`plugin`

The tracking issue for this feature is: #29597

This feature is part of "compiler plugins." It will often be used with the `rustc_private` feature.

`rustc` can load compiler plugins, which are user-provided libraries that extend the compiler's behavior with new lint checks, etc.

A plugin is a dynamic library crate with a designated *registrar* function that registers extensions with `rustc`. Other crates can load these extensions using the crate attribute `#![plugin(...)]`. See the `rustc_driver::plugin` documentation for more about the mechanics of defining and loading a plugin.

In the vast majority of cases, a plugin should *only* be used through `#![plugin]` and not through an `extern crate` item. Linking a plugin would pull in all of librustc_ast and librustc as dependencies of your crate. This is generally unwanted unless you are building another plugin.

The usual practice is to put compiler plugins in their own crate, separate from any `macro_rules!` macros or ordinary Rust code meant to be used by consumers of a library.

Lint plugins

Plugins can extend Rust's lint infrastructure with additional checks for code style, safety, etc. Now let's write a plugin `lint-plugin-test.rs` that warns about any item named `lintme`.

```
#![feature(box_syntax, rustc_private)]

extern crate rustc_ast;

// Load rustc as a plugin to get macros
extern crate rustc_driver;
#[macro_use]
extern crate rustc_lint;
#[macro_use]
extern crate rustc_session;

use rustc_driver::plugin::Registry;
use rustc_lint::{EarlyContext, EarlyLintPass, LintArray, LintContext, LintPass};
use rustc_ast::ast;
declare_lint!(TEST_LINT, Warn, "Warn about items named 'lintme'");

declare_lint_pass!(Pass => [TEST_LINT]);

impl EarlyLintPass for Pass {
    fn check_item(&mut self, cx: &EarlyContext, it: &ast::Item) {
        if it.ident.name.as_str() == "lintme" {
            cx.lint(TEST_LINT, |lint| {
                lint.build("item is named 'lintme'").set_span(it.span).emit()
            });
        }
    }
}

#[no_mangle]
fn __rustc_plugin_registrar(reg: &mut Registry) {
    reg.lint_store.register_lints(&[&TEST_LINT]);
    reg.lint_store.register_early_pass(|| box Pass);
}
```

Then code like

```
#![feature(plugin)]
#![plugin(lint_plugin_test)]

fn lintme() { }
```

will produce a compiler warning:

```
foo.rs:4:1: 4:16 warning: item is named 'lintme', #[warn(test_lint)] on by default
foo.rs:4 fn lintme() { }
         ^~~~~~~~~~~~~~
```

The components of a lint plugin are:

- one or more `declare_lint!` invocations, which define static `Lint` structs;
- a struct holding any state needed by the lint pass (here, none);
- a `LintPass` implementation defining how to check each syntax element. A single `LintPass` may call `span_lint` for several different `Lint` s, but should register them all through the `get_lints` method.

Lint passes are syntax traversals, but they run at a late stage of compilation where type information is available. `rustc` 's built-in lints mostly use the same infrastructure as lint plugins, and provide examples of how to access type information.

Lints defined by plugins are controlled by the usual attributes and compiler flags, e.g. `#[allow(test_lint)]` or `-A test-lint` . These identifiers are derived from the first argument to `declare_lint!` , with appropriate case and punctuation conversion.

You can run `rustc -W help foo.rs` to see a list of lints known to `rustc` , including those provided by plugins loaded by `foo.rs` .