Rust Reference Manual

July 12, 2012

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 at `http://doc.rust-lang.org/doc/tutorial.html` to help acquire such background familiarity.

This document also does not serve as a reference to the core or standard libraries included in the language distribution. Those libraries are documented separately by extracting documentation attributes from their source code. Formatted documentation can be found at the following locations:

- Core library: `http://doc.rust-lang.org/doc/core`
- Standard library: `http://doc.rust-lang.org/doc/std`

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 on grammar: 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 small number of productions in Rust's grammar permit Unicode codepoints outside the ASCII range; these productions are defined in terms of character properties given by the Unicode standard, rather than ASCII-range codepoints. These are given in the section Special Unicode 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. No normalization is performed during input processing. Most Rust grammar rules are defined in terms of printable ASCII-range codepoints, but a small number are defined in terms of Unicode properties or explicit codepoint lists. [1]

3.2 Special Unicode Productions

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

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

3.2.1 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` is `non_null` restricted to exclude `U+002F` (`/`)
- `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 : block_comment | non_star * | '*' non_slash ;
line_comment : "//" non_eol * ;
```

Comments in Rust code follow the general C++ style of line and block-comment forms, with proper nesting of block-comment delimiters. 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 in crate files are the following strings:

```
import export use mod
```

The keywords in source files are the following strings:

```
alt again assert
break
check claim class const copy
drop
else enum export extern
fail false fn for
if iface impl import
let log loop
mod mut
pure
ret
```

```
true trait type
unchecked unsafe
while
```

Any of these have special meaning in their respective grammars, and 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
```

Nil and boolean literals The *nil value*, the only value of the type by 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 = map::hashmap<int,str>;  // Type arguments used in a type expression
let x = id::<int>(10);           // Type arguments used in a call expression
```

4 Crates and source files

Rust is a *compiled* language. Its semantics are divided along a *phase distinction* between compile-time and run-time. Those semantic rules that have a *static interpretation* govern the success or failure of compilation. A program that fails to compile due to violation of a compile-time rule has no defined semantics at run-time; the compiler should halt with an error report, and produce no executable artifact.

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

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.

Crates are provided to the Rust compiler through two kinds of file:

- *crate files*, that end in `.rc` and each define a `crate`.
- *source files*, that end in `.rs` and each define a `module`.

The Rust compiler is always invoked with a single input file, and always produces a single output crate.

When the Rust compiler is invoked with a crate file, it reads the *explicit* definition of the crate it's compiling from that file, and populates the crate with modules derived from all the source files referenced by the crate, reading and processing all the referenced modules at once.

When the Rust compiler is invoked with a source file, it creates an *implicit* crate and treats the source file as though it was referenced as the sole module populating this implicit crate. The module name is derived from the source file name, with the `.rs` extension removed.

4.1 Crate files

```
crate : attribute [ ';' | attribute* directive ]
      | directive ;
directive : view_item | dir_directive | source_directive ;
```

A crate file contains a crate definition, for which the production above defines the grammar. It is a declarative grammar that guides the compiler in assembling a crate from component source files.[2] A crate file describes:

- Attributes about the crate, such as author, name, version, and copyright. These are used for linking, versioning and distributing crates.
- The source-file and directory modules that make up the crate.
- Any `use`, `import` or `export` view items that apply to the anonymous module at the top-level of the crate's module tree.

An example of a crate file:

```
// 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" ];
```

[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.

```
// Import a module.
use std (ver = "1.0");

// Define some modules.
#[path = "foo.rs"]
mod foo;
mod bar {
    #[path =  "quux.rs"]
    mod quux;
}
```

4.1.1 Dir directives

A `dir_directive` forms a module in the module tree making up the crate, as well as implicitly relating that module to a directory in the filesystem containing source files and/or further subdirectories. The filesystem directory associated with a `dir_directive` module can either be explicit, or if omitted, is implicitly the same name as the module.

A `source_directive` references a source file, either explicitly or implicitly by combining the module name with the file extension `.rs`. The module contained in that source file is bound to the module path formed by the `dir_directive` modules containing the `source_directive`.

4.2 Source files

A source file contains a `module`: that is, a sequence of zero or more `item` definitions. Each source file is an implicit module, the name and location of which – in the module tree of the current crate – is defined from outside the source file: either by an explicit `source_directive` in a referencing crate file, or by the filename of the source file itself.

A source file that contains a `main` function can be compiled to an executable. If a `main` function is present, it must have no type parameters and no constraints. Its return type must be `nil` and it must either have no arguments, or a single argument of type `[str]`.

5 Items and attributes

A crate is a collection of items, each of which may have some number of attributes attached to it.

5.1 Items

```
item : mod_item | fn_item | type_item | enum_item
     | res_item | iface_item | impl_item | foreign_mod_item ;
```

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, remain constant during execution, and may reside in read-only memory.

There are several kinds of item:

- modules
- functions
- type definitions
- enumerations
- resources
- interfaces
- 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 exact locations in which sub-items may be declared is given by the grammar.

5.1.1 Type Parameters

All items except modules may be *parametrized* 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 the type of the item; in order to refer to the type-parametrized item, a referencing path must in general provide type arguments as a list of comma-separated types enclosed within angle brackets. 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.

5.1.2 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 {
        // ...
    }
}
```

View items

```
view_item : use_decl | import_decl | export_decl ;
```

A view item manages the namespace of a module; it does not define new items but simply changes the visibility of other items. There are several kinds of view item:

- use declarations
- import declarations
- export declarations

Use declarations

```
use_decl : "use" ident [ '(' link_attrs ')' ] ? ;
link_attrs : link_attr [ ',' link_attrs ] + ;
link_attr : ident '=' literal ;
```

A *use declaration* specifies a dependency on an external crate. The external crate is then imported into the declaring scope as the `ident` provided in the `use_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`.

Two examples of `use` declarations:

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

use std; // equivalent to: use std ( name = "std" );

use ruststd (name = "std"); // linking to 'std' under another name
```

Import declarations

```
import_decl : "import" ident [ '=' path
                             | "::" path_glob ] ;

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

An *import declaration* creates one or more local name bindings synonymous with some other path. Usually an import declaration is used to shorten the path required to refer to a module item.

Note: unlike many languages, Rust's `import` declarations do *not* declare linkage-dependency with external crates. Linkage dependencies are independently declared with `use` declarations.

Imports support a number of "convenience" notations:

- Importing as a different name than the imported name, using the syntax `import x = p::q::r;`.

- Importing a list of paths differing only in final element, using the glob-like brace syntax `import a::b::{c,d,e,f};`
- Importing all paths matching a given prefix, using the glob-like asterisk syntax `import a::b::*;`

An example of imports:

```
import foo = core::info;
import core::float::sin;
import core::str::{slice, hash};
import core::option::some;

fn main() {
    // Equivalent to 'log(core::info, core::float::sin(1.0));'
    log(foo, sin(1.0));

    // Equivalent to 'log(core::info, core::option::some(1.0));'
    log(info, some(1.0));

    // Equivalent to 'log(core::info,
    //                      core::str::hash(core::str::slice("foo", 0u, 1u)));'
    log(info, hash(slice("foo", 0u, 1u)));
}
```

Export declarations

```
export_decl : "export" ident [ ',' ident ] *
            | "export" ident "::{}"
            | "export" ident '{' ident [ ',' ident ] * '}' ;
```

An *export declaration* restricts the set of local names within a module that can be accessed from code outside the module. By default, all *local items* in a module are exported; imported paths are not automatically re-exported by default. If a module contains an explicit `export` declaration, this declaration replaces the default export with the export specified.

An example of an export:

```
mod foo {
    export primary;

    fn primary() {
        helper(1, 2);
        helper(3, 4);
    }

    fn helper(x: int, y: int) {
        // ...
    }
}

fn main() {
    foo::primary();  // Will compile.
}
```

If, instead of calling `foo::primary` in main, you were to call `foo::helper` then it would fail to compile:

```
    foo::helper(2,3) // ERROR: will not compile.
```

Multiple names may be exported from a single export declaration:

```
mod foo {
    export primary, secondary;

    fn primary() {
        helper(1, 2);
        helper(3, 4);
    }

    fn secondary() {
        // ...
    }

    fn helper(x: int, y: int) {
        // ...
    }
}
```

When exporting the name of an `enum` type `t`, by default, the module does *not* implicitly export any of `t`'s constructors. For example:

```
mod foo {
    export t;

    enum t {a, b, c}
}
```

Here, `foo` imports `t`, but not `a`, `b`, and `c`.

5.1.3 Functions

A *function item* defines a sequence of statements and an optional final expression associated 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 `ret` 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 `ret` expression applied to the final-expression.

An example of a function:

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

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) -> ! {
    log(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 `ret` 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 {
     ret 42;
```

```
    }
    else {
      my_err("Bad number!");
    }
}
```

The typechecker would complain that `f` doesn't return a value in the `else` branch. Adding the `!` annotation on `my_err` would express that `f` requires no explicit `ret`, as if it returns control to the caller, it returns a value (true because it never returns control).

Predicate functions Any pure boolean function is called a *predicate function*, and may be used in a constraint, as part of the static typestate system. A predicate declaration is identical to a function declaration, except that it is declared with the additional keyword `pure`. In addition, the typechecker checks the body of a predicate with a restricted set of typechecking rules. A predicate

- may not contain an assignment or self-call expression; and
- may only call other predicates, not general functions.

An example of a predicate:

```
pure fn lt_42(x: int) -> bool {
    ret (x < 42);
}
```

A non-boolean function may also be declared with `pure fn`. This allows predicates to call non-boolean functions as long as they are pure. For example:

```
pure fn pure_length<T>(ls: list<T>) -> uint { /* ... */ }

pure fn nonempty_list<T>(ls: list<T>) -> bool { pure_length(ls) > 0u }
```

In this example, `nonempty_list` is a predicate—it can be used in a typestate constraint—but the auxiliary function `pure_length` is not.

TODO: should actually define referential transparency.

The effect checking rules previously enumerated are a restricted set of typechecking rules meant to approximate the universe of observably referentially transparent Rust procedures conservatively. Sometimes, these rules are *too* restrictive. Rust allows programmers to violate these rules by writing predicates that the compiler cannot prove to be referentially transparent, using an escape-hatch feature called "unchecked blocks". When writing code that uses unchecked blocks, programmers should always be aware that they have an obligation to show that the code *behaves* referentially transparently at all times, even if the compiler cannot *prove* automatically that the code is referentially transparent. In the presence of unchecked blocks, the compiler provides no static guarantee that the code will behave as expected at runtime. Rather, the programmer has an independent obligation to verify the semantics of the predicates they write.

TODO: last two sentences are vague.

An example of a predicate that uses an unchecked block:

```
fn pure_foldl<T, U: copy>(ls: list<T>, u: U, f: fn(&&T, &&U) -> U) -> U {
    alt ls {
      nil { u }
      cons(hd, tl) { f(hd, pure_foldl(*tl, f(hd, u), f)) }
    }
}

pure fn pure_length<T>(ls: list<T>) -> uint {
    fn count<T>(_t: T, &&u: uint) -> uint { u + 1u }
    unchecked {
        pure_foldl(ls, 0u, count)
    }
}
```

Despite its name, `pure_foldl` is a `fn`, not a `pure fn`, because there is no way in Rust to specify that the higher-order function argument `f` is a pure function. So, to use `foldl` in a pure list length function that a predicate could then use, we must use an `unchecked` block wrapped around the call to `pure_foldl` in the definition of `pure_length`.

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

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

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

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

Extern functions may not be called from Rust code, but their value may be taken as an unsafe `u8` pointer.

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

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

5.1.4 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 (pinned, unique or shared).

For example, the type {`x: u8, y: u8`} defines the set of immutable values that are composite records, each containing two unsigned 8-bit integers accessed through the components `x` and `y`, and laid out in memory with the `x` component preceding the `y` component.

5.1.5 Enumerations

An *enumeration item* simultaneously declares a new 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. Note that `enum` previously was referred to as a `tag`, however this definition has been deprecated. While `tag` is no longer used, the two are synonymous.

The constructors of an `enum` type may be recursive: that is, each constructor may take an argument that refers, directly or indirectly, to the enumerated type the constructor is a member of. Such recursion has restrictions:

- Recursive types can be introduced only through `enum` constructors.
- A recursive `enum` item must have at least one non-recursive constructor (in order to give the recursion a basis case).
- The recursive argument of recursive `enum` constructors must be *box* values (in order to bound the in-memory size of the constructor).
- Recursive type definitions can cross module boundaries, but not module *visibility* boundaries or crate boundaries (in order to simplify the module system).

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

```
enum animal {
  dog,
  cat
}

let mut a: animal = dog;
a = cat;
```

An example of a *recursive* `enum` item and its use:

```
enum list<T> {
  nil,
  cons(T, @list<T>)
}

let a: list<int> = cons(7, @cons(13, @nil));
```

5.1.6 Classes

TODO: more about classes

Classes are named record types that may have a destructor associated with them, as well as fields and methods. For historical reasons, we may call a class with a destructor and a single field a "resource".

A *class item* declares a class type:

```
class file_descriptor {
    let fd: libc::c_int;
    new(fd: libc::c_int) { self.fd = fd; }
    drop { libc::close(self.fd); }
}
```

Calling the `file_descriptor` constructor function on an integer will produce a value with the `file_descriptor` type. Resource types have a noncopyable type kind, and thus may not be copied. Class types that don't have destructors may be copied if all their fields are copyable. The semantics guarantee that for each constructed resource value, the destructor will run once: when the value is disposed of (barring drastic program termination that somehow prevents unwinding from taking place). For stack-allocated values, disposal happens when the value goes out of scope. For values in shared boxes, it happens when the reference count of the box reaches zero.

The argument or arguments to the class constructor may be stored in the class's named fields, and can be accessed by a field reference. In this case, the `file_descriptor`'s data field would be accessed like `f.fd`, if `f` is a value of type `file_descriptor`.

5.1.7 Interfaces

An *interface item* describes a set of method types. *implementation items* can be used to provide implementations of those methods for a specific type.

```
iface shape {
    fn draw(surface);
    fn bounding_box() -> bounding_box;
}
```

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

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

```
iface seq<T> {
   fn len() -> uint;
   fn elt_at(n: uint) -> T;
   fn iter(fn(T));
}
```

Generic functions may use interfaces as bounds on their type parameters. This will have two effects: only types that implement the interface can be used to instantiate the parameter, and within the generic function, the methods of the interface 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);
}
```

Interface items also define a type with the same name as the interface. Values of this type are created by casting values (of a type for which an implementation of the given interface is in scope) to the interface type.

```
let myshape: shape = mycircle as shape;
```

The resulting value is a reference counted box containing the value that was cast along with information that identify the methods of the implementation that was used. Values with an interface type can always have methods of their interface called on them, and can be used to instantiate type parameters that are bounded on their interface.

5.1.8 Implementations

An *implementation item* provides an implementation of an interface for a type.

```
type circle = {radius: float, center: point};

impl circle_shape of shape for circle {
    fn draw(s: surface) { do_draw_circle(s, self); }
    fn bounding_box() -> bounding_box {
        let r = self.radius;
        {x: self.center.x - r, y: self.center.y - r,
         width: 2.0 * r, height: 2.0 * r}
    }
}
```

This defines an implementation named `circle_shape` of interface `shape` for type `circle`. The name of the implementation is the name by which it is imported and exported, but has no further significance. It may be omitted to default to the name of the interface that was implemented. Implementation names do not conflict the way other names do: multiple implementations with the same name may exist in a scope at the same time.

It is possible to define an implementation without referencing an interface. The methods in such an implementation can only be used statically (as direct calls on the values of the type that the implementation targets). In such an implementation, the `of` clause is not given, and the name is mandatory.

```
impl uint_loops for uint {
    fn times(f: fn(uint)) {
        let mut i = 0u;
        while i < self { f(i); i += 1u; }
    }
}
```

When an interface is specified, all methods declared as part of the interface must be present, with matching types and type parameter counts, in the implementation.

An implementation can take type parameters, which can be different from the type parameters taken by the interface it implements. They are written after the name of the implementation, or if that is not specified, after the `impl` keyword.

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

5.1.9 Foreign modules

```
foreign_mod_item : "extern mod" ident '{' foreign_mod '} ;
foreign_mod : [ foreign_fn ] * ;
```

Foreign modules form the basis for Rust's foreign function interface. A foreign module describes functions in external, non-Rust libraries. Functions within foreign modules are declared the same as other Rust functions, with the exception that they may not have a body and are instead terminated by a semi-colon.

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

Functions within foreign modules may be called by Rust code as it would any normal function and the Rust compiler will automatically translate between the Rust ABI and the foreign ABI.

The name of the foreign module has special meaning to the Rust compiler in that it will treat the module name as the name of a library to link to, performing the linking as appropriate for the target platform. The name given for the foreign module will be transformed in a platform-specific way to determine the name of the library. For example, on Linux the name of the foreign module is prefixed with 'lib' and suffixed with '.so', so the foreign mod 'rustrt' would be linked to a library named 'librustrt.so'.

A number of attributes control the behavior of foreign modules.

By default foreign modules assume that the library they are calling use the standard C "cdecl" ABI. Other ABI's may be specified using the `abi` attribute as in

```
// Interface to the Windows API
#[abi = "stdcall"]
extern mod kernel32 { }
```

The `link_name` attribute allows the default library naming behavior to be overriden by explicitly specifying the name of the library.

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

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

5.2 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 piece of metadata 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 are applied to an entity by placing them within a hash-list (`#[...]`) as either a prefix to the entity or as a semicolon-delimited declaration within the entity body.

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
```

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

```
#[cfg(target_os="linux")]
mod bar {
  // ...
}

// A documentation attribute
#[doc = "Add two numbers together."]
fn add(x: int, y: int) { x + y }
```

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 will be made 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 `link` attribute, for describing linkage metadata for a crate.
- The `test` attribute, for marking functions as unit tests.

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

6 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.

6.1 Statements

A *statement* is a component of a block, which is in turn a component of an outer expression or function. When a function is spawned into a task, the task *executes* statements in an order determined by the body of the enclosing function. Each statement causes the task to perform certain actions.

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

6.1.1 Declaration statements

A *declaration statement* is one that introduces a *name* into the enclosing statement block. The declared name may denote a new slot or a new item.

Item declarations An *item declaration statement* has a syntactic form identical to an item declaration within a module. Declaring an item – a function, enumeration, type, resource, interface, 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* has one of two forms:

- `let pattern optional-init`;
- `let pattern` : `type optional-init`;

Where `type` is a type expression, `pattern` is an irrefutable pattern (often just the name of a single slot), and `optional-init` is an optional initializer. If present, the initializer consists of either an assignment operator (`=`) or move operator (`<-`), followed by an expression.

Both forms introduce a new slot into the enclosing block scope. The new slot is visible from the point of declaration until the end of the enclosing block scope.

The former form, with no type annotation, causes the compiler to infer the static type of the slot through unification with the types of values assigned to the slot in the remaining code in the block scope. Inference only occurs on frame-local variable, not argument slots. Function signatures must always declare types for all argument slots.

6.1.2 Expression statements

An *expression statement* is one that evaluates an expression and drops its result. The purpose of an expression statement is often to cause the side effects of the expression's evaluation.

6.2 Expressions

An expression plays the dual roles of causing side effects and producing a *value*. Expressions are said to *evaluate to* a value, and the side effects are caused during *evaluation*. Many expressions contain sub-expressions as operands; the definition of each kind of expression dictates whether or not, and in which order, it will evaluate its sub-expressions, and how the expression's value derives from the value of its sub-expressions.

In this way, the structure of execution – both the overall sequence of observable side effects and the final produced value – is dictated by the structure of expressions. Blocks themselves are expressions, so the nesting sequence of block, statement, expression, and block can repeatedly nest to an arbitrary depth.

6.2.1 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 nil value.

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

6.2.2 Tuple expressions

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

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

6.2.3 Record expressions

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

A *record expression* is one or more comma-separated name-value pairs enclosed by braces. A fieldname can be any identifier (including keywords), and is separated from its value expression by a colon. To indicate that a field is mutable, the `mut` keyword is written before its name.

```
{x: 10f, y: 20f};
{name: "Joe", age: 35u, score: 100_000};
{ident: "X", mut count: 0u};
```

The order of the fields in a record expression is significant, and determines the type of the resulting value. {`a: u8, b: u8`} and {`b: u8, a: u8`} are two different fields.

A record expression can terminate with the word `with` followed by an expression to denote a functional update. The expression following `with` (the base) must be of a record type that includes at least all the fields mentioned in the record expression. A new record will be created, of 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. The ordering of the fields in such a record expression is not significant.

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

6.2.4 Field expressions

```
field_expr : expr '.' expr
```

A dot can be used to access a field in a record.

```
myrecord.myfield;
{a: 10, b: 20}.a;
```

A field access on a record is an *lval* 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 boxed record, it is automatically derferenced to make the field access possible.

Field access syntax is overloaded for interface method access. When no matching field is found, or the expression to the left of the dot is not a (boxed) record, an implementation that matches this type and the given method name is looked up instead, and the result of the expression is this method, with its *self* argument bound to the expression on the left of the dot.

6.2.5 Vector expressions

```
vec_expr : '[' "mut" ? [ expr [ ',' expr ] * ] ? ']'
```

A *vector expression* is written by enclosing zero or more comma-separated expressions of uniform type in square brackets. The keyword `mut` can be written after the opening bracket to indicate that the elements of the resulting vector may be mutated. When no mutability is specified, the vector is immutable.

```
~[1, 2, 3, 4];
~["a", "b", "c", "d"];
~[mut 0u8, 0u8, 0u8, 0u8];
```

6.2.6 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 *lval* 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];
(~[mut 'x', 'y'])[1] = 'z';
(~["a", "b"])[10]; // fails
```

6.2.7 Unary operator expressions

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

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

`*` Dereference. When applied to a box or resource type, it accesses the inner value. For mutable boxes, the resulting *lval* 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 shared, reference-counted box, whereas `~` creates a unique box.

6.2.8 Binary operator expressions

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

Binary operators expressions are given in terms of operator precedence.

Arithmetic operators Binary arithmetic expressions require both their operands to be of the same type, and can be applied only to numeric types, with the exception of `+`, which acts both as addition operator on numbers and as concatenate operator on vectors and strings.

`+` Addition and vector/string concatenation.

`-` Subtraction.

`*` Multiplication.

`/` Division.

`%` Remainder.

Bitwise operators Bitwise operators apply only to integer types, and perform their operation on the bits of the two's complement representation of the values.

`&` And.

| Inclusive or.

ˆ Exclusive or.

<< Logical left shift.

>> Logical right shift.

>>> Arithmetic right shift.

Lazy boolean operators The operators || and `&&` may be applied to operands of boolean type. The first performs the 'or' operation, and the second the 'and' operation. They differ from | and `&` in that the right-hand operand is only evaluated when the left-hand operand does not already determine the outcome 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

`==` Equal to.

`!=` Unequal to.

< Less than.

> Greater than.

<= Less than or equal.

>= Greater than or equal.

The binary comparison operators can be applied to any two operands of the same type, and produce a boolean value.

TODO details on how types are descended during comparison.

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. An unsafe pointer value can be cast to or from any integral type or unsafe 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;
  ret sum / sz;
}
```

A cast is a *trivial cast* iff the type of the casted expression and the target type are identical after replacing all occurrences of `int`, `uint`, `float` with their machine type equivalents of the target architecture in both types.

Binary move expressions A *binary move expression* consists of an *lval* followed by a left-pointing arrow (<-) and an *rval* expression.

Evaluating a move expression causes, as a side effect, the *rval* to be *moved* into the *lval*. If the *rval* was itself an *lval*, it must be a local variable, as it will be de-initialized in the process.

Evaluating a move expression does not change reference counts, nor does it cause a deep copy of any unique structure pointed to by the moved *rval*. Instead, the move expression represents an indivisible *transfer of ownership* from the right-hand-side to the left-hand-side of the expression. No allocation or destruction is entailed.

An example of three different move expressions:

```
x <- a;
x[i] <- b;
y.z <- c;
```

Swap expressions A *swap expression* consists of an *lval* followed by a bi-directional arrow (<->) and another *lval* expression.

Evaluating a swap expression causes, as a side effect, the values held in the left-hand-side and right-hand-side *lvals* to be exchanged indivisibly.

Evaluating a swap expression neither changes reference counts nor deeply copies any unique structure pointed to by the moved *rval*. Instead, the swap expression represents an indivisible *exchange of ownership* between the right-hand-side and the left-hand-side of the expression. No allocation or destruction is entailed.

An example of three different swap expressions:

```
x <-> a;
x[i] <-> a[i];
y.z <-> b.c;
```

Assignment expressions An *assignment expression* consists of an *lval* expression followed by an equals sign (=) and an *rval* expression.

Evaluating an assignment expression is equivalent to evaluating a binary move expression applied to a unary copy expression. For example, the following two expressions have the same effect:

```
x = y;
x <- copy y;
```

The former is just more terse and familiar.

Operator-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`.

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.

6.2.9 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;
```

6.2.10 Unary copy expressions

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

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.

Shared boxes (type `@`) are, as usual, shallow-copied, as they may be cyclic. Unique boxes, vectors and similar unique types are deep-copied.

Since the binary assignment operator `=` performs a copy 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(vec: ~[mut int]) {
   vec[0] = 10;
}

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

mutate(copy v);   // Pass a copy

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

6.2.11 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.

A call expression statically requires that the precondition declared in the callee's signature is satisfied by the expression prestate. In this way, typestates propagate through function boundaries.

An example of a call expression:

```
let x: int = add(1, 2);
```

6.2.12 Shared function expressions

TODO.

6.2.13 Unique function expressions

TODO.

6.2.14 While loops

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

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:

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

6.2.15 Infinite loops

A `loop` expression denotes an infinite loop:

```
loop_expr : "loop" '{' block '}';
```

For a block `b`, the expression `loop b` is semantically equivalent to `while true b`. However, `loop`s differ from `while` loops in that the typestate analysis pass takes into account that `loop`s are infinite.

For example, the following (contrived) function uses a `loop` with a `ret` expression:

```
fn count() -> bool {
  let mut i = 0;
  loop {
    i += 1;
    if i == 20 { ret true; }
  }
}
```

This function compiles, because typestate recognizes that the `loop` never terminates (except non-locally, with `ret`), thus there is no need to insert a spurious `fail` or `ret` after the `loop`. If `loop` were replaced with `while true`, the function would be rejected because from the compiler's perspective, there would be a control path along which `count` does not return a value (that is, if the loop condition is always false).

6.2.16 Break expressions

```
break_expr : "break" ;
```

Executing a `break` expression immediately terminates the innermost loop enclosing it. It is only permitted in the body of a loop.

6.2.17 Again expressions

```
again_expr : "again" ;
```

Evaluating an `again` 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.

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

6.2.18 For expressions

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

A *for loop* is controlled by a vector or string. The for loop bounds-checks the underlying sequence *once* when initiating the loop, then repeatedly executes the loop body with the loop variable referencing the successive elements of the underlying sequence, one iteration per sequence element.

An example a for loop:

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

for v.each |e| {
    bar(e);
}
```

6.2.19 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.

6.2.20 Alternative expressions

```
alt_expr : "alt" expr '{' alt_arm [ '|' alt_arm ] * '}' ;

alt_arm : alt_pat '{' block '}' ;

alt_pat : pat [ "to" pat ] ? [ "if" expr ] ;
```

An `alt` 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, records and tuples, variable binding specifications, wildcards (`*`), and placeholders (`_`). An `alt` 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));

alt x {
    cons(_, @nil) { fail "singleton list"; }
    cons(*)       { ret; }
    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 `alt` 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 `alt`, 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 `alt` expression:

```
enum list<X> { nil, cons(X, @list<X>) }

let x: list<int> = cons(10, @cons(11, @nil));

alt x {
    cons(a, @cons(b, _)) {
        process_pair(a,b);
    }
    cons(10, _) {
        process_ten();
    }
    nil {
        ret;
    }
    _ {
        fail;
    }
}
```

Records can also be pattern-matched and their fields bound to variables. When matching fields of a record, the fields being matched are specified first, then a placeholder (_) represents the remaining fields.

```
fn main() {
    let r = {
        player: "ralph",
        stats: load_stats(),
        options: {
            choose: true,
            size: "small"
        }
    };

    alt r {
      {options: {choose: true, _}, _} {
        choose_player(r)
      }
      {player: p, options: {size: "small", _}, _} {
        log(info, p + " is small");
      }
      _ {
        next_player();
      }
    }
}
```

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

```
let message = alt x {
  0 | 1  { "not many" }
  2 to 9 { "a few" }
  _      { "lots" }
};
```

Finally, alt 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 = alt maybe_digit {
  some(x) if x < 10 { process_digit(x) }
  some(x) { process_other(x) }
  none { fail }
};
```

6.2.21 Fail expressions

```
fail_expr : "fail" expr ? ;
```

Evaluating a `fail` expression causes a task to enter the *failing* state. In the *failing* state, a task unwinds its stack, destroying all frames and freeing all resources until it reaches its entry frame, at which point it halts execution in the *dead* state.

6.2.22 Note expressions

```
note_expr : "note" expr ;
```

Note: Note expressions are not yet supported by the compiler.

A `note` expression has no effect during normal execution. The purpose of a `note` expression is to provide additional diagnostic information to the logging subsystem during task failure. See log expressions. Using `note` expressions, normal diagnostic logging can be kept relatively sparse, while still providing verbose diagnostic "back-traces" when a task fails.

When a task is failing, control frames *unwind* from the innermost frame to the outermost, and from the innermost lexical block within an unwinding frame to the outermost. When unwinding a lexical block, the runtime processes all the `note` expressions in the block sequentially, from the first expression of the block to the last. During processing, a `note` expression has equivalent meaning to a `log` expression: it causes the runtime to append the argument of the `note` to the internal logging diagnostic buffer.

An example of a `note` expression:

```
fn read_file_lines(path: str) -> ~[str] {
    note path;
    let r: [str];
    let f: file = open_read(path);
    lines(f) |s| {
        r += ~[s];
    }
    ret r;
}
```

In this example, if the task fails while attempting to open or read a file, the runtime will log the path name that was being read. If the function completes normally, the runtime will not log the path.

A value that is marked by a `note` expression is *not* copied aside when control passes through the `note`. In other words, if a `note` expression notes a particular `lval`, and code after the `note` mutates that slot, and then a subsequent failure occurs, the *mutated* value will be logged during unwinding, *not* the original value that was denoted by the `lval` at the moment control passed through the `note` expression.

6.2.23 Return expressions

```
ret_expr : "ret" expr ? ;
```

Return expressions are denoted with the keyword `ret`. Evaluating a `ret` expression[4] 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 `ret` expression:

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

6.2.24 Log expressions

```
log_expr : "log" '(' level ',' expr ')' ;
```

Evaluating a `log` expression may, depending on runtime configuration, cause a value to be appended to an internal diagnostic logging buffer provided by the runtime or emitted to a system console. Log expressions are enabled or disabled dynamically at run-time on a per-task and per-item basis. See logging system.

Each `log` expression must be provided with a *level* argument in addition to the value to log. The logging level is a `u32` value, where lower levels indicate more-urgent levels of logging. By default, the lowest four logging levels (`0_u32 ... 3_u32`) are predefined as the constants `error`, `warn`, `info` and `debug` in the `core` library.

Additionally, the macros `#error`, `#warn`, `#info` and `#debug` are defined in the default syntax-extension namespace. These expand into calls to the logging facility composed with calls to the `#fmt` string formatting syntax-extension.

The following examples all produce the same output, logged at the `error` logging level:

```
// Full version, logging a value.
log(core::error, "file not found: " + filename);

// Log-level abbreviated, since core::* is imported by default.
```

[4] A `ret` expression is analogous to a `return` expression in the C family.

```
log(error, "file not found: " + filename);

// Formatting the message using a format-string and #fmt
log(error, #fmt("file not found: %s", filename));

// Using the #error macro, that expands to the previous call.
#error("file not found: %s", filename);
```

A `log` expression is *not evaluated* when logging at the specified logging-level, module or task is disabled at runtime. This makes inactive `log` expressions very cheap; they should be used extensively in Rust code, as diagnostic aids, as they add little overhead beyond a single integer-compare and branch at runtime.

Logging is presently implemented as a language built-in feature, as it makes use of compiler-provided logic for allocating the associated per-module logging-control structures visible to the runtime, and lazily evaluating arguments. In the future, as more of the supporting compiler-provided logic is moved into libraries, logging is likely to move to a component of the core library. It is best to use the macro forms of logging (*#error*, *#debug*, etc.) to minimize disruption to code using the logging facility when it is changed.

6.2.25 Check expressions

```
check_expr : "check" call_expr ;
```

A `check` expression connects dynamic assertions made at run-time to the static typestate system. A `check` expression takes a constraint to check at run-time. If the constraint holds at run-time, control passes through the `check` and on to the next expression in the enclosing block. If the condition fails to hold at run-time, the `check` expression behaves as a `fail` expression.

The typestate algorithm is built around `check` expressions, and in particular the fact that control *will not pass* a check expression with a condition that fails to hold. The typestate algorithm can therefore assume that the (static) postcondition of a `check` expression includes the checked constraint itself. From there, the typestate algorithm can perform dataflow calculations on subsequent expressions, propagating conditions forward and statically comparing implied states and their specifications.

```
pure fn even(x: int) -> bool {
    ret x & 1 == 0;
}

fn print_even(x: int) : even(x) {
    print(x);
}

fn test() {
    let y: int = 8;

    // Cannot call print_even(y) here.

    check even(y);

    // Can call print_even(y) here, since even(y) now holds.
    print_even(y);
}
```

6.2.26 Prove expressions

Note: Prove expressions are not yet supported by the compiler.

```
prove_expr : "prove" call_expr ;
```

A `prove` expression has no run-time effect. Its purpose is to statically check (and document) that its argument constraint holds at its expression entry point. If its argument typestate does not hold, under the typestate algorithm, the program containing it will fail to compile.

6.2.27 Claim expressions

```
claim_expr : "claim" call_expr ;
```

A `claim` expression is an unsafe variant on a `check` expression that is not actually checked at runtime. Thus, using a `claim` implies a proof obligation to ensure—without compiler assistance—that an assertion always holds.

Setting a runtime flag can turn all `claim` expressions into `check` expressions in a compiled Rust program, but the default is to not check the assertion contained in a `claim`. The idea behind `claim` is that performance profiling might identify a few bottlenecks in the code where actually checking a given callee's predicate is too expensive; `claim` allows the code to typecheck without removing the predicate check at every other call site.

6.2.28 If-Check expressions

An `if check` expression combines a `if` expression and a `check` expression in an indivisible unit that can be used to build more complex conditional control-flow than the `check` expression affords.

In fact, `if check` is a "more primitive" expression than `check`; instances of the latter can be rewritten as instances of the former. The following two examples are equivalent:

Example using `check`:

```
check even(x);
print_even(x);
```

Equivalent example using `if check`:

```
if check even(x) {
    print_even(x);
} else {
    fail;
}
```

6.2.29 Assert expressions

```
assert_expr : "assert" expr ;
```

An `assert` expression is similar to a `check` expression, except the condition may be any boolean-typed expression, and the compiler makes no use of the knowledge that the condition holds if the program continues to execute after the `assert`.

6.2.30 Syntax extension expressions

```
syntax_ext_expr : '#' ident paren_expr_list ? brace_match ? ;
```

Rust provides a notation for *syntax extension*. The notation for invoking a syntax extension is a marked syntactic form that can appear as an expression in the body of a Rust program.

After parsing, a syntax-extension invocation is expanded into a Rust expression. The name of the extension determines the translation performed. In future versions of Rust, user-provided syntax extensions aside from macros will be provided via external crates.

At present, only a set of built-in syntax extensions, as well as macros introduced inline in source code using the `macro` extension, may be used. The current built-in syntax extensions are:

- `fmt` expands into code to produce a formatted string, similar to `printf` from C.
- `env` expands into a string literal containing the value of that environment variable at compile-time.
- `concat_idents` expands into an identifier which is the concatenation of its arguments.
- `ident_to_str` expands into a string literal containing the name of its argument (which must be a literal).
- `log_syntax` causes the compiler to pretty-print its arguments.

Finally, `macro` is used to define a new macro. A macro can abstract over second-class Rust concepts that are present in syntax. The arguments to `macro` are pairs (two-element vectors). The pairs consist of an invocation and the syntax to expand into. An example:

```
#macro([#apply[fn, [args, ...]], fn(args, ...)]);
```

In this case, the invocation `#apply[sum, 5, 8, 6]` expands to `sum(5,8,6)`. If ... follows an expression (which need not be as simple as a single identifier) in the input syntax, the matcher will expect an arbitrary number of occurrences of the thing preceding it, and bind syntax to the identifiers it contains. If it follows an expression in the output syntax, it will transcribe that expression repeatedly, according to the identifiers (bound to syntax) that it contains.

The behaviour of `...` is known as Macro By Example. It allows you to write a macro with arbitrary repetition by specifying only one case of that repetition, and following it by `...`, both where the repeated input is matched, and where the repeated output must be transcribed. A more sophisticated example:

```
#macro([#zip_literals[[x, ...], [y, ...]), [[x, y], ...]]);
#macro([#unzip_literals[[x, y], ...], [[x, ...], [y, ...]]]);
```

In this case, `#zip_literals[[1,2,3], [1,2,3]]` expands to `[[1,1],[2,2],[3,3]]`, and `#unzip_literals[[1,1], [2,2], [3,3]]` expands to `[[1,2,3],[1,2,3]]`.

Macro expansion takes place outside-in: that is, `#unzip_literals[#zip_literals[[1,2,3],[1,2,3]]]` will fail because `unzip_literals` expects a list, not a macro invocation, as an argument.

The macro system currently has some limitations. It's not possible to destructure anything other than vector literals (therefore, the arguments to complicated macros will tend to be an ocean of square brackets). Macro invocations and `...` can only appear in expression positions. Finally, macro expansion is currently unhygienic. That is, name collisions between macro-generated and user-written code can cause unintentional capture.

Future versions of Rust will address these issues.

7 Types and typestates

7.1 Types

Every slot and value in a Rust program has a type. The *type* of a *value* defines the interpretation of the memory holding it. The type of a *slot* may also include constraints.

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. In addition, every built-in type or type-constructor name is reserved as a *keyword* in Rust; they cannot be used as user-defined identifiers in any context.

7.1.1 Primitive types

The primitive types are the following:

- The "nil" type `()`, having the single "nil" value `()`.[5]
- 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.

[5]The "nil" value `()` is *not* a sentinel "null pointer" value for reference slots; the "nil" 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`[6] 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`[7] 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.

7.1.2 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.

7.1.3 Record types

The record type-constructor forms a new heterogeneous product of values.[8] Fields of a record type are accessed by name and are arranged in memory in the order specified by the record type.

An example of a record type and its use:

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

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

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

[8] The record type-constructor is analogous to the `struct` type-constructor in the Algol/C family, the *record* types of the ML family, or the *structure* types of the Lisp family.

7.1.4 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. Single-element tuples are not legal; all tuples have two or more values.

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 = (int,str);
let p: pair = (10,"hello");
let (a, b) = p;
assert b != "world";
```

7.1.5 Vector types

The vector type-constructor represents a homogeneous array of values of a given type. A vector has a fixed size. The kind of a vector type depends on the kind of its member type, as with other simple structural types.

An example of a vector type and its use:

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

Vectors always *allocate* a storage region sufficient to store the first power of two worth of elements greater than or equal to the size of the vector. This behaviour supports idiomatic in-place “growth” of a mutable slot holding a vector:

```
let mut v: ~[int] = ~[1, 2, 3];
v += ~[4, 5, 6];
```

Normal vector concatenation causes the allocation of a fresh vector to hold the result; in this case, however, the slot holding the vector recycles the underlying storage in-place (since the reference-count of the underlying storage is equal to 1).

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

7.1.6 Enumerated types

An *enumerated type* is a nominal, heterogeneous disjoint union type.[9] An `enum` *item* consists of a number of *constructors*, each of which is independently named and takes an optional tuple of arguments.

Enumerated types cannot be denoted *structurally* as types, but must be denoted by named reference to an *enumeration* item.

7.1.7 Box types

Box types are represented as pointers. There are three flavours of pointers:

Shared boxes (`@`) These are reference-counted boxes. Their type is written `@content`, for example `@int` means a shared box containing an integer. Copying a value of such a type means copying the pointer and increasing the reference count.

Unique boxes (`~`) Unique boxes have only a single owner, and are freed when their owner releases them. They are written `~content`. Copying a unique box involves copying the contents into a new box.

Unsafe pointers (`*`) Unsafe pointers are pointers without safety guarantees or language-enforced semantics. Their type is written `*content`. They can be copied and dropped freely. Dereferencing an unsafe pointer is part of the unsafe sub-dialect of Rust.

7.1.8 Function types

The function type-constructor `fn` forms new function types. A function type consists of a sequence of input slots, an optional set of input constraints and an output slot.

An example of a `fn` type:

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

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

let mut x = add(5,7);

type binop = fn(int,int) -> int;
let bo: binop = add;
x = bo(5,7);
```

7.2 Type kinds

Types in Rust are categorized into three kinds, based on whether they allow copying of their values, and sending to different tasks. The kinds are:

Sendable Values with a sendable type can be safely sent to another task. This kind includes scalars, unique pointers, unique closures, and structural types containing only other sendable types.

Copyable This kind includes all types that can be copied. All types with sendable kind are copyable, as are shared boxes, shared closures, interface types, and structural types built out of these.

Noncopyable Resource types, and every type that includes a resource without storing it in a shared box, may not be copied. Types of sendable or copyable type can always be used in places where a noncopyable type is expected, so in effect this kind includes all types.

These form a hierarchy. The noncopyable kind is the widest, including all types in the language. The copyable kind is a subset of that, and the sendable kind is a subset of the copyable kind.

Any operation that causes a value to be copied requires the type of that value to be of copyable kind. Type parameter types are assumed to be noncopyable, unless one of the special bounds `send` or `copy` is declared for it. For example, this is not a valid program:

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

Putting `x` into a shared box involves copying, and the `T` parameter is assumed to be noncopyable. 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.

7.3 Typestate system

Rust programs have a static semantics that determine the types of values produced by each expression, as well as the *predicates* that hold over slots in the environment at each point in time during execution.

The latter semantics – the dataflow analysis of predicates holding over slots – is called the *typestate* system.

7.3.1 Points

Control flows from statement to statement in a block, and through the evaluation of each expression, from one sub-expression to another. This sequential control flow is specified as a set of *points*, each of which has a set of points before and after it in the implied control flow.

For example, this code:

```
s = "hello, world";
io::println(s);
```

Consists of 2 statements, 3 expressions and 12 points:

- the point before the first statement
- the point before evaluating the static initializer `"hello, world"`
- the point after evaluating the static initializer `"hello, world"`
- the point after the first statement
- the point before the second statement
- the point before evaluating the function value `println`
- the point after evaluating the function value `println`
- the point before evaluating the arguments to `println`

- the point before evaluating the symbol `s`
- the point after evaluating the symbol `s`
- the point after evaluating the arguments to `println`
- the point after the second statement

Whereas this code:

```
io::println(x() + y());
```

Consists of 1 statement, 7 expressions and 14 points:

- the point before the statement
- the point before evaluating the function value `println`
- the point after evaluating the function value `println`
- the point before evaluating the arguments to `println`
- the point before evaluating the arguments to `+`
- the point before evaluating the function value `x`
- the point after evaluating the function value `x`
- the point before evaluating the arguments to `x`
- the point after evaluating the arguments to `x`
- the point before evaluating the function value `y`
- the point after evaluating the function value `y`
- the point before evaluating the arguments to `y`
- the point after evaluating the arguments to `y`
- the point after evaluating the arguments to `+`
- the point after evaluating the arguments to `println`

The typestate system reasons over points, rather than statements or expressions. This may seem counter-intuitive, but points are the more primitive concept. Another way of thinking about a point is as a set of *instants in time* at which the state of a task is fixed. By contrast, a statement or expression represents a *duration in time*, during which the state of the task changes. The typestate system is concerned with constraining the possible states of a task's memory at *instants*; it is meaningless to speak of the state of a task's memory "at" a statement or expression, as each statement or expression is likely to change the contents of memory.

7.3.2 Control flow graph

Each *point* can be considered a vertex in a directed *graph.* Each kind of expression or statement implies a number of points *and edges* in this graph. The edges connect the points within each statement or expression, as well as between those points and those of nearby statements and expressions in the program. The edges between points represent *possible* indivisible control transfers that might occur during execution.

This implicit graph is called the *control-flow graph*, or *CFG*.

7.3.3 Constraints

A *predicate* is a pure boolean function declared with the keywords `pure fn`.

A *constraint* is a predicate applied to specific slots.

For example, consider the following code:

```
pure fn is_less_than(a: int, b: int) -> bool {
     ret a < b;
}

fn test() {
   let x: int = 10;
   let y: int = 20;
   check is_less_than(x,y);
}
```

This example defines the predicate `is_less_than`, and applies it to the slots `x` and `y`. The constraint being checked on the third line of the function is `is_less_than(x,y)`.

Predicates can only apply to slots holding immutable values. The slots a predicate applies to can themselves be mutable, but the types of values held in those slots must be immutable.

7.3.4 Conditions

A *condition* is a set of zero or more constraints.

Each *point* has an associated *condition*:

- The *precondition* of a statement or expression is the condition required at in the point before it.

- The *postcondition* of a statement or expression is the condition enforced in the point after it.

Any constraint present in the precondition and *absent* in the postcondition is considered to be *dropped* by the statement or expression.

7.3.5 Calculated typestates

The typestate checking system *calculates* an additional condition for each point called its *typestate*. For a given statement or expression, we call the two typestates associated with its two points the prestate and a poststate.

- The *prestate* of a statement or expression is the typestate of the point before it.
- The *poststate* of a statement or expression is the typestate of the point after it.

A *typestate* is a condition that has *been determined by the typestate algorithm* to hold at a point. This is a subtle but important point to understand: preconditions and postconditions are *inputs* to the typestate algorithm; prestates and poststates are *outputs* from the typestate algorithm.

The typestate algorithm analyses the preconditions and postconditions of every statement and expression in a block, and computes a condition for each typestate. Specifically:

- Initially, every typestate is empty.
- Each statement or expression's poststate is given the union of the its prestate, precondition, and postcondition.
- Each statement or expression's poststate has the difference between its precondition and postcondition removed.
- Each statement or expression's prestate is given the intersection of the poststates of every predecessor point in the CFG.
- The previous three steps are repeated until no typestates in the block change.

The typestate algorithm is a very conventional dataflow calculation, and can be performed using bit-set operations, with one bit per predicate and one bit-set per condition.

After the typestates of a block are computed, the typestate algorithm checks that every constraint in the precondition of a statement is satisfied by its prestate. If any preconditions are not satisfied, the mismatch is considered a static (compile-time) error.

7.3.6 Typestate checks

The key mechanism that connects run-time semantics and compile-time analysis of typestates is the use of `check` expressions. A `check` expression guarantees that *if* control were to proceed past it, the predicate associated with the `check` would have succeeded, so the constraint being checked *statically* holds in subsequent points.^[A `check` expression is similar to an `assert` call in a C program, with the significant difference that the Rust compiler *tracks* the constraint that each `check` expression enforces. Naturally, `check` expressions cannot be omitted from a "production build" of a Rust program the same way `asserts` are frequently disabled in deployed C programs.}

It is important to understand that the typestate system has *no insight* into the meaning of a particular predicate. Predicates and constraints are not evaluated in any way at compile time. Predicates are treated as specific (but unknown) functions applied to specific (also unknown) slots. All the typestate system does is track which of those predicates – whatever they calculate – *must have been checked already* in order for program control to reach a particular point in the CFG. The fundamental building block, therefore, is the `check` statement, which tells the typestate system "if control passes this point, the checked predicate holds".

From this building block, constraints can be propagated to function signatures and constrained types, and the responsibility to `check` a constraint pushed further and further away from the site at which the program requires it to hold in order to execute properly.

8 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.

8.1 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*.

8.1.1 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: shared boxes – which may be subject to garbage collection – and unique 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.

8.1.2 Memory ownership

A task owns all memory it can *safely* reach through local variables, shared or unique boxes, and/or references. Sharing memory between tasks can only be accomplished using *unsafe* constructs, such as raw pointer operations or calling C code.

When a task sends a value satisfying the `send` interface over a channel, 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` interface: it is only instantiated for (transitively) unique kinds of data constructor and pointers, never shared pointers.

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

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

8.1.3 Memory slots

A task's stack contains slots.

A *slot* is a component of a stack frame. A slot is either a *local variable* or a *reference*.

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.

A *reference* references a value outside the frame. It may refer to a value allocated in another frame *or* a boxed value in the heap. The reference-formation rules ensure that the referent will outlive the reference.

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`).

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 condition is guaranteed by the typestate system.

References are created for function arguments. If the compiler can not prove that the referred-to value will outlive the reference, it will try to set aside a copy of that value to refer to. If this is not semantically safe (for example, if the referred-to value contains mutable fields), it will reject the program. If the compiler deems copying the value expensive, it will warn.

A function can be declared to take an argument by mutable reference. This allows the function to write to the slot that the reference refers to.

An example function that accepts an value by mutable reference:

```
fn incr(&i: int) {
    i = i + 1;
}
```

8.1.4 Memory boxes

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

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

A *unique box* type or value is constructed by the prefix *tilde* sigil `~`.

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

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

An example of constructing one shared box type and value, and one unique 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:

```
let x = @{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);
}
```

8.2 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.

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 a number of threads is used based on the number of concurrent physical CPUs detected at startup, but this can be changed dynamically at runtime. When the number of tasks exceeds the number of threads – which is quite possible – the tasks are multiplexed onto the threads [10]

8.2.1 Communication between tasks

With the exception of *unsafe* blocks, Rust tasks are isolated from interfering with one another's memory directly. Instead of manipulating shared storage, Rust tasks communicate with one another using a typed, asynchronous, simplex message-passing system.

A *port* is a communication endpoint that can *receive* messages. Ports receive messages from channels.

A *channel* is a communication endpoint that can *send* messages. Channels send messages to ports.

Each port is implicitly boxed and mutable; as such a port has a unique per-task identity and cannot be replicated or transmitted. If a port value is copied, both copies refer to the *same* port. New ports can be constructed dynamically and stored in data structures.

Each channel is bound to a port when the channel is constructed, so the destination port for a channel must exist before the channel itself. A channel cannot be rebound to a different port from the one it was constructed with.

Channels are weak: a channel does not keep the port it is bound to alive. Ports are owned by their allocating task and cannot be sent over channels; if a task dies its ports die with it, and all channels bound to those ports no longer function. Messages sent to a channel connected to a dead port will be dropped.

Channels are immutable types with meaning known to the runtime; channels can be sent over channels.

Many channels can be bound to the same port, but each channel is bound to a single port. In other words, channels and ports exist in an N:1 relationship, N channels to 1 port. [11]

[10]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 as tasks can give better results. The M:N scheduling in Rust exists to support very large numbers of tasks in contexts where threads are too resource-intensive to use in a similar volume. The cost of threads varies substantially per operating system, and is sometimes quite low, so this flexibility is not always worth exploiting.

[11]It may help to remember nautical terminology when differentiating channels from ports. Many different waterways – channels – may lead to the same port.

Each port and channel can carry only one type of message. The message type is encoded as a parameter of the channel or port type. The message type of a channel is equal to the message type of the port it is bound to. The types of messages must satisfy the `send` built-in interface.

Messages are generally sent asynchronously, with optional rate-limiting on the transmit side. Each port contains a message queue and sending a message over a channel merely means inserting it into the associated port's queue; message receipt is the responsibility of the receiving task.

Messages are sent on channels and received on ports using standard library functions.

8.2.2 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 receive call on a port, or attempts a rate-limited blocking send on a channel. When the communication expression can be completed – when a message arrives at a sender, or a queue drains sufficiently to complete a rate-limited send – 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` expression. 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.

8.2.3 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 `core::task::yield`, which deschedules it immediately. Entering any other non-executing state (blocked, dead) similarly deschedules the task.

8.2.4 Spawning tasks

A call to `core::task::spawn`, passing a 0-argument function as its single argument, causes the runtime to construct a new task executing the passed function. The passed function is referred to as the *entry function* for the spawned task, and any captured environment is carries is moved from the spawning task to the spawned task before the spawned task begins execution.

The result of a `spawn` call is a `core::task::task` value.

An example of a `spawn` call:

```
let po = comm::port();
let ch = comm::chan(po);

do task::spawn {
    // let task run, do other things
    // ...
    comm::send(ch, true);
};

let result = comm::recv(po);
```

8.2.5 Sending values into channels

Sending a value into a channel is done by a library call to `core::comm::send`, which takes a channel and a value to send, and moves the value into the channel's outgoing buffer.

An example of a send:

```
let po = comm::port();
let ch = comm::chan(po);
comm::send(ch, "hello, world");
```

8.2.6 Receiving values from ports

Receiving a value is done by a call to the `recv` method on a value of type `core::comm::port`. This call causes the receiving task to enter the *blocked reading* state until a value arrives in the port's receive queue, at which time the port deques a value to return, and un-blocks the receiving task.

An example of a *receive*:

```
let s = comm::recv(po);
```

9 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.

9.0.7 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 in use. 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 boxed values.

9.0.8 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.

9.0.9 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.

9.0.10 Logging system

The runtime contains a system for directing logging expressions to a logging console and/or internal logging buffers. Logging expressions 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 0 to 3, inclusive. Level 0 is the error level, 1 is warning, 2 info, and 3 debug. Any logs less than or equal to the specified level will be output. If not specified then log level 3 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 psuedo-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` psuedo-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

10 Appendix: Rationales and design tradeoffs

TODO.

11 Appendix: Influences and further references

11.1 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 deterministic destructor system of C++.
- The typeclass system of Haskell.
- The lexical identifier rule of Python.
- The block syntax of Ruby.