an expression language. This means that most forms of value-producing or effect-causing evaluation are directed by the uniform syntax category of *expressions*. Each kind of expression can typically *nest* within each other kind of expression, and rules for evaluation of expressions involve specifying both the value produced by the expression and the order in which its sub-expressions are themselves evaluated.

In contrast, statements in Rust serve *mostly* to contain and explicitly sequence expression evaluation.

7.1 Statements

A *statement* is a component of a block, which is in turn a component of an outer expression or function.

Rust has two kinds of statement: declaration statements and expression statements.

7.1.1 Declaration statements

A *declaration statement* is one that introduces one or more *names* into the enclosing statement block. The declared names may denote new slots or new items.

Item declarations An *item declaration statement* has a syntactic form identical to an item declaration within a module. Declaring an item – a function, enumeration, structure, type, static, trait, implementation or module – locally within a statement block is simply a way of restricting its scope to a narrow region containing all of its uses; it is otherwise identical in meaning to declaring the item outside the statement block.

Note: there is no implicit capture of the function's dynamic environment when declaring a function-local item.

Slot declarations

```
let_decl : "let" pat [':' type ] ? [ init ] ? ';' ;
init : [ '=' ] expr ;
```

A *slot declaration* introduces a new set of slots, given by a pattern. The pattern may be followed by a type annotation, and/or an initializer expression. When no type annotation is given, the compiler will infer the type, or signal an error if insufficient type information is available for definite inference. Any slots introduced by a slot declaration are visible from the point of declaration until the end of the enclosing block scope.

7.1.2 Expression statements

An *expression statement* is one that evaluates an expression and ignores its result. The type of an expression statement `e;` is always `()`, regardless of the type of `e`. As a rule, an expression statement's purpose is to trigger the effects of evaluating its expression.

7.2 Expressions

An expression may have two roles: it always produces a *value*, and it may have *effects* (otherwise known as "side effects"). An expression *evaluates to* a value, and has effects during *evaluation*. Many expressions contain sub-expressions (operands). The meaning of each kind of expression dictates several things: * Whether or not to evaluate the sub-expressions when evaluating the expression * The order in which to evaluate the sub-expressions * How to combine the sub-expressions' values to obtain the value of the expression.

In this way, the structure of expressions dictates the structure of execution. Blocks are just another kind of expression, so blocks, statements, expressions, and blocks again can recursively nest inside each other to an arbitrary depth.

Lvalues, rvalues and temporaries Expressions are divided into two main categories: *lvalues* and *rvalues*. Likewise within each expression, sub-expressions may occur in *lvalue context* or *rvalue context*. The evaluation of an expression depends both on its own category and the context it occurs within.

An lvalue is an expression that represents a memory location. These expressions are paths (which refer to local variables, function and method arguments, or static variables), dereferences (`*expr`), indexing expressions (`expr[expr]`), and field references (`expr.f`). All other expressions are rvalues.

The left operand of an assignment or compound-assignment expression is an lvalue context, as is the single operand of a unary borrow. All other expression contexts are rvalue contexts.

When an lvalue is evaluated in an *lvalue context*, it denotes a memory location; when evaluated in an *rvalue context*, it denotes the value held *in* that memory location.

When an rvalue is used in lvalue context, a temporary un-named lvalue is created and used instead. A temporary's lifetime equals the largest lifetime of any reference that points to it.

Moved and copied types When a local variable is used as an rvalue the variable will either be moved or copied, depending on its type. For types that contain owning pointers or values that implement the special trait `Drop`, the variable is moved. All other types are copied.

7.2.1 Literal expressions

A *literal expression* consists of one of the literal forms described earlier. It directly describes a number, character, string, boolean value, or the unit value.

```
();         // unit type
"hello";    // string type
'5';        // character type
5;          // integer type
```

7.2.2 Path expressions

A path used as an expression context denotes either a local variable or an item. Path expressions are lvalues.

7.2.3 Tuple expressions

Tuples are written by enclosing one or more comma-separated expressions in parentheses. They are used to create tuple-typed values.

```
(0,);
(0.0, 4.5);
("a", 4u, true);
```

7.2.4 Structure expressions

```
struct_expr : expr_path '{' ident ':' expr
                      [ ',' ident ':' expr ] *
                      [ ".." expr ] '}' |
              expr_path '(' expr
                      [ ',' expr ] * ')' |
              expr_path ;
```

There are several forms of structure expressions. A *structure expression* consists of the path of a structure item, followed by a brace-enclosed list of one or more comma-separated name-value pairs, providing the field values of a new instance of the structure. A field name can be any identifier, and is separated from its value expression by a colon. The location denoted by a structure field is mutable if and only if the enclosing structure is mutable.

A *tuple structure expression* consists of the path of a structure item, followed by a parenthesized list of one or more comma-separated expressions (in other words, the path of a structure item followed by a tuple expression). The structure item must be a tuple structure item.

A *unit-like structure expression* consists only of the path of a structure item.

The following are examples of structure expressions:

```
# struct Point { x: f64, y: f64 }
# struct TuplePoint(f64, f64);
# mod game { pub struct User<'a> { name: &'a str, age: uint, score: uint } }
# struct Cookie; fn some_fn<T>(t: T) {}
Point {x: 10.0, y: 20.0};
TuplePoint(10.0, 20.0);
let u = game::User {name: "Joe", age: 35, score: 100_000};
some_fn::<Cookie>(Cookie);
```

A structure expression forms a new value of the named structure type. Note that for a given *unit-like* structure type, this will always be the same value.

A structure expression can terminate with the syntax `..` followed by an expression to denote a functional update. The expression following `..` (the base) must have the same structure type as the new structure type being formed. The entire expression denotes the result of allocating a new structure (with the same type as the base expression) with the given values for the fields that were explicitly specified and the values in the base record for all other fields.

```
# struct Point3d { x: int, y: int, z: int }
let base = Point3d {x: 1, y: 2, z: 3};
Point3d {y: 0, z: 10, .. base};
```

7.2.5 Block expressions

```
block_expr : '{' [ view_item ] *
                 [ stmt ';' | item ] *
                 [ expr ] '}' ;
```

A *block expression* is similar to a module in terms of the declarations that are possible. Each block conceptually introduces a new namespace scope. View items can bring new names into scopes and declared items are in scope for only the block itself.

A block will execute each statement sequentially, and then execute the expression (if given). If the final expression is omitted, the type and return value of the block are `()`, but if it is provided, the type and return value of the block are that of the expression itself.

7.2.6 Method-call expressions

```
method_call_expr : expr '.' ident paren_expr_list ;
```

A *method call* consists of an expression followed by a single dot, an identifier, and a parenthesized expression-list. Method calls are resolved to methods on specific traits, either statically dispatching to a method if the exact `self`-type of the left-hand-side is known, or dynamically dispatching if the left-hand-side expression is an indirect object type.

7.2.7 Field expressions

```
field_expr : expr '.' ident ;
```

A *field expression* consists of an expression followed by a single dot and an identifier, when not immediately followed by a parenthesized expression-list (the latter is a method call expression). A field expression denotes a field of a structure.

```
myrecord.myfield;
foo().x;
(Struct {a: 10, b: 20}).a;
```

A field access on a record is an lvalue referring to the value of that field. When the field is mutable, it can be assigned to.

When the type of the expression to the left of the dot is a pointer to a record or structure, it is automatically dereferenced to make the field access possible.

7.2.8 Vector expressions

```
vec_expr : '[' "mut" ? vec_elems? ']' ;

vec_elems : [expr [',' expr]*] | [expr ',' ".." expr] ;
```

A *vector expression* is written by enclosing zero or more comma-separated expressions of uniform type in square brackets.

In the `[expr ',' ".." expr]` form, the expression after the `".."` must be a constant expression that can be evaluated at compile time, such as a literal or a static item.

```
[1, 2, 3, 4];
["a", "b", "c", "d"];
[0, ..128];             // vector with 128 zeros
[0u8, 0u8, 0u8, 0u8];
```

7.2.9 Index expressions

```
idx_expr : expr '[' expr ']' ;
```

Vector-typed expressions can be indexed by writing a square-bracket-enclosed expression (the index) after them. When the vector is mutable, the resulting lvalue can be assigned to.

Indices are zero-based, and may be of any integral type. Vector access is bounds-checked at run-time. When the check fails, it will put the task in a *failing state*.

```
# use std::task;
# do task::spawn {

([1, 2, 3, 4])[0];
(["a", "b"])[10]; // fails

# }
```

7.2.10 Unary operator expressions

Rust defines six symbolic unary operators. They are all written as prefix operators, before the expression they apply to.

`-` Negation. May only be applied to numeric types.

`*` Dereference. When applied to a pointer it denotes the pointed-to location. For pointers to mutable locations, the resulting lvalue can be assigned to. On non-pointer types, it calls the `deref` method of the `std::ops::Deref` trait, or the `deref_mut` method of the `std::ops::DerefMut` trait (if implemented by the type and required for an outer expression that will or could mutate the dereference), and produces the result of dereferencing the `&` or `&mut` borrowed pointer returned from the overload method.

`!` Logical negation. On the boolean type, this flips between `true` and `false`. On integer types, this inverts the individual bits in the two's complement representation of the value.

`~` Boxing operators. Allocate a box to hold the value they are applied to, and store the value in it. `~` creates an owned box.

`&` Borrow operator. Returns a reference, pointing to its operand. The operand of a borrow is statically proven to outlive the resulting pointer. If the borrow-checker cannot prove this, it is a compilation error.

7.2.11 Binary operator expressions

```
binop_expr : expr binop expr ;
```

Binary operators expressions are given in terms of operator precedence.

Arithmetic operators Binary arithmetic expressions are syntactic sugar for calls to built-in traits, defined in the `std::ops` module of the `std` library. This means that arithmetic operators can be overridden for user-defined types. The default meaning of the operators on standard types is given here.

`+` Addition and vector/string concatenation. Calls the `add` method on the `std::ops::Add` trait.

`-` Subtraction. Calls the `sub` method on the `std::ops::Sub` trait.

`*` Multiplication. Calls the `mul` method on the `std::ops::Mul` trait.

`/` Quotient. Calls the `div` method on the `std::ops::Div` trait.

`%` Remainder. Calls the `rem` method on the `std::ops::Rem` trait.

Bitwise operators Like the arithmetic operators, bitwise operators are syntactic sugar for calls to methods of built-in traits. This means that bitwise operators can be overridden for user-defined types. The default meaning of the operators on standard types is given here.

`&` And. Calls the `bitand` method of the `std::ops::BitAnd` trait.

`|` Inclusive or. Calls the `bitor` method of the `std::ops::BitOr` trait.

`^` Exclusive or. Calls the `bitxor` method of the `std::ops::BitXor` trait.

`<<` Logical left shift. Calls the `shl` method of the `std::ops::Shl` trait.

`>>` Logical right shift. Calls the `shr` method of the `std::ops::Shr` trait.

Lazy boolean operators The operators `||` and `&&` may be applied to operands of boolean type. The `||` operator denotes logical 'or', and the `&&` operator denotes logical 'and'. They differ from `|` and `&` in that the right-hand operand is only evaluated when the left-hand operand does not already determine the result of the expression. That is, `||` only evaluates its right-hand operand when the left-hand operand evaluates to `false`, and `&&` only when it evaluates to `true`.

Comparison operators Comparison operators are, like the arithmetic operators, and bitwise operators, syntactic sugar for calls to built-in traits. This means that comparison operators can be overridden for user-defined types. The default meaning of the operators on standard types is given here.

`==` Equal to. Calls the `eq` method on the `std::cmp::Eq` trait.

`!=` Unequal to. Calls the `ne` method on the `std::cmp::Eq` trait.

`<` Less than. Calls the `lt` method on the `std::cmp::Ord` trait.

`>` Greater than. Calls the `gt` method on the `std::cmp::Ord` trait.

`<=` Less than or equal. Calls the `le` method on the `std::cmp::Ord` trait.

`>=` Greater than or equal. Calls the `ge` method on the `std::cmp::Ord` trait.

Type cast expressions A type cast expression is denoted with the binary operator `as`.

Executing an `as` expression casts the value on the left-hand side to the type on the right-hand side.

A numeric value can be cast to any numeric type. A raw pointer value can be cast to or from any integral type or raw pointer type. Any other cast is unsupported and will fail to compile.

An example of an `as` expression:

```
# fn sum(v: &[f64]) -> f64 { 0.0 }
# fn len(v: &[f64]) -> int { 0 }

fn avg(v: &[f64]) -> f64 {
  let sum: f64 = sum(v);
  let sz: f64 = len(v) as f64;
  return sum / sz;
}
```

Assignment expressions An *assignment expression* consists of an lvalue expression followed by an equals sign (`=`) and an rvalue expression.

Evaluating an assignment expression either copies or moves its right-hand operand to its left-hand operand.

```
# let mut x = 0;
# let y = 0;

x = y;
```

Compound assignment expressions The `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, and `>>` operators may be composed with the `=` operator. The expression `lval OP= val` is equivalent to `lval = lval OP val`. For example, `x = x + 1` may be written as `x += 1`.

Any such expression always has the `unit` type.

Operator precedence The precedence of Rust binary operators is ordered as follows, going from strong to weak:

```
* / %
as
+ -
<< >>
&
^
|
< > <= >=
== !=
&&
||
=
```

Operators at the same precedence level are evaluated left-to-right. Unary operators have the same precedence level and it is stronger than any of the binary operators'.

7.2.12 Grouped expressions

An expression enclosed in parentheses evaluates to the result of the enclosed expression. Parentheses can be used to explicitly specify evaluation order within an expression.

```
paren_expr : '(' expr ')' ;
```

An example of a parenthesized expression:

```
let x = (2 + 3) * 4;
```

7.2.13 Call expressions

```
expr_list : [ expr [ ',' expr ]* ] ? ;
paren_expr_list : '(' expr_list ')' ;
call_expr : expr paren_expr_list ;
```

A *call expression* invokes a function, providing zero or more input slots and an optional reference slot to serve as the function's output, bound to the `lval` on the right hand side of the call. If the function eventually returns, then the expression completes.

Some examples of call expressions:

```
# use std::from_str::FromStr;
# fn add(x: int, y: int) -> int { 0 }

let x: int = add(1, 2);
let pi: Option<f32> = FromStr::from_str("3.14");
```

7.2.14 Lambda expressions

```
ident_list : [ ident [ ',' ident ]* ] ? ;
lambda_expr : '|' ident_list '|' expr ;
```

A *lambda expression* (sometimes called an "anonymous function expression") defines a function and denotes it as a value, in a single expression. A lambda expression is a pipe-symbol-delimited (`|`) list of identifiers followed by an expression.

A lambda expression denotes a function that maps a list of parameters (`ident_list`) onto the expression that follows the `ident_list`. The identifiers in the `ident_list` are the parameters to the function. These parameters' types need not be specified, as the compiler infers them from context.

Lambda expressions are most useful when passing functions as arguments to other functions, as an abbreviation for defining and capturing a separate function.

Significantly, lambda expressions *capture their environment*, which regular function definitions do not. The exact type of capture depends on the function type inferred for the lambda expression. In the simplest and least-expensive form (analogous to a `|| { }` expression), the lambda expression captures its environment by reference, effectively borrowing pointers to all outer variables mentioned inside the function. Alternately, the compiler may infer that a lambda expression should copy or move values (depending on their type.) from the environment into the lambda expression's captured environment.

In this example, we define a function `ten_times` that takes a higher-order function argument, and call it with a lambda expression as an argument.

```
fn ten_times(f: |int|) {
    let mut i = 0;
    while i < 10 {
        f(i);
        i += 1;
    }
}

ten_times(|j| println!("hello, {}", j));
```

7.2.15 While loops

```
while_expr : "while" expr '{' block '}' ;
```

A `while` loop begins by evaluating the boolean loop conditional expression. If the loop conditional expression evaluates to `true`, the loop body block executes and control returns to the loop conditional expression. If the loop conditional expression evaluates to `false`, the `while` expression completes.

An example:

```
let mut i = 0;

while i < 10 {
    println!("hello");
    i = i + 1;
}
```

7.2.16 Infinite loops

The keyword `loop` in Rust appears both in *loop expressions* and in *continue expressions*. A loop expression denotes an infinite loop; see Continue expressions for continue expressions.

```
loop_expr : [ lifetime ':' ] "loop" '{' block '}';
```

A `loop` expression may optionally have a *label*. If a label is present, then labeled `break` and `loop` expressions nested within this loop may exit out of this loop or return control to its head. See Break expressions.

7.2.17 Break expressions

```
break_expr : "break" [ lifetime ];
```

A `break` expression has an optional `label`. If the label is absent, then executing a `break` expression immediately terminates the innermost loop enclosing it. It is only permitted in the body of a loop. If the label is present, then `break foo` terminates the loop with label `foo`, which need not be the innermost label enclosing the `break` expression, but must enclose it.

7.2.18 Continue expressions

```
continue_expr : "loop" [ lifetime ];
```

A continue expression, written `loop`, also has an optional `label`. If the label is absent, then executing a `loop` expression immediately terminates the current iteration of the innermost loop enclosing it, returning control to the loop *head*. In the case of a `while` loop, the head is the conditional expression controlling the loop. In the case of a `for` loop, the head is the call-expression controlling the loop. If the label is present, then `loop foo` returns control to the head of the loop with label `foo`, which need not be the innermost label enclosing the `break` expression, but must enclose it.

A `loop` expression is only permitted in the body of a loop.

7.2.19 For expressions

```
for_expr : "for" pat "in" expr '{' block '}' ;
```

A `for` expression is a syntactic construct for looping over elements provided by an implementation of `std::iter::Iterator`.

An example of a for loop over the contents of a vector:

```
# type Foo = int;
# fn bar(f: Foo) { }
# let a = 0;
# let b = 0;
# let c = 0;

let v: &[Foo] = &[a, b, c];

for e in v.iter() {
    bar(*e);
}
```

An example of a for loop over a series of integers:

```
# fn bar(b:uint) { }
for i in range(0u, 256) {
    bar(i);
}
```

7.2.20 If expressions

```
if_expr : "if" expr '{' block '}'
          else_tail ? ;

else_tail : "else" [ if_expr
                   | '{' block '}' ] ;
```

An `if` expression is a conditional branch in program control. The form of an `if` expression is a condition expression, followed by a consequent block, any number of `else if` conditions and blocks, and an optional trailing `else` block. The condition expressions must have type `bool`. If a condition expression evaluates to `true`, the consequent block is executed and any subsequent `else if` or `else` block is skipped. If a condition expression evaluates to `false`, the consequent block is skipped and any subsequent `else if` condition is evaluated. If all `if` and `else if` conditions evaluate to `false` then any `else` block is executed.

7.2.21 Match expressions

```
match_expr : "match" expr '{' match_arm [ '|' match_arm ] * '}' ;

match_arm : match_pat "=>" [ expr "," | '{' block '}' ] ;

match_pat : pat [ ".." pat ] ? [ "if" expr ] ;
```

A `match` expression branches on a *pattern*. The exact form of matching that occurs depends on the pattern. Patterns consist of some combination of literals, destructured vectors or enum constructors, structures, records and tuples, variable binding specifications, wildcards (`..`), and placeholders (`_`). A `match` expression has a *head expression*, which is the value to compare to the patterns. The type of the patterns must equal the type of the head expression.

In a pattern whose head expression has an `enum` type, a placeholder (`_`) stands for a *single* data field, whereas a wildcard `..` stands for *all* the fields of a particular variant. For example:

```
enum List<X> { Nil, Cons(X, ~List<X>) }

let x: List<int> = Cons(10, ~Cons(11, ~Nil));

match x {
    Cons(_, ~Nil) => fail!("singleton list"),
    Cons(..)      => return,
    Nil           => fail!("empty list")
}
```

The first pattern matches lists constructed by applying `Cons` to any head value, and a tail value of `~Nil`. The second pattern matches *any* list constructed with `Cons`, ignoring the values of its arguments. The difference between `_` and `..` is that the pattern `C(_)` is only type-correct if `C` has exactly one argument, while the pattern `C(..)` is type-correct for any enum variant `C`, regardless of how many arguments `C` has.

Used inside a vector pattern, `..` stands for any number of elements. This wildcard can be used at most once for a given vector, which implies that it cannot be used to specifically match elements that are at an unknown distance from both ends of a vector, like `[.., 42, ..]`. If followed by a variable name, it will bind the corresponding slice to the variable. Example:

```
fn is_symmetric(list: &[uint]) -> bool {
    match list {
        [] | [_]                   => true,
        [x, ..inside, y] if x == y => is_symmetric(inside),
        _                          => false
    }
}

fn main() {
    let sym     = &[0, 1, 4, 2, 4, 1, 0];
    let not_sym = &[0, 1, 7, 2, 4, 1, 0];
    assert!(is_symmetric(sym));
    assert!(!is_symmetric(not_sym));
}
```

A `match` behaves differently depending on whether or not the head expression is an lvalue or an rvalue. If the head expression is an rvalue, it is first evaluated into a temporary location, and the resulting value is sequentially compared to the patterns in the arms until a match is found. The first arm with a matching pattern is chosen as the branch target of the `match`, any variables bound by the pattern are assigned to local variables in the arm's block, and control enters the block.

When the head expression is an lvalue, the match does not allocate a temporary location (however, a by-value binding may copy or move from the lvalue). When possible, it is preferable to match on lvalues, as the lifetime of these matches inherits the lifetime of the lvalue, rather than being restricted to the inside of the match.

An example of a `match` expression:

```
# fn process_pair(a: int, b: int) { }
# fn process_ten() { }
```

```
enum List<X> { Nil, Cons(X, ~List<X>) }

let x: List<int> = Cons(10, ~Cons(11, ~Nil));

match x {
    Cons(a, ~Cons(b, _)) => {
        process_pair(a,b);
    }
    Cons(10, _) => {
        process_ten();
    }
    Nil => {
        return;
    }
    _ => {
        fail!();
    }
}
```

Patterns that bind variables default to binding to a copy or move of the matched value (depending on the matched value's type). This can be changed to bind to a reference by using the `ref` keyword, or to a mutable reference using `ref mut`.

Subpatterns can also be bound to variables by the use of the syntax `variable @ pattern`. For example:

```
enum List { Nil, Cons(uint, ~List) }

fn is_sorted(list: &List) -> bool {
    match *list {
        Nil | Cons(_, ~Nil) => true,
        Cons(x, ref r @ ~Cons(y, _)) => (x <= y) && is_sorted(*r)
    }
}

fn main() {
    let a = Cons(6, ~Cons(7, ~Cons(42, ~Nil)));
    assert!(is_sorted(&a));
}
```

Patterns can also dereference pointers by using the `&`, `~` or `@` symbols, as appropriate. For example, these two matches on `x: &int` are equivalent:

```
# let x = &3;
let y = match *x { 0 => "zero", _ => "some" };
let z = match x { &0 => "zero", _ => "some" };

assert_eq!(y, z);
```

A pattern that's just an identifier, like `Nil` in the previous example, could either refer to an enum variant that's in scope, or bind a new variable. The compiler resolves this ambiguity by forbidding variable bindings that occur in `match` patterns from shadowing names of variants that are in scope. For example, wherever `List` is in scope, a `match` pattern would not be able to bind `Nil` as a new name. The compiler interprets a variable pattern `x` as a binding *only* if there is no variant named `x` in scope. A convention you can use to avoid conflicts is simply to name variants with upper-case letters, and local variables with lower-case letters.

Multiple match patterns may be joined with the `|` operator. A range of values may be specified with `..`. For example:

```
# let x = 2;

let message = match x {
  0 | 1  => "not many",
  2 .. 9 => "a few",
  _      => "lots"
};
```

Range patterns only work on scalar types (like integers and characters; not like vectors and structs, which have sub-components). A range pattern may not be a sub-range of another range pattern inside the same `match`.

Finally, match patterns can accept *pattern guards* to further refine the criteria for matching a case. Pattern guards appear after the pattern and consist of a bool-typed expression following the `if` keyword. A pattern guard may refer to the variables bound within the pattern they follow.

```
# let maybe_digit = Some(0);
# fn process_digit(i: int) { }
# fn process_other(i: int) { }

let message = match maybe_digit {
  Some(x) if x < 10 => process_digit(x),
  Some(x) => process_other(x),
  None => fail!()
};
```

7.2.22 Return expressions

```
return_expr : "return" expr ? ;
```

Return expressions are denoted with the keyword `return`. Evaluating a `return` expression moves its argument into the output slot of the current function, destroys the current function activation frame, and transfers control to the caller frame.

An example of a `return` expression:

```
fn max(a: int, b: int) -> int {
   if a > b {
      return a;
   }
   return b;
}
```

8 Type system

8.1 Types

Every slot, item and value in a Rust program has a type. The *type* of a *value* defines the interpretation of the memory holding it.

Built-in types and type-constructors are tightly integrated into the language, in nontrivial ways that are not possible to emulate in user-defined types. User-defined types have limited capabilities.

8.1.1 Primitive types

The primitive types are the following:

- The "unit" type `()`, having the single "unit" value `()` (occasionally called "nil"). [4]
- The boolean type `bool` with values `true` and `false`.
- The machine types.
- The machine-dependent integer and floating-point types.

[4]The "unit" value `()` is *not* a sentinel "null pointer" value for reference slots; the "unit" type is the implicit return type from functions otherwise lacking a return type, and can be used in other contexts (such as message-sending or type-parametric code) as a zero-size type.

Machine types The machine types are the following:

- The unsigned word types `u8`, `u16`, `u32` and `u64`, with values drawn from the integer intervals $[0, 2^8 - 1]$, $[0, 2^{16} - 1]$, $[0, 2^{32} - 1]$ and $[0, 2^{64} - 1]$ respectively.
- The signed two's complement word types `i8`, `i16`, `i32` and `i64`, with values drawn from the integer intervals $[-(2^7), 2^7 - 1]$, $[-(2^{15}), 2^{15} - 1]$, $[-(2^{31}), 2^{31} - 1]$, $[-(2^{63}), 2^{63} - 1]$ respectively.
- The IEEE 754-2008 `binary32` and `binary64` floating-point types: `f32` and `f64`, respectively.

Machine-dependent integer types The Rust type `uint`[5] is an unsigned integer type with target-machine-dependent size. Its size, in bits, is equal to the number of bits required to hold any memory address on the target machine.

The Rust type `int`[6] is a two's complement signed integer type with target-machine-dependent size. Its size, in bits, is equal to the size of the rust type `uint` on the same target machine.

8.1.2 Textual types

The types `char` and `str` hold textual data.

A value of type `char` is a Unicode scalar value (ie. a code point that is not a surrogate), represented as a 32-bit unsigned word in the 0x0000 to 0xD7FF or 0xE000 to 0x10FFFF range. A `[char]` vector is effectively an UCS-4 / UTF-32 string.

A value of type `str` is a Unicode string, represented as a vector of 8-bit unsigned bytes holding a sequence of UTF-8 codepoints. Since `str` is of unknown size, it is not a *first class* type, but can only be instantiated through a pointer type, such as `&str` or `~str`.

8.1.3 Tuple types

The tuple type-constructor forms a new heterogeneous product of values similar to the record type-constructor. The differences are as follows:

- tuple elements cannot be mutable, unlike record fields
- tuple elements are not named and can be accessed only by pattern-matching

[5] A Rust `uint` is analogous to a C99 `uintptr_t`.
[6] A Rust `int` is analogous to a C99 `intptr_t`.

Tuple types and values are denoted by listing the types or values of their elements, respectively, in a parenthesized, comma-separated list.

The members of a tuple are laid out in memory contiguously, like a record, in order specified by the tuple type.

An example of a tuple type and its use:

```
type Pair<'a> = (int,&'a str);
let p: Pair<'static> = (10,"hello");
let (a, b) = p;
assert!(b != "world");
```

8.1.4 Vector types

The vector type constructor represents a homogeneous array of values of a given type. A vector has a fixed size. (Operations like `vec.push` operate solely on owned vectors.) A vector type can be annotated with a *definite* size, such as `[int, ..10]`. Such a definite-sized vector type is a first-class type, since its size is known statically. A vector without such a size is said to be of *indefinite* size, and is therefore not a *first-class* type. An indefinite-size vector can only be instantiated through a pointer type, such as `&[T]` or `~[T]`. The kind of a vector type depends on the kind of its element type, as with other simple structural types.

Expressions producing vectors of definite size cannot be evaluated in a context expecting a vector of indefinite size; one must copy the definite-sized vector contents into a distinct vector of indefinite size.

An example of a vector type and its use:

```
let v: &[int] = &[7, 5, 3];
let i: int = v[2];
assert!(i == 3);
```

All in-bounds elements of a vector are always initialized, and access to a vector is always bounds-checked.

8.1.5 Structure types

A `struct` *type* is a heterogeneous product of other types, called the *fields* of the type. [7]

New instances of a `struct` can be constructed with a struct expression.

[7] `struct` types are analogous `struct` types in C, the *record* types of the ML family, or the *structure* types of the Lisp family.

The memory order of fields in a `struct` is given by the item defining it. Fields may be given in any order in a corresponding struct *expression*; the resulting `struct` value will always be laid out in memory in the order specified by the corresponding *item*.

The fields of a `struct` may be qualified by visibility modifiers, to restrict access to implementation-private data in a structure.

A *tuple struct* type is just like a structure type, except that the fields are anonymous.

A *unit-like struct* type is like a structure type, except that it has no fields. The one value constructed by the associated structure expression is the only value that inhabits such a type.

8.1.6 Enumerated types

An *enumerated type* is a nominal, heterogeneous disjoint union type, denoted by the name of an `enum` item. [8]

An `enum` item declares both the type and a number of *variant constructors*, each of which is independently named and takes an optional tuple of arguments.

New instances of an `enum` can be constructed by calling one of the variant constructors, in a call expression.

Any `enum` value consumes as much memory as the largest variant constructor for its corresponding `enum` type.

Enum types cannot be denoted *structurally* as types, but must be denoted by named reference to an `enum` item.

8.1.7 Recursive types

Nominal types – enumerations and structures – may be recursive. That is, each `enum` constructor or `struct` field may refer, directly or indirectly, to the enclosing `enum` or `struct` type itself. Such recursion has restrictions:

- Recursive types must include a nominal type in the recursion (not mere type definitions, or other structural types such as vectors or tuples).
- A recursive `enum` item must have at least one non-recursive constructor (in order to give the recursion a basis case).
- The size of a recursive type must be finite; in other words the recursive fields of the type must be pointer types.

[8] The `enum` type is analogous to a `data` constructor declaration in ML, or a *pick ADT* in Limbo.

- Recursive type definitions can cross module boundaries, but not module *visibility* boundaries, or crate boundaries (in order to simplify the module system and type checker).

An example of a *recursive* type and its use:

```
enum List<T> {
  Nil,
  Cons(T, ~List<T>)
}

let a: List<int> = Cons(7, ~Cons(13, ~Nil));
```

8.1.8 Pointer types

All pointers in Rust are explicit first-class values. They can be copied, stored into data structures, and returned from functions. There are four varieties of pointer in Rust:

Owning pointers (`~`) These point to owned heap allocations (or "boxes") in the shared, inter-task heap. Each owned box has a single owning pointer; pointer and pointee retain a 1:1 relationship at all times. Owning pointers are written `~content`, for example `~int` means an owning pointer to an owned box containing an integer. Copying an owned box is a "deep" operation: it involves allocating a new owned box and copying the contents of the old box into the new box. Releasing an owning pointer immediately releases its corresponding owned box.

References (`&`) These point to memory *owned by some other value.* References arise by (automatic) conversion from owning pointers, managed pointers, or by applying the borrowing operator `&` to some other value, including lvalues, rvalues or temporaries. References are written `&content`, or in some cases `&'f content` for some lifetime-variable `f`, for example `&int` means a reference to an integer. Copying a reference is a "shallow" operation: it involves only copying the pointer itself. Releasing a reference typically has no effect on the value it points to, with the exception of temporary values, which are released when the last reference to them is released.

Raw pointers (`*`) Raw pointers are pointers without safety or liveness guarantees. Raw pointers are written `*content`, for example `*int` means a raw pointer to an integer. Copying or dropping a raw pointer has no effect on the lifecycle of any other value. Dereferencing a raw pointer or converting it to any other pointer type is an `unsafe` operation. Raw pointers are generally discouraged in Rust code; they exist to support interoperability with foreign code, and writing performance-critical or low-level functions.

8.1.9 Function types

The function type constructor `fn` forms new function types. A function type consists of a possibly-empty set of function-type modifiers (such as `unsafe` or `extern`), a sequence of input types and an output type.

An example of a `fn` type:

```
fn add(x: int, y: int) -> int {
  return x + y;
}

let mut x = add(5,7);

type Binop<'a> = 'a |int,int| -> int;
let bo: Binop = add;
x = bo(5,7);
```

8.1.10 Closure types

The type of a closure mapping an input of type `A` to an output of type `B` is `|A| -> B`. A closure with no arguments or return values has type `||`.

An example of creating and calling a closure:

```
let captured_var = 10;

let closure_no_args = || println!("captured_var={}", captured_var);

let closure_args = |arg: int| -> int {
  println!("captured_var={}, arg={}", captured_var, arg);
  arg // Note lack of semicolon after 'arg'
};

fn call_closure(c1: ||, c2: |int| -> int) {
  c1();
  c2(2);
}

call_closure(closure_no_args, closure_args);
```

8.1.11 Object types

Every trait item (see traits) defines a type with the same name as the trait. This type is called the *object type* of the trait. Object types permit "late binding" of methods, dispatched using *virtual method tables* ("vtables"). Whereas most calls to trait methods are "early bound" (statically resolved) to specific implementations at compile time, a call to a method on an object type is only resolved to a vtable entry at compile time. The actual implementation for each vtable entry can vary on an object-by-object basis.

Given a pointer-typed expression `E` of type `&T` or `~T`, where `T` implements trait `R`, casting `E` to the corresponding pointer type `&R` or `~R` results in a value of the *object type* `R`. This result is represented as a pair of pointers: the vtable pointer for the `T` implementation of `R`, and the pointer value of `E`.

An example of an object type:

```
trait Printable {
  fn to_string(&self) -> ~str;
}

impl Printable for int {
  fn to_string(&self) -> ~str { self.to_str() }
}

fn print(a: ~Printable) {
   println!("{}", a.to_string());
}

fn main() {
   print(~10 as ~Printable);
}
```

In this example, the trait `Printable` occurs as an object type in both the type signature of `print`, and the cast expression in `main`.

8.1.12 Type parameters

Within the body of an item that has type parameter declarations, the names of its type parameters are types:

```
fn map<A: Clone, B: Clone>(f: |A| -> B, xs: &[A]) -> ~[B] {
    if xs.len() == 0 {
       return ~[];
    }
    let first: B = f(xs[0].clone());
    let rest: ~[B] = map(f, xs.slice(1, xs.len()));
    return ~[first] + rest;
}
```

Here, `first` has type `B`, referring to `map`'s `B` type parameter; and `rest` has type `~[B]`, a vector type with element type `B`.

8.1.13 Self types

The special type `self` has a meaning within methods inside an impl item. It refers to the type of the implicit `self` argument. For example, in:

```
trait Printable {
  fn make_string(&self) -> ~str;
}

impl Printable for ~str {
    fn make_string(&self) -> ~str {
        (*self).clone()
    }
}
```

`self` refers to the value of type `~str` that is the receiver for a call to the method `make_string`.

8.2 Type kinds

Types in Rust are categorized into kinds, based on various properties of the components of the type. The kinds are:

`Send` Types of this kind can be safely sent between tasks. This kind includes scalars, owning pointers, owned closures, and structural types containing only other owned types. All `Send` types are `'static`.

`Copy` Types of this kind consist of "Plain Old Data" which can be copied by simply moving bits. All values of this kind can be implicitly copied. This kind includes scalars and immutable references, as well as structural types containing other `Copy` types.

`'static` Types of this kind do not contain any references (except for references with the `static` lifetime, which are allowed). This can be a useful guarantee for code that breaks borrowing assumptions using `unsafe` operations.

`Drop` This is not strictly a kind, but its presence interacts with kinds: the `Drop` trait provides a single method `drop` that takes no parameters, and is run when values of the type are dropped. Such a method is called a "destructor", and are always executed in "top-down" order: a value is completely destroyed before any of the values it owns run their destructors. Only `Send` types can implement `Drop`.

Default Types with destructors, closure environments, and various other *non-first-class* types, are not copyable at all. Such types can usually only be accessed through pointers, or in some cases, moved between mutable locations.

Kinds can be supplied as *bounds* on type parameters, like traits, in which case the parameter is constrained to types satisfying that kind.

By default, type parameters do not carry any assumed kind-bounds at all. When instantiating a type parameter, the kind bounds on the parameter are checked to be the same or narrower than the kind of the type that it is instantiated with.

Sending operations are not part of the Rust language, but are implemented in the library. Generic functions that send values bound the kind of these values to sendable.

9 Memory and concurrency models

Rust has a memory model centered around concurrently-executing *tasks*. Thus its memory model and its concurrency model are best discussed simultaneously, as parts of each only make sense when considered from the perspective of the other.

When reading about the memory model, keep in mind that it is partitioned in order to support tasks; and when reading about tasks, keep in mind that their isolation and communication mechanisms are only possible due to the ownership and lifetime semantics of the memory model.

9.1 Memory model

A Rust program's memory consists of a static set of *items*, a set of tasks each with its own *stack*, and a *heap*. Immutable portions of the heap may be shared between tasks, mutable portions may not.

Allocations in the stack consist of *slots*, and allocations in the heap consist of *boxes*.

9.1.1 Memory allocation and lifetime

The *items* of a program are those functions, modules and types that have their value calculated at compile-time and stored uniquely in the memory image of the rust process. Items are neither dynamically allocated nor freed.

A task's *stack* consists of activation frames automatically allocated on entry to each function as the task executes. A stack allocation is reclaimed when control leaves the frame containing it.

The *heap* is a general term that describes two separate sets of boxes: managed boxes – which may be subject to garbage collection – and owned boxes. The lifetime of an allocation in the heap depends on the lifetime of the box values pointing to it. Since box values may themselves be passed in and out of frames, or stored in the heap, heap allocations may outlive the frame they are allocated within.

9.1.2 Memory ownership

A task owns all memory it can *safely* reach through local variables, as well as managed, owned boxes and references.

When a task sends a value that has the `Send` trait to another task, it loses ownership of the value sent and can no longer refer to it. This is statically guaranteed by the combined use of "move semantics", and the compiler-checked *meaning* of the `Send` trait: it is only instantiated for (transitively) sendable kinds of data constructor and pointers, never including managed boxes or references.

When a stack frame is exited, its local allocations are all released, and its references to boxes (both managed and owned) are dropped.

A managed box may (in the case of a recursive, mutable managed type) be cyclic; in this case the release of memory inside the managed structure may be deferred until task-local garbage collection can reclaim it. Code can ensure no such delayed deallocation occurs by restricting itself to owned boxes and similar unmanaged kinds of data.

When a task finishes, its stack is necessarily empty and it therefore has no references to any boxes; the remainder of its heap is immediately freed.

9.1.3 Memory slots

A task's stack contains slots.

A *slot* is a component of a stack frame, either a function parameter, a temporary, or a local variable.

A *local variable* (or *stack-local* allocation) holds a value directly, allocated within the stack's memory. The value is a part of the stack frame.

Local variables are immutable unless declared otherwise like: `let mut x = ...`.

Function parameters are immutable unless declared with `mut`. The `mut` keyword applies only to the following parameter (so `|mut x, y|` and `fn f(mut x: ~int, y: ~int)` declare one mutable variable `x` and one immutable variable `y`).

Methods that take either `self` or `~self` can optionally place them in a mutable slot by prefixing them with `mut` (similar to regular arguments):

```
trait Changer {
    fn change(mut self) -> Self;
    fn modify(mut ~self) -> ~Self;
}
```

Local variables are not initialized when allocated; the entire frame worth of local variables are allocated at once, on frame-entry, in an uninitialized state. Subsequent statements within a function may or may not initialize the local variables. Local variables can be used only after they have been initialized; this is enforced by the compiler.

9.1.4 Owned boxes

An *owned box* is a reference to a heap allocation holding another value, which is constructed by the prefix *tilde* sigil ~

An example of an owned box type and value:

```
let x: ~int = ~10;
```

Owned box values exist in 1:1 correspondence with their heap allocation copying an owned box value makes a shallow copy of the pointer Rust will consider a shallow copy of an owned box to move ownership of the value. After a value has been moved, the source location cannot be used unless it is reinitialized.

```
let x: ~int = ~10;
let y = x;
// attempting to use ‘x‘ will result in an error here
```

9.2 Tasks

An executing Rust program consists of a tree of tasks. A Rust *task* consists of an entry function, a stack, a set of outgoing communication channels and incoming communication ports, and ownership of some portion of the heap of a single operating-system process. (We expect that many programs will not use channels and ports directly, but will instead use higher-level abstractions provided in standard libraries, such as pipes.)

Multiple Rust tasks may coexist in a single operating-system process. The runtime scheduler maps tasks to a certain number of operating-system threads. By default, the scheduler chooses the number of threads based on the number of concurrent physical CPUs detected at startup. It’s also possible to override this choice at runtime. When the number of tasks exceeds the number of threads – which is likely – the scheduler multiplexes the tasks onto threads.[9]

[9] This is an M:N scheduler, which is known to give suboptimal results for CPU-bound

9.2.1 Communication between tasks

Rust tasks are isolated and generally unable to interfere with one another's memory directly, except through `unsafe` code. All contact between tasks is mediated by safe forms of ownership transfer, and data races on memory are prohibited by the type system.

Inter-task communication and co-ordination facilities are provided in the standard library. These include:

- synchronous and asynchronous communication channels with various communication topologies
- read-only and read-write shared variables with various safe mutual exclusion patterns
- simple locks and semaphores

When such facilities carry values, the values are restricted to the `Send` type-kind. Restricting communication interfaces to this kind ensures that no references or managed pointers move between tasks. Thus access to an entire data structure can be mediated through its owning "root" value; no further locking or copying is required to avoid data races within the substructure of such a value.

9.2.2 Task lifecycle

The *lifecycle* of a task consists of a finite set of states and events that cause transitions between the states. The lifecycle states of a task are:

- running
- blocked
- failing
- dead

A task begins its lifecycle – once it has been spawned – in the *running* state. In this state it executes the statements of its entry function, and any functions called by the entry function.

concurrency problems. In such cases, running with the same number of threads and tasks can yield better results. Rust has M:N scheduling in order to support very large numbers of tasks in contexts where threads are too resource-intensive to use in large number. The cost of threads varies substantially per operating system, and is sometimes quite low, so this flexibility is not always worth exploiting.

A task may transition from the *running* state to the *blocked* state any time it makes a blocking communication call. When the call can be completed – when a message arrives at a sender, or a buffer opens to receive a message – then the blocked task will unblock and transition back to *running*.

A task may transition to the *failing* state at any time, due being killed by some external event or internally, from the evaluation of a `fail!()` macro. Once *failing*, a task unwinds its stack and transitions to the *dead* state. Unwinding the stack of a task is done by the task itself, on its own control stack. If a value with a destructor is freed during unwinding, the code for the destructor is run, also on the task's control stack. Running the destructor code causes a temporary transition to a *running* state, and allows the destructor code to cause any subsequent state transitions. The original task of unwinding and failing thereby may suspend temporarily, and may involve (recursive) unwinding of the stack of a failed destructor. Nonetheless, the outermost unwinding activity will continue until the stack is unwound and the task transitions to the *dead* state. There is no way to "recover" from task failure. Once a task has temporarily suspended its unwinding in the *failing* state, failure occurring from within this destructor results in *hard* failure. A hard failure currently results in the process aborting.

A task in the *dead* state cannot transition to other states; it exists only to have its termination status inspected by other tasks, and/or to await reclamation when the last reference to it drops.

9.2.3 Task scheduling

The currently scheduled task is given a finite *time slice* in which to execute, after which it is *descheduled* at a loop-edge or similar preemption point, and another task within is scheduled, pseudo-randomly.

An executing task can yield control at any time, by making a library call to `std::task::yield`, which deschedules it immediately. Entering any other non-executing state (blocked, dead) similarly deschedules the task.

10 Runtime services, linkage and debugging

The Rust *runtime* is a relatively compact collection of C++ and Rust code that provides fundamental services and datatypes to all Rust tasks at run-time. It is smaller and simpler than many modern language runtimes. It is tightly integrated into the language's execution model of memory, tasks, communication and logging.

> **Note:** The runtime library will merge with the `std` library in future versions of Rust.

10.0.4 Memory allocation

The runtime memory-management system is based on a *service-provider interface*, through which the runtime requests blocks of memory from its environment and releases them back to its environment when they are no longer needed. The default implementation of the service-provider interface consists of the C runtime functions `malloc` and `free`.

The runtime memory-management system, in turn, supplies Rust tasks with facilities for allocating releasing stacks, as well as allocating and freeing heap data.

10.0.5 Built in types

The runtime provides C and Rust code to assist with various built-in types, such as vectors, strings, and the low level communication system (ports, channels, tasks).

Support for other built-in types such as simple types, tuples, records, and enums is open-coded by the Rust compiler.

10.0.6 Task scheduling and communication

The runtime provides code to manage inter-task communication. This includes the system of task-lifecycle state transitions depending on the contents of queues, as well as code to copy values between queues and their recipients and to serialize values for transmission over operating-system inter-process communication facilities.

10.0.7 Linkage

The Rust compiler supports various methods to link crates together both statically and dynamically. This section will explore the various methods to link Rust crates together, and more information about native libraries can be found in the ffi tutorial.

In one session of compilation, the compiler can generate multiple artifacts through the usage of command line flags and the `crate_type` attribute.

- `--crate-type=bin`, `#[crate_type = "bin"]` - A runnable executable will be produced. This requires that there is a `main` function in the crate which will be run when the program begins executing. This will link in all Rust and native dependencies, producing a distributable binary.

- `--crate-type=lib`, `#[crate_type = "lib"]` - A Rust library will be produced. This is an ambiguous concept as to what exactly is produced because a library can manifest itself in several forms. The purpose of this generic `lib` option is to generate the "compiler recommended" style of library. The output library will always be usable by rustc, but the actual type of library may change from time-to-time. The remaining output types are all different flavors of libraries, and the `lib` type can be seen as an alias for one of them (but the actual one is compiler-defined).
- `--crate-type=dylib`, `#[crate_type = "dylib"]` - A dynamic Rust library will be produced. This is different from the `lib` output type in that this forces dynamic library generation. The resulting dynamic library can be used as a dependency for other libraries and/or executables. This output type will create `*.so` files on linux, `*.dylib` files on osx, and `*.dll` files on windows.
- `--crate-type=staticlib`, `#[crate_type = "staticlib"]` - A static system library will be produced. This is different from other library outputs in that the Rust compiler will never attempt to link to `staticlib` outputs. The purpose of this output type is to create a static library containing all of the local crate's code along with all upstream dependencies. The static library is actually a `*.a` archive on linux and osx and a `*.lib` file on windows. This format is recommended for use in situtations such as linking Rust code into an existing non-Rust application because it will not have dynamic dependencies on other Rust code.
- `--crate-type=rlib`, `#[crate_type = "rlib"]` - A "Rust library" file will be produced. This is used as an intermediate artifact and can be thought of as a "static Rust library". These `rlib` files, unlike `staticlib` files, are interpreted by the Rust compiler in future linkage. This essentially means that `rustc` will look for metadata in `rlib` files like it looks for metadata in dynamic libraries. This form of output is used to produce statically linked executables as well as `staticlib` outputs.

Note that these outputs are stackable in the sense that if multiple are specified, then the compiler will produce each form of output at once without having to recompile.

With all these different kinds of outputs, if crate A depends on crate B, then the compiler could find B in various different forms throughout the system. The only forms looked for by the compiler, however, are the `rlib` format and the dynamic library format. With these two options for a dependent library, the compiler must at some point make a choice between these two formats. With this in mind, the compiler follows these rules when determining what format of dependencies will be used:

1. If a dynamic library is being produced, then it is required for all upstream Rust dependencies to also be dynamic. This is a limitation of the current

implementation of the linkage model. The reason behind this limitation is to prevent multiple copies of the same upstream library from showing up, and in the future it is planned to support a mixture of dynamic and static linking.

When producing a dynamic library, the compiler will generate an error if an upstream dependency could not be found, and also if an upstream dependency could only be found in an `rlib` format. Remember that `staticlib` formats are always ignored by `rustc` for crate-linking purposes.

2. If a static library is being produced, all upstream dependecies are required to be available in `rlib` formats. This requirement stems from the same reasons that a dynamic library must have all dynamic dependencies.

Note that it is impossible to link in native dynamic dependencies to a static library, and in this case warnings will be printed about all unlinked native dynamic dependencies.

3. If an `rlib` file is being produced, then there are no restrictions on what format the upstream dependencies are available in. It is simply required that all upstream dependencies be available for reading metadata from.

The reason for this is that `rlib` files do not contain any of their upstream dependencies. It wouldn't be very efficient for all `rlib` files to contain a copy of `libstd.rlib`!

4. If an executable is being produced, then things get a little interesting. As with the above limitations in dynamic and static libraries, it is required for all upstream dependencies to be in the same format. The next question is whether to prefer a dynamic or a static format. The compiler currently favors static linking over dynamic linking, but this can be inverted with the `-C prefer-dynamic` flag to the compiler.

What this means is that first the compiler will attempt to find all upstream dependencies as `rlib` files, and if successful, it will create a statically linked executable. If an upstream dependency is missing as an `rlib` file, then the compiler will force all dependencies to be dynamic and will generate errors if dynamic versions could not be found.

In general, `--crate-type=bin` or `--crate-type=lib` should be sufficient for all compilation needs, and the other options are just available if more fine-grained control is desired over the output format of a Rust crate.

10.0.8 Logging system

The runtime contains a system for directing logging expressions to a logging console and/or internal logging buffers. Logging can be enabled per module.

Logging output is enabled by setting the `RUST_LOG` environment variable. `RUST_LOG` accepts a logging specification made up of a comma-separated list of paths, with optional log levels. For each module containing log expressions, if `RUST_LOG` contains the path to that module or a parent of that module, then logs of the appropriate level will be output to the console.

The path to a module consists of the crate name, any parent modules, then the module itself, all separated by double colons (`::`). The optional log level can be appended to the module path with an equals sign (`=`) followed by the log level, from 1 to 4, inclusive. Level 1 is the error level, 2 is warning, 3 info, and 4 debug. You can also use the symbolic constants `error`, `warn`, `info`, and `debug`. Any logs less than or equal to the specified level will be output. If not specified then log level 4 is assumed. Debug messages can be omitted by passing `--cfg ndebug` to `rustc`.

As an example, to see all the logs generated by the compiler, you would set `RUST_LOG` to `rustc`, which is the crate name (as specified in its `crate_id` attribute). To narrow down the logs to just crate resolution, you would set it to `rustc::metadata::creader`. To see just error logging use `rustc=0`.

Note that when compiling source files that don't specify a crate name the crate is given a default name that matches the source file, with the extension removed. In that case, to turn on logging for a program compiled from, e.g. `helloworld.rs`, `RUST_LOG` should be set to `helloworld`.

As a convenience, the logging spec can also be set to a special pseudo-crate, `::help`. In this case, when the application starts, the runtime will simply output a list of loaded modules containing log expressions, then exit.

Logging Expressions Rust provides several macros to log information. Here's a simple Rust program that demonstrates all four of them:

```
#[feature(phase)];
#[phase(syntax, link)] extern crate log;

fn main() {
    error!("This is an error log")
    warn!("This is a warn log")
    info!("this is an info log")
    debug!("This is a debug log")
}
```

These four log levels correspond to levels 1-4, as controlled by `RUST_LOG`:

```
$ RUST_LOG=rust=3 ./rust
This is an error log
This is a warn log
this is an info log
```

11 Appendix: Rationales and design tradeoffs

TODO.

12 Appendix: Influences and further references

12.1 Influences

> The essential problem that must be solved in making a fault-tolerant software system is therefore that of fault-isolation. Different programmers will write different modules, some modules will be correct, others will have errors. We do not want the errors in one module to adversely affect the behaviour of a module which does not have any errors.
>
> — Joe Armstrong

> In our approach, all data is private to some process, and processes can only communicate through communications channels. *Security*, as used in this paper, is the property which guarantees that processes in a system cannot affect each other except by explicit communication.
>
> When security is absent, nothing which can be proven about a single module in isolation can be guaranteed to hold when that module is embedded in a system [. . .]
>
> — Robert Strom and Shaula Yemini

> Concurrent and applicative programming complement each other. The ability to send messages on channels provides I/O without side effects, while the avoidance of shared data helps keep concurrent processes from colliding.
>
> — Rob Pike

Rust is not a particularly original language. It may however appear unusual by contemporary standards, as its design elements are drawn from a number of "historical" languages that have, with a few exceptions, fallen out of favour. Five prominent lineages contribute the most, though their influences have come and gone during the course of Rust's development:

- The NIL (1981) and Hermes (1990) family. These languages were developed by Robert Strom, Shaula Yemini, David Bacon and others in their group at IBM Watson Research Center (Yorktown Heights, NY, USA).
- The Erlang (1987) language, developed by Joe Armstrong, Robert Virding, Claes Wikström, Mike Williams and others in their group at the Ericsson Computer Science Laboratory (Älvsjö, Stockholm, Sweden) .
- The Sather (1990) language, developed by Stephen Omohundro, Chu-Cheow Lim, Heinz Schmidt and others in their group at The International Computer Science Institute of the University of California, Berkeley (Berkeley, CA, USA).
- The Newsqueak (1988), Alef (1995), and Limbo (1996) family. These languages were developed by Rob Pike, Phil Winterbottom, Sean Dorward and others in their group at Bell Labs Computing Sciences Research Center (Murray Hill, NJ, USA).
- The Napier (1985) and Napier88 (1988) family. These languages were developed by Malcolm Atkinson, Ron Morrison and others in their group at the University of St. Andrews (St. Andrews, Fife, UK).

Additional specific influences can be seen from the following languages:

- The structural algebraic types and compilation manager of SML.
- The attribute and assembly systems of C#.
- The references and deterministic destructor system of C++.
- The memory region systems of the ML Kit and Cyclone.
- The typeclass system of Haskell.
- The lexical identifier rule of Python.
- The block syntax of Ruby.