Rust Reference Manual

July 3, 2013

0.7 (a2db7c1 2013-07-02 09:25:44 -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]

---

[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.

3.2 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` (`"`)

3.3 Comments

```
comment : block_comment | line_comment ;
block_comment : "/*" block_comment_body * '*' + '/' ;
block_comment_body : non_star * | '*' + non_slash_or_star ;
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 *three* slashes (`///`), and block comments beginning with a 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 comment's text.

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
copy
do drop
else enum extern
```

```
false fn for
if impl
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 * '"' ;

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

string_body : non_double_quote
            | '\x5c' [ '\x22' | common_escape ] ;

common_escape : '\x5c'
              | 'n' | 'r' | 't'
              | '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 ;
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 (\).

Some additional *escapes* are available in either character or 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*.

Number literals

```
num_lit : nonzero_dec [ dec_digit | '_' ] * num_suffix ?
        | '0' [       [ dec_digit | '_' ] + num_suffix ?
              | 'b'   [ '1' | '0' | '_' ] + 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 three 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.
- 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
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 is of type `float`. 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 three floating-point suffixes: `f` (for the base `float` type), `f32`, and `f64` (the 32-bit and 64-bit floating point types).

Examples of floating-point literals of various forms:

```
123.0;                             // type float
0.1;                               // type float
3f;                                // type float
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:

```
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:

- `fmt!` : format data into a string
- `env!` : look up an environment variable's value at compile time
- `stringify!` : pretty-print the Rust expression given as an argument
- `proto!` : define a protocol for inter-task communication
- `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, with the exception of `proto!`, are expressions with values. `proto!` is an item, defining a new name.

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 lock-step, 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_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 typically have the extension `.rs` but, by convention, source files that represent crates have the extension `.rc`, called *crate files*.

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. Atributes on the anonymous crate module define important metadata that influences the behavior of the compiler.

```
// Linkage attributes
#[ link(name = "projx",
        vers = "2.5",
        uuid = "9cccc5d5-aceb-4af5-8285-811211826b82") ];

// Additional metadata attributes
#[ desc = "Project X" ];
#[ license = "BSD" ];
#[ author = "Jane Doe" ];

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

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

---

[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.

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.

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 {
        ...
    }
    fn cos(f: f64) -> f64 {
        ...
    }
    fn tan(f: f64) -> f64 {
        ...
    }
}
```

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_mod_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 mod` declarations
- `use` declarations

Extern mod declarations

```
extern_mod_decl : "extern" "mod" ident [ '(' link_attrs ')' ] ? ;
link_attrs : link_attr [ ',' link_attrs ] + ;
link_attr : ident '=' literal ;
```

An *`extern mod` 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_mod_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 `link_attrs` provided in the `use_decl` against any `#link` attributes that were declared on the external crate when it was compiled. If no `link_attrs` are provided, a default `name` attribute is assumed, equal to the `ident` given in the `use_decl`.

Three examples of `extern mod` declarations:

```
extern mod pcre (uuid = "54aba0f8-a7b1-4beb-92f1-4cf625264841");

extern mod extra; // equivalent to: extern mod extra ( name = "extra" );

extern mod rustextra (name = "extra"); // linking to 'extra' under another name
```

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.

Note: Unlike in many languages, `use` declarations in Rust do *not* declare linkage dependency with external crates. Rather, `extern mod` 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::float::sin;
use std::option::{Some, None};

fn main() {
    // Equivalent to 'info!(std::float::sin(1.0));'
    info!(sin(1.0));

    // Equivalent to 'info!(~[std::option::Some(1.0), std::option::None]);'
    info!(~[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: ∼∼ 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::*`.

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: &fn(T)) {
    for seq.iter().advance |elt| { f(elt); }
}
fn map<T, U>(seq: &[T], f: &fn(T) -> U) -> ~[U] {
    let mut acc = ~[];
    for seq.iter().advance |elt| { 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, `sys::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 always be moved, but they can only be copied when the parameter is given a `Copy` bound.

```
fn id<T: Copy>(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.

Unsafe functions Unsafe functions are those containing unsafe operations that are not contained in an `unsafe` block. Such a function must be prefixed with the keyword `unsafe`.

Unsafe operations are those that potentially violate the memory-safety guarantees of Rust's static semantics. Specifically, the following operations are considered unsafe:

- Dereferencing a raw pointer.
- Casting a raw pointer to a safe pointer type.
- Calling an unsafe function.

Unsafe blocks A block of code can also be prefixed with the `unsafe` keyword, to permit a sequence of unsafe operations in an otherwise-safe function. This facility exists because the static semantics of Rust are a necessary approximation of the dynamic semantics. When a programmer has sufficient conviction that a sequence of 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", to the surrounding context.

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) -> ! {
    info!(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 $\perp$ ("bottom") that unifies with any type. Rust has no syntax for $\perp$.

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 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.

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

Extern functions may not be called from Rust code, but Rust code may take their value as a raw `u8` pointer.

```
let fptr: *u8 = new_vec;
```

The primary motivation for extern functions is to create callbacks for foreign functions that expect to receive function pointers.

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, float), Cat { name: ∼str, weight: float } }

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 borrowed pointers 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<’self> {
    mybits: [uint, ..2],
    mystring: &’self str
}

static bits_n_strings: BitsNStrings<’static> = BitsNStrings {
    mybits: bits,
    mystring: string
};
```

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.

```
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, &fn(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:

```
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.

```
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 float {
    fn from_int(n: int) -> float { n as float }
}
let x: float = Num::from_int(42);
```

Traits may inherit from other traits. For example, in

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

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. Refering to the previous example of `trait Circle : Shape`:

```
fn radius_times_area<T: Circle>(c: T) -> float {
    // ‘c‘ is both a Circle and a Shape
    c.radius() * c.area()
}
```

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

```
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 Circle {
    radius: float,
    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.

```
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.

```
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 the `abi` attribute as in

```
// Interface to the Windows API
#[abi = "stdcall"]
extern { }
```

The `link_name` attribute allows the name of the library to be specified.

```
#[link_name = "crypto"]
extern { }
```

The `nolink` attribute tells the Rust compiler not to do any linking for the external block. This is particularly useful for creating external blocks for libc, which tends to not follow standard library naming conventions and is linked to all Rust programs anyway.

6.2 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() {
  ...
}

// 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;
```

---

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

> **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.
- 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 crate.
- The `test` attribute, for marking functions as unit tests.
- The `allow`, `warn`, `forbid`, and `deny` attributes, for controlling lint checks. Lint checks supported by the compiler can be found via `rustc -W help`.
- The `deriving` attribute, for automatically generating implementations of certain traits.
- The `static_assert` attribute, for asserting that a static bool is true at compiletime

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

6.2.1 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.

`copy` Can be implicitly copied.

`owned` Are uniquely owned.

`durable` Contain borrowed pointers.

`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 borrowed pointer to a mutable value.

`return_to_mut` Release a borrowed pointer created with `return_to_mut`

`check_not_borrowed` Fail if a value has existing borrowed pointers 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`.

6.2.2 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

```
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 `extra`.
- `Clone` and `DeepClone`, to perform (deep) copies.
- `IterBytes`, to iterate over the bytes in a data type.
- `Rand`, to create a random instance of a data type.
- `Zero`, to create an zero (or empty) instance of a data type.
- `ToStr`, to convert to a string. For a type with this instance, `obj.to_str()` has similar output as `fmt!("%?", obj)`, but it differs in that each constituent field of the type must also implement `ToStr` and will have `field.to_str()` invoked to build up the result.

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.

Path, field and index expressions are lvalues. All other expressions are rvalues.

The left operand of an assignment, binary move or compound-assignment expression is an lvalue context, as is the single operand of a unary borrow, or move expression, and *both* operands of a swap expression. 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 borrowed pointer 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 mutable fields or owning pointers, 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,);
(0f, 4.5f);
("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:

```
Point {x: 10f, y: 20f};
TuplePoint(10f, 20f);
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.

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

7.2.5 Record expressions

```
rec_expr : '{' ident ':' expr
               [ ',' ident ':' expr ] *
               [ ".." expr ] '}'
```

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;
{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 derferenced 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*.

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

7.2.10 Unary operator expressions

Rust defines six symbolic unary operators, in addition to the unary copy and move 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. For enums that have only a single variant, containing a single parameter, the dereference operator accesses this parameter.

`!` 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.

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

`&` Borrow operator. Returns a borrowed pointer, pointing to its operand. The operand of a borrowed pointer 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 avg(v: &[float]) -> float {
  let sum: float = sum(v);
  let sz: float = len(v) as float;
  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.

```
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 Unary copy expressions

```
copy_expr : "copy" expr ;
```

> **Note:** `copy` expressions are deprecated. It's preferable to use the `Clone` trait and `clone()` method.

A *unary copy expression* consists of the unary `copy` operator applied to some argument expression.

Evaluating a copy expression first evaluates the argument expression, then copies the resulting value, allocating any memory necessary to hold the new copy.

Managed boxes (type `@`) are, as usual, shallow-copied, as are raw and borrowed pointers. Owned boxes, owned vectors and similar owned types are deep-copied.

Since the binary assignment operator `=` performs a copy or move implicitly, the unary copy operator is typically only used to cause an argument to a function to be copied and passed by value.

An example of a copy expression:

```
fn mutate(mut vec: ~[int]) {
   vec[0] = 10;
}

let v = ~[1,2,3];

mutate(copy v);   // Pass a copy

assert!(v[0] == 1); // Original was not modified
```

7.2.14 Unary move expressions

```
move_expr : "move" expr ;
```

A *unary move expression* is similar to a unary copy expression, except that it can only be applied to a local variable, and it performs a *move* on its operand, rather than a copy. That is, the memory location denoted by its operand is de-initialized after evaluation, and the resulting value is a shallow copy of the operand, even if the operand is an owning type.

> **Note:** In future versions of Rust, `move` may be removed as a separate operator; moves are now automatically performed for most cases `move` would be appropriate.

7.2.15 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:

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

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

ten_times(|j| println(fmt!("hello, %d", j)));
```

7.2.17 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\n");
    i = i + 1;
}
```

7.2.18 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.

7.2.19 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.

7.2.20 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.

7.2.21 Do expressions

```
do_expr : "do" expr [ '|' ident_list '|' ] ? '{' block '}' ;
```

A *do expression* provides a more-familiar block-syntax for a lambda expression, including a special translation of return expressions inside the supplied block.

Any occurrence of a return expression inside this `block` expression is rewritten as a reference to an (anonymous) flag set in the caller's environment, which is checked on return from the `expr` and, if set, causes a corresponding return from the caller. In this way, the meaning of `return` statements in language built-in control blocks is preserved, if they are rewritten using lambda functions and `do` expressions as abstractions.

The optional `ident_list` and `block` provided in a `do` expression are parsed as though they constitute a lambda expression; if the `ident_list` is missing, an empty `ident_list` is implied.

The lambda expression is then provided as a *trailing argument* to the outermost call or method call expression in the `expr` following `do`. If the `expr` is a path expression, it is parsed as though it is a call expression. If the `expr` is a field expression, it is parsed as though it is a method call expression.

In this example, both calls to `f` are equivalent:

```
f(|j| g(j));

do f |j| {
    g(j);
}
```

In this example, both calls to the (binary) function `k` are equivalent:

```
k(3, |j| l(j));

do k(3) |j| {
   l(j);
}
```

7.2.22 For expressions

```
for_expr : "for" expr [ '|' ident_list '|' ] ? '{' block '}' ;
```

A *for expression* is similar to a `do` expression, in that it provides a special block-form of lambda expression, suited to passing the `block` function to a higher-order function implementing a loop.

In contrast to a `do` expression, a `for` expression is designed to work with methods such as `each` and `times`, that require the body block to return a boolean. The `for` expression accommodates this by implicitly returning `true` at the end of each block, unless a `break` expression is evaluated.

In addition, `break` and `loop` expressions are rewritten inside `for` expressions in the same way that `return` expressions are, with a combination of local flag variables, and early boolean-valued returns from the `block` function, such that the meaning of `break` and `loop` is preserved in a primitive loop when rewritten as a `for` loop controlled by a higher order function.

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

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

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

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

```
for uint::range(0, 256) |i| {
    bar(i);
}
```

7.2.23 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.

7.2.24 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 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.

To execute an `match` expression, first the head expression is evaluated, then its 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.

An example of an `match` expression:

```
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 made explicit using the `copy` keyword, changed to bind to a borrowed pointer by using the `ref` keyword, or to a mutable borrowed pointer using `ref mut`.

A pattern that's just an identifier, like `Nil` in the previous answer, 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 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 message = match maybe_digit {
  Some(x) if x < 10 => process_digit(x),
  Some(x) => process_other(x),
  None => fail!()
};
```

7.2.25 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.

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.

---

[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-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.

Machine-dependent floating point type The Rust type `float` is a machine-specific type equal to one of the supported Rust floating-point machine types (`f32` or `f64`). It is the largest floating-point type that is directly supported by hardware on the target machine, or if the target machine has no floating-point hardware support, the largest floating-point type supported by the software floating-point library used to support the other floating-point machine types.

Note that due to the preference for hardware-supported floating-point, the type `float` may not be equal to the largest *supported* floating-point type.

8.1.2 Textual types

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

A value of type `char` is a Unicode character, represented as a 32-bit unsigned word holding a UCS-4 codepoint.

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`, `@str` or ~.

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

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

---

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

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

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<'self> = (int,&'self 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, written with a trailing asterisk and integer literal, 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]`, `@[T]` or ~. 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.

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

---

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

`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:

Managed pointers (`@`) These point to managed heap allocations (or "boxes") in the task-local, managed heap. Managed pointers are written `@content`, for example `@int` means a managed pointer to a managed box containing an integer. Copying a managed pointer is a "shallow" operation: it involves only copying the pointer itself (as well as any reference-count or GC-barriers required by the managed heap). Dropping a managed pointer does not necessarily release the box it points to; the lifecycles of managed boxes are subject to an unspecified garbage collection algorithm.

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 ∼, for example ∼ 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.

Borrowed pointers (`&`) These point to memory *owned by some other value*. Borrowed pointers arise by (automatic) conversion from owning pointers, managed pointers, or by applying the borrowing operator `&` to some other value, including lvalues, rvalues or temporaries. Borrowed pointers are written `&content`, or in some cases `&f/content` for some lifetime-variable `f`, for example `&int` means a borrowed pointer to an integer. Copying a borrowed pointer is a "shallow" operation: it involves only copying the pointer itself. Releasing a borrowed pointer typically has no effect on the value it points to, with the exception of temporary values, which are released when the last borrowed pointer 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 is 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<'self> = &'self fn(int,int) -> int;
let bo: Binop = add;
x = bo(5,7);
```

8.1.10 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_str(&self) -> ~str;
}

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

fn print(a: @Printable) {
   println(a.to_str());
}

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`.

8.1.11 Type parameters

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

```
fn map<A: Copy, B: Copy>(f: &fn(A) -> B, xs: &[A]) -> ~[B] {
   if xs.len() == 0 { return ~[]; }
   let first: B = f(copy xs[0]);
   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 $\sim$, a vector type with element type `B`.

8.1.12 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 { copy *self }
}
```

`self` refers to the value of type ∼ 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:

`Freeze` Types of this kind are deeply immutable; they contain no mutable memory locations directly or indirectly via pointers.

`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`.

`Static` Types of this kind do not contain any borrowed pointers; this can be a useful guarantee for code that breaks borrowing assumptions using `unsafe` operations.

`Copy` This kind includes all types that can be copied. All types with sendable kind are copyable, as are managed boxes, managed closures, trait types, and structural types built out of these. Types with destructors (types that implement `Drop`) can not implement `Copy`.

`Drop` This is not strictly a kind, but its presence interacts with kinds: the `Drop` trait provides a single method `finalize` 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 that do not implement `Copy` can implement `Drop`.

> **Note:** The `finalize` method may be renamed in future versions of Rust.

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.

Any operation that causes a value to be copied requires the type of that value to be of copyable kind, so the `Copy` bound is frequently required on function type parameters. For example, this is not a valid program:

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

Putting `x` into a managed box involves copying, and the `T` parameter has the default (non-copyable) kind. To change that, a bound is declared:

```
fn box<T: Copy>(x: T) -> @T { @x }
```

Calling this second version of `box` on a noncopyable type is not allowed. 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, owning and borrowed pointers.

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 or borrowed pointers.

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 with `let mut`. The `mut` keyword applies to all local variables declared within that declaration (so `let mut (x, y) = ...` declares two mutable variables, `x` and `y`).

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:` ∼ declare one mutable variable `x` and one immutable variable `y`).

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 Memory boxes

A *box* is a reference to a heap allocation holding another value. There are two kinds of boxes: *managed boxes* and *owned boxes*.

A *managed box* type or value is constructed by the prefix *at* sigil `@`.

An *owned box* type or value is constructed by the prefix *tilde* sigil ∼.

Multiple managed box values can point to the same heap allocation; copying a managed box value makes a shallow copy of the pointer (optionally incrementing a reference count, if the managed box is implemented through reference-counting).

Owned box values exist in 1:1 correspondence with their heap allocation; copying an owned box value makes a deep copy of the heap allocation and produces a pointer to the new allocation.

An example of constructing one managed box type and value, and one owned box type and value:

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

Some operations (such as field selection) implicitly dereference boxes. An example of an *implicit dereference* operation performed on box values:

```
struct Foo { y: int }
let x = @Foo{y: 10};
assert!(x.y == 10);
```

Other operations act on box values as single-word-sized address values. For these operations, to access the value held in the box requires an explicit dereference of the box value. Explicitly dereferencing a box is indicated with the unary *star* operator `*`. Examples of such *explicit dereference* operations are:

- copying box values (`x = y`)
- passing box values to functions (`f(x,y)`)

An example of an explicit-dereference operation performed on box values:

```
fn takes_boxed(b: @int) {
}

fn takes_unboxed(b: int) {
}

fn main() {
    let x: @int = @10;
    takes_boxed(x);
    takes_unboxed(*x);
}
```

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]

9.2.1 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.

---

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

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 borrowed 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.

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. The unwinding procedure of hard failure frees resources but does not execute destructors. The original (soft) failure is still resumed at the point where it was temporarily suspended.

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, extending and 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 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. Any logs less than or equal to the specified level will be output. If not specified then log level 4 is assumed.

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 `link` 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 either `.rs` or `.rc` 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.

The Rust runtime itself generates logging information. The runtime's logs are generated for a number of artificial modules in the `::rt` pseudo-crate, and can be enabled just like the logs for any standard module. The full list of runtime logging modules follows.

- `::rt::mem` Memory management
- `::rt::comm` Messaging and task communication
- `::rt::task` Task management
- `::rt::dom` Task scheduling
- `::rt::trace` Unused
- `::rt::cache` Type descriptor cache
- `::rt::upcall` Compiler-generated runtime calls
- `::rt::timer` The scheduler timer
- `::rt::gc` Garbage collection
- `::rt::stdlib` Functions used directly by the standard library
- `::rt::kern` The runtime kernel
- `::rt::backtrace` Log a backtrace on task failure
- `::rt::callback` Unused

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

```
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
rust: ~"\"This is an error log\""
rust: ~"\"This is a warn log\""
rust: ~"\"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 stack-growth implementation of Go.
- 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.