LALRPOP

LALRPOP is a parser generator, similar in principle to YACC, ANTLR, Menhir, and other such programs. In general, it has the grand ambition of being the most usable parser generator ever. This ambition is most certainly not fully realized: right now, it's fairly standard, maybe even a bit subpar in some areas. But hey, it's young. For the most part, this README is intended to describe the current behavior of LALRPOP, but in some places it includes notes for planned future changes.

Crash course on parsers

If you've never worked with a parser generator before, or aren't really familiar with context-free grammars, this section is just a *very brief* introduction into the basic idea. Basically a grammar is a nice way of writing out what kinds of inputs are legal. In our example, we want to support parenthesized numbers, so things like `123`, `(123)`, etc. We can express this with a simple grammar like:

```
Term = Num | "(" Term ")"
```

Here we say we are trying to parse a *term*, and a term can either be a number (`Num`) or some other term enclosing in parentheses (here I did not define what a number is, but in the real LALRPOP example we'll do that with a regular expression). Now imagine a potential input like `((123))`. We can show how this would be parsed by writing out something called a "parse tree":

```
(   (   1   2   3   )   )
|   |   |       |   |   |
|   |   +-Num-+    |   |
|   |     |         |   |
|   |    Term       |   |
|   |     |         |   |
|   +---Term----+   |
|         |             |
+-------Term-------+
```

Here you can see that we parsed `((123))` by finding a `Num` in the middle, calling that `Num` a `Term`, and matching up the parentheses to form two more terms on top of that.

Note that this parse tree is not a data structure but more a visualization of the parse. I mean, you *can* build up a parse tree as a data structure, but typically you don't want to: it is more detailed than you need. For example, you may not be that interested in the no-op conversion from a `Num` to a `Term`. The other weird thing about a parse tree is that it is intimately tied to your grammar, but often you have some existing data structures you would like to parse into -- so if you built up a parse tree, you'd then have to convert from the parse tree into those data structures, and that might be annoying.

Therefore, what a parser generator usually does, is instead let you choose how to represent each node in the parse tree, and how to do the conversions. You give each nonterminal a type, which can be any Rust type, and you write code that will execute each time a new node in the parse tree would have been constructed. In fact, in the examples that follow, we'll eventually build up something like a parse tree, but in the beginning, we won't do that at all. Instead, we'll represent each number and term as an `i32`, and we'll propagate this value around.

To make this a bit more concrete, here's a version of the grammar above written in LALRPOP notation (we'll revisit this again in more detail of course). You can see that the `Term` nonterminal has been given the type `i32`, and that each of the definitions has some code that follows a `=>` symbol. This is the code that will execute to convert from the thing that was matched (like a number, or a parenthesized term) into an `i32`:

```
Term: i32 = {
    Num => /* ... number code ... */,
    "(" Term ")" => /* ... parenthesized code ... */,
};
```

OK, that's enough background, let's do this for real!

For getting started with LALRPOP, it's probably best if you read the tutorial, which will introduce you to the syntax of LALRPOP files and so forth.

But if you've done this before, or you're just the impatient sort, here is a quick 'cheat sheet' for setting up your project. First, add the following lines to your `Cargo.toml`:

```
[package]
...
build = "build.rs" # LALRPOP preprocessing

# The generated code depends on lalrpop-util.
#
# The generated tokenizer depends on the regex crate.
#
# (If you write your own tokenizer, or already have the regex
# crate, you can skip this dependency.)
[dependencies]
lalrpop-util = "0.16.2"
regex = "0.2.0"

# Add a build-time dependency on the lalrpop library:
[build-dependencies]
lalrpop = "0.16.2"
```

Next create a `build.rs` file that looks like:

```
extern crate lalrpop;

fn main() {
    lalrpop::process_root().unwrap();
}
```

(If you already have a `build.rs` file, you should be able to just call `process_root` in addition to whatever else that file is doing.)

That's it! Note that `process_root` simply uses the default settings. If you want to configure how LALRPOP executes, see the advanced setup section.

Running manually

If you prefer, you can also run the `lalrpop` crate as an executable. Simply run `cargo install lalrpop` and then you will get a `lalrpop` binary you can execute, like so:

```
lalrpop file.lalrpop
```

This will generate `file.rs` for you. Note that it only executes if `file.lalrpop` is newer than `file.rs`; if you'd prefer to execute unconditionally, pass `-f` (also try `--help` for other options).

This is a tutorial for how to write a complete parser for a simple calculator using

LALRPOP.

If you are unfamiliar with what a parser generator is, you should read Crash course on parsers first.

This tutorial is still incomplete. Here are some topics that I aim to cover when I get time to write about them:

- Advice for resolving shift-reduce and reduce-reduce conflicts
- Passing state and type/lifetime parameters to your action code (see e.g. this test invoked from here).
- Location tracking with `@L` and `@R` (see e.g. this test).
- Integrating with external tokenizers (see e.g. this test invoked from here).
- Conditional macros (no good test to point you at yet, sorry)
- Fallible action code that produces a `Result` (see e.g. this test invoked from here).
- Converting to use `LALR(1)` instead of `LR(1)` (see e.g. this test invoked from here).
- Plans for future features

Adding LALRPOP to your `Cargo.toml` file

LALRPOP works as a preprocessor that is integrated with cargo. When LALRPOP is invoked, it will search your source directory for files with the extension `lalrpop` and create corresponding `rs` files. So, for example, if we have a file `calculator.lalrpop`, the preprocessor will create a Rust file `calculator.rs`. As an aside, the syntax of LALRPOP intentionally hews fairly close to Rust, so it should be possible to use the Rust plugin to edit lalrpop files as well, as long as it's not too picky (the emacs rust-mode, in particular, works just fine).

To start, let's use `cargo new` to make a new project. We'll call it `calculator`:

```
> cargo new --bin calculator
```

We now have to edit the generated `calculator/Cargo.toml` file to invoke the LALRPOP preprocessor. The resulting file should look something like:

```
[package]
name = "calculator"
version = "0.1.0"
authors = ["Niko Matsakis <niko@alum.mit.edu>"]

[build-dependencies] # <-- We added this and everything after!
lalrpop = "0.16.2"

[dependencies]
lalrpop-util = "0.16.2"
regex = "0.2.1"
```

Cargo can run build scripts as a pre-processing step, named `build.rs` by default. The `[build-dependencies]` section specifies the dependencies for build scripts -- in this case, just LALRPOP.

The `[dependencies]` section describes the dependencies that LALRPOP needs at runtime. All LALRPOP parsers require at least the `lalrpop-util` crate. In addition, if you don't want to write the lexer by hand, you need to add a dependency on the regex crate. (If you don't know what a lexer is, don't worry, it's not important just now, though we will cover it in the next section; if you *do* know what a lexer is, and you want to know how to write a lexer by hand and use it with LALRPOP, then check out the lexer tutorial.)

Next we have to add `build.rs` itself. This should just look like the following:

```
extern crate lalrpop;

fn main() {
    lalrpop::process_root().unwrap();
}
```

The function `process_root` processes your `src` directory, converting all `lalrpop` files into `rs` files. It is smart enough to check timestamps and do nothing if the `rs` file is newer than the `lalrpop` file, and to mark the generated `rs` file as read-only. It returns an `io::Result<()>`, so the `unwrap()` call just asserts that no file-system errors occurred.

NOTE: On Windows, the necessary APIs are not yet stable, so timestamp checking is disabled.

Parsing parenthesized numbers

OK, now we're all set to start making a LALRPOP grammar. Before we tackle full expressions, let's start with something simple -- really simple. Let's just start with parenthesized integers, like `123` or `(123)` or even (hold on to your hats) `(((123)))`. Wow.

To handle this, we'll need to add a `calculator1.lalrpop` as shown below. Note: to make explaining things easier, this version is maximally explicit; the next section will make it shorter by employing some shorthands that LALRPOP offers.

```
use std::str::FromStr;

grammar;

pub Term: i32 = {
    <n:Num> => n,
    "(" <t:Term> ")" => t,
};

Num: i32 = <s:r"[0-9]+"> => i32::from_str(s).unwrap();
```

Let's look at this bit by bit. The first part of the file is the `use` statement and the `grammar` declaration. You'll find these at the top of every LALRPOP grammar. Just as in Rust, the `use` statement just brings names in scope: in fact, these `use` statements are just copied verbatim into the generated Rust code as needed.

A note about underscores and hygiene: LALRPOP generates its own names that begin with at least two leading underscores. To avoid conflicts, it will insert more underscores if it sees that you use identifiers that also have two underscores. But if you use glob imports that bring in names beginning with `__`, you may find you have invisible conflicts. To avoid this, don't use a glob (or define some other name with two underscores somewhere else).

Nonterminal declarations. After the `grammar` declaration comes a series of *nonterminal declarations*. This grammar has two nonterminals, `Term` and `Num`. A nonterminal is just a name that we give to something which can be parsed. Each nonterminal is then defined in terms of other things.

Let's start with `Num`, at the end of the file, which is declared as follows:

```
Num: i32 =
    <s:r"[0-9]+"> => i32::from_str(s).unwrap();
```

This declaration says that the type of `Num` is `i32`. This means that when we parse a `Num` from the input text, we will produce a value of type `i32`. The definition of `Num` is `<s:r"[0-9]+">`. Let's look at this from the inside out. The notation `r"[0-9]+"` is a regex literal -- this is the same as a Rust raw string. (And, just as in Rust, you can use hashes if you need to embed quotes, like `r#"..."..."#`.) It will match against a string of characters that matches the regular expression: in this case, some number of digits. The result of this match will be a slice `&'input str` into the input text that we are parsing (no copies are made).

This regular expression is wrapped in angle brackets and labeled: `<s:r"[0-9]+">`. In general, angle brackets are used in LALRPOP to indicate the values that will be used by the *action code* -- that is, the code that executes when a `Num` is parsed. In this case, the string that matches the regular expression is bound to the name `s`, and the action code `i32::from_str(s).unwrap()` parses that string and creates an `i32`. Hence the result of parsing a `Num` is an `i32`.

OK, now let's look at the nonterminal `Term`:

```
pub Term: i32 = {
    <n:Num> => n,
    "(" <t:Term> ")" => t,
};
```

First, this nonterminal is declared as `pub`. That means that LALRPOP will generate a public struct (named, as we will see, `TermParser`) that you can use to parse strings as `Term`. Private nonterminals (like `Num`) can only be used within the grammar itself, not from outside.

The `Term` nonterminal has two alternative definitions, which is indicated by writing `{ alternative1, alternative2 }`. In this case, the first alternative is `<n:Num>`, meaning that a term can be just a number; so `22` is a term. The second alternative is `"(" <t:Term> ")"`, which indicates that a term can also be a parenthesized term; so `(22)` is a term, as is `((22))`, `((((((22))))))`, and so on.

Invoking the parser. OK, so we wrote our parser, how do we use it? For every nonterminal `Foo` declared as `pub`, LALRPOP will export a `FooParser` struct with a `parse` method that you can call to parse a string as that nonterminal. Here is a simple test that we've added to our [`main.rs`][main] file which uses this struct to test our `Term` nonterminal:

```
#[macro_use] extern crate lalrpop_util;

lalrpop_mod!(pub calculator1); // synthesized by LALRPOP

#[test]
fn calculator1() {
    assert!(calculator1::TermParser::new().parse("22").is_ok());
    assert!(calculator1::TermParser::new().parse("(22)").is_ok());
    assert!(calculator1::TermParser::new().parse("((((22))))").is_ok());
    assert!(calculator1::TermParser::new().parse("((22)").is_err());
}
```

The full signature of the parse method looks like this:

```
fn parse<'input>(&self, input: &'input str)
                     -> Result<i32, ParseError<usize,(usize, &'input
str),()>>
                     //         ~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                     //          |                        |
                     // Result upon success               |
                     //                                   |
                     //             Error enum defined in the
lalrpop_util crate
{
    ...
}
```

Type inference

OK, now that we understand the calculator1 example, let's look at some of the shorthands that LALRPOP offers to make it more concise. This code is found in the calculator2 demo.

To start, let's look at the definition of `Term` we saw before:

```
pub Term: i32 = {
    <n:Num> => n,
    "(" <t:Term> ")" => t,
};
```

The action code here is somewhat interesting. In both cases, it's not doing any new work, it's just selecting a value that was produced by another nonterminal. This turns out to be pretty common. So common, in fact, that LALRPOP offers some shorthand notation for it. Here is the definition of `Term` from the calculator2 demo:

```
pub Term = { Num, "(" <Term> ")" };
```

Here, we have no action code at all. If there is no action code, LALRPOP synthesizes action code which just takes the value of the things being matched. In the case of the first alternative, `Num`, there is only one thing being matched, so that means that `Term` will produce the same value as the `Num` we parsed, whatever that was.

In the case of the second alternative, `"(" <Term> ")"`, there are three things being matched. Here we use the angle brackets to select which item(s) we want to take the value of --- we selected only one, so the result is that we take the value of the `Term` we parsed. If we selected more than one, the result would be a tuple of all the selected items. If we did not select any (i.e., `"(" Term ")"`), the result would be a tuple of all the items, and hence the result would be of type `(&'input str, i32, &'input str)`.

Speaking of types, you may have noticed that `Term` has no type annotation. Since we didn't write out own action code, we can omit the type annotation and let LALRPOP infer it for us. In this case, LALRPOP can see that `Term` must have the same type as `Num`, and hence that the type must be `i32`.

OK, let's look at the definition of `Num` we saw before from calculator1:

```
Num: i32 = <s:r"[0-9]+"> => i32::from_str(s).unwrap();
```

This definition too can be made somewhat shorter. In calculator2, you will find:

```
Num: i32 = r"[0-9]+" => i32::from_str(<>).unwrap();
```

Here, instead of giving the regular expression a name `s`, we modified the action code to use the funky expression `<>`. This is a shorthand that says "synthesize names for the matched values and insert a comma-separated list here". In this case, there is only one matched value, `r"[0-9]+"`, and it produces a `&'input str`, so LALRPOP will insert a synthetic variable for that value. Note that we still have custom action code, so we still need a type annotation.

To control what values are selected when you use the `<>` expression in your action code, you can use angle brackets as we saw before. Here are some examples of alternatives and how they are expanded to give you the idea:

Alternative	Equivalent to
`A => bar(<>)`	`<a:A> => bar(a)`
`A B => bar(<>)`	`<a:A> <b:B> => bar(a, b)`
`A B => (<>)`	`<a:A> <b:B> => (a, b)`
`<A> B => bar(<>)`	`<a:A> B => bar(a)`
`<p:A> B => bar(<>)`	`<p:A> B => bar(p)`
`<A> <B> => bar(<>)`	`<a:A> <b:B> => bar(a, b)`
`<p:A> <q:B> => bar(<>)`	`<p:A> <q:B> => bar(p, q)`
`<p:A> B => Foo {<>}`	`<p:A> B => Foo {p:p}`
`<p:A> <q:B> => Foo {<>}`	`<p:A> <q:B> => Foo {p:p, q:q}`

The `<>` expressions also works with struct constructors (like `Foo {...}` in examples above). This works out well if the names of your parsed values match the names of your struct fields.

Controlling the lexer

This example dives a bit deeper into how LALRPOP works. In particular, it dives into the meaning of those strings and regular expression that we used in the previous tutorial, and how they are used to process the input string (a process which you can control). This first step of breaking up the input using regular expressions is often called **lexing** or **tokenizing**.

If you're comfortable with the idea of a lexer or tokenizer, you may wish to skip ahead to the calculator3 example, which covers parsing bigger expressions, and come back here only when you find you want more control. You may also be interested in the tutorial on writing a custom lexer.

Terminals vs nonterminals

You may have noticed that our grammar included two distinct kinds of symbols. There were the nonterminals, `Term` and `Num`, which we defined by specifying a series of symbols that they must match, along with some action code that should execute once they have matched:

```
   Num: i32 = r"[0-9]+" => i32::from_str(<>).unwrap();
// ~~~  ~~~   ~~~~~~~~~    ~~~~~~~~~~~~~~~~~~~~~~~~~~
// |    |     |                Action code
// |    |     Symbol(s) that should match
// |    Return type
// Name of nonterminal
```

But there are also **terminals**, which consist of the string literals and regular expressions sprinkled throughout the grammar. (Terminals are also often called **tokens**, and I will use the terms interchangeably.)

This distinction between terminals and nonterminals is very important to how LALRPOP works. In fact, when LALRPOP generates a parser, it always works in a two-phase process. The first phase is called the **lexer** or **tokenizer**. It has the job of figuring out the sequence of **terminals**: so basically it analyzes the raw characters of your text and breaks them into a series of terminals. It does this without having any idea about your grammar or where you are in your grammar. Next, the parser proper is a bit of code that looks at this stream of tokens and figures out which nonterminals apply:

```
        +-------------------+    +---------------------+
Text -> | Lexer             | -> | Parser              |
        |                   |    |                     |
        | Applies regex to  |    | Consumes terminals, |
        | produce terminals |    | executes your code  |
        +-------------------+    | as it recognizes    |
                                 | nonterminals        |
                                 +---------------------+
```

LALRPOP's default lexer is based on regular expressions. By default, it works by extracting all the terminals (e.g., `"("` or `r"\d+"`) from your grammar and compiling them into one big list. At runtime, it will walk over the string and, at each point, find the longest match from the literals and regular expressions in your grammar and produces one of those. As an example, let's look again at our example grammar:

```
pub Term: i32 = {
    <n:Num> => n,
    "(" <t:Term> ")" => t,
};

Num: i32 = <s:r"[0-9]+"> => i32::from_str(s).unwrap();
```

This grammar in fact contains three terminals:

- `"("` -- a string literal, which must match exactly
- `")"` -- a string literal, which must match exactly
- `r"[0-9]+"` -- a regular expression

When we generate a lexer, it is effectively going to be checking for each of these three terminals in a loop, sort of like this pseudocode:

```
let mut i = 0; // index into string
loop {
    skip whitespace; // we do this implicitly, at least by default
    if (data at index i is "(") { produce "("; }
    else if (data at index i is ")") { produce ")"; }
    else if (data at index i matches regex "[0-9]+") { produce
r"[0-9]+"; }
}
```

Note that this has nothing to do with your grammar. For example, the tokenizer would happily tokenize a string like this one, which doesn't fit our grammar:

```
  (  22   44  )     )
  ^  ^^   ^^  ^     ^
  |  |    |   |     ")" terminal
  |  |    |   |
  |  |    |   ")" terminal
  |  +----+
  |  |
  |  2 r"[0-9]+" terminals
  |
  "(" terminal
```

When these tokens are fed into the **parser**, it would notice that we have one left paren but then two numbers (`r"[0-9]+"` terminals), and hence report an error.

Precedence of fixed strings

Terminals in LALRPOP can be specified (by default) in two ways. As a fixed string (like `"("`) or a regular expression (like `r[0-9]+`). There is actually an important difference: if, at some point in the input, both a fixed string **and** a regular expression could match, LALRPOP gives the fixed string precedence. To demonstrate this, let's modify our parser. If you recall, the current parser parses parenthesized numbers, producing a `i32` . We're going to modify if to produce a **string**, and we'll add an "easter egg" so that `22` (or `(22)` , `((22))` , etc) produces the string `"Twenty-two"` :

```
pub Term = {
    Num,
    "(" <Term> ")",
    "22" => format!("Twenty-two!"),
};

Num: String = r"[0-9]+" => <>.to_string();
```

If we write some simple unit tests, we can see that in fact an input of `22` has matched the string literal. Interestingly, the input `222` matches the regular expression instead; this is because LALRPOP prefers to find the **longest** match first. After that, if there are two matches of equal length, it prefers the fixed string:

```
#[test]
fn calculator2b() {
    let result = calculator2b::TermParser::new().parse("33").unwrap();
    assert_eq!(result, "33");

    let result = calculator2b::TermParser::new().parse("(22)").unwrap();
    assert_eq!(result, "Twenty-two!");

    let result = calculator2b::TermParser::new().parse("
(222)").unwrap();
    assert_eq!(result, "222");
}
```

Ambiguities between regular expressions

In the previous section, we saw that fixed strings have precedence over regular expressions. But what if we have two regular expressions that can match the same input? Which one wins? For example, consider this variation of the grammar above, where we also try to support parenthesized **identifiers** like `((foo22))`:

```
pub Term = {
    Num,
    "(" <Term> ")",
    "22" => format!("Twenty-two!"),
    r"\w+" => format!("Id({})", <>), // <-- we added this
};

Num: String = r"[0-9]+" => <>.to_string();
```

Here I've written the regular expression `r\w+`. However, if you check out the docs for regex, you'll see that `\w` is defined to match alphabetic characters but also digits. So there is actually an ambiguity here: if we have something like `123`, it could be considered to match either `r"[0-9]+"` **or** `r"\w+"`. If you try this grammar, you'll find that LALRPOP helpfully reports an error:

```
error: ambiguity detected between the terminal `r#"\w+"#` and the
terminal `r#"[0-9]+"#`

      r"\w+" => <>.to_string(),
      ~~~~~~
```

There are various ways to fix this. We might try adjusting our regular expression so that the first character cannot be a number, so perhaps something like `r"[[:alpha:]]\w*"`. This will work, but it actually matches something different than what we had before (e.g., `123foo` will not be considered to match, for better or worse). And anyway it's not always convenient to make your regular expressions completely disjoint like that. Another option is to use a `match` declaration, which lets you control the precedence between regular expressions.

Simple `match` declarations

A `match` declaration lets you explicitly give the precedence between terminals. In its simplest form, it consists of just ordering regular expressions and string literals into groups, with the higher precedence items coming first. So, for example, we could resolve our conflict above by giving `r"[0-9]+"` **precedence** over `r"\w+"`, thus saying that if something can be lexed as a number, we'll do that, and otherwise consider it to be an identifier.

```
match {
    r"[0-9]+"
} else {
    r"\w+",
    _
}
```

Here the match contains two levels; each level can have more than one item in it. The top-level contains only `r"[0-9]+"`, which means that this regular expression is given highest priority. The next level contains `r\w+`, so that will match afterwards.

The final `_` indicates that other string literals and regular expressions that appear elsewhere in the grammar (e.g., `"("` or `"22"`) should be added into that final level of precedence (without an `_`, it is illegal to use a terminal that does not appear in the match declaration).

If we add this `match` section into our example, we'll find that it compiles, but it doesn't work exactly like we wanted. Let's update our unit test a bit to include some identifier examples::

```
#[test]
fn calculator2b() {
    // These will all work:

    let result = calculator2b::TermParser::new().parse("33").unwrap();
    assert_eq!(result, "33");

    let result =
calculator2b::TermParser::new().parse("foo33").unwrap();
    assert_eq!(result, "Id(foo33)");

    let result = calculator2b::TermParser::new().parse("
(foo33)").unwrap();
    assert_eq!(result, "Id(foo33)");

    // This one will fail:

    let result = calculator2b::TermParser::new().parse("(22)").unwrap();
    assert_eq!(result, "Twenty-two!");
}
```

The problem comes about when we parse `22`. Before, the fixed string `22` got precedence, but with the new match declaration, we've explicitly stated that the regular expression `r"[0-9]+"` has full precedence. Since the `22` is not listed explicitly, it gets added at the last level, where the `_` appears. We can fix this by adjusting our `match` to mention `22` explicitly:

```
match {
    r"[0-9]+",
    "22"
} else {
    r"\w+",
    _
}
```

This raises the interesting question of what the precedence is **within** a match rung -- after all, both the regex and `"22"` can match the same string. The answer is that within a match rung, fixed literals get precedence over regular expressions, just as before, and all regular expressions must not overlap.

With this new `match` declaration, we will find that our tests all pass.

Renaming `match` declarations

There is one final twist before we reach the final version of our example that you will find in the repository. We can also use `match` declarations to give names to regular expressions, so that we don't have to type them directly in our grammar. For example, maybe instead of writing `r"\w+"`, we would prefer to write `ID`. We could do that by modifying the match declaration like so:

```
match {
    r"[0-9]+",
    "22"
} else {
    r"\w+" => ID, // <-- give a name here
    _
}
```

And then adjusting the definition of `Term` to reference `ID` instead:

```
pub Term = {
    Num,
    "(" <Term> ")",
    "22" => format!("Twenty-two!"),
    ID => format!("Id({})", <>), // <-- changed this
};
```

In fact, the match declaration can map a regular expression to any kind of symbol you want (i.e., you can also map to a string literal or even a regular expression). Whatever symbol appears after the `=>` is what you should use in your grammar. As an example, in some languages have case-insensitive keywords; if you wanted to write `"BEGIN"` in the grammar itself, but have that map to a regular expression in the lexer, you might write:

```
match {
    r"(?i)begin" => "BEGIN",
    ...
}
```

And now any reference in your grammar to `"BEGIN"` will actually match any capitalization.

Handling full expressions

Now we are ready to extend our calculator to cover the full range of arithmetic expressions (well, at least the ones you learned in elementary school). Here is the next calculator example, calculator3:

```
use std::str::FromStr;

grammar;

pub Expr: i32 = {
    <l:Expr> "+" <r:Factor> => l + r,
    <l:Expr> "-" <r:Factor> => l - r,
    Factor,
};

Factor: i32 = {
    <l:Factor> "*" <r:Term> => l * r,
    <l:Factor> "/" <r:Term> => l / r,
    Term,
};

Term: i32 = {
    Num,
    "(" <Expr> ")",
};

Num: i32 = {
    r"[0-9]+" => i32::from_str(<>).unwrap(),
};
```

Perhaps the most interesting thing about this example is the way it encodes precedence. The idea of precedence of course is that in an expression like `2+3*4`, we want to do the multiplication first, and then the addition. LALRPOP doesn't have any built-in features for giving precedence to operators, mostly because I consider those to be creepy, but it's pretty straightforward to express precedence in your grammar by structuring it in tiers -- for example, here we have the nonterminal `Expr`, which covers all expressions. It consists of a series of factors that are added or subtracted from one another. A `Factor` is then a series of terms that are multiplied or divided. Finally, a `Term` is either a single number or, using parenthesis, an entire expr.

Abstracting from this example, the typical pattern for encoding precedence is to have one nonterminal per precedence level, where you begin with the operators of lowest precedence (`+`, `-`), add in the next highest precedence level (`*`, `/`), and finish with the bare "atomic" expressions like `Num`. Finally, you add in a parenthesized version of your top-level as an atomic expression, which lets people reset.

To see why this works, consider the two possible parse trees for something like `2+3*4`:

```
2 + 3   *    4          2   +  3   *    4
| | |   |    |          |   |  |   |    |
| | +-Factor-+    OR    +-Expr-+   |    |
| |     |                   |      |    |
+-Expr -+                   +----Factor-+
```

In the first one, we give multiplication higher precedence, and in the second one, we (incorrectly) give addition higher precedence. If you look at the grammar now, you can see that the second one is impossible: a `Factor` cannot have an `Expr` as its left-hand side. This is the purpose of the tiers: to force the parser into the precedence you want.

Building ASTs

Of course, most of the time, when you're parsing you don't want to compute a value, you want to build up some kind of data structure. Here's a quick example to show how that is done in LALRPOP. First, we need to *define* the data structure we will build. We're going to use a very simple `enum`:

```
pub enum Expr {
    Number(i32),
    Op(Box<Expr>, Opcode, Box<Expr>),
}

pub enum Opcode {
    Mul,
    Div,
    Add,
    Sub,
}
```

We put this code into an `ast.rs` module in our project, along with some `Debug` impls so that things pretty-print nicely. Now we will create the calculator4 example, which will build up this tree. To start, let's just look at the `Expr` nonterminal, which will show you most everything of how it is done (the most interesting lines have been flagged with comments):

```
use std::str::FromStr;
use ast::{Expr, Opcode}; // (0)

grammar;

pub Expr: Box<Expr> = { // (1)
    Expr ExprOp Factor => Box::new(Expr::Op(<>)), // (2)
    Factor,
};

ExprOp: Opcode = { // (3)
    "+" => Opcode::Add,
    "-" => Opcode::Sub,
};
```

First off, we have to import these new names into our file by adding a `use` statement (0). Next, we want to produce `Box<Expr>` values, so we change the type of `Expr` (and `Factor` and `Term`) to `Box<Expr>` (1). The action code changes accordingly in (2); here we've used the `<>` expansion to supply three arguments to `Expr::Op`. Finally, just for concision, we introduced an `ExprOp` nonterminal (3) to cover the two opcodes, which now trigger the same action code (before they triggered different action code, so we could do an addition vs a subtraction).

The definition of `Factor` is transformed in a similar way:

```
Factor: Box<Expr> = {
    Factor FactorOp Term => Box::new(Expr::Op(<>)),
    Term,
};

FactorOp: Opcode = {
    "*" => Opcode::Mul,
    "/" => Opcode::Div,
};
```

And finally we adjust the definitions of `Term` and `Num`. Here, we convert from a raw `i32` into a `Box<Expr>` when we transition from `Num` to `Term` (4):

```
Term: Box<Expr> = {
    Num => Box::new(Expr::Number(<>)), // (4)
    "(" <Expr> ")"
};

Num: i32 = {
    r"[0-9]+" => i32::from_str(<>).unwrap()
};
```

And that's it! Now we can test it by adding some code to our main.rs file that parses an expression and formats it using the `Debug` impl:

```
pub mod calculator4;
pub mod ast;

#[test]
fn calculator4() {
    let expr = calculator4::ExprParser::new()
        .parse("22 * 44 + 66")
        .unwrap();
    assert_eq!(&format!("{:?}", expr), "((22 * 44) + 66)");
}
```

Macros

Frequently when writing grammars we encounter repetitive constructs that we would like to copy-and-paste. A common example is defining something like a "comma-separated list". Imagine we wanted to parse a comma-separated list of expressions (with an optional trailing comma, of course). If we had to write this out in full, it would look something like:

```
Exprs: Vec<Box<Expr>> = {
    Exprs "," Expr => ...,
    Expr => vec![<>],
}
```

Of course, this doesn't handle trailing commas, and I've omitted the action code. If we added those, it would get a bit more complicated. So far, this is fine, but then what happens when we later want a comma-separated list of terms? Do we just copy-and-paste everything?

LALRPOP offers a better option. You can define macros. In fact, LALRPOP comes with four macros builtin: `*`, `?`, `+`, and `(...)`. So you can write something like `Expr?` to mean "an optional `Expr`". This will have type `Option<Box<Expr>>` (since `Expr` alone has type `Box<Expr>`). Similarly, you can write `Expr*` or `Expr+` to get a `Vec<Expr>` (with minimum length 0 and 1 respectively). The final macro is parentheses, which is a shorthand for creating a new nonterminal. This lets you write things like `(<Expr> ",")?` to mean an "optionally parse an `Expr` followed by a comma". Note the angle brackets around `Expr`: these ensures that the value of the `(<Expr> ",")` is the value of the expression, and not a tuple of the expression and the comma. This means that `(<Expr> ",")?` would have the type type

`Option<Box<Expr>>` (and not `Option<(Box<Expr>, &'input str)>`).

Using these operations we can define `Exprs` in terms of a macro `Comma<T>` that creates a comma-separated list of `T`, whatever `T` is (this definition appears in calculator5):

```
pub Exprs = Comma<Expr>; // (0)

Comma<T>: Vec<T> = { // (1)
    <v:(<T> ",")*> <e:T?> => match e { // (2)
        None => v,
        Some(e) => {
            let mut v = v;
            v.push(e);
            v
        }
    }
};
```

The definition of `Exprs` on line (0) is fairly obvious, I think. It just uses a macro `Comma<Expr>`. Let's take a look then at the definition of `Comma<T>` on line (1). This is sort of dense, so let's unpack it. First, `T` is some terminal or nonterminal, but note that we can also use it as a type: when the macro is expanded, the `T` in the type will be replaced with "whatever the type of `T` is".

Next, on (2), we parse `<v:(<T> ",")*> <e:T?>`. That's a lot of symbols, so let's first remove all the angle brackets, which just serve to tell LALRPOP what values you want to propagate and which you want to discard. In that case, we have: `(T ",")* T?`. Hopefully you can see that this matches a comma-separated list with an optional trailing comma. Now let's add those angle-brackets back in. In the parentheses, we get `(<T> ",")*` -- this just means that we keep the value of the `T` but discard the value of the comma when we build our vector. Then we capture that vector and call it `v`: `<v:(<T> ",")*>`. Finally, we capture the optional trailing element `e`: `<e:T?>`. This means the Rust code has two variables available to it, `v: Vec<T>` and `e: Option<T>`. The action code itself should then be fairly clear -- if `e` is `Some`, it appends it to the vector and returns the result.

As another example of using macros, you may recall the precedence tiers we saw in calculator4 (`Expr`, `Factor`, etc), which had a sort of repetitive structure. You could factor that out using a macro. In this case, it's a recursive macro:

```
Tier<Op,NextTier>: Box<Expr> = {
    Tier<Op,NextTier> Op NextTier => Box::new(Expr::Op(<>)),
    NextTier
};

Expr = Tier<ExprOp, Factor>;
Factor = Tier<FactorOp, Term>;

ExprOp: Opcode = { // (3)
    "+" => Opcode::Add,
    "-" => Opcode::Sub,
};

FactorOp: Opcode = {
    "*" => Opcode::Mul,
    "/" => Opcode::Div,
};
```

And, of course, we have to add some tests to main.rs file:

```
#[macro_use] extern crate lalrpop_util;

lalrpop_mod!(pub calculator5);

#[test]
fn calculator5() {
    let expr = calculator5::ExprsParser::new().parse("").unwrap();
    assert_eq!(&format!("{:?}", expr), "[]");

    let expr = calculator5::ExprsParser::new()
        .parse("22 * 44 + 66")
        .unwrap();
    assert_eq!(&format!("{:?}", expr), "[((22 * 44) + 66)]");

    let expr = calculator5::ExprsParser::new()
        .parse("22 * 44 + 66,")
        .unwrap();
    assert_eq!(&format!("{:?}", expr), "[((22 * 44) + 66)]");

    let expr = calculator5::ExprsParser::new()
        .parse("22 * 44 + 66, 13*3")
        .unwrap();
    assert_eq!(&format!("{:?}", expr), "[((22 * 44) + 66), (13 * 3)]");

    let expr = calculator5::ExprsParser::new()
        .parse("22 * 44 + 66, 13*3,")
        .unwrap();
    assert_eq!(&format!("{:?}", expr), "[((22 * 44) + 66), (13 * 3)]");
}
```

Error recovery

By default, the parser will stop as soon as it encounters an error. Sometimes though we would like to try and recover and keep going. LALRPOP can support this, but you have to help it by defining various "error recovery" points in your grammar. This is done by using a special `!` token: this token only occurs when the parser encounters an error in the input. When an error does occur, the parser will try to recover and keep going; it does this by injecting the `!` token into the stream, executing any actions that it can, and then dropping input tokens until it finds something that lets it continue.

Let's see how we can use error recovery to attempt to find multiple errors during parsing. First we need a way to return multiple errors as this is not something that LALRPOP does by itself so we add a `Vec` storing the errors we found during parsing. Since the result of `!` contains a token, error recovery requires that tokens can be cloned.

```
grammar<'err>(errors: &'err mut Vec<ErrorRecovery<usize, (usize, &'input
str), ()>>);
```

Since an alternative containing `!` is expected to return the same type of value as the other alternatives in the production we add an extra variant to `Expr` to indicate that an error was found.

```
pub enum Expr {
    Number(i32),
    Op(Box<Expr>, Opcode, Box<Expr>),
    Error,
}
```

Finally we modify the grammar, adding a third alternative containing `!` which simply stores the `ErrorRecovery` value received from `!` in `errors` and returns an `Expr::Error`. The value of the error token will be a `ParseError` value.

```
Term: Box<Expr> = {
    Num => Box::new(Expr::Number(<>)),
    "(" <Expr> ")",
    ! => { errors.push(<>); Box::new(Expr::Error) },
};
```

Now we can add a test that includes various errors (e.g., missing operands). You can see that the parser recovered from missing operands by inserting this `!` token where necessary.

```
#[test]
fn calculator6() {
    let mut errors = Vec::new();

    let expr = calculator6::ExprsParser::new()
        .parse(&mut errors, "22 * + 3")
        .unwrap();
    assert_eq!(&format!("{:?}", expr), "[((22 * error) + 3)]");

    let expr = calculator6::ExprsParser::new()
        .parse(&mut errors, "22 * 44 + 66, *3")
        .unwrap();
    assert_eq!(&format!("{:?}", expr), "[((22 * 44) + 66), (error *
3)]");

    let expr = calculator6::ExprsParser::new()
        .parse(&mut errors, "*")
        .unwrap();
    assert_eq!(&format!("{:?}", expr), "[(error * error)]");

    assert_eq!(errors.len(), 4);
}
```

Writing a custom lexer

Let's say we want to parse the Whitespace language, so we've put together a grammar like the following:

```
pub Program = <Statement*>;

Statement: ast::Stmt = {
    " " <StackOp>,
    "\t" " " <MathOp>,
    "\t" "\t" <HeapOp>,
    "\n" <FlowCtrl>,
    "\t" "\n" <Io>,
};

StackOp: ast::Stmt = {
    " " <Number> => ast::Stmt::Push(<>),
    "\n" " " => ast::Stmt::Dup,
    "\n" "\t" => ast::Stmt::Swap,
    "\n" "\n" => ast::Stmt::Discard,
};

MathOp: ast::Stmt = {
    " " " " => ast::Stmt::Add,
    " " "\t" => ast::Stmt::Sub,
    " " "\n" => ast::Stmt::Mul,
    "\t" " " => ast::Stmt::Div,
    "\t" "\t" => ast::Stmt::Mod,
};

// Remainder omitted
```

Naturally, it doesn't work. By default, LALRPOP generates a tokenizer that skips all whitespace -- including newlines. What we *want* is to capture whitespace characters and ignore the rest as comments, and LALRPOP does the opposite of that.

At the moment, LALRPOP doesn't allow you to configure the default tokenizer. In the future it will become quite flexible, but for now we have to write our own.

Let's start by defining the stream format. The parser will accept an iterator where each item in the stream has the following structure:

```
pub type Spanned<Tok, Loc, Error> = Result<(Loc, Tok, Loc), Error>;
```

`Loc` is typically just a `usize`, representing a byte offset into the input string. Each token is accompanied by two of them, marking the start and end positions where it was found. `Error` can be pretty much anything you choose. And of course `Tok` is the meat of the stream, defining what possible values the tokens themselves can have. Following the conventions of Rust iterators, we'll signal a valid token with `Some(Ok(...))`, an error with `Some(Err(...))`, and EOF with `None`.

(Note that the term "tokenizer" normally refers to a piece of code that simply splits up the stream, whereas a "lexer" also tags each token with its lexical category. What we're writing is the latter.)

Whitespace is a simple language from a lexical standpoint, with only three valid tokens:

```
pub enum Tok {
    Space,
    Tab,
    Linefeed,
}
```

Everything else is a comment. There are no invalid lexes, so we'll define our own error type, a void enum:

```
pub enum LexicalError {
    // Not possible
}
```

Now for the lexer itself. We'll take a string slice as its input. For each token we process, we'll want to know the character value, and the byte offset in the string where it begins. We can do that by wrapping the `CharIndices` iterator, which yields tuples of `(usize, char)` representing exactly that information.

```
use std::str::CharIndices;

pub struct Lexer<'input> {
    chars: CharIndices<'input>,
}

impl<'input> Lexer<'input> {
    pub fn new(input: &'input str) -> Self {
        Lexer { chars: input.char_indices() }
    }
}
```

(The lifetime parameter `'input` indicates that the Lexer cannot outlive the string it's trying to parse.)

Let's review our rules:

- For a space character, we output `Tok::Space`.
- For a tab character, we output `Tok::Tab`.
- For a linefeed (newline) character, we output `Tok::Linefeed`.

- We skip all other characters.
- If we've reached the end of the string, we'll return `None` to signal EOF.

Writing a lexer for a language with multi-character tokens can get very complicated, but this is so straightforward, we can translate it directly into code without thinking very hard. Here's our `Iterator` implementation:

```
impl<'input> Iterator for Lexer<'input> {
    type Item = Spanned<Tok, usize, LexicalError>;

    fn next(&mut self) -> Option<Self::Item> {
        loop {
            match self.chars.next() {
                Some((i, ' ')) => return Some(Ok((i, Tok::Space, i+1))),
                Some((i, '\t')) => return Some(Ok((i, Tok::Tab, i+1))),
                Some((i, '\n')) => return Some(Ok((i, Tok::Linefeed,
i+1))),

                None => return None, // End of file
                _ => continue, // Comment; skip this character
            }
        }
    }
}
```

That's it. That's all we need.

Updating the parser

To use this with LALRPOP, we need to expose its API to the parser. It's pretty easy to do, but also somewhat magical, so pay close attention. Pick a convenient place in the grammar file (I chose the bottom) and insert an `extern` block:

```
extern {
    // ...
}
```

Now we tell LALRPOP about the `Location` and `Error` types, as if we're writing a trait:

```
extern {
    type Location = usize;
    type Error = lexer::LexicalError;

    // ...
}
```

We expose the `Tok` type by kinda sorta redeclaring it:

```
extern {
    type Location = usize;
    type Error = lexer::LexicalError;

    enum lexer::Tok {
        // ...
    }
}
```

Now we have to declare each of our terminals. For each variant of `Tok`, we pick what name the parser will see, and write a pattern of the form `name => lexer::Tok::Variant`, similar to how action code works in grammar rules. The name can be an identifier, or a string literal. We'll use the latter.

Here's the whole thing:

```
extern {
    type Location = usize;
    type Error = lexer::LexicalError;

    enum lexer::Tok {
        " " => lexer::Tok::Space,
        "\t" => lexer::Tok::Tab,
        "\n" => lexer::Tok::Linefeed,
    }
}
```

From now on, the parser will take a `Lexer` as its input instead of a string slice, like so:

```
    let lexer = lexer::Lexer::new("\n\n\n");
    match parser::parse_Program(lexer) {
        ...
    }
```

And any time we write a string literal in the grammar, it'll substitute a variant of our

`Tok` enum. This means **we don't have to change any of the rules we already wrote!** This will work as-is:

```
FlowCtrl: ast::Stmt = {
    " " " " <Label> => ast::Stmt::Mark(<>),
    " " "\t" <Label> => ast::Stmt::Call(<>),
    " " "\n" <Label> => ast::Stmt::Jump(<>),
    "\t" " " <Label> => ast::Stmt::Jz(<>),
    "\t" "\t" <Label> => ast::Stmt::Js(<>),
    "\t" "\n" => ast::Stmt::Return,
    "\n" "\n" => ast::Stmt::Exit,
};
```

The complete grammar is available in `whitespace/src/parser.lalrpop`.

Where to go from here

Things to try that apply to lexers in general:

- Longer tokens
- Tokens that require tracking internal lexer state

Things to try that are LALRPOP-specific:

- Persuade a lexer generator to output the `Spanned` format
- Make this tutorial better

When you setup LALRPOP, you create a `build.rs` file that looks something like this:

```
extern crate lalrpop;

fn main() {
    lalrpop::process_root().unwrap();
}
```

This `process_root()` call simply applies the default configuration: so it will transform `.lalrpop` files into `.rs` files *in-place* (in your `src` directory), and it will only do so if the `.lalrpop` file has actually changed. But you can also use the `Configuration` struct to get more detailed control.

For example, to **force** the use of colors in the output (ignoring the TTY settings), you might make your `build.rs` file look like so:

```
extern crate lalrpop;

fn main() {
    lalrpop::Configuration::new()
        .always_use_colors()
        .process_current_dir();
}
```

Up to version 0.15, LALRPOP was generating its files in the same directory of the input files. Since 0.16, files are generated in the Cargo's **output directory**.

If you want to keep the previous behaviour, you can use `generate_in_source_tree` in your configuration:

```
extern crate lalrpop;

fn main() {
    lalrpop::Configuration::new()
        .generate_in_source_tree()
        .process();
}
```

For each `foo.lalrpop` file you can simply have `mod foo;` in your source tree. The `lalrpop_mod` macro is not useful in this mode.

Contributors

Here is a list of the contributors who have helped improving LALRPOP. This list may be incomplete. The "canonical list" can be found in the git history; if you have contributed but you are not in this list, please add yourself!

- nikomatsakis
- Marwes
- wieczyk
- joerivanruth
- ahmedcharles
- malleusinferni
- dflemstr
- shahn
- federicomenaquintero
- fhahn
- jonas-schievink
- oconnor0

- minijackson
- fitzgen
- ruuda
- wagenet
- pyfisch
- vmx
- dagit
- ashleygwilliams
- brson
- serprex
- pensivearchitect
- larsluthman
- mchesser
- notriddle
- nixpulvis
- Nemikolh
- nick70
- paupino